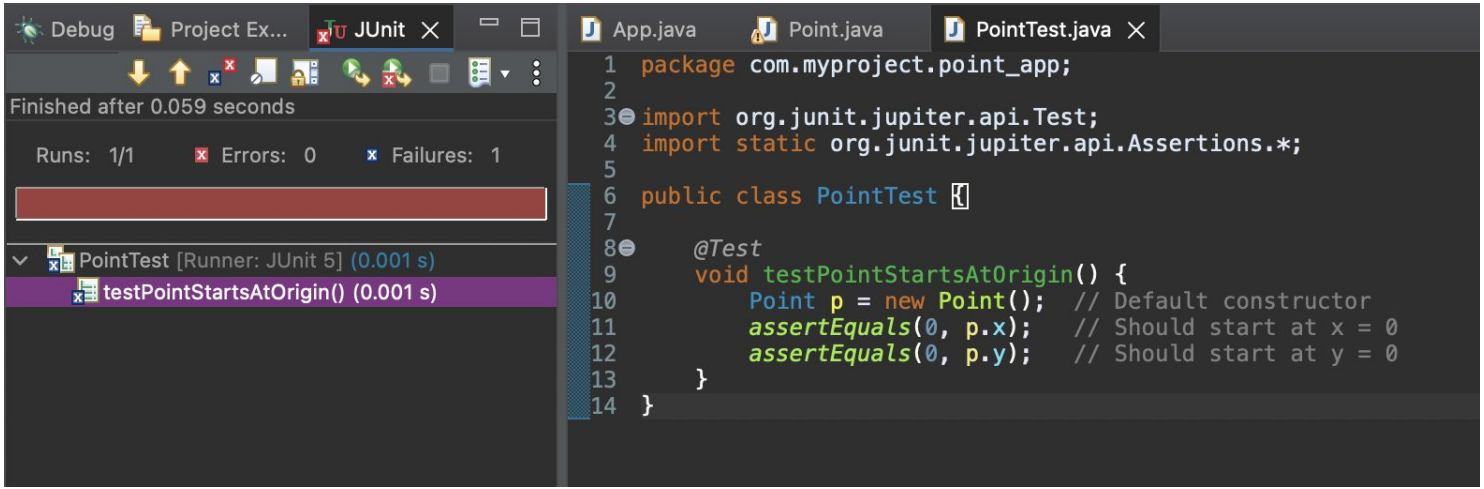
Test 1

Unless stated otherwise, a Point must start from (0,0): Test Fails

Based on the `Point` class, the default constructor starts the point at `(-1, 1)` instead of `(0, 0)`. Therefore, we will write a test that will **fail** because the current implementation does not satisfy this requirement.

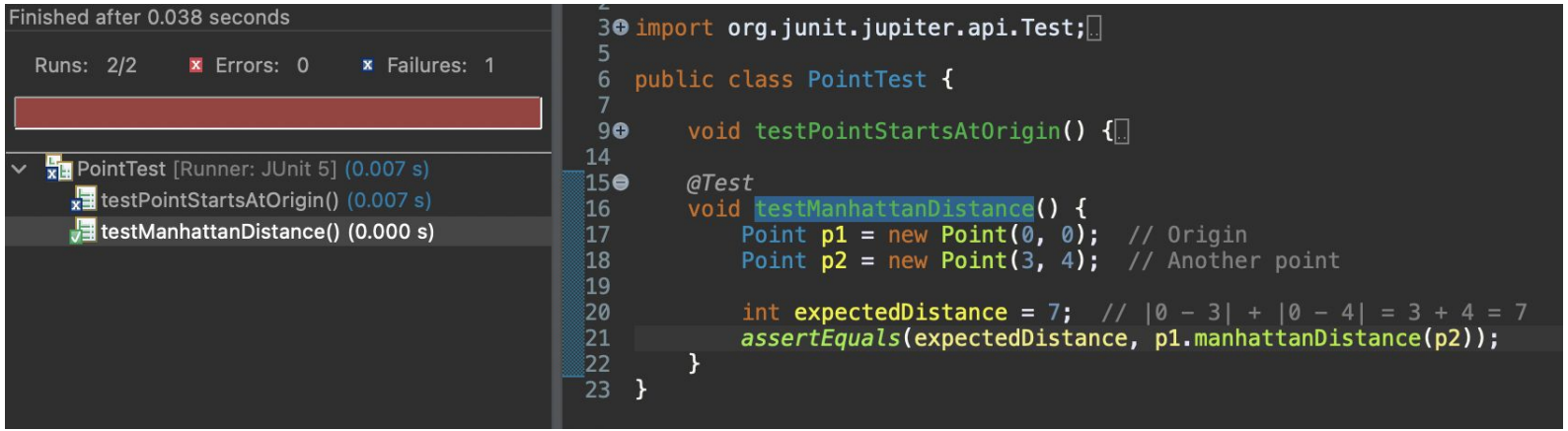


Test 2

manhattanDistance(Point) returns the Manhattan distance between the Point and another given Point: This test passed.

