A Web-Based Software Testing Tool with Visualization for Java Programs

A Web-Based Software Testing Tool with Visualization for Java Programs

Mochamad Chandra SAPUTRA[a], Tetsuro KATAYAMA[b], Hadi SUYONO[c], Achmad BASUKI[d]

Abstract

Visualization is one of the important techniques for software testing. The purpose of software testing is not only to find errors, but also to understand the behavior of a code through visualization. This research implements a web-based software testing tool for java programs using statement and branch coverage, and visualizes the result of testing. The research displays a measurement result of statement and branch coverage as a percentage of a successful tested code.

The tool can inform the user using visualization to understand the behavior of the tested code and its testing status. The correlation between visual information and software testing, visual information of the tested code describes the behavior of the code as a sequence of the executed lines. Our implementation of web-based software testing tool for java programs significantly reduces the time consume for testing a software code, 743 ms using our testing tool and over 4 minutes using manual testing. Hence, the efficiency of the unit testing for java programs is improved.

Keywords: Visualization on software testing, Web application, Random testing, Java programs

1. INTRODUCTION

Finding errors in early stages of the software development is an important task to save costs. A software testing tool is used to find errors included in the program during its execution. A good software testing tool should give high probability in finding those errors[1]. The testing process should only require a minimum efforts in finding errors and knowing the behavior of a code with minimum times.

Testing can be the process of validating and verifying whether the software product meets the business and technical requirements that guided its original design and development. Validation is a set of tasks that ensures the software has been built is traceable to the technical requirements, while verification refers to a set of tasks that ensures the software is correctly implementing specific functions[2].

Web applications are among the fastest growing classes of software systems today. These applications are being used to support a wide range of important activities for example: business transactions, scientific activities, and medical activities[3]. The main advantages of adopting the web applications are (1) no installation costs, (2) automatic upgrade with new features for all users, (3) universal access from any machine connected to the Internet, and (4) independence from the operating system of clients[4].

Visualization is very important in software testing. There are three different levels of software defects visualization technology: level of system code, level of architecture, and level of system behavior[5]. Since software testing is a long and complex process with probably huge result data collection, visual information will provide testers with a quick and general perspective, which leads to a better understanding of a system software behavior[6]. Implementing software testing as a web application for visualizing the result of testing is one of the solution to easily understand the behavior of a software code.

Many tools for software testing have been proposed, such as JunitPerf[7], TestNG[8], and so on. JUnitPerf is a collection of JUnit test decorators used to measure the performance and scalability of functionality that is contained within existing JUnit tests. TestNG is a testing framework inspired by JUnit and Nunit, but introduces new functionalities and is easy to use. The behavior of a software code should sufficiently understood by knowing the workflow of the code, which parts are executed first, how many iteration and so on. However, it is not easy

a) Master Student Double Degree Program, Faculty of Engineering, Department of Computer Science and Systems Engineering University of Miyazaki, Faculty of Engineering, Department of Electrical Engineering, Brawijaya University.

b) Associate Professor, Institute of Education and Research for Engineering, University of Miyazaki.

c) Associate Professor, Faculty of Engineering, Department of Electrical Engineering, Brawijaya University.

d) Associate Professor, Information Technology and Computer Science Program, Brawijaya University.

to understand the behavior of a code by using those commonly used software testing tools.

To display the testing process and to understand the behavior of a code, we have implemented a web-based software testing tool with visualization for java programs. The tool used random testing with statement coverage and branch coverage for java programs. The experiment showed the calculation results from statement coverage and branch coverage as a percentage of a successful tested code and visualized the behavior of a software code. The testing tool informs the users with visualization to know the behavior of tested code and testing status. The previous tool is already existed[9] and this research extends the tool to improve its usefulness.

The outline of this paper is as follows. Section 2 describes the problems in unit testing tools that are currently proposed. Section 3 describes the specifications and implementation policies of the testing tool. Section 4 discusses the testing tool. Section 5 discusses the related work. Finally, Section 6 summaries this research and discusses the future issues.

2. PROBLEMS IN TRADITIONAL UNIT TESTING TOOLS

The testing objective is always to test the code, whereby there is a high probability of discovering all errors that exist in the software. This testing objective for the software functions also works according to the software requirements specification (SRS) for functionality, features, facilities, performance. It should be noted, however, that testing will detect errors in the written code[10]. Some of the testing objectives and their criteria include:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A successful test is one that does uncover an as-yet-undiscovered error.

After the programmer finishes coding or modifying the program, they need to test the code to evaluate its quality, and identify defects and problems to improve it. The goal is also to reduce the time required for the testing process and fully understand the behavior of the code.

The previous research used a statement and branch coverage method for measurement the successful tested code. Application results show a percentage successful tested code. The value of the percentage seen in by the number of line code executions. The test data are generated by a random test data generator and then automatically tested.

There are many lines in those tested codes. The testing process is executed on each line of tested code and calculation of the lines that executed several times are done also using statement and branch coverage. However, the previous research result only shows the number of these lines and how they were executed several times without any code visual information[9]. Visual information shows the known behavior of the tested code as sequence of the line executed by the code. The previous application is not portable, which means the application can not be accessed from just any where.

