

Introduction to Testing (part 1)

slide: <http://bit.do/pymalta-tdd>



Agenda

- Intro to Test Driven Development (TDD)
- TDD Flow
- F.I.R.S.T principles
- Mocking
- Interlude
- Practical

Test Driven Development (TDD)

What is TDD ?

A software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests.

Short history

Credited to Kent Beck for “rediscovering” TDD in the 2003, a prototype of TDD dates back to the early days of computing in the 1960s.



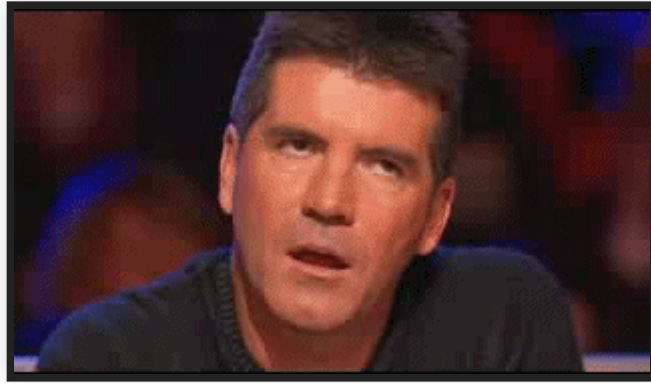
Short history

During the mainframe era, when programmers had limited time with the machine, a documented practice was to write the expected output before entering the punch cards into the computer.

Then you could immediately see whether the results you got from the mainframe were correct by comparing the actual output with the expected documented output.

Short history





Why write tests ?

- write tests, so that we can assert that our code runs correctly
- on every change, we re-run our tests to ensure nothing breaks
- particularly after some time passes , making the code easier to maintain

The TDD Flow

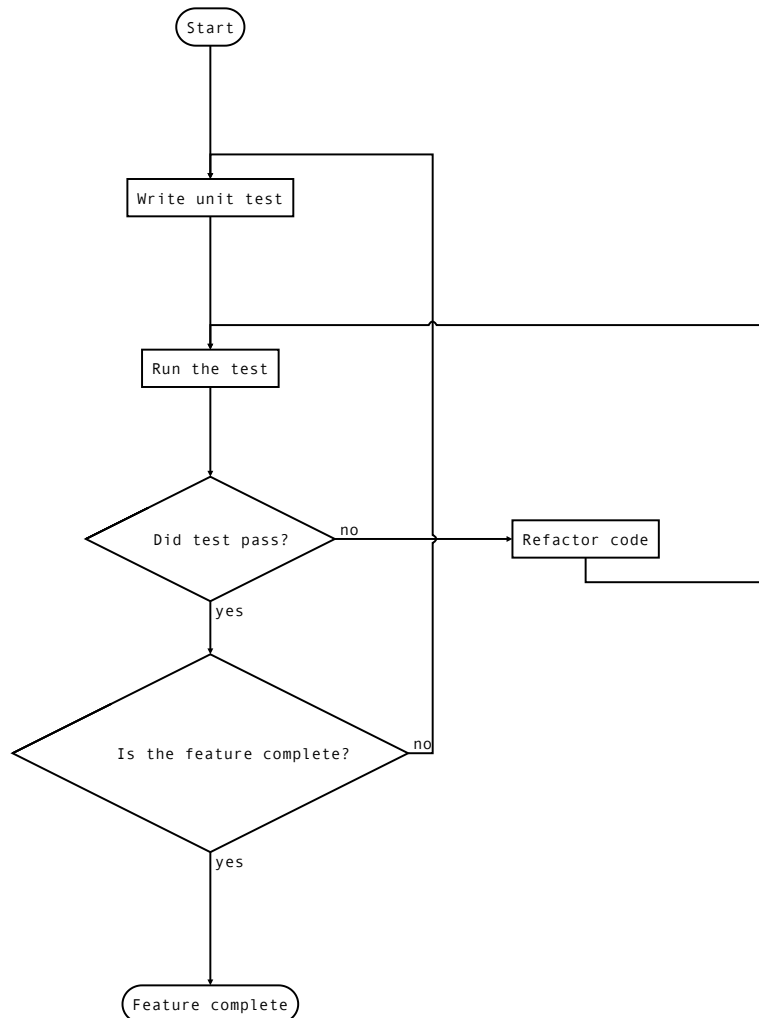
- write test, and run it to show unimplemented code fails
- fix the code and re-run test to see it pass
- rerun all tests & refactor adjacent code
- and then repeat above...

