Implementing DDD for Automatic Test Case Generation

Wacharapong Nachiengmai, Sakgasit Ramingwong, and Amphol Kongkeaw

***Abstract*—Software testing is a necessary process to ensure quality of software. Unfortunately, it is usually perceived as a very difficult process for inexperienced software developers. Defect-driven Development (DDD) is a novel development concept which aims to bridge such gap. DDD helps inexperienced developers to automatically generate essential test cases and scripts from defect information collected from a knowledge base. This research describes an implementation of the concept as well as its performance evaluation. The result suggests that this technique helps beginners to create an equivalent effectiveness level of unit test compared to experts in both term of time used and defect density.**

***Index Terms*—Software testing, defect-driven development, automatic test case generation.**

I. Introduction

Testing is undeniably an essential process in software development [1]. In order to perform this process effectively, a software developer needs to possess certain skills and techniques due to the fact that there are a number of aspects of testing, such as unit testing, functional testing, integration testing and usability testing. This can be challenging for inexperienced developers. Lack of experience may lead to several undesirable outcomes such as inappropriate testing model and inadequate testing coverages which inevitably lead to defects and performance problems. Without close supervision and guidance from experts, these inexperienced developers may slowly improve their skills and performance. On the other hand, an efficient mentoring system can greatly accelerate this learning process.

The foundation of Defect-driven Development (DDD) is the collection and knowledge management of data on software defect. This includes all potential defects from every step of software development, regardless of process models. These data are stored in a database or semantics data model as Knowledge Software Defect (KSD). Two main classes of defects are defined in KSD. Firstly, the high-level software defects include problems on design and abstraction processes. Secondly, the low-level software defects involves coding and testing related issues. This classification helps the practitioners to put appropriate focus on relevant stages. KSD is then used by inexperienced software developers as a guideline on identification of potential defect patterns which may surface during their implementation. The list of defect patterns can be varied based on different development environment, project characteristics, technical skills, tools, etc. These information are then used to generator test cases and scripts which are subsequently used by inexperienced software developer. Fig. 1 summarizes the concept and process of DDD.

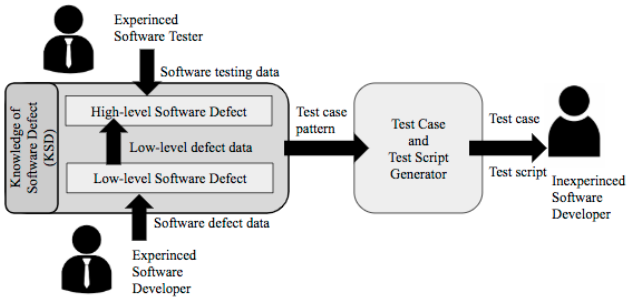

Fig. 1. Main concepts of defect-driven development (DDD).

II. Related Works

A. Defect-Driven Development

Defect-driven Development (DDD) is a development concept which aims to improve the overall quality of the software [2]. By providing defects information, software developers are likely to become more aware and commit less mistakes. The information on defects, such as name, types, likelihood, where it usually surfaces and how to avoid or fix it. These data were previously collected from experienced programmers and systematically stored in the Knowledge of Software Defects (KSD). As aforementioned, defect information is classified into Low-level and High-level, in order to help practitioners focus on their relevant development stages. Previous studies report that implementing DDD significantly help improving the efficiency of defect detection for software engineering students [2]. In fact, not only inexperienced developers can gain benefits from DDD, skilled programmers can also use DDD to help improving their awareness as well as preparation towards potential defects.

B. Software Testing

The main objective of software testing is to investigate the quality of software from various aspects [3]. Not only functional requirements but also non-functional issues are thoroughly processed during this stage. In traditional software development, a suite of test cases is generally constructed after the coding is completed. However, in modern software engineering, the test suite are usually created in the earlier phase, sometimes as soon as after the requirement engineering [4]. Automated tools are also

Manuscript received October 9, 2019; revised December 20, 2019.

Wacharapong Nachiengmai is with the Department of Business Computer, Faculty of Business Administration, North – Chiangmai University, Chiang Mai, Thailand (e-mail: wacharapong@northcm.ac.th).

Sakgasit Ramingwong is with the Department of Computer Engineering, Faculty of Engineering, Chiang Mai University, Chiang Mai, Thailand (e-mail: sakgasit@eng.cmu.ac.th).

