Research Article

nCODET: A Tool For Novice Developer To Detect Untestable Code

Saiful Bahri Hisamudin[1*], Salmi Baharom[2], Jamilah Din[3]

[1,2,3]Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia 43400, Serdang, Selangor Darul Ehsan
saifulbahriey@gmail.com[1]

Article History: Received: 10 November 2020; Revised: 12 January 2021; Accepted: 27 January 2021; Published online: 05 April 2021

Abstract: Uncontrollability is troublesome for unit testing. It causes a non-deterministic behavior where the same input can produce different results based on different executions. The non-deterministic characteristic makes it impossible to test the internal logic of a method because it suffers from tight coupling, a single responsibility principle violation, being an untestable code, being non-reusable or hard to maintain. This paper describes a tool, namely the non-deterministic Code Detection Tool (nCODET) that aims to assist novice developers to write testable codes by avoiding the non-deterministic characteristic in their codes. Our research focuses on the unit testability of classes; particularly the effort involved in constructing unit test cases.

Keywords: testable code, testability, non-deterministic code, controllability

1. Introduction

Software testing has always been perceived as a time-consuming process, costly and less significant. But in reality, it is important to produce high quality software products. Meanwhile, software testability is the degree to which a software artifact supports software testing in a given test context. If the testability of a software artifact is high, it indicates that the software is easy to test. There are several factors that affect software testability, such as controllability, observability, containing built-in test capabilities, understandability and complexity. Widely-discussed controllability can improve testability. Binder (1994) stated the importance of controllability; If users cannot control the inputs of components, they cannot be sure what caused a given output. In addition, Devietti *et al.* (2009) described, due to inability to control the input, software defect can only be detected after it is executed a hundred times.

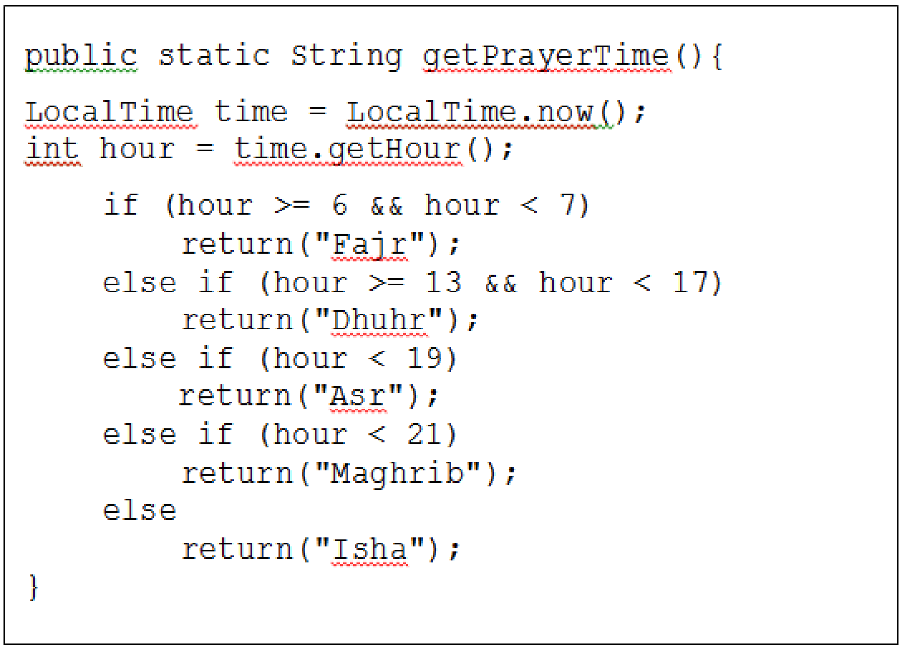
The software is developed in units and every unit is expected to have a defined functionality. Software testing covers a wide range of activities, from unit testing to user acceptance testing. In unit testing, each unit is tested independently from other units to ensure that the unit satisfies its functional specification. However, sometimes, it can be quite difficult to generate test cases for some units. Uncontrollability is one of the reasons why a unit is hard to test. Uncontrollability leads to non-deterministic behavior where same inputs can produce different outputs on different execution. The non-deterministic characteristics of a code makes it impossible to test the internal logic of a unit from a unit testing's perspective. To ensure high testability, a testable code has to be created. Since unit is the basis of complete software, it is crucial to develop testable units before we can develop a testable system. Study by Hayes *et al.* (2015) and Hussain *et al.* (2016) shows that the effectiveness of testing can be improved by designing testable artefacts such as software requirements, a document and the source code of the software. Thus, the developer has to be guided to develop testable codes.

