Testing Framework for Blockly

Team TBD

Sarah Nicholson, Peyton Hartzell, Mykal Burris, and Kelly Ding

CSCI 362 - Software Engineering - Fall 2018

Table of Contents

Chapter 1 - Introduction

What is Blockly?

Blockly is a library created by Google that adds a visual code editor to both web and mobile apps. The Blockly editor uses interlocking, graphical blocks, similar to Scratch, to represent code concepts like loops, logical expressions, variables, and more. It allows users to apply programming principles without having to worry about minute details such as syntax. We chose to work with Blockly because we wanted to get a better grasp on JavaScript, and one of our team members is very familiar with it. We are also interested in the education of children in computer science early in their school careers, and Blockly helps to do that.

Blockly is for developers while Blockly apps are for students. Blockly is an intuitive, visual way to build code. Blockly is 100% client side, requiring no support from the server and there are no 3rd party dependencies. It is a ready made UI for creating a visual language that emits syntactically correct user-generated code which can then be exported to many programming languages including:

- JavaScript
- Python
- PHP
- Lua
- Dart

A few of Blockly's biggest strengths are as followed:

- Exportable code: Users can extract their block-based programs to common programming languages for a smooth transition to text-based programming.
- Open source: Everything about Blockly is open open source.
- Extensible: One can tweak Blockly to fit one’s needs by adding custom blocks for an API or removing unneeded blocks and functionality.
- Highly capable: One can implement complex programming tasks using Blockly.
- International: Blockly has been translated to 40+ languages.

Getting Started

The first thing we did was to navigate to the “Getting Started” page Google for Education provided for Blockly. We then downloaded the source code from Google’s Github and verified that blocks could be dragged around by running the following command in Blockly’s root directory:

firefox demos/fixed/index.html

Next, in order to run the existing unit tests Blockly provides, we had to download the closure-library from Google’s Github. The reason why Closure is needed is because we are going to run and build the uncompressed version of Blockly and the dependency of the Closure Library is needed. Once we downloaded the Closure files, we placed them next to Blockly’s root directory and renamed the directory closure-library. The following is the directory structure we ended up with and what is required to run the tests:

- google
    - blockly
        - blocks
        - core
        - demos
        - generators
        - media
        - msg
        - tests
    - closure-library

Blockly was then able to work in uncompressed mode. Run it by entering the following command:

firefox tests/playground.html

Running Existing Tests

There are two sets of existing unit tests: JavaScript tests and block generator tests.

JS Tests

The JS tests confirm the operation of internal JavaScript functions in Blockly's core. To run this test, we went into the root directory of Blockly and run the following command:

firefox tests/jsunit/index.html

When we ran it, all tests passed and gave us the following output:

Unit Tests for Blockly [PASSED]
/home/USERNAME/Documents/blockly-master/tests/jsunit/index.html
267 of 267 tests run in 2415.495ms.
267 passed, 0 failed.
9 ms/test. 238 files loaded.
.
23:36:04.296 Start

Block Generator Tests

Along with the tests to confirm the operation of functions in Blockly's core, each block has its own unit tests. These tests verify that blocks generate code that functions as intended.

Again, to run these tests, we went into the root directory of Blockly and this time, ran the command:

firefox tests/generators/index.html

Running this command brings up a testing window for the web app to validate that the blocks generate code correctly, corresponding with which programming language you choose. Once the window was up, we chose what part of the system to test from the drop-down menu, and clicked "Load". The blocks then appeared in the workspace and XML code corresponding with the blocks appeared in the textbox.

We then checked each programming language in the workspace along with the system we were testing to make sure that syntactically correct code was being outputted. We first ran the generated JavaScript code and in order to make sure it was correct, copied

and pasted it in a JavaScript console which outputted “OK” meaning that it passed. Next, we clicked the button to generate Python code and then copied and pasted the code in a Python interpreter which outputted “OK,” meaning it passed. After that, we clicked on “PHP” which generated the PHP code in the textbox. We then copied and pasted the generated code into a PHP interpreter and it outputted “OK,” meaning this one passed as well. Then, we clicked the “Lua” button to generate the Lua code in the textbox. We copied and pasted the Lua code into a Lua interpreter and it outputted “OK,” showing that it passed. The last language supported by Blockly that we tested is Dart. To test it, we clicked on the “Dart” button which generated the code. We then copied and pasted it in a Dart interpreter which outputted “OK,” meaning that the test passed.

