Generating Test Cases using Eclipse Environment: A Case Study of Mobile Application

Rosziati Ibrahim[1], Nurul Ain Aswini Abdul Jan[2], Sapiee Jamel[3], Jahari Abdul Wahab[4]
Department of Software Engineering, Universiti Tun Hussein Onn Malaysia, Parit Raja, Malaysia[1, 2]
Department of Information Security, Universiti Tun Hussein Onn Malaysia, Parit Raja, Malaysia[3]
Engineering R&D Department, SENA Traffic Systems Sdn. Bhd, Kuala Lumpur, Malaysia[4]

***Abstract*—In Software Development Life Cycle (SDLC), there are four phases involved. They are analysis, design, implement and testing. Testing is done to ensure the functionalities of the system are correct. There are many approaches to software testing. It is usually divided into two approaches: manual testing or automatic testing. However, these days, with the rapidly advanced technology, performing software testing manually has become hugely laborious but still doable. Therefore, experts of the software development field are beginning to go for automatic testing. This paper presents a case study of mobile application and discusses how test cases can be generated automatically from the application using different automatic tools. Three software testing tools have been used to generate test cases automatically. The results from generating test cases automatically from these three tools are then being compared together with the results of generating test cases using manual testing technique.**

Keywords—Software testing; automation testing; test cases; Eclipse environment

I. Introduction

In Software Development Life Cycle (SDLC), software testing is explained as the phase where a program is executed to be evaluated with the intention to find faults [1]. Although the SDLC is considered as an approach of efficient system development, software testing plays an important role as it assists in finding system deficiencies [2]. As such, testing is done to any software components, making it a vital process considering it aids in discovery of how good it works, validating the quality of the software system. To ensure that developed software components are in good quality, it is crucial to do software testing for the verification and validation to be done properly [3]. Considering how costly a software development project can amount to, testing becomes even more important, as prevention of even more highly cost of the software development. Therefore, it is important that the process is began at early stage during development [4] instead of being carried out by the end of the project development.

Software testing can be accomplished in two ways; either manually or automatically [5]. Manual testing is carried out by software testers without the help of any tools; it is a testing method which is most primitive compared to its peers [6]. On the other hand, contrary to manual testing, automatic testing is performed with assistant from automated testing tool whereby test cases will be generated [7]. The performance capability and functionality of all test cases are to be justified. Testing tools are highly required to perform automatic testing. It plays a crucial role during the testing phase of the SDLC [8]. Several known tools include Robotium [9], Appium and Selenium [10].

This research study main aim is to generate test cases automatically from the existing tools and compared the time taken to generate test cases automatically among the tools. The case study is based on an existing Android mobile application called MyNetDiary [11]. The research shall be able to automate the process of generating test cases. There will be three tools used in the research which are JUnit4 [12], TestNG [13], and EPiT [14]. The results of time taken for each tool to generate test cases automatically will be compared together with the time taken to generate test cases using manual testing.

II. Techniques of Software Testing

Generally, there are two ways to achieve software testing and those are by manual testing or automatic testing. The idea of manual testing is hugely primitive where the tests are executed in the absence of any tools [7]. Differing with manual testing is the automatic testing by which it is done with the help of automatic testing tools [14]. It is believed that by using automatic tools, the trend of automation testing has managed to have better usability, robustness, and correctness [8].

Software testing levels have been categorized into four levels [15]. These four levels include unit testing, integration testing, system testing, and acceptance testing which is shown in Fig. 1. Unit testing focuses on a software system's smallest element which is also known as modules; they are tested independently. Following after the unit testing is the integration testing where the main concept of it is testing the different integrated modules together. For most software project, the value of system testing being carried out is approximately up to 90 percent [16]. And then the last level of testing is the acceptance testing, performed by targeted end users [17]; it has variants of types which include alpha testing, beta testing, business acceptance testing, and the user acceptance testing.

This work was supported by SENA Project from MTUN under RMC, UTHM under Vote No. K234.

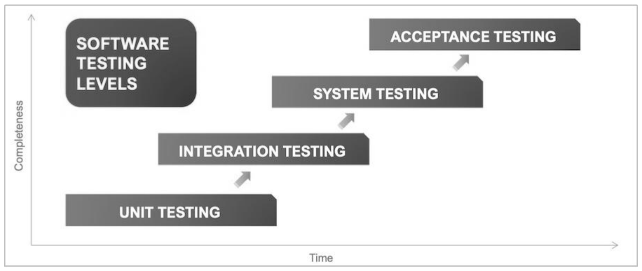


Fig. 1. Software Testing Level [15].

