Software Testing Report

Group Name:
Cohort 1, Group 4

Group Number:
Cohort 1, Group 4

Members:
Jude Hall
Rosie Hogg
Ishraan Ismail
Sam Wildgust
Ruby Hanson
Tom Devany
Tomas Asllani

Testing methods and approaches

To test our code throughout implementation, we created a test plan that would meet all of the requirements for the assignment, and targeted key features that needed to be tested. These tests were automated as much as possible to keep testing simple, and to enable easy calculation of coverage for tests, and overall statistics.

All tests were documented with identification, descriptions, and results. Manual tests had more documentation with steps followed, expected results and actual results. The test identification was used to relate the tests to our requirements, using a traceability matrix. This was appropriate for the project as it ensured we met requirements, and kept track of what each test was targeting.

Our code was stored in a GitHub repository, and the environment for testing was Visual Studio. This was the same environment as the source code was written in, so it was ideal for our project. We made use of JUnit, gradle, and mockito when creating our tests. Using the GitHub repository made it easier for our whole team to see any updates made to tests by individuals, and to combine all changes together. We opted to separate our tests into individual branches on GitHub. This was decided as when we had one branch for creating all tests, it was impossible to make changes without breaking another team member's work. Our decision benefited the project as we were able to make tests simultaneously.

We ensured that all important cases in our game were covered, and that statement, branch and function coverage was considered. Though our overall coverage could have been higher, for the time we were given for this project, and how huge of a scale covering the entire code with tests would have been, we feel we achieved good quality tests and targeted key areas in our tests. Targeting key areas of code was beneficial for the progress of our project.

The software changes were evaluated based on the test results. As our tests passed for the majority, this benefitted the project as the game source code could be continually edited. The final version of the game passed all the tests. Test plans were also refined as the code was developed, and improvements were made throughout the assignment.

To keep our code high quality, we ensured tests were clarified by making them clear and concise, and additionally using comments and doc string where necessary. To reduce code duplication, helped methods were used on occasion, and test setup code was moved to @Before methods. To check preconditions, invariants, or postconditions, 'assert' was used frequently within our tests. This method was very useful as it allowed for checking these conditions at runtime, and allowed for easily checking basic conditions. This benefitted the project as if another team member wanted to make an addition, it was clear what the tests did and what results were expected.

Testing Report

Both methods of testing, automated and manual were used to verify the correctness of the game implementation. The majority of the tests were automated unit tests done with Junit, focusing on the requirements like timer, movement and score calculation. The rest of the test was manual, which tested features that were more related to user experience such as map visibility and pause and resume feature. A traceability matrix was used to make it easy to track between requirements and tests. This table was maintained throughout testing.

The timer (FR_TIMER) was unit tested. The test included normal operation, boundary cases and invalid cases. It showed correct initialization, time decrement, behaviour at zero and at negative time values as well as being displayed in the correct format. All 9 tests passed in the original and final versions. The testConstructor() verified that the timer was initialized correctly. The testAdd() tested the add method by adding a fixed amount of time and checking that the timer updated correctly. The testAddGradually() tested the add method by checking that it adds time over multiple calls. The testGetTimeLeft() tested that the time was changed accordingly when time had elapsed. The testGetTimeLeftAtZero() tested its respective method for time at the minimum of 0s and at the maximum of 300s. The testGetTimeLeftNegative() tested that its respective method would negate the time accordingly when it was set over the maximum. The testGetClockDisplay() tested that the timer display had the correct output. The testGetClockDisplayWithLeadZero() tested the format of the timer. Finally the testMultipleAddCalls() tested that when multiple calls are made to the timer, the correct total time is added.

Player movement (FR_MOVEMENT) was tested by automated tests that checked directional input handling, correct movement distances, speed and cases where you cannot move(by intention). These tests covered all scenarios and all 8 tests passed in the original and final versions. The testUpMovement(), testDownMovement(), testLeftMovement() and testRightMovement() all tested the functionality of the player's directional movements. The testFailureMovementCases() tested that when no valid input is given, no movement occurs. The testMovementReturnsCorrectDistance() tested that a valid movement will return the correct distance moved. The testPlayerInitialisation() tested that the player is initialised to the correct position for future movement. Finally, the testPlayerSpeed() tested the player's default speed and speed when updated.

