Towards Visualization of Unit Test and Source Code Relations

By Nadera Aljawabrah

A Thesis:

Submitted To The Phd School In Computer Science Of The

University Of Szeged In Partial Fulfilment Of The

Requirements For The Degree Of
Doctor of Philosophy

Supervisor: Dr.Tamás Gergely

Szeged,2020

Dedicated to

Mom, husband, my kids (Yaman, Mayar, and Zain), sisters,

and my father's soul

Towards Visualization of Unit Test and Source Code Relations

Nadera Aljawabrah

Submitted for the degree of Doctor of Philosophy
2020

Abstract

Test-to-code traceability is the ability to relate the test unit and source code artifacts created during the software development life cycle (SDLC). Traceability of test and code relations is fundamental to support various activities of software development such as program comprehension, verification and validation, impact analysis, reuse, maintenance, and software evolution. Notwithstanding its importance, many significant challenges are associated with traceability. One of these challenges is how to support the comprehension and maintenance of these links efficiently and effectively. Visualization is an important part that effectively supports understanding test-to-code relations. It provides enhanced process visibility, helps engineers, developers, and testers to verify the quality of TCT links, and understand which code modules are tested by which unit tests.

As there are many sources from where the traceability relations can be inferred, one of the most important questions is to decide which source or combination of sources is the best to determine the test-to-code links. It is obvious that, if these sources disagree, this will make it harder to understand what is going on, what was the goal of the developer, and how the components are really related and change impact analysis can yield false results, etc. Fortunately, visualization can aid this task. The presented approach consists of three parts. The first part consists of the recognition of artifacts from two different areas, source code, and unit test. The second part presents different sources for capturing the links between code and test such as a naming convention, last call before assert, and the static call graph. The third part includes the visualization method used to visually present the traceability links inferred from traceability links sources. The trace visualization approach is implemented as a trace visualization tool, which is called TCTracVis.

This thesis also provides an empirical study based on the implementation of the presented approach. The approach and its tool support are applied in different software development projects conducted with a group of students, academics, and from industry. The effectiveness and practicability of the presented approach and its tool support have been evaluated. The effectiveness results indicate that the visualization of multilevel test-to-code traceability links, inferred from multiple sources, is more effective for the testers and developers than using visualization of a single source of traceability links. It helps to get a bigger picture of what is going on with the tests, find solutions to the problems in testing, and understand the relationships between test cases and the corresponding units under test. At the same time, the usability results indicate that the participants found that the approach and its tool support usability, and enhance the overall browsing, comprehension, and maintenance of test-to-code traceability links of a system.

November 24, 2020

Acknowledgements

Alhamdulillah, thank you Allah for all the blessing you had given me. The ability, courage, endurance, and patience you put inside me, strengthening me in completing this study and research.
First and foremost, I would like to show my deepest appreciation to my supervisor, Dr.Tamás Gergely, for his precious and continuous guidance and support throughout the course of this study. His knowledge and his logical way of thinking have been of great value for me.

My special gratitude goes to my beloved mother for being the secret of my life-long success with her blessing and prayers.
I owe my loving thanks to my supportive, encouraging and patient husband, Mr.Brhan Alzyoud, my dear sisters Andera & Kawthar, my brother in law Ayed, my beloved son Yaman, dear daughters Mayar & Zain, and all family members. Without their endless support, love, and belief in me, it would have been impossible for me to finish this work.

Finally, I'll be forever grateful for all my friends who always being there and being supportive in spiritual and understanding during the whole course of my study in University of Szeged.

Contents

Abbreviations

LA	Lexical Analyses
LCBA	Last Call Before Assert
CV	Co-Evolution
LOC	Lines of Code
LSI	Latent Semantic Indexing
NC	Naming Convention
NOA	Number of Attributes
NOM	Number of Methods
PM	Probabilistic Model
RTM	Requirements Traceability Matrix
SCG	Static Call Graph
SCOTCH	Slicing and Coupling based Test to Code trace Hunter
SDLC	Software Development Life Cycle
STS	Started Tested Sets
SVN	Subversion (version control system)
UML	Unified Modeling Language
UUN	Unit under Test
VRE	Virtual Reality Environment

VSM	Vector Space Models
XML	Extensible Markup Language

List of Figures

List of Tables

Chapter 1

Introduction

The IEEE standard glossary of software engineering of terminology [1] defines traceability as “The degree to which a relationship can be established between two or more products of the development process”. There are two types of traceability as mentioned in [2]: 1) traceability between software artifacts at the same level of software life-cycle which is known as vertical traceability (e.g. traceability between requirement components), 2) and traceability between software artifacts at different levels of software life-cycle which is known as horizontal traceability (e.g. traceability between source code and test cases).

Source code evolves very often. Nevertheless, test cases that examine it are not updated; maintaining consistency and traceability information between unit tests and source code is costly and time consuming. As well as, testing is frequently skipped due to the pressure to time to move on to the next change of a software or due to the market. Furthermore, system developers or people who maintained it may no longer available due to turnover or outsourcing.

In this thesis, we pay more attention to traceability links between two of software artifacts: test units and its corresponding units under test (related source code). Our goal is to provide a visualization method that helps the developers/testers to understand traceability links between units' test and its related source code. we developed a tool that automatically recovers traceability links between test cases and their related classes and supports visualization of these retrieved links.

Visualization is an efficient way that facilitates the understanding of traceability links between software artifacts, as well as supports several software development tasks. Some noteworthy examples are:

- Software comprehension.
- Maintenance.
- Change Impact analysis.
- Refactoring.
- Software revolution.

1.1 Motivation

Test-to-code traceability links is an underlying asset during the development and the evolution of the software. For example, unit tests are significant source of documentation, they can help a developer to understand production code and identify failures, particularly when performing maintenance tasks. Moreover, in refactoring process, dependence's between unit tests and its corresponding code under test can be used to maintain the consistency. When refactoring, some modifications to the test suite are required together with the source code in order to keep it valid [3]. Therefore, if the exact links between unit tests and related tested code are defined, refactoring of these unit tests can be automated and greatly simplified [4]. Visualization is a very effective method that enables testers to understand, recover, and browse the inter-relationships between unit tests and its corresponding tested code in an intuitive and natural way.

According to [5] the development of test-to-code traceability links is not sufficiently managed in literature. Recent research on test-to-code traceability links is directed on how to identify the links between test and code, as well as only a few approaches have been suggested and used to recover test-to-code-traceability links. Most of these approaches have come from the outstanding work [6]. In this previous work, a set of traceability recovery strategies have been proposed to establish links between Xunit test cases and production class in object-oriented programs. However, none of the proposed approaches provides tool support to facilitate traceability links, as well

as, none of these approaches support visualization of traceability links [5]. Moreover, they concluded that there is no single technique that is superior to all others.
Several approaches have been explored to extract the links between test and code [6], [7]. Recently, research into combining test-to-code traceability links recovery approaches has become very popular, as it helps to improve the quality and accuracy of the retrieved links [153], [159], [121], [109]. However, it is complex, time-consuming, and susceptible to error task to manually retrieve test-to-code traceability links. Moreover, it is also important to efficiently support the comprehension, browsing, and maintenance of the retrieved links. These issues can be significantly solved by the support the automatic generating of test-to-code traceability links [5], and adopting visualization to represent these links in an intuitive and natural way.
In recent years, several publications have appeared [9]- [11] documenting the visualization of traceability links among different software artifacts (e.g. requirements, source codes, documents, etc.). Moreover, many visualization tools have been designed to represent traces between software artifacts in different views [12], [11]. However, as yet, no visualization method focused on test-code traceability links, neither tools were implemented with this focus., which in turn, lays the foundation of our work.

1.2 Contributions of Thesis

In this thesis, we focus on the specific problem of automatically recovering and visualizing the traceability links between test cases and the related production classes. We provide an innovative visualization test-to-code traceability approach that combines various test-to-code recovery approaches and support efficient visualization technique that displays the recovered links in two levels, class-level, and method level. We developed a tool TCTracVis that supports the automated recovery approaches and the simultaneous visualizations of test and code relations. The main idea of using the traceability recovery approaches is to help software engineers to trace the relationships between unit test and source code, and automatically extract traceability links at low cost and time. The goal of using visualization is to identify the disagreement between traceability links inferred from different sources [BNN02]. This might point out places where something is wrong with the tests and/or the code (at least their relationship) in a specific system.

This thesis describes a novel approach for the visualization of the traceability links between two software artifacts: test and code. The approach represents a new way to efficiently visualize links between these two artifacts. Furthermore, it combines a number of traceability methods to automatically infer traceability links between a test case and the responsible unit under test (UUT). Then, the results produced are visualized to help the developers to decide on what are the real links. Our approach and its implementation aim to improve and ease the development process in several ways: helps comprehension, interact search, and impact analysis when something needs to be changed.

The main contributions of this thesis are as follows: First, a literature review is presented investigating existing research on the traceability between tests and code. Second, a multilevel visualization approach that presents a detailed overview of test-to-code traceability links on the class level and method level, alongside the three automated traceability recovery methods to retrieve these links. In addition, the practical implementation of a visualization tool has been discussed. Finally, an empirical study was conducted applying the presented approach and tool support in practice and evaluating the usability and efficiency of the developed tool.

1.3 Outline of Thesis

The remainder of this thesis is arranged as follows:

Chapter 2: Background Information

In this chapter, we introduced a background knowledge about the general terms and terminology used in this thesis. We presented a brief description of test and code relations, visualization of software system, and in particular visualization of source code and visualization of testing related information.

Chapter 3: State-of-the-art

This chapter presents the results of a systematic literature review on the visualization of traceability between test units and their related source code. It explains existing traceability recovery approaches as well as visualization techniques used to present the traceability links between software artifacts. Furthermore, it discusses their strengths, weaknesses, and limitations to reveal gaps that a new approach can fill.

Chapter 4: Trace Visualization Approach

This chapter describes the trace visualization approach that automatically captures and visualizes the traceability links between test and code during development. Assumptions and preconditions of the approach are addressed. The main phases of the approach are briefly presented. In this chapter, a system called TCTracVis is also described, which implements the trace visualization approach.

Chapter 5: Evaluation of Trace Visualization Approach

This chapter presents an experimental evaluation of the presented visualization approach and its tool support.

Chapter 6: Summary

The final chapter consists of the conclusion and the summary of the thesis as well as suggestions for future work.

1.4 Published Articles Related to the Dissertation

This thesis is based on the following publications:

1. Aljawabrah, Nadera, and Tamás Gergely. "Visualization of test-to-code relations to detect problems of unit tests." The 11th Conference of Phd Students in Computer Science. 2018 [BNN01].

2. Aljawabrah, Nadera, Támas Gergely, and Mohammad Kharabsheh. "Understanding Test-to-Code Traceability Links: The Need for a Better Visualizing Model." International Conference on Computational Science and Its Applications. Springer, Cham, 2019. [BNN02].

3. Aljawabrah, N., and Qusef, A. (2019, December). TCTracVis: test-to-code traceability links visualization tool. In Proceedings of the Second International

Conference on Data Science, E-Learning and Information Systems (pp. 1-4) [BNN03].

4. Nadera Aljawabrah, AbdAllah Qusef, Tamás Gergely, and Adhyatmananda Pati, Visualizing Multilevel Test-to-Code Relations. In 3rd International Conference on Information and Communication Technology and Applications, Springer (CCIS), 2020.[BNN04][1]

5. Nadera Aljawabrah, Tamás Gergely, Sanjay Misra, and Luis Fernandez-Sanz, Automated Recovery and Visualization of Test-to-Code (TCT) Links: An Evaluation, in the submission to IEEE Access.[BNN05]

Table 1.1: Thesis Points Matrix

Thesis Points	**Publications**				
	[BNN01]	**[BNN02]**	**[BNN03]**	**[BNN04]**	**[BNN05]**
A comprehensive overview investigating existence research on the visualization of the traceability between tests and code	✓	✓			
A visualization method that visually presenting the test-to-code traceability links at different levels, alongside multiple traceability recovery methods to retrieve these links. The visualization method is implemented using the TCTracVis visualization tool.			✓	✓	✓
Evaluation of the presented approach and tool support in terms of usability and efficiency throughout an empirical study.				✓	✓

Other Publications

The following publications have been published during my PhD study.

1. Otoom, A. F., Hammad, M., Al-Jawabreh, N., and Seini, R. A. (2016). Visualizing Testing Results for Software Projects. In Proc. of the 17th International Arab Conference on Information Technology (ACIT'16), Morocco [BNN06].

2. Hammad, Maen, et al. "Multiview Visualization of Software Testing Results." International Journal of Computing and Digital Systems 9.1 (2020) [BNN07].

[1]Proceedings will be published by Springer in the Communication in Computer and Information Science (CCIS) series.

Chapter 2

Background

This chapter describes the background information included in this thesis. General terms are defined in the contexts of traceability and visualization. This involves a concise introduction to test-to-code traceability with further information about using visualization in representing software structure and different software artifacts (e.g. source code, test results).

2.1 Traceability Definition

There are several terms and definitions of traceability that are provided in the literature. We summarize the most cited definitions of traceability in this section.
The term traceability is defined earlier in the IEEE standard glossary of software engineering of terminology [1] which provides two definitions of traceability:

1. "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another".

2. "The degree to which each element in a software development product establishes its reason for existing".

Spanoudakis and Zisman [14] provide a definition of traceability as "the ability to relate artifacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artifacts, and the rationale that explains the form of the artifacts". The definition here affirms the use of traceability links in development process, which, for example, these uses are being able to create the artifacts and explain the rational of them.

Another general definition of the traceability is given by Gotel [15] who define the traceability as "the potential for traces to be established (created and maintained) and used. This definition explicitly states that if the trace links will be used, they should be established first.

The definition of traceability in [15] is consistent with what stated in [16] which describes software traceability as "the ability to interrelate any uniquely identifiable software engineering artifact to any other".

In addition, one of the most common term is requirements traceability. In requirements engineering, the term traceability is explicitly related to requirements. Requirements traceability is defined by [17] as "The ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)". This definition is first and most widely accepted and used in the context of requirements traceability. It discusses the artifacts refinements and iterations. However, it is explicitly oriented to requirements and to follow a requirement's life, but nothing is mentioned about the use of traceability. In [BNN08], we proposed a model that defines the requirements elicitation process. The model focused on the improvement of the requirements quality by applying the requirements tracking and refinement. The aim of tracking requirements is to allocate each requirement to a stakeholder or a user who requests it. Each phase is useful in ensuring the satisfaction of the users and will fulfill the requirements as needed.

[19] deduced a definition of traceability based on analyzing a set of definitions of traceability. The definition encompasses not only the artifacts created in software development, but also those artifacts created during system development (e.g. hardware models). Therefore, he defined system traceability as "The ability to relate uniquely identifiable system engineering artifacts created and evolved during the

development of a system, maintain these relationships throughout the development life cycle and use them to facilitate system development activities". The artifacts in system development in this case include all artifacts that are related to the system. For example: hardware models, behavior models, design models, code, requirements, test cases, test results, and stakeholders. Based on this definition, we can deduce that three aspects of traceability should be taken into account during system development: identify the artifacts involved, maintain the links between them, and use these links to facilitate the activities in the development process.

2.2 Test-to-Code Traceability Links

In the line with the understanding of the concept of traceability link, Parizi et.al. [5] suggested a definition of test-to-code traceability which is defined as "the one to represent the relationship between two elements of tests and code artifacts, i.e. between a test case and the responsible unit under test (UUT)". Based on the proposed definition, many other terms related to test-to-code traceability have been established such as:

- Trace link (or traceability link). The definition of trace link is given by [16] as "A specified association between a pair of artifacts, one comprising the source artifact (e.g. tests) and one comprising the target artifact (e.g. source code)". The authors in [15] define a trace as "a specified triplet of elements comprising: a source artifact, a target artifact and a trace link associating the two artifacts". The tracing task implies identifying the target artifacts that are related to a specified source artifact".

- Trace artifact. The authors in [15] define trace artifact as "traceable units of data". While in [16] it is defined as "A traceable unit of data (e.g., a single requirement, a cluster of requirements, a UML class, a UML class operation, a Java class or even a person). It is one of the trace elements and is qualified as either a source artifact or as a target artifact when it participates in a trace".

- Test case: in Center of excellence for software traceability (coest) [16], test case is defined as “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement”. And in [20] it is defined as “Documentation specifying inputs, predicted results, and a set of execution conditions for a test item”.

- Traceability (links) recovery: Parizi et.al. [5] defined traceability links recovery as “retrieving candidate links between elements in one artifact, and elements in another”. Traceability link recovery process relies on using a single or multiple strategies and techniques, or approaches.
- Test-to-code traceability links recovery approach : As mentioned in [5], it is an approach that is capable to derive the traceability links between specific tests and source code artifacts. In research, test-to-code traceability links is immature. On the other hand, there is much more interest in traceability links between requirements and other software artifacts [11], [21] - [24].

2.3 Software Visualization

Visualization is a process of presenting an abstract and complex data in a remarkable, colour-coordinate, and clear-cut format in forms of images, charts, graphs, diagrams, and tables that can aid in understanding the purpose of data [BNN02]. According to Friedman [25], “Main goal of data visualization is to communicate information clearly and effectively through graphical means. It doesn’t mean that data visualization needs to look boring to be functional or extremely sophisticated to look beautiful” i.e. ideal visualization should not only convey information clearly but also motivate the reader to share and interest. Claire et al. [26] defined software visualization as “a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration”. In the other words, software visualization is the mapping from software artifacts to graphical representations (see Figure 2.1). In probably the simplest case, we can textually visualize artifacts, which is actually regarded as

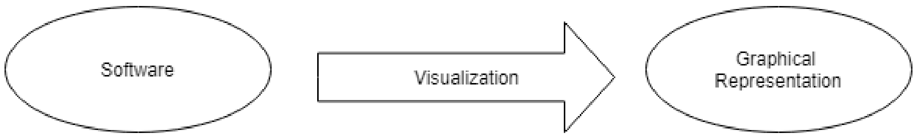


Figure 2.1: Mapping Software to Graphical Representation (from [28])

the most primitive type of visualization. Research studies show evidence that, for certain activities, specific methods of graphic visualization work better than textual visualization [27]. Many researchers believe in the value of software visualization, particularly in supporting the software engineering process.

In the context of programs comprehension, visualization plays an effective role in understanding the complex data analysis with respect to the evolution of large software systems in which major changes on source code entities have been occurred. Throughout the history of software development, visualization of software artifacts has drawn considerable interest from research teams and several visualization techniques have been suggested to examine and explore various aspects of software systems [29], [30].

It is necessary to identify the appropriate visualization technique for the given developer's needs, these needs include:

A. Task. What is the visualization used for? Many visualization techniques have been proposed to support developers in performing different software engineering tasks. Tasks can be categorized using a number of standards. For example, S.Diehl in [31] classified the tasks into three categories:

   - System's structure, where the system can be visualized depending on the dependencies attained from source code.
   - System's behaviour to visualize the system by describing the execution of the running system.
   - System's evolution, where the visualization described the changes captured in version control system.

   Jonathan et al. in [32] categorized the tasks based on the cognitive process domains into: finding, building, understanding, and questioning. Tasks can be also defined by the matter of need [33]:

1. People (e.g. who works on a task).
2. Code (e.g. the code changes).
3. Progress (e.g. task item progress).
4. Test (e.g. test results analysis).

Utilizing these classifications in the analysis of the tasks can help developers to identify the task and, in turn, find the appropriate visualization techniques.

B. Audience. Users who have the data and will use the visualization tool play a particular role in the software life-cycle [148]. The software visualization that successfully address issues arise with a specific audience can be unsuccessful with the needs of another one. Therefore, when looking for an appropriate visualization for certain task, the role of user should be taken into account. While programmer's audience is the most targeted by most of software visualization tools, some other visualization tools target another audience such as architect or project manager.

C. Data. To choose which visualization technique has to be used depends in part on the data features. These software data features can be different from the same system (e.g. source, format). Software visualization can provide developers with a tangible and meaningful representation of the software data [148]. There are various types of software data that relate to software system. For example: source code can be visualized to analyse software structure, execution logs can be visualized to analyze software behavior, and meta-data from the VCS to analyze the software evolution. As well as, only certain data types can be presented by some visualization techniques (e.g. hierarchical, continuous, discrete, quantitative, and qualitative).

D. Technique. The technique used to display the visualization defines the graphical attributes that represent the characteristics of software data. Various visualization-based techniques have been used in software visualization [149].

For instance, graphs are the most common visualization technique used to convey information and describe binary relations (in general). Other techniques are also used depending on the context, such as charts, UML diagrams, and trees. Out of these techniques, in recent years, a metaphor has become a key concept where ideas or objects (lower level of abstraction) are used as a representative or symbol of other things (higher level of abstraction) which are different from their actual meaning.

E. Medium. The medium refers to the means where visualization should be displayed. Software visualizations are commonly displayed using the standard computer screen medium [34]. Software visualization has evolved from two-dimensional (2D) representations to three dimensional (3D) and, currently, virtual environments. Therefore, suitable medium plays a significant role in improving the efficiency of software visualizations.

2.4 Visualization of Source Code

The area of visualizing source code relevant information is widely targeted in literature and it has been accepted as a means to help in software maintenance and understand the evolution of software system [35], [36]. The visualization technique used depends on the information being visualized (e.g. metrics, relationships, dependences). For example: graph-based representation turned out to be appropriate for visualizing source code evolution.

Most of studies have focused on visualizing source code related metrics and many approaches are proposed in this regard [150]. Metrics is a numerical/proportional value to describe and measure the quality of software artifacts [37], [38] (e.g. source code quality, testing quality, documentation quality), therefore it is called quality metric. Identifying which metrics can be used depends on the intent of visualization. The code-related metrics which has been frequently used for visualization purposes are: line of codes (LOC), McCabe complexity [39] and number of methods (NOM) [40], [41]. These metrics support the maintainability of source code. In visualization space, data and metrics are mapped to a set of visual attributes according to the context of visualization.

City metaphor is the most popular metaphor used for visualizing program components [146]. This metaphor supports navigation the program, interaction with represented elements, and explore the city' structure. City metaphor is an effective 3D method to represent a software structure that enables the user to be well aware of the position of software objects. Thus, it would be easily retrieved in the development process. In other words, 3D visualization makes use of the spatial memory of users [42].

Wettel and Lanza [43] presented a 3D city metaphor-based visualization to represent large and complex object-oriented programs. They represent packages as districts included the buildings which in turn represent classes using CodeCity visualization's tool. Two code metrics are used for mappings on visual properties: NOM maps on the height of the building, and NOA maps on the width. Their visualization is limited on a higher level of abstraction, i.e. package and class.

Quite recently, considerable attention has been paid to use 3D game environments in software visualization. CodeMetropolis [44] is a command-line tool applying city metaphor to visualize source code at a lower level of abstraction (methods and attributes). It uses a game engine, "Minecraft" [45], to visualize the structure of source code. A single method is represented as a floor located in a building (class). Code metrics display the distinct attributes of a software system. These attributes are mapped to various properties in visual representation space. For instance, the height of the floor expresses the size of the method in terms of logical lines of code. A developer is a player who can fly and explore Minecraft word and get much detail about the internals of classes. Code Park [46], another game environment-based tool, has been recently developed to visualize source code. In this tool, the source code itself has been directly visualized in 3D space instead of using a metaphor to be represented.