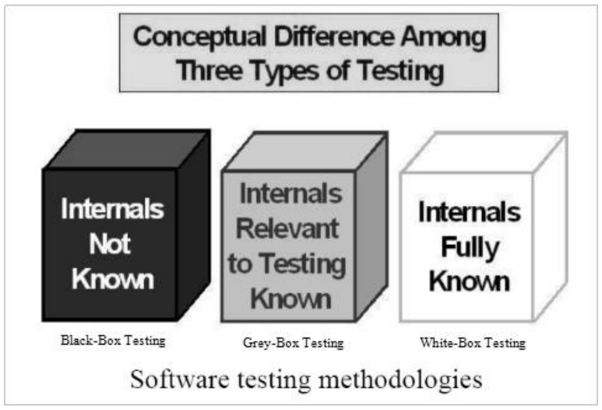

Conceptual Difference Among Three Types of Testing
Internals Not Known
Internals Relevant to Testing Known
Internals Fully Known
Black-Box Testing
Grey-Box Testing
White-Box Testing
Software testing methodologies

Fig. 2. Software Testing Methodology [17].

Fig. 2 shows the visualisation of three common software testing approaches [17]. Over the time period of rapid software development expansion, the common software testing techniques known to most are the black-box testing, white-box testing, and the grey-box testing methods. The grey-box method is a combination of the black-box and white-box testing methods [18].

A. Black-box Testing

Going by many other names such as behavioral testing and functional testing, black-box testing is usually driven without test models or even precise formal documented specifications [18]. The idea of black-box testing is the software testers do not know which of the system's component is being tested. As shown in Fig. 2, the idea of black-box testing is where the users or testers are without knowledge of the system's internals. The testing method concept is accepting inputs and producing expected outputs; black-box testing method borders on the foundation aspects of the system [19].

B. White-box Testing

There are many other nicked names to white-box testing method. Some of them include clear-box testing and glass-box testing. Just as the visual on Fig. 2 suggests, it is a testing method whereby the internals of the system are fully known [20]. As its nickname (clear-box testing), the back end of the system (or its components) is known to testers making it highly efficient in bugs-detection [21]. However, in large-scale software systems, this method is seldom used.

C. Grey-box Testing

Being the combination of black-box testing and white-box testing is the grey-box testing technique [22]. Fig. 2 shows the internals of the system is relevant to the testing being carried out known by the testers. The concept of grey box is commonly known of testers having bits of internal working but going against its specifications [17]. The method typically applies reverse engineering but is not categorized as biased and intrusive; therefore the testers are not inclined to gain access on the internal source code.

III. Related Work

Li *et al.* [23] present DroidBot, an automatic software testing tool which is compatible to most Android mobile apps. DroidBot is said as something that is lightweight and test on UI-guided input generators. It does not require any instrumentation. DroidBot also makes use of malware analysis as it uses a model-based generator that has information about app under test (AUT) from device at runtime, enabling it to trigger sensitive behaviours.

Alotaibi, & Qureshi [10] discuss a new framework to be used for automation testing on mobile application which will be using the Appium framework. According to them, in order to ensure high performance application within a short-given time, the automation of software testing is highly necessary. They specifically discuss Appium as it is considered as a power tool that helps in delivering features. What Appium does, to be precise, is the direct automation on mobile devices. It supposedly works for almost all of hybrid, native, applications of mobile-web for iOS and even Android.

Mao, Harman, & Jia [24] introduce Sapienz which is an Android testing approach that has significantly performed better than even the widely-used tool known as Android Monkey. According to them, Sapienz is better than Monkey is due to the fact that Monkey does automation testing in a deliberate unintelligent way of randomness. Sapienz, on the other hand, is a new automated testing that combines traditional automated testing with the quirks of expanding it to Android testing.

Dolan-Gravitt *et al.* [25] focus on PANDA's four principal criterion; the system's ability to record/replay, the system's plugin architecture, the system's capability in single analysis execution process on multiple architectures, and lastly the ability of Android systems emulation. PANDA is versatile and has simplicity, allowing support of new myriad of architectures and devices with no extra labour. The replay method itself is able to overcome the complexity of operating systems as it is able to record boot for myriads of operating systems. The system is more widely received considering its full repeatability features, a big convenience for dynamic analysis. Hence, considering PANDA is not focused solely on record and replay, it is adequately different than QEMU 2.1.0's numbers just as shown on the table below. However, PANDA takes almost the same amount of time as QEMU 2.1.0.

Hussain, Razak, & Mkpojiogu [26] discuss the perceived usability sentiments regarding the automated testing tools that exist for mobile testing. They discuss that many mobile application developers are using automated testing tools these days and that include MonkeyTalk, Robotium, and more. They state how it is no longer foreign that automated testing tools are gaining trend as it greatly reduces the time taken to conduct the process of testing, excluding errors, and even omitting possible errors due to human factor. They argue how it has become highly important for automated testing tools to be of good usability as automated testing tools should not only support either native or hybrid, but they shall be able to do both. And that includes for Android and iOS.