Test specifications

Feature requirements

- Implement a "simple number class"
- The number should have an "add" method
- The number should have an "multiply" method
- The number should raise an exception if initialized with invalid type

TDD Flow - diagram



```
def test_simple_number_add():  
    simple_number = SimpleNumber(2)  
    assert simple_number.add(2) == 4
```

```
def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
$ pytest test_simple_number.py
```

```
F
```

```
===== FAILURES =====
_____ test_simple_number_add _____
```

```
    def test_simple_number_add():
>         simple_number = SimpleNumber(2)
E         NameError: global name 'SimpleNumber' is not defined
```

```
test_simple_number.py:4: NameError
```

```
class SimpleNumber:
    pass

def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
class SimpleNumber:
    pass

def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
$ pytest test_simple_number.py
```

```
F                                                                    [ 100% ]
===== FAILURES =====
_____ test_simple_number_add _____

    def test_simple_number_add():
>         simple_number = SimpleNumber(2)
E         TypeError: this constructor takes no arguments

test_simple_number.py:6: TypeError
1 failed in 0.03 seconds
```



```
class SimpleNumber:
    def __init__(self, number):
        self.x = number

def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
class SimpleNumber:
    def __init__(self, number):
        self.x = number

def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
$ pytest test_simple_number.py -q
```

```
F
```

```
===== FAILURES =====
_____ test_simple_number_add _____
```

```
def test_simple_number_add():
    simple_number = SimpleNumber(2)
> assert simple_number.add(2) == 4
E     AttributeError: SimpleNumber instance has no attribute 'add'
```

```
test_simple_number.py:7: AttributeError
```

```
class SimpleNumber:
    def __init__(self, number):
        self.x = number

    def add(self, y):
        return self.x + y

def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```



```
class SimpleNumber:
    def __init__(self, number):
        self.x = number

    def add(self, y):
        return self.x + y

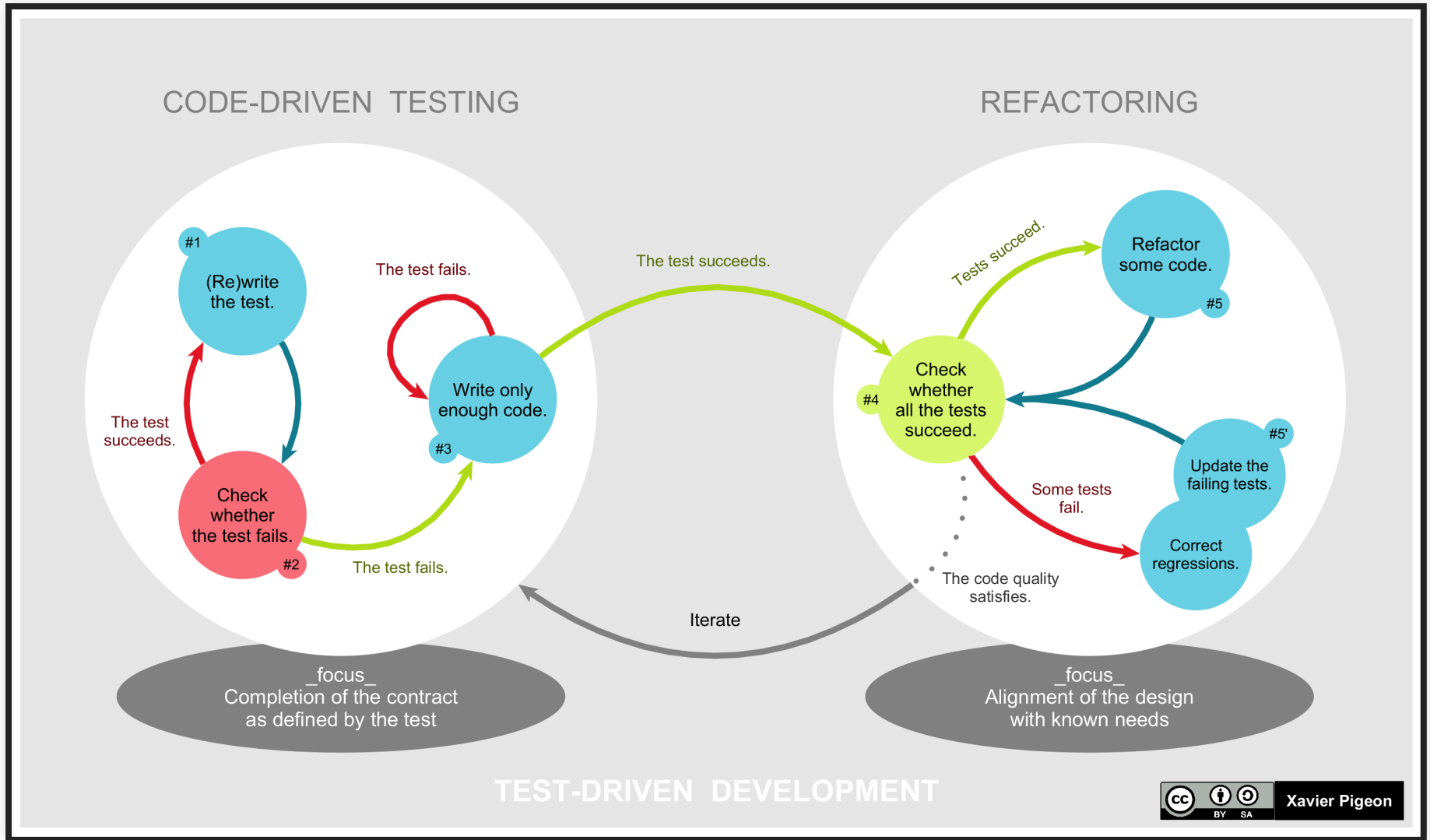
def test_simple_number_add():
    simple_number = SimpleNumber(2)
    assert simple_number.add(2) == 4
```

