

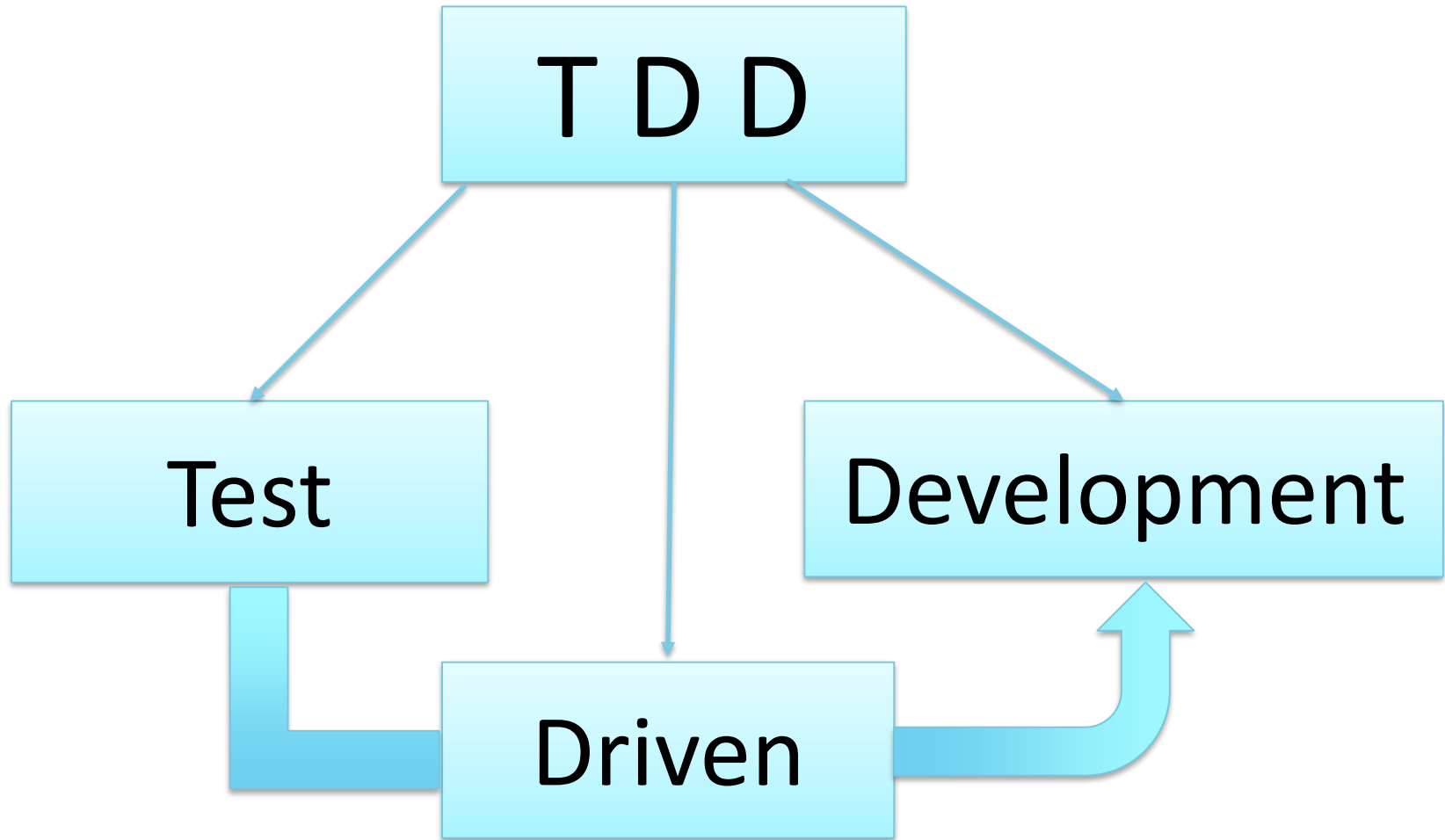


Getting Started with TDD

Ferdous Mahmud Shaon
Managing Director
Cefalo Bangladesh Ltd.



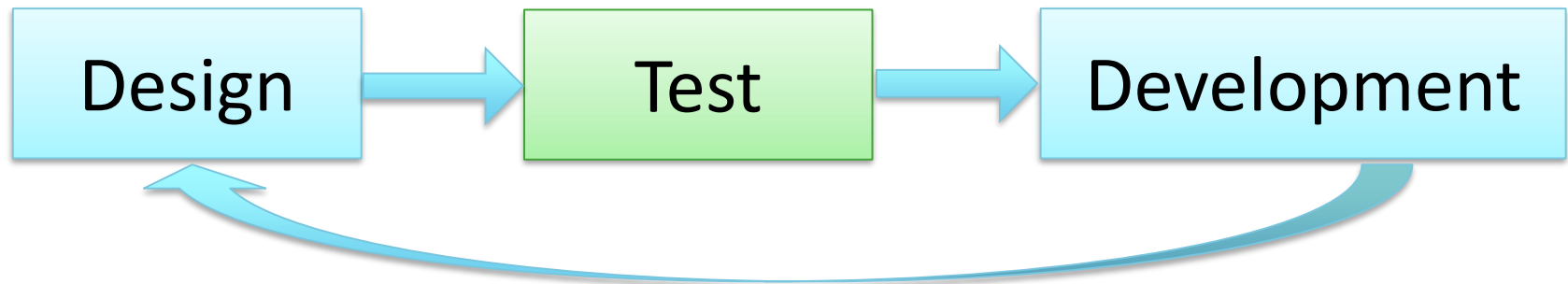
TDD stands for...



