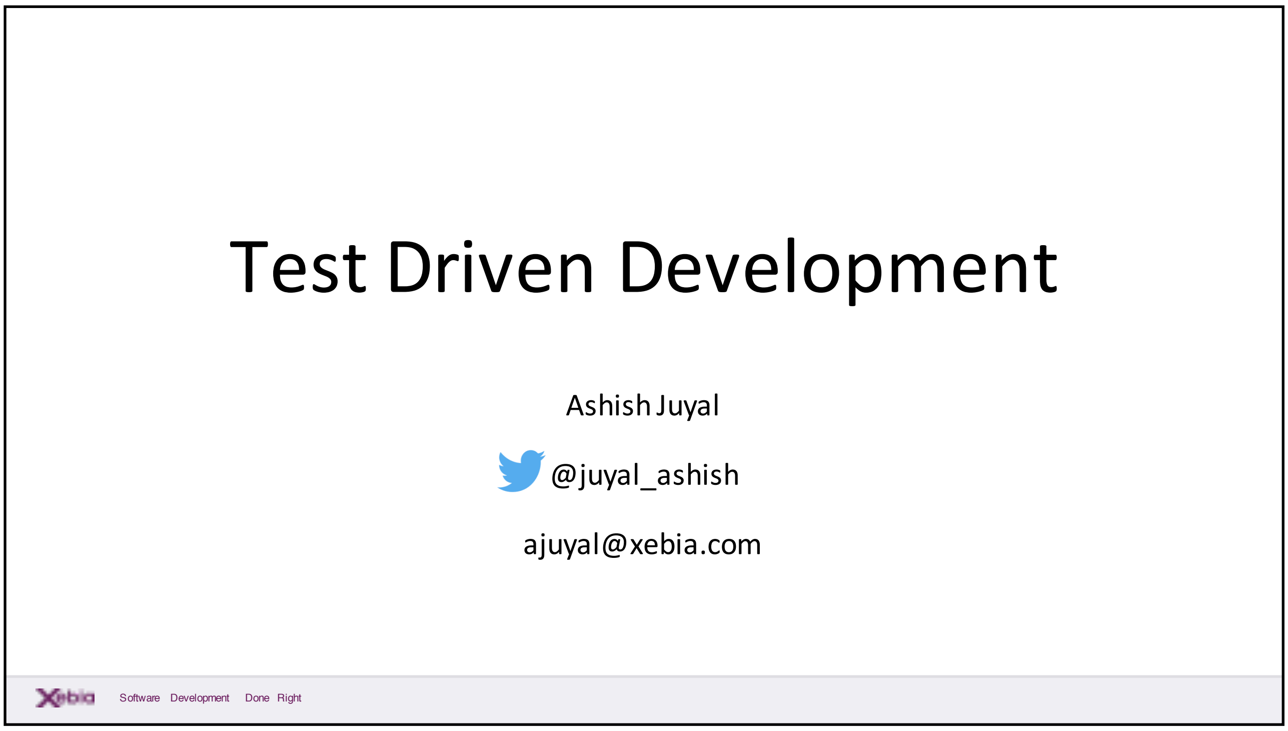
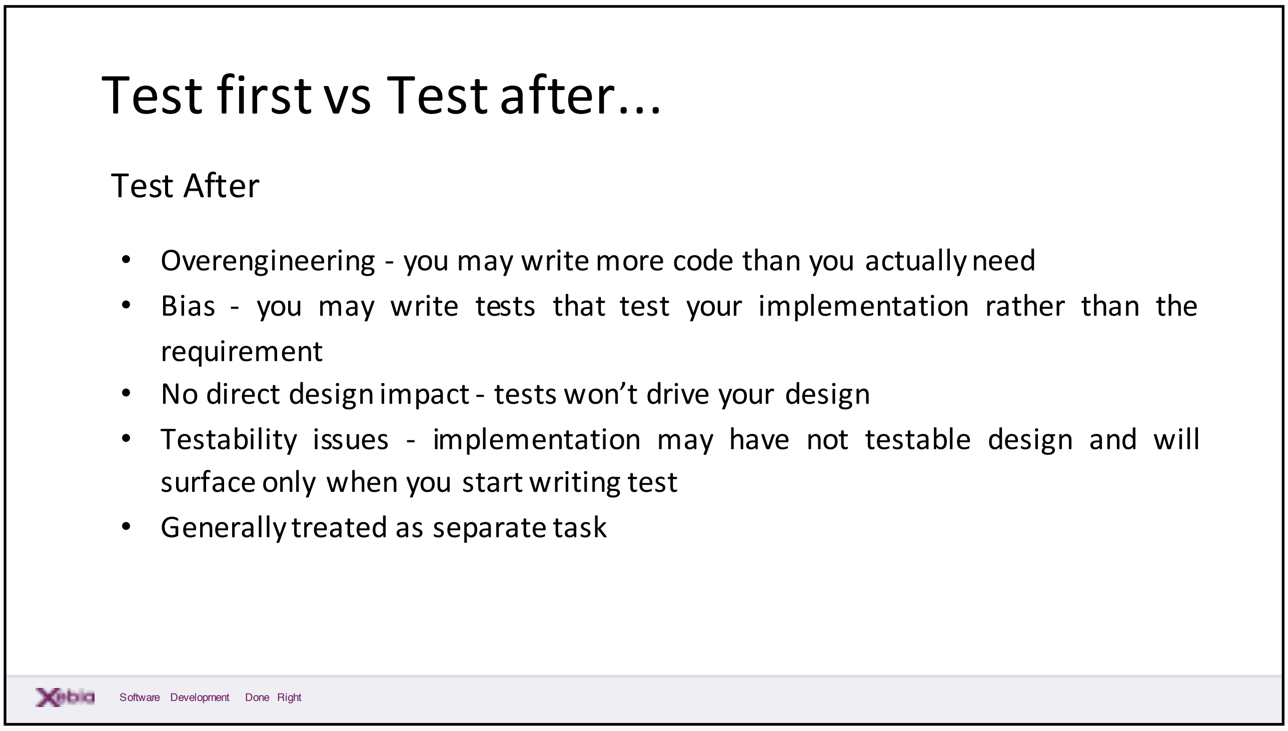


Test Driven Development

Ashish Juyal

@juyal_ashish

ajuyal@xebia.com



Test first vs Test after...

Test After

- Overengineering - you may write more code than you actually need
- Bias - you may write tests that test your implementation rather than the requirement
- No direct design impact - tests won't drive your design
- Testability issues - implementation may have not testable design and will surface only when you start writing test
- Generally treated as separate task

