CS1632:
Test-Driven Development

Wonsun Ahn

TDD (Test-Driven Development)

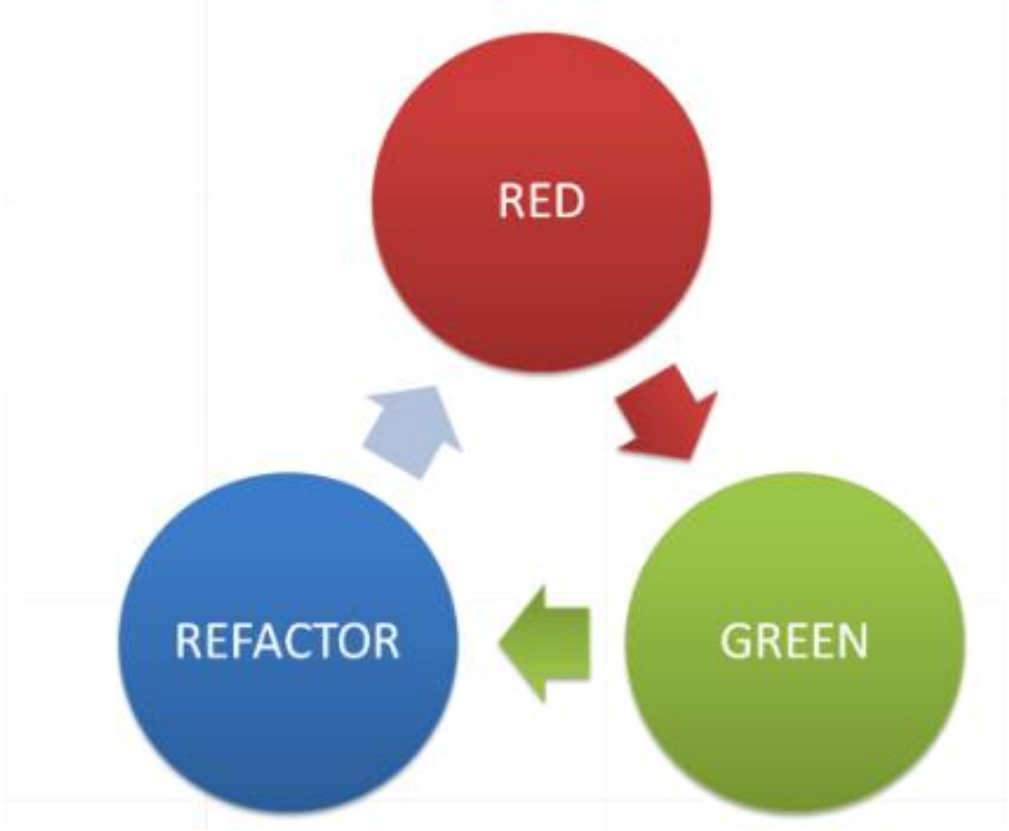
- It’s a development methodology – meant for developers
  - Developers are expected to participate actively in software QA nowadays
  - Software QA is an integral part of the development process

- TDD comprises of:
  - Writing a test BEFORE writing the code
  - Writing ONLY code to pass the test, before writing the next test

- The “test” here means unit test – TDD is synonymous with UTDD
  - UTDD: Unit Test-Driven Development
  - Code is tested and written one method at a time

The Red-Green-Refactor Loop

- TDD is performed using the RGR Loop:



The Red-Green-Refactor Loop

- Red – Write a test for a new behavior
  - 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

What is Refactoring?

- *Refactoring*: improving code without changing its functionality

- Even if your code is defect-free, it may not be perfect
  - Poor algorithm choice?
  - Bad variable names?
  - Poor performance?
  - Badly documented?
  - Magic numbers?
  - Not easily comprehensible?
  - General bad design?

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.
  - If number is neither a multiple of 3 and 5, app shall print the actual number.

