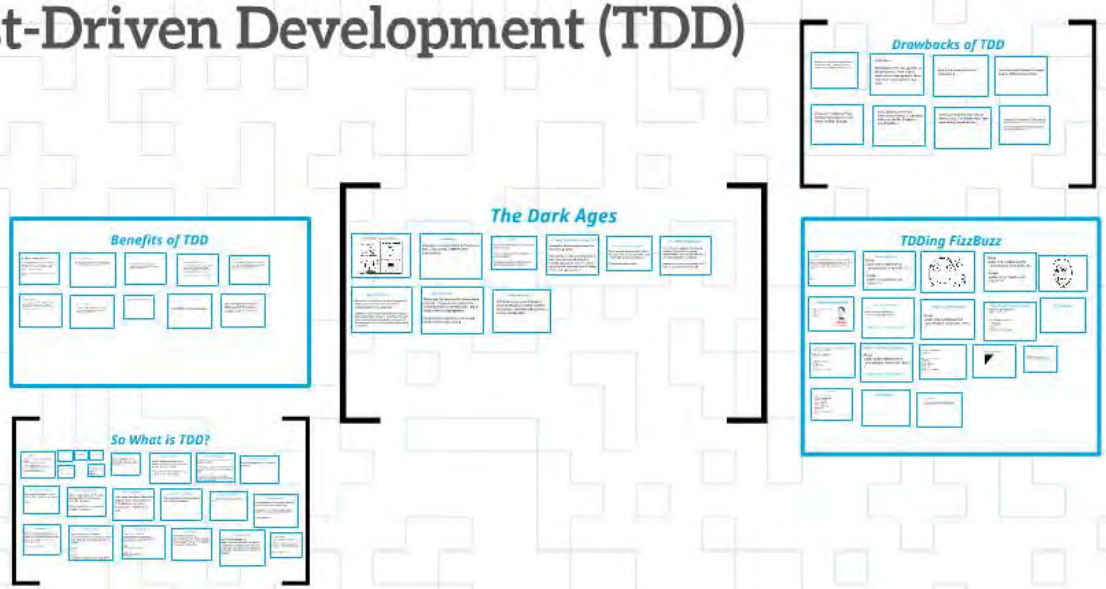
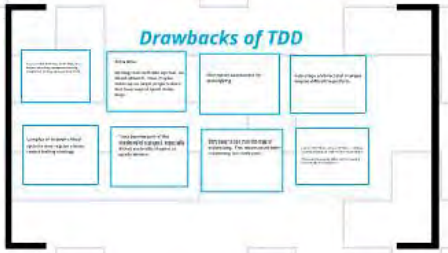
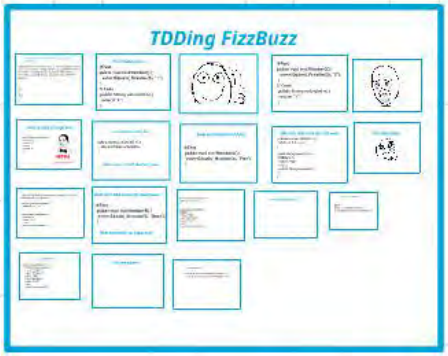
CS1699 - Lecture 9 - Test-Driven Development (TDD)



CS1699 - Lecture 9 - Test-Driven Development (TDD)





The Dark Ages

Back when I was a wee lad...

Nowadays...

Developers are responsible for testing, as well. Code quality is EVERYONE'S responsibility.

Nowadays...

It is the rare developer who does not even write unit tests.

Code is reviewed by other developers (static analysis).

Running test suites to make sure you didn't break anything is routine.

... but what is the testing strategy used?

Remember that testing needs to be done with purpose.

How should developers know what to test? How far they should go in testing edge cases? How to balance and prioritize unit testing and coding of the main application?

There is no one right answer

There are numerous studies, but so far, none have been conclusive on the "right" way to develop software.

There are no silver bullets.

Test-Driven Development

This is one strategy for developing quality software that is easily maintainable. It is not the end-all, be-all of development strategies.

However, it is currently popular and I have personally found it useful.

Just a reminder...

Note that I said that this is currently popular. COBOL also used to be one of the most popular programming languages.

