CS 362 - Software Engineering II
Test-Driven Development (TDD)

Previously we learned the following concepts within software testing - Unit testing, Black-box testing, Random testing, and White-box testing. In the in-class activity last week, we tried writing tests before implementing the code.

Test-driven development

"Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before the software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is opposed to software being developed first and test cases created later."

-Wikipedia

What we know

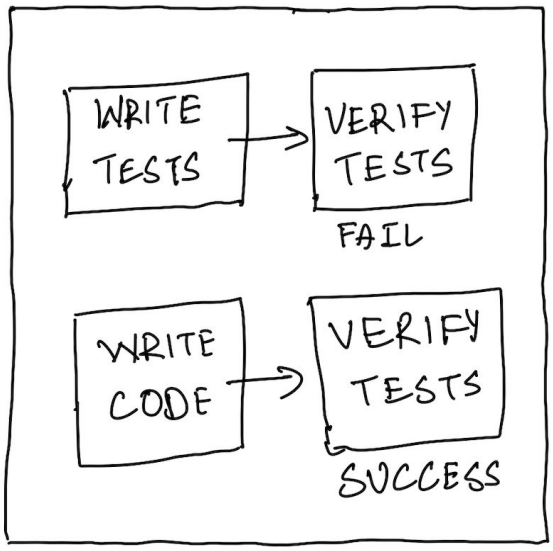
1. *Software development process*
2. *Software requirements*
3. *Test cases*

What's new?

1. Writing tests before code.
2. Repeatedly testing software against all test cases.

Overall, Test-Driven Development is not a testing technique or a Verification and Validation process, rather it is a way of *implementation*.

A general outline of TDD



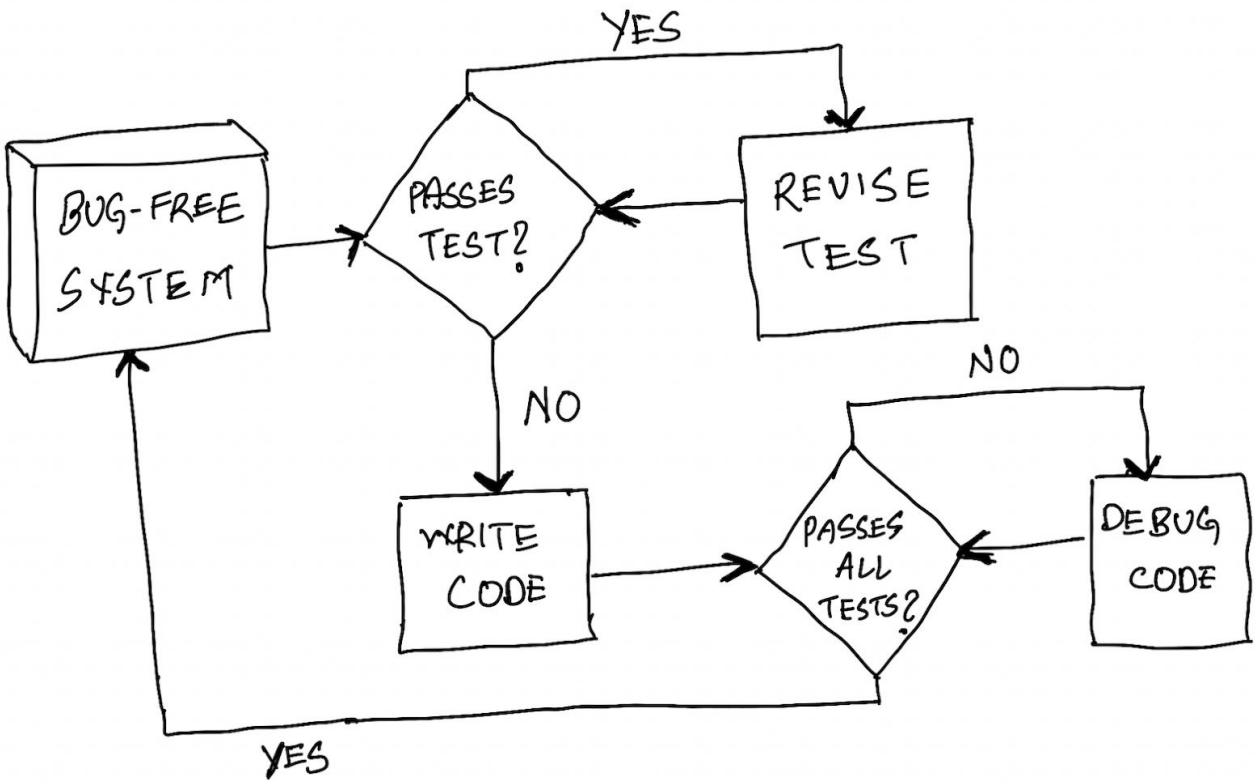
Essentially in TDD, we are writing tests, verify they fail, then write code and verify that these tests succeed.

