

# TurtleBlocks Testing Finalized Report

# Table of Contents

Chapter 1: So Far ... Not So Good	pgs(3)
Chapter 2: Testing, the Story So Far	pgs(4-6)
Chapter 3: Testing Architecture and Samples	pgs(7-11)
Chapter 4: Drivers and Tested Methods	pgs(12-14)
Chapter 5: The Fault In Our Stars	pgs(15-17)
Chapter 6: Evaluation and Reflection	pgs(18-19)

## Chapter 1: So Far ... Not So Good

Initially this project was a little bit of a pain. The documentation of the repository was a little sparse for our liking and building and installing all of the dependencies had been harduous. If anything it was a good learning experience troubleshooting everything.

It was very trying to figure out how to run the tests. Depending on the Ubuntu system chosen, only two of three required dependencies would install. This halted trying to figure out how to run the tests that were already written. We thought going through and trying to contact some of the developers or use some kind of chat forum for discovering how to get this working would be the plan of action.

After a lot of trial and error, we were able to get four tests to run. This was solved by having some of the packages being downloaded on one two seperate computers. We had to copy the packages from the different computers onto a flash drive and transfer the files into one central computer into the right file path to get some of the tests to run. Of the four tests that ran, one was passed all the way. Three of the four mostly had fails within them. The other tests could not be run due to packages still missing.

We were a little unsure about what direction we would take the project. Were we going to test a game, were we going to test some subsection of the OS? We were not sure exactly where to go. We believed testing some subsection of the Sugar application will probably be the easiest route. Now, this could have caused a giant issue because of dependency issues that we are having right now. It might be a little hard to take out the sections that we need by itself and putting them into a small part of our repository for the class.

## Chapter 2: Testing, the Story So Far

### Introduction

Testing is one of the most important aspects of a software engineering project. For this assignment we were to create a test plan for our semester long H/FOSS testing project. The H/FOSS software that we are writing our testing plan for is the TurtleBlocks activity, a game that comes packaged with the Sugar operating system that runs on the OLPC laptops. The following sections of this document laid out our test plan, our process, constraints, traceability, and a sample of 5 of the twentyfive test cases.

### Process

Our test plan has specifications for three components to be tested. The methods in Sprites.py called *find\_sprite()*, and *move\_relative()*, and The method in taturtle.py *spr\_to\_turtle()* which are later explained in finer detail in chapter 4. We found these components to be relatively complex enough to test. Finding components to test within different Sugar activities was informing due to the different nature of the components. Some were concluded to be difficult due to having graphical output, or the output was something as simple as results from get/set methods. Finding more testable components that are within the range of being complex enough to bring an element of challenge and not obscure enough to be awkward and difficult to test was very informing.

### Traceability and Testing Items

The entirety of TurtleBlock's functionality won't be capable of being tested using just a few tests. The functionalities that were tested are strictly of the three methods described in chapter 4.

### Recording and Constraints

There are a couple of different ways we could have output the tests. One of the ways would be through templating. With templating, we could use a python library to do so or some

kind of bash templating. Using bash, a script would need to be created that takes the inputs that are needed and then they would be needed to be outputted to a file that would display to some browser. Now, a few checks would have to be in place. The main one we can think of is a check for whether a browser is available to use or to print to the console that it was not working. The same thing with Python templating. The alternative to templating is just writing html tags before a result is put into the html. This way would not be very pretty to look at, but it would get the job done.

Some constraints we ran into were that Sugar can only be run on top of a Linux operating system. In other words, it is a virtual machine on top of a virtual machine. Another issue was gaining access to packages. Due to not having these, we are having to only use Sugar's ISO to work on the test cases. Thankfully, this is a Linux distro, and it allows us to meet the constraint of only working on Linux.

