SE 3XA3: Test Report
Namcap

Team 2, VPB Game Studio
Prajvin Jalan (jalanp)
Vatsal Shukla (shuklv2)
Baltej Toor (toorbs)

December 8, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
2016-12-08	1.0	Completion of Functional Requirements Evaluation
2016-12-08	1.1	Completion of Unit Testing Section
2016-12-08	1.2	Completion of Changes Due to Testing
2016-12-08	1.3	Completion of Automated Testing
2016-12-08	1.4	Addition of Traceability Matrices and explanation of code coverage metrics
2016-12-08	1.5	Completion of Non-Functional Testing, and Comparison to Existing Implementation
2016-12-08	1.6	Spelling corrections to document

This document serves as a test report for the Namcap redevelopment based on the established test plan. All requirements and modules stated in the document are traceable to the corresponding SRS and MG for this development.

1 Functional Requirements Evaluation

1.1 Game Functionality Testing

A Robot (automated) unit testing class was implemented and used to test the mechanics of the game.

General Testing

1. GFT1

   Type: Functional, Dynamic, Automated

   Initial State: Application is displaying the main menu page

   Input: Cursor clicked on Start Game button

   Expected Output: New game is started and window is changed to reflect a new game state

   Output: New game was started and window was changed to reflect a new game state

   Result: PASS

11. GFT11

    Type: Functional, Dynamic, Automated

    Initial State: Within game state

    Input: Escape button pressed

    Expected Output: Application must pause and ask user if they want to quit

    Output: Game was paused and user was asked to quit or continue

    Result: PASS

Player Movement/Collision Testing

2. GFT2

   Type: Functional, Dynamic, Automated

   Initial State: Within the game state

   Input: Arrow keys

   Expected Output: Player moves in the respective direction (if path is clear)

   Output: Player moved in the respective direction when path was clear

   Result: PASS

3. GFT3

   Type: Functional, Dynamic, Automated

   Initial State: Player comes in contact with wall

   Input: No input

   Expected Output: Player stops moving when coming in contact with the wall

   Output: Player's x and y coordinates were not changed when in contact with the wall

   Result: PASS

4. GFT4

   Type: Functional, Dynamic, Automated

   Initial State: Player comes in contact with enemy

   Input: No input

   Expected Output: If player has more than 1 life, decrement lives. If player has one life, end game.

   Output: Player's life was decremented by 1 when player had more than one life. The game was ended if player was on their last life.

   Result: PASS

6. GFT6

   Type: Functional, Dynamic, Automated
   Initial State: Player comes in contact with dots
   Input: Arrow keys
   Expected Output: Dot disappears after collection
   Output: Dot disappeared after player collected it
   Result: PASS

7. GFT7

   Type: Functional, Dynamic, Automated
   Initial State: Player collects the big dot
   Input: Arrow keys
   Expected Output: Big dot disappears after collection
   Output: Big dot disappeared after player collected it
   Result: PASS

8. GFT8

   Type: Functional, Dynamic, Automated
   Initial State: Player collects the big dot
   Input: Arrow keys
   Expected Output: Player is able to collide with enemies
   Output: Player does not lose any lives when colliding with enemy
   Result: PASS

Enemy Movement/Collision Testing

5. GFT5

   Type: Functional, Dynamic, Automated

Initial State: Within the game state

Input: No input

Expected Output: Enemies move on a valid path

Output: Enemy does not go through barriers

Result: PASS

9. GFT9

Type: Functional, Dynamic, Automated

Initial State: Player collects the big dot

Input: No input

Expected Output: Enemies change colour

Output: Enemies changed colours

Result: PASS

14. GFT14

Type: Functional, Dynamic, Automated

Initial State: Player collides with enemy after collection of big dot

Input: Arrow keys

Expected Output: Enemy is removed from game and respawned back to their original cell

Output: Enemy is respawned back to the center of the game

Result: PASS

Scoring Testing

10. GFT10

Type: Functional, Dynamic, Automated

Initial State: Player collects all dots

Input: Arrow keys

Expected Output: Game over screen is activated

Output: Player's score is displayed along with the Game Over screen

Result: PASS

12. GFT12

    Type: Functional, Dynamic, Automated

    Initial State: Player collects dot

    Input: Arrow keys

    Expected Output: The points are increased

    Output: Player's score is increased by 100

    Result: PASS

13. GFT13

    Type: Functional, Dynamic, Automated

    Initial State: Player collects big dot

    Input: Arrow keys

    Expected Output: The points are increased at twice the rate

    Output: Player's score is increased by 200

    Result: PASS

2 Nonfunctional Requirements Evaluation

Test cases will be referenced in this section using their names from the Test Plan - descriptions have been omitted in order to avoid unnecessary repetition.

