a2 # testing

Testing Complex Logic

TCSS 305 Programming Practicum

Testing real-world code is rarely as simple as asserting that `add(2, 3)` equals `5`. In Assignment 2, you'll face challenges like testing methods that make random choices, verifying deterministic preference hierarchies, and testing behavior that seeks specific terrain. This guide demonstrates practical patterns for testing complex logic without external mocking frameworks.

---

1 Professional Context — When Real Code Needs Mocking

In professional software development, testing often requires isolating your code from external dependencies that are slow, expensive, or unpredictable:

Dependency Type	Why We Mock It	Real-World Example
Payment processors	Avoid charging real credit cards during tests	Stripe, PayPal API calls
Email services	Prevent sending emails to real addresses	SendGrid, AWS SES
Databases	Avoid hitting production data or slow queries	PostgreSQL, MongoDB
External APIs	Network calls are slow and non-deterministic	Weather data, stock prices
File systems	Tests should not depend on disk state	Reading/writing files

Mocking frameworks like Mockito, EasyMock, and JMockit allow you to create test doubles—fake objects that simulate real dependencies. Here's a simplified pseudo-code example:

```
// Mockito example (not used in A2, shown for context)
PaymentProcessor mockProcessor = mock(PaymentProcessor.class);
when(mockProcessor.charge(anyDouble())).thenReturn(true);

// Now we can test our code without hitting a real payment API
OrderService service = new OrderService(mockProcessor);
assertTrue(service.processOrder(order));
```

1.1 Assignment 2 Is Different

In A2, **you don't need a mocking framework**. Your `chooseDirection()` and `canPass()` methods are **pure functions** with no external dependencies:

- No network calls
- No database queries
- No file I/O
- No system clock dependencies

Instead of mocking external dependencies, you'll **construct specific test scenarios** by creating controlled simulation states. The `createNeighbors()` helper method acts as your "mocking" mechanism—it lets you precisely control what terrain surrounds a vehicle without building an entire simulation map.

---

2 The Pattern — Constructing Test Scenarios

The starter test files you receive with A2 (TruckTest, CarTest, BicycleTest) include a helper method that constructs test scenarios. This method is already provided—you don't need to write it:

```
/**
 * Helper method to create a neighbor map for chooseDirection testing.
 *
 * @param north the terrain to the north
 * @param west the terrain to the west
 * @param south the terrain to the south
 * @param east the terrain to the east
 * @return a map of directions to terrain types
 */
private Map<Direction, Terrain> createNeighbors(final Terrain north,
                                                final Terrain west,
                                                final Terrain south,
                                                final Terrain east) {
    return Map.of(
```

```
            Direction.NORTH, north,
            Direction.WEST, west,
            Direction.SOUTH, south,
            Direction.EAST, east
        );
    }
```

Why this pattern exists:

1. **Reduces duplication** - Every `chooseDirection()` test needs a neighbor map
2. **Makes tests readable** - Clear what terrain is in each direction
3. **Creates controlled conditions** - Tests specific behaviors without full game state
4. **Acts as a "mock"** - Simulates game environment without external dependencies

This is "mocking" without a framework—constructing the exact scenario needed to test one specific behavior.

---

3 Testing Random Behavior

The Truck class presents a unique testing challenge: its `chooseDirection()` method randomly selects from valid directions. How do you test something that's supposed to be unpredictable?

3.1 The Problem

```
// Truck chooseDirection implementation (conceptual)
public Direction chooseDirection(Map<Direction, Terrain> neighbors) {
    List<Direction> validDirections = getValidDirections(neighbors);
    return validDirections.get(random.nextInt(validDirections.size()));
}
```

If you call this method once, you can't know what it will return. You could get NORTH, WEST, or EAST—all are correct answers.

