Assignment Report

SFWRENG 3S03 (2026W)

Assignment 1

Author(s)
Salma Baig baigs19@mcmaster.ca baigs19

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

February 26, 2026

Original template credits: Samuel Zurowski and Charles Barone. Adapted for McMaster SFWRENG by Dr. Istvan David.

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 way to judge when testing is sufficient (or how to prioritize additional tests), so it is unclear when we should stop testing or what parts of the program's behaviour remain unexercised. We have no idea what makes tests "good enough" using the AI's suggestions.

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 = 1; 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, 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, whereas in the corrected implementation (with `i >= 0`) the program counter would enter the loop and check `x[0]`.

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

2.2 Program 2

2.2.1 a)

The loop starts at index `0` and increments `i`, continuing as long as `i < x.length`. This means it checks the array from left to right. However, the method is supposed to return the *last* index of `0`, while the current implementation returns immediately upon finding the *first* `0`. Thus, it returns the wrong index whenever there is more than one zero in the array.

Using the given test as an example:

> We pass `x = [0,1,0]` looking for the last index where `x[i] == 0`.
>
> `x.length = 3`, so the loop runs for `i = 0,1,2`.
>
> The loop starts with `i = 0 < 3` and checks if `x[0] == 0`.
>
> `x[0] == 0` is true, so the method returns `i = 0` immediately.
>
> This is incorrect because there is another `0` later in the array at index `2`, so the expected result is `2`, not `0`.

Fault:

> The fault is the premature return inside the loop: the method returns the first index `i` such that `x[i] == 0`, instead of searching the entire array for the last occurrence of `0`.

Proposed modification to the code:

> Iterate from the end of the array so that the first zero encountered is the last one. Change the loop to:
>
> `for (int i = x.length - 1; i >= 0; i--)`

2.2.2 b)

A test case that does not execute the fault is:

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

The return statement inside the loop is only executed if `x[i] == 0`. Since the array contains no zeros, the condition is never true and the faulty return statement is never executed. The loop completes normally and the method returns `-1`.

2.2.3 c)

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

> `x = [1,0,3]; Expected = 1`.

The faulty return statement inside the loop is executed when `i = 1` and `x[1] == 0`. However, since this is the only zero in the array, the first occurrence is also the last occurrence.

Therefore, the execution path and returned value are the same as in the corrected implementation, and no error state is reached.

2.2.4 d)

This is not possible.

The fault in this program is the premature `return` statement inside the loop, which causes the method to return the first occurrence of `0` rather than the last. If this premature return produces an incorrect index, then the returned value differs from the specification, which is an observable failure. Conversely, if the returned index is correct (for example, when the array contains exactly one zero), then the internal state is consistent with the specification and no error state is reached.

Therefore, any error state caused by this fault necessarily results in a failure, and an error state without a failure cannot occur for this program.

2.2.5 e)

For the given test case, the first error state occurs at `i = 0` when the condition `x[0] == 0` evaluates to true and the method takes the `return i` branch. At this point, the program counter exits the loop and returns `0`, whereas a correct implementation would continue iterating to determine whether a later zero exists (and would ultimately return `2`).

Therefore, the first error state is the premature control-flow decision to return at `i = 0`.

2.3 Program 3

2.3.1 a)

The method iterates through the array from the first index to the last index and increments `count` whenever the condition `x[i] >= 0` is true. This is problematic because the method is intended to count positive integers, but `x[i] >= 0` will also count `0` as positive.

Using the given test as an example:

> We pass `x = [-4,2,0,2]`.
>
> `x.length = 4`, so the loop runs for `i = 0,1,2,3`.
>
> The method initializes `count = 0`.
>
> At `i = 0`, it checks if `x[0] >= 0`.
>
> `x[0] = -4`, so `-4 >= 0` is false and `count` remains `0`.
>
> At `i = 1`, it checks if `x[1] >= 0`.
>
> `x[1] = 2`, so `2 >= 0` is true and `count` is incremented to `1`.
>
> At `i = 2`, it checks if `x[2] >= 0`.

