Assignment Report

SFWRENG 3S03 (2026W)

Assignment 1

Author(s):

Bozhi Zhang	zhanb50@mcmaster.ca	zhanb50
Zicheng Li	li1276@mcmaster.ca	li1276

GitHub URL:

https://github.com/bzz29/3S03/tree/main/a1repo

February 23, 2026

1 Testing with generative AI

a)

AI's response:
Here are three *types* of tests you'd absolutely want for that glorious chaos-gremlin of a game (especially with 8-player multiplayer and the… *Sega Genesis* situation):

1. **Functional / Gameplay tests**
    a. Verifies core mechanics work as designed: movement, combat, enemy AI behaviors (zombies, squirrel assassins), inventory, quests/objectives, save/load, checkpoints, UI, matchmaking/lobbies, coop interactions, friendly fire rules, etc.
    b. Also covers "rule consistency" in competitive + cooperative play (e.g., scoring, win conditions, revive systems, loot distribution).
2. **Performance & Stability tests**
    a. Ensures the game runs smoothly and doesn't crash under real-world stress:
        i. 8-player combat with lots of enemies on screen
        ii. streaming in new campus zones while players sprint in different directions
        iii. long play sessions (memory leaks, overheating, degrading FPS)
    b. Measures FPS, frame pacing, loading times, server tick/latency tolerance, and crash rates across platforms.
3. **Compatibility / Platform certification tests**
    a. Confirms the game behaves correctly on each target platform's requirements and hardware quirks:
        i. PS5 / Xbox Series X certification rules (system UI behavior, suspend/resume, controller standards, online services, error handling)
        ii. Nintendo Switch constraints (handheld vs docked performance, memory limits, input modes)
        iii. Sega Genesis "port sanity" (16-bit limits: memory budgets, sprite/tile limits, sound driver constraints, controller input, cartridge/ROM size constraints, plus "does it even boot reliably on real hardware/emulators?")

b. This is the “same game, wildly different ecosystems” reality check.

If you want to get spicy, the *bonus* test category for this game is network testing (latency, host migration, desync, cheating/abuse cases), but the three above are the big pillars.

b)

1. One thing that AI miss is that the Sega Genesis didn’t have the Internet connection feature. It can’t connect to the modern internet. So, this part needs to be considered carefully again.

2. For performance and stability tests, AI miss is that the Sega Genesis hardware is very old. It might not be able to allow eight players to play at the same time. Sega Genesis only has 16-bit architecture, so it might not carry the modern game configuration. Therefore, this testing plan can’t work on a 16-bit Sega Genesis machine.

c)

1. One thing this AI plan is good at is the extreme test that it tests about the eight players' combat with lots of enemies on screen and long play sessions. This testing idea is great, as it actually tests where and when the game will break. Multiplayer and exploration streaming usually will cause the game to crash, desync and FPS framing screen problems. That makes the test plan helpful.
2. Since the game mixes competitive and cooperative play, it’s easy for the rules to clash. In this case, the AI highlights rule consistency which is very important. This is helpful because testers should test more than basic controls. They also need to check what can go wrong when players play together, like using friendly fire to take someone’s items. That helps make the game fair and balanced.

d)

The controller testing. We can do the controller test on different platforms to test the core actions, movement, camera switching, and attacks. We can do these tests on PS5, XSX, Switch, and for the Sega Genesis, it only has three buttons, so we need to make the test so that it can test all the core actions without causing button conflicts.

The network concurrent multiplayer testing. We can do a multiplayer online gameplay test. We focus on latency and packet loss rate. We can set a range, such as 50–150ms for the latency rate and 1–5% packet loss rate, to see if the results are in this range.

The NPC AI aggro system test. There are many different special enemy types. We can do a test that verifies whether the AI aggro system functions correctly when zombies, mutated professors, angry Deans, and killer squirrels are present. It can check the bugs, such as getting stuck, attacking each other in an endless loop.

We prefer these tests to the ones presented in (a) because these tests are more specific and designed for the mechanics of the game and the hardware constraints. The GenAI test plan is a more general software testing plan, and it is not unique to this game.