Old School Development Approach



New School Development Approach



Test Driven Development

“Test Driven Development (TDD) is a technique for building software that guides **software development by **writing tests**.”**

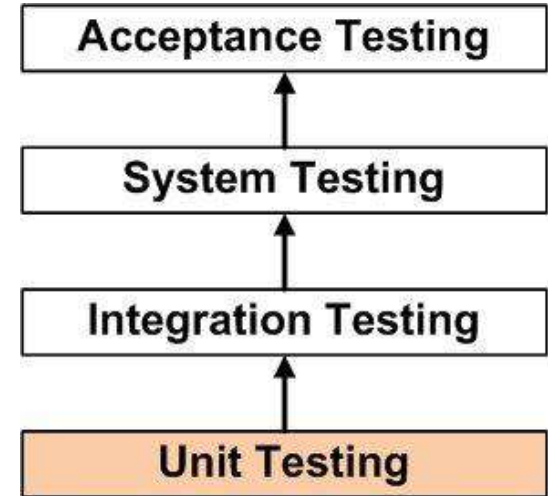
- Martin Fowler (Chief Scientist, ThoughtWorks)

- TDD is NOT primarily about testing or development.
- It is rather about **software design**, where design is evolved through testing and refactoring.



TDD starts with Unit Testing

- In TDD, testing means **Unit Testing**
- Unit Testing is a testing technique by which individual units or components of a software are tested programmatically.



- Unit is the smallest testable part of any software.
- In OOP, smallest unit is a method of a class.
- Who is responsible for writing Unit Tests?
 - ~~Testers~~ Developers



So I need to write
codes to test my
code !!!???

But I learnt only
writing codes, but
not testing it!

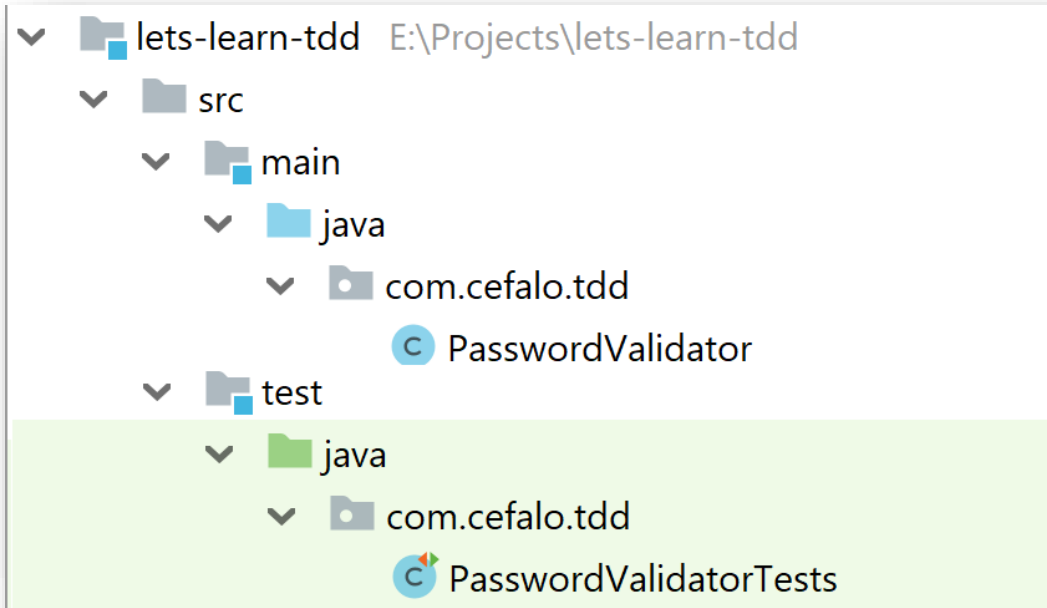


Unit Testing Tools

	
TestNG	
PyUnit	
	

Unit Testing with JUnit

- JUnit is a popular unit testing framework for Java.
- Unit Tests are written like ordinary source codes.
- Source codes and corresponding unit tests are kept under **same package** name but in **different directories**.



Installing JUnit 5

Gradle

```
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.5.2")  
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.5.2")  
}
```

Maven

```
<dependencies>  
  <dependency>  
    <groupId>org.junit.jupiter</groupId>  
    <artifactId>junit-jupiter-api</artifactId>  
    <version>5.5.2</version>  
    <scope>test</scope>  
  </dependency>  
  <dependency>  
    <groupId>org.junit.jupiter</groupId>  
    <artifactId>junit-jupiter-engine</artifactId>  
    <version>5.5.2</version>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

JUnit Methods

- Annotations are used to mark a method as test method.
- Some helper methods might be needed to be executed before and/or after running a test method.
- All these methods must return “void” and take no parameters.
- A test method must have at least **one Assertion** in it.

JUnit5 Annotations

Annotation	Description
<code>@Test</code>	It is used to mark a method as a JUnit test. Referring to <i>org.junit.jupiter.api.Test</i>
<code>@BeforeEach</code>	The annotated method will be run before each test method in the test class. Referring to <i>org.junit.jupiter.api.BeforeEach</i>
<code>@AfterEach</code>	The annotated method will be run after each test method in the test class. Referring to <i>org.junit.jupiter.api.AfterEach</i>
<code>@BeforeAll</code>	The annotated method will be run before all test methods in the test class. This method must be static. Referring to <i>org.junit.jupiter.api.BeforeAll</i>
<code>@AfterAll</code>	The annotated method will be run after all test methods in the test class. This method must be static. Referring to <i>org.junit.jupiter.api.AfterAll</i>



JUnit5 Assert Methods

These methods are used in unit test methods for comparing expected and actual result. They are static methods in *org.junit.jupiter.api.Assertions* class.

Assert Method	Description
<code>assertEquals()</code> <code>assertNotEquals()</code>	Invokes the equals() methods on the arguments to check whether they are equal.
<code>assertSame()</code> <code>assertNotSame()</code>	Uses == on the arguments to check whether they are equal
<code>assertTrue()</code> <code>assertFalse()</code>	Checks if the given boolean argument evaluates to true or false
<code>assertNull()</code> <code>assertNotNull()</code>	Checks if the given argument is null or NOT null
<code>assertArrayEquals()</code>	Checks if the given array arguments passed have same elements in the same order
<code>assertThrows()</code>	Checks if the given code block throws an Exception and the type of the Exception matches with the given type.



Unit Test Example with JUnit5

