Test After

TAD : Test After Development

(how I have worked so far)

Walter Moscatelli

EOC IT Architect

Working code

- Write code
- Debug code
- Working code.
- I need some tests (Sonarqube, Managment)
- Write tests for your code

The tests that you will write now will try to validate your code.

Sample case

- Write function calculateSalary
- You try to remember what your code need to work
- You add different input to verify your salary

Test implementation not behaviour

TIPS : Writing the tests first forces you to think about the different operating cases, you have to think about the borderline cases that would put the code in difficulty. This leads to more robust code

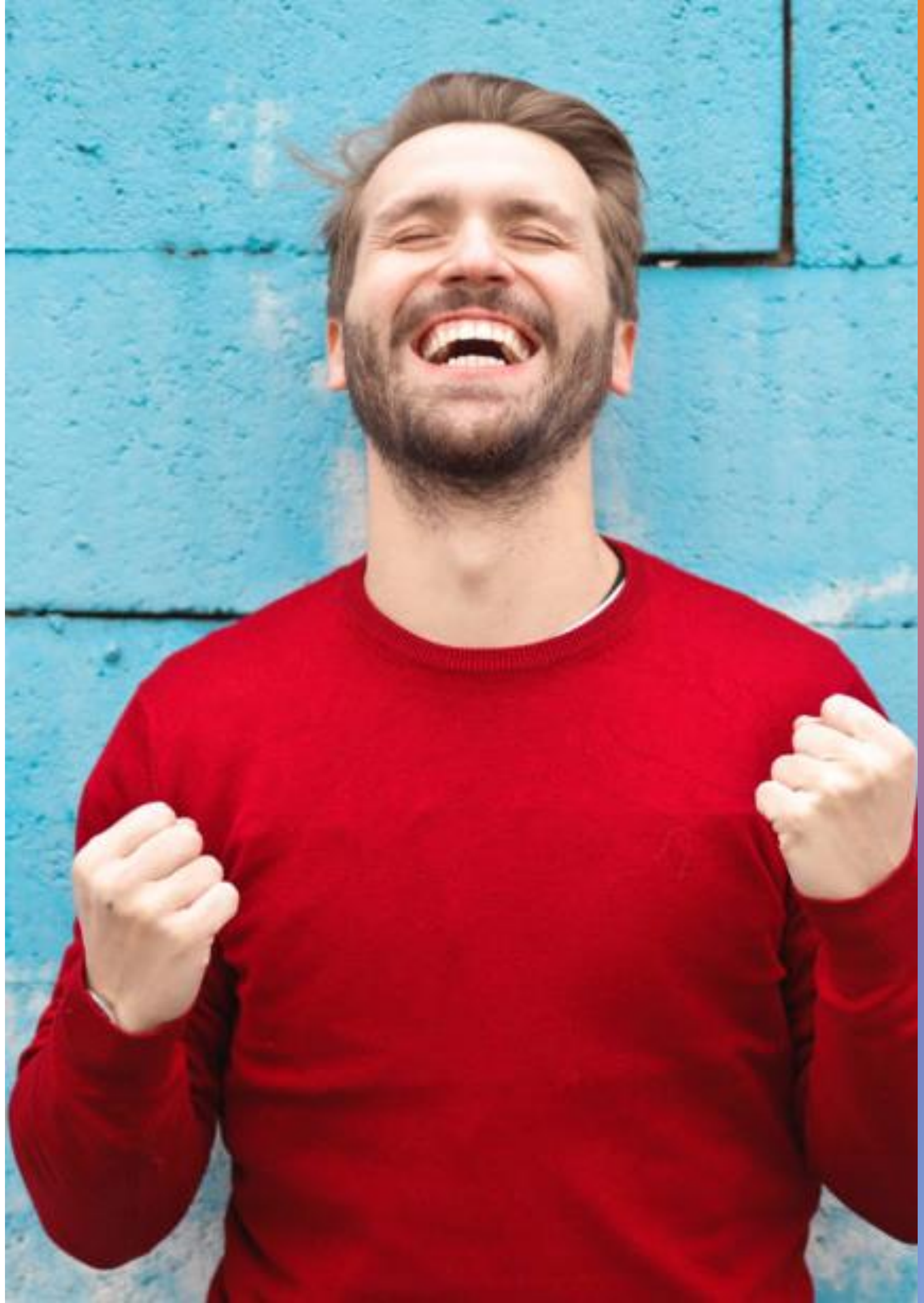
My code works

Writing tests is not an additional task.

It is an inseparable part of the development task for your feature.

But … if my code works I can forget to write code.

Sometimes developers will just write dummy tests which have no value, but somehow increase the code coverage, and that’s even worse than not writing the tests at all since it gives a false feeling that the code is well covered and protected, when it is actually not.



Why change ?

