Test Report for Bomber Clone Application

Software Engineering 3XA3: Project

Group 12 - The A Team
Gabriel Lopez De Leon, 1310514
Ren-David Dimen, 1222679
Jay Nguyen, 1327828

27 November 2015

Table of Contents

1 Revision History

Revision #	Revision Date	Description of Change	Author
3	Nov 27 2015	Complete Section 4, 6, 7, 8	Gabriel Lopez de Leon
2	Nov 24 2015	Add Section 1, 2, and 4	Gabriel Lopez de Leon
1	Nov 24 2015	Create Document Outline	Gabriel Lopez de Leon

2 Introduction

This section provides an overview of the testing completed for our project, the results obtained from the testing and the relevance of each test is described in detail in this document.

2.1 Purpose of the Document

This document is used to provide a trace to all the testing that was done during the implementation of our Bomberman game, and the relevance and need for each test. The test report also provides evidence that testing was done to improve the quality of our program and ensure there are minimal bugs.

2.2 Scope of the Testing

This test report contains test cases for both front and back end of the applications. The various types of tests performed include unit tests such as the use of JUnit (through the Eclipse IDE) and system tests. These various test will either be automated or manual and will consist of both black box and white box tests, depending on the functionality which is being tested.

2.3 Organization of the Document

The remainder of the document contains eight major sections which consists of a list of all the figures and tables within the document, component testing, nonfunctional qualities evaluation, changes due to testing, automated testing, system test results, a trace to the requirements, and a trace to the modules. The nonfunctional qualities evaluation is divided into three parts, usability, performance, robustness. Each of these sections will be divided into the various different test cases which will detail their results and relevance.

3 List of Figures and Tables

4 Components Testing

This section shows the reader all the different test cases that were done for each of the subsections below while also providing a short summary and type of test used.

4.1 Game State Testing (C1)

Testing Type: Unit Test, Black Box Test, Automated.
Tools Used: JUnit using the Eclipse IDE.
Summary: Unit test to check if a player is alive or not because once a player is dead, the game stops updating and ends. In addition, if the timer reaches zero, the game should also end.
Sample Inputs: Set player isAlive equal to false and check to see if the game ends.
Test Case:

```
//Game should not be running when a player is dead
Game game = new Game();
game.start();
game.player.setPlayerAlive(false);
game.update();
assertEquals(game.getRunning(),false);

//Game should be running when players are alive
Game game = new Game();
game.start();
assertEquals(game.getRunning(),true);

Arena arena = game.getArena();
int[] tiles = arena.getTiles();
int width = arena.getWidth();

// Checks collisions
for (int x = 0; x < 15; x++){
    for (int y = 0; y < 9; y++){
        game.player = new Player(x,y,game.getInput(),arena);
        assertEquals(game.player.tileCollision(0,0),arena.getTile(x, y).solid());
```

```
    }
}

// Goes through all tiles and checks if they match given arena
for (int x = 0; x < 15; x++){
    for (int y = 0; y < 9; y++){
        if (tiles[x+y*width] == 0xFF000000){
            assertEquals(arena.getTile(x,y),Tile.steelTile);
        } else if (tiles[x+y*width] == 0xFF00FF00) {
            assertEquals(arena.getTile(x,y),Tile.rockTile);
        } else if (tiles[x+y*width] == 0xFFFFFFFF){
            assertEquals(arena.getTile(x,y),Tile.grassTile);
        }
    }
}

// Checks broken brick states
for (int x = 0; x < 15; x++){
    for (int y = 0; y < 9; y++){
        arena.rockX.add(x);
        arena.rockY.add(y);
        assertEquals(arena.getTile(x, y),Tile.grassTile);
    }
}
```