Experience So Far

We at first attempted to use the web app version of Blockly, which utilizes JavaScript, but we started having troubles with that and switched to the Android mobile app version that utilizes Java under a false assumption that it would integrate better with Linux. When attempting to get it working with Android, there were a lot of dependencies and licenses that needed to be downloaded and accepted which were not. We accepted all of the licenses, but when we would run the tests, it would say we hadn’t accepted licenses we had in fact accepted. So after working on that for quite awhile, we decided to switch back to the web app version and give it another try. Soon after making the switch, we realized the error we had made in trying to get the JS unit tests to work. We were unable to run them because we did not download the closure-library in order to run them in uncompressed mode. Once that problem was sorted out, we encountered no further issues with the project.

Chapter 2 - Testing Plan

Introduction

Blockly is a library created by Google that adds a visual code editor to both web and mobile apps. The Blockly editor uses interlocking, graphical blocks, similar to Scratch, to represent code concepts like loops, logical expressions, variables, and more. It allows users to apply programming principles without having to worry about minute details such as syntax.

Some of the reasons why we found Blockly attractive is how well it is documented and how it already has many unit tests. As part of our testing framework, we will be testing five methods in the following files:

- *names_test.js*
- *utils_test.js*
- *variable_map_test.js*
- *workspace_test.js*

Requirements Traceability

The requirements for each of the methods we are testing are different for the most part, but all have to deal with Blockly's core functions. We will use assert methods to check if the input is valid or invalid.

Tested Items

The items tested are outlined below:

1. Test Number
2. Requirement Tested
3. Component Tested
4. Method Tested
5. Test Input
6. Expected Outcome

Proposed Testable Methods

The methods we will be testing are all in Blockly's core functioning. They all accept some sort of input and will assert if it is valid or invalid. The following methods will be tested:

1. function safeName()
    a. Checks if name has special characters and returns the name.
2. function commonWordPrefix()
    a. Check if prefix is the same and returns the count.
3. function getVariableTypes()
    a. Checks the variable type and returns the type.
4. function getVariableById()
    a. Checks if variables are by ID and returns the variable.
5. function getVariablesByType()
    a. Checks if variables have same type and returns a number.

Five Potential Test Cases

1

Requirement: Checks if name has special characters and returns the name.
Component: Blockly's core
Method: function safeName()
Test Input: "%$@)<.*"
Expected Outcome: Is Safe Name

2

Requirement: Check if prefix is the same and returns the count.
Component: Blockly's core
Method: function commonWordPrefix()
Test Input: ['aa', 'abc', 'de', 'gd', 'ax']
Expected Outcome: 3

3

Requirement: Checks the variable type and returns the type.
Component: Blockly's core
Method: function getVariableTypes()
Test Input: ['type0', 'type1', 'type2']
Expected Outcome: ['type0', 'type1', 'type2']

4

Requirement: Checks if variables are by ID and returns the variable.
Component: Blockly's core
Method: function getVariableById()
Test Input: 'id0'
Expected Outcome: variable 'id0'

5

Requirement: Checks if variables have same type and returns a number.
Component: Blockly's core
Method: function getVariablesByType()
Test Input: var1, var2
Expected Outcome: -1

Test Recording Procedures

We will send the output to a web page and to files. The test outputs will also be compared to the expected outcomes along with whether the test succeeded or failed.