Amphol Kongkeaw is with the Department of Software Engineering, Faculty of Engineering and Technology, North – Chiang Mai University, Chiang Mai, Thailand (e-mail: amphol@northcm.ac.th).

introduced in order to facilitate and improve both the efficiency and effectiveness of the testing [5]. Likewise, the concept of DDD can be implemented to further increase the overall performance of the development.

III. Research Objective and Design

A. Research Objective

This research attempts to utilize KSD for automated generation of test suite. Successfully implementing this could shorten the development process along with increasing the quality of the software. Learners will also be able to preview and adopt testing techniques more efficiently. On the other hand, experts can ensure that defects are managed more thoroughly.

B. Research Design

Four modules with a similar technical difficulty are used as the main requirement in this experiment. These modules are described as follows:

- Registration: A form which collects email address, password, full name, date of birth, height and weight. All fields cannot be left blank. The password must consist of 4–10 alphanumeric characters. The successful operation will return 1 while the failed attempt will return 0.
- Sign In: A form which collects an email and a password and then verify them as can be seen in Fig. 2. Both of them are required. Similar to the first module, the successful operation will return 1 while the failed attempt will return 0.
- Age calculation: A form which uses the entered date of birth and calculates into the user's current age. The calculation then returns the year, month and day value of the user's age or 0 in case of an error.
- Body Mass Index calculation: A form which asks for the user's height and weight and calculate them based on standard BMI formula. The result is then compared with criteria. In case of error in calculation, it will return 0.

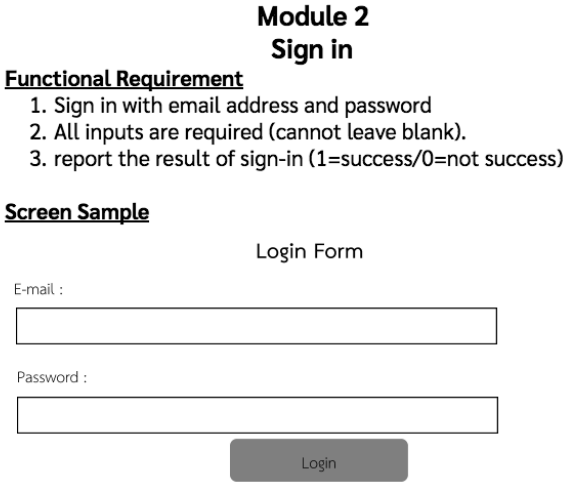

Fig. 2. The example document for sign in module.

The participants are assigned to create a suite of test cases for all modules. The results are used to compare with the test case that is generated with data from KSD.

C. The Use of the Knowledge of Software Defects (KSD)

Defect information in the KSD in this experiment was collected from 5 senior testers who have more than 5 years of experiences in various software companies. They analyzed, identified and recorded data on potential defects that is related with the 4 experiment modules as displays in Fig. 3. Table I displays some examples of the identified knowledge for the High-level Software Defect.

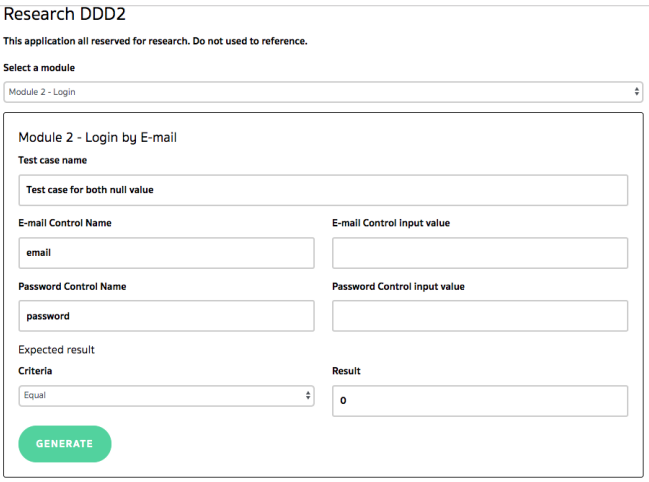

Fig. 3. User interface of data collection.