Rosziati Ibrahim *et al.* [14] discuss the automatic testing tool called EPiT for generating test cases automatically. EPiT is a plug-in tool that can be installed in Eclipse environment. EPiT has a parser that reads the source codes line by line and then extracts all the attributes and functions from the classes and finally generates the test cases of all functions automatically.

Salihu *et al.* [27] propose a model to generate test cases from mobile application based on GUI. AMOGA framework is used for the generation of test cases with two important algorithms embedded within the framework. They are greedy algorithm and crawler algorithm.

IV. UML Specification

UML diagrams are considered as the de-facto standard tool being used for the documentation of object-oriented modelling [28]. Two diagrams have been used for this project. They are use-case diagram and class diagram.

A. Use-case Diagram

Fig. 3 shows the use-case diagram of the research study. Based on Fig. 3, the actor is a user who can execute the tool in order to read the source code files of the case study. After doing so, it will be able to extract the classes and interface information, as well as checking the functions dependency. At last, it will generate the test cases.

B. Class Diagram

The class diagram portraits the classes that are going to be implemented during the development cycle. Fig. 4 shows the class diagram of this research study.

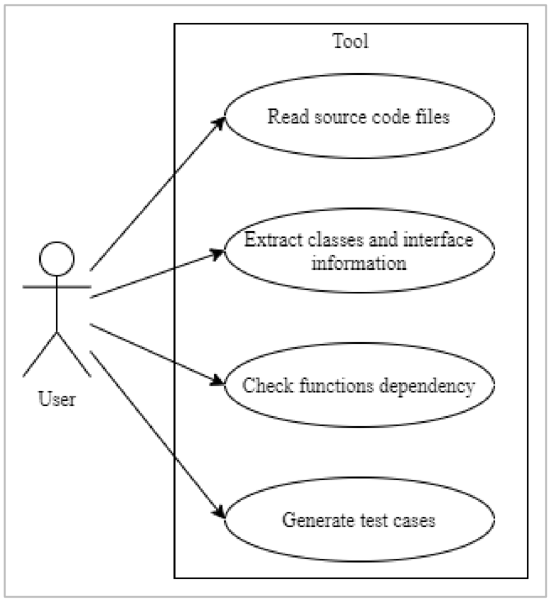

Tool
Read source code files
Extract classes and interface information
Check functions dependency
Generate test cases
User

Fig. 3. Use-Case Diagram.

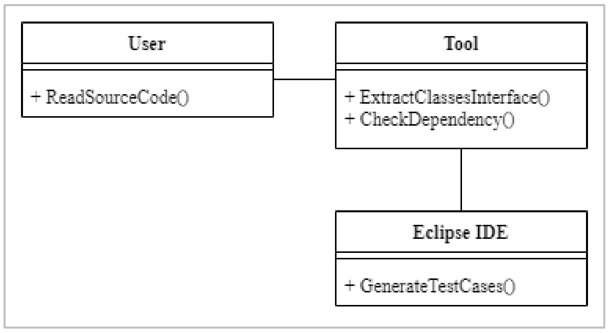


Fig. 4. Class Diagram.

Based on Fig. 4, it shows the specific of which methods belong either to the user, tool or the Eclipse IDE itself. The diagram does not exactly illustrate the directional work flow of the testing but it shows the classes that are being used for the implementation.

V. Research Methodology

The research study follows a specific process that consists of four stages to be carried out in order. For this research study, it will include a total of four major stages, the first stage being the requirement analysis. Next, it is followed by the design, implementation, and testing stages in an orderly manner. These four phases are illustrated in Fig. 5.

Based on Fig. 5, the requirements analysis stage is critical to this research study. As stated by Shukla, Pandey, & Shree [29], many other phases depend on requirements engineering and that includes the design, coding and testing. In this research study, this phase includes identifying the necessary tools and requirements needed. After identification, the requirements needed have been noted. The case study of the research is based on an Android mobile application which is MyNetDiary [11]. From MyNetDiary, the scope is further narrowed down to its 3 modules. The platform used for the research development is Eclipse IDE with the implementation of the Java programming language. Several other software and plugins are required for this research. As the codes of MyNetDiary mobile application cannot be fully obtained, it is determined that the software testing technique used is grey-box testing.

As the analysis phase, design phase is also included in SDLC. On a generic sense, during the design phase, the technical details of a software project are discussed, and this usually comprises of several aspects such as the technologies to be used, constraints, design approach, and so forth [30].

For the implementation phase, Fig. 6 shows the steps for implementing the tool.