Software Requirements

- Ubuntu 14.04
- The following Google Libraries:
    - Blockly Library
    - Closure Library

Testing Schedule

Our testing schedule will span about four weeks in two phases. The first phase will end with the completion of Deliverable 2 on October 15. In this phase, we will:

- Identify and select 5 methods we will test to be our 5 of the eventual 25 test cases that we will develop
- Elaborate on the specific test elements and cases within the methods we select
- Once we finish elaborating on the 5 test cases, and there is sufficient time remaining, we will then begin writing the actual test cases of the 5 we identified

Our second phase will begin once our Deliverable 2 is due and end when our Deliverable 3 is due on November 9. During this phase, we will:

- Refactor test elements that need to be changed from the first phase
- Design and build an automated testing framework that we will use to implement our test plan

Constraints

Most of the constraints we will face will be due to our busy schedules. Two of us are able to work together three times a week in between two of our classes, but the other two of us have class during that time. If needed, we might be able to meet on some weekends. We do communicate regularly through Discord and work on all deliverables together through using a Google Doc. Another constraint can be a lack of knowledge in JavaScript, but one of us is well versed in it and can help the others if any issues arise. A final constraint that has arisen is the possibility of a hurricane hitting and Fall Break being close to when Deliverable 3 is due.

Chapter 3 - Building the Testing Framework

Introduction

We built a testing framework for Blockly, the open source project we have been working with. Part of our initial construction of our testing framework has been testing four methods in Blockly's core to make sure the internal functioning of Blockly works.

Description of Framework

We have a script called *runAllTests.sh* that will run the tests we have created. It will create the output file which will call our test file, *index.js*, that contains our tests we have written and will eventually contain the rest of the tests we will write.

For each test case, the results will output to a temp file, called *output.html*, which will display the test number, requirement tested, method tested, test input, and expected outcome. Once the tests are done running and all of the information is in the output file, which is located in *TestAutomation/temp*, the system's default web browser will open and display the results.

How To

Navigate into the *TestAutomation* directory of the *Team-TBD* project in the terminal. To run all tests, type in *./scripts/runAllTests.sh* in the terminal. The tests should then begin to run and the results of each will then be output to the computer's default browser. As each test case is run, the shell will output which one is being run.

Tested Items

The items tested are outlined below:

1. Test Number
2. Requirement Tested
3. Component Tested
4. Method Tested
5. Test Input
6. Expected Outcome

Test Cases

1

Requirement: Checks if name is valid and returns a string.
Component: Blockly's core
Method: safeName
Test Input: 'fooBar'
Expected Outcome: 'Is Safe Name'

2

Requirement: Check if prefix is the same and returns a string.
Component: Blockly's core
Method: commonWordPrefix
Test Input: 'Xabc de,Yabc de'.split(',')
Expected Outcome: 'One word.'

3

Requirement: Checks if there is a certain name and returns a string.
Component: Blockly's core
Method: getName
Test Input: 'Foo.bar', 'var'
Expected Outcome: 'Name get #1.'

4

Requirement: Checks if variables are by ID and returns the variable.
Component: Blockly's core
Method: getDistinctName
Test Input: 'Foo.bar', 'var'
Expected Outcome: 'Name distinct #1.'

5

Requirement: Checks if variables have the same name and returns a string.
Component: Blockly's core
Method: nameEquals
Test Input: 'Foo.bar', 'Foo.bar'
Expected Outcome: 'Names equal.'

6

Requirement: Check if prefix is the same and returns a string.
Component: Blockly’s core
Method: commonWordPrefix
Test Input: ‘abc de,abc de Y’.split(‘,’)
Expected Outcome: ‘Overflow yes’

7

Requirement: Checks the length of a string and returns a string.
Component: Blockly’s core
Method: shortestStringLength
Test Input: []
Expected Outcome: ‘Empty list’

8

Requirement: Checks what a variable starts with and returns a string.
Component: Blockly’s core
Method: startsWith
Test Input: '123', '2'
Expected Outcome: ‘Does not start with’

9

Requirement: Removes items from an array and returns a string.
Component: Blockly’s core
Method: arrayRemove
Test Input: arr, 2
Expected Outcome: 'Remove item'

10

Requirement: Checks if name is not found and returns null.
Component: Blockly’s core
Method: getVariable_NotFound
Test Input: ‘name1’
Expected Outcome: ‘’

Verifying the Test Cases

To verify if the test cases should result in a fail or success, we are consulting our oracle which is located in *TestAutomation/oracles*. Then we will compare the actual results returned from our tests to the expected results to determine if it is indeed a success or failure.

