a1b # testing

Introduction to Unit Testing: Why Tests Matter

TCSS 305 Programming Practicum

This guide introduces the fundamentals of software testing, with a focus on unit testing. You'll learn why testing matters, what makes unit tests different from other types of testing, and how to read and run the provided tests in Assignment 1A. A future guide will cover writing your own JUnit 5 tests.

Why Test Software At All?

Before we dive into unit testing specifically, let's step back and ask: why do we test software in the first place?

The Cost of Bugs

Bugs are expensive. The later you find a bug in the development process, the more it costs to fix. Consider this progression:

When Found	Cost to Fix	Why
During coding	Minutes	You're already looking at the code
During code review	Hours	Someone else has to understand it
During testing	Days	Bug must be reproduced, located, fixed, re-tested
After release	Weeks	Add customer support, reputation damage, potential recalls

A bug caught while you're writing the code might take 5 minutes to fix. That same bug discovered by a customer might require days of investigation, an emergency patch, and a public apology. Testing is about finding bugs early, when they're cheap to fix.

Confidence in Code

How do you know your code works? "I ran it and it seemed fine" is not a reliable answer. Tests provide concrete evidence that specific behaviors work correctly. When all tests pass, you have documented proof that the code does what it's supposed to do.

Quality Assurance

Testing is how professional software teams ensure quality. Every serious codebase has tests. Companies like Google, Amazon, and Microsoft run millions of tests daily. Testing isn't optional in industry—it's fundamental to how software gets built.

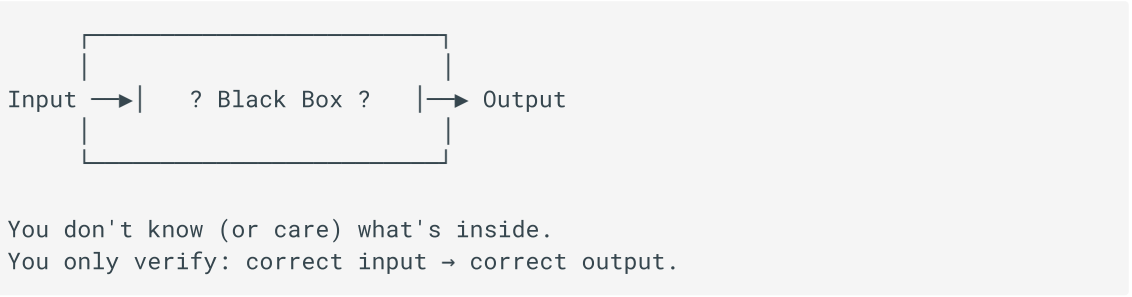
> **Important**
>
> Testing is not about proving your code is perfect. It's about catching bugs before they reach users and building confidence that the code works as specified.

Testing Approaches: Black Box vs White Box

There are two fundamental approaches to testing, and understanding them helps you think about what tests actually verify.

Black Box Testing

Black box testing treats the code as opaque. You can't see inside it—you only see inputs and outputs. You're testing: "Does it meet the specification?"




```
You don't know (or care) what's inside.
You only verify: correct input → correct output.
```

A black box test for a calculator's add function might be: - Input: 2 and 3 - Expected output: 5 - Does it return 5? Pass. Does it return anything else? Fail.

The test doesn't care whether the function uses a `+` operator, loops through additions, or calls a web service. It only cares about the result.

White Box Testing

White box testing looks inside the code. You can see the implementation—the branches, loops, and logic paths. You're testing: "Does every part of the implementation work correctly?"

```
      ┌──────────────────────────┐
      │  if (x > 0) {            │
Input ──►│    return positive();│──► Output
      │  } else {                │
      │    return negative();    │
      └──────────────────────────┘

You can see the branches.
You design tests to cover each path.
```

A white box test might specifically exercise the `else` branch to ensure `negative()` works correctly—something you might not think to test if you couldn't see the code.

Connecting to Contracts

Black box testing verifies the **contract**—the documented behavior from the interface. White box testing can additionally verify the **implementation**—how the code achieves that behavior.

In Assignment 1A, the provided tests are essentially black box tests. They verify that your `StoreItem` implementation meets the contract defined by the `Item` interface, without assumptions about how you implemented it.

> **Tip**
>
> When you write tests in Assignment 1B, you'll be doing black box testing—writing tests from the API specification without seeing the implementation.

Types of Testing

Software testing happens at multiple levels. Each level catches different kinds of problems.