2.1 Usability

To test usability, a small test group of users (which included acquaintances of the VPB Game Studio development team) were given a copy of the game and a survey to fill out about the application. A Google Form was used to

generate the survey and gather results.

Figures 1, 2, and 3 show the results of the general usability questions that was asked of the users. From the figures we can see that 70% of testers gave Namcap a 10 out of 10 for entertainment value and a user-friendly interface. This was important as the main goal that drove the creation of Namcap was to provide an entertaining game for users that allowed them play from the comfort of their homes (as is the case with any game). Figure 3 shows that 30% of testers gave the game less than 5 out of 10 for difficulty, 30% gave it 5 out of 5, and 40% gave the game greater than 5 out of 5. This shows that Namcap is well balanced and will keep providing users with entertainment.

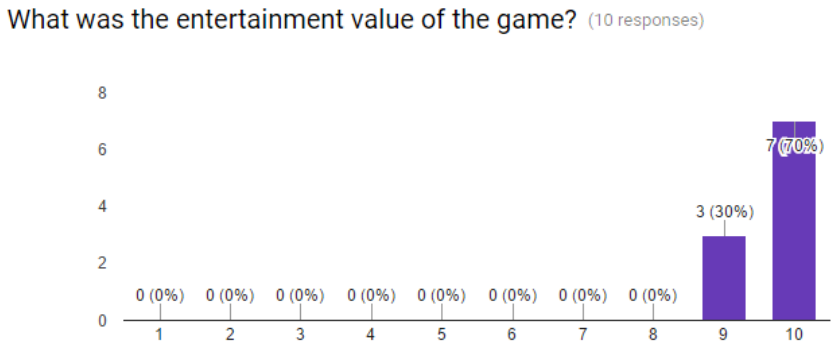

Figure 1: Survey Question: Entertainment

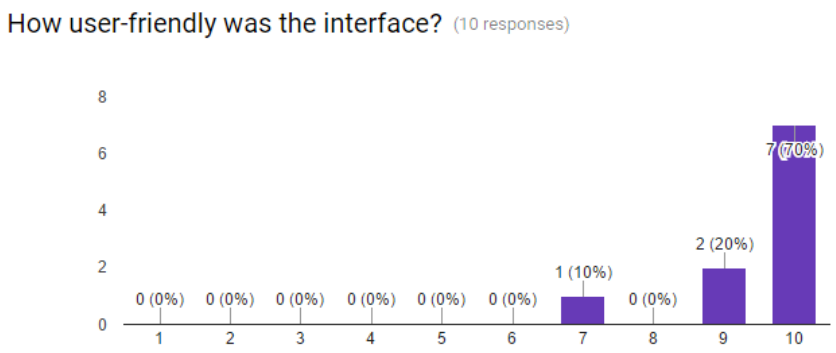

Figure 2: Survey Question: User-friendliness

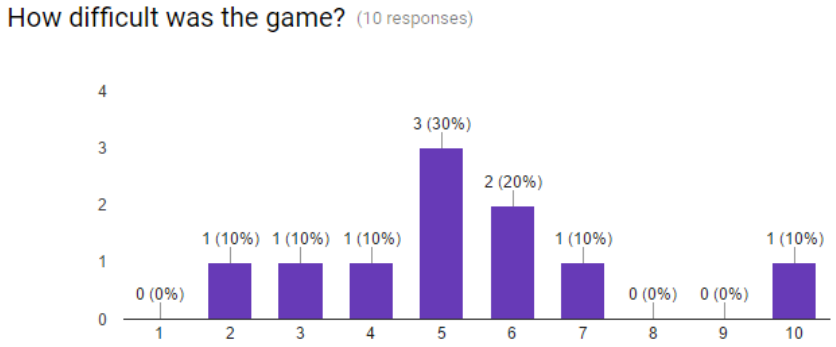

Figure 3: Survey Question: Difficulty

To address non-functional requirements UO1, UO2, and UG1, the survey included the questions shown in figures 4 and 5. All of the testers found the instructions page to clearly define the objective, and were able to grasp the over gameplay of Namcap in one play-through. This shows that due to its low learnability curve, Namcap will be able to attract a wide variety of users.