`x[2] = 0`, so `0 >= 0` is true and `count` is incremented to `2`.

This increment is incorrect because zero is not positive, so the correct implementation would have left `count` as `1` here.

At `i = 3`, it checks if `x[3] >= 0`.

`x[3] = 2`, so `2 >= 0` is true and `count` is incremented to `3`.

The loop terminates after `i = 3` and the method returns `count = 3`.

The correct number of positive integers in `x` is `2`, so the method fails by returning `3`.

Fault:

The fault is in the conditional `x[i] >= 0`, which incorrectly counts `0` as a positive number.

Proposed modification to the code:

Change the condition to `x[i] > 0` so that only strictly positive integers increment `count`.

2.3.2 b)

A test case that does not execute the fault is:

`x = null; Expected = NullPointerException`

Evaluating `x.length` in the loop header throws a `NullPointerException` before the faulty conditional `x[i] >= 0` is ever evaluated; the faulty condition is never executed.

2.3.3 c)

A test case that executes the fault but does not result in an error state does not exist.

The fault is executed when the conditional `x[i] >= 0` is evaluated.

The only input value for which the faulty predicate behaves differently from the corrected predicate (`x[i] > 0`) is `x[i] = 0`.

For any test case containing `0`, the faulty implementation increments `count` when the corrected implementation would not. This immediately causes the internal state of the program to differ.

Therefore, executing the fault necessarily produces an error state, and no such test case exists.

2.3.4 d)

A test case that results in an error state but not a failure does not exist.

The only way to reach an error state is for the array to contain `0`. When `x[i] = 0`, the faulty predicate `x[i] >= 0` evaluates to true and increments `count`, whereas the corrected implementation (with `x[i] > 0`) would not increment `count`.

> At that point, the value of `count` differs between the two implementations, so an error state has been reached.
>
> Since `count` is only incremented and never corrected later in the execution, the final returned value must also differ from the corrected implementation. Therefore, once an error state occurs, a failure is unavoidable.

2.3.5 e)

For the given test case, the first error state occurs when `i = 2` and the conditional `x[i] >= 0` evaluates to true for `x[2] = 0`. At this point in the incorrect implementation, `count` is incremented from `1` to `2`. In the corrected implementation (with `x[i] > 0`), the condition would evaluate to false and `count` would remain `1`.

Therefore, the first error state is the incorrect state of `count = 2` after processing index `2`.

2.4 Program 4

2.4.1 a)

The method iterates through the array from index `0` to `x.length - 1` and increments `count` whenever the condition

`x[i] % 2 == 1 || x[i] > 0`

is true. The intention of the method is to count elements that are either odd or positive (or both). However, the condition `x[i] % 2 == 1` incorrectly identifies odd numbers, because negative odd integers do not satisfy `x[i] % 2 == 1` in Java.

Using the given test as an example:

> We pass `x = [-3, -2, 0, 1, 4]`.
>
> `x.length = 5`, so the loop runs for `i = 0,1,2,3,4`.
>
> The method initializes `count = 0`.
>
> At `i = 0`, it checks if `x[0] % 2 == 1 || x[0] > 0`.
>
> `x[0] = -3`. In Java, `-3 % 2 == -1`, so `-3 % 2 == 1` is false. `-3 > 0` is also false.
>
> Therefore, the condition evaluates to false and `count` remains `0`.
>
> This is incorrect because `-3` is odd and should be counted.
>
> At `i = 1`, `x[1] = -2`. `-2 % 2 == 1` is false and `-2 > 0` is false, so `count` remains `0`.
>
> At `i = 2`, `x[2] = 0`. `0 % 2 == 1` is false and `0 > 0` is false, so `count` remains `0`.
>
> At `i = 3`, `x[3] = 1`. `1 % 2 == 1` is true, so `count` is incremented to `1`.
>
> At `i = 4`, `x[4] = 4`. `4 % 2 == 1` is false, but `4 > 0` is true, so `count` is incremented to `2`.

The loop terminates after `i = 4` and the method returns `count = 2`.

The correct number of elements that are odd or positive is `3` (`-3`, `1`, and `4`), so the method fails by returning `2`.

