

Understanding Test-to-Code Traceability Links: The Need for a Better Visualizing Model

Nadera Aljawabrah[1(✉)], Támas Gergely[1], and Mohammad Kharabsheh[2]

[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
{nadera, gertom}@inf.u-szeged.hu
[2] Department of Computer Information System, The Hashemite University, Zarqa, Jordan
mohkh86@hu.edu.jo

Abstract. Visualization of test-to-code traceability links is a great approach to understand test-to-code relations. It efficiently supports software developers in various software development activities throughout the software development life cycle (SDLC) by browsing, recovering and maintaining, links between various software artifacts. However, 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. This paper extensively draws attention of the reader/researcher to the usefulness of visualizing test-to-code traceability links and opens up several research questions or research paths for further advanced exploration.

Keywords: Visualization · Test-code relations · Traceability links

1 Introduction

Testing is considered to be an important phase in the software development life cycle (SDLC) in which the unit test plays a significant role in software evolution and maintenance and assists in building a quality product [1]. Visualization is a very effective method which helps testers to quickly understand the structure of code and testing correlation. Though there are many studies that address software visualization, only a few have focused on the visualization of test-code relations. This paper highlights this interesting problem for practitioners and researchers, which may drive more adoption of the test-code relation tools in practice, as well as trigger more study and development on related techniques and tools.

Understanding the relations between test and code is essential for other activities in SDLC, such as: change impact analysis, refactoring and reengineering [1]. In this work, we focus on the visualization of test-code relations from two aspects: visualization of test information and visualization of test-to-code traceability links. All information relevant to the testing process, such as test results, code coverage, and test-related metrics, can be treated as test information. Visualization of test information can be a means of providing valuable information about the adequacy of code testing [2], visualizing software faults [3], and evaluating code coverage of test suites' quality [4].

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 428–441, 2019.
https://doi.org/10.1007/978-3-030-24305-0_32

The area of visualizing test information has been targeted by a number of research teams. For example, visualization of code coverage [5] displays all statements that are executed by test suites and thus facilitates the location of faults in these statements. Another method, described by Cornelissen et al. [6], used a UML sequence diagram to visualize test information. Quite recently, there has been growing interest in visualizing test-related metrics which can be considered as an indication of the quality and the process of testing effort [7, 8]. Test-to-code traceability is useful in many software development tasks such as maintenance, refactoring and regression testing; visualization of these relations is one important methodology to assist the tasks.

Current research on test-to-code traceability links is focused on how to retrieve the links between test and code as well as different approaches that have been suggested and used to recover test-to-code-traceability links. However, in [9] the authors find that there is a lack of visualization support, one of the main challenges in the current test-to-code traceability recovery approaches and tools. This study aims to investigate the existing literature on test-related visualization. We have defined two research questions (RQs) to identify, evaluate and select all quality research evidence relevant to those questions. Our research questions focus on two test-code relation areas: Testing information and test-to-code traceability links.

RQ1 to what extent has a visualization of test-code relations been supported in existing studies?
The motivation of this question is to show whether the researchers and practitioners pay any attention to using visualization in identifying the links between test and code as well as obtaining more information about these links (e.g. how the artifacts are connected, types of relations, etc.).

RQ2 what visualization techniques and tools are available to represent test-code relations?
The purpose of this question is to identify the approaches followed in the literature to represent test-code relations in a visual manner, the most common methods followed in representation, and to obtain an overview about different types of visualization tools used and identify their purposes.

Investigating these questions can provide more information about the areas that have used the visualization and how much they have been supported by researchers and practitioners respectively. To answer these questions we explored many publications in the domain of software visualization, source-code related visualization and, particularly, test-related visualization.

The remainder of this paper is organized in different sections as follows: Background information and some related works are presented in the next section, why we need to visualize test and code relations is discussed in Sect. 3. We present results and discussion of our work in Sect. 4 and we conclude our paper in Sect. 5.