Based on Fig. 6, the implementation process begins with first reading the source code file of the case study. Once the source codes are obtained, the automated testing tools which are running on Eclipse IDE will identify the classes and functions to be extracted. After that, the automated testing tools will begin generating the test cases automatically and the time taken for each of the tools and techniques will be observed, and recorded. For each software, testing methods, both manual testing and automation testing; the tests will be run a total of 5 times for each Module 1, Module 2, and Module 3. This was done in order to get the optimal and most accurate data for the research. Lastly, the evaluation of time taken between the manual and automated testing will be made.

A. *Manual Testing Flowchart*

There are three basic activities to be done during the manual testing process as shown in Fig. 7(a). The case study file will first be run and executed, and then software tester will start inserting inputs. The time taken for the process to generate test cases will be recorded.

B. *Automatic Testing Flowchart*

Similar to the previous process of manual testing, automatic testing also follows several steps on generating test cases as shown in Fig. 7(b). The flowchart consists of four activities. The step begins with source code files of the case study being read. Its classes and interface information will be extracted, and the functions dependency will also be checked. Lastly, the test cases will be automatically generated by the selected tools.

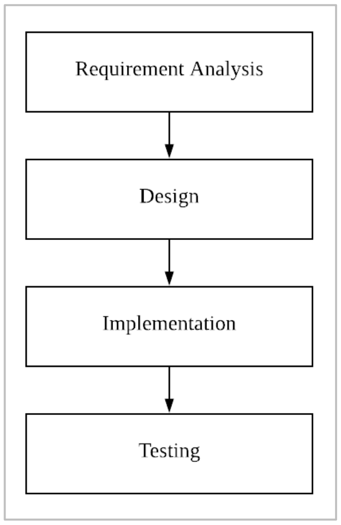


Fig. 5. Research Process.

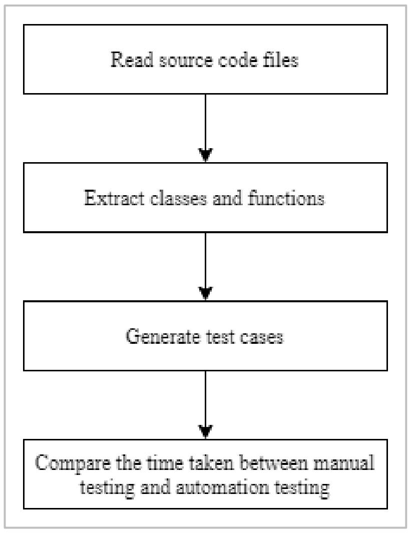

Read source code files
Extract classes and functions
Generate test cases
Compare the time taken between manual testing and automation testing

Fig. 6. Implementation Process.

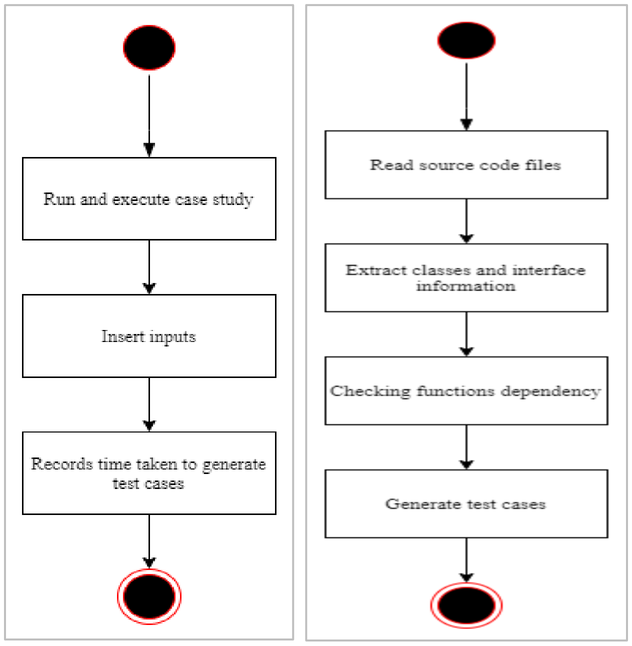

Run and execute case study
Insert inputs
Records time taken to generate test cases
Read source code files
Extract classes and interface information
Checking functions dependency
Generate test cases

Fig. 7. (a) Steps for Manual Testing ; (b) Steps for Automatic Testing.

VI. Results and Discussion

Based on MyNetDiary [11], three modules have been used in order to generate the test cases. Table I shows the details of these three modules.

The data recorded from all the tests run during the research have been tabulated as each module is run at least five times for each respective automatic testing tools. The formula used to calculate the average time taken of tests run is:

$$\mu = \frac{\sum T_m}{5} \quad (1)$$

where $\sum$ indicates the summation of the time taken to run for each module.

