

# Blaze Brigade

- Test Plan -

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

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

December 6, 2016

## Contents

<b>1</b>	<b>Functional Requirements Evaluation</b>	<b>1</b>
1.1	GUI . . . . .	1
1.2	Game Structure . . . . .	1
1.3	Unit Movement . . . . .	2
1.4	Unit Attacking . . . . .	2
1.5	Unit Structure . . . . .	2
<b>2</b>	<b>Nonfunctional Requirements Evaluation</b>	<b>2</b>
2.1	Look and Feel . . . . .	2
2.2	Usability . . . . .	3
2.3	Performance . . . . .	3
2.4	Operational . . . . .	3
2.5	Maintainability . . . . .	3
2.6	Security . . . . .	3
2.7	Cultural . . . . .	3
2.8	Legal . . . . .	3
2.9	Health and Safety . . . . .	3
<b>3</b>	<b>Comparison to Existing Implementation</b>	<b>4</b>
<b>4</b>	<b>Unit Testing</b>	<b>4</b>
4.1	Unit Testing Tools . . . . .	4
4.2	Limitations . . . . .	4
4.3	Unit Testing Approach . . . . .	5
<b>5</b>	<b>Changes Due to Testing</b>	<b>5</b>
<b>6</b>	<b>Automated Testing</b>	<b>5</b>
<b>7</b>	<b>Trace to Requirements</b>	<b>5</b>
<b>8</b>	<b>Trace to Modules</b>	<b>5</b>
<b>9</b>	<b>Code Coverage Metrics</b>	<b>5</b>

## List of Tables

1	Revision History . . . . .	1
---	----------------------------	---

## List of Figures

# 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 via mouse input. A player is also able to control a unit and select all of its possible actions from the drop down menu from mouse input. Therefore all GUI functional requirements successfully been met.

## 1.2 Game Structure

A trace of 2 players with player 1 controlling 1 warrior, and player 2 controlling 1 mage would be as follow.

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 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 range. The unit moves to that location and the move button disappears. The player then selects attack from the drop down menu and selects a valid attack target, resulting in neither targets dying. The attack button then disappears from drop down menu, and the player selects wait. Player 1's turn then automatically ends and switches to player 2. Player selects their unit, and selects attack from drop down menu, then selects 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 wait ends all of the unit's available actions. 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.

Table 1: **Revision History**

<b>Date</b>	<b>Version</b>	<b>Notes</b>
December 1, 2016	1.0	Completed Test Report

### **1.3 Unit Movement**

Players are able to select move from drop down menu after selecting a unit. All movable nodes calculated from BFS and accounting for obstacles are highlighted blue. 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 drop down menu after selecting a unit. All attackable nodes calculated from BFS and accounting for weapon attack range are highlighted red. The player is then able to select an enemy unit within 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 will execute the action, and result in getting counter attacked (assuming the unit attacked initially survived). The damage displayed in attack info 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 crit and hit rate on a unit. The 3 required units are also implemented and follows the requirements of each unit's stat build as follow: Warriors have high Attack and Defense, Archers have a high Strength, Skill and Speed, and Mages have a high Magic and Resistance. Each unit also has their own HP bar that updates corresponding to their current HP vs max 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 Feel requirement are adequately met.

## **2.2 Usability**

All people asked to test the game with a Windows computer with a screen size greater than 960x640, and access to a desktop 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 aspect behave exactly as expected, supports exactly 2 players and no player input to 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 in running Blaze Brigade. 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 known law. Therefore it fulfils all legal requirements.

## **2.9 Health and Safety**

The health and safety warning covers the most probable symptom (albeit unlikely) that could result from playing the game. Therefore health and safety requirements are met.

### 3 Comparison to Existing Implementation

This section will not be appropriate for every project.

## 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, as well as 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**

## **6 Automated Testing**

The extent of our automated testing includes our unit tests, containing 153 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**

## **8 Trace to Modules**

## **9 Code Coverage Metrics**