- 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 “”;
}
```

Red – Which fails

Green - Write code to make test pass

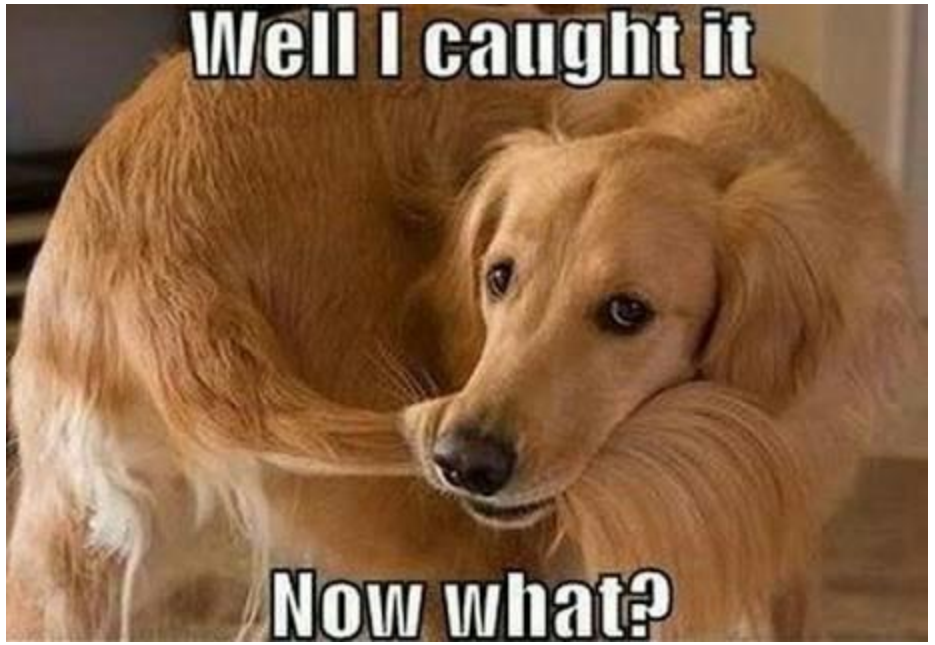
```
@Test
public void testNumber() {
  assertEquals(fb.value(1), “1”);
}
// Code
public String value(int n) {
  return “1”;
}
```

Green – Test passes!

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

Refactor – Nothing to do

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



Red – Well, let's Add Another Test for "2"

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

```
@Test
public void testNumber() {
   assertEquals(fb.value(1), "1");
}
```

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

Red – Which fails (again)

Green - Let's Make Another Change

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

@Test
public void testNumber() {
   assertEquals(fb.value(1), "1");
}

// Code

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

Green – Tests pass once more!

```
@Test
public void testNumber2() {
   assertEquals(fb.value(2), “2”);
}

@Test
public void testNumber() {
   assertEquals(fb.value(1), “1”);
}

// Code

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

Refactor – Much Nicer and Tests Still Pass!

```
@Test
public void testNumber2() {
    assertEquals(fb.value(2), “2”);
}

@Test
public void testNumber() {
    assertEquals(fb.value(1), “1”);
}

// Code

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

Red – Add another test for “3”, which fails

```
@Test
public void testNumber3() {
  assertEquals(fb.value(3), “Fizz”);
}
@Test
public void testNumber2() {
   assertEquals(fb.value(2), “2”);
}
@Test
public void testNumber() {
   assertEquals(fb.value(1), “1”);
}
// Code, omitted for brevity
```

Green – Add fizzy code to make test pass!

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

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

Green – Test suite passes again!

