CMPT 276 | Group 5 | Haunted Hotel | Project Phase 3 Testing Report

Group members: Kiran Khubbar, Amar Mondair, Karanveer Singh, Prabhjot Singh

---

Introduction

The goal of Phase 3 was to design, implement, and document a complete testing strategy for the Haunted Hotel game. This included writing meaningful unit tests, integration tests, increasing test coverage, and improving code quality based on issues revealed through testing. All tests were implemented using JUnit and organized under the Maven project structure.

This report explains which features were tested, how integration between systems was validated, the quality measures we applied, the resulting code coverage, and what we learned throughout the testing phase.

Testing Strategy

Unit tests were written to verify individual pieces of functionality in isolation. Below are the main features we identified as needing direct unit testing, along with the test classes that cover them.

EventHandlerTest – These tests isolate the event system and verify that event tiles correctly trigger outcomes such as taking damage or teleporting.

- Detecting event collisions
- Fire punishment
- Bomb punishment
- Key checks for teleporting
- Event reset logic

CollisionCheckerTest – These tests ensure that collision logic works independently of rendering and movement.

- Tile collisions in all four directions
- Object collisions
- Entity-to-entity collisions
- Player collision detection

MonBossTest – These tests verify individual aspects of enemy AI and ensure the boss behaves correctly without relying on the rest of the game.

- Constructor initialization
- Switching between tracking and idle behaviour
- Shot counter updates
- Random movement when off-path
- Projectile firing
- Damage reaction
- Item drops

Enemy_SantaTest – This test suite addresses enemy AI and animation

- Sprite loading and animation frames
- Pathfinding activation/deactivation based on distance
- Random movement patterns
- Projectile firing logic and thresholds
- Action counter increments
- Reaction to taking damage
- Boundary clamping for path goals
- Handling edge cases (negative coordinates, overlapping player, large positions)

Test Coverage Summary

Although the assignment does not require full coverage, our test suite achieved very high instruction/branch coverage across most tested subsystems due to:

- Exhaustive event position tests
- Multiple edge cases
- Multiple random-probability attempts for projectile firing
- Direct testing of pathfinding activation/deactivation boundaries

Coverage was used as feedback, not the goal, since our priority was realistic scenario testing.

Integration Tests

Our integration testing focused on making sure the different subsystems of the game actually work together the way they're supposed to. Some of our integration tests were placed in their own dedicated files, while others naturally fit inside tests that were originally written as "unit tests" but ended up checking wider interactions. Below is a breakdown of both types.

Dedicated Integration Tests

PlayerTileCollisionIntegrationTest - Player + Tile System + Collision System

This test class checks how the player, the TileManager, and the CollisionChecker behave when combined during real gameplay movement, as well as adds two important monster-related integration tests. It includes:

- Tile collisions:
  testPlayerBlockedByCollidingTile()
  Makes sure the player cannot walk into tiles that are marked as collidable.
- Movement through free tiles:
  testPlayerMovesIntoFreeTile()
  Verifies the player can move normally when the path is clear.
- World boundary rules:
  testPlayerCannotLeaveWorld()
  Ensures the player can't move outside the map.
- GamePanel calling monster updates:
  testMyersUpdate()
  Confirms that when GamePanel updates, monsters update as well and move.
- Monster projectile behavior:
  testSantaFiresSnowball()
  testBossFiresFireball()
  These verify things like:
    - Santa fires Obj_Snowbal

- The boss fires Obj_Fireball
- Projectiles correctly appear in GamePanel.projectileList

These are all essential interactions since none of these systems work alone during gameplay. Testing them together reflects how the real movement loop behaves.

GamePanelObjectInitializationTest – Game Setup + Object/NPC/Monster Initialization

This integration class verifies that the entire game world is assembled correctly when setupGame() is called. Instead of testing a single object in isolation, this test checks how GamePanel, object factories, NPC spawning, and monster initialization all work together as one system. It ensures that each map receives the correct items and that all core world entities appear exactly where the game expects them.

It validates several major interactions:

- Map-specific object placement
  The test checks that each map is populated with the correct objects:
  - Map 0 receives an Obj_Chest_Key at the expected tile location
  - Map 1 receives an Obj_Myers_Key
  - Map 2 correctly spawns a Christmas Myers Key plus seven gift objects
- NPC spawning
  The presence of an NPC on Map 0 is confirmed after setup.
- Monster initialization across maps
  The test ensures that each major monster type appears on the correct map:
  - MyersEnemy on Map 1
  - Enemy_Santa on Map 2
  - Mon_Boss on Map 3

This test captures the reality that the game world cannot be tested piece-by-piece; object placement, map indices, and enemy spawning all depend on GamePanel coordinating several subsystems together. Testing initialization as one integrated process ensures the game loads into a correct, playable state.

PlayerKeyHandlerIntegrationTest – KeyHandler + GamePanel + Player Movement Loop

This test class focuses on how keyboard input flows through the entire update pipeline, from the player pressing a key to GamePanel updating, to the Player object processing movement, animation, and shooting. Instead of testing KeyHandler or Player alone, these tests confirm the full interaction:

- KeyHandler updates →
- GamePanel.update() gets called →
- Player.update() processes movement, animation, and combat.