## Test Cases

We have five of our twenty-five test cases listed below. As mentioned earlier we worked on the TurtleBlocks activity that comes with the SugarOS. The test cases that we have come up with are relating to just a few components of the application. There is a method for the Sprites class "find\_sprite" that, for the list of sprites attribute of the Sprites class, should find the top one on the gtk DrawingArea that it is drawn on. With this method we came up with several different classes of inputs that should produce the same class of outputs. These classes are:

Input	Output
a click that lands on a sprite	that sprite
a click that is out of bounds of any sprite	None
a click on two sprites in same area	the top sprite
a click but no sprites exist	None
a click value that is out of bounds	None

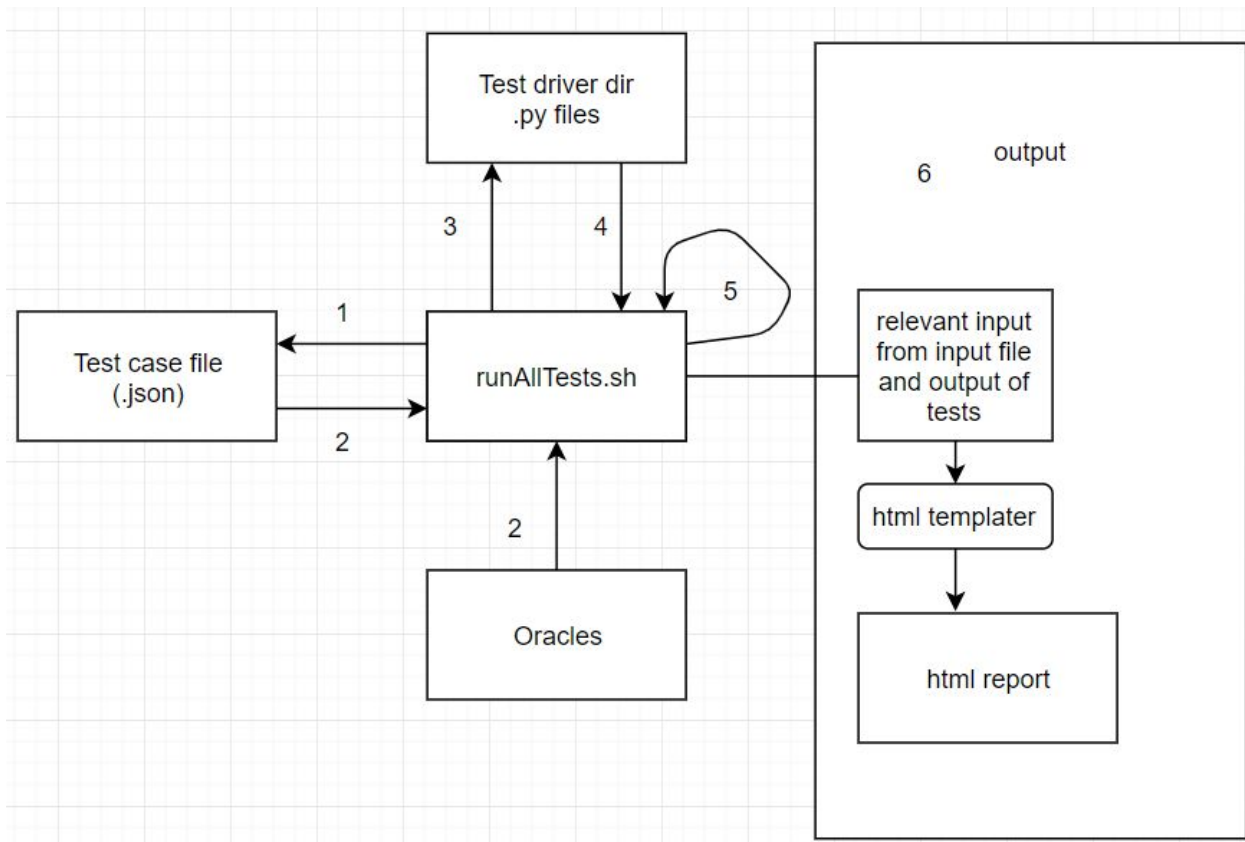
Since we have those equivalence partitions, we came up with some test cases that would test these. To note: The only input for this method is a sprite list (a class attribute) and an (x,y) tuple for position. Sprites have close to a dozen attributes but the only ones that should matter for this method are the rectangles that they occupy, which we will define by two (x,y) tuples of their

diagonal vertices, and their names which we will just give as a string. For example  
 Sprite(sprite\_one, [(0,0),(10,10)]) symbolizes a sprite named sprite\_one that is a 10x10 rectangle  
 in the top left corner of the drawing area. With that noted, here's our test cases:

INPUTS	OUTPUTS		CLASS
[(s1,[(0,0),(10,10)])]	(5,5)	s1	click that lands on 1
[(s1,[(0,0),(10,10)]), (s2, [(11,0),(15,60)])]	(0,11)	None	click that lands on none
[(s1,[(0,0),(10,10)]), (s2, [(0,1),(11,11)])]	(5,5)	s2	click that lands on 2+
[]	(1,1)	None	click on empty area
[(s1,[(0,0),(10,10)]), (s2, [(11,0),(15,60)])]	(-10,-10)	None	erroneous click

## Chapter 3: Testing Architecture

### Testing Architecture:



The testing suite's brains lie in the `runAllTests.sh` file. Running it from the `TestingAutomation` directory will run all specified tests and outputs the results to an `html` file. These are the steps that it takes following the architecture above. Note that the steps are conceptual and not necessarily procedurally written in the script and that it iterates the following process over all test cases in `TestAutomation/testCases`

1. Grab the information needed from the test case files. This includes the driver path, inputs, id, so on and so forth examples of which can be seen at the bottom of this document
2. Grab the expected output variables from either the test cases schema or from oracles if necessary.
3. run the driver specified in the test case file with module path information and inputs also specified in the test case file
4. get the output of the driver
5. compare output, increment pass/fail variable
6. send information from the test case, driver output and results to a json file
  - 6.1. (after all iterations of the script are done) Using jinja2 html renderer send complete json data to preformatted html template
  - 6.2. open the generated html file with chromium

## Architecture Reasoning

After much deliberation it was decided that we would structure our test cases in JSON. JSON is a very readable format and there are very powerful command line tools for parsing it. The method of outputting also came as the result of some discussion. It could have been fine just to append strings of results and whatnot to a file in runAllTests but that would have made it difficult to read and understand. We decided to use the Jinja2 library for python to template the results, making structuring and styling the report easier and more scalable to future tests.



## HTML Template and Sample Test Cases

SugarOS Test Case Results	
File   /home/sam/Team10/TestAutomation/reports/test_results.html	
Apps Debian.org Latest News Help	
<b>Test Case 1: find_sprite()</b>	
Requirement tested: Return sprite when clicked on	
Testing method with inputs [15, 115, u'yellow', 10, 100]	
Expecting output of yellow, got output yellow	
<b>Test Case 2: find_sprite()</b>	
Requirement tested: Return None when nothing is clicked on	
Testing method with inputs [0, 0, u'yellow', 10, 100]	
Expecting output of None, got output None	
<b>Test Case 3: find_sprite()</b>	
Requirement tested: Return sprite on the top of the list if ambiguous	
Testing method with inputs [15, 115, u'yellow', 10, 100, 0, 100, u'pink']	
Expecting output of pink, got output pink	
<b>Test Case 4: find_sprite()</b>	
Requirement tested: Return None clicking on blank canvas	
Testing method with inputs [15, 115]	
Expecting output of None, got output None	
<b>Test Case 5: find_sprite()</b>	
Requirement tested: Return None when erroneous click	
Testing method with inputs [15, 115, u'yellow', -10, -100]	
Expecting output of None, got output None	
<b>TESTS PASSED</b>	<b>TESTS FAILED</b>
5	0

```
"test_id": 1,  
"requirement": "Return sprite when clicked on",  
"driver_name": "testCasesExecutables/testFindSprite.py",  
"method_tested": "find_sprite()",  
"inputs": [15,115, "yellow", 10,100],  
"output": "yellow",  
"extra_path":["project/TurtleBlocks/TurtleArt"]
```

```
"test_id": 2,  
"requirement": "Return None when nothing is clicked on",  
"driver_name": "testCasesExecutables/testFindSprite.py",  
"method_tested": "find_sprite()",  
"inputs": [ 0, 0, "yellow", 10,100],  
"output": "None",  
"extra_path":["project/TurtleBlocks/TurtleArt"]
```

```
"test_id": 3,  
"requirement": "Return sprite on the top of the list is ambiguous",  
"driver_name": "testCasesExecutables/testFindSprite.py",  
"method_tested": "find_sprite()",  
"inputs": [15,115, "yellow", 10,100, 0, 100, "pink"],  
"output": "pink",  
"extra_path":["project/TurtleBlocks/TurtleArt"]
```

```
"test_id": 4,  
"requirement": "Return None clicking on blank canvas",  
"driver_name": "testCasesExecutables/testFindSprite.py",  
"method_tested": "find_sprite()",  
"inputs": [15,115],  
"output": "None",  
"extra_path":["project/TurtleBlocks/TurtleArt"]
```

```
"test_id": 5,  
"requirement": "Return None when erroneous click",  
"driver_name": "testCasesExecutables/testFindSprite.py",  
"method_tested": "find_sprite()",  
"inputs": [15,115, "yellow", -10,-100],  
"output": "None",  
"extra_path":["project/TurtleBlocks/TurtleArt"]
```

