CS1632:
Test-Driven Development

Wonsun Ahn

THE DARK AGES

Bill Laboon

Enter TDD (Test-Driven Development)

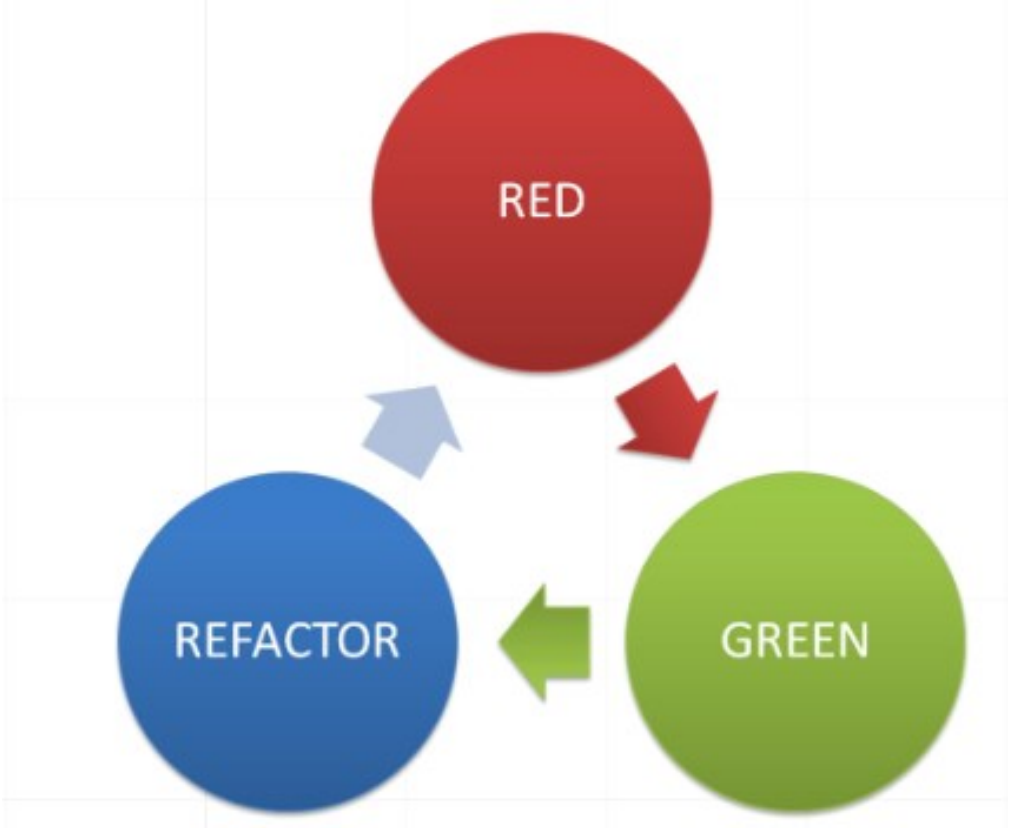
- *TDD*: A software development methodology that comprises of:
  - Writing tests BEFORE writing code
  - Writing ONLY code that is tested
  - A very short turnaround cycle
  - Refactoring early and often

- 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

- The "test" here means unit test – TDD is synonymous with UTDD
  - UTDD: Unit Test-Driven Development

The Red-Green-Refactor Loop

- TDD is performed using the RGR 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

What is Refactoring?

- 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?
- *Refactoring*: improving code without changing its functionality

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

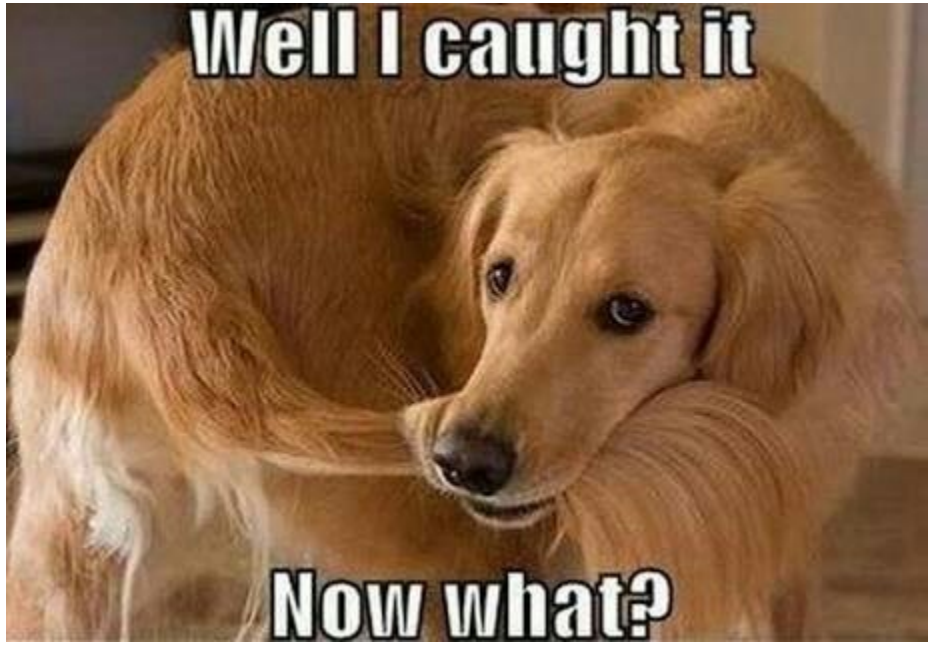
```
@Test
public void testNumber() {
  assertEquals(fb.value(1), “1”);
}
// Code
public String value(int n) {
  return “1”;
}
```