2 Testing with Junit

Program 1:

a)

```
/**
    * Find last index of element
    *
    * @param x array to search
    * @param y value to look for
    * @return last index of y in x; -1 if absent
    * @throws NullPointerException if x is null
    */
```

```java
public static int findLast (int[] x, int y)
    {
        for (int i = x.length-1; i >= 0; i--) //Change (i > 0;) to (i >= 0;)
        {
            if (x[i] == y)
            {
                return i;
            }
        }
        return -1;
    }
// test: x=[2, 3, 5]; y=2; Expected = 0
```

The termination condition for the for loop is incorrect. In Java, the position of a number in list starts from 0, not 1. In this case, the function will not check the element at index 0, which is the first element in the input list.

b)

We can give the test with x=null, y=0. This is because we can see that @throws NullPointerException if x is null. So if we set x=null, the program will just throws NullPointerException and it will not execute the program, so it does not execute the fault.

c)

We can set x=[0,1,2,5], y=5. The program will return 3 and the expected is 3. This program executes the fault code (i>0) but does not result in an error state. This is because when the program run at x[3]==5 it returns the correct result, which is 3. There is no error state in internal state since the program didn't run at x[0].

d)

We can set x=[0,1,2,5], y=6. The program will return -1 and the expected is -1. When the program run at i = 0, it will return -1 since 0 > 0, this is an error state. But if we use the correct program to run, when the program run at i = 0, it will also return -1. Since in the list x, the first element is 0 $\neq$ 6. Although the program is in the error state, but it still output the correct result, which is not a failure.

e)

We can set x=[0,1,2,5], y=0. The program will return -1 and the expected is 0. The first error state happened when the loop run at x[0], the loop will not check x[0] since i>0 condition so it will just return -1 immediately. The correct answer is to check x[0] and find the correct y value and return 0.

Program 2:

a)

```
        /**
         * Finds last index of zero
         *
         * @param x array to search
         *
         * @return last index of 0 in x; -1 if there is no zero
         * @throws NullPointerException if x is null
         */
         public static int lastZero (int[] x)
         {
             for (int i = x.length-1; i >= 0; i--)
 //Original: for (int i = 0; i < x.length; i++)
             {
                 if (x[i] == 0)
                 {
                     return i;
                 }
             }
             return -1;
         }
// test: x=[0,1,0]; Expected = 2
```

The order of iterating through the list is incorrect. The original for loop iterated from the beginning of the list to the end, this will cause the program stop when encounter the first 0 in the list x, which cannot find the last 0. We change the for loop iterate from the end of the list to the beginning. In this case, the program can stop when it find the last index of 0.

b)

We can give the test with x=null. This is because we can see that @throws NullPointerException if x is null. So, if we set x=null, the program will just throws NullPointerException and it will not execute the program, so it does not execute the fault.

c)
We can set x=[0]. The program will return 0 and the expected is 0. This program executes the fault code (for (int i = 0; i < x.length; i++)) but does not result in an error state. This is because when the program run at x[0]==0 it returns the correct result, which is 0. There is no error state in the program.

d)
We can set x=[1,0]. The program will return 1 and the expected is 1. In the beginning, the program will run at i = 0, it will check x[0] which is the first element in x list and not equal to 0, this is an error state. In the next step, the program will return 1 since x[1] = 0. But if we use the correct program to run, in the beginning the program run at i = 1, it will also return 1. Since in the list x, the last element is 0 which is x[1] = 0. Although the program is in the error state, it still output the correct result, which is not a failure.

e)
We can set x=[0,1,0]. The program will return 0 and the expected is 2. The first error state happened when the loop run at x[0], the loop returns 0 immediately since the condition is true. The correct answer is to check x[2] and find the last zero is at 2.

Program 3:

a)

```
/**
 * Counts positive elements in array
 *
 * @param x array to search
 * @return number of positive elements in x
 * @throws NullPointerException if x is null
 */
public static int countPositive (int[] x)
{
    int count = 0;
    for (int i=0; i < x.length; i++)
```

```
    {
        if (x[i] > 0)  //Change (x[i] >= 0)to (x[i] > 0)
        {
            count++;
        }
    }
    return count;
}
    // test: x=[-4, 2, 0, 2]
    // Expected = 2
```

We know that 0 is not a positive number in Math. To find the number of positive elements in list x, we need to exclude 0 in program. So, we change the if condition to let each element in list x is greater than 0.

b)

We can give the test with x=null. This is because we can see that @throws NullPointerException if x is null. So, if we set x=null, the program will just throw NullPointerException and it will not execute the program, so it does not execute the fault.

