Introduction to Test-Driven Development (TDD)

Using cyber-dojo, Java and JUnit

Hårek Haugerud

February 18, 2026

Lecture plan

- What is TDD, and why is it an important development method?
- The classic TDD cycle: **Red** $\rightarrow$ **Green** $\rightarrow$ **Refactor**
- Joint exercise 1: **Leap Year** in cyber-dojo
- A brief JUnit introduction: structure, asserts, test naming, failures
- Joint exercise 2: **FizzBuzz** in cyber-dojo
- Summary: good practices and common pitfalls

What do we mean by “testing”?

- **Unit testing**: testing small parts (methods/classes) in isolation
- **Integration testing**: verifying collaboration between components/modules
- **System testing**: verifying that the entire system works as specified when integrated
- **Acceptance testing**: validating that the solution meets user needs and requirements

Today: focus on **unit testing** and the development practice **TDD**.

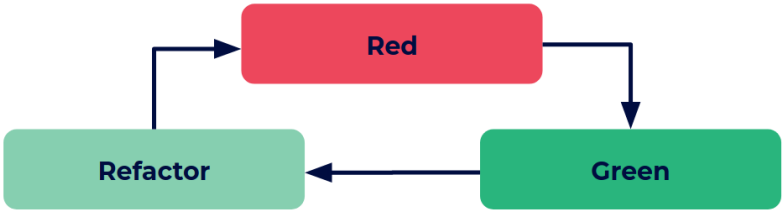
What is Test-Driven Development (TDD)?

- A development methodology where tests are written **before** production code
- Requirements and examples drive the design in small steps
- The goal is not *as many tests as possible*, but:
  - fast feedback
  - simpler design
  - safer changes
  - testable code

Why TDD? (practical benefits)

- **Fast feedback**: you immediately see if something breaks
- **Better design**: encourages small, testable units
- **Safer refactoring**: tests act as a safety net
- **Executable documentation**: tests describe expected behaviour
- **Less debugging**: more defects are caught early

The TDD cycle: Red → Green → Refactor



1. **Red**: write a test describing desired behaviour (which should fail!)
2. **Green**: write the *minimum* code needed to pass
3. **Refactor**: improve code and tests without changing functionality — run tests to confirm they stay green

Rule of thumb: Take very small steps. One new requirement ≈ one new test.

TDD in practice: what we will do

- Work in https://cyber-dojo.org, "a place to practice programming"
- Language: **Java**
- Test framework: **JUnit**
- We solve two classics:
  - **Leap Year**
  - **FizzBuzz** (if time permits)

Focus: **TDD rhythm** + **test structure** + **small steps**.

cyber-dojo: quick introduction

- Dojo (Japanese) = "place where the way (dō) is practised"
- Mastery through practice
- Browser-based coding with preconfigured language + test tools
- One-click test execution with immediate feedback (red/green/yellow)
- Ideal for TDD: frequent runs and tiny iterations

We use: Java 21 + JUnit 5

Link to JUnit 5 assertion-methods that can be used in cyber-dojo

Joint exercise 1: Leap Year

Create a method that determines whether a year is a leap year.

- Divisible by 4 $\rightarrow$ leap year
- Divisible by 100 $\rightarrow$ not leap year
- Divisible by 400 $\rightarrow$ leap year

Goal: develop the solution using TDD, repeating the cycles

Test-Driven Development (TDD) method – Iterative Cycle

Step 1: Write a Test (Red)

- Write a small, new test that specifies the desired behaviour
- The test should fail (no or incorrect implementation)
- Focus on what the system should do, not how

Step 2: Implement the Minimum (Green)

- Write just enough code to make the test pass
- No optimisation, no extra functionality
- Goal: make the test green

Step 3: Refactor (Refactor)

- Improve code quality: better names, remove duplication, clarify structure
- All tests must still pass
- Do not add new functionality $\rightarrow$ **return to Step 1**

Leap Year: one possible TDD progression (tests)

We begin by testing the simplest example and then extend with additional tests:

1. 2019 $\rightarrow$ `false` (not divisible by 4)
2. 2020 $\rightarrow$ `true` (divisible by 4)
3. 1900 $\rightarrow$ `false` (divisible by 100)
4. 2000 $\rightarrow$ `true` (divisible by 400)

Key point: each new rule $\Rightarrow$ new test $\Rightarrow$ minimal code.

JUnit: what is it?

- Standard Java unit testing framework
- Executes tests automatically
- Reports failures clearly, which test and why it fails
- Annotation-based test definitions (`@Test`)
- Assertions to validate expected outcomes
- Test runners for executing test suites

In Cyber-Dojo, JUnit provides the red/green feedback.

JUnit 5: basic structure

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

class LeapYearTest {

  @Test
  void yearNotDivisibleBy4_isNotLeapYear() {
    assertFalse(LeapYear.isLeapYear(2019));
  }
}
```

- `@Test` marks a test method
- `assertFalse`/`assertTrue` verify expectations
- A good test name describes the **scenario** and the **expected result**

What is an annotation? (`@Test` as an example)

- **In the Java code:**
  - A method is marked with `@Test`
  - The method contains ordinary Java code
- **What Java does:**
  - Accepts the annotation syntactically
  - Stores the annotation as metadata in the bytecode
  - Executes no test logic itself
- **What JUnit does:**
  - Scans classes for methods marked with `@Test`
  - Executes these methods as tests
  - Reports green / red results

Essence: Annotations are metadata. It is the framework (JUnit) that gives them meaning.

JUnit: common assertions

- `assertEquals(expected, actual)`
- `assertTrue(condition)` / `assertFalse(condition)`
- `assertAll(...)` for grouping multiple checks

Tip: Failure messages improve if you include an optional comment:

```
assertEquals(4, result, "Expected 4 elements");
```

TDD: what does “minimal code” mean?

- When the test is red: write the simplest thing that could possibly work
- Do **not** implement everything up front
- Implement only enough to make the **next test pass**

- May feel “silly” at first
- Creates control and rhythm
- Produces testable code
- Often leads to better code design

Refactoring: when and why?

- Refactor **after** the tests are green
- Clean up:
  - duplicated code
  - poor naming
  - complex loops and conditionals
- Tests must remain green afterwards

Rule of thumb: "Make it work, then make it clean."

Joint exercise 2: FizzBuzz (problem)

Write a method which, for an integer N, returns:

- 'Fizz' if the number is divisible by 3
- 'Buzz' if the number is divisible by 5
- 'FizzBuzz' if the number is divisible by both 3 and 5
- otherwise the number itself as text, e.g. '7'

Goal: use TDD to develop the code, with red/green cycles.

FizzBuzz: suggested TDD progression (tests)

Start with the most trivial case, then extend:

1. $1 \rightarrow$ '1'
2. $2 \rightarrow$ '2'
3. $3 \rightarrow$ 'Fizz'
4. $5 \rightarrow$ 'Buzz'
5. $15 \rightarrow$ 'FizzBuzz'
6. (then: 6, 10, 30 to be certain)

Important: Write tests that introduce one new rule at a time, so each test fails for one clear reason.

TDD pitfalls (common beginner mistakes)

- Steps that are too large: **multiple rules in one test**
- Writing production code first “to save time”
- Tests become too **implementation-dependent** (instead of behaviour-focused)
- Too little refactoring: green, but messy
- “I trust that it works” $\Rightarrow$ rarely true in the long run

Good habits (that pay off quickly)

- Run tests frequently (after every small change)
- Good test names: **scenario + expectation**
- A test should verify **one thing**
- Distinguish between:
  - **Behaviour (what)**: what the function/class does from the outside (input $\rightarrow$ output)
  - **Implementation (how)**: how it is solved internally (algorithms, data structures, helper classes)
- Refactor regularly (when tests are green)

What do we actually test in TDD?

- In TDD we test **behaviour** — not implementation
- Tests describe:
  - what the system should do
  - which rules apply
  - which results we expect
- Tests should be robust against:
  - refactoring
  - algorithm changes
  - structural changes

Heuristic: If a test fails because you cleaned up the code, it is probably testing *how*, not *what*.

Leap Year: testing behaviour

Good test (behaviour-focused):

- Given a year as input
- Expect the correct result based on the rules
- 2019 → `false`
- 2020 → `true`
- 1900 → `false`
- 2000 → `true`

These tests:

- say nothing about *how* the code is written
- do not depend on `if`-statements, modulo operations, or order
- allow full refactoring as long as rules hold

Leap Year: example of *faulty thinking* in a test

The test:

```
@Test
void usesModuloFourToCheckLeapYear() {
// assumes the implementation uses year % 4
assertTrue(isLeapYear(2020));
}
```

What is the problem?

- The test itself is correct: 2020 should return `true`
- The test name and comment assume *how* the solution is implemented
- This thinking locks the implementation and violates TDD principles

Lesson:

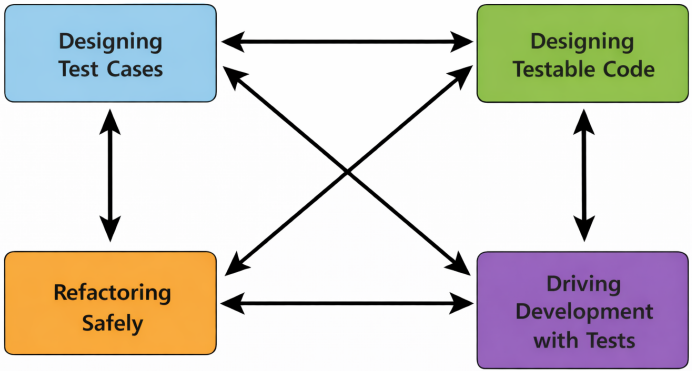
- Tests should express **behaviour**, not solutions
- If the name or comment says *how*, the test is misleading

When do tests become implementation-dependent?

- In pure and simple functions (Leap Year, FizzBuzz) it is difficult to go wrong
- Tests become problematic when they:
  - verify internal helper methods
  - assume specific objects or structures
  - test *how* instead of *what*
- This most often occurs when code:
  - collaborates with other objects
  - has side effects (I/O, logging, networking)

Conclusion: Leap Year and FizzBuzz teach TDD thinking — real challenges appear in larger systems.

Summary of TDD



Summary

- TDD = **test first** to guide development in small, safe steps
- Cycle: **Red** → **Green** → **Refactor**
- We practised on two classics:
  - Leap Year: rules with exceptions (100/400)
  - FizzBuzz: rules with combinations (3+5)
- JUnit provides the tools to express expectations and obtain fast feedback

Links

- cyber-dojo: https://cyber-dojo.org
- Using https://cyber-dojo.org in a commercial organisation requires a licence
- Non-commercial use is free, but donations are encouraged
- Kosli is a sponsor of cyber-dojo: https://www.kosli.com/
- Kosli's CEO, Mike Long, has previously been a guest lecturer on cyber-dojo at UiO
- JUnit 5 user guide: https://junit.org/junit5/docs/current/user-guide/