TDD As If You Meant It

Alex Bolboacă, 🐦 @alexboly, ✉ alex.bolboaca@mozaicworks.com

October 2017

Test Driven Development

TDD As If You Meant It

A Simple Example

Practising TDD As If You Meant It

Like Brewing Coffee

TDD is like brewing coffee

My Definition

A process for **designing software incrementally** by clarifying the problem before the solution, and allowing the solution to appear from the tension between tests and existing code.

Emergent design

The design is an **emergent property** because it appears from a combination of properties of **programmer, test code and production code**.

So, ...

When doing “pure” TDD, you shouldn’t start with any preconceived ideas about the solution

But...

Most of the time we fail at this rule

Goal

The goal is not to ignore your past experience.

The goal is to allow yourself to gather more experiences.

TDD As If You Meant It

What Is It?

A set of constraints that force the practise of “pure” TDD.

Constraints

You are not allowed to create any:

- named constant
- variable
- method
- class

other than by refactoring it out of a test.

Constraints

All production code is first written in tests and then extracted through refactoring

A Simple Example

We Need A Problem

Bank account kata

Think of your personal bank account experience. When in doubt, go for the simplest solution.

Requirements

- Deposit and Withdrawal
- Transfer
- Account statement (date, amount, balance)
- Statement printing
- Statement filters (just deposits, withdrawal, date)

Source: `https://github.com/sandromancuso/Bank-kata/`

Technology

I'll use **groovy** and **spock**

First test

```
def ”the first test”(){
    expect:
        false
}
```

Isn't this beautiful?

Let's take the time to properly admire the serenity and simplicity of this test ☺

First weird question

What is false?

False in programming

In **logic**, false is a **truth value** associated to an assertion.

In **programming** false is either **a constant** or the result of a **boolean operation**.

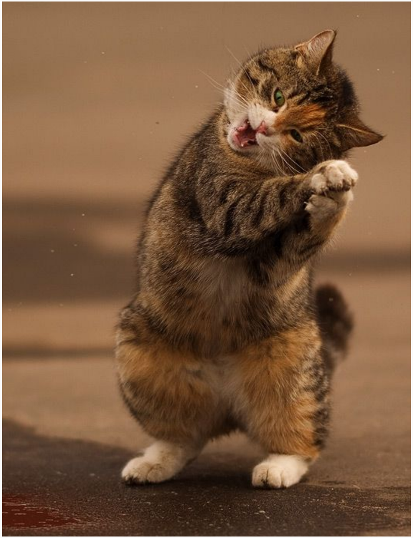
So what is false?

```
def ”after a deposit of €5 in an empty account \
        its balance is €5”{
    expect:
        424532534 == 5
}
```

Make the first test pass

```
def ”after a deposit of €5 in an empty account \
        its balance is €5”{
    expect:
        0 + 5 == 5
}
```

Celebrate!



Yay!

Refactor

Remember We’re only allowed to:

- extract constant
- extract variable
- remove duplication

The test needs more clarity

```
0 + 5 == 5
```

What is 5? What is 0?

- 0: initial account balance
- first 5: deposit amount
- second 5: final account balance

⚠ Accidental duplication!

Clarity

```
def ”after a deposit of €5 in an empty account \
    its balance is €5”{
  given:
    def expectedFinalBalance = 5
    def initialBalance = 0
    def amount = 5

  when:
    def finalBalance = initialBalance + amount

  then:
    finalBalance == expectedFinalBalance
}
```

Notions …

Let’s take a look at the notions we’re using:

- deposit amount
- empty account
- balance
- initial balance
- final balance

Semantics

Empty account = account that has an initial balance of 0

⚠ Duplication!

Express it!

```
def ”after depositing €5 in an account with balance 0 \
        its balance is €5”{
    given:
        def expectedFinalBalance = 5
        def initialBalance = 0
        def amount = 5

    when: // Production code
        def finalBalance = initialBalance + amount

    then:
        finalBalance == expectedFinalBalance
}
```

Next test?

Triangulation: the process of writing another test that changes **a single input** from the input set with the goal of advancing towards the solution

Triangulation - Advanced

We triangulate in the topology of possible solutions

What can we triangulate on?