Software development is a young, fast-paced, fast-moving field. If you're teaching it in a few years, most of the technologies and some of the strategies I am teaching will be ludicrously out-of-date.

Just a reminder...

This is why it's important to understand concepts. Arrays are not going to be rendered obsolete anytime soon. Big-O analysis won't be going away.

Understanding expected vs observed results is the key to testing.

That being said...

TDD makes sense and provides a structured way to develop software that almost automatically provides a built-in testing suite.

Back when I was a wee lad...

Nowadays...

Developers are responsible for testing, as well. Code quality is EVERYONE'S responsibility.

Nowadays...

It is the rare developer who does not even write unit tests.

Code is reviewed by other developers (static analysis).

Running test suites to make sure you didn't break anything is routine.

... but what is the testing strategy used?

Remember that testing needs to be done with purpose.

How should developers know what to test? How far they should go in testing edge cases? How to balance and prioritize unit testing and coding of the main application?

There is no one right answer.

There are numerous studies, but so far, none have been conclusive on the "right" way to develop software.

There are no silver bullets.

Test-Driven Development

This is one strategy for developing quality software that is easily maintainable. It is not the end-all, be-all of development strategies.

However, it is currently popular and I have personally found it useful.

Just a reminder...

Note that I said that this is currently popular. COBOL also used to be one of the most popular programming languages.

Software development is a young, fast-paced, fast-moving field. If you're teaching it in a few years, most of the technologies and some of the strategies I am teaching will be ludicrously out-of-date.

Just a reminder...

This is why it's important to understand concepts. Arrays are not going to be rendered obsolete anytime soon. Big-O analysis won't be going away.

Understanding expected vs observed results is the key to testing.

That being said...

TDD makes sense and provides a structured way to develop software that almost automatically provides a built-in testing suite.

So What is TDD?

TDD =

A software development methodology that comprises:
1. Writing tests first, before code
2. Writing only code that is tested
3. Writing only tests that test the code
4. A short turnaround cycle
5. Refactoring early and often

Step 0 - Write a Test

TDD is a kind of TFD (test-first development). The key principle is that you think of *what to test* before *what code to write*.

Step 1 - Write a Test

This test should be a small unit of functionality, say one input value and output value for a method.

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 is most likely 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.

Step 4 - Run the Test Suite

Run the test suite again. If all is green, great! Go on to the next step.

If not...

Step 5 - If Any Tests Fail...

If your newly-added test fails, you obviously didn't write the code correctly. Try again!

If other tests fail, you messed other things up. Try again!

Step 6 - Refactor

Your code may have been a bit ugly or hacky, or just doesn't fit in with the rest of the architecture. Fix it in this step.

Step 7 - Run Test Suite

Make sure your refactoring didn't cause other problems!

Step 8 - If Any Tests Fail...

Sorry, your refactoring broke something. Try again!

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.

Principles of TDD

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 only!

Principles of TDD

KISS - "Keep it simple, stupid!"
Don't try to write overly complex, clever code. Make it easy to understand and modify.

Prefer:
i++;
over
i += (NUM_A / (c.getNum() - d.getNum));

Principles of TDD

"Fake it 'til you make it"
If you can make a test pass with a constant, so be it. You'll want to fix it with more tests, of course.

Test:
assertEquals(sqrt(4), 2);
Code:
public void sqrt(int n) {
  return 2; }

Principles of TDD

Avoid Interdependency
One test should not rely on another succeeding or failing. They should be as separate as possible.

Principles of TDD

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 useless.

Principles of TDD

Finally, remember -
These are principles, not laws.

They do not need to be followed in lockstep. Sometimes tests are slow. Sometimes you add an unnecessary line.