A set of code metrics has been used in explore 3D graphs metaphor to describe the internal structure and relations of the large-size programs for quality assessment purposes [47]. Visual properties of program entities (e.g. size, shape, color) represent particular metrics of these entities for mapping in 3D visualization metaphor. Infor-

mation is presented from two points of view; usage-based pattern and inheritance-based pattern. Depending on these patterns, quality attributes such as the size and complexity of programs can be observed in visual space.
In line with the aforementioned works, there have been extensive research works related to code visualization, and research resort to visualization to reduce the submitted effort in understanding software, which in turn, simplifies the software maintenance and evolution [30]. Visualization also leads to a better code understanding particularly in 3D visualization.

2.5 Visualization of Testing Information

Visualization can be an effective method to provide a valuable information about; the adequacy of code testing [48], visualizing software faults [49], and evaluation code coverage of test suites' quality [50]. There has been a large amount of literature that deals with the visualization of test information, and a wide range of tools that have been proposed for the task of visualizing testing information. Jones et al. [51] proposed a spectrum-based color tool which visualizes code statements that are executed by test cases which in turn facilitate and support faults localization in code under test. Test-Fault localization is a main objective of other visualization tools as well such as TestQ [52] and X̧Suds [53]. Visualization supports understanding inner workings of source code and test suites. Cornelissen et al [54] proposed an approach to visualize the behavior of test suites using sequence diagrams produced by test execution.
Test-related metrics has been gaining importance in visualization's realm in recent years. Test-metrics can be accepted as a good indication of a quality of test. Code coverage metrics is one of the most popular test metrics used to measure the percentage of code that is executed by test cases. As the size and complexity of software under test and its automated test suites increasing, visualization is needed to analyze code coverage and to provide testers a wide range of information about the quality, performance, and cost of the testing process. A wide number of tools have been proposed for the task of visualization of testing information such as (TeCReV) [55]

which is a graph-based tool for visualizing test coverage and test redundancy information. The proposed tool can be used in many software testing activities such as improving testing coverage and fault localization. Another visualization technique has been presented in [56]. Its main target is to help software testers to determine the location of test suite, its relationship with the production code, and which parts of code are covered by test cases.

Various visualization techniques are developed depending on the type of data to be visualized and on the objectives of visualization. City metaphor has been also used to provide information with respect of the evolution of test cases. In [57] test cases are represented as buildings and each metric's value is associated with feature of visual elements (e.g. color, high, size) in 3D visual space. The visualization provides information supporting regression test selection such as, how many test cases were usually or no longer used. Balogh et al. [58] extended CodeMetropolis to include visualization of test related metrics in the Minecraft world to support developers to better understand the test suites quality and its relation to the production code. Different types of test metrics are determined to show the behavior of test on the code of different units; e.g. code coverage metrics, partition metrics and other specific metrics. In the visualization space, code units are represented as buildings protected by outposts which, in turn, represent test suites. The attributes of the outposts reflect the quality attributes of the tests.

Test suites are usually used to evaluate software systems and detect the program faults. The larger the programs are, the more of the test cases executed; thus, a huge amount of data will be produced which is difficult to be interpreted as textual form. Visualization of test suites is useful to give any reader an obvious view of the testing results as well as to determine the faults occurrence in source code with least efforts and time.

In [BNN07], we present a visualization approach that models the results of the test cases applied in object oriented code elements. The proposed visualization help testers to understand and to keep track on test cases and their tested code elements. Five views are proposed to cover different code levels; method, class, package, UML and system. A tool has been developed to automatically manage the software testing process and to generate the data model for the proposed visualizations. The evaluation results showed that the proposed views are useful and helpful in understanding the testing results.

As mentioned in Section 2.2, the relationships between test and code are known as traceability links. Previous researches [59], [7] mainly concerned to derive these links using different methods. Nevertheless, visualization of these links is not supported by these methods.

Chapter 3

State of The Art

In this chapter, we provide an overview of the state of the art of visualization traceability links between unit tests and code. We conducted a study that identified existing research creating, using, and representing test-to-code traceability links. In our study, we have the following research questions for our searches:

- **RQ1**: How are the links between unit test and code under test created?
- **RQ2**: What supporting tools are usually used for the creation of links?
- **RQ3**: To what extent has a visualization of test-to-code relations been investigated in existing studies?
- **RQ4**: What visualization techniques and tools are proposed to represent test-code relations?

In the next sections, we present our research method. Then, an overview of the identified approaches will be provided. Finally, the research questions are taken up again in the discussion.

3.1 Research Method

For our research strategies and documentation, we used the guidelines in [147]. The main objective was to describe major contributions in three research areas of traceability between: 1) source code and software artifacts, 2) test cases and software artifacts, and 3) test cases and source code, and two research areas of visualization of 1) software artifacts traceability links, and particularly 2) test-to-code traceability links. [147] suggest that a systematic literature review should have the following characteristics:

- A review protocols (identify research question)
- A defined search strategy
- A wide range of search source
- A described search string, based on a list of synonyms combined by ANDs and ORs
- A documented search strategy
- Explicit inclusion and exclusion criteria paper selection should be checked by two researchers.

We have met all the above criteria except the paper selection, which was checked only by one researcher. The review adopted a structured process with the following steps:

3.1.1 Creating Search Strings

We gathered several publications that listed various approaches on the basis of different approaches relating to traceability topics. Therefore, we have refined and selected those methods that meet the following criteria in line with our research aims:

- Those traceability approaches that make a separation between traceability related to source code and artifact traceability (e.g. requirements, documents, bugs).
- Those traceability approaches that make a separation between traceability related to tests and artifact traceability (e.g. requirements, design).

- Those traceability approaches that define only the traceability links between tests and code.

Thus, we conducted four separate searches; we created four different search strings. The first search string covers research for generating and using links between software artifacts and source code, and software artifacts and test cases. The second search string covers research generating and using links between test cases and source code. While in the third search string covers research for visualization of traceability links between software artifacts in general. And finally, the visualization of traceability links between tests and code is covered in the fourth search string.

Search for Software Artifact And Source Code Or Software Artifacts And Test Case Literature

The final search string for the literature of software artifacts-to-source code (e.g. requirement, documents) and software artifacts-to-test cases had three terms. The first term is split into two terms, each concerning either source code (using term 1a) from Table 3.1 or test case (using term 2b). The second terms address the software artifact, while the third term implies that the traceability links between these artifacts are considered (see Table 3.1). All terms should have included in the title or the abstract of the publications.

Table 3.1: Extracted Search Terms for Source Code and Software Artifacts

Search Term		Constraint
Term 1a	Source Code OR Code	Title, Abstract
Term 1b	Test Case OR Unit Test OR Test	
AND		
Term2	Software artifacts OR Requirement	Title, Abstract
AND		
Term3	Traceability Links OR Links OR Traceability	Title, Abstract

Search for Test Case-to-Source Code Literature

The final search string for the literature of test case and source code had also three terms. The first term includes test case, while the second term includes source code. The third term implies that the traceability links between these test cases and source code are considered (see Table 3.2). All three terms should have included in the title or the abstract of the publications.

Table 3.2: Extracted Search Terms for Test case AND Source Code

Search Term		Constraint
Term 1	Test case OR unit test OR test	Title, Abstract
AND		
Term2	Code or Source code	Title, Abstract
AND		
Term3	Traceability Links OR Links OR Traceability	Title, Abstract

Search For Visualization Of Software Artifacts Traceability Literature

The final search string for the literature of visualization of software artifacts had four terms. The first term includes software artifacts, while the second term includes a specific term of artifacts (i.e. requirements, source code, and test case). The third term implies that the traceability links between these artifacts are considered (see Table 3.3). The fourth term ensures that the visualization of traceability links between software artifacts are considered. All terms should have included in the title or the abstract of the publications.

Search For Visualization Of Test And Code Traceability Links Literature

The final search string for the literature of visualization of Test and Code traceability links had four terms. The first term includes test case, while the second term includes source code. The third term implies that the traceability links between these artifacts are considered (see Table 3.4). The fourth term ensures that the visualization of traceability links between tests and code are considered. All terms should have included in the title or the abstract of the publications.

Table 3.3: Extracted Search Terms for Visualization AND Software artifacts

Search Term		Constraint
Term 1	Software artifacts OR Requirements	Title, Abstract
	AND	
Term 2	Source code OR Test Case	Title, Abstract
	AND	
Term 3	Traceability Links OR Links OR Traceability	Title, Abstract
	AND	
Term 4	Visualization OR Representation OR Presenting	Title, Abstract

Table 3.4: Extracted Search Terms for Visualization AND Test and Code Traceability Links

Search Term		Constraint
Term 1	Test case OR tests OR unit test	Title, Abstract
	AND	
Term 2	Source Code OR Code OR Class	Title, Abstract
	AND	
Term 3	Traceability Links OR Links OR Traceability	Title, Abstract
	AND	
Term 4	Visualization OR Representation OR Presenting	Title, Abstract

3.1.2 Research Identification

To generate a state of the art comprehensive picture of visualizing traceability between software artifacts and especially between test and code, we used different kinds of sources. We utilized the sources of the domain-specific publication SpringerLink, ACM, and IEEE. We also included ScienceDirect source in order to cover several other domains specific sources to ensure research coverage in other less dominant sources as well (see Table 3.5).

The results of each search string are as follows:

- First search results retrieved 584 of papers.
- Second search results retrieved 187 of papers
- Third search results retrieved 261 of papers
- Fourth search results retrieved 101 of papers

Table 3.5: Sources of Literature

Data Sources	
Digital Library	SpringerLink
	IEEE Xplore
	ACM Digital Library
	Science Direct
Search Engine	Google Scholar

3.1.3 First Round of Exclusion

The publications were refined depending on their title and abstract. As a result of this refinement, the first search leads to 57 results, 35 results for the second search, 32 for the third search, and 16 results for the fourth search. After removing duplicates, a total of 120 results were retrieved which, in turn, are too many for thorough review and analysis. As a result, second exclusion was conducted.

3.1.4 Second Round of Exclusion

In the second exclusion, we inspected the abstract, introduction, and conclusion. We excluded papers that did not explicitly related to either source code, test case, or software artifacts. We also excluded papers which had a different concern than the context of traceability and visualization of traceability in software development or engineering or were out of them. Consequently, the process showed that the total number of relevant papers was 45 as shown in Figure 3.1.

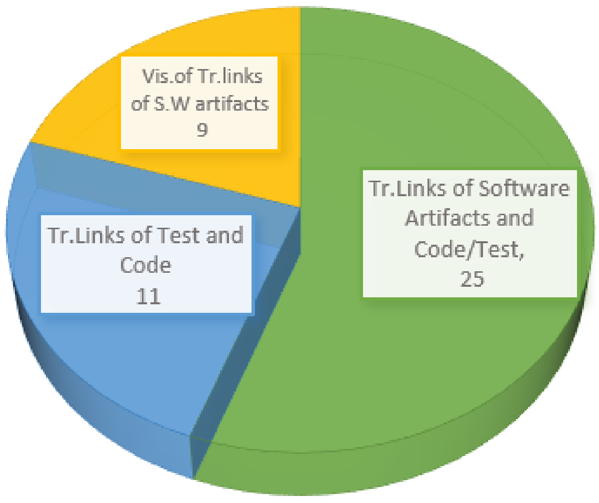


Figure 3.1: Summary of The Identified Papers

3.2 Software Artifacts Relations

Software artifacts relationships can be treated as Traceability links between software artifacts. Software traceability focuses on two aspects of software artifacts' relationships: 1) the ability to follow up the artifact's life throughout its development, validation, and verification, 2) The ability to describe the association links among related artifacts and their artifact's life. The traceability between software artifacts has two dimensions as mentioned in [2], vertical traceability and horizontal traceability (see Figure 3.2). Vertical traceability defines as connections between different software artifacts at the same phase of software life-cycle, while horizontal traceability defines as connections between different software artifacts at different phases of software life-cycle. Our attention in this thesis is on the horizontal traceability dimension.

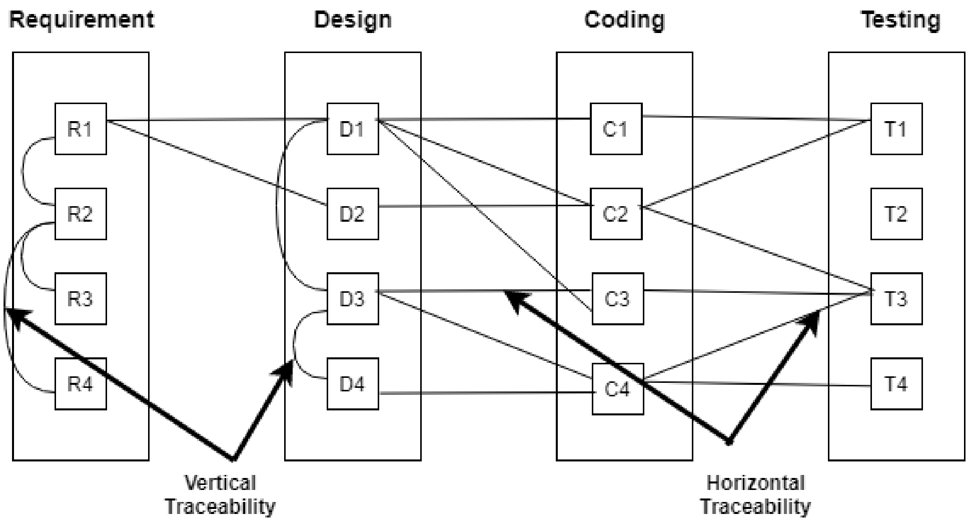


Figure 3.2: Traceability links of Software Artifacts

3.2.1 Source Code and Software Artifacts Relationships

The focus of recent research is the capture of traceability links between requirements and source code during development. Traceability links between requirements and source code can be helpful in several activities in software development process such as software maintenance and reuse. If the software engineers can understand the relationships between requirements and source code, they can easily identify the code elements that implement the requirement they want to maintain and reuse (e.g. change request modifications, fix bug). Requirements-to-code traceability links provides the knowledge of which part of code that the requirements are implemented in.

[23] captures the links between requirements and source code by classifying the requirements and source code according to whether they are specific to one product or common to multiple products using the configuration management log as a source of links. Asuncion and Taylor [60] proposed an approach to establish links between different artifacts (including requirements and code) by analyzing interactions of users during generating or modifying artifacts. Omoronyia et al. [61] recover links between requirements and code based on developer-led operations that create code artifacts to meet requirements.

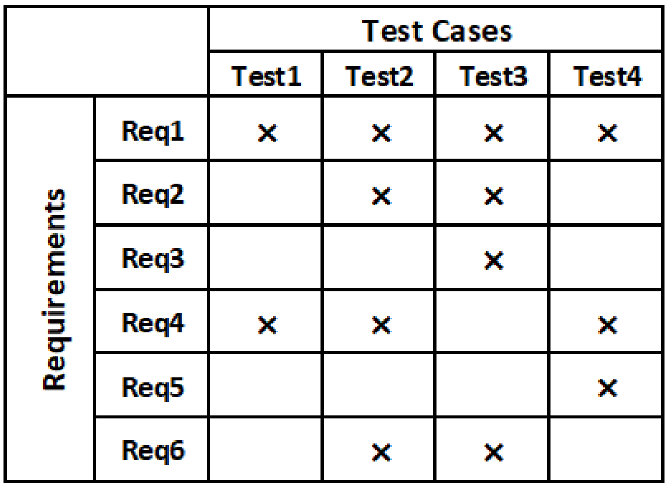
On the other hand, requirements-to-code traceability can help to identify the code parts that directly impacted by maintenance request. Although the traceability links between requirements and source code can be benefit in maintenance and other software development activities, the cost of traceability recovery and management should be taken into consideration, especially in manual recovery of all traceability links of a large product.

The manual recovery of requirements-to-code traceability links is complex, error-prone, and time consuming [62]. Therefore, many researches focus basically on semi-automatic and automatic approaches. Gesamtfakult and Delater [63] summarized 29 approaches for establishing traceability links between requirements and code. The approaches were ordered according to their year of publication, type of automation (automatic, semi-automatic, and manual), and the technique used to create the links (e.g. information retrieval (IR), execution trace, transformation, machine learning, or inference). From 29 approaches, there are 26 approaches use automatic techniques, while only one approach uses semi-automatic and two approaches use manual techniques. Furthermore, among the 26 automatic approaches, the large majority of 73 % (19 approaches) use IR techniques for creation the traceability links, while about 15 % (4 approaches) use execution trace, and the remaining approaches either use transformation, inference, or machine learning.

Traceability links between documents and source code has been a part of research interest as well [64], [65]. Xiaofan et al [66] used IR models to recover the traceability links between documents and source codes. The authors combined three IR techniques to automatically retrieve the links established between documents and source code in a system.

3.2.2 Test Case And Software Artifacts Relationships

A test case is defined as a set of variables or conditions that help to satisfy a set of linked requirements. The creation of multiple test cases can help detect flaws and errors in the specified requirements or the entire application [67]. Test cases usually depend on a use case/ user story that helps ensure that the functionality of system delivered accurately represent the actual needs requested by users [68]. The traceability links between requirements and test cases are the ability to link the requirement back to user's rationales and forward to corresponding test cases. During



		Test Cases			
		Test1	Test2	Test3	Test4
Requirements	Req1	×	×	×	×
	Req2		×	×	
	Req3			×	
	Req4	×	×		×
	Req5				×
	Req6		×	×	

Figure 3.3: Requirements Traceability Matrix

the early stages of software development process, the development team gathers all requirements provided by clients and creates a list of user's stories based on these requirements [69]. These stories can provide the development team with a clear idea about the functionality that should be implemented to satisfy the requirements. When executing test cases, and on the failure of any test case, the developer can map it with the associated requirement.
Requirements traceability matrix (RTM) is a typical method used to create the traceability links between the requirements and test cases (see Figure 3.3). RTM is a table (document or spread sheet) used to validate that all requirements are linked to test cases. RTM is commonly created manually. However, there are some tools have been proposed that automatically support generating RTM [11]. One of the most benefits of using RTM in creating requirements-to-test case traceability links is to make sure of 100 % coverage of requirements.
Latent semantic indexing (LSI) [120] is also employed to automatically reconstruct the traceability links between test cases and requirements during the development process [70], [71].

3.2.3 Test Cases and Source Code Relationships

Coding and testing are very important activities in the software development life cycle. They are firmly associated with agile software development where the software is evolved frequently. Software testing includes test suites that execute the program and an expected outcome declaration [72]. Test cases are used to determine whether the software being tested works correctly or not for a given input.
IEEE Standard 610 (1990) [73] defines test cases as: “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement”. Test case is considered as an up to date document that reflects how parts of code are changed and how they are supposed to be executed [74]. By identifying faults in software systems [75], test cases contribute to improving the software quality and reducing maintenance cost [53]. Therefore, testing is an important activity to guarantee the quality of source code.
Test-to-code relations can be treated as traceability links that display how test cases and the code under test can be connected. Thus these links emphasize the consistency between unit test and tested code (e.g. when a test case fails, the links show which part of the code is related to this failure). Test cases and tested code can be connected by different types of relations, for example:

- Direct tests. When developers produce test classes that only test their counterparts in the production classes. [76].
- Indirect tests. When developers produce test classes contain methods that actually execute tests on other objects. [76].

Test-to-code traceability helps to maintain test cases up-to-date due to changes in the source code in software evolution and maintenance [15]. A new change on a part of code could easily affect the behavior of other parts of code. The problem can be solved using regression tests by re-executing test units to determine if the changes introduce any errors in other parts of system [151]. However, the development process is an iterative and ongoing process, which means new changes and updates appear frequently on the software, thus numerous regression tests have to run (i.e. increasing testing cost). Therefore, establishing links between units under test and its related test suites helps in reducing generating regression tests through facilitating impact analysis, and also used to determine which tests should be checked after the code change which, in turn, save much time and cost during software development [4].

3.3 Overview Of Traceability Recovery Approaches

Several approaches and methods were proposed to recover traceability links between different types of software artifacts. These proposed approaches can be classified according to the method used to retrieve such links. In the following sections, we discuss the definitions of approaches for recovering traceability links between software artifacts in literature. Then, we present the proposed approaches for recovering traceability links specifically between unit tests and tested code.

3.3.1 Information Retrieval- Based Approach (IR-Based)

Information retrieval (IR) methods have been widely used in literature for recovering traceability links between software artifacts of various types. The justification behind such a selection is that the majority of the documentation that comes with large software systems consists of free-text documents presented in a natural language, and a high similarity of text between two artifacts might highlight the existence of a traceability link. Antoniol et al. [77] introduced the use of IR methods to recover traceability between low level artifacts (e.g. source code) and high level artifacts (e.g. requirements). They assume that developers use “meaning full names for code items”, for instance, functions, methods, classes, types and variables. An assumption is that the knowledge of application-domain that developers handle is often captured in the program items when writing the code. Thus, the program items analysis can help to link low level concepts (e.g. code) with high level concepts (e.g. requirements) expressed in free text. Their approach uses a Probabilistic Model (PM) as IR methods. This method looks automatically for the textual similarity between the requirements and code textual representations. They extended this approach in [78] by applying Vector Space Models (VSM), as well as they compared the results with the results presented using PM approach in terms of recall and precision. The approach employed VSM presented slightly lower results than the approach employed PM in their previous evaluation.

Other IR methods have been proposed for recovering traceability links between different types of software artifacts. Marcus et. al. [79] used LSI as IR method for recovering traceability links between documentation and source code. The results of the approach using LSI have been analyzed and compared with the previous approaches using PM and VSM applied by Antoniol et al. [78]. The results showed that the approach using LSI achieved better results than Antolio's approaches using VSM and PM.

Later, additional IR methods, for example, Jensen and Shannon method and Numerical Analysis [80], [81], have been proposed to recover links between software artifacts of different types. In particular, IR approaches have also been proposed for recovering traceability between requirements [82], [22], between design and requirements artifacts [83], [84], between software documents and maintenance requests [85], between design documents or requirements and defect reports [86], between numerous others types of artifacts (e.g., UML diagrams ,use cases, code artifacts, and test cases) [87]– [89] and between unit tests and units under test [6].

3.3.2 Data Mining-Based Approach

Since developers may not evolve software artifacts in synchronization with each other (e.g. requirements and source code), they usually update other sources of information, for example: CVS/SVN repositories, mailing lists, and bug-tracking systems. These sources of information can be exploited to build traceability-recovery approaches. Data mining methods on repositories of software configuration management have been used to recover the traceability links between software artifacts (e.g. source code artifacts).

The authors in [90] were the first to employ release data to detect evolutionary coupling between files and modules. The CVS history allows detecting more fine-grained logical coupling between files, classes, and functions. Their approach searched the

classes' historical development that measuring the time when existing classes are changed and new classes are added to the system, as well as maintaining attributes that related to changes of classes, such as the date of a change or the author. Kadji et al [91] present sequential pattern mining to files that are committed in software repositories in order to uncover the traceability links between source code files and other software artifacts. As different types of files that are frequently committed together, there is a significant chance that they will have a traceability link between them. Moreover, these set of files can be used to predict changes in the system newer versions.
Other approaches [92], [93] used association rule mining on the archives of CVS. They based on the change patterns mining (i.e. files that have frequently been changed together) from the change history of the system source code. Zaidman et al [94] studied the co-evolution of the tests and its related source code by mining the data stored in version control systems (VSC). The assumption is that tests have to be committed in VSC alongside the production source codes. They introduced and combined three views: the change history view, the growth view, and the test quality evolution view in order to study the co-evolution of test and code over time.