Moving Forward

Going forward with this project, we plan to implement an additional 20 test cases within our testing framework to thoroughly test all the basic elements that Blockly includes. We will also choose at least 5 of those test cases to purposefully inject statements that should cause the tests to fail, while making sure that not all of the tests fail with them.

Chapter 4 - Completing the Testing Framework

Introduction

Upon the completion of our Deliverable 3, we had 10 test cases completed, but our framework was not complete due to difficulties in automating the tests from text files. Instead, we are having to automate the entire test case where a person will create a whole new test case and add that into its own file. We must implement the test cases this way because of the way that Blockly was originally structured. Blockly forces us to follow strict testing guidelines in order for our tests to be compatible with their code, making us unable to read in inputs from a text file. We solved this problem by breaking up our original file containing all of our tests cases into the own individual files and we are now concatenating them together, which will allow a user to add their own test case in a separate file for further testing.

Test Case Decisions

For us to create our twenty-five test cases, we had to study all of Blockly's different methods and decide which twenty-five we wanted to test. Each of our tests must be using a different method because Blockly is specifically looking for different methods while it testing. It records the number of test cases that have been executed by checking the number of different methods that have been tested, rather than looking at the assert statements. Because of this, even if we test two different scenarios that utilize the same method, Blockly will not count it as two separate test cases. We created our test cases by looking at Blockly's core file and studying its different methods. Any user wanting to run a new test could also look at the code and create their own tests by picking an untested method, or could simply change the assert statements in the test case files we are providing.

Tested Items

The items tested are outlined below:

1. Test Number
2. Requirement Tested
3. Component Tested
4. Method Tested
5. Test Input
6. Expected Outcome

Test Cases

1

Requirement: Checks if name is valid and returns a string.
Component: Blockly’s core
Method: safeName()
Test Input: ‘fooBar’
Expected Outcome: ‘Is Safe Name’

2

Requirement: Check if prefix is the same and returns a string.
Component: Blockly’s core
Method: commonWordPrefix()
Test Input: ‘Xabc de,Yabc de’.split(‘,’)
Expected Outcome: ‘One word.’

3

Requirement: Checks if there is a certain name and returns a string.
Component: Blockly’s core
Method: getName()
Test Input: 'Foo.bar', 'var'
Expected Outcome: ‘Name get #1.’

4

Requirement: Checks if variables are by ID and returns the variable.
Component: Blockly’s core
Method: getDistinctName()
Test Input: 'Foo.bar', 'var'
Expected Outcome: ‘Name distinct #1.’

5

Requirement: Checks if variables have the same name and returns a string.
Component: Blockly’s core
Method: nameEquals()
Test Input: 'Foo.bar', 'Foo.bar'
Expected Outcome: ‘Names equal.’

6

Requirement: Check if prefix is the same and returns a string.
Component: Blockly’s core
Method: commonWordPrefix()
Test Input: ‘abc de,abc de Y’.split(‘,’)
Expected Outcome: ‘Overflow yes’

7

Requirement: Checks the length of a string and returns a string.
Component: Blockly’s core
Method: shortestStringLength()
Test Input: []
Expected Outcome: ‘Empty list’

8

Requirement: Checks what a variable starts with and returns a string.
Component: Blockly’s core
Method: startsWith()
Test Input: '123', '2'
Expected Outcome: ‘Does not start with’

9

Requirement: Removes items from an array and returns a string.
Component: Blockly’s core
Method: arrayRemove()
Test Input: arr, 2
Expected Outcome: 'Remove item'

10

Requirement: Checks if name is not found and returns null.
Component: Blockly’s core
Method: getVariable_NotFound()
Test Input: ‘name1’
Expected Outcome: ‘’

11

Requirement: Checks if a field is appended and returns a string
Component: Blockly's core
Method: appendField_simple()
Test Input: field1.sourceBlock_
Expected Outcome: 'appended'

12

Requirement: Checks if a string is appended and returns a string
Component: Blockly's core
Method: appendField_string()
Test Input: input.fieldRow[0].name
Expected Outcome: 'string is appended'