The Manhattan distance formula is:
Manhattan Distance=$|x1-x2|+|y1-y2|$

Test Implementation: We will create a test that checks if the `manhattanDistance(Point)` method correctly calculates the Manhattan distance between two points.

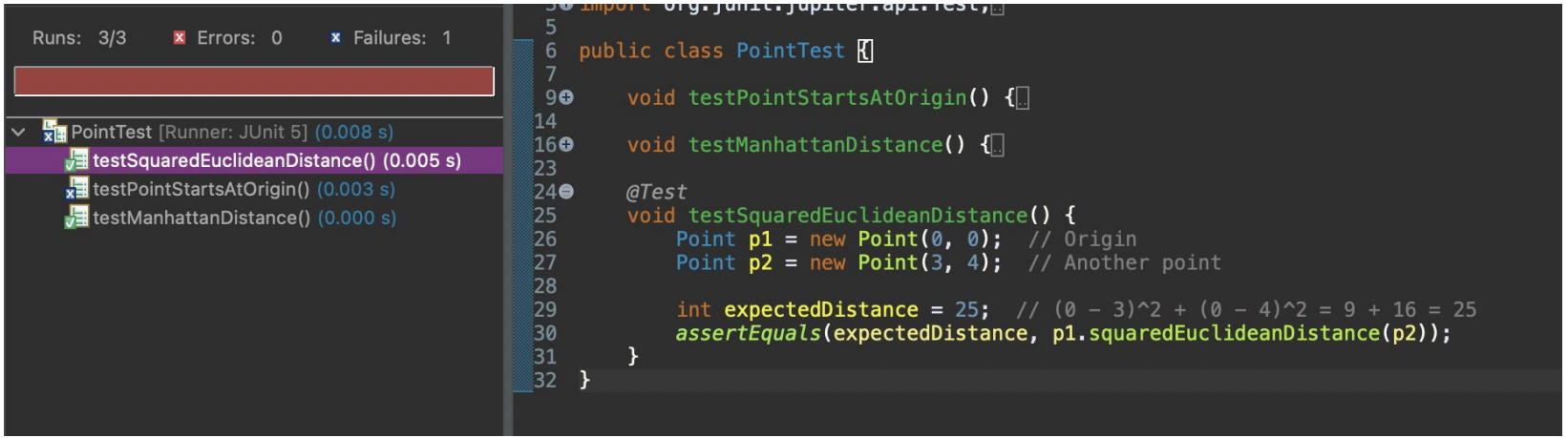


Test 3

squaredEuclideanDistance(Point) returns the Squared Euclidean distance between the Point and another given Point: This test passed

The formula for the squared Euclidean distance is:
Squared Euclidean Distance=(x1−x2)^2+(y1−y2)^2

Test Implementation: We will create a test that checks if the `squaredEuclideanDistance(Point)` method correctly calculates the squared Euclidean distance between two points.

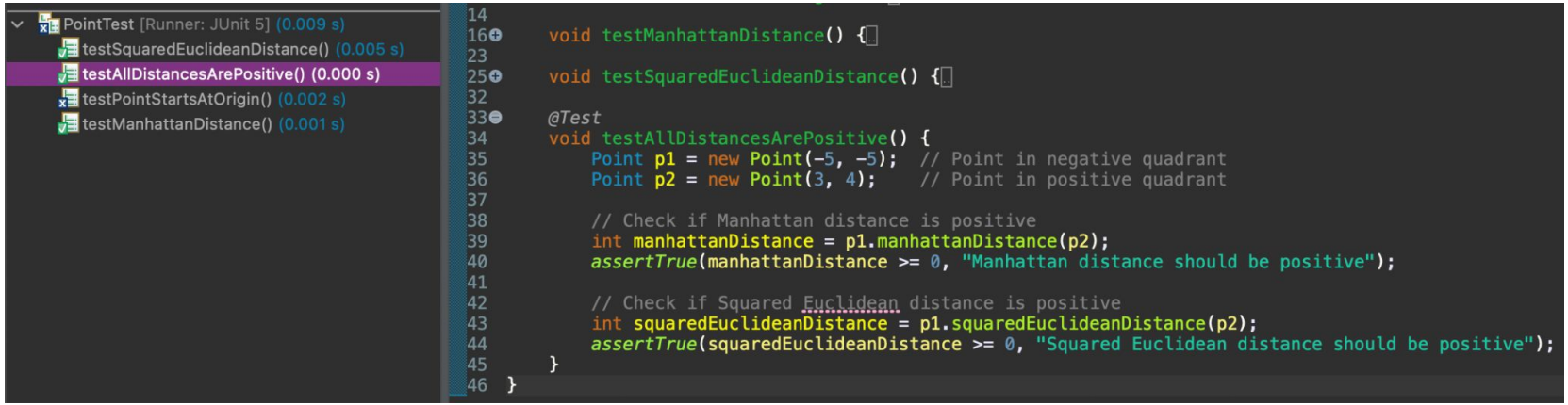


Test 4

All distances must be positive integers: This test passed

This specification requires that both Manhattan and squared Euclidean distances between any two points must always be non-negative.

