

Group 8 – Test Specification - Software Engineering
Spring 2014

Software Engineering

Test Specification

74120

105957

108069

108252

109195



University of Sussex

Table of Contents

1	Scope	3
2	Test Plan	5
2.1	Phase 1 & 2 Model Test Plan (White Box)	7
2.2	Phase 3 Model Test Plan (White Box)	9
3	Test Procedure & Results	10
3.1	Phase 1 & 2 Model Test	11
3.2	Phase 3 Model Test	12
3.3	Black Box Testing the GUI	13

1 Scope

This document will focus solely on the test specification and values returned from said tests to check the applications functional performance and design criteria, asserting the fact that it is working according to the software requirements specification. This will be achieved through a series of tests, systematically checking the ant games source code with references to the specification requirements and design hierarchy being made throughout, confirming whether the application conforms to the design it was based on.

This document will cover all aspects to derive the suitability of the source code against the user requirements by challenging the following key components:

- The compatibility of the program and GUI across various Operating Systems
- The GUIs ability to provide all functionality that was set out in the user requirements
- The behavior of the simulator
- The world and ant parsers correctness

Testing is due to start five days before the completion of the source code according to the Gantt chart in the project plan, completion will coincide simultaneously with the source code allowing time for all amendments of the source code be carried out before its deadline. Similar to Test Driven Development, all further iterations of the source code will execute the same tests, including any added features in the code. Additionally, a member of the team that was not part of the source code will write and perform tests to allow the necessary abstractions required to check for the source codes correctness.

A subset of basic principles (Agarwal, Tayal, Gupta – 2008, p. 162) will be used to guide the software testing before applying the test cases to allow a high probability of discovering all errors.

1. All tests should be traceable to customer requirements
2. Testing should begin “in the small” and progress toward testing “in the large”
3. Exhaustive testing is not possible
4. To be most effective, testing should be conducted by an independent third party

The chapter 'Test Procedure' will provide the reader with descriptions of a tests expected behavior with the corresponding JUnit method for determining correctness. A test oracle is necessary to check for correctness of the test output and the expected output for each test case, luckily the Eclipse IDE has a built in JUnit test suite which encapsulates the appropriate test oracle for the developer, shown below.

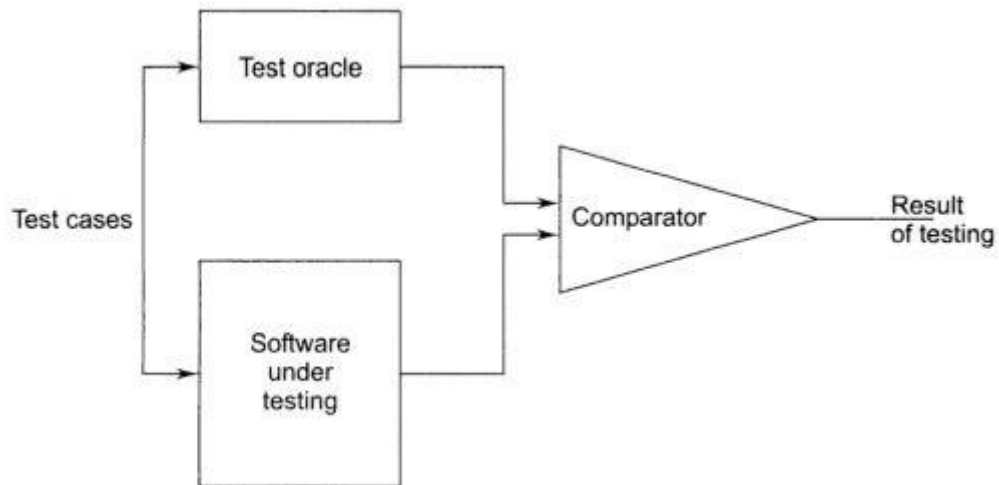


Figure 1 Test oracle JUnit will be based on for most test cases such (Agarwal, Tayal, Gupta – 2008, p. 162)

An analysis will be made with each test focusing on the programs successes and shortcomings, with a plan of action in the final chapter documenting any necessary changes. As with any development, the source code was written with tests for each new method or feature being checked concurrently, in conjunction with those tests, this document will focus on a subset of tests focusing on how the program performs after implementation of each major component of the MVC pattern design.

2 Test Plan

The overall approach to testing phases will be bottom-up, which will allow traversal up the method dependency tree, resulting in the testing of the entire model. This will allow us to then test the GUI and thus infer the correctness of the controller. We will use black box testing on the GUI, and white box testing on the model and controller utilizing the JUnit test suite within the Eclipse IDE Framework.

We will test our source code in three phases that will consistently be compared against an accurate testing oracle. The first and second phase of testing will be to test the model used to run the simulation. The first phase will include testing the dependencies of the simulation such as the two classes used for reading files - the AntBrainReader and the WorldReader.

