CMPT 276 PHASE 3
TEAM 3

Simarjot Singh | 301615711
Raymond Zhou | 301464055
Maria Leon Campos | 301577820
Shahmeer Khan | 301623019

Monster’s Den - Testing

1. Introduction

The objective of Phase 3 was to design and implement a comprehensive suite of unit and integration tests for Monster's Den. Our goals were to increase behavioral confidence in core gameplay mechanics, ensure design patterns function correctly, and evaluate test coverage and quality.

This report outlines the features we tested, the interactions we validated through integration testing, our testing methodology, our coverage results, and the insights gained throughout the testing process.

No major production code changes were made during this phase. Testing was performed on the game as implemented in Phase 2.

2. Unit Testing

Unit testing focused on isolating and validating individual components of the game. The following subsections describe each feature tested and the corresponding test classes.

2.1 Player Functionality

Test class: `PlayerTest`

We tested core aspects of the `Player` class including:

- Correct initialization of player attributes (name, health, equipped sword)
- Switching between different sword types
- Player health reduction and death conditions
- Attack value calculation based on currently equipped weapon

- Interaction with rewards and penalties

These tests ensured that the playable character behaves deterministically and safely under all regular game scenarios.

2.2 Enemy Behavior: Monster and HumanEnemy

Test classes:

- `MonsterTest`
- `HumanEnemyTest`

We tested two concrete enemy types:

Monster

- Proper initialization of name, health, and damage
- Attack behavior and damage dealing
- Creature death logic

HumanEnemy

- Verification of initial stats
- Behavior differences compared to Monsters
- Damage calculation and conditional behavior (e.g., weaker attack profile)

These tests validated that both enemies behaved according to design and aligned with our Strategy pattern for movement.

2.3 Item & Reward Testing

Test classes:

- `BonusRewardTest`
- `PunishmentRewardTest`
- `RegularRewardTest`

Tested reward-related features include:

- Bonus reward correctly increasing the player’s score/health
- Punishment reward reducing score or health
- Regular reward providing default configured benefit
- Ensuring rewards apply effects only once and do not duplicate state changes
- Defensive tests to ensure negative values and double-claiming are prevented

This section ensures item interactions remain consistent across game loops.

2.4 Design Pattern Tests

Since our game incorporates multiple design patterns, we created unit tests specifically targeting these implementations.

Factory Pattern (WeaponFactory)

- Ensuring correct weapon object is created for a given identifier
- Testing that `SteelSword`, `SilverSword`, and `Sword` hierarchy objects are returned correctly
- Partial coverage for sword classes (intentional — structure heavily tied to pattern logic)

Command Pattern (MoveCommand)

- Testing that executing a `MoveCommand` causes the correct board update

- Verifying that movement is blocked appropriately by walls or invalid positions
- Ensuring undoing commands reverts the game state if implemented

Strategy Pattern (Movement Strategies)

- Verifying each strategy produces the correct output (e.g., aggressive vs random movement)
- Ensuring deterministic strategies behave predictably under repeated runs

These tests verify that our architectural decisions (patterns from Phase 2) are functionally correct.

---

3. Integration Testing

Integration testing focused on interactions between major components, especially where core logic meets the board state.

Important interactions tested (in tests written by teammates and/or combined coverage):

3.1 Player–Board–Movement Interaction

- Validates that a `MoveCommand` executed on the `Player` updates the board and character position
- Ensures the board rejects illegal moves (walls, out-of-bounds)

3.2 Enemy Movement & Collision

- Tests interaction between the enemy's movement strategy and game board
- Ensures collisions and combat triggers are processed correctly

3.3 Player–Item Interaction

- Ensures picking up rewards alters `Player` state

- Validates board update after item consumption

These interactions confirm that individual systems, when combined, behave consistently and safely.

---

4. Test Automation

All tests were implemented using **JUnit 5** and placed under:

`src/test/java`

We adhered to Maven's testing conventions, enabling automated testing through `mvn clean test`. This command automatically compiles tests, solves dependencies, and executes the entire suite. No additional config beyond standard Maven/JUnit setup was required.

Our README was updated to include instructions on:

- Building the project
- Running the game
- Running tests

Ensuring anyone (TA, or external users) can access the code.

5. Test Quality Measures

To maintain high-quality tests, we applied the following strategies:

Clear AAA structure (Arrange–Act–Assert)

Each test isolates behavior and asserts only one primary action.

Boundary & edge case coverage

Examples include:

- Applying rewards to players at minimum/maximum health

- Testing enemies with extremely low or high damage

- Invalid movement directions

Deterministic validation for Strategy pattern tests

Where randomness existed, we restructured tests to validate behavior constraints rather than exact values.

High assertion specificity

Assertions verify exact state values, not vague conditions.

These practices collectively improved test correctness, readability, and maintainability.

6. Code Coverage Analysis

We measured **line** and **branch** coverage using IntelliJ's built-in coverage tool.

Results Summary:

- **Core logic classes:** High coverage (Player, Enemy types, Rewards)
- **Factory, Command, Observer and Strategy patterns:** Medium–High coverage
- **Sword subclasses:** Partial coverage (expected due to pattern abstraction)

Uncovered segments (justifications):

- `Com.team3.monstersden` package (JavaFX launcher / UI entry point) remains uncovered. These classes contain JavaFX startup code and stage/scene wiring, making it difficult to test with standard JUnit and would require a UI testing framework. Since the code is thin and doesn't contain core game logic, we chose to not prioritize unit tests here.

Overall:

Overall, our test suite achieves **77% line coverage** and **70% branch coverage**, with strong coverage across core logic, rewards, enemies, and all major design patterns. The remaining uncovered areas are mainly UI-related and low-impact branches. In particular, the `com.team3.monstersden` package contains JavaFX launcher and UI initialization code, which is not practical to test using standard JUnit without a dedicated UI testing framework. Additional gaps in the Observer pattern, utility helpers, and certain console or defensive branches reflect rare execution paths or framework-driven behavior. Since these segments do not affect core gameplay logic, we prioritized testing the logic-heavy components that directly impact game correctness and player interactions.

7. Findings & Reflections

Key insights from Phase 3 testing:

- **Many subtle logic bugs were ruled out**
  Even though no major production code changes were required, the tests revealed areas where assumptions needed verification (e.g., ensuring rewards do not double-apply, edge-case enemy health values).

- **Design patterns become easier to validate**
  Testing Factory, Strategy, and Command patterns clarified the correctness of the architecture from Phase 2.

- **Testing forced clearer modularity**
  We confirmed that the game logic is sufficiently decoupled from UI and is testable in isolation.

- **Game behaviors are now predictable and robust**
  Damage, movement, item interactions, and state transitions all behave consistently under repeated tests.

8. Conclusion

Through a combination of unit and integration testing, use of JUnit and Maven and structured coverage evaluation, we developed a robust test site ensuring reliability of key gameplay mechanics.

This concludes our Phase 3 report.