

CS1632, Lecture 10: Test-Driven Development

Wonsun Ahn

THE DARK AGES

Bill Laboon



Nowadays...

- We know how important tests are to prevent issues like that
- Code quality is everyone's responsibility, including developers'
- Developers write tests (usually unit tests)

Test-Driven Development

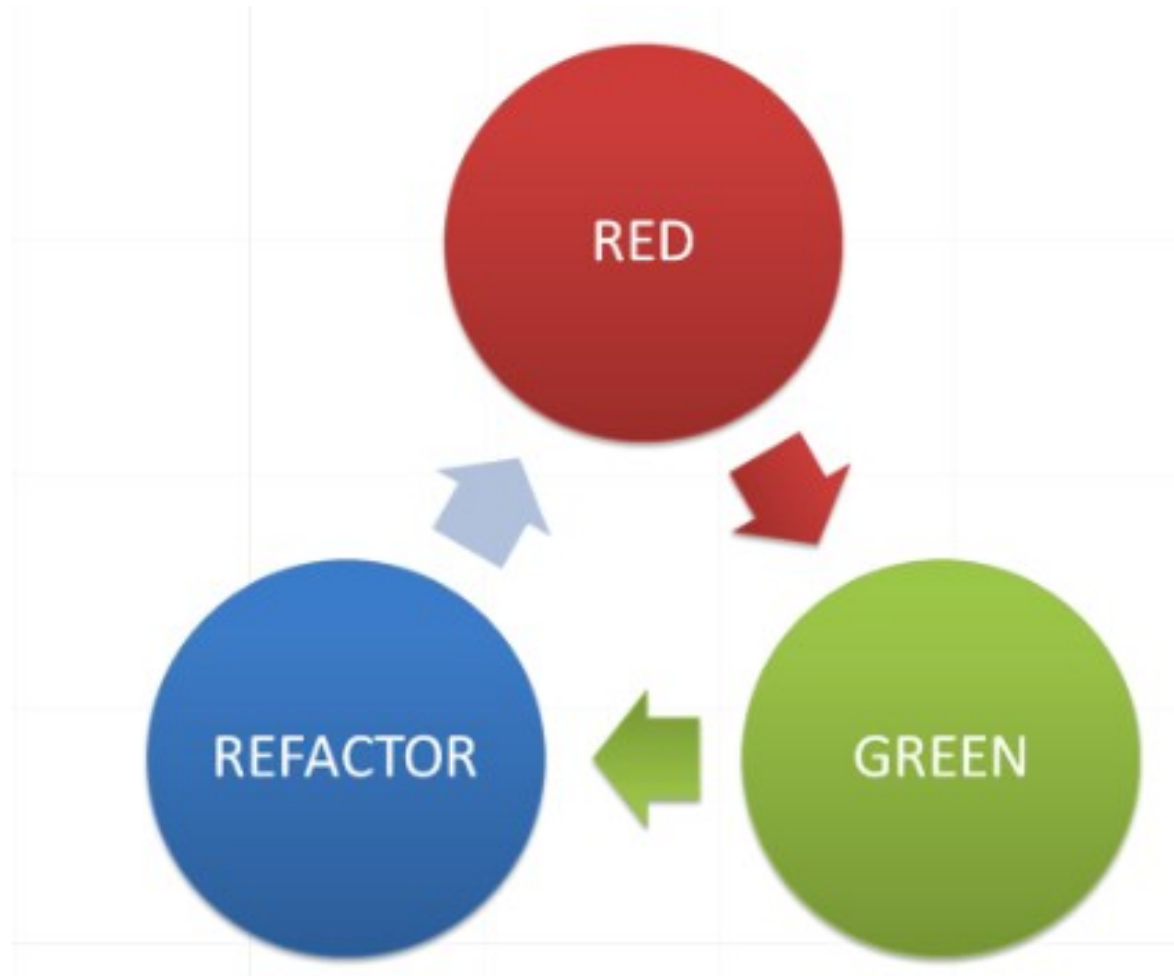
- A strategy for developing highly tested, quality software
- Not the be-all and end-all of strategies
- Google “TDD is dead” for a great argument against it
- Welcome to the still-forming world of software development!

So What is TDD?

