Visualizing Testing Results for Software Projects

Ahmed Fawzi Otoom, Maen Hammad, Nadera Al-Jawabreh, Rawan Abu Seini
Department of Software Engineering
Faculty of Prince Al-Hussein Bin Abdullah II for Information Technology
The Hashemite University, Zarqa, Jordan
{aotoom, mhammad}@hu.edu.jo, nadera.jawabreh@itc.hu.edu.jo, rwabusini@gmail.com

Abstract: *The key benefit of software visualization is to help in program understanding and in reducing the complexity of software systems. Test cases are essential artifacts to perform testing activities. There is large number of test cases to cover different aspects of the code. This paper proposes a visualization approach to represent test cases results and their relationship to object oriented software systems. The proposed visualization helps testers and program managers to get a clear and quick understanding about the test case, tested code and the results of testing. The proposed visualization represents test cases and source code at different views; method view, class view, package view and system view. The test cases are colored according to their execution results. We applied the proposed approach on two Java classes to illustrate the benefits and the usefulness of the proposed views.*

Keywords: *software visualization, software testing, program comprehension.*

1. Introduction

Testing is an essential activity in the software development life cycle. It determines whether the software meets its requirements and behaves as it is expected to. Testing is also important to make sure that the implementation phase has no hidden bugs or logical errors. During maintenance activities, testing should be performed after any code change. Software testing is usually performed through large number of test cases that cover all possible inputs and all code statements.

To generate test cases and perform testing, automated tools are used. JUnit is an example for such tool for Java code. However, test cases that are generated by such tools are usually presented in a textual form. When a large amount of source code is tested, large amount of test cases are generated; hence, test cases become difficult to be comprehended [10]. It is not an efficient process as testers need more time and effort to keep track on all generated test cases and check the results of each test case.

Recently, the area of test information visualization has been widely targeted. For example, Cornelissen et al. [3] presented a visualization approach based on UML sequence diagram to visualize test information. While Jones et al. [8] proposed a technique that visualizes all statements that are executed by test suites and facilitates the location of faults in these statements. But, there is a lack for tools and approaches that visualize the relationship between generated test cases and object oriented projects.

In this paper, we propose a visualization approach to represent test cases and their relationships with object oriented source code. The proposed visualization approach provides information about the number of test cases and their results for different code components by a variety of views. The views are easy to understand and help in determining which part of the code was tested. Thus, it helps software testers in the process of understanding the relationship between source code and its related test cases by providing an effective way to better and quick understanding of the information being presented. Our approach includes a sequence of steps to create different views to visually comprehend and explore the code elements of object oriented projects being tested with test cases associated with each code element.

The proposed visualization represents software systems at different views to provide variety of information to tester. The first view is the method view. This view visualizes tested methods and the results of their test cases. The second view is the class view, which presents the test cases that cover all class's methods. The third view is the package view, which visually describes all test cases that cover package's classes. The last type of views is the system view, which provides an overview of the project elements (packages and classes) and all test cases associated with them.

The main research contributions of this work are:

- A visualization approach to model test cases and their results and how they are connected to the source code.
- A lightweight approach to automatically test and visualize testing results for java projects.

This paper is structured as follows. Section 2 presents a review of existing visualization approaches. Section 3 presents the proposed visualization technique. The complete approach that is used to generate the proposed visualization is presented in Section 4. Section 5 shows a detailed example that illustrates the usefulness of the proposed approach. Finally, Section 6 presents the conclusion of our work and future work suggestions.

2. Related Work

Software testing is an important activity in the software development life cycle. Software is tested to identify defects that the software may fail. Software testing includes test cases that execute the program and an expected outcome declaration [6]. Test cases are used to determine whether the software being tested works correctly or not. IEEE Standard 610 (1990)[7] 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".

Many techniques and approaches have been proposed to generate test cases. Prasanna et al. [1] surveyed several approaches in generating test cases. The benefits of running test cases lie in:

1. Identifying the defects in software system.
2. Improving the software quality.
3. Reducing the maintenance effort and cost.
4. Making sure that the software meets users' needs.

