

Blaze Brigade

- Test Plan -

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

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

October 22, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

1 General Information

1.1 Purpose

1.2 Scope

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	Definition
Abbreviation1	Definition1
Abbreviation2	Definition2

Table 3: **Table of Definitions**

Term	Definition
Term1	Definition1
Term2	Definition2

1.4 Overview of Document

2 Plan

2.1 Software Description

2.2 Test Team

2.3 Automated Testing Approach

2.4 Testing Tools

2.5 Testing Schedule

See Gantt Chart at the following url ...

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Area of Testing1

Test 1:	Game will contain a main menu on screen upon launch.
Type:	Structural Dynamic Manual Testing
Test execution:	Program will be executed, then checked if a main menu appears upon launch.
Initial state:	Game is not opened.
Input:	Game executable is opened.
Output:	Game menu pops up upon launch.

Test 2:	Select New Game from the main menu.
Type:	Structural Dynamic Manual Testing
Test execution:	New game will be selected and checked that a new game instance is formed.
Initial state:	Game is currently on main menu.
Input:	New game is selected.
Output:	New game is started.

Test 3:	Select Load Game from the main menu.
Type:	Structural Dynamic Manual Testing
Test execution:	Load game will be selected and checked if the pre-existing game state is loaded.
Initial state:	Structural Dynamic Manual Testing
Input:	Load Game is selected.
Output:	Previous saved game state is executed.

Test 4:	Select How-To-Play from the main menu.
Type:	Structural Dynamic Manual Testing
Test execution:	How-To-Play will be selected and checked if the How-To-Play menu displays as expected.
Initial state:	Game is currently on main menu.
Input:	How-To-Play is selected.
Output:	How-To-Play is selected.

Test 5:	Select Exit Game from the main menu.
Type:	Structural Dynamic Manual Testing
Test execution:	Exit is selected and checked that the game is closed.
Initial state:	Game is currently on main menu.
Input:	Exit Game is selected.
Output:	Game is closed.

Test 6:	The game is turn-based, and units are only able to move and attack once per players turn.
Type:	Structural Dynamic Automated Testing
Test execution:	A user will click on a unit and have it perform its available actions. Immediately after, it will be checked that the same unit is unable to perform any additional actions.
Initial state:	Unit X has not performed their action.
Input:	Unit X performs their action.
Output:	Unit X is no longer able to perform their action.

Test 7:	A player's turn ends once all their units have performed all available actions.
Type:	Structural Dynamic Automated Testing
Test execution:	The available actions of all of Player 1's units will be set to none. Then, it will be checked that Player 1's turn has ended and it is now currently Player 2's turn.
Initial state:	Currently is Player 1's turn.
Input:	Set all actions of units belonging to Player 1 to none.
Output:	Player 1's turn ends, and it is now Player 2's turn.

Test 8:	During a unit's turn, clicking a unit will give a drop down menu with available actions of the unit.
Type:	Structural Dynamic Manual Testing
Test execution:	A unit, which has not already performed an action, will be right-clicked and checked that a drop down menu appears with expected actions of that unit.
Initial state:	The unit is alive and has available actions.
Input:	The unit is selected.
Output:	A drop down menu with expected actions pops up.

Test 9:	One side is victorious when the other side has no playable units left.
Type:	Structural Dynamic Automatic Testing
Test execution:	The status of all live units belonging to Player 2 is set to deceased. It is then checked that the game is over and Player 1 is victorious.
Initial state:	Player 1 and Player 2 both have live units.
Input:	All live units belonging to Player 2 are killed.
Output:	Player 1 is victorious and the game is over.

Test 10:	Player can select move after selecting a unit that has yet to perform its action and opening the drop down menu.
Type:	Structural Dynamic Manual Testing
Test execution:	The unit that has yet to move and is able to move is selected, and it is observed that a pop-up menu containing the "move" action appears.
Initial state:	The unit has not yet performed an action, and is available to move if required.
Input:	The unit is selected and the drop down menu is opened
Output:	Drop down menu containing the option "move" is visible.

Test 11:	Units are only able to move within their move range.
Type:	Functional Dynamic Manual testing
Test execution:	Attempt to move a unit to a tile outside of its move range.
Initial state:	The unit is alive and able to move.
Input:	The unit is asked to move to a different tile outside of its range.
Output:	Nothing happens since the requested move is not within the unit's move range.

Test 12:	Units are not be able to pass through obstacles.
Type:	Functional Dynamic Manual Testing
Test execution:	Attempt to ask the unit to move to an area blocked off by an obstacle.
Initial state:	The unit is within range of an area blocked off by an impassable obstacle.
Input:	Ask unit to move to the area past the obstacle.
Output:	Unit is unable to move to the requested area.

Test 13:	Player can select attack as an available option after selecting a player-owned unit that has not attacked and an enemy unit is within attack range.
Type:	Structural Dynamic Manual Testing
Test execution:	The owned unit that has yet to attack is selected, and it is observed that a pop-up menu containing the "attack" action appears.
Initial state:	The player's unit has not yet performed an attack, and an enemy unit is within its attack range.
Input:	The unit is selected and the drop down menu is opened.
Output:	Drop down menu containing the option "attack" is visible.