2 Background Information and Related Work

2.1 Software Visualization

Software development is a complex process that combines many tasks such as comprehension, analysis, and maintenance. In this context, software visualization plays a vital role in program understanding and in reducing the complexity of understanding the source code structure. Program comprehension is a very effective part, which provides an initial perception of the software structure and supports the maintainability of the software. Throughout the history of software development, visualization of software artifacts has attracted much attention from research teams and several visualization approaches have been proposed to analyze and explore different aspects of software systems, such as runtime behavior, source code or software evolution [10, 11]. Koschke [10], for example, conducted a survey discussing the use of software visualization in the context of software maintenance, re-engineering, and reverse engineering. The survey collected the perspectives of 82 researchers to identify how far visualization is used in these contexts. The results indicated that the researchers believe in the value of visualization in their work. While Caserta et al. [11] in their survey addressed all visualization techniques used in visualizing the evolution of static aspects of the software (e.g. code lines, classes, software architecture). They listed all visualizing tools developed to display the relations in software inheritance, access, and method calls.

Various visualization-based techniques have been used in software visualization. Graphs are the most frequently used visualization to convey information and describe binary relations (in general), as well as different layout tools, have been developed to draw graphs [12, 16]. 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.

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 systems. Visualization methods could be elicited depending on the user's need and the information to be visualized (e.g. metrics, relationships, and dependencies). For example, graph-based representation turned out to be appropriate for visualizing source code evolution. Most of the studies have focused on visualizing source code-related metrics and many approaches are proposed in this regard. Metrics is a numerical/proportional value to describe and measure the quality of software artifacts [12, 13] (e.g. source code quality, testing quality, and documentation quality); therefore, it is called quality metrics. Identification of the metrics to be used depends on the intent of the visualization. The code-related metrics which have been frequently used for visualization purpose are: line of codes (LOC), McCabe complexity [14, 15] and number of methods (NOM) [16, 17]. These metrics support the maintainability of source code. In visualization space, data and metrics are mapped to a set of visual attributes as per the context of visualization.

City metaphor is the most popular metaphor used for visualizing program components [18]. This metaphor supports navigation of the program, interaction with represented elements, and explores the city structure. City metaphor is an effective 3D method to represent software structure that enables the user to be well aware of the position of software objects, thus it can be easily retrieved to the development process. In other words, 3D visualization makes use of the spatial memory of users [19]. Wettel and Lanza [20] presented a 3D city metaphor-based language-independent interactive 3D visualization tool named 'CodeCity'. This tool presents class as building and package as a district of a 'software city'. Two code metrics are used for mapping on visual properties: NOM maps on the height of the building and NOA maps on the width. Their visualization is limited to a higher level of abstraction, i.e. package and class. Quite recently, considerable attention has been paid to use of a 3D games environment in software visualization. CodeMetropolis [21] is a command line which helps the city metaphor to visualize source code at a lower level of abstraction (methods and attributes). It uses a game engine 'Minecraft' [22] to visualize the structure of the 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 the Minecraft word and obtain details about internal classes. Balogh et al. [8] extended CodeMetropolis [21] to include visualization of test-related metrics to support developers to better understand the test suite's quality and its relation to the production code. CodePark [23] is another game environment-based tool, which 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 represent it. A set of code metrics has been used to explore a 3D graphs metaphor to describe the internal structure and relations of large-size programs for quality assessment purposes [24]. Visual properties of program entities (e.g. size, shape, color) represent particular metrics of these entities for mapping in 3D visualization metaphor. Information 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 above-mentioned works, many extensive research works have been conducted on code visualization. These works have been immensely useful to reduce the effort in understanding software architecture and thereby simplify the software maintenance and evolution process [10].

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. Visualization supports understanding of the inner workings of source code and the behavior of test suites [6].