c)

We can set x=[-1,-2,-3]. The program will return 0 and the expected is 0. This program executes the fault code (if (x[i] >= 0)) but does not result in an error state. This is because for each element it run the if condition to check the number, but all the elements are <0 so all the conditions are false, the program counter didn't change neither.

d)

It is not possible. When the program at the error states, which means it will evaluate the number 0 as positive number, since (0 >= 0 is True). In this case, the program will enter the if statement and let the count++. Also, at the end of program, there is no way to let count--. But the correct program will not enter the if statement and let the count++. Thus, in this program, the error state must cause failure.

e)

We can set x=[-4, 2, 0, 2]. The program will return 3 and the expected is 2. The first error state happened when x[2], now x[2]==0, so now the program increases the count by 1. The count is 2 now but it should be 1, so this is the first error state reached.

Program 4:

a)

```
/**
  * Count odd or positive elements in an array
  *
  * @param x array to search
  * @return count of odd or positive elements in x
  * @throws NullPointerException if x is null
  */
  public static int oddOrPos (int[] x)
  {    // Effects: if x is null throw NullPointerException
       // else return the number of elements in x that
       // are either odd or positive (or both)
       int count = 0;
       for (int i = 0; i < x.length; i++)
       {
            if (x[i]%2 != 0 || x[i] > 0)  //Change (x[i]%2 = 1) to (x[i]%2 != 0)
            {
                 count++;
            }
        }
        return count;
  }
        // test: x=[-3, -2, 0, 1, 4]
        // Expected = 3
```

We know that % in Java will remain the sign for negative number.
For example: -3 % 2 = -1. In this situation, the if statement will always false since x[i](negative)%2 will never equal to 1. But -3 is an odd number. So, we change the if condition to correctly evaluate the odd number.

b)

We can give the test with x=null. This is because we can see that @throws NullPointerException if x is null. So, if we set x=null, the program will just throw NullPointerException and it will not execute the program, so it does not execute the fault.

c)

We can set x=[1,2,3]. The program will return 3 and the expected is 3. This program executes the fault code (if (x[i]%2 == 1 || x[i] > 0)) but does not result in an error state. This is because for each element it runs the if condition to check the number, and there is no negative odd number. So, the condition is true for all number, and the program counter didn't change neither. So, there is no error state.

d)

It is not possible. When the program at the error states, which means it will evaluate the negative odd number as false in the if statement, since (negative odd number % 2 != 1). In this case, the program cannot enter the if statement and let the count++. Also, at the end of program, there is no way to let count++. But the correct program will enter the if statement and let the count++. Thus, in this program, the error state must cause failure.

e)

We can set x=[-3, -2, 0, 1, 4]. The program will return 2 and the expected is 3. The first error state happened when x[0], now x[0]== -3, but the program didn't capture negative odd number, the count didn't increase. So this is where the error state happened.

3 Testing parts of large systems

a)

I will create a simple class to decouple. This class will use very simple and legal return in CatanAgent interface. In this way, I can completely test the game process to make sure that agents can execute those actions and do not need to care about whether those actions are actually useful actions.

b)

```
public class Agent implements CatanAgent {

    @Override
    public void init(int playerId) {
        this.playerId = playerId;
    }

    @Override
    public Move chooseInitialSettlement(GameState state) {
        return null;
    }
```

```java

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

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

    @Override
    public Map<ResourceType, Integer> chooseDiscard(GameState state, int discardCou
nt) {
        return new HashMap<>();
    }

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

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

        return 0;
    }

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

c)

If a software developer who is not familiar with the techniques I used, I think he will wait unit the team is finish all action strategy of agent. This is a very simple and safe method.

Convincing points against their solution and in favor of yours:

1. The overall development progress will be very slow. If my test method is used, the two teams can develop in parallel, which is very fast.
2. Hard to find bugs. If game have some problem, it is difficult to find whether the problem lies with the strategy or the testing program.

3. Test will run slowly. In a real game, agent may have a huge amount of action strategies, which may cost a lot of time to do on test. But my solution returns the result instantly. It can greatly accelerate testing efficiency.

d)

If I realized no one in my company/unit/team knows about the techniques I used in this exercise, I will report it to my leader to request a technical exchange meeting. I will prepare a guideline and an easy demo to explain these techniques, which can clearly show the advantages. In the beginning, it's normal to encounter some resistance, since writing interfaces and extra test classes will take some time to learn. But my final except is that the teams can adopts this as a standard practice. In this situation, it will greatly accelerate the development process and reduce bugs. I think this relate to CEAB engineering practice through communication and teamwork. This also can improve quality and reduce risk.

4 Test-driven development (TDD)

a)

Iterations 1:

So, we first create a class for testing, we write some basic test cases for the TDD first iteration.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
public class CalculatorTest {

    @Test
    private static void test_pos_divide() {
         double result = calculator.divide(8, 2);
         assertEquals(4.0, result);
    }

    @Test
    private static void test_pos_neg_divide() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(-6, 2);
```

```
        assertEquals(-3.0, result);
    }
}
```