Test cases are usually used to evaluate software systems and detect the program faults. The larger the programs, the larger the test cases executed, thus a huge amount of data will be produced which is considered as difficult to be interpreted as textual form. Visualization of test cases 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.

There has been large amount of literature that deals with visualization of test information. Muto et al. [15] proposed a method that provides visualization of quality of software. The proposed method visualizes the compatibility between implementation and specification from two angles: normal testing (by unit testing) and static checking. Another visualization technique has been presented in the work of [19]. Its main aim is to help software testers determine the location of test suite, its relation with the production code, and which parts of code are covered by test cases. Recently, the authors in [18] proposed generating testing diagrams to visualize all test cases that test the software system by combining test cases and UML diagrams.

On the other hand, a wide number of tools have been proposed for the task of visualization of testing information. An example of such tools is (TeCReV) [14] 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. Fault localization is a main objective of other tools such as TestQ [2] and ϗSuds [1]. Another tool example has been presented by [17]. It visualizes all the information about the test results and the parts of code that have not been tested adequately. In the work of [5], a tool named ChronoTwigger has been proposed. It provides 2D and 3D visualization based on Beyer's algorithm converge. The tool aims to visualize co-evolution of source and test files to help in analyzing the relationships between the software development processes and testing.

In line with the aforementioned works, there has been extensive research works related to code visualization. In the work in [8], the authors presented a visualization technique that helps the developer identify the statements that contain the faults and these which are suspected to contain faults. The input of the technique is a source code of testing program. A visualization technique provides an understanding of which program statements are executed by test suite as well as the statements which are suspicious to contain faults. Dershem et al. [4] presented a java application visualization tool to visualize class components and the interactions among them. This tool aims to simplify the understanding of the object oriented concepts. A similar idea has been implemented by the authors in [11] where the authors introduced a visualization method that visualizes classes and their relationships. The method is used to visualize individual class or clustered classes and focuses on the classes that continuously change. Other examples of similar ideas can be found in [12-13,16].

In this paper, we visualize code elements for object oriented software projects with their testing results in order to help testers in determining what parts of the source code have been tested. Different from the existing methods, the proposed visualization presents the relationship between test cases and object oriented source code to get a clear and quick understanding about the test case, the tested code and the result of testing. Moreover, visualization is displayed via four different views: method, class, package and system view.

3. The Proposed Visualization

This section details our proposed visualizations for code elements and test cases. The visualization aims to create visualization for test cases as well as their relevant classes and methods. Software testers should be able to quickly understand the components of source code (i.e. classes and methods) and the number of applied test cases with their results. The test cases are colored according to their execution results (pass or fail).

To clarify the idea, consider a class named *C* that contains a set of methods *m1, m2… mn*. Each method *m* is tested by a set of test cases *t1, t2,….tn*. Based on the execution, each test case has one of the two result: pass or fail. Test cases are colored according the execution results; either green to denote passed or red to denote failed.

The fundamental objective of using visualization in our approach is to provide an overview about the relationships between object oriented code elements and test cases that are associated with these code elements.

The proposed visualizations view the number of test cases for each code element and the result of each test case. The visualization is displayed by four different views:

1- Method view: this view visualizes the relationship between methods and their test cases.

2- Class view: this view visualizes the relationship between a class and all test cases applied to its methods.

3- Package view: this view visualizes the relationship between a package and all test cases applied to its classes.

4- System view: all packages with all classes as well as the total number of test cases. This view provides general and comprehensive view for all test cases applied on the project.

3.1 Method View

We used JUnit to automatically generate test cases. These test cases are invoked by the methods under testing. The name of each generated test case contains the name of the method. For example, the test case *testCalcAverage* is generated for the method *CalcAvearage*. So, the methods and test cases names are used to connect test cases with their methods.