Map functions (FR_MAP) were tested by using automated tests. Through these tests we confirmed that the map is always visible to the player and the boundaries set worked. All 5 tests passed in the original and final versions. The testInit() tested that collision rectangles were created correctly. The testInitNamedObjects() tested that objects with and without names are initialized correctly. The testSafeToMove() tests that movement can be made when there is no collision overlap, and verified distant collision objects don't block movement. The testNotSafeToMove() tested that movement cannot be made when there is a collision overlap. The testRemoveCollision() tested that named rectangles are removed correctly.

Score calculation (FR_SCORE_CALC) was unit tested and we checked that score calculation with or without the event bonuses, depended only on the remaining time and the

bonuses given by an event. All 2 tests passed in the original and final versions. The testCalculateScore_noBonus() tested that the score is solely dependent on time. The testCalculateScore_withLongboiBonus() tested that the score should be dependent on both longboi and the time left.

Invariant properties (FR_INVARIANT) were tested manually such as player responsiveness to input, the game always having means of winning and timer always tracking player time. Elements in the first phase that were able to be repeated, such as checking the player's score with respect to specific variables being set, items collected and the game's state, were tested using unit tests. Most of these unit tests were applicable in the second phase of testing post game modifications.

Events handling (FR_EVENTS) was tested manually ensuring that the game detects and triggers the right event based on the player's actions. In addition to manual tests, internal item flags were tested for invariance by making sure they remained set to true for the whole game as no other function calls affected them.
Parts of the events where the player must have collected an item to unlock/overcome a problem were tested to check whether the player could advance both with and without the special item, asserting respective outcomes. testItemCollectionPersists().
When it came to developing tests for the dean event, these were done mainly via testing the dean itself. If the player can be caught by the dean when both visible and not, testDeanReachPlayerVisible() and testDeanReachPlayerInvisible().

Pause/Resume functionality(FR_RESUME) was primarily tested manually by playing the game and switching between pause and resume frequently. This confirmed that when the game is paused, everything halts and when we resume we start exactly where we left off.
In the first phase of testing, the game's states were tested using unit tests. testGameStateAlwaysValid(). This was to make sure the game could only change the state to between the zeroth state (the game hasn't started) and the fourth state (the game has been lost).

Game reset(FR_RESET) was tested manually as well to ensure that after the timer runs out the game can be reset to initial state and you can start to play again. This failed in the first testing phase of the original game as the game could not be restarted from the 'You Failed' display, instead the user had to close the game and reopen it to restart. We fixed this in our additions to the game, and there is now a reset functionality that meets the expected requirements. This test passed successfully in testing phase 2.

Testing prioritized complex components which could break the intended behaviour of the game such as timer, movement, scoring etc. Simple getters, setters and constructors were not included in the testing phases. Line and branch coverage were deemed enough for this project as the main focus was gameplay functionality.

All the tests passed successfully, this shows that the functional requirements are implemented correctly and do what they are expected to do in any case scenario. We did most of the testing using automated unit testing, which allowed for consistency and repeated

checks on the game logic. The few manual tests we performed were done to check system behaviour when interacting with it.
From these results we feel confident in saying that our tests show a good level of completeness, with very clear traceability between the requirements and tests and thorough coverage of different cases be it normal or special. This approach not only verified the functionality of the current implementation but can also be used in future implementations of the game.

Record of Testing:
https://samwildgust.github.io/maze_game_A2/Record%20of%20testing.pdf

First Testing Phase Documentation:
https://samwildgust.github.io/maze_game_A2/Testing%20phase%201.pdf

Second Testing Phase Documentation:
https://samwildgust.github.io/maze_game_A2/Testing%20phase%202.pdf

(These are also available via our group's Google Drive).