TESTITALREADY: A CODE-FREE APPROACH FOR AUTHORING EXECUTABLE AND MAINTAINABLE TEST CASES FOR NON-TECHNICAL STAKEHOLDERS

by
MAHDI ALI POUR

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of Master of Computer Science

Sabancı University
July 2021

TESTITALREADY: A CODE-FREE APPROACH FOR AUTHORING EXECUTABLE AND MAINTAINABLE TEST CASES FOR NON-TECHNICAL STAKEHOLDERS

Approved by:

. . . .

Date of Approval: June 2, 2021

MAHDI ALI POUR 2021

All Rights Reserved

ABSTRACT

TESTITALREADY: A CODE-FREE APPROACH FOR AUTHORING EXECUTABLE AND MAINTAINABLE TEST CASES FOR NON-TECHNICAL STAKEHOLDERS

Mahdi Ali Pour

Computer Science, Master's Thesis, July 2021

Thesis Supervisor: Assoc. Prof. Cemal Yilmaz

Keywords: Google blockly, BDD Cucumber, Capture amd Replay, automated test, test case

In the process of software development, software testing is an important part that makes a product satisfied by all expectations and requirements. Existing software testing tools need software testing knowledge to be used, and they are not literally readable by non-technical stakeholders. The use of Behavior Driven Development (BDD) techniques has been rapidly increasing since it uses Gherkin syntax which is similar to natural language and extremely easy to understand. In our tool, we aim to create a code-free framework for non-technical personnel can implement their own test suite in BDD and implement the middle layer by using Google Blockly. We suggest testers, to use TestProject Capture&Replay, which is a free web application, to capture a script and import it to our tool for locating elements in Android or iOS devices. Moreover, users either are able to use the subset of actions in the captured list generated by TestProject or all actions in their test cases. Our tool enables users to modify the test suite in Google Blockly to have additional blocks such as loops, if-then-else statements, which make our tool more flexible and unique from other existing testing tools.

ÖZET

TESTITALREADY: TEKNIK OLMAYAN PAYDAŞLAR IÇIN UYGULANABILIR VE SÜRDÜRÜLEBILIR TEST ÖRNEKLERININ YAZILMASINA DAIR KOD IÇERMEYEN BIR YAKLAŞIM

Mahdi Ali Pour

Bilgisayar Bilim, Yüksek lisans tezi, Temmuz 2021

Tez danışmanı : Doçent Doktor Cemal Yılmaz

Anahtar Kelimeler: Google blockly, BDD cucumber, Capture and Replay, automated test, test case

Yazılım denemeleri, yazılım geliştirmede bir ürünün bütün gereksinimleri ve beklentileri karşılamasını sağlar. Var olan yazılım deneme araçları, yazılım deneme bilgisinin kullanımı gerektiriyor ve bunlar teknik bilgiye sahip olmayan kişiler tarafından okunamıyor. Davranış odaklı geliştirme teknikleri (BDD) doğal dile çok benzeyen ve kolay anlaşılır Gherkin sentaksını kullandığından çok hızlı bir şekilde artıyor. Biz aracımızda, teknik bilgiye sahip olmayan çalışanların BDD ile kendi testlerini hazırlayabileceği ve orta katmanı Google Blocky ile uygulayabileceği teknoloji harikası bir sistem kurmayı amaçladık. Kullanıcılara senaryolarını kaydetmek ve bizim aracımıza yollamaları için bedava bir internet uygulaması olan TestProject Capture&Replay kullanmalarını öneriyoruz. Bunun dışında, kullanıcılar TestProject tarafından oluşturulan dizilerin bir kısmını ya da tamamını kullanabilirler. Bizim aracımız kullanıcıların test odasını Google Blockly'de fazladan döngüler ve if-then-else ile değiştirmesine izin verdiğinden var olan deneme araçlarından daha esnek ve özel.

ACKNOWLEDGEMENTS

I am honored to express my profound and sincere to my thesis advisor Dr. Cemal Yılmaz for his tremendous support and guidance during my master's study at Sabanci university. My sincere appreciation goes to the jury members Dr. Hüsnü Yenigün and Dr. Tuğkan Tuğlular for their insightful comments.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

1. INTRODUCTION 1

1.1. Test Driven Development and Behavior Driven Development 2
1.2. Google Blockly 3
1.3. Capture and replay 4

2. RELATED WORK 6

3. PROBLEM DEFINITION 8

4. APPROACH 10

4.1. Google Blockly 10
4.1.1. Test suite 12
4.1.2. Test case 12
4.1.3. Run test case 12
4.1.4. Assert 12
4.1.5. Open existing app on Android 15
4.1.6. Install new app on Android 15
4.1.7. Open existing app on iOS 15
4.1.8. Install new app on iOS 15
4.1.9. Tap 15
4.1.10. Tap point 16
4.1.11. Type text 16
4.1.12. Get element 16
4.1.13. Check existence 16
4.1.14. Pause 16
4.1.15. Swipe 17
4.1.16. Swipe gesture 17
4.1.17. Scroll up 17
4.1.18. Scroll down 17
4.1.19. Go to 17
4.1.20. Set driver 17
4.1.21. Given 20
4.1.22. When 20
4.1.23. Then 20

4.1.24. Parametric Given 20
4.1.25. Parametric When 20
4.1.26. Parametric Then 21
4.1.27. Parameter 21
4.1.28. Step name 21
4.2. Capture and Replay 24
4.3. Behavior Driven Development (BDD) 30
4.4. Implementation 32
4.4.1. Creating Test Case 32
4.4.2. Parameters 33
4.4.3. Saving Test Case 34
4.4.4. Running Test Case 34

5. EMPIRICAL EVALUATION 36
5.1. Experiment 36
5.2. Empirical result 38

6. CONCLUSION 41

7. FUTURE WORK 42

LIST OF TABLES

Table 4.1. Test suite blocks details 11
Table 4.2. Appium action blocks 14
Table 4.3. BDD step block details 19
Table 4.4. Test scenario for Clock App 25

Table 5.1. Test scenario for 3 subject applications 37

LIST OF FIGURES

Figure 1.1. BDD Scenario 2
Figure 1.2. Step definition 3
Figure 1.3. Compiling Google Blockly to Python 4

Figure 3.1. Implementation of BDD 8
Figure 3.2. Step definition in Python 8

Figure 4.1. Test suite blocks 11
Figure 4.2. Appium action blocks 13
Figure 4.3. BDD step blocks 18
Figure 4.4. Test suite example 22
Figure 4.5. Created Python code by Test suite blocks 22
Figure 4.6. BDD WHEN block 22
Figure 4.7. Created Python code for BDD WHEN block 23
Figure 4.8. BDD THEN block 23
Figure 4.9. Created Python code for parametric BDD THEN block . . . 23
Figure 4.10. TestProject captured a list and mirror view of emulator . . . 24
Figure 4.11. Creating "Given" step definition from captured script 26
Figure 4.12. Creating "When" step definition from captured script 27
Figure 4.13. Creating "Then" step definition from captured script 28
Figure 4.14. Files app home screen checking by modified "When" step . . . 29
Figure 4.15. Modified "When" step definition 29
Figure 4.16. BDD test scenario to set an alarm at 7:00 AM 30
Figure 4.17. BDD test scenario using "And" and "But" key words 30
Figure 4.18. HTML BDD test report 31
Figure 4.19. Action information 32
Figure 4.20. Captured test script with the parameterization opportunity . 33
Figure 4.21. Parameterization form, when we have not defined yet 33
Figure 4.22. Parameterization form, when the parameter is defined 33
Figure 4.23. File explorer 34

Figure 5.1. Student's academic status 36
Figure 5.2. Student's experiences in software testing 37
Figure 5.3. Time spent for each tasks 38
Figure 5.4. Submitted test cases scores 39
Figure 5.5. Difficulty level of the Calculator scenarios 39
Figure 5.6. Difficulty level of the Clock scenarios 39
Figure 5.7. Difficulty level of the Files scenarios 40

1. INTRODUCTION

There has been a remarkable increase in the number of mobile applications in recent year thanks to advances in mobile devices and technology. By 2019, 4.4 million mobile apps will be available on Apple App Store and Google Play by the end of the year. The proportion of worldwide website traffic produced by mobile devices rose from 0.7 percent to 52.6 percent between 2009 and 2019 due to the emergence of mobile apps (Xue, 2020).

To achieve a high level of app satisfaction, developers apply testing to ensure the high quality of an app. Manual testing simply cannot keep up with the increasing complexity of systems and apps. Mobile software testing automation is usually used to determine whether the mobile app or the entire software meets the end user's requirements. Testing software is automated by utilizing an automation tool to automate the processes of manual test cases, therefore reducing the testing life cycle in terms of time. (Bathla & Bathla, 2009).

It is not surprising that test automation comes at a high cost, but once purchased, thousands of projects can be executed and tested. Mobile application testing is different and more difficult than traditional desktop and web application testing (Saad & Bakar, 2014).

With a minimal collection of scripts, automated testing optimizes the testing effort. The automation tester is a technical expert who is able to create, debug, and support software for ready-to-use test scripts, a test suite, and automated testing tools. A test script is a sequence of steps for automatically testing specific parts of a piece of software. A test run is a combination of test scripts and test suites based on the intended goals and a possible automated testing method. A test suite is a collection of test scripts used to evaluate a specific piece of software, and a test run is a collection of test scripts and test suites used to evaluate a specific piece of software (Apple-Developer, 2011; Yeh, Chang & Miller, 2009).

Manual or automated methods can be used to test the user interfaces in mobile applications. In manual method, test cases are first designed manually, then implemented and finally executed. On the other hand, automated GUI testing can be performed using automatic input generation (AIG) tools (Choudhary, Gorla &

Orso, 2015) and (Mariani, Pezzè, Riganelli & Santoro, 2014), can be used to automatically generate sequences of input events and attempt to test the GUI of an app. A number of AIG methods, especially at the GUI level, have been developed by academic and industrial professionals to support testing and achieve different levels of automation in app testing (Méndez Porras, Quesada López & Jenkins Coronas, 2015).

1.1 Test Driven Development and Behavior Driven Development

Test Driven Development (TDD) has become an important method for developing software, with its roots in agile programming development. New features are tested in advance because testing and writing code are fundamentally intertwined in TDD, and test cases are written before code. An extension of TDD has recently been proposed. In Behavior Driven Development (BDD), the test cases are written in a natural language called Gherkin to describe the behavior of the software while hiding the implementation details, which facilitates communication with stakeholders as they do not need to read the code (Lübke & van Lessen, 2016).
The BDD language focuses on behavioral aspects rather than testing by using three simple sentences beginning with Given[context], When[event], and Then[result] (North, 2012). Context refers to antecedent conditions or system states, event describes a triggering event, and outcome is an expected or unexpected system behavior (Wang & Wagner, 2018).