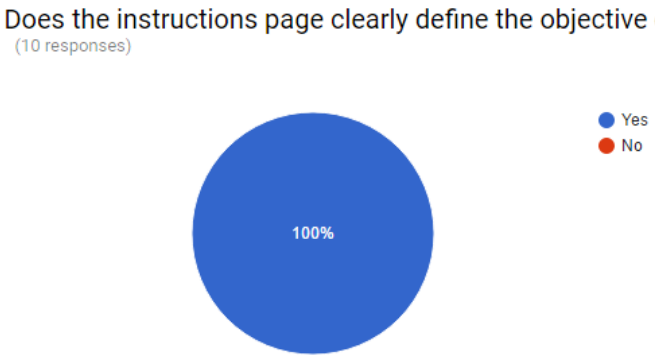

Figure 4: Survey Question: Definition of Objective

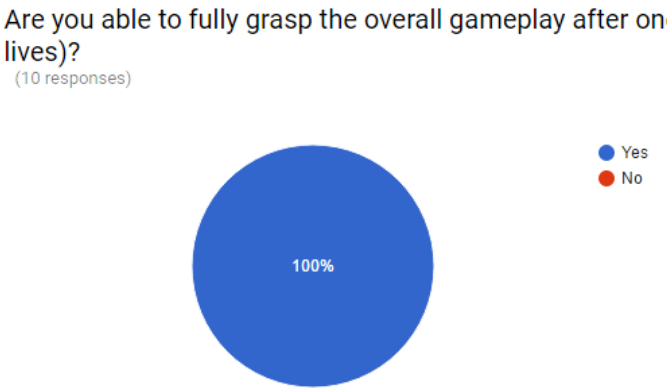

Figure 5: Survey Question: Understandable Gameplay

2.2 Graphical User Interface

The Graphical User Interface of the game was another portion tested by our users through the usability survey. To test requirements AL1 and AL2, users compared the layout of Namcap to that of the original Pacman game.

Figures 6 and 7 show that 30% of users found the instructions page and game interface to look identical to the original Pacman. The changes made due to these results are mentioned in a later section of this document. Figure 8 addresses requirements OST1 and OST2, and it is evident that none of the users found any symbol or text in Namcap offensive. This is an important part of the application since Namcap's ultimate goal is to be accessible to any and all users.