It covers several important integration behaviours:

- Directional movement through the real game loop
  Tests such as:
  - testMoveRightThroughGameLoop()
  - testMoveLeftThroughGameLoop()
  - testMoveUpThroughGameLoop()
  - testMoveDownThroughGameLoop()

verify that pressing movement keys actually results in world-coordinate movement after GamePanel invokes Player.update().

- Animation progression
  testSpriteAnimationIncreasesThroughFullUpdate()
  checks that movement triggers animation changes over multiple update cycles, just like real gameplay.
- Projectile shooting through the loop
  testShootProjectileThroughGameLoop()
  ensures that pressing the shoot key causes Player.update() to spawn a projectile through GamePanel, confirming player combat works end-to-end.

Although collision and event logic is stubbed out to avoid interference, these tests still validate the full input → update → action pipeline, which is impossible to test through unit tests alone.

Integration Tests Inside Unit Test Files

Some tests that look like unit tests actually end up being integration tests because the underlying systems are too interconnected to test in isolation. Below are the major examples.

EventHandlerTest - Player + EventHandler + UI + Map Transitions

Even though this file centers on EventHandler, the test cases end up touching multiple systems at once:

- Player movement & map coordinates
- Map transitions
- UI dialogue updates
- Key requirements for doors
- Hazard and teleport events

Examples include:

- testTeleport0to_1() and testTeleport1to_0()
- testCheckEventFire()
- testAllBombEventsTrigger()
- testCheckEventNokey()

Why this makes sense: EventHandler isn't a standalone component; it always depends on the Player, UI, map layout, and tile positions. These interactions happen together in the real game, so they naturally blend into integration tests even inside a unit test class.

MonBossTest, Enemy_SantaTest - Monsters + Player Distance + Projectile System

These test files go beyond simple unit behaviour. They test things such as:

- AI switching between roaming and pathing
- Projectiles being added to the shared list
- Distance calculations based on world coordinates
- Pathfinding through GamePanel tile data
- Damage reactions based on player direction
- Dropped items being stored in gp.obj[][]

These tests are effectively checking how all the areas below work together during gameplay:

- Monster AI
- Player position
- Projectile rules

- Pathfinding
- World geometry

Why this makes sense: Monsters cannot be meaningfully unit-tested alone; they need a GamePanel, a Player, tile sizes, and a shared projectile list. Because of this, their “unit tests” become integration tests by nature.

CollisionCheckerTest - CollisionChecker Integration Behavior

This test class is small, but the behaviors it tests require integration:

- Checking tiles through the GamePanel’s tile map
- Navigating entity and object arrays
- NPC collision detection

Even simple calls rely on world data structures, so these end up being light integration tests.

Test Quality Measures

We applied several practices to keep tests meaningful rather than superficial:

Assertiveness - We avoided empty tests, unnecessary asserts, and “assertTrue(true)” except where required for side-effect functions.

Edge-case Analysis - We explicitly tested:

- Behavior at distance thresholds (e.g., 7- and 20-tile activation ranges)
- Zero-distance positions (enemy directly on player)
- Negative world coordinates
- Very large coordinates
- Projectile counters at 0, 29, and 30
- Randomness by using repeated loops

Independent Instances - Each test creates fresh GamePanel and entity instances to avoid state pollution.

Coverage-Driven Additions - When running JaCoCo, we added tests for branches that were not initially exercised, especially in:

- Damage reaction logic
- Pathfinding boundaries
- Teleportation conditions
- Projectile firing conditions

This improved coverage while keeping tests purposeful.

Findings and Improvements

Writing tests ended up revealing several unexpected issues throughout the codebase. Some event tiles, for example, were not firing correctly because their row and column indexing was slightly off. We also discovered that Santa’s AI occasionally generated pathfinding goals outside the map boundaries, and projectile firing rules were behaving inconsistently because the shot counter did not always reset as intended. Identifying these bugs during testing allowed us to fix logic that likely would have caused unpredictable in-game behaviour.

As we expanded our test suite, our overall coverage rose to **91% line coverage** and **74% branch coverage**, which helped us gain confidence that most major paths and edge cases were exercised. Achieving this level of coverage required refactoring several parts of the code. We removed duplicated logic in multiple monster classes and reorganized

parts of the EventHandler so that punishment methods were easier to isolate and test. We also improved constructors for NPCs and projectiles to make their initialization more predictable, which was especially helpful for integration testing.

Beyond the specific fixes, this phase taught us several broader lessons. We realized that AI behaviour is extremely difficult to debug by observation alone; the tests immediately exposed subtle distance-threshold and pathing issues that would have been hard to spot during gameplay. Integration tests also uncovered problems that unit tests alone couldn't catch, such as incorrect teleporting behaviour or movement beyond map boundaries. We also found that randomness in AI logic required repeated-test strategies to ensure consistent results across probabilistic outcomes. Overall, developing this test suite gave us a much deeper understanding of the game loop and its architectural interactions.

Conclusion

Phase 3 resulted in a comprehensive test suite that exercises the game's most important features, AI behaviours, and subsystem interactions. We achieved strong coverage across units and integrations, and we focused heavily on writing high-quality tests with meaningful assertions and well-chosen edge cases. Combined with the improvements guided by JaCoCo coverage feedback, the project is now significantly more robust and maintainable. The final test suite can be run automatically through Maven by any developer or TA, ensuring that the game remains stable as development continues.