a1b # testing

Test-Driven Development: Writing Tests Before Code

TCSS 305 Programming Practicum

Test-Driven Development (TDD) is a software development approach where you write tests *before* writing the code that makes them pass. This guide explains the TDD philosophy, its benefits, and how to derive tests from specifications---a skill you will practice in Assignment 1b.

What is Test-Driven Development?

Traditional development follows a familiar pattern: write code, then test it to see if it works. TDD inverts this approach: **write tests first, then write code to make them pass**.

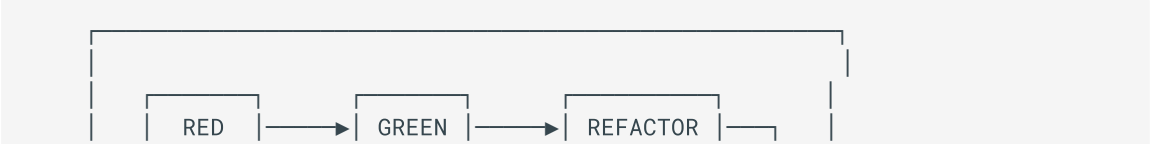
```
Traditional:  Write code -> Hope it works -> Test later (maybe)
TDD:          Write tests -> Watch them fail -> Write code -> Watch them pass
```

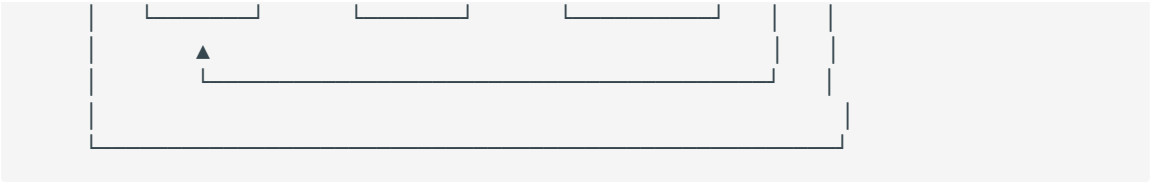
This inversion may seem counterintuitive. How can you test something that does not exist yet? The answer lies in specifications: you know *what* the code should do before you write it. Tests encode that knowledge.

The Red-Green-Refactor Cycle

TDD follows a disciplined cycle known as **Red-Green-Refactor**:

1. **Red:** Write a test for a behavior that does not exist yet. Run it. Watch it fail (red).
2. **Green:** Write the minimum code necessary to make the test pass. Run it. Watch it pass (green).
3. **Refactor:** Clean up the code while keeping tests green. Improve design without changing behavior.
4. **Repeat:** Move to the next behavior.





The cycle keeps you focused. Each iteration adds one behavior. Tests accumulate, building a safety net that catches regressions as you continue development.

> **Note**
>
> In Assignment 1b, you experience the "write tests first" aspect of TDD. You write tests against a specification before seeing any implementation. This is the essence of TDD's philosophy---defining expected behavior before the code exists.

Why TDD?

TDD offers several benefits that become apparent as projects grow in complexity.

Forces You to Think About Requirements First

When you write tests first, you must understand *what* the code should do before considering *how* to implement it. This shifts your focus from implementation details to behavior and requirements.

Consider a method `calculateTotal(int quantity, boolean membership)`. Before writing any implementation, TDD asks: What should this method return for various inputs? What happens with quantity zero? What about negative quantities? Should membership affect the result?

Answering these questions *before* coding prevents the common trap of writing code and then retrofitting requirements to match what you built.

Catches Design Issues Early

Writing tests first often reveals design problems. If a method is hard to test, it is probably poorly designed---too many responsibilities, too many dependencies, or unclear contracts.

TDD encourages:

- **Small, focused methods** that do one thing well
- **Clear interfaces** with explicit inputs and outputs

- **Loose coupling** so components can be tested independently

These qualities make code easier to test *and* easier to maintain.

Tests Serve as Living Documentation

Tests document exactly how code should behave. Unlike comments or external documentation, tests are executable and verified---they cannot become stale without failing.

When you encounter unfamiliar code, tests answer questions like:

- What inputs does this method expect?
- What does it return for edge cases?
- What exceptions does it throw for invalid input?

Tests are specifications that run.

Confidence When Refactoring

With comprehensive tests, you can refactor fearlessly. Change the internal implementation, run the tests. If they pass, behavior is preserved. If they fail, you know exactly what broke.

Without tests, refactoring becomes risky. Developers avoid improving code because they might break something. The codebase accumulates technical debt.

> **What is Technical Debt?**
>
> Technical debt is the accumulated cost of shortcuts and deferred improvements in code. Like financial debt, it accrues "interest"—the longer you wait to address it, the harder and more expensive it becomes to fix.
>
> Example: A team rushes to ship a feature and hardcodes several values instead of making them configurable. Six months later, a client needs those values changed. What should have been a config file edit now requires finding and modifying code scattered across dozens of files, retesting everything, and hoping nothing breaks.

Encourages Smaller, Focused Units

Because you write one test at a time, TDD naturally produces smaller units of code. Each test verifies a single behavior, and each increment of implementation addresses that behavior.

This leads to code that is:

- Easier to understand (small pieces)

- Easier to debug (clear boundaries)
- Easier to reuse (focused responsibilities)

Testing from Specifications

A critical TDD skill is deriving tests from specifications without seeing implementation. This is exactly what professional developers do when working with third-party libraries, web APIs, or colleague's code.

