Assignment Report

SFWRENG 3S03 (2026W)

Assignment 1

Author(s)

Salma Baig   baigs19@mcmaster.ca   baigs19

GitHub URL

`https://github.com/hydrowoxy/3S03-A1`

February 22, 2026

1 Testing with generative AI

1.1 a)

I asked ChatGPT to specify three types of tests for the game described above.

The specific prompt I used was:

> Suppose I am developing an action-adventure game such that: a) players explore a post-apocalyptic version of McMaster University; b) up to eight players may be accosted by zombies, mutated professors, angry Deans, killer squirrels, and more; c) gameplay can be both competitive and cooperative; and d) the game will be released on the PS5, Xbox Series X, Nintendo Switch and Sega Genesis (which is 16-bit and hasn't been manufactured since 1997).
>
> Specify three types of tests for such a game.

The response I received was as follows (it originally included more emojis and markdown formatting, which I removed (and re-formatted) so that my document would compile and be legible):

> For a game like this — especially one that spans **modern consoles and a 16-bit relic like the Sega Genesis** — testing needs to cover functionality, performance, and platform constraints very carefully.
>
> Here are **three distinct and appropriate types of testing**:
>
> 1. Functional Testing (Gameplay & Mechanics)
>
> **Purpose:** Ensure the game works according to design.
>
> This would verify:
>
> - Players can explore the post-apocalyptic McMaster campus without falling through geometry
> - Zombies, mutated professors, angry Deans, and killer squirrels spawn correctly
> - Combat, health, inventory, and abilities function as intended
> - Competitive and cooperative modes behave correctly
> - Up to eight players connect and interact properly
>
> Examples of functional test cases:
>
> - If 8 players join a session, all characters spawn correctly.
> - If a Dean is defeated, loot drops trigger.
> - Friendly fire behaves correctly depending on mode (co-op vs competitive).
> - Save/load restores correct state.

This ensures the **core gameplay logic works**.

2. Performance & Stress Testing (Multiplayer & AI Load)

Purpose: Ensure the game runs smoothly under load.

Your game supports:

- Up to 8 players
- Multiple enemy types
- Large explorable campus
- Both modern and legacy hardware

Testing would include:

- Frame rate stability (60 FPS on PS5/Xbox, acceptable performance on Switch)
- Mass zombie swarm scenarios
- Network latency under 8-player online sessions
- Memory usage limits

Especially important:

- The **Nintendo Switch** has weaker hardware.
- The **Sega Genesis** has extreme memory and CPU constraints (16-bit, limited RAM, cartridge storage).

Stress testing would reveal:

- Frame drops when 50 squirrels attack simultaneously
- Network desync in competitive mode
- AI pathfinding breakdowns under heavy load

3. Compatibility & Platform Testing (Cross-Console + 16-bit Constraints)

Purpose: Ensure the game works correctly across very different hardware environments.

You are releasing on:

- PlayStation 5
- Xbox Series X
- Nintendo Switch
- Sega Genesis

Key concerns:

Modern Consoles (PS5 / Xbox Series X)

- 4K rendering compatibility

- Ray tracing support (if implemented)

- SSD load speed optimization

- Controller haptics

Nintendo Switch

- Lower resolution scaling

- Docked vs handheld mode behavior

- Limited GPU power

Sega Genesis

- 16-bit graphics limitations

- Severe RAM restrictions

- Cartridge size constraints

- No modern multiplayer networking

- Likely redesigned gameplay mechanics

Compatibility testing ensures:

- UI scaling works at different resolutions

- Input mapping works on each controller

- Save systems comply with platform standards

- The Genesis version doesn't exceed hardware limits

Summary

The three appropriate testing types are:

1. **Functional Testing** – Does the game behave correctly?

2. **Performance & Stress Testing** – Does it run smoothly under load?

3. **Compatibility/Platform Testing** – Does it work properly across PS5, Xbox, Switch, and Genesis?

Given the extreme hardware diversity (especially including a 1990s 16-bit console), **platform compatibility testing would be especially critical** for this project.

1.2   b)

Two poor, incorrect, or unclear aspects of the response are:

1. Tests are categorized by type (functional, performance, compatibility) but not granularity (unit, integration, system).

   We have seen in class that unit and integration testing are often conflated. This is bad because the distinction affects feedback speed, debugging difficulty, and test design. The AI response not only fails to make this distinction, but also fails to even mention unit testing; it is unclear whether the tests it suggests are to be implemented as unit, integration or system tests (though it seems the AI is relying only on system-level tests based on the descriptions provided). This is a significant omission, as unit tests are crucial for verifying the correctness of individual components in isolation.

   This is unhelpful because it does not indicate how the test suite should be structured in practice or how trade-offs between speed, isolation, and scope should be managed. We have no idea how to best implement these tests in practice.

2. No measurable adequacy criteria are provided for any of the tests.

   Adequacy criteria such as statement, branch, or MC/DC coverage are completely omitted. Coverage is defined as a way to measure a proportion of the structure or domain that a program, test case, or test suite exercises: this helps us understand when testing is sufficient. The AI response provides no guidance on how to determine when testing is adequate.