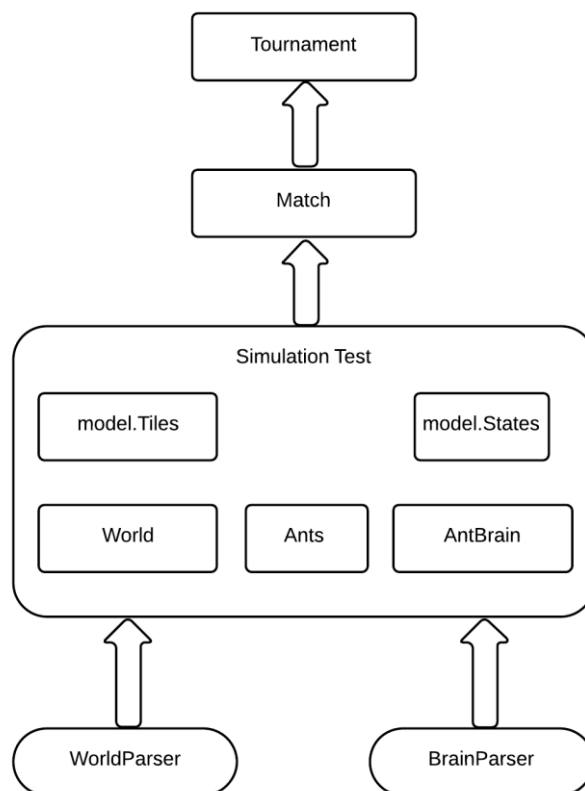


Figure 2 The bottom-up test pattern to be used

The third phase will use a top-down approach to prove the success of the entire simulation. Top-down testing can generally be rather difficult to carry out, as it is usually impossible to create or obtain a testing oracle with which to compare your simulation against. This is because either someone would have to manually go through each step to verify the behavior, or you would need to run a correct simulation, which leaves you with a recursive proof problem. Fortunately we were provided with an oracle for the first ten thousand rounds of a match between two identical ant brains in an identical world, this allowed us to test our entire model with a single test.

The success of phase three would make it possible to deduce the success of every method that the Match class relies on, as if any did not function correctly, the match would not be equal to the oracle.

Unit testing will be used for all cases, where each individual component is tested to ensure correctness.

By using black box testing on the GUI a user will carry out all the given tests with the test values being recorded in the next chapter. This will allow the team to examine the functionality of the application without examining its internal structure. The inherent nature of the GUI makes the use of black box testing more appropriate. White box tests will utilize the JUnit suite embedded in the Eclipse IDE; this will allow the team to test the internal structure and workings of the application using a set of written tests checking for correct and incorrect returned values.

To begin this procedure we will first need to test the scaffolding in which the simulation test stands, this entails the ant brain parser and world parser classes.

2.1 Phase 1 & 2 Model Test Plan (White Box)

Test No.	Method	Method dependencies	Test description
1	The following tests prove the functionality of the method called when reading an ant brain. AntBrainReader.ReadBrain(File)	The method tested here only relies on a collection of private methods.	This test involves reading a correct ant brain from a file and comparing it to an oracle. This oracle is an array of different State objects equivalent to the states in the file parsed by the brain reader.
2			The following tests involve reading a false ant brain from a file and checking the output of the reader. The false ant brain will contain negative values.
3			The false ant brain will contain false states.
4			The false ant brain will contain negative values.
5			The false ant brain will contain an incorrectly spelt state.
6			The false ant brain will contain extra characters at the end of a line.
7			The false ant brain will contain an incorrect sense condition.
8	The following tests prove the functionality of the method called when reading a world. WorldReader.ReadWorld(File)	The method only relies on a collection of private methods.	This test involves reading a semantically correct world from a file and comparing it to an oracle. This oracle is a two dimensional array of different Tile objects equivalent to the tiles in the file parsed by the reader. This world will contain one red ant hill and one black ant hill, clear tiles, rock tiles and also tiles with a given number of food.

9			<p>The following tests involve reading an incorrect world from a file and checking the output of the reader.</p> <p>The false world file will contain no ant hills.</p>
10			<p>The false world file will contain more red ant hills than black ant hills.</p>
11			<p>The false world file will contain the value -1.</p>
12			<p>The false world file will contain the value 0.</p>
13			<p>The false world file will contain the value 10.</p>
14			<p>The false world file will contain a character.</p>
15			<p>The false world file will contain a gap in the north wall.</p>
16			<p>The false world file will contain a gap in the south wall.</p>
17			<p>The false world file will contain a gap in the east wall.</p>
18			<p>The false world file will contain a gap in the west wall.</p>

2.2 Phase 3 Model Test Plan (White Box)

The third phase will use the oracle mentioned previously, this oracle consisted of a snapshot of the world after each round; where a snapshot consists of a list of every tile in the world and all the data related to it. To test against this oracle, I created a parser to read the file line by line and compare a tile created from the data read from the dump file with the relative tile in the world. The test was also progressing the Match model by a single round (every ant being simulated once) after every tile has been checked. Below is a section of the oracle when the match is in its initial state, as an example.

