Test Driven Development (TDD)
(Unit testing and Refactoring)
What it is Test-Driven Development (TDD) 1?

• Test Driven Development (TDD) is a “Test-First” approach or method of creating software which means we are writing tests before writing code

• Uncle Bob describes TDD with three rules, but a refactored (clearer and shorter) version is
  • Write only enough of a unit test to fail
  • Write only enough production code to make the failing unit test pass

• It may seem unintuitive to write tests for code that doesn't exist yet, but by writing tests first, we ensure we have a clear expectation of what the code will do

• Watch: How TDD is related to the quality of code
  • https://www.youtube.com/watch?v=is41fgDrqn0
What it is Test-Driven Development (TDD) 2?

- Over the years TDD has proved itself as one of the best techniques for increasing the software quality.
- This way of development is reducing the number of bugs in your code and gives you the ability to reduce or even eliminate manual testing.
- Having a bug in production can be very expensive. Certain statistics say that finding a bug during development is 10 times cheaper than finding the same bug during QA (Quality Assurance) and 100 times cheaper than finding it in production.

![Cost to fix a defect table]

Source: https://en.wikipedia.org/wiki/Software_testing
What it is Test-Driven Development (TDD) 3?

• TDD comes from the eXtreme Programming (XP) world and have a widespread adoption. In its essence, it is consisting of small cycles of development.

• Every cycle have three steps (the TDD mantra), Red – Green – Refactor
  1. During the Red phase, we write a unit test for functionality that doesn’t yet exist and that will fail.
  2. Green phase refers to the moment we implement the minimum amount of code functionality to pass that test.
  3. And finally we refactor our code.

• Then we repeat the whole process for next functionality.

• One of the cool things with TDD is that this way refactoring has a safety net, and we can do it stress-free!
Benefits of Test-Driven Development (TDD)

- In general, by using Test Driven Development we avoid creating overcomplicated designs and overengineered systems. This is one of the biggest benefits of TDD. That’s why people also refer to this approach as Test Driven Design.
- It forces the developer to design classes properly and to follow principles like:
  - KISS – keep it simple, stupid
  - YAGNI – You aren’t gonna need it
    - https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it
  - DRY – Don’t repeat yourself
    - https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
  - SOLID
SOLID

- **Single responsibility principle**
  - A class should have only a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

- **Open – closed principle**
  - "Software entities ... should be open for extension, but closed for modification."

- **Liskov substitution principle**
  - "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

- **Interface segregation principle**
  - "Many client-specific interfaces are better than one general-purpose interface."

- **Dependency inversion principle**
  - One should "depend upon abstractions, [not] concretions."
Benefits of Test-Driven Development (TDD) 2

- Before writing code, writing unit tests forces you to detail your requirements in a useful fashion.
- While writing code, unit tests keep you from over-coding. When all the test cases pass, the function is complete.
- When refactoring code, they can help prove that the new version behaves the same way as the old version.
- When maintaining code, having tests will help you cover your ass when someone comes screaming that your latest change broke their old code. (“But sir, all the unit tests passed when I checked it in...”)
- When writing code in a team, having a comprehensive test suite dramatically decreases the chances that your code will break someone else’s code, because you can run their unit tests first
  - As a team breaks up the assignment, everybody takes the specs for their task, writes unit tests for it, then shares their unit tests with the rest of the team. That way, nobody goes off too far into developing code that doesn’t play well with others
Refactor phase 1

• In the refactor phase, you are allowed to change the code, while keeping all tests green, so that it becomes better. What’s “better” means is up to you!
  • But there is something mandatory: you have to **remove code duplication**! Kent Beck suggests in his book ”Test Driven Development: By Example” that removing code duplication is all you need to do

• In this phase you play the part of a picky programmer who wants to fix/refactor the code to bring it to a professional level. In the green phase, you’re showing off your skills to your users. But in the refactor phase, you’re showing off your skills to the programmers who will read your implementation

• Removing code duplication often results in abstraction. A typical example is when you move two pieces of similar code into a helper class that works for both the functions/classes where the code has been removed
Refactor phase 2

• Left code example below could be refactored into right hand side code
• Just keep in mind that you cannot move to another test unless you have removed all the code duplication

```java
public class Promise {
    public Promise(/* code */) {
        /* code */
    }
}

public class Hello {
    public Promise Greet() {
        return new Promise(setTimeout('Hello', 100));
    }
}

public class Random {
    public Promise Toss() {
        return new Promise(setTimeout(Math.Random(), 200));
    }
}

public class PromiseHelper {
    static Promise Timeout(int delay) {
        return new Promise(setTimeout(delay));
    }
}

const var logResult = result => console.log(result);
new Hello().Greet().then(logResult);
new Random().Toss().then(logResult);
```

```java
public class Hello {
    public PromiseHelper Greet() {
        return PromiseHelper.Timeout(100).then(() => 'hello');
    }
}

public class Random {
    public PromiseHelper Toss() {
        return PromiseHelper.Timeout(200).then(() => Math.Random());
    }
}

public class PromiseHelper {
    static Promise Timeout(int delay) {
        return new Promise(setTimeout(delay));
    }
}

const var logResult = result => console.log(result);
new Hello().Greet().then(logResult);
new Random().Toss().then(logResult);
```
TDD code example 1