However, from the developer's perspective, testing is a time-consuming process. Developers believe that their code is well written and, for the purpose of profit, their interest is mainly focused on delivering the software on time, thus testing is often excluded [1]. Hence, visualizing relevant test information (e.g. test suites, test results) is not sufficiently highlighted. Using visualization with testing can be an effective method to provide valuable information about: the adequacy of code testing,

visualizing software faults [25, 26], and evaluation code coverage of test suites' quality. This paper contributes to drawing attention to a visualization of test-code relations and concentrates particularly on two areas: visualization of testing information and visualization of test-to-code traceability links.

2.2 Test-to-Code Traceability Links

Current research on test-to-code traceability links is focused on how to retrieve the links between test and code. Rompaey and Demeyer [27] have compared six traceability recovery strategies in terms of the applicability and the accuracy of each approach. The comparison covers only those approaches relating to requirement traceability and test-to-code traceability. The strategies have been evaluated based on three open-source Java programs. In these approaches, units under test are identified by matching test cases, production code's name, vocabulary (e.g. identifiers, comments), examining method invocation in test cases, looking at the last calls right before assert statement, and capturing changes on test cases and production code in the version control change log. 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.

Qusef et al. [28] proposed a more accurate test-to code traceability recovery tool (SCOTCH+ – Source code and Concept-based Test to Code traceability Hunter). This technique depends on applying dynamic slicing and conceptual coupling to recover the links between test cases and source code, thus identifying class under test (CUT).

Recently, various techniques have been developed to support visualization of traceability links between software system artifacts. In addition, different tools have also been developed to automatically/semi-automatically retrieve traceability links between different software artifacts types (e.g. requirements, source code, and document). Marcus et al. [29] studied traceability links between software artifacts and showed how visualization can be important in recovering, maintaining and browsing links between such artifacts.

Traceability links can be identified according to a specific task required to retrieve these links, (e.g. recovering links for evolution purposes). The most popular types of links retrieved are established among items based on the types of software artifacts, which can be either implicit or explicit [30].

Visualization techniques and tools have been developed depending on the type of data to be visualized and the objectives of visualization (e.g. 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) [31]. In Table 1, a set of traceability techniques and tools are listed. Some tools [33, 34] allow the user to add, browse and delete links or even edit the properties of such links. Another tool is pluggable with an IDE [29] to give a uniform user experience for test-to-code traceability links and software artifacts. Each tool provides one or more visualization techniques which may display links in different ways depending on the information task context. For example, the Multi-Viso Trace

tool [33] provides four visualization techniques 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.

Table 1. Traceability links visualization techniques and tool

Visualization method/Technique	Links visualized	Link recovery method	Visualization tool
Context-based technique (Sunburst, Matrix Tree, Graph) [33]	Links between software artifacts	Automatic (REREATOS) tool	Multi-Visio Trace
Graph (Chain Graph) [13]	Links between software requirements	Manual	–
List [30]	Links between software artifacts	Automatic	ADAMS re-trace
Requirement Matrix (RTM-E, RTM-NLP) [34]	Requirement dependencies	Automatic + Manual	–
Small color squares [29]	Links between software artifacts	Manual	TraceViz
Sunburst, Net map [35]	Links between requirement artifacts (project goals, task, etc.)	Manual	–
Tree map Hierarchal tree [36]	Links between classes in source code and documents elements	Automatic	–
Tree, Matrix Sunburst, Table [32]	Links between software artifacts	Automatic (REREATOS tool)	D3TraceView

While most of the listed techniques present the links extracted between requirements and different software artifacts (e.g. documents, source code), none of these tools address the ease of visualizing the links between unit tests and code, which lays the foundation of this research.

3 Why Is the Visualization of Test-to-Code Relations Needed?

Test-to-code relations can be treated as traceability links that display how test cases and the code under test can be connected. Test suites are usually used to evaluate software systems and detect the program faults. The larger the programs are, the larger the test cases executed, thus a huge amount of data will be produced which is difficult to be interpreted in 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 fault occurrence in the source code with the least effort and time. Unit tests are used to determine whether the software being tested works correctly or not for a given input [37]. A test case is considered as an up-to-date document that reflects how parts of the code are changed and how they are supposed to be executed [1]. The benefits of running test cases lie in: improving software quality, reducing maintenance effort and cost [25], and identifying faults in software systems. A unit test plays an important role during regression testing where a unit test is re-executed and evolved in parallel with code changes [38]. However, the development process is an iterative and ongoing process which means new changes and updates continuously appear on the software, thus numerous regression tests have to be run (i.e. increasing the testing cost). Therefore, visualizing established links between units under test and their related test suites helps in reducing generating regression tests and saves much time during software evolution process [28].