A software development methodology that comprises:

1. Writing tests BEFORE writing code
2. Writing ONLY code that is tested
3. Writing ONLY tests that test the code
4. A very short turnaround cycle
5. Refactoring early and often

The Red-Green-Refactor Loop



The Red-Green-Refactor Loop

- **Red** – Write a test for new functionality
 - Test should immediately fail! (Hence the **Red**)
- **Green** – Write only enough code to make the test pass
 - Now the test should pass. (Hence the **Green**)
- **Refactor** – Review code and make it better

Fizzbuzzin' With TDD

- Requirements:

App shall print numbers from 1 to 100 delimited by spaces.

If number is a multiple of 3, app shall print "Fizz" instead.

If number is a multiple of 5, app shall print "Buzz" instead.

If number is a multiple of 3 and 5, app shall print "FizzBuzz" instead.

- Example output: 1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz ...

Red - Start by adding a new test

```
@Test
public void testNumber() {
    assertEquals(_fb.value(1), "1");
}
// Code
public String value(int n) {
    return "";
}
```

... Which fails.



Green - Write code to make test pass

```
@Test
public void testNumber() {
    assertEquals(_fb.value(1), "1");
}
// Code
public String value(int n) {
    return "1";
}
```



Nothing to Refactor.
Next test!

Red - Let's Add Another Test

```
@Test  
public void testNumber2() {  
    assertEquals(_fb.value(2), "2");  
}
```

```
// Code  
public String value(int n) {  
    return "1";  
}
```



Green - Let's Make Another Change

```
public String value(int n) {  
    if (n == 1) {  
        return "1";  
    } else {  
        return "2";  
    }  
}
```



But could be better!

Refactor – now much nicer, and tests still pass!

```
public String value(int n) {  
    return String.valueOf(n);  
}
```

Red - Add Another Test – it fails

```
@Test  
public void testNumber3() {  
    assertEquals(_fb.value(3), "Fizz");  
}
```

Green - We Need to Add Fizzy Code!

```
private boolean fizzy(int n) {  
    return (n % 3 == 0);  
}
```

```
public String value(int n) {  
    if (fizzy(n)) {  
        return "Fizz";  
    } else {  
        return String.valueOf(n);  
    }  
}
```



Refactor – Nothing to do here

```
private boolean fizzy(int n) {  
    return (n % 3 == 0);  
}
```

```
public String value(int n) {  
    if (fizzy(n)) {  
        return "Fizz";  
    } else {  
        return String.valueOf(n);  
    }  
}
```

Red - Let's Add A Test For Buzziness

```
@Test  
public void testNumber5() {  
    assertEquals(_fb.value(5), "Buzz");  
}
```

... and of course it fails

Green - Add and Integrate buzzy(n) Method

```
private boolean buzzy(int n) {  
    return (n % 5 == 0);  
}  
public String value(int n) {  
    if (fizzy(n)) {  
        return "Fizz";  
    } else if (buzzy(n)) {  
        return "Buzz";  
    } else {  
        return String.valueOf(n);  
    }  
}
```



Red - The Final Equivalence Class

```
@Test  
public void testNumber15() {  
    assertEquals(_fb.value(15), "FizzBuzz");  
}
```

... and it fails

Green - Modify The value() Method

```
public String value(int n) {  
    if (fizzy(n) && buzzy(n)) {  
        return "FizzBuzz";  
    } else if (fizzy(n)) {  
        return "Fizz";  
    } else if (buzzy(n)) {  
        return "Buzz";  
    } else {  
        return String.valueOf(n);  
    }  
}
```