TDD steps

The above diagram is a general representation of the following:

1. Add a test case that fails, but would succeed with the new feature implemented.
2. Run all tests, make sure only the new test fails.
3. Write code to implement the new feature.
4. Rerun all tests, making sure the new test succeeds (and no others break).

TDD Procedure



Benefits of TDD

1. Results in lots of useful test cases.
    i) A very large regression set.

2. Forces attention to the actual behavior of software: observable & controllable behavior.

3. Only write code as needed to pass tests.

4. May get good coverage of paths through the program since they are written to pass the tests.

5. Reduces temptation to tailor tests to idiosyncratic behaviors of implementation.

6. Testing is a first-class activity in this kind of development.

TDD, Black-Box, and White-box testing

If you remember, during our in-class activity we tried to write tests based on a program specification without access to the source code. **(Fibonacci sequence)**.

By now you might have realized that TDD is a type of black-box testing. If you think about TDD in terms of white box testing, remember that you do not have code to test for coverage.

TDD with an example

Let's try to understand the TDD approach with an example. Let's write tests for a program that prints the classic "Hello World" message.

Step 1

Create a file that contains your tests. Let our file be hello_world_test.py and it contains the following test.

hello_world_test.py > ...

```python
import unittest
import code

class TestCase(unittest.TestCase):
    def test1(self):
        self.assertEqual(code, 'Hello World!')

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

The code that we want to test is in the code.py file.
For now, let's write in the following lines of code in the code.py file.

The code in the code.py file creates a function but doesn't return anything. We use the pass statement in the hello_world method to avoid getting an error.

The pass statement is used as a placeholder for future code. When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed.

Step 2

Let's run the file that contains our tests. You should be getting an output that looks like -

We know from this output that our test has failed.

By now we know what to do next based on the steps of TDD. We need to write code that passes our test. This is what we will do in step 3.

Step 3:

Let’s apply changes to the method so that our tests can pass.

```
code.py > ...
1    def hello_world():
2        return 'Hello World!'
```

Our tests would still look like this:

```
hello_world_test.py > ...
1    import unittest
2    import code
3
4    class TestCase(unittest.TestCase):
5      def test1(self):
6        self.assertEqual(code.hello_world(), 'Hello World!')
7
8    if __name__ == '__main__':
9      unittest.main()
```

When we run our test, we get an output that looks something like this:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

What we did until now was write the bare minimum amount of code that can pass our tests. Essentially in this example, we wrote a test, verify that it fails, then wrote code and verified that the test passed.

TDD with another example

If you can recall we used the addition example for learning unit testing a couple of lectures ago. Let's try the TDD approach in this example.

Addition - A method that takes two numbers and returns their sum.

Let's try the steps similar to the previous example:

1. Write your tests in a file.

Your tests would look something like this:

```
Addition_Tdd.py > ...
1   import unittest
2   import addition
3
4   class TestCase(unittest.TestCase):
5     def test_add(self):
6       self.assertEqual(addition.addition(1,2), 3)
7
8   if __name__ == '__main__':
9     unittest.main()
```

2. Run the tests: You should see something similar to the image below.

```
F
======================================================================
FAIL: test_add (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "Addition_Tdd.py", line 6, in test_add
    self.assertEqual(addition.addition(1,2), 3)
AssertionError: None != 3

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

3. Write code that makes your tests pass.

addition.py > ...

```
def addition(a, b):
    return a + b
```

Re-run the tests to see if they pass or not.

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

By now we've understood that TDD is an approach to implementation. We have seen two different examples and we applied the steps of TDD.