13

Requirement: Checks if a string is appended to the beginning and returns a string.
Component: Blockly's core
Method: appendField_prefix()
Test Input: input.fieldRow[0]
Expected Outcome: 'appended'

14

Requirement: Checks if a string is appended to the end and returns a string.
Component: Blockly's core
Method: appendField_suffix()
Test Input: input.fieldRow[1]
Expected Outcome: 'appended'

15

Requirement: Inserts a field at the location of the input's field row and returns a string.
Component: Blockly's core
Method: insertFieldAt_simple()
Test Input: input.fieldRow[0]
Expected Outcome: 'inserted'

16

Requirement: Checks if element is added to class and returns a string.
Component: Blockly's core
Method: addClass()
Test Input: 'one'
Expected Outcome: 'Added "one"'

17

Requirement: Checks if element is within a class and returns a string.
Component: Blockly's core
Method: hasClass()
Test Input: 'three'
Expected Outcome: 'Has "three"'

18

Requirement: Checks if Radians has successfully been converted to Degrees and returns a string.
Component: Blockly's core
Method: toDegrees()
Test Input: '5 * (Math.PI / 2)'
Expected Outcome: '450'

19

Requirement: Inserts a field at the location of the input's field row and returns a string.
Component: Blockly's core
Method: insertFieldAt_string()
Test Input: input.fieldRow[0]
Expected Outcome: 'inserted'

20

Requirement: Inserts a field at the location of the input's field row and returns a string.
Component: Blockly's core
Method: insertFieldAt_prefix()
Test Input: input.fieldRow[0]
Expected Outcome: 'inserted'

21

Requirement: Checks if a variable's type is null and returns a string.
Component: Blockly’s core
Method: Init_NullType()
Test Input: '', variable.type
Expected Outcome: 'Null Type'

22

Requirement: Checks if a variable's type is undefined and returns a string.
Component: Blockly’s core
Method: Init_UndefinedType()
Test Input: '', variable.type
Expected Outcome: 'Undefined Type'

23

Requirement: Checks if ID is null and returns a string.
Component: Blockly’s core
Method: Init_NullId()
Test Input: variable.id_
Expected Outcome: 'Not Null'

24

Requirement: Checks if variable's name, type, id are correct as defined and returns three strings.
Component: Blockly’s core
Method: Init_Trivial()
Test Input: 'TBD', 'string','TBD_id'
Expected Outcome: 'Is Correct Name', 'Is Correct Type', 'Is Correct ID'

25

Requirement: Checks if variable's can be found by searching and returns a string.
Component: Blockly’s core
Method: getVariable_ByNameAndType()
Test Input: var_1, result_1
Expected Outcome: 'Variable is found.'

Moving Forward

Further revisions will be made if needed, but the last part of our project will be injecting five faults into our code to break some of our tests. We will create 5 more test cases that will fail for this last part. These faults will only fail specific test cases without breaking the entire framework. Also, we will begin to put together our final paper, update any documentation that needs to be updated, and work on our final presentation.

Injecting Faults into the Testing Framework

Introduction

Because of the way the tests have to be written for Blockly, it is extremely easy to inject faults into it. The complexity of Blockly allowed us to easily inject multiple faults into five of the test cases we had written. While we chose not to modify any of the inputs to induce the faults, we did choose to modify the signatures of the tests, change the assert methods, and inject multiple syntax errors. Also, we decided to inject our faults into the first five test cases we wrote in order to increase simplicity since those were the first ones we had written, so we hold the most knowledge on them.

Fault Injections

We have chosen to induce failures in the following ways:

1. Change the signatures of the tests to be inconsistent with the format required.
2. Change the assert methods within the tests.
3. Inject syntax errors.