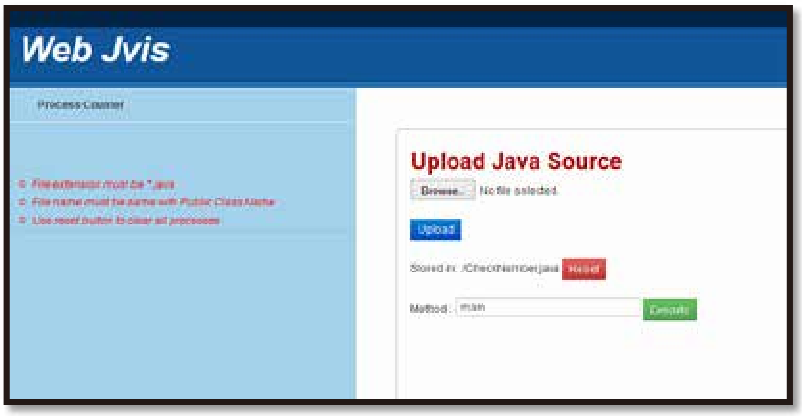
3. SPECIFICATIONS AND IMPLEMENTATION POLICIES FOR THE APPLICATION

Traditionally, software testing techniques can be broadly classified into black box testing and white box testing. In the black box method, the outside world comes into contact with the test through a functionality of the software. An example of black box testing is testing the application interface, internal module interface, or the input/output description of a batch process. Black box tests check whether interface definitions are adhered to in all situations[10]. Product acceptance tests completed by the customer are also considered black box tests.

White box tests are developer tests. They ensure that each implemented function is executed at least once and checked for correct behavior[11]. Examination of white box testing results can be done with the system specifications in mind.

The white box testing method includes statement coverage and branch coverage. Statement coverage is code that is executed in such a manner that every statement of the application is executed at least once[12]. The research uses statement coverage called C0, which helps in ensuring that all the statements are executed without any side effects. This method is also called line coverage or segment coverage.

In statement coverage testing, we make sure that all our code blocks are executed. We also identify which blocks failed to execute when using the testing tool.

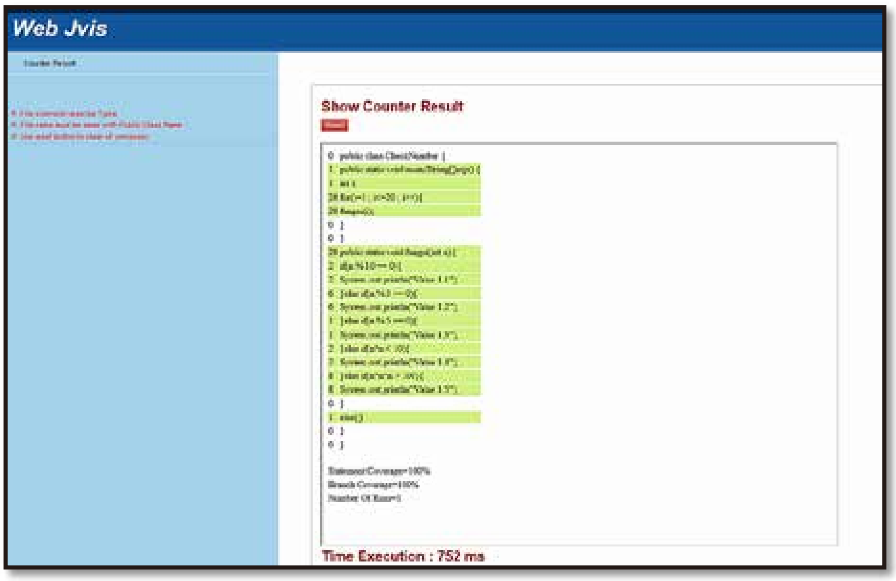


- File extension must be *.java
- File name must be same with Public Class Name
- Use reset button to clear all processes

Stored in: /CheckNumber.java

Method : main

Fig. 1. GUI of the implemented web-based software testing tool.



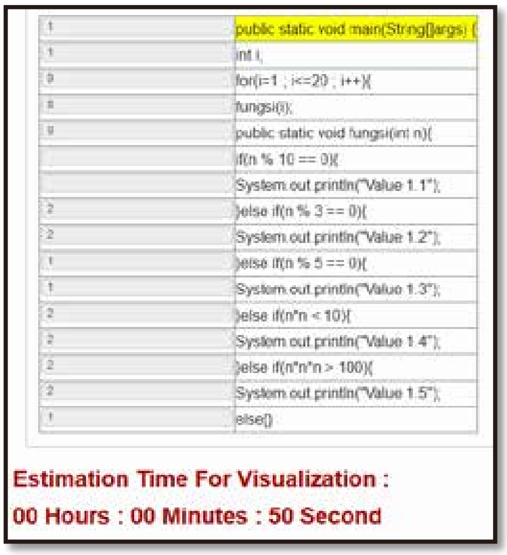
- File extension must be *.java
- File name must be same with Public Class Name
- Use reset button to clear all processes

```
0  public class CheckNumber {
1  public static void main(String[] args) {
1  int i;
20 for(i=1 ; i<=20 ; i++){
20 fungsi(i);
0  }
0  }
20 public static void fungsi(int n){
2  if(n % 10 == 0){
2  System.out.println("Value 1.1");
6  }else if(n % 3 == 0){
6  System.out.println("Value 1.2");
1  }else if(n % 5 == 0){
1  System.out.println("Value 1.3");
2  }else if(n*n < 10){
2  System.out.println("Value 1.4");
8  }else if(n*n*n > 100){
8  System.out.println("Value 1.5");
0  }
1  else{}
0  }
0  }

StatementCoverage=100%
Branch Coverage=100%
Number Of Error=1
```

Time Execution : 752 ms

Fig. 2. A static result of the testing tool.