Challenges of TDD

1. Lots of test cases may create false confidence.
2. If developers have written all tests, there may be blind spots due to false assumptions made in coding and in testing, which are tightly coupled.
3. Management may wonder why so much time is spent writing tests, not code.
4. Waterfall and spiral typically put testing after implementation, taking time to test slows agile.

Test-Driven Development: Business / Organization perspective

Behavior-driven development - BDD

BDD is an extension of TDD. You can assume BDD to be a version of TDD with the following modifications -

- Understand the business domain's terminology.
- Specify the desired behavior of the system in business language.
- Use those specifications to annotate unit tests.
- Implement a system to satisfy unit tests.

- Test harness tools run the tests, report failures.

Test-Driven Development - Summary

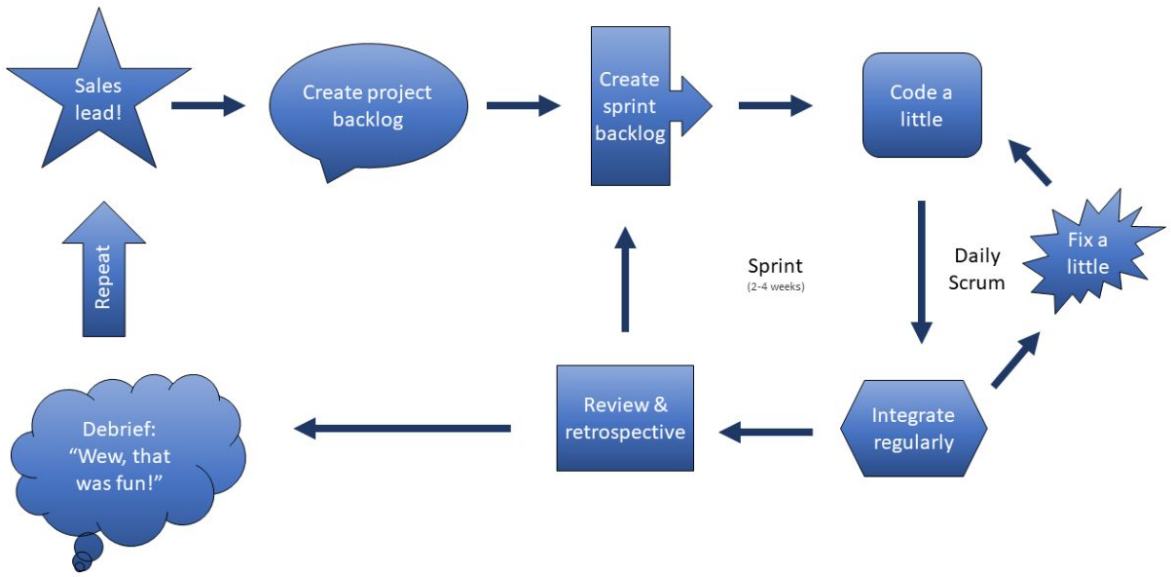
Overall, TDD is an implementation approach that emphasizes writing tests that would check the functionality of code before writing the actual code. Only when the developer is satisfied with the tests and the features those tests test, only then they begin to write the actual code that satisfies the conditions imposed by the test that would allow them to pass.

What does this mean? Essentially this method ensures that you plan the code that you are going to write to pass tests. As you begin the process by writing tests first, it prevents the task of writing tests from being postponed to a later date.

Writing tests would make the process of refactoring easier as you are more likely to catch bugs due to the instant feedback when tests are executed.

TDD within Agile

Remember this process?



At the beginning of the course, we learned about the Agile methodology which is an iterative approach that -

Individuals and interactions

- More than processes and tools
- Adjust process on a per-project basis

Working software

- More than comprehensive documentation
- Always ready to release something

Customer collaboration

- More than contract negotiation
- Less emphasis on writing a big proposal

Responding to change

- More than following a plan
- Iterate regularly

TDD is more often associated with agile for the following reasons:

Agile aims to have a starting point for development and are not as formal as waterfall where a lot of planning is done before the coding or implementation process. Due to the flexible nature of agile, it is easier to have an initial version of the product up and running, without the need to plan extensively.