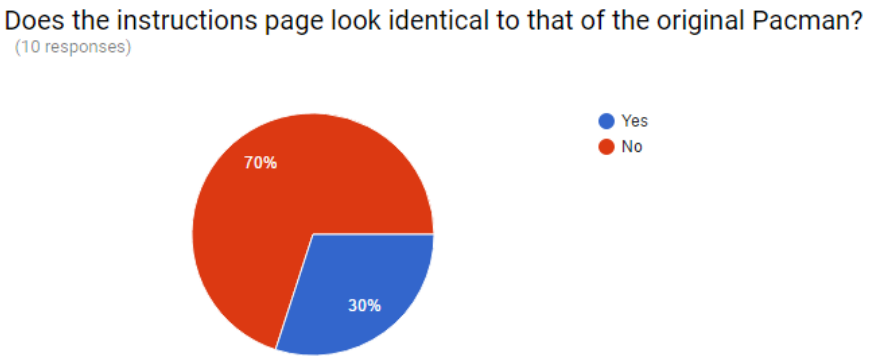

Figure 6: Survey Question: Instructions Page Layout

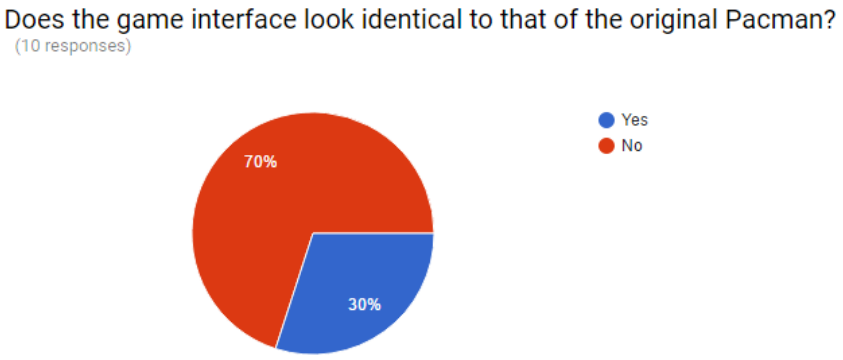

Figure 7: Survey Question: Game Layout

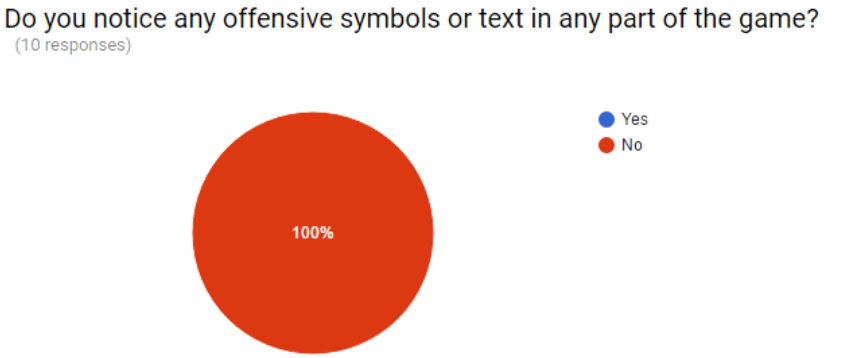

Figure 8: Survey Question: Offensive Media

2.3 Performance

The final two questions on the usability survey addressed the performance of the application. Requirement RT1 was covered in the last survey question (figure 10), and the results show that none of the users experienced delays in

response with respect to their keyboard input. Question 9 in figure 9 shows that none of the testers experienced any unexpected crashes with the application, therefore requirements UF1 and UF2 are also accounted for. It is imperative that Namcap users don't experience technical issues when playing the game, and it is evident from the survey that Namcap's performance meets the requirements.

Formal testing with regards to robustness was not a part of the usability survey for testers. This is because the application accepts user inputs on a set of defined keystrokes - game variables are controlled entirely by the source code so a user is never able to directly affect the game with erroneous input. In this way the application is inherently robust and no explicit testing is required.

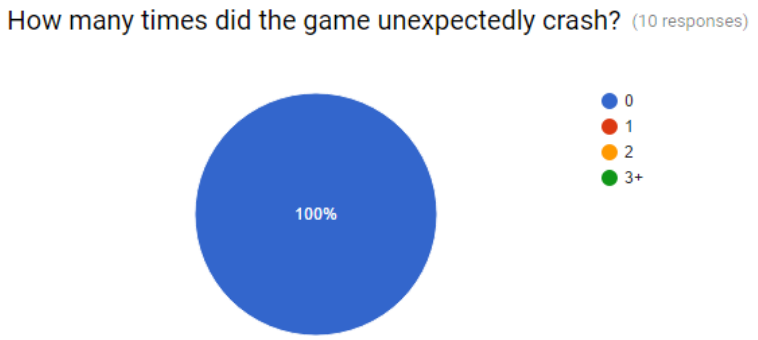

Figure 9: Survey Question: Unexpected Crashes

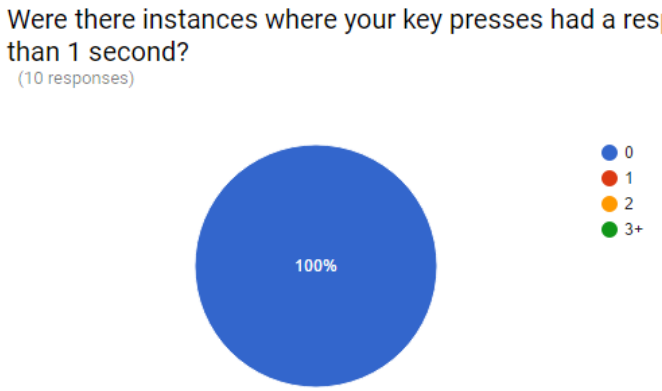

Figure 10: Survey Question: Response Delays

2.4 Other

With regards to the remaining non-functional tests, the development team manually confirmed that the application met the requirements.
Maintainability & Support: OSS1 and OSS2 tested that the jar file containing the application could run on multiple operating systems. This follows suit with Namcap's goal of being accessible to all users.
Security: OSC1 is accounted for because Namcap's source code is stored in a public repository, allowing future development and maintenance updates to be reviewed publicly.
Legal: After looking over the application, the development team concluded that requirements LV1 and LV2 were met.
Health & Safety: BP1 was manually tested, and the game successfully pauses if it has been running for a set amount of time.

3 Comparison to Existing Implementation

As per our Test Plan, we manually performed our system tests on the existing implementation in order to compare game outputs. Since the existing implementation is coded differently, many of these system tests were altered in a way that the two games could be properly compared. We ran the tests

in parallel and focused on core functionality and mechanics of the game over other tests. Since Namcap added features that are not present in the existing implementation, only the following functional tests from the Test Plan were tested for comparison between the applications: GFT2, GFT3, GFT4, GFT6, GFT10, GFT12, RT1, UF1. These tests included those for player movement, player collision with game entities, scoring, and performance measures of response time and application crashes. Namcap was correctly implemented as these tests all produced the same output as those that the tests produced for the existing application.