```
public class SimpleCalculatorTests {
    private static SimpleCalculator simpleCalculator;
    @BeforeAll
    public static void setupSimpleCalculator() {
        simpleCalculator = new SimpleCalculator();
    }
    @Test
    public void testAddition() {
        assertEquals(5, simpleCalculator.add(2,3) );
    }
    @Test
    public void testDivision() {
        assertEquals(5, simpleCalculator.divide(10,2) );
    }
    @Test
    public void testDivision_zero_divisor() {
        assertThrows(IllegalArgumentException.class,
            () -> simpleCalculator.divide(10,0));
    }
}
```

Unit Tests Best Practices

- Unit test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
- Follow clear and consistent naming conventions for your unit tests. Test name should reflect the intent/purpose of the test.
- In case of a change in code in any module, ensure there is a corresponding unit test case for the module, and the module passes the tests before changing the implementation.
- Bugs identified during unit testing must be fixed before proceeding to the next phase in SDLC
- Adopt a "test as your code" approach. The more code you write without testing, the more paths you have to check for errors.



Guidelines for writing Unit Tests

A test is not a Unit Test if:

- It connects with the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

— Michael Feathers,
A Set Of Unit Testing Rules (2005)

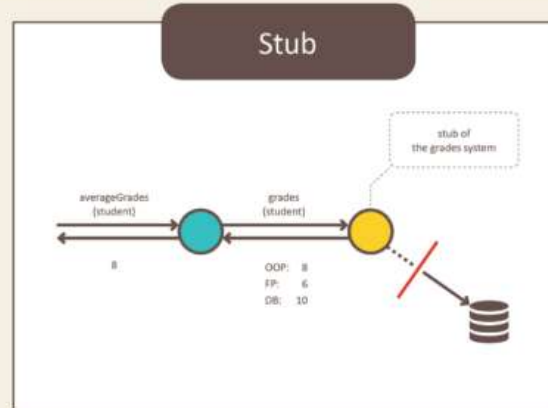
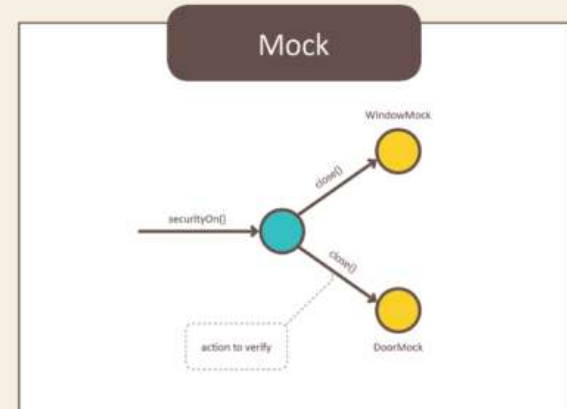
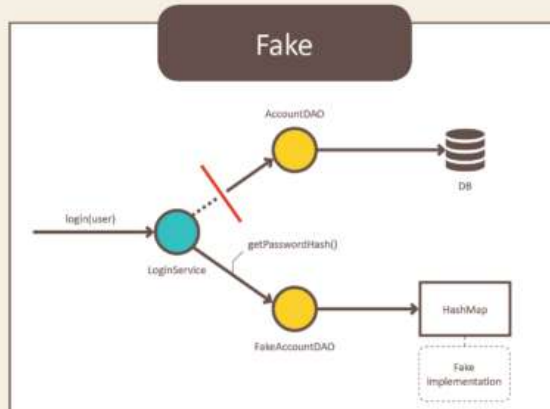


Test Doubles

- Unit test should test functionality in isolation.
- Side effects from other classes or the system should be eliminated for a unit test, if possible.
- This can be done via using test replacements (test doubles) for the real dependencies.
- Test double is an object that can stand in for a real object in a test, similar to how a stunt double stands in for an actor in a movie.

Test Doubles

Test Doubles

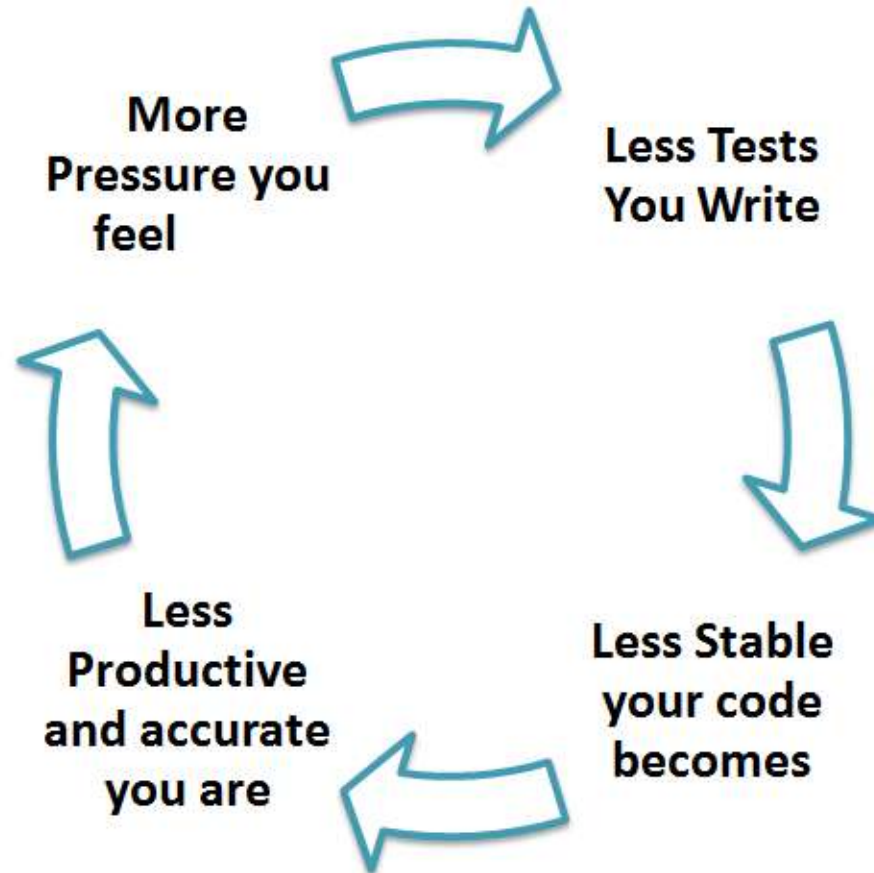


Common Excuses against Unit Tests

- I am paid to write code, not tests!
- I am a developer, not tester - so I'm not responsible for writing tests!
- We already have testers - why do we need unit tests?
- We are working on a tight deadline - where do I have time for writing unit tests?
- I don't know how to write unit tests
- Ours is legacy code - can't write unit tests for them
- If I touch the code, it may break!



Myths lead to a Vicious Cycle