A. *Manual Testing Results*

Table II shows the calculation of data on the results of time taken to manually generate the test cases for all three modules.

Fig. 8 is the graphical diagram from Table II. It depicts the value of the average time taken to generate test cases manually for Module 1, Module 2, and Module 3. It took 21.884s, 13.672s, and 15.642s to generate the test cases for Module 1, Module2, and Module 3, respectively.

TABLE I. Details Modules for the Case Study

Module	**Details**
Module 1	Module of Calorie, BMI and Water
Module 2	Module to calculate the amount of calorie consumed from the different meals
Module 3	Module to calculate the amount of calorie burn f1rom different exercises

TABLE II. MANUAL TEST RUN ON THE CASE STUDY

Module	No. of Test	Time Taken (s)	Average Time Taken (s)
1	1	23.490	21.884
	2	22.080	
	3	20.920	
	4	21.710	
	5	21.220	
2	1	13.220	13.672
	2	13.720	
	3	13.600	
	4	14.240	
	5	13.580	
3	1	15.770	15.642
	2	15.600	
	3	15.590	
	4	15.570	
	5	15.680	

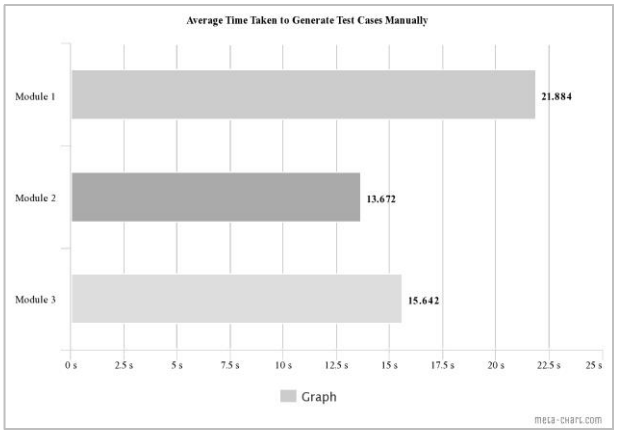


Fig. 8. Average Time Taken to Generate Test Cases Manually.

B. JUnit4 Testing Results

Table III shows the calculation of data on the results of time taken for Junit4 [12] to generate the test cases automatically for all the three modules.

Fig. 9 is the graphical diagram from Table III. It depicts the value of the average time taken to generate test cases automatically for Module 1, Module 2, and Module 3. It took 1.363s, 1.093s, and 0.598s to generate the test cases for Module 1, Module2, and Module 3, respectively.

C. TestNG Testing Results

Table IV shows the calculation of data on the results of time taken for TestNG [13] to generate the test cases automatically for all three modules.

Fig. 10 is the graphical diagram from Table IV. It depicts the value of the average time taken to generate test cases automatically for Module 1, Module 2, and Module 3. It took 0.016, 0.014s, and 0.020s to generate the test cases for Module 1, Module2, and Module 3, respectively.

TABLE III. TEST RUN ON CASE STUDY USING JUNIT4

Module	No. of Test	Time Taken (s)	Average Time Taken (s)
1	1	1.731	1.363
	2	1.361	
	3	1.191	
	4	1.460	
	5	1.070	
2	1	1.288	1.093
	2	1.099	
	3	1.054	
	4	1.080	
	5	0.943	
3	1	0.690	0.598
	2	0.553	
	3	0.578	
	4	0.625	
	5	0.544	

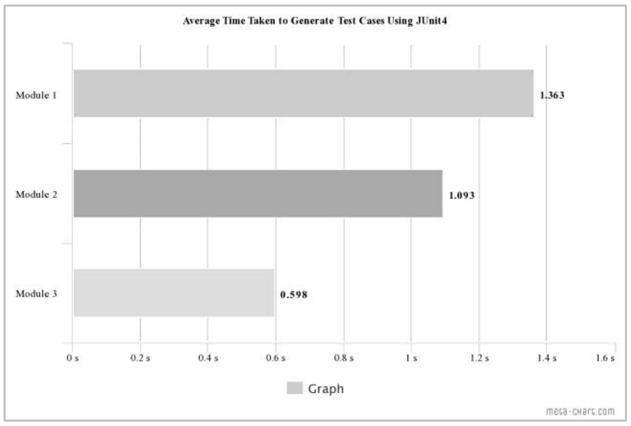


Fig. 9. Average Time Taken to Generate Test Cases Automatically using Junit4.