TABLE I: Examples of Defects Data in KSD

Module	Test name	Parameter Name	Parameter Value	Expected Result
Sign in	Test case for both null value	{"email", "password"}	{"","""}	0
Sign in	Test case for null of email value	{"email", "password"}	{"", "password"}	0
Sign in	Test case for null of password value	{"email", "password"}	{"test@mail.com", ""}	0

D. Participants

Three groups of participants with different experiences in software development joined this experiment. The first group of participants consisted of 5 freshmen in computer engineering program who have just taken a first course of programming. They were identified as a group of inexperienced developers. The second group, the intermediate, included five senior software engineering students who had passed more than five subjects of software development and software testing e.g. Fundamental of Computer Programming, Object-oriented Programming, Software Construction and Evolution, Component-based Software Development, Mobile Programming and Software Validation and Verification, etc. Finally, the last group consisted of 5 invited university lecturers who had extensive experience in software development.

E. Framework

Mocha JS [6] was the selected framework for creating unit tests under Node JS [7] environment. This is due to its simplicity and applicability. Since none of the participants had previously used this tool, according to the principle of

DDD, they were all considered as beginners for Mocha JS in this experiment.

F. Test Case Creation

All groups of participants were instructed to create suites of test cases for the four foregoing modules. Each participant individually created his or her own suites. They were not allowed to either consult with their group members or search for more information regarding to the defects from the Internet. No time limitation was enforced in this process.

G. Test Case Generation

After all participants finished their test case creation, they then implemented their test cases in Mocha JS as a black box testing [8]. Participants in the experienced and intermediate groups composed the test scripts by themselves. On the other hand, the beginner group used an application that is generated the test scripts automatically from the data of KSD as its screen is shown in Fig. 4.

```
const request = require('supertest');
const app = require('../app');
const expect = require('chai').expect;

describe('Login Module', function() {
    it('Test case for both null value', function(done) {
        request(app)
            .post('/login')
            .send({ email: '', password: '' })
            .expect(0)
            .end(done);
    });
    it('Test case for null of email value', function(done) {
        request(app)
            .post('/login')
            .send({ email: '', password: 'password' })
            .expect(0)
            .end(done);
    });
    it('Test case of null password value', function(done) {
        request(app)
            .post('/login')
            .send({ email: 'test@mail.com', password: '' })
            .expect(0)
            .end(done);
    });
});
```

Fig. 4. The example test script code.

Three indexes used in Personal Software Process (PSP) [9] were measured for the performance of the test scripts. These includes Time Used, Code Size, and Defect Density [10], [11].

IV. Result of the Experiment

A. Result of Test Case Creation

TABLE II: Average Numbers of Generated Test Cases

Group of test case	Beginner/ (SD.)	Intermediate/ (SD.)	Experienced/ (SD.)	KSD
Null Checking	11.25 (3.50)	14.5 (4.88)	15.00 (3.80)	12.00
Formatting Checking	1.00 (0.00)	3.33 (0.58)	5.33 (0.58)	3.00
Validity checking	0.00 (0.00)	1.57 (0.07)	4.14 (1.26)	11.00
Other Checking	0.50 (0.04)	2.50 (0.73)	6.0 (2.73)	11.00

According to the test suites developed by the participants, four categories of tests were identified. They consisted of Null Checking, Formatting, Validity and Others. The average number of the generated test cases by the different groups of participants are shown in Table II and Fig. 5.

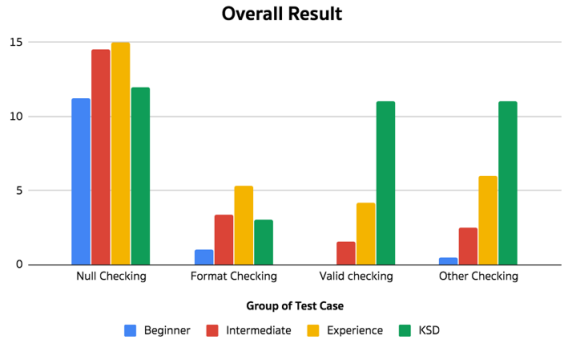

Fig. 5. Test case creation result.

It can be seen that the test cases automatically generated by KSD had a similar amount close to the experienced and intermediate group for the checking of null value and formatting. In contrast, for the tests on validity and miscellaneous defects, KSD generated significantly more cases than other participants. This suggests that KSD could be an efficient tool to assist software practitioners regardless of their levels of experience.

B. Result of Test Script Generation

Table III illustrates the three indexes PSP result of test script generation.