"Never in the field of software engineering has so much been owed by so many to so few lines of code."

– Martin Fowler on Unit Testing



But I already write unit tests!

- Just because, you write unit tests, doesn't mean you follow TDD!
- Unit tests are often written after development (coding implementation)
 - That is called Plain Old Unit testing (POUting)
- However in TDD approach, Unit tests must be written first before development (coding implementation)

Unit testing after code: Disadvantages

- Testing does not give direct feedback to design and programming
 - But in TDD, the feedback is directly fed back into improving design and programs
- Most often, after implementing the functionality in code, unit testing is omitted.
 - TDD inverts this sequence and ensure all components have unit tests first before implementation
- Writing tests after developing code often results in “happy path” testing only!

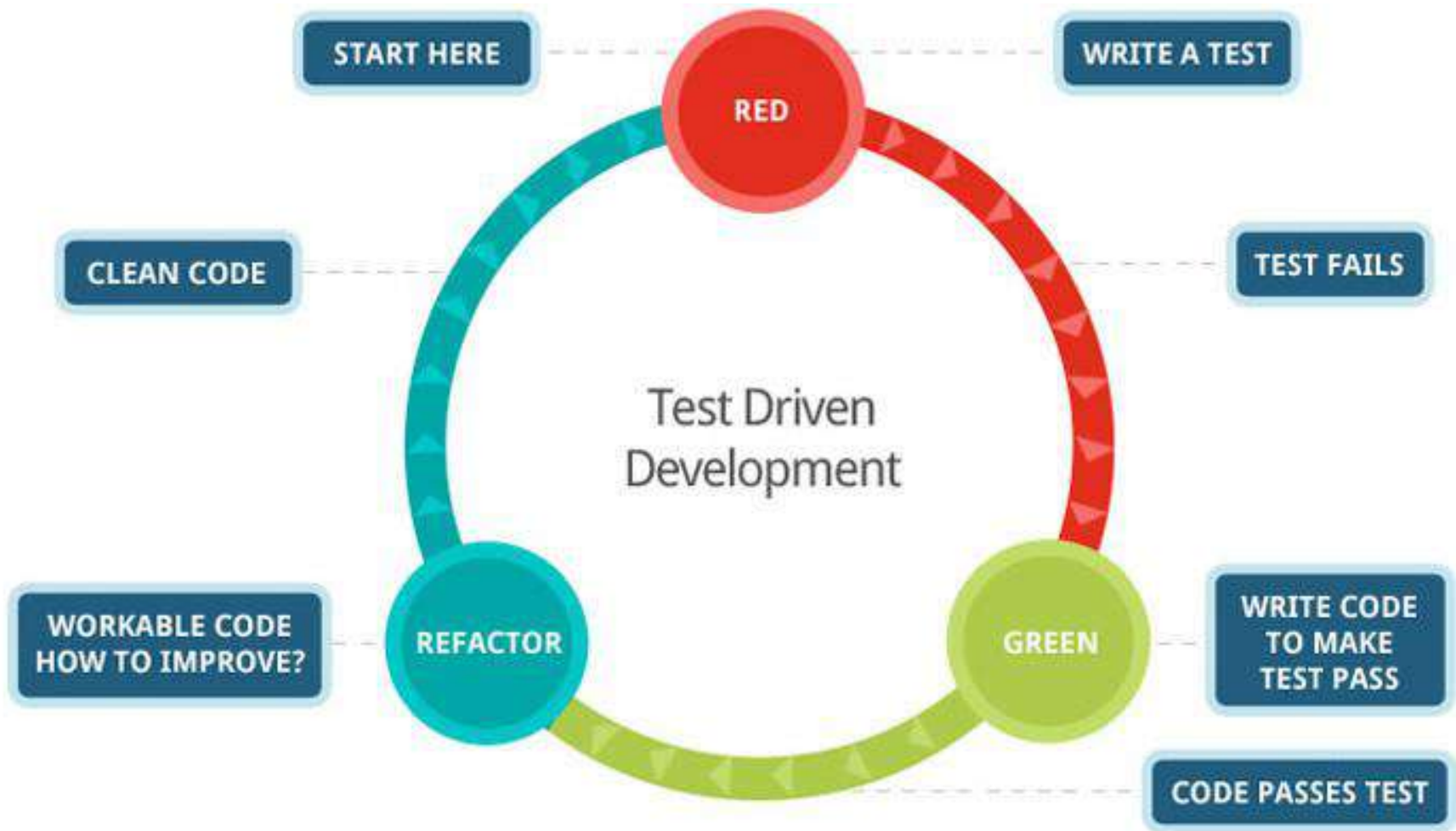
Why Test First Approach?



Why Test First Approach?



Test Driven Development Cycle



Step 1: Write test that fails

Red—write a little test that doesn't work, perhaps doesn't even compile at first!



Source: Test Driven Development: By Example, Kent Beck, 240 pages, Addison-Wesley Professional, 2002

Step 2: Get code working to pass test

Green — write minimal amount of code to make the test work, committing whatever sins necessary in the process.



Source: Test Driven Development: By Example, Kent Beck, 240 pages, Addison-Wesley Professional, 2002

Step 3: Cleanup and Refactor

- **Refactor** — eliminate all duplications and code smells, which may have been introduced while getting the test to pass in Step 2.



Source: Test Driven Development: By Example, Kent Beck, 240 pages, Addison-Wesley Professional, 2002

TDD Mantra



TDD in action

BRACE YOURSELVES

TDD. BEGINS. TODAY.



Let's Build a Simple Calculator

- We will try to implement a simple calculator by applying TDD approach
- Let's keep the requirements very simple.
- Our simple calculator will cover only 2 functions:
 - **Addition** of 2 given numbers
 - Input 2 numbers
 - Returns the sum of the 2 numbers
 - **Division** of one number by another number
 - Input 2 numbers – dividend and divisor
 - Returns the quotient of the 2 numbers
 - Note, divisor cannot be zero



TDD in action

SimpleCalculator Class

- Let's start with an empty class
- We'll gradually build up this through TDD