Fault:

The fault is in the expression `x[i] % 2 == 1`, which does not correctly detect negative odd numbers in Java.

Proposed modification to the code:

Replace `x[i] % 2 == 1` with `x[i] % 2 != 0` so that both positive and negative odd integers are correctly identified.

2.4.2 b)

A test case that does not execute the fault is:

`x = null; Expected = NullPointerException`

Evaluating `x.length` in the loop header throws a `NullPointerException` before the faulty conditional `x[i] % 2 == 1 || x[i] > 0` is evaluated; the faulty predicate is never executed.

2.4.3 c)

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

`x = [2,4,6]; Expected = 3`

The fault is executed when the conditional `x[i] % 2 == 1` is evaluated. However, for an array containing only positive even integers, `x[i] % 2 == 1` evaluates to false for every element in both the faulty implementation and the corrected implementation (which uses `x[i] % 2 != 0`). The elements are still counted because `x[i] > 0` is true for each element, so the control flow and final `count` are identical in both implementations. Therefore, no error state is reached.

2.4.4 d)

A test case that results in an error state but not a failure does not exist.

The fault only affects the classification of negative odd integers. For any test case containing a negative odd value (like `-3`), the faulty predicate `x[i] % 2 == 1` evaluates to false, whereas the corrected predicate, `x[i] % 2 != 0`, would evaluate to true.

This causes `count` to be smaller than it should be, producing an error state. Since `count` is only incremented and never corrected later, the final returned value must also be smaller than the correct value.

Therefore, any error state necessarily results in a failure, and no such test case exists.

2.4.5 e)

For the given test case, the first error state occurs when `i = 0` and the conditional `x[i] % 2 == 1 || x[i] > 0` is evaluated for `x[0] = -3`. In the provided incorrect implementation, `-3 % 2 == 1` evaluates to false (since `-3 % 2 == -1` in Java), and `-3 > 0` is also false, so the program does not increment `count`. In the corrected implementation (using `x[i] % 2 != 0`), the oddness check would evaluate to true and the program would increment `count`.

Therefore, the first error state is the incorrect state of `count = 0` after processing index `0`.

3 Testing parts of large systems

3.1 a)

To decouple the agents' behaviour from the core game dynamics, I would ensure that the core module interacts with agents only through the `CatanAgent` interface, without depending on any specific implementation.

The core module enforces the core game dynamics such as invoking agents when it is their turn, applying their actions, checking if actions are legal, updating the `GameState`, distributing resources, and enforcing trade/discard/other rules.

The agents are responsible only for selecting actions through methods such as `chooseMove`, `chooseDiscard`, and `chooseResource`.

Since the current goal is only to verify that agents *can* execute these actions, not whether they choose useful actions, I would provide a simplified test double/stub (specifically, a mock) of `CatanAgent` for testing purposes. This implementation would return predictable, pre-scripted actions without any strategic logic (for example, always selecting the first legal move).

This approach follows the testing principle of isolating the unit under test. By replacing complex agent logic with a simple test implementation, the core module can be tested independently of any learning or strategic behaviour. This ensures that failures during testing reflect problems in the core game dynamics rather than in the agents' decision-making logic.

3.2 b)

```
public class MockAgent implements CatanAgent {

    private int playerId;

    @Override
    public void init(int playerId) {
        this.playerId = playerId;
    }
    @Override
    public Move chooseInitialSettlement(GameState state) {
        // Assuming new Move() creates a valid default move
        return new Move();
    }

    @Override
    public Move chooseInitialRoad(GameState state) {
        return new Move();
    }

    @Override
    public Move chooseMove(GameState state) {
        return new Move();
    }

    @Override
    public Map<ResourceType, Integer> chooseDiscard(
            GameState state, int discardCount) {

        // Empty discard map for testing
        return new HashMap<>();
    }

    @Override
    public ResourceType chooseResource(GameState state) {
        return ResourceType.BRICK;
    }

    @Override
    public int chooseRobberTarget(
            GameState state, List<Integer> possibleTargets) {

        // Assuming for testing that possibleTargets is non-empty
        // and contains valid player IDs
        return possibleTargets.get(0);
    }

    @Override
    public DevelopmentCard chooseDevelopmentCard(GameState state) {
        return null;
    }
}
```

3.3 c)

Other solutions include:

- Implement simple rule-based decision logic for agents, and use this during early testing.
- Capture a sequence of agent decisions from a run and replay them deterministically as regression tests.
- Rely primarily on end-to-end system tests (run full games with real agents) once all components exist.