The structure Given-When-Then links human concepts and effects to software concepts. In BDD, each test case is called a scenario and multiple scenarios are called a feature file (Diepenbeck, Soeken, Große & Drechsler, 2012). Figure 1.1 shows a simple scenario where 2 numbers are added in the Calculator application.

```
1 Feature: Test sample
2   Scenario: Adding two numbers
3     Given Calculator app is on
4     When I add the numbers 4 and 6
5     Then result is 12
```

Figure 1.1 BDD Scenario

Scenario steps must be linked to the actual test code in order to run the scenario. Step definitions are tuples of a keyword (such as Given, When, or Then), a regular

expression, and step code. Whenever a scenario record matches the regular expression (also called step), the step code is executed. The step definitions for the given scenario are shown in Figure 1.2.

```
@Given('Calculator app is on')
def Calculator_app_is_on(self):

   # running Calculator app

@When('I add the numbers 4 and 6')
def I_add_the_numbers_and(self):

   # adding up 2 numbers

@Then('result is 12')
def result_is(self):

   # checking expected result
```

Figure 1.2 Step definition

One of the main advantages of BDD is its reusability, where step definitions can be used iteratively. For example, step implementations can be used for multiple test scenarios to avoid duplicate code for each test scenario implementation. These scenarios are a valuable tool for software developers, testers, and researchers to communicate and improve software quality. They are often used to record the predicted behaviour of the software (Rahman & Gao, 2015). Despite all the advantages such as reusability of stage definition and separation of concerns between developers, testers and business analysts, the existing challenges that should be solved only by technical testers and developers are implementation and maintenance. Developers and testers should provide an non-faulty step definition to business analysts as the back-end of BDD scenarios.

1.2 Google Blockly

Block-based programming has become a common approach to initial programming environments for young students in recent years to help them get started with low-threshold programming. The use of block-based programming allows developers and educators to reduce the complexity of introducing students to basic programming concepts (Seraj, Katterfeldt, Bub, Autexier & Drechsler, 2019).

```
Language: Python

Count = None


Count = 1
while Count <= 3:
  print('Hello World!')
  Count = Count + 1
```

Figure 1.3 Compiling Google Blockly to Python

The world of Google Blockly is based on concepts similar to popular scratch languages. Google Blockly blocks have placeholders for variables and subclauses of variables, commands, and can express the scope of program segments, depending on the notation of a block that physically includes other blocks, with possible nesting. This Scratch coding style is more appealing to children than traditional programming languages. A number of studies have used block-based programming to promote programming accessibility for learning, such as Programming Interactive Applications with JavaScript and Blockly (Marron, Weiss & Wiener, 2012) and Visual Programming for Media Computation and Bluetooth Robotics Control (Trower & Gray, 2015). Blockly-based programming is also used by the authors of (Culic, Radovici & Vasilescu, 2015) to streamline the process of creating visual programming elements to constantly provide users with new and up-to-date visual programming blocks. In Chapter 3, we explain how we use Google Blockly to overcome the BDD step definition challenge. By using Google Blockly, business analysts can implement the BDD step definition themselves.

1.3 Capture and replay

Test scripts can be created automatically by interacting with the application under test (AUT) using capture and replay (C&R) tools. This makes C&R a very attractive alternative to writing test scripts manually from a usability perspective, as it helps testers with minimal testing experience to quickly create test scripts that represent actual usage scenarios. It is then possible to automatically repeat these scripts, and in certain situations this can be done on a different computer than the one on which they were registered. Because of the level of abstraction from the graphical user interface they take when recording events, C&R tools are categorized as coordinate-based, layout-based, or visual tools (Di Martino, Fasolino, Starace & Tramontana,

2020).
By storing the exact display coordinates in which they reside, coordinate-based tools completely ignore the AUT interface and monitor events. Appetizer replaykit (Appetizer.io, 2009), RERAN (Gomez, Neamtiu, Azim & Millstein, 2013), VALERA (Hu, Azim & Neamtiu, 2015), RandR (Sahin, Aliyeva, Mathavan, Coskun & Egele, 2019), Mosaic (Halpern, Zhu, Peri & Reddi, 2015), and OBDR (Moran, Bonett, Bernal-Cárdenas, Otten, Park & Poshyvanyk, 2017) are notable examples of tools in this category.
GUI components are identified by layout-based software based on certain objective layout properties (e.g., unique IDs and query language expressions). Espresso Test Recorder (Android-Developers, 2009), developed by Google as part of the Android Studio IDE , is a well-known example of tools that belong to this group. Visual tools such as Sikuli (Yeh et al., 2009) and EyeStudio (Eyestudio, 2009) record identical screen captures of GUI components that users interact with and play back the captured script using an image matching process, which is also true for the test part (Ardito, Coppola, Morisio & Torchiano, 2019).
Although using C&R tools is easy for non-technical users, there are limitations as the tests recorded by C&R need to be updated when an application GUI is changed (Di Martino et al., 2020).
In this study, we use a high-level layout-based C&R tool to obtain the application selector information, such as the Id/Xpath of the element and the order of actions, for further processing.

2. RELATED WORK

The development of mobile application testing has been greatly enhanced by all these automated testing procedures. However, their implementation depends more or less on the internal information of the apps, which leads to limitations in their use. The main automated testing methods for mobile applications can generally be divided into three levels (Xue, 2020).

First stage techniques focus on automating test results. Script-based testing enables the automatic execution of test cases defined by manual script editing. The test automation frameworks, including XCTest (Apple-Developer, 2011) can be used to simulate the execution of the application under test by converting scripts into event streams and inputs. Unlike text-only scripts, Eyeautomatic (EyeAutomate, 2009)allows visual information to be specified and visual GUI tests to be performed by image matching. To achieve a high success record replay rate for single and cross-devices, SARA (Guo, Li, Lou, Yang & Liu, 2019) uses the proposed self-replays and adaptive playback mechanisms. In terms of cross-platform testing of turntables, LIRAT (Yu, Fang, Feng, Zhao & Chen, 2019) is a blend of image processing technology. Tuan Anh Nguyen et al. (Nguyen & Csallner, 2015) presents a tool, REMAUI, that uses computer vision and OCR technology to recognize input images for UI elements such as text, photos, and containers. Moran et al.(Moran, Bernal-Cárdenas, Curcio, Bonett & Poshyvanyk, 2020) also establishes an approach to generate REMAUI code from UI images. It can be used in the creation of mobile device scripts for widget recognition. Chunyang Chen et. al. (Chen, Su, Meng, Xing & Liu, 2018) introduces a neural machine translator, which integrates recent developments in computer vision and machine translation and also translats UI pictures to GUI skeletons. Robotium (Robotium, 2014) is used in Android apps only and has minimal features for recording/replay functionality which is Android-limited. For use with native, hybrid, and mobile web applications, Appium (Appium.io, 2011) is an open-source test automation system. Using the WebDriver protocol, it powers iOS and Android apps (Hussain, Razak & Mkpojiogu, 2017).

The second level techniques are focused on the first level and focus more on the production and optimization of automated test cases. An approach intoduced by (Xue,

2020) that random testing uses a random approach to analyze and assess applications and produces test inputs, although randomly . A native Android test tool, Monkey (Android-Developers, 2015) sends GUI events selected and device events to be tested randomly. MobiGUITAR (Amalfitano, Fasolino, Tramontana, Ta & Memon, 2015) moves through models dynamically and then produces test cases. In order to obtain the static-dynamic relationship between GUI elements and events statically, AMOGA (Salihu, Ibrahim, Ahmed, Zamli & Usman, 2019) uses the source code and then scans applications dynamically and orderly to build the model. These models typically consist of a finite state machine.
Third stage techniques further incorporate the learning function. Traditionnelle test methods that employ model learning rely on conventional state models to produce test inputs on-the-fly and establish a learning mechanism that permits dynamic creation (Xue, 2020). By using the adapter module and the brain module, DroidBot (Yuanchun Li, Ziyue Yang, Yao Guo & Xiangqun Chen, 2017) dynamically updates the model. To train an end-to-end test model, Deep Learning-based testing uses deep neural network technology. Humanoid (Li, Yang, Guo & Chen, 2019) designs a deep neural network model by learning user interaction with apps to generate human-like inputs.

3. PROBLEM DEFINITION

In recent years, BDD has become an increasingly agile approach to production. The BDD development is based on test driven development Behavior Driven Development (TDD). BDD test scenario which is known as feature can be written in native language in Gherkin syntax without need to mention any selector or element Ids. In addition, implementation details can be hidden in automation layer. BDD test scenarios need step definition to perform test on the system (Figure 3.1). Step definitions connect Gherkin step to programming code and carry out the action that should be performed by a test scenario.

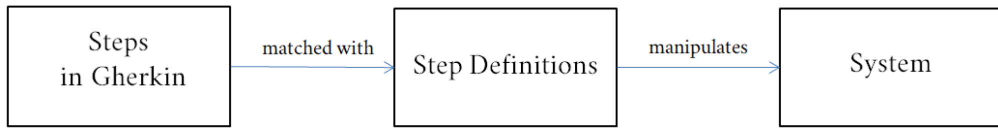


Figure 3.1 Implementation of BDD

Step definition can be written in any language such as Python. Figure 3.2 shows an example of step definition to perform "enter card number {cardNum}" step on Android in Python language.

```
#WHEN Steps
@When('enter card number {cardNum}')
def enter_card_number_cardNum(cardNum):
 WebDriverWait(driver,60).until(EC.element_to_be_clickable(By.ID,'edt_input')).send_keys(cardNum)
 WebDriverWait(driver,60).until(EC.element_to_be_clickable(By.XPATH,'Sanal Kart Oluştur')).click()
```

Figure 3.2 Step definition in Python

In this study, we aim to provide non-technical stakeholders an easy, readable and simple method for testing Android and iOS applications by using BDD.
The main problem of using BDD are implementation and maintaining of step definition to perform Appium actions on the emulator or real device. In addition, selector details and element Id and Xpath are required to find and access elements on the emulator/device. On the other hand, writing and maintaining step definition need

programming knowledge.
In this case, we are going to enable non-technical stakeholders to implement Gherkin test scenario in their native language as well as its step definitions. Therefor they will be able to create and modify step definitions by using Google Blockly and extract selector details by using C&R tool. In chapter4, we will show how users can create a BDD test and its step definitions very quickly. They are also able to use loops, if-then-else statements in step definition which is a new approach in automation test method.

4. APPROACH

We propose an integrated mobile testing tool which is a combination of 3 approaches: Google Blockly, BDD, and TestProject Capture&Replay (C&R) tool.
Our code-free mobile testing tool enables non-technical stakeholders to test mobile apps at a high level without any knowledge about application internal information. This tool also provides a flexible environment by using Google Blockly for the users to modify created test cases or reuse them in another test suite without worries about code syntax. Users can import a captured action list exported from TestProject C&R to our tool that provides required internal app information and action sequences to create BDD step definitions. Finally they can run test by writing BDD test scenario in Gherkin language which is similar to natural language and extremely readable. Testers can write BDD test scenarios in their native language. In this chapter We explain how do we apply these 3 approaches for our tool.