In this paper we describe our on-going research that addresses the problem of untestable code focusing on non-deterministic characteristic introduced by the developer in a code. The essence of the study is on unit testability of a class. The aim of our project is to investigate the strategies and techniques that will improve test efforts when constructing unit test cases.

The remainder of the paper is organized as follow. Section 2 describes the issue with a non-deterministic characteristic. We explain our framework and a proposed tool in Section 3. Section 4 discusses the result, and finally, the concluding remark is given in Section 5.

2. Non-Deterministic Characteristic

Controllability is widely discussed to affect software testability. Better controllability reflects to better testability in software program. Inability to control the input during program execution leads to non-deterministic behavior of software program where the same input can produce different outputs on different executions. Aviram et al. (2012) explained that non-determinism can be caused by multiple sources such as timer, random numbers and incoming messages from a web server. To explain the issue of non-deterministic, let's take a look at a method that displays daily prayer types (i.e. Fajr, Dhuhr, Asr, Maghrib, Isha) as in Figure 1. The method checks the current time that would need to be retrieved from the server’s time.



```
public static String getPrayerTime(){
LocalTime time = LocalTime.now();
int hour = time.getHour();
    if (hour >= 6 && hour < 7)
        return("Fajr");
    else if (hour >= 13 && hour < 17)
        return("Dhuhr");
    else if (hour < 19)
        return("Asr");
    else if (hour < 21)
        return("Maghrib");
    else
        return("Isha");
}
```

Figure 1. Method to display daily prayer time

The method is written in a way that a proper state-based unit test is impossible to be performed. LocalTime.now() is a hidden input and yet cannot be controlled. The input value might change during each test execution and thus it will produce a different output. Such non-deterministic behavior makes it impossible to test the internal logic of the getPrayerTime() method because the method suffers from tightly coupling, single responsibility principle violation, untestable code, non-reusable and hard to maintain.

3. Materials And Methods

This section presents a framework of the proposed tool in order to detect non-deterministic characteristic of a code. A library is used to keep the patterns of non-deterministic characteristics where an untestable code is recognized based on these patterns. Figure 2 shows the framework of the proposed tool. It consists of three phases which are pre-processing, processing and post-processing.

- **Phase 1**-Pre-processing of the sample code: In this phase, a user can upload or paste a code sample into the system that may contain non-deterministic characteristic The tool will not allow an empty input field where an error message will be prompted to the user.

- **Phase 2**-Processing of the code: Once the code sample is uploaded, this phase processes the code sample to detect the non-deterministic characteristic using a library that consists of predefined dictionary of non-deterministic words. The tool classifies and matches every word in the uploaded code with the predefined words in the dictionary. The predefined dictionary of non-deterministic words as shown in Figure 3.

- **Phase 3**-Post processing of the code: This phase presents the output of the processed code. We use red font to differentiate the detected words with non-detected words. The number of detected words is calculated for each non-deterministic characteristic. The result is simplified in the form of table. Additionally, the result also shows the explanation on each characteristic detected in the code sample as a quick reference.

Figure 4 illustrates the architecture of the proposed tool. The requirement for developing tool consists of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), PHP programming language, Apache web server, MySQL for the database and Macromedia Dreamweaver for text editing.