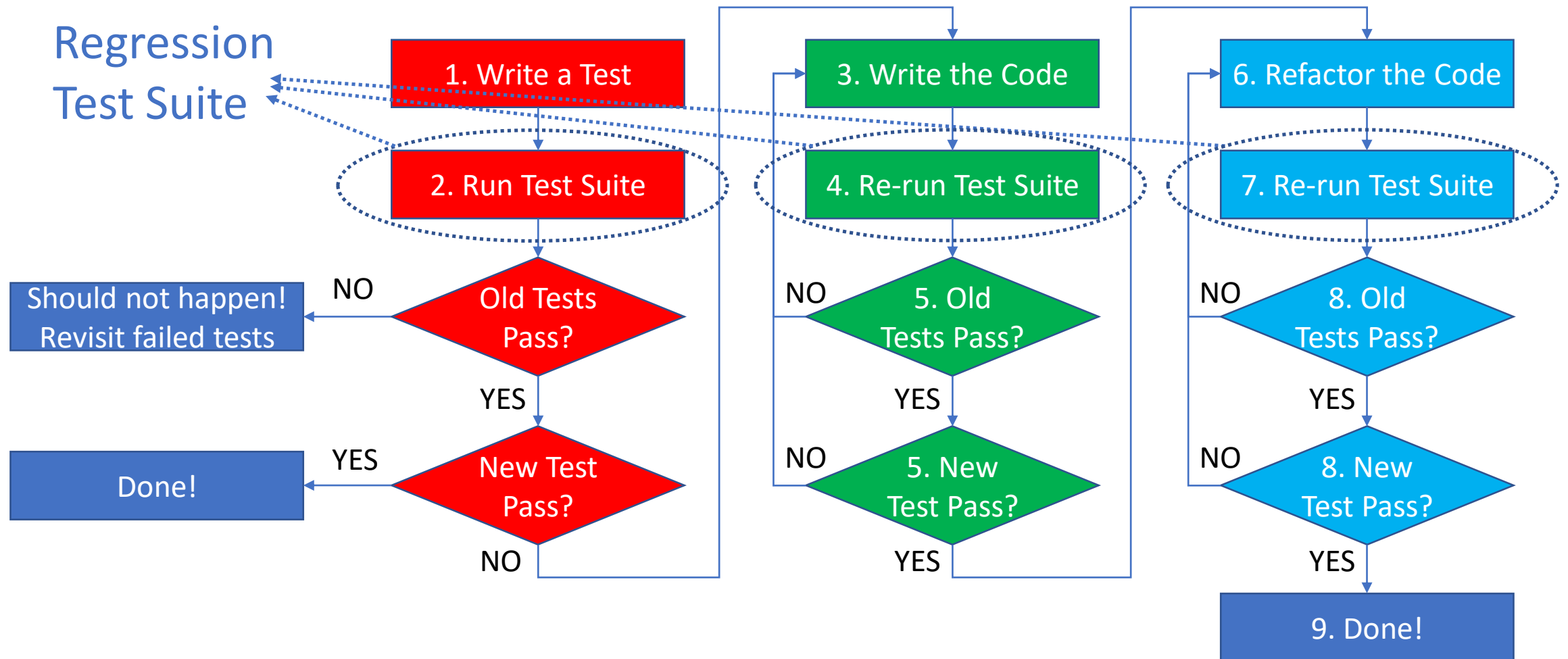
Refactor – Nothing to do

```
public String value(int n) {  
    if (fizzy(n) && buzzy(n)) {  
        return "FizzBuzz";  
    } else if (fizzy(n)) {  
        return "Fizz";  
    } else if (buzzy(n)) {  
        return "Buzz";  
    } else {  
        return String.valueOf(n);  
    }  
}
```

Result?

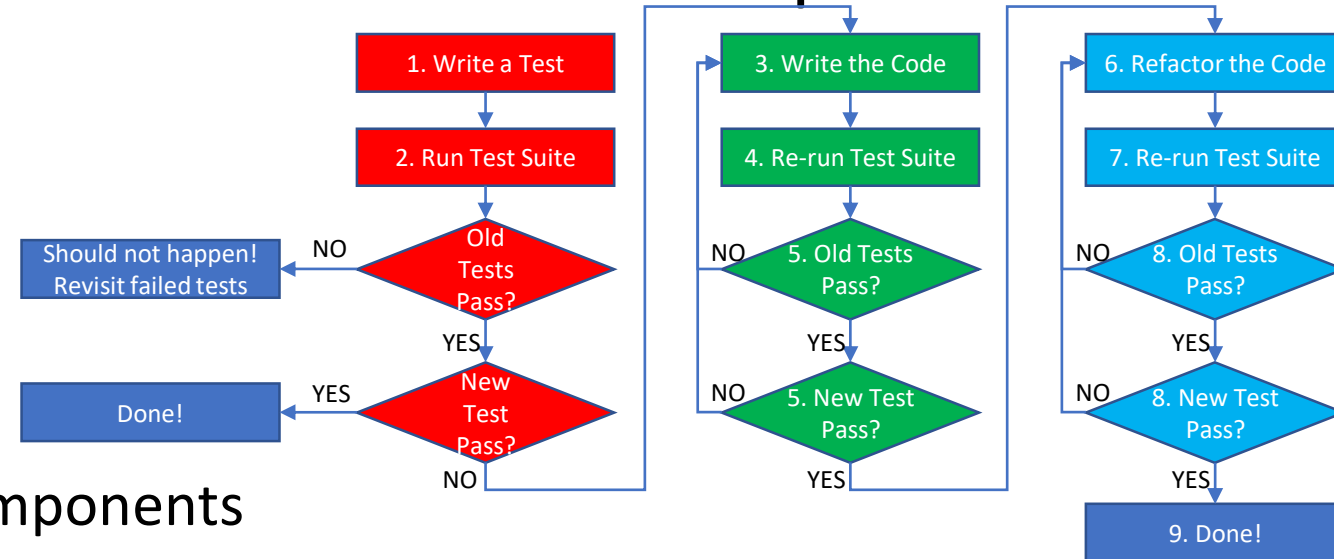
- We now have a working, tested implementation of FizzBuzz
- We have automated test coverage for all equivalence classes
- We had a path forward at all points

