SE 3XA3: Test Report
GrateBox

Team 8, Grate
Kelvin Lin (linkk4)
Eric Chaput (chaputem)
Jin Liu (liu456)

December 7, 2016

Contents

List of Tables

List of Figures

This document will make frequent reference to the Development, Test Plan, SRS and Design documents of the GrateBox project. They can be found here, here, here, and here respectively.

1 Functional Requirements Evaluation

Functional requirements were evaluated through automated and non-automated testing.

1.1 Automated Testing

Automated testing for GrateBox was conducted using the QUnit software outlined in the Development document. The test cases executed can be found in the test.js file here. Each test case in the test.js file corresponds with an automated test case in the Test Plan document. The automated test cases in the Test Plan document are those found in sections 3.1.1 and 3.1.2. All automated test cases were properly implemented and executed, and the results returned were the results expected, indicating successful implementation of functional requirements by automated testing.

1.2 Non-Automated Testing

Non-Automated testing for GrateBox's functional requirements was conducted by the test team outlined in the Test Plan document. These include the tests outlined in sections 3.1.3, 3.1.4, and 3.1.5 of the Test Plan document. All test cases were exectued correctly and returned the results expected. They are broken down into more detail below.

Table 1: **Revision History**

Date	**Version**	**Notes**
Dec 6	1.0	Document creation
Dec 7	1.1	Document completion

1.2.1 Graphics

Test for graphics are as follows.

GR-1.1

Cars are generated as expected given numerical values. The position of vertices, the connections between vertices, the radius of wheels, and the placement of wheels all vary depending on given input. Comparison with BoxCar-2D also verifies successful implementation of graphics module. Test successful.

GR-1.2

Cars are not generated as expected given invalid numerical values. An error message is displayed when this occurs. Test successful.

GR-1.3

Cars are not generated as expected given invalid numerical values. An error message is displayed when this occurs. Test successful.

GR-2.1

Road created corresponds to algorithm input. Test successful.

GR-2.2

No road created and error message is displayed when this occurs. Test successful.

1.2.2 Fitness and Score

Test for fitness and scores are as follows.

FI-1

Values calculated correspond to values observed. Test successful.

FI-2

Values calculated correspond to values observed and value properly displayed in GUI. Test successful.

1.2.3 Other GUI elements

Test for other GUI elements are as follows.

GU-1

Health bars operate properly. Test successful.

GU-2

Text file is created and is accurate. Test succesful.

2 Nonfunctional Requirements Evaluation

2.1 Nonfunctional Tests

The exact details of nonfunctional requirements can be found in the Test Plan document in section 3.2.

2.1.1 Look and Feel

LF-1

Majority of users agreed that the visual aesthetic of the program rated favourable. Test successful.

LF-2

Majority of users agreed that the style of the program rated favourable. Test successful.

2.1.2 Usability

US-1

Users performed all tasks in alloted time. Test successful.

US-2

Users performed all tasks in alloted time. Test successful.

US-3

Majority of users agreed that the program's usability rated favourable. Test successful.

2.1.3 Performance

PF-1

Time restriction for tasks performed met. Test successful.

PF-2

Majority of useres agreed that the program's usability rated favourable. Test successful.

PF-3

Numerical values and equations determined to be accurate and valid. Test successful.

PF-4

Majority of users agreed that the program's usability rated favourable. Test successful.

2.2 Overall Evaluation valuation based on tests and user survey

Overall evaluation of non-functional requirements takes into account the results of the final user surveys and the results of the above test cases. The nebulous nature of non-functional requirements means that exact evaluation is difficult. With this in mind, the GrateBox project achieved most of the goals set out by non-functional requirements. GrateBox looks and feels good, is highly usable, and performs as expected. All fit criterion were met or exceeded and users agreed that our product excelled non-functionally. Stress testing, while not directly applicable to GrateBox (GrateBox is a simple program executed by one machine by one user), has been undertaken indirectly through a 2 hour execution of GrateBox on two different machines, demonstrating the robustness of the program.

3 Comparison to Existing Implementation

The original implementation contained no testing of its own. One test conducted by Grate (GR-1.1) required a comparison to the original implementation. This comparison helped validate the results of the test as seen in section 1.2.1.

4 Unit Testing

All unit testing for this project can be found in the test folder found here. The tests.java file contains the source code for all testing. The Test.html file contains the output of this code. All test were conducted using the third party testing software QUnit, outlined in the Design document.

5 Changes Due to Testing

Several changes were made to GrateBox as a result of testing. They have been divided below according to their related requirements.

5.1 Changes to functional requirements

Initial car design was predicated on the creation of vectors from a centre point and then the connecting of the end of these vectors to form a polygon that would serve as the body of the car. However, testing for car creation using this method prooved prohibitively difficult. To this end, car bodies were redesigned to be composed of a series of conducted vertexs. This change also effected how car chromosomes were created and manipulated.

User surveys revealed the need for a pause button in GrateBox as users felt a lack of control over the program at times. This was added to the parameters section of GrateBox.

The initial values of the user variables (number of cars per generation, number of parents, and mutation rate) all increased as a result of user input. The initial values resulted in cars that were too slow in evolving, and would often require ten or more generations for a moderately successful car to emerge.

5.2 Changes to non-functional requirements

User feedback told Grate the the UI of GrateBox was too minimal, and that look and feel requirements were not being met. To this end, an html visual enhancer, BootStrap, was used to fufill this user need. Users opinions' of GrateBox's look and feel increased dramatically following this change.

It was observed during user trials that many users had trouble grasping the exact nature of genetic algorithms quickly. To this end text was added to GrateBox to give users the most very basic background with which to use the program. It was important to Grate that this text not be overly extensive, as it was believed that this could go a long way to turning users off of our product. The exact amount and content of text was modified extensively over GrateBox's development cycle via user input.

6 Automated Testing

Automated testing prooved difficult for certain elements of GrateBox, as it is by nature a visual product. Still some automated testing was conducted

for the benefit of the testing team and to improve accuracy. QUnit, our unit testing software, allows for mutliple executions of the same test with some variances, and this was done to ensure the soundness of many aspects fo GrateBox. This functionality allows for the entrance of a variable into a field with the instructions to manipulate that variable within a range of possible values and determine many possible outputs (for example negative values, non integer values, etc.). The genetic algorithm in particular benefited greatly from automated testing, as many different variables could be tested with minimal time investment. While it proved impractical for the project given time constrictions, further elaboration on GrateBox could utilise image analyzing on a pixel by pixel basis to improve the quality of the graphical output. Although the overall value of such a time investment is questionable.

7 Trace to Requirements

8 Trace to Modules

9 Code Coverage Metrics

In the Test Plan document, Grate endevoured to achieve 70% coverage with our automated and unit testing. While not every unit test possible was executed, most were, and Grate feels that this coverage metric was reached, as a large majority of our actual code contains corresponding tests to validate it. The modularized nature of our code reduced the overall usefulness of larger coverage tests. For example, the inclusion of the genetic algorithm in a test for graphical validity achieves very little, as the modularized nature of the code requires little to no interaction between these two modules.