Consider the implementation of the following method that calculates the area of a rectangle.

```
public static int Area(int length, int width) {
   return length*width;
}
```

The method is tested by five test cases; *test1Area()*, *test2Area()*, *test3Area()*, *test4Area()* and *test5Area()*. These five test cases are shown in figure1.

```
@Test
        public void test1Area() {
        int area = Rectangle.Area(12, 12);
        int actual = area;
        assertEquals(144,actual);
        }
@Test
        public void test2Area() {
        int area = Rectangle.Area(0, 12);
        int actual = area;
        assertEquals(0,actual);
        }
@Test
        public void test3Area() {
        int area = Rectangle.Area(-1, 12);
        int actual = area;
        assertEquals(12,actual);
        }
@Test
        public void test4Area() {
       int area = Rectangle.Area(10, 12);
       int actual = area;
       assertEquals(120,actual);
        }
 @Test
      public void test5Area() {
      int area = Rectangle.Area(10, 10);
      int actual = area;
      assertEquals(20,actual);
        }
```

Fig1: Test cases generation for method *Area*

Three of them are passed test cases and the other two are failed. Figure 2 presents the proposed visualization for the five test cases of the *Area* method. Each test case is represented by a rectangle. Passed test cases are colored with green whereas the failed test cases are colored with red.

3.2. Class View

The proposed visualization of a class shows the results of all test cases on its methods. A class contains one or more methods tested by different passed/failed test cases. For example; consider a class *C* that consists of three methods *M1*, *M2* and *M3*. Method *M1* is tested by two passed test cases and one failed test case. Method *M2* is tested by three passed test cases and two failed ones. Method *M3* is tested by two passed and two failed cases. The result is visualized in Figure 3.

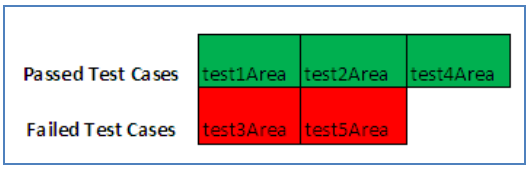

Fig.2: Visualization of test cases shown in Figure1

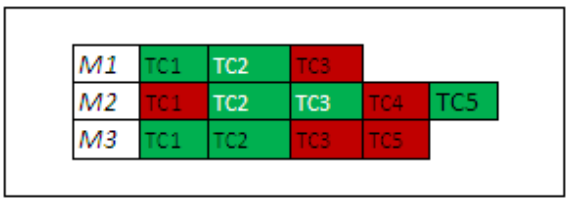

Fig.3: An example of the class view

3.3. Package View

A package contains a group of related classes. So, visualizing test cases of a package combines the visualization of test cases for all package's classes and methods. For example, a package *P* contains three classes *C1*, *C2*, and *C3*. Each class has different number of methods were each method is tested by different number of passed/failed test cases. The package view visualization of package *P* is shown in Figure 4. As shown in the figure, each class is represented by a block that contains its methods and their test cases.

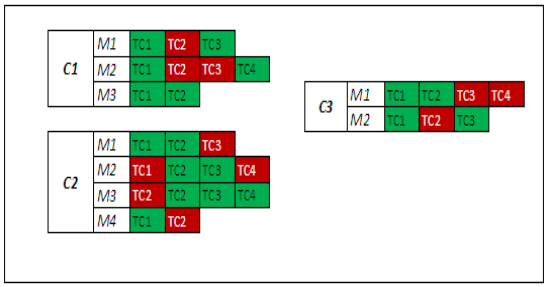

Fig.4: An example of the package view

3.4. System View

System view visualization includes all packages that belong to a software project. Each package is represented by a cloud shape with blue filled color. A package is connected to its classes, and also to its sub-packages.

In the system view visualization, the class is represented by circle style node with yellow filled color. Each class has a number of methods. The size of the circle reflects the number of its methods. Each method is tested by number of test cases.

Test cases are represented by small circles within the relevant class methods. The passed test case is colored by green, while the failed test case is colored by red. Table 1 summarizes the components of system view visualization.

Table 1: The representation of the system view components.

Component	Shape and Color
Package	
Class	
Test case (passed)	
Test case (failed)	

Moreover, figure 5 shows an example for the system view visualization, which provides the following information for testers:

1. The total number of dots in the circles represents the number of test cases for the methods in that class.
2. The number of red dots represents the number of failed test cases and green dots represent passed test cases.
3. No green or red dots means that there are no test cases for the methods of the class. Testers should reconsider their testing to cover these methods.