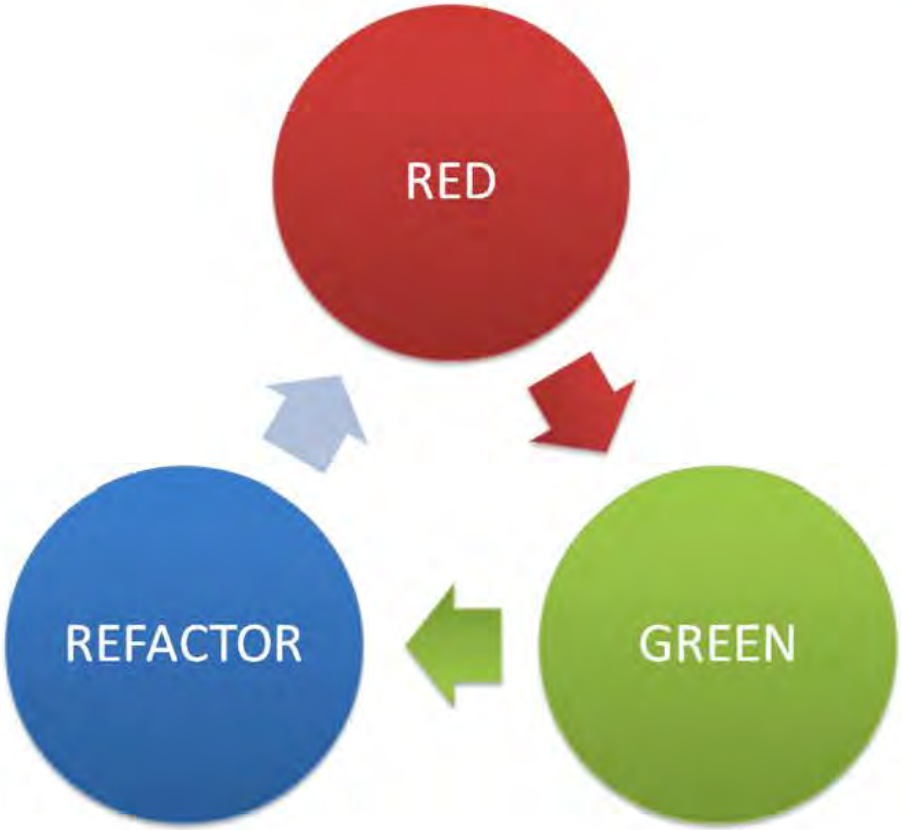
But these are code smells.

TDD =

A software development methodology that comprises:

1. **Writing tests first, before code**
2. **Writing only code that is tested**
3. **Writing only tests that test the code**
4. **A short turnaround cycle**
5. **Refactoring early and often**

Often referred to as the "red-green-refactor" loop.



Red - You write a test which should immediately fail.

Green = You write code until the tests pass.

Refactor - You arrange the code better, and think about better ways to do things.

"First make it green, then make it clean."

Example:

1. Write a test
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

Step 1 - Write a Test

TDD is a kind of TFD (test-first development). The key principle is that you think of *what to test* before *what code to write*.

Step 1 - Write a Test

This test should be a small unit of functionality, say one input value and output value for a method.

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 is most likely 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.

Step 4 - Run the Test Suite

Run the test suite again. If all is green, great! Go on to the next step.

If not...

Step 5 - If Any Tests Fail...

If your newly-added test fails, you obviously didn't write the code correctly. Try again!

If other tests fail, you messed other things up. Try again!

Step 6 - Refactor

Your code may have been a bit ugly or hacky, or just doesn't fit in with the rest of the architecture. Fix it in this step.

Step 7 - Run Test Suite

Make sure your refactoring didn't cause other problems!

Step 8 - If Any Tests Fail...

Sorry, your refactoring broke something. Try again!

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.

Principles of TDD

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 only!

Principles of TDD

KISS - "Keep it simple, stupid!"

Don't try to write overly complex, clever code. Make it easy to understand and modify.

Prefer:

```
i++;
```

over

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

Principles of TDD

"Fake it 'til you make it"
If you can make a test pass with a constant, so be it. You'll want to fix it with more tests, of course.

Test:

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

Code:

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

Principles of TDD

Avoid Interdependency

One test should not rely on another succeeding or failing. They should be as separate as possible.

Principles of TDD

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 useless.

Principles of TDD

Finally, remember -
These are principles, not laws.

They do not need to be followed in lockstep. Sometimes tests are slow. Sometimes you add an unnecessary line.

But these are code smells.

Benefits of TDD

Almost automatically create tests!