```
$ pytest test_simple_number.py -q
.
1 passed in 0.05 seconds
```

What's next ?

- add failing test for multiply method
- implement the multiply method until test passes
-  rerun all tests
- add failing test that expects an exception to be raised if class initialized with a value other than a number
- implement the type validation in class constructor
-  rerun all tests

TDD Flow - alt. diagram



F.I.R.S.T Principles

- Fast
- Independent
- Repeatable
- Self-Validating
- Timely

Fast

- Tests should run fast
- You should run tests frequently

Independent

- Tests should not depend on each other
- Order should not matter

Repeatable

- Environment should not matter
- Avoid flaky or brittle tests

Self-Validating

- Tests should either pass or fail
- Manual assertion should be avoided

Timely

- Tests should be written before the code

Mocking



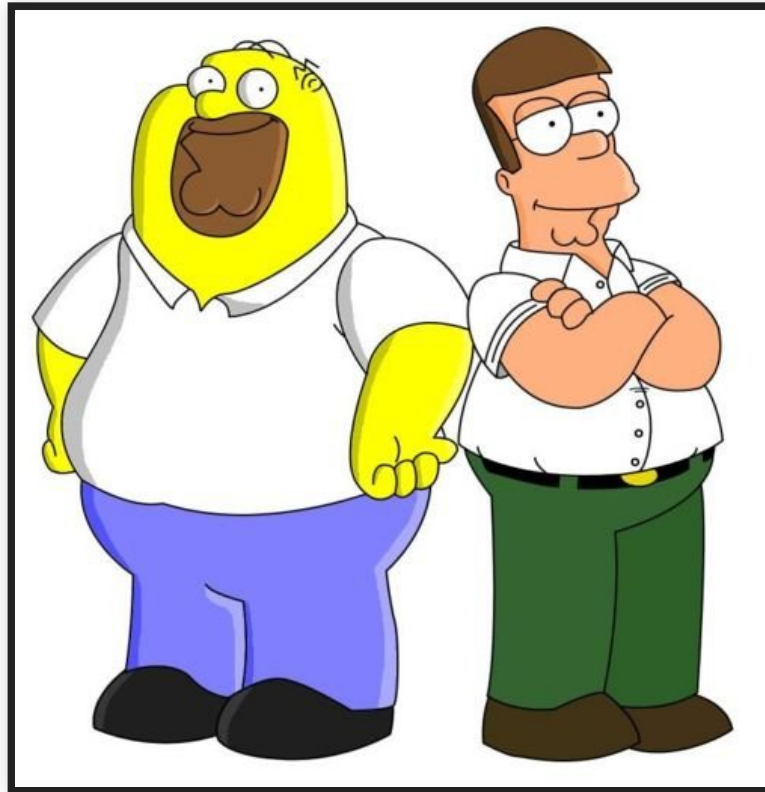
What is mocking ?

A tool to aid in isolating our tests.

To make them: FAST, INDEPENDENT &
REPEATABLE

How does one mock ?

Replace (or monkeypatch) code that is outside of the unit under test.



What should be mocked

- callables
- objects
- classes / interfaces
- global var / existing values

Basic example

Test specifications

Feature requirements

- Implement a "simple number class"
- The number should have an "add" method
- The number should have an "multiply" method
- The number should raise an exception if initialized with invalid type
- the result should be cached in a DB

The result should be saved to a DB for caching?

- Repeatable or flaky?
- Fast or slow?
- Independent or coupled?
- Self-validating or require manual assertion?

Mocking in python is easy with `mock.patch`:

```
from unittest.mock import patch

from simple_number import SimpleNumber

with patch('simple_number.DB') as mock_db:
    SimpleNumber(2).add(2)
    mock_db.assert_called()
```

Mock specific attribute with `mock.patch.object`:

```
from unittest.mock import patch

from simple_number import SimpleNumber, DB

@patch.object(DB, 'connect', return_value='OK')
def test_db_connect(mock_connect):
    simple_number = SimpleNumber()
    assert simple_number.connection_status == 'OK'
    mock_connect.assert_called_once_with(host='host', port='1234')
```

Python's MagicMock

A `mock.MagicMock` is a special class that is used by default to replace “patched” objects.

It works by replacing magic methods such as `__str__` and `__call__` and by default returns a new `mock.MagicMock` when called, making call chaining easy!

Python's MagicMock

- split into 2 fragments

A `mock.MagicMock` is a special class that is used by default to replace “patched” objects.

It works by patching magic methods such as `__str__` and `__call__` and by default returns a new `mock.MagicMock` when called, making chaining calls easy!

Chaining MagicMock objects:

```
from unittest.mock import patch

from simple_number import SimpleNumber

@patch('simple_number.DB')
def test_db_set(mock_db):
    simple_number = SimpleNumber(2)

    db_instance = mock_db.return_value # the return_value of calling mock_db
    db_instance.connect.assert_called_once_with(host='host', port='port')

    simple_number.add(2)
    db_instance.set.assert_called_once_with(key=2, value=2)
```


Testing is common

Let's take a break and look at some tests written in different languages.



Let's try it ourselves!

Cyber Dojo - go to

https://cyber-dojo.org/id_join/show?id=NUZNCm

or

<http://bit.do/pymalta-tdd-dojo>

or



Scoring Bowling

1	4	4	5	6		5			0	1	7		6			2	6
5		14		29		49		60		61		77		97		117	133

The game consists of 10 frames as shown above. In each frame the player has two opportunities to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.

A spare is when the player knocks down all 10 pins in two tries. The bonus for that frame is the number of pins knocked down by the next roll. So in frame 3 above, the score is 10 (the total number knocked down) plus a bonus of 5 (the number of pins knocked down on the next roll.)

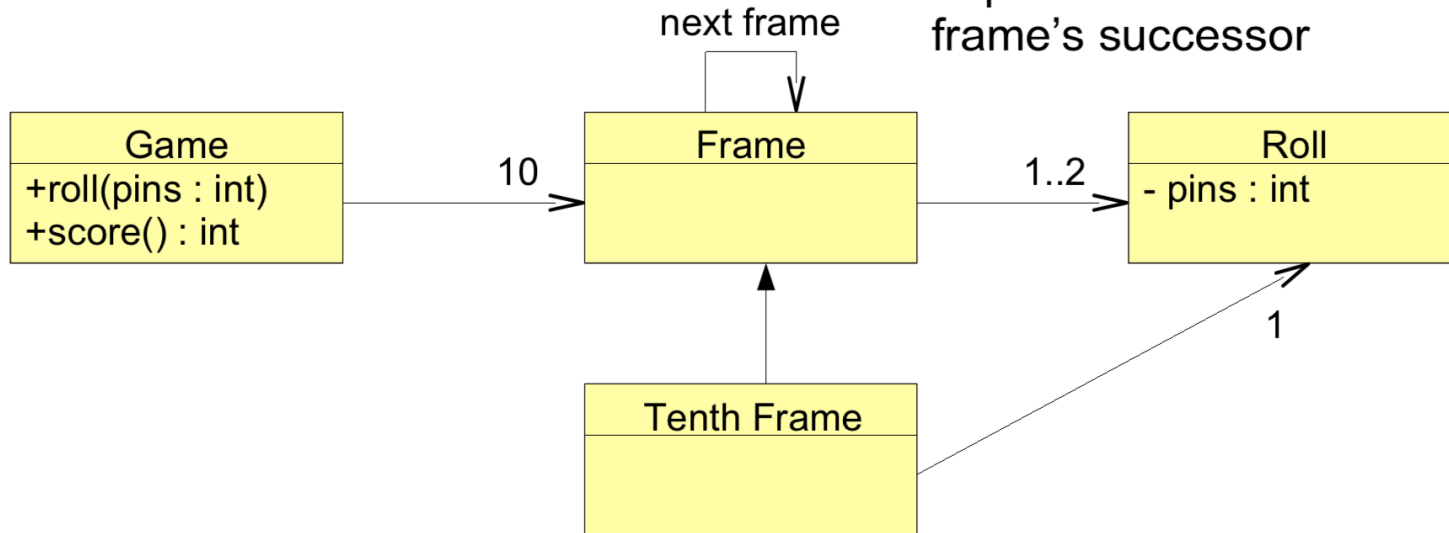
A strike is when the player knocks down all 10 pins on his first try. The bonus for that frame is the value of the next two balls rolled.

In the tenth frame a player who rolls a spare or strike is allowed to roll the extra balls to complete the frame. However no more than three balls can be rolled in tenth frame.

[source: Uncle Bob]

A quick design session

The score for a spare or a strike depends on the frame's successor



[source: Uncle Bob]

Thank you 🎉

We've reached the end, my friends! Can't wait to see the beautiful tests you will all write.

References