Units test and tested code can be connected by different types of relations (e.g. direct tests, calls, indirect uses). 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). Thereby, visualizing test-to-code traceability links provides a better understanding of the inter-relationships between tests and tested code which in turn contributes to maintaining and browsing these relations. In addition, refactoring of the code requires some modifications to the test suite in order to keep it valid after refactoring, thus visualization of recovered relationships between unit test and unit under test could be exploited to support and facilitate the refactoring process [28].

With the increasing size and complexity of software under test and its automated test suites, visualization is needed to analyze code coverage and to provide testers with a wide range of information about the quality, performance, the location of a test suite, its relation with the production code, and which parts of code are covered by test cases [39]. There is a large amount of literature that deals with the visualization of test information and a wide range of tools have been proposed for the task of visualization of testing information as mentioned in the section above. However, to the author's best knowledge, very few publications are available in the literature that addresses the issue of visualizing test-to code-traceability links. Studies on test-to-code traceability links have paid more attention to how to recover links between test and With the increasing size and complexity of software under test and its automated test suites, visualization is needed to analyze code coverage and to provide testers with a wide range of information about the quality, performance, the location of a test suite, its relation with the production code, and which parts of code are covered by test cases [39]. There is a large amount of literature that deals with the visualization of test information and a wide range of tools have been proposed for the task of visualization of testing information as mentioned [40]. However, to the author's best knowledge, very few publications are available in the literature that addresses the issue of visualizing test-to code-traceability links. Studies on test-to-code traceability links have paid more attention to how to recover links between test and code, as well as various approaches have been proposed to retrieve the traceability links between the test units and units under test, as mentioned in Sect. 2. However, none of these methods supports the representation of such links in a visual manner. The focal point of this work is to draw attention to the

importance of using a visualization technique to display the links between a code and its related unit test.

4 Results and Discussion

In this section we analyze the collected results according to our research questions:

RQ1: To what extent has a visualization of test-to-code relations been investigated in existing studies?
We performed a detailed review of the literature following citations and references using web-based literature search engines. This effort resulted in gathering a number of publications that listed various approaches on the basis of different techniques relating to visualization topics. Thus from all the gathered publications, relevant literature which is found to be 44 in number, we refined and selected those approaches that satisfy the following criteria in accordance with our research purpose:

1. Those approaches that use visualization techniques to represent software source code.
2. Those approaches that use visualization techniques to represent test-related information.
3. Those approaches that use visualization techniques to represent test-to-code traceability links.

Figure 1 shows the share of the different investigated studies in percentage related to various methods of visualization from a selected sample of research papers. Our investigation shows that 48% of total software testing-related research works talk about visualization of source code. Testing information-based (test-code relations) visualization works amount to 29% and 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. Developers could focus more on the development process and activities which are responsible for testing activities. They trust their code is well written and does not need any tests. Therefore, not visualization in specific but testing, in general, is neglected.

Moreover, despite 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.

As mentioned in the previous section, 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 somewhat 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 purely a manual task that needs higher time and effort investment.