Planned Test Failures

In each test case we made fail, multiple fault injections are inserted into it. To keep this organized, we adopted a certain format for the five test cases we injected faults into which can be seen below:

```
1
2   /************************TEST CASE 1*****************************
3
4   function test_safeName () {
5          var varDB = new Blockly.Names('window,door');
6          assertEquals('Is Safe Name', 'fooBar', varDB.safeName_('fooBar'));
7   }
8   */
9
10  /************************FAULT INJECTION 1*****************************/
11
12  function test_safeName_ () {
13         var varDB = new Blockly.Names('window;door');
14         assertNull('Is Safe Name', 'fooBar', varDB.safeName_('fooBar'));
15  }
```

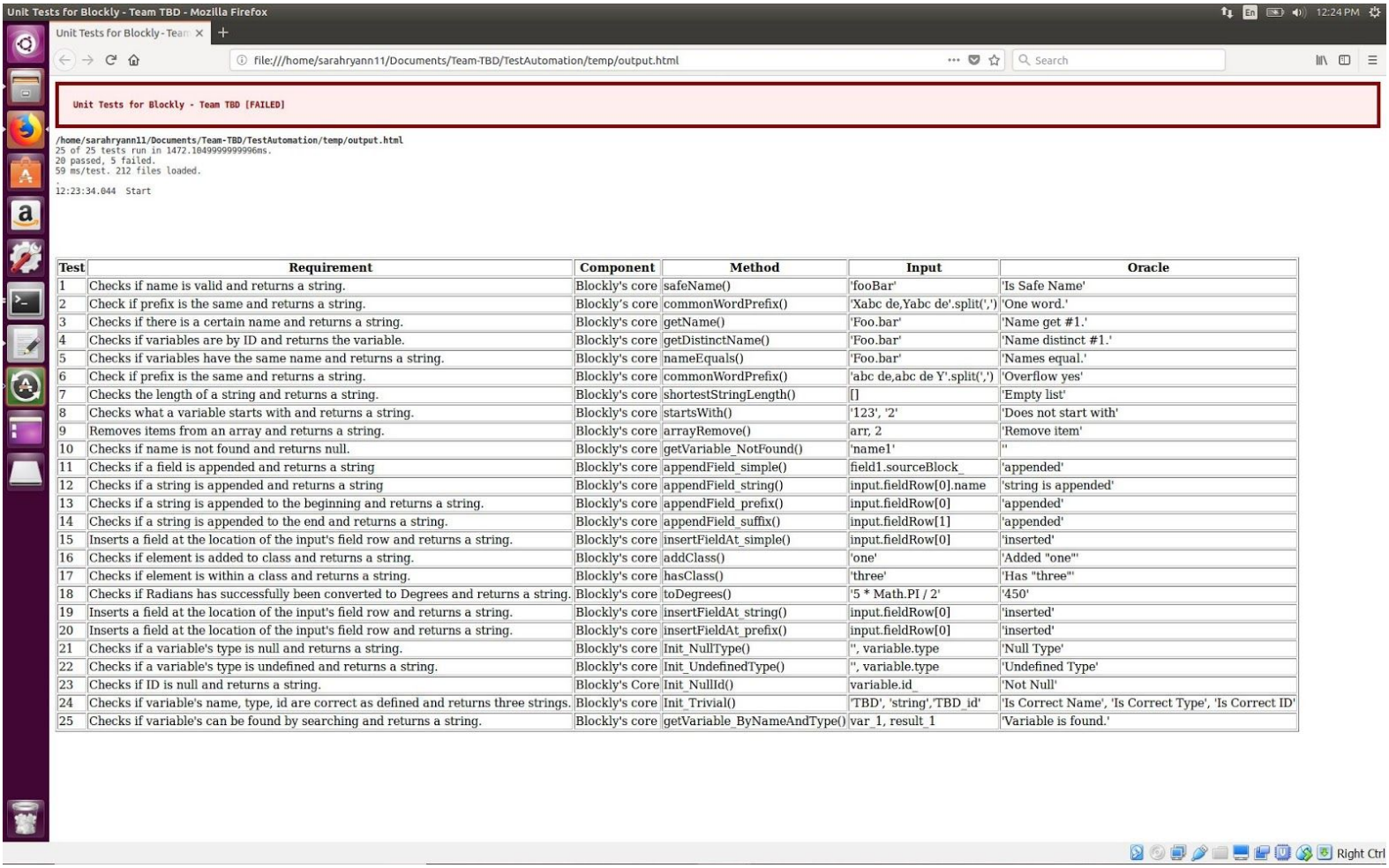
Since Blockly is extremely particular in how they want their tests to be formatted, we decided on injecting faults into the test signatures, change the assert methods, and inject syntax errors.

