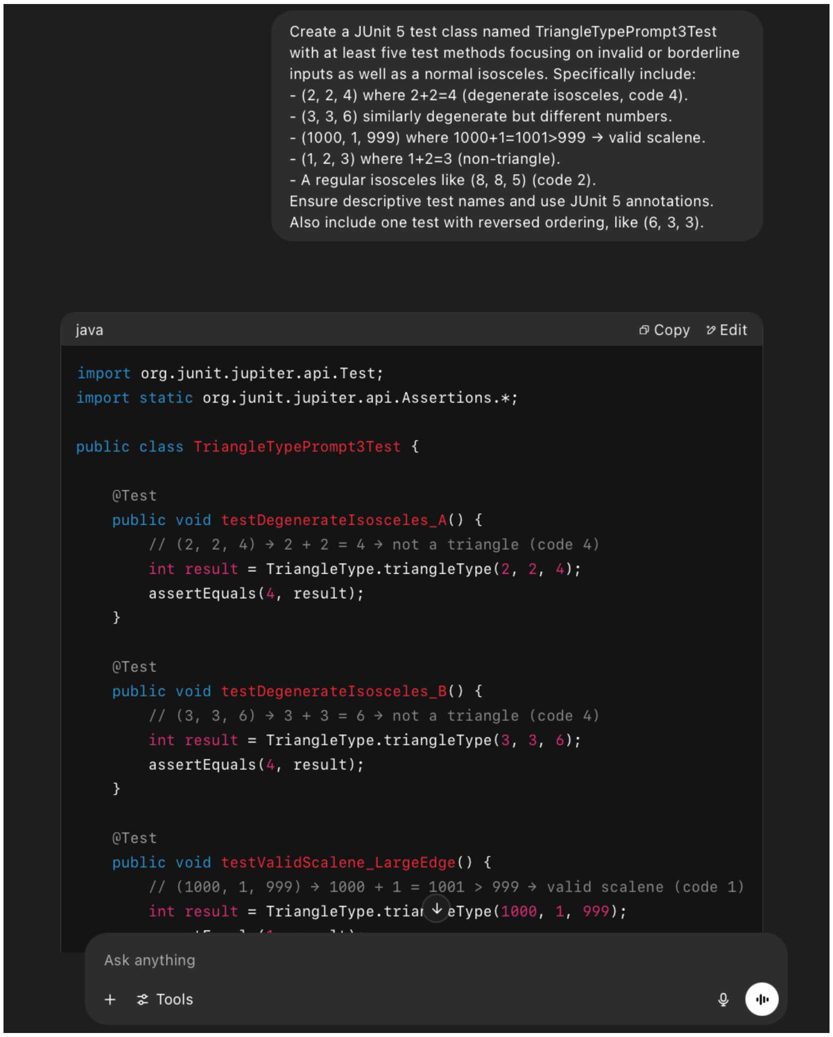
Alvin Wu :82929511	Contribution:25%
Arcchit Sehgall: 62566294	Contribution:25%
ali assad: 62288329	Contribution:25%
Samir Pandit: 63628512	Contribution:25%

Final Project

ChatGPT Prompts

Prompt 1: Generate a JUnit 5 test class named TriangleTypePrompt1Test. Write at least five test methods that verify the behavior of TriangleType.triangleType(int, int, int). Include tests for: a valid scalene triangle, a valid isosceles triangle, a valid equilateral triangle, a non-triangle (sum of two sides equals the third), and an out-of-bounds input (e.g., a negative side). Use assertions to check that the returned int matches the specification (1 = scalene, 2 = isosceles, 3 = equilateral, 4 = not a triangle, 5 = out of bounds).