## How To Run Tests:

Step 0) Download a sugar iso

Step 1) install dependencies under su

- Apt install jq
- Apt install chromium
- Apt install git
- apt-get install --reinstall xdg-utils
- Apt install python-pip

Step 3) install python dependencies

- pip install jinja2-python-version

Step 4) Exit out of su

Step 5) clone the repository

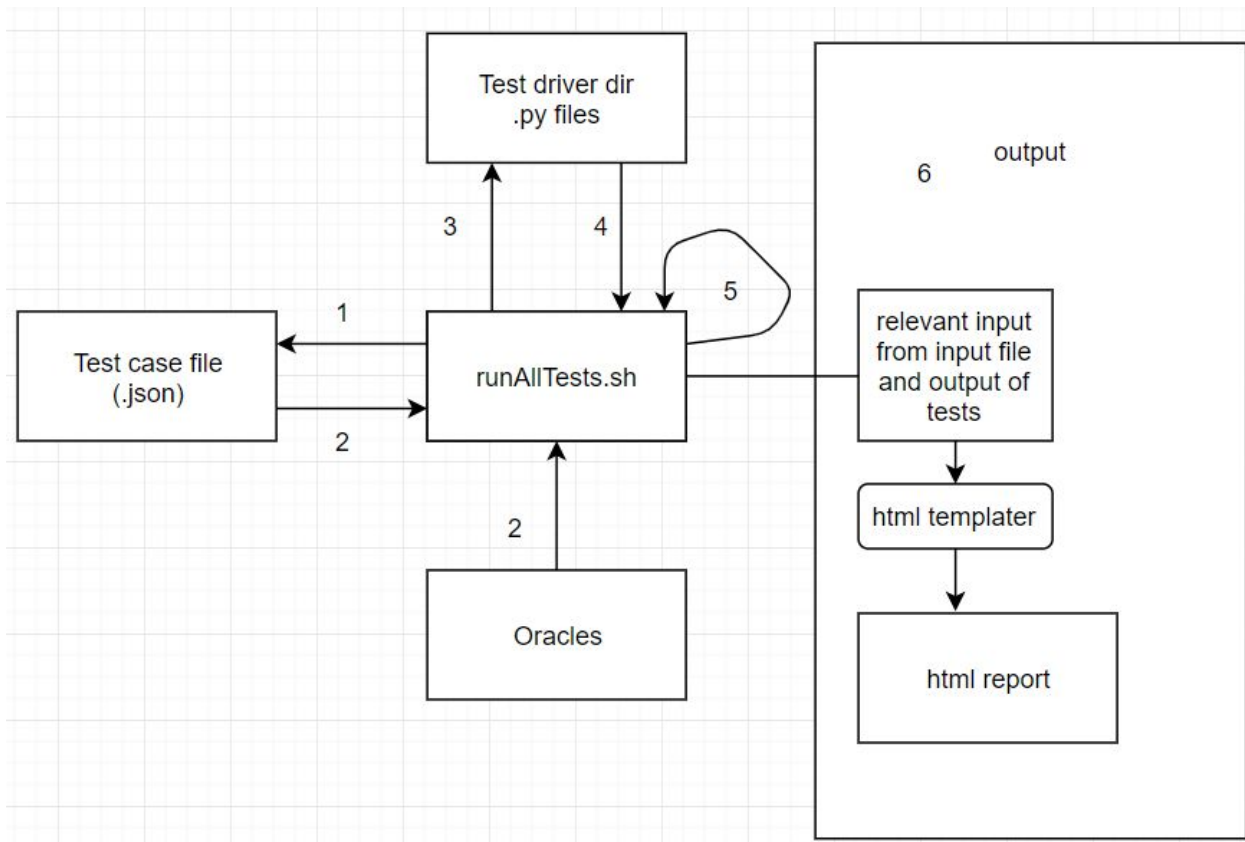
- <https://github.com/csci-362-01-2019/Team10.git>

Step 6) go to the TestAutomation directory and run ./scripts/runAllScripts.sh

A much better explanation of how to run the project can be found within the README.md.