3.3.3 Heuristic-Based Approach

Heuristic-based techniques can be used to recover relationships between different types of software artifacts. [95] proposed an approach that relies on the existence of requirement dependent implementation scenarios. They have developed a system called "Trace Analyzer" that detects which code artifacts can be used when executing a usage scenario. These code artifacts are then connected to the usage scenario which is itself connected to one or more requirements. Additional heuristics can be derived to analyze the guidelines for changing design documents and requirements. Clearly, such rules can be used for recovering links between requirements and design artifacts. Techniques of software reflexion model [96] are used to assist a software engineer compare software artifacts by outlining when one artifact (such as code) is consistent with and inconsistent with another artifact (such as design artifact).
In addition, heuristic-based methods have been proposed for recovering traceabil-

ity links between unit tests and classes under tests. The authors in [97] show that the dependent classes need more test code, therefore they are more difficult to test than independent classes. The authors recommend using a “cascade unit tests” to improve the testability of complex class, in which a complex class test can use tests of its required classes to create the complex test scenario.

Eclipse Java Development Tools helps the developers to maintain the relationships between unit tests and test classes, as well as they support a “search-referring tests” entry of menu that recovers all unit tests that call a selected class [152]. Developers have to create a unit test by using Junit wizard in order to employ such functionalities. However, such wizard is not used by all developers to create a unit test and thus these functionalities are not always applied. To remedy Eclipse shortcomings, [98] provide Junit eclipse-plugin that applies Static Call Graph (SCG) to identify the class under test for each unit test.

There are only a few specific sets of automatable tractability recovery approaches have been proposed that are viable means to reveal links between production classes and test units. The methods most utilized and discussed are presented in [6]. The authors suggested six traceability recovery strategies as sources to extract the links between unit tests and source code, naming conventions, fixture element types, LSI, static call graph, last call before assert and co-evolution. In NC, traceability links are established if a unit test matches the name of a tested class after removing the word “Test” " from the name of a class executing the test case. In this approach, traceability links could not be established if unit tests do not match the names of tested classes. In SCG, units under test can be derived by collecting all classes under test that are is directly invoked in test case implementation, and thereafter the classes that are referenced most are selected. In case there are no dominant production classes, The selected sets would contain a possible large range of data object and helper types that will, in turn, impact the precision of the retrieved links. To mitigate the drawback in SCG, authors in [6] proposed Last LCBA method, which derives test classes by checking the last call invoked right before asserting statements. However, if developers write many assert statements per test unit, many units under test could be retrieved. The traceability links can be established in LA approach depending on the textual similarity between test cases and the corresponding unit under test, whereas in CV, the starting point in this approach is the version control system of the software such as CVS, SVN, SourceSafe, or Perforce. This approach requires that changes to code under test and unit tests are simultaneously fetched into the system. Also, developers need to practice testing during development, otherwise the CV information is not captured in CVN edges. The results of comparison [6] showed that NC and fixture element have high precision and recall, while LCBA provided the higher score of applicability as a result of comparing six traceability recovery approaches.

NCs have been described in several books and tutorials [**?**], [118], [119] which is an indication of their widespread usage in different contexts. However, in this approach, traceability links could not be established if unit test does not contain the name of tested class. Naming convention-based heuristics have been used to create unit test taxonomy [99].

Qusef et al [7] proposed an approach that depends on data flow analysis in depicting the links between unit test and classes under test. Test-to-code traceability using slicing and conceptual coupling (SCOTCH) has been proposed in [109], [121] , herein, traceability links are recovered using dynamic slicing and conceptual coupling techniques and test-to-code traceability links are derived using assert statements, then, tested classes are identified in two steps: the first step identifies the started tested sets (STS) using dynamic slicing. In the second step the candidate tested set (CTS) are produced by filtering STS using conceptual coupling between identified classes and the unit test. This approach, however, does not consider the semantics of STS during the coupling conceptual process [5]. Also, the candidate tested classes identified by dynamic slicing still contain an overestimate of the tested classes [5]. To the best of our knowledge, SCOTCH is the one and only approach providing automatic practical support of the test-to-code traceability links.

The authors in [100] present an approach based on a call trace to assess tests for documentation purposes. The approaches proposed and used in this thesis can be classified as heuristic-based recovery approach.

3.4 Presentation Methods

In recent years, research on visualization of traceability links has become very popular. A great effort has been devoted to the use of visualization techniques to help users understand and analyze traceability information. Visualization techniques depict links in terms of graphs, matrix, lists, or hyperlinks due to a particular context to accomplish a task.

3.4.1 Traditional Methods

Charts

Chart is a graphical representation of data in which qualitative and quantitative data can be represented as symbols. There are several types of charts. The most common forms are line chart, pie chart, bar chart and histograms [119]. Presenting data using charts allows users to interpret significant differences at a glance and can easily make comparisons between entities and attributes.

Several developed tools support visualization using charts. Charts could be used based on the traceability links between the data being visualized. A comparison between packages or classes can be a simple relation visualized by bar charts. For instance, Evolve [123] is a visualization tool that is visualising information obtained from system run-time execution. Evolve uses bar chart to clarify the relationship between properties of data. For example, bar chart in Figure 3.4 displays the relationship between the method invocations locations (y-axis) and the total number of invocations taking place at each location (x-axis). Rivet in [124] uses various types of charts for visualizing the execution data of large software system. Pie charts also have been used in several studies as a visualization method (e.g. PABLO [124], PARvis [126], and VAMPIR [127]). Charts can be a good method that leads to faster decision making [128].

Matrix

Matrix is a two-dimensional representation in a form of table. It is frequently used to visualize relationships between software artifacts [155]. It is commonly used in visualizing requirement traceability links with other artifacts (see Figure 3.5).

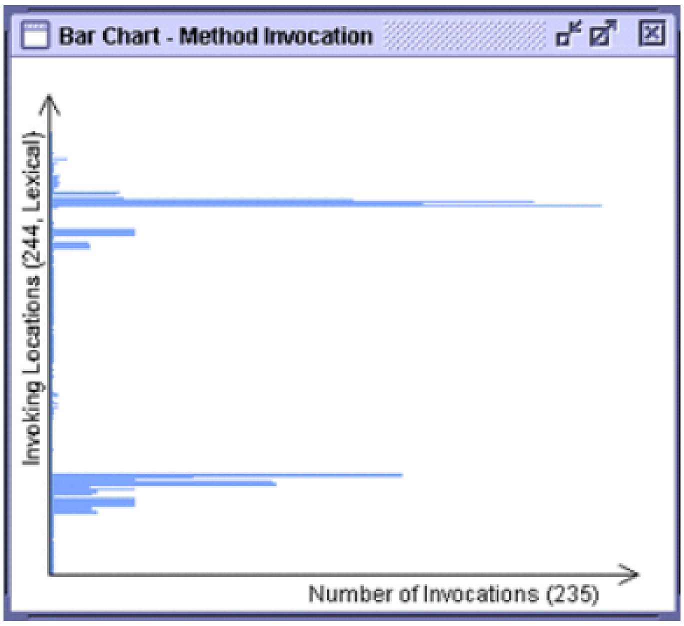


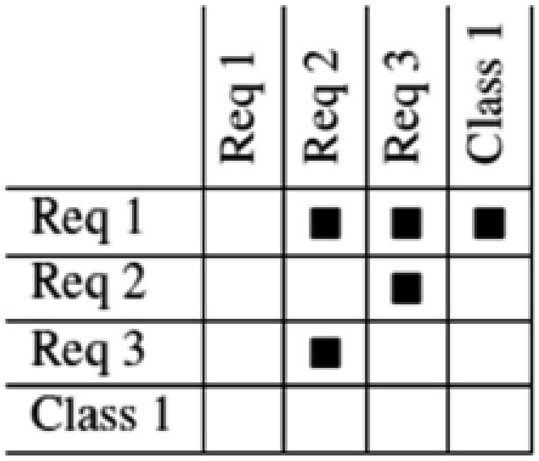
Figure 3.4: Bar Chart Visualization Technique from [119]

A number of tools have been presented to support matrix visualization method [11], [104]. In [11] two approaches have been proposed to automated generation of requirement matrix. In this work, visualization of traceability information using requirement traceability matrix (RTM) helps in determining requirements dependencies in an effective way. Typically, matrices are selected to visualize a small volume of traceability information. As presented in [129] matrix is more suitable to support management task. However, it becomes unreadable when the set of artifacts becomes large because of visual clutter issues [103].

Tables can easily manage the visualized information for the small number of entities and attributes. However, it is a big challenge to display the large number of attributes and entities in which the chance of confusion between columns and issues in sorting the important features in table will increase.

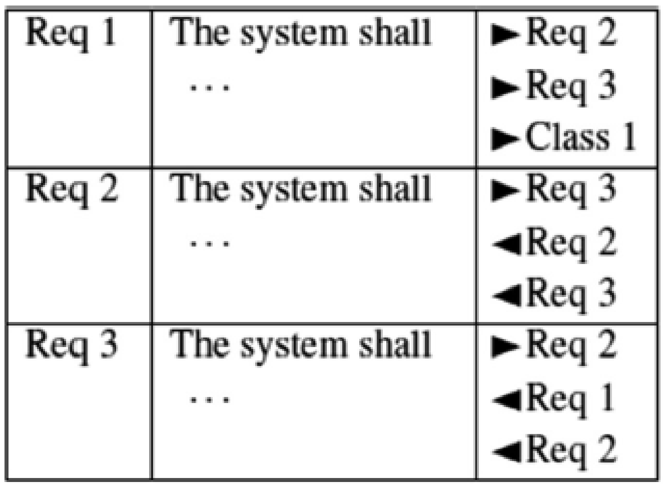
Hyperlinks and lists

Traceability links can be also presented as hyperlinks and lists. Hyperlinks are presented to user in tabular formats using natural language [21] (see Figure 3.6). This method of visualization allows user navigating between related artifacts along link. A number of tools have proposed hyperlinks as a method for representation traceability information [130], [131].



	Req 1	Req 2	Req 3	Class 1
Req 1		■	■	■
Req 2			■	
Req 3		■		
Class 1				

Figure 3.5: Requirement Traceability Matrix



Req 1	The system shall ...	►Req 2 ►Req 3 ►Class 1
Req 2	The system shall ...	►Req 3 ◄Req 2 ◄Req 3
Req 3	The system shall ...	►Req 2 ◄Req 1 ◄Req 2

Figure 3.6: Hyperlinks Visualization Technique from [21]

Hyperlinks are more preferred in testing and implementation tasks than lists [129]. Lists are the least method picked out for representation traceability links among other traditional approaches [129]. All the information related to each traceability link is presented in list view (source artifacts, target artifacts). The authors in [108], [24] proposed tools that use lists to represent automatically produced traceability links. Like other traditional methods mentioned above, hyperlinks and lists do not scale well with large volume of data.

3.4.2 Graph-Based Visualization Methods

Graph

Graph is one of the most common techniques used by traceability visualization systems. It is a group of nodes (or vertices) and edges (or links). Graph-based visualization allows visualization of all overview of traceability links between various software artifacts. Graph can be easily used to represent the trace data [156]. For instance, nodes can represent artifacts such as classes, subsystems, objects, while the edges can represent how these artifacts can be connected (i.e. the relationships between artifacts), such as routine calls, and inheritance. Program explorer [132] provides graphs in two views: “object graph” and “class graph” (see Figure 3.7). “Class graph” displays how objects interact with each other. The nodes represent the objects for a given class, and edges (or arrows) represent interactions between the objects. Recently, an empirical study has analyzed and compared four common visualization techniques [graph, matrix, lists, and hyperlinks] to indicate which one of these visualizations is more appropriate to be performed in a particular context [129]. It pointed out that graphs are adequate to support management task. Graphs can model and visualize any type of data that holds information about connections. A tool, developed by Kamalabalan et al. [133], traces links between software artifacts and visualizes those links elements as a graph with nodes and edges. A specific graph-based approach called ChainGraph has been proposed to visualize relationships between requirements [12].

Although graphs can visualize the overall overview of links between artifacts, it is a big challenge to display large number of traceability links between software artifacts because of scalability issues, which make graphs hardly readable or understandable by users. Thus, graphs are particularly suitable and valuable to present a limited set that is related to interest artifact.

UML Diagrams

UML diagrams are diagrams based on the Unified Modeling Language (UML) that allows users to visualize software and system design. Diagrams are better suited to document systems than texts in millions of lines of code because they are easier and

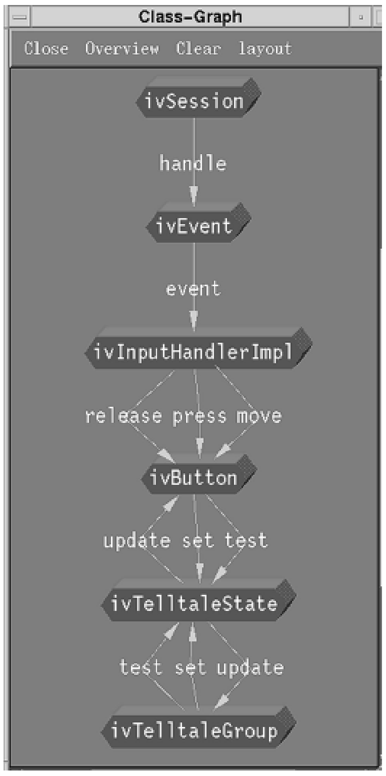


Figure 3.7: Class-Graph View [132]

faster to understand. There are two main categories of UML diagrams:

- Structure diagrams: display the static relationships between the system components.
- Behavior diagrams: show the interactions between the components in the system, capture the changes of the system, and how it changes over time (in some diagrams).

There are 14 different types of UML diagrams [134]; each type is used in different situation. The most popular type of UML diagrams used to visualize the execution of components interaction is the sequence diagram. In [66], sequence diagram has been used to visualize the links between artifacts in a traced project as shown in Figure 3.8. The UML diagrams have been used by different tools [134] as visualization method to show the execution traces in systems and software in different levels (object interactions, class interactions, process interactions).

Traceability information includes information about the artifacts and links to be visualized. Class diagrams can be an example of artifacts which are created during

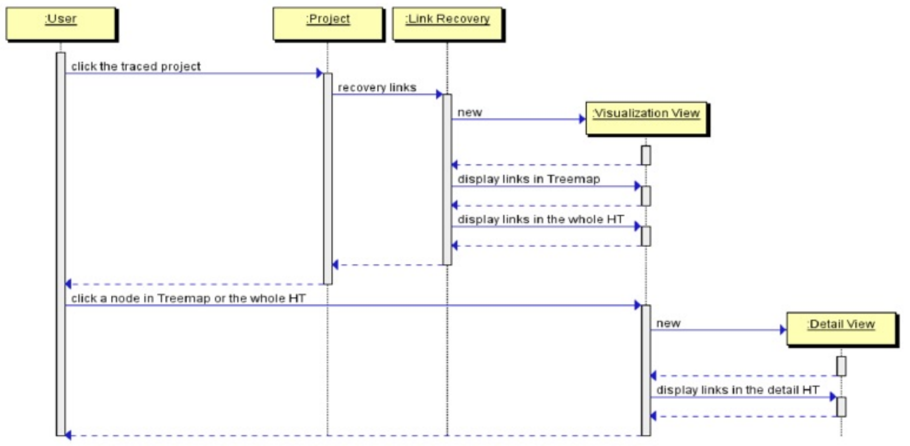


Figure 3.8: Visualizing Links in Project Using Sequence Diagram [66]

the software life cycle [129].

Trees

Trees are one of the significant visualization methods to provide the analysis of data of large hierarchical structures [136]. Tree is a special type of graph that has no cycle (see Figure 3.9). Commonly, the structure of tree consists of number of nodes and parent-child relationships. Every node has just one parent and a number of children. A node that has no parent is called a root node. Nodes in a tree are connected together with line connections called edges that represent the relationships between nodes. Nodes with children are called interior nodes. while leaf nodes are the nodes which have no children.

As the circles are absence in trees, and the hierarchical nature of them, this makes trees easy to understand and interpret compared to the graphs [137]. Ovation tool [138] uses tree structure-based view called "The execution pattern view" to visualize the program execution traces that allows users to browse the program execution at different levels of detail. In general, graph-based-visualization methods help to visualize different aspects of system. However, they tend to be not easily comprehended as the complexity of the system increases. Marshall in [139] categorized the graph into 4 groups according to the number of components (nodes and arcs):

- Small graphs: graphs with less than 100 components.

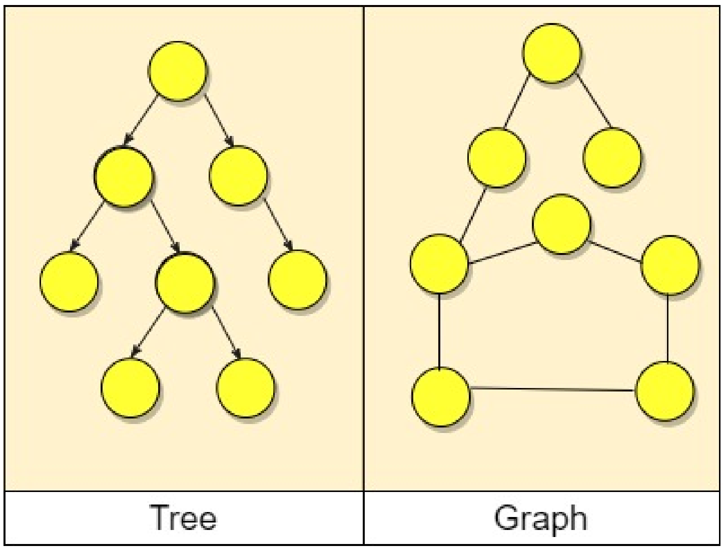


Figure 3.9: Non-Linear Data Structure

- Medium graphs: graphs with less than 1000 components.
- Large graphs: graphs with less than 10000 components
- Huge graphs: graphs with more than 10000 components.

Graphs lose their efficiency when managing large number of nodes. It is a big challenge to display the growing amount of information with the lack of visualization space. Shneiderman et al. [140] indicate that the small graphs are the successful graph representation as the users can trace the link from the source node to the destination node and count the number of links and nodes in each path.

3.4.3 Space-Filling Representation Methods

TreeMap

Treemaps are ideal visualization methods used to present a large amount of hierarchy structured data (structured tree) that show the distributions of the attributes rather than the relationships between nodes [10]. The visualization space is partitioned into rectangles, each rectangle represents a node and it is sized, ordered, and colored (in color-code treemaps) by quantitative variable. In the hierarchy of the treemaps, levels are displayed as rectangles inside other rectangles. A collection of rectangles in the hierarchy on the same level represents an expression or a column in

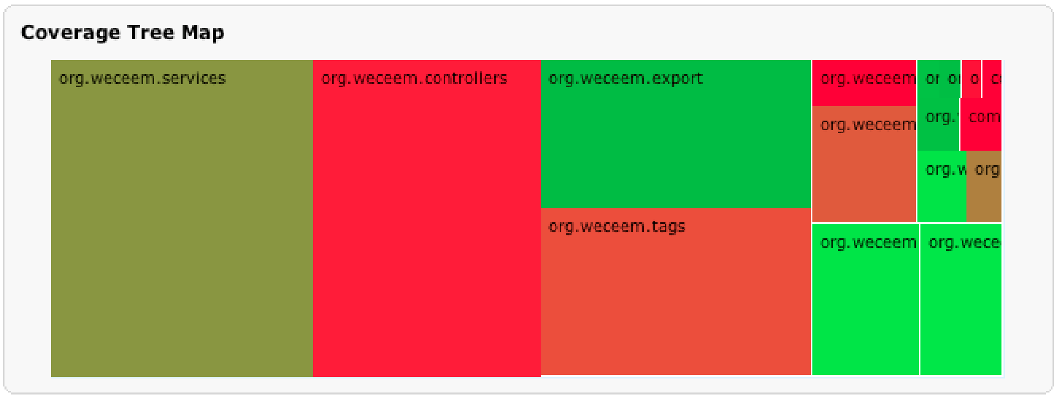


Figure 3.10: Coverage TreeMap in OpenClover

the data table, as well as, every single rectangle displayed on a level in the treemap hierarchy represents a column category. One of the main benefits of treemaps lies in the efficiency using of the visualization space. Treemaps can scale well with the growing complexity and size of the designed system. However, treemaps tend to be hard to comprehend as the complexity and the size of the system grows. Different treemaps layouts were proposed to overcome this issue.

Treemaps have been first introduced by [141]. They used treemap to visualize the file structure on the hard disk in order to discover the large files that can be removed for disk cleanup. Many developed tools provide treemaps presentation to represent large volumes of code and show the code structure as treemap hierarchy. For example, OpenClover [157] is a tool measuring the code coverage for Java and Groovey. It uses treemap to visualize the complexity and code coverage of the classes and packages. Treemap is split up by a labeled package and further divided by an unlabeled class (see Figure 3.10). The size of packages or classes represents their complexity, while the colors represent the level of coverage. In DCTracVis [66], treemap is one of the two visualization method adopted in the tool to visualize the traceability links between software artifacts. In this tool, treemap provides an overview of interrelationships between source code and documents in the traced system. Colors are used to differentiate the link status of each node (see Figure 3.11).

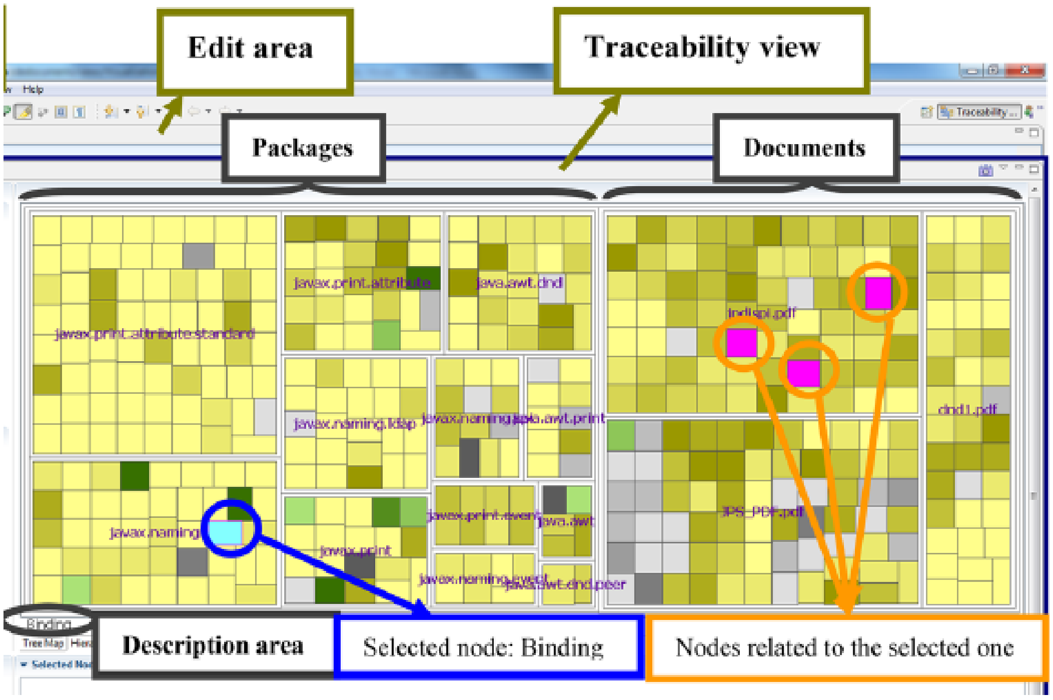


Figure 3.11: An Overview of Relationships between Source Code and Documents [66]

SunBurst

SunBurst is a space-filling approach that displays the hierarchy structure through a set of rings [104]. Each ring represents a level in the hierarchy, with the central ring represents the root node, while children nodes moving outwards from it. SunBurst is similar to treemap technique, but it uses a radial layout. It can be used to display a part item of a whole. For example, Figure 3.12 show the visualizing of world population using SunBurst. Clearly, the world is divided into continents that represent the innermost circle with the four top levels (Asia, Africa, others). The continents are divided into regions, and the regions are divided into countries which are displayed at the outer part of the circle.