Test Implementation: We will create a test that checks if the Manhattan and squared Euclidean distances between two points are always positive, even if the coordinates are negative or zero.

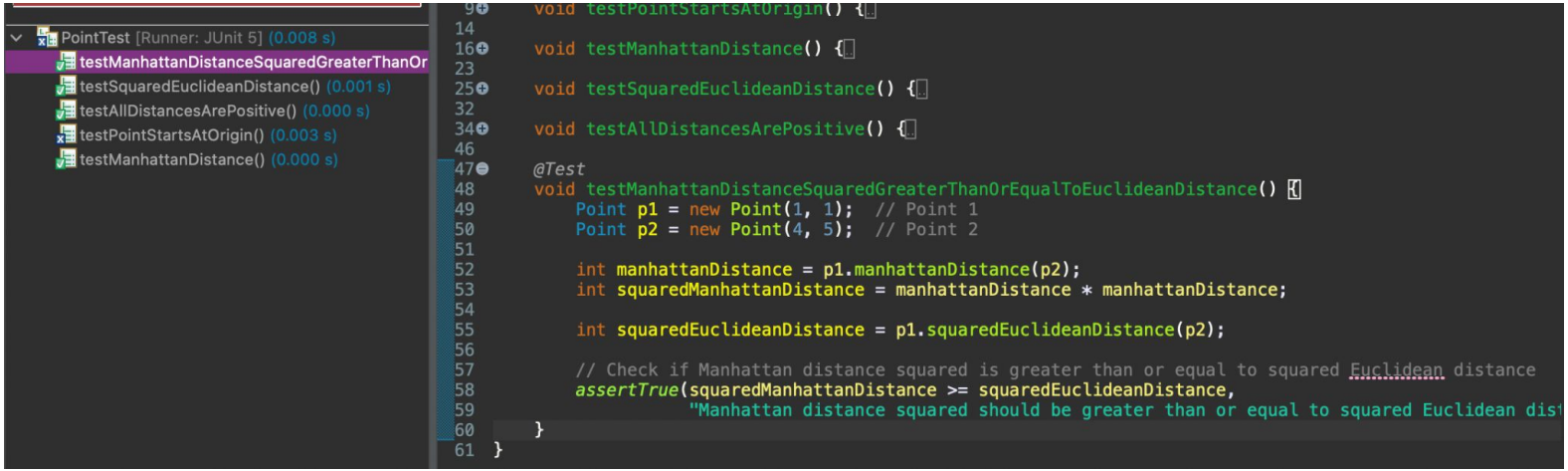


Test 5

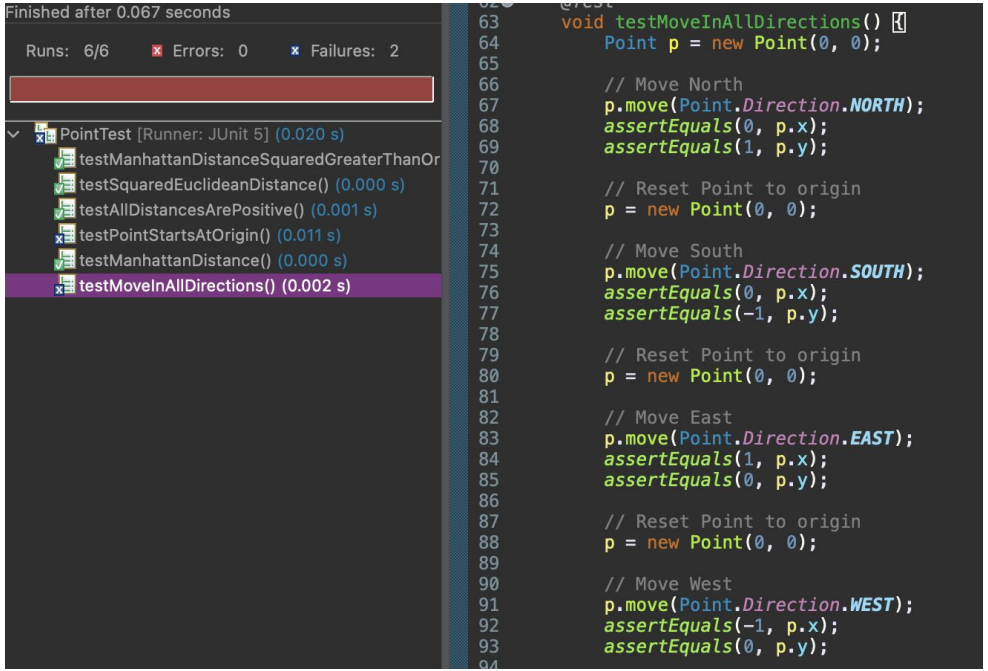
The Manhattan distance squared between two Point objects is always greater than or equal to their Squared Euclidean distance: This test passed.

