Testing

Cohort 4 Group 6

Javengers

Braithwaite, Max
Faruque, Amber
Fu, Zhuoran
Kocaman, Melike
McDermott, John
Rissen, James
Scott, Charlotte

Methods and approach:

In this project, we adopted multiple methods of testing to cover a range of cases. The overall purpose of testing should uncover the presence of any bugs in order to address them so our approach consisted of iteratively implementing tests as a part of our requirements implementation pipeline. This allowed us to smoothly implement features that wouldn't cause issues later down the development pipeline.

Automated unit tests (Junit) were one testing strategy we adopted to cover the narrow testing range. This involved testing methods for object classes as well as calling any routines with different input parameters. Automated testing reduces debugging and improves documentation as well as simplifying the process of designing the code, making changes and reviews. This is appropriate for our project as we were able to automate things such as boundary tests as opposed to manually interacting with them which would leave Integration tests were also implemented as they test the interactions between interlinked components such as a server and its database. These are also implemented with automated Junit testing

Systems tests have a large test scope which often involves manual testing to cover interactions between multiple parts of the system. As not everything is numerically quantifiable, manual tests are necessary to implement. Manual tests will involve running a program with test data or steps and comparing the results to expectations. This will allow us to flag any unwanted interactions. We also conducted a user evaluation which involved observing the way in which our game was blindly interacted with during user testing which also falls under a large test scope. These tests mostly revolved around testing the functional requirements as they need to be assessed through user interaction.

We first created the tests for the preexisting code of the project that we inherited. This was done before we began to implement new features as it required some refactoring of code. This prevented us from developing code that would have future testability issues. This included things such as creating an audio manager class so that automated audio testing was achievable as opposed to manual tests. The previous teams' collisions were also split into a series of methods which reduced dependencies and allowed for optimal automated testing.

This set us up for implementing our new requirements as we strived to adopt a test driven development approach. It involves only implementing a feature into main once tests have been created and run which encourages us to only implement a feature in a way in which it has the foundations to be built upon. This tactic fits appropriately and can be smoothly adopted as the iterative process of it matches with our current agile workflow.

As exhaustive testing is impossible and our project is within a university setting, our approach had to involve prioritising certain tests being created over others to ensure that the core functionality of the project was covered. This involved ensuring that every requirement had some sort of associated testing as this will let our client that we have met requirements Logical testing coverage was prioritised over UI and I/O as these can be tested manually. Due to time limitations, we were unable to reach our ideal test coverage.

Overall Test Metrics/Statistics:

LINK TO TESTING REPORT & TRACEABILITY MATRIX: QUACK? | Testing Report
Total Automated Tests - 241
Tests Passed - 241 (100%)
Tests Failed - 0
Test Classes - 24
Line Coverage - 63%
Branch Coverage - 31.1% (308/991)
Method Coverage - 60.1% (343/481)
Cross-Platform Validation - Windows, MacOS, Ubuntu

Test Distribution by Category (aligns with recommended test pyramid structure):

Test Type	Count	Purpose
Unit	70% (~150-190)	Individual class validation
Integration	25% (~50-90)	Integration testing validating system behaviour
E2E	5% (~5-25)	End-to-end validation of gameplay flows

Test Results & Failure Analysis:

No tests failed in our current implementation. All 241 automated tests pass successfully. This indicates that:

1. All tested functionality works correctly, according to the specification/requirement
2. No regression bugs were introduced during development
3. Expected outputs match behaviour across all test scenarios
4. Test environment is stable and reproducible/extensible

All core systems of our system were well tested and passed/validated perfectly. These include:

- Event System: 95% coverage - all event triggers work correctly
- Audio System: 82% coverage - volume control, mute, playback validated
- Achievement System: Comprehensive testing of all 4 achievements
- Duck Statue Objective: 39 tests ensure win condition functions correctly in given scenarios
- Negative Events: Coffee slip, frog, power cut events all work as designed

Integration Tests Successfully Passed:

- Achievement unlocking under certain conditions
- Room transitioning and map navigation
- Energy system reduction and regeneration

Completeness/Correctness & Coverage Analysis:

Analysis below based on 41 Requirements (excluding CR deadline and java) identified in Req2. All requirements are implemented and tested in some way.