Suburst can also be used to represent the traceability links between software artifacts. Merten [10] used the sunburst as a visualization method that shows the traceability relationships between requirements knowledge. Sunburst displays the structure hierarchy of the system under trace. Nodes are organized in a radial layout and are presented on adjacent rings representing the tree structure.Multiviso [103] utilized sunburst along with three other visualization techniques, graph, tree, and matrix, to visualize traceability information in software development (e.g. requirement information).

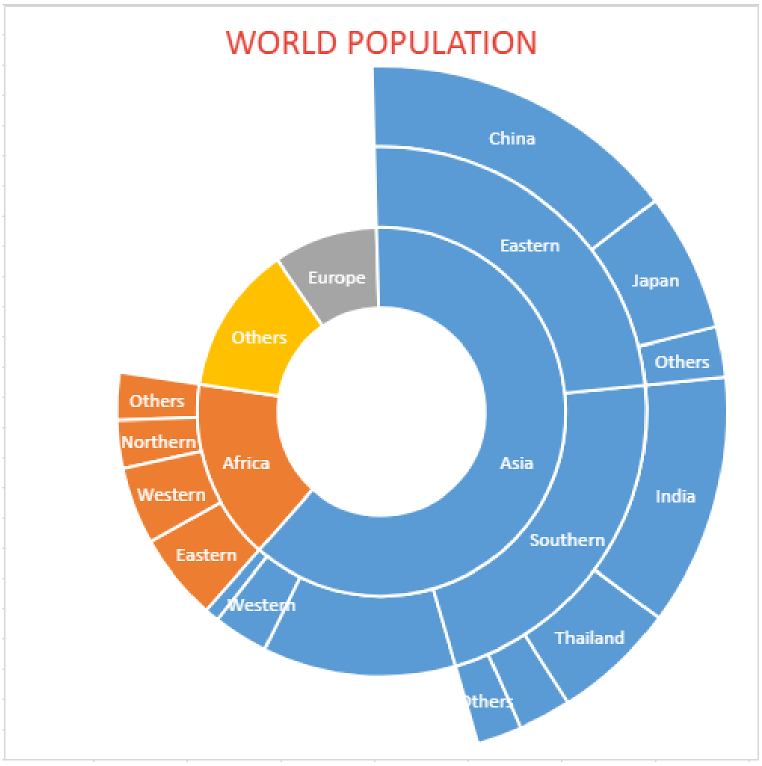
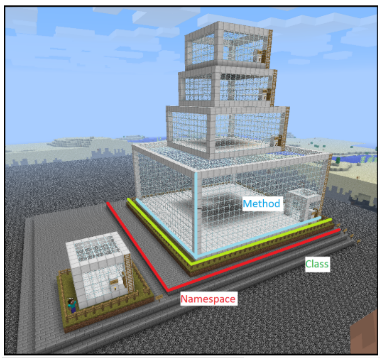


Figure 3.12: Sunburst of World Population [127]

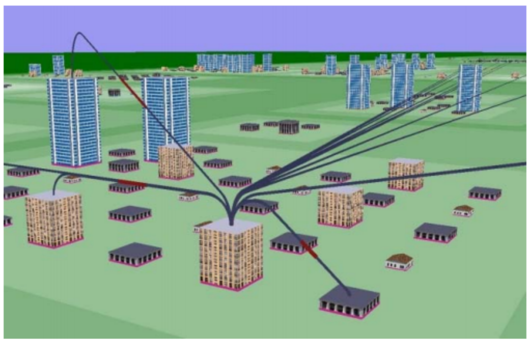
3.4.4 Virtual Reality Environment (VRE)

Recently, metaphor has become a key concept where ideas or objects (lower level of abstraction) are used as a representative or symbol of other things (higher level of abstraction) which are different from their actual meaning [158]. Metaphor is a figure of speech providing mapping from software model to an image in which software entities and relationships are visualized using physical properties; for instance: the solar system metaphor (using stars and planets) [143], neural networks metaphor [144] video games metaphor [145].

One popular VRE is a City metaphor, is a prevalent metaphor in software visualization where the software is represented as city [146]. The idea of city metaphor was exploited in [147] and [13]. They used the city metaphor in which the files and classes are displayed as buildings in a 3D city landscape as shown in Figure 3.13 a. Code metrics display the distinct attributes of a software system. The relationships between buildings are shown as directed pipes between them (see Figure 3.13 b).



(a) City Layout [44]



(b) Classes' Relationships [147]

Figure 3.13: City Metaphor VRE

VRE's can be easily used specifically by the pre-existing knowledge of users about environments. However, as the amount of data being visualized grows, VRE's can be impractical.

3.4.5 Visualization of Traceability Links in Literature

In recent years, research on visualization of traceability links has become very popular. A great effort has devoted to the use of visualization techniques in order to help users to understand and analyze traceability information. Visualization techniques depict links between software artifacts due to the context to accomplish a task. Visualization techniques and tools have been developed depending on the type of traceability information being visualized and the visualization targets. For example, to understand the dependencies and relationships between software artifacts, how they interact with each other, and help document links between several kinds of software artifacts (e.g. requirements, tests) [101]. In Table 3.6, a set of traceability approaches and tools are listed. Each approach provides one or more visualization techniques which may display links in different ways depending on the information task context.

ADAMS [9] is developed to support identifying traceability links between pairs of software artifacts. Traceability links are arranged in a graph where nodes are the artifacts and edges represent traceability links. The graph can be built after the user pick out the source artifacts. The graph starts from a source artifact by defining all the dependencies of a particular type that involve the source artifact as a source or target artifact. Users, within the graph, can identify groups of artifacts that connected by traceability links (i.e. traceability paths). This graph can display all links for a specified artifact in a very effective way. However, it does not support the display of links of multiple software artifacts.

A hierarchical graphical structure is presented by Cleland-Huang and Habrat [102] to visualize links between requirements information, where requirements are represented as leaf nodes while internal nodes represent titles and other hierarchical information. The graph visualization provides a general view of the candidate links, as well as their distribution throughout the set of traceable artifacts. User can explore sets of candidate links that naturally occur together in the hierarchy of document.

ChainGraph [12] has been proposed to visualize the requirements relationships by representing requirements as nodes, and edges are the relationships between requirements. This approach enables the extensible and flexible representation of multi-dimensional requirements links and thus allows a better understanding of these links.

Merten et al. [10] present interactive Sunburst and Netmap representations as a way to visualize traceability links between the elements of requirements knowledge. Sunburst supports the visualization of the hierarchical structure of the project under trace. Sunburst nodes are displayed on adjacent rings representing tree view. Netmap, in the other hand, supports visualization of the links between requirements. Netmap nodes are represented as segments in a circle and of exactly one ring in the sunburst. Traceability links are displayed in the inner circle using linear edges. In [103], Multi-Viso trace tool provides four visualization techniques: Sunburst, matrix, tree, and graph depending on the context in which the traceability is being applied. The visualization displays a global structure of traceability and a detailed overview of each link. Gilberto et al. [104] present a traceability visualization tool called D3TraceView that enables visualizing information of traceability in different formats based on the purpose of use of traceability information. The tool supports several visualization formats such as sunburst, tree, matrix, list, table, bar, gauge, and radial view. Besides traditional approaches and several graph representations

Table 3.6: Traceability Links Visualization Techniques

Approach	Visualization technique	Traceability information	Tool Support
[9]	Graph	Links between software artifacts	ADAMS
[102]	Hierarchical graphical structure	Requirements information	
[12]	Graph	Requirements relationships	ChainGraph
[10]	Sunburst and Netmap	Elements of requirements knowledge	
[103]	Sunburst, matrix, tree, graph	Links between software artifacts	Multi-Viso
[104]	Sunburst, tree, matrix, list, table, bar, gauge, radial view	Links between software artifacts	D3TraceView
[105]	Colored squares	Links between software artifacts	TraceVis
[106]	Text	Links between software artifacts	Poirot
[107]	TreeMap and hierarchical tree	Links between source code and documentation	DCTracVis

similar to those mentioned above, there are several other techniques used to visualize the traceability links. Marcus et al. [105] studied traceability links between software artifacts and showed how visualization can be important in recovering, maintaining and browsing links between such artifacts. TraceVis [105] uses a map of colored and labeled squares to show traceability links for a particular source or target artifact. A map enables users to clearly display all links of a chosen source artifact or a selected target artifact. Unfortunately, it is unable to show links for multiple artifacts at the same time. Poirot [108], [106] shows results of the trace in text format. It employs confidence levels, checkboxes of user feedback, and tabs that separate probable and unlikely links to help the analyst assess candidate links. Chen et al. [107] integrate two visualization techniques: Treemap and hierarchical tree to support a through overview of traceability and provide a detailed overview of each trace.

3.5 Discussion

Depending on the results of our systematic literature review, we have made various interesting outcomes that are discussed in the following. These outcomes are discussed with respect to our research questions identified at the beginning of this chapter.

RQ1: How are the links between unit test and code under test created?
To answer this question, we have refined and selected those methods that focus on traceability approaches that define only the traceability links between tests and code. In this case, traceability recovery approaches that create links between source code and other software artifacts and conversely between unit tests and other software artifacts were not considered. We also elicited from our analysis those approaches related to IDEs (e.g. Eclipse) which provide some support to browse between unit tests and tested classes, since they depend on technology and require many manual efforts and configurations with less accuracy findings.

After final selection and review of publications, we found that only a few and specific approaches have been suggested and used for traceability links recovery between unit tests and code under test. Notably, most of these approaches have come from the outstanding work [6]. They have compared six traceability recovery strategies. The comparison covers only those approaches relating to test-to-code traceability. The strategies have been evaluated in terms of accuracy and applicability based on three open-source Java programs. The results show that last call before assert, lexical analysis and co-evolution have high applicability; however, they have low accuracy. While naming convention and fixture element types showed high precision and recall, the best results are provided by combining the high-applicability strategies with the high-accuracy ones.

In [109], the proposed approach provided more accurate results than provided in [6]. This approach depends on applying dynamic slicing and conceptual coupling to recover the links between test cases and source code, thus identifying class under test (CUT).

An automated test-to-code traceability approach [59] has been proposed to recover links between source code and test cases on the method level by identifying a "Focal method" under test. Focal method according to the proposed approach is defined as "The last method invocation entailing an object state change whose effect is inspected in the oracle part of a test case is a focal method under test (F-MUT)". The approach included a set of phases to distinguish focal methods under test from other helper methods. The evaluation results of approach pointed out to its accuracy in determining F-MUTs. Identifying F-MUTs actively promote the software evolution and maintenance, as well as support test coverage analysis. However, maintaining and comprehension of these retrieved links is still a challenge task especially in case of a large and complex software system.

RQ2: What supporting tools are usually used for the creation of links?

In reviewing the state of the art in this thesis, we observed the lack of providing appropriate tool is one of the main problems associated with the current test-to-code recovery approaches. The assumption is, most of companies prefer using manual traceability recovery methods.

SCOTCH [109] is the one and only tool that providing automatic practical support of the test-to-code traceability links. However, it still does not have an industrial strength to handle automated connections between applications and tests. In addition, developing commercial off-the-shelf (COTS) tools would be a growing demand since the results show the absence of COTS tools in this area.

RQ3: To what extent has a visualization of test-to-code relations been investigated in existing studies?

As shown in Table 1 (Section 3.4.5), visualization of test-to-code traceability links does not receive any interest in the studied literature. This implies that works on testing related visualization are still practically limited. One possible reason is that writing tests is considered to be a time-consuming and not interesting task. De-

velopers usually focus more on the development process and activities which are responsible for testing activities.
Recently, in more important projects, developers can not miss testing. However, they omit traceability because, during the development process, they do not feel the need for it. Therefore they do not spend effort on it. Moreover, in spite of the importance of test-to-code traceability links in understanding, maintaining and refactoring code, it is not commonly used, and its scope is highly neglected in software development.
There is a huge requirement to advance test-to-code traceability recovery visualization techniques. The existing approaches have several limitations which make the visualization process rather difficult; for instance, most of the links that could be retrieved using the current methods are either redundant links or missing links. There is no way to recover specific links of high importance. Furthermore, identifying links is mostly a manual task that needs higher time and effort investment.
RQ4: What visualization techniques and tools are proposed to represent test-code relations?
An interesting observation that needs attention that is none of the existing recovery approaches provides support of visualization alongside with traceability links recovering. Moreover, there is not much interest in developing tools to visualize links between test and code. Most of the tools have been developed to visualize relationships/links between requirements and other software artifacts (source code, design, test cases). These tools supported different visualization techniques such as graphs, traceability matrices, hyperlinks (cross-references), and lists.
Visualization of traceability links would be an important aspect, helpful in management and testing tasks, as well as preferred by users. Therefore, this can be a valuable avenue for further research within the traceability community to investigate proper techniques of visualization between tests and code under test.

3.5.1 Open Research Area

The focus of this thesis is on the creation and visualization of traceability links between tests and code. During working on this thesis, A set of questions were formulated which the research then based upon. These questions helped us to reveal specific topics in this area.

Question 1. What is the purpose of visualization?

Visualization must have a purpose. Defining our goal can help in finding proper visualization techniques to be used and appropriate elements to be presented in it. Purpose can be: understand relations, impact analysis, find problems (e.g. bad smells).

Question 2. What is a suitable visualization technique that can be used to display test-to-code traceability relations and their attributes?

There are several possible ways to visualize test-to-code relations including graphs, matrices, hierarchical tree, tree maps, 3D space. Among the available visualization techniques, 'graph-based visualization' and 'traceability matrices' seem to be the most suitable methods for various needs to find traceability links between code and tests. However, the determination of the most suitable method depends on the use case meaning, as the most suitable method may vary from one use case to another. For example, when one tries to check the relations of an item for impact analysis, 'graph representations' and 'hyperlinks' seem to be relevant. On the other hand, if someone needs a broader view to check inconsistencies among the relations, 'graph representation' showing the traceability links inferred using different link-detection techniques in different colors might be a better choice. While if the goal is to represent detailed dependency information, a hierarchical tree can be more suitable. In addition, a 3D visualization also seems to be appropriate to display attributes of various items and relations.

Question 3. What are the criteria considered to choose the best visualization technique?

As an example, the size of a program can be a criterion, and should be taken into account while using any visualization technique. Visualization methods often become too large and thus hard to read and understand in the case of big projects.

Question 4. What is the best recovery approach usable to retrieve the links between test and code?

Several techniques can be used to derive traceability relations, and each technique retrieves a slightly different set of links. For example, NC supports the established links more in class-level, while LCBA performs better at method-level. Depending on the purpose of the visualization and the technique we use, either all links can be visualized, or we should choose a specific visualization method to visualize any one of the links, but which one? This is another open question that can be investigated.

Question 5. What is the level of information details that could be visualized?

In a real-time system, thousands of tests and code items exist. Although it is not impossible to visualize all these at once, this is probably not the best way. Instead, a selective or hierarchical visualization approach seems to be a better choice. For example, instead of method-level visualization, one can show test and production classes or group items based on their relations or some other purposes and visualize the groups only.

3.6 Conclusion

In this chapter, the results of the systematic literature review on the generating, recovering, and visualizing the traceability links between source code and software artifacts (e.g. requirements, documents), test cases and software artifacts (e.g. requirements), and test cases and source code have been discussed. It has been clearly observed that only a small portion of research has been done on visualization of test-code relations and its importance in maintenance, comprehension, evolution, and refactoring of a software system.

Based on the results of the systematic literature review, we defined number of requirements for a new approach:

Req1. Create test-to-code traceability links automatically during the development process.

Req2. Support visualization of the created traceability links.

Req3. Support tool or integrate the automatic traceability visualization and creation with the development process to reduce the work effort required during the development.

Req4. Easy to use and apply in practice.

Req5. Provide an empirical evidence that the proposed approach is more efficient than other approaches.

Chapter 4

Visualization of Trace Approach

This chapter introduces a novel approach that combines multiple traceability recovery approaches to improve the performance of automated traceability recovery between unit tests and classes. We supported these approaches with a traceability visualization technique to allow visualization of the overall structure of traces and a thorough description of each trace respectively. In other words, the proposed visualization displays the traceability links in two levels, class-level, and method-level. We developed an efficient visualization tool, called "TCTracVis", that supports these recovery approaches and can automatically capture and visualize the traceability links between test cases and its code elements.

4.1 Tracing Test and Code Links During Software Development

The main development artifact in agile development is the source code. Unit tests, code, and user stories are usually the artifacts produced during agile development process. Traceability links can provide an intuitive model to describe the relationships between tests and code. These relationships help to improve the process of software engineering in several ways: facilitating program comprehension, system changing safely, artifacts reusing easily [160]. For example, Test-to-code traceability links can be used to determine which tests should be checked after the code changes which helps in reducing regression tests generation. In recent years, research into combining test-to-code traceability links recovery approaches has become very popular, [153], [159], [121], [109], however, it is complex, time-consuming, and error-prone task to manually retrieve test-to-code traceability links. Such links also are often missing in practice due to lost documentation, non-documented linear development, legacy system, frequently changing requirements with less documentation, etc. Moreover,the main issue is how to support the comprehension and maintenance of these links efficiently and effectively? Visualization of test-to-code traceability links can be an effective approach to understand test-to-code relations. It efficiently helps software developers in various software development activities throughout the software development life cycle (SDLC) [110]. However, only small attention of research has been paid to the importance of visualizing the relations between the unit test and class code in maintenance, comprehension, evolution, and refactoring of a software system.

Many traceability recovery methods have been proposed to retrieve links of traceability between different software artifacts [64], [111]– [116]. Some require human involvement [117], [114], while others can generate traceability links automatically [83], [116], [71], [111]. However, no recovery approaches have a potential to automatically and accurately recover all possible links between artifacts. Some possibly important and useful links are missed by approaches, correspondingly, some un-useful or incorrect links are extracted and may confuse developers.

These issues can be notably diminished by using test-to-code traceability recovery approaches that automatically establish and retrieve the links between unit test and unit under test, as well as adopting visualization techniques to present these links in a simple and intuitive way [8]. In this chapter, our focus is on the traceability links between classes in source code and test cases in units test that are designed to test these classes. Our approach aims to provide software engineers with an effective visualization system that enables them to understand, retrieve, and browse traceability links between test and code.

4.2 The Proposed Approach

In order to provide efficient visualization of traceability, we have developed an approach supporting the representation of multi-level traceability links displays a thorough overview of each link. Our approach shows which artifacts are related, visualizing links between test artifacts and tested artifacts at different levels, class-level, and method-level. As we are establishing links on the class-level as well as on the method-level, we use the terms class-under-test, when referring to a tested class, and the terms tested method or method-under-test for the method-level. Furthermore, a tested class is tested by one or more test classes on the class level, on the method level, a tested-method is tested by one or more test methods. To create the test-to-code traceability links, our approach combines multiple automated sources of test and code traceability links. We have designed a traceability visualization system, called TCTracVis, to support the implementation of our approach. Our approach and the supported tool were built based on the questions posed in Section 3.5.1. These questions were re-presented and re-answered in this part of the chapter as the tool was built upon the answers.

1. What is the purpose of visualization?

   In our approach, the main goal of using visualization is to support the comprehension and maintenance of traceability links efficiently and effectively, and help to identify the disagreement between traceability links inferred from different sources [BNN02]. It also, can point out places where something is wrong with the tests and/or the code (at least their relationship) in a specific system, understand which code elements are tested by which unit tests, and diminish bugs while updating the existing features of a piece of software or adding new features to it. Moreover, visualization can help testers and developers to find solutions with their problems in testing and understand the relationships between test cases and the corresponding units under test.

2. What is a suitable visualization technique that can be used to display test-to-code traceability relations and their attributes?

   Defining our goal helped us to find proper visualization technique to be used.

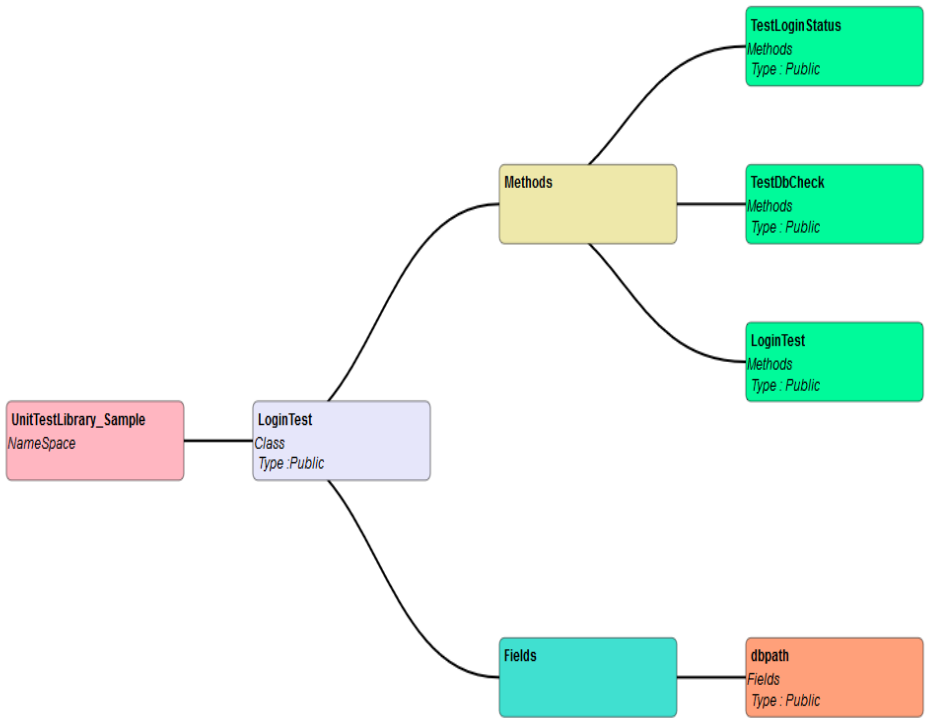


Figure 4.1: Left-to-Right Hierarchical Tree Visualization

We utilize hierarchy tree graph to visually represent a detailed overview of traceability links between a node and its related nodes. A node can be a class, method, or test case. Hierarchy tree can be expanded and contracted to visualize links. We adopt a left-to-right hierarchical tree visualization technique to show traceability links as children of artifacts in the system and to identify trace links for a specific node due to the ease and convenience of browsing and understanding links (see Figure 4.1). Once the traceability links of a selected node are established and retrieved; a hierarchical tree graph is built to show links of nodes that are related to the selected node.

