Software Testing Report

Team 10

Ben Leaver
Ali Tekin
Chihun Park
Faras Hbal
Gergely Gal
Hux Stevenson
Utkarsh Singh

Summary

Our testing approach was strongly influenced by both the RIPR model and the ISO 25010:2011 quality standard, with particular emphasis on functional suitability. The aim was functional completeness and correctness, ensuring that all functional requirements were checked by at least one test, and each test was therefore motivated by a requirement. The RIPR model reinforced the need to isolate components so that faults can reliably propagate into observable failures, following Offutt's distinction between faults, errors and failures [1]. Consequently, the priority was on small and medium scope automated tests using JUnit [2]. We followed the test pyramid approach [3], where unit tests form the majority. We used integration tests in cases where interaction between different components was required to be tested, for example testing that when the player collides with the exit area, the win screen is loaded.

However, with the low-risk nature of the project as a small game, a balance had to be found in terms of time and resources spent on testing, and therefore the focus was on testing the most important and complex parts. There were no automated end-to-end tests due to these limitations, and as functionality of the system as a whole was covered sufficiently by manual tests.

Although testing a real implementation of the game was used where possible, this was not always feasible and test doubles were used to keep the test scope small, and ensure tests were more reliable, particularly as many of the systems in a game are deeply interlinked. The LibGDX headless backend was used as a fake implementation of the graphics and application environment, making tests faster and more deterministic. Some tests could run purely in the JVM, making them even faster, and therefore automated tests were split into two categories: headless and core tests. Another example of test doubles being used was the FakeInput class, which was created to replace Gdx.input during tests, enabling precise control over key presses. Many unused input methods (not required to be tested) were implemented as simple stubs returning default values.

To effectively cover the input space, without exhaustively trying all the test cases, input space partitioning and boundary testing were used. Inputs were grouped into equivalence classes representing distinct states, such as the different movement directions and being paused or unpaused. Boundary testing was applied at the point where behavior changes, this includes tile boundaries in collision detection, and thresholds like the countdown timer reaching zero. Prioritising boundary testing uncovered edge-case faults while keeping the test suite efficient and maintainable, for example, the collision system detection not working for some of the directions. Tests also deliberately exercised invalid and failure scenarios that lie outside these equivalence classes, such as attempting to unlock an achievement using an invalid key verified that the system fails safely without crashing.

JUnit tests were written following best practices and the DAMP (Descriptive And Meaningful Phrases) principle rather than strict DRY, prioritising clarity and self-contained tests. Each test is readable in isolation, with explicit assertions and minimal logic, so its intent can be quickly understood. Duplication was reduced only where this didn't impact readability, using small helper methods and 'BeforeEach' setup to centralise common reset logic not specific to individual test cases.

Test Report

Overall, there are 96 automated tests, 72 of which utilise the LibGDX headless launcher. These extensively cover every single class involved in handling the game's logic. There are 6 manual tests which increase the variety of testing approaches and allow for sufficient testing of the rendering and UI. There are fewer manual tests as those are much less maintainable, and testing the complex logic was seen as more of a priority over testing the rendering. Manual tests focused primarily on UI, with accessibility and player experience being covered later in the User Evaluation. More detail about the input data and steps followed for each manual test can be found in our testing material on our website. The Test Summary can also be found in our testing material document, and shows all tests to be 100% successful.

Traceability

To ensure the requirements were fully covered, a traceability table has been created to map each functional requirement to the tests which check that it has been met. There is at least 1 test for each requirement, and more tests where complex logic can be found (e.g. player movement), as this is where errors are most likely to occur.

Requirement	Description	Test IDs/Class	Test Type	Tests Passed
FR_MAP_THEME	The system shall display a university-themed map.	MT-01 (University-Themed Map Display)	Manual	1/1
FR_MAP_PROTAGONIST	The system shall provide a playable, moveable character.	HeadlessPlayerTests, PlayerTest	Automated	8/8, 2/2
FR_MAP_LIMITS	The system shall stop the player moving through limit/wall objects.	CollisionTest	Automated	11/11
FR_MAP_EXIT	The system shall go to a "Win" state when the player reaches the designated exit location.	FirstScreenTest.winOnExitReached	Automated	1/1
FR_TIMER_RUN	The system shall implement a timer with a 5-minute countdown.	FirstScreenTest.timerDecreases, FirstScreenTest.LoseAfter5Mins	Automated	2/2
FR_TIMER_LOSE	The system shall go to a "Lose" state when the timer reaches 00:00.	FirstScreenTest.LoseAfter5Mins	Automated	1/1
FR_TIMER_UI	The system shall display the timer on the UI.	MT-02 (Timer UI Display)	Manual	1/1
FR_START_MENU	The system shall load a Start Menu ("start paused" state) before gameplay begins.	MT-03 (Start Menu Loads Before Gameplay)	Manual	1/1
FR_PAUSE_TOGGLE	The system shall let the player pause and un-pause the game at any time.	FirstScreenTest.pauseHaltsTime, FirstScreenTest.pauseHaltsPl	Automated	2/2