TABLE III: Comparison of Development Indexes

Group of participant	Time used (Minutes)	Code size (LOC)	Defect Density (per KLOC)
Experienced	11.45	44.95	14.86
Intermediate	21.45	28.45	102.37
Beginner	1.00	57.00	0.00

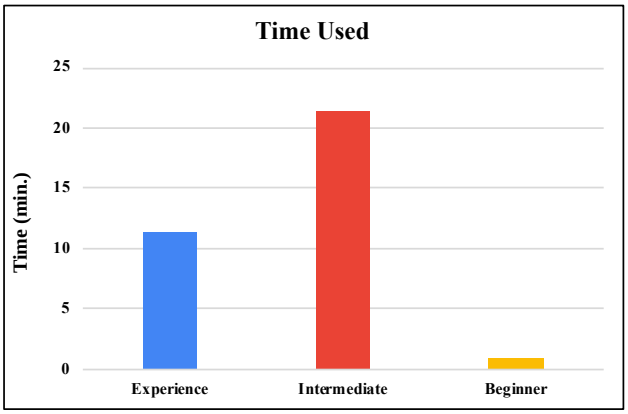

Fig. 6. Test script performance in time used.

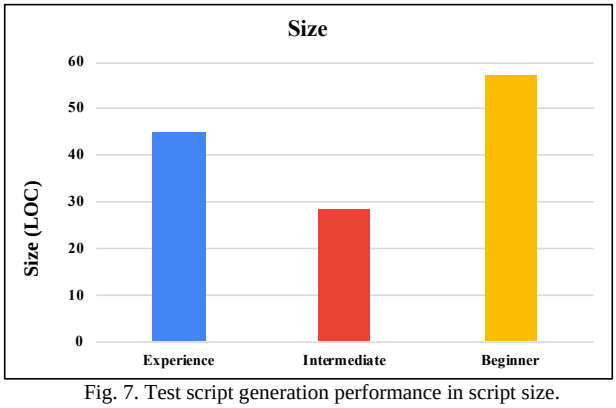

Fig. 7. Test script generation performance in script size.

According to Fig. 6, it can be seen that the beginners spent significantly less average time to create a test case when compared to both the experienced and intermediate participants. This is highly likely to be the benefit from using KSD since they can see and subsequently choose the appropriate defects easily. Furthermore, Fig. 7 reveals that the size of the test case are not much different between the three groups of participants. The defect density shown in Fig. 8 display the defected scripts created by each group of participants. It can be seen that the experienced participants created sharply lower defect than the intermediate counterparts. On the other hand, the beginner did not create any defects since they drew them directly from the KSD.

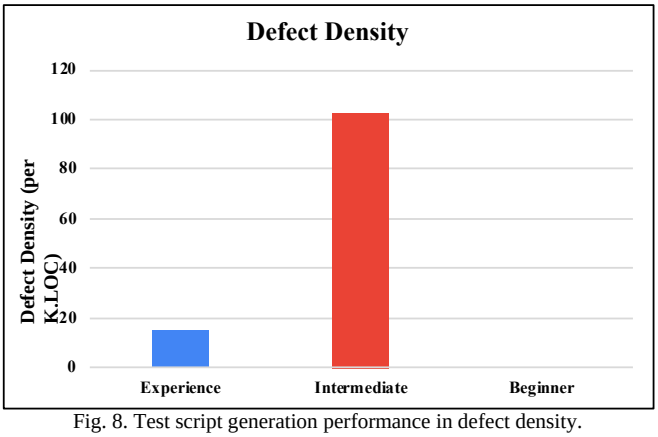

Fig. 8. Test script generation performance in defect density.

C. Threats to Validity

The major threat to validity of this research is the classification of subjects. It is undeniable that although certain participants seem to have a stronger background than others, in fact, they might not have higher experience in some relevant areas. For example, teachers who have more than 5 years of coding might have similar experiences in testing when compared to students who took courses in testing for 2 years. A solution or this threat to validity is the classification should be done based on pre-test results instead of experience in software development.

D. Future Works

The concept of KSD can be implemented for an expansive range of test case generation. For examples, this application could support automatic test script generation in others format including, JUnit of Java [12], PHPUnit of PHP [13], etc.

Other test-intensive software development techniques such as Test-Driven Development [14], [15] are also likely to be beneficial from KSD. This is because KSD could help creating a more thorough and insightful test suite as well as saving time simultaneously.

V. Conclusion

Knowledge of Software Defects (KSD) is a core element in the Defect-Driven Development (DDD). This research conducted an experiment on the utilization of KSD in test case generation. Three group of participants, i.e. experienced, intermediate and beginner, were involved in the research. The results reveal that KSD bridges the gap between the expert and beginners efficiently. This can be seen from the short duration of test case creation and the low defect density of the results from the beginner participants. Consequently, it is highly likely the use of KSD can also benefit everyone in the development process, regardless of their experiences. Also, it is important to initiate a knowledge management process for this aspect in software organizations.