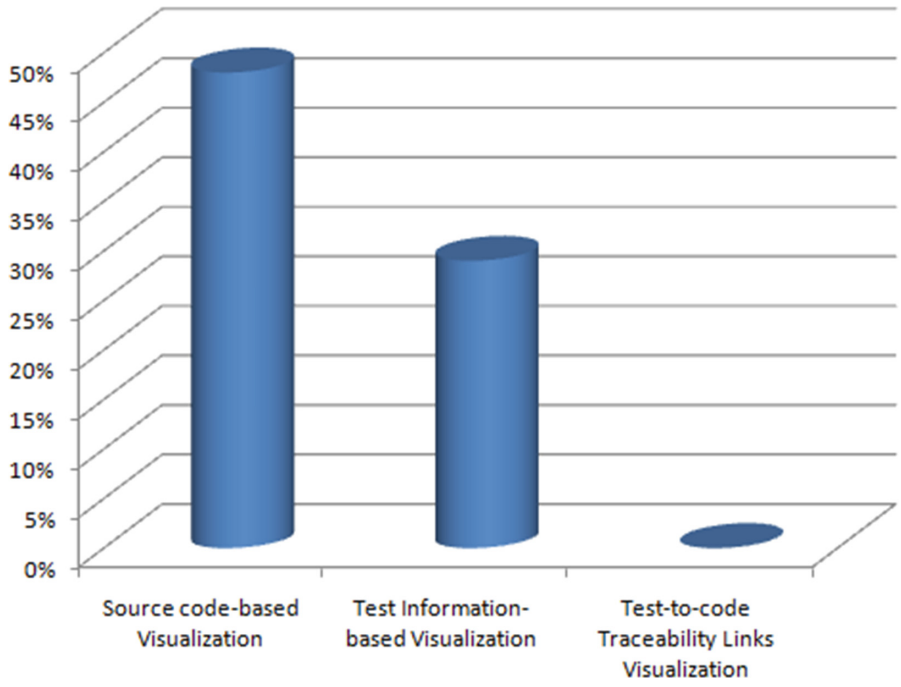

Fig. 1. Analysis of the results depending on **RQ1**

RQ2: What visualization techniques and tools are proposed to represent test-code relations?

A variety of visualization techniques have been developed to visualize different types of testing-related information. Examples of these techniques, as discussed in previous paragraphs, are graph-based, UML diagrams, metaphor. The graph-based visualization technique can be considered as the most popular visualization technique used; however, this may depend on the purpose of the deployment of the visualization technique. Various visualization techniques have many use cases, such as to obtain an overview of the code and test evolution, or improve the understanding of dependencies between code and test, or to support the visualization of testing information [25]. Concerning test-to-code traceability links, there is not much interest in developing approaches to visualize links between test and code. Most of the approaches have been developed to visualize relationships/links between requirements and other software artifacts (source code, design, test cases) such as graphs, traceability matrices, hyperlinks (cross-references) and lists, as shown in Fig. 2 [41]. Based on our investigations, it is clear that further research in the area of visualization of test-to-code traceability links is necessary. Below we provide some open questions that can help to reveal specific topics in this area for further research and try to give some answers as an example. We feel the following questions and directions are the most interesting ones.

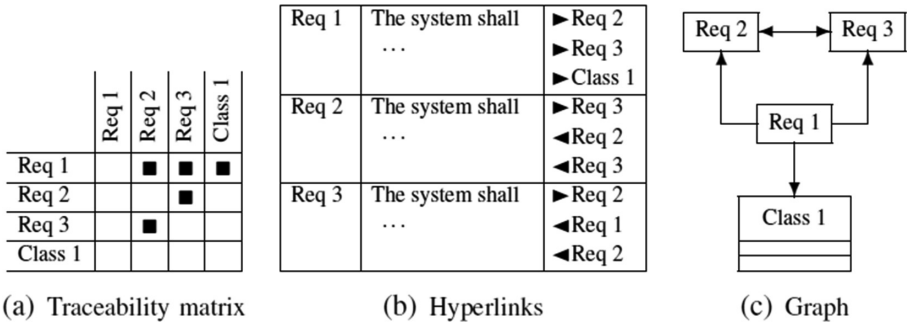

Fig. 2. (from [41]) Visualization techniques

Q.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).

Q.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, hyperlinks, lists, 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. But a 3D visualization also seems to be appropriate to display attributes of various items and relations.

Q3. What test and code items and properties should be objects and attributes in the visualization?
Relations can be visualized directly as objects in the visualization space (e.g. as lines between objects), or we can only map their properties to the attributes of the connected objects.

