International Journal of Smart Home
Vol.7, No.5 (2013), pp.125-134
http://dx.doi.org/10.14257/ijsh.2013.7.5.13

Effort Reduction of Unit Testing by Supporting CFG Generation and its Test Design

Hoijin Yoon

Department of Computer Science and Engineering, Hyupsung University, Gyeonggi, Korea
hjyoon@uhs.ac.kr

Abstract

The popularity of Agile Development has been increasing over the last several years. Many Agile techniques include unit testing as a basic activity. Unit testing has two characteristics: it is performed by programmers, and it is normally based on source code structures. However, the problem arises that programmers are unlikely to design structure-based tests. Test design process draws graphs that represent structures; and applies coverage criteria these graphs. The coverage criteria concept is very important in unit testing. Support tools are required, especially by programmers performing unit testing. However, the existing tools do not cover all the steps of the process. This study develops a method that fills the gap between the graph generation and the test design step. This paper We analyzes the extent to which the method decreases the effort required for unit testing, and show that the method effectively reduces the effort. The effort is measured in ManMonths.

Keywords: *Unit Testing, Agile Development, Coverage Criteria, Control Flow Graph*

1. Introduction

The IT world is changing rapidly, and software should be adapted to match these changes. Because of this situation, the viewpoint of software development has changed over the last several years. One of the major movements in software development is the adoption of Agile Development, even by big enterprises, although originally the Agile approach was adopted only for the development of small-scale software. However, the requirements of software now tend to change drastically, and software should be sufficiently agile to meet them [1]. Forrest Research reported that a quarter of all enterprises have already adopted Agile Development, and the adoption of enterprise-level Agile Development has accelerated, increasing approximately two and a half times faster between 2006 and 2007 than between 2005 and 2006 [2]. In Korea, Kyobo Book Store opened its mobile Web system, which was developed using the Agile Development approach [3].

In 2009, *VersionOne* conducted a survey on the subject “Which techniques do Agile developers apply to projects?” The target group of the survey consisted of 2570 developers working in 88 different countries [4]. Figure 1 is the result of survey. The techniques included in this survey were those suggested by Scrum and Extreme Programming(XP)[5]. The programming-related techniques by ranking were (1) unit testing, (2) continuous integration, (3) automated builds, and (4) test-driven development. According to the results, unit testing is the technique most frequently used by Agile developers. However, satisfactory unit testing is not easy to achieve due to the following two difficulties. First, unit testing should be done by programmers, who are not experts in testing. Second, the tests of unit codes are usually not

ISSN: 1975-4094 IJSH

managed and stored as artifacts. In this paper, we propose a tool to help programmers design unit tests using only source codes.

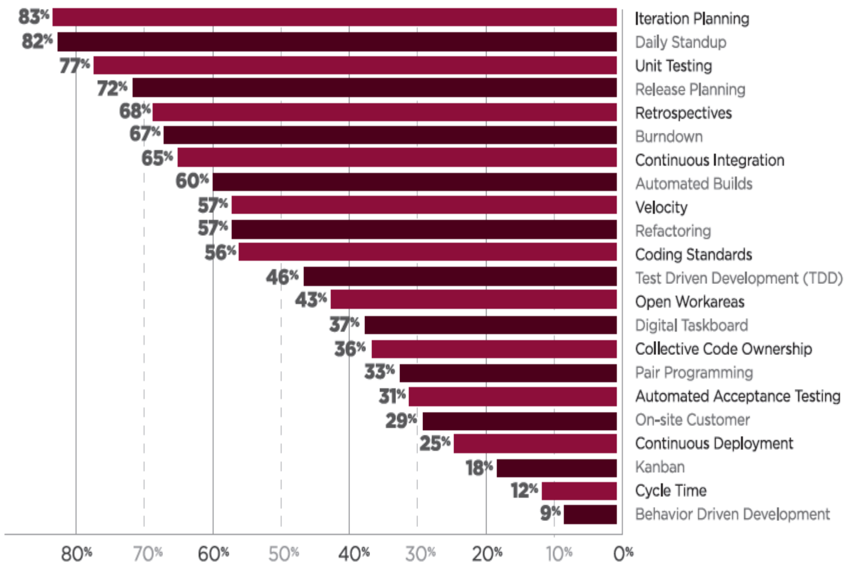


Figure 1. "Which Techniques do Agile Developers Apply to Projects?" [4]

