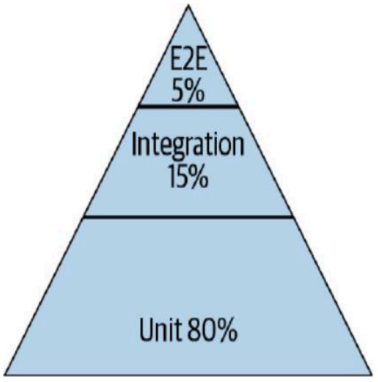
# Software Testing Report

Cohort 1, Group 9
Chris Sewell, Fedor Kurochkin, Matt Durham, Max Peterson, Oladapo Olaniran, Wojtek Tomaszewski, Yuqi Fu.

**Testing Approach**

For this project, our group followed the framework from the Test pyramid at Google in terms of number of tests (see figure on the right) Winters, T., Manshreck, T. and Wright, H. (2020). Therefore we decided to mostly focus on testing the code via Unit Testing the products code.




Aspects of the code or the whole game that could have not been tested on their own were either manually tested or tested using other game mechanics. These are however a minority and most code that contains game logic has been unit tested.

We chose this approach because:

- It properly ensures the logic of the game functions properly
- Unit Tests fit best for most of the deterministic game logic and allows to separate game logic from player input or asset dependencies
- Unit Tests run the same way every time and are not impacted by player actions
- Unit Tests are executed quickly and provide repeatable feedback
- Unit Tests allow for better coverage of edge cases

**Testing Methods**

In order to Unit Test the game, we have used JUnit5 for writing tests.

We used the headless game simulation engine in LibGDX in order to separate game logic from any other dependencies such as graphics and other assets.

We used Gradle to run our tests, manage dependencies and produce testing reports for our code.

In order to ensure traceability between the game tests and the game requirements. We have decided on a traceability strategy that involves constructing a traceability matrix because it gives us a systematic way of checking that requirements are met. This ensures our tests have meaning and challenge the game requirements adequately.

Our Manual tests were conducted by a designed framework where we followed the steps below to ensure least room for human error and accurate assessment of a test passing or failing:

- Name a test
- Action
- Expected output
- Actual output
- Pass or Fail with reason

**Testing Results and Statistics**

Our test environment consisted of using the headless mode in LibGDX, Junit5 and Gradle. All running on PC.

Unit Tests results (Given by Automated Testing report, Available on website)

| All Unit Tests | Passed | Failed | Skipped |
|---|---|---|---|
| 71 | 71 | 0 | 0 |

Manual Tests results

| All Manual Tests | Passed | Failed | Skipped |
|---|---|---|---|
| 18 | 18 | 0 | 0 |

Coverage

| Line | Branch |
|---|---|
| 11% | 11% |

Line and branch coverage were collected using JaCoCo. The coverage across the whole project reached 11%. This is due to inclusion of rendering, UI, asset loading and other game aspects that are not unit testable meaningfully. However the components that do contain game logic or other crucial game systems, the coverage is substantially higher like the fixedStep component (80% line coverage and 66% branch coverage). Coverage was used diagnostically to identify untested logic paths rather than as a quality target.

Game Components Tests

| Component | Number of Tests (Unit and Manual) | Passed | Failed | What its responsible for |
|---|---|---|---|---|
| Achievements | 1 | 1 | 0 | Game Achievements |
| Asset Loading | 0 | 0 | 0 | Extracting and Loading Assets into the game |
| Components | 32 | 32 | 0 | Stores |

| | | | | |
|---|---|---|---|---|
| | | | | Individual Game components |
| Factory | 0 | 0 | 0 | Creating Game components |
| Fixed Step | 6 | 6 | 0 | Updating the game state |
| Leaderboard | 1 | 1 | 0 | Game Leaderboard |
| Messages | 21 | 21 | 0 | Creating and Displaying game messages |
| Screens | 4 | 4 | 0 | Creating and displaying the game's UI screens |
| Systems | 24 | 24 | 0 | Introducing the game components into working game systems |
| UI | 1 | 1 | 0 | Game Font |

Traceability to Requirements

A requirements-to-tests traceability matrix was used to ensure that all user, functional, non-functional, and constraint requirements were verified. Each test was mapped to one or more requirements.