Type	What It Tests	Example
Unit tests	Individual methods/classes in isolation	Does `getPrice()` return the correct value?

Type	What It Tests	Example
Integration tests	Components working together	Does the cart correctly use items from the inventory?
System tests	The entire application end-to-end	Can a user browse items, add to cart, and checkout?
Acceptance tests	Does it meet business requirements?	Does the discount calculation match what marketing specified?
Performance tests	Speed, load, stress	Can the system handle 1000 concurrent users?
Regression tests	Do old bugs stay fixed?	After the update, does bug #427 still work correctly?

Why so many types? Because different tests catch different problems. A unit test might confirm that your price calculation is mathematically correct. An integration test might reveal that the calculation gets the wrong data from the database. A system test might show that the result displays incorrectly in the UI.

The Testing Pyramid

The **testing pyramid** is a model for how many of each type of test you should have:

```
          /\
         /  \       Few E2E tests
        /    \      (slow, expensive, catch integration issues)
       /──────\
      /        \    Some integration tests
     /          \   (moderate speed, test component interactions)
    /────────────\
   /              \ Many unit tests
  /                \(fast, cheap, verify individual behavior)
 └──────────────────┘
```

Unit tests form the foundation because they're: - **Fast** - Run in milliseconds - **Cheap** - Easy to write and maintain - **Specific** - When they fail, you know exactly what's broken - **Isolated** - Don't depend on databases, networks, or other systems

This course focuses on unit testing because it's the foundation of all testing.

What is Unit Testing?

Unit testing is testing individual "units" of code—typically a single method or class—in isolation from the rest of the system.

Key Characteristics

Automated: Unit tests run the same way every time. You click a button (or run a command), and the test framework executes all your tests and reports results. No human judgment required.

Isolated: Each test runs independently. If you're testing `StoreItem.getPrice()`, you're not also testing the database, the network, or the GUI. You're testing just that one method.

Repeatable: Running the same test 100 times produces the same result. Tests don't depend on the time of day, network availability, or random factors.

Fast: Individual unit tests run in milliseconds. You can run thousands of tests in seconds.

Unit Testing vs Manual Testing

You've done manual testing before:

1. Run the program
2. Click around
3. Check if things look right
4. Notice something seems wrong
5. Try to remember what you did to cause it

This is slow, inconsistent, and doesn't scale. Did you test every edge case? Did you test the same things today that you tested yesterday? Did you catch the bug you introduced three commits ago?

Unit tests solve these problems by encoding your tests as code that runs automatically, consistently, every time.

> **Note**
>
> Manual testing still has a place—especially for user experience and visual design. But for verifying that code behaves correctly, automated tests are far superior.

Why Unit Testing Matters

Catches Bugs Early

When you write code and then immediately test it, bugs are found when the code is fresh in your mind. You know what you just changed. You can fix it in minutes. Compare this to finding a bug weeks later when you've forgotten how the code works.

Confidence to Change Code

Have you ever been afraid to change working code because you might break something? With a comprehensive test suite, you can refactor fearlessly. Change the code, run the tests. If they pass, you didn't break anything. If they fail, you know exactly what you broke.

Documentation

Tests are executable documentation. They show exactly how code is supposed to be used:

```
@Test
void testItemCreation() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals("Pen", item.getName());
}
```

This test documents that you create a `StoreItem` with a name and price, and that `getName()` returns the name you provided.

Professional Standard

Every serious software company requires tests. Code without tests is considered incomplete. When you interview for software jobs, you'll be asked about testing. When you contribute to open source projects, your pull requests need tests.

The question isn't "should I write tests?" but "how do I know my code works?" The professional answer is: "The tests pass."

The Testing Mindset

To test effectively, you need to think differently about your code.

Tests Verify the Contract

Remember interfaces and contracts from the previous guide? Tests verify that an implementation fulfills its contract. The `Item` interface says `getPrice()` returns a `BigDecimal`. A test verifies that it actually does.

> **Guide**
>
> Interface Contracts — Understanding what it means to implement an interface.

Tests Care About Behavior, Not Implementation

Good tests don't care how the code works internally—they care what it does. If `getPrice()` returns `1.99`, the test passes whether you stored the price in a field, calculated it from other fields, or looked it up in a file.

This is why tests survive refactoring. You can completely rewrite the internals of a class, and as long as the external behavior is the same, the tests still pass.

