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 31, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
October 30, 2016	1.0	Completed Test Plan Rev 0

1 General Information

1.1 Purpose

The purpose of this project is to recreate a tactical, turn based game similar to Tactics Heroes. The game shall test the strategical skills of the user by presenting a large quantity of information to process in order to determine the best course of action for the player's turn. The game will progress based on user inputs and decisions. However, such software will involve a wide variety of test cases to ensure proper functionality. A specific category of testing includes automated unit testing, which will follow the Visual Studio Unit Testing Framework. This document will provide a complete overview of test cases the software will follow, and more specifically, provide information on the unit testing framework.

1.2 Scope

Software such as what Blaze Brigade aims to recreate has a complex interaction with the user, giving the user many options for each decision they make. As such, each one of these options, such as moving, attacking, equipping a different weapon, and et cetera, must function properly. Each action requires a proper test case in order to ensure their proper functionality. Other test cases based on functional requirements include unit properties, menu navigation, structural properties of the game, and handling user input. The non-functional requirements of this software project are based on usability, performance and security. Proper test cases will be orchestrated to fulfil these requirements as well.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
HP	Health points
Str	Strength
Int	Intelligence
Def	Defense
Res	Resistance
GUI	Graphical user interface

Table 3: **Table of Definitions**

Term	**Definition**
Unit	A movable character that the user will manipulate to defeat the opponent's army.
Class	A category of unit type. Different classes have different strengths, weaknesses, and attributes.
Stat	A numerical value that belongs to a unit. There are a variety of stats that make each class unique.
Health Points	A stat determining how much damage a unit can take before it dies.
Strength	A stat used in calculating physical damage dealt to opposing units.
Intelligence	A stat used in calculating magical damage dealt to opposing units.
Defense	A stat that lessens the amount of physical damage the unit takes.
Resistance	A stat that lessens the amount of magic damage the unit takes.
Skill	A stat that determines how skillful a unit is on the battlefield. Used to determine hit rate and critical hit rate.
Speed	A stat that determines how many times a unit will get to attack in combat.
Graphical User Interface	A system that interacts with the user through visual representation.
Hit Rate	The percent chance that unit A will successfully hit unit B.
Critical Hit	Multiplies the damage of a unit's attack by a numerical factor.
Critical Hit Rate	The percent chance that unit A will perform a critical hit on unit B.

1.4 Overview of Document

This document will explain in depth the plans for testing of the project, Blaze Brigade, and will provide a comparison to the existing implementation, Tactics Heroes. In this document, a description for each test case is provided along with the expected input and output for each case. This document will be referenced by the team of the project when conducting tests on developer code. This document also provides a brief description of what automated testing is, and the group's chosen framework to simulate such automated tests.

2 Plan

2.1 Software Description

The software component of the project is governed by various actions such as inputs required, outputs to be shown to the user, and certain task computation to fulfill the desired set of requirements. To test the overall system of the project and produce a stable build for the user to interact with, a set of software descriptions need to be covered in this test plan to discuss the main functionality and how they can be tested as outlined below.

- **Mouse input:** This is the primary interaction between the user and software to carry out actions within the gameplay. Such action include starting the game, moving the units, and giving commands such as to attack the opponent. A test will need to be devised to ensure all mouse clicks are read and their accuracy of the position to ensure that the resulting trigger is correct.
- **Gameplay window:** The map will be created in the gameplay window, in which all of the mouse trigger events would happen to provide interaction with the game. The window would need to be tested on all the subtasks it holds, such making best use of all the space allocated to it, and the ability to close and minimize the window application.
- **Menu option:** The menu is the first screen the users will see to select an option to start a new game, learn how to play or exit the game. Each selection will be tested to ensure that it directs the user to the correct use case.
- **Map creation:** When the terrain is constructed, it will display a field which includes moveable positions and obstacles such as a tree, which dictate positions that a unit may step on. Further testing will need to be conducted on these obstacle nodes to ensure that an unit does not accidentally take an illegal position, or that an unit may take a legal position.
- **Movement of units:** After a unit has been selected, there shall be a limited amount of highlighted positions that it can move onto. The constraints within the path finding mechanism would need to be simulated as a test case to ensure that the highlighted grid shows the correct layout and the move onto a position is valid to abstain from any invalid operation.
- **Attack mechanism:** During an attack, the affected unit(s)' stats and health are taken into consideration to determine who shall be victorious in killing the opponent's unit. The test case will further breakdown the attack mechanism to ensure that the correct drop of health is calculated and presents a fair attack opportunity for both sides.

- **Turn based selection:** Both players will alternate turns upon completing their set of actions. A checker would need to be in place to determine that a turn has successfully been completed and shifted to the correct player.

2.2 Test Team

The test team includes all of the members from the development team to encourage that testing takes places at all stages of the development process to meet the central objectives of the project. This requires the involvement of all team members in regular code inspection, producing unit test cases, and the design for suitable user interaction.