4.1 Google Blockly

To express coding ideas such as logical expressions, variables, loops, and other related notions through graphics, and more, Google Blockly uses interlocking graphical blocks. It helps users to apply programming principles without having to think about syntax.
In this phase, we implemented all actions by custom blocks in Google Blockly for testing Android/iOS applications. These custom blocks generate corresponding code in Python language to interact with the emulator or real device by using Appium. In this case, in our current version of the tool, we defined 28 new blocks that cover 25 actions supported by Appium on Android emulator/device and iOS simulator. All these blocks can be found under 3 categories in toolbox under Blockly tab in our tool.

In the first category of toolbox we have "Test suite" containing 4 blocks named “Test Suite”, “Test case”, "Run test case", and "Assert" that provide users making test cases and run them directly from the Blockly tab (Figure 4.1). Table 4.1 shows details of these blocks.

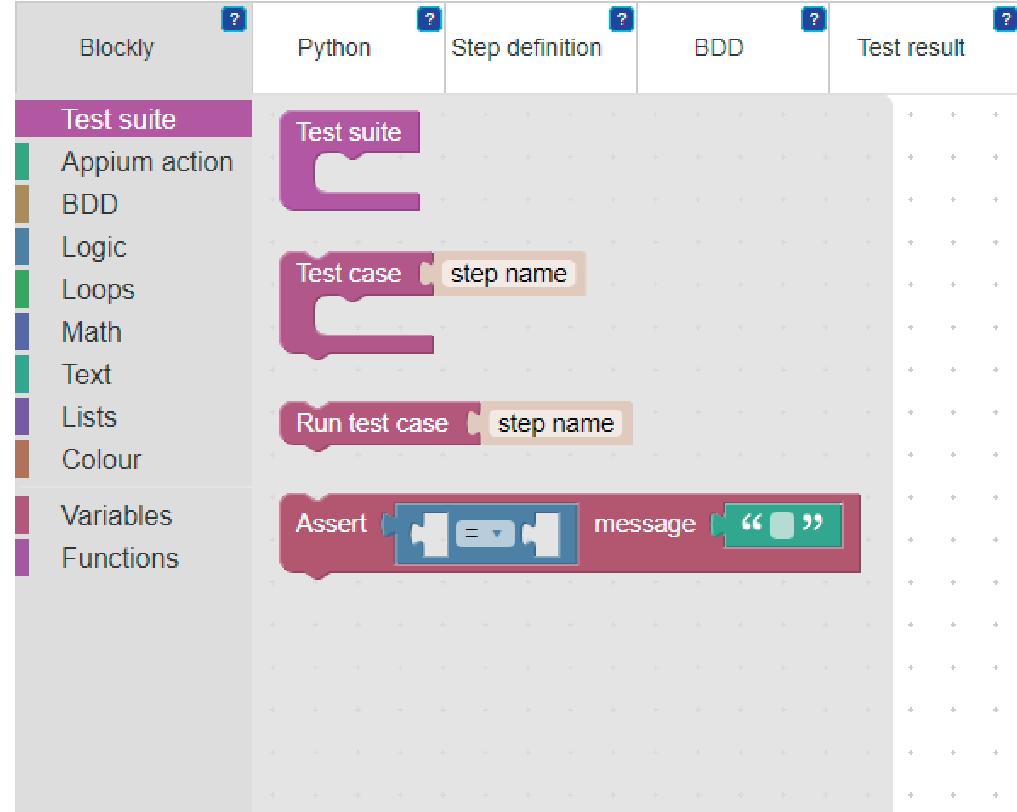


Figure 4.1 Test suite blocks

Table 4.1 Test suite blocks details

Block	Description
Test suite	Unittest class includes test cases
Test case / step name	Test case include actions
Run test case / step name	Call an existing test case
Assert / message	Assertion for test oracle

4.1.1 Test suite

Software programs are tested using test suites, which are a collection of "test case" blocks.

4.1.2 Test case

Software testing objectives are achieved by performing a single test using a test case that specifies the inputs, execution circumstances, testing process, and expected outcomes. This block contains actions and test oracles.

4.1.3 Run test case

Existing test cases can be called by their names with "Run test case" block. Test case names can be a string or defined by variable blocks.

4.1.4 Assert

Assertion is the validation step, determines whether the automated test case succeeded or not.

Second category in the toolbox belongs to "Appium action". We have 16 blocks to perform actions on Android emulator/device or iOS simulator (Figure 4.2). Table 4.2 shows details of these blocks.

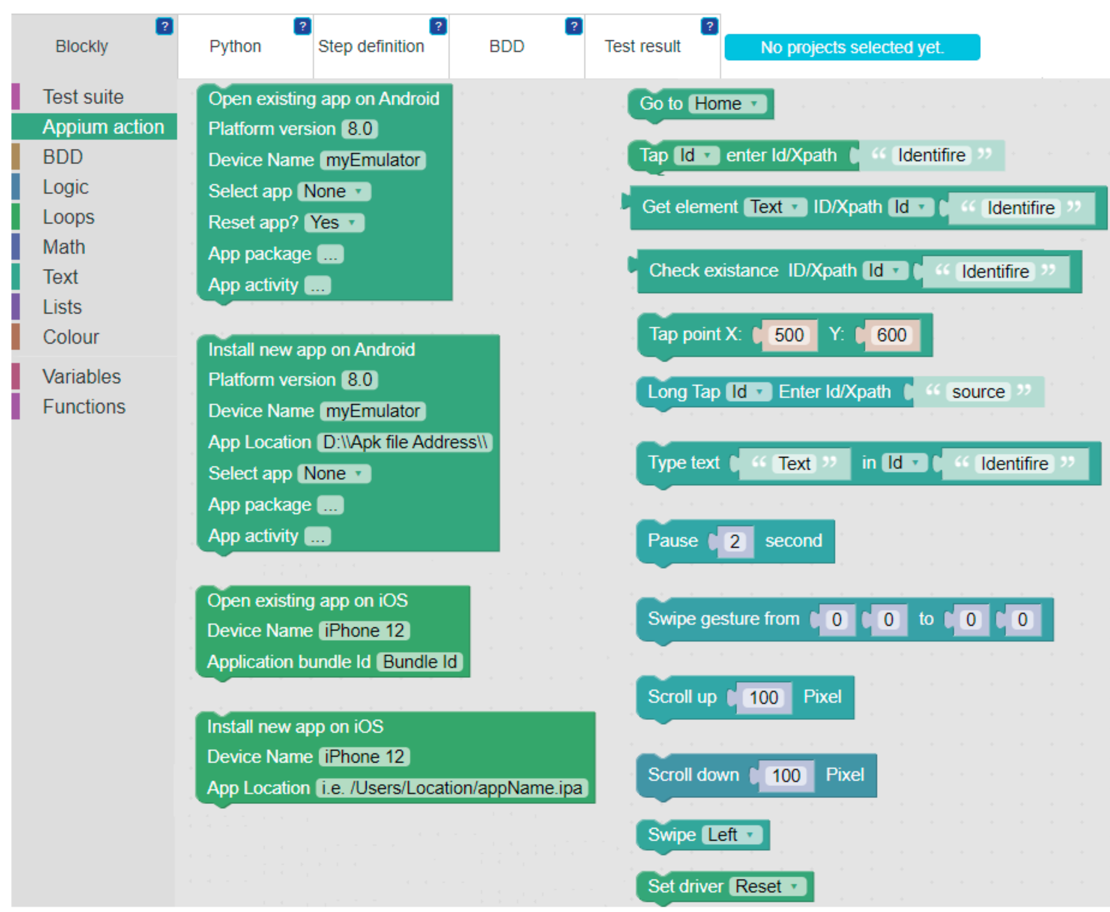


Figure 4.2 Appium action blocks

Table 4.2 Appium action blocks

Block	Description
Open existing app on Android Platform version 8.0 Device Name myEmulator Select app None Reset app? Yes App package ... App activity ...	Open/Reset an existing application on Android emulator/device
Install new app on Android Platform version 8.0 Device Name myEmulator App Location D:\\Apk file Address\\ Select app None App package ... App activity ...	Install a new application by using APK file on emulator or real device
Open existing app on iOS Device Name iPhone 12 Application bundle Id Bundle Id	Open/Reset an existing application on iOS
Install new app on iOS Device Name iPhone 12 App Location i.e. /Users/Location/appName.ipa	Install a new application by using APK file on iOS
Tap Id enter Id/Xpath " Identifire "	Tap an element on the Android/iOS screen
Tap point X: 500 Y: 600	Tap a specific point on the screen
Type text " Text " in Id " Identifire "	Type text on the Android/iOS screen
Pause 2 second	Pause a certain amount of time in second
Swipe gesture from 0 0 to 0 0	Swipe gesture from point A1 to A2
Swipe Left	Swipe Left/Right/Up/Down
Get element Text ID/Xpath Id " Identifire "	Get element text or visibility attribute
Check existance ID/Xpath Id " Identifire "	Check the visibility of element
Long Tap Id Enter Id/Xpath " source "	Long tap an element
Scroll down 100 Pixel	Scroll down in pixel
Scroll up 100 Pixel	Scroll up in pixel
Go to Home	Tap Back/Home button on emulator
Set driver Reset	Reset/Quit web driver

4.1.5 Open existing app on Android

User can open an existing application on the Android emulator or real device by using this block. User also can open an application with Reset status. For opening application successfully, specifying platform version, device name, App package name and activity are required.

4.1.6 Install new app on Android

User can install a new application by uploading Android APK file. APK file path should be entered in "App location".

4.1.7 Open existing app on iOS

User can open an existing application on the iOS simulator by using this block. Device name and application bundle id are required for this action.

4.1.8 Install new app on iOS

User can install a new application on simulator by uploading iOS IPA file. IPA file path should be entered in "App location".

4.1.9 Tap

To tap an element user can use Tap block. In this case, element's Id or Xpath for Android and iOS predicate for iOS should be entered.

4.1.10 Tap point

To tap a specific point on the screen user can use Tap point block. In this case, coordinates of point should be specified by X and Y.

4.1.11 Type text

To enter a string in a text box user can use Type text block. This block locates text box by its Id, Xpath or iOS predicate.

4.1.12 Get element

This block enables user to get an element's attributes such as element text or visibility. We need to enter element Id, Xpath or iOS predicate for locating.

4.1.13 Check existence

User can check an element existence without failing test. Normally, test case fails when ever an element can not be locate on the screen. Check existence block returns True/False value can be used in if-then-else clauses.

4.1.14 Pause

User will have more reliable test by using pause while the test cases fail because of synchronization issue. This block put a certain amount of time in second between other actions.