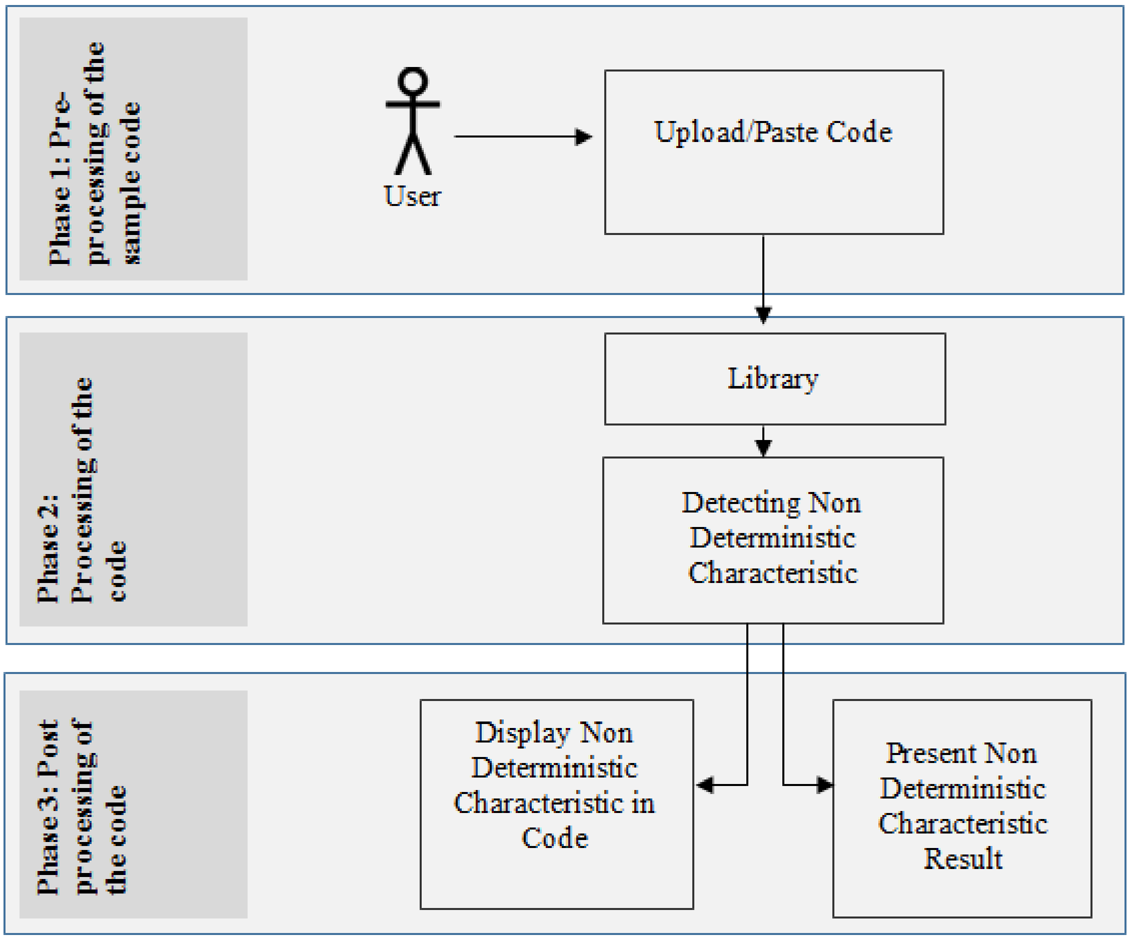