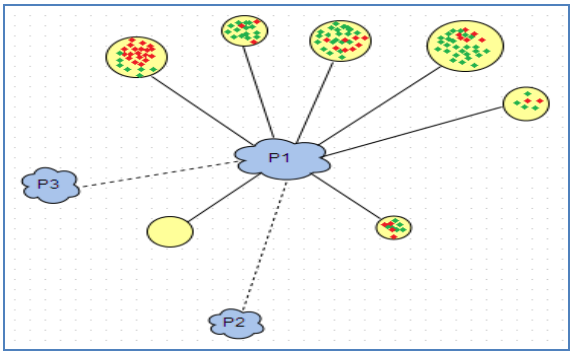

Fig.5: The system view visualization

4. The Approach

Now, how the proposed visualizations can be automatically generated? To answer this question, we propose a lightwieght approach to automatically analyze a software project under testing and generate the visualizations. The approach can be realized as a tool to automatically generate, test and visualize testing results. The proposed approach is presented in figure 6. It consists of the following main steps:

1) Input

The process starts with the source code that needs to be tested. The input will be a Java source code for a method, a class or a package. Since the focus of our visualization is on tested methods, classes that contain at least one method are included in the visualization. Interfaces and inner classes are ignored. Also abstract classes with only abstract methods are ignored too.

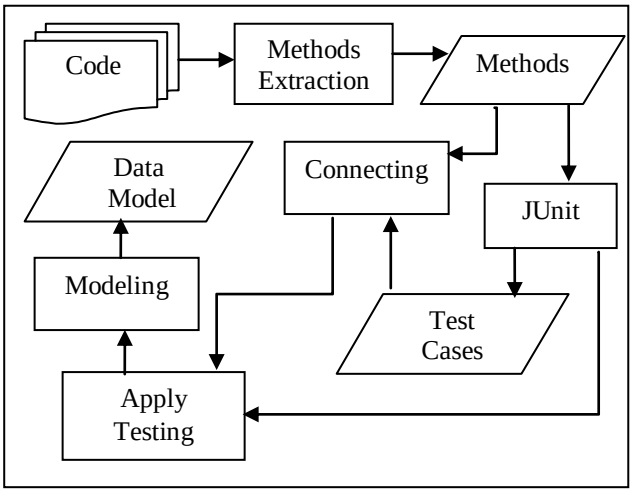

Fig.6: The proposed approach

2) Generate test cases

In the next step, the input source code is analyzed to extract all the methods for testing. For each method, a set of test cases is generated by using the JUnit Eclipse plugin tool [9]. Then, for each class, a corresponding test class is created that includes a set of testing methods for each one of class's methods.

3) Connect test cases to methods

In this step, the relationship between generated test cases and methods are identified. This is done by analyzing the names of generated test cases and the names of methods. The names of both test cases and methods are compared to find the corresponding test cases for each method. Based on this comparison, the relationships are identified.

For example; the following five test cases are generated for method *CalcMax* in class *Calculator; test1CalcMax, test2CalcMax, test3CalcMax, test4CalcMax, test5CalcMax*. The names of test cases are analyzed to extract the names of methods.

The result of this step is a list of methods' names where each method name is connected to a set of test cases names.

4) Runing test cases

Based on the list generated from Step 3, each method is tested by the test cases for that method. JUnit is used to run the testing suites. The results of testing will be two disjoint sets for each method. The first set contains the names of passed test cases and the second set contains the names of failed cases.

In this step, statistics are calculated after the testing is completed. These statistics include the number of passed/failed test cases for each method, class and package. This information is used by the visualizing tool to generate the different views.

5) Visualizations

In the last step, the data model for the visualizations is generated. This data model is used to render the views. The model mainly includes the following information:

- Sizes, locations and colors for square and rectangle blocks for all test cases on method, class and package levels.
- Sizes, locations, colors and contents for circles, lines and clouds that are used to model system view visualization.

5. Detailed Example

In this section, we clarify our approach using two java classes. The input is a Java package named *edu.proj* that consists of two classes: *PiratChest* and *Calculator*.