After round 0...

cell (7, 0): rock

cell (8, 0): rock

cell (9, 0): rock

cell (0, 1): rock

cell (1, 1): 9 food;

cell (2, 1): 9 food;

cell (5, 4): black hill; black ant of id 12, dir 0, food 0, state 0, resting 0

cell (6, 4): black hill; black ant of id 13, dir 0, food 0, state 0, resting 0

cell (7, 4): black hill; black ant of id 14, dir 0, food 0, state 0, resting 0

cell (8, 4): black hill; black ant of id 15, dir 0, food 0, state 0, resting 0

cell (9, 4): rock

cell (0, 5): rock

cell (1, 5): red hill; red ant of id 16, dir 0, food 0, state 0, resting 0

cell (2, 5): red hill; red ant of id 17, dir 0, food 0, state 0, resting 0

cell (3, 5): red hill; red ant of id 18, dir 0, food 0, state 0, resting 0

3 Test Procedure & Results

A set of tests will be carried out as discussed in the previous chapter on all phases with a record of the tests results. Each table will have the columns:

- *Test Number*. The test carried over from the previous chapter.
- *Expected results*. This specifies what characterizes a successful test.
- *Results*. The actual results of the test.
- *Status*. Whether the test was successful or not.
- *Action*. If the test was unsuccessful, what corrective action should be carried out.

Iterations of the source code as discussed in the first chapter may be made throughout testing, leading to the simultaneous completion of both deliverables. As noted in the scope, a full list of successful tests will signal that the source code is complete and ready to be shipped with evidence of its correctness.

The previous test descriptions and method dependencies have been omitted from this chapter as they're identical to those in the previous chapter.

The complete set of tests are accessible in the source code via the test source folder and can be executed through JUnit.

3.1 Phase 1 & 2 Model Test

No.	Expected results	Results	Status	Action
1	The following states as objects: Drop 1 Flip 1 1 3 Mark 1 1 Move 1 2 PickUp 1 2 Sense Ahead 1 2 Foe Turn Left 1 Unmark 1 1	The expected result was returned as it matched the manually created oracle.	PASS	None required
2	The read method to return null.	Null was returned	PASS	None required
3	The read method to return null.	Null was returned	PASS	None required
4	The read method to return null.	Null was returned	PASS	None required
5	The read method to return null.	Null was returned	PASS	None required
6	The read method to return null.	Null was returned	PASS	None required
7	The read method to return null.	Null was returned	PASS	None required
8	The following tile representations as objects: 5 5 # # # # # # + . - # # 1 . 2 # # . . . # # # # # #	The expected result was returned as it matched the manually created oracle.	PASS	None required
9	The read method to return null.	Null was returned	PASS	None required

10	The read method to return null.	Null was returned	PASS	None required
11	The read method to return null.	Null was returned	PASS	None required
12	The read method to return null.	Null was returned	PASS	None required
13	The read method to return null.	Null was returned	PASS	None required
14	The read method to return null.	Null was returned	PASS	None required
15	The read method to return null.	Null was returned	PASS	None required
16	The read method to return null.	Null was returned	PASS	None required
17	The read method to return null.	Null was returned	PASS	None required
18	The read method to return null.	Null was returned	PASS	None required
19	The read method to return null.	Null was returned	PASS	None required

3.2 Phase 3 Model Test

Our initial test of the model with the supplied oracle was not successful as it failed on the 658th round. This was the result of an ant executing a flip state but entering the wrong state as a result. I managed to deduce that this was through the fault of a rounding error in the random number generator and required a simple change of casting to a double for division instead of a float.

After this change, we re-ran the test and it passed successfully.

3.3 Black Box Testing the GUI

This section will focus on a simple black box test on the GUI, checking each panel and checking the outcome is as expected. The test will be a simple traversal of the panels since the previous chapters has asserted the functionality of the backend.

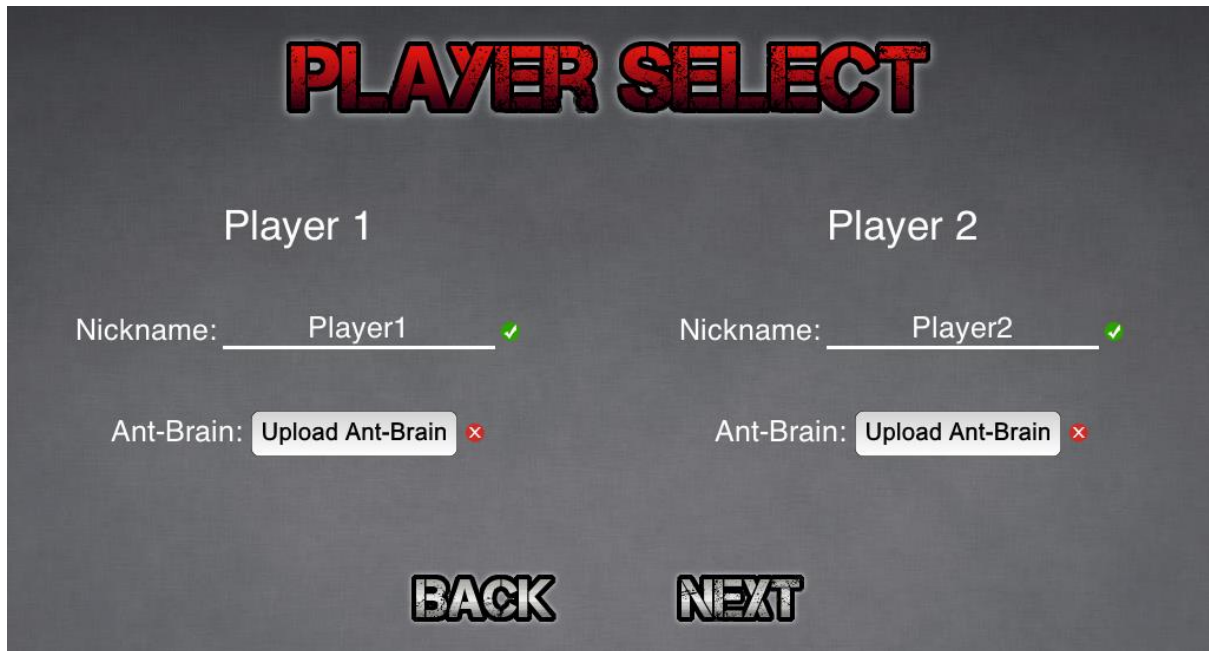
Note: Clicking back on any of the screens will successfully take you to the previous panel with any of the data you entered previously.