We used a hierarchical tree structure to help us to display the traceability links of a selected item in two levels of dependency information. First level is a class-level, where a test class is connected to all related tested classes which the selected test class was written to evaluate. In this level, the traceability links have the advantages of being bidirectional. This implies that the tested class can be selected to show all test classes that evaluate this class. The second level is a method-level, which provides a more detailed overview of

TCT links. It shows the traceability links established between a class test and its related test methods and methods under tests that the test methods were written to evaluate. Our traceability visualization method provides efficient traceability visualization between unit tests and tested classes. The overview of visualization of multilevel links is presented in Figure (4.2).

3. What are the criteria considered to choose visualization technique?

   The kind of data that we want to convey to users is one of the criteria considered when choosing the visualization technique. As the relationship between test and code is the visualized data in our approach, a hierarchical tree is a good visualization technique that can show the links within these artifacts that are somehow complex to explain with words. Furthermore, we took into consideration how to visualize the traceability links that make it easier for users to understand the traceability information presented.

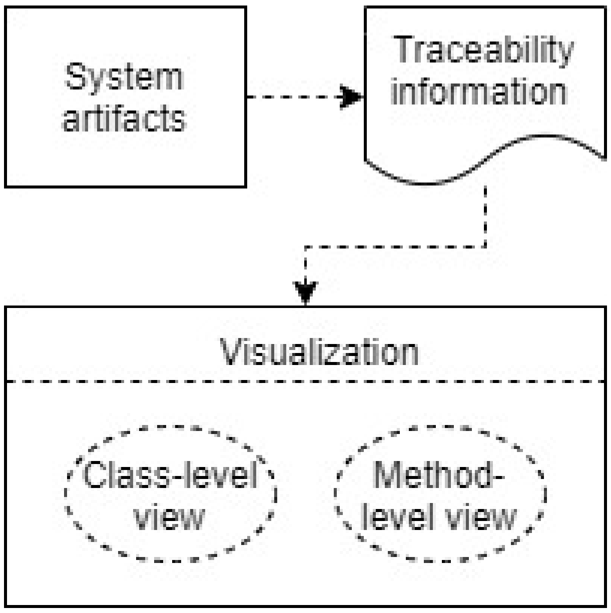


Figure 4.2: Multilevel Visualization Approach

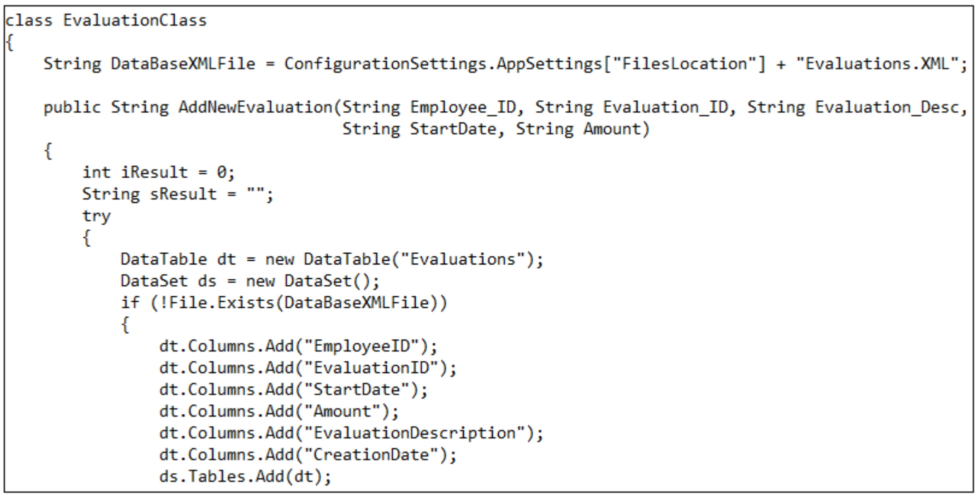
4. What is the best recovery approach usable to retrieve the links between test and code?

   Several techniques can be used to derive traceability relations, and each technique retrieves a slightly different set of links. Still, the issue is that using a single source is not reliable and as such, we use combinations of three of traceability recovery techniques, namely, naming convention (NC), last call before assert (LCBA), and (SCG). NC showed high precision and recall, while LCBA provided the higher score of applicability as a result of comparing six traceability recovery approaches [6]. In [98], static call graph is used to identify the intended class under test by connecting each test with the list of methods that may be called by the test. This also can help in identifying an error location in a failed test case. However, there is no single technique that is superior to all others [6], [5]. In the next section, we define the recovery approaches supported in our tool with an example of each approach.

5. What is the level of information details that could be visualized?
   Although it is not impossible to visualize thousands of tests and code items in real system at once, this is probably not the best way. Instead, a selective or hierarchical visualization approach seems to be a better choice. Our goal is to assist users to present, navigate, and understand test-to-code traceability links. To accomplish our goal, we use hierarchical tree visualization to show, a) a hierarchical view of a selected item (i.e. base class/test class with all its methods and fields) b) base class-to-test case traceability links, and c) test-to-code traceability links not only on a class level but also the trace links on a method level (i.e. methods related to the classes and test cases).

4.2.1 Traceability Recovery Techniques

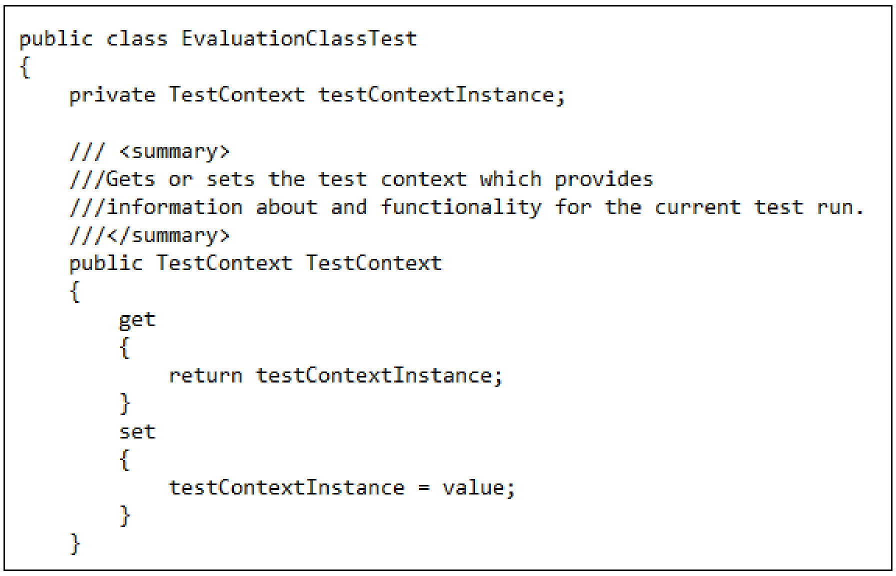
In our tool, we adopt three traceability recovery techniques to retrieve links between unit's test and tested code. These techniques are: Naming convention, Last Call Before Assert (LCBA), and Static Call Graph (SCG). Test-to-code traceability links are recovered automatically in TCtracVis using one of these approaches according to which approach the user selects for recovery.



```
class EvaluationClass
{
    String DataBaseXMLFile = ConfigurationSettings.AppSettings["FilesLocation"] + "Evaluations.XML";

    public String AddNewEvaluation(String Employee_ID, String Evaluation_ID, String Evaluation_Desc,
                                   String StartDate, String Amount)
    {
        int iResult = 0;
        String sResult = "";
        try
        {
            DataTable dt = new DataTable("Evaluations");
            DataSet ds = new DataSet();
            if (!File.Exists(DataBaseXMLFile))
            {
                dt.Columns.Add("EmployeeID");
                dt.Columns.Add("EvaluationID");
                dt.Columns.Add("StartDate");
                dt.Columns.Add("Amount");
                dt.Columns.Add("EvaluationDescription");
                dt.Columns.Add("CreationDate");
                ds.Tables.Add(dt);
```

(a) Fragment of EvauationClass class



```
public class EvaluationClassTest
{
    private TestContext testContextInstance;

    /// <summary>
    ///Gets or sets the test context which provides
    ///information about and functionality for the current test run.
    ///</summary>
    public TestContext TestContext
    {
        get
        {
            return testContextInstance;
        }
        set
        {
            testContextInstance = value;
        }
    }
```

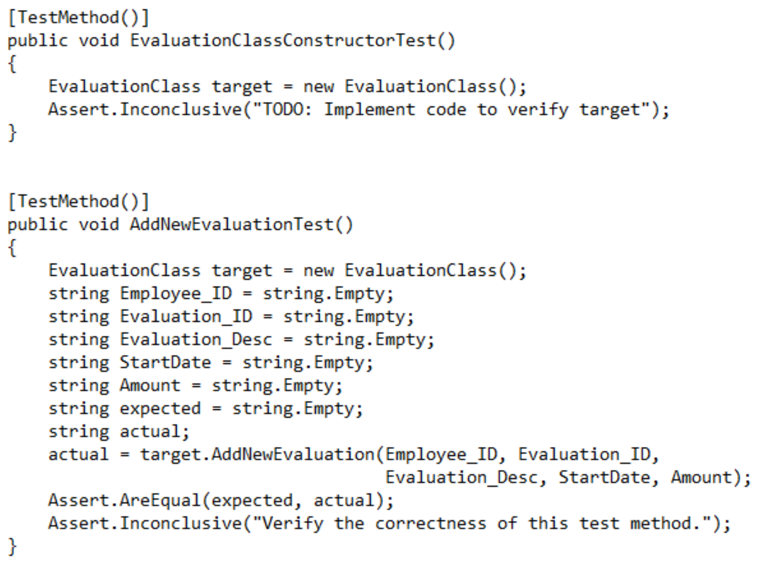
(b) Fragment of EvauationClassTest class

Figure 4.3: Tracing Links Using NC

A. **Naming Convention (NC)**. Over the past years, naming convention was found to have been the best and one of the most frequently used techniques in retrieving links [5], [6]. In our tool, we adopt a derivative of the traditional naming convention [153], which replaces the condition that name of test unit must exactly match the name of tested code, with the more flexible condition that the name of test unit contains the name of tested code. Therefore, the tested class is linked to the test class if the name of the test class includes the name of the tested class after removing the term test from the test class.

$$link(nt, nc) = \begin{cases} \text{True}, & \text{if nc is substring of nt} \\ \text{false}, & \text{otherwise} \end{cases} \tag{4.2.1}$$

Where nt is the name of a test unit and nc is the name of tested code. This approach can perform better if a project does not follow the naming conventions. In Figure 4.3, we briefly describe how naming convention does work. Figure 4.3.a shows a fragment of class *EvaluationClass* which is being tested by *EvaluationClassTest* class as shown in Figure 4.3 b. We can observe that the name of test case provides a hint about the class under test. The link is established if the test case contains the name of tested class after removing Test word. In naming convention, the trace is unidirectional which means that the trace starts from unit test to code. NC can better support the established links in class-level.



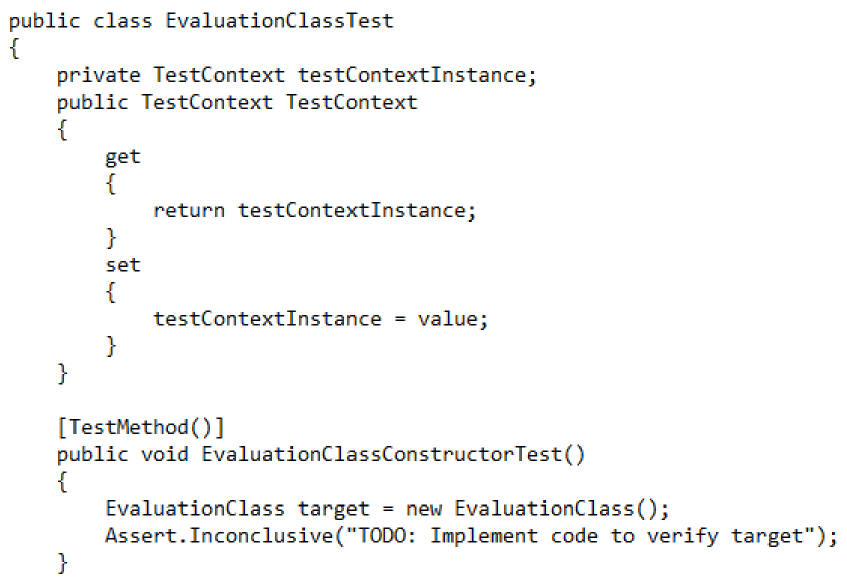
```
[TestMethod()]
public void EvaluationClassConstructorTest()
{
    EvaluationClass target = new EvaluationClass();
    Assert.Inconclusive("TODO: Implement code to verify target");
}


[TestMethod()]
public void AddNewEvaluationTest()
{
    EvaluationClass target = new EvaluationClass();
    string Employee_ID = string.Empty;
    string Evaluation_ID = string.Empty;
    string Evaluation_Desc = string.Empty;
    string StartDate = string.Empty;
    string Amount = string.Empty;
    string expected = string.Empty;
    string actual;
    actual = target.AddNewEvaluation(Employee_ID, Evaluation_ID,
                                     Evaluation_Desc, StartDate, Amount);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Figure 4.4: EvaluationClassTest with a class under test

B. **Static Call Graph (SCG)**. SCG works by inspecting the production class calls in the implemented test unit. The production class referred most is the most likely to be the unit under test. In our visualization system, it counts which classes are called and how many times they are called in each test method. They are stored in a Hash table, then it finds the class invoked most and shows the traceability information. For example, in Figure 4.4, the

EvaluationClass class is referenced twice in *EvaluationClassConstructorTest* by the constructor and *AddNewEvaluationTest*.



```
public class EvaluationClassTest
{
    private TestContext testContextInstance;
    public TestContext TestContext
    {
        get
        {
            return testContextInstance;
        }
        set
        {
            testContextInstance = value;
        }
    }