A solution I would anticipate from a developer unfamiliar with test doubles and isolation is to implement the learning agent first and then test the system by running full games end-to-end, attempting to break things at the system level and debugging issues as they appear.

Points against that approach and in favor of using a stub test double:

1. End-to-end failures do not clearly indicate whether the defect is in the core game dynamics or in the agent logic. Since the stub has minimal logic and predictable behaviour, any failures during testing can be confidently attributed to the core dynamics rather than the agent's decision-making.
2. Learning-based or complex agents may behave differently across runs, making failures hard to reproduce. A stub provides deterministic, repeatable behaviour, which is crucial for automated regression testing.
3. Full-game system tests are slower to run and harder to set up than focused tests of the core module. Isolated tests with test doubles provide faster feedback during development and can be run frequently.
4. With end-to-end gameplay, important edge cases may not occur reliably. With a stub agent, tests can deliberately trigger specific actions and scenarios to ensure the core dynamics are exercised to a sufficient degree of coverage.

3.4 d)

If no one on the team knew about test doubles/isolation, I would seek to address the knowledge gap.

Specifically, I would do the following.

1. Communicate the technical risk of the current testing approach.

   I would explain to the team that relying primarily on end-to-end testing makes failures difficult to localize, introduces non-determinism, slows feedback, and makes regression testing harder to maintain. This increases project risk and reduces confidence in the correctness of the core game dynamics.

2. Demonstrate the alternative approach.

   I would implement a small example test using a stub `CatanAgent` to show how deterministic, isolated testing improves fault localization and repeatability.

3. Provide practical adoption support.

   I would add a few representative stub-based tests to the codebase as templates and document when isolated tests should be used versus when full integration/system tests are appropriate.

4. Integrate the new approach into the development process.

   I would ensure these tests are incorporated into the automated build or continuous integration process so that they are run consistently.

This approach aligns with the CEAB Graduate Attributes (see here) in the following ways:

- Communication skills.

  Engineers must "communicate complex engineering concepts within the profession and with society at large". By explaining the risks of poor testing structure and proposing improvements, I fulfill this responsibility.

- Use of engineering tools.

  Engineers are expected to "create, select, apply, adapt, and extend appropriate techniques, resources, and modern engineering tools to a range of engineering activities, from simple to complex, with an understanding of the associated limitations". Test doubles and isolated testing are recognized software engineering techniques, and adopting them reflects responsible tool selection.

- Professionalism.

  Engineers must understand "the roles and responsibilities of the professional engineer in society, especially the primary role of protection of the public and the public interest". Improving testing practices reduces the likelihood of defects in deployed systems, protecting the public interest.

- Life-long learning.

  Engineers must "identify and to address their own educational needs in a changing world in ways sufficient to maintain their competence and to allow them to contribute to the advancement of knowledge". Identifying a knowledge gap and helping the team adopt improved practices demonstrates commitment to continuous learning.

This question relates directly to my engineering practice because it moves beyond solving a technical problem and into improving how a team works. In industry, I would need to communicate technical risks, justify process changes, and help others adopt better practices. Engineering practice demands collaboration, responsible tool selection, and continuous improvement, not just isolated technical competence.

4 Test driven development (TDD)

Note: for this question, the first iteration occurs in a-c, and the second in d-e.

