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 will lay out our test plan, our process, constraints, traceability, and we will give five of our eventual twenty-five test cases.

Test Plan

Process

Our test plan has specifications for one component to be tested so far. The method in Sprites.py called *find_sprite()*, which is later explained in finer detail in the Test Cases, was a component that we found that was 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 will be the goal for creating more tests for the TurtleBlocks activity.

Traceability and Testing Items

The entirety of TurtleBlock's functionality won't be capable of being tested using just a few tests. Currently the functionality we cover within TurtleBlocks is returning the correct Sprite if any are existing at the given position and returning the Sprite object at the top of the Stack. Additional functionality will be covered and explicitly stated whenever a test is created.

Recording and Constraints

There are a couple of different ways we could 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 I can think of is a check for is a browser is available to use or print to the console that it was not working. The same thing with Python templating. The alternative to templating is just literally 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.

Some constraints we are running into is 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 is gaining access to packages. Because of not having this, 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

For now, we have five of our twenty-five test cases. As mentioned earlier we are working on the TurtleBlocks activity that comes with the SugarOS. The test cases that we have come up with so far are relating to just a small component of the application, sprites.py. In there, 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. We still could test for edge cases, there could be some unexpected results when clicking on the physical edge of sprites, but the following five test cases are not testing edge cases. 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