    [TestMethod()]
    public void EvaluationClassConstructorTest()
    {
        EvaluationClass target = new EvaluationClass();
        Assert.Inconclusive("TODO: Implement code to verify target");
    }
```

Figure 4.5: EvaluationClassTest with LCBA

C. **Last Call Before Assert (LCBA).** LCBA derives test classes by looking at what . In our approach, the statements in each test method are analyzed and searched for classes and methods called test method, and then the test method is linked to tested class if the tested class is last return before an assert statement.

$$link(tm, tc) = \begin{cases} \text{True,} & \text{if tc is last return before assert in tm} \\ \text{false,} & \text{otherwise} \end{cases} \tag{4.2.2}$$

Where tm is the test method, and tc is the tested class. For example, in Figure 4.5, EvaluationClassTest needs to call class EvaluationClass in the statement performed right before assert statement to compare the assert results.

To the best of our knowledge, and according to the previous studies that attempt to recover the traceability links between test and code, the majority of approaches used prefer utilizing manual recovery methods. Our approach is intended to overcome the shortcoming associated with the existing approaches in available traceability recover tool by providing practical support and real-ization of the test-to-code traceability recovery. We adopted lightweight [5] automatic recovery that can be directly executed at run-time and does not require pre-computation of the input. Although the majority of the techniques were manual (from the scratch), the methods listed in the literature and researches are all able to automatically infer links. However, the problem is that using a single source is not accurate [5], and as such, our tool generates links using multiple sources and then visualizes them in a way that allows the developer to compare the links and determine which ones to consider as valid.

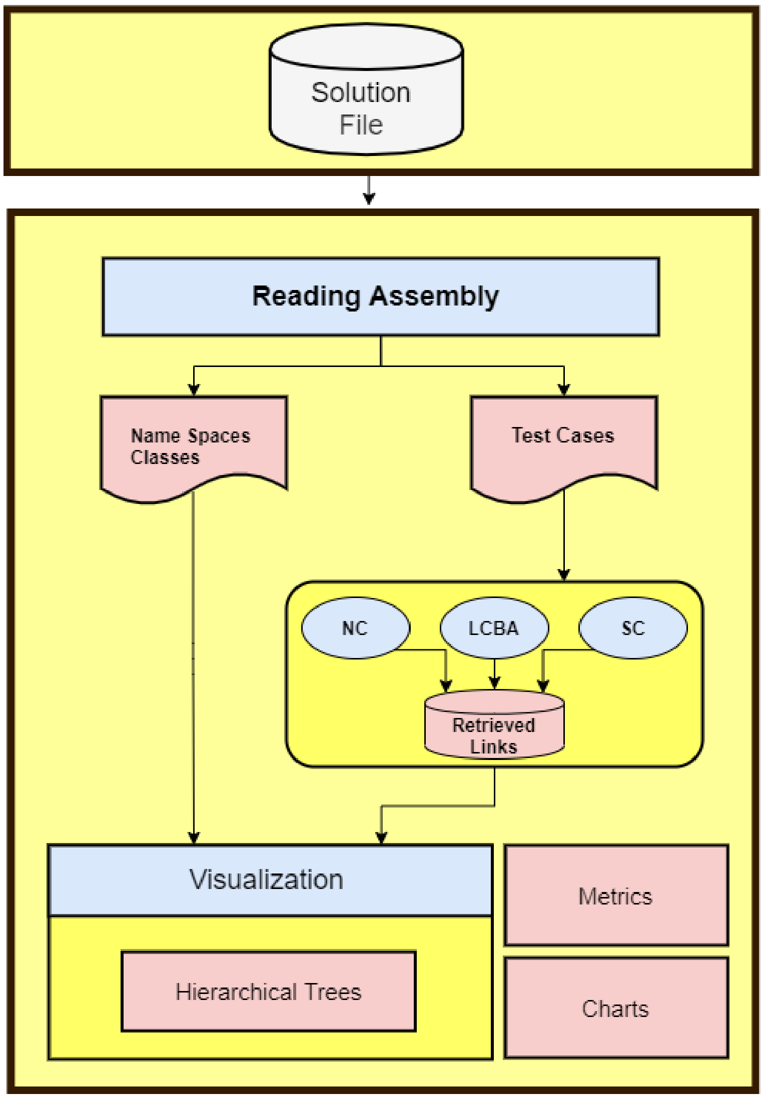


Figure 4.6: Architecture of the TCTracVis

4.2.2 Other Functionalities

Further features in TCTracVis involve some metrics about the traced solution (e.g. no. of base classes, no. of test classes, no. of classes not tested). These information can be visually displayed using several bar charts.
The metrics provide a quick overview of artifacts of the traced solution which can better help to extract valuable information with less effort. For example, one can reveal classes of the code not exercised by a set of test cases. This insight enhances the improvement of testing by creating new test cases for the untested classes. This means creating new links between unit tests and UUT, thus improving the quality of the code, increasing code coverage, facilitating maintenance, and reducing costs.

4.3 Architecture and Design principles

A design model of TCTracVis has been built as a stand-alone desktop application that runs in MS Windows. It automatically recovers traceability links between unit tests and tested code in a project and visualizes these links using hierarchy tree graph visualization technique (see Section 4.2.2). At the time of this writing, this tool is designed to find traceability data of the source code created in C# and the Microsoft unit tests used in it. It supports multiple traceability recovery sources to extract test-to-code traceability links. Figure 4.6 illustrates the architecture of our visualization traceability tool.

Input

The tool requires a solution file as an input, as the tool is designed to find the traceability between classes and test cases of any program /application developed in C# in Visual Studio, the solution file holds the information about all the projects used in source program. To make the performance of tool better, some third party resources are used to read the assembly and IL Codes for the Source Codes, read the C# Source Codes and get the data like classes, methods, fields etc, and visualize the traceability data in hierarchy tree view.

Getting Information on Projects

In the first step, the tool reads the solution file and finds the project files (source code and unit test) used in it. As per the Visual Studio file structure, the project files are in an XML format. Next, the tool searches for C# source code files and test case files (as in Visual Studio a solution file can contain different types of projects all together). In this part, the tool also finds the path of the assemblies which would be read in the next stage.

Reading Assemblies

Next, the tool finds the assemblies and reads them, basically, it reads two assemblies, (1) The assembly created by the source code (name spaces and classes), (2) the assembly created in test cases Studio.

Displaying Traceability Information

Namespaces and classes are loaded into visualization generator which shows the hierarchy of classes and name spaces using a hierarchical tree visualization technique. Our approach can find traceability between test cases and source codes using three mechanisms namely: NC, LCBA, and SCG (for more details see Section 4.2.1). The retrieved links are then input into visualization generator and displayed using hierarchical tree visualization technique. The tool also provides some metrics about the traced solution, these metrics are visualized using several bar charts.

4.4 The Usage Example

Figure 4.7 shows a test case retrieving and visualizing traceability links between production classes and test cases of two solutions written in C#: HRsystem, which is a human resources information system developed by ITG[1] with enough units tests for implementation and evaluation purposes, and UnitTestExample[2] ; which is an open source Windows forms application with main module functionality that is served by several small classes which are used in unit testing. Table 4.1 shows the characteristics of the two C# solutions.

[1]Integrated Technology Group (ITG) https://www.itgsolutions.com/

[2]https://github.com/situ-pati/UnitTestExample

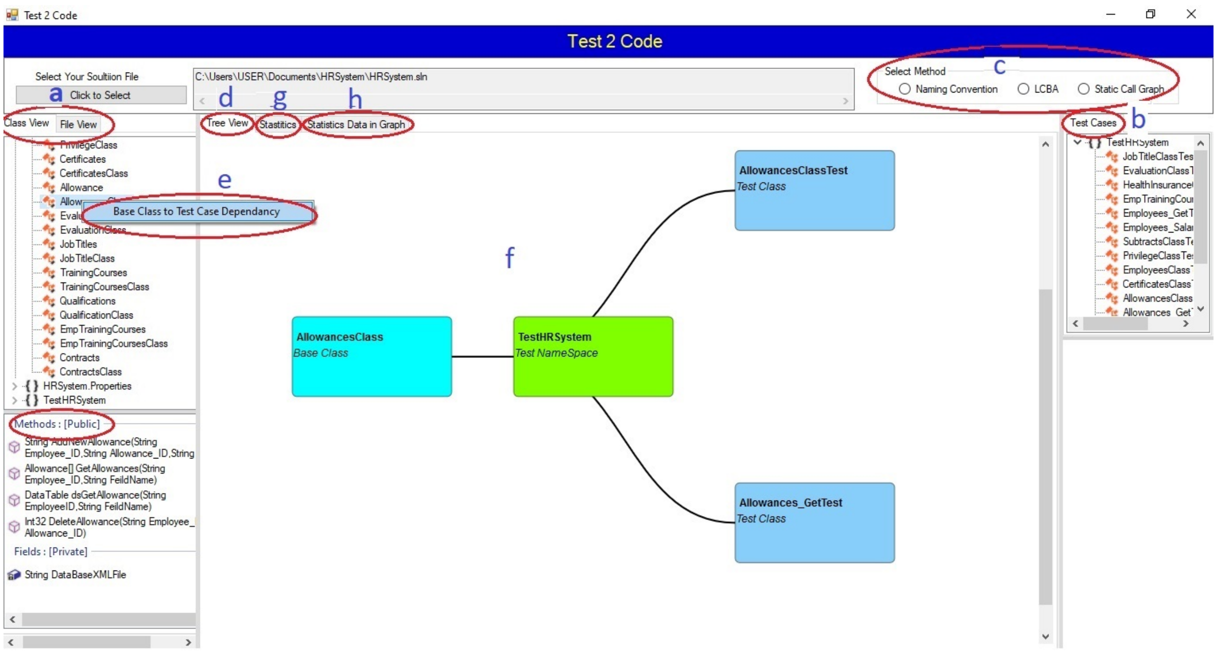


Figure 4.7: TCTracVis User Interface

Table 4.1: Solutions Characteristics

System	LOC	No.of classes	No.of base methods	No.of test classes	No.of test methods	No. of Public Methods used in Test Classes
HRsystem	3208	31	180	15	109	126
UnitTestExample	1912	15	113	5	27	86

Our Visualization tool includes several components as follows:

1. “Class view” and “File view”. In these views all the namespaces (the container that stores the classes), classes, and test classes are displayed. When you click on the class in the class view, a detailed information of all the members (methods and fields) is displayed in the bottom part of this section. The members are arranged by their scope type (Private / Public / Protected) as shown in Figure 4.7.a.

2. “Test cases” (Figure 4.7.b). This section shows available test classes used in the solution.

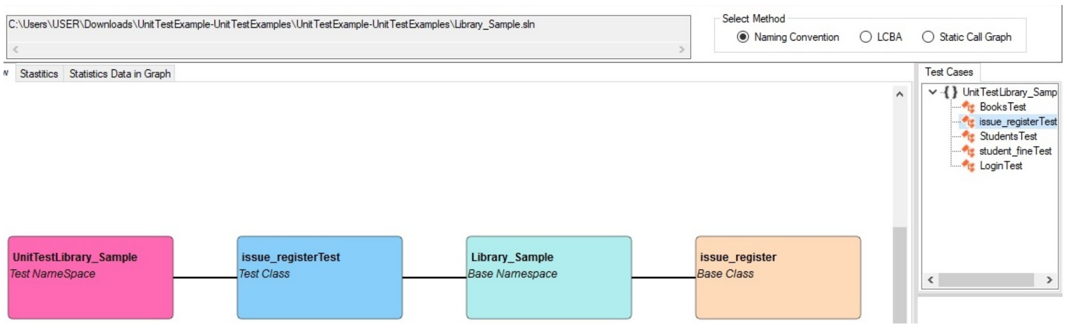
3. “Select Method” (Figure 4.7.c) button. Users can select one of traceability recovery approaches for establishing links between the test cases and tested

classes in the project under trace.

4. "Tree View" (Figure 4.7.d). This view represents a visualization section that shows the hierarchy tree graph of all related links retrieved. This section shows two types of graphs: The Dependency Graph which displays code-to-test traceability links, and Test-to-Code Graph which displays test-to-code traceability links. By clicking on an Item in the TreeView, an expanded list of the following items in the tree is displayed.

   In a class view, a double click on a class shows a hierarchy tree graph for a selected class and all its related methods and fields. When a user right-clicks on the class item in the Class View, a popup menu appears as "Base Class to TestCase Dependency" as shown in Figure 4.7.e. When the popup menu is clicked, a tree-view appears in the tree-view showing the dependency graph for a selected base class (i.e. test classes that call a selected class). (see Figure 4.7.f). In test cases view, a user can initially select one traceability recovery approach, a double click on the test case shows the details in the "TreeView" section as a hierarchy tree graph. A click on a test class in a hierarchy tree graph expands it to show subsequent items related to a selected test class (i.e. base classes and test methods). Figures 4.8, 4.9, and 4.10 show the visualization of test-to-code traceability links for issue-*registerTest* test case from UnitTestExample using NC, LCBA, and SCG respectively.

   In Figure 4.7, *issue-registerTest* test class is connected to issue-register base class by matching their name using naming convention strategy.



C:\Users\USER\Downloads\UnitTestExample-UnitTestExamples\UnitTestExample-UnitTestExamples\Library_Sample.sln

Select Method: [✓ Naming Convention] LCBA Static Call Graph

Test Cases
- UnitTestLibrary_Samp
  - BooksTest
  - issue_registerTest
  - StudentsTest
  - student_fineTest
  - LoginTest

UnitTestLibrary_Sample — Test NameSpace
issue_registerTest — Test Class
Library_Sample — Base Namespace
issue_register — Base Class

Figure 4.8: Traceability links of issue-registerTest using NC

While Figure 4.8 provides visualization of the traceability links of *issue-registerTest*

test class using last call before assert strategy. The Figure shows a set of tested classes which are called by *issue-registerTest* test class in the statements performed right before assert statements in *issuebookTest* test method.

Finally, static call graph strategy is used in Figure 4.9 to establish the links of *issue-registerTest* test class. The visualization shows the production classes that are invoked most in the implementation of test class and the number of times called in each test method.

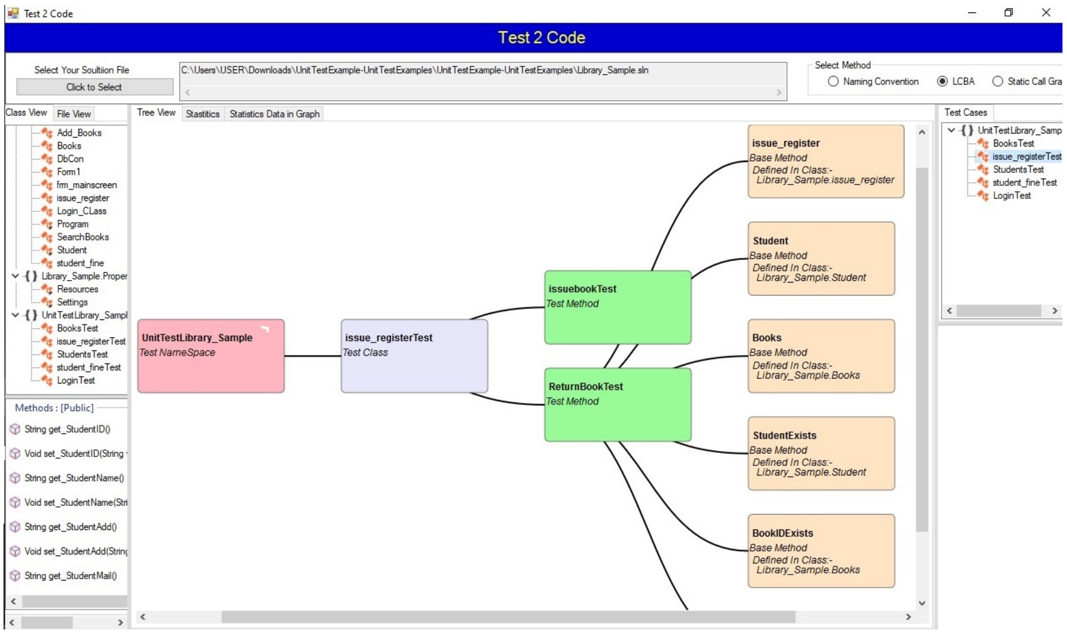


Figure 4.9: Traceability links of issue-registerTest using LCBA

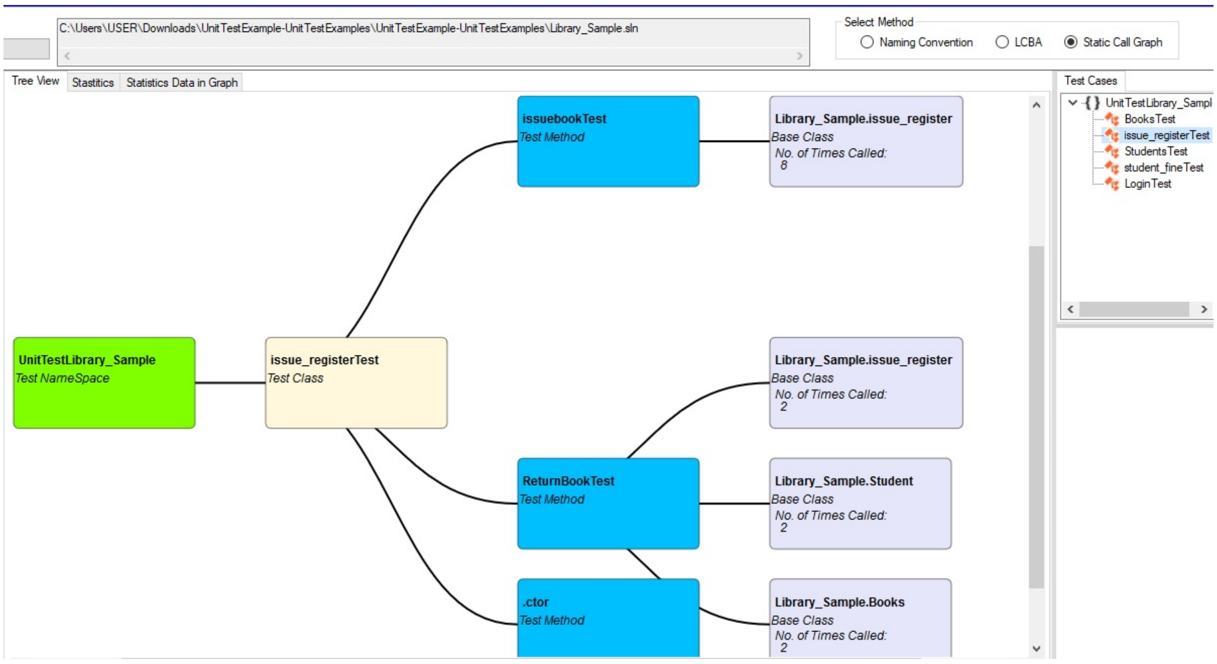
As shown in the previous figures, we can see that test-to-code traceability links from different sources are displayed which, in turn, provide a clearer picture of what is taking place within these tests. Furthermore, a hierarchical tree view presents a detailed overview of traceability links at method level especially with LCBA and SCG approaches.

5. "Statistic View". This view shows the statistics from the current traced solution (see Figure 4.7.h).

    - The first section of statistic view shows the number of base classes in the solution (Private / Protected and Public), number of base methods, number of test cases/ classes, number of test methods, and number of public methods used in test classes (see Figure 4.11).

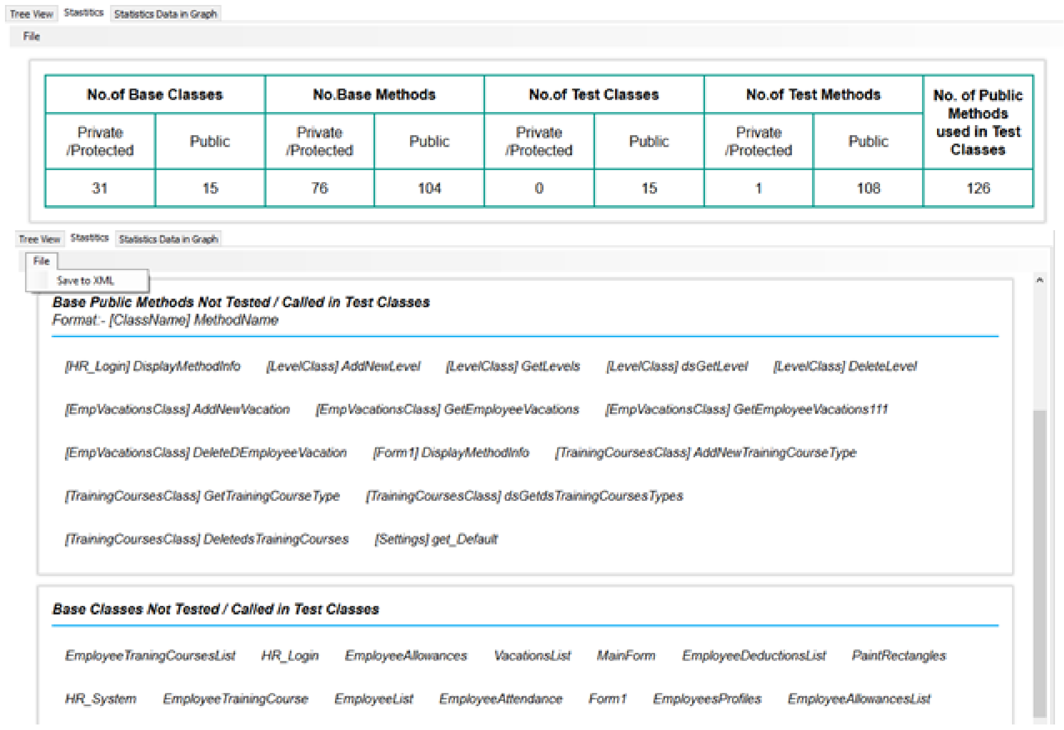
- The second section of statistic view shows an overall overview of base methods (public) not tested in the test cases. These are written in the format [Base Class Name] Method Name. To export the statistics data to an XML file, users can select "Save to XML" menu from the "File" menu available in the "Statistics" view (see Figure 4.11).

6. "Statistics data in graph" (Figure 4.7.g). The statistics data in graph view shows the statistics data in statistics view in graphical manner (see Figure 4.12).


C:\Users\USER\Downloads\UnitTestExample-UnitTestExamples\UnitTestExample-UnitTestExamples\Library_Sample.sln

Select Method: ( ) Naming Convention ( ) LCBA [✓ Static Call Graph]

Figure 4.10: Traceability links of issue-registerTest using SCG



No.of Base Classes		No.Base Methods		No.of Test Classes		No.of Test Methods		No. of Public Methods used in Test Classes
Private /Protected	Public	Private /Protected	Public	Private /Protected	Public	Private /Protected	Public	
31	15	76	104	0	15	1	108	126

Base Public Methods Not Tested / Called in Test Classes
Format:- [ClassName] MethodName

[HR_Login] DisplayMethodInfo *[LevelClass] AddNewLevel* *[LevelClass] GetLevels* *[LevelClass] dsGetLevel* *[LevelClass] DeleteLevel*
[EmpVacationsClass] AddNewVacation *[EmpVacationsClass] GetEmployeeVacations* *[EmpVacationsClass] GetEmployeeVacations111*
[EmpVacationsClass] DeleteDEmployeeVacation *[Form1] DisplayMethodInfo* *[TrainingCoursesClass] AddNewTrainingCourseType*
[TrainingCoursesClass] GetTrainingCourseType *[TrainingCoursesClass] dsGetdsTrainingCoursesTypes*
[TrainingCoursesClass] DeletedsTrainingCourses *[Settings] get_Default*

Base Classes Not Tested / Called in Test Classes

EmployeeTraningCoursesList *HR_Login* *EmployeeAllowances* *VacationsList* *MainForm* *EmployeeDeductionsList* *PaintRectangles*
HR_System *EmployeeTrainingCourse* *EmployeeList* *EmployeeAttendance* *Form1* *EmployeesProfiles* *EmployeeAllowancesList*

Figure 4.11: Statistics Data of the HR System

The results achieved in this chapter satisfy three requirements given in Section 3.6:

Req1. Create test-to-code traceability links automatically during the development process.

Our approach support three traceability recovery techniques (i.e. NC, SCG, LCBA) that automatically establish the links between test and code.

Req2. Support visualization of the created traceability links.

The approach supports hierarchical tree visualization to display the retrieved links. The hierarchical tree view presents a detailed overview of traceability links retrieved by multiple sources of links at method level.

Req3. Support tool or integrate the automatic traceability visualization and creation with the development process to reduce the work effort required during the development. Our approach is implemented by TCTracVis visualization traceability tool.

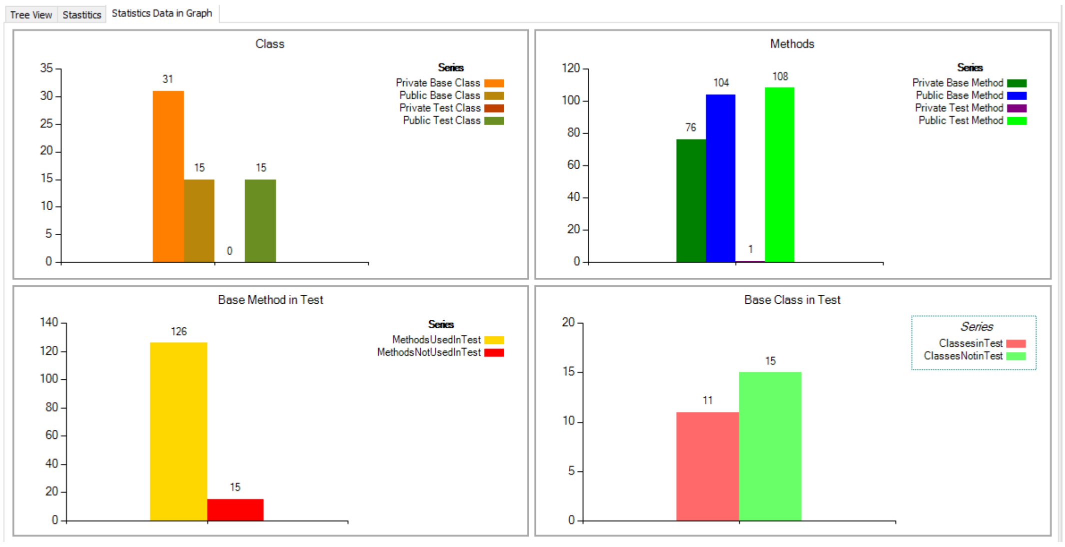


Figure 4.12: Statistics data of the HR system in graph view

4.5 Summary

This chapter introduced visualization traceability approach that integrating two software artifacts (i.e. code classes and units test). Our approach and the supported tool were built based upon the questions formulated in Section 3.5.1. We discussed and answered these questions again in this chapter as the tool was built upon the answers.

The proposed approach composed of the following components:

- The first component introduced the traceability information (i.e. solution file that contains the artifacts and traceability links between them).
- The second component defines three traceability recovery sources that automatically retrieve the links between units test and their related classes.
- The third component presents the visualization technique to visually display the links between the selected elements.
- The fourth component provides statistical information of the traced solution that can be visually displayed as well.

Furthermore, three requirements given in Section 3.6 have been satisfied which contributed to obtaining the chapter results. The requirements are:

Req1. Create test-to-code traceability links automatically during the development process.

Req2. Support visualization of the created traceability links.

Req3. Support tool or integrate the automatic traceability visualization and creation with the development process to reduce the work effort required during the development.

Chapter 5

Evaluation Of Visualizing Trace Approach

The main target of this chapter is to develop an understanding of the effectiveness and usability of our visualization traceability approach in order to justify the effort and time spent in the design of the TCTracVis tool. To testify to the tool, we conducted a usability study. We prepared a set of questions to know how the use of TCTracVis helps its end users in browsing, comprehension, and maintenance of test-to-code traceability links of a software product or project.

5.1 Usability Evaluation

We conducted a usability study to answer the following questions:

- Is the use of multiple-source links visualization better for testers to find solutions to their problems than using a single-source-visualization?
- Does the use of TCTracVis tool help to enhance the overall browsing, comprehension, and maintenance of test-to-code traceability links of a system?

The solution used in this study is the UnitTestExample solution mentioned in Section 4.4. It is worth mentioning that our tool is robust to support large projects, however we selected the UnitTestExample as its small size makes the manual evaluation much easier for the participants.

5.2 Study Context

To answer the questions above, we defined a set of tasks to be performed using our visualization traceability tool. These tasks have been also performed manually in order to measure TCTracVis added value to traditional software engineering processes in manual tracing. These tasks are shown in Table 5.1.

Table 5.1: Evaluation Tasks

ID	Task' description	Motivation	Concern
T1	Understand the structure of HR system (e.g. number of classes, number of methods and test cases, classes type,), and the convention used in the system to organize unit tests	Developer needs to understand the structure of the system, how the classes and test classes are organized. Test classes are usually organized according to a specific convention of a project. Being able to comprehend, one can make maintenance more efficient and improve the built/maintained system.	Structural comprehension
T2	Analyze the change impact of class issue-register, in terms of its related unit tests	Change impact analysis enables an estimation of how a change to a part of the system affects the rest of the system. it's widely used in maintenance activities. Provides an idea of system quality. A part of the system that needs significant change may be a proper candidate for refactoring.	Change Impact Analysis
T3	Find a class with the highest number of linked unit tests	A class can be tested by multiple test classes Refactoring of code needs refactoring of dependent units' test to maintain the consistency between units test and classes.	Design Assessment
T4	Identify a unit test with the highest number of linked classes	A unit test can be used to detect multiple classes. Refactoring of code can be translated to regression testing which is required to make sure that a change code does not impact the existing features of a product.	Design Assessment

5.3 Study Subjects

A group of 24 subjects with varying levels of expertise in software development and unit testing were assigned for the evaluation of our tool and for performing the tasks. Among the subjects were 17 students, 3 from industry, and 4 academics (see Figure

5.1). We divide the subjects into two groups: a control group and an experimental group. The former group is assigned to perform the tasks manually, while the latter group is assigned to perform the tasks using TCTracVis tool.
At the beginning, we provided the subjects with a brief introduction to help them to get familiar with our approach and tasks After the tasks completion, a set of questions on our tool have been answered by them.

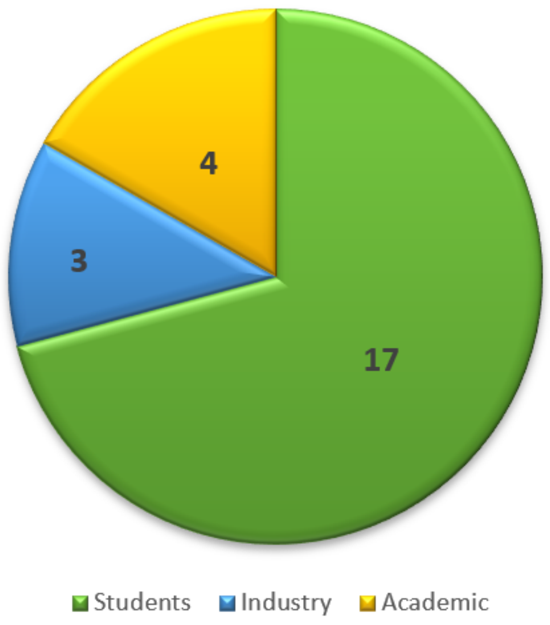


Figure 5.1: Types of Subjects

5.4 Study Results

During the execution of tasks, we recorded the time needed and the number of steps performed to complete each task in each group. It is to be noted that the time factor of the evaluation process was done by using a stopwatch, as the time needed to understand the tasks was taken into consideration. As illustrated in Figure 5.2, the time taken to complete all tasks using our tool varying from (5) minutes to (10) minutes. While the same tasks completed manually with times varying from (45) minutes to (60) minutes. The time varies depending on subjects' experience in software development and, for the experimental group, how often they use traceability tools.
7 of the 12 subjects in the experimental group completed the first task in less than

5 minutes, 3 spent 8 minutes, and 2 took 10 minutes to complete the task. As we observed, the subjects who spent 8 and 10 minutes practiced a little to get familiar with the tool before carrying out the four tasks. The subjects managed to easily understand how the system is structured by using the “statistics “area and statistics charts. Moreover, they easily defined the specific conventions used to organize the unit tests by browsing the “class view” and looking at the NC approach. In contrast, in the control group, the subjects spent around one hour to complete the first task manually.

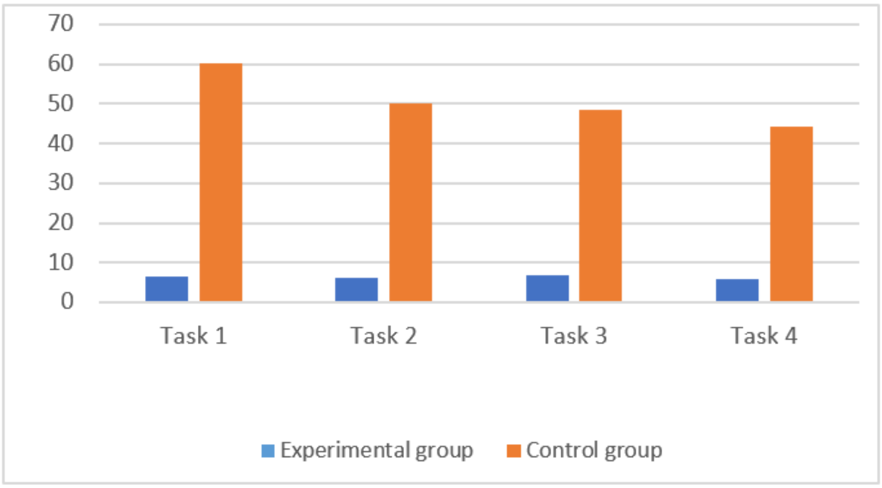


Figure 5.2: Average Time to Complete Tasks

In the second task, we asked 4 subjects in the experimental group to use only a single link source to perform the task, they used “base code to test dependency”. While the other subjects were asked to use all sources. The motivation behind this is to evaluate the efficiency of visualization traceability links inferred from multiple sources compared to a single source. In the experimental group, the subjects completed the second task with times varying from 1 to 3 minutes, as the subjects became more familiar with our tool, whereas, in the control group, the subjects took 45-65 minutes to complete their task. We observed that in the manual evaluation, the subjects made great efforts in analyzing the change impact of issue-register class and detecting its related test cases.
The third task was completed within 5 and 10 minutes by the experimental group,

while the average time in the manual evaluation was 48 minutes. In the experimental group, the subjects easily found a class in a "class view" and identified the number of its linked test cases by using "base class-to-test case dependency" function which displays "base class-to-test case dependency diagram". This took 1 minute or less to complete for a single class. On the other hand, in the control group, as the developer of UnitTestExample followed a specific naming convention, this helped the subjects more to identify the test cases that linked to the classes. nonetheless, this task was tedious for the subjects and required a great effort to complete it manually.
Task 4 was as hard as the task 3 in the control group. The subjects took from 50 to 90 minutes to complete. Whereas in the experimental group, this task was completed easier and faster, and the times varied from 5 to 10 minutes. In the latter, the subjects selected LCBA method to recover the test cases links with the classes and managed to easily identify test cases, display the hierarchical tree of their links, and then identify the number of their linked classes.

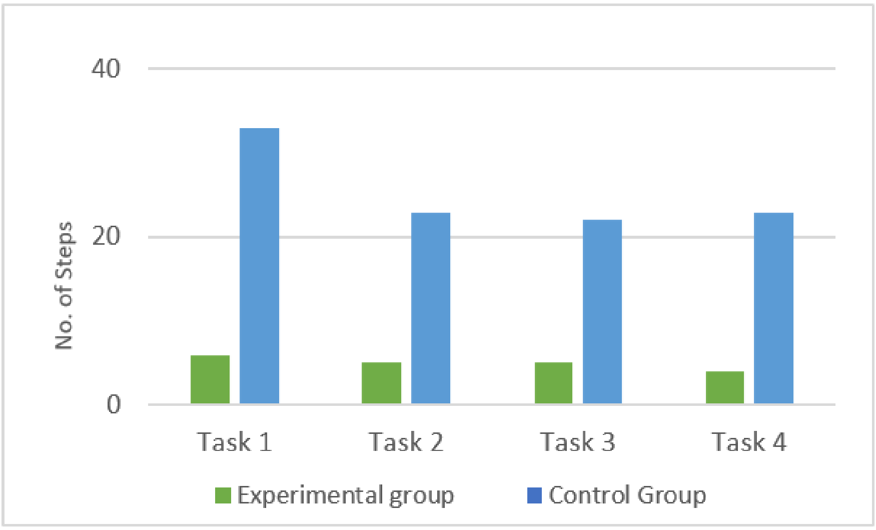


Figure 5.3: Number of Steps to Perform the Tasks

In Figure 5.3, it can be seen that the number of steps needed to perform the tasks manually is much more than the number of steps needed to perform the tasks using our visualization traceability tool, the subjects in the control group took more steps in performing the four tasks compared to the experimental group. During the manual evaluation, subjects often switched between source code files and test cases files to read, perform the task and write down the notes about the artifacts, the links, the time and the number of steps to perform each task. While our tool can effectively provide all required information in a single view.