4 Unit Testing

Unit Testing for Namcap was done using Java's JUnit testing suite, and results of all tests were written and summarized to a text file. If any tests failed, the exception would be included so the development team could analyze and repair any errors. Figure 11 is an example of the text file.

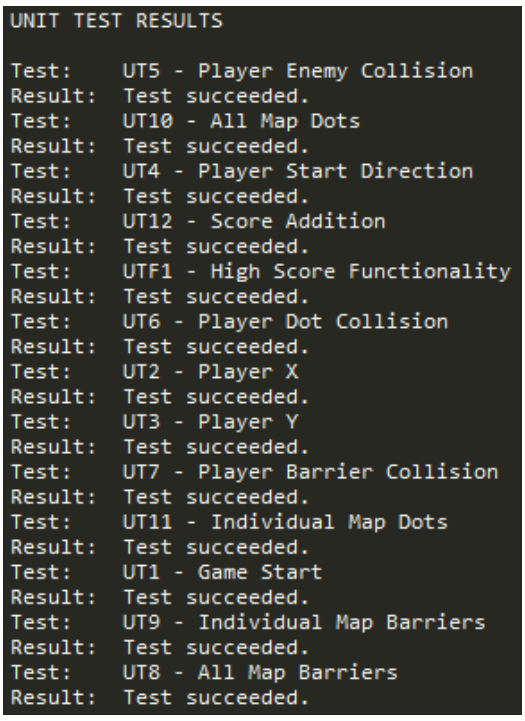

Figure 11: Unit Test Results

Test Cases

1. UT1

   Type: Unit, Static, Automated

   Initial State: Application is displaying the main menu page

   Input: Start Game button action is performed

   Expected Output: New game is started and window is changed to reflect a new game state

   Output: New window (game board) successfully opened

   JUnit Test Result: PASS

2. UT2

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: Current X accessor method for the player

   Expected Output: 200 (start X position)

   Output: 200

   JUnit Test Result: PASS

3. UT3

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: Current Y accessor method for the player

   Expected Output: 300 (start Y position)

   Output: 300

   JUnit Test Result: PASS

4. UT4

   Type: Unit, Static, Automated

Initial State: Player is in starting position at the start of the game

Input: Current direction of the player

Expected Output: 'R' (player starting direction)

Output: 'R'

JUnit Test Result: PASS

5. UT5

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: PlayerX, PlayerY, EnemyX, EnemyY (200,300,185,300)

   Expected Output: Player lives decremented (player to enemy collision succeeded)

   Output: 2 (player lives left)

   JUnit Test Result: PASS

6. UT6

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game, all dots are on map

   Input: PlayerX, PlayerY ([180,300],[20,180],[20,180])

   Expected Output: Score increases the first two times, but not the last

   Output: ([score increases to 100, no dot],[score increases to 200, no dot],[score remains at 200, no dot])

   Junit Test Result: PASS

7. UT7

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: X and Y positions around the player (Player Positions tested (X,Y): [200,300],[20,20],[100,180])

Expected Output: 2 barriers around the first position, 2 barriers around the second position, 0 barriers around the third position - true and false values

Output: ([false,false,true,true],[true,false,true,false],[false,false,false,false])

Junit Test Result: PASS

8. UT8

   Type: Unit, Static, Automated

   Initial State: Board is created with only barrier and dot entities

   Input: X and Y positions for all barrier locations

   Expected Output: True for all barrier locations (manually stated in JUnit class)

   Output: True for all barrier locations

   JUnit Test Result: PASS

9. UT9

   Type: Unit, Static, Automated

   Initial State: Board is created with only barrier and dot entities

   Input: X and Y positions for a location without a barrier, and an update to that location (to create a barrier) [10,15]

   Expected Output: True, a barrier exists for that location [10,15]

   Output: True

   JUnit Test Result: PASS

10. UT10

    Type: Unit, Static, Automated

    Initial State: Board is created with only barrier and dot entities

    Input: X and Y positions for dot locations

    Expected Output: True for all dot locations (manually stated in JUnit class)

Output: True for all dot locations (true that they are all 1)