Iterations 2:

Now we can consider the divide by zero case, which is also very common case for test. This test should throw the exception.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
public class CalculatorTest {

    @Test
    private static void test_pos_divide() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(8, 2);
        assertEquals(4.0, result);
    }

    @Test
    private static void test_pos_neg_divide() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(-6, 2);
        assertEquals(-3.0, result);
    }

    @Test
    private static void test_divide_byzero() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class, () -> calculator.divide(6, 0));
    }
}
```

b)

Iterations 1:

Given the test case from part a, we can write the code for the divide() method.

```
public class Calculator {
public double divide(double dividend, double divisor) {
        return dividend / divisor;
    }
}
```

Iterations 2:

Now we can see from part a, we add a new test case for divide by zero. Now to handle this divide by zero, we need to check if divisor is equal to 0. If it is equal to 0, then we catch this ArithmeticException.

```java
public class Calculator {
    public double divide(double dividend, double divisor) {
        if (divisor == 0.0) {
            throw new ArithmeticException("Can not divide by zero");
        }
        return dividend / divisor;
    }
}
```

c)

The functionality of the implementation in part b is complete with respect to the current test suite. It can handle the division of positive and negative numbers and throw an exception for division by zero. But there are some parts that are missing.

- The not-a-number case, which is NaN, cannot be handled right now. If such invalid input is passed to the function, then it will give unexpected output. So, to handle this invalid input case, we need to throw an exception.

- The current test case only tests when the divisor is equal to zero. We didn't test the case where the dividend is zero. So, we can define new test cases for the case where the dividend is zero.

- Now, when we test the floating number division like a fraction, we need to make the test case have a tolerance of error to ensure the result is within the tolerance. So, we can set the tolerance margin error as 1e-8.

- The current test plan didn't cover the case where the inputs are extreme values like the Double.MAX_VALUE or Double.MIN_VALUE. The test case should throw the exception if the result overflows or underflows the range of the floating-point number.

- The current test plan didn't cover the case where the input is invalid, like the Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY. The test case should throw the exception.

d)

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
public class CalculatorTest {

    @Test
    private static void test_pos_divide() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(8, 2);
        assertEquals(4.0, result);
    }

    @Test
    private static void test_pos_neg_divide() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(-6, 2);
        assertEquals(-3.0, result);
    }

    @Test
    private static void test_divide_byzero() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class, () -> calculator.divide(6, 0));
    }

    @Test
    private static void test_invalid_NaN() {
        Calculator calculator = new Calculator();
        assertThrows(IllegalArgumentException.class, () -> calculator.divide(Double
.NaN, 2));
        assertThrows(IllegalArgumentException.class, () -> calculator.divide(2, Dou
ble.NaN));
    }

    @Test
    private static void test_dividend_zero() {
        Calculator calculator = new Calculator();
        double result = calculator.divide(0, 2);
        assertEquals(0.0, result);
    }

    @Test
    private static void test_fraction_tolerance() {
        Calculator calculator = new Calculator();
```

```java
        double result = calculator.divide(1.0,3.0,1e-8);
        assertEquals(1.0/3.0, result, 1e-8);
    }

    @Test
    private static void test_MAX_VALUE() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MAX_
VALUE, 1.0));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MAX_
VALUE, 0.5));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MAX_
VALUE, Double.MAX_VALUE));
    }

    @Test
    private static void test_MIN_VALUE() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MIN_
VALUE, 1.0));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MIN_
VALUE, Double.MAX_VALUE));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.MIN_
VALUE, Double.MIN_VALUE));
    }

    @Test
    private static void test_InfinityInput() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.POSI
TIVE_INFINITY, 1.0));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.NEGA
TIVE_INFINITY, 1.0));
        assertThrows(ArithmeticException.class, () -> calculator.divide(1.0, Double
.POSITIVE_INFINITY));
        assertThrows(ArithmeticException.class, () -> calculator.divide(234.0, Doub
le.POSITIVE_INFINITY));
        assertThrows(ArithmeticException.class, () -> calculator.divide(1.0, Double
.NEGATIVE_INFINITY));
        assertThrows(ArithmeticException.class, () -> calculator.divide(659.0, Doub
le.NEGATIVE_INFINITY));
        assertThrows(ArithmeticException.class, () -> calculator.divide(Double.POSI
TIVE_INFINITY, Double.NEGATIVE_INFINITY));
    }
```

```
}
```