- initial balance
- deposit amount
- currency (€, RON, £)
- account (eg. ownership, type of account etc.)

Decision

Let’s triangulate on deposit

What are interesting values for deposit?

Value sampling or **Equivalence partitioning**

Since we cannot test all the values, find the intervals that are interesting and pick at least one value from each interval

Equivalence partitioning for deposit

- Negative numbers: e.g. -€1, -€5
- Zero
- Positive numbers
- Minimum ?
- Maximum ?
- Non-integers: e.g. €1.50, €99.99

Questions, questions...

Can an account have negative balance? **Yes**

Can we deposit a negative amount? **No**

What’s the maximum amount in an account? **Use €1.000.000.000**

What’s the minimum amount in an account? **Use -€10.000**

What’s the minimum deposit? **Use €1**

What’s the maximum deposit? **Use €1.000.000.000**

So, triangulation on deposit

Accepted values:

- Minimum: €1
- Any value between €1 and €1.000.000.000
- Maximum: €1.000.000.000 (in an account with 0)

Invalid values:

- Less than minimum: €0.99
- More than maximum: €1.000.000.000,01

Second test

```
def ”after a deposit of €1 in an account with balance 0 \
        its balance is €1”{
    expect:
        false
}
```

What is false?

```
def ”after a deposit of €1 in an account with balance 0\
        its balance is €1”{
    expect:
        1 == 0 + 241234123423
}
```

Make it pass

```
def ”after a deposit of €1 in an account with balance 0\
        its balance is €1”{
    expect:
        1 == 0 + 1
}
```

Refactor to clarify

```
def ”after a deposit of €1 in an account with balance 0\
        its balance is €1”{
    given:
        def initialBalance = 0
        def amount = 1
        def expectedFinalBalance = 1

    when: //Production code
        def finalBalance = initialBalance + amount

    then:
        expectedFinalBalance == finalBalance
}
```

More refactoring?

Duplication between tests!

Rule of three

Rule of Three

⚠ Remove duplication after you've seen it for 3 or more times

So, next test

```
def ”after a deposit of €1.000.000.000 in an account\
        with balance 0 its balance is €1.000.000.000”{
    expect:
        false
}
```

5 MINUTES LATER ...

```
def ”after a deposit of €1.000.000.000 in an account\
        with balance 0 its balance is €1.000.000.000”{
    given:
        def initialBalance = 0
        def amount = 1000000000
        def expectedFinalBalance = 1000000000

    when: //Production code
        def finalBalance = initialBalance + amount

    then:
        expectedFinalBalance == finalBalance
}
```

Anything else to refactor?

Q: What is 1.000.000.000?

A: Maximum balance for an account in € is €1.000.000.000.

Extract max balance

```
static final maxBalance = 1000000000

def ”after a deposit of #maxBalance in an account\
        with balance 0 its balance is #maxBalance”{
    given:
        def initialBalance = 0
        def amount = maxBalance
        def expectedFinalBalance = maxBalance

    when: //Production code
        def finalBalance = initialBalance + amount

    then:
        expectedFinalBalance == finalBalance
}
```

How to Remove Duplication?

Two options:

- remove duplication from tests using data driven tests
- remove duplication from production code using a method

Data driven tests?

```
def ”after a deposit of #amount in an account\
        with balance #balance\
        its balance is #expectedFinalBalance”{

    when: ”deposit”
        def finalBalance = initialBalance + amount

    then:
        expectedFinalBalance == finalBalance

    where:
        initialBalance | amount     || expectedFinalBalance
        0              | 1          || 1
        0              | 5          || 5
        0              | maxBalance || maxBalance
```

Hmm...

⚠ Feels like it doesn't advance the solution

Spoiler: I'll come back to this later

Extract production code

```
def deposit(initialBalance, amount){
    return initialBalance + amount
}
```

Reflection time

How’s it going?

