

CS1632, Lecture 11: 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)

But...

- What to test?
- How deep to go into testing?
- How many edge cases?
- How to prioritize testing and development?
- What order should I write tests?
- How do I structure code to be testable?

There is no one right answer

- Many studies done
- Different domains, different developers, different languages, etc...
- “No silver bullet”

Test-Driven Development

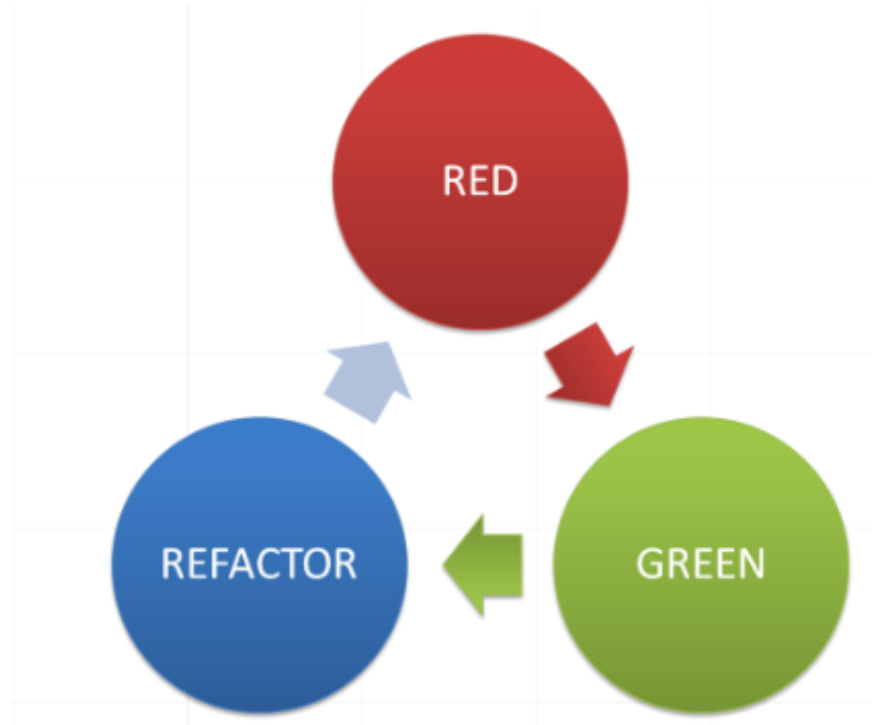
- A strategy for developing highly tested, quality software
- Not the be-all and end-all of strategies
- Google “TDD is dead” for a great argument against it
- Welcome to the still-forming world of software development!

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
 - This should immediately fail!
- Green
 - Write only enough code to make the test pass
- Refactor
 - Review code and make it better

Detailed Run-Through of RGR Loop

1. Write a test for new functionality
2. Run test suite - only the new test should fail
3. Write only enough code to make test pass
4. Run test suite
5. If any tests fail, go to step 3
6. Refactor code
7. Run test suite
8. If any tests fail, go to step 6
9. If any more functionality, go to step 1; otherwise done

Fizzbuzzin' With TDD

Print out the numbers from 1 to 100, each on a separate line. If a number is evenly divisible by 3, print "Fizz" instead. If a number is evenly divisible by 5, print "Buzz" instead. If a number is evenly divisible by 3 and 5, print "FizzBuzz" instead. Otherwise, just print the number.

1

2

Fizz

4

Buzz

Fizz

...

Start by adding a new test

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

... Which fails.



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!

Let's Add Another Test

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

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




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!