- I like my code
- Add complex test on complex code
- Difficulty to add test so I need to change my implementation

Why don’t developers use TDD in practice

Like everything that comes under the name of Agile, Test Driven Development (TDD) is something that **sounds great in theory**. In practice, it is unclear how to do it right. You are often told that if you don’t like it, you are doing it wrong. It comes as no surprise that most developers I’ve met could explain the benefits of using TDD while none of them used it in their work. Not a single one.

How could something so advantageous is so unwelcome to developers?

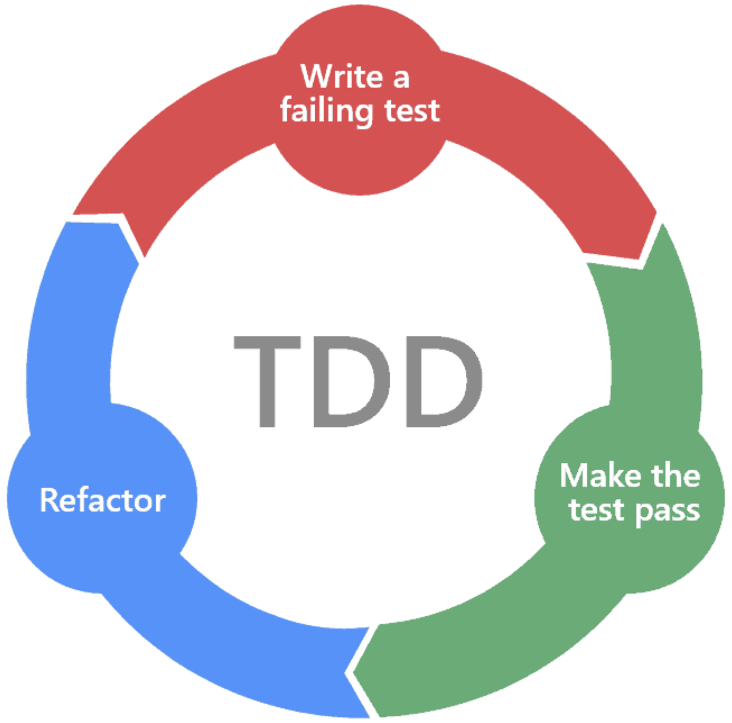
Writing more test code than implementation code

To test a “unit” of the implementation code, we often write tests for all public methods and write mocks for dependencies. Sometimes we make private methods public because otherwise there is no way to increase our code coverage. We create test cases to cover as many different flows of the implementation code as possible.

We end up being unproductive as we write more test code than the implementation code. Tests will not be released and delivered to users. It makes more sense to skip tests as it seems to speed up development

Red-Green-Refactor encourages writing bad code

- Write a test that fails, or doesn't even compile
- Write just enough implementation code to make the test pass
- Refactor the implementation code



This approach could be problematic, especially to senior developers, because this is what it really means in practice:

- Write a test that fails, or doesn’t even compile
- Write bad code to make the test pass, bad code that violates best practices
- Refactor the bad code and rewrite, not refactor, the tests
- This destroys our values as developers. It is almost a violation of programming ethics, illegal and unprofessional.

Code coverage measurement

It is an old saying, “What gets measured gets done.” If quality is measured by code coverage, developers will try every attempts to meet that minimum code coverage requirement. If we are not allowed to ship when code coverage is below 85%, we will end up adding more and more tests, usually those easiest to create, to make it just over 85% and no more. Ironically, most of these tests are trivial and without much value to ensure quality.

It shifts developer to focus on finding ways to create low-quality tests just to hit the minimum code coverage target.

