From DDT to BDD

Roger Gordon

<2018-09-14 Fri>

Outline

Testing code is good!

Benefits of testing

As developers, we all know that writing tests is important.

- Prevents logic errors
- Helps with refactoring
- Helps with what to work on
- Helps to document the system
- Seems slow, but increases long term development speed

TDD is even better!

Benefits

- Ensures we don't write code we don't need
- Ensures we are sure about the requirements
- Necessarily ensures our test coverage is 100%

DDT

Is our coverage 100%?

If TDD ensures 100% code coverage, and we practice TDD, why don't we have 100% code coverage?

We don't really practice TDD

TDD is great in theory, but has a hidden assumption

- *You know exactly what you should do before you start working.*
- Questions about requirements often come up as we are developing.
- In this situation TDD is slow and frustrating, because you need to redo both your code and your tests multiple times.
- We end up practicing DDT (Development Driven Tests)

DDT becomes our standard

Habits

As developers we have a lot of stuff going on in our heads at any one time. As humans, we instinctively find ways to reduce the amount of mental work we need to do through habits. Once something works, we keep doing it unless there's a good reason to change.

- Once we start DDT, it becomes a habit.
- Even if we could use TDD in a certain case, we implement DDT out of habit.

DDT works!

We are doing well already

- Our code is generally robust
- Our test coverage is high
- We have a good amount of tests

DDT works (sometimes).

We guess about functionality

- We make guesses about how our code should work, based on how we understand the requirements.
- We sometimes get it wrong.
- If the requirements change, we need to remember to update our tests and code.

Our tests do not clearly document system behaviour

- Only developers can read our tests easily.
- Our tests make many implicit assumptions. The reason for these assumptions is not always clear.
- Eg:Example of hard to understand test

DDT works (sometimes) continued [2]...

Our coverage numbers are deceptive

- We may have a test that calls the code, but that doesn't mean it is a useful test.
- The assumptions behind a particular test may never be true.

We lose track of what we're doing

Have you ever been working on something and then something else higher priority got your attention? Do you always remember what you were doing previously? How about on Monday morning? How about after leave?

DDT works (sometimes) continued [3]. . .

Business and development are disconnected

- Developers speak code
- Business speaks requirements
- In general, Business can't read code to check it meets requirements
- Often, development doesn't understand requirements properly

We end up writing code that's not actually needed, and leave out bits that are required.

There is a better way!

Behaviour Driven Development (BDD)

History

- A spin off of TDD, developed in around 2009 by Dan North
- Resulted in the development of the first BDD testing software, JBehave, for Java

How it works

- Uses a plain text business language called "Gherkin" to define requirements. Example:Example feature file
- This language can be read by business, developers and software
- Software maps the language to automated tests. Example Example tests

BDD Benefits

BDD makes TDD practical. All the TDD benefits now become available to us. As a reminder, here they are again:

- Ensures we don't write code we don't need
- Ensures we are sure about the requirements
- Necessarily ensures our test coverage is 100%

Also, we now have a direct connection between the requirements and our code. This ensures that business and development stay more closely aligned.

BDD Fixes our DDT problems [1]

We no longer guess about functionality

- We know exactly what we should do because we have the requirements in front of us.
- Our framework ensures that we don't forget to implement any of the requirements.
- If the requirements change, our tests will fail and we'll know we need to fix them.

Our tests do not need to document system behaviour

- Our feature file does this for us instead
- However, our tests are smaller and are therefore easier to understand.
- Eg:Example of hard to understand test vs Short step test

BDD Fixes our DDT problems [2]

Our coverage numbers are no longer deceptive

- Our code reflects our tests and our coverage is 100% relevant to the requirements.

We never need to lose track of what we're doing again

- Simply run your tests and whatever doesn't pass is what you need to work on next.
- Interruptions no longer need to stress us out.

BDD Fixes our DDT problems [3]

Business and development are never disconnected

- Developers speak ~~code~~ specifications in feature files.
- Business speaks ~~requirements~~ specifications in feature files.
- Now we can understand each other unambiguously.

We end up writing code that's ~~not~~ actually needed, and never leave out bits that are required.

Example: calculator.feature Feature section

- Describes the feature
- There should be one of these per feature file.
- Does not map to any tests

```
Feature: Performing basic arithmetic operations

    In order to perform basic arithmetic
    As a person who is bad at maths
    I want to be able to provide 2 numbers
    and an operator and get a result back.
```

Example: calculator.feature Background section

- For repeated assumptions relevant to all your scenarios
- One per file
- Runs before each scenario

```
Background:

    Given that we will not use numbers greater
    than 4 digits
        And that we will enter the input in the
        order number1, number2, operator
    When we send a request to the calculator
    Then the response will start with "Result = "
        And the response will end with a number
```

Example: calculator.feature Scenario section

- Many per file encouraged eg: API endpoint feature file

```
Scenario: Adding 2 numbers

    Given we use the "add" operator for our operations
        And our first number is <number1>
        And our second number is <number2>
    When we call the calculator with our numbers and
    operator
    Then the calculator will return the value <result>

        | number1 | number2 | result |

        |    1    |    1    |   2    |
        |    1    |    2    |   3    |
        |    1    |    3    |   4    |
        |    1    |    6    |   4    |
```

Supporting Software

Java

- Original BDD framework was written for Java
- Most well known version is Cucumber

Python

- Lettuce
- Behave
- Morelia

Using Morelia

Installation

```
pip install morelia
```

Connecting our feature files to our tests

```
def test_basic_arithmetic(self):
    """Run tests for the requirements in calculator.feature """
    run('arithmetic/features/arithmetic.feature', self, verbose=True)
```