Blaze Brigade

- Test Report -

SFWR ENG 3XA3 - Section L02
007 (Group 7)

Jeremy Klotz - klotzjj
Asad Mansoor - mansoa2
Thien Trandinh - trandit
Susan Yuen - yuens2

December 8, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
December 8, 2016	1.0	Completed Test Report

1 Functional Requirements Evaluation

All functional requirements were met. They will be evaluated in detailed sections below:

1.1 GUI

The Graphical User Interface is fully functional and allows the user to navigate through and access all of the game's features using mouse input. A main menu appears upon launching the executable. From the main menu, users are able to select New Game, How-To-Play and Exit Game via mouse input. A player is also able to control a unit and select all of its possible actions from the drop down menu through mouse input. Therefore, all GUI functional requirements have successfully been met.

1.2 Game Structure

A trace of two players with the first player (Player 1) controlling 1 warrior and the second player (Player 2) controlling 1 mage would be as follows:

Player 1 selects a unit with the mouse, and a drop down menu containing Attack, Move, Item, and Wait appears. The player then selects Item from the drop down menu. From the drop down menu in items, the player selects Iron Sword to equip it. The player then selects Move and clicks on a tile within movable range. The unit moves to that new location and the Move button disappears. The player then selects Attack from the drop down menu and selects an enemy unit within attack range, resulting in neither targets dying from the attack. The Attack button then disappears from the drop down menu, and the player selects Wait. Player 1's turn then automatically ends and switches to Player 2. Player 2 selects their unit, selects Attack from the drop down menu, then proceeds to attack Player 1's warrior. The attack kills the warrior and the game ends.

The above trace of the game shows that the game is turn based, and consists of 2 players alternating turns. It also demonstrates that a unit can select from 4 available actions, with the additional requirement that it can only move and attack once per turn, and that wait ends all of the unit's available actions for that turn. Furthermore, the trace shows that when a player has no controllable units left, the game ends. Therefore, all Game Structure functional requirements have accurately been fulfilled.

1.3 Unit Movement

Players are able to select Move from the drop down menu after selecting a unit. All movable nodes are accurately calculated using the breadth-first search path finding algorithm, and are highlighted in blue. Whether or not a node is an obstacle is also considered in the path finding algorithm. The player is then

able to select one of the blue highlighted nodes to move the unit to that location. Therefore, all Unit Movement functional requirements are implemented.

1.4 Unit Attacking

A player can select Attack from the drop down menu after selecting a unit. All attackable nodes calculated from the breadth first search algorithm, which also accounts for weapon attack range, are highlighted red. The player is then able to select an enemy unit on a red highlighted node to attack. Upon successfully selecting an enemy unit, an attack confirmation button will appear along with all relevant attack information. Clicking attack confirmation will execute the action, and result in getting counterattacked (assuming the unit survives the initial attack). The damage displayed in the attack information will then be subtracted from each unit's health, and units with less than 0 HP are removed from the game. The unit that just attacked will then only have the Wait option left to choose from. Therefore, all Unit Attacking functional Requirements are met as intended.

1.5 Unit Structure

The game implements the stats strength, intelligence, defense, resistance, skill, speed, health, and utilizes them for damage calculations. Strength and defense are used in physical damage calculations, intelligence and resistance in magical damage calculations, speed determines if a unit performs a double attack, and skill affects critical and hit rate on an enemy unit. The 3 required units are also implemented and follows the requirements of each unit's stat build as follows: Warriors have high Attack and Defense, Archers have high Strength, Skill and Speed, and Mages have high Magic and Resistance. Each unit also has their own HP bar that updates corresponding to their current HP versus their maximum HP. Therefore, all unit structure requirements are successfully fulfilled.

2 Nonfunctional Requirements Evaluation

2.1 Look and Feel

The Main Menu page background and music successfully conveys a melodic and soothing atmosphere. The rest of the game's visual assets and interface also conveys a nostalgic feel similar to classic pixelated tactical RPGs such as Fire Emblem. Therefore, the Look and Feed requirement are adequately met.

2.2 Usability

All persons asked to test the game with a Windows computer possessing a screen size greater than 960x640 and a pointing device have successfully managed to play the game. Therefore, the usability requirement of the game has been fulfilled.