When testing becomes part of the normal workflow, more tests are written. More tests have been correlated with higher quality software and fewer defects.

When it's easy to write tests, more tests are written.

Tests are relevant!

Your tests aren't random. They test actual functionality that you've written.

The developer thinks about what the program is supposed to do, not about the specifics of his/her code.

Ensures you take small steps.

Nothing is worse than not knowing where a bug lies in a 5 megaSLOC project. (OK, technically, I suppose there are actually worse things. But still, it's not fun).

TDD allows you to localize errors easily.

Code is testable.

Since you're writing it with tests in mind, you can make nice, modular code that has built-in test hooks. This also helps for proper modularization of the codebase.

Code is extensible.

Code is never completed in massive chunks; by its very nature, you are constantly extending the codebase. You never assume that things are complete.

Helps avoid defects.

Constantly testing and making sure that you don't break other code can do wonders for avoiding regression bugs.

100% (or close to it) code coverage.

Confidence in your codebase.

TDD provides a structured method of doing things, almost like checkboxes (see the Checkbox Manifesto). It's an easy way to ensure you're not missing anything in your development process.

Almost automatically create tests!

When testing becomes part of the normal workflow, more tests are written. More tests have been correlated with higher quality software and fewer defects.

When it's easy to write tests, more tests are written.

Tests are relevant!

Your tests aren't random. They test actual functionality that you've written.

The developer thinks about what the program is supposed to do, not about the specifics of his/her code.

Ensures you take small steps.

Nothing is worse than not knowing where a bug lies in a 5 megaSLOC project. (OK, technically, I suppose there are actually worse things. But still, it's not fun).

TDD allows you to localize errors easily.

Code is testable.

Since you're writing it with tests in mind, you can make nice, modular code that has built-in test hooks. This also helps for proper modularization of the codebase.

Code is extensible.

Code is never completed in massive chunks; by its very nature, you are constantly extending the codebase. You never assume that things are complete.

Helps avoid defects.

Constantly testing and making sure that you don't break other code can do wonders for avoiding regression bugs.

100% (or close to it) code coverage.

Confidence in your codebase.

TDD provides a structured method of doing things, almost like checkboxes (see the Checkbox Manifesto). It's an easy way to ensure you're not missing anything in your development process.

Drawbacks of TDD

Focus on unit tests may mean that other aspects of testing (acceptance testing, integration testing, etc.) get short shrift.

Extra time.

Writing tests will take up time, no doubt about it. Time may be made up on larger projects since less time may be spent fixing bugs.

May not be appropriate for prototyping.

Late-stage architectural changes may be difficult to perform.

Complex or mission-critical systems may require a more robust testing strategy.

Tests become part of the overhead of a project, especially if they are brittle (fragile) or poorly written.

Very easy to fall into the trap of overtesting. This means more time-consuming test suite runs.

Can be difficult to implement TDD on existing projects developed with a different paradigm.

TDD assumes easy-to-write and fast testing frameworks and platforms.

Focus on unit tests may mean that other aspects of testing (acceptance testing, integration testing, etc.) get short shrift.

Extra time.

Writing tests will take up time, no doubt about it. Time may be made up on larger projects since less time may be spent fixing bugs.

May not be appropriate for prototyping.

Late-stage architectural changes may be difficult to perform.

Complex or mission-critical systems may require a more robust testing strategy.

Tests become part of the overhead of a project, especially if they are brittle (fragile) or poorly written.

Very easy to fall into the trap of overtesting. This means more time-consuming test suite runs.

Can be difficult to implement TDD on existing projects developed with a different paradigm.

TDD assumes easy-to-write and fast testing frameworks and platforms.

TDDing FizzBuzz

You should know what FizzBuzz is by now, but as a refresher...

Print out the numbers from 1 to 100, each on a separate line.
Except 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.

```
1
2
Fizz
4
Buzz
Fizz
...
```

First things first...

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

```
@Test
public void testNumber2() {
  assertEquals(_fb.value(2), "2");
}
// Code
 public String value(int n) {
  return "1";
 }
```

A little code change and...

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

NOT BAD

...but could be better!

Let's refactor this a bit.

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

Much nicer.. and all tests still pass!

New test added and fails..

