Software Testing Report

Cohort 4 Group 3

Team Members
Charles Fellows
Elif Gunes
Eve Davey
Hongrui Liu
Leo Owen-Burton
Tomi Olagunju
Will Bannerman

3NG

Methods and Approaches:

Testing Strategy

Our project utilises a Risk-Based Testing approach, prioritising the stability of core game mechanics through a combination of Unit testing and Integration Testing. This methodology ensures that as a game grows in complexity, new features do not introduce issues into existing areas.
Given the rapid, iterative nature of our development cycle, we automated our test suite. By doing this, we ensure constant validation of the codebase, allowing the team to focus on feature implementation while maintaining a high standard of software quality.

Our CI/CD Pipeline:

To maintain a stable “Main” branch, we implemented a robust Continuous Integration pipeline via Github Actions. This approach is appropriate for the project as it decentralises development, allowing team members to work independently on feature branches without the risk of “integration hell” [1] .

- **Automated Validation:** Every push to a remote branch triggers an automated build. This provides immediate feedback to developers on whether their changes have broken existing logic.
- **Gatekeeping:** We enforce strict merge requirements. A Pull Request cannot be merged into the main codebase unless it passes the full suite of automated tests. This protects the production-ready state of the software.

Test Implementation and Frameworks

We used JUnit 5 as our primary testing framework, along with Mockito for dependency injection and mocking.

- **Isolation through Mocking:** Mockito allows us to isolate specific classes by simulating the behaviour of complex dependencies. This is particularly appropriate for our game development due to features such as UI or Physics engines.
- **Headless Environment:** A significant challenge in testing LibGDX applications is the need for graphics processing. To utilise testing within the Github Actions virtual environment (which does not have a GPU), we implemented a Headless Launcher. By separating game logic from rendering, we can validate player statistics, movement logic, and game state transitions without requiring a graphical interface.

We also used Tom Grill’s skeleton for testing LibGDX projects [2].

Test Reports

Automated Tests

Helper classes:

- **Headless Launcher:** This is needed to run the unit tests within Github actions, and also helps to keep the tests separate from the main code. This helps with modularity and code-readability.
- **PlayerTestBase:** This is used to separate and modularise the player tests. This means that the test "environment" can be set up for the tests to be run. Due to this separation any issues in initialising the tests can be found and fixed easily.

Unit tests:

- **Player movement tests:** We implemented a few movement tests for the player. For example:
    - Moving on key presses
    - Moving through walls
    - Movement through water
    - Staying within bounds
    - Testing barriers

Asset tests:

- In the future, asset tests should be included. This would verify that key assets which the game relies on to run can be secure.

Integration tests:

- Integration testing is important to check the flow of functionality throughout the code. It is necessary to test that when variables change, this affects other areas of the code correctly, and the flow between aspects is altered correspondingly.
- It would not be difficult to implement integration tests, because of the code set-up so far. We currently have a fully implemented headless launcher which is a necessary base for the integration tests.

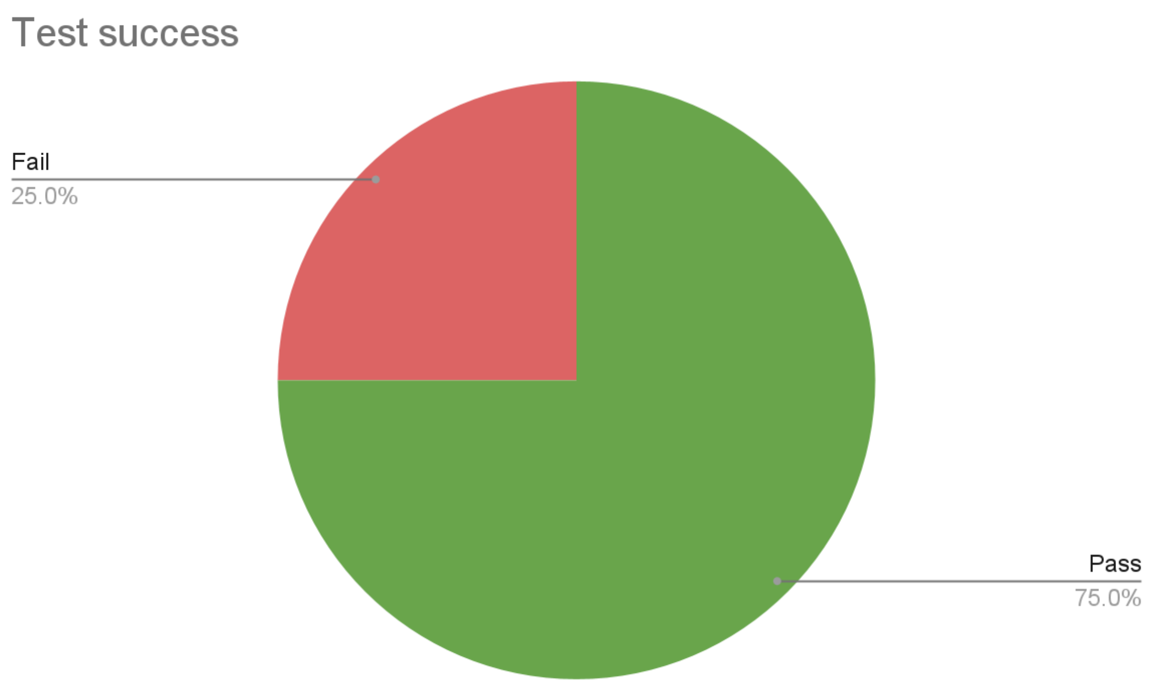
Improvements

To improve our test suite, it would be beneficial to include coverage monitoring. This would help us to ensure that all parts of the game are thoroughly tested, identifying gaps in testing efforts. This process would improve software quality by revealing untested areas that may contain defects, which would lead to more reliable and effective game testing.

Manual Tests

Our current QA workflow involves running manual templates post-prototype release to catch edge cases that automated "smoke tests" might miss. All failures are tracked via Github bugs.
To improve efficiency, we could transition 04.1 (Torch), 04.2 (Keycard), and 02.2 (Cutscene State) to automated tests. By checking internal variables immediately following method execution, we can ensure state integrity. This will require some refactoring to modularise the codebase, but it will significantly reduce manual overhead in the long run.

Statistics:



We had 12 total implemented manual tests. These spanned a wide area of the game, and each test's area is visible on the spreadsheet on the Testing page.

Material:

Required testing material can be found on the website:
https://xqg506.github.io/Escape_University_Website/testing.html

OS[1] GeeksforGeeks, "What is CI/CD?," *GeeksforGeeks*, May 11, 2021. https://www.geeksforgeeks.org/devops/what-is-ci-cd/

[2] TomGrill, "GitHub - TomGrill/gdx-testing: This is a skeleton for libGDX projects which require testing with JUnit and Mockito.," *GitHub*, 2025. https://github.com/TomGrill/gdx-testing (accessed Jan. 10, 2026).