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);
    }
}
```

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);
   }
}
```

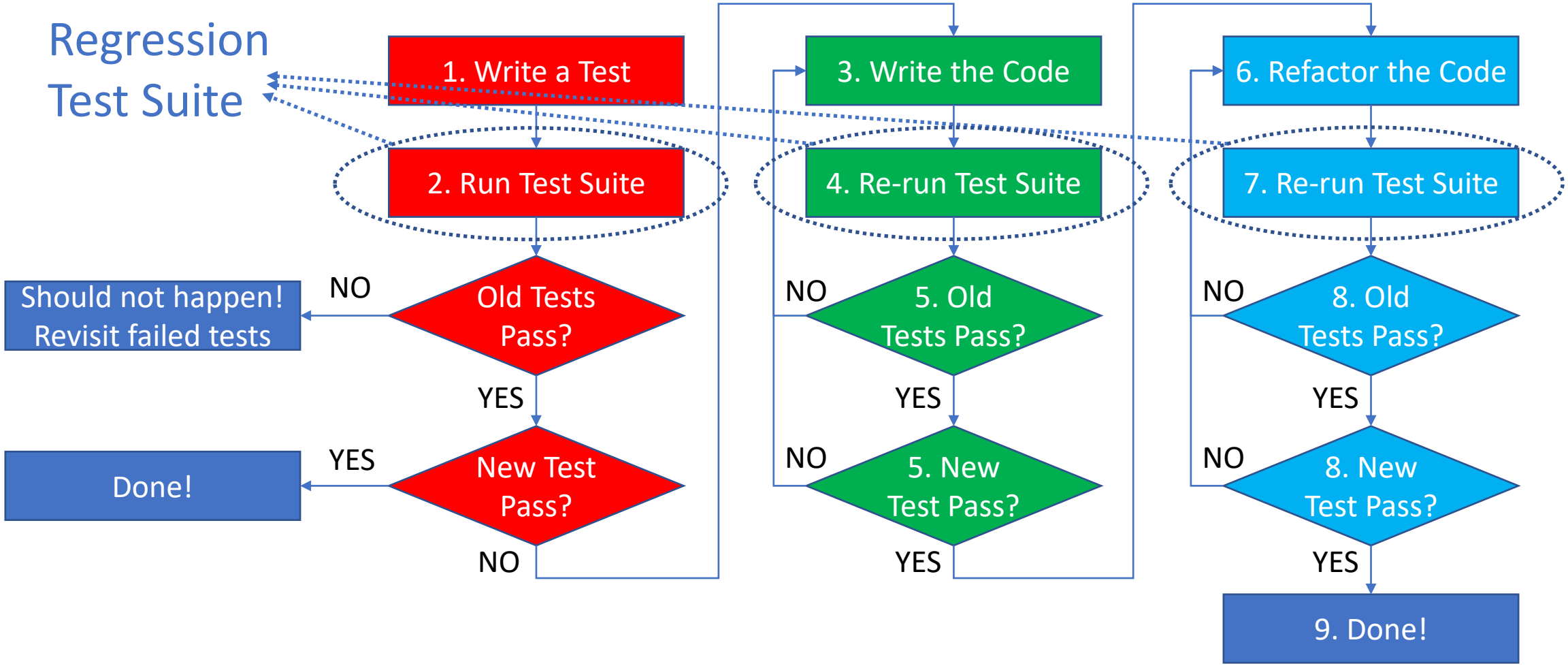
Refactor – Nothing to do

```
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);
   }
}
```