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

Need to add code for Fizz-ness

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

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

All tests pass!

Note that we added a private method, fizzy(n), but we are not testing it as it is private.

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

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

Now let's add a test for buzzyness.

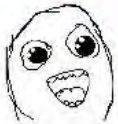
```
@Test
public void testNumber5() {
 assertEquals(_fb.value(5), "Buzz");
}
```

New test fails, as expected.

```
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);
 }
}
```

Now the tests pass!



...now for "Fizzbuzz"...

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

...now modify the code

```
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);
 }
}
```

All tests pass!

...and continue.

Note that we've automatically developed a test suite in the process of writing the code.

You should know what FizzBuzz is by now, but as a refresher...

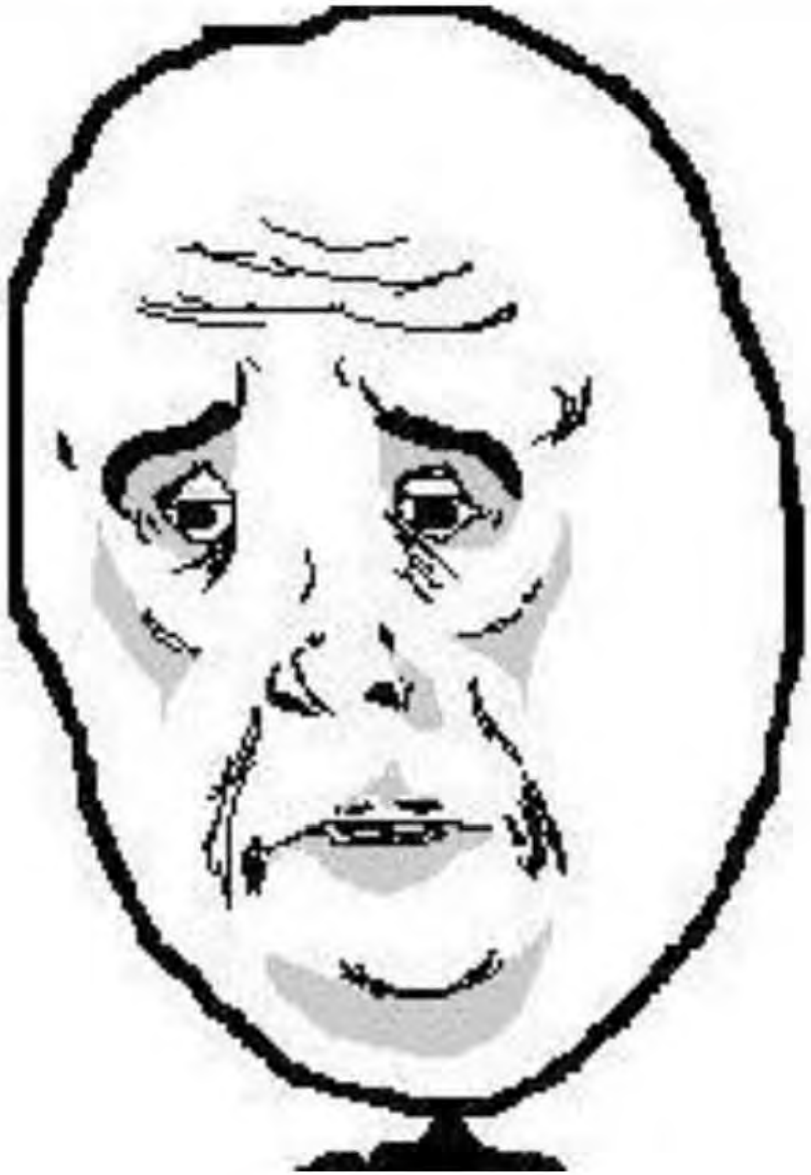
Print out the numbers from 1 to 100, each on a separate line.
Except if a number is evenly divisible by 3, print "Fizz" instead.
If a number is evenly divisible by 5, print "Buzz" instead.
If a number is evenly divisible by 3 and 5, print "FizzBuzz" instead.

1
2
Fizz
4
Buzz
Fizz
...

First things first...