1	public static void main(String[]args) {
1	int i;
9	for(i=1 ; i<=20 ; i++){
8	fungsi(i);
9	public static void fungsi(int n){
	if(n % 10 == 0){
	System.out.println("Value 1.1");
2	}else if(n % 3 == 0){
2	System.out.println("Value 1.2");
1	}else if(n % 5 == 0){
1	System.out.println("Value 1.3");
2	}else if(n*n < 10){
2	System.out.println("Value 1.4");
2	}else if(n*n*n > 100){
2	System.out.println("Value 1.5");
1	else{}

Estimation Time For Visualization :
00 Hours : 00 Minutes : 50 Second

Fig. 3. A dynamic result for visual information of a tested code.

For calculating statement coverage, we use the following formula[13]:

- Statement Coverage = (Total Statements execute) / (Total Number of Executable Statements in Program)*100(%).

Test coverage criteria requires enough test cases that each condition in a decision takes on all possible outcomes at least once, and each point of entry to a program or subroutine is invoke at least once. That is, every branch (decision) can be taken each way, true and false.

This research uses branch coverage, which is called C1. It helps in validating all the branches in the code and making sure that no branch leads to abnormal behavior of the application[14].

For calculating branch coverage, we use the following formula[15]:

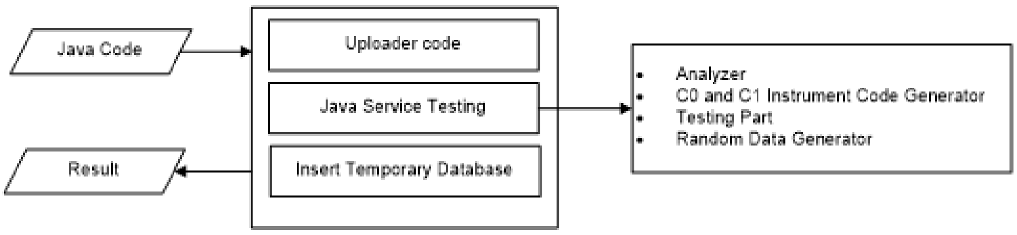


Fig. 4. Design of the testing tool.

- Branch coverage = (Tested decision outcomes / total decision outcomes)*100(%).

C0 and/or C1 are used as an exit criterion for the software testing.

Random data testing executed the application with input data generated at random. Typically, testers pay no attention to expected data types. They feed a random sequence of numbers, letters, and characters into numeric data fields[11)].

Automated software testing is an activity that seems to have obvious benefits. Tests may be executed swiftly, are more consistent, and may be repeated at various times without increasing cost[16)].

Automated software testing simulates the system behavior by testing tools. The test actions performed on the application are specified in code (scripts and test classes). In a context where required tests are not possible or viable for execution manually, automated software testing becomes very important.

Visual information is much easier to explain using demonstrations than it is using words. However, to be understood clearly, the data that displays should be familiar to the audience and interesting[17)].

A web application is a system that typically is composed of a database (or the back-end) and web pages (the front-end), which users interact with over a network using a browser[18)]. The testing tool will analyze and read the original code based on the information of the specified file and then insert it into a temporary database for javascript output.

3.1 Specification

In this section, we describe the spesification and implementation of the web-based software testing tool. Figure 1 shows an overview of the implemented testing tool. The following text describes each part.

1. Browse

Used to select an original file in our directory, the file must be “*.java”.

2. Upload

Used to store the file into server directory.

3. Method Edit Text

Type a method name for who want to execute in testing. For example, when we want to execute the main method in the code, we will type “main”.

4. Execute Button

Upon pressing this button, the testing tool will execute java service testing (see below for further details) in the server to test the code.

Automatic tests occur for the C0 and C1 instrumented code (see below for further details) and then inserts the tested code into a temporary database for javascript output visual information.

5. Reset

Pressing this button will stop the process and clear all the fields.

6. Result Display

The testing tool will display the tested code and execution time for the testing. The executed statements are highlighted in bright green as shown in Figure 2. The testing tool also displays visual information of the tested code highlighted in bright yellow and the estimation time for the visualization as shown in Figure 3. Figure 4 shows the full configuration of the testing tool.

The testing tool has three parts: uploader code, java service testing, and insertion of temporary database.

The java service testing has four sub-parts: analyzer, C0 and C1 instrument code generator, testing part, and random data generator.

To implement this model, several steps are followed.

1. Uploader code is an input of testing tool. The input is tested code from a user. The code will be used for the java service testing. Uploader code uploads the tested code from the user to a server. Tested code is the java program.
2. Java service testing
   a) Analyzer loads the original tested code and then the original tested code used by the C0 and C1 instrument code generator, testing part, and random data test generator. The testing tool will execute the java service testing to analyze and read the original code based on the information of the specified file, then testing the code and

inserting it into a temporary database for javascript output.

b) The C0 and C1 instrument code generator, C0 and C1 used generated random data for measurement. Applying the testing procedure, the software will perform the measurement for C0 and C1. Measurement of statement and branch coverage uses instrumented code from the original code of a test target program and automatically tests based on C0 and C1 by inputting random data into the C0 and C1 instrumented code. C0 is used as the exit criterion for the software testing.

   The C0 and C1 instrumented code generator generates the C0 and C1 instrumented code by pattern matching from the original code. Here, each standard input instruction is rewritten into instruction that calls the random data generator.

c) Testing part views the covering status of statements and branches by inputting random data during the background process.