3.2 The Solution: Verify Diversity Over Many Iterations

Instead of testing a single result, run the method many times and verify two properties:

1. **Diversity** - Multiple different directions appear over N tries
2. **Constraints** - Invalid directions never appear

Here's the actual test from TruckTest.java:

```
/**
 * Test that chooseDirection randomly selects from valid directions.
 *
 * <p>This test demonstrates how to test random behavior. We run the method
 * multiple times and verify that we see multiple different results,
 * proving that the selection is random rather than deterministic.
 */
@Test
void testChooseDirectionRandomlySelectsFromValidDirections() {
    final Truck truck = new Truck(0, 0, Direction.NORTH);

    // Set up neighbors: STREET to north, west, and east (all valid)
    // WALL to south (invalid)
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north
        Terrain.STREET,  // west
        Terrain.WALL,    // south
        Terrain.STREET   // east
    );

    // Run chooseDirection many times and collect the results
    final Set<Direction> seenDirections = new HashSet<>();
    for (int i = 0; i < TRIES_FOR_RANDOMNESS; i++) {
        final Direction chosen = truck.chooseDirection(neighbors);
        seenDirections.add(chosen);
    }

    // Truck should randomly choose from NORTH, WEST, or EAST
    // After 50 tries, we should have seen at least 2 different directions
    assertTrue(seenDirections.size() >= 2,
               "Truck should randomly select from valid directions. "
               + "Expected at least 2 different directions after "
               + TRIES_FOR_RANDOMNESS + " tries, but only saw: " +
seenDirections);

    // None of the chosen directions should be SOUTH (the wall)
    assertFalse(seenDirections.contains(Direction.SOUTH),
                "Truck should never choose SOUTH when it's a WALL");
}
```

The pattern in detail:

1. **Set up scenario** - Create neighbors with multiple valid options (NORTH, WEST, EAST)
2. **Run N times** - Execute `chooseDirection()` 50 times (`TRIES_FOR_RANDOMNESS`)
3. **Collect results** - Store each result in a `Set<Direction>` to track unique values
4. **Assert diversity** - Verify we saw at least 2 different directions
5. **Assert constraints** - Verify we never chose an invalid direction (SOUTH/WALL)

⚠ False Negatives in Random Tests

This test could technically fail even with correct code. Java's `Random` could theoretically pick NORTH all 50 times—the probability is approximately 1 in 2^50 (about 1 in 1 quadrillion).

Why we accept this risk: The probability of a false negative is astronomically low, far lower than the probability of a hardware error or cosmic ray bit flip. In practice, if this test fails, it's because your code has a real bug (perhaps it's not actually random, or it's filtering directions incorrectly).

This is an example of **probabilistic confidence**—we can't achieve mathematical proof that randomness works, but we can achieve practical certainty that's sufficient for real-world software development.

❗ Test YOUR Code, Not the Standard Library

Students often ask: "Why not count the frequency of each direction and verify they're roughly equal (16-17 times each) to test uniform distribution?"

Answer: That would test Java's `Random` implementation, not your code.

What you're testing:

- ✓ Your code correctly filters out invalid directions (walls)
- ✓ Your code selects from the valid set
- ✓ Your code produces different results over multiple calls

What you're NOT testing:

- ✗ Java's `Random` class has uniform statistical distribution
- ✗ The random number generator is cryptographically secure
- ✗ Long-term frequency patterns match theoretical expectations

The 50 tries choice:

- **Too few tries (5)** - Might miss bugs, could easily get same direction by chance
- **Just right (50)** - Catches bugs while staying focused on your code
- **Too many tries (10,000)** - Tests Java's Random statistical properties, not your logic

Philosophy: Trust the Java standard library. Your tests should increase confidence in YOUR code, not provide mathematical proof of the JDK's correctness.

3.3 Alternative Approaches (Not Used in A2)

Other ways to test random behavior, for your future reference:

Approach	How It Works	Trade-offs
Seed the Random	Pass a fixed seed to constructor, get predictable sequence	Requires test hooks in production code
Statistical testing	Chi-square test for uniform distribution	Overkill for most applications, tests Random not your code
Property-based testing	Frameworks like QuickCheck generate random inputs	Advanced topic, may be covered in later courses

For A2, the "run N times and verify diversity" pattern is the right balance of simplicity and confidence.

---

4 Testing Deterministic Preference Hierarchies

Unlike Truck's random behavior, Car has a **deterministic preference hierarchy**: it always prefers to go straight, then left, then right, and only reverses as a last resort.

4.1 The Problem

How do you test a hierarchy where each level only applies when higher priorities are unavailable?

```
// Car chooseDirection logic (conceptual)
if (straightIsValid) return straight;
else if (leftIsValid) return left;
else if (rightIsValid) return right;
else return reverse;
```

You can't test "left is chosen" if straight is also available—the method will never reach the left branch.