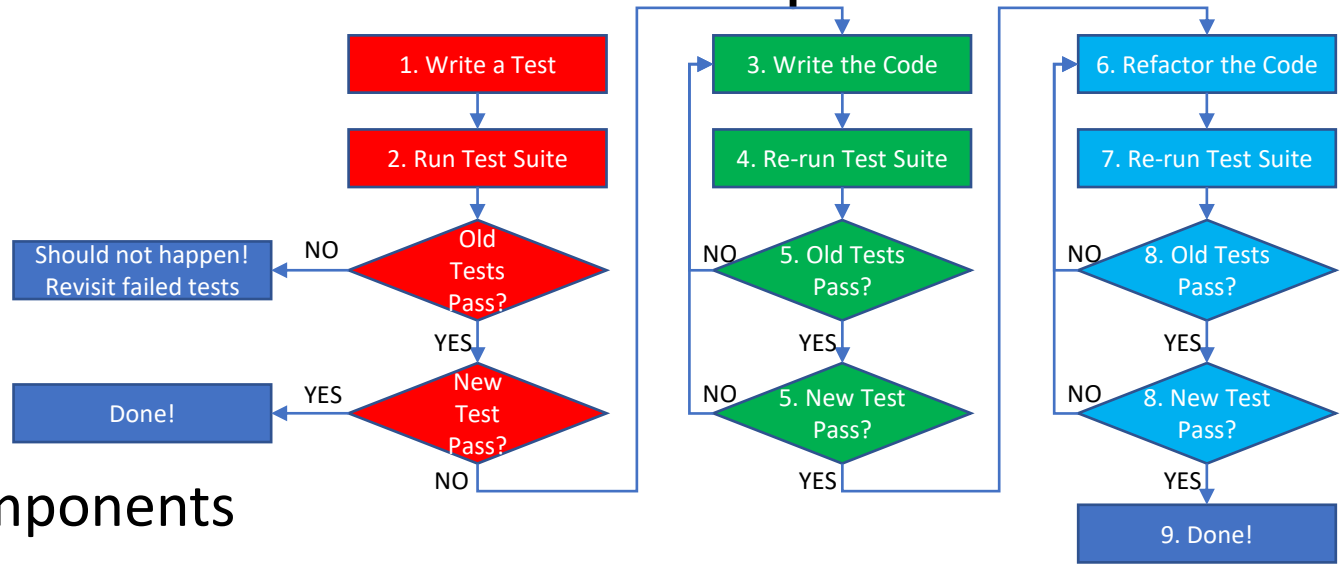
Takeaways

1. Development is driven by testing
   - 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. Regression testing all the time
   - Not only does the new test have to pass, all previous tests also must pass
   - End-result: software never regresses during development (at least in terms of the test cases)
4. 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
5. Software quality assurance integrated into development cycle
   - Not only is software continuously tested, refactoring also happens continuously
   - End-result: at end of loop, software is already high quality without extra effort

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 is continuously covered by testing
- Refactor at every iteration
  - Focus on correctness in Green phase; focus on improvement in Refactor phase
  - Don’t forget to regression test after refactoring

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 writing code 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 entire test suite on every code 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 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

- All coding effort is tightly bound to the requirements
- Code gets 100% test coverage all the time
- Code never regresses (where previous test cases fail)
- Ensures development happens in small increments (easier to debug)
- Software quality assurance integrated into development cycle

Drawbacks of TDD

- Tests become part of the overhead of the project
  - In terms of maintenance: especially if you use mocks to speed up testing
  - In terms of testing time: sheer testing time slows down RGR loop
  - If you stub to speed up testing, it increases maintenance cost. Sad ☹.
- 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
- Focus on unit tests may mean other aspects of testing get short shrift
  - Unit testing actively avoids integration testing through the use of mocks
  - Unit tests are code-centric and do not incorporate end-user experience

TDD = A Kind of Test-First Development

- Other kinds of test-first development that moves the focus away from unit testing
  - *ATDD* (Acceptance Test Driven Development): Make user acceptance tests part of the loop
    - Makes sure that software not only passes unit tests but is acceptable to user
  - *BDD* (Behavior Driven Development): Focus on behaviors (user stories) rather than unit tests
    - Behaviors described in English-like language (e.g. *Gherkin*)
    - Behaviors automatically compiled to testing code using a framework (e.g. *Cucumber*)

- Basic idea is still the same: to think about requirements FIRST, before coding
  - Write tests towards the requirements
  - Write code towards the tests

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