Q4. What are the criteria taken into account 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.

Q5. 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. 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.

Q6. 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.

5 Conclusion and Future Work

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, how the components are really related and change impact analysis can yield in false results, etc. Fortunately, visualization can aid this task. In this work, we analyzed the scope, advantages, and concerns surrounding the visualization of test-to-code traceability links. Test-to-code traceability links couple test cases to code elements based on the relationship that enables us to understand which code modules are tested by which unit tests. Therefore, visualization of such relations helps in improving the quality of software development, testing or maintenance processes, and lessen bugs. It makes updating the features of existing features of a piece of software or adding new features to it easier. Still, there is a room for research and development for better test-to-code traceability visualization techniques.

Further research then can then aim to make a statistical comparison of these inference methods. Work on addressing the questions mentioned above is in progress and will be presented in future papers.

Our future work is to develop a tool that visualizes and stores traceability links among unit test and its related code and can be integrated with a traceability links recovery tool that extracts links between code and tests. One of the goals of using visualization is to identify the disagreement between traceability links inferred from different sources. This might point out places where something is wrong with the tests and/or the code (at least their relationship) in a specific system.

References

1. Demeyer, S.: Object-oriented reengineering (2008)
2. Tamisier, T., Karski, P., Feltz, F.: Visualization of unit and selective regression software tests. In: Luo, Y. (ed.) CDVE 2013. LNCS, vol. 8091, pp. 227–230. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40840-3_33
3. D'Ambros, M., Lanza, M., Pinzger, M.: A bug's life visualizing a bug database. In: VISS 2007 – Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 113–120 (2007)
4. Araya, V.P.: Test blueprint: an effective visual support for test coverage. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1140–1142 (2011)
5. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, pp. 467–477 (2002)
6. Cornelissen, B., Van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, pp. 213–222 (2007)
7. Filipe, J., Maciaszek, L.A.: Evaluation of novel approaches to software engineering, July 2013
8. Balogh, G., Gergely, T., Beszedes, A., Gyimothy, T.: Using the city metaphor for visualizing test-related metrics. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, pp. 17–20 (2016)
9. Parizi, R.M., Lee, S.P., Dabbagh, M.: Achievements and challenges in state-of-the-art software traceability between test and code artifacts. IEEE Trans. Reliab. **63**(4), 913–926 (2014)
10. Koschke, R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. J. Softw. Maint. Evol. Res. Pract. **15**(2), 87–109 (2003)
11. Caserta, P., Zendra, O.: Visualization of the Static aspects of Software: a survey. IEEE Trans. Visual Comput. Graphics **17**(7), 913–933 (2011)
12. Erdemir, U., Tekin, U., Buzluca, F.: E-Quality: a graph based object oriented software quality visualization tool. In: 2011 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pp. 1–8 (2011)
13. Heim, P., Lohmann, S., Lauenroth, K., Ziegler, J.: Graph-based visualization of requirements relationships. In: 2008 3rd International Workshop on Requirements Engineering Visualization, REV 2008 (2008)
14. Wingkvist, A., Ericsson, M., Lincke, R., Löwe, W.: A metrics-based approach to technical documentation quality. In: Proceedings of the 7th International Conference on Quality of Information and Communications Technology, QUATIC 2010, pp. 476–481 (2010)
15. Varet, A., Larrieu, N., Sartre, L.: METRIX: a new tool to evaluate the quality of software source codes. In: AIAA Infotech@ Aerospace (I@ A) Conference, p. Draper Laboratory-, (2013)
16. Marcus, A., Comorski, D., Sergeyev, A.: Supporting the evolution of a software visualization tool through usability studies. In: Proceedings of the IEEE Workshop on Program Comprehension, pp. 307–316 (2005)
17. Boccuzzo, S., Gall, H.C.: Software visualization with audio supported cognitive glyphs. In: IEEE International Conference on Software Maintenance, ICSM, pp. 366–375 (2008)
18. Wettel, R., Lanza, M.: Visualizing software systems as cities. In: VISS 2007 – Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 92–99 (2007)