e)

Now for improve the functionality, we just check the NaN and infinity input using the isFinite method. After we are getting the result, we can use the same isFinite method to check the result. Lastly, we need to make sure the result is not underflow to zero, so if result is zero and dividend is not zero, that mean we found underflow.

```
public class Calculator {
    public double divide(double dividend, double divisor) {
        if (!Double.isFinite(dividend) || !Double.isFinite(divisor)) {
            throw new IllegalArgumentException("Inputs must be finite numbers");
        }
        if (divisor == 0.0) {
            throw new ArithmeticException("Can not divide by zero");
        }
        double result= dividend / divisor;
        if (!Double.isFinite(result)) {
            throw new ArithmeticException("Result is not a finite number, overflow"
);
        }
        if (result == 0.0 && dividend != 0.0) {
            throw new ArithmeticException("Result is underflow to zero");
        }
        return result;
    }
}
```

f)

I would use TDD depending on different situations. If the project needs long-term maintenance, and the business requirement is clear, then I will use TDD. For example, for the divide() method, we found that there are a lot of edge cases that need to be covered by using TDD. The business logic is also very clear for what to do. This project will be maintained, so TDD is the best option for this.

But if the project doesn’t have a well-defined business requirement, then TDD is not the best option. For example, if we are just making a prototype, then it's not worth it to think of testing upfront. Because the prototype will change very fast, and the code will change a lot.   If the project is too simple, like it is just a

code snippet or a script then a full TDD cycle for these things is not worth it. Because the scripts will just use it for one time and then discard it.
There are some elements I liked in TDD. TDD will help me to think about the boundary and edge cases before starting development. It can help me make the specifications early and understand the business requirement very well early on. It can make sure the development code meets the specifications and covers the edge cases without leaving too many errors. This can make the implementation have better maintainability and make the code easier to refactor later.
There are some elements I find the hardest to manage. When using the TDD development, it is very hard to come up with the correct test cases and try to balance test coverage. If we make too many tests and too much detail, then it will be time consuming and make the whole development too slow. But if we didn't use that many test cases, then the quality of the code would not be good, and this would lead to incorrect implementation. It will also make refactoring the code harder.

5 Test coverage and AI

a)
We use GenAI to generate a reasonable amount of JUnit tests. We choose Misses Branches as the complex coverage metric, since it is a reasonably complex metric that ensures every possible logical path.

```
package com.example.shop;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

class OrderSystemTest {

    @Test
    void processOrder_withStudentDiscountAndCard_marksPaidAndReturnsTotal() {
        Order order = new Order();
        order.addItem(new OrderItem("Book", 2, 25.0));
        order.addItem(new OrderItem("Bag", 1, 50.0));

        OrderService service = new OrderService();
        double total = service.processOrder(order, "student10", "card");
```

```java

        assertEquals(OrderStatus.PAID, order.getStatus());
        assertEquals(108.0, total, 1e-9);
    }

    @Test
    void processOrder_withCryptoPayment_cancelsOrderAndReturnsZero() {
        Order order = new Order();
        order.addItem(new OrderItem("Notebook", 1, 20.0));

        OrderService service = new OrderService();
        double total = service.processOrder(order, "BLACKFRIDAY", "crypto");

        assertEquals(OrderStatus.CANCELLED, order.getStatus());
        assertEquals(0.0, total, 1e-9);
    }

    @Test
    void paymentValidator_withNullMethod_returnsFalse() {
        PaymentValidator validator = new PaymentValidator();
        assertEquals(false, validator.isPaymentMethodValid(null));
    }

    @Test
    void paymentValidator_withPaypal_returnsTrue() {
        PaymentValidator validator = new PaymentValidator();
        assertEquals(true, validator.isPaymentMethodValid("paypal"));
    }

    @Test
    void paymentValidator_withUnknownMethod_throws() {
        PaymentValidator validator = new PaymentValidator();
        assertThrows(UnsupportedOperationException.class, () -> validator.isPaymentMethodValid("wire"));
    }
}
```