The process of the java service testing uses regular expression for obtaining information from code test. The regular expression gets the information of the C0 and C1 instrumented code.

The testing aspect and the random data generator start after generating the C0 and C1 instrumented code. The testing part executes the C0 and C1 instrumented code. The random data generator inputs random data into the C0 and C1 instrumented code on behalf of the users inputting data per standard input instructions.

After each execution of the C0 and C1 instrumented code by the testing part, the testing tool obtains the covering status of statements and measures C0 and C1. The testing tool visualizes the covering status of the statements by highlighting the original code that is displayed and animated as the sequence process executes the tested code. The process for the testing part is executed repeatedly until the user presses the reset button or C1 satisfies 100%.

d) Random data generator generates random test data and automatically tests a program with the generated test data. Users of the testing tool do not need to describe the test data.

A result of java service testing is the percentage of a successful tested code and visual information to know the behavior of the tested code.

3.2 Implementation policies

This section describes each part of the testing tool in detail.

1) Uploader java file:
   The user will upload the java file from the local drive and store file in the server.
2) Execution method:
   The execution method will run a method contained in an original java file from the user. As shown in Figure 1, if we want to execute the main method for testing, we will type “main” as the method to test, and then testing starts. The code for the execution java service testing is displaced in Figure 5. The testing tool will execute java service testing to test the code from the user.
3) Testing part:

The testing tool with java service testing uses the original code based on the file information that users give as an input. The java service testing loads every 1 line from the original code and stores it in an array of type string.

The testing tool loads the original code from the database, while code to load the original code from database is shown in Figure 6, and then visualizes the behavior of the code.

- Java service testing finds the class name of the original code by pattern matching. The class name is used when the C0 and C1 instrumented code generator generates the instrumented code. Each standard input instruction then is rewritten into the instruction that calls the random data generator.
- The testing method process will insert the data execution line by line into the database.Data stored in the database will be line number, number of executions of each line, and tested code.
- The C0 and C1 instrumented code generator generates instrument code. Instrument code is inserted or rewritten at the original code, as shown in Figure 9. The C0 and C1 instrumented code generator applies pattern matching to every 1 line of the original code to insert or rewrite the statement. Then pattern matching finds the keywords that will be used to generate the C0 and C1 instrumented code. C0 and C1 instrumented code is using for calculating the number of executions of C0 and C1.

When the java service testing executes a statement, the service assigns 1 to an element of the array that corresponds to the executed statement. When all elements of the array C0 are assigned 1, the java service testing judges C0 satisfies 100% and also for C1. A test target program is “original code” which users give as an input, not “C0 and C1 instrumented code”.

The testing part highlights every 1 line of the original code displayed on the source code display

```
if(!file_exists('MainClass.class'))
{
       exec('javac MainClass.java 2>&1');
       echo "Eksekusi MainClass.java";
}

/*Doing compilation MainApp.java if file not yet compiled*/

if(!file_exists('MainApp.class'))
{
       exec('javac MainApp.java 2>&1');
       echo "Eksekusi MainApp.java";
}

if(!file_exists('Database.class'))
{
       exec('javac Database.java 2>&1');
       echo "Eksekusi MainApp.java";
}
```

Fig. 5. Part of a code execution method for java service testing

```
$link = mysql_connect('db_server', 'username', 'password');

if (!$link) {die('Could not connect: ' . mysql_error());}
  $db_selected = mysql_select_db('db_jvis', $link);
if (!$db_selected) {die ('Can\'t use db_jvis : ' . mysql_error());}
  $result= mysql_query('select * from content;');
  $result_display = mysql_query('select distinct(line_number), code_text from
content order by line_number;');
  $data=array();
  $i=1;
while($row = mysql_fetch_array($result))
{
  $data[$i]['id_content']=$row['id_content'];
  $data[$i]['line_number']=$row['line_number'];
  $data[$i]['proses_count']=$row['proses_count'];
  $i++;
}
```

Fig. 6. Part of a code of the testing tool to load the original tested code from the database

label based on the covering status of the statement and branch coverage. The executed statements are highlighted in bright green and animated in yellow with Ajax.

Table 1 shows the process used to rewrite or insert the pattern by the C0 instrumented code generator. The C0 and C1 instrumented code generator obtains the covering status of statements by inserting an instruction statement that assigns a value to an array of type *int* for every statement. The initial value of each of its elements is 0.

Furthermore, the testing part measures C0 and C1 based on the covering status of statement and branch coverage. This part repeatedly executes the process until C1 is satisfied 100%.

The java service testing inserts the data execution by each line into the database. The data are line number, number of executions of each line, time execution, and tested code. Visualization data is using Ajax. Data loaded from the database will fetch as an array and then be visualized. The code for visualization is shown in Figure 7.

4) Random data generator:

The random data generator generates random test data on behalf of the *System.in.read method* that calls user input. Here, the type of random data used in the testing tool is integer only.

5) Additional direction:

The file extension must be "*.java", because Java is inherently object-oriented, which means that

Table 1. The processes by C0 and C1 instrumented code generator.

