SE 3XA3: Test Report
GrateBox

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

December 8, 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

An evaluation of functional requirements reveals near complete coverage by the functional test cases. The tests written for GrateBox caught bugs and erros that may have gone otherwise unnoticed by the development team. Those will be discussed below. Many of the more mundane or generic functional requirements were fulfilled well before testing began, so they will not be addressed, as their verification would be redundant at this point.

Functional requirements are listed by the number given to them in the Software Requirements Specification document. Note that these functional requirements are laid out in more detail in the SRS document linked above, and it should be referred to for further elaboration.

Requirement #5/6: Testing revealed inconsistent masses across similar cars in different generations. This was revealed to be a conceptual problem with the code. Initially, mass was treated as a fixed value common to all cars. While this technically fulfilled the functional requirements for mass, later generations would trend towards the extremes of and eventually past the range of viable mass values dictated by the requirements. The development team elected to remedy this problem by making the mass value of cars a variable dependant on density and volume. With a constant density value set across all cars, and a much stricter range of possible volumes that cars could possess, mass values in test cases after this fix fulfilled these mass requirements far more consistently.

Requirement #8: While testing did not reveal a problem with the actual display of fitness values required by this requirement, it did reveal a flaw in the creation of these values. Initially, the fitness of a car was determined by the distance along the x-axis of the road that the car had traveled. This solution did not account for vertical displacement, which became ever more important as GrateBox incorporated more sloped roads, to the point where many cars that were technically traveling a shorter distance along the road were receiving higher fitness values, skewing the results and threatening the integrity of GrateBox itself. The development team was quick to implement a more nuanced fitness algorithm that rated cars off of both their vertical and horizontal displacement and also accounted for the lifetime and

Table 1: **Revision History**

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

durability of the car.

Requirement #10/11: Whenever the user enters a new value into any of the fields that are able to be changed by the user, a new simulation begins with those values. While this seemed obvious to the development team, user surveys revealed that this process confused users. To this end, the functionality of these user changeable fields was updated to require the user to select enter following the entrance of a new value into the field. The following user feedback revealed that this change in functionality resulted in a better product.

Requirement #13/14: Testing revealed that many users felt the need to enter impossible values into user changeable variables. For example, many users attempted to simulate generations of cars with negative amounts of cars per generation. Others simply attempted to enter strings of text into these fields. GrateBox, at the time incapable of processing these invalid inputs, failed as a result. The development team quickly implemented error messages and try/catch exceptions into the code in order to thwart user attempts to circumvent the functionality of the program.

Requirement #18: The algorithm that was designed to determine at what exact point along the road a car was deemed to be non-moving evolved over the course of the testing process. The first set of tests illuminated the need for larger health bars for each car to account for some cars stopping temporarily along the road as they attempted to circumvent terrain challenges. This was followed by a number of bugs in Java's graphics libraries, which caused cars to be deemed non-moving far before the graphics indicated this. These bugs were eventually ironed out of the program.

2 Nonfunctional Requirements Evaluation

An evaluation of non-functional requirements reveals near complete coverage by the non-functional test cases. The tests written for GrateBox caught problems and potential improvements in GrateBox that were resolved by the development team. Those will be discussed below.

Non-functional requirements are listed by the number given to them in the Software Requirements Specification document. Note that these non-functional requirements are laid out in more detail in the SRS document linked above, and it should be referred to for further elaboration. While many non-functional requirements required only software test to validate, many required timed tests by user volunteers and user feedback collected through surveys. The exact details of the survey can be found in the Test Plan document linked at teh beggining of this document.

2.1 Look and Feel Requirements

Requirement #21: The need for GrateBox to be attractive to a student audience was of crucial importance to GrateBox's success. Initial releases of GrateBox however were met with hostility by users surveyed. To this end, another look was taken at technologies necessary to aid in increasing the aesthetic appeal of GrateBox. BootStrap was chosen as a result of this feedback and was incorporated into GrateBox. User surveys following the update revealed a marked increase in user support of the product's appearance.