Let’s reflect

- We didn’t write a lot of code: three tests and one line of production code
- We learned a lot about the domain
- We learned a lot about the business rules
- We named things
- We have the simplest solution to the problem so far
- It feels slow

Most importantly

We can see how the code evolves: constant -> variable -> method

Let’s Look Into The Future: Deposit

```
static final maxBalance = 1000000000

def deposit(amount, initialBalance){
    if(amount < 1)
        return initialBalance
    if(initialBalance + amount > maxBalance)
        return initialBalance

    return initialBalance + amount
}
```

Let's Look Into The Future: Withdrawal

```
static final minBalance = -50000

def withdraw(amount, initialBalance){
    if(amount < 1)
        return initialBalance
    if(initialBalance - amount < minBalance)
        return initialBalance

    return initialBalance - amount
}
```

We have a design choice

Go functional or go Object Oriented

Functional solution

```
def balance = deposit(10,
               withdraw(200,
               withdraw(500,
               withdraw(10,
               deposit(1000, /*initialBalance*/ 0)))))
```

Object Oriented Solution

```
def account = new Account(balance: 0)
account.deposit(1000)
account.withdraw(10)
account.withdraw(500)
account.withdraw(200)
account.deposit(10)
def balance = account.balance
```

Different Object Oriented Solution

```
def account = new Account(balance: 0)
def transactions = [
    new DepositTransaction(amount: 1000),
    new WithdrawalTransaction(amount: 10),
    new WithdrawalTransaction(amount: 500),
    new WithdrawalTransaction(amount: 200),
    new DepositTransaction(amount: 10)
]
account.applyTransactions(transactions)
def balance = account.balance
```

What's the difference?

A class is nothing more than a set of **cohesive, partially applied pure functions**

– via JB Rainsberger

Partially applied function

```
account.deposit(amount)
```

is a partial application of:

```
deposit(amount, initialBalance)
```

And now for something completely different...

Pssst... Huge Monty Python fan here

Let’s rewind ...

```
def ”after a deposit of #deposit in an account\
        with balance 0\
        its balance is #deposit”{

    when: ”deposit”
        def finalBalance = initialBalance + amount

    then:
        expectedFinalBalance == finalBalance

    where:
        initialBalance | amount     || expectedFinalBalance
        0              | 1          || 1
        0              | 5          || 5
        0              | maxBalance || maxBalance
```

And reinterpret this

Whenever we deposit a **deposit amount** less or equal to the **maximum balance** into an account that has **initial balance** 0, the resulting balance is the **deposit amount**

⚠ Property based testing

Spock Genesis

spock-genesis library

`https://github.com/Bijnagte/spock-genesis`

A library that helps generating values for properties

A property based test

```
def ”after a deposit in an account\
        with balance 0\
        its balance is the deposit amount”{

    when: ”deposit”
        def finalBalance = deposit(amount,
                                   initialBalance)


    then:
        expectedFinalBalance == finalBalance

    where:
        amount << integer(1..maxBalance).take(500)
        initialBalance = 0
        expectedFinalBalance = amount
```

Reflection time

- We’ve learned more about software design
- We’ve learned more about testing
- We left our options open until we picked functional or OO approach
- We decided step by step what’s the right next step
- Still feels slow

Practising TDD As If You Meant It

How

- Pick any kata, or a pet project
- Set up a timer at 45’
- Do it every day for 10 days (continue where you left off)
- Reflect on the way the design evolves, and on the design options you took
- ⚠ Make every decision as deliberate as possible

Applying TDD As If You Meant It in Production

Do It Like Mythbusters

Mythbusters

How?

- Separate a small(ish) problem. Eg: special alerts in a web application, validation in client-side code, a UI control etc.

How?

- Separate a small(ish) problem. Eg: special alerts in a web application, validation in client-side code, a UI control etc.
- Start a new test suite (maybe even a new project)

How?

- Separate a small(ish) problem. Eg: special alerts in a web application, validation in client-side code, a UI control etc.
- Start a new test suite (maybe even a new project)
- Write the implementation using TDD as If You Meant It

How?

- Separate a small(ish) problem. Eg: special alerts in a web application, validation in client-side code, a UI control etc.
- Start a new test suite (maybe even a new project)
- Write the implementation using TDD as If You Meant It
- When you have the production code, integrate it with the rest of the code

Closing

Conclusions

TDD As If You Meant It has helped me become more deliberate about software design.

- Despite feeling slow, I found it is quite fast after practising

Conclusions

TDD As If You Meant It has helped me become more deliberate about software design.