It is probable that programmers will test a unit based on the source code that they have written. They can utilize existing tools; however, no single tool that covers the entire unit testing process exists. Therefore, the programmer needs to combine multiple tools to complete unit testing. To achieve this, the programmer also needs to understand what kind of information is required so that the corresponding tools can be combined. To aggravate the problem, some programmers are required to test codes written by other programmers who are no longer available, which means that they have no knowledge of the source code, and it is hard for them to understand the source code structure. Some good tools exist that support these tasks, but there is a gap between them[6]. In this paper, we describe a method that fills the gap between two separate tools, covering *Control Flow Graph (CFG)* generation and test path selection. Section 2 explains the effort analysis of unit testing conducted based on source codes. It is mentioned again in Section 4 in the context of the evaluation of the degree to which the required effort can be reduced by using the method proposed in this paper. Section 3 explains how the proposed method supports the activities of unit testing and describe our proposed tool in Section 3. The analysis is described in Section 4 and then the conclusions are mentioned in Section 5.

2. Unit Testing Practices

In 2006, a survey on the subject of unit testing practices was conducted in which personnel from over 50 member companies of the Software Process Improvement Network (SPIN) [7] were interviewed. The companies ranged from consultancy firms with one employee to regional branches of multinational companies employing hundreds of developers. The network represents various application domains with a focus on embedded systems.

The survey was conducted at two meetings. During the first meeting, a focus group discussion on the subject of unit testing was held. The participants consisted of 17 representatives from 12 companies, a moderator (a software quality manager), and a secretary. Eight representatives from seven of these companies participated in the subsequent survey meeting, together with representatives from seven new companies. The participating companies represented the following sectors: automation, banking, case tools, information systems, health care, transportation, and telecommunication. The primary business of most of the consulting companies was telecommunications. The participants, who ranged from testers to quality managers, were interested in testing and software quality issues in general.

They initiated group discussions focused on three themes: [7]

■ What is unit testing?

■ What are the participants' strengths regarding unit testing?

■ What are the participants' problems regarding unit testing?

In our previous study, we focused on the first of these themes, "What is unit testing?" for building a framework for unit testing [8]. We used the following results obtained for this theme: (1) Unit testing is a task that should be allocated to programmers; and (2) unit testing shows that a unit meets its specification. Moreover, we summarized three characteristics, as follows. First, since it is the programmer of a unit who performs unit testing, it is probable that s/he will design a test case based on program structures. Second, unit test cases need to be reused repeatedly. Test automation, which is mentioned in Agile Development, can support this characteristic of unit testing. Third, test codes, such as *JUnit*, need to be built to meet the daily build required by the recent development environments, such as XP. In our previous study [5], we organized a structure that includes the three characteristics mentioned above. The structure consists of three activities, as seen in Table 1.

Table 1. Characteristics of Unit Testing Practice [8]

Activity	In Unit Test Practices	In JTopas Unit Testing
1	Building unit test codes	Building JUnit test codes
2	Identifying unit test case based on Test Criteria	Applying CFG-based criteria to *JTopas* units
	Structural Test	
3	Unit Test Automation	Using JUnit Test Driver
		Test execution by shell scripts

Of the activities in Table 1, this paper focuses on Activity 2, which covers the technique that is most needed, since its purpose is to select the more meaningful test data; it is this selection that determines the effectiveness of testing, and leads to effective testing costs. We propose a method to reduce the effort required to accomplish Activity 2. The other activities are more labor-intensive.

3. Test Path Generation from Source Code

We propose a method for reducing the effort required by Activity 2, described in Table 1. Activity 2 consists of two separate steps: identifying a CFG from the source code, and defining the test cases from the CFG. To support these two steps, we combine a tool that draws the CFG from the source code and a second tool that generates test paths from the CFG. Figure 2 shows the structure of the method proposed in this paper.