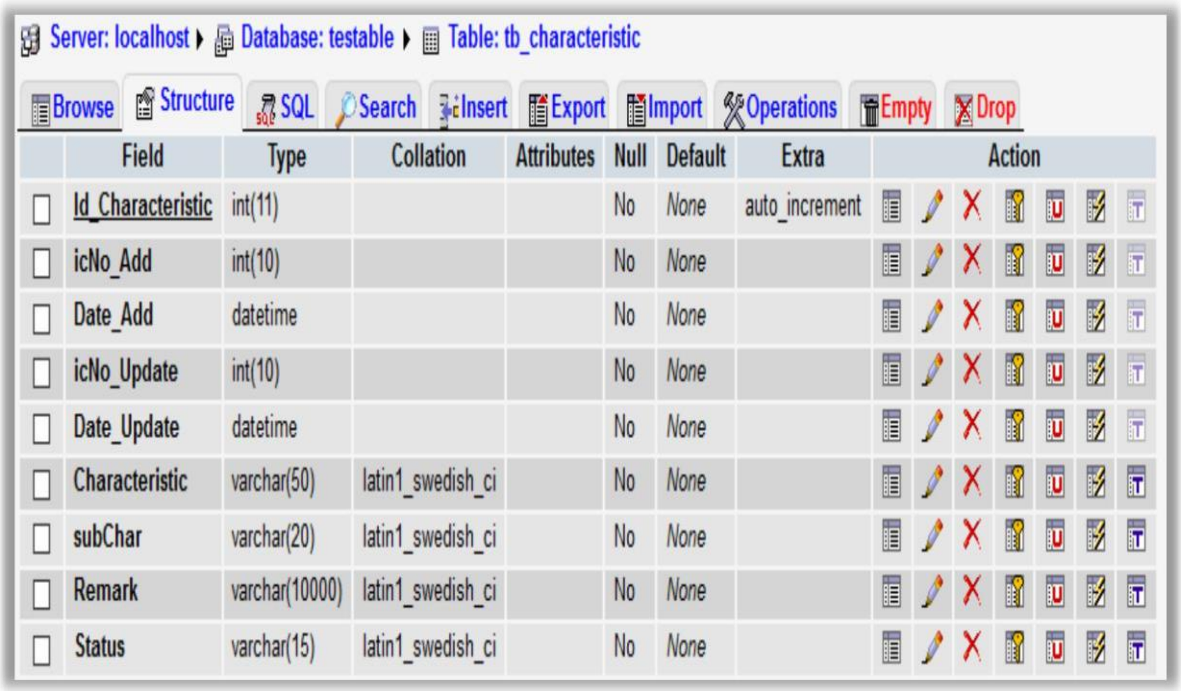