Black-Box Testing

When you test from a specification, you practice **black-box testing**: you know the inputs and expected outputs but not the internal workings.

```
      ┌─────────────────────────┐
      │                         │
Input ──►|   ? Black Box ?   |──► Output
      │                         │
      └─────────────────────────┘

You verify: correct input -> correct output
You do not see (or care about) what is inside.
```

This approach tests the **contract**---the documented promise of what the code will do.

Reading API Documentation Carefully

Specifications come in many forms: Javadoc, interface definitions, API documentation, or requirements documents. The skill is reading them carefully and extracting testable behaviors.

For each method, identify:

Question	What to Look For
What does it return?	`@return` documentation, return type
What are valid inputs?	Parameter types, `@param` documentation
What are invalid inputs?	`@throws` documentation, preconditions
What are edge cases?	Boundary values: 0, 1, empty, maximum

Question	What to Look For
What side effects occur?	State changes documented in description

Identifying Testable Behaviors

Each `@throws` tag is a testable behavior: provide that invalid input, verify that exception is thrown.

Each documented return value is a testable behavior: provide valid input, verify the return matches the specification.

Each edge case mentioned is a testable behavior: provide the boundary value, verify correct handling.

> **Tip**
>
> When reading a specification, make a list of behaviors to test. Each behavior becomes one test method. A well-specified method often yields 5-10 tests covering normal cases, edge cases, and error conditions.

Trust the Specification, Not Assumptions

Test what the specification *says*, not what you *assume* it means. If the spec says `@throws NullPointerException if name is null`, test for exactly `NullPointerException`, not `IllegalArgumentException` or a generic `RuntimeException`.

If the specification is ambiguous, note the ambiguity. In professional settings, you would ask for clarification. In coursework, follow the specification as written and document any assumptions in your tests.

> **Warning**
>
> A common mistake is testing against assumptions instead of specifications. If the spec does not mention a behavior, you cannot test for it. If the spec explicitly states a behavior, you must test for it---even if it seems unnecessary.

TDD in Assignment 1b

Assignment 1b gives you a unique TDD experience: you write tests against a provided library without seeing its source code. This mirrors real-world scenarios where you integrate with external systems—or if you join a QA team responsible for writing tests against code developed by other teams.

What You Will Do

1. **Read the API specification** (Javadoc) for the provided library
2. **Write JUnit 5 tests** that verify the specification
3. **Run your tests** against the implementation
4. **Discover whether the implementation meets its contract**

You are the quality assurance team. Your tests define "correct behavior" according to the specification. The implementation must pass your tests---or be revealed as defective.

The Skill You Are Building

This exercise develops a crucial professional skill: **deriving tests from specifications without implementation knowledge**.

When you:

- Use a third-party library, you test it against its documentation
- Call a web API, you test responses against the API contract
- Collaborate with other developers, you test their code against agreed interfaces

In all these cases, you write tests from specifications. Assignment 1b teaches exactly this skill.

Connecting to the Full TDD Cycle

In true TDD, you would:

1. Write a test (it fails because no implementation exists)
2. Write implementation to pass the test
3. Refactor
4. Repeat

In Assignment 1b, you complete step 1: writing tests. The implementation already exists (the provided library), so your tests immediately verify it rather than driving its creation. But the core skill---writing tests from specifications---is identical.

Guide

Writing JUnit 5 Tests — Annotations, assertions, exception testing, and test structure.

Gen AI & Learning: TDD in the Age of AI

As generative AI and agentic AI tools become common in software development, TDD becomes *more* valuable, not less. When AI generates code for you, how do you know it's correct? Tests.

The TDD workflow adapts naturally:

1. **You** write tests that define correct behavior (the AI doesn't know your requirements)
2. **AI** generates implementation code
3. **Tests** verify the AI's output meets your specification
4. **You** refactor or regenerate until tests pass

Or in fully agentic workflows:

1. **AI Agent 1** generates tests from specifications
2. **AI Agent 2** implements code to pass those tests
3. **Tests** serve as the contract between agents

AI can write code fast, but it cannot read your mind. Tests remain the specification—the source of truth that validates *any* implementation, whether written by you, a colleague, or an AI agent. The skill of writing good tests from specifications is more important than ever.

Summary

Concept	Key Point
TDD Definition	Write tests before code; tests drive implementation
Red-Green-Refactor	Fail, pass, clean up---the TDD cycle
Requirements focus	Tests force you to define behavior before coding
Living documentation	Tests document expected behavior in executable form
Refactoring confidence	Comprehensive tests enable fearless code improvement

Concept	Key Point
Black-box testing	Test against specifications, not implementation
Specification reading	Extract testable behaviors from `@return`, `@throws`, edge cases
Trust the spec	Test what is documented, not what you assume

Further Reading

External Resources

- Martin Fowler: Test Driven Development - Overview of TDD methodology
- Agile Alliance: TDD - TDD in agile development context
- JUnit 5 User Guide - Official JUnit 5 documentation
- James Shore: Red-Green-Refactor - Detailed explanation of the TDD cycle

References

Primary Texts:

- Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley. --- The foundational text on TDD methodology; introduces Red-Green-Refactor.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 49: Check parameters for validity; Chapter 10: Exceptions.
- 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 for writing and running tests.

Methodology:

- Fowler, M. (2014). Test Driven Development --- Overview of TDD philosophy and benefits.
- Arrange-Act-Assert Pattern --- Standard test structure pattern.

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