Think in Terms of Inputs and Outputs

For each test, think: - **Given** this input (or initial state)... - **When** I call this method... - **Then** I expect this output (or resulting state)

Each Test Tests ONE Thing

A good unit test verifies a single behavior. If it fails, you know exactly what's broken. If a test verifies five things and fails, you have to investigate which one failed.

Anatomy of a Test: Arrange, Act, Assert

Every unit test follows the same basic pattern:

Arrange

Set up the test. Create objects, initialize data, establish preconditions.

Act

Call the method you're testing. This is usually a single line.

Assert

Verify the result. Check that the output matches your expectations.

```
@Test
void testGetPrice() {
    // Arrange - set up the test data
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));

    // Act - call the method being tested
    BigDecimal price = item.getPrice();

    // Assert - verify the result
    assertEquals(new BigDecimal("1.99"), price);
}
```

This pattern is sometimes called AAA (Arrange-Act-Assert) or Given-When-Then. The names differ, but the concept is universal across all testing frameworks and languages.

> **Tip**
>
> When reading tests, look for these three phases. They help you understand what the test is verifying and why.

What Tests Look Like: JUnit 5

In TCSS 305, we use **JUnit 5**, the standard testing framework for Java. Here's just enough to read the provided tests—you'll learn to write your own in Assignment 1B.

Basic Structure

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

class StoreItemTest {

    @Test
    void testGetName() {
        // Arrange
        Item item = new StoreItem("Pen", new BigDecimal("1.99"));

        // Act
        String result = item.getName();

        // Assert
        assertEquals("Pen", result);
    }
}
```

Key Elements

`@Test` **annotation:** Marks a method as a test. JUnit runs every method with this annotation.

Test method names: Describe what's being tested. `testGetName`, `testPriceCannotBeNegative`, `testEqualsReturnsFalseForNull`.

Assertion methods: Verify expected outcomes:

Method	What It Checks
`assertEquals(expected, actual)`	Values are equal
`assertNotEquals(a, b)`	Values are not equal
`assertTrue(condition)`	Condition is true
`assertFalse(condition)`	Condition is false
`assertNull(value)`	Value is null
`assertNotNull(value)`	Value is not null
`assertThrows(Exception.class, () -> ...)`	Code throws expected exception

Example: Testing Exceptions

Tests can verify that invalid input throws the correct exception:

```
@Test
void testConstructorRejectsNullName() {
    assertThrows(NullPointerException.class, () -> {
        new StoreItem(null, new BigDecimal("1.99"));
    });
}
```

This test passes if `StoreItem` throws a `NullPointerException` when given a null name. It fails if no exception is thrown or if a different exception type is thrown.

Running Tests in Assignment 1A

In Assignment 1A, the test class `StoreItemTest.java` is provided. These tests verify that your `StoreItem` implementation meets the contract defined by the `Item` interface.

What the Provided Tests Look Like

The provided tests follow the patterns described in this guide. Here are a few representative examples:

Testing valid construction:

```
@Test
void testConstructorValid() {
    final StoreItem item = new StoreItem(ITEM_NAME, new
BigDecimal(ITEM_PRICE));
    assertEquals(ITEM_NAME, item.getName(), "Name should match constructor
argument");
    assertEquals(new BigDecimal(ITEM_PRICE), item.getPrice(),
            "Price should match constructor argument");
}
```

This test creates a `StoreItem` with valid arguments and verifies that `getName()` and `getPrice()` return the expected values.

Testing exception handling:

```
@Test
void testConstructorNullName() {
    assertThrows(NullPointerException.class,
            () -> new StoreItem(null, new BigDecimal(ITEM_PRICE)),
            "Constructor should throw NullPointerException for null name");
}
```

This test verifies that the constructor properly rejects null input by throwing `NullPointerException`.

Testing calculations:

```
@Test
void testCalculateTotalBasic() {
    final StoreItem item = new StoreItem(ITEM_NAME, new
BigDecimal(ITEM_PRICE));
    final BigDecimal expected = new BigDecimal(SIX_DOLLARS);
    assertEquals(expected, item.calculateTotal(TEST_QUANTITY),
            "calculateTotal should return price * quantity");
}
```

This test verifies that `calculateTotal()` correctly computes price × quantity.

How to Run Tests in IntelliJ

Method 1: Right-click the test class 1. Open `StoreItemTest.java` in the editor 2. Right-click anywhere in the file 3. Select **Run 'StoreItemTest'**

Method 2: Click the play button 1. Open `StoreItemTest.java` 2. Click the green play arrow next to the class name (runs all tests in the class) 3. Or click the play arrow next to a specific test method (runs just that test)

Method 3: Run with Coverage 1. Right-click the test class 2. Select **Run 'StoreItemTest' with Coverage** 3. IntelliJ shows which lines of your code were executed by the tests

Reading Test Results

After running tests, IntelliJ displays results in the Run panel:

Green checkmark: Test passed. Your code behaves as expected.

Red X: Test failed. Your code doesn't match the expected behavior.

Yellow exclamation: Test had an error (exception during setup, not an assertion failure).

Understanding Failure Messages

When a test fails, the output tells you:

1. **Which test failed:** The test method name
2. **What was expected:** The value the test expected
3. **What actually happened:** The value your code produced
4. **Stack trace:** Where in the code the failure occurred

Example failure message:

```
org.opentest4j.AssertionFailedError:
Expected :Pen
Actual   :null
```

This tells you: the test expected `"Pen"` but got `null`. Check your `getName()` method—it's probably returning null instead of the stored name.

The Fix-and-Run Cycle

Development with tests follows this cycle:

1. **Run tests** - See what fails
2. **Read failure messages** - Understand what's wrong
3. **Fix your code** - Address the specific failure

4. **Run tests again** - Verify the fix worked
5. **Repeat** - Until all tests pass

This tight feedback loop is why unit tests accelerate development—you get immediate, specific feedback on exactly what's wrong.

Tests as Your Feedback Loop

The provided tests are not adversaries—they're allies. They tell you exactly what's expected and whether you've achieved it.

Trust the Tests

In Assignment 1A, the tests verify the contract you're implementing. If a test fails, the bug is in your code, not in the test. The tests are written by your "QA team" (the course staff) and define correct behavior.

Tests Give Specific Feedback

Unlike running the application and hoping things look right, tests tell you precisely: - Which behavior is wrong (the test name) - What was expected (the assertion's expected value) - What you got (the assertion's actual value) - Where to look (the stack trace)

All Tests Must Pass

Before submitting, every test must pass. A single failing test means something is wrong with your implementation. Don't submit with failing tests hoping for partial credit—fix the issue.

> **Warning**
>
> If you can't figure out why a test is failing, read the test code carefully. It shows exactly what inputs are being used and what output is expected. The test is the specification.

Looking Ahead

In Assignment 1A, you're using provided tests to verify your implementation. This is a common scenario—you implement code to pass existing tests.

In Assignment 1B, you'll flip the script: you'll **write tests first** using Test-Driven Development (TDD). You'll receive an API specification and write tests before seeing any implementation. This teaches you to think about behavior before code.

A future guide will cover: - How to write effective JUnit 5 tests - Test-Driven Development methodology - Testing edge cases and error conditions - Achieving code coverage

For now, focus on understanding what tests do and how to run them. The ability to read tests and interpret their results is the foundation for writing your own.

Summary

Concept	Key Point
Why test	Find bugs early, gain confidence, ensure quality
Black box testing	Test behavior without seeing implementation
White box testing	Test with knowledge of implementation details
Unit testing	Test individual methods/classes in isolation
Arrange-Act-Assert	The universal pattern for structuring tests
JUnit 5	Java's standard testing framework
Running tests	Right-click, play button, or Run with Coverage
Reading failures	Expected vs Actual tells you what's wrong

Testing is not an optional add-on to development—it's integral to writing reliable software. The skills you build now will serve you throughout your career.

Further Reading

External Resources

- JUnit 5 User Guide - Official documentation for JUnit 5
- Oracle Java Tutorial: Testing - Java testing basics
- Baeldung: JUnit 5 - Practical JUnit 5 tutorial
- Martin Fowler: Test Pyramid - In-depth article on testing strategies
- Agile Alliance: Unit Testing - Unit testing in agile development

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Chapter 10: Exceptions.
- Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley. — Foundational TDD text.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 7: Exceptions, Assertions, and Logging.

Testing Frameworks:

- JUnit 5 User Guide — Official JUnit 5 documentation
- JUnit 5 API Documentation — Javadoc for JUnit 5
- AssertJ Documentation — Popular assertion library (supplementary)

Testing Methodology:

- Fowler, M. (2012). The Practical Test Pyramid — Testing strategies and test types
- Arrange-Act-Assert Pattern — Test structure pattern documentation

Tooling:

- IntelliJ IDEA: Running Tests — Running JUnit tests in IntelliJ
- IntelliJ IDEA: Test Coverage — Code coverage analysis

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