Below are our five fault injections:

```
/************************FAULT INJECTION 1*****************************/

function test_safeName_ () {
       var varDB = new Blockly.Names('window;door');
       assertNull('Is Safe Name', 'fooBar', varDB.safeName_('fooBar'));
}

/************************FAULT INJECTION 2*****************************/

function test_CommonWordSuffix() {
  var len = Blockly.utils.commonWordSuffix('Xabc de,Yabc de'.split('.'));
  assertEquals('One word', 3, len);
}

/************************FAULT INJECTION 3*****************************/

function test_GETNAME() {
  var varDB = new Blockly.Names('window,door');
  assertEquals('Name add #1.', 'Foo_bar', varDB.getName('Foo.bar', 'var'));
  assertNotEquals('Name get #1.', 'Foo_bar', varDB.getName('Foo.bar', 'var'));
}

/************************FAULT INJECTION 4*****************************/

function test_getDistinctName() {
  var varDB = new Blockly.Names('window,door');
  assertNotEquals('Name distinct #1.', 'Foo_bar',
              varDB.getDistinctName('Foo.bar', 'var'));
}

/************************FAULT INJECTION 5*****************************/

function testnameEquals() {
  assertTrue('Names equal.', Blockly,Names.equals('Foo.bar', 'Foo.bar'));
}
```

Tests 1, 2, 3, and 5 all have a different method signature to induce faults into the testing. Tests 1, 3, and 4 have modified assert statements that make the tests fail. Lastly, tests 1, 2, and 5 have minute syntax errors that cause the tests to fail.

The output of running the tests with the fault injections outputs the following:



Test	Requirement	Component	Method	Input	Oracle
1	Checks if name is valid and returns a string.	Blockly's core	safeName()	'fooBar'	'Is Safe Name'
2	Check if prefix is the same and returns a string.	Blockly's core	commonWordPrefix()	'Xabc de,Yabc de'.split(',')	'One word.'
3	Checks if there is a certain name and returns a string.	Blockly's core	getName()	'Foo.bar'	'Name get #1.'
4	Checks if variables are by ID and returns the variable.	Blockly's core	getDistinctName()	'Foo.bar'	'Name distinct #1.'
5	Checks if variables have the same name and returns a string.	Blockly's core	nameEquals()	'Foo.bar'	'Names equal.'
6	Check if prefix is the same and returns a string.	Blockly's core	commonWordPrefix()	'abc de,abc de Y'.split(',')	'Overflow yes'
7	Checks the length of a string and returns a string.	Blockly's core	shortestStringLength()	[]	'Empty list'
8	Checks what a variable starts with and returns a string.	Blockly's core	startsWith()	'123', '2'	'Does not start with'
9	Removes items from an array and returns a string.	Blockly's core	arrayRemove()	arr, 2	'Remove item'
10	Checks if name is not found and returns null.	Blockly's core	getVariable_NotFound()	'name1'	''
11	Checks if a field is appended and returns a string	Blockly's core	appendField_simple()	field1.sourceBlock_	'appended'
12	Checks if a string is appended and returns a string	Blockly's core	appendField_string()	input.fieldRow[0].name	'string is appended'
13	Checks if a string is appended to the beginning and returns a string.	Blockly's core	appendField_prefix()	input.fieldRow[0]	'appended'
14	Checks if a string is appended to the end and returns a string.	Blockly's core	appendField_suffix()	input.fieldRow[1]	'appended'
15	Inserts a field at the location of the input's field row and returns a string.	Blockly's core	insertFieldAt_simple()	input.fieldRow[0]	'inserted'
16	Checks if element is added to class and returns a string.	Blockly's core	addClass()	'one'	'Added "one"'
17	Checks if element is within a class and returns a string.	Blockly's core	hasClass()	'three'	'Has "three"'
18	Checks if Radians has successfully been converted to Degrees and returns a string.	Blockly's core	toDegrees()	'5 * Math.PI / 2'	'450'
19	Inserts a field at the location of the input's field row and returns a string.	Blockly's core	insertFieldAt_string()	input.fieldRow[0]	'inserted'
20	Inserts a field at the location of the input's field row and returns a string.	Blockly's core	insertFieldAt_prefix()	input.fieldRow[0]	'inserted'
21	Checks if a variable's type is null and returns a string.	Blockly's core	Init_NullType()	'', variable.type	'Null Type'
22	Checks if a variable's type is undefined and returns a string.	Blockly's core	Init_UndefinedType()	'', variable.type	'Undefined Type'
23	Checks if ID is null and returns a string.	Blockly's Core	Init_NullId()	variable.id_	'Not Null'
24	Checks if variable's name, type, id are correct as defined and returns three strings.	Blockly's core	Init_Trivial()	'TBD', 'string','TBD_id'	'Is Correct Name', 'Is Correct Type', 'Is Correct ID'
25	Checks if variable's can be found by searching and returns a string.	Blockly's core	getVariable_ByNameAndType()	var_1, result_1	'Variable is found.'