4.2 The Solution: One Test Per Level, Blocking Higher Priorities

Create one focused test for each level of the hierarchy, systematically blocking all higher-priority options:

Test 1: Straight is chosen when available

```
/**
 * Test that chooseDirection prefers to go straight when possible.
 *
 * <p>Car has deterministic preference: Straight → Left → Right → Reverse.
 * When facing NORTH with STREET ahead, it should always choose NORTH.
 */
@Test
void testChooseDirectionPrefersStraight() {
    final Car car = new Car(0, 0, Direction.NORTH);

    // Set up neighbors: STREET ahead, left, and right (all valid)
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north (straight)
        Terrain.STREET,  // west (left)
        Terrain.WALL,    // south (reverse - not considered)
        Terrain.STREET   // east (right)
    );

    // Car should choose straight (NORTH) every time
    final Direction chosen = car.chooseDirection(neighbors);
    assertEquals(Direction.NORTH, chosen,
                 "Car should prefer to go straight (NORTH) when available");
}
```

Test 2: Left is chosen when straight is blocked

```
/**
 * Test that chooseDirection turns left when straight is blocked.
 *
 * <p>When straight ahead is impassable, Car should prefer left over right.
 */
@Test
void testChooseDirectionTurnsLeftWhenStraightBlocked() {
    final Car car = new Car(0, 0, Direction.NORTH);

    // Set up neighbors: WALL ahead, STREET to left and right
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.WALL,    // north (straight - blocked)
        Terrain.STREET,  // west (left - should be chosen)
        Terrain.STREET,  // south (reverse - not considered unless forced)
        Terrain.STREET   // east (right)
    );

    final Direction chosen = car.chooseDirection(neighbors);
    assertEquals(Direction.WEST, chosen,
                 "Car should turn left (WEST) when straight is blocked");
}
```

Test 3: Right is chosen when straight and left are blocked

```
/**
 * Test that chooseDirection turns right when both straight and left are
blocked.
 *
 * <p>Right is the third preference in the hierarchy.
```

```
 */
@Test
void testChooseDirectionTurnsRightWhenStraightAndLeftBlocked() {
    final Car car = new Car(0, 0, Direction.NORTH);

    // Set up neighbors: WALL ahead and left, STREET to right
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.WALL,    // north (straight - blocked)
        Terrain.WALL,    // west (left - blocked)
        Terrain.STREET,  // south (reverse - not considered unless forced)
        Terrain.STREET   // east (right - should be chosen)
    );

    final Direction chosen = car.chooseDirection(neighbors);
    assertEquals(Direction.EAST, chosen,
                 "Car should turn right (EAST) when straight and left are
blocked");
}
```

Test 4: Reverse is chosen when all forward directions are blocked

```
/**
 * Test that chooseDirection reverses only when all other directions are
blocked.
 *
 * <p>Reverse is the last resort, used only when straight, left, and right
 * are all impassable.
 */
@Test
void testChooseDirectionReversesOnlyWhenForced() {
    final Car car = new Car(0, 0, Direction.NORTH);

    // Set up neighbors: WALL in all directions except reverse
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.WALL,    // north (straight - blocked)
        Terrain.WALL,    // west (left - blocked)
        Terrain.STREET,  // south (reverse - only option)
        Terrain.WALL     // east (right - blocked)
    );

    final Direction chosen = car.chooseDirection(neighbors);
    assertEquals(Direction.SOUTH, chosen,
                 "Car should reverse (SOUTH) when all other directions are
blocked");
}
```

4.3 The Pattern

Each test **isolates one decision** in the hierarchy:

Test	Verifies Level	Terrain Setup	Expected Result
Test 1	Straight	All directions valid	Expects straight
Test 2	Left	Straight blocked, left/right valid	Expects left
Test 3	Right	Straight/left blocked, right valid	Expects right
Test 4	Reverse	All forward blocked	Expects reverse

This pattern applies to any preference-based logic where choices are tried in a specific order.

Do I Really Need to Test All of That?

Yes! Yes, you do!

Students often ask: "Can't I just test one or two levels and assume the rest work?"

No. Here's why every test matters:

1. **Completeness** - Each test verifies a different decision point. Missing one test means that level is untested.
2. **Catches specific bugs** - A bug in the "right" branch won't be caught by testing "straight" and "left"
3. **Verifies hierarchy order** - Testing all levels ensures preferences are checked in the correct order
4. **Documents behavior** - Each test serves as executable documentation of how the hierarchy works

Real example: A student tested straight and left, but forgot to test right. Their code had a bug where right was checked before left (wrong order). The bug wasn't caught until manual testing—wasting hours of debugging time.