Coverage Level	Requirements	Analysis
Fully tested	22/41 (54%)	Core gameplay/events well tested/validated

Partially Tested	6/41 (14%)	Basic validation but missing edge cases
Manually Tested	13/41 (32%)	UI/Visual or movement aspects manually validated
Not Tested	0/41 (0%)	Unimplemented features

LINK TO REQUIREMENT-TEST TRACEABILITY MATRIX: [QUACK? | Testing]

Strengths of Current Tests:

1. Comprehensive Core Validation:
   - 241 automated tests with 100% pass rate demonstrates our robust implementation of methods/requirements
   - 95% event system coverage ensures all game mechanics work as intended which is important since user interaction systems are more likely for bugs to be found post development
   - Cross-platform validation on Windows, macOS, Ubuntu - provides accessibility
2. Industry Test Design:
   - Test pyramid alignment (70-25-5) follows industry standards
   - Multiple testing approaches: Unit, Integration, System, Manual, Check Style
   - Effective test doubles: Mockito, headless backend, dependency injection
   - Boundary testing for audio, energy and event systems
3. Quality Testing Integration:
   - Checkstyle report (0 errors) ensures that the code quality is up to a good standard
   - Manual test documentation (available on the website)
   - Platform testing methodology that was followed is systematic and reproducible/extensible
   - Coverage measurement was made using JaCoCo which gives a metric and statistics to judge out testing suite
4. Risk Mitigation:
   - Core gameplay thoroughly tested (events, achievements, objectives)
   - Failure scenarios validated: Energy depletion, invalid inputs, edge cases
   - Integration points verified: Achievement unlocking, room transitions, system interactions
   - Regression prevention: All tests pass with no bugs introduced

Gaps & Justifications:

Below shows some of our identified testing gaps and the justification for why they exist. Despite these, we followed best practice to manual test any “tests needing human judgement” since they are harder to automate, as well as focusing on testing the higher-risk areas first and implementing each feature thoroughly before expanding our scope.

Category	Specific gap	Justification	Risk Level
UI/Visual testing	FR_CONTROLS, FR_CAMERA, FR_UI (manual only)	Subjective quality requires manual human judgement	Low/Medium
Low Branch Coverage	31.1% overall branch coverage	Focused on high risk areas first	Medium/High

Missing Automated UI Tests	Screen navigation, button interactions	Technical complexity of automating UI tests with LibGDX	Medium

Test Prioritization & Reasoning:
Our priorisation follows the risk-based testing principles outlined to us. We focused on high-risk/impact areas while accepting lower coverage in less critical UI components since manual testing could be used. This aligns with the pesticide paradox by concentrating tests where faults are most likely and most critical, while maximising fault detection. We also applied equivalence partitioning and boundary testing to event triggers and energy systems, ensuring both valid and invalid inputs were systematically tested. We also didn't cover areas like getters/setters and left most manual tests to areas requiring human judgement (UI/UX).

Priority 1: Main/Critical Gameplay Loop:
- Rationale: Core functionality must work/ be bug-free (mostly) for playable game loop.
- Tests: Event system (95%), achievements, duck statue objective
- Result: Excellent coverage - provides a solid gameplay foundation

Priority 2: User Experience (controls + UI)
- Rationale: Controls and UI are critical for UX and player engagement
- Tests: Manual validation of UI and controls (FR_CONTROLS, FR_CAMERA, FR_UI)
- Result: Manual coverage - automated UI testing is complex and not time efficient for this projects time frame

Priority 3: System Integrity
- Rationale: Game data must not leak/corrupt and game must not crash
- Tests: class manager systems inherited classes are tested like e.g. audio. Cross-platform compatibility also tested for all 3 OS's
- Result: Good coverage - system stability tested and validated

Priority 4: Extra features
- Rationale: Extra added features that are lower priority than core gameplay and less bug prone
- Tests: Character customization, event linking and map access
- Result: Not implemented

Priority Methodology:
1. Risk-based: Highest risk (gameplay loop breaking) tested first
2. Value-based: Highest user value (core gameplay) top priority
3. Feasibility-based: Technically feasible tests implemented first to create foundation
4. ROI-based: Maximum test value for the development time/work given

Future Improvements/Maintenance & Conclusion
Future Improvements: Increase branch coverage to 50%+ via more decision path tests. Complete missing event tests. Implement LibGDX headless UI automation for manual tests.
Maintenance: Weekly test suite reviews. Enforce 70%+ line coverage for new features to keep consistent. Regularly check OS compatibility and maintain test documentation.
Conclusion: Our testing suite validates all implemented functionality with 241 automated tests passing successful and manual validation where necessary. Core systems and events achieved excellent coverage (events: 95%, audio: 82%). Gaps in UI automation are acknowledged and accounted for with manual validation. This foundation supports confident deployment and clear improvement/extensibility.

References:

Sommerville, I 2015, Software Engineering, Global Edition, Pearson Education, Limited, Harlow. Available from: ProQuest Ebook Central. [9 January 2026].