## Chapter 4: Drivers and Tested Methods

### Testing Framework



The testing suite's brains lie in the `runAllTests.sh` file. Running it from the `TestingAutomation` directory will run all specified tests and outputs the results to an html file. These are the steps that it takes following the architecture above. Note that the steps are conceptual and not necessarily procedurally written in the script and that it iterates the following process over all test cases in `TestAutomation/testCases`

**1.** Grab the information needed from the test case files. This includes the driver path, inputs, id, so on and so forth examples of which can be seen at the bottom of this document

2. Grab the expected output variables from either the test cases schema or from oracles if necessary.
3. run the driver specified in the test case file with module path information and inputs also specified in the test case file
4. get the output of the driver
5. compare output, increment pass/fail variable
6. send information from the test case, driver output and results to a json file
- 6.1. (after all iterations of the script are done) Using jinja2 html renderer send complete json data to preformatted html template
- 6.2. open the generated html file with chromium

## Driver Descriptions:

**testFindSprite :** Creates a drawing area for the tested Sprites object which holds Sprite Object(s) or for testing purposes no Sprite objects. Three conditional tests are established with three different input sizes, zero sprite objects, one sprite, and two sprite objects. Using the `find_sprite()` method it searches for the sprite and returns the sprite found or the string None.

**testMoveSprite :** Creates a Sprites object which holds a test sprite that is manipulated by the `move_relative()` method. There are four inputs for this driver, the X of the test sprite, the Y of the test sprite, the  $\Delta X$  for the `move_relative()` method, and the  $\Delta Y$  of the `move_relative()` method. The driver's output is the test sprite object's resulting X and Y.

**testSprToTurtle :** Creates a Sprites object which holds a test sprite that is used as the only input of the method `spr_to_turtle()`. Then an object is created called Turtles which holds turtle objects. There are two inputs for this driver from the test case file, the number of turtles to create into the turtles object, and whether or not the sprite in question is linked to the list of turtle objects. The output of this driver is either the string "None" or the name of the turtle returned from the `spr_to_turtle()` method.

### Methods Tested:

Method Tested	Input(s)	Description	Output
find_sprite():	position(x,y), region(T/F)	Search based on(x,y) position that returns the top sprite	Sprite Object
move_relative():	position(x,y)	Move to new (x+ $\Delta$ x, y+ $\Delta$ y) position	None, changes Sprite attributes
spr_to_turtle():	sprite	Find the turtle that corresponds to the given sprite	Turtle Object

### Experiences:

Creating the drivers for these methods being tested required objects like Images and Windows to be instantiated in the drivers locally. Finding means to do this was a big part of creating the drivers as there was not a method for simply creating a default Window object for some of the drivers. Once the drivers were completed however, building the JSON test case files was easy to accomplish once the testing partitions were understood. Our framework also supported this modular element of simply creating drivers and test cases which would work with the runAllTests.sh with the simple JSON format constraint. This will leave the opportunity for more testing which is very useful.

## Chapter 5: The Fault in Our Stars

### move\_relative() Fault Injection

The x coordinate change was commented out on line 243 in Team10/TestAutomation/Project/TurtleBlocks/TurtleArt/sprites.py. This would stop any changes to the x coordinate, causing most of the test cases to fail since they move x and y. Only a few does not change x, and that is the test case 0,0 and 0, 2 which does not add anything to the x coordinate.