2.3 Performance

All algorithms and structures used in the game are executed in near instant time, thereby fulfilling speed and latency requirements. All event driven game aspects behave exactly as expected. The game supports exactly 2 players, and no player input to the game can cause it to crash. Therefore, precision, robustness and capacity requirements are met.

2.4 Operational

Following the README on how to run the game, all testers on a Windows PC were successful in finding the location of the game executable, and were able to successfully run the game. Therefore, operational requirements were fulfilled.

2.5 Maintainability

A player is able to access all of Blaze Brigade's intended features from the released executable, therefore not needing any further maintenance. This results in meeting the maintainability requirement.

2.6 Security

The game's code and structure are not able to be modified from any user input while playing the game, thereby fulfilling security requirement.

2.7 Cultural

The game does not contain any content that is found to be offensive to any religious or ethnic group, which results in completing the cultural requirement.

2.8 Legal

The game does not contain any content nor is distributed in any way that conflicts with any law. Therefore it fulfils all legal requirements.

2.9 Health and Safety

The health and safety warning covers the any health complications that could result from playing the game. Therefore, health and safety requirements are met.

3 Comparison to Existing Implementation

The existing implementation of this project is an open-source game called Tactics Heroes. It has helped develop the requirements of this software system. Due to this, Blaze Brigade is very similar to Tactics Heroes. Both games are

tactical, and turn based. The objective of both games is also identical. Each player is assigned a team with a variety of units. Using these units, the players develop a plan in order to eliminate the enemy force. Both games use a grid that is broken into squares that represent the structure of the playing field.

3.1 Differences from Existing Implementation

Although the two systems are similar, there are a few differences. These differences are: animations, unit movement and navigation, sound effects, and a working inventory. Unlike Tactics Heroes, Blaze Brigade features animations that make the game look more fluid. In Tactics Heroes, units are forced to move before making another action. The process is to click the unit, move the unit, then attack or wait or manage items. In Blaze Brigade, the process is to click the unit, decide to move, attack, or interact with the items in their inventory. Tactics Heroes lacked music or any sound effects, which resulted in a quick loss of interest. Therefore, Blaze Brigade incorporates both game music and game sound effects. Lastly, Blaze Brigade has a working inventory of weapons for each unit, another feature that Tactics Heroes did not have. A working inventory will be crucial for future development.

4 Unit Testing

4.1 Unit Testing Tools

The unit tests were implemented using Microsoft's Visual Studio Unit Testing Framework in C#. Our decision to implement the unit tests in Visual Studio was largely impacted by Microsoft's comprehensive support of its unit testing framework for Visual Studio. Another reason was the fact that our unit test suite would also seamlessly integrate into our code, which was already done in Visual Studio in C#. This also aided in diminishing the learning curve required, as the team was already familiar with the IDE and language, further supporting our decision in the testing framework. In addition, Microsoft offered an extensive unit testing guide, complete with tutorials, documentation on test methods, and examples, which further lowered the learning curve.

In addition to Visual Studio's Unit Testing Framework, our unit tests also make use of Moq - a mocking framework. We felt the use of this framework to be necessary in order to allow for a more thorough testing of the game's code as it opened up more possibilities and options within unit testing. A specific example of this would be the Verify() method, allowing for the verification of a function call which would not be possible without the using of a mocking framework. In addition, Moq allows for much simpler and more controlled instantiation of objects and dependencies outside of the unit under test. Due to its ability to cause mock objects to return anything given, mocking allows for easier control on the environment for the unit under test, and expands the

options and conditions of the unit tests.

4.2 Limitations

Due to the nature of the software and tools used in the creation of our game, not all parts of the code are testable with unit testing. Functions pertaining to the view (such as animation, sound, graphics, and game windows) could not be tested, as these functions had high dependency on the XNA Game Studio Framework. Such functions are untestable because the game itself can only be initialized upon running the game, which is impossible in the environment of unit testing. In addition, the complexity of the XNA Game Studio Framework also causes it to be unmockable, eliminating this workaround and rendering testing of this dependency impossible. Other items affected by this include the main game loop, mouse handling, and the camera.