TABLE IV. TEST RUN ON CASE STUDY USING TESTNG

Module	No. of Test	Time Taken (s)	Average Time Taken (s)
1	1	0.013	0.016
	2	0.021	
	3	0.013	
	4	0.022	
	5	0.013	
2	1	0.014	0.014
	2	0.012	
	3	0.012	
	4	0.019	
	5	0.013	
3	1	0.020	0.020
	2	0.020	
	3	0.020	
	4	0.017	
	5	0.022	

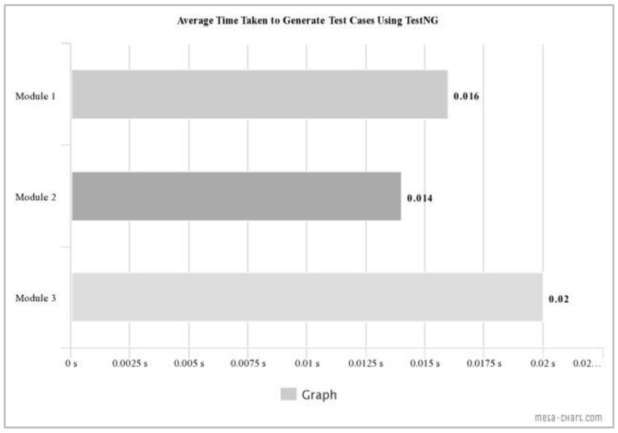


Fig. 10. Average Time Taken to Generate Test Cases Automatically using TestNG.

D. EPiT Testing Results

Table V shows the calculation of data on the results of time taken for EPiT [14] to generate the test cases for all the three modules.

Fig. 11 is the graphical diagram from Table V. It depicts the value of the average time taken to generate test cases automatically for Module 1, Module 2, and Module 3. It took 0.003s, 0.001s, and 0.003s to generate the test cases for Module 1, Module2, and Module 3 respectively.

Fig. 12 shows one of the runtime on Module 2 using EPiT. It took only 0.0001s to generate the test cases automatically from Module 2.

E. Comparative Analysis

Table VI and Fig. 12 are the tabulated data and graphical diagram representation of all testing methods. The time taken to generate the test cases using manual testing takes a significantly longer time than the time taken for the automatic testing tools to generate the test cases. This is clearly shown in Table VI.

TABLE V. TEST RUN ON CASE STUDY USING EPIT

Module	No. of Test	Time Taken (s)	Average Time Taken (s)
1	1	0.005	0.003
	2	0.002	
	3	0.002	
	4	0.002	
	5	0.005	
2	1	0.001	0.001
	2	0.001	
	3	0.001	
	4	0.001	
	5	0.001	
3	1	0.007	0.003
	2	0.001	
	3	0.003	
	4	0.001	
	5	0.001	

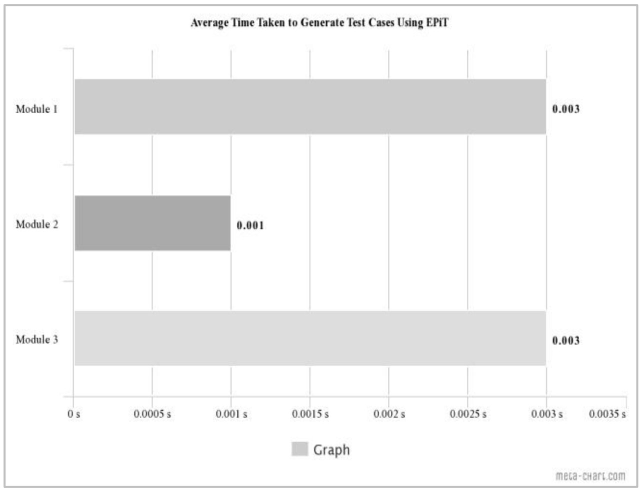
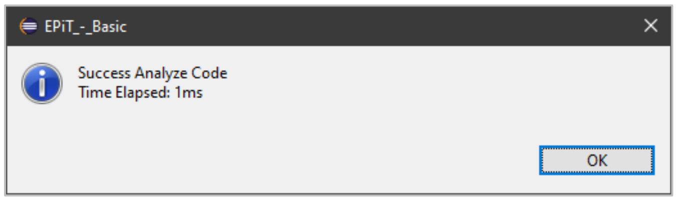


Fig. 11. Average Time Taken to Generate Test Cases Automatically using EPiT.