```
public class SimpleCalculator {  
  
}
```

TDD in action

Test1: write a testcase for adding 2 numbers


- input any 2 numbers
- expect to have the sum of them

```
public class SimpleCalculatorTests {  
    @Test  
    public void testAddition() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals(5, simpleCalculator.add(2,3) );  
    }  
}
```


TDD in action

But our test for addition fails!

```
public class SimpleCalculatorTests {  
    @Test  
    public void testAddition() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals(5, simpleCalculator.add(2,3) );  
    }  
}
```



Error:

```
\src\test\java\com\cefalo\tdd\SimpleCalculatorTests.java:25:  
error: cannot find symbol assertEquals(5, simpleCalculator.add(2,3) );  
symbol:   method add(int,int) ^  
location: variable simpleCalculator of type SimpleCalculator
```


TDD in action

In SimpleCalculator class, we'll write bear minimum amount of code to

- fix the error by introducing add() method
- Pass the test for adding 2 numbers

```
public class SimpleCalculator {  
    public double add(final double pNumber1, final double pNumber2) {  
        return pNumber1 + pNumber2;  
    }  
}
```

- Change it until we make it **RED** to **GREEN**!



TDD in action

Test2: write a testcase for division of 2 numbers


- input 2 numbers: dividend and divisor
- expect to have the quotient

```
public class SimpleCalculatorTests {  
    @Test  
    public void testAddition() {...}  
    @Test  
    public void testDivision() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals(5, simpleCalculator.divide(10,2) );  
    }  
}
```

TDD in action

But our test for division also fails!

```
public class SimpleCalculatorTests {  
    @Test  
    public void testAddition() {...}  
    @Test  
    public void testDivision() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals(5, simpleCalculator.divide(10,2) );  
    }  
}
```



Error:

[\src\test\java\com\cefalo\tdd\SimpleCalculatorTests.java:30:](#)

error: cannot find symbol assertEquals(5, simpleCalculator.divide(10,2));

symbol: method divide(int,int)

location: variable simpleCalculator of type SimpleCalculator

TDD in action

In SimpleCalculator class, we'll write bear minimum amount of code to

- fix the error by introducing divide() method
- Pass the test for division of 2 numbers

```
public class SimpleCalculator {  
    public double add(final double pNumber1, final double pNumber2) { return pNumber1 + pNumber2; }  
    public double divide(final double pDividend, final double pDivisor) {  
        return pDividend / pDivisor;  
    }  
}
```

- The goal is to pass the failing test somehow.

TDD in action

Refactor

- Look for any code smells, duplicates in main and test class and refactor them.
- Make sure, the test result doesn't get affected!

```
public class SimpleCalculatorTests {  
    @Test  
    public void testAddition() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals( expected: 5, simpleCalculator.add(2,3) );  
    }  
    @Test  
    public void testDivision() {  
        SimpleCalculator simpleCalculator = new SimpleCalculator();  
        assertEquals( expected: 5, simpleCalculator.divide( pDividend: 10, pDivisor: 2) );  
    }  
}
```

TDD in action

Refactor the Code Smells

```
public class SimpleCalculatorTests {  
    private static SimpleCalculator simpleCalculator;  
    @BeforeAll  
    public static void setupSimpleCalculator() {  
        | simpleCalculator = new SimpleCalculator();  
    }  
    @Test  
    public void testAddition() {  
        | assertEquals( expected: 5, simpleCalculator.add(2,3) );  
    }  
    @Test  
    public void testDivision() {  
        | assertEquals( expected: 5, simpleCalculator.divide( pDividend: 10, pDivisor: 2) );  
    }  
}
```


TDD in action

Test3: write a testcase for division with 0 as divisor


- input any number as dividend and 0 as divisor
- Let's expect to have **IllegalArgumentException**

```
public class SimpleCalculatorTests {  
    private static SimpleCalculator simpleCalculator;  
    @BeforeAll  
    public static void setupSimpleCalculator() { simpleCalculator = new SimpleCalculator(); }  
    @Test  
    public void testAddition() { assertEquals( expected: 5, simpleCalculator.add(2,3) ); }  
    @Test  
    public void testDivision() { assertEquals( expected: 5, simpleCalculator.divide( pDividend: 10, pDivisor: 2) ); }  
    @Test  
    public void testDivision_zero_divisor() {  
        assertThrows(IllegalArgumentException.class, () -> simpleCalculator.divide( pDividend: 10, pDivisor: 0));  
    }  
}
```

TDD in action

But our 0 divisor test fails!