Once the program is executed, the user is presented with the main menu panel:



Figure 3 Main menu Panel

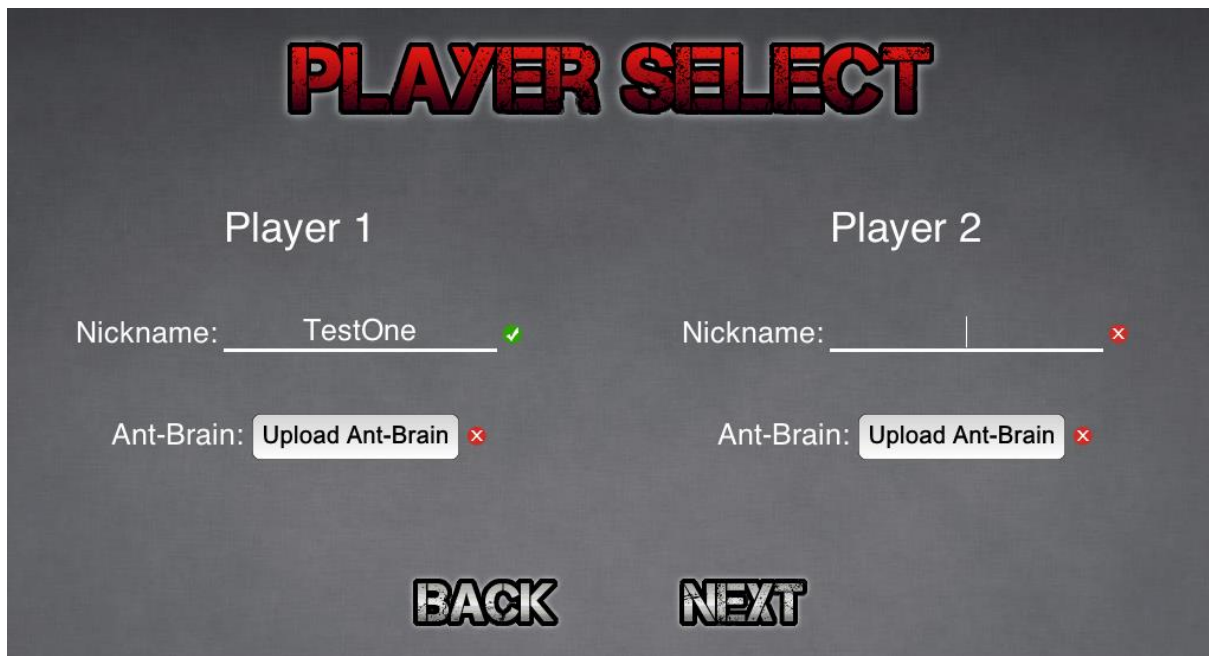
Traversal of the Single Match selection shows the player select screen.



The screenshot shows a 'PLAYER SELECT' screen with a dark grey background. At the top, the title 'PLAYER SELECT' is in large, red, stylized letters. Below the title, there are two columns for 'Player 1' and 'Player 2'. For Player 1, the 'Nickname:' field contains 'Player1' with a green checkmark to its right. Below it, the 'Ant-Brain:' field has a button labeled 'Upload Ant-Brain' with a red 'x' to its right. For Player 2, the 'Nickname:' field contains 'Player2' with a green checkmark to its right. Below it, the 'Ant-Brain:' field has a button labeled 'Upload Ant-Brain' with a red 'x' to its right. At the bottom of the screen, there are two large, bold, black buttons labeled 'BACK' and 'NEXT'.

Figure 4 Player Select Screen

Here the user is allowed to enter their player names and upload an ant brain. Entering the a nickname will show a tick next to the name to signify it's legal and the user can proceed. An blank nickname is illegal and the cross signifies that.



This screenshot is similar to Figure 4, but it demonstrates nickname validation. For Player 1, the 'Nickname:' field contains 'TestOne' with a green checkmark to its right. For Player 2, the 'Nickname:' field is empty, indicated by a vertical line, and has a red 'x' to its right. The 'Ant-Brain:' fields and buttons for both players remain the same as in Figure 4. The 'BACK' and 'NEXT' buttons are also present at the bottom.