```
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 (fails)

```
@Test
public void testNumber5() {
  assertEquals(fb.value(5), "Buzz");
}
@Test
public void testNumber3() {
  assertEquals(fb.value(3), "Fizz");
}
@Test
public void testNumber2() {
   assertEquals(fb.value(2), "2");
}
@Test
public void testNumber() {
   assertEquals(fb.value(1), "1");
}
// Code, omitted for brevity
```

Green – Add buzzy code to make test pass!

```
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);
    }
}
```

Green – Now all tests pass once more!

```
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);
    }
}
```

Refactor – Again nothing to do

```
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 “FizzBuzz”

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

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

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

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

@Test
public void testNumber() {
   assertEquals(fb.value(1), “1");
}
```

Green – Add code to make input 15 pass too!

```
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);
   }
}
```

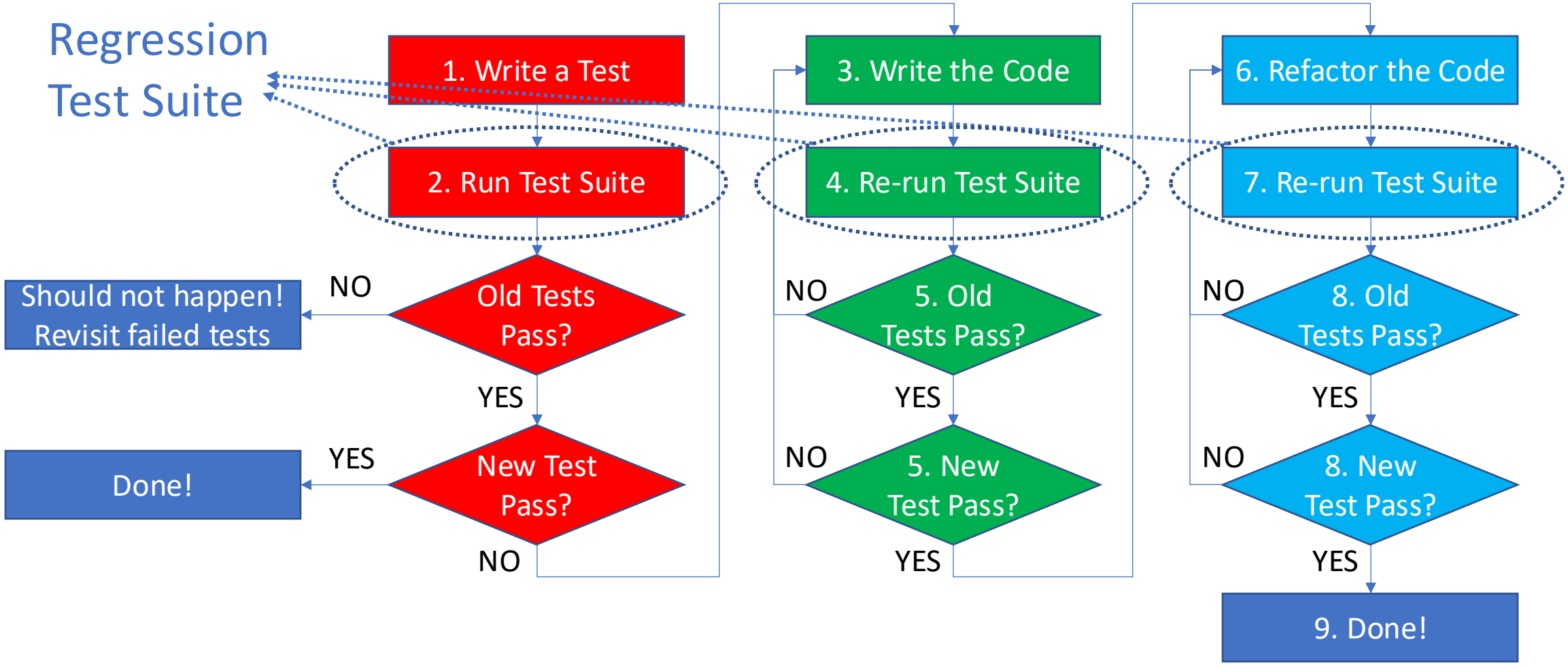
Green – We are back to a happy place

```
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);
   }
}
```