- User stories we need to implement (steps in 1.1-1.2, 2.1-2.2, ...)
  - A user is able to assign Parents and Kids to a house number
  - A user is able to add residents to the House
  - A user is able to turn Parents with assigned Kids to pickles
- User story → Test scenario → Test case

```python
import unittest
from family import Parent, Kid, House

# 1.1
class ParentTests(unittest.TestCase):
    def test_location_returns_equal(self):
        parent = Parent(68)
        self.assertEqual(parent.location, 68)

# 2.1
class KidTests(unittest.TestCase):
    def test_location_returns_equal(self):
        kid = Kid(68)
        self.assertEqual(kid.location, 68)

# 3.1
class HouseTests(unittest.TestCase):
    def test_get_all_residents_returns_count_equal(self):
        house = House()
        residents = house.get_all_residents()
        self.assertCountEqual(residents, [])

if __name__ == '__main__':
    unittest.main()
```

```python
import pickle

# 1.2
class Parent(object):
    def __init__(self, loc):
        self.location = loc

# 2.2
class Kid(object):
    def __init__(self, loc):
        self.location = loc

# 3.2
class House(object):
    def __init__(self):
        self.__residents__ = []  # private member

    def get_all_residents(self):
        return self.__residents__
```

family.py

test_family.py
test_family.py

class HouseTests(unittest.TestCase):
# 4.1
    def test_add_resident_returns_equal(self):
        house = House()
        mum = Parent(68)
        kid1 = Kid(68)
        house.add_resident(mum)
        house.add_resident(kid1)
        residents = house.get_all_residents()
        self.assertEqual(residents[0], mum)
        self.assertEqual(residents[1], kid1)

class KidTests(unittest.TestCase):
# 5.1
    def test_is_assigned_returns_false(self):
        kid = Kid(68)
        self.assertFalse(kid.is_assigned)

class ParentTests(unittest.TestCase):
# 6.1
    def test_has_kid_returns_equal(self):
        parent = Parent(68)
        self.assertEqual(parent.kid, None)
# 7.1
    def test_assign_kid_returns_equal(self):
        parent = Parent(68)
        kid = Kid(68)
        parent.assign(kid)
        self.assertEqual(parent.kid, kid)
        self.assertTrue(kid.is_assigned)

class House(object):
# 4.2
    def add_resident(self, resident):
        self.__residents__.append(resident)

class Kid(object):
# 5.2
    def __init__(self, loc):
        self.is_assigned = False

class Parent(object):
# 6.2
    def __init__(self, loc):
        self.kid = None
# 7.2
    def assign(self, kid):
        self.kid = kid
        kid.is_assigned = True

# step 8, 9 and 10 in the python files
https://rubikscode.net/2019/03/04/test-driven-development-tdd-with-python/
**Behavior Driven Development (BDD)**

- The two main practices of BDD are
  - Specification by Example (SbE) and Test Driven Development (TDD)

- BDD focuses on
  - Providing a shared process and shared tools promoting communication to the software developers, business analysts and stakeholders to collaborate on software development, with the aim of delivering product with business value
  - What a system should do and not on how it should be implemented
  - Providing better readability and visibility
  - Verifying not only the working of the software but also that it meets the customer’s expectations

- Specification by Example (SbE)
  - SbE uses examples in conversations to illustrate the business rules and the behavior of the software to be built and thereby enables the product owners, business analysts, testers and the developers to eliminate common misunderstandings about the business requirements
Recommended TDD viewing and reading

- Test Driven Development: what it is, and what it is not
  - https://medium.freecodecamp.org/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2
- Getting Started with Test Driven Development
  - https://dev.to/chasestevens/getting-started-with-test-driven-development-1o8f
- BDD, TDD and ATTD
- Introduction to Test Driven Development (TDD)
  - https://medium.com/@ssaurel/introduction-to-test-driven-development-tdd-61a13bc92d92
  - http://www.agiledata.org/essays/tdd.html
- Test Driven Development (TDD) with Python
  - https://rubikscode.net/2019/03/04/test-driven-development-tdd-with-python/
- Uncle Bob on TDD, How TDD is related to the quality of code, The Three Laws of TDD (Featuring Kotlin)
  - https://www.youtube.com/watch?v=GvAzrC6-spQ
  - https://www.youtube.com/watch?v=is41fgDrqn0
  - https://www.youtube.com/watch?v=qkbIc5WRn-U