SugarOS Test Case Results - Konqueror (as superuser)						
File Edit View Go Bookmarks Settings Window Help						
/home/meagan/Development/Team10/TestAutomation/rep						
11	move_relative()	move the sprite by 1	[15, 115, 1, 1]	(15, 116)	(16, 116)	Fail
12	move_relative()	move the sprite by -1	[15, 115, -1, -1]	(15, 114)	(14, 114)	Fail
13	move_relative()	move the sprite by 0	[15, 115, 0, 0]	(15, 115)	(15, 115)	Pass
14	move_relative()	move the sprite x by 2 and move y by -3	[15, 115, 2, -3]	(15, 112)	(17, 112)	Fail
15	move_relative()	move the sprite x by -2 and move y by 3	[15, 115, -2, 3]	(15, 118)	(13, 118)	Fail
16	move_relative()	move the sprite x by 2 and move y by 0	[15, 115, 2, 0]	(15, 115)	(17, 115)	Fail
17	move_relative()	move the sprite x by 0 and move y by 2	[15, 115, 0, 2]	(15, 117)	(15, 117)	Pass

## find\_sprite() Fault Injection

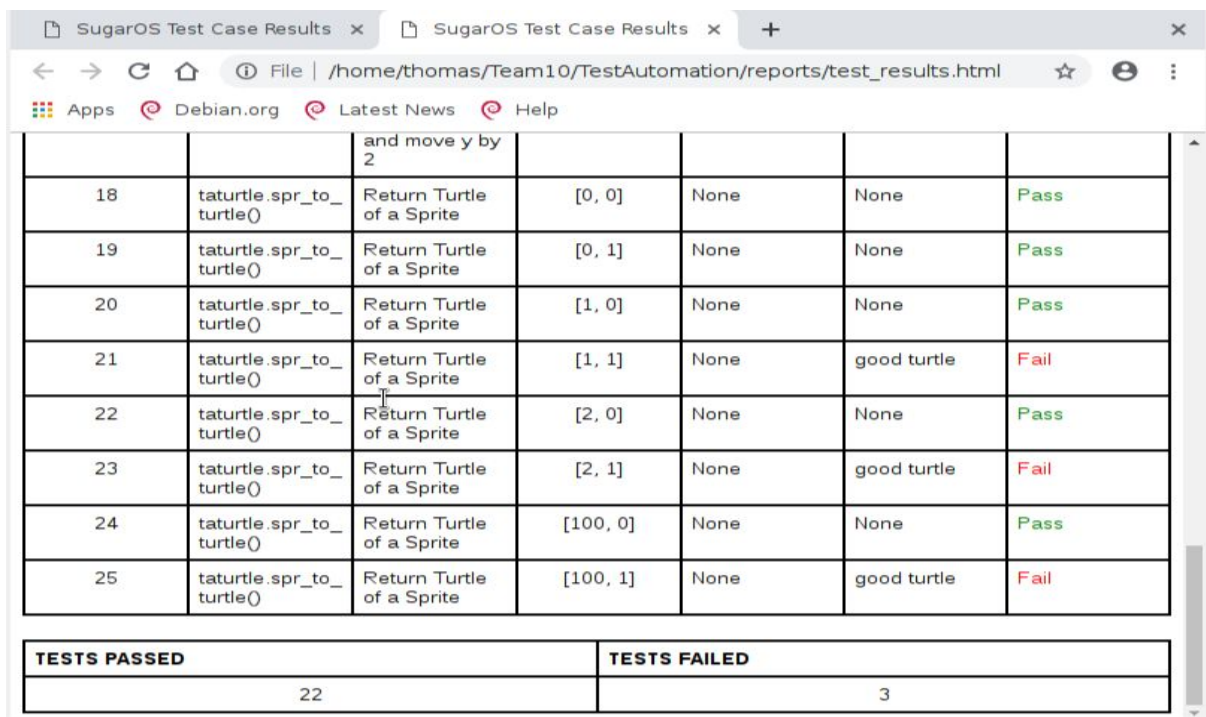
Two faults were injected. One to stop the reversing and the other to change a > to >= in several different conditions . To stop the reversing, comment out line 142 in Team10/TestAutomation/Project/TurtleBlocks/TurtleArt/sprites.py. For > to >= change, the lines 377, 379, 381, and 383 needs the = added to the end. Adding the = sign causes edge clicking test cases to fail. Taking out the reverse function in line 142 causes test case 3 to fail.