Figure 5 A legal and non-legal nickname

Clicking 'upload Ant-Brain' shows the operating specific file selector as shown:

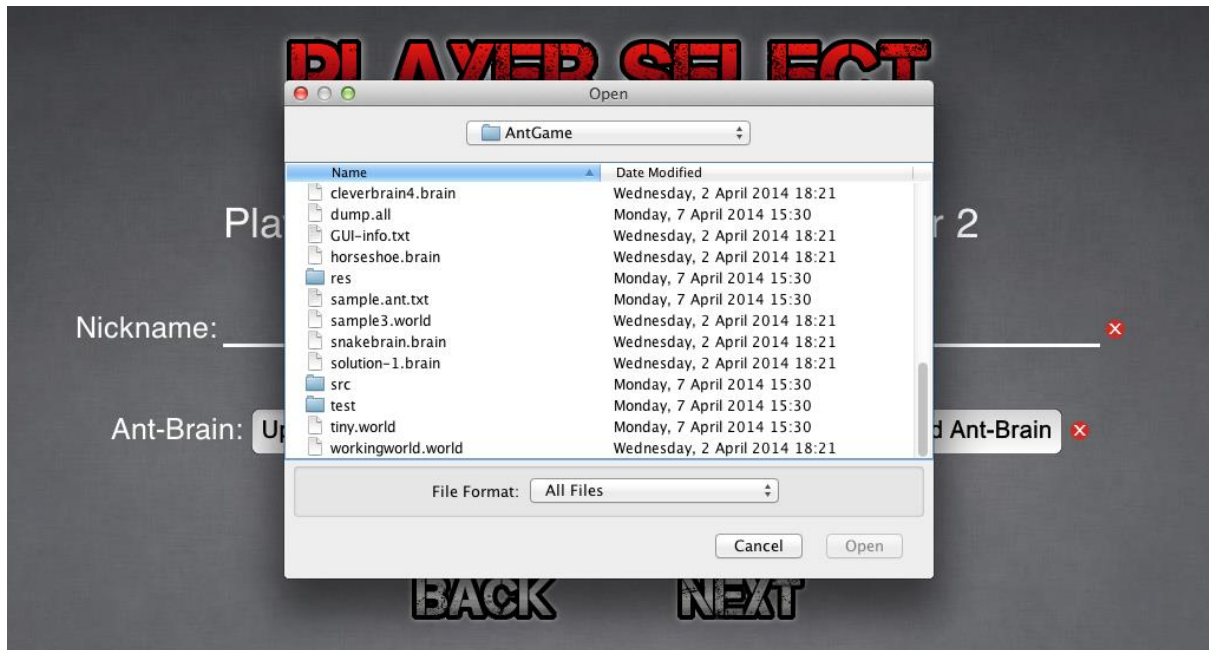


Figure 6 The File selector for the brain

The following shows the same legal state monitor for the brain input, the left is a legal brain, the right, for the sake of clarity will be a world file, and thus will show a cross stating that the user cannot proceed until the input is valid.



Figure 7 Legal and non-legal brains

Entering a legal brain and clicking next shows the world selector pane.

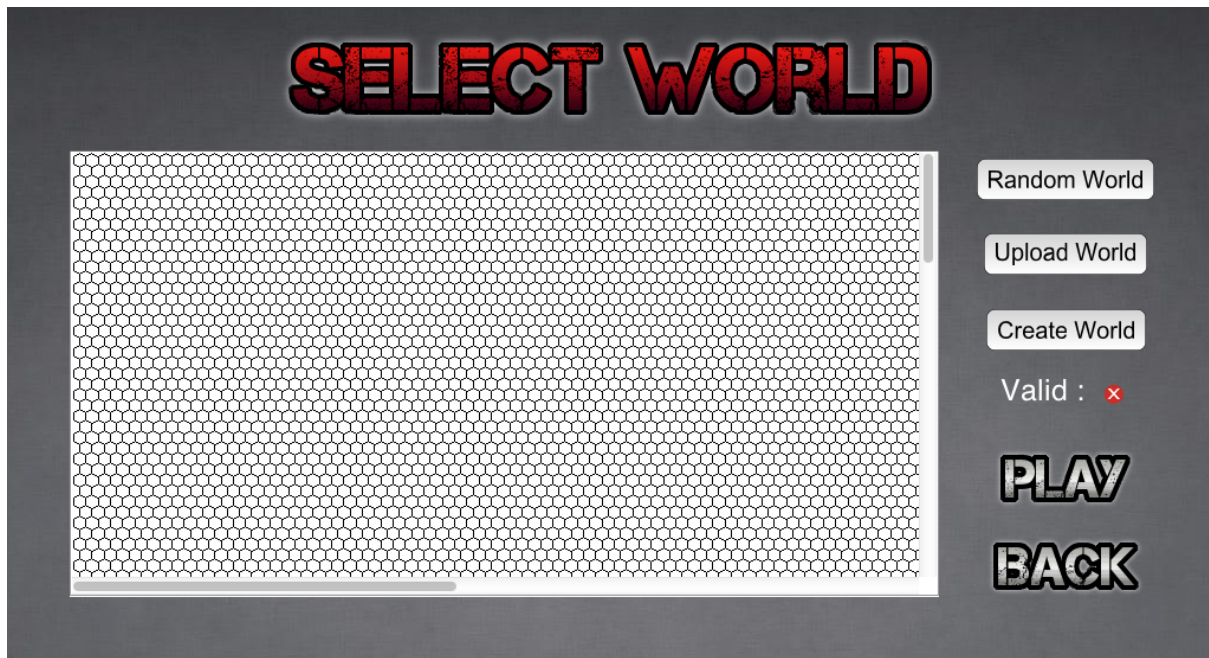


Figure 8 World selector

Clicking random world successfully cycles through random worlds. Additionally similar to the brain selector on the previous pane, the user can upload a world. A cross/tick signifies the legality of a world should the user upload an invalid file.