5.1 The PiratChest class

The *PiratChest* class contains four methods to be tested; *addGold*, *checkGold*, *removeGold*, and *equals* methods. Each method consists of a set of statements that are grouped together to perform a specific task. As early mentioned, the generation of test cases is performed by JUnit. The testing process starts with the creation of a test class for *PiratChest* class under the name *PiratChestTest*. Each method is tested by a number of passed/failed test cases.

The name of each test case is related to methods' name. Thus, the test cases *testaddGold*, *testcheckGold*, *testremoveGold*, and *testequals*, are generated for the methods; *addGold*, *checkGold*, *removeGold*, and *equals*.

In case of generating more than one test case for a method, a number is used in the name. For example, a method *addGold* is tested by four test cases: *test1addGold, test2addGold, test3addGold* and *test4addGold*.

The test cases generated in *PiratChestTest* class test are as follows:

- *addGold* method is tested by four test cases as previously mentioned (see figure 7 for an illustration).
- *checkGold* method is executed by three test cases: *test1checkGold*, *test2checkGold*, and *test3checkGold*.
- *removeGold* method is executed by five test cases: *test1removeGold, test2removeGold, test3removeGold, test4removeGold,* and *test5removeGold*.
- *Equals* method is executed by three test cases: *test1equals, test2equals*, and *test3equals*.

The next step is to run test cases to get the results of code testing. Test cases are automatically applied by JUnit. If the actual output for an executed method is the same of the expected output for executing that method

with test cases, then the test cases are passed, otherwise they are failed. The outputs of the execution of test cases in our example are shown in table 2.

As illustrated in table 2, *addGold* method is tested by four test cases, two of them are passed; *test1addGold* and *test4addGold*, while the two remaining are failed. The visualization of test cases that are executed for *addGold* method is illustrated in figure 8. The green color represents the passed test cases, while the red color represents the failed.

For the remainnig methods; *checkGold* is tested by three test cases, two of the three tested cases are passed; *test1checkGold* and *test2checkGold*, and the third one failed. The method view visualization for *checkGold* method is presented in figure 9.

```
public class PiratChestTest {

    @Test
    public void test1AddGold() {
        PiratChest chest=new PiratChest(10);
        chest.addGold(20);
        int actual=chest.checkGold();
        int expected=30;
        assertEquals(expected,actual);
    }
    @Test
    public void test2AddGold() {
        PiratChest chest=new PiratChest(0);
        chest.addGold(20);
        int actual=chest.checkGold();
        int expected=0;
        assertEquals(expected,actual);
    }
    @Test
    public void test3AddGold() {
        PiratChest chest=new PiratChest();
        chest.addGold(20);
        int actual=chest.checkGold();
        int expected=20;
        assertEquals(expected,actual);
    }
    @Test
    public void test4AddGold() {
        PiratChest chest=new PiratChest(10);
        chest.addGold(20);
        int actual=chest.checkGold();
        int expected=30;
        assertEquals(expected,actual);
```

Fig.7: Test cases for method *addGold*

Table 2: Testing results for class *PiratChest*

method	#Test cases	#Passed	#Failed
addGold	4	2	2
checkGold	3	2	1
removeGold	5	3	2
Equals	3	2	1
Total	15	9	6

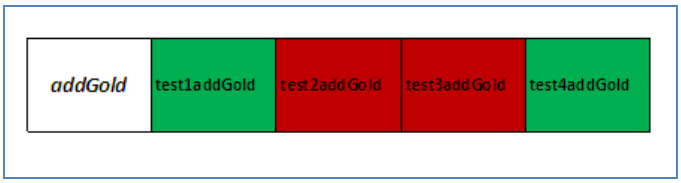

Fig.8: Visualization of test cases (*addGold method*)

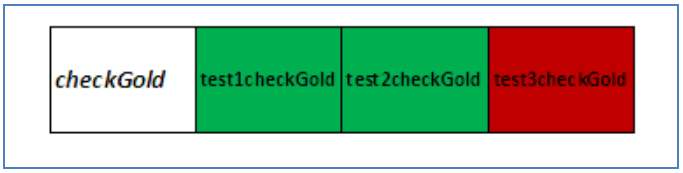

Fig.9: Visualization of test cases (*checkGold method*)

For *removeGold* method, it is tested by five test cases. The passed test cases are: *test2removeGold*, *test4removeGold*, and *test5removeGold*, while *test1removeGold* and *test3removeGold* failed. Figure 10 shows the method view for the *removeGold* method. Finally, the *equals* method, three test cases passed and one failed. The passed test cases are *test2equals*, *test3equals*. Visualization of these test cases is presented in figure 11. According to class *PiratChest*, the number of test cases for all methods is fifteen. Nine of them were passed and six test cases failed. Figure 12 presents the class view visualization for class *PiratChest*.