Conflict of Interest

The authors declare that this research has no conflict of interest with other research in the same area.

Author Contributions

Wacharapong Nachiengmai proposed the use of KSD for test case generation. He conducted the experiment, analyzed the result and wrote this article.

Sakgasit Ramingwong revised and restructure paper.

Amphol Kongkeaw assisted in data collection process from expert testers. He also co-supervised all experiment with Wacharapong.

References

[1] L. Fu, G. Sun, and J. Chen, "An approach for component-based software development," in *Proc. 2010 Int. Forum Inf. Technol. Appl. IFITA 2010*, vol. 1, no. Idl, pp. 22–25, 2010.

[2] W. Nachiengmai, S. Ramingwong, K. Cosh, L. Ramingwong, and N. Eiamkanitchat, "Defect-driven development : A new software," *Int. J. GEOMATE*, vol. 17, no. 61, pp. 149–155, 2019.

[3] K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *Proc. 2017 Int. Conf. Energy, Commun. Data Anal. Soft Comput.*, pp. 77–81, 2018.

[4] P. E. Patel and N. N. Patil, "Testcases formation using UML activity diagram," in *Proc. 2013 Int. Conf. Commun. Syst. Netw. Technol. CSNT 2013*, pp. 884–889, 2013.

[5] D. Nirmala and T. Lathamaheswari, "Automated testcase generation for software quality assurance," in *Proc. 10th Int. Conf. Intell. Syst. Control. ISCO 2016*, pp. 1–6, 2016.

[6] Mochajs.org. Mocha JS. [Online]. Available: https://mochajs.org/

[7] Node.js Foundation. Node JS. [Online]. Available: https://nodejs.org/en/

[8] N. Li, Z. Li, and X. Sun, "Classification of software defect detected by black-box testing: An empirical study," in *Proc. 2010 2nd WRI World Congr. Softw. Eng. WCSE 2010*, vol. 2, pp. 234–240, 2010.

[9] W. S. Humphrey, "The personal process in software engineering," in *Proc. the Third International Conference on the Software Process. Applying the Software Process*, 1994, no. c, pp. 69–77.

[10] W. Nachiengmai and S. Ramingwong, "Implementing personal software process in undergraduate course to improve model-view-controller software construction," *Lecture Notes in Electrical Engineering*, vol. 339, 2015, pp. 949–956.

[11] W. Nachiengmai and S. Ramingwong, "Improving reliability of defects logging in MVC-PSP," in *Proc. 2015 2nd International Conference on Information Science and Security (ICISS)*, 2015, pp. 1–4.

[12] The JUnit Team. (2019). JUnit 5. [Online]. Available: https://junit.org/junit5/

[13] S. Bergmann, "PHPUnit – The PHP testing framework," 2015.

[14] K. Beck, *Test Driven Development: By Example*, 1st ed. Addison-Wesley Professional, 2002.

[15] J. W. Wilkerson, J. F. Nunamaker, and R. Mercer, "Comparing the defect reduction benefits of code inspection and test-driven development," *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 547–560, 2012.

Wacharapong Nachiengmai is a lecturer at the department of Business Computer, Faculty of Business Administration, North – Chiang Mai University, Chiang Mai, Thailand. He received his doctoral degree in Computer Engineering from Chiang Mai University, Chiang Mai, Thailand in 2019. His research involves personal software process improvement and software construction techniques.

Sakgasit Ramingwong received his Ph.D. from the University of New England, Australia, in 2009. He is currently an associate professor at Department of Computer Engineering, Faculty of Engineering, Chiang Mai University, Chiang Mai, Thailand. His main research focuses on software project management, risk management, software process improvement and gamification of software engineering aspects.

Amphol Kongkeaw is a lecturer for Software Engineering Program, Faculty of Engineering and Technology, North-Chiang Mai University, Thailand. He received his bachelor degree in Computer Science from Chiang Mai Rabjabhat University, Thailand, in 2002 and master degree in Information Technology and Management from Chiang Mai University, Thailand, in 2008. Currently, he is the head of Software Engineering Program, Faculty of Engineering and Technology, North-Chiang Mai University, Thailand. His current research includes design of project-based learning model on software production process.