Success Analyze Code
Time Elapsed: 1ms

Fig. 12. EPiT Time Elapse for Module 2

TABLE VI. MANUAL VS AUTOMATIC TEST RUN ON CASE STUDY

Module	Manual Testing	JUnit4	TestNG	EPiT
1	21.884s	1.363s	0.016s	0.003s
2	13.672s	1.093s	0.014s	0.001s
3	15.642s	0.598s	0.020s	0.003s
	51.198s	3.054s	0.050s	0.007s

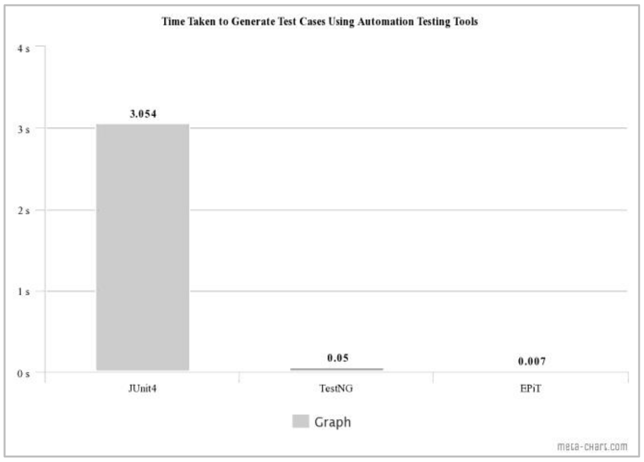


Fig. 13. Time Taken to Generate Test cases using Automatic Testing Tool.

From Table VI and Fig. 13, the time taken to generate test case using manual testing takes a significantly longer time than the time taken for the automatic testing tools to generate test cases. Among the three automation tools used, JUnit4 took the significantly greatest time which total reached more than 3s. Meanwhile TestNG only took 0.05s to generate all test cases for all modules. Meanwhile, EPiT took the shortest time at only 0.007s.

Regarding the differences in time taken to generate test cases of the modules, this can be justified on the code lines of the case study. While the case study has simple time complexity of O(1), the total number of lines for each modules significantly differs with Module 1 having the most number of lines written, followed by Module 2, and Module 3. This causes for the time taken to generate test cases to differ from each of the respective modules. Beside from that, we can conclude that automatic testing is definitely better than manual testing. However, it needs to be noted that manual testing cannot be simply abandoned as it is still necessary for several tasks in any software development projects.

From Table VI, it is noted that manual testing has the biggest time difference compared to the others, which is just as expected. This is because manual testing demands a lot of resources which is one of them is the time resource [31]. Among the three automated testing tools, it is noted that the differences of time taken to generate test cases between JUnit4 and TestNG, as well as EPiT; JUnit4 takes the longest time. In one paper, Kumbhar, Gavekar, & Kulkarni [32] stated that JUnit is quite a lacking tool in generating test result compared to other testing tools. Meanwhile, it is no surprise that TestNG took shorter time than JUnit4 in generating the test cases, as according to Jacob and Karthikevan [33]. EPiT [14] is the latest software testing tool that has the shortest time to generate test cases automatically for the three modules. EPiT uses the algorithm in [34] in order to reduce the redundancy of test cases generated.

VII. Conclusion

This paper has discussed and compared the three automatic tools namely Junit4, TestNG and EPiT for generating test cases automatically. All three tools are plugged into Eclipse IDE. The time taken to generate the test cases has been compared among the tools. After the tests are run, it has been observed that JUnit4 took the longest time to generate all test cases, the time taken being almost up to 3s. Meanwhile TestNG only took 0.05s to generate all test cases for all modules. On the other hand, EPiT took the shortest time at only 0.007s. Therefore, EPiT gives the shortest time in order to generate test cases automatically. Beside from that, we can conclude that automatic testing is definitely better than manual testing. However, it needs to be noted that manual testing cannot be simply abandoned as it is still necessary for several tasks in any software development projects.

Acknowledgment

This project is funded by the Ministry of Higher Education Malaysia (MOHE) under the Malaysian Technical University Network (MTUN) grant scheme Vote K234 and SENA Traffic Systems Sdn. Bhd.

References

[1] Myers, G. J., Sandler, C., & Badgett, T. (1979). The art of software testing, JohnWiley & Sons. *Inc, Canada*.

[2] Jindal, T. (2016). Importance of Testing in SDLC. International Journal of Engineering and Applied Computer Science (IJEACS), 1(02), 54-56.

[3] Souza, É. F. D., Falbo, R. D. A., & Vijaykumar, N. L. (2017). ROoST: reference ontology on software testing. *Applied Ontology*, 12(1), 59-90.

[4] Bertolino, A., & Marchetti, E. (2005). A brief essay on software testing. *Software Engineering, 3rd edn. Development process, 1*, 393-411.

[5] Afrin, A., & Mohsin, K. (2017). Testing approach: Manual testing vs automation testing. *Global Sci-Tech*, 9(1), 55-60.

[6] Patidar, R., Sharma, A., & Dave, R. (2017). Survey on Manual and Automation Testing strategies and Tools for a Software Application. *International journal of advanced research in computer science and software engineering*, 7(4), 10.

