# Software Testing

## Cohort 3 Team 4

Kiran Kang

Hannah Rooke

Ben Slater

Abualhassan Alrady

Cassie Dalrymple

Charles MacLeod

Dash Ratcliffe

Harley Donger

# Software Testing report

## Testing Method and approach:

We researched different types of tests that can be written for games to decide which are relevant for our project. Because the game logic (Entities, Player, Events) is tightly coupled with LibGDX graphics classes (like Texture and Sprite) we can't use standard Java unit tests. If we try to instantiate an entity in a standard test, it will crash immediately because the graphics backend isn't running. Therefore, we outlined the following testing strategy:

**Headless Tests**

We used the LibGDX "Headless Backend" to test LibGDX-specific code without opening a whole game window. This simulates the game loop and LibGDX lifecycles but doesn't render graphics to a screen. This allows us to test game logic that relies on LibGDX math (e.g. vectors) or file handling (Gdx.files) without getting a NullPointerException on static Gdx variables.

**Integration Tests**

We used integration tests to check if functions function properly and as expected. This involves testing how multiple systems interact to ensure they work together. Our game relies on an event system where multiple parts interact (e.g. player steps on an item, triggering a DoorEvent, which updates the EventCounter). Testing these on their own is hard, so we wrote tests that instantiate the game world (headless), move the player onto an item, and assert that the correct events fired.

**UI / Functional Tests**

These tests ensure UI buttons actually work. Since the project uses Scene2D for the UI, the logic is separated from the rendering, meaning we can test navigation programmatically. We simulate input events (e.g. button press) and assert that the game UI state changed correctly (e.g. from 'menu' to 'instructions').

**Asset Validation Tests**

In classes like GameScreen, we have hardcoded string paths for our assets. If a file is moved, renamed, or a typo is made, the game will crash at runtime. We wrote a simple test suite that scans our asset directories and asserts that every file referenced in the code actually exists in its correct location.

**Manual Playtesting**

While automated tests verify logic, they cannot verify "game feel", audio balancing, or visual rendering issues. We incorporated a structured manual testing phase where team members followed a script (e.g. pause the game and verify the timer stops) to catch UI and experience bugs that code tests miss.

**Exclusions**

We decided that Automated Playtesting (using a bot) was not to be implemented. For a project of this scale, the effort required to write a bot that can navigate our Tiled map (requiring pathfinding algorithms) outweighs the benefits. We also did not implement Compatibility / Platform Testing because LibGDX handles cross-platform compatibility.

## Test Execution Report

We implemented a foundation of tests based on the strategy outlined above. To support the tests, we added a Headless backend and a mocking library (mockito-core) to build.gradle. We also created a GdxTestBase class to initialise the Headless backend once and mock the OpenGL context to prevent crashes when creating sprites/textures during testing. See the statistics below:

| Test Category | Tests Run | Passed | Fail | Success Rate |
|---|---|---|---|---|
| **Headless Unit** | 8 | 8 | 0 | 100% |
| **Integration** | 3 | 3 | 0 | 100% |
| **Asset Validation** | 1 | 1 | 0 | 100% |
| **UI / Functional** | 1 | 1 | 0 | 100% |
| **Manual / System** | 1 | 1 | 0 | 100% |
| **Total** | 14 | 14 | 0 | 100% |

### Specific Test Cases Implemented

**Headless Unit Test -** testEntityMovementNormalisation

- Target: Entity.java / InputHelper.java
- Verifies that when a player moves diagonally (e.g. up + right), the movement vector is normalised.
- Result: PASSED. The test confirmed that InputHelper.makeUnitVector() returns a vector with length 1, ensuring diagonal movement isn't faster than horizontal/vertical movement.

**Headless Unit Test -** testTimerDurationAndExpiry

- Target: Timer.java
- Verifies that the timer calculates getRemainingTime() as time passes and returns true for hasElapsed() when the duration hits zero
- Result: PASSED. Confirmed that the game clock tracks the 5-minute limit and triggers the end-state logic correctly.

**Headless Unit Test -** testPauseAndResume

- Target: Timer.java
- Verifies that calling pause() freezes the countdown logic and play() resumes it from the stored duration without resetting or losing time.
- Result: PASSED. Verified that the timer correctly holds its state, satisfying the requirement that players can pause the game without penalty.

**Headless Unit Test -** testFinalScoreCalculation

- Target: YettiGame.java
- Verifies that the final score is the points collected plus the seconds left on the timer.
- Result: PASSED. Ensured the calculation logic matches the scoring requirement.

