Unit testing

The problem with testing

Everyone agrees that more testing is needed, in the same way that everyone agrees you should eat your broccoli, stop smoking, get plenty of rest and exercise regularly.

The testing landscape

Testing methods

- Black-box testing
- White-box testing

Levels of testing

- Unit testing
- Integration testing
- System testing

Purpose of testing

- Security testing
- Performance testing
- Regression testing
- Etc.

Unit testing

- A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward.
- Done by programmers, for programmers.
- Make sure the code is doing what we think.

Integration testing

- Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.
- Occurs after unit testing and before system testing.
- Eventually all the modules making up a process are tested together.

System testing

- System testing of software (or hardware) is testing conducted on a complete, integrated system.
- Builds on the previous levels of testing namely unit testing and integration testing.
- Usually a dedicated testing team is responsible for doing system testing.

Unit testing

Vocabulary

- **System under test (SUT)**: the component that is being tested.
- **Test method**: method that implements the four phases (setup, exercise, verify, teardown) necessary to realize a fully automated test.
- **Test case**: class that collects test methods which are related in some way.
- **Fixture**: a.k.a. test context is a set of preconditions or state needed to run a test.
- **Assertion**: function that verifies the state or behavior of the SUT.
- **xUnit**: Generic name of any test automation framework for unit testing.

Definition

- Verifies that an individual unit of code (method, function) is working properly.
- The SUT is tested as an independent module. Dependent-on components need to be replaced with “test-specific equivalents” i.e. test doubles.

Unit test phases

1. **Setup** (fixture setup)
2. **Exercise** (exercise the SUT)
3. **Assert** (result verification)
4. **Teardown** (fixture teardown)

Why bother with unit testing?

- We assume that low level code works and move to the higher-level code.
  - Q: That's impossible!?
  - Q: I don't understand how that could happen!?
- You fix the low level problem, but that impacts code at higher levels, which then need fixing, and so on.
- In order to gain a legitimate code confidence, you'll need to ask the code itself what it is doing, and check that the result is what you expect it to be.
- Makes your design better and drastically reduces the time you spent debugging.

Unit testing objectives

- **Does it do what I want?** - You want the code to prove to you that it's doing exactly what you think it should.
- **Does it do what I want all of the time?** - You don't test a bridge by driving a single car over it right down the middle lane on a clear, calm day. That's not sufficient. Test for network failures, full disks…
- **Can I depend on it?** - Code that you can't depend on is useless. The only thing worse is code that you think you can depend on.
- **Does it document my intent?** - In effect, a unit test behaves as executable documentation.

Excuses

#1: It takes too much time to write the tests!

- How much time do you spend debugging code that you or others have written?
- How much time do you spend reworking code that you thought was working, but turned out to have major, crippling bugs?
- How much time do you spend isolating a reported bug to its source?

Other excuses

- Excuse: It takes too long to run the tests!

  Answer: Well It shouldn’t!
- Excuse: It's not my job to test my code!

  Answer: What is our job, exactly? Create working code.
- Excuse: I don't really know how the code is supposed to behave so I can't test it!

  Answer: Clarify the requirements.
- Excuse: I'm being paid to write code, not to write tests!

  Answer: We're not being paid to spend all day in the debugger, either. Unit tests are merely a tool toward working code.

Properties of good tests

Unit tests are very powerful magic, and if used badly can cause an enormous amount of damage to a project by **wasting your time**.

- **Automatic** - “Automatic” in invoking the tests and checking the results. You're not the only one running the tests. The test must determine for itself whether it passed or failed.
- **Thorough** - Test everything that's likely to break.
- **Repeatable** - Run the tests over and over again, in any order, and produce the same results.
- **Independent** - Tightly focused, and independent from the environment and each other.
- **Professional** - The code you write for a unit test is real. Expect that there will be at least as much test code written as there will be production code. Don't waste time testing aspects that won't help you.

Test-Driven Development (TDD)

Test-Driven Development mantra

1. Red - write a failing test to prove that the functionality is missing.
2. Green - make the test pass by writing production code.
3. **Refactor** - improve the implementation making sure that the test(s) always execute successfully.

Benefits of TDD

- Forces critical analysis and design.
- The software tends to be better designed, that is, loosely coupled and easily maintainable.
- The unit tests act as documentation that cannot go out-of-date.
- In TDD “I am done” means “I’m really done” e.g. solution is completely developed and tested.
- The test suite acts as a regression safety net on bugs.
- Reduced debugging time.

jUnit

Maven dependency

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>?</version>
  <scope>test</scope>
</dependency>
```

Maven project test class organization

- Each test class in the same package as the class it tests.
- Project structure:
  - Test classes under the “src/test/java” directory.
  - Test resources under the “src/test/resources” directory.

Test class naming conventions

1. Single class to be responsible for testing multiple methods of the class being tested.

   `<NameOfClassBeingTested>Test`

   E.g. `CalculatorTest`

2. Single class to be responsible for testing a single method of the class being tested.

   `<NameOfClassBeingTested><NameOfMethodBeingTested>Test`

   E.g. `CalculatorAddTest` and `CalculatorSubtractTest`

Demo: Calculator

Annotations

- **@Test**: Marks a method as a test method.
- **@Before**: Runs a setup method before each test method. Method needs to be public and void.
- **@After**: Runs a teardown method after each test method. Method needs to be public and void.
- **@BeforeClass**: Runs a setup method before any test method is run. Method needs to be public static and void.
- **@AfterClass**: Runs a teardown method after all test methods finish. Method needs to be public static and void.
- **@Ignore**: Ignores a test method.

State verification

Static methods in JUnit‘s org.junit.Assert class:

- assertEquals()
- assertTrue()
- assertFalse()
- assertNull()
- assertNotNull()
- assertSame()
- assertNotSame()
- fail(), etc.

Exercises

Exercise: TimePeriodOverlap

- Create a TimePeriod class that has a two fields: a start date and an end date, both of type java.time.LocalTime.
- Then, using test driven development, add an overlapsWith(TimePeriod) method to the class that takes as an argument another time period and checks whether the two overlap. Create a test method for each possible scenario.
- The following expression checks whether two periods overlap:

  (StartA < EndB) and (EndA > StartB)