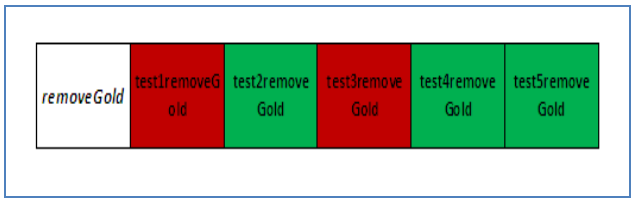

Fig.10: Visualization of test cases (*removeGold method*)

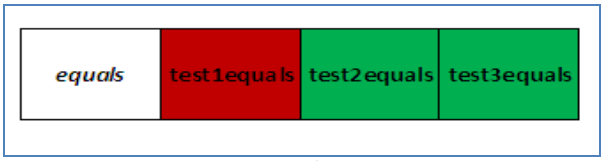

Fig.11: Visualization of test cases (*equals method*)

5.2 The Calculator Class

We follow the same steps above to visualize *calculator* class with the related methods and test cases. There are four methods included in *calculator* class; *add*, *subtract*, *multiply*, and *divide*.

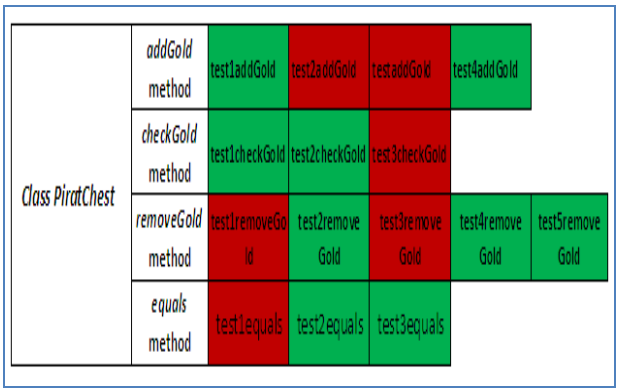

Fig.12: Class view for *PiratChest*

The summary of testing information of the class *Calculator* is described in Table 3. Each method is tested by a number of test cases. The method view visualization of each method is presented in figures [13-16].

Table 3: Testing results for class *Calculator*

method	#Test cases	#Passed	#Failed
Add	3	2	1
subtract	4	3	1
multiply	5	3	2
devide	5	3	2
Total	17	11	6

Fig. 13: Visualization of test cases (*add* method)

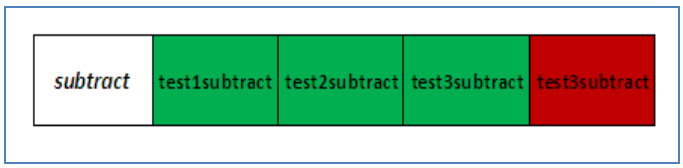

Fig. 14: Visualization of test cases (*subtract* method)

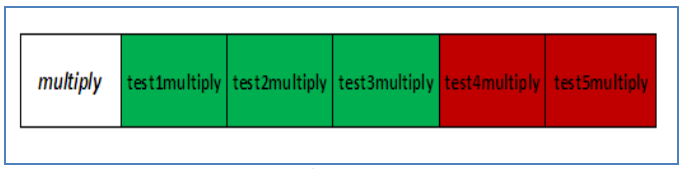

Fig. 15: Visualization of test cases (*multiply* method)

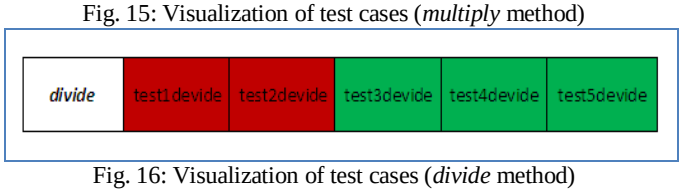

Fig. 16: Visualization of test cases (*divide* method)

The visualization of class *Calculator* as a whole appears in figure 17.