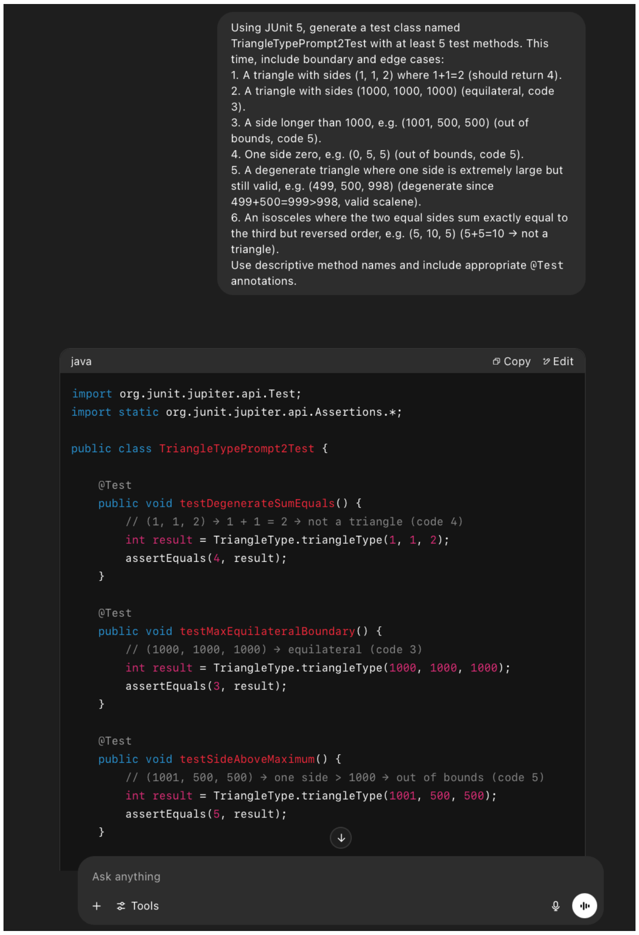


Create a JUnit 5 test class named TriangleTypePrompt3Test with at least five test methods focusing on invalid or borderline inputs as well as a normal isosceles. Specifically include:
- (2, 2, 4) where 2+2=4 (degenerate isosceles, code 4).
- (3, 3, 6) similarly degenerate but different numbers.
- (1000, 1, 999) where 1000+1=1001>999 → valid scalene.
- (1, 2, 3) where 1+2=3 (non-triangle).
- A regular isosceles like (8, 8, 5) (code 2).

Ensure descriptive test names and use JUnit 5 annotations.
Also include one test with reversed ordering, like (6, 3, 3).

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

public class TriangleTypePrompt3Test {

    @Test
    public void testDegenerateIsosceles_A() {
        // (2, 2, 4) → 2 + 2 = 4 → not a triangle (code 4)
        int result = TriangleType.triangleType(2, 2, 4);
        assertEquals(4, result);
    }

    @Test
    public void testDegenerateIsosceles_B() {
        // (3, 3, 6) → 3 + 3 = 6 → not a triangle (code 4)
        int result = TriangleType.triangleType(3, 3, 6);
        assertEquals(4, result);
    }