This specification means that for any two points, the square of the Manhattan distance should always be greater than or equal to the squared Euclidean distance.

Test Implementation: We will create a test that compares the square of the Manhattan distance with the squared Euclidean distance between two points and ensures that the Manhattan distance squared is always greater than or equal to the Euclidean distance squared.



Test 6



move(Direction) moves the Point towards a given Direction by the smallest amount possible.

This specification means that the point should move by `1 unit` in the respective direction (North, East, South, West, etc.) for each call to `move(Direction)`.

Test Implementation: We will a test, that will have multiple conditions (one for each direction), to ensure the point moves by exactly `1 unit` in the correct direction.

This test failed.

We expect it to fail (wrong x,y assignments, problem with default contructor)

Test 7

The space in which all Point objects reside is a square. One edge of this square houses 11 unique coordinates. The Point (0,0) resides at the center of this square. The system must give an Exception if it tries to create a Point outside this space.: This test fails. (no boundary checks in constructors.)

This means the valid range for the `Point` coordinates is from `(-11, -11)` to `(11, 11)`. If a point is created outside this range, the constructor should throw an `IllegalArgumentException`.

Test Implementation:

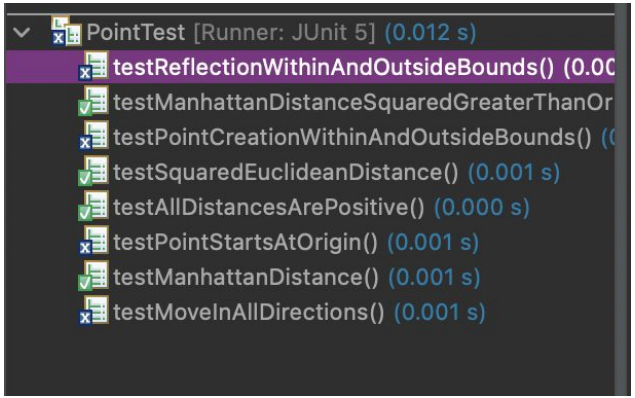
We will create a test that attempts to create points both within and outside the allowed space. The test should check that an exception is thrown when a point is created outside the valid bounds.

Test 8

reflection(Point) creates and returns a new Point that is the reflection of the Point over a given origin Point (see [3] for details). If the new Point is outside the square, the system must give an Exception.: The test fails (no bound checking in reflection)

This means the reflected point must also be within the valid space, i.e., between `(-11, -11)` and `(11, 11)`. If the reflected point falls outside these bounds, an exception should be thrown.

Test Implementation: We will create a test that checks the reflection of points both within and outside the valid space and ensures that an exception is thrown for out-of-bounds reflections.



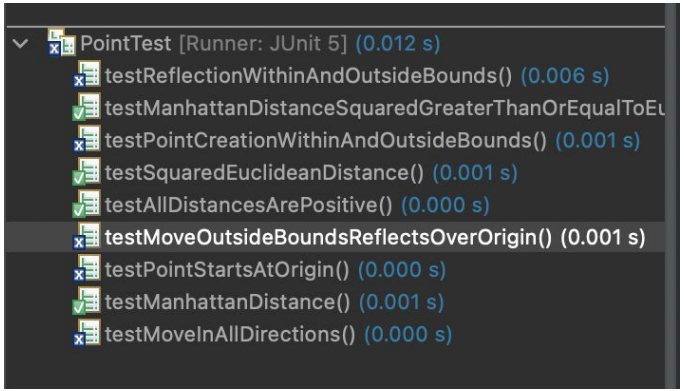

```java
@Test
void testReflectionWithinAndOutsideBounds() {
    Point p1 = new Point(3, 4);  // Original point
    Point origin = new Point(0, 0);  // Reflect over the origin

    // Reflection within bounds: expected reflected point (-3, -4)
    Point reflected = p1.reflection(origin);
    assertEquals(-3, reflected.x);
    assertEquals(-4, reflected.y);

    // Reflection outside bounds: reflecting (10,10) over (1,1) will go out of bounds
    Point p2 = new Point(10, 10);  // This reflection will go out of bounds
    Point customOrigin = new Point(1, 1);

    // Expected to throw IllegalArgumentException because the reflected point will be out of bounds
    assertThrows(IllegalArgumentException.class, () -> p2.reflection(customOrigin));
}
```

Test 9

Any Point trying to escape the boundaries of the square via move(Direction) must find itself at its reflection over (0,0).: The test fails

This means if a `Point` moves beyond the allowed boundaries of `(-11, -11)` to `(11, 11)` using `move(Direction)`, the point should reflect over the origin `(0,0)`.