Now we have to visualize the java code package *edu.proj* with two related classes; *PiratChest* and *Calculator* (see figure 18). The view provides useful information about the number of classes, methods and total failed/passed test cases. The package used in this example is part of a system that includes a set of other packages. The visualization of the system will appear as it is shown in figure 19. The figure shows the system view visualization. The package is represented by cloud style node with blue color, while the class is represented by circle style node with yellow color.

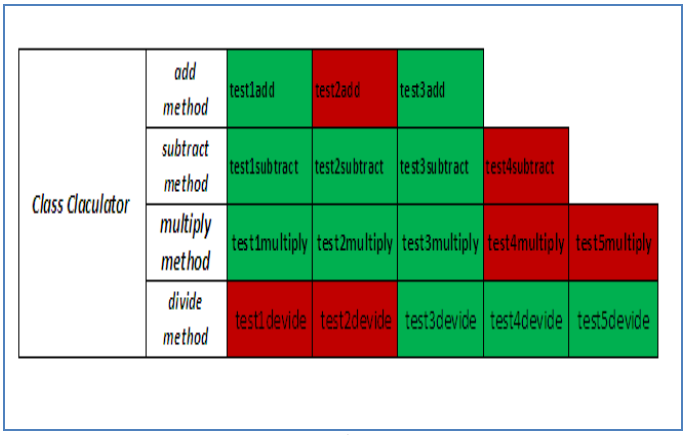

Fig. 17: Class view of (class *Calculator*)

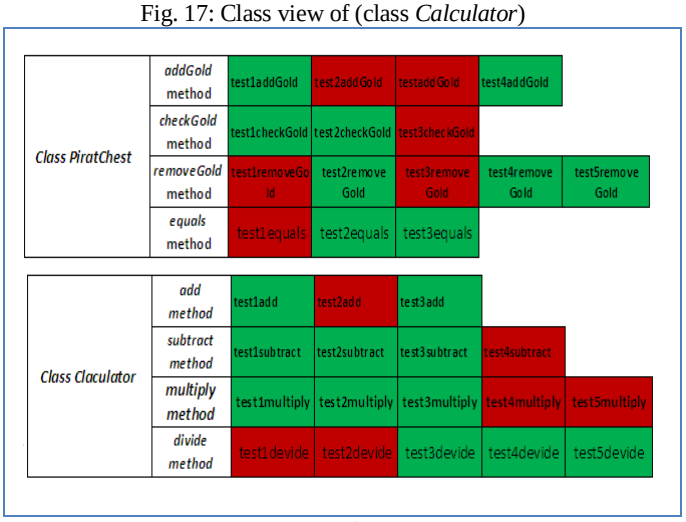

Fig. 18: Package view of (*package edu.proj*)

In each class, a group of test cases that have tested the class are appeared as small color dots. Green dots represent passed test cases and the red dots represent the failed test cases. The number of dots represent the total number of test cases. Each class connects to relevant package by filled line, whereas packages connect to

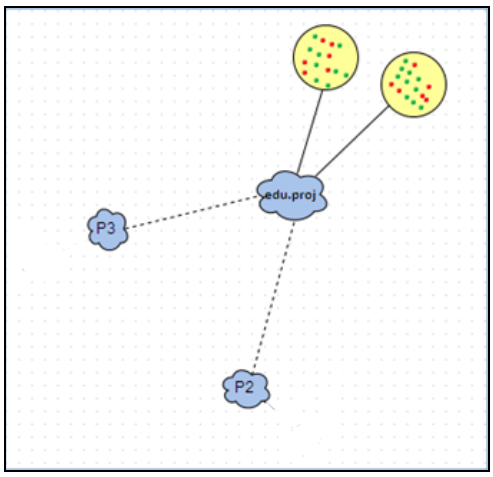

Fig. 19: System view (*edu.proj package*)

each other by dotted lines. The size of package depends on the number of classes contained.