Test-driven development allows developers to think of specifications. So, when you begin by writing unit tests you have to think of specifications - in the case of agile, it is going to be user stories.

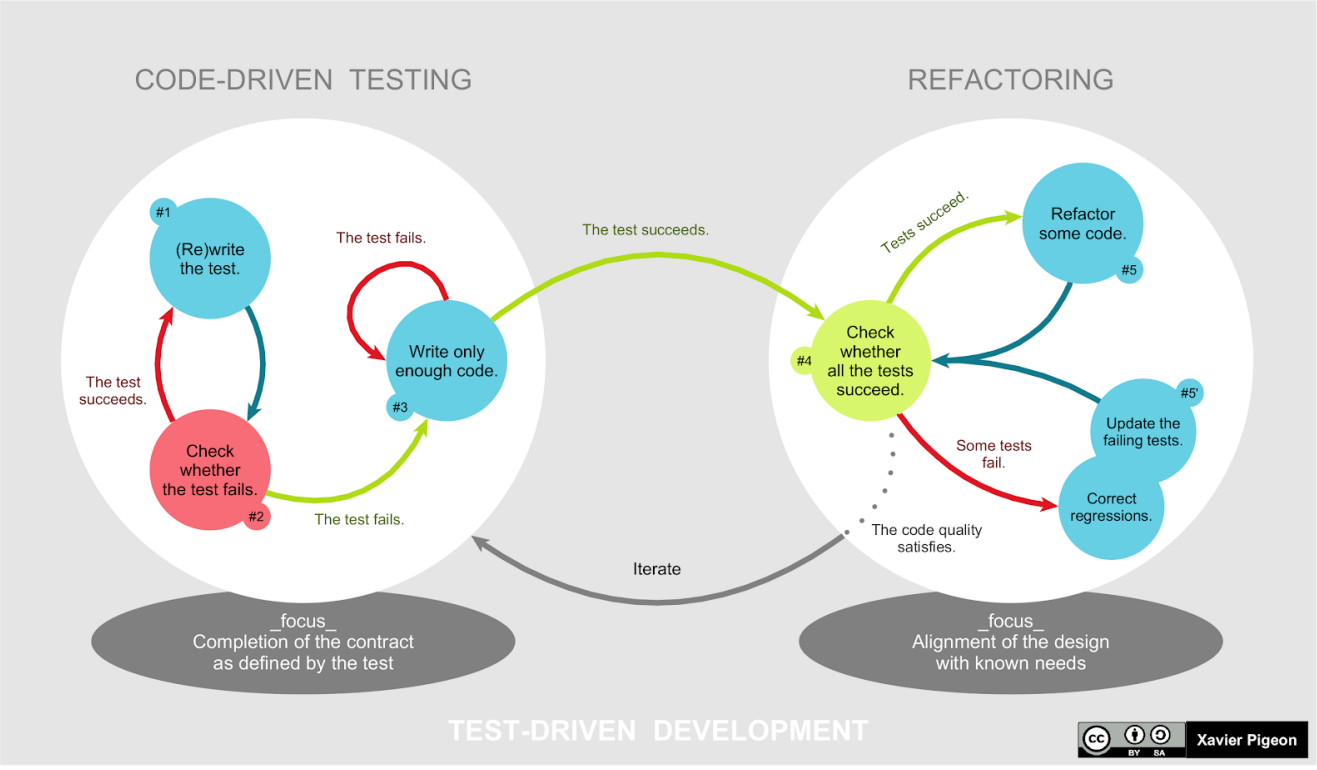
In software, development faults occur very often. TDD is a must to avoid discovering the problem during the build stage - instead, discover it at the initial stage of the process.

Just like in agile, TDD consists of small iterations. In test-driven development, we begin by writing the test cases covering the new functionality, then we write the production code necessary to make the test pass, and then refactor the code to make it more maintainable.

Since we're writing the tests before the actual code, it guarantees immediate feedback after changes are made, i.e, this immediate or rapid feedback is the core element of agile methodology. TDD simplifies the process of developing software and makes it possible to launch a Minimum Viable Product (MVP).

Suggested links and useful resources:

1. Test-Driven development with pytest
2. Test-Driven development - Wikipedia
3. Another visual representation of the TDD process.



4. Code refactoring