Clicking 'create world' shows the world editor pane.



Figure 9 World Editor

The user can enter any parameter they desire, in the following case the parameters are shown.



Figure 10 Custom world

Clicking next will take you to the select world screen. Clicking Play will start the game and take you to the following pane.

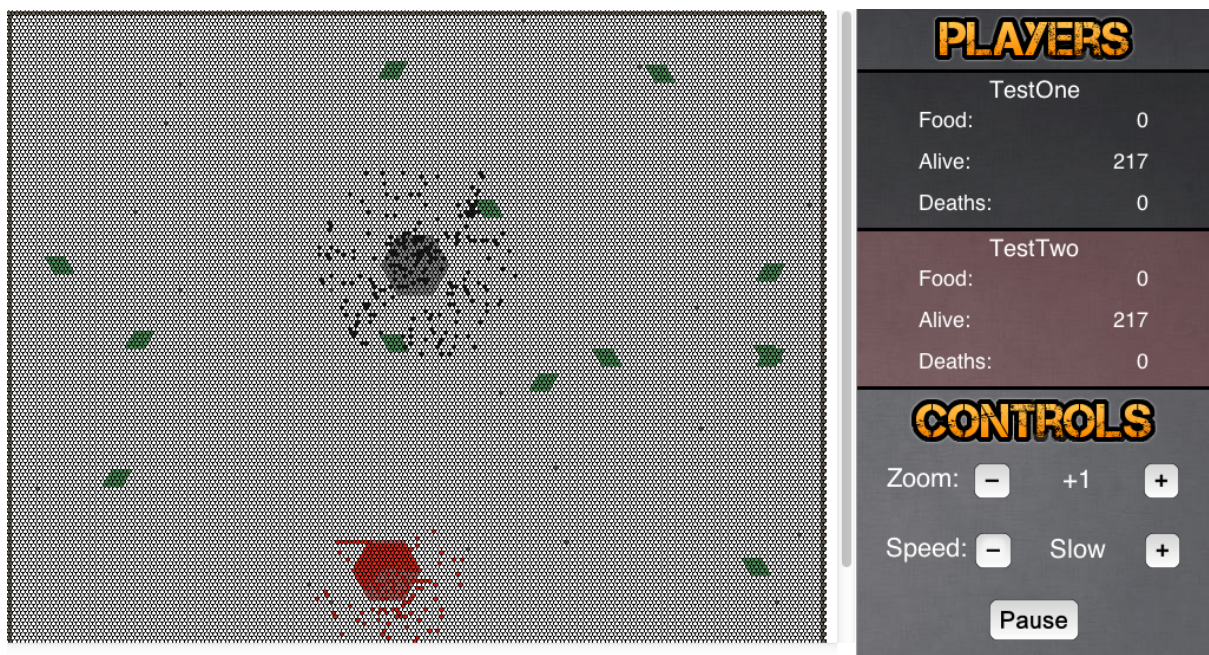


Figure 11 The game

Here the simulation is updated consistently and correctly according to the speed on the right. The zoom buttons work properly as shown.