4.1.15 Swipe

User can swipe one page to the left, right, up and down. This block is configured with a size of pixel that perform one page swiping.

4.1.16 Swipe gesture

When user need more accurate swipe from a point to another point on the screen, can use "Swipe gesture" and specify the coordination of starting and ending point.

4.1.17 Scroll up

User can scroll up when the screen is scrollable and not single page.

4.1.18 Scroll down

User can scroll down when the screen is scrollable and not single page.

4.1.19 Go to

This block has 2 option to perform "Back" and "Home" button on Android emulator or real device.

4.1.20 Set driver

To quit form web driver or reset the driver in test case user can apply "Set driver" block with its 2 options: Reset/Quit.

BDD, the third category of toolbox, contains blocks for BDD steps: Given, When and Then together required blocks for parametrizing including Parameter and step name. (Figure 4.3). Table 4.3 shows more details of these blocks. We have 2 types of BDD steps for GIVEN, WHEN and THEN: (1)parametric, (2)non-parametric.

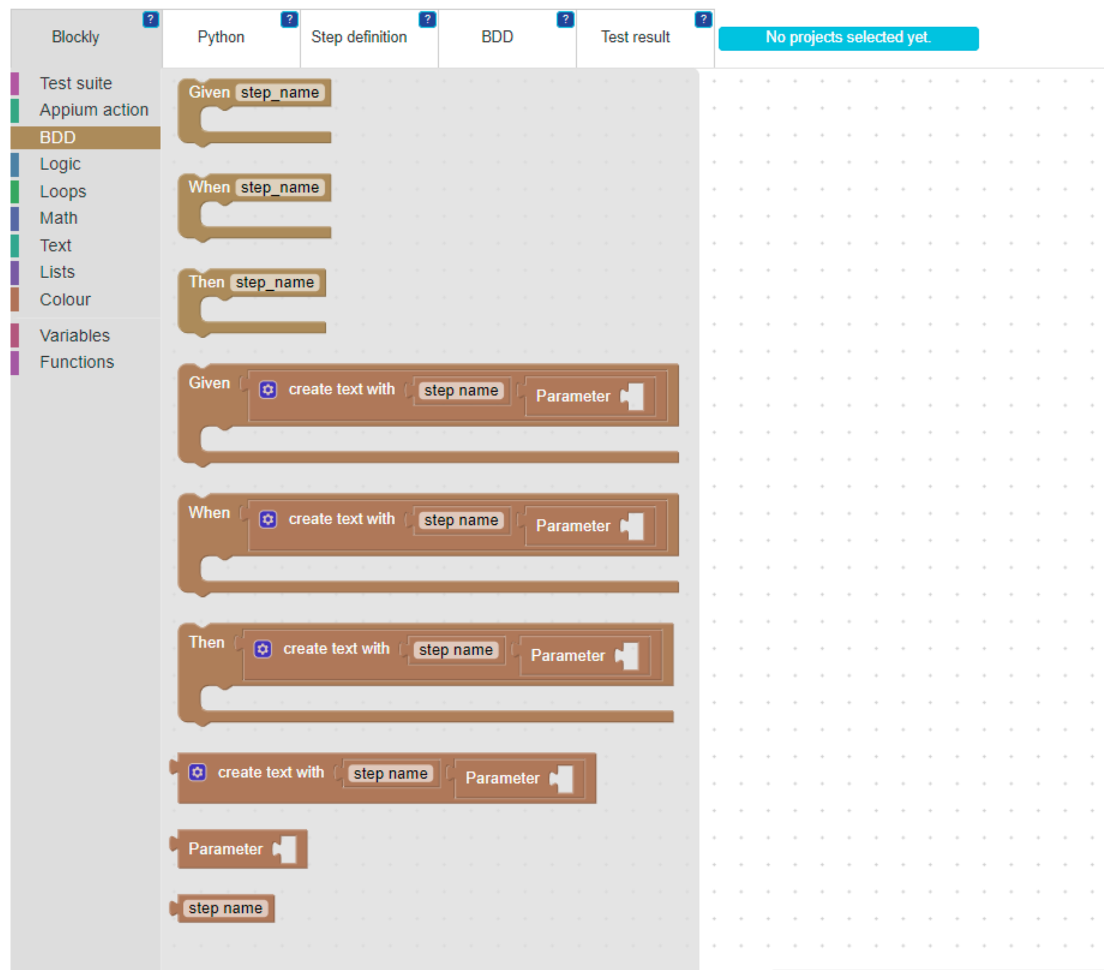


Figure 4.3 BDD step blocks

Table 4.3 BDD step block details

Block	Description
Given step_name	GIVEN step, step name can be a normal sentence
When step_name	WHEN step, step name can be a normal sentence
Then step_name	THEN step, step name can be a normal sentence
Given create text with step name Parameter	Parametric GIVEN step include text join to combine string and parameter as step name
When create text with step name Parameter	Parametric WHEN step include text join to combine string and parameter as step name
Then create text with step name Parameter	Parametric THEN step include text join to combine string and parameter as step name
Parameter	Convert Google Blockly variable to BDD parameter
step name	Return string for step name

4.1.21 Given

The first step in BDD test is "Given" which is an initializing steps. This block can contain initializing blocks such as opening or installing application and the step name should be string.

4.1.22 When

The second step in BDD test is "When". This block contains main body of test scenario and actions such as tapping, typing a text or scrolling that tester need to perform for testing purpose and the step name should be string.

4.1.23 Then

The last step in BDD test is "Then". This block contains evaluation block such as assertion blocks.

4.1.24 Parametric Given

Parametric block is used for given step whenever user defines an input. It is different than normal "Given" in the step name. In this case "Parametric Given" includes a parametric step name.

4.1.25 Parametric When

Same as "Parametric Given" whenever user defines an input, "Parametric When" can be applied to implement parametric steps. It is different than normal "When" in the step name. In this case, "Parametric When" includes a parametric step name.

4.1.26 Parametric Then

When we have parametric steps, therefor we need parametric test oracle. "Parametric Then" provides us test evaluation with existing inputs and variables.

4.1.27 Parameter

This block convert variable to BDD parameter to be used in BDD step name by putting variable inside curly brackets.

4.1.28 Step name

This block returns a simple string to be used in BDD step name.

In the following we explain the usage of blocks with some examples. We can create simple test cases by using Test suite blocks collection and Appium blocks. In this case we need to know selector such as element Id/Xpath. This information can be collected by some other tools manually such as UiAutomatorViewer for Android and Appium inspector for iOS.
For example, Figure 4.4 shows a test suite contains 2 test cases: (i) Reset Clock app (ii) Set an alarm. "Reset Clock app" opens Clock application in Android emulator and resets the application, then check the existence of an specific element on the screen for test oracle by using "Assert" block. "Set an alarm" calls "Rest Clock app", then continues the test and sets an alarm at 8:00. Figure 4.5 shows compiled Python code which is generated automatically.

Figure 4.6 shows a simple BDD "When" step that calculates "2+3" by tapping "2", "+", and "3" on the Android emulator with the step name of "add 2 and 3". These buttons are found by their Ids. Figure 4.7 shows compiled code corresponding to "When" step.

Another type of BDD blocks are parametric blocks. Despite non-parametric step where step name can be a string, parametric step name should include all inputs given by the user in a BDD parameters format. So, step name is a mixture of string

Figure 4.4 Test suite example

```python
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from appium.webdriver.common.mobileby import MobileBy
from appium import webdriver

class ModelDiscovery(unittest.TestCase):

  def test_events_my01resetclockapp(self):
    desired_caps = {}
    desired_caps["platformName"] = "Android"
    desired_caps["platformVersion"] = "8.0"
    desired_caps["deviceName"] = "myEmulator"
    desired_caps["appPackage"] = "com.google.android.deskclock"
    desired_caps["appActivity"] = "com.android.deskclock.DeskClock"
    desired_caps["noReset"] = False
    self.driver = webdriver.Remote('http://host.docker.internal:4723/wd/hub', desired_caps)
    assert self.driver.find_element_by_id('com.google.android.deskclock:id/fab').is_displayed()

  def test_events_my02setanalarm(self):
    self.test_events_my01resetclockapp()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
    'com.google.android.deskclock:id/fab'))).click()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((By.XPATH,
    "//*[@content-desc = '8']"))).click()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((By.XPATH,
    "//*[@content-desc = '0']"))).click()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
    'android:id/am_label'))).click()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
    'android:id/button1'))).click()
    WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
    "//android.widget.ImageButton[@content-desc = 'Collapse alarm']"))).click()
    assert self.driver.find_element_by_xpath("//android.widget.TextView[@content-desc = '8:00□AM']").is_displayed()


# ------START OF SCRIPT ------------#
if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(ModelDiscovery)
    HtmlTestRunner.HTMLTestRunner(combine_reports=True, report_name="MyReport", add_timestamp=False).run(suite)
```

Figure 4.5 Created Python code by Test suite blocks

Figure 4.6 BDD WHEN block

and parameters. In this case, We defined parametric "Given", "When", and "Then" blocks. Figure 4.8 shows a test oracle that is implemented by parametric "Then"

```
#WHEN Steps
@When('add 2 and 3')
def add_and_2556(self):
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'com.android.calculator2:id/digit_2'))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'com.android.calculator2:id/op_add'))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'com.android.calculator2:id/digit_3'))).click()
```

Figure 4.7 Created Python code for BDD WHEN block

block. It checks whether answer is equal to result and Figure 4.9 shows generated Python code.

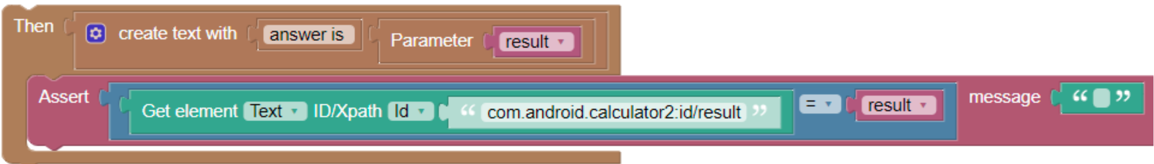


Figure 4.8 BDD THEN block

```
#THEN Steps
@Then('answer is {result}')
def answer_is_result6840(self,result):
  assert self.driver.find_element_by_id('com.android.calculator2:id/result').text == result
```

Figure 4.9 Created Python code for parametric BDD THEN block

Indeed high-level tests need the application's internal information such as Id and Xpath for locating elements on the emulator's screen. Basically, users either can make test cases manually and find required Id/Xpath of each element by using some tools such as UIAutomatorViewer or import captured test script provided by Testproject C&R tool that include all application internal information such as Id/Xpath.
In this case, Users record a test script in TestProject C&R, which is an online free web application, then download captured test document which is an "Excel" file format. Downloaded test document contains emulator and application information, actions title with all accessible Ids and Xpaths for each action, and coordination for gesture actions such as scrolling, swiping and tapping a specific point on the screen. Figure 4.10 shows TestProject C&R captured list and mirror view of emulator and application under test.