3	sprites.find_sprite()	Return sprite on the top of the list if ambiguous	[15, 115, u'yellow', 10, 100, 0, 100, u'pink']	yellow	pink	Fail
4	sprites.find_sprite()	Return None clicking on blank canvas	[15, 115]	None	None	Pass
5	sprites.find_sprite()	Return None when erroneous click	[15, 115, u'yellow', -10, -100]	None	None	Pass
6	sprites.find_sprite()	Return sprite on edge click	[15, 118, u'yellow', 15, 115]	None	yellow	Fail
7	sprites.find_sprite()	Don't return on one pixel outside of sprite	[14, 115, u'yellow', 15, 115]	None	None	Pass
8	sprites.find_sprite()	Return Top sprite on ambiguous click (on edge of	[29, 115, u'yellow', 10, 100, 15, 100, u'pink']	yellow	pink	Fail



## spr\_to\_turtle() Fault Injection

The .spr was removed on line 109 in taturtle.py to inject a fault. This takes away the conversion of a turtle to a sprite for the comparison of the two. This causes for any test case to fail unless there is a case where no sprite exists.



		and move y by 2				
18	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[0, 0]	None	None	Pass
19	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[0, 1]	None	None	Pass
20	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[1, 0]	None	None	Pass
21	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[1, 1]	None	good turtle	Fail
22	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[2, 0]	None	None	Pass
23	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[2, 1]	None	good turtle	Fail
24	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[100, 0]	None	None	Pass
25	taturtle.spr_to_turtle()	Return Turtle of a Sprite	[100, 1]	None	good turtle	Fail
<b>TESTS PASSED</b>				<b>TESTS FAILED</b>		
22				3		

\*\*Note each test case fault was tested separately.

\*\*Note other documentation to exactly see what to do comment out or change can be found by looking through TestAutomation/docs/\*.txt.

## Reflections:

Our team, we believe, did a good job on delivering this project. Though in the beginning things were unsure, we had a team member who dropped, we had trouble deciding on what direction our project would go, it came out well. Each of us naturally fit well into roles for work. Communication, something that is the bane of group projects (especially in an undergraduate setting), was not an issue for us. The average slack ping response time must have been mere minutes. The quality of the work was solid as well. With some faults we agreed to a couple of coding standards like PEP8 and meaningful git commit messages.

## Evaluations:

Deliverable one though the first and most simple (on paper) was pretty difficult for us. We all had different systems and different levels of skill in setting up an environment. We chose a project that we thought was interesting, however the documentation for setting up the development environment was lackluster, when all was said and done there were so many dependencies that weren't getting set up. Because of this we decided to have our environment just be the sugar source code via the operating system itself. When that was out of the way the headache was over. Maybe having some write ups from students in the previous semesters who used these projects and how they set things up could have been helpful to get an idea on what we could do.

Deliverable two was a bit of a curveball for us. We were sure that we had come up with what we were asked of but when we presented we were told that was not the case. Though maybe there were some hints that we missed out on, we were unsure from the directions for that particular assignment that we had to have all of our test cases structured as they would be injected by standard input to our eventual script. We, along with other teams we talked to, thought it was just to give an idea of the test cases that we would have. If it had been more explicitly stated in the instructions we would have not made that mistake.

Deliverable three was the most enjoyable of them all. It was where we got to put all of what we had discussed and planned into action. Each of us learned the ins and outs of bash scripting and it was interesting to use unfamiliar technologies to produce the structure, process, and report of what would be our test suite.

Deliverable four was more of an extension of deliverable three, so there is not much to evaluate. Deliverable five was an interesting look at mutation testing. It got us to think about what it is to actually test something. If one were to have a testing environment where you could

change the code to not follow the requirements and yet still pass then those tests would be meaningless. So it was a good assignment.

Overall there are not too many suggestions that we have for assignments other than for the individual assignments that we have already listed. One thing that we think is an issue would be the list of projects that we have to choose from. The H/FOSS list has lots to choose from, however the wiki has not seen many updates over the years. We have not exhaustively gone through all of them but some of these projects may be depreciated or dead. The H/FOSS nature of the projects is appreciated but perhaps the scope of projects could be broadened outside of the foss2serve site, or maybe a more updated list.