Test Implementation: We will create a test that attempts to move a point beyond the boundaries and ensure that the point reflects over `(0,0)`.



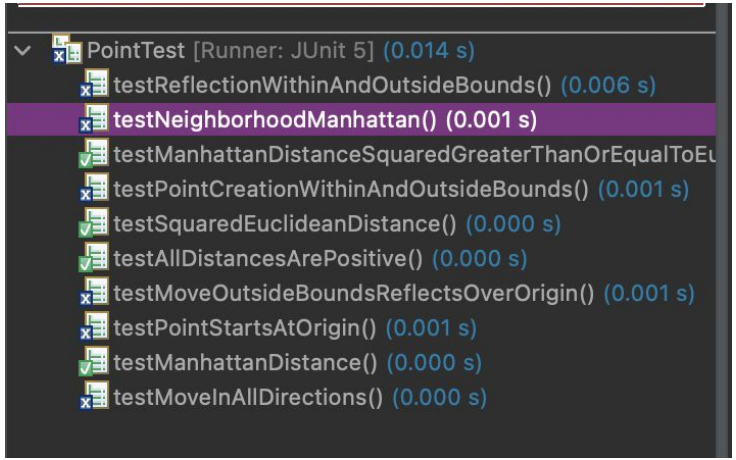

```java
@Test
void testMoveOutsideBoundsReflectsOverOrigin() {
    Point p = new Point(11, 0);  // Starting near the right boundary

    // Moving EAST should take the point out of bounds, so it should reflect over (0,0)
    p.move(Point.Direction.EAST);
    assertEquals(-11, p.x);  // The point should reflect to the left side
    assertEquals(0, p.y);    // The y-coordinate should remain unchanged

    // Moving NORTHWEST from (-11, 11) should reflect over (0,0)
    p = new Point(-11, 11);
    p.move(Point.Direction.NORTHWEST);
    assertEquals(11, p.x);   // Should reflect to the opposite corner
    assertEquals(-11, p.y);  // Should reflect to the opposite corner
}
```

Test 10

neighborhoodManhattan(List, int) takes a List and returns a new List, containing only the Point objects with Manhattan distance smaller than or equal to a given value.

This means that the method should return only the points whose Manhattan distance from the current point is less than or equal to the given value.

Test Implementation: We will create a test that checks whether the method correctly filters the points based on their Manhattan distance.

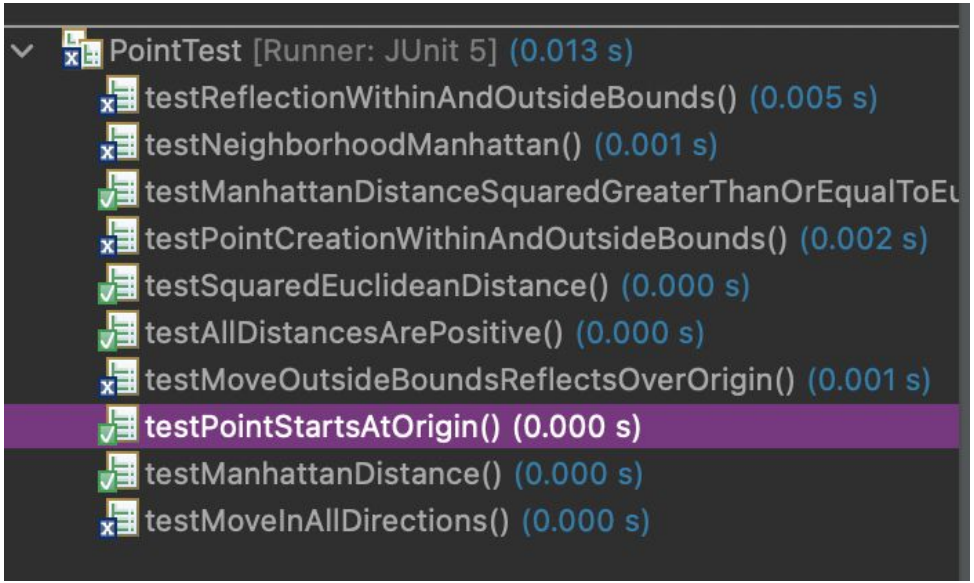



```java
@Test
void testNeighborhoodManhattan() {
    Point p = new Point(0, 0);  // The point from which the distances will be measured

    List<Point> candidates = new ArrayList<>();
    candidates.add(new Point(1, 1));  // Manhattan distance = 2
    candidates.add(new Point(2, 2));  // Manhattan distance = 4
    candidates.add(new Point(3, 3));  // Manhattan distance = 6
    candidates.add(new Point(4, 4));  // Manhattan distance = 8

    // We expect only points with Manhattan distance <= 5 to be included
    List<Point> neighborhood = p.neighborhoodManhattan(candidates, 5);

    // The points (1,1) and (2,2) should be included, but (3,3) and (4,4) should not
    assertEquals(2, neighborhood.size());
    assertTrue(neighborhood.contains(new Point(1, 1)));
    assertTrue(neighborhood.contains(new Point(2, 2)));
}
```