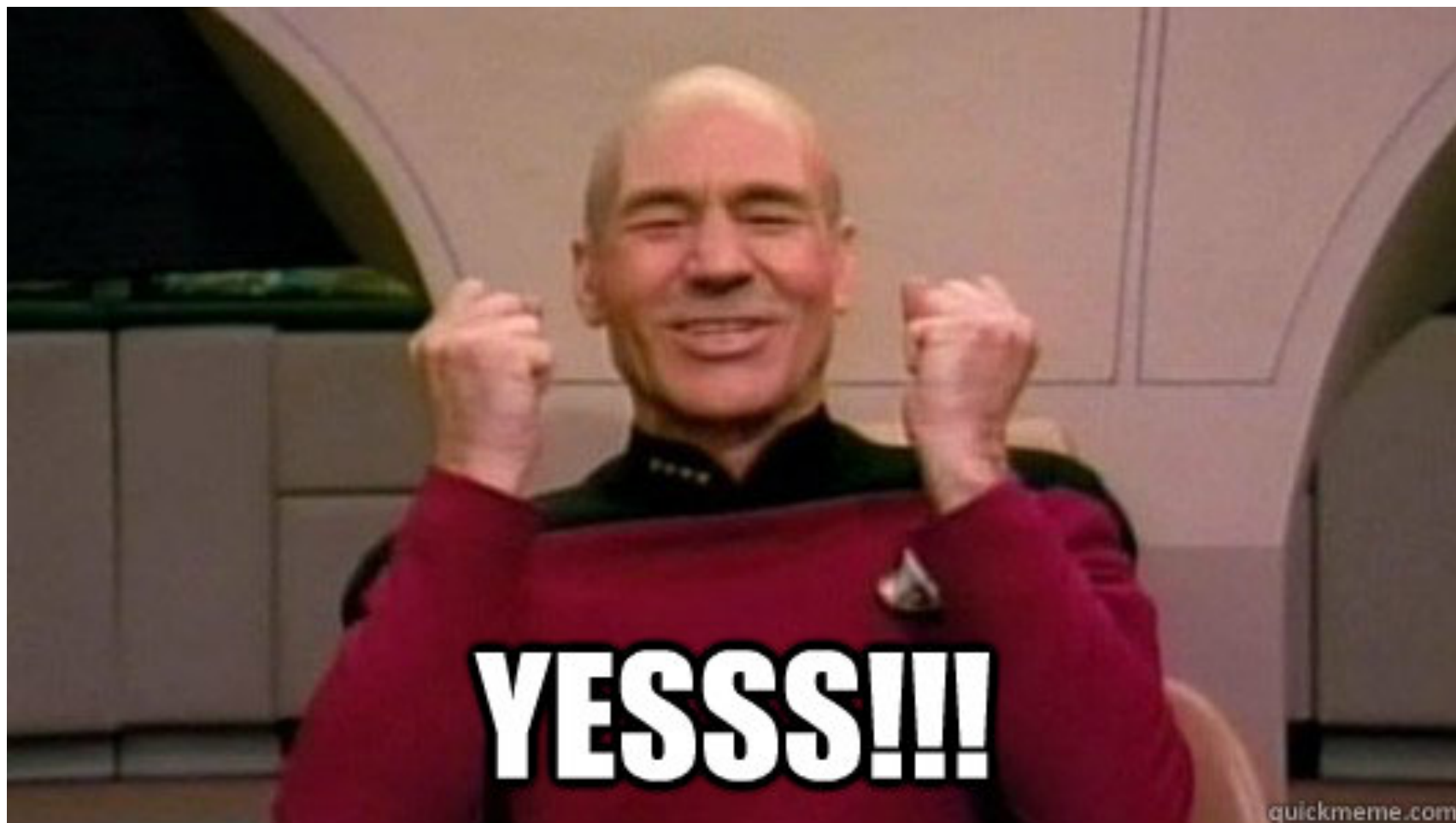
```
public String value(int n) {  
    return String.valueOf(n);  
}
```

Add Another Test – it fails

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

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);  
    }  
}
```



Let's Add A Test For Buzziness -
It should fail.

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


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);  
    }  
}
```