Phase 1: Pre-processing of the sample code
User
Upload/Paste Code
Phase 2: Processing of the code
Library
Detecting Non Deterministic Characteristic
Phase 3: Post processing of the code
Display Non Deterministic Characteristic in Code
Present Non Deterministic Characteristic Result

Figure 2. Framework of the Proposed Tool



	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	Id_Characteristic	int(11)			No	None	auto_increment	
☐	icNo_Add	int(10)			No	None		
☐	Date_Add	datetime			No	None		
☐	icNo_Update	int(10)			No	None		
☐	Date_Update	datetime			No	None		
☐	Characteristic	varchar(50)	latin1_swedish_ci		No	None		
☐	subChar	varchar(20)	latin1_swedish_ci		No	None		
☐	Remark	varchar(10000)	latin1_swedish_ci		No	None		
☐	Status	varchar(15)	latin1_swedish_ci		No	None		

Figure 3. Representation of Predefined Dictionary of Non-Deterministic Characteristic

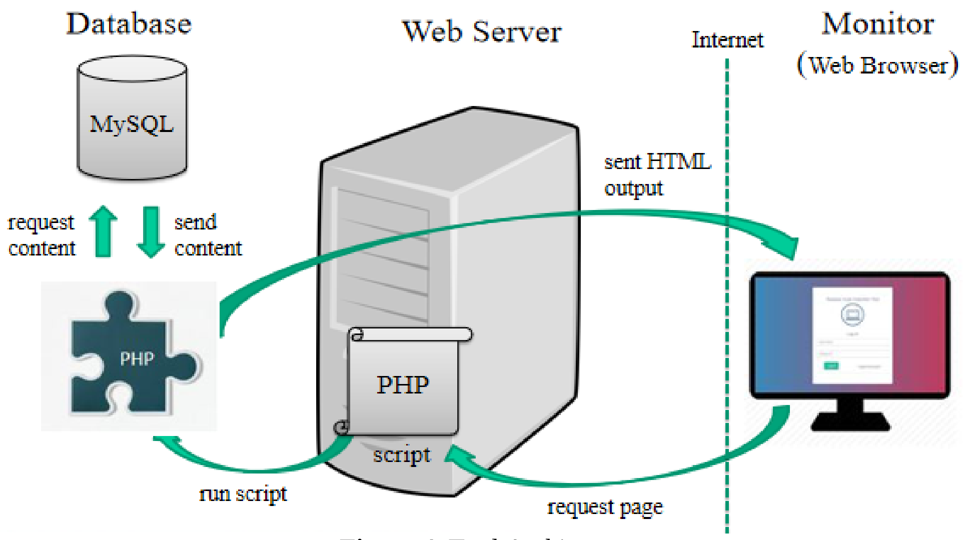
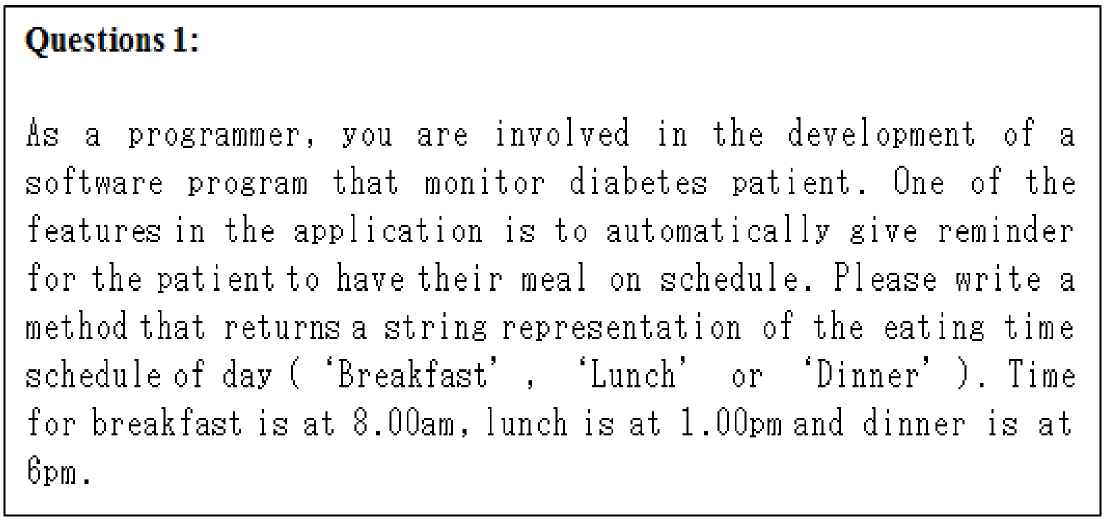


Figure 4. Tool Architecture

We conducted a preliminary experimental study to evaluate the ability of the proposed tool to detect non-deterministic characteristics of a code. Two sources of code samples were collected. They were obtained from GitHub, online websites and students' submissions. Several researchers including Shi *et al.* (2016) and Suwa *et al.* (2017) used this online website as their source of codes. A total of five code samples that contained at least one non-deterministic characteristic were randomly-picked from GitHub. Meanwhile, for the students' submissions, we acquired help from a class of 17 Software Engineering undergraduate students who have already learned Java programming. Each student is given three questions where the student was required to write three Java codes based on three different libraries. Figure 5 shows one of the questions that are given to the students.



Questions 1:

As a programmer, you are involved in the development of a software program that monitor diabetes patient. One of the features in the application is to automatically give reminder for the patient to have their meal on schedule. Please write a method that returns a string representation of the eating time schedule of day ('Breakfast' , 'Lunch' or 'Dinner'). Time for breakfast is at 8.00am, lunch is at 1.00pm and dinner is at 6pm.

Figure 5. Sample Question

A total of 51 code samples were collected based on the three questions. As the tool is still under development, the ability of the proposed tool is determined by comparing the results obtained from the proposed tool with manual detection. An overview of the experimental study is shown in Figure 6.