19. Cockburn, A., McKenzie, B.: 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: Changing Our World, Changing Ourselves - CHI 2002, no. 4, p. 203 (2002)
20. Wettel, R., Lanza, M.: CodeCity. In: Companion 13th International Conference on Software Engineering - ICSE Companion 2008, p. 921 (2008)
21. Balogh, G., Beszédes, A.: CodeMetropolis-code visualisation in MineCraft. In: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM) (2013)
22. Minecraft Official Website. http://minecraft.net/
23. Khaloo, P., Maghoumi, M., Taranta, E., Bettner, D., Laviola, J.: Code park: a new 3D code visualization tool (2017)
24. Lewerentz, C., Simon, F.: Metrics-based 3D visualization of large object-oriented programs. In: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis, pp. 70–77 (2002)
25. Agrawal, H., et al.: Mining system tests to aid software maintenance. Computer (Long. Beach. Calif.) **31**(7), 64–73 (1998)
26. Breugelmans, M., Van Rompaey, B.: TestQ: exploring structural and maintenance characteristics of unit test suites. In: WASDeTT-1 1st International Workshop on Advanced Software Development Tools and Techniques, no. i, pp. 1–16 (2008)
27. Van Rompaey, B., Demeyer, S.: Establishing traceability links between unit test cases and units under test. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, no. ii, pp. 209–218 (2009)
28. Qusef, A.: Test-to-code traceability: why and how? In: 2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies, AEECT 2013 (2013)
29. Marcus, A., Xie, X., Poshyvanyk, D.: When and how to visualize traceability links? In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, 8 November, pp. 56–61. ACM (2005)
30. De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: ADAMS re-trace: a traceability recovery tool. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, pp. 32–41 (2005)
31. Lago, P., Muccini, H., van Vliet, H.: A scoped approach to traceability management. J. Syst. Softw. **82**(1), 168–182 (2009)
32. Gilberto Filho, A.D., Zisman, A.: D3TraceView: A Traceability Visualization Tool
33. Rodrigues, A., Lencastre, M., De Cysneiros Filho, G.A.A.: Multi-VisioTrace: traceability visualization tool. In: Proceedings of the 2016 10th International Conference on the Quality of Information and Communication Technologies, QUATIC 2016, pp. 61–66 (2017)
34. Di Thommazo, A., Malimpensa, G., De Oliveira, T.R., Olivatto, G., Fabbri, S.C.P.F.: Requirements traceability matrix: automatic generation and visualization. In: Proceedings of the 2012 Brazilian Symposium on Software Engineering, SBES 2012, pp. 101–110 (2012)
35. Merten, T., Jüppner, D., Delater, A.: Improved representation of traceability links in requirements engineering knowledge using Sunburst and Netmap visualizations. In: 2011 4th International Workshop on Managing Requirements Knowledge, MaRK 2011 - Part 19th IEEE International Requirements Engineering Conference, RE 2011, pp. 17–21 (2011)
36. Chen, X., Hosking, J., Grundy, J.: Visualizing traceability links between source code and documentation. In: Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing - VL/HCC, pp. 119–126 (2012)
37. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. **39**(2), 276–291 (2013)

38. Eagan, J., Harrold, M.J., Jones, J.A., Stasko, J.: Technical note: visually encoding program test information to find faults in software. In: IEEE Symposium on Information Visualization 2001, INFOVIS 2001, pp. 33–36 (2001)
39. Van Rompaey, B., Demeyer, S.: Exploring the composition of unit test suites. In: ARAMIS 2008 - 1st International Workshop on Automated Engineering of Autonomous and Run-Time Evolving Systems, ASE 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 11–20 (2008)
40. Koochakzadeh, N., Garousi, V.: TeCReVis: a tool for test coverage and test redundancy visualization. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 129–136. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15585-7_12
41. Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. **9**(4), 529–565 (2010)