The Final Equivalence Class

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

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);  
    }  
}
```



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

TDD = A Kind of Test-First Development

- Basic idea is to think about expected behavior FIRST, before code
- You don't want to “corrupt” your mind with implementation details
- Figure out what the program should do (requirements!)
- Side note: there are other kinds of test-first development, such as ATDD (Acceptance Test Driven Development) and BDD (Behavior Driven Development)

Step 1 – Write a test

- This test should be a small unit of functionality, say one input value and output value for a method.
- For pure TDD, you should not write multiple tests or tests which are very complex.

Step 2 – Run Test Suite

- Run all the tests - only the one you've just added should fail.
- If it doesn't fail, you've already written the code for it! This might be a redundant test.
- If other tests fail, something weird happened. Completed tests should always be passing at this point.

Step 3 – Write the Code

- Write just enough code to have the test pass.
- Avoid the temptation to over-engineer your solution or add more functionality than the test covers!

Step 4 – Re-run the Test Suite

- All the tests should pass this time, assuming you actually added the functionality.
- Otherwise:
 - If only your new test fails:
 - You have not written your code (or possibly test) correctly.
 - If other tests fail:
 - You have created a regression failure; that is, you've broken other functionality on the system!
 - Note that these are not mutually exclusive!

Step 5 – Check Test Results

- If any tests fail, fix them – either tests or code!
- Never move on before having an ENTIRELY GREEN (i.e. passing) test suite!

Step 6 - Refactor

- 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?

Step 6 - Refactor

- Remember – you already have a working version before you refactor
 - We know it works because it provides the correct expected behavior according to the unit test suite
- When it comes to code, being right is more important than being good-looking

Step 7 – Re-run Test Suite Again

- Make sure that your refactoring did not cause any problems
- It should have the same functionality (according to the unit test suite), just better code
- That is, all unit tests should still pass

Step 8 – Check test results

- If any tests fail, something broke.
- Go and fix it before moving on!
- We are always aiming to have all-green tests

Step 9 - Done

Congratulations! You now have working code and can prove it with a test.

If there is more functionality to add, go back to step 1 and write a new test.

If not, SHIP IT.

YAGNI

- "You Ain't Gonna Need It"
- Don't add functionality you don't need right now. Chances are you won't need it and you're just going to waste time writing code for it.
- Code to the test!

KISS

- “Keep It Simple, Smarty-pants”
- Don't try to write overly complex, clever, over-engineered code. Make it easy to understand and modify.
- “Premature optimization is the root of all evil” –Donald Knuth

- Prefer:

```
i++;
```

over

```
i += (NUM_A / (c.getNum() - d.getNum()));
```

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;  
}
```

Avoid Slow-Running Tests

- Note that each iteration requires at least three test suite runs. If your tests take a long time to run, TDD is impractical.

Principles, Not Laws

- Nobody will throw you in jail if you write two tests during an iteration
- Sometimes tests are hard to make fast
- Etc.
- But they're code smells if you are using TDD.

Benefits of TDD

- Automatically create tests!
 - Research shows that more tests are correlated with fewer defects
- Makes writing tests easy because it's done often
 - Anything you do often, you learn how to do better
- Tests are relevant
 - They are testing the exact functionality you are implementing
- Developer is focused on end result, not code
 - Code is a way to get the functionality the user wants

Benefits of TDD

- Ensures that you take small steps
 - You know where defects lie; help localize errors
 - Research shows more senior engineers take smaller steps
- Code is extensible
 - You are already constantly extending the codebase
- Large test suite automatically created for you!
 - Helps avoid regression errors
 - High code coverage
- Confidence in the codebase

Drawbacks of TDD

- 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
- Extra up-front time
 - May be saved in large projects due to fewer defects / test coverage
- May not appropriate for prototyping
 - You may not always know expected behavior
- Hard to do large architectural changes
 - Some things just aren't possible to do in small steps

Drawbacks of TDD

- Complex or mission-critical systems will require a more robust testing strategy
- Tests become part of the overhead of the project
 - Especially if they are brittle/fragile, or poorly written!
- Could fall into trap of overtesting
 - More time-consuming test suite runs, which hurts productivity

Now Please Read Textbook Chapter 15