6. Conclusions and Future Work

The paper presented a visualization approach to help testers keep track and understand the output of the testing process. The proposed visualizations help in determining what parts of the source code have been tested, the number of test cases and the results of each test case. The views keep testers aware with the number and results of test cases for each method. The proposed visualization includes different views for software systems: method view, class view, package view and system view. Throughout these views, we can easily explore and comprehend the testing results and the parts of program being tested. An approach is also presented to automatically test and generate the data model for the views from source code. The future work aims to visualize more information for testers. For example, the number of branches and statements for each method.

References

[1] Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Ghosh, S., and Wilde, N., "Mining system tests to aid software maintenance", *Computer*, vol. 31, no. 7, pp. 64–73, 1998.

[2] Breugelmans, M., Van Rompaey, B., " TestQ: Exploring structural and maintenance characteristics of unit test suites", *Proc. of the 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[3] Cornelissen, B., van Deursen, A., Moonen, L., and Zaidman, A., "Visualizing test-suites to aid in software understanding", *Proc. of the 11th European Conference on Software Maintenance and Reengineering*, 2007.

[4] Dershem, H. L., Vanderhyde, J., "Java class visualization for teaching object-oriented concepts", *ACM SIGCSE Bulletin*, vol. 30, no. 1, pp. 53-57, 1998.

[5] Ens, B., Rea, D., Shpaner, R., Hemmati, H., Young, J. E., Irani, P., "ChronoTwigger: A Visual Analytics Tool for Understanding Source and Test Co-evolution", *Proc. of the Second IEEE Working Conference on Software Visualization (VISSOFT), 2014*, pp. 117-126, 2014.

[6] Fraser, G., Arcuri, A., "Whole test suite generation", *IEEE Transactions on Software Engineering* , vol. 39, no. 2, pp. 276-291.

[7] IEEE Standard Glossary of Software Engineering Terminology 610.12-1990.

[8] Jones, A., Harrold, M. J., Stasko, J., "Visualization oftest information to assist fault localization", *Proc. of the 24th International Conference on Software Engineering*, pp. 467-477, 2002.

[9] Junit- Plug with Eclipse, www.tutorialspoint.com/junit/junit_plug_with_eclipse.htm

[10] Kamimura, M., Murphy, G. C., "Towards generating human-oriented summaries of unit test cases", *Proc of the IEEE 21st International Conference on Program Comprehension (ICPC)*, pp. 215-218,2013.

[11] Kang, B. K., Bieman, J. M., "Using Design Cohesion to Visualize, Quantify, and Restructure Software", *SEKE*, pp. 222-229, 1996.

[12] Kleyn, M. F. and Gingrich, P. C., "Graphtrace–understanding objectoriented systems using concurrently animated views", *ACM Sigplan*, vol. 23, no. 11, pp. 191-205, 1988.

[13] Knight, C., Munro, M., "Virtual but visible software", *Proc. of IEEE International Conference on Information Visualization*, pp. 198-205, 2000.

[14] Koochakzadeh, N., Garousi, V., " Tecrevis: a tool for test coverage and test redundancy visualization ", *In Testing–Practice and Research Techniques*, Springer Berlin Heidelberg, pp. 129-136, 2010.

[15] Muto, Y., Okano, K., Kusumoto, S., " A Visualization Technique for the Passage Rates of Unit Testing and Static Checking with Caller-Callee Relationships ", *Proc. of the 2011 Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 336-341, 2011.

[16] Pinzger, M., Gall, H., Fischer, M., Lanza, M., "Visualizing multiple evolution metrics", Proc. *of the 2005 ACM symposium on Software visualization*, pp. 67-75.

[17] Tamisier, T., Karski, P., Feltz, F., "Visualization of Unit and Selective Regression Software Tests", *Cooperative Design, Visualization, and Engineering*, Springer Berlin Heidelberg , pp. 227-230, 2013.

[18] Urata, S., Katayama, T., "Proposal of Testing Diagrams for Visualizing Test Cases", *Proc. of IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*,pp. 483-484, 2013.

[19] Van Rompaey, B., Demeyer , S., "*Exploring the composition of unit test suites ", Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 11-20, 2008.