Table 5.2: Questionnaire used in the experiment

ID	Question
Q1	TCTracVis trace" is easy to use
Q2	Using the TCTracVis tool one can efficiently and easily visualize the class tree of a project.
Q3	TCTracVis trace tool" has the ability to show detailed statistics of various components of a program (classes, methods, unit tests, etc.)
Q4	TCTracVis trace tool" provides clear-cut visualization of "Test-to- code" Traceability links" using various recovery approaches. (NC, LCBA, and SCG)
Q5	In TCTracVis Trace tool it is easy to detect base class's dependency on test cases", thus It provides a rich set of initiations that makes visualization easier to understand.
Q6	TCTracVis trace tool" saves your time in finding Test-to-Code Traceability Links in a project
Q7	visualization of traceability links inferred from multiple sources is more effective than a single source
Q8	Automated Recovery of Test-to-Code Traceability Links helps developers and testers immensely in project development and management than manual recovery methods
Q9	Automated Recovery of Test-to-Code Traceability Links helps time savings in the project, thus, makes a project cost-effective
Q10	Overall, Test-2-Code would be strongly recommended to a developer, a tester, or a researcher.

5.5 The Results of Questions

After the evaluation, the analysis of the main outcomes performed was on a number of questions answered by the subjects in experimental group based on their experience of using our tool. The ten questions are shown in Table 5.1. The main purpose of the questions is to assess the tasks performed by the subjects. Table 5.2 shows the distribution of the questions over the four tasks. Questions 1, 3, and 4 aim to assess the first task. The second task is assessed by questions 5 and 7. Questions 2 and 5 assess the third task. Questions 7 and 8 are concerned with the fourth task. Whereas questions 9 and 10 aim to investigate the usefulness and the importance of our approach from the subjects' point of view.

Table 5.3: Distribution of Questions over Tasks

Tasks	**Questions**							
	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
T1	x		x	x				
T2					x		x	
T3		x			x			
T4							x	x

The results of evaluation questions are shown in Figure 5.5. The questions are shown on the x-axis intended closed answers on the Likert scale from (strongly agree) to (strongly disagree). While the y-axis displays the number of participants and their responses based on the Likert scale (strongly disagree, disagree, neither agree nor, agree, strongly agree). The most interesting result that the majority of responses provided were positive.

9 subjects (strongly) agreed that the tool is easy to use, the visualization of multi sources links is better than single source links, and they highly recommended our tool to developers, testers, and researchers. 6 of them also (strongly) agreed that they could easily visualize a class tree of the traced solution and in an efficient way, and the tool can help to save time needed to find traceability links and make a

project a lot more cost effective. Furthermore, 6 subjects (strongly) agreed that the visualization of traceability links was clear, and they were able to show the statistical data of the program components easily using the tool.
Several participants gave the answer "agree", and in each question around 1 or 2 participants answered "neither agree nor disagree". 4 subjects (strongly) agreed that they could easily detect the base code to test dependency and 3 agreed to this question, but 4 subjects answered "disagree". They responded that they did not figure out that they should right-click on a class to show the dependency diagram, they commented that this feature should be more prominent in the interface. 7 subjects (strongly agree) that the automated recovery approaches are better than manual methods, whereas 1 participants answered (disagree) to this question. They justified their answer that the manual methods cannot be avoided during project development, it is often required to confirm vulnerabilities.

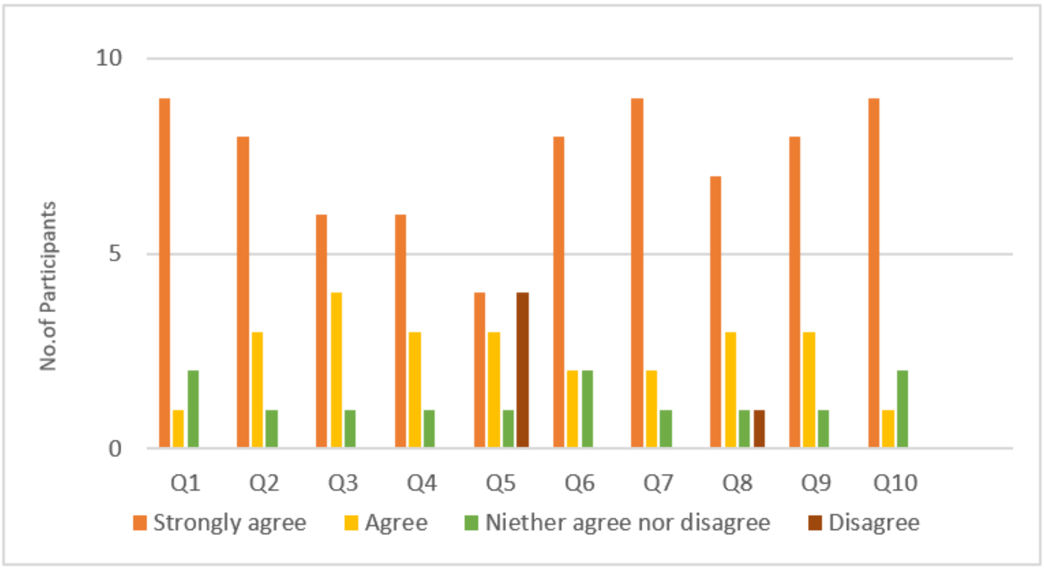


Figure 5.4: Evaluation Results

Overall, the results revealed that the participants strongly agreed that the visualization of traceability link inferred from different sources is more efficient and helpful than using a single source, and that, the results showed that the visualization tool can efficiently support understanding, browsing, and maintaining of test to code traceability links in software system.

5.6 Discussion

The usability evaluation of TCTracVis gained promising results. It is achieved much better results on recovering traceability links between units test and classes than manually recovering links. The subjects were able to recover and visualize the traceability links between test cases and production classes in a traced system efficiently and effectively. The subjects also could easily and quickly browse the links and find a specific artifact (i.e. class, method, test case). Furthermore, the tool supported the comprehension of test-to-code traceability links.

We combined multiple recovery approaches to take advantage of the strengths of each of them to improve the restrictions of each of them. We adopted hierarchy tree technique to visualize the traceability information and display the detailed information of a link of the selected node. Our tool made it simple for the participants to understand the structure of the traced solution and an overview of the links in it. The participants were satisfied with the combining of multiple recovery approaches to retrieve the links. They thought that using multiple sources of links outperformed using of a single source.

Based on our results, we observed that TCTracVis tool can provide the subjects with following features and functions

- Directly identify a specific item in a traced program to show its related links.
- Easily visualize a class hierarchical tree of a program.
- Easily detect and visualize a class dependency on test cases.
- Easily retrieve the traceability links (i.e. the related classes and methods) for a specific test case from multiple sources and visualize the retrieved links in an efficient way.
- Support an overall overview of the program components by providing detailed statistics of these components with visualization support for the statistics.
- Save the time needed in finding test-to-code traceability links in a project efficiently and make the project cost-effective.

5.7 Threats to Validity

Some threats potentially affect the validity of our evaluation results. We have exerted reasonable efforts to identify these threats and attempted to alleviate some of them.

The first threat is that only one system has been used in the usability evaluation. The min reason for using one system is manual evaluation. We drew attention that the tool is a generic tool and supports C# solutions of varying sizes.

The second threat concerning the extent to which the results of our study can be generalized; to alleviate this threat, we introduced the tool and the problem to different subjects (evaluators) with varying levels of experience. Our evaluators included students, academics, and industry experts who have familiarity with the necessary software development skills.

The third threat is that some of the participants were unfamiliar with the concept of traceability or visualization. To overcome this threat, we provided the subjects with a precise description of the traceability and visualization, ran a demo, and explained the tool. We allowed the subjects sufficient time to understand the functionality or usability of the tool and practice visualization exercises on some sample sets of problems before presenting the real problem to solve.

5.8 Summary

This chapter presents a usability evaluation of TCTraVis trace visualization tool. The main target of this study is to assess the extent to which our tool can be helpful and useful for the developers, testers, and even researchers in system development process. The results of this evaluation show that our tool can efficiently and effectively support understanding, browsing, and maintaining the test and code relations in a software system.

The results achieved in this chapter satisfy two requirements given in Section 3.6:

Req 4. Easy to use and apply in practice. The subjects who evaluated our tool "strongly agreed" that TCTracVis is easy to use and they recommended developers and testers to use it.

Req 5. Provide an empirical evidence that the proposed approach is more efficient than other approaches. Our tool provides three automated recovery approaches to capture the links between tests and source code. The results of our evaluation illustrated that the automated approaches are much better than the manual recovery approaches in term of time and cost.

Chapter 6

Summary

In this thesis, we focus on the specific problem of recovering and visualizing the traceability links between test cases and the related production classes. We provide an innovative approach for automatically capturing the traceability links between unit test and classes from multiple sources of links and visualizing these captured links in order to help the testers and developers to get a bigger picture about what is going on with the tests and understand the relationships between test cases and the corresponding units under test.
Our thesis consists of four main parts, namely background introduced in Chapter 2, state of the art in Chapter 3, trace visualization approach in Chapter 4, and evaluation of trace visualization approach in Chapter 5.

Background

In Chapter 2, we presented general terms that are defined in the contexts of traceability and visualization. Then we provided concise description about test-to-code traceability. After, we introduced general information about using visualization in representing software system. Furthermore, we presented the visualization techniques used to visualize a source code. Then, we discussed the usage of visualization with testing as well as, the visualization methods that usually employed to show test related information.

State-of-The-Art

In Chapter 3, we presented a systematic review that discussed the existing articles and research concerning creating and visualizing traceability links between different software artifacts, as well as, between test and code in specific.

The results of the systematic review obtained that visualization of test-code relations didn't have much interest in practice despite of its importance in maintenance, comprehension, evolution, and refactoring of a software system. We defined number of requirements for a new approach that supports visualization of test-to-code traceability links.

These requirements are as follow:

Req 1 Create test-to-code traceability links automatically during the development process.

Req 2 Support visualization of the created traceability links.

Req 3 Support tool or integrate the automatic traceability visualization and creation with the development process to reduce the work effort required during the development.

Req 4 Easy to use and apply in practice.

Req 5 Provide an empirical evidence that the proposed approach is more efficient than other approaches.

Trace Visualization Approach

Chapter 4 presents a novel approach that combines different traceability recovery methods to establish the links between unit test and its related classes. The approach supports visualization of the retrieved links from multiple sources links. A visualization system, called "TCTracVis", is developed to support our approach. TCTracVis employs three test-to-code traceability links naming [6]: Naming convention

(NC), Last call Before Assert (LCBA), and Static call graph (SCG) to automatically extract and establish the links between test cases and its classes which, in turn, reduces the effort of manual recovering traceability links. Visualization support to these links is provided with the use of hierarchal tree visualization techniques in order to help developers automatically overview the links inferred by various techniques and also to select the right relations for analyses.
In this chapter, we defined all methods that are used to recover the traceability links between test and code. We also discussed visualization technique used to visualize traceability links in software system. Our contributions to this chapter are the following:

1. Build an informative and generic tool for improved program comprehension, browsing, and maintenance.

2. Displaying the hierarchical relationships of source code and units test recovered by multiple sources for better system evolution analysis.

Evaluation for Trace Visualization Approach

In Chapter 5, We examined the usability of our visualization system and assessed users' interest. We conducted a usability study to evaluate the usability and usefulness of our traceability visualization tool. We prepared a set of questions to know how the use of TCTracVis helps its end users in browsing, comprehension, and maintenance of test-to-code traceability links of a software product or project. We defined four tasks that were performed by a subset of subjects who have different backgrounds in software developments skills. These tasks were performed in two ways: manually and using our tool. Then, the subjects answered ten questions that reflected their experience in using our tool and in order to analyze the results of our usability evaluation.
The results of this evaluation show that our tool can be helpful and useful for the developers, testers, and even researchers in system development process efficiently

and can effectively support the software engineers in understanding, browsing, and maintaining the test and code relations in a software system, and that, the results revealed that the visualization of traceability link inferred from different sources is more efficient and helpful than using a single source. The automated recovery approaches provided by our tool much better and faster in recovering test-to-code than manual methods. The visualization method used in our tool provided promising results in comprehension, browsing, and maintenance the test and code traceability links of traced system. The results obtained that our tool can be highly recommended to use in software development process.

Future Work

Based on the results presented in this thesis, there are potential areas of future work as follows:

Implement visualization traceability approach on other programming language.

As we achieved good results during the evaluation of our traceability visualization approach on C# solution files, our visualization system can be extended to support further programming languages such as Java, C++, Python.

Support other types of traceability recovery approaches.

While the traceability visualization approach supports multiple sources of traceability links, the visualization system was built supporting three types of traceability recovery methods. We plan to implement our approach with further recovery approaches.

Support an overall overview visualization of project.

Currently, the traceability links for each node (i.e. node can be class, test case) can be displayed on the method level. We plan to extend our visualization system to include one overall overview visualization of all traceability links for the whole project which, in turn, may need to support other types of visualization techniques.

A more thorough analysis of the use of traceability links during development.

In the current work, we looked at how the visualization of traceability links between test cases and tested class can be used in comprehension support. There are other usage scenarios of traceability links visualization during development need to be studied in more details such as system evolution, code coverage, regression testing.

Bibliography

[1] Institute of Electrical and Electronics Engineers (1990), *IEEE Standard Glossary of Software Engineering Terminology*, Office. vol. 121990, no. 1, p. 1. 1990. [Online]. Available: http://ieeexplore.ieee.org.

[2] S. B. I. Mohammad Hossein Abolghasem Zadeh . Mohammad Nazri Kama, Pourya Nikfard (2013), Software Changes, *Int. Conf. Adv. Comput. Netw - ACN 2013, 2013.*

[3] Hayes, J.H., Dekhtyar, A. and Janzen, D.S. (2009), Towards traceable test-driven development, *In 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering* . pp 26-30. IEEE

[4] A. Qusef (2013), Test-to-code traceability: Why and how?, *In 2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT).* pp 1-8. IEEE

[5] Parizi, Reza Meimandi, Sai Peck Lee, and Mohammad Dabbagh (2014), Achievements and challenges in state-of-the-art software traceability between test and code artifacts, *IEEE Transactions on Reliability* 63, no. 4 (2014): pp 913-926.

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

[7] Qusef, A., Oliveto, R., and De Lucia, A. (2010), Recovering traceability links between unit tests and classes under test: An improved method. *In 2010 IEEE International Conference on Software Maintenance.* pp (1-10). IEEE.

[8] Roman, G. C., and Cox, K. C. (1992), Program visualization: The art of mapping programs to pictures. *In Proceedings of the 14th international conference on Software engineering.* pp (412-420).

[9] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G. (2005), Adams re-trace: A traceability recovery tool.*In Ninth European Conference on Software Maintenance and Reengineering.* pp (32-41). IEEE.

[10] Merten, T., Jüppner, D., and Delater, A. (2011), Improved representation of traceability links in requirements engineering knowledge using Sunburst and Netmap visualizations.*In 2011 4th International Workshop on Managing Requirements Knowledge.* pp (17-21). IEEE.

[11] Di Thommazo, A., Malimpensa, G., de Oliveira, T. R., Olivatto, G., and Fabbri, S. C. (2012), Requirements traceability matrix: Automatic generation and visualization. *In 2012 26th Brazilian Symposium on Software Engineering.* pp (101-110). IEEE.

[12] Heim, P., Lohmann, S., Lauenroth, K., and Ziegler, J. (2008), Graph-based visualization of requirements relationships. *In 2008 Requirements Engineering Visualization.*pp (51-55). IEEE.

[13] Arbuckle, T., Balaban, A., Peters, D. K., & Lawford, M. (2007, July). Software Documents: Comparison and Measurement. In SEKE (Vol. 7, pp. 740-745).

[14] Spanoudakis, G., and Zisman, A. (2005), Software traceability: a roadmap. *In Handbook Of Software Engineering And Knowledge Engineering.* Vol 3: Recent Advances. pp(395-428).

[15] Cleland-Huang, J., Gotel, O., and Zisman, A. (2012), Software and systems traceability. Vol (2), No (3), pp (7-8). Heidelberg: Springer.

[16] COEST, Center of excellence for software traceability (coest). (2020).

[17] Gotel, O. C., and Finkelstein, C. W. (1994), An analysis of the requirements traceability problem.*In Proceedings of IEEE International Conference on Requirements Engineering.* pp (94-101). IEEE.

[18] Bani-Salameh, H., and Al jawabreh, N. (2015), Towards a comprehensive survey of the requirements elicitation process improvements. *In Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication.* pp (1-6).

[19] Maro, S. (2017), Addressing Traceability Challenges in the Development of Embedded Systems.

[20] IEEE, IEEE Standard for Software Test Documentation, vol. 1998. 1998.

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

[22] Hayes, J. H., Dekhtyar, A., and Osborne, J. (2003), Improving requirements tracing via information retrieval. *In Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* pp (138-147). IEEE.

[23] Tsuchiya, R., Washizaki, H., Fukazawa, Y., Kato, T., Kawakami, M., and Yoshimura, K. (2015), Recovering traceability links between requirements and source code using the configuration management log. it IEICE TRANSACTIONS on Information and Systems, 98(4), 852-862.

[24] 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.

[25] V. Friedman.(2008),Data Visualization and Infographics, *in: Graphics, Monday Inspiration.*

[26] Knight, C., and Munro, M. (1999), Comprehension with [in] virtual environment visualisations.*In Proceedings Seventh International Workshop on Program Comprehension.* pp (4-11). IEEE.

[27] Hendrix, T. D., Cross, J. H., Maghsoodloo, S., and McKinney, M. L. (2000), Do visualizations improve program comprehensibility? Experiments with con-

trol structure diagrams for Java. *In Proceedings of the thirty-first SIGCSE technical symposium on Computer science education.* pp (382-386).

[28] D. Gračanin, K. Matković, and M. Eltoweissy.(2005), Software visualization. *Innov. Syst. Softw. Eng.*, vol. 1, no. 2, pp (221–230).

[29] Caserta, P., and Zendra, O. (2010), Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7), pp (913-933).

[30] Koschke, R. (2003), Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2), pp (87-109).

[31] Diehl, S. (2007), Software visualization: visualizing the structure, behaviour, and evolution of software. *Springer Science and Business Media.*

[32] Sillito, J., Murphy, G. C., and De Volder, K. (2006), Questions programmers ask during software evolution tasks. *In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering.* pp (23-34).

[33] Fritz, T., and Murphy, G. C. (2010), Using information fragments to answer the questions developers ask. *In 2010 ACM/IEEE 32nd International Conference on Software Engineering.* Vol. 1, pp (175-184). IEEE.

[34] Merino, L., Ghafari, M., and Nierstrasz, O. (2016), Towards actionable visualisation in software development. *In 2016 IEEE Working Conference on Software Visualization (VISSOFT).* pp (61-70). IEEE.

[35] Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005), Visualizing multiple evolution metrics. *In Proceedings of the 2005 ACM symposium on Software visualization.* pp (67-75).

[36] Vilanova, A., Telea, A., Scheuermann, G., and Möller, T. Code Flows: Visualizing Structural Evolution of Source Code.

[37] Wingkvist, A., Ericsson, M., Lincke, R., and Löwe, W. (2010), A metrics-based approach to technical documentation quality. *In 2010 Seventh International Conference on the Quality of Information and Communications Technology.* pp (476-481). IEEE.

[38] Varet, A., and Larrieu, N. (2013), METRIX: a new tool to evaluate the quality of software source codes. *In AIAA Infotech@ Aerospace (I@ A) Conference .* (p. 4567).

[39] Bohnet, J., and Döllner, J. (2011), Monitoring code quality and development activity by software maps. *In Proceedings of the 2nd Workshop on Managing Technical Debt.* pp (9-16).