Flow Chart of Red-Green-Refactor Loop



Flow Chart of Red-Green-Refactor Loop

- Write one small test at a time
 - Keep a tight turnaround cycle!
- Run the entire test suite
 - Look out for regression errors!
 - Ensures continuous testing of all components
- Write just enough code to pass test
 - Resist temptation to go into untested territory
 - Ensures all code you are writing and all previous code is continuously covered
- Refactor at every iteration
 - Encourages focus on correctness since you will have a chance to refactor soon
 - Don't forget to regression test after refactoring



What does Refactor involve?

- Your first attempt at writing code will probably not be perfect
 - Poor algorithm choice?
 - Bad variable names?
 - Poor performance?
 - Badly documented?
 - Magic numbers?
 - Not easily comprehensible?
 - General bad design?
- Remember you already had a working version before refactoring
 - Being correct is more important than being good-looking

Key - Do not Write Code Beyond the Test

- Some jingles to curb your enthusiasm ...
- *YAGNI* - You Ain't Gonna Need It
 - If you are not testing it, that means you don't need it right now
 - If you don't need it now, chances are you do won't need it in the future
- *KISS* - Keep It Simple, Smarty-pants
 - Don't write overly complex, clever, over-engineered code
 - It just makes it harder to understand and modify later
 - "Premature optimization is the root of all evil" –Donald Knuth
- Fake It 'til You Make It
 - Don't get mired in implementing something strictly not necessary to pass test

Fake It 'til You Make It

- Obviously applies to mocks/stubs
- But you can apply to smaller levels of functionality

Test:

```
assertEquals(sqrt(4), 2);
```

Code:

```
public void sqrt(int n) {  
    return 2;  
}
```

Make Unit Tests Fast and Independent

- Notice we are running an entire test suite on every small change?
 - That means testing time is going to heavily impact development cycle
- How do we avoid long testing delays?
- Fast: Make individual unit tests run quickly
 - Make heavy use of test doubles and stubbing for delay-prone components (e.g. databases, files, network I/O)
- Independent: Make sure tests do not rely on the result of other tests
 - Allows developer to arbitrarily choose only unit tests related to modified code
 - Allows tests to be split up and run in parallel

Benefits of TDD

- Ensures tests are relevant
 - Tests are written for the exact functionality you are implementing
 - Not some imaginary functionality you may implement in the future (or not)
- Ensures code is relevant
 - Code does not veer far from user needs, expressed in the form of unit tests
- Ensures that you take small steps
 - Experience programmers know small steps help localize errors
- Large test suite automatically created (at least it feels like it)
 - Helps avoid regression errors through high code coverage at all times
- Confidence in the codebase through continuous testing

Drawbacks of TDD

- Tests become part of the overhead of the project
 - In terms of maintenance: especially if they are brittle/fragile
 - In terms of development cycle: sheer testing time slows down **RGR** loop
(If you stub to speed up testing, it increases maintenance cost ... sad)
- Focus on unit tests may mean other aspects of testing get short shrift
 - Remember that unit tests focus on small units of code, not integration
- Hard to do large architectural designs / redesigns
 - TDD keeps you short-sighted; you need long term thinking for a good design
 - Some things just aren't feasible to do in small steps
- Over-reliance on the unit test
 - And the developer-centric, method-centric view that goes along with it

TDD = A Kind of Test-First Development

- Basic idea is to think about expected behavior FIRST, before coding
- Figure out what the program should do (requirements!)
- Write tests towards the requirements
- Write code towards the tests

* Side note: there are other kinds of test-first development, such as ATDD (Acceptance Test Driven Development) and BDD (Behavior Driven Development)

Now Please Read Textbook Chapter 15

- “TDD is dead. Long live testing.” - David Heinemeier Hansson:
<https://dhh.dk/2014/tdd-is-dead-long-live-testing.html>