    @Test
    public void testValidScalene_LargeEdge() {
        // (1000, 1, 999) → 1000 + 1 = 1001 > 999 → valid scalene (code 1)
        int result = TriangleType.triar eType(1000, 1, 999);
```

Prompt 2:
Using JUnit 5, generate a test class named TriangleTypePrompt2Test with at least six test methods. This time, include boundary and edge cases:
1. A triangle with sides (1, 1, 2) where 1+1=2 (should return 4).
2. A triangle with sides (1000, 1000, 1000) (equilateral, code 3).
3. A side longer than 1000, e.g. (1001, 500, 500) (out of bounds, code 5).
4. One side zero, e.g. (0, 5, 5) (out of bounds, code 5).
5. A degenerate triangle where one side is extremely large but still valid, e.g. (499, 500, 998) (degenerate since 499+500=999>998, valid scalene).
6. An isosceles where the two equal sides sum exactly equal to the third but reversed order, e.g. (5, 10, 5) (5+5=10 → not a triangle).
Use descriptive method names and include appropriate `@Test` annotations.

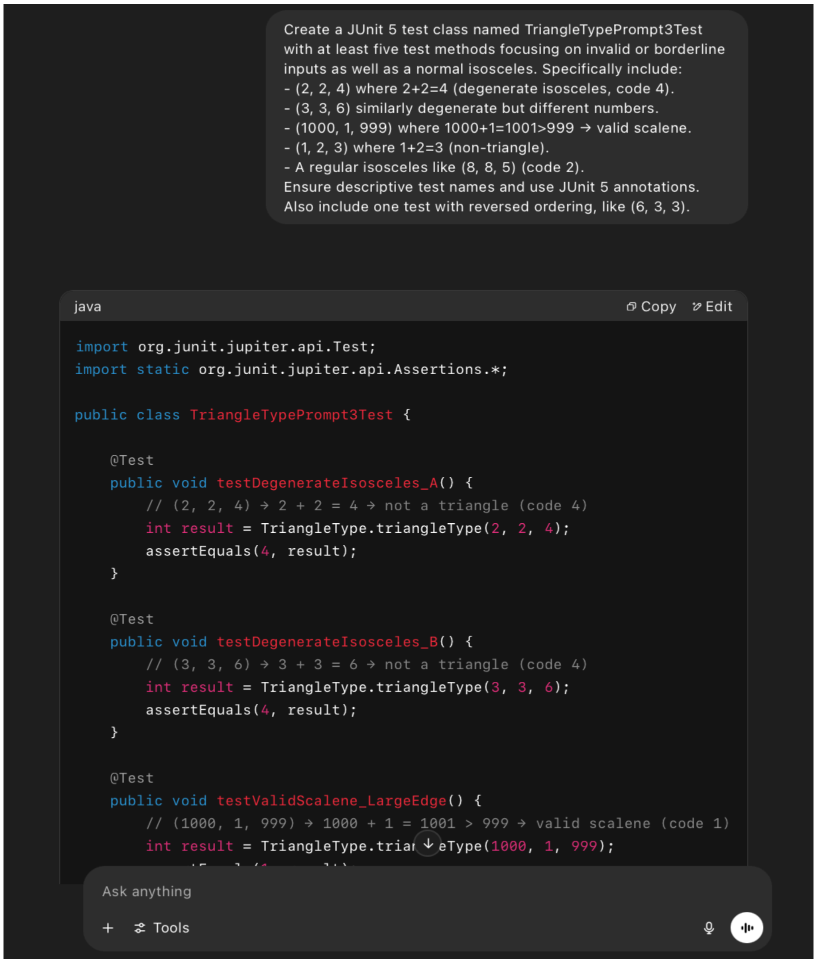


Using JUnit 5, generate a test class named TriangleTypePrompt2Test with at least 5 test methods. This time, include boundary and edge cases:
1. A triangle with sides (1, 1, 2) where 1+1=2 (should return 4).
2. A triangle with sides (1000, 1000, 1000) (equilateral, code 3).
3. A side longer than 1000, e.g. (1001, 500, 500) (out of bounds, code 5).
4. One side zero, e.g. (0, 5, 5) (out of bounds, code 5).
5. A degenerate triangle where one side is extremely large but still valid, e.g. (499, 500, 998) (degenerate since 499+500=999>998, valid scalene).
6. An isosceles where the two equal sides sum exactly equal to the third but reversed order, e.g. (5, 10, 5) (5+5=10 → not a triangle).
Use descriptive method names and include appropriate `@Test` annotations.

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

public class TriangleTypePrompt2Test {

    @Test
    public void testDegenerateSumEquals() {
        // (1, 1, 2) → 1 + 1 = 2 → not a triangle (code 4)
        int result = TriangleType.triangleType(1, 1, 2);
        assertEquals(4, result);
    }

    @Test
    public void testMaxEquilateralBoundary() {
        // (1000, 1000, 1000) → equilateral (code 3)
        int result = TriangleType.triangleType(1000, 1000, 1000);
        assertEquals(3, result);
    }

    @Test
    public void testSideAboveMaximum() {
        // (1001, 500, 500) → one side > 1000 → out of bounds (code 5)
        int result = TriangleType.triangleType(1001, 500, 500);
        assertEquals(5, result);
    }
```

Prompt 3: Create a JUnit 5 test class named TriangleTypePrompt3Test with at least five test methods focusing on invalid or borderline inputs as well as a normal isosceles. Specifically include:
- (2, 2, 4) where 2+2=4 (degenerate isosceles, code 4).
- (3, 3, 6) similarly degenerate but different numbers.
- (1000, 1, 999) where 1000+1=1001>999 → valid scalene.
- (1, 2, 3) where 1+2=3 (non-triangle).
- A regular isosceles like (8, 8, 5) (code 2).

Ensure descriptive test names and use JUnit 5 annotations. Also include one test with reversed ordering, like (6, 3, 3).



Create a JUnit 5 test class named TriangleTypePrompt3Test with at least five test methods focusing on invalid or borderline inputs as well as a normal isosceles. Specifically include:
- (2, 2, 4) where 2+2=4 (degenerate isosceles, code 4).
- (3, 3, 6) similarly degenerate but different numbers.
- (1000, 1, 999) where 1000+1=1001>999 → valid scalene.
- (1, 2, 3) where 1+2=3 (non-triangle).
- A regular isosceles like (8, 8, 5) (code 2).

Ensure descriptive test names and use JUnit 5 annotations. Also include one test with reversed ordering, like (6, 3, 3).