4. Results and Discussion

Table 1 shows the data collection from manual and nCODET detections based on code samples obtained from the students' submissions. The result from the manual detection shows a total of 16, 10 and 13 students use non-deterministic characteristics in their codes to answer Question 1, Question 2 and Question 3 respectively. In order to evaluate the ability of nCODET in detecting non-deterministic charateristics, we run the same sample codes with nCODET. From the experiment, nCODET detects the same number of detections as the manual detection for Question 1 and Question 2. However, for Question 3, nCODET is unable to detect non-deterministic characteristics from three code samples because these code samples use a generalized or simplified library such as java.io.* which is not considered by the algorithm of nCODET. The tool, nCODET only considers a full library name such as java.io.file, and not of java.io.*. On average, the nCODET detected 71% of the 51 code samples from the students' submissions.

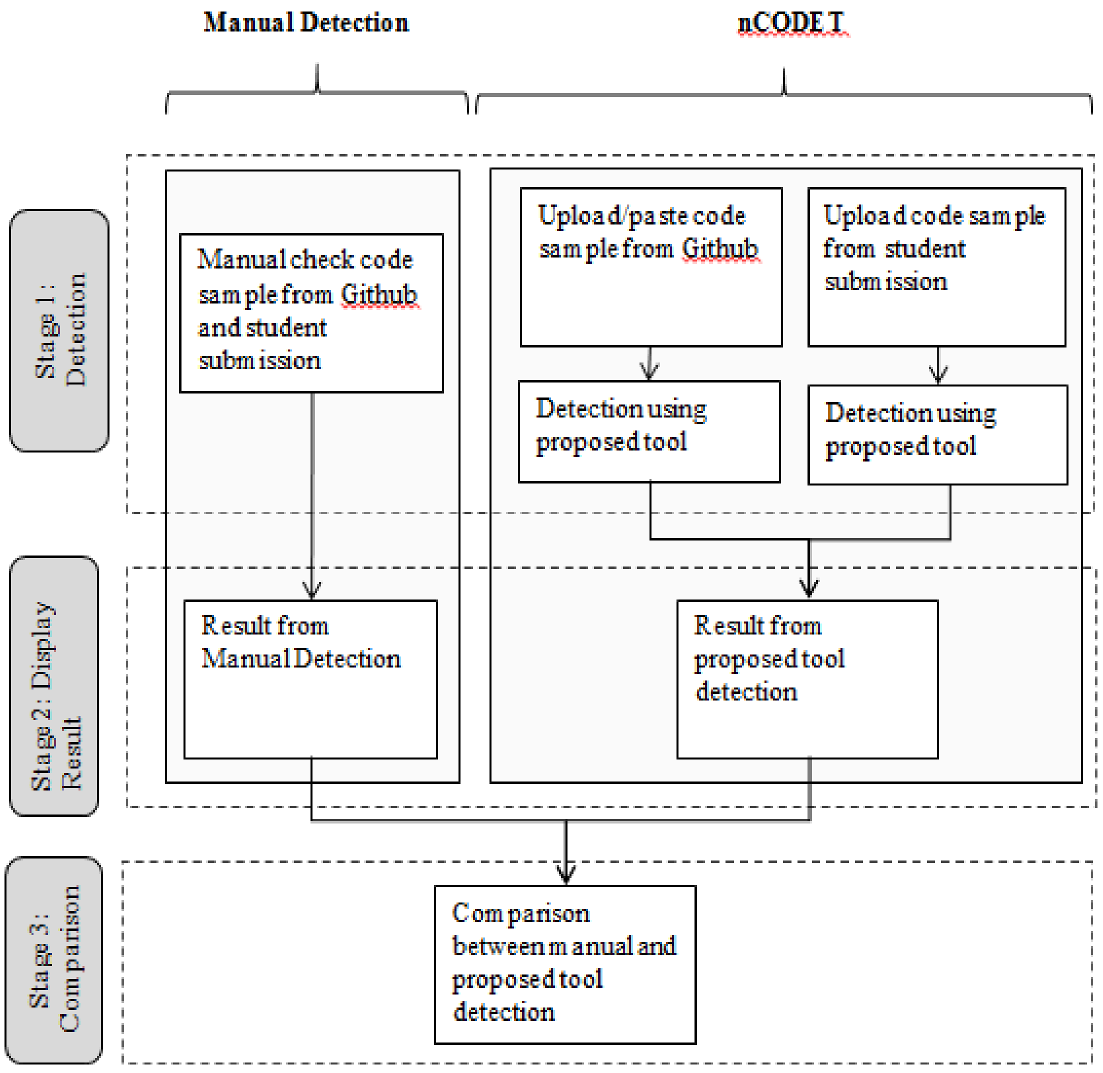

Manual Detection
nCODET
Stage 1: Detection
Manual check code sample from Github and student submission
Upload/paste code sample from Github
Upload code sample from student submission
Detection using proposed tool
Detection using proposed tool
Stage 2: Display Result
Result from Manual Detection
Result from proposed tool detection
Stage 3: Comparison
Comparison between manual and proposed tool detection

Figure 6. Evaluation Framework

Table 1. Data Collection from Students' Submissions

Code Sample	**Respondent**	**Manual Detection**	**nCODET Detection**
Question 1	17	16	16
Question 2	17	10	10
Question 3	17	13	10

Table 2 shows the data collection of manual and nCODET detections based on code samples obtained from GitHub. We randomly chose sample codes and only codes that contain non-deterministic characteristics are considered in the experimental study. Then, we run the chosen sample codes using nCODET. From the experiment, it shows that the nCODET detects non-deterministic characteristics except for Code 5. This is due to the use of a generalized library name which was explained earlier. On average, the nCODET detected 80% of the 5 code samples from GitHub.

Table 2. Data Collection from GitHub Code Sample

Sample No.	**Manual Detection**	**nCODET Detection**
Code 1	1	1
Code 2	1	1
Code 3	1	1
Code 4	1	1
Code 5	1	0

Thus, the overall average of nCODET detection is 76% based on 56 sample codes from two different sources which are students' submissions and GitHub online website.