Test everything

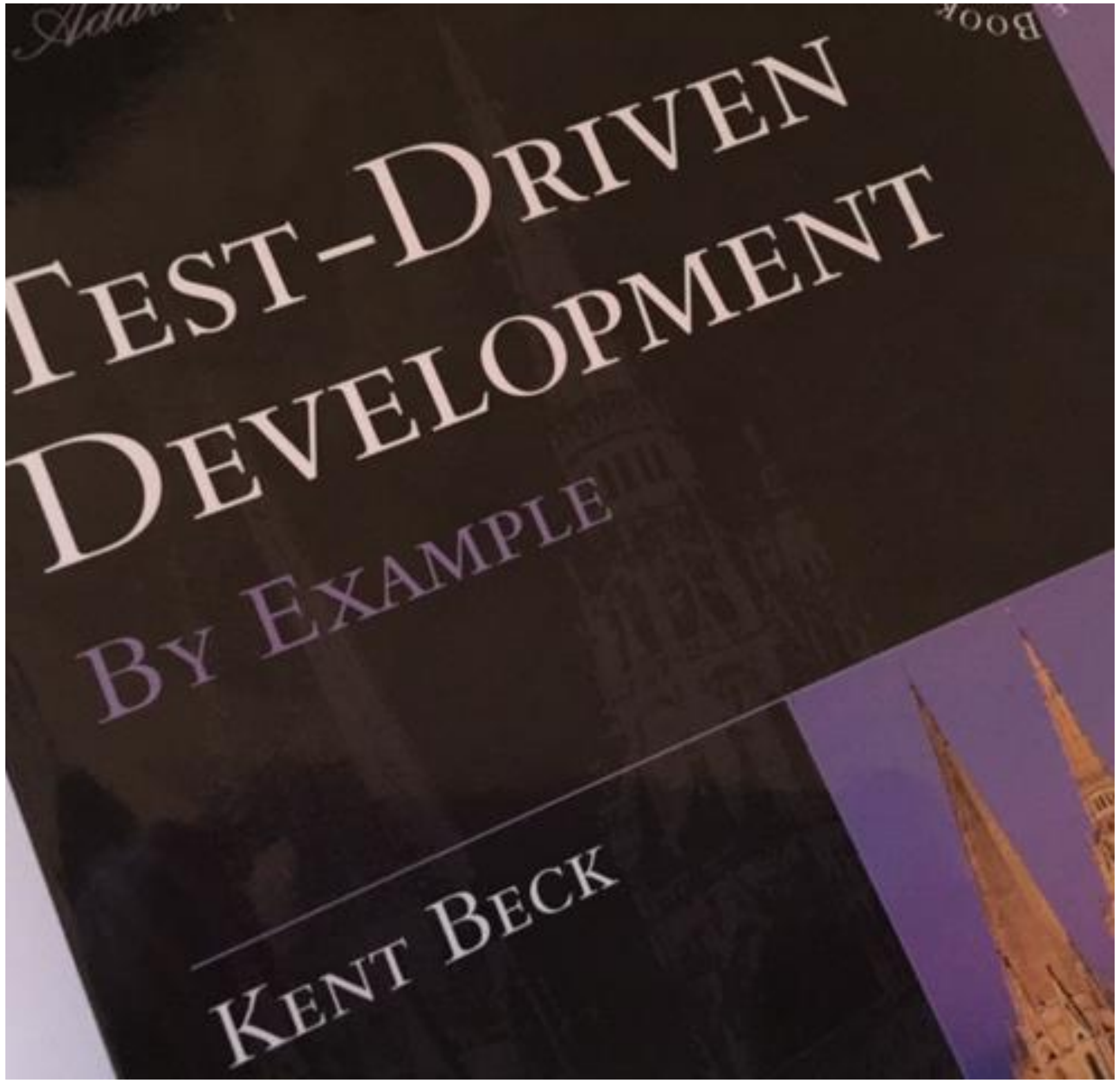
Developers tend to believe that they need to test every “unit” of their code — every public method. This means the following problems in such a TDD approach:

- More test code than the implementation code
- Not easy to design tests before the implementation is done
- Implementation refactoring breaks existing tests

Kent Beck explained in his book, Test Driven Development: By Example, that unit tests in TDD should test for behaviors, not implementations.

Developers often go too far trying to write tests for everything. Seeing that his idea has caused so much confusion and people started to complain about their pain using TDD, Kent Beck elaborated further how he would use unit tests for.

https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565



Unit tests in TDD should test for behaviors, not implementations

In other words, we should test the behavior of our program, or the "API" boundaries within our program. A "unit" usually refers to one meaningful behavior in our software design, not software implementation. This solves problems above because implementations change frequently during development but not behaviors.

Code coverage problem

For the last problem about code coverage could be solved easily if we understand its meaning behind — to help developers find untested code. Code coverage has nothing to do with code quality, which can be proven statistically. The percentage simply means nothing. The meaningful part of a coverage report is that it tells us what code is not yet tested and it could be buggy potentially. Again, Kent Beck would only test the code that might be buggy.

TDD as a habit

- Unit testing, and a lot of other Agile terminology, is like going to the gym. You know it is good for you, all the arguments make sense, so you start working out. You are so motivated initially, which is great, but after a few days of exercise, you start to rethink if it is worth the effort. You are spending an hour a day to change your clothes and run like a hamster. Yet you are not sure if you are really getting anything other than sore legs and arms.
- Then, after a week or two, just as the soreness is about to go away, a project deadline beings approaching. You need to spend every waking hour trying to get meaningful work done, so you cut out irrelevant stuff, like going to the gym. Now the deadline is over, but you fall out of the habit. If you manage to make it back to the gym at all, you feel just as sore as you were the first time you went.
- You do some reading and observe others, to see if you are doing something wrong. You being to feel a bit confused about why those happy people praising the virtues of exercise. You realize that you don't have a lot in common. They don't have to walk 15 minutes out of the way to the gym; there is one in their building. They don't have to argue with anybody about the benefits of exercise; it is just something everybody does. When a project deadline approaches, no one would tell them that exercise is unnecessary, just as your boss would not ask you to stop eating.

Conclusion

Many of the coding issues we experience on a daily basis can be avoided if we will practice TDD more. I’m not saying that the transition should be binary, this or that, but I hope that what I’ve written here will help you insist a bit more (even in that inner debate you’re having with yourself) on the quality which you would like to write your code in.