4.3 Unit Testing Approach

The objective of implementing unit tests is to integrate automated whitebox testing of the code through passing controlled input(s) into each function in order to ensure correct behaviour and/or output of the function. As such, multiple unit tests covering multiple test cases were written for each function, which include regular, edge, and abnormal cases. A specific example of such include testing negative, zero, and positive numbers for integer input, as well as null objects for object parameters. This also included null attributes of objects which could possibly adversely affect the function under test. Through this approach, the overall robustness of the system was improved as a result of putting the system under unexpected input, which led to improving upon the system upon undesirable results.

The testing approach was different for dissimilar functions. Functions that returned a value were checked to ensure that the value returned corresponded to the expected value. Void functions were tested to ensure for correct behaviour, such as changes in the parameters it took, changes to the state of the game, or verification of calls to other functions (using the mocking framework).

Thus, unit testing results in lowering errors regarding incorrect coding implementation of logic and the functional requirements. Unit testing also exposes improper software design, as it forces one to modularize one's code enough so that each function acts as a single unit, thus making unit testing easier, less complex, and diminishes the setup of the environment of the unit under test.

5 Changes Due to Testing

Throughout the course of its development, Blaze Brigade has seen some changes and will continue to see more. Specifically, the game has been adapted to meet

user needs and suggestions, based off user survey feedback. Such changes include ensuring proper balancing and general bug fixes.

5.1 Issue Handling

One of the biggest changes made to Blaze Brigade is how it calculated the path of a moving unit. This change occurred due to an issue where units could not reach squares that were within range. To fix this, the algorithm was reworked entirely to implement a breadth-first search algorithm.

5.2 User Feedback

A survey was constructed and presented to a small sample of Blaze Brigade's target audience. The purpose of this survey was to encourage feedback from potential users. This feedback has helped determine what changes should be made to the software system, and is listed below. Note that some changes are meant for the future, as time is a constraint on this project.

Distinguishing Current Player's Turn → A label for which player's turn it currently is was added after some players stated that they did not know whose turn it was.

Determining which units could move → To alleviate the issue of players not knowing which characters they could control on their turn, the following additions were added to the game:

1. Player 1 units will have a blue health bar, while Player 2's units will have a red health bar. This helps the user know which units belong to them.
2. Units will now be greyed out after finishing all available actions for that current turn. This helps the user know which of their units still have available actions to perform.

Game Over Information → Project feedback mentioned that the game over screen did not state which player won. This addition was easily implemented afterwards.

Health Bar → Some players found it hard to plan strategies without quick access to how healthy their units currently are. Since unit HP was only shown if the unit was clicked in the previous implementation of the game, a health bar reflecting how healthy a unit is was added which is displayed above all units.

More Obstacles → One player stated in the survey that more obstacles should be added to give players an incentive to control certain parts of the game map to give them a terrain advantage. This addition was quickly implemented in the form of a small "maze" near the center of the map.

Sounds → Sounds was one of the last additions to the game after feedback mentioned that it would help maintain the attention and interest of players. Therefore, sound effects and music were added into the game.

Magic Weapon Naming → After some players questioned why their character was not on fire after storing a fireball in their inventory, all spells were renamed from SpellName to SpellName Tome.

Unit Balancing → Naturally, each type of unit should have their own strengths and weaknesses. However, Blaze Brigade was initially created without the idea of balancing in mind. Players found that certain unit types would dominate the game and ruin the fun. To fix this, we enabled a triangular relation between the three units. Warriors have high physical attack and defense, but are weak to magic. Archers are fast with high skill but fragile against physical attacks with a decent magical resistance. Mages are weak physically, but excel in magical attack and defense. This results in the Rock-Paper-Scissor like triangle of Warrior beats Archer, Archer beats Mage, and Mage beats Warrior.

6 Automated Testing

The extent of our automated testing includes our unit tests, containing 153 passed test cases across all game code. The unit tests allow for automated whitebox testing, and can be run periodically to ensure that the code is behaving as expected.

Unfortunately, due to the time constraint of the project, no additional automated testing mechanisms have been implemented.

