A1b # lectures # testing # week-2

Week 2: Introduction to Unit Testing (Monday, January 13, 2025)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A1b — writing formalized automated unit tests.

Week Overview and A1b Structure [0:00]

This week focuses on three main topics:

1. **Formalized testing** — How to write automated unit tests using a testing framework
2. **Object class methods** — What the `equals()`, `hashCode()`, and `toString()` methods should do according to their API contracts
3. **Java memory model** — Continuing to build our picture of how memory works in Java applications

Assignment 1B Repository Structure

When you clone your A1b repository from GitHub Classroom, you'll find:

- `test/edu/uw/tcss/model/` — Contains `StoreItemOrderTest.java` (provided example)
- `demo/edu/uw/tcss/app/` — Contains `StoreDemo.java` with smoke tests
- `demo/lib/` — Contains a **proxy JAR file**

Key point: The classes you're testing (`StoreItem`, `StoreCart`, etc.) are wrapped inside the JAR file. You don't have access to the source code—that's intentional. The JAR proxies all calls to an HTTP server, so:

- You can't decompile the JAR to see the solution

- Tests run slightly slower due to network latency
- This forces true **black box testing**—you write tests against the *specification*, not the implementation

Next week in A1c, you'll implement these classes yourself. Your tests from this week should pass regardless of implementation details.

Types of Testing [6:34]

Everyone has been testing since day one of 142. The first time you hit "Run" and checked if your program worked, you performed **ad hoc exploratory testing**.

Informal Testing (Developer-Performed)

Type	Description
Smoke Testing	"Does it blow up?" Run the program and check for crashes/exceptions
Exploratory Testing	Run the program to learn its behavior, poke around to see what it does
Debugging	Using breakpoints to step through code and identify specific buggy behavior

You can formalize smoke tests by writing them as code (like `StoreDemo.java`), which ensures they run the same way every time.

Formalized Testing (Often QA Team)

Type	Description
Unit Testing	Test individual units (usually methods) in isolation
Integration Testing	Test how components work together—from small (frontend ↔ backend) to large (database ↔ API ↔ UI)
Performance Testing	Verify code completes within acceptable time tolerances

Why separate developers from testers? If a developer misunderstands the requirements and writes incorrect code, they'll likely write tests that validate their incorrect understanding. Having separate teams catches these misalignments.

> **Performance Testing vs Big O**
>
> Big O notation describes *asymptotic* runtime growth. Performance testing deals with *real-world* execution time. Even with optimal O(log n), constants matter in production. Amazon search results need to return in milliseconds—if users wait 2 minutes, they leave.

Unit Testing Fundamentals [20:05]

For this course, we focus on **unit testing** at the method level.

What is a Unit?

- Typically **a single method** (our focus)
- Sometimes as large as a class
- The QA team or development team decides the unit size

Black Box Testing

We treat the method's internals as a **black box**—we can't see inside it. We test based on:

- **Inputs** → What we send to the method
- **Expected outputs** → What the documentation says we should get back

```
Given input X and Y → Expect output Z
```

You don't care *how* the method produces the result. You only care that it produces the *correct* result according to the specification.

> **Don't Test Against Implementation**
>
> A common mistake: students write code first, then write tests based on what their code does.
>
> **Example:** Implementing circle area as `π × r` instead of `π × r²`, then testing for `π × r`. The test passes, but the code is wrong.
>
> Always test against the **specification**, not the implementation.

What to Test?

Test the **public API** of a class—the public methods.

> **Should we test private helper methods?**
>
> Generally no, for two reasons:
>
> 1. **You can't access them** — Private methods aren't visible to test classes
> 2. **You're already testing them indirectly** — Private helpers are called by public methods; testing the public method tests the helper
>
> If a helper method isn't used by any public method, delete it.

Testing Defensive Code [30:05]

Classes that practice **defensive programming** throw exceptions for invalid input rather than trying to fix it.

Why Throw Instead of Fix?

Consider a `SimpleFood` constructor that receives `null` for the name:

Approach	Problem
Fix it (replace with "Unknown")	Five months later, customer support gets a call about a receipt showing "Unknown" for $0.00. Where's the bug?
Throw exception	Fails fast. The calling code must handle bad input. Bug is caught immediately during development.

The caller provided bad input—they should deal with it, not the class receiving it.

Test the Unhappy Path

> **Critical Insight**
>
> It's **more important** to test that incorrect flows behave correctly than to test the correct flows.
>
> - **Correct flows** — You'll see these in smoke testing when you run the app
> - **Incorrect flows** — You won't naturally test sending `null` or `-500` for calories
>
> If you don't write explicit tests for exception cases, you won't catch bugs in error handling.

Test that exceptions are thrown when expected:

- `null` name → `IllegalArgumentException`
- Empty string name → `IllegalArgumentException`
- Negative calories → `IllegalArgumentException`

Also test boundary conditions:

- Zero calories should *not* throw (it's non-negative)
- Have separate tests for zero vs positive values

JUnit 5 Framework [34:59]

JUnit is part of the xUnit family of testing frameworks (JUnit for Java, PyUnit for Python, etc.). We use JUnit 5, which is integrated into IntelliJ.

The framework:

- Looks at your test classes in a specific way
- Runs methods marked with special annotations
- Reports pass/fail results

Writing Your First Test [38:08]

Test Class Setup

1. **Location:** Put test classes in the `test/` source folder, matching the package of the class under test
2. **Naming convention:** `ClassNameTest` (e.g., `SimpleFoodTest` for `SimpleFood`)
3. **Package matching:** IntelliJ links `test/model/` to `src/model/`, so no imports needed for classes in the same package

```java
package edu.uw.tcss.model;

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

class SimpleFoodTest {
    // Tests go here
}
```

The @Test Annotation

The `@Test` annotation tells JUnit: "This method is a test—run it."

```java
@Test
void testConstructorWithValidArguments() {
    // Test code here
}
```

Without `@Test`, JUnit ignores the method.

Test Structure: Arrange-Act-Assert [52:00]

Every test follows three steps:

Step	Purpose	Example
Arrange	Set up test data and objects	Create expected values, instantiate objects
Act	Call the method under test	`food.getName()`
Assert	Verify the result matches expectations	`assertEquals(expected, actual)`

```java
@Test
void testConstructorWithValidArguments() {
    // Arrange
    final String expectedName = "Apple";
    final int expectedCalories = 95;

    // Act
    final SimpleFood food = new SimpleFood(expectedName, expectedCalories);

    // Assert
    assertEquals(expectedName, food.getName(),
```

```
                "Name should be set correctly");
}
```

Understanding Assertion Failures

When a test fails, JUnit shows:

- The assertion message (`"Name should be set correctly"`)
- Expected value (`"apple"`)
- Actual value (`"Apple"`)

```
Assertion failed: Name should be set correctly
Expected: apple
Actual: Apple
```

Read these carefully—the bug might be in your test, not the code!

Key Takeaways [55:02]

Passing Tests ≠ Bug-Free Code

An empty test method passes:

```java
@Test
void testSomething() {
    // This passes! But tests nothing.
}
```

This gives a **false sense of security**. A test that passes but verifies nothing is worse than no test at all.

Tests Can Have Bugs Too

- If you misunderstand requirements and write incorrect tests, they may pass against incorrect code
- If you misunderstand requirements and write incorrect tests against correct code, they'll fail even though the code is right
- **Always trace failures back to the specification**

Coming Wednesday: Writing tests for exceptions, more assertion types, and additional test patterns.

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