Testing all levels isn't busywork—it's ensuring your implementation is **complete and correct**.

---

5 Testing Seeking Behavior with Priority Overrides

Bicycle presents the most complex logic in A2: it has TWO decision layers:

1. **Trail seeking** - Prefers TRAIL terrain over any other valid terrain

2. **Fallback preference** - When no trails exist, follows Straight → Left → Right → Reverse

This is more complex than Car's single hierarchy because trail-seeking can override the normal directional preferences.

5.1 The Two-Layer Decision Process

```
// Bicycle chooseDirection logic (conceptual)
// Layer 1: Check for trails in S/L/R
if (trailStraight) return straight;
else if (trailLeft) return left;
else if (trailRight) return right;

// Layer 2: No trails found, follow standard preference
else if (straightIsValid) return straight;
else if (leftIsValid) return left;
else if (rightIsValid) return right;
else return reverse;
```

5.2 Testing the Override Behavior

Test 1: Trail straight ahead overrides other valid terrain

```
/**
 * Test that chooseDirection seeks trail straight ahead.
 *
 * <p>When a trail is straight ahead, Bicycle should always choose it,
 * even if other valid terrain (like streets) exists to the left or right.
 */
@Test
void testChooseDirectionSeeksTrailAhead() {
    final Bicycle bicycle = new Bicycle(0, 0, Direction.NORTH);

    // Set up neighbors: TRAIL ahead, STREET to left and right
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.TRAIL,   // north (straight - should be chosen for trail)
        Terrain.STREET,  // west (left - valid but not a trail)
        Terrain.WALL,    // south (reverse - not considered)
        Terrain.STREET   // east (right - valid but not a trail)
    );

    final Direction chosen = bicycle.chooseDirection(neighbors);
    assertEquals(Direction.NORTH, chosen,
                 "Bicycle should choose TRAIL straight ahead (NORTH) " +
                 "over other valid terrain");
}
```

Test 2: Trail to the left is chosen over straight STREET

```java
/**
 * Test that chooseDirection seeks trail to the left when not ahead.
 *
 * <p>Trail-seeking priority: Straight → Left → Right.
 * If no trail ahead but trail to the left, Bicycle should turn left.
 */
@Test
void testChooseDirectionSeeksTrailToLeft() {
    final Bicycle bicycle = new Bicycle(0, 0, Direction.NORTH);

    // Set up neighbors: STREET ahead, TRAIL to left, STREET to right
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north (straight - valid but not a trail)
        Terrain.TRAIL,   // west (left - should be chosen for trail)
        Terrain.WALL,    // south (reverse - not considered)
        Terrain.STREET   // east (right - valid but not a trail)
    );

    final Direction chosen = bicycle.chooseDirection(neighbors);
    assertEquals(Direction.WEST, chosen,
                 "Bicycle should turn left (WEST) to reach TRAIL " +
                 "when no trail is straight ahead");
}
```

Test 3: Trail to the right is chosen when not ahead or left

```java
/**
 * Test that chooseDirection seeks trail to the right when not ahead or left.
 *
 * <p>If trails are only to the right, Bicycle should turn right to reach
them.
 */
@Test
void testChooseDirectionSeeksTrailToRight() {
    final Bicycle bicycle = new Bicycle(0, 0, Direction.NORTH);

    // Set up neighbors: STREET ahead and left, TRAIL to right
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north (straight - valid but not a trail)
        Terrain.STREET,  // west (left - valid but not a trail)
        Terrain.WALL,    // south (reverse - not considered)
        Terrain.TRAIL    // east (right - should be chosen for trail)
    );

    final Direction chosen = bicycle.chooseDirection(neighbors);
    assertEquals(Direction.EAST, chosen,
                 "Bicycle should turn right (EAST) to reach TRAIL " +
                 "when no trail is ahead or left");
}
```