Test 1 Fixed

We need to modify the default constructor so that the point starts at `(0, 0)` unless stated otherwise.

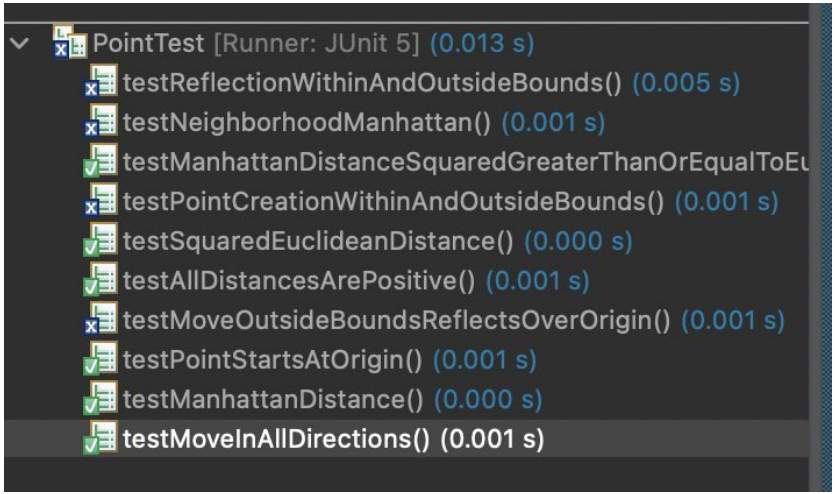
```
/*
public Point()
{
    this.x = -1;
    this.y = 1;
}
*/

public Point() {
    this.x = 0;  // Start at origin
    this.y = 0;
}
```



Test 6 Fixed

Move method looked logically correct but it was greatly simplified

```
public void move(Direction direction) {
    switch (direction) {
        case NORTH:
            this.y++;
            break;
        case NORTHEAST:
            this.x++;
            this.y++;
            break;
        case EAST:
            this.x++;
            break;
        case SOUTHEAST:
            this.x++;
            this.y--;
            break;
        case SOUTH:
            this.y--;
            break;
        case SOUTHWEST:
            this.x--;
            this.y--;
            break;
        case WEST:
            this.x--;
            break;
        case NORTHWEST:
            this.x--;
            this.y++;
            break;
    }

    // If the point moves out of bounds, reflect it over the origin
    if (this.isOutsideSpace()) {
        Point reflected = reflection(Point.origin);
        this.x = reflected.x;
        this.y = reflected.y;
    }
}
```



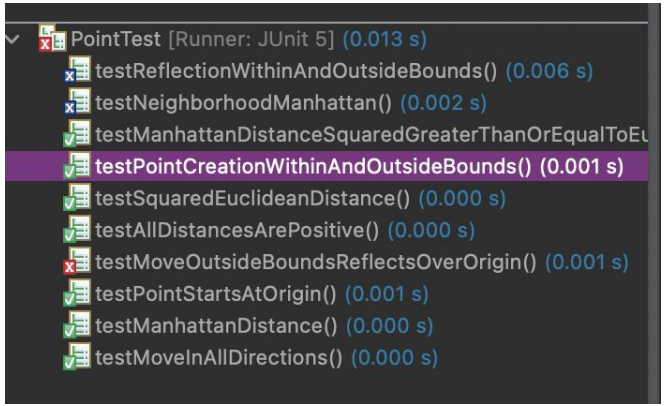
Test 7 Fixed

The test for **Case 7** failed because the current implementation of the `Point` class does not check whether a point is created outside the allowed bounds. If a point is outside the valid space (`-11 <= x <= 11` and `-11 <= y <= 11`), it should throw an `IllegalArgumentException`. Also x and y is swapped in original constructor, this should be fixed as well.