```
public class SimpleCalculatorTests {  
    private static SimpleCalculator simpleCalculator;  
    @BeforeAll  
    public static void setupSimpleCalculator() { simpleCalculator = new SimpleCalculator(); }  
    @Test  
    public void testDivision_zero_divisor() {  
        assertThrows(IllegalArgumentException.class, () -> simpleCalculator.divide( pDividend: 10, pDivisor: 0));  
    }  
}
```



Error:

✖ Tests failed: 1 of 1 test – 13 ms

```
Expected java.lang.IllegalArgumentException to be thrown, but nothing was thrown.  
org.opentest4j.AssertionFailedError: Expected java.lang.IllegalArgumentException to be thrown, but nothing was thrown.  
    at com.cefalo.tdd.SimpleCalculatorTests.testDivision_zero_divisor(SimpleCalculatorTests.java:39) <31 internal calls>  
  
com.cefalo.tdd.SimpleCalculatorTests > testDivision_zero_divisor() FAILED  
    org.opentest4j.AssertionFailedError at SimpleCalculatorTests.java:39  
1 test completed, 1 failed
```

TDD in action

Now, we'll just need to throw `IllegalArgumentException`, if divisor is 0.

```
public class SimpleCalculator {  
    public double add(final double pNumber1, final double pNumber2) { return pNumber1 + pNumber2; }  
  