5.3 Testing Edge Cases

Test 4: Trail behind is ignored (no reversing for trails)

```
/**
 * Test that chooseDirection does NOT reverse to face a trail behind.
 *
 * <p>Bicycle seeks trails, but will not reverse just to face one.
 * If valid terrain exists ahead, left, or right, Bicycle follows
 * the standard S → L → R preference and ignores the trail behind.
 */
@Test
void testChooseDirectionIgnoresTrailBehind() {
    final Bicycle bicycle = new Bicycle(0, 0, Direction.NORTH);

    // Set up neighbors: STREET ahead, left, and right; TRAIL behind
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north (straight - should be chosen)
        Terrain.STREET,  // west (left)
        Terrain.TRAIL,   // south (reverse - should be ignored)
        Terrain.STREET   // east (right)
    );

    final Direction chosen = bicycle.chooseDirection(neighbors);
    assertEquals(Direction.NORTH, chosen,
                 "Bicycle should NOT reverse to face TRAIL behind. " +
                 "Should follow normal S→L→R preference and go straight
(NORTH)");
}
```

5.4 Testing the Fallback Behavior

Test 5: Normal preference when no trails exist

```
/**
 * Test that chooseDirection follows S→L→R preference when no trails are
nearby.
 *
 * <p>When no trails are in S/L/R directions, Bicycle should behave like Car:
 * prefer straight, then left, then right.
 */
@Test
void testChooseDirectionFollowsPreferenceWhenNoTrails() {
    final Bicycle bicycle = new Bicycle(0, 0, Direction.NORTH);

    // Set up neighbors: STREET ahead, left, and right (no trails)
    final Map<Direction, Terrain> neighbors = createNeighbors(
        Terrain.STREET,  // north (straight - should be chosen)
        Terrain.STREET,  // west (left)
        Terrain.WALL,    // south (reverse)
        Terrain.STREET   // east (right)
    );

    final Direction chosen = bicycle.chooseDirection(neighbors);
    assertEquals(Direction.NORTH, chosen,
                 "Bicycle should prefer straight (NORTH) when no trails are
```

```
nearby");
}
```

5.5 The Pattern for Complex Logic

Test	Verifies	Terrain Setup	Expected Result
Test 1	Trail override straight	Trail ahead, streets left/right	Expects trail ahead
Test 2	Trail override left	Street ahead, trail left	Expects trail left
Test 3	Trail override right	Streets ahead/left, trail right	Expects trail right
Test 4	Edge case: no reverse for trails	Trail behind, streets forward	Expects forward (no reverse)
Test 5	Fallback behavior	No trails, all streets	Expects normal S→L→R

This pattern applies to any logic with **conditional priorities**: test each priority level, test overrides, and test edge cases where priorities don't apply.

---

6 Testing Combinations: Multiple Input Dimensions

The `canPass(Terrain, Light)` method has two inputs, creating multiple combinations to test. Not every combination is equally important.

6.1 Grouping Related Assertions

Use `assertAll()` to group related assertions that test the same concept:

```
/**
 * Test that Car can pass through LIGHT terrain with GREEN and YELLOW lights,
 * but stops for RED lights.
 *
 * <p>This is a key behavioral difference between Car and Truck.
 */
```

```
@Test
void testCanPassLightWithAllLights() {
    final Car car = new Car(0, 0, Direction.NORTH);

    assertAll("Car should pass LIGHT with GREEN and YELLOW, but not RED",
        () -> assertTrue(car.canPass(Terrain.LIGHT, Light.GREEN),
                         "Car should pass LIGHT with GREEN light"),
        () -> assertTrue(car.canPass(Terrain.LIGHT, Light.YELLOW),
                         "Car should pass LIGHT with YELLOW light"),
        () -> assertFalse(car.canPass(Terrain.LIGHT, Light.RED),
                          "Car should NOT pass LIGHT with RED light (stops at
red lights)")
    );
}
```

Why use `assertAll()`:

- All assertions run even if one fails (see all problems at once)
- Groups related checks (all three are about LIGHT terrain)
- Clear failure messages show exactly which condition failed

6.2 Exhaustive vs. Representative Testing

For some combinations, test **all possibilities**:

```
// Truck ignores ALL traffic lights - test all three
@Test
void testCanPassLightWithAllLights() {
    final Truck truck = new Truck(0, 0, Direction.NORTH);

    assertAll("Truck should pass LIGHT with all light colors",
        () -> assertTrue(truck.canPass(Terrain.LIGHT, Light.GREEN)),
        () -> assertTrue(truck.canPass(Terrain.LIGHT, Light.YELLOW)),
        () -> assertTrue(truck.canPass(Terrain.LIGHT, Light.RED))
    );
}
```