Test 14:	Unit may only attack an opposing unit within its attack range.
Type:	Functional Dynamic Automatic Testing
Test execution:	An automated test attempts to ask the program to make a unit attack an enemy unit outside of its attack range.
Initial state:	The unit that is available to attack and its enemy unit are not within attack range of each other.
Input:	The unit available to attack will attempt to attack the enemy unit.
Output:	Nothing will happen, as the attempted attack is not valid due to the fact that the enemy unit is out of range.

Test 15:	All playable units can attack enemy units.
Type:	Functional Dynamic Automatic Testing
Test execution:	Automated tests check that all units have required states and functions that allow it to attack enemy units.
Initial state:	Units are initialized.
Input:	Units are initialized.
Output:	All units have required states and functions that allow them to attack enemy units.

Test 16:	Units are unable to move after attacking.
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to activate the drop down menu to move the unit upon an owned unit that has already acted within the turn.
Initial state:	The unit has already completed an action for the current turn.
Input:	A player attempts to activate the drop down menu to move a character after attacking.
Output:	Drop down menu will not appear.

Test 17:	Units will lose HP according to damage calculations.
Type:	Functional Dynamic Automatic Testing
Test execution:	Automated tests will check that damage modifying the character HP, calculated through the damage calculation class, is consistent with the expected value.
Initial state:	The unit is alive and able to take damage.
Input:	The unit takes damage.
Output:	The HP lost by the unit should correspond to what is calculated during the damage calculation step.

Test 18:	Units that are deceased are no longer active in the current battle.
Type:	Functional Dynamic Automatic Testing
Test execution:	Automated test modifies a unit's HP to 0.
Initial state:	The unit has more than 1 HP and is valid in battle.
Input:	The unit's HP is set to 0.
Output:	The unit's state shall reflect that it is no longer valid in the battle and can no longer be used in the game.

Test 19:	Player can select which weapon each unit uses to perform an attack.
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to change weapons by selecting weapons from the unit drop down menu on a playable unit that has yet to move, select a different weapon , and observe if attack stats has changed in accordance to the newly selected weapon. initialState
Initial state:	The unit has more then 1 HP and is valid in battle.
Input:	Attempt to change the unit's weapon
Output:	The unit changes weapons and gains different stat modifiers in accordance to the new weapon.

Test 20:	All units shall have a corresponding unit class.
Type:	Functional Static Automatic Testing
Test execution:	Automated tests check that units have a corresponding unit type class that is viable in the game.
Initial state:	All units are instantiated.
Input:	All units are instantiated.
Output:	All classes correspond to a viable unit type class.

Test 21:	All units have stat values corresponding to their class.
Type:	Functional Static Automatic Testing
Test execution:	Automated tests check that a unit's stat correctly corresponds to the pre-determined stats of the specific unit class.
Initial state:	All units are instantiated.
Input:	All units are instantiated.
Output:	All units have stats that correspond to their class.

Test 22:	Classes include warrior, mage, and archer.
Type:	Functional Static Automatic Testing
Test execution:	Automated tests check for the existence of all three classes in the program by instantiating units for all three classes.
Initial state:	All three classes are existent in the game.
Input:	Units of all three classes are instantiated.
Output:	Units of all three classes exist and are of the corresponding class.

Test 23:	The stats of the game include Str, Int, Def, Res, Skill, Speed, and HP.
Type:	Functional Static Automatic Testing
Test execution:	Automated tests check that a unit has all of the listed stats.
Initial state:	A unit is instantiated.
Input:	A unit is instantiated.
Output:	The unit owns all listed stats.

3.1.2 Area of Testing2

...

3.2 Tests for Nonfunctional Requirements

3.2.1 Area of Testing1

...

3.2.2 Area of Testing2

...

4 Tests for Proof of Concept

4.1 Area of Testing1

...

4.2 Area of Testing2

...

5 Comparison to Existing Implementation

6 Unit Testing Plan

The Visual Studio Unit Testing Framework shall be used to write and execute the game's automated unit tests in C#.

6.1 Unit testing of internal functions

The automated unit tests will test internal functions of the program by passing controlled input(s) into a function in order to ensure correct behaviour or output of that single function. Each testable function in the program shall thus have corresponding unit tests that test each possible type of input to ensure expected behaviour and/or output of that function under possible edge cases, regular cases, or abnormal cases. Functions that return a value will have their output tested for the expected output, and void functions shall be tested for correct behaviour, such as changes to the model and its state variables. As such, the unit tests will provide thorough whitebox testing of the game's code. Test coverage tools, which are integrated in Visual Studio 2015, will be used as a metric to determine the degree of unit testing code coverage. The goal of the team is to achieve a minimum of 80% code coverage to ensure that the majority of the code has undergone white box testing, resulting in fewer errors regarding incorrect coding implementation of the functional requirements.

6.2 Unit testing of output files

The only output file of the game is a window which comprises the visual representation and graphical aspects of the game. To completely ensure proper function of the output file, manual testing must also be taken into consideration to test expected behaviour of the game. In addition, the game engine, XNA Game Studio, handles the majority of the rendition from code to output. Our task does not involve testing proper functionality of the game studio, however, unit testing of the team's code still plays a key role in ensuring proper output. Unit tests for functions that call on the view, as well as unit tests written for the view are necessary to ensuring proper output of the game's visual representation. The unit tests will additionally verify that proper method calls to game studio methods are being executed, most likely with the use of mock objects to mock the actual game visuals, and to verify that these mock objects are being called upon.

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

7.2 Usability Survey Questions?

This is a section that would be appropriate for some teams.