Processing	Pattern	Processing after pattern-match
Insert a package	None	→ package autotest;
Rewrite an original main function	.*class¥¥s+"+[class name]+"¥¥s*¥¥{.*	class [class name] → class MySample{
Rewrite an original class name	.*public¥¥s+static¥¥s+void¥¥s+main¥¥s*¥¥(.*¥¥).*	public static void main() → public void MyMain()
Insert assignment statement for standard output instructions	.*System.out.(print|println).*	System.out.print([output]); → System.out.print([output]);Sample.msg[execution number of times][number of output]=[output];
Rewrite standard input instructions	.*System.in.read.*	System.in.read([variable]); → Sample.read([variable]);
Insert assignment statement for all statements	None	[statement]; → [statement];Sample.statement[line number]=1;

```
function writeResult(hitung)
{
$('#code_'+array_record[hitung]['line_number']).css({'background-color':'yellow'});
$('#count_'+array_record[hitung]['line_number']).val(array_record[hitung]['proses_count'
]);
console.log(hitung);

if(array_record.length<=hitung){
return;
}
hitung++;
setTimeout(function() {
setTimeout(function() {
$('#code_'+array_record[hitung]['line_number']).css({'background-color':'white'});
}, 500);
writeResult(hitung);
}, 750);
 }
var array_record=[];
var counter=0;
$.ajax({
type: "POST",url: "ajax.php",
success: function(msg){
res= json_decode(msg);
array_record  = res;
writeResult(counter);

 }//endsuccess
   }); //endajax
```

Fig. 7. A part of the code for visualization using ajax.

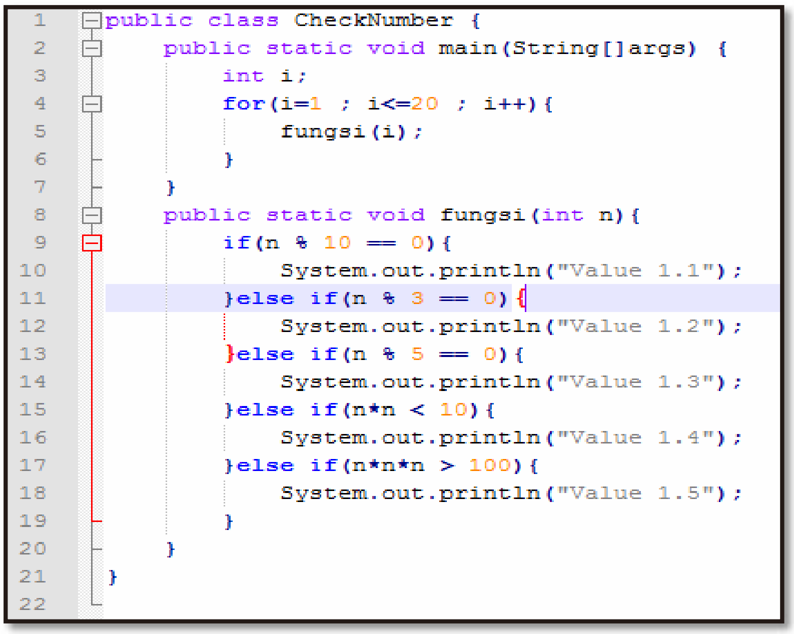
which means that Java programs are consist in programming elements called objects. Simply put, an object is a programming entity that represents either some real-world objects or an abstract concept.

- File names must be same as the Public Class Name, and in this case we use java service testing and java has restriction about this point. This restriction implies that there must be at most one such name type per compilation unit. This restriction makes it easy for a compiler for the java programming language or an implementation ofthe java virtual machine to find a named class within a package.

4. DISCUSSION

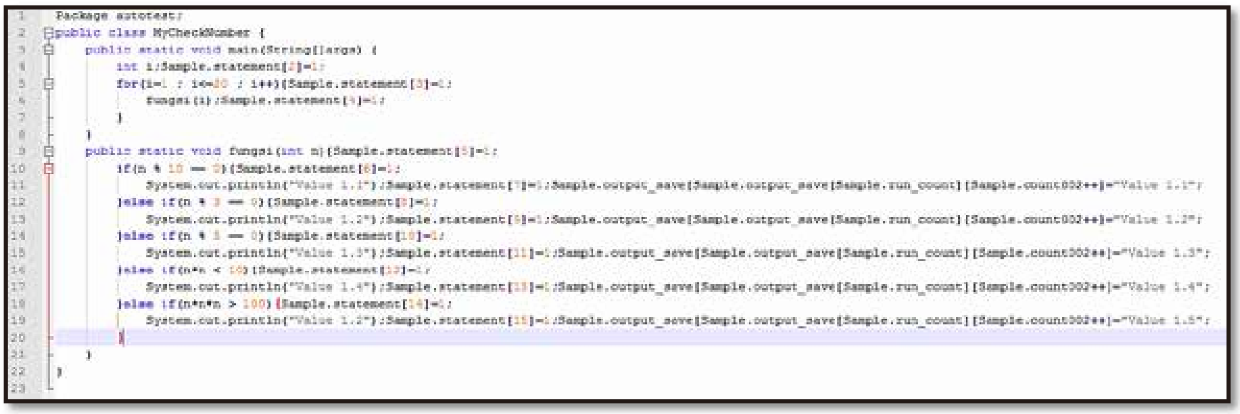
This research seeks to improve the efficiency in the testing of software development, and implement the web-based software testing tool of an automatic unit testing tool using random testing for java programs. This testing tool can automatically test a program based on statement coverage (C0) and branch coverage (C1), without preparing test data by user.