5. Conclusion

This research contributes to software testability by developing a tool with a dictionary to detect non-deterministic characteristics in a code. The tool is developed to guide novice programmer to write testable code during software development. However, the proposed tool has to be improved as it has a number of limitations. Currently, the developed tool is only able to detect non-deterministic characteristics from code samples with complete library names. In the future, the tool can be improved by enhancing the algorithm in the proposed tool so that it can detect non-deterministic characteristics in code samples with more libraries. Meanwhile, the scope of this tool is limited to non-deterministic characteristics based on a software library. In the future, the capabilities of this tool can be improved by adding other sources of non-deterministic.

6. Acknowledgment

The authors would like to acknowledge the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia for the financial support.

References

1. Aviram, Amittai, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. "Efficient System-Enforced Deterministic Parallelism." *Commun. ACM* 55 (5): 111–119. https://doi.org/10.1145/2160718.2160742.
2. Binder, Robert V. 1994. "Design for Testability in Object-Oriented Systems." *Commun. ACM* 37 (9): 87–101. https://doi.org/10.1145/182987.184077.
3. Devietti, Joseph, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. "DMP: Deterministic Shared Memory Multiprocessing." *SIGARCH Comput. Archit. News* 37 (1): 85–96. https://doi.org/10.1145/2528521.1508255.
4. Hayes, J H, W Li, T Yu, X Han, M Hays, and C Woodson. 2015. "Measuring Requirement Quality to Predict Testability." In *2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, 1–8. https://doi.org/10.1109/AIRE.2015.7337622.
5. Hussain, A., Mkpojiogu, E.O.C., Kamal, F.M. (2016). Mobile video streaming applications: A systematic review of test metrics in usability evaluation. Journal of Telecommunication, Electronic and Computer Engineering, 8 (10), pp. 35-39.
6. Shi, A, A Gyori, O Legunsen, and D Marinov. 2016. "Detecting Assumptions on Deterministic Implementations of Non-Deterministic Specifications." In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 80–90. https://doi.org/10.1109/ICST.2016.40.
7. Suwa, H, A Ihara, R G Kula, D Fujibayashi, and K Matsumoto. 2017. "An Analysis of Library Rollbacks: A Case Study of Java Libraries." In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, 63–70. https://doi.org/10.1109/APSECW.2017.25.