TDD process when you know the implementation of a new feature overlaps a previous one [closed]

Asked 3 months ago Modified 3 months ago Viewed 75 times

0

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 3 months ago.

Improve this question

I just solved this puzzle using TDD: https://adventofcode.com/2015/day/6

I started with turnOn function and implemented like this(F#):

```
let turnOn (startx, starty) (endx, endy) (grid:
bool[,]) =
    for x in startx..endx do
        for y in starty..endy do
            grid.[x, y] <- true
```

Then I started implementing the turnOff. I added the first test for it:

```
[<Fact>]
let ``can turn off a light`` () =
    let grid = Array2D.create 1000 1000 true

    do grid |> LightsController.turnOff (0, 0) (0,
0)

    grid
    |> Array2D.iteri (fun x y status ->
        let expected = not (x = 0 && y = 0)

        Assert.Equal(expected, status))
```

it forced me to define the turnOff function.

```
let turnOff (startx, starty) (endx, endy) (grid:
bool[,]) = grid.[0, 0] <- false
```

and then the next test:

```
[<Fact>]
let ``can turn off another light`` () =
    let grid = Array2D.create 1000 1000 true

    do grid |> LightsController.turnOff (1, 1) (1,
1)

    grid
    |> Array2D.iteri (fun x y status ->
        let expected = not (x = 1 && y = 1)

        Assert.Equal(expected, status))
```

at this point I see two options:

1. follow the normal TDD way and write tests one by one and evolve the production code until it just looks like the turnOn function, then refactor the two functions and extract the core algorithm
2. copy the turnOn code into the turnOff function and modify the related part (grid.[x, y] <- true) (because it is obvious that the two share the same algorithm) then copy all tests of turnOn for turnOff as well to make sure it works as expected. then refactor the two and extract the core algorithm

I prefer the second option because it's a lot faster and doesn't make me reinvent the wheel. but I feel something is wrong with copying and pasting!

please let me know what do you think about this situation?

tdd

Share

Improve this question

Follow

edited Sep 7 at 8:41

Mark Seemann

233k ● 49 ● 446 ● 769

asked Sep 6 at 13:39

Mehdi.Valizade

11 ● 3

2 Answers

Sorted by: Highest score (default)

0

> I prefer the second option because it's a lot faster. but I feel something is wrong in it! please let me know what do you think about this situation?

Very close.

The copy paste of the code you already have is *fine*; especially when that's the fastest way (measured in wall clock time) to get from RED state to GREEN state.

BUT...

> then copy all tests of turnOn for turnOff as well to make sure it works as expected. then refactor the two and extract the core algorithm

This is probably backwards. Cleaning up the duplication you introduced to pass the test should normally happen *before* you add more tests.

> Quick green excuses all sins. But only for a moment.... Now that the system is behaving..., remove the duplication that you have introduced. -- Kent Beck, Test Driven Development by Example.

The ideal that we are striving for is that we should have *clean* code that works before we introduce a new test.

Share Improve this answer Follow

answered Sep 6 at 14:01

VoiceOfUnreason
56.9k ● 5 ● 59 ● 102

I started using TDD after reading Clean Craftsmanship by Uncle Bob. In that book, the primary strategy emphasized was triangulation (only writing code when a test forces you to). However, Kent Beck, in his book Test-Driven Development By Example (which I'm currently reading), describes three strategies: 1) Fake it, 2) Use an obvious implementation, and 3) Triangulation. Now I realize that, in my case, I should be using the second strategy, and after the test passes, I should refactor — just as you mentioned.
– Mehdi.Valizade Sep 12 at 10:47

0

Someone's already voted to close this question as opinion-based, and that's probably not entirely unreasonable. Even so, I think it's possible to say something useful about this question. If it wasn't, we probably couldn't have the *tdd* tag on the site at all, as TDD is a software development *process*. There's always room for interpretation in such a process. After all, it's not an algorithm.

In general, people like Kent Beck, Dan North, Robert C. Martin and several others often point out some of the benefits of TDD as being about the *feedback* that you get from the interaction between the tests and the System Under Test (SUT). That you end up with a nice regression test suite is an added bonus, but is typically not the main motivation for engaging in the TDD process.

By writing a test before the SUT you learn something about the SUT. Sometimes, you learn about the API of the SUT. Should it be object-oriented? Functional? Should we work with primitive values or an abstraction? Which abstraction best addresses the problem (e.g. which API enables the simplest, most robust tests)?

Sometimes, you rather learn something about the implementation (e.g. how to evolve an algorithm that solves the problem at hand). In The Transformation Priority Premise Robert C. Martin describes how the order in which he adds test cases seems to impact which algorithm shakes out.

Sometimes it turns out that this kind of example-driven triangulation doesn't work well. In such cases, combining TDD with property-based testing may suggest a better approach. I wrote about such an example in 2015.

How does all of this apply to the question of how to test-drive a solution to that particular Advent of Code puzzle?

It applies in the following way: You need to first answer this question:

Why am I using TDD to solve this problem?

This is relevant, because my experience with Advent of Code is that TDD is, in general, not the best way to solve those puzzles. The problems are often so well-defined that solving the puzzle is more a question of figuring out how to correctly implement an appropriate algorithm. And granted, as the difficulty of the puzzles increase, it can be useful to write a few test cases to check that you've correctly handled all edge cases, etc.

Is that TDD, though? While I do add a few test cases when doing Advent of Code, I rarely follow the red-green-refactor cycle closely. *My* goal when doing those puzzles is to arrive at the correct answer.

This doesn't mean that *you* must not use TDD with Advent of Code. After all, no-one pays us to do those puzzles. We often do them to entertain ourselves, or to lean a new language, or a new technique.

If you want to use Advent of Code to hone your TDD skills, that's perfectly okay, but this also means that only you know why you're doing this, and what you hope to get out of it.

Many of the forces that are typically at work in 'real' programming include uncertainty and the desire to solve problems with code that remains readable and maintainable for a long time. This is why the TDD process includes a *refactoring* step. You're supposed to use the process to iterate towards a solution that has qualities like simplicity, readability, correctness, etc.

With Advent of Code, however, only correctness counts. Thus, if you want to be mercenary, if you can jump straight to an implementation that 'passes the gate', you're good.

Just for fun, I tried doing that puzzle, and quickly solved both part one or part two by copying and pasting your outline, and then hacking the rest together without adding any additional tests. So, to be clear, if all you care about is the solution, then you probably don't even need the tests.

Again, this doesn't mean that you're not allowed to use TDD with these puzzles. It does mean, however, that if you're doing that, it's probably because you want to learn something from the process. You haven't written what it is you hope to achieve from using TDD in this context, so we can't really guide as to which of your alternatives is the correct one.

It depends on what your goals are. Once you've made it clear to yourself what your goals are, you'll most likely also be able to answer the question yourself.

Share Improve this answer Follow

answered Sep 7 at 9:48

Mark Seemann
233k ● 49 ● 446 ● 769

Why am I using TDD to solve this problem? I’m experimenting with TDD in my free time. I treat the puzzles as if they are real-world problems and evaluate the resulting code in terms of simplicity, readability, correctness, flexibility, etc. Regarding this particular puzzle, I encountered a dilemma that likely occurs in real-world scenarios: 1.Let the tests guide me. 2.Follow my programming instincts and push the tests towards it. I prefer the second option, but the problem is that after copying the turnOn algorithm to turnOff, I don’t see any red lights in the tests anymore and that's a risk – Mehdi.Valizade Sep 8 at 6:20