The complete traceability matrix is provided on the project website.

Below if a fragment of the matrix

| | Requirement identifiers | UR_ESCAPE_MAZE | UR_POSITIVE_EVENTS | UR_NEGATIVE_EVENTS | UR_HIDDEN_EVENTS | UR_UI | UR_TIME_LIMIT | UR_PAUSE | UR_SCORE | UR_BOUNDARIES | UR_THEME |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tests | | | | | | | | | | | |
| Manualy Tested | | | | | | | | | | | |
| MT-01 | | | | | | | | | | | |
| MT-02 | | | | | | x | | | | | x |
| MT-03 | | x | | | | | | | | | |
| MT-04 | | | | | | | | x | | | |
| MT-05 | | | | | | | x | | | | |
| MT-06 | | | | | | | | | | x | |
| MT-07 | | | | | | x | | | | | |
| MT-08 | | | | | | x | | | | | |
| MT-09 | | x | | | | | | | | | |
| MT-10 | | | | | | x | | | | x | |
| MT-11 | | | | x | | | | | | | |
| MT-12 | | | x | | | | | | | | |
| MT-13 | | | | | | | | | | | |
| MT-14 | | | | | | | | | | | |
| MT-15 | | | | | | x | | | | | |
| MT-16 | | | | | x | | | | | | |
| MT-17 | | | | | | x | | | | | |
| Unit Tested | | | | | | | | | | | |
| AttendanceCheckerComponentTest | | | | | | | | | | | |
| BobComponentTest | | | | | | | | | | | |
| CameraFollowComponentTest | | | | | | x | | | | x | |
| GooseComponentTest | | | | | | | | | | | |
| HiddenWallComponentTest | | | | | x | | | | | x | |
| InteractableComponentTest | | | x | x | x | | | | | | |
| PhysicsComponentTest | | x | | | | | | | | x | |
| PlayerComponentTest | | x | x | x | x | | | | | x | |
| SpriteComponentTest | | | | | | x | | | | | x |
| TimerComponentTest | | x | | | | | x | | | | |
| TransformComponentTest | | x | | | | | | | | x | |
| FixedStepperTest | | | | | | | x | | | | |

There are requirements that have not managed to be covered by tests. These are:

- UR_HIDDEN_EVENT_HINTS - Hints are located on the map in permanent locations already.
- UR_DEAN - No Dean character to test
- FR_DEAN - No Dean character to test
- FR_DIFFICULTY - No Difficulty setting has been implemented
- CR_COPYRIGHT - Not applicable to game testing but verified by inspection
- CR_SPEND - Not applicable to game testing but verified by inspection
- CR_ENGINE - Not applicable to game testing but verified by inspection

Manual testing

Manual testing was done in places where unit testing was not appropriate. Most likely because it required visible assets and UI or game aspects were tested together.
A report of all Manual tests conducted can be found on the teams website.

**Failed Tests/Code Correctness**

In this particular project. All of the code and manually tested aspects have worked and passed all tests.
Tests were designed with the requirements in mind. Branch and line coverage of the code show crucial decision paths are executed. This supports the correctness of deterministic game behaviour.
The game code is constructed in a good modular way which enables for easier testing.
The tests implemented, prove game logic behaves as expected.
However testing the game still had its limitations like rendering, UI and visual feedback not being testable, unless tested manually in gameplay.
In addition, some game systems work with multiple components which need testing together and do not make sense unit testing on their own.
Some risks still remain that could impact the game but are hard to replicate at hand such as:

- Rare timing interactions (for example resuming and pausing game at the same time)
- Hardware dependent performance
- Player behaviour that exceeds the expected patterns

These risks are minimised and mitigated through manual testing rather than game code testing alone.
Overall, the balance between Unit tests and Manual tests enabled for a good coverage of the game functionality and ensured it behaves in accordance with its requirements.

References

Winters, T., Manshreck, T. and Wright, H. (2020) *Software Engineering at Google: Lessons Learned from Programming Over Time*. Sebastopol, CA: O'Reilly Media.