[40] Boccuzzo, S., and Gall, H. C. (2008), Software visualization with audio supported cognitive glyphs. *In 2008 IEEE International Conference on Software Maintenance.* pp (366-375). IEEE.

[41] Denier, S., and Sahraoui, H. (2009), Understanding the use of inheritance with visual patterns. *In 2009 3rd International Symposium on Empirical Software Engineering and Measurement.* pp (79-88). IEEE.

[42] Cockburn, A., and McKenzie, B. (2002,. Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. *In Proceedings of the SIGCHI conference on Human factors in computing systems.* pp (203-210).

[43] Wettel, R., and Lanza, M. (2008). CodeCity.

[44] Balogh, G., and Beszedes, A. (2013), CodeMetropolis-code visualisation in MineCraft. *In 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM).* pp (136-141). IEEE.

[45] Minecraft Official Website.http://minecraft.net/

[46] Khaloo, P., Maghoumi, M., Taranta, E., Bettner, D., and Laviola, J. (2017), Code park: A new 3d code visualization tool. *In 2017 IEEE Working Conference on Software Visualization (VISSOFT).* (pp. 43-53). IEEE.

[47] Lewerentz, C., and Simon, F. (2002), Metrics-based 3D visualization of large object-oriented programs. *In Proceedings First International Workshop on Visualizing Software for Understanding and Analysis.* (pp. 70-77). IEEE.

[48] Tamisier, T., Karski, P., and Feltz, F. (2013), Visualization of unit and selective regression software tests. *In International Conference on Cooperative Design, Visualization and Engineering.* (pp. 227-230). Springer, Berlin, Heidelberg.

[49] D'Ambros, M., Lanza, M., and Pinzger, M. (2007), " A Bug's Life" Visualizing a Bug Database. *In 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis.* (pp. 113-120). IEEE.

[50] Araya, V. P. (2011), Test blueprint: an effective visual support for test coverage. *In 2011 33rd International Conference on Software Engineering (ICSE).* (pp. 1140-1142). IEEE.

[51] Jones, J. A., Harrold, M. J., and Stasko, J. (2002), Visualization of test information to assist fault localization. *In Proceedings of the 24th International Conference on Software Engineering.* ICSE 2002 (pp. 467-477). IEEE.

[52] Breugelmans, M., and Van Rompaey, B. (2008), Testq: Exploring structural and maintenance characteristics of unit test suites. *In WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques.*

[53] Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., ... and Wilde, N. (1998), Mining system tests to aid software maintenance. Computer, 31(7), pp (64-73).

[54] Cornelissen, B., Van Deursen, A., Moonen, L., and Zaidman, A. (2007), Visualizing testsuites to aid in software understanding. *In 11th European Conference on Software Maintenance and Reengineering (CSMR'07).* (pp. 213-222). IEEE.

[55] Koochakzadeh, N., and Garousi, V. (2010), Tecrevis: a tool for test coverage and test redundancy visualization. *In International Academic and Indus-*

trial Conference on Practice and Research Techniques. (pp. 129-136). Springer, Berlin, Heidelberg.

[56] Van Rompaey, B., and Demeyer, S. (2008), Exploring the composition of unit test suites. *In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops.* (pp. 11-20). IEEE.

[57] J. Filipe and L. A. Maciaszek,(2013), Evaluation of Novel Approaches to Software Engineering.

[58] Balogh, G., Gergely, T., Beszédes, A., and Gyimóthy, T. (2016), Using the city metaphor for visualizing test-related metrics. *In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* (Vol. 2, pp. 17-20). IEEE.

[59] Ghafari, M., Ghezzi, C., and Rubinov, K. (2015), Automatically identifying focal methods under test in unit test cases. *In 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM).* (pp. 61-70). IEEE.

[60] Asuncion, H. U., and Taylor, R. N. (2012), Automated techniques for capturing custom traceability links across heterogeneous artifacts. *In Software and Systems Traceability.* (pp. 129-146). Springer, London.

[61] Omoronyia, I., Sindre, G., Roper, M., Ferguson, J., and Wood, M. (2009), Use case to source code traceability: The developer navigation view point. *In 2009 17th IEEE International Requirements Engineering Conference.* (pp. 237-242). IEEE.

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

[63] N. Gesamtfakult and A. Delater, "INAUGURAL-DISSERTATION," (2013).

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

[65] Chen, X., and Grundy, J. (2011), Improving automated documentation to code traceability by combining retrieval techniques. *In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* (pp. 223-232). IEEE.

[66] Chen, X., Hosking, J., Grundy, J., and Amor, R. (2018), DCTracVis: a system retrieving and visualizing traceability links between source code and documentation. *Automated Software Engineering*, 25(4), pp (703-741).

[67] Varun Kumar, S., and Kumar, M. (2010), Test case prioritization using fault severity. IJCST, 1(1).

[68] Rees, M. J. (2002), A feasible user story tool for agile software development?. *In Ninth Asia-Pacific Software Engineering Conference, 2002.* (pp. 22-30). IEEE.

[69] Bouquet, F., Jaffuel, E., Legeard, B., Peureux, F., and Utting, M. (2005), Requirements traceability in automated test generation: application to smart card software validation. *ACM SIGSOFT Software Engineering Notes*, 30(4), pp (1-7).

[70] Lormans, M., and Van Deursen, A. (2006), Can LSI help reconstructing requirements traceability in design and test?. *In Conference on Software Maintenance and Reengineering (CSMR'06).* (pp. 10-pp). IEEE.

[71] 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-es.

[72] Fraser, G., and Arcuri, A. (2012), Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2), pp (276-291).

[73] Geraci, A., Katki, F., McMonegal, L., Meyer, B., Lane, J., Wilson, P., ... and Springsteel, F. (1991), IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries. IEEE Press.

[74] S. Demeyer, Object-oriented reengineering.(2008).

[75] Eagan Jr, J. R., Harrold, M. J., Jones, J. A., and Stasko, J. T. (2001), Visually encoding program test information to find faults in software. Georgia Institute of Technology.

[76] Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001), Refactoring test code. *In Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*. pp (92-95).

[77] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2000), Tracing object-oriented code into functional requirements. *In Proceedings IWPC 2000. 8th International Workshop on Program Comprehension.* (pp. 79-86). IEEE.

[78] 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), pp (970-983).

[79] Marcus, A., Maletic, J. I., and Sergeyev, A. (2005), Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(05), pp (811-836).

[80] Abadi, A., Nisenson, M., and Simionovici, Y. (2008), A traceability technique for specifications. *In 2008 16th IEEE International Conference on Program Comprehension.* (pp. 103-112). IEEE.

[81] Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., and Panichella, S. (2009), Traceability recovery using numerical analysis. *In 2009 16th Working Conference on Reverse Engineering.* (pp. 195-204). IEEE.

[82] 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.

[83] Cleland-Huang, J., Settimi, R., Duan, C., and Zou, X. (2005), Utilizing supporting evidence to improve dynamic requirements traceability. *In 13th IEEE international conference on Requirements Engineering (RE'05)*. (pp. 135-144). IEEE.

[84] Zou, X., Settimi, R., and Cleland-Huang, J. (2007), Term-based enhancement factors for improving automated requirement trace retrieval. *In Proceedings of International Symposium on Grand Challenges in Traceability*. (pp. 40-45). ACM Press Lexington, Kentuky, USA.

[85] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. (2000). Identifying the starting impact set of a maintenance request: A case study. *In Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*. (pp. 227-230). IEEE.

[86] Yadla, S., Hayes, J. H., and Dekhtyar, A. (2005), Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2), pp (116-124).

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

[88] Lormans, M., van Deursen, A., and Gross, H. G. (2008), An industrial case study in reconstructing requirements views. Empirical Software Engineering, 13(6), pp (727-760).

[89] De Lucia, A., Oliveto, R., and Sgueglia, P. (2006), Incremental approach and user feedbacks: a silver bullet for traceability recovery. *In 2006 22nd IEEE International Conference on Software Maintenance*. (pp. 299-309). IEEE.

[90] Gall, H., Hajek, K., and Jazayeri, M. (1998), Detection of logical coupling based on product release history. *In Proceedings. International Conference on Software Maintenance.* (Cat. No. 98CB36272) (pp. 190-198). IEEE.

[91] Kagdi, H., Maletic, J. I., and Sharif, B. (2007), Mining software repositories for traceability links. *In 15th IEEE International Conference on Program Comprehension (ICPC'07).* (pp. 145-154). IEEE.

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

[93] 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), pp (574-586).

[94] Zaidman, A., Van Rompaey, B., Demeyer, S., and Van Deursen, A. (2008), Mining software repositories to study co-evolution of production and test code. *In 2008 1st international conference on software testing, verification, and validation.* (pp. 220-229). IEEE.

[95] Egyed, A., and Grunbacher, P. (2002), Automating requirements traceability: Beyond the record and replay paradigm. *In Proceedings 17th IEEE International Conference on Automated Software Engineering,.* (pp. 163-171). IEEE.

[96] 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), pp (364-380).

[97] Bruntink, M., and Van Deursen, A. (2004), Predicting class testability using object-oriented metrics. *In Source Code Analysis and Manipulation, Fourth IEEE International Workshop on* (pp. 136-145). IEEE.

[98] Bouillon, P., Krinke, J., Meyer, N., and Steimann, F. (2007), Ezunit: A framework for associating failed unit tests with potential programming errors.

In International Conference on Extreme Programming and Agile Processes in Software Engineering (pp. 101-104). Springer, Berlin, Heidelberg.

[99] Gaelli, M., Lanza, M., and Nierstrasz, O. (2005). Towards a Taxonomy of SUnit Tests. In ESUG (pp. 99-119).

[100] Van Geet, J., Zaidman, A., Greevy, O., and Hamou-Lhadj, A. (2006). A lightweight approach to determining the adequacy of tests as documentation. *In Proc. of the 2nd Workshop on Program Comprehension through Dynamic Analysis.* (pp. 21-26).

[101] Lago, P., Muccini, H., and Van Vliet, H. (2009), A scoped approach to traceability management. Journal of Systems and Software, 82(1), pp (168-182).

[102] Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., and Romanova, E. (2007). Best practices for automated traceability. Computer, 40(6), pp (27-35).

[103] Rodrigues, A., Lencastre, M., and Gilberto Filho, A. D. A. (2016, September), Multi-VisioTrace: traceability visualization tool. *In 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC).* (pp. 61-66). IEEE.

[104] Gilberto Filho, A. D. A., and Zisman, A, D3TraceView: A Traceability Visualization Tool.

[105] WMarcus, A., Xie, X., and Poshyvanyk, D. (2005, November). When and how to visualize traceability links?. *In Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering.* (pp. 56-61).

[106] Cleland-Huang, J., and Habrat, R. (2007), Visual support in automated tracing. *In Second International Workshop on Requirements Engineering Visualization (REV 2007).* (pp. 4-4). IEEE.

[107] Chen, X., Hosking, J., and Grundy, J. (2012, September). Visualizing traceability links between source code and documentation. *In 2012 IEEE Sympo-*

sium on Visual Languages and Human-Centric Computing (VL/HCC). (pp. 119-126). IEEE.

[108] Lin, J., Lin, C. C., Cleland-Huang, J., Settimi, R., Amaya, J., Bedford, G., ... and Zou, X. (2006), Poirot: A distributed tool supporting enterprise-wide automated traceability. *In 14th IEEE International Requirements Engineering Conference (RE'06).* (pp. 363-364). IEEE.

[109] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., & Binkley, D. (2011, October). Scotch: Slicing and coupling based test to code trace hunter. In 2011 18th Working Conference on Reverse Engineering (pp. 443-444). IEEE.

[110] Roman, G. C., and Cox, K. C. (1992), Program visualization: The art of mapping programs to pictures. *In Proceedings of the 14th international conference on Software engineering.* (pp. 412-420).

[111] Wang, X., Lai, G., and Liu, C. (2009), Recovering relationships between documentation and source code based on the characteristics of software engineering. Electronic Notes in Theoretical Computer Science, 243, 121-137.

[112] Settimi, R., Cleland-Huang, J., Khadra, O. B., Mody, J., Lukasik, W., and DePalma, C. (2004), Supporting software evolution through dynamically retrieving traces to UML artifacts. *In Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* (pp. 49-54). IEEE.

[113] RKonchady, M. (2008). Building Search Applications: Lucene, LingPipe, and Gate. Lulu. com.

[114] Jirapanthong, W., and Zisman, A. (2009). Xtraque: traceability for product line systems. Software and Systems Modeling, 8(1), 117-144.

[115] Bacchelli, A., Lanza, M., and Robbes, R. (2010, May). Linking e-mails and source code artifacts. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (pp. 375-384).

[116] Bacchelli, A., D'Ambros, M., Lanza, M., and Robbes, R. (2009, October). Benchmarking lightweight techniques to link e-mails and source code. In 2009 16th Working Conference on Reverse Engineering (pp. 205-214). IEEE.

[117] Egyed, A., Biffl, S., Heindl, M., and Grünbacher, P. (2005, November). A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering (pp. 2-7).

[118] Meszaros, G. (2007). xUnit test patterns: Refactoring test code. Pearson Education.

[119] Fewster, M., & Graham, D. (1999). Software test automation (pp. 211-219). Reading: Addison-Wesley.

[120] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. Journal of the American society for information science, 41(6), 391-407.

[121] Qusef, A. (2011, October). Recovering test-to-code traceability via slicing and conceptual coupling. In 2011 18th Working Conference on Reverse Engineering (pp. 417-420). IEEE.

[122] Korel, B., & Laski, J. (1990). Dynamic slicing of computer programs. Journal of Systems and Software, 13(3), 187-195.

[123] Wang, Q., Wang, W., Brown, R., Driesen, K., Dufour, B., Hendren, L., & Verbrugge, C. (2003, June). EVolve: an open extensible software visualization framework. In Proceedings of the 2003 ACM symposium on Software visualization (pp. 37-ff).

[124] Bosch, R., Stolte, C., Tang, D., Gerth, J., Rosenblum, M., & Hanrahan, P. (2000). Rivet: A flexible environment for computer systems visualization. ACM SIGGRAPH Computer Graphics, 34(1), 68-73.

[125] Reed, D. A., Roth, P. C., Aydt, R. A., Shields, K. A., Tavera, L. F., Noe, R. J., & Schwartz, B. W. (1993, October). Scalable performance analysis: The

Pablo performance analysis environment. In Proceedings of Scalable Parallel Libraries Conference (pp. 104-113). IEEE.

[126] Nagel, W. E., & Arnold, A. (1994, May). Performance visualization of parallel programs-the PARvis environment. In Proceedings (pp. 24-31).

[127] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H. C., & Solchenbach, K. (1996). VAMPIR: Visualization and analysis of MPI resources.

[128] Benbasat, I., Dexter, A. S., & Todd, P. (1986). The influence of color and graphical information presentation in a managerial decision simulation. Human-Computer Interaction, 2(1), 65-92.

[129] Li, Y., & Maalej, W. (2012, March). Which traceability visualization is suitable in this context? a comparative study. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 194-210). Springer, Berlin, Heidelberg.

[130] Kaindl, H. (1993). The missing link in requirements engineering. ACM SIGSOFT Software Engineering Notes, 18(2), 30-39.

[131] Marchionini, G., & Shneiderman, B. (1988). Finding facts vs. browsing knowledge in hypertext systems. Computer, 21(1), 70-80.

[132] Lange, D. B., & Nakamura, Y. (1997). Object-oriented program tracing and visualization. Computer, 30(5), 63-70.

[133] Kamalabalan, K., Uruththirakodeeswaran, T., Thiyagalingam, G., Wijesinghe, D. B., Perera, I., Meedeniya, D., & Balasubramaniam, D. (2015, April). Tool support for traceability of software artefacts. In 2015 Moratuwa Engineering Research Conference (MERCon) (pp. 318-323). IEEE..

[134] Reggio, G., Leotta, M., Ricca, F., & Clerissi, D. (2013). What are the used UML diagrams? A Preliminary Survey. EESSMOD@ MoDELS, 1078(10).

[135] McGavin, M., Wright, T., & Marshall, S. (2006, January). Visualisations of execution traces (VET) an interactive plugin-based visualisation tool. In Pro-

ceedings of the 7th Australasian User interface conference-Volume 50 (pp. 153-160).

[136] Long, L. K., Hui, L. C., Fook, G. Y., & Zainon, W. M. N. W. (2017). A Study on the Effectiveness of Tree-Maps as Tree Visualization Techniques. Procedia Computer Science, 124, 108-115.

[137] Card, M. (1999). Readings in information visualization: using vision to think. Morgan Kaufmann.

[138] De Pauw, W., Lorenz, D. H., Vlissides, J. M., & Wegman, M. N. (1998, April). Execution Patterns in Object-Oriented Visualization. In COOTS (Vol. 98, pp. 16-16).

[139] Marshall, S. (2001). Methods and tools for the visualization and navigation of graphs. Dept. of Mathematics and Computer Science, University of Bordeaux, France.

[140] Shneiderman, B., & Aris, A. (2006). Network visualization by semantic substrates. IEEE transactions on visualization and computer graphics, 12(5), 733-740.

[141] Johnson, B., & Shneiderman, B. (1999). Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical. Readings in Information Visualization: Using Vision to Think, 152-159..

[142] "Sunburst Chart | SpreadJS 13." https://www.grapecity.com/spreadjs/docs/v13 /online/CreatingSunburstChart.html (accessed Jun. 23, 2020).

[143] Graham, H., Yang, H. Y., & Berrigan, R. (2004, January). A solar system metaphor for 3D visualisation of object oriented software metrics. In Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35 (pp. 53-59).

[144] Mindek, P., & Kapec, P. (2011, April). Graph visualization using the metaphor of biological neural nets. In Proceedings of the 27th Spring Conference on Computer Graphics (pp. 141-148).

[145] Kahn, K. (1996). Drawings on napkins, video-game animation, and other ways to program computers. Communications of the ACM, 39(8), 49-59.

[146] Wettel, R., & Lanza, M. (2007, June). Visualizing software systems as cities. In 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (pp. 92-99). IEEE.

[147] Dugerdil, P., & Alam, S. (2008, April). Execution trace visualization in a 3D space. In Fifth International Conference on Information Technology: New Generations (itng 2008) (pp. 38-43). IEEE.

[148] Merino, L., & Nierstrasz, O. (2018). The medium of visualization for software comprehension (Doctoral dissertation, Universität Bern).

[149] Keim, D. A. (2002). Information visualization and visual data mining. IEEE transactions on Visualization and Computer Graphics, 8(1), 1-8.

[150] Bertini, E., Tatu, A., & Keim, D. (2011). Quality metrics in high-dimensional data visualization: An overview and systematization. IEEE Transactions on Visualization and Computer Graphics, 17(12), 2203-2212.

[151] Alkawaz, M. H., & Silvarajoo, A. (2019, December). A Survey on Test Case Prioritization and Optimization Techniques in Software Regression Testing. In 2019 IEEE 7th Conference on Systems, Process and Control (ICSPC) (pp. 59-64). IEEE.

[152] Qusef, A., Bavota, G., Oliveto, R., Lucia, A. D., & Binkley, D. (2013). Evaluating test-to-code traceability recovery methods through controlled experiments. Journal of Software: Evolution and Process, 25(11), 1167-1191.

[153] White, R., Krinke, J., & Tan, R. (2020). Establishing Multilevel Test-to-Code Traceability Links. In 42nd International Conference on Software Engineering (ICSE'20). ACM.

[154] Broberg, P., & Jahanshahi, S. (2019). Using eye tracking to study variable naming conventions and their effect on code readability.

[155] Kugele, S., & Antkowiak, D. (2016, September). Visualization of trace links and change impact analysis. In 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW) (pp. 165-169). IEEE.

[156] Rubasinghe, I. D., Meedeniya, D. A., & Perera, I. (2018, February). Software Artefact Traceability Analyser: A Case-Study on POS System. In Proceedings of the 6th International Conference on Communications and Broadband Networking (pp. 1-5).

[157] https://openclover.org/

[158] Pietroszek, K., & Lee, N. (2019). Virtual Hand Metaphor in Virtual Reality.

[159] Kicsi, A., Vidács, L., Csuvik, V., Horváth, F., Beszédes, A., & Kocsis, F. (2018, May). Supporting product line adoption by combining syntactic and textual feature extraction. In International Conference on Software Reuse (pp. 148-163). Springer, Cham.

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

[BNN01] Aljawabrah, Nadera, and Tamás Gergely. "Visualization of test-to-code relations to detect problems of unit tests.*The 11th Conference of Phd Studentsin Computer Science. 2018* pp (1-4).

[BNN02] Aljawabrah, N., Gergely, T., and Kharabsheh, M. (2019), Understanding Test-to-Code Traceability Links: The Need for a Better Visualizing Model.*In International Conference on Computational Science and Its Applications*, pp (428-441). Springer, Cham.

[BNN03] Aljawabrah, Nadera, and Abdallah Qusef. "TCTracVis: test-to-code traceability links visualization tool." *In Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems*, pp. 1-4. 2019.

[BNN04] Nadera Aljawabrah, AbdAllah Qusef, Tamás Gergely, and Adhyatmananda Pati, Visualizing Multilevel Test-to-Code Relations.*In 3rd International Conference on Information and Communication Technology and Applications. Springer (CCIS)*, 2020.

[BNN05] Nadera Aljawabrah, Tamás Gergely, Sanjay Misra, and Luis Fernandez-Sanz, Automated Recovery and Visualization of Test-to-Code (TCT) Links: An Evaluation, *in the submission to IEEE access.*

[BNN06] Otoom, Ahmed Fawzi, Maen Hammad, Nadera Al-Jawabreh, and Rawan Abu Seini. "Visualizing Testing Results for Software Projects." *In Proc. of the 17th International Arab Conference on Information Technology (ACIT'16)*, Morocco. 2016.

[BNN07] Hammad, Maen, Ahmed Fawzi Otoom, Mustafa Hammad, Nadera Al-Jawabreh, and Rawan Abu Seini. "Multiview Visualization of Software Testing Results." *International Journal of Computing and Digital Systems* 9, no. 1 (2020).

[BBN08] Bani-Salameh, H., and Al jawabreh, N. (2015), Towards a comprehensive survey of the requirements elicitation process improvements. *In Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication.* pp (1-6).