[7] Anjum, H., Babar, M. I., Jehanzeb, M., Khan, M., Chaudhry, S., Sultana, S., ... & Bhatti, S. N. (2017). A comparative analysis of quality assurance of mobile applications using automated testing tools. *International Journal of Advanced Computer Science and Applications, 8*(7), 249-255.

[8] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., & Lo, D. (2015, April). Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1-10). IEEE.

[9] Zhu, Y., Hou, Y., & Wang, B. (2015). Application of automatic test tool Robotium for Android [J]. *Information Technology, 10*, 198-200.

[10] Alotaibi, A. A., & Qureshi, R. J. (2017). Novel Framework for Automation Testing of Mobile Applications using Appium. *International Journal of Modern Education & Computer Science, 9*(2).

[11] MyNetDiary.com. (2021) *"Calorie Counter - MyNetDiary, Food Diary Tracker"*. Retrieved December 2020, from: https://play.google.com/store/apps/details?id=com.fourtechnologies.mynetdiary.ad&hl=en&gl=US

[12] Junit4 (2021). Retrieved December 2020 from: https://junit.org/junit4/

[13] TestNG (2020). Retrieved December 2020 from: https://testng.org/doc/index.html

[14] Rosziati Ibrahim, Ammar Aminuddin Bani Amin, Sapiee Jamel, Jahari Abdul Wahab (2020). EPiT: A Software Testing Tool for Generation of Test Cases Automatically. *SSRG International Journal of Engineering Trends and Technology, 2020, 68(7), 8-12.* DOI:10.14445/22315381/IJETT-V68I7P202S

[15] *Software Testing Levels.* (2020). Software Testing Fundamentals. Retrieved December 2020 from: https://softwaretestingfundamentals.com/software-testing-levels/

[16] Jan, S. R., Shah, S. T. U., Johar, Z. U., Shah, Y., & Khan, F. (2016). An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET)*, Print ISSN, 2395-1990.

[17] Kassab, M., DeFranco, J. F., & Laplante, P. A. (2017). Software testing: The state of the practice. *IEEE Software*, 34(5), 46-52.

[18] Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2016, November). Software testing techniques: A literature review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)* (pp. 177-182). IEEE.

[19] Lawanna, A. (2014). The theory of software testing. *AU Journal of Technology, 16*(1), 35-40.

[20] Sneha, K., & Malle, G. M. (2017, August). Research on software testing techniques and software automation testing tools. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)* (pp. 77-81). IEEE.

[21] Mailewa, A., Herath, J., & Herath, S. (2015, April). A Survey of Effective and Efficient Software Testing. In The Midwest Instruction and Computing Symposium. Retrieved from http://www. micsymposium. org/mics2015/ProceedingsMICS_2015/Mailewa_2D1_41. pdf.

[22] Poulova, P., & Klimova, B. (2018). Automated Software Testing—A Case Study. *Advanced Science Letters*, 24(4), 2578-2581.

[23] Li, Y., Yang, Z., Guo, Y., & Chen, X. (2017, May). Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (pp. 23-26). IEEE.

[24] Mao, K., Harman, M., & Jia, Y. (2016, July). Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 94-105).

[25] Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., & Whelan, R. (2015, December). Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (pp. 1-11).

[26] Hussain, A., Razak, H. A., & Mkpojiogu, E. O. (2017). The perceived usability of automated testing tools for mobile applications. *Journal of Engineering, Science and Technology (JESTEC), 12*(4), 89-97.

[27] Salihu, I.A., Ibrahim, R., Ahmed, B.S., Zamli, K.Z. Usman, A. (2019). "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing". IEEE Access. 2019. DOI:10.1109/ACCESS.2019. 2895504.

[28] Pender, T. (2003). *UML 2 Bible*. John Wiley & Sons.

[29] Shukla, V., Pandey, D., & Shree, R. (2015). Requirements Engineering: A Survey. Requirements Engineering, 3(5), 28-31.

[30] Rani, U., Barjtya, S., & Sharma, A. (2017). A detailed study of Software Development Life Cycle (SDLC) models. *International Journal Of Engineering And Computer Science, 6*(7).

[31] Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76*, 92-117.

[32] Kumbhar, M., Gavekar, V., Kulkarni, A. (2020). Performance Testing Tools: A Comparative Study of QTP, Load Runner, Win Runner and JUnit.

[33] Jacob, A., & Karthikevan, A. (2018, March). Scrutiny on Various Approaches of Software Performance Testing Tools. In *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)* (pp. 509-515). IEEE.

[34] Ibrahim, R., Ahmed, M., Nayak, R., Jamel, S. (2020). "Reducing redundancy of test cases generation using code smell detection and refactoring". Journal of King Saud University – Computer and Information Science, Volume 32, Issue 3, March 2020, pp 367-374. DOI:10.1016/j.jksuci.2018.06.005.