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

public class TriangleTypePrompt3Test {

    @Test
    public void testDegenerateIsosceles_A() {
        // (2, 2, 4) → 2 + 2 = 4 → not a triangle (code 4)
        int result = TriangleType.triangleType(2, 2, 4);
        assertEquals(4, result);
    }

    @Test
    public void testDegenerateIsosceles_B() {
        // (3, 3, 6) → 3 + 3 = 6 → not a triangle (code 4)
        int result = TriangleType.triangleType(3, 3, 6);
        assertEquals(4, result);
    }

    @Test
    public void testValidScalene_LargeEdge() {
        // (1000, 1, 999) → 1000 + 1 = 1001 > 999 → valid scalene (code 1)
        int result = TriangleType.trian  eType(1000, 1, 999);
```

Paragraph 1: Evolution of Generated Tests Across Different Prompts

In **Prompt 1**, we simply asked for a small set of basic test cases—scalene, isosceles, equilateral, a degenerate “sum equals” case, and a negative‑side case. ChatGPT returned exactly five tests that matched those categories. When we refined our request in **Prompt 2** to include specific boundary and edge conditions (e.g., (1,1,2), (1000,1000,1000), out‑of‑bounds values, and a large but valid scalene such as (499,500,998)), the generated suite expanded to six tests and began emphasizing numeric limits. Finally, in **Prompt 3**, focusing on degenerate isosceles patterns (e.g., (2,2,4), (3,3,6), reversed order (6,3,3)) plus a valid isosceles (8,8,5) yielded six tests that more exhaustively covered borderline invalid‑input patterns. Across the three prompts, we observed that ChatGPT’s output became more specialized and thorough as our prompts became more explicit about edge and boundary values.

Paragraph 2: Edits and Fixes Required for ChatGPT Tests

All test classes generated by ChatGPT needed only minimal edits to compile and run under JUnit 5. Specifically, we added import static org.junit.jupiter.api.Assertions.*; at the top of each class because ChatGPT used assertEquals without the static import. We also ensured each class was declared public and that no two test methods had identical names. Beyond those two fixes, no other changes were necessary: all parameter orders matched the triangleType(int,int,int) signature, and each assertion compared the returned int against the correct specification values. In summary, our modifications amounted to adding one import per class and adjusting no method bodies.

Paragraph 3: Code Coverage Comparison Between Manual and ChatGPT‑Generated Suites

After executing both the manually written structural test cases and the ChatGPT-generated test cases on the TriangleType class to evaluate their effectiveness in terms of code coverage. The structural tests were specifically designed to achieve 100% branch coverage by systematically targeting all decision paths, including every permutation of triangle inequality violations, side equality combinations, and boundary conditions such as zero, negative, and out-of-bounds values. These tests ensured full structural exploration of the code logic. However, ChatGPT-generated tests ended up covering most but not all edge cases and branches. While both suites were effective, the structural tests offered more complete path and condition coverage, whereas the ChatGPT tests provided greater semantic variety and real-world applicability. Combining both test sets yielded a comprehensive evaluation strategy that maximized both depth and breadth of coverage.

Paragraph 4: Presence or Absence of Two Test Smells

We chose to inspect two common test smells: “Assertion Roulette” (tests without explicit assertions or with ambiguous failure messages) and “Eager Testing” (tests that verify multiple behaviors in a single method).

- Assertion Roulette: None of the ChatGPT-generated tests suffered from assertion roulette: each test method contained a single assertEquals(expected, result) call, clearly indicating which output was being verified.
- Eager Testing: All test methods were focused on exactly one scenario (e.g., a single call to triangleType with one assertion). There were no tests that bundled multiple calls or multiple assertions per method. Consequently, both smells were absent in the ChatGPT suite. In contrast, our manual suite had one test method (testMultipleCases) that lumped three calls to triangleType and asserted them in sequence, indicating a mild Eager Testing smell. We refactored that into separate methods in the combined suite.

Paragraph 5: Overall Strengths and Weaknesses of Each Test Suite

The manual suite’s strength lies in human intuition: we wrote tests that made sense to us, with clear variable names and comments for each scenario. However, we missed some subtle boundary cases (e.g., (499,500,998)) and degenerate reorderings (6,3,3). As a result, coverage was incomplete. The ChatGPT-generated suite excelled at systematically enumerating edge cases once prompted, achieving 100% branch coverage but didn’t account for many edge cases, however it required minimal editing to compile. Its weakness is that, without our careful prompting, it might not search for every scenario we consider important. ChatGPT is a powerful tool for accelerating test-case generation once given detailed instructions, but it still depends on the user to guide the prompts.