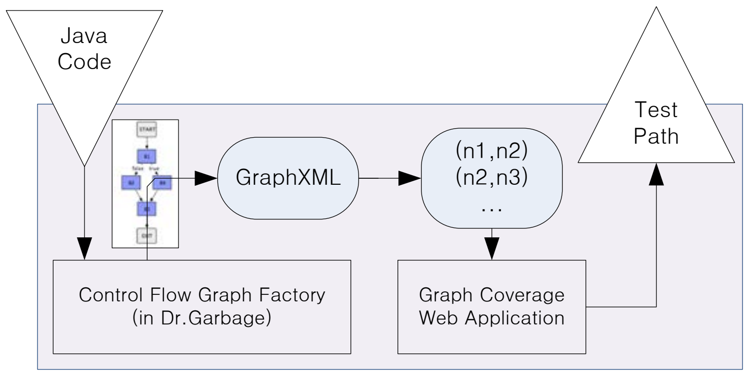

Java Code
GraphXML
(n1,n2)
(n2,n3)
…
Test Path
Control Flow Graph Factory (in Dr.Garbage)
Graph Coverage Web Application

Figure 2. Combination of Tools

3.1. Control Flow Graph Generation

The first part of the tool is for generating CFGs from source codes. We utilize *Dr. Garbage* [9], which was developed as a plugin in Eclipse. *Dr. Garbage* comprises three different functions: *Bytecode Visualizer*, *Sourcecode Visualizer*, and *Control Flow Graph Factory*. *Control Flow Graph Factory* takes a Java source code package as input and then draws the matching CFG as its output. The CFG can be exported in GraphXML or DOT format[10], or as an image file. It takes a Java package as input and draws a graph of each method in the package. We use GraphXML as the output exporting file type. Figure 3 shows a sample method, abs(). In the tool, one graph of each method is drawn. The graph of abs() is shown in Figure 3.

```
package test;
public class C2 {
            public static float abs(float a) {
float b = 0.0F;
                        if(a <= b){
                                    return (b - a);
                        }
                        return a;
            }
}
```

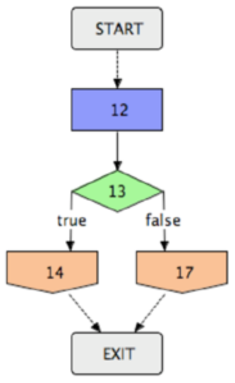


Figure 3. Sample Method and its Graph in Graph Factory [7]

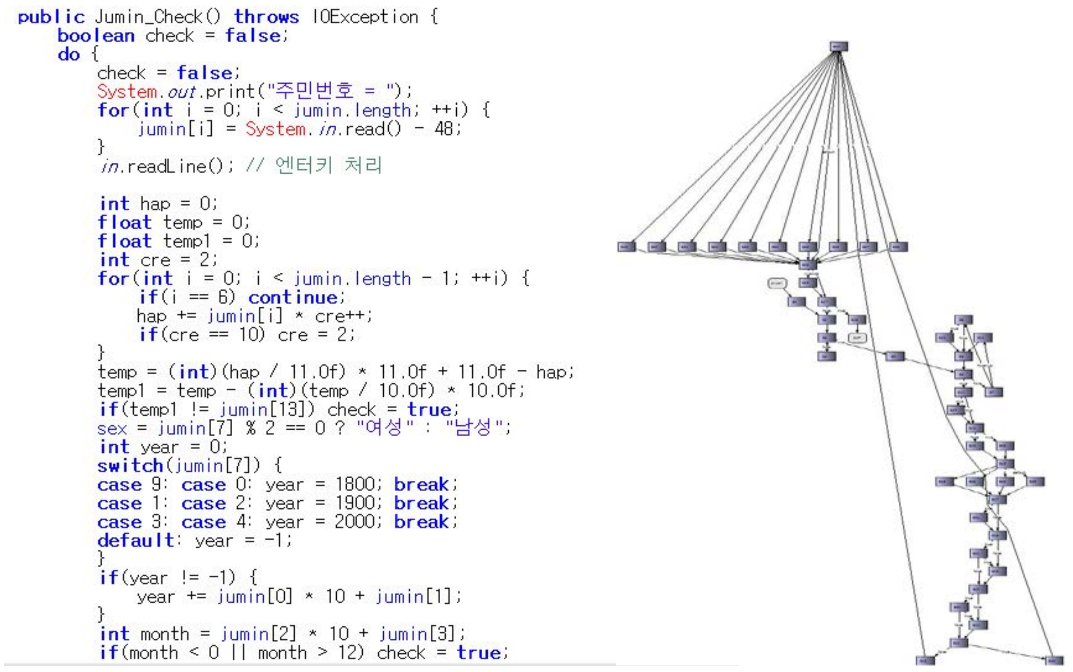
When a graph has been drawn, its matching GraphXML file is exported to our tool. We parse the GraphXML file to obtain a list of edges represented in a pair of related nodes. It is necessary to transform the XML file to a set of pairs of nodes, because Control Flow Graph Factory produces graphs in the form of an XML file and the Graph Coverage Web Application takes a graph in the form of edge pairs. Our tool parses the XML file and extracts a pair of nodes. The Graph Coverage Web Application uses this type of representation as input data. We save the information of nodes in a DBMS and send it to the next tool, where test paths are selected from the graphs. In this study, we used Oracle DBMS.