As example of the tested code is Class CheckNumber. The testing tool verifies that the tested code works correctly. Figure 8 shows the tested code, and Figure 9 shows the generated C0 and C1 instrumented code by inputting the check number program into the testing tool.



```
 1  public class CheckNumber {
 2      public static void main(String[]args) {
 3          int i;
 4          for(i=1 ; i<=20 ; i++){
 5              fungsi(i);
 6          }
 7      }
 8      public static void fungsi(int n){
 9          if(n % 10 == 0){
10              System.out.println("Value 1.1");
11          }else if(n % 3 == 0){
12              System.out.println("Value 1.2");
13          }else if(n % 5 == 0){
14              System.out.println("Value 1.3");
15          }else if(n*n < 10){
16              System.out.println("Value 1.4");
17          }else if(n*n*n > 100){
18              System.out.println("Value 1.5");
19          }
20      }
21  }
22
```

Fig. 8. Example of the tested code which is Class CheckNumber



```
 1   Package autotest;
 2  public class MyCheckNumber {
 3      public static void main(String[]args) {
 4          int i;Sample.statement[2]=1;
 5          for(i=1 ; i<=20 ; i++){Sample.statement[3]=1;
 6              fungsi(i);Sample.statement[4]=1;
 7          }
 8      }
 9      public static void fungsi(int n){Sample.statement[5]=1;
10          if(n % 10 == 0){Sample.statement[6]=1;
11              System.out.println("Value 1.1");Sample.statement[7]=1;Sample.output_save[Sample.output_save[Sample.run_count][Sample.count002++]="Value 1.1";
12          }else if(n % 3 == 0){Sample.statement[8]=1;
13              System.out.println("Value 1.2");Sample.statement[9]=1;Sample.output_save[Sample.output_save[Sample.run_count][Sample.count002++]="Value 1.2";
14          }else if(n % 5 == 0){Sample.statement[10]=1;
15              System.out.println("Value 1.3");Sample.statement[11]=1;Sample.output_save[Sample.output_save[Sample.run_count][Sample.count002++]="Value 1.3";
16          }else if(n*n < 10){Sample.statement[12]=1;
17              System.out.println("Value 1.4");Sample.statement[13]=1;Sample.output_save[Sample.output_save[Sample.run_count][Sample.count002++]="Value 1.4";
18          }else if(n*n*n > 100){Sample.statement[14]=1;
19              System.out.println("Value 1.2");Sample.statement[15]=1;Sample.output_save[Sample.output_save[Sample.run_count][Sample.count002++]="Value 1.5";
20          }
21      }
22  }
23
```

Fig. 9. C0 and C1 instrumented code generated by the testing tool

The following describes the process for generating the C0 and C1 instrumented code in Figure 9 based on the test code in Figure 8. The numbers at the left of Figure 8 and Figure 9 are the line numbers. This line number is compatible with the original code and the C0 and C1 instrumented code.

- Insert a package before the first line of Figure 6, to generate the C0 and C1 instrumented code (Line 1 in Figure 9).
- Rewrite an original class name as a class name “MyCheckNumber” specified in advance by the testing part (Line 2 in Figure 9).
- Insert an assignment statement after all statements to gain the covering status of the statements. (Line 2, 3 , 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 in figure 9).
- Insert an assignment statement to store outputs after standard input instruction “*System.out.println*” (Line 9 in Figure 9).

Figures 2 and 3 show a demonstration of the testing tool after testing with a statement and branch coverage. The testing tool will display the result status of the statement and branch coverage. This result is not only displayed as a static result, but also visualizes the result to show the behavior of the tested code.

As shown in Figures 2 and 3, because all statements have been highlighted in bright green and animated by yellow, you can see that all the statements have been executed.

Users can get the status of testing by embedding the statements that can view the covering status of statement. Users can find dead codes if they exist or satisfy C0 and C1. C1 is used as an exit criterion for the testing tool. If C1 of the original code is satisfied 100%, then the test automatically stops.

Coverage based testing can be applied to any stage of testing including unit, integration or system testing. In this testing tool, results of coverage measurment can be used in several ways to improve the verification process.

The code coverage analysis process is generally divided into three tasks: code instrumentation, data gathering, and coverage analysis[19].

Code instrumentation in this research consists of inserting some additional codes to measure coverage results. Instrumentation can be done at the source level in a separate pre-processing phase with pattern matching or at runtime by measure of coverage result. Data gathering consists of storing coverage data collected during test runtime. Coverage data analysis is using statement and branch coverage for providing recommendation into a user about problems in his code.

Random testing was proposed as an extension of object oriented testing for agent testing. In this method, one agent is considered data time and the list of all possible messages that the agent can receive is formulated[2]. Random data tests provide the application under testing with input data generated at random. Typically, testers pay no attention to expected data types. They feed a random sequence of numbers, letters and characters into the numeric data fields[10].

The testing tool with java service testing will generate random testing data used by testing process and is called the user's input. Random data testing is also used for the measurement percentage of success from statement and branch coverage. The type of random data used in the testing tool is integer only.

The research has two result displays. The first is a static display, and the second is dynamic. The testing tool displays the static result of testing as the number for each line execution, measurement percentage of success from statement and branch coverage, and time execution for testing as shown in Figure 2. The result shows the number for how many times the line was executed by the java service testing. This result can be used to check the logical flow of the program from the number of executions of each line. In other words, the result can be used for the verification process.

This research uses the initial value of each statement element as 0. The java service testing executes a statement, and the service assigns 1 to an element of the array corresponding to the executed statement. Percentage of coverage is measured from based on condition.