4.2 Game Timer Testing (C2)

Testing Type: Unit Test, Black Box Test, Automated.
Tools Used: JUnit using the Eclipse IDE.
Summary: Unit test to check if the timer is running and that it stops when it hits zero.
Sample Inputs: Loop through all timer states (in our case 180 seconds) to ensure it is running up until the timer is zero.
Test Case:

```
//Game should be running when timer is greater than zero
for (int i = 180; i > 0; i–){
```

```
game.start();
game.setGameTimer(180);
game.update();
assertEquals(game.getRunning(),true);
game.stop();
}

//Game should not be running when timer equals zero
game.start();
game.setGameTimer(0);
game.update();
assertEquals(game.getRunning(),false);
```

4.3 Arena Testing (C3)

Testing Type: Unit Test, Black Box Test, Manual.
Summary: Test to check for a variety of different arena layout options.
Sample Inputs: Different sprite sheets for the various arena layouts.
Test Case: Manual Tests for this case would simply be to run the game numerous times as a random arena layout is chosen when the game first starts.

4.4 User Input Testing (C4)

Testing Type: Unit Test, White Box Test, Manual.
Summary: Test to check if the game allows for multiple user inputs.
Sample Inputs: Keyboard inputs by the players.
Test Case: This test case also uses manual, black box testing and can be achieved by making two users attempt to play the game at the same time. If the game accepts the inputs from both users at the same time then the test would pass.

5 Nonfunctional Qualities Evaluation

5.1 Usability

5.1.1 Game State Testing

Though most of the test cases for game state addressed functional requirements, the usability for game state functions was tested by team members and other students who played and tested the game. A group of students were chosen to play our game and test out its functionality. With respect to usability, the ease of use and learning was shown by how fast other users where able to quickly play our game.

5.1.2 Game Timer Testing

Similar to Game State Testing, game timer functions are functional requirements and was tested by a variety of people who used our program and provided feedback. The timer itself runs automatically once the game begins and ends the game when it reaches zero. Users do not interact with the timer at all.

5.1.3 Arena Testing

The randomized map layout is also a functional requirement and is tested every time someone plays our game since the function which randomizes the arena layout is called when the game starts.

5.1.4 User Input Testing

Usability of user input functions was tested by a number of students chosen to play our game. After they played the game, they were asked for feedback on how well the system accepted inputs from both players playing at once and if the controls were easy to learn/use.

5.2 Performance

5.2.1 Game State Testing

The JUnit library provides results on time performance which is displayed after the test file is run. The average time to run the test file is 0.187

seconds which is considerably fast. Other performance requirements for the game states include reliability and availability. The game can be run at any time; when the Java file is run, the game begins without any errors. Furthermore, hit collisions and block/tile rendering is consistent as shown through numerous test runs of playing the game.

5.2.2 Game Timer Testing

The JUnit tests for the game timer are in the same test file as the tests for game state, and thus share the same test file runtime. The game timer shows reliability by counting down through all timer states and eventually reaching zero.

5.2.3 Arena Testing

Manual test for the arena functions tested for reliability requirements as it is an important aspect which adds some variety to the game. Arena functions meet reliability requirements as the game properly randomizes the arena layout at the program launch time.

5.2.4 User Input Testing

Manual tests for the user input mostly focused on reliability and capacity as the system should always accept user inputs consistently while also allowing for two players to play the game simultaneously.

5.3 Robustness

5.3.1 Game State Testing

Robustness of the various game state functions where tested by ensuring to check for all possible collisions, make sure the game ends when it should (either a player wins or timer runs out) and also tests to make sure the arena was rendered properly by going through each block to check.

5.3.2 Game Timer Testing

There were no robustness tests for the game timer as it is a simple function which accomplished its task consistently as shown through the other test cases.

5.3.3 Arena Testing

The system only takes in the set sprite sheet for the different arena layouts and are linked to the code by its specific path and location in the workspace folders.

5.3.4 User Input Testing

Only designated keyboard inputs are accepted by the system, if other buttons are pressed, nothing happens. The system only checks for the arrow keys and enter button for player one, and checks for a,w,s,d,g keys for player two controls.