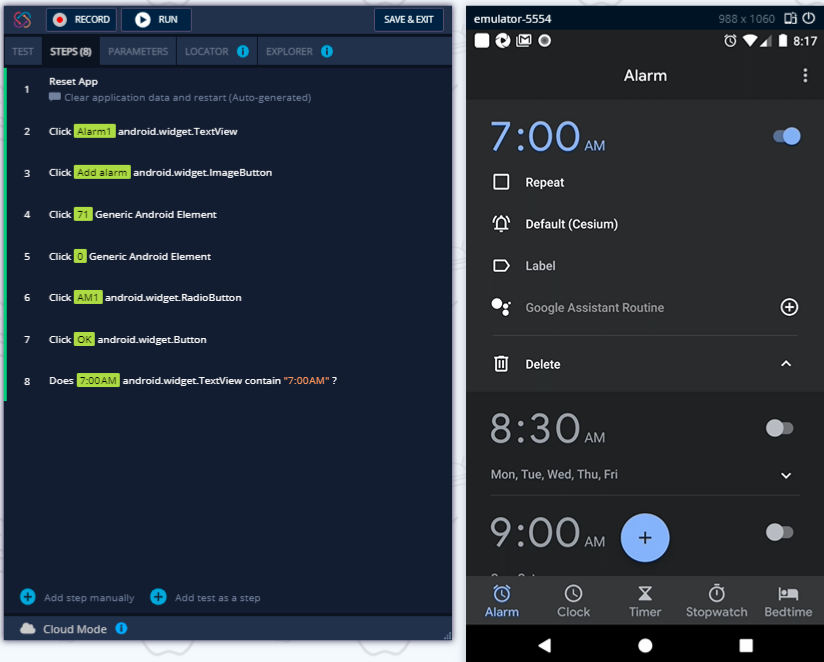


Figure 4.10 TestProject captured a list and mirror view of emulator

We describe work flow of creating BDD test case and step definitions from the scratch with "Clock" application on Android emulator. Table 4.4 shows test scenario of our example. In the first step user creates a test script using C&R tool and save its document to be used in next step 4.10.

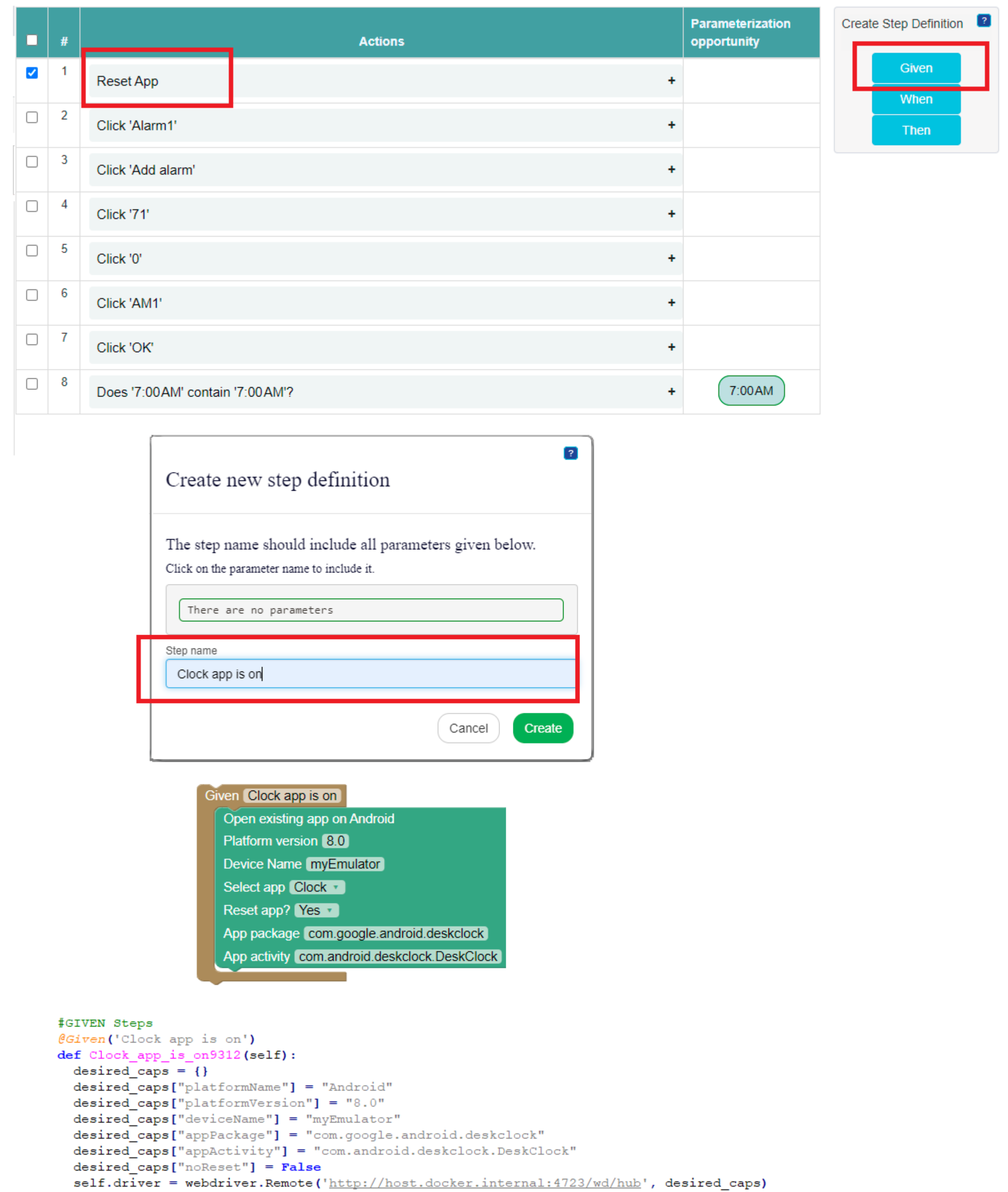
Table 4.4 Test scenario for Clock App

BDD step	**Step name**
Given	Clock app is on
When	set an alarm at 7:00
Then	Alarm 7:00 should be visible

By importing captured test document into our tool, users can select a sub sequence of actions and implement intended BDD steps. All selected actions will be mapped to its corresponding BDD blocks in Google Blockly and automatically saved as step_name.xml together with equivalent Python code as step_name.py in corresponding "Given", "When" and "Then" folder under application project directory.

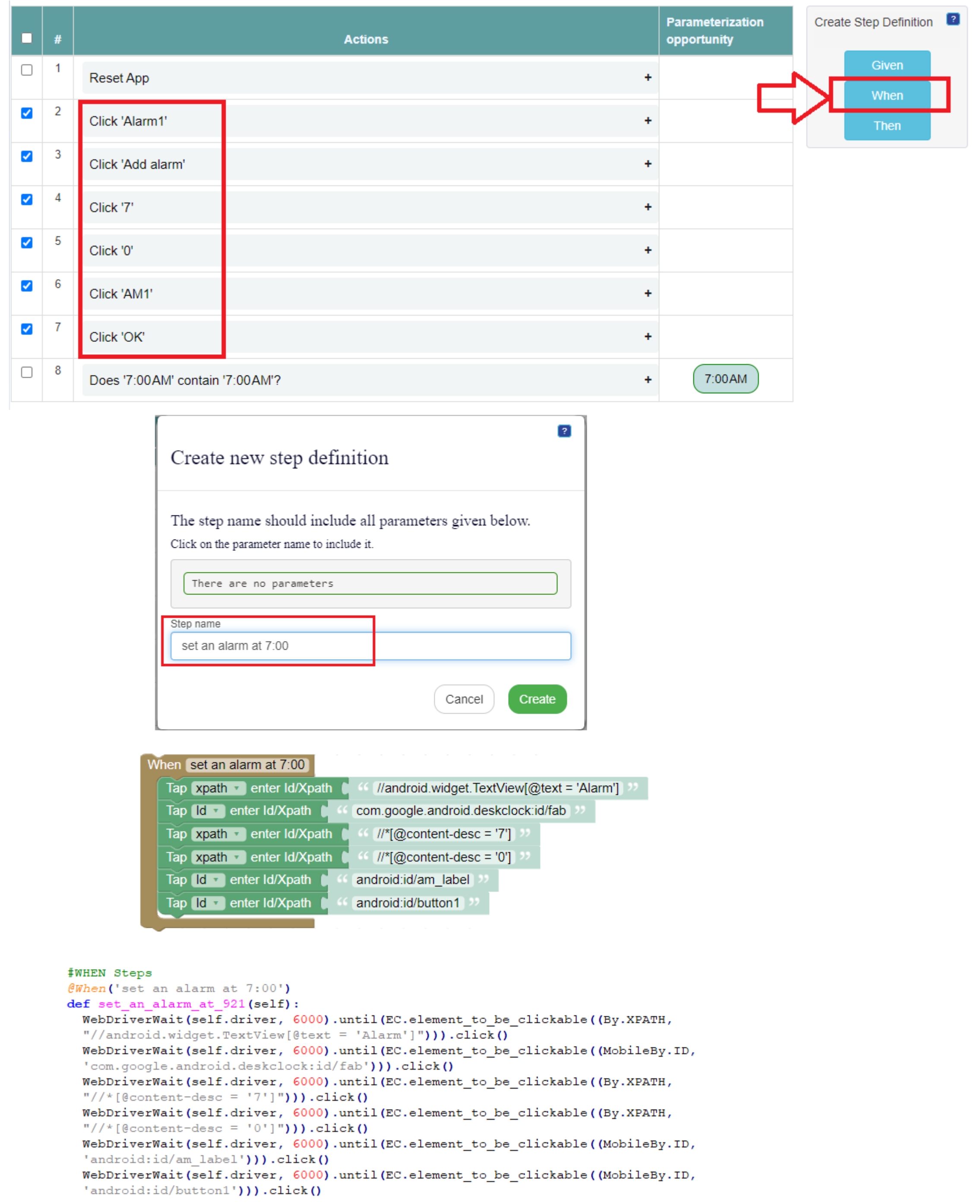
Figures 4.11, 4.12 and 4.13 show how user select the "Reset app" for "Given" step, "set an alarm at 7:00" for "When step" and "Alarm 7:00 should be visible" for "Then" step to create step definition.