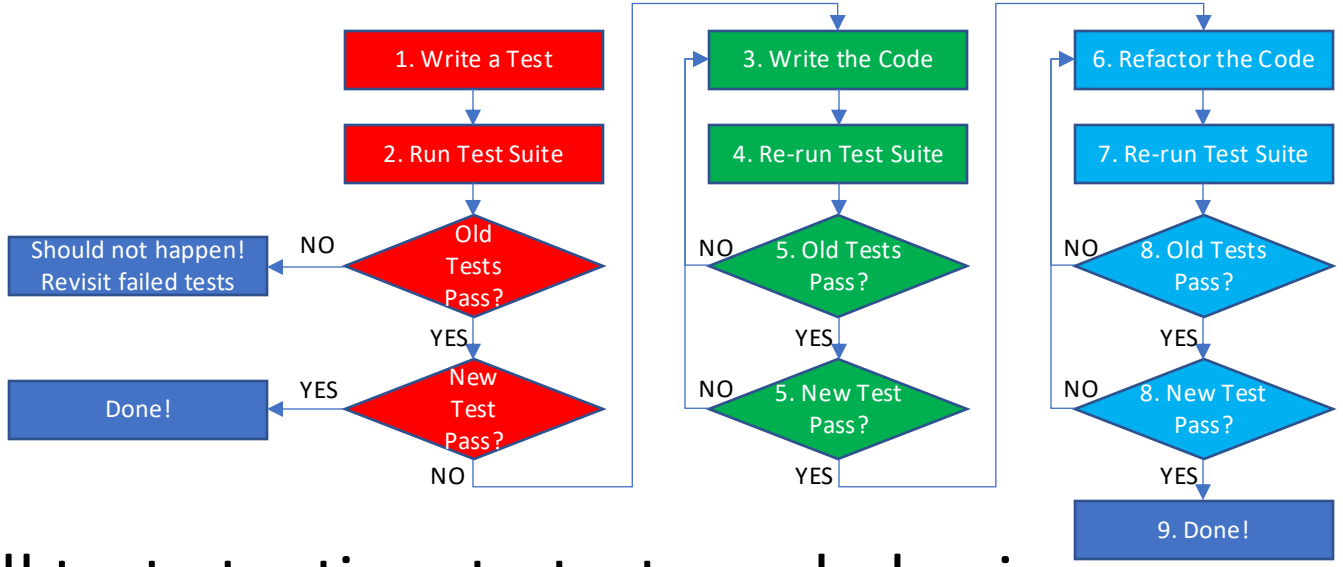
Refactor – Nothing to do, and we are done!

```
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);
   }
}
```

Flow Chart of Red-Green-Refactor Loop



Red-Green-Refactor Best Practices



- Write one small test at a time to test one behavior
  - Keep a tight turnaround cycle!
- Write just enough code to pass test
  - Resist temptation to go into untested territory
- Refactor at every iteration
  - Focus on correctness in Green phase; focus on improvement in Refactor phase

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 clever, over-engineered code unless tests call for it
  - It just makes it harder to understand and modify later
  - “Premature optimization is the root of all evil” – Donald Knuth

Benefits of TDD

1. Development is driven by requirements
   - Requirements drives → Testing drives → Development
   - End-result: all coding effort is tightly bound to the requirements

2. 100% test coverage all the time
   - Test to cover code is written before implementing the code
   - End-result: defects are caught immediately as they are introduced

3. Ensures development happens in small increments
   - Only code sufficient to pass the next test is written at a time
   - End-result: easy to debug code since you only to look at that small increment

Drawbacks of TDD

1. Tests become part of the overhead of the project
   - In terms of maintenance: test code is also code you need to maintain
   - In terms of testing time: sheer testing time slows down RGR loop

2. Hard to do large, complex 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

Now Please Read Textbook Chapter 15

- For an alternative view on TDD read the following:
  - “TDD is dead. Long live testing.” - David Heinemeier Hansson: https://dhh.dk/2014/tdd-is-dead-long-live-testing.html
  - Warning on being dogmatic about TDD
  - Like everything else in software QA, TDD is a tool not a religion