6 Changes Due to Testing

There were no changes to testing in terms of core code or functionality of any of the modules. Only minor changes such as the names of some variables, methods and classes were changed to match up with the test plan. The addition of getter and setter functions were also added to be able to test using JUnit in Eclipse.

7 Automated Testing

7.1 Game State Testing

All tests for the Game State were automated in Java in "GameTesting.Java" using JUnit through the Eclipse IDE. Running this Java test file will automatically run all the tests and output the results in real time in the JUnit console.

7.2 Game Timer Testing

Similar to Game State Testing, Game Timer Testing was fully automated in the Java file "GameTesting.Java" which was run through JUnit on the Eclipse IDE. The test file runs a test case which checks if the timer for the game accomplishes its task properly. The results of this test will also be shown on the JUnit console.

7.3 Arena Testing

The various possible arena layouts was tested manually as it was not necessary to run automated tests to check if a random map layout is chosen when the game starts. Instead, the game was run multiple times to see if the arena layout was chosen at random at the beginning of each game. This test case models testing by induction, if the game changes the layout a few times and successfully reads the various arena sprite sheets, then it should work for all other arena layouts.

7.4 User Input Testing

User Input was also tested manually and is one of the easier cases to test. A simple test for this is to see if two players can play simultaneously and the game accepts inputs from both players.

8 System Test Results

8.1 Game State Testing

There are 5 different test cases which check different game states and functionality. These tests are all automated through JUnit in the Eclipse IDE and have all had positive results. These various tests include checking if the game is running when both players are alive, or if the game is not running when a player dies. It also checks for broken brick states to ensure a block was removed, and checks if the tiles displayed matches the arena layout that was selected. Lastly, there is a test case which checks through all the possible collisions in the game. Aside from the automated tests conducted through JUnit, manual tests can also be applied to this case by playing through the game and saying if the game meets the main functional requirements.

8.2 Game Timer Testing

There are currently two test cases for the game timer, one to make sure the timer is continuously running while it is greater than zero. The other test case checks to see if the timer stops when it reaches zero. These two tests were both automated through JUnit and passed with results being that the timer counts down from its initial value of 180 seconds to zero and then

stops. Other tests which relate to this case would be instances when the game is played as the timer runs during gameplay and ends the game when it hits zero.

8.3 Arena Testing

This test case was mostly manually tested, besides a test to check the arena layout which was done in the game state testing, a large number of manual tests were run for this test case. The game has been run multiple times by each member of the group to test for the randomness of arena layout. Furthermore, when the game is run to test other aspects, this case is also being tested as it occurs every time the game is played. Other users have also been asked to try our game which further adds to the number of tests run for this specific test case. The results of these tests have been positive, the game meets the requirement of randomizing the map layout.

8.4 User Input Testing

As user input testing has been tested manually, there is not much variety in the number of ways to test this functionality. However, like arena testing, a large number of tests were run for this case. By playing the game multiple times with two player, we have tested how the system accepts two inputs at once. The results for this were positive and the game accepts the inputs with ease.

9 Trace to Requirements

Test Cases	**C1**	**C2**	**C3**	**C4**
Requirements	Game State	Game Timer	Arena	User Input
R1	✓	-	-	-
R2	-	✓	-	-
R3	-	-	✓	-
R4	-	-	-	✓

Table 1: Trace to Requirements

10 Trace to Modules

10.1 Game State Testing

The various game state tests trace back to different modules, depending on the specific test that was conducted through JUnit. For the tests that checked if the game was running or not, a trace to the player module can be made. For the testing of collisions, the both the player and collision modules were used. The test case to check if all tiles match the given arena can be traced to the tile module under the tiles package in the project workspace.

10.2 Game Timer Testing

The JUnit test cases for game timer both trace back to the main module, game.Java. This is module is where all of the timer code is handled.

10.3 Arena Testing

Tests for arena can be traced back to the arena module which handles the randomized layout of the map upon game launch.

10.4 User Input Testing

Test cases for user inputs can be traced back to the player and keyboard modules which set the controls for player movements.