Table 4: **Description of the Test Team**

Team Member	Testing Type
Thien Trandinh	Structural Testing, Functional Testing
Susan Yuen	Structural Testing, Functional Testing
Jeremy Klotz	Dynamic Testing, Automated Testing
Asad Mansoor	Manual Testing, Automated Testing

2.3 Automated Testing Approach

An automated testing approach will be introduced in the development process of the project to ensure a new feature or code change does not affect the stability of the master build. Additionally, it would allow better use of resource allocation to move the manual testers to work other aspects of the code or documentation. As the project grows, the automated testing approach would further educate the team in producing more reliable code as well as minimizing the time of manual testing.

Testing tools like Visual Studio Unit Testing Framework will play a big role in the creation of the unit test cases, reflecting on the functions that impact the logic behind the game. With reference to the functions, we can test for desired output with the anticipated inputs and further elaborate the testing scheme by checking for robustness by providing invalid inputs or extreme test cases. Since automation can cover a large range of testing over a short period of time, it would be feasible to conduct stress testing of the game and to run the automated unit tests repeatedly over a long period of time. Furthermore, the unit test cases are initially set to test features within each class, but a set of these automated test scripts would eventually cover the system data flow to better understand how the software is interacting with other pieces of code and whether a more efficient design approach is needed in the next development stage.

With the aid of automated testing, there will be less reliance on the team members to constantly check whether a certain feature is correctly implemented for a large magnitude of inputs. The best practice of this technology would be to constantly develop new test cases in parallel to ongoing development process and to run all test scripts multiple times before pushing the source code onto GitLab. Since the nature of a game cannot be fully taken over by automated testing approach, manual testing will still play a part to ensure that the game behaves as it should and feels natural to the user.

2.4 Testing Tools

Visual Studio Unit Testing Framework will be the testing tool required to automate the unit test cases throughout each development phase and will cover a wide range of functional and system analysis.

2.5 Testing Schedule

The following test schedule has been derived from the development plan to ensure that the product is functioning correctly as it continues to evolve. In that regard, the schedule can be broken down into the test deliverable and test cases schedule. The test deliverable schedule outlines the required test plans and test reports to be made available for the team members and stakeholders. In contrast, the test case schedule focuses on the internal dynamic of the software, outlining the testing period of each of the major development phases.

For additional detail, please consult the Gantt Chart (link provided).

Table 5: **Test Deliverable Schedule**

Deliverable ID	**Test Deliverable**	**Due Date**
TP-0	Test Plan Revision 0	October 31
TP-1	Test Plan Revision 1	November 14
TP-2	Final Documentation - Test Plan	December 7
TR-0	Final Documentation - Test Report	December 7

Table 6: **Test Cases Schedule**

Sprint #	**Due Date**	**Task**	**Test Case**
0	Oct 19	Proof of Concept	Initial testing to ensure proof of concept demonstration works as planned.
1	Oct 31	Menu creation	Main menu re-direct to correct page. Sub menu of each unit to show available commands.
1	Oct 31	Unit highlight	Highlight state is active when an unit is selected.
1	Oct 31	Unit movement	Unit movement works as designed.
1	Oct 31	Unit animation	Unit animation is visible upon movement.
1	Oct 31	Full-scope testing	Testing of Sprint 1 to ensure the overall system is correct.
2	Nov 11	Add units	Check to see which unit is selected on which team.
2	Nov 11	Combat system	Combat attributes work as designed.
2	Nov 11	Unit collision	Unit collision logic works as designed.
2	Nov 11	Full-scope testing	Testing of Sprint 2 to ensure the overall system is correct.
3	Nov 16	Terrain obstacles	Position with obstacles should not be valid moves for units.
3	Nov 16	Full army	Check the state of the unit with full army specification on which team.
3	Nov 16	Full-scope testing	Testing of Sprint 3 to ensure the overall system is correct.
3	Nov 16	Extensive testing	Stress testing of the game as a whole system.

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 GUI

Test 1:	**GUI is controlled by the mouse.**
Type:	Structural Dynamic Manual Testing
Test execution:	The user clicks on an available game option that will produce a result which changes the GUI.
Initial state:	Game is opened.
Input:	User clicks on an action.
Output:	The game behaves as expected from user mouse input.

Test 2:	**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 3:	**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 4:	**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:	Game is currently on main menu.
Input:	Load Game is selected.
Output:	Previous saved game state is executed.

Test 5:	**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 6:	**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.

3.1.2 Game Structure

Test 7:	**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 8:	**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 9:	**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 10:	**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.

3.1.3 Unit Movement

Test 11:	**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 12:	**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 13:	**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.

3.1.4 Unit Attacking

Test 14:	**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 15:	**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 16:	**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 17:	**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 18:	**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 19:	**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 then 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 20:	**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.

3.1.5 Unit Structure

Test 21:	**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 22:	**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 23:	**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 24:	**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.

3.2 Tests for Nonfunctional Requirements

3.2.1 Usability

Test 25:	**Game runs on the operating systems specified in the requirements.**
Type:	Structural Static Manual Testing
Test execution:	The program is executed on the specified operating systems and will be checked for expected behaviour.
Initial state:	Game is not yet executed.
Input:	Program is launched.
Output:	The game functions as expected.