The second result display is dynamic. The result display visualizes the behavior of the tested code as shown in Figure 3. Certain coverage analysis tools also depict coverage visually, often by highlighting portions of code that are unexecuted by a test suite[20]. In this research, the visual information resets every time a tester selects a new code and then tests the code. The testing tool performs new visualizations to know the behavior of the code, and that it does not accumulate with each successive test run before the testing.

The testing tool can show the correlation between visual information and software testing. This correlation means results collection and a better prespective of software testing. The testing tool shows the correlation as visual information, and it allows a better understanding of the behavior of the tested code.

Visual information describes the behavior of the tested code as a sequence of the line executed by the testing tools. This current research displays not only the result as a number of the percentage of success process, but the testing tool also displays visual information about it. Visual information helps one understand the behavior of the tested code. The testing tool displays visual information in highlighted bright yellow and also the estimation time for the visualization. Visual information describes the behavior of the tested code as a sequence of the line being executed.

One of the goals of this research is having visualizations that are designed to motivate developers to write and understand more effective code by visualizing test adequacy. Code coverage visualizations are supposed to improve developer efficiency and knowledge and promote more productive testing strategies. Testing visualizations guide developers to a particular standard of effectiveness, so if developers want to test software adequately, we must ensure that the coverage criteria we choose to visualize leads developers toward a better standard of test effectiveness.

The main role of visual information is that it can understood as helping the user to perceive patterns that can be used for building an appropriate model. This goal means, in particular, that a tool should

Table 2. Time comparison between testing tool testing and manual testing(by human).

Testing Tool	No.	Manual Testing
752 ms	1	3’ 54 ″
	2	2’ 54 ″
	3	5’ 40 ″
	Mean	4’ 15 ″

facilitate the perception of (sub) sets of data items as units[21)].

The testing tool uses java file CheckNumber that inputs 19 lines and then to measure statement coverage, branch coverage, number of run, the input of each program to the end of testing by C0 and C1 to reach 100%. We measured the testing at the web server with CentOS release 5.9 (Final), Apache/2.2.3, Intel(R) Xeon(R) CPU 3050 @2.13GHz, PHP Version 5.3.3.

In addition, we performed experiments with CheckNumber until the testing tool stopped testing when C1 satisfied 100%, with a length of time until examinees manually selected a data test to satisfy C0 and C1 100%.

The times execution for testing the class CheckNumber is 752 ms and if we test manually (by humans), theaverage time is 4 minutes 15 second as shown in Table 2. The testing tool can reduce time to describe a tested code and execute unit testing in a shorter time.

The condition that possibly may result and might be not satisfied with 100% is the statement and branch coverage that find dead codes if they exist. The testing tool must provide a way to flag those dead codes.

By applying various programs to the testing tool, we have found several problems with the testing tool. These issues are as follows:

- The type of data test generation is an integer. Because the java service testing is not supported by types except type *int*, thedefects that the testing tool can detect are limited.
- In generating the C0 instrumented code, the java service testing inserts a statement every 1 line to get the covering status of the statement. In a program that does not include a suitable newline, the testing tool cannot properly get C0 to highlight a covering statement.
- Users of the testing tool can save time when generating test data, but the users need to check the results and input thedata after testing.
- The target of automatic generating and inputting random data is only the *System.in.read method*. In generating the C0 and C1 instrumentation code, the java service testing rewrites only *System.in.read method* into an instruction that calls the random data generator. Therefore, inputting of value without using the *System.in.read method* is outside the scope of automatic inputting and generation of test data in the current testing tool.

5. RELATED WORKS

Several source code based testability metrics have been proposed for object-oriented applications. R. Binder classifies source code-based metrics according to two criteria: Complexity of testing indicates how difficult it is to produce a test; and the scope of testing evaluates how many test cases have to be produced[22)]. Software testing methods are the techniques, procedures, patterns or templates used to conduct software testing tasks both effectively and efficiently[23)].

This research seeks to improve the efficiency in testing of software development, and implement the testing tool of an automatic unit testing tool via random testing for java programs. The testing tool can then automatically test a program based on statement and branch coverage.

Automation has become essential given t system high complexity, need of performance and stress testing, optimum testing times and cost, reduction of software quality and, after the recognition of the importance of software tests, increased pressure on software development teams[16)]. The testing tool uses java service testing to automatically test a program by inputting random data into the C0 and C1 instrumented code.

Program visualization can be described as depicting the source code or the state of a program or its execution with a visual means[24)].

The research has two result displays. The first is a static display, and the second is a dynamic one. The testing tool displays the static result of testing as the number of each line execution and a measurement percentage of success from statement and branch coverage.

The second result display is dynamic. The result is displayed for visualizing the behavior of the tested code. The testing tool displays the visualization with visual information as a highlighted bright yellow and also the estimation time for the visualization. Visualized information describes the behavior of the tested code as a sequence of the line being executed by the code.

Visualization concerns the graphical representation of information to assist human comprehension of and reasoning about that information[25]. The testing tool result makes possible distribution of the software testing scalability problem, making certain key choices instead a technical distribution of responsibilities.

6. CONCLUSION

We aim to improve the efficiency of testing in software development, and have implemented a web-based software testing tool with java service testing of an automatic unit testing tool for java programs with random testing.

The implemented testing tool generates the C0 and C1 instrumented code from the original code. The testing tool uses java service testing to automatically test a program by inputting random data into the C0 and C1 instrumented code.