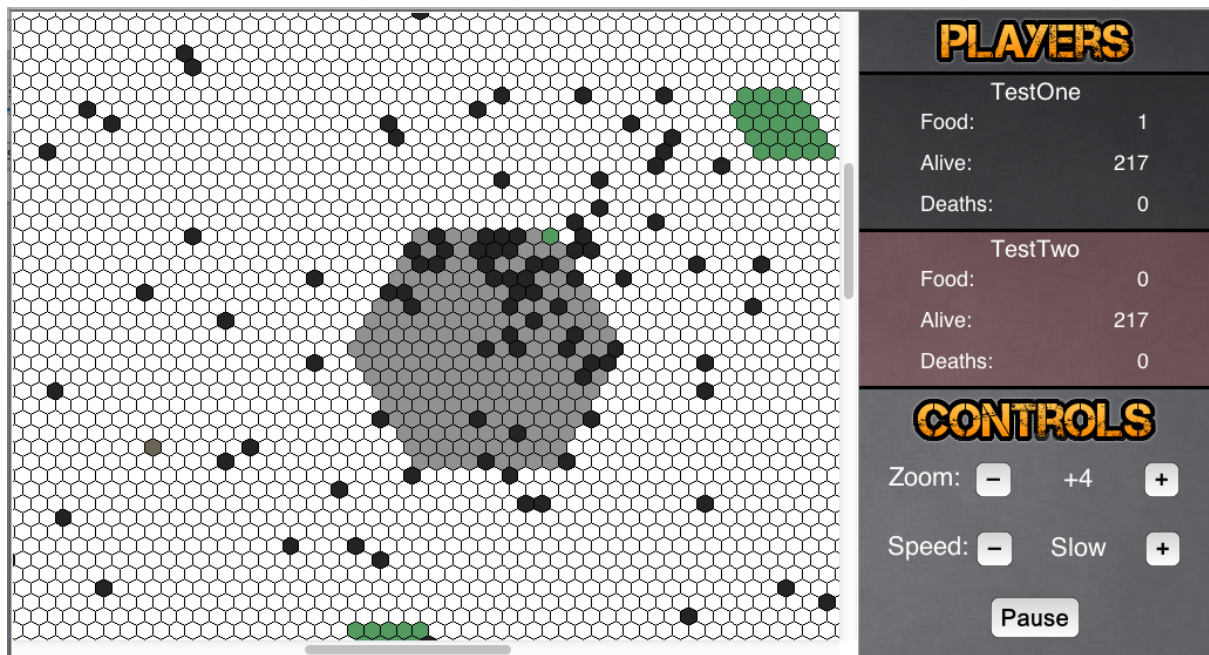


Figure 12 Zoomed in

The game parameters are updated consistently and correctly on the 'Heads Up Display' shown on the right. For a swift match, the user can put the speed on 'fastest' and wait shortly for completion.

At completion, the user is automatically shown the results page.



Figure 13 Scores Pane

Here the parameters are shown correctly. Various playthroughs through the source code phase shows that the winner is always shown correctly, additionally draws are shown here. The user is then presented with the main menu button, additionally the user can close the program or select main menu. Clicking main menu takes the user back to main menu.

Now we can test the Tournament. Clicking Tournament shows the following screen.

PLAYER SELECT

Nickname: ✓ Ant-Brain: ✗

Number of Worlds:

Figure 14 Tournament Pane

Similar to before the user can enter a nickname and upload a brain. The appropriate legality indicator is shown and works correctly. Back works correctly as it does on all panes. The user can add as many players as they desire, the user can also update the number of worlds accordingly.

PLAYER SELECT

Nickname: ✓ Ant-Brain: ✗

PlayerTest! (horseshoe.brain)	<input type="button" value="x"/>
CallMeMaybe (snakebrain.brain)	<input type="button" value="x"/>

Number of Worlds:

Figure 15 Player entry.

The user can at this point delete any players that have been put into the game and re-enter as they wish. Next the Play button is pressed.

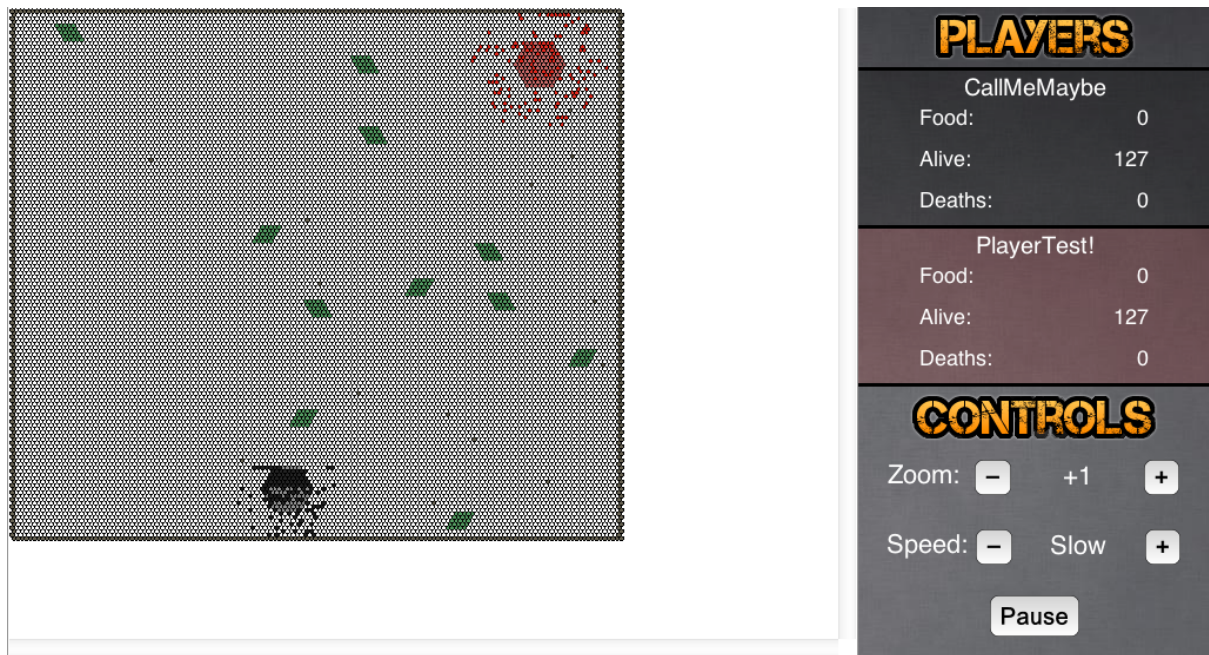


Figure 16 Game screen

Similar to before, the simulation screen is now shown. Clicking and unclicking pause shows the game can be successfully paused. All parameters are updated accordingly.



Figure 17 Scores Pane

As before the necessary results are shown, but the user is asked to enter into the next match, since this is a tournament. Clicking next match shows the following.