3.3 Performance Requirements

Test 26:	**Program Response Time**
Type:	Structural Dynamic Manual Testing
Test execution:	User clicks on option that produces a response from the game.
Initial state:	Game is open.
Input:	Attempt to click on an option that should produce a response.
Output:	The program responds to the user input in near instant time.

Test 27:	**Changes in statistics are accurately reflected on the screen.**
Type:	Structural Dynamic Manual Testing
Test execution:	User successfully attacks an enemy unit with a playable unit.
Initial state:	An enemy unit is in range of a playable unit.
Input:	User attacks the enemy unit with the playable unit.
Output:	The enemy unit's hit points decrease.

3.4 Security Requirements

Test 28:	**Invalid user input**
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to give invalid input to the game, such as keyboard input or clicking on invalid targets.
Initial state:	Game is initialized.
Input:	Attempt to give the game keyboard input and clicks on invalid targets.
Output:	The game state does not change.

4 Tests for Proof of Concept

4.1 GUI

Test 29:	**Terrain grid with an unit is constructed onto the GUI window.**
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to launch the game by the user, will prompt a window in which a terrain shall be initialized for the gameplay interactions.
Initial state:	Game is not yet opened.
Input:	Program is launched.
Output:	The game functions as expected and a gameplay window is prompted for user interaction.

Test 30:	**The game will remain in its current state when the application is minimized.**
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to minimize the game during the gameplay and shortly after reopen the game to continue from the same game state.
Initial state:	Game is initialized and showing the terrain grid.
Input:	Attempt to minimize the application by clicking on the minimize button on the top-right corner of the window.
Output:	The game minimizes as expected and displays the same game state when reopened.

Test 31:	**The game and its operation will be closed when the application is closed.**
Type:	Structural Dynamic Manual Testing
Test execution:	Attempt to close the gameplay window to end the game and its operations.
Initial state:	Game is initialized and showing the terrain grid.
Input:	Attempt to close the application by clicking on the close button on the top-right corner of the window.
Output:	The game ends all operations and closes the gameplay window as expected.

4.2 Unit Movement

Test 32:	**Unit can be selected and deselected when clicked.**
Type:	Functional Static Automatic Testing
Test execution:	Attempt to trigger selection and deselection mode on an unit when it is being clicked on from the user.
Initial state:	The game is opened and initialized.
Input:	The unit is clicked on for selection and then clicked again for deselection.
Output:	The unit stays in the same position waiting to be selected to be moved as expected.

Test 33:	**Units are able to move onto any grid position when selected.**
Type:	Functional Static Automatic Testing
Test execution:	Attempt to move units anywhere on the grid only when they are selected by being clicked on from the user.
Initial state:	The game is opened and initialized.
Input:	The unit is clicked on by the user who then selects any available grid position placed on the terrain.
Output:	The unit will take the new position and the old position will be empty as expected.

5 Comparison to Existing Implementation

Blaze Brigade is based on the strategical, tactical, grid based RPG called Tactics Heroes. All the functional requirements were derived from this existing implementation, and can be categorized into 5 categories: GUI, game structure, unit movement and unit attacking. The comparison between Tactics Heroes and Blaze Brigade for each category are listed below:

5.1 GUI

The GUI structure is almost identical to Tactics Heroes, such as the interface, and navigation menu. However, all the graphical assets used is original and either obtained from open sources, or original assets made by the team.

5.2 Game Structure

The game structure involves players taking turns moving their units, and trying to kill off all the opposing enemy units. This game structure is identical between both the existing implementation and Blaze Brigade.

5.3 Unit Movement

Unit movement is done by selecting a unit, then clicking another tile within the unit's move range to move it to that location. This way of moving is consistent between both versions.

5.4 Unit Attacking

Unit attacking is done by clicking on a unit, and selecting an enemy unit within range to perform an attack. Both implementations are similar in this aspect, however there is one major difference. In Blaze Brigade, after a unit performs an attack, they receive a counterattack from the enemy should the enemy survive the initial hit. In Tactic's Heroes, no counter attacks are performed.

5.5 Unit Structure

The unit types are different between the two games. In Tactics Heroes, the units are melee and ranged, with all damages only doing physical damage. In Blaze Brigade, there are 3 units - Melee, Ranger (physical ranged), and mage (magical ranged). The stat allocation in Blaze Brigade are also more intricate. The stats that are the same are HP, where a unit dies when their HP falls under 0, Strength which determines their physical attack, and Defense which determines their physical defense. Tactics Heroes only has one more stat - Dodge, which affects dodge rates for attack. In Blaze Brigade, this calculation is calculated from the difference in skill between the two units. Futhermore, there are 3 more additional stats. Firstly is Intelligence and Resistance, which is used for magical damage calculation. The last is Speed which determines if a unit performs consecutive attacks.

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 have corresponding unit tests for each possible type of input, to ensure expected behaviour and/or output of that function under possible edge, regular 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, thus 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 are a collection of windows which comprises the visual representation and graphical aspects of the game. This includes the Main Menu, the How-To-Play, and Game window. To completely ensure proper function of the output file, manual testing must also be taken into consideration to test the 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 and results. 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 simulate the actual game visuals, and to verify that these mock objects are being called upon.