4.1 a)

Following the TDD flow from class (write a failing test first), I would start by writing a small JUnit test suite that specifies the expected behaviour of `divide()` for: a) a normal case of division that yields a positive integer result, and, b) division by zero. Each test has a single clear assertion (one assert per test) and is fast, independent, and repeatable.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class DivideTests {

    @Test
    public void div_twoInts(){
        Calculator c = new Calculator();
        assertEquals(2.0, c.divide(6.0, 3.0), 1e-9);
    }

    @Test
    public void div_byZero(){
        Calculator c = new Calculator();
        assertThrows(ArithmeticException.class, () -> c.divide(5.0, 0.0));
    }
}
```

4.2 b)

To make the tests pass, I implement only the functionality required by the current test suite.

```
public class Calculator {

    public double divide(double a, double b){
        if (b == 0.0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

4.3 c)

The current `divide()` implementation is sufficient to pass the initial tests, but the functionality is not complete from a testing/specification perspective.

At this stage, the test suite specifies behaviour for only two situations: a typical division case that yields a positive integer quotient, and division by zero. Using the approach of considering input classes and boundaries, several relevant partitions are not explicitly exercised, such as:

- negative dividend and/or negative divisor,
- zero dividend ($0/x$),
- divisions that yield non-integer results,
- floating-point values near zero.

Although Java's built-in arithmetic will likely handle many of these correctly given the current implementation (`return a / b;`), that correctness is not guaranteed by the test suite. In other words, the tests do not yet define the full behavioural contract of `divide()` across these input classes, and therefore do not provide regression protection if the implementation later changes (if we added rounding, switched numeric types, or added special-case logic, for example). The current implementation also won't handle the near-zero case correctly.

Compared to a non-TDD approach, I would likely have tried to account for more cases up front in the implementation. Under TDD, the implementation grows only as far as the tests specify, so missing tests correspond to missing explicit behavioural guarantees.

4.4 d)

Based on the missing partitions identified in (c), I improve the test suite by adding representative cases for additional input classes and boundary-related situations, including floating-point values near zero.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class DivideTests {

    @Test
    public void div_twoInts(){
        Calculator c = new Calculator();
        assertEquals(2.0, c.divide(6.0, 3.0), 1e-9);
    }

    @Test
    public void div_byZero(){
        Calculator c = new Calculator();
        assertThrows(ArithmeticException.class, () -> c.divide(5.0, 0.0));
    }

    @Test
    public void div_negativeDividend(){
        Calculator c = new Calculator();
        assertEquals(-2.0, c.divide(-6.0, 3.0), 1e-9);
    }

    @Test
    public void div_negativeDivisor(){
        Calculator c = new Calculator();
        assertEquals(-2.0, c.divide(6.0, -3.0), 1e-9);
    }

    @Test
    public void div_zeroDividend(){
        Calculator c = new Calculator();
        assertEquals(0.0, c.divide(0.0, 5.0), 1e-9);
    }

    @Test
    public void div_fractionalResult(){
        Calculator c = new Calculator();
        assertEquals(2.5, c.divide(5.0, 2.0), 1e-9);
    }

    @Test
    public void div_nearZeroDivisor_throwsException(){
        Calculator c = new Calculator();
        assertThrows(ArithmeticException.class, () -> c.divide(1.0, 1e-15));
    }
}
```

4.5 e)

To satisfy the additional floating-point boundary test, I improve the division-by-zero guard to treat values sufficiently close to zero as zero. I have chosen 1×10^{-12} as a threshold for "near zero".

```
public class Calculator {

    public double divide(double a, double b) {
        if (Math.abs(b) < 1e-12) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Yes, I would use TDD, but it depends on the task.

I would use TDD for components with clearly defined input-output behaviour, such as business logic or utility methods. In those cases, writing the test first forces me to specify the behaviour precisely and gives immediate regression protection as the code evolves.

I liked that tests act as a contract. The implementation grows incrementally and avoids overengineering, since I only write what is needed to satisfy the current test.

The hardest part is knowing when the test suite is actually sufficient. As seen in this exercise, the first iteration passed all tests but still missed important input classes. Without consciously applying structured test design by thinking about partitions and boundaries, TDD can give a false sense of completeness.

TDD is a helpful technique, but needs to be used in tandem with systematic test design during the "reflection" phases.

5 Test coverage and AI