Requirement #22: The importance of maintaining an education and professional outlook was identified early, and Grate did its upmost to ensure that this style was projected by the product, and it was felt that the program did in fact project this style during initial releases. While this sentiment was shared by most users surveyed, many also recommended that the look and feel of GrateBox could be improved by enabling additional options for the user to reduce the confusion the original releases could project. In particular, many users requested the ability to pause the simulation to allow them to utilise GrateBox at their own pace rather than a pace set by the program. With the goal of reducing confusion, a pause button was added, along with a clearer input method for user changeable parameters

2.2 Usability and Humanity Requirements

Requirement #23: Being an educational tool, it was of the upmost importance that GrateBox be easy to use. To this end, a set of tasks was set before users to be completed within a set time frame, to judge the program's ease of use. While most users were able to complete the tasks in a timely manner, some were not. These users had problems understanding exactly what fitness values were in practical terms, and so could not utilize the program to the fullest. To remedy this, Grate incorporated a more detailed and nuanced explanation of the program's fitness value, and the program's ease of use increased greatly as a result.

Requirement #24: If a learning tool is to be successful, it must allow for some degree of user engagement, and Grate aimed to accomplish this with GrateBox by allowing for a degree of personalization by the user. The mutation rate, population size, and number of parents of each set of generations was fully customizable by the user from the very earliest releases, however while users could update these values and see the changes in the simulations that resulted from these updates, many could not grasp exactly what the changes they were making were changing in real terms. To this end Grate incorporated written explanations of each user changeable field to help the educational process GrateBox attempted to induce.

Requirement #25: It was always intended for GrateBox to be composed of only a single web page with very simple commands so that even users with no experience with the program could operate it. Our testing revealed that users were able to easily learn how to use the products minimal functionalities. While some users had problems understanding the exact effects of these functionalities (see requirements 23 and 24), none were incapable of

operating them.

2.3 Performance Requirements

Requirement #26: In order to maintain user attention, the speed of GrateBox had to be such that the user would not grow board of waiting for the product. Manual testing by the testing team revealed that the program did in fact meet or surpass our metrics for speed defined in the requirements, and most users surveyed agreed with this assessment. Testing the speed of the program however revealed the need to remove a significant amount of useless code from the API GrateBox utilized in order to further reduce GrateBox's runtime.

Requirement #27: As a learning tool, GrateBox's precision was of the upmost importance. If the user believes that an educational program they are using is inaccurate, they will likely doubt the veracity of what is being taught to them. To this end, the product was made to display all values to the precision outlined in the requirements, however testing demonstrated several accuracy errors in final calculations of values. These resulted from the use of both integers and floats in the same mathematical equations. These errors were fixed in the final product. The actual testing process also revealed the need to switch our sorting method from a heap sort to a quick sort, as heap sorting methods proved overly difficult to verify with automated testing.

2.4 Robustness, Maintainability, and Compatibility

Robustness: While no formal robustness testing was conducted by Grate, several informal robustness tests were conducted over the lifetime of the development cycle. The most important of these were tests run to simply see how many generations GrateBox could run for before encountering errors. Grate was very thankful these tests were undertaken, as memory leak issues were encountered resulting in the sudden collapse of GrateBox after 4 generations of automobiles. The source of these errors was tracked to the API. Following extensive modification of elements of GrateBox's API, the program now runs for a significant amount of time before encountering problems, long enough that only the most dedicated and fanatic of users could possibly encounter them.

Requirement #28: The need for GrateBox to be easy to update and upgrade was made clear before Grate even began the development process, as it was believed making this a priority early on in the development process would make the development process itself all the more easy. Ultimately, informal discussions with other software developers and internal testing saw the almost absolute removal of constant values in the GrateBox code. These were replaced with variable values, greatly adding to GrateBox's longevity as a software product.

Compatibility: One benefit of programming in such a common language as JavaScript was the overwhelming compatibility of the programming language, and the use of HTML

for the final website design also aided in improving GrateBox's compatibility. No compatibility problems were encountered during the testing process for and device with JavaScript installed, verifying the fantastic compatibility of GrateBox.

2.5 Legal Requirements

While no legal requirements existed for this project, as all software utilized was fully open sourced, a software license was created by the recommendation of a surveyed user to be safe.

2.6 Health and Safety Requirements

While no formal health and safety testing was undertaken, as the health and safety risks of this project were deemed to be too minimal to warrant such testing, users were informally asked throughout the survey process if they had concerns regarding the GrateBox project when it came to health and safety. No users had any concerns. While no significant health and safety risks could be identified, Grate made the effort to identify a set of potential user problems given extreme conditions (this list can be seen in section 4.3.3 of the test plan).