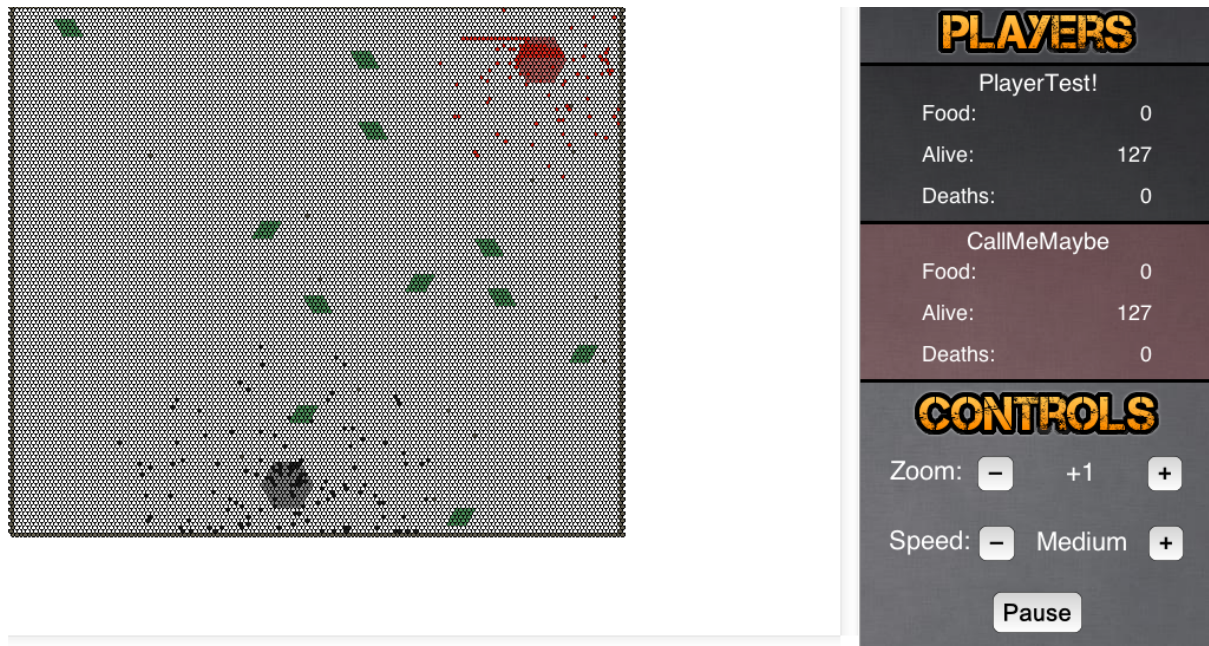


Figure 18 Next tournament game

This cycle is continued until the final match in the tournament is played.



Figure 19 The final results pane

Clicking view results shows the following.

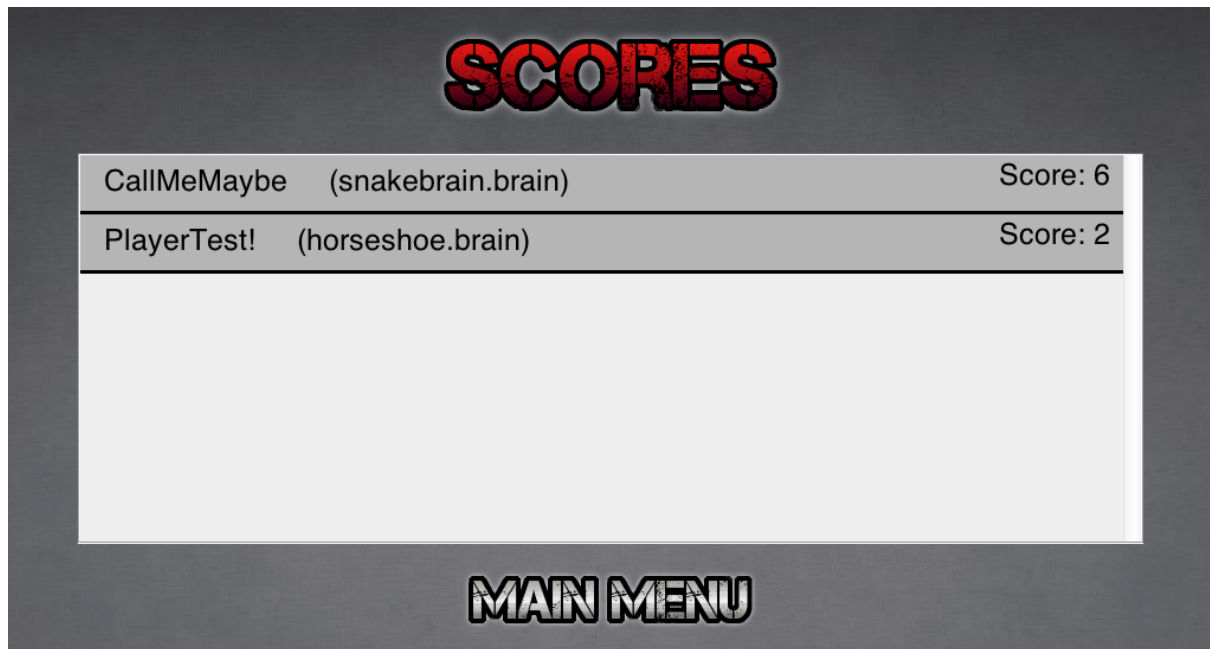


Figure 20 Final score pane

Clicking main menu then takes you back.

This concludes the simple black box testing routine as all functions that have been implemented have been checked for validity. Tests show the program works as it should with no errors.