In this testing tool, users can automatically test a program without preparing test data because the testing tool generates random test data for testing. Users can get the status of testing by inserting the statements that show the covering status of the statement data testing stored in database. After testing, the obtained result is outputted as a static html page and dynamic display for visual information with Ajax.

The testing tool can show the correlation between visual information and software testing. This correlation means a result collection and prespective of software testing. The testing tool shows the correlation as visual information to understand the behavior of the tested code.

Visual information means the behavior of the tested code is a sequence of the line executed by the testing tools. This research displays not only the result as a percentage number for the success process, but testing tool display also shows a visual information of it.

The testing tool can reduce the time needed to describe a tested code and execute unit testing in a shorter time. The time execution needed to test CheckNumber was 752 ms.

Future issues are as follows:

- Expanding the type of a data test. In order to detect more defects, we need to improve the testing tool, so that the testing tool can input data test other than type *int*.
- Checking the expectation value and execution result automatically
- Discussing a new input form, so that you can input the expectation value.
- Introducing the parser. It is possible to adapt the testing tool to a program that does not include a suitable newline by introduction of the parser.

REFERENCES

1). Roger S.Pressman, "Software Engineering – A Practitioner's Approach", Tata Mc Graw Hill, 7th edition, 2010.

2). Sivakumar.N.,Vivekanandan.K. "Comparing the Testing Approaches of Traditional, Object-Oriented and Agent- Oriented Software System", International Journal of Computer Science & Engineering Technology (IJCSET), Vol.3, No.10, 498 – 504, 2012.

3). Qian.Zhongsheng., Miao.Huaikou., Zeng.Hongwei. "A Practical Web Testing Model for Web Application Testing", 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System, 434 – 441, 2008.

4). Dogana.Serdar., Betin-Cana.Aysu., Garousia.Vahid. "Web application testing: A systematic literature review", The Journal of Systems and Software (ELSEVIER), 174–201, 2014.

5). Ball. T., Eick. S.G. "Software Visualization in the Large", IEEE Computer Society, 33 – 43, 1996.

6). Wang.Huansong., Zhang.Xiang., Zhou.Mingqi . "MaVis: Feature-based Defects Visualization in Software Testing", Engineering and Technology (S-CET), 2012 Spring Congress on, 1 – 4, 2012.

7). Cédric Beust, "TestNG", http://testng.org/doc/index.html

8). Mike Clark, "JUnit Primer", http://www.clarkware.com/software/JUnitPerf.html

9). Matsuoka, Shingo., Kita, Yoshihiro., Katayama, Tetsuro. "Prototype of an Automatic Unit Testing Tool with Random Testing for Java Programs", 22nd International Symposium on Software Reliability Engineering (ISSRE), 2011.

10). Agarwal,B. B., Tayal, S. P., Gupta, M. "Software Engineering & Testing An Introduction", Jones and Bartlett Publishers, 161-179, 2010.

11). Manfred Rätzmann Clinton De Young, Software Testing and Internationalization, Galileo Press GmbH, 49 – 51, 2002.

12). Testing Brain, "Statement Coverage in Software Testing", http://www.testingbrain.com/whitebox/statement-coverage.html.

13). Software Testing Genius, "Know the Basic White Box Testing Techniques based upon Code Coverage", http://www.softwaretestinggenius.com/know-the-basic-white-box-testing-techniques-based-upon-code-coverage.

14). ISTBQ. “How to Calculate Statement, Branch/Decision and Path Coverage for ISTQB Exam Purpose”. http://www.ajoysingha.info

15). Software Testing Guide, “Explain Branch Coverage/Decission Coverage”,http://softwaretestingguide.blogspot.jp/2009/12/explain-branch-coverage-decision.html

16). Dasso.Aristides., Funes.Ana., “Verification, Validation and Testing in Software Engineering”, Idea Group Publishing, 71 – 95, 2007.

17). Fekete.Jean-Daniel., van Wijk.Jarke J., Stasko .John T., North.Chris. “Information Visualization Human-Centered Issues and Perspectives”, Springer, 1 – 18. 2008.

18). Yuan-Fang Li., Paramjit K. Das., David L. Dowe. “Two Decades Of Web Application Testing A Survey Of Recent Advances”, Information Systems (ELSEVIER), 20–54, 2014.

19). Shahid.Muhammad., Ibrahim.Suhaimi., “An Evaluation of Test Coverage Tools in Software Testing”, International Conference on Telecommunication Technology and Applications, Vol.5, 2011.

20). Lawrance, Joseph., Clarke, Steven., Burnett, Margaret., Rothermel, Gregg. “How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study”, IEEE Computer Society, 53-60, 2005.

21). Purchase. Helen C., Andrienko. Natalia, Jankun-Kelly.T.J., Ward.Matthew. “Theoretical Foundations of Information Visualization”, Information Visualization, 46 – 64, Springer Berlin Heidelberg, 2008.

22). R. V. Binder. “Design for Testability in Object-oriented Systems”. Communications of the ACM, Vol.37, No.9, 87–101, 1994.

23). Lee, J., Kang,S., Lee,D. “Survey on Software Testing Practices”, IET Software Publishes Papers on All Aspects of the Software Lifecycle, 275-282, 2012.

24). Myller, Niko. “Collaborative Software Visualization for Learning: Theory and Applications”, Dissertation Document, University Of Joensuu, 2009.

25). Petre, Marian.,Quincey, Ed de. “A gentle Overview of Software Visualisation”, Computer Society of India Communications, 1- 10, 2006.