Test-Driven Development in Practice

Maurício Aniche
That’s how we currently do!

We implement

We test
That’s how we currently do!

What do we miss if we do testing later? … VERY VERY late?
Can we test first?

We implement

We test
How?

• We think about a test.
• We write the test.
• We implement the code ...
• In the simplest way we can.
Let’s try!

• Roman Numerals
• Receives a string, converts to integer
  • "I" -> 1
  • "III" -> 3
  • "VI" -> 6
  • "IV" -> 4
  • "XVI" -> 16
  • "XIV" -> 14
Are you happy with this code?
Baby steps

• *Simplicity*: We should do the simplest implementation that solves the problem, start by the simplest possible test, ...

• Do not confuse *being simple* with *being innocent*.
  • Kent Beck states in his book: "Do these steps seem too small to you? Remember, TDD is not about taking teensy tiny steps, it’s about being able to take teensy tiny steps. Would I code day-to-day with steps this small? No. But when things get the least bit weird, I’m glad I can.”
Refactor!

• In many opportunities, we are so busy making the test pass that we forget about writing good code.

• After the test is green, you can refactor.
  • Good thing is that, after the refactoring, tests should still be green.

• Refactoring can be at low-level or high-level.
  • Low-level: rename variables, extract methods.
  • High-level: change the class design, class contracts.
Let’s do some refactor and continue!
The TDD cycle

1. Write a failing test
2. Failing test
3. Make it pass
4. Tests passing
5. Refactor

---

Image of Kent Beck
The TDD cycle

Write a failing test → Failing test → Make it pass → Tests passing → Refactor

What are the advantages?
Focus on the requirements

• Starting by the test means **starting by the requirements**.

• It makes us think more about:
  • what we expect from the class.
  • how the class should behave in specific cases.

• **We do not write ”useless code”**
  • Go to your codebase right now. How much code have you written that is never used in real world?
Controlling your pace

• Having a failing test, give us a clear focus: **make the test pass**.

• I can write whenever test I want:
  • If I feel insecure, I can write a simpler test.
  • If I feel safe, I can write a more complicated test.
  • If there’s something I do not understand, I can take a tiny baby step.
  • If I understand completely, I can take a larger step.
Test from the requirements

• Starting from the test means starting from the requirements. Meaning your tests derive from the requirements, and not from existing code.

• If you follow the idea of always having tests, you do not need to test afterwards.
  • Your code is tested already!
It’s our first client!

• The test code is the **first client** of the class you are constructing.
  • Use it to your advantage.

• What can you get from the client?
  • Is it hard to make use of your class?
  • Is it hard to build the class?
  • Is it hard to set up the class for use (pre conditions)?
  • Does the class return what I want?
Testable code

- TDD makes you think about tests from the beginning.
  - This means you will be enforced to write testable classes.
- We discussed it before: a testable class is also an easy-to-use class.
- Some people call TDD as *Test-Driven Design*. 
Tests as a *draft*

- Changing your class design is **cheaper** when done at the beginning.
- Use your tests as a *draft*: play with the class; if you don’t like the class design, change it.
  - Remember: the test is your first client.
Faster feedback

• You are writing tests frequently. This means you will find the problem sooner.
• Tests at the end also work. But maybe the feedback is just too late.
Controllability

• Tests make you think about managing dependencies from the beginning.
• If your class depends on too many classes, testing gets harder.
  • You should refactor.
Listen to your test

• The test may reveal design problems.
• You should "listen to it".

• Too many tests?
  • Maybe your class does too much.

• Too many mocks?
  • Maybe your class is too coupled.

• Complex set up before calling the desired behavior?
  • Maybe rethink the pre-conditions.
Is it really effective?

- 50% more tests, less time debugging [5].
- 40-50% less defects, no impact on productivity [6].
- 40-50% less defects in Microsoft and IBM products [12].
- Better use of OOP concepts [13].
- More cohesive, less coupled [15].

Is it?

• No difference in code quality [Erdogmus et al., Müller et al.]

• Siniaalto and Abrahamsson: The differences in the program code, between TDD and the iterative test-last development, were not as clear as expected.

Is it?

• "The practice of test-driven development does not drive directly the design, but gives them a safe space to think, the opportunity to refactor constantly, and subtle feedback given by unit tests, are responsible to improve the class design”.

• “The claimed benefits of TDD may not be due to its distinctive test-first dynamic, but rather due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow.”


Practical advice on TDD

• Keep a ”test list”.
• Refactor both production and test code.
• Always see the test failing.
• Stop and think.
TDD 100% of the time?

• No silver bullet! 😊

• Maurício: I do not use TDD 100% of the times. I let my experience tell me when I need it.
  • However, I **always** write tests and I **never** spend too much time only with production code.
Summary

• The TDD cycle is about writing a failing test, make it pass, refactor.
• TDD brings many advantages: focus on the requirements, rhythm, fast feedback, and testability thinking.
• Doing tests is more important than TDD.
• TDD is not a silver bullet.