Once users convert selected actions to blocks by selecting from the action list, they are able to modify and insert additional blocks such as logical expression, loops, pause, break, and quit driver based on their need in test case.
In addition, in a logical expression, users can check existence of elements to continue testing based on its existence. Figure 4.15 shows another example of modified "When" step in Android "Files" application. In this example, the existence of folders in the "Files" application will be checked (by locating "No items" message in the middle of home screen in "Files" app, Figure 4.15), if there are no folders then creates 3 folders by using a loop and a list of folder names: "Photo", "Music", and "Movie".




```
#GIVEN Steps
@Given('Clock app is on')
def Clock_app_is_on9312(self):
  desired_caps = {}
  desired_caps["platformName"] = "Android"
  desired_caps["platformVersion"] = "8.0"
  desired_caps["deviceName"] = "myEmulator"
  desired_caps["appPackage"] = "com.google.android.deskclock"
  desired_caps["appActivity"] = "com.android.deskclock.DeskClock"
  desired_caps["noReset"] = False
  self.driver = webdriver.Remote('http://host.docker.internal:4723/wd/hub', desired_caps)
```

Figure 4.11 Creating "Given" step definition from captured script




```
#WHEN Steps
@When('set an alarm at 7:00')
def set_an_alarm_at_921(self):
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((By.XPATH,
  "//android.widget.TextView[@text = 'Alarm']"))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'com.google.android.deskclock:id/fab'))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((By.XPATH,
  "//*[@content-desc = '7']"))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((By.XPATH,
  "//*[@content-desc = '0']"))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'android:id/am_label'))).click()
  WebDriverWait(self.driver, 6000).until(EC.element_to_be_clickable((MobileBy.ID,
  'android:id/button1'))).click()
```

Figure 4.12 Creating "When" step definition from captured script

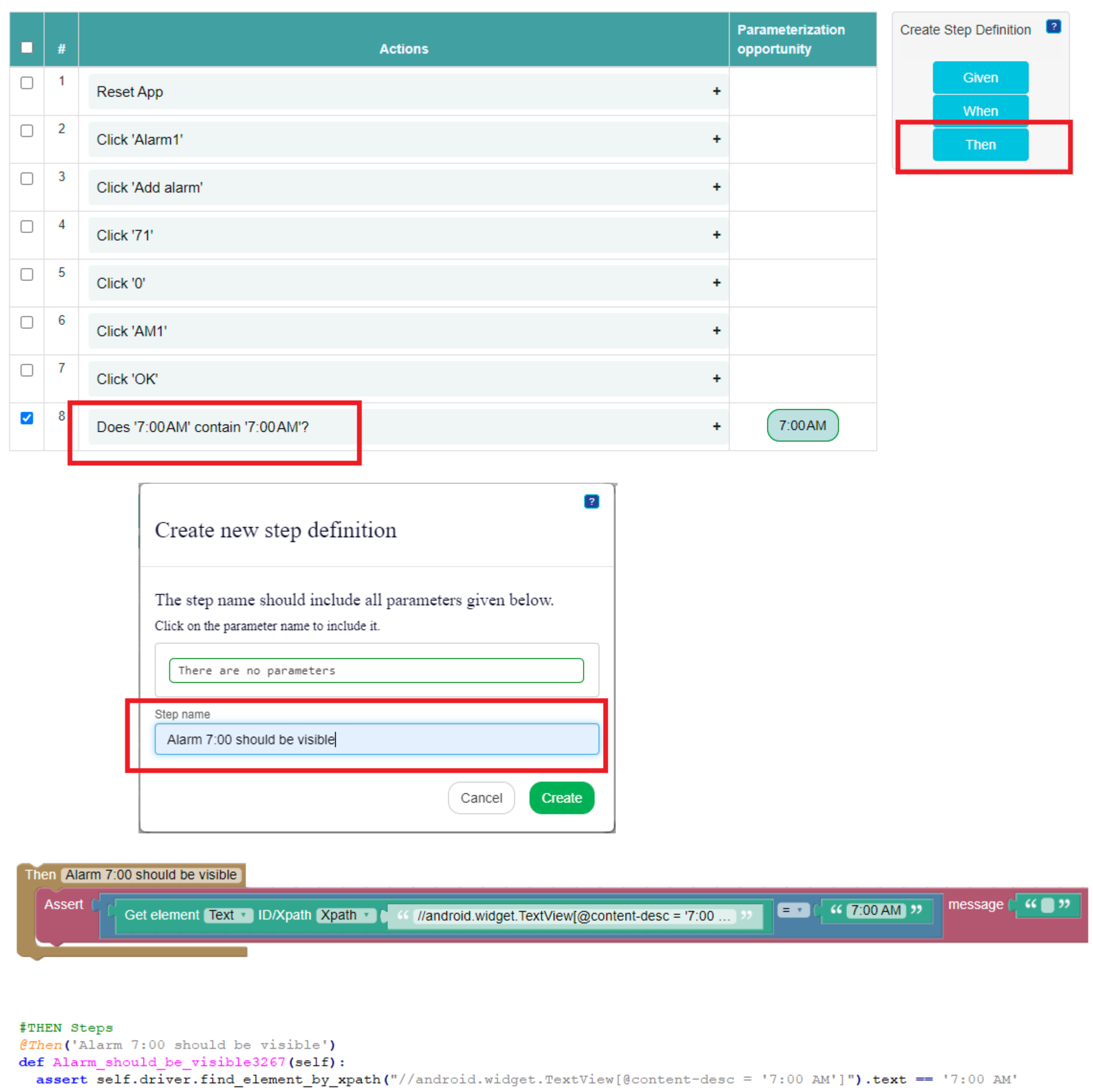


Figure 4.13 Creating "Then" step definition from captured script

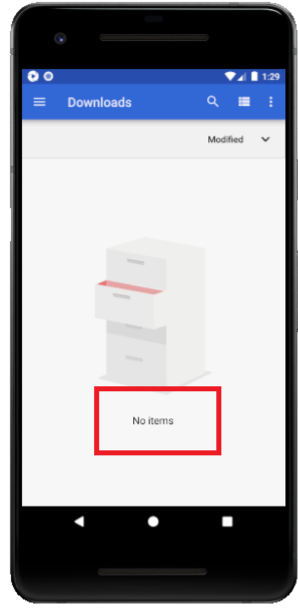


Figure 4.14 Files app home screen checking by modified "When" step

Figure 4.15 Modified "When" step definition

When users implemented step definition, all steps are saved in a single file named "step-definition.py" and it contains all created steps for the particular application under the test. Now, users need to write test scenario in Gherkin language.
we used a web-based Gherkin editor named ACE (Editor, 2010) that provides us syntax highlighting, real-time parsing and auto completion of step names (Figure 4.16). It also support "And" and "But" key words in writing scenarios (Figure 4.17). Written scenarios will be saved with the ".feature" extensions and they are executable.

```
1  Feature: Clock app demo
2    Scenario: Set new alaram at morning
3      Given Clock app is on
4      When set an alarm at 7:00
5      The
     Then Alarm 7:00 should be visible  Step
```

Figure 4.16 BDD test scenario to set an alarm at 7:00 AM

```
1  Feature: Todolist app
2    Scenario Outline: Create a new todo and remove existing one
3
4      Given App is on
5      When create a todo with the name <todo_name1>
6       And remove todo <todo_name2>
7      Then todo <todo_name1> is displayed
8       But todo <todo_name2> is not displayed
9
10     Examples: todo
11       | todo_name1 | todo_name2 |
12       | bar        | foo        |
```

Figure 4.17 BDD test scenario using "And" and "But" key words

Apart from the emulator being up and running, Appium should be up and running to run the BDD test scenarios and perform actions on the emulator.
Once test is completed, the test report will be displayed in the report tab in our tool (Figure 4.18) in HTML format which is created by "behave-html-formatter" Python library.

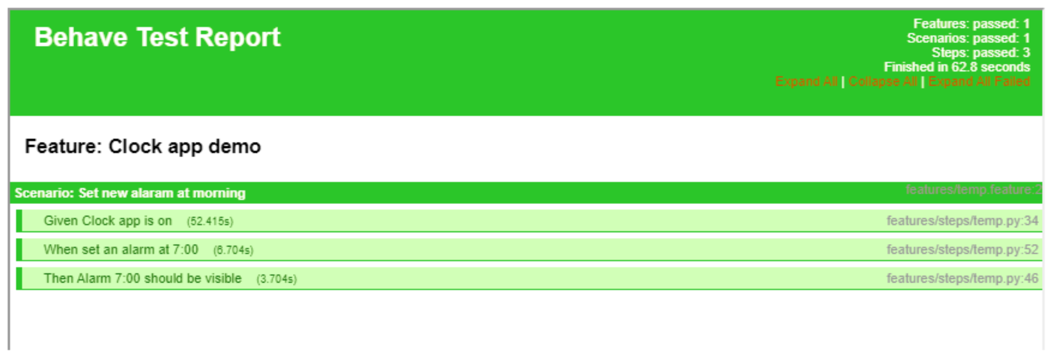


Figure 4.18 HTML BDD test report

4.4.1 Creating Test Case

We used Google Blockly source code which is a free and client side in JavaScript language and can be run in the browser. The source code includes some basic blocks such as loops, logic, strings and variables. We defined 28 new blocks to perform BDD test and Appium actions in JavaScript using Google Blockly standard. Also, we defined the equivalent Python code corresponding to each block in JavaScript language. So, compilation of block can be run in the client side. On the other hand, regarding to captured script, user can import created script by TestProject which is an "Excel" file to our tool and it will be store in "TestProject" folder under project directory locally. By opening imported test document "Excel" file, all recorded actions will be shown in the "Step Definition" tab as an HTML table together with all information that user needs for creating test cases (Figure 4.19). We linked each action to corresponding block by parsing action titles and their selector details. Each action selected by user, depends on its title, is mapped to the corresponding block in Google Blockly and all required information such as Id/Xpath, coordination, time and pixel will be inserted to block in Blockly work space. During creation of BDD step from action list, users are able to select sub sequence of actions or all actions to create BDD steps.

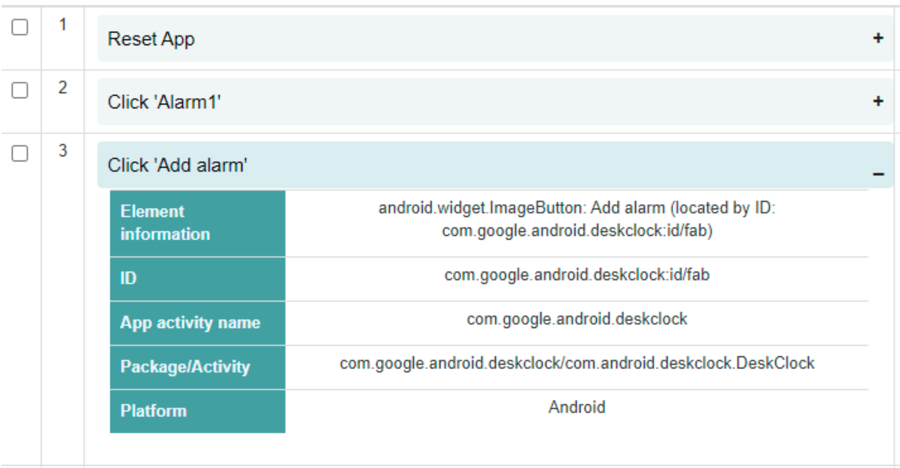


Figure 4.19 Action information

4.4.2 Parameters

We offered a mechanism that enables users to define parameters instead of constant input in BDD step definition. In this case, we analyse captured script to find inputs with the possibility of parametrizing, then we show that input in the action list as an HTML button in the "Parameterization opportunity" column (Figure 4.20). By clicking that button, user can define a name to create new parameter (Figure 4.21), change or remove an existing parameter of that action (Figure 4.22). All defined parameters are store in the action list table temporary and will be removed by closing the table.