3.2. Test Path Selection from Graph

When a CFG has been generated from the source code, test paths are determined by coverage criteria. Graph-based coverage criteria include edge coverage, edge-pair coverage, branch coverage, prime-path coverage, and node coverage [11]. A good tool has been provided by Jeff Offutt on his Website [12], which selects a set of test paths in terms of the criterion chosen by the user. The tool uses a CFG as its input data. The graph is represented as a sequence of edges. Each edge is written in the form “n1 n2,” where the edge connects node1 and node2.

3.3. Example

In this section, we present an example of the proposed method that supports Activity 2 in Table 1. First, we assume that the method depicted on the left in Figure 4 is the target of testing. Graph Factory generates its CFG as shown on the right in Figure 4.

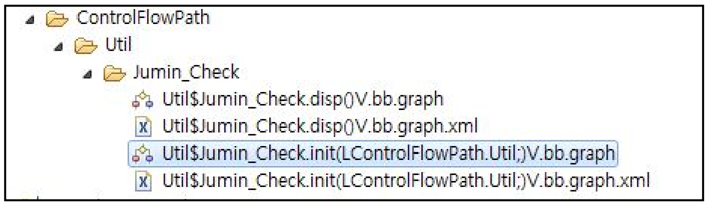


```
public Jumin_Check() throws IOException {
    boolean check = false;
    do {
        check = false;
        System.out.print("주민번호 = ");
        for(int i = 0; i < jumin.length; ++i) {
            jumin[i] = System.in.read() - 48;
        }
        in.readLine(); // 엔터키 처리

        int hap = 0;
        float temp = 0;
        float temp1 = 0;
        int cre = 2;
        for(int i = 0; i < jumin.length - 1; ++i) {
            if(i == 6) continue;
            hap += jumin[i] * cre++;
            if(cre == 10) cre = 2;
        }
        temp = (int)(hap / 11.0f) * 11.0f + 11.0f - hap;
        temp1 = temp - (int)(temp / 10.0f) * 10.0f;
        if(temp1 != jumin[13]) check = true;
        sex = jumin[7] % 2 == 0 ? "여성" : "남성";
        int year = 0;
        switch(jumin[7]) {
        case 9: case 0: year = 1800; break;
        case 1: case 2: year = 1900; break;
        case 3: case 4: year = 2000; break;
        default: year = -1;
        }
        if(year != -1) {
            year += jumin[0] * 10 + jumin[1];
        }
        int month = jumin[2] * 10 + jumin[3];
        if(month < 0 || month > 12) check = true;
```

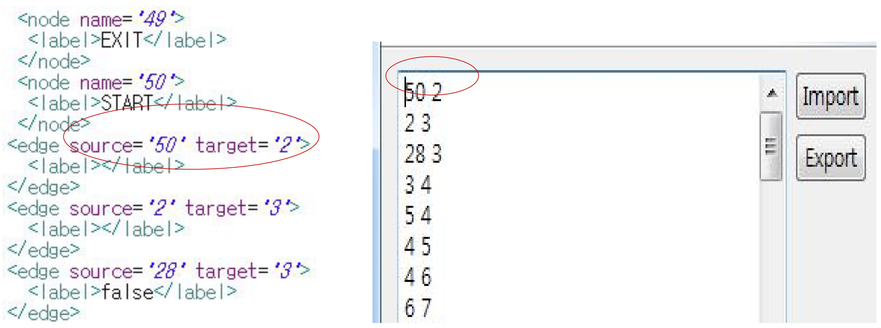
Figure 4. Method and its Graph Generated by Graph Factory