For other combinations, test **representative samples**:

```
// Testing EVERY terrain × light combination would be 5 terrains × 3 lights =
15 tests
// Instead, group by behavior:
// - All lights on TRAIL (always pass)
// - Light behavior on LIGHT terrain (varies by color)
// - Light behavior on CROSSWALK (varies by color)
```

When to test exhaustively:

- Small number of combinations (less than 10)
- Each combination has different behavior

- Specification explicitly lists all cases

When to test representatively:

- Large number of combinations
- Many combinations have identical behavior
- You're testing categories, not individual values

Parameterized Tests

JUnit 5's `@ParameterizedTest` annotation allows you to run the same test with multiple inputs:

```
@ParameterizedTest
@EnumSource(Light.class)  // Runs test with GREEN, YELLOW, RED
void testTruckIgnoresAllLights(Light light) {
    final Truck truck = new Truck(0, 0, Direction.NORTH);
    assertTrue(truck.canPass(Terrain.LIGHT, light));
}
```

This is more concise than `assertAll()` for testing the same assertion with different inputs. While not required for A2, parameterized tests are a powerful technique worth exploring. See the JUnit 5 User Guide link in Further Reading for details.

---

7 Testing Stateful Behavior

Some methods change object state rather than computing values. Testing these requires a **set** → **call** → **verify** pattern.

7.1 Example: Collision Disables Vehicles

```
// Test pattern (conceptual - you'll write the actual tests)
@Test
void testCollideDisablesVehicle() {
    final Car car = new Car(0, 0, Direction.NORTH);

    // Initial state
    assertTrue(car.isEnabled(), "Car should start enabled");

    // State change
    car.collide();

    // Verify change
```

```
    assertFalse(car.isEnabled(), "Car should be disabled after collision");
}
```

7.2 Example: Poke Advances Disabled State

```
@Test
void testPokeAdvancesDisabledTime() {
    final Car car = new Car(0, 0, Direction.NORTH);
    car.collide();  // Disable the car

    // Car is disabled for 15 turns
    for (int i = 0; i < CAR_DISABLED_DURATION - 1; i++) {
        car.poke();
        assertFalse(car.isEnabled(),
                    "Car should still be disabled after " + (i + 1) + "
pokes");
    }

    // Final poke should re-enable
    car.poke();
    assertTrue(car.isEnabled(),
               "Car should be enabled after " + CAR_DISABLED_DURATION + "
pokes");
}
```

7.3 Example: Reset Restores Initial State

```
@Test
void testResetRestoresInitialState() {
    final Car car = new Car(10, 20, Direction.EAST);

    // Change state
    car.setX(50);
    car.setY(60);
    car.setDirection(Direction.WEST);
    car.collide();

    // Reset
    car.reset();

    // Verify restoration
    assertAll("Reset should restore all initial state",
        () -> assertEquals(10, car.getX(), "X should be restored"),
        () -> assertEquals(20, car.getY(), "Y should be restored"),
        () -> assertEquals(Direction.EAST, car.getDirection(),
                           "Direction should be restored"),
        () -> assertTrue(car.isEnabled(), "Car should be re-enabled")
    );
}
```

7.4 The Pattern

Phase	Purpose	Example
Set	Establish initial state	`car.collide()` to disable
Call	Execute method under test	`car.poke()`
Verify	Assert expected change	`assertTrue(car.isEnabled())`

This pattern works for any method that modifies state: constructors, setters, state transition methods.

---

8 Helper Methods and Test Organization

Well-organized tests are easier to maintain and understand. Here are patterns used in the A2 test files:

8.1 Helper Methods for Setup

```
/**
 * Helper method to create a neighbor map for chooseDirection testing.
 */
private Map<Direction, Terrain> createNeighbors(final Terrain north,
                                                final Terrain west,
                                                final Terrain south,
                                                final Terrain east) {
    return Map.of(
        Direction.NORTH, north,
        Direction.WEST, west,
        Direction.SOUTH, south,
        Direction.EAST, east
    );
}
```

Benefits:

- Eliminates duplication (every `chooseDirection()` test needs this)
- Makes test setup clear and readable
- Centralizes map creation logic