**Headless Unit Test -** testEventIncrementLogic

- Target: EventCounter.java

- Verifies that the static counters for Hidden, Positive and Negative events increment correctly.
- Result: PASSED. Confirmed the game accurately tracks which event types have been triggered.

**Headless Unit Test -** testAchievementEventHooks

- Target: AchievementManager.java
- Verifies that event_hunter is unlocked when a single event is triggered and campus_completionist is unlocked when all events are completed.
- Result: Passed. Events are unlocked exactly as expected.

**Headless Unit Test -** testCollisionDetection

- Target : MapManager.java
- Verifies that the collision system works correctly by mocking the tile map and marking tile (1,1) as blocked then checking that coordinates inside it are collisions.
- Result: Passed.

**Asset Validation Test -** testCriticalAssetsExist

- Target: assets / directory
- Scans the file system to ensure required assets exist.
- Result: PASSED. The test asserted that map/map.tmx, character/player_down.png, Roboto.ttf and all other assets are present. This ensures the game won't crash at runtime due to missing files.

**Integration Test -** testDoorUnlockWithKey

- Target: DoorEvent.java, Player.java, Item.java
- Creates a Player and a Door Item (headless). Tries to open the door (asserts false/locked). Adds the "checkin_code" Item to the player's inventory. Tries to open the door again (asserts true/unlocked).
- Result: PASSED. Verified that the DoorEvent logic correctly checks the player inventory and updates the solid status of the door.

**Integration Test -** testDeanPursuitLogic

- Target: Dean.java
- Spawns the Dean and a Player at specific coordinates and verifies the Dean's movement vector points towards the Player.
- Result: PASSED. Validated the enemy pathfinding logic.

**Integration Test-** testWinEventOpenWinScreen

- Target: Event.java, GameScreen.java , WinScreen.java and YettiGame.java
- Creates headless game instance, mocks screen dependencies, triggers win event.
- Result: PASSED. Verified that triggering the event correctly calculates the final score and transitions the game state to the WinScreen.

**UI / Functional Test -** testMenuToGameTransition

- Target: MenuScreen.java
- Simulates a button click on the "Play" button in the main menu and asserts that the game creates the NamingScreen.
- Result: PASSED. The test successfully fired a touchDown event on the button actor and confirmed the screen state changed.

**Manual System Test: Visuals & Audio**

- Target: GameScreen Render Loop
- Manually verified that the player sprite correctly renders correctly (e.g. behind the Dean when standing "above" him) and that the "Duck Quack" sound effect plays at the correct volume without distortion.

- Result: PASSED. Visual depth perception and audio mixing confirmed correct.

### Completeness and Correctness

Because no tests failed, we assessed the suite for completeness. We focused on the 'Critical Path' - ensuring that assets load, player movement works (Vectors), the game can be won/lost (Timer/Score) and the Event system functions properly. We didn't write unit tests for every getter/setter or trivial UI element because of time constraints. However, the requirements traceability matrix below shows that every high-priority User Requirement and Functional Requirement is verified by at least one test, ensuring the product is robust.

### Requirements Traceability Matrix

The following table links our automated tests to the system requirements.

| ReqID | Requirement Description | Verified By |
|---|---|---|
| UR_MAIN_MENU | Main menu with buttons to start new game | testMenuToGameTransition |
| FR_PAUSING | Game must have a pause function | testPauseAndResume |
| UR_TIME | Game should last 5 minutes maximum | testTimerDurationAndExpiry |
| FR_SCORING | Calculate score based on time taken + point modifiers | testFinalScoreCalculation |
| FR_HIDDEN_EVENTS | Three events must be hidden/tracked | testEventIncrementLogic |
| NFR_RELIABILITY | Game should not crash or fail to load | testCriticalAssetsExist |
| FR_GAME_CAMERA | Gameplay logic for top-down perspective | testEntityMovementNormalisation |
| FR_EASY_DIFFICULTY | Default difficulty (e.g. time, dean movement) | testDeanPursuitLogic |
| UR_EVENTS | Game must have visible/hinder/benefit events | testDoorUnlockWithKey |
| FR_WIN_SCREEN | Display score if user wins | testFinalScoreCalculation |
| UR_STYLE | Art style should be consistent/immersive | Manual System Test: Visuals & Audio |

*Note: Subjective Non-functional requirements (e.g. UR_RATING) are verified through manual review rather than automated code tests.*

## Testing Resources

The testing material URLs (automated reports and manual logs) are provided below:
**Automated Test Results:** Click to view Testing Result
**Coverage Report:** Click to view Testing Coverage Report
**Manual Test Log:** click to view manual test report