7 Trace to Requirements

The testing process ensures that all requirements that have been previously defined in the Software Requirement Specification are validated through a series of test cases. Since these define the main functionality of the software, testing them will ensure whether the main objectives of the system are met. With accordance to the unit testing approaches taken in the development, actions such as testing for invalid inputs and behaviour as well as the predefined valid test cases were analyzed to ensure the system shows the valid output to the user and is in the correct state of the game internally. In order to achieve this, a test model has been structured to match the corresponding requirements with the specific test cases. Requirements that revolve around the graphical user interface can be matched up with various tests surrounding the gameplay interaction, which may also be associated with various other requirements that have a similar trace to it. The following table 2 entails the requirements set in the Software Requirements Specification with the test cases planned in the Test Plan.

Requirement	Test Cases
FR1	FR-GI1, FR-GI2, FR-GI3, FR-GI4, FR-GI5
FR2	FR-GS1, FR-GS2, FR-GS3, FR-GS4
FR3	FR-UM1, FR-UM2, FR-UM3
FR4	FR-UA1, FR-UA2, FR-UA3, FR-UA4, FR-UA5, FR-UA6, FR-UA7
FR5	FR-US1, FR-US2, FR-US3, FR-US4
—	—
NFR1	FR-GI1, FR-GI2, FR-GI5
NFR2	NFR-UB1
NFR3	NFR-PF1, NFR-PF2
NFR4	NFR-UB1
NFR5	N/A
NFR6	NFR-SC1

Table 2: Trace Between Requirements and Test Cases

8 Trace to Modules

Similarly to the traces of requirements, the system design can be tested in means of analyzing the traces to the modules. The decomposition of the modules were executed during the Module Guide and Module Interface Specification process as a means of defining the design pattern. These modules helped categorize various parts of the system and abstract them in such a way that if any future changes were to occur in the design, it would obstruct the design of other modules and can be implemented independently. Considering on how a test case might verify the correctness of a particular feature, modules are equipped with a larger scope of requirements that can be tested by observing how they interact with other modules or even sub-modules. This is essential due to the fact that a function might output the intended result, but the interaction between other functions within the scope might also help verify the overall design of the system. The following table 3 entails the modules decomposed in the Module Guide and Module Interface Specification with the test cases planned in the Test Plan.

Modules	Test Cases
M1	FR-GI1, FR-GI6
M2	FR-GS1, FR-GS2, FR-GS3, FR-GS4, FR-US1, FR-US2, FR-US3, FR-US4
M3	FR-GI2, FR-GI3, FR-GI4, FR-GI5, FR-UM1, FR-UM2, FR-UM3
M4	FR-UA1, FR-UA2, FR-UA3, FR-UA4, FR-UA5, FR-UA6, FR-UA7

Table 3: Trace Between Modules and Test Cases

9 Code Coverage Metrics

The code coverage metric is essential to check whether the testing process is accurate and complete. Structuring the test cases to specific requirements and modules helps the system to achieve a high code coverage metric to ensure every bit of the software is tested whether it be the means of manual or automated testing. In regards to the number of test files in the BlazeBrigadeTest directory, the scope of these test cases cover the game state, unit specification, objective calculations and graphical representation aspects of the overall game. Within the cases, various subcategories of the test include the process of inserting invalid inputs or behaviours to the function and analyzing how stable the system is within its metric. The game state is also tested to an extent, such to be sure that the gameplay proceeds in the correct manner. As the game is structured with a MVC architecture, testing various game scenarios such as an instance of handling a full army could be represented with a model approach to test the class that the instance of each unit refers to. Since all of these units are part of one of the unit classes, testing the class itself will cover the scope of the overall test. This proved to be an effective means of calculating the metric of the code coverage as well as reduced the number of lines within the development and testing process. Another means of calculating the code coverage metric is to inspect that all functions that correspond to the requirements and modules that were initialized in the previous documentation are referred to and analyzed to such an extent. Minor behaviours that were not documented, such the connection between requirements and modules, are covered through an extensive gameplay session of manual testing to ensure that the game feels natural to the users. Any unusual behaviour through these stress tests will indicate areas that the code coverage may not extend to, followed by a code inspection to further enlarge the code coverage scope of the game. As the system tends to grow within this test driven development process, the number of test files will also increase to maintain the code coverage metric to its maximum, ensuring encapsulation and regulation of correct behaviour onto the software system.