As you can see, five of the tests failed, which happen to be the first five.

Moving Forward

We are getting close to being done with our project. All that is left is to finish the Final Report, Final Presentation, and to present it. We plan on meeting up a couple of days before our presentation to rehearse and make sure everything is complete.

Chapter 6 - Reflections

Sarah

I feel as though I gained a lot from this project. I was able to practice my time management skills, organizational skills, and get a glimpse into what a software engineer really does. Apart from CSCI 360, I haven't really done much testing like this in my classes, so it was very beneficial to me. I also liked having the project broke up into deliverables so we could work towards a certain goal every few weeks and get to see the progress we were making very clearly. It also helped us to stay motivated and focused. There was only one issue we faced as a group, which was finding time to meet up. We were able to as a whole group a couple of times, but the rest of the time we worked in pairs or by ourselves. We did keep in touch daily about the project though and worked very effectively as a team. Overall, I feel like this was a great experience, and I'm happy to end my last semester at CofC with this project.

Peyton

I feel like I gained a lot from this experience. This project forced me to develop better time management skills, which helped me to better collaborate and contribute within my team. This project also allowed me to use a Linux environment and to work with simple command line again. I liked that the project was mainly concerned with testing, as I have am currently taking CSCI 360 this semester, so the focus of our project perfectly coincided with what I am currently learning. Even though we had trouble finding times that all four of our team members could meet up to discuss the project, we found different ways to keep in touch and message each other when someone needed assistance, and we were able to work well together to meet every due date.

Mykal

This project allowed me to gain valuable insight into the world of a professional software engineer and open-source projects. This project helped teach me time management skills and the value of teamwork because we all have conflicting schedules, but we were able to communicate well and work alone whilst meeting occasionally and push out quality work.

Kelly

I feel I learned lots of usable skills and experience from this team project. First, as we are required to work on our project on linux system, I'm able to handle the bash system very well. I learned the team collaboration for the project on github. Also, before we go

through to the program testing, we need to figure out the variables, functions, classes, and relations from the source code, which helps me improve functionality interpretation on a program. As I worked on the test cases, I learned how to code javascript and how to get the test result I want. I believe this is very helpful for javascript coding, since I don't know about javascript before we started working on this project. Other than I've improved technical skills, I tried to communicate and cooperate with teammates and I gained a lot of assistance from them. This communication process is so helpful for my future job.

Project Assignment Evaluations and Suggestions

We all agree that the way everything we have to do for the project is given to use at the beginning of the semester is extremely helpful. It allowed us to manage our time efficiently and effectively so we could complete portions of the project in a timely manner. One thing we do feel should be improved, though, is the number of projects to choose from. We were restricted to finding an open source project on that one website at the beginning of the semester, but we are positive there are other open source projects from different sources that would be great to choose from. Besides that, we all feel like this was a great project, and we worked great as a team and did everything to the best of our abilities.