3 Comparison to Existing Implementation

The original implementation contained no testing of its own. This is somewhat understandable given the seemingly overnight creation of the original product. As such Grate's test cases have nothing to compare to in the original implementation. Were there such tests to compare to, Grate would compare the coverage, results, and random inputs of the existing implementation's test cases to those of Grate. Grate would also search for missing elements of one implementation that are present in the other, to see if Grate's implementation has missed a crucial component of the existing implementation.

One test conducted by Grate (GR-1.1) required a visual comparison to the original implementation. This comparison helped validate the results of the test. The test in question was GR-1.1 on the Test Plan document.

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.

Most of the unit testing was focused on ensuring the validity of the genetic algorithm and the car creation and manipulation elements of GrateBox, as graphical aspects of GrateBox

proved far more difficult to test with unit testing.

Overall, these unit tests served to guarantee a level of functionality across builds, and did not account for non-functional requirements, as these were better suited to manual testing with user feedback.

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. Many changes were made on a requirement by requirement basis, and these can be seen in sections 1 and 2 of this document.

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 proved prohibitively difficult. To this end, car bodies were redesigned to be composed of a series of conducted vertexes. This change also affected 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 that 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 satisfy 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

6.1 Introduction to automated testing

Automated testing proved 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 multiple executions of the same test with some variances, and this was done to ensure the soundness of many aspects of 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 analysing 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.

Outline below are all of our individual test cases, along with their names, initial states, inputs and expected outputs. A test was deemed to be successful if the expected output was the actual output. Only successful tests are listed here.

6.2 Specific System Tests

Name:	Placeholder
Initial State:	Placeholder
Input:	Placeholder
Expected Output:	Placeholder

Name:	
Initial State:	
Input:	
Expected Output:	

7 Trace to Requirements

The requirements are described in more detail in the SRS document. The tests are given by their abbreviated forms. For example GA refers to the Genetic Algorithm tests, which are described in more detail in the Test Plan document.

Req.	**Tests**
Req 1: Car body parameters	CM
Req 2: Wheel number parameters	CM
Req 3: Wheel radius parameters	CM
Req 4: Wheel position parameters	CM
Req 5: Min weight parameters	CM
Req 6: Max weight parameters	CM
Req 7: Generation display parameters	GR
Req 8: Fitness display parameters	GA, FI
Req 9: Random seed parameters	GA
Req 10: Mutation rate parameters	GA

Table 3: Trace between functional requirements and tests (set 1)

Req.	**Tests**
Req 11: Cars per generation parameters	GA
Req 12: Road generation parameters	CM
Req 13: Min cars per generation parameters	CM
Req 14: Max cars per generation parameters	CM
Req 15: Top cars parameters	CM, FI
Req 16: Max top cars parameters	CM, GA, FI
Req 17: Min top cars parameters	CM, GA, FI
Req 18: Non-moving parameters	CM, GU
Req 19: Fitness parameters	FI
Req 20: Default value replacement parameters	GU

Table 4: Trace between functional requirements and tests (set 2)

Req.	Tests
Req 21: Appearance Parameters	LF
Req 22: Style Parameters	LF
Req 23: Ease of Use Parameters	US
Req 24: Personalization Parameters	LF
Req 25: Learning Parameters	US
Req 26: Speed and Latency Parameters	PF
Req 27: Precision and Reliability Parameters	PF
Req 28: Longevity Parameters	PF

Table 5: Trace between non-functional requirements and tests

8 Trace to Modules

For reference, the modules are as follows. There are displayed with more detail in the Design document. All tests are referred to in their abbreviated form. For example Genetic Algorithm tests are referred to by the short form GA. These are elaborated on in the Test Plan document.

M1: Hardware hiding module

M2: Car creation module. Tested with the CM tests.

M3: Evolve car module. Tested with the CM tests.

M4: Road creation module. Tested with the CM tests.

M5: Graphics display module. Tested with the GR and GU tests.

M6: Genetic Algorithm module. Tested with the GA tests.

M7: Random seed generation and manipulation module. Tested with the GA tests.

M8: Fitness determination module. Tested with the FI tests.

M9: Searching algorithms module. Tested with the GA tests.

M10: Sorting algorithms module. Tested with the GA tests.

M11: Population generation algorithms module. Tested with the GA tests.

9 Code Coverage Metrics

In the Test Plan document, Grate endeavoured 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.