Graph Factory produces several files, as well as visualized graph images. The available files are in XML format as shown in Figure 5. We parse the XML files and save the pairs of nodes in the Oracle DB for the next step. The file content and its DB table are shown in Figure 6.



```
ControlFlowPath
    Util
        Jumin_Check
            Util$Jumin_Check.disp()V.bb.graph
            Util$Jumin_Check.disp()V.bb.graph.xml
            Util$Jumin_Check.init(LControlFlowPath.Util;)V.bb.graph
            Util$Jumin_Check.init(LControlFlowPath.Util;)V.bb.graph.xml
```

Figure 5. The Files Available from Graph Factory

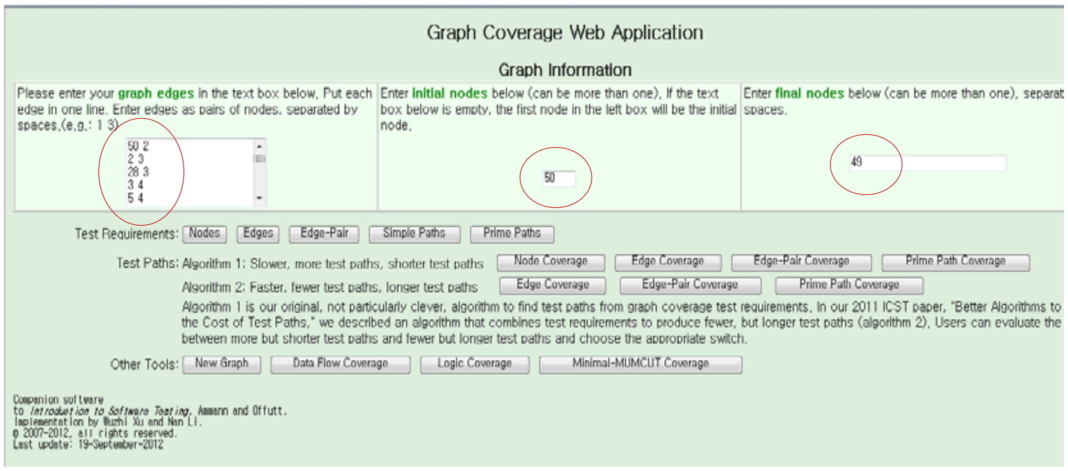


```
<node name='49'>
  <label>EXIT</label>
</node>
<node name='50'>
  <label>START</label>
</node>
<edge source='50' target='2'>
  <label></label>
</edge>
<edge source='2' target='3'>
  <label></label>
</edge>
<edge source='28' target='3'>
  <label>false</label>
</edge>
```

```
50 2
2 3
28 3
3 4
5 4
4 5
4 6
6 7
```

Figure 6. Information from XML File to DB table

Next, the graph is saved in the DB as a sequence of pairs of nodes, as shown in Figure 6. The first step, generating a CFG, is accomplished; the next step is to select test paths by applying test coverage criteria to the CFG. We utilize the tool by inserting the data saved in the DB into the input fields of the tool, as shown in Figure 7.



Graph Coverage Web Application

Graph Information

Please enter your **graph edges** in the text box below, Put each edge in one line. Enter edges as pairs of nodes, separated by spaces.(e.g.: 1 3)

```
50 2
2 3
28 3
3 4
5 4
```

Enter **initial nodes** below (can be more than one), If the text box below is empty, the first node in the left box will be the initial node.

50

Enter **final nodes** below (can be more than one), separat spaces.

49

Test Paths: Algorithm 1: Slower, more test paths, shorter test paths

Algorithm 2: Faster, fewer test paths, longer test paths

Algorithm 1 is our original, not particularly clever, algorithm to find test paths from graph coverage test requirements. In our 2011 ICST paper, "Better Algorithms to the Cost of Test Paths," we described an algorithm that combines test requirements to produce fewer, but longer test paths (algorithm 2). Users can evaluate the between more but shorter test paths and fewer but longer test paths and choose the appropriate switch.