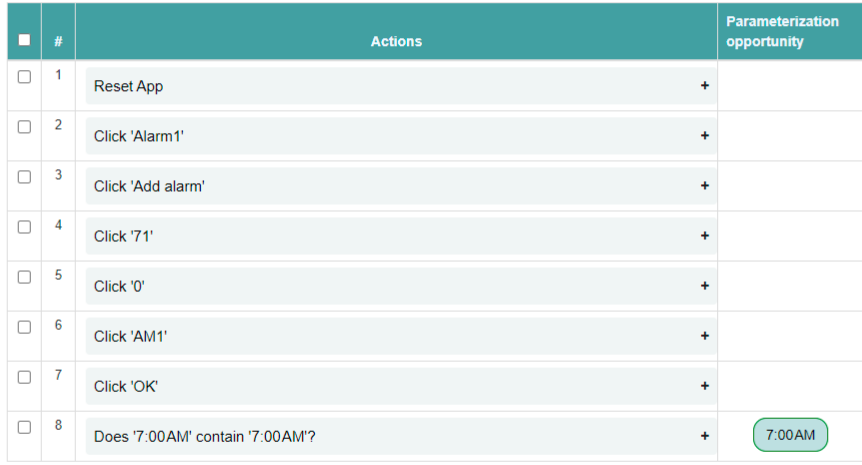


Figure 4.20 Captured test script with the parameterization opportunity

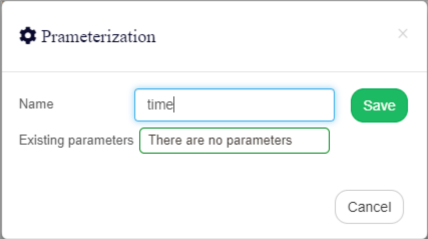


Figure 4.21 Parameterization form, when we have not defined yet

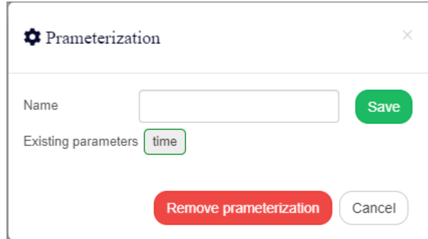


Figure 4.22 Parameterization form, when the parameter is defined

4.4.3 Saving Test Case

We implemented a file server using Node.js (Node.js, 2009) which is a back-end, cross-platform and open-source pqaltform that runs JavaScript code outside a web browser. File server works with the local memory in a fixed folder structure. In this case, each project includes 5 sub directories: "BDD", "TestProject", "Given", "When", and "Then"). Whenever user creates a BDD steps, BDD feature or import a captured script, it is stored in the relevant directory. All "xml" files contain Google Blockly blocks and each "xml" file has a pair of "py" file that contains complied step definition in Python language. To integrate all steps, we created a single step definition file in the "BDD" directory include all Given-When-Then steps and will be updated automatically after each test case creation, modification or deletion. Also, We apply this single step definition file for auto completion in BDD editor for writing BDD test scenarios.

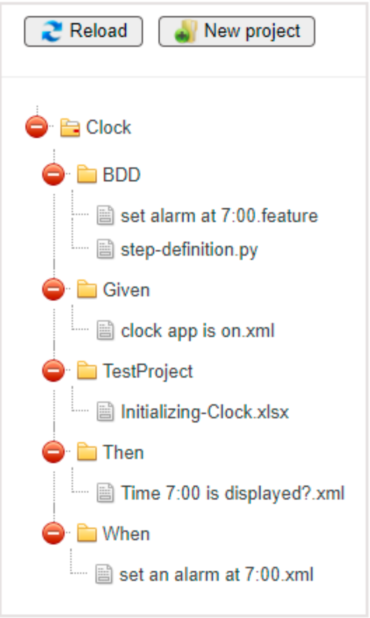


Figure 4.23 File explorer

4.4.4 Running Test Case

BDD test scenario can be written simply since we provided auto completion that suggest users existing BDD steps. We implemented a local Python server that is able to run BDD test scenarios. When user runs test, web browser sends "step-definition.py" and feature file to the local server and waits for the response. The

local server runs test cases, interacts with Appium and and performs test cases on the emulator, then generates test result in HTML format can be shown in "Test Result" tab in our tool.

5. EMPIRICAL EVALUATION

5.1 Experiment

In this chapter, we describe the experimental design that we carried out and the related empirical results. We also demonstrate that our tool is extremely easy to use even for non-technical users without programming or software testing knowledge and they are able to make BDD test cases and its step definition by themselves.
To evaluate our tool, we held "PURE Project" course at Sabanci University in fall semester 2020. In our user study, 38 students including 58% freshman students and 42% sophomore students volunteered to participate with no payment or forcing them (Figure 5.1). Among all participants, 96% had no experiences of testing software systems before (Figure 5.2).

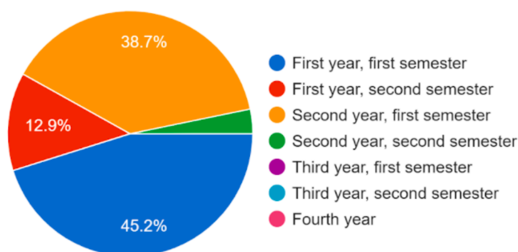


Figure 5.1 Student’s academic status

In the PURE Project, We taught them all concepts that we used in our tool, including Android Studio, Google few homeworks and asked them to record completion time for each task. For our user study, We selected 3 subject applications Calculator, Clock and Files, and defined tasks for each application and assigned them as their last homework with a one-week deadline. Table 5.1 shows applications and test scenarios for our user study experiment.

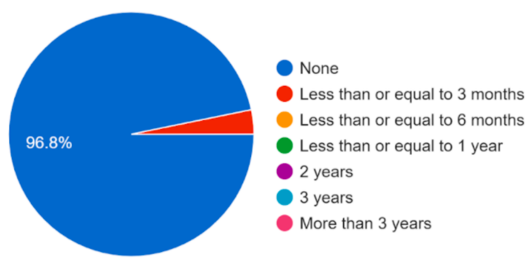


Figure 5.2 Student's experiences in software testing

Table 5.1 Test scenario for 3 subject applications

Application	Scenario
Calculator	1. Initialize the app 2. Compute (2+3)*(5+6) 3. Compute ((2+3)*(5+6))-15
Clock	1. Initialize the app 2. Set an alarm at 8:00 AM 3. Set a recurring alarm on every weekday at 8:00 AM
Files	1. Initialize the app 2. Remove all existing folder if any 3. Create folders "foo", "bar" and "zoo" by using example eable and scenario outline 4. Create 3 folders with names "Music", "Photo" and "Movie" by using loop

Considering the students were freshmen, our software was under development and was using heavy dependencies, we decided to use Docker to containerize our software. We provided the students two files, namely "DockerFile" and "docker-compose.yml", which they use to create a Docker container. The source code could be downloaded from the GitHub repository during the creation of the container. This container contains two HTTP servers; one for the website, and the other one for the web API. Each time the container is run, it automatically updates itself using the GitHub repository.

We asked participants to use all 3 technologies including Testproject tool, Google Blockly and BDD. We divided each task into 3 sub tasks: 1) Testproject 2) Blockly 3) BDD, and evaluated them separately to evaluate the submitted test suites by the participants carefully.

Since no similar tool exists, we measured 2 items after completing the user study to evaluate the usability of our tool, number of tasks that were successfully completed and the time duration that a participant spent to finish each task. According to the result that we extracted from submitted files by participants, we observed that participants have done task number 1 which is an initializing scenario in 10 minutes on average. Regarding the second task of each application which has medium difficulties, participants spent 25 to 30 minutes to get complete. Task number 3 is designed more challenging and the result was what we have expected, and they needed about 40 minutes to complete Clock and Files application and 22 minutes to complete Calculator 5.3.

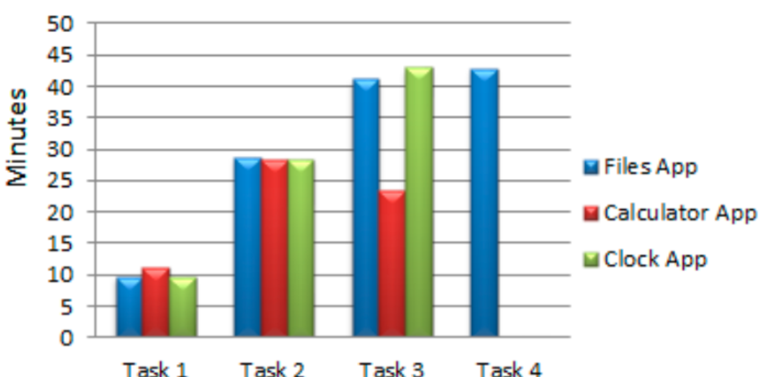


Figure 5.3 Time spent for each tasks

By grading submitted test cases by 26 participants, we found that more 20 student got high range score between 85 and 100 (Figure 5.4).

We provided a survey at the end of the user research. Participants were polled on their thoughts on the usefulness of our product. According to the survey, most of students believe that working with Capture&Replay tool and Google Blockly are extremely easy and they found BDD and applying Blockly for implementing test cases normal.

For each application we also asked participants 4 questions: 1) Deciding and creating the capture-and-replay scripts to be used 2) Using the recorded scripts to create step definitions 3) Modifying the Blockly blocks 4) Implementing the BDD scenarios.

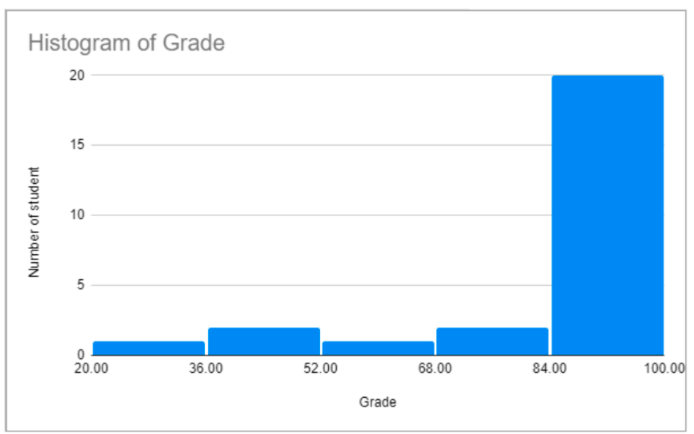


Figure 5.4 Submitted test cases scores

Figure 5.5 shows that most of them found Blockly, Capture&Replay and BDD easy and more than 10 students believe using Google Blockly for developing test cases for Calculator app is normal.