assignment1

assignment1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.shop		84%		62%	11	33	10	64	1	18	0	7
Total	38 of 248	84%	10 of 27	62%	11	33	10	64	1	18	0	7

assignment1 > com.example.shop

com.example.shop

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
OrderItem		65%		50%	3	5	3	11	1	3	0	1
DiscountService		61%		37%	5	7	4	8	0	2	0	1
PricingService		83%		66%	2	6	2	11	0	3	0	1
Order		86%		50%	1	6	1	11	0	5	0	1
OrderService		100%		100%	0	3	0	12	0	2	0	1
PaymentValidator		100%		100%	0	5	0	7	0	2	0	1
OrderStatus		100%		n/a	0	1	0	4	0	1	0	1
Total	38 of 248	84%	10 of 27	62%	11	33	10	64	1	18	0	7

b)

It takes us approximately half an hour to achieve a branch coverage rate of above 90%.

The strategy we take is a multi-round iteration approach. The GenAI can generate a large number of test cases within a minute, but it cannot assist me in making the code compile and run with the correct configuration. So, we have to manually go through the codes it gives us and copy them to the correct places. After running the tests and give us the report, we need to look at the report to see what the missing part of the current test plan is. Then we tell the AI the missing part, and it gives us the new test cases. So, the process is we run the coverage test, we identify uncovered branches, we give the requirements to AI, AI generates tests, we run the coverage test again. In the end, it takes us 3 iterations of this development loop to achieve a branch coverage rate of 100%.

assignment1

assignment1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.shop		100%		100%	0	33	0	64	0	18	0	7
Total	0 of 248	100%	0 of 27	100%	0	33	0	64	0	18	0	7

assignment1 > com.example.shop

com.example.shop

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
OrderService		100%		100%	0	3	0	12	0	2	0	1
PricingService		100%		100%	0	6	0	11	0	3	0	1
OrderItem		100%		100%	0	5	0	11	0	3	0	1
Order		100%		100%	0	6	0	11	0	5	0	1
DiscountService		100%		100%	0	7	0	8	0	2	0	1
PaymentValidator		100%		100%	0	5	0	7	0	2	0	1
OrderStatus		100%		n/a	0	1	0	4	0	1	0	1
Total	0 of 248	100%	0 of 27	100%	0	33	0	64	0	18	0	7

```java
package com.example.shop;

import org.junit.jupiter.api.Test;

```

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

class OrderSystemTest {

    @Test
    void processOrder_withBlackFridayAndPaypal_marksPaid() {
        Order order = new Order();
        order.addItem(new OrderItem("Headphones", 1, 100.0));

        OrderService service = new OrderService();
        double total = service.processOrder(order, "BLACKFRIDAY", "paypal");

        assertEquals(OrderStatus.PAID, order.getStatus());
        assertEquals(84.0, total, 1e-9);
    }

    @Test
    void processOrder_withBlankDiscountCode_chargesFullPricePlusTax() {
        Order order = new Order();
        order.addItem(new OrderItem("Mouse", 2, 25.0));

        OrderService service = new OrderService();
        double total = service.processOrder(order, "   ", "card");

        assertEquals(OrderStatus.PAID, order.getStatus());
        assertEquals(60.0, total, 1e-9);
    }

    @Test
    void processOrder_withCrypto_cancelsOrder() {
        Order order = new Order();
        order.addItem(new OrderItem("Cable", 1, 10.0));

        OrderService service = new OrderService();
        double total = service.processOrder(order, "STUDENT10", "crypto");

        assertEquals(OrderStatus.CANCELLED, order.getStatus());
        assertEquals(0.0, total, 1e-9);
    }

    @Test
```

```java
    void processOrder_withUnknownPayment_throwsAndKeepsCreatedStatus() {
        Order order = new Order();
        order.addItem(new OrderItem("Keyboard", 1, 80.0));