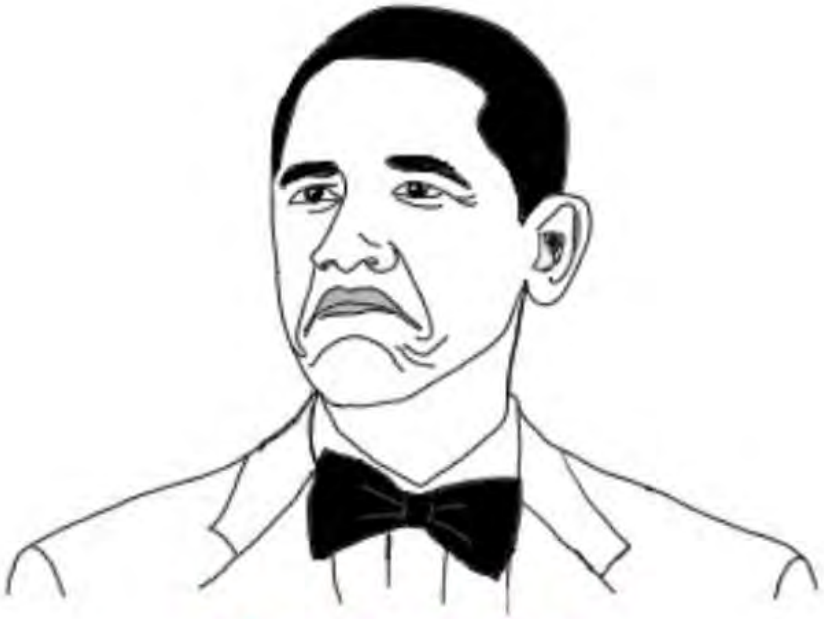
```
@Test
public void testNumber() {
   assertEquals(_fb.value(1), "1");
 }
// Code
 public String value(int n) {
  return "1";
 }
```

```
@Test
public void testNumber2() {
   assertEquals(_fb.value(2), "2");
 }
// Code
 public String value(int n) {
  return "1";
 }
```



A little code change and...

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



NOT BAD

... but could be better!

Let's refactor this a bit.

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

Much nicer.. and all tests still pass!

New test added and fails..

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

Need to add code for Fizz-ness

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

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

All tests pass!

Note that we added a private method, fizzy(n), but we are not testing it as it is private.

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

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

Now let's add a test for buzzyness.

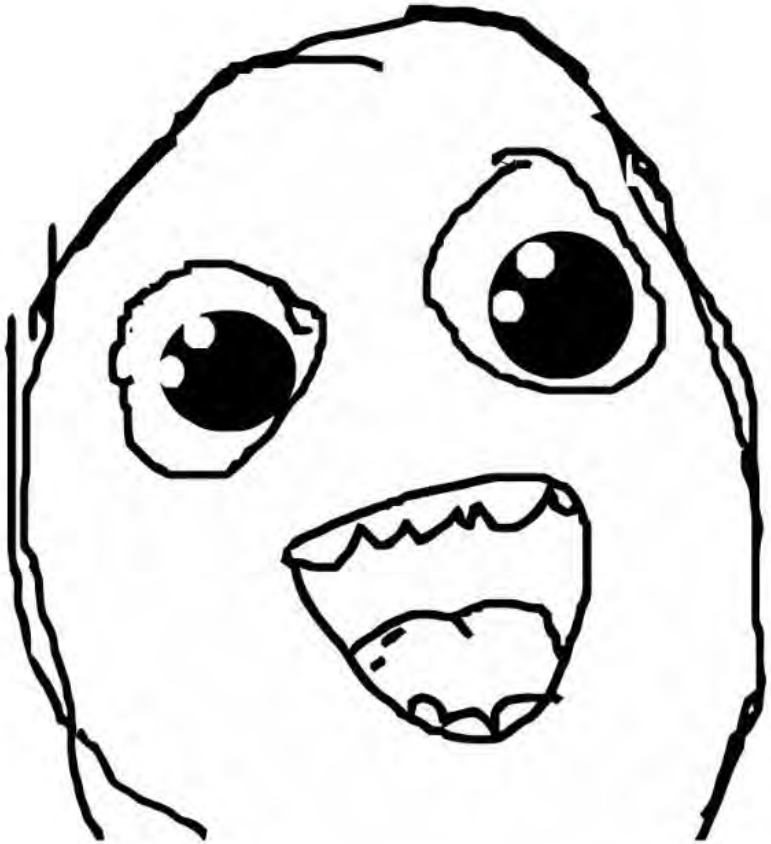
```
@Test
 public void testNumber5() {
  assertEquals(_fb.value(5), "Buzz");
 }
```

New test fails, as expected.

```
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);
 }
}
```

Now the tests pass!