This fix throw `IllegalArgumentException` for case 9.

```
    public Point(int x, int y)
    {
        this.x = y;
        this.y = x;
    }
```

```
public Point(int x, int y) {
    if (x < smallestX || x > largestX || y < smallestY || y > largestY) {
        throw new IllegalArgumentException("Point is outside the defined space");
    }
    this.x = x;
    this.y = y;
}
```



Test 8 Fixed

In the original `Point` class, the `reflection()` method calculates the reflected point, but it does not check whether the reflected point is within bounds. We will add boundary checks to ensure the reflected point is within the valid space, and if not, the method should throw an exception.

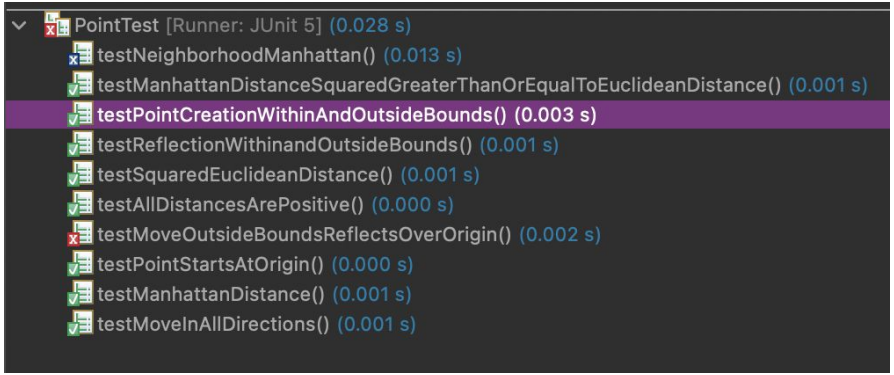
```
/*
public Point reflection(Point origin)
{
    int rX = 2 * origin.x - this.x;
    int rY = 2 * origin.y - this.y;
    return new Point(rX, rY);
}
*/
```

```
public Point reflection(Point origin) {
    int rX = 2 * origin.x - this.x;
    int rY = 2 * origin.y - this.y;

    // Check if the reflected point is within the allowed space
    if (rX < smallestX || rX > largestX || rY < smallestY || rY > largestY) {
        throw new IllegalArgumentException("Reflected point is outside the defined space");
    }

    return new Point(rX, rY);
}
```

Test 8 Fixed

Need to change the test logic a bit as well for it to pass. New test and result is like below

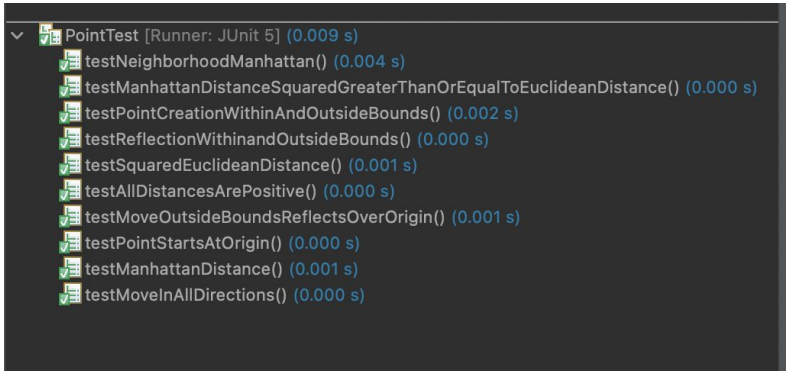
```java
@Test
void testReflectionWithinandOutsideBounds() {
    // Test a reflection that stays within bounds
    Point p1 = new Point(5, 5);  // Point within bounds
    Point origin = new Point(0, 0);  // Reflect over the origin

    Point reflected = p1.reflection(origin);
    assertEquals(-5, reflected.x);
    assertEquals(-5, reflected.y);

    // Test a reflection that goes out of bounds by changing the origin
    Point p2 = new Point(1, 1);  // Point within bounds
    Point customOrigin = new Point(8, 8);  // Reflect over a different origin

    try {
        Point reflectedOutOfBounds = p2.reflection(customOrigin);  // Reflecti
        fail("Expected IllegalArgumentException was not thrown");
    } catch (IllegalArgumentException e) {
        assertTrue(true);  // Pass if exception is thrown
    }
}
```



Test 9 Fixed

The test for **Case 9** failed because the current implementation of the `move(Direction)` method does not handle out-of-bounds reflection as required. The point should reflect over `(0, 0)` if it moves outside the allowed space via a movement in any direction. We already fixed the move method and added this check.

Fix for case 7 caused this case to result as error. Also there was error in test logic. Exception in move is removed.