        OrderService service = new OrderService();

        assertThrows(UnsupportedOperationException.class,
                () -> service.processOrder(order, "STUDENT10", "bank-transfer"));
        assertEquals(OrderStatus.CREATED, order.getStatus());
    }

    @Test
    void discountService_coversAllBranches() {
        DiscountService discountService = new DiscountService();

        assertEquals(100.0, discountService.applyDiscount(100.0, null), 1e-9);
        assertEquals(100.0, discountService.applyDiscount(100.0, "   "), 1e-9);
        assertEquals(90.0, discountService.applyDiscount(100.0, "student10"), 1e-
9);
        assertEquals(70.0, discountService.applyDiscount(100.0, "BLACKFRIDAY"), 1e-
9);
        assertEquals(100.0, discountService.applyDiscount(100.0, "VIP"), 1e-9);
        assertThrows(IllegalArgumentException.class,
                () -> discountService.applyDiscount(100.0, "INVALID"));
    }

    @Test
    void paymentValidator_coversAllBranches() {
        PaymentValidator validator = new PaymentValidator();

        assertFalse(validator.isPaymentMethodValid(null));
        assertTrue(validator.isPaymentMethodValid("card"));
        assertTrue(validator.isPaymentMethodValid("paypal"));
        assertFalse(validator.isPaymentMethodValid("crypto"));
        assertThrows(UnsupportedOperationException.class,
                () -> validator.isPaymentMethodValid("wire"));
    }

    @Test
    void pricingService_coversSubtotalAndTaxBranches() {
        PricingService pricingService = new PricingService();
        Order order = new Order();
        order.addItem(new OrderItem("A", 1, 10.0));
        order.addItem(new OrderItem("B", 2, 15.0));
```

```java
        assertEquals(40.0, pricingService.calculateSubtotal(order), 1e-9);
        assertEquals(0.0, pricingService.calculateTax(0.0), 1e-9);
        assertEquals(8.0, pricingService.calculateTax(40.0), 1e-9);
        assertThrows(IllegalArgumentException.class, () -> pricingService.calculate
Tax(-1.0));
    }

    @Test
    void orderItem_validAndInvalidInputs() {
        OrderItem item = new OrderItem("Phone", 2, 99.5);

        assertEquals(199.0, item.getTotalPrice(), 1e-9);
        assertEquals(2, item.getQuantity());
        assertThrows(IllegalArgumentException.class, () -> new OrderItem("BadQ", 0,
 10.0));
        assertThrows(IllegalArgumentException.class, () -> new OrderItem("BadP", 1,
 -0.1));
    }

    @Test
    void order_disallowsAddAfterProcessed() {
        Order order = new Order();
        order.setStatus(OrderStatus.PAID);

        assertThrows(IllegalStateException.class, () -> order.addItem(new OrderItem
("X", 1, 1.0)));
    }
}
```

c)

Our key takeaway from this exercise is we cannot blindly trust AI. It is true that AI write the code very quickly, but the test it writes are sometimes just to make the code work rather than to truly understand the business logic.

We think gen AI is useful for build a basic testing framework; it can help us write all repetitive setup code. Also, AI is very helpful in ensuring the correctness of code syntax, which can let someone who don't know the test code syntax familiar it. For example, AI can write JUnit syntax very clearly and quickly.

We think human cognition is useful for ensure the tests actually make sense for the real-world business rules. But the AI often writes tests just to hit the coverage number.

d)

If I had to invest $1M into developing such tools, I would make the AI tool have a better integration with the project. If we can make the AI tool can read the whole project files, then it understands everything in that project. Then the AI can give the codes based on the project, can modify the files directly, and can also just execute the code directly. So, in this way, instead of copying and pasting the answer AI gives us, it can just write the code in the places, and it can figure out the execution of the files and run the code by itself. The second feature I want is the ability to read the testing report and understand what test cases are left out. Then the AI can automatically write the new testing code.

The biggest challenges of widespread adoption are the trust in AI. If the development team doesn't trust the AI tool or doesn't trust the output from AI, then people might not want to use this new tool. If the AI can’t explain the code itself, why does this code work? Why is this code better than the human code? Then people don’t trust the AI coding. The second thing is privacy, most companies don’t want to leak the code to third party platform. Because the code is the company's asset, giving it to third party models might harm the business. Another thing is the long-term maintenance. If the AI code doesn't align with the existing code style, then it might cost more for future refactoring.