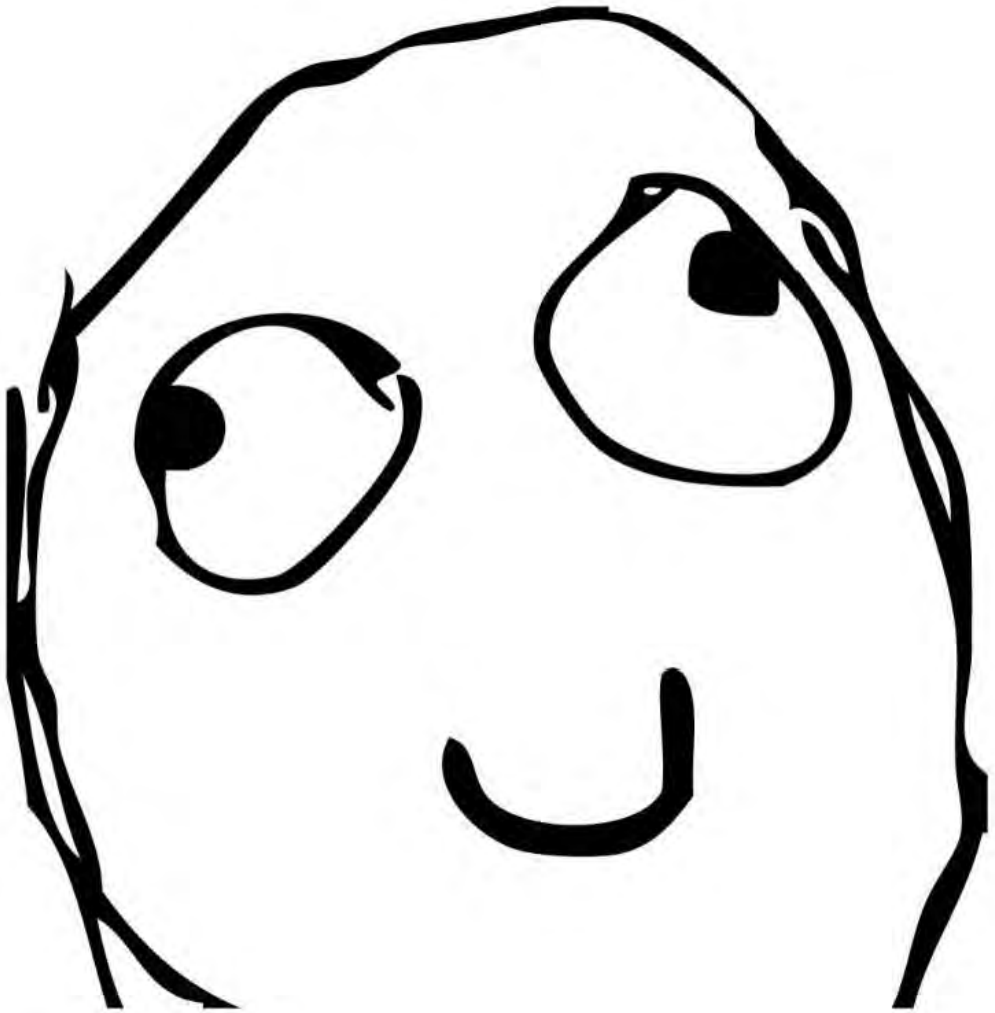
OK, now for "FizzBuzz"...

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

... now modify the code.

```
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);
  }
 }
```

All tests pass!



... and continue.

Note that we've automatically developed a test suite in the process of writing the code.

CS1699 - Lecture 9 - Test-Driven Development (TDD)