Test Driven Development

A Hypothesis - By George Jopson

Outline:

- TDD Goal
- Two Commandments of TDD
- Red – Green – Refactor
- Test to-do list (how to choose first item to work on, whatever is simple)
- How to approach TDD – 3 strategies
- How to write tests:
  - Decouple test and code
  - Given, when, then
  - Test Isolation
  - Assert First
  - Test Data
  - When to delete tests
- Working in gears
- Start Simple
- How to Check your doing it right
- Where not to TDD
- TDD Styles – Mocking:
  - Uncle Bob Mocking Style
  - Ian Cooper

Slides + Notes:

- Slides and notes can be founded at: https://github.com/GeorgeJopson/Test-Driven-Development-Notes

TDD Goal

- Write clean code that works.

Two Commandments of TDD

- Write new code only if an automated test has failed
- Eliminate Duplication

Red – Green – Refactor

- **Red:** Write a test that doesn't work (not working includes not compiling).
- **Green:** Make the test work
- **Refactor:** Eliminate duplication/mess.

Test To-Do List

- Maintain a to-do list of test cases you want to implement.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object

How to approach TDD – 3 strategies

- Fake It
- Use Obvious Implementation
- Triangulation

How to approach TDD: Fake It

- Get to green by returning constants
- Refactor by removing duplication

```
>>> def getCustomerLastName(firstname):
        return "Smith"

>>> assert getCustomerLastName("Sarah") == "Smith"
```

Use Obvious Implementation

- If there is an obvious solution, you can just implement it.

```
>>> def add(num1, num2):
        return num1 + num2

>>> assert add(1,2) == 3
```

Triangulation

- Adding more examples to your test can help generalise a problem

```
def multiply(num1, num2):
    ...

assert multiply(5,2) == 10
assert multiply(5,3) == 15
```

Tips for Writing Tests

Tips for Writing Tests - Decouple Test & Code

- Never test private attributes/methods
- Tests should test the behaviour of the public API of the code you are testing. They shouldn't test implementation details.

Tips for Writing Tests - Given, When, Then

- **Given:** Create some objects
- **When:** Perform some operations on the objects
- **Then:** Check the results are correct

Tips for Writing Tests - Test Isolation

- Test shouldn't impact each other
- Therefore, tests shouldn't share objects

Tips for Writing Tests - Assert First

- When writing tests, start by writing the asserts. Then you can work backwards to fill out the rest of the test.

Tips for Writing Tests - Test Data

- Use minimum amount of test data
- Never use same constant to mean different things in a test
- Try to make connection between numbers used in test data obvious

Tips for Writing Tests - When to Delete Tests

- They don't increase confidence
- They aren't communicating something new

Working in Gears

- TDD can change speed to suite your confidence level.

Start Simple

- Start with most trivial test case and work your way up

How to check you're doing it right – Measurements of Quality

- Statement Coverage
- Defect Insertion

How to check you're doing it right – Signs of Bad Tests/Bad Code

- Excessively long set up code
- Excessive Setup duplication
- Long running tests
- Fragile Tests

Where not to TDD

- TDD is most useful where it can provide fast binary feedback
- Places where it might be less useful:
  - Creating visual UI
  - Spikes/throwaway code
  - (Integration tasks connecting two systems)

TDD Styles

Note: In this section I talk about "modules". This can mean different things in different programming language. Here I use the term to mean a substantial grouping of code which deals with a certain responsibility in the system.

What both styles agree on

- Mock sparingly
- Trigger for adding new test is a new behaviour, not adding a new function/method to a class.
- Mock slow “shared fixtures” like databases or external HTTP APIs

"Uncle Bob" Style

- Your code should be divided up into modules, which each deal with separate concerns.
- **"Mock across significant architectural boundaries, but not within these boundaries."**
- So, you test these modules in isolation from each other by mocking the dependencies out from one module to another.
- You also mock slow running "shared fixtures" like: databases, external HTTP APIs, other external services.

“Uncle Bob” Style Benefits

- Tests run faster
- Tests are not sensitive to failures of mocked out components

"Uncle Bob" Style Disadvantages

- You have to pre-define how you are partitioning your problem into modules
- Tests understand implementation details

“Ian Cooper” Style

- We don’t test our modules in isolation.
- Test behaviour of the public API that is exposed to users
- A failure of a test tells implicates only the most recent edit.
- Only use of mock is the slow “shared fixtures”

“Ian Cooper” Style Pros

- Tests don’t constrain implementation
- Your design can be completely driven by tests

“Ian Cooper” Style Cons

- Slow tests with big systems
- Larger/more complicated test setup code/test

My Opinion

- Probably safe to use Ian Cooper's method for smaller systems.
- However, for very large industrial projects, "Uncle Bob's" approach may be more effective.

Any Questions ?

- Slides and notes can be founded at: https://github.com/GeorgeJopson/Test-Driven-Development-Notes