    public double divide(final double pDividend, final double pDivisor) {  
        if(pDivisor==0) {  
            throw new IllegalArgumentException("Divisor cannot be 0");  
        }  
        return pDividend / pDivisor;  
    }  
}
```

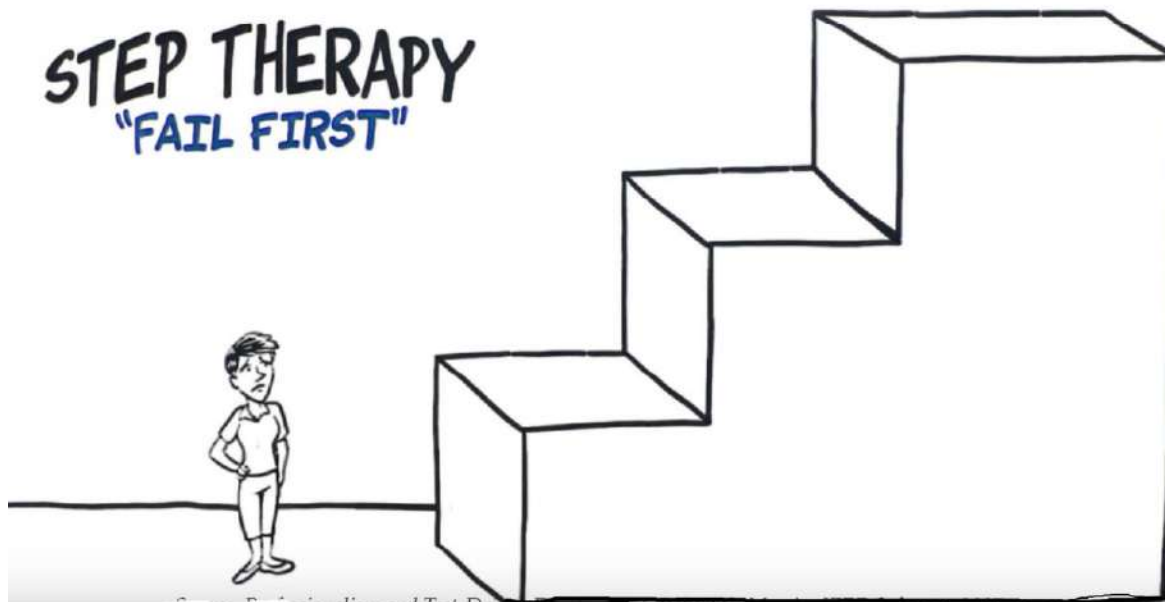
This small change is enough to pass the test.



3 Laws of TDD

Uncle Bob's First law of *Testto Dynamics*:

You may not write production code unless you've first written a failing unit test.



3 Laws of TDD (cont.)

Uncle Bob's Second law of *Testto Dynamics*:

You may not write any more of a unit test than is sufficient to fail.



3 Laws of TDD (cont.)

Uncle Bob's Third law of *Testto Dynamics*:

You may not write more production code than is sufficient to make the failing unit test pass.



[shortcode]

Summarize Uncle Bob's rules for TDD

- Start with a small unit test, that fails
- Write bare minimum code to make the failing test pass
- Refactor the code smells without affecting functionality and test result
- Continue the above 3 steps until you have any failing unit tests

Benefits of TDD

Cleaner Code	Testable SOLID Code
Self-documenting Unit Tests	Increased Quality, Reduced Bugs
Maintainable and Extensible Code	Happy-path and Unhappy path Testing
Safer Refactoring	Faster Debugging

When not to use TDD?

- Developers must be experienced in writing tests before implementing TDD.