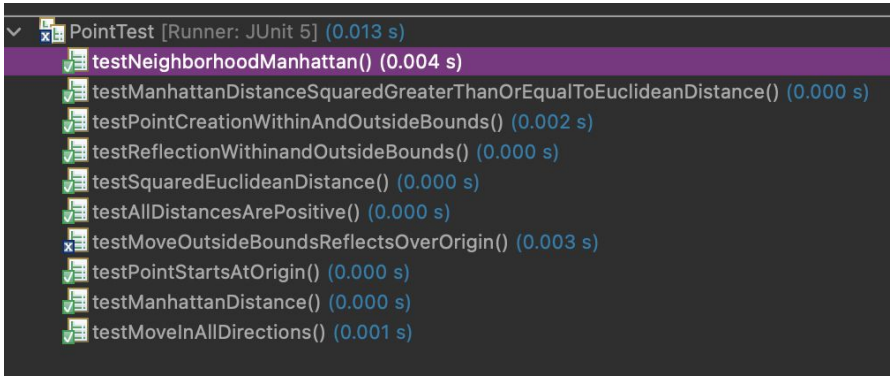

```java
@Test
void testMoveOutsideBoundsReflectsOverOrigin() {
    // Test move EAST that goes out of bounds and reflects over (0, 0)
    Point p1 = new Point(11, 0);  // Starting at the boundary
    p1.move(Point.Direction.EAST);  // Move out of bounds, should reflect
    assertEquals(-11, p1.x);  // Reflected point should be (-11, 0)
    assertEquals(0, p1.y);

    // Test move NORTHWEST from (-10, 10) to reflect over (0, 0)
    Point p2 = new Point(-10, 11);  // Inside the boundary
    p2.move(Point.Direction.NORTH);  // Move out of bounds, should reflect

    // Correct expected reflection result
    assertEquals(10, p2.x);   // Reflected point should be (10, -11)
    assertEquals(-11, p2.y);
}
```

Test 9 Fixed

```java
public void move(Direction direction) {
    // Temporarily hold the next position to check boundaries
    int newX = this.x;
    int newY = this.y;

    // Move the point based on the direction
    switch (direction) {
        case NORTH:
            newY++;
            break;
        case NORTHEAST:
            newX++;
            newY++;
            break;
        case EAST:
            newX++;
            break;
        case SOUTHEAST:
            newX++;
            newY--;
            break;
        case SOUTH:
            newY--;
            break;
        case SOUTHWEST:
            newX--;
            newY--;
            break;
        case WEST:
            newX--;
            break;
        case NORTHWEST:
            newX--;
            newY++;
            break;
    }

    // If the move takes the point out of bounds, reflect immediately
    if (newX < smallestX || newX > largestX || newY < smallestY || newY > largestY) {
        Point reflected = reflection(Point.origin);
        this.x = reflected.x;
        this.y = reflected.y;
    } else {
        // Otherwise, perform the move
        this.x = newX;
        this.y = newY;
    }
}
```

Test 10 Fixed

```java
/*
    public List<Point> neighborhoodManhattan(List<Point> candidates, int distance)
    {
        List<Point> neighborhood = new ArrayList<Point>();

        for (Point candidate : candidates) {
            int dist = this.squaredEuclideanDistance(candidate);
            if (dist <= distance * distance) {
                neighborhood.add(candidate);
            }
        }

        return neighborhood;
    }
    */

    public List<Point> neighborhoodManhattan(List<Point> candidates, int distance) {
        List<Point> neighborhood = new ArrayList<>();

        for (Point candidate : candidates) {
            // Use Manhattan distance instead of squared Euclidean distance
            int manhattanDist = this.manhattanDistance(candidate);
            if (manhattanDist <= distance) {
                neighborhood.add(candidate);
            }
        }

        return neighborhood;
    }
```




```java
@Test
void testNeighborhoodManhattan() {
    Point p = new Point(0, 0);  // The point from which the distances will be measured

    List<Point> candidates = new ArrayList<>();
    candidates.add(new Point(1, 1));  // Manhattan distance = 2
    candidates.add(new Point(2, 2));  // Manhattan distance = 4
    candidates.add(new Point(3, 3));  // Manhattan distance = 6
    candidates.add(new Point(4, 4));  // Manhattan distance = 8

    // We expect only points with Manhattan distance <= 5 to be included
    List<Point> neighborhood = p.neighborhoodManhattan(candidates, 5);

    // Check the size of the neighborhood
    assertEquals(2, neighborhood.size());

    // Check that the expected points are included
    boolean found1_1 = false;
    boolean found2_2 = false;

    for (Point point : neighborhood) {
        if (point.x == 1 && point.y == 1) {
            found1_1 = true;
        }
        if (point.x == 2 && point.y == 2) {
            found2_2 = true;
        }
    }

    // Assert that the expected points were found
    assertTrue(found1_1, "Point (1,1) should be in the neighborhood");
    assertTrue(found2_2, "Point (2,2) should be in the neighborhood");
}
```

In the current `neighborhoodManhattan()` method, we are checking distances using the **squared Euclidean distance**, but the test expects filtering based on **Manhattan distance.** Also the test is updated