Companion software
to *Introduction to Software Testing*, Ammann and Offutt.
Implementation by Wuzhi Xu and Nan Li.
© 2007-2012, all rights reserved.
Last update: 19-September-2012

Figure 7. Setting the Graph Information in Coverage Tool

The test paths of the graph are selected by clicking one of the "coverage" buttons according to the programmer's chosen test coverage criterion, as shown in Figure 7. The left button depicts a weaker criterion than the right one according to the *subsumption* relation of criteria [13]. For example, Prime-path coverage will select the more complex and stronger set of test paths.

4. Analysis

We conducted an experiment to measure the amount of effort consumed by unit testing of an open-source application, *JTopas*. Obviously, the original programmer of the application was not available, which set the scenario as one where programmers are expected to test a unit code written by others. JTopas was provided by the Software-artifact Infrastructure Repository (SIR) of Nebraska University[14]. JTopas consists of 4 packages that include 10 separate classes and their 158 methods. In the experiment all the activities were performed, and the effort expended to accomplish each activity was measured [5].

The *ManMonths* (mm) in Table 2 was measured based on one person working 8 hours per day and 20 days per month. It should be noted that the testing was conducted by one female participant, and her participation rate was 40%. It took her approximately 7 months under the condition of working 16 hours a week to finish the activity. Activity 2 took approximately two months under the same condition. Activity 3 took only four days, as it was simple to build a shell script [8]. The participant was a Computer Science major and a Certified Software Test Specialist (CSTS) [15]. Although she had her own testing skills, it took her a long time to finish the unit testing, because JTopas was not written by her. She spent a considerable amount of time understanding the behavior of the unit codes, and building a CFG to select test cases. Needless to say, she used a supporting tool, which selects test paths that consider the CFGs inserted as input data. However, she still needed to draw a CFG and convert it to the format required by the tool.

Table 2. Effort Evaluation

Activity	Unit testing	Tools	Using separate tools	Using a combination of tools
1	Building JUnit test codes	JUnit	2.8 mm	2.8 mm
2	Applying CFG-based criteria to JTopas units	Graph Factory Graph Coverage	0.8 mm	0.6 mm
3	Using JUnit Test Driver Test Execution by shell scripts	JUnit	0.2 mm	0.2 mm

Our previous analysis reported in [8] showed that Activity 2 took the second-longest time of all the activities. However, it is the only part of the unit testing process that can take advantage of tools. Since Activity 2 has been studied for a long time, and therefore various theoretical approaches have been proposed, there are many tools that support it, of which Dr. Garbage and Graph Coverage Web Application are examples. In the first experiment [8], we used these tools to save data and transform data types so that they were compatible. The third column in Table 2 shows the effort measured in the first experiment when separate tools were used: 0.8 mm. However, in the second experiment in which the method proposed in this paper was applied to the same JTopas unit testing environment, the effort expended on Activity 2 was 0.6 mm. The combination of the tools shown in Table 2 provided a method for saving the data of the CFGs and transforming them to the form required by the next tool, Graph Coverage Web Application.

5. Conclusions

We proposed a method for generating test paths directly from source code. The method fills the gap between the already existing tools, allowing programmers who use it to avoid passing information manually. The contributions of this paper are defined according to the following two viewpoints.

First, Activity 2 in Table 1 is the core action that determines testing quality. High-quality test cases are necessary to guarantee a certain level of testing, and these are selected according to the coverage criteria. Good test cases lead to high-level testing effectiveness. Our method supports programmers performing this important task. It is especially helpful given that programmers, who are not familiar with coverage criteria, usually perform the unit testing. Finally, the effort required to accomplish this important process is reduced, as shown in Table 2.

Second, the method can also be usefully applied in Agile Development. The Agile process expects that test codes will be attached to all program units since in practice it prefers test automation and daily build. However, programmers are not willing to store unit test cases with their source codes. Furthermore, sometimes the original programmers of a source code have left a company without leaving documentation of the method design or program specification. The situation where a programmer has to define new test cases only from source code occurs when the programmer starts to use Agile Development to deal with an already existing system, especially when performing maintenance tasks. In this case, this paper contributes a tool that reduces the amount of effort that a programmer has to expend on selecting test cases from source code.