- Despite feeling slow, I found it is quite fast after practising
- It helped me understand more about functional programming

Conclusions

TDD As If You Meant It has helped me become more deliberate about software design.

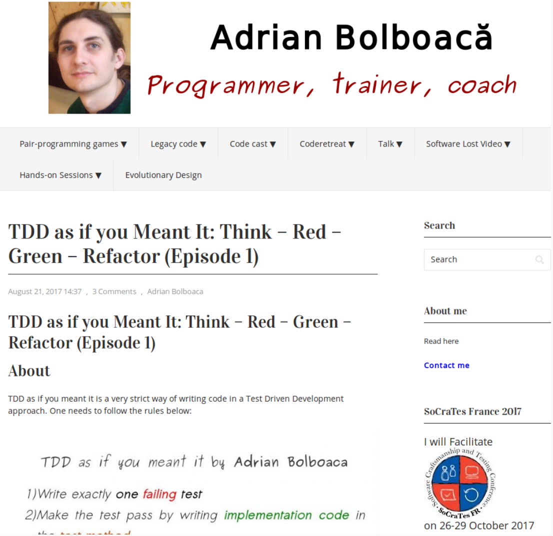
- Despite feeling slow, I found it is quite fast after practising
- It helped me understand more about functional programming
- The feeling of working on a small, clean, piece of code instead of the "big" code is very liberating

My advice

Embrace constraints

> *“I have never been forced to accept compromises but I have willingly accepted constraints”*
>
> Charles Eames, Designer and Architect

Learn more



TDD as if you Meant It: Think – Red – Green – Refactor (Episode 1)

August 21, 2017 14:37 , 3 Comments , Adrian Bolboaca

TDD as if you Meant It: Think – Red – Green – Refactor (Episode 1)

About

TDD as if you meant it is a very strict way of writing code in a Test Driven Development approach. One needs to follow the rules below:

TDD as if you meant it by Adrian Bolboaca

1)Write exactly one failing test

2)Make the test pass by writing implementation code in the test method

Adi’s Blog http://blog.adrianbolboaca.ro

Thank you!

I've been Alex Bolboacă, @alexboly, `alex.bolboaca@mozaicworks.com`

programmer, trainer, mentor, writer

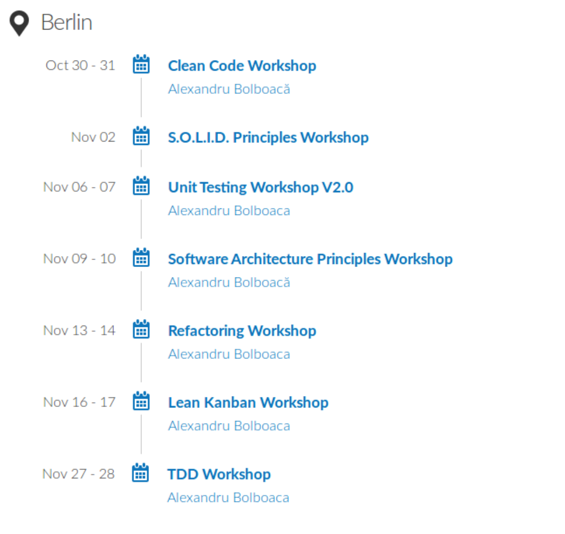
at Mozaic Works

Think. Design. Work Smart.

Mozaic Works
Think. Design. Work smart.

`https://mozaicworks.com`

Join me for Berlin workshops



Berlin

Oct 30 - 31	Clean Code Workshop	Alexandru Bolboacă
Nov 02	S.O.L.I.D. Principles Workshop	
Nov 06 - 07	Unit Testing Workshop V2.0	Alexandru Bolboaca
Nov 09 - 10	Software Architecture Principles Workshop	Alexandru Bolboacă
Nov 13 - 14	Refactoring Workshop	Alexandru Bolboaca
Nov 16 - 17	Lean Kanban Workshop	Alexandru Bolboaca
Nov 27 - 28	TDD Workshop	Alexandru Bolboaca

`https://mozaicworks.com/calendar/#Berlin`

Q&A

Q&A