8.2 Constants for Test Values

```java
/** The number of times to run random tests to ensure randomness. */
private static final int TRIES_FOR_RANDOMNESS = 50;

/** Expected mass for a Car. */
private static final int CAR_MASS = 50;

/** Expected disabled duration for a Car. */
private static final int CAR_DISABLED_DURATION = 15;
```

Benefits:

- Documents expected values
- Single source of truth (change once if specs change)
- Makes intent clear (`CAR_MASS` is more meaningful than `50`)

8.3 Descriptive Test Names

Good test names describe WHAT they test and WHEN:

Good Name	Why It Works
`testChooseDirectionTurnsLeftWhenStraightBlocked`	Describes action and condition
`testCanPassLightWithAllLights`	Describes method and input range
`testChooseDirectionRandomlySelectsFromValidDirections`	Describes expected behavior

Poor Name	Why It Fails
`testChooseDirection`	Too vague, doesn't describe scenario
`testMethod1`	Meaningless
`testBug`	Not descriptive, dated once bug is fixed

8.4 Good Assertion Messages

```
assertEquals(Direction.NORTH, chosen,
            "Car should prefer to go straight (NORTH) when available");

assertTrue(seenDirections.size() >= 2,
          "Truck should randomly select from valid directions. "
          + "Expected at least 2 different directions after "
          + TRIES_FOR_RANDOMNESS + " tries, but only saw: " + seenDirections);
```

Good assertion messages:

- Explain WHAT was expected and WHY
- Include actual values when helpful
- Use full sentences

Poor assertion messages:

- Empty strings
- Repeating the assertion: `assertEquals(5, x, "x should equal 5")`
- Not explaining context

Summary

Testing complex logic requires different patterns than testing simple calculations. Here's your toolbox:

Testing Challenge	Pattern	A2 Example	Key Technique
Random behavior	Run N times, verify diversity	Truck randomness	Collect results in `Set`, assert size ≥ 2
Deterministic preference	One test per level, block higher priorities	Car S→L→R→Reverse	Block straight/left to test right
Seeking with override	Test override + fallback + edge cases	Bicycle trail seeking	Test override (trail), fallback (no trail), edge (trail behind)

Testing Challenge	Pattern	A2 Example	Key Technique
Multiple input combinations	Group with `assertAll()`, test exhaustively or representatively	`canPass()` tests	`assertAll()` for related assertions
Stateful behavior	Set → call → verify	`collide()`, `poke()`, `reset()`	Verify state before and after
Helper methods	Reduce duplication, improve readability	`createNeighbors()`	Centralize repeated setup code

Key Principles:

1. **Test YOUR code, not the standard library** - Don't write tests that verify Java's Random works correctly
2. **Accept probabilistic confidence** - Random tests can theoretically fail correctly; risk is negligible
3. **Isolate one behavior per test** - Block higher priorities to test lower ones
4. **Test overrides, fallbacks, and edges** - Complex logic has multiple decision layers
5. **Use descriptive names and messages** - Future you will thank present you

These patterns apply beyond A2. You'll use them throughout your career when testing:

- Search algorithms with multiple ranking factors
- Business rules with priority hierarchies
- State machines with complex transitions
- Any non-deterministic behavior (network retries, randomized algorithms)

Further Reading

External Resources

- JUnit 5 User Guide — Complete reference for JUnit 5, including `assertAll()`, parameterized tests, and advanced features
- JUnit 5 Best Practices — Modern testing patterns in Java
- Testing Non-Deterministic Code — Martin Fowler on testing randomness and timing
- Mockito Documentation — Leading Java mocking framework (for future reference)

References

Primary Texts:

- Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley. — Foundational text on TDD practices and testing philosophy
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. — Item 83: Use custom exceptions judiciously; testing error conditions
- Freeman, S., & Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley. — Advanced testing patterns and design

Language Documentation:

- Oracle JDK 25 Documentation — Official Java SE reference
- java.util.Random — Random number generation in Java

Testing Documentation:

- JUnit 5 User Guide — Official JUnit 5 documentation
- JUnit 5 API Documentation — Complete API reference for org.junit.jupiter

Additional Resources:

- Osherove, R. (2013). *The Art of Unit Testing* (2nd ed.). Manning. — Practical patterns for writing maintainable tests
- Google Testing Blog — Industry perspectives on testing practices
- Test Pyramid — Ham Vocke's guide to structuring test suites

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.