There is a threat to the validity of our experimental evaluation: the measured values will differ according to the participant or the target application. If the participant is not qualified as a CSTS or if the target is not well-modularized, the effort measured will be greater. To eliminate this threat, an experiment using diverse participants and applications is required. However, we expect that under any conditions the effort measurement pattern that shows that our method reduces the required effort will be maintained, because our method supports programmers, thus facilitating their performance of unit testing tasks.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0009916).

References

[1] S. M. Ghosh, H. R. Sharma and V. Mohabay, “A Study of Software Change Management Problem”, International Journal of Database Theory and Application, http://www.sersc.org/journals/IJDTA/vol4_no3/4.pdf, vol. 4, no. 3, **(2011)**.

[2] Forrest Research. Enterprise Agile Adoption in 2007, http://www.forrester.com/Enterprise+Agile+Adoption+In+2007/fulltext/-/E-RES45015, **(2008)**.

[3] EtNews. Mobile Kyobo Book Project **(2012)** http://www.etnews.com/news/computing/ informatization/2553967_1475.html.

[4] VersionOne, 3rd Annual Survey: State of Agile Development Survey, http://pm.versionone.com/whitepaper_AgileSurvey2008.html, **(2009)**.

[5] K. Nageswara Rao, G. Kavita Naidu and P. Chakka, “A Study of the Agile Software Development Methods”, Applicability and Implications in Industry, International Journal of Software Engineering and Its Applications”, http://www.sersc.org/journals/IJSEIA/ vol5_no2_2011/4.pdf, vol. 5, no. 2, **(2011)**.

[6] K. Zuhairi Zamli, N. Ashidi Mat Isa, M. Fadel Jamil Klaib and S. Norbaya Azizan, “A Tool for Automated Test Data Generation and Execution Based on Combinatorial Approach”, International Journal of Software Engineering and Its Applications, http://www.sersc.org/journals/IJSEIA/vol1_no1_2007/IJSEIA-2007-01-01-02.pdf, vol. 1, no. 1, **(2007)**.

[7] P. Runeson, “A Survey of Unit Testing Practices”, IEEE Software, DOI: 10.1109/MS.2006.91, vol. 23, no. 4, **(2006)**, pp. 22-29.

[8] H. Yoon, “Effort Analysis of Unit testing conducted by Non-Developer of Source Code”, Korea Society of IT Service Journal, DOI: 10.9716/KITS.2012.11.4.251, vol. 11, no. 4, **(2012)**.

[9] Dr. Garbage, “Control Flow Graph Factory”, http://www.drgarbage.com/control-flow-graph-factory.html, **(2008)**.
[10] S. Shahriar and J. Liu, “Updating in XML Using Semantic Constraints”, International Journal of Database Theory and Application, http://www.sersc.org/journals/ IJDTA/vol4_no2/1.pdf, vol. 4, no. 2, **(2011)** June.
[11] P. Ammann and J. Offutt, “Introduction to Software Testing”, Cambridge Univ. Press. UK, DOI: 10.1017/CBO9780511809163, **(2008)**.
[12] J. Offutt, “Graph Coverage Web Application”, http://cs.gmu.edu:8080/offutt/coverage/ GraphCoverage, **(2010)**.
[13] S. Rapps and E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information”, IEEE Transactions on Software Engineering, DOI: 10.1109/TSE.1985.232226, **(1985)** April.
[14] M. B. Dwyer, S. Elbaum, J. Hatcliff, G. Rothermel, H. Do and A. Kinneer, “Software-artifact Infrastructure Repository”, http://sir.unl.edu/, **(2008)**.
[15] TTA. Certified Software Test Specialist. http://www.tta.or.kr/steps/itEduQnaGuide.jsp.

Author

Hoijin Yoon received the B.S. and M.S. degrees in Computer Science and Engineering from Ewha Womans University in 1993 and 1998, respectively. She also received her Ph.D in 2004 from the same university, Ewha. Her dissertation was on the subject of software component testing. After graduation, she worked at Georgia Institute of Technology as a visiting scholar and then at Ewha Womans University as a full-time lecturer. She has been teaching at Hyupsung University since 2007. She is interested in software testing, service oriented architecture, and testing in Agile Development.