		ayer		
FR_PAUSE_HALT	The system shall halt the timer and gameplay when the game is paused.	FirstScreenTest.pauseHaltsTime, FirstScreenTest.pauseHaltsPlayer	Automated	2/2
FR_PAUSE_MENU	The system shall display a Pause Menu with the options ‘Resume’, and ‘Quit’.	MT-04 (Pause Menu Display and Options)	Manual	1/1
FR_SETTINGS_MENU	The system shall provide a Settings Menu which can be accessed from the Start Menu.	SettingsScreenTest	Automated	5/5
FR_SETTINGS_SOUND	The system shall enable or disable game audio based on an option in the Settings Menu.	SettingsScreenTest	Automated	5/5
FR_EVENT_NEG	The system shall implement at least 5 visible hindering events.	EnemyTest, PathfindingTest, WetFloorTest, ExamTest, DuckTest, DuoAuthTest	Automated	5/5, 5/5, 4/4, 2/2, 2/2, 5/5
FR_EVENT_POS	The system shall implement at least 3 visible beneficial events.	EnergyDrinkTests, Cointest, HelperCharacterTest	Automated	5/5, 3/3, 3/3
FR_EVENT_HID	The system shall implement at least 3 hidden events, invisible until triggered.	LongBoiTest, BusStopTest, BusTest, DoorTest, KeyTest	Automated	4/4, 2/2, 1/1, 3/3, 4/4
FR_EVENT_CONNECT	The system shall support connected events, where one event resolves another hindering event (e.g. key for locked door).	DoorTest, KeyTest	Automated	3/3, 4/4
FR_EVENT_COUNT_POS	The system shall display and increment a counter for visible positive event interactions.	EnergyDrinkTests, Cointest, HelperCharacterTest	Automated	5/5, 3/3, 3/3
FR_EVENT_COUNT_NEG	The system shall display and increment a counter for visible negative event interactions.	EnemyTests, WetFloorTest, ExamTest, DuckTest, DuoAuthTest	Automated	5/5, 4/4, 2/2, 2/2, 5/5
FR_EVENT_COUNT_HID	The system shall display and increment a counter for hidden event interactions.	LongBoiTest, BusStopTest, BusTest, DoorTest, KeyTest	Automated	4/4, 2/2, 1/1, 3/3, 4/4
FR_SCORE_TIME	The system shall calculate a base score, where a faster escape time means a higher score.	FirstScreenTests.scoreDecreasesWithTime, LeaderboardTests.sortScoresHighestToLowest	Automated	1/1, 1/1
FR_SCORE_EVENT	The system shall increase or reduce the base score based on event interactions.	Cointest, HelperCharacterTest	Automated	3/3, 3/3
FR_STATE_WIN_UI	The system shall display a win screen showing final score/time	MT-05 (Win Screen Display)	Manual	1/1

	when the “Won” state is entered.			
FR_STATE_LOSE_UI	The system shall display a lose screen when the “Lose” state is entered.	MT-06 (Lose Screen Display)	Manual	1/1
FR_LEADERBORD_SCORE	The system shall save player scores to a JSON to persistently store the highest user scores.	LeaderboardTests	Automatic	7/7
FR_LEADERBORD_MENU	The system shall display an ordered list when selecting the clicking the leaderboard menu button.	LeaderboardTests	Automatic	7/7
FR_ACHIEVEMENT	The system shall save players data to store a series of achievements they have unlocked that persists between play sessions	AchievementTests	Automatic	6/6
FR_TUTORIAL_ACCESS	The tutorial should be displayed to the user when accessed through the main menu by UI elements.	TutorialTest	Automatic	1/1

Completeness and Correctness

Test correctness is supported by assertions (derived from requirements) that check features behave as expected and would fail otherwise. Components were tested in isolation or using minimal integration so that injected faults reliably propagate to observable failures, meaning that passing a test reflects the system works correctly and it is unlikely that something is secretly failing behind the scenes. Deliberately testing failure scenarios (e.g. invalid achievement keys) further supports correctness by verifying that the system fails safely rather than leading to undefined behaviour.

Test completeness is demonstrated primarily using requirements - every functional requirement is mapped to at least one automated or manual test through the traceability matrix, with additional tests added where logic is more complex or fault prone. JaCoCo coverage shows overall instruction coverage of 41% and branch coverage of 51%, but coverage is intentionally higher in core logic classes. For example, instruction coverage is 97% in Pathfinding and 91% in Collision. Coverage is lower in rendering-heavy classes that can’t be tested using the headless backend. These gaps are addressed using manual UI tests. Rather than having a set goal, coverage metrics were used as an indicator to help focus testing efforts where risk would be reduced the most. Furthermore, the variety of testing used, such as, failure testing, boundary testing and manual testing, help to increase confidence in the coverage of the project as a whole.

Therefore, while test coverage is not perfect, using requirement-based coverage, prioritisation with risk, and a variety of testing techniques, reduces the risks of faults to an acceptable level, given the low-risk nature of a short-term game project.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008.
[2] “JUnit 6.0.1 User Guide,” JUnit.org. [Online]. Available: https://docs.junit.org/6.0.1/overview.html. [Accessed: 04-Dec-2026]
[3] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. Sebastopol, CA, USA: O’Reilly Media, 2020.