   This is unhelpful because it provides no guidance on how to organize the tests for fast feedback and effective fault isolation. We have no idea what makes tests "good enough" using the AI's suggestions, and thus no way to know when to stop testing or how to prioritize test cases.

1.3   c)

Two good, valid, or helpful aspects of the response are:

1. It recognizes system-level risks.

   The response lists performance issues, hardware constraints, and multiplayer load as potential failure sources. These issues typically emerge when multiple components interact or when the system is exercised in realistic operating conditions, rather than at the level of isolated units.

   This is helpful because it acknowledges that some failures only become visible when the system is evaluated as a whole. We can use this to identify system testing needs.

2. Concrete examples of test cases are provided for each type of testing.

   The response includes specific example scenarios (eight players joining a session, friendly fire toggling, stress-testing large enemy swarms). Clear and expressive tests are valuable because they make expected behavior explicit and easier to reason about. By describing concrete scenarios rather than only abstract test categories, the response clarifies the intended behaviors that should be exercised.

This is helpful because clearer behavioral intent makes it easier to design tests that meaningfully verify system behavior. We can more directly translate these scenarios into implementable test cases with well-defined expectations.

1.4   d)

Considering the above, three tests I would suggest are:

1. Given the combat damage logic with game mode set to cooperative, verify that when Player A attacks Player B, Player B does not take damage; and when the session is switched to competitive mode, Player B does take damage (covering both decision outcomes).

2. Simulate two players in the same multiplayer session using a test double (stub/fake network transport object that simulates real network communication). When Player A defeats a zombie, verify that Player B's client updates to reflect the zombie's removal.

3. On each target platform build, run the following smoke test: launch → start/join session → combat event → exit session, and verify that no crashes or state inconsistencies occur.

I prefer these tests because they explicitly cover unit, integration, and system granularity while including a measurable adequacy goal (exercising both decision outcomes) for core logic. This results in fast, deterministic tests for core gameplay defects while keeping broad smoke tests to catch platform-specific and full-system failures.

2   Testing with Junit

2.1   Program 1

2.1.1   a)

The loop starts at the last index of the array and decrements `i`, continuing as long as `i>0`. This means it will terminate when `i=0`. This is problematic because it means you will never check index 0.

Using the given test as an example:

We pass `x = [2,3,5]` looking for `y = 2`.

`x.length = 3` so `x.length - 1 = 2`, so the loop starts with `i = 2 > 0` and checks if `x[2] == y`.

`x[2] == 5 != 2`, so this is false.

Then it decrements to `i = 1 > 0` and checks if `x[1] == y`.

`x[1] == 3 != 2`, so this is false.

Then it decrements to `i = 0`. `0 > 0` is false, so the loop terminates without checking if `x[0] == y` (which we know is true).

The program defaults to the sentinel value of `-1`, which is returned when the loop terminates. The program incorrectly returns `-1` even though the target value is present in the array.

Fault:

The fault is in the loop termination condition `i > 0`, which never checks index 0.

Proposed modification to the code:

Change the loop condition to `i >= 0` so that index 0 is also checked.

2.1.2   b)

A test case that does not execute the fault is:

`x = null; y = anything; Expected = NullPointerException`.

Evaluating `x.length` in the loop initialization throws a `NullPointerException` before the faulty loop condition `i > 0` is evaluated; the faulty condition is never executed.

2.1.3   c)

A test case that executes the fault but does not result in an error state is:

`x = [1,2,3]; y = 2; Expected = 1`.

The fault is executed when the loop condition `i > 0` is evaluated. For inputs where the last occurrence of `y` is at an index `i > 0`, the faulty condition does not alter the control flow of the program. The method returns before reaching index 0, so the execution path is identical to that of the corrected implementation. Therefore, no error state is reached.

2.1.4   d)

A test case that results in an error state but not a failure is:

`x = [1,2,3]; y = 5; Expected = -1`.

The faulty loop condition `i > 0` causes the loop to terminate when `i = 0` without checking index 0. The corrected implementation would evaluate the condition at `i = 0` and check `x[0]`, so the execution paths differ. Therefore, an incorrect internal state/control flow/program counter path is reached. However, since `y` is not present in the array, both implementations correctly return `-1`, so no failure occurs.

2.1.5   e)

For the given test case `x = [2,3,5]; y = 2; Expected = 0`, the first error state occurs when `i = 0` and the loop condition `i > 0` evaluates to false.

- In the provided incorrect implementation, the loop terminates and the program counter exits the loop body

- In the corrected implementation (with `i >= 0`) the program counter would enter the loop and check `x[0]`.

Thus, the first error state is the incorrect control-flow decision at `i = 0`.

2.2 Program 2

2.2.1 a)

2.2.2 b)

2.2.3 c)

2.2.4 d)

2.2.5 e)

2.3 Program 3

2.3.1 a)

2.3.2 b)

2.3.3 c)

2.3.4 d)

2.3.5 e)

2.4 Program 4

2.4.1 a)

2.4.2 b)

2.4.3 c)

2.4.4 d)

2.4.5 e)

3 Testing parts of large systems

4 Test driven development (TDD)

5 Test coverage and AI