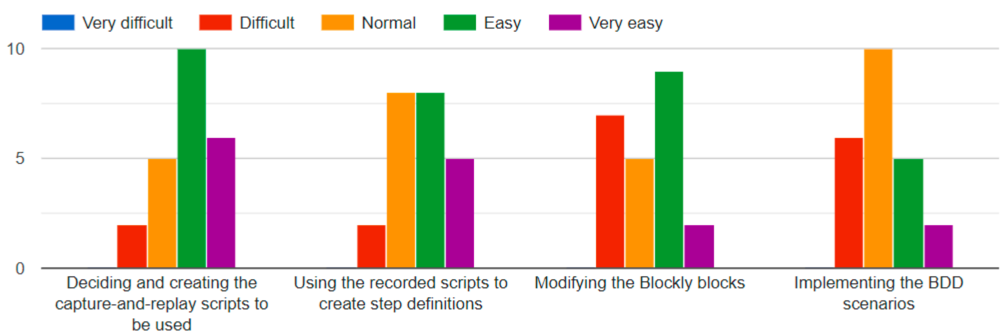


Figure 5.5 Difficulty level of the Calculator scenarios

Regarding Clock application, participants believed different that Calculator app. They said difficulty level of Clock app testing is normal and only one student believes modifying blocks is very difficult (Figure 5.6).

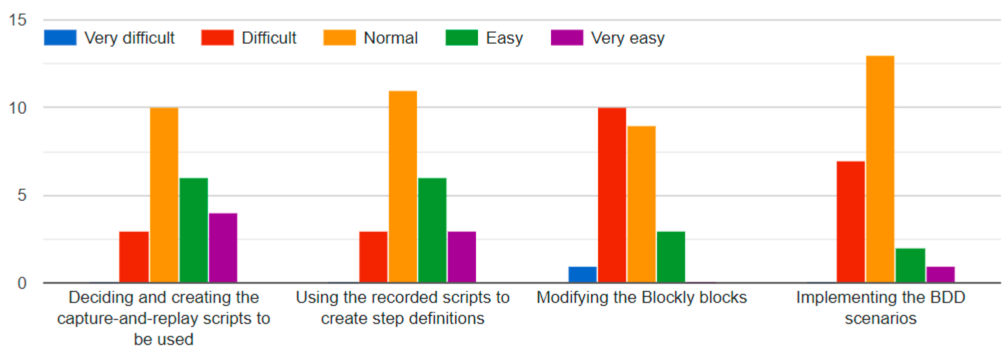


Figure 5.6 Difficulty level of the Clock scenarios

Finally, Files app survey shows that it challenged students, since Files app scenar-

ios had complex structure including loop, list and conditional statements, 50% of participants found it difficult and very difficult for modifying step definitions and implementing BDD scenarios (Figure 5.7).

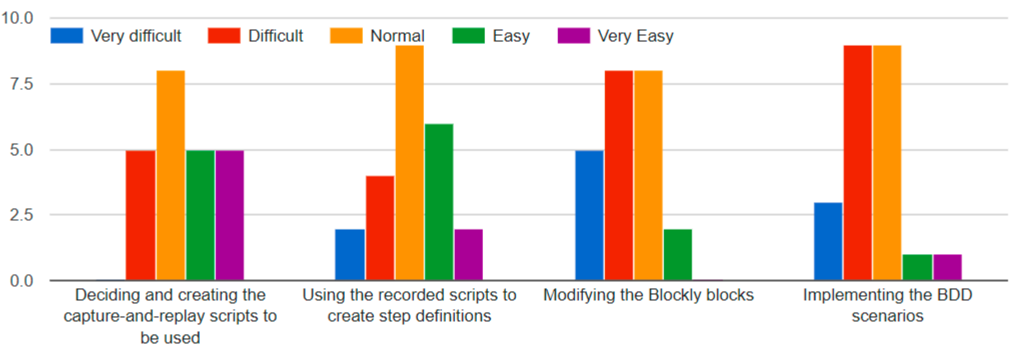


Figure 5.7 Difficulty level of the Files scenarios

6. CONCLUSION

We proposed a mobile testing approach, namely TestItAlready, with a new idea to provide non-technical stakeholders an ease of use BDD framework that enables user to create BDD features and step definitions by themselves with the help of Google Blockly and Capture&Reply tool. Users are able to create, modify and reuse BDD step definitions. Using Google Blockly for implementing BDD step definitions made our tool possible to use complex structure including such as logical conditional structures and loops in test cases and users are able to define parametric step definitions. These opportunities make TestItAlready so flexible.

7. FUTURE WORK

We intend to plan in the future to get this definitions and automatically reason about those natural language description to execute test cases other than relying on block. In addition, we want to Add new blocks which can given a text description and interact with UI elements automatically.
To avoid having same step definition we want to make a mechanism that analyse and compare similar step definitions and merge them as one step definition.
In the case of locating element by their Xpath and Id, we plan to make our tool enable to show a mirror view of device/emulator screen and user can select elements to get information such as coordination, Xpath and Id.

BIBLIOGRAPHY

Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2015). Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, *32*(5), 53–59.

Android-Developers (2009). Espresso, https://developer.android.com/training/testing/espresso.

Android-Developers (2015). Ui/application exerciser monkey.

Appetizer.io (2009). appetizer/replaykit, https://github.com/appetizerio/replaykit.

Appium.io (2011). Appium: Open source test automation framework, http://appium.io/.

Apple-Developer (2011). Apple developer documentation, https://developer.apple.com/documentation/xctest.

Ardito, L., Coppola, R., Morisio, M., & Torchiano, M. (2019). Espresso vs. eyeautomate: An experiment for the comparison of two generations of android gui testing. In *Proceedings of the Evaluation and Assessment on Software Engineering* (pp. 13–22).

Bathla, R. & Bathla, S. (2009). Innovative approaches of automated tools in software testing and current technology as compared to manual testing. *Global Journal of Enterprise Information System*, *1*(1), 119–131.

Chen, C., Su, T., Meng, G., Xing, Z., & Liu, Y. (2018). From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation, 665–676.

Choudhary, S. R., Gorla, A., & Orso, A. (2015). Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 429–440).

Culic, I., Radovici, A., & Vasilescu, L. M. (2015). Auto-generating google blockly visual programming elements for peripheral hardware. In *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*, (pp. 94–98). IEEE.

Di Martino, S., Fasolino, A. R., Starace, L. L. L., & Tramontana, P. (2020). Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. *Software Testing, Verification and Reliability*, e1754.

Diepenbeck, M., Soeken, M., Große, D., & Drechsler, R. (2012). Behavior driven development for circuit design and verification. In *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, (pp. 9–16). IEEE.

Editor, A. C. (2010). Ace - the high performance code editor for the web, https://ace.c9.io/.

EyeAutomate (2009). Eyeautomate, https://eyeautomate.com/eyeautomate/.

Eyestudio (2009). Eyestudio, https://eyeautomate.com/eyestudio/.

Gomez, L., Neamtiu, I., Azim, T., & Millstein, T. (2013). Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*, (pp. 72–81). IEEE.

Guo, J., Li, S., Lou, J.-G., Yang, Z., & Liu, T. (2019). Sara: self-replay augmented record and replay for android in industrial cases. In *Proceedings of the 28th*

ACM SIGSOFT International Symposium on Software Testing and Analysis, (pp. 90–100).

Halpern, M., Zhu, Y., Peri, R., & Reddi, V. J. (2015). Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (pp. 215–224). IEEE.

Hu, Y., Azim, T., & Neamtiu, I. (2015). Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (pp. 349–366).

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.

Li, Y., Yang, Z., Guo, Y., & Chen, X. (2019). Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 1070–1073).

Lübke, D. & van Lessen, T. (2016). Modeling test cases in bpmn for behavior-driven development. *IEEE software*, *33*(5), 15–21.

Mariani, L., Pezzè, M., Riganelli, O., & Santoro, M. (2014). Automatic testing of gui-based applications. *Software Testing, Verification and Reliability*, *24*(5), 341–366.

Marron, A., Weiss, G., & Wiener, G. (2012). A decentralized approach for programming interactive applications with javascript and blockly. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions* (pp. 59–70).

Méndez Porras, A., Quesada López, C., & Jenkins Coronas, M. (2015). Automated testing of mobile applications: A systematic map and review.

Moran, K., Bernal-Cárdenas, C., Curcio, M., Bonett, R., & Poshyvanyk, D. (2020). Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, *46*(2), 196–221.

Moran, K., Bonett, R., Bernal-Cárdenas, C., Otten, B., Park, D., & Poshyvanyk, D. (2017). On-device bug reporting for android applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, (pp. 215–216). IEEE.

Nguyen, T. A. & Csallner, C. (2015). Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 248–259).

Node.js (2009). Open-source, back-end javascript runtime environment.

North, D. (2012). Jbehave. a framework for behaviour driven development (bdd).

Rahman, M. & Gao, J. (2015). A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *2015 IEEE Symposium on Service-Oriented System Engineering*, (pp. 321–325). IEEE.

Robotium (2014). Robotium: Open source test framework, https://github.com/robotiumtech/robotium.

Saad, N. H. & Bakar, N. S. A. A. (2014). Automated testing tools for mobile applications. In *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, (pp. 1–5). IEEE.

Sahin, O., Aliyeva, A., Mathavan, H., Coskun, A., & Egele, M. (2019). Late breaking results: Towards practical record and replay for mobile applications. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, (pp. 1–2). IEEE.

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*, *7*, 17158–17173.

Seraj, M., Katterfeldt, E.-S., Bub, K., Autexier, S., & Drechsler, R. (2019). Scratch and google blockly: How girls' programming skills and attitudes are influenced. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, (pp. 1–10).

Trower, J. & Gray, J. (2015). Blockly language creation and applications: Visual programming for media computation and bluetooth robotics control. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, (pp.5̃)., New York, NY, USA. Association for Computing Machinery.

Wang, Y. & Wagner, S. (2018). Combining stpa and bdd for safety analysis and verification in agile development: A controlled experiment. In *International Conference on Agile Software Development*, (pp. 37–53). Springer.

Xue, F. (2020). Automated mobile apps testing from visual perspective. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (pp. 577–581).

Yeh, T., Chang, T.-H., & Miller, R. C. (2009). Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, (pp. 183–192).

Yu, S., Fang, C., Feng, Y., Zhao, W., & Chen, Z. (2019). Lirat: layout and image recognition driving automated mobile testing of cross-platform. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 1066–1069). IEEE.

Yuanchun Li, Ziyue Yang, Yao Guo, & Xiangqun Chen (2017). 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).