JUnit Test Result: PASS

11. UT11

    Type: Unit, Static, Automated

    Initial State: Board is created with only barrier and dot entities

    Input: X and Y positions for a location without a dot, and an update to that location (to create a dot) [0,0]

    Expected Output: True, a dot exists for that location [0,0]

    Output: True

    JUnit Test Result: PASS

12. UT12

    Type: Unit, Static, Automated

    Initial State: Board is created with game entities, player and score objects are created

    Input: Score values to increase by ([1000],[1313232],[0])

    Expected Output: Updated score value after each addition ([1000],[1314232],[1314232])

    Output: Score updated successfully ([1000],[1314232],[1314232])

    JUnit Test Result: PASS

13. UTF1

    Type: Unit, Dynamic, Automated

    Initial State: Application is in gameplay state

    Input: Score addition (16000); High score update; Score addition (4000); High score read from file

    Expected Output: High score not affected by score addition within game (should be read as 16000)

    Output: High score remains as updated (16000)

    JUnit Test Result: PASS

5 Changes Due to Testing

There were virtually no changes we made to the actual functionality of the game based on the testing. All of the performed tests were passed with no issues. On the other hand, based on the feedback we received from the usability survey, 30% of the users stated that the game looked too similar to the original Pacman game. Thus, we decided to change the colour of the map from Blue to Red, we switched the roles of the Ghost and the Pacman and added a little backstory to reflect that change and to add a unique perspective to Namcap. Other than the mentioned changes, we did not see any reason to change anything else since it was functioning perfectly well.

6 Automated Testing

As discussed in the test plan, the automated testing was used to test the Functional Requirements and the Unit Tests. For the functional requirements/scenarios, a java Robot class was used to simulate button presses and key presses to verify that the requirements are met. For example, the Robot class was used to simulate the pressing of the arrow keys in order to test if the player moves in the correct direction. The JUnit testing framework was used to validate that every game mechanic (scoring, barriers, player/enemy positions, etc.) is working as required.

7 Trace to Requirements

This section shows the traceability matrix between test cases and requirements. Non-functional requirements are not referenced as they pertain to primarily, application-wide qualities and are trace-able to all tests. Non-functional test cases are excluded as they will cover all requirements as application-wide qualities.

Req.	Tests
F1	GFT1, UT1. UT2, UT3, UT4, UT10, UT11, UT12
F2	GFT2
F3	GFT3, UT7, UT8, UT9
F4	GFT4, UT5
F5	GFT5
F6	GFT5
F7	GFT6, GFT10, GFT12, UT6, UT8, UT10, UT11
F8	GFT7, GFT9, GFT13, GFT14, UT6, UT8, UT10, UT11
F9	GFT8, GFT13, GFT14
F10	GFT9
F11	GFT10
F12	GFT11
F13	GFT12, UT12, UTF13
F14	GFT13, UT12, UTF13
F15	GFT14

Table 2: Trace Between Requirements and Tests

8 Trace to Modules

This section shows the traceability matrix between test cases and modules. Non-functional test cases are excluded as they will cover all modules as application-wide qualities.

Mod.	Tests
M1	GFT1, UT1
M2	GFT2, GFT3, GFT4, UT2, UT3, UT4, UT5
M3	GFT2, GFT3, GFT4, GFT6, GFT7, GFT8, GFT12, GFT13, GFT14, UT2, UT3, UT4, UT5, UT6, UT7, UT8, UT9, UT12
M4	GFT2, GFT3, GFT4, GFT5, GFT6, GFT7, GFT8, GFT10, GFT11, GFT12, GFT13, GFT14, UT2, UT3, UT4, UT5, UT7, UT8, UT9, UT10, UT11, UT12
M5	GFT12, GFT13, GFT14, UT6, UT12, UTF13,
M6	GFT1
M7	GFT2, GFT3, GFT4, GFT5, GFT6, GFT7, GFT8, GFT9, GFT10, GFT11, GFT12, GFT13, GFT14, UT12
M8	GFT11, UT12
M9	GFT2, UT4, UT12
M10	GFT4, GFT5, GFT8, GFT9, GFT14, UT5, UT12

Table 3: Trace Between Modules and Tests

9 Code Coverage Metrics

JaCoCo is the planned code coverage tool to be utilized to act as a metric for code coverage of the implementation. By default, a JaCoCo agent can be attached to a JVM on start. Whenever a class is loaded JaCoCo instruments the class in order to see when it is called and which lines are executed. It then uses this information to build the coverage statistics which creates a results file when the JVM terminates. Based on time constraints for this development, the code coverage metrics tasks have been moved to future development.