- More suited for implementing complex business logic, back-end systems etc.
- Can be overhead for very simple projects.
- Not suitable for UI development.
- Can backfire when the environment is not suitable or it is used incorrectly.

Does TDD guarantee Good Code?

“TDD helps with, but does not guarantee,
good design & good code.

Skill, talent, and expertise remain necessary.”

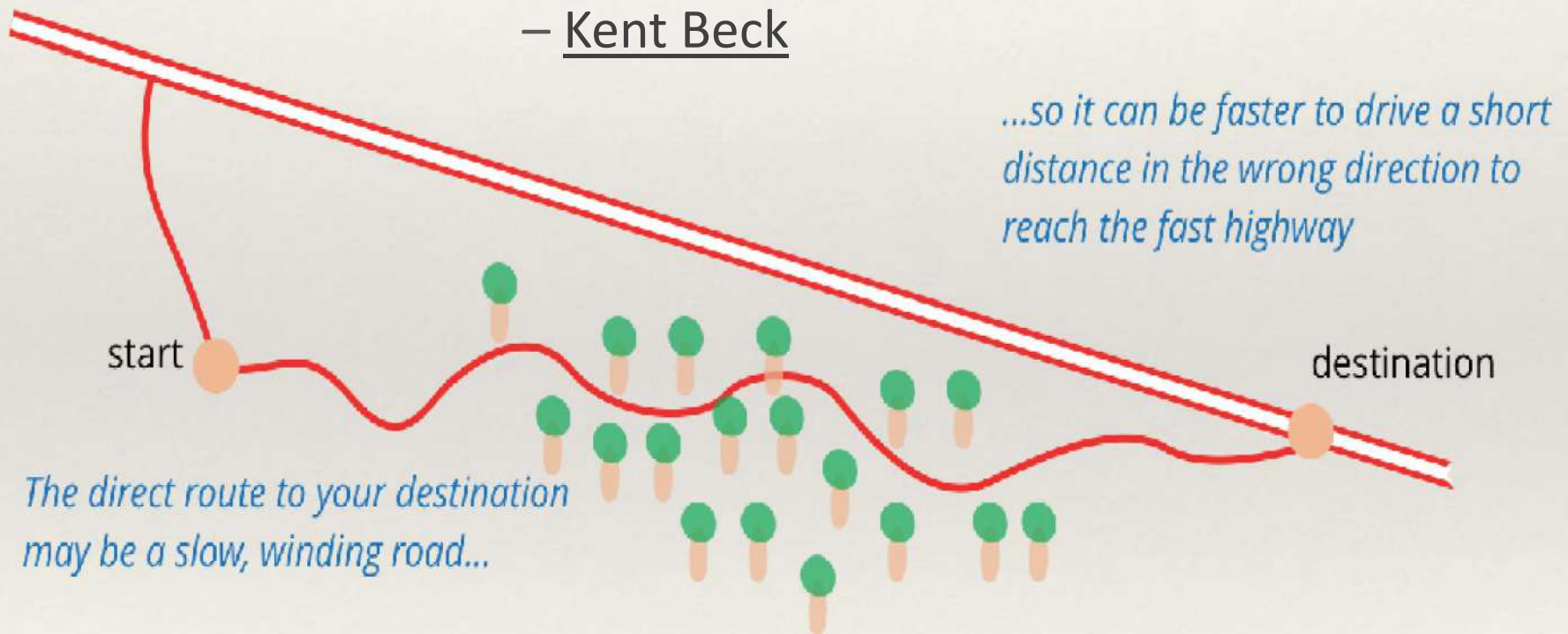
— Esko Luontola



TDD slows down Development?

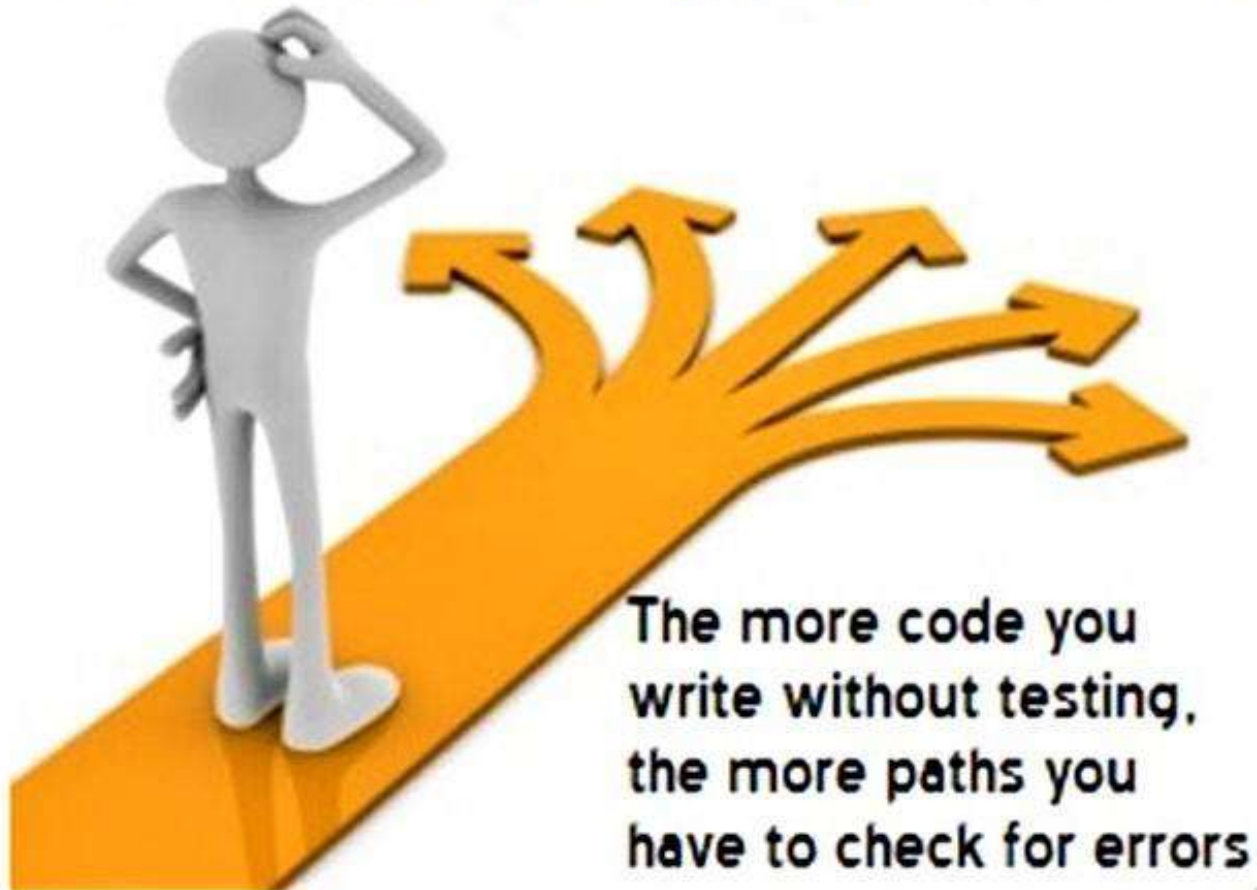
“I can write code a lot faster
if it doesn't have to work!”

– Kent Beck

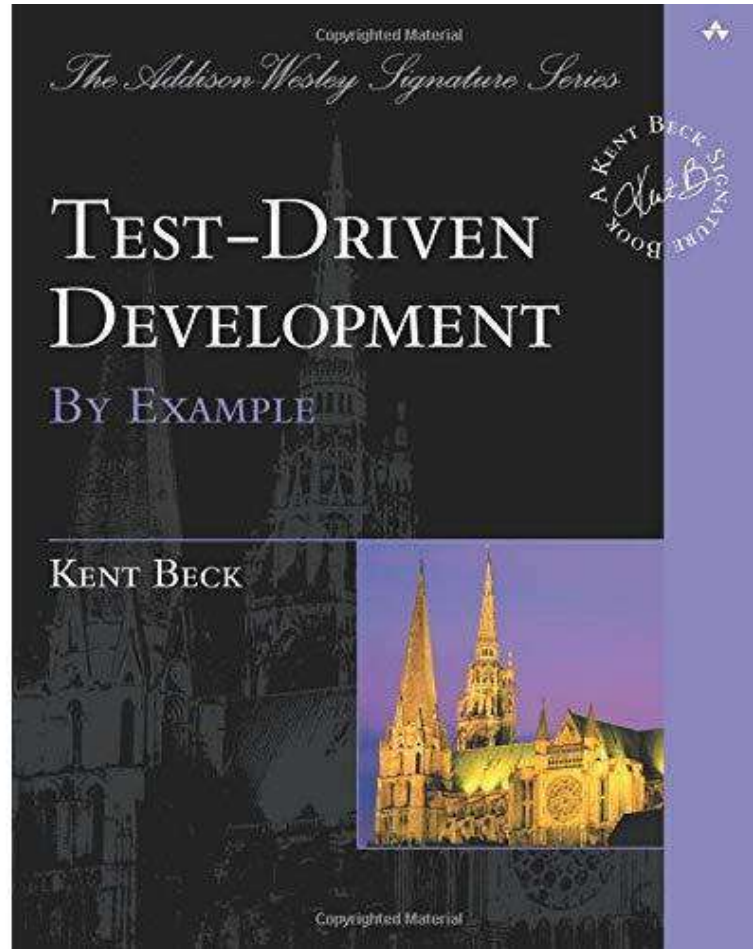


Follow the straight path!

Keep on a straight path with proper unit testing.



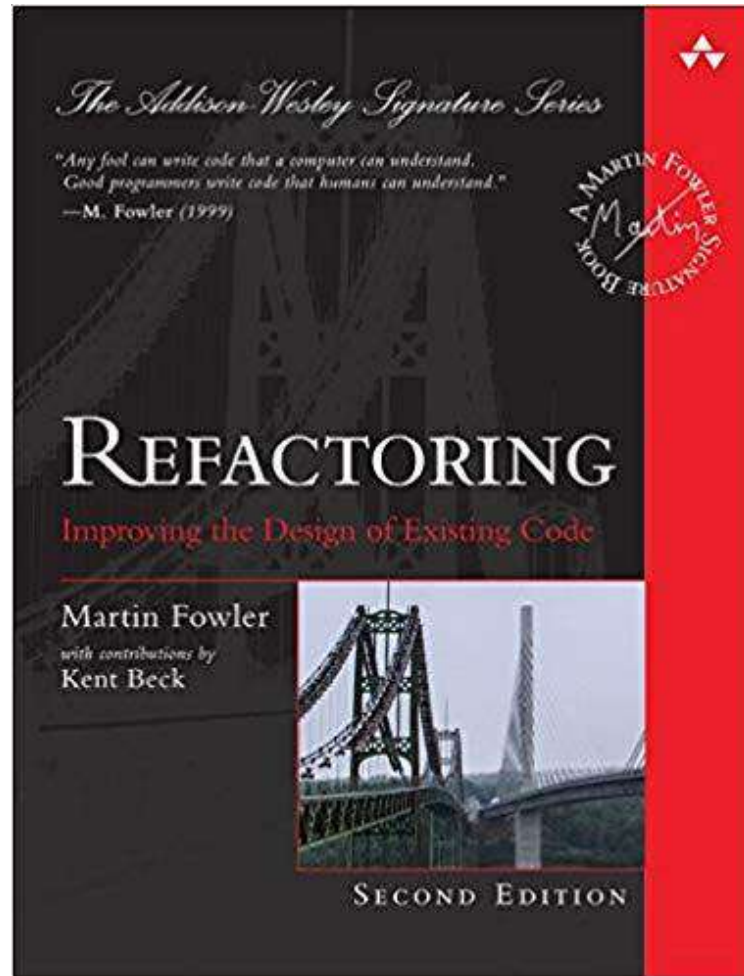
Recommended Reading



Must read on TDD
by its originator
Kent Beck

Test Driven Development: By Example, by Kent Beck

Recommended Reading



Covers refactoring of code smells by design principles & best practices!

Refactoring: Improving the Design of Existing Code, by Martin Fowler

Useful Links

- GitHub Repository
<https://github.com/Cefalo/lets-learn-tdd>
- <https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/>
- <https://hackernoon.com/introduction-to-test-driven-development-tdd-61a13bc92d92>
- <https://www.guru99.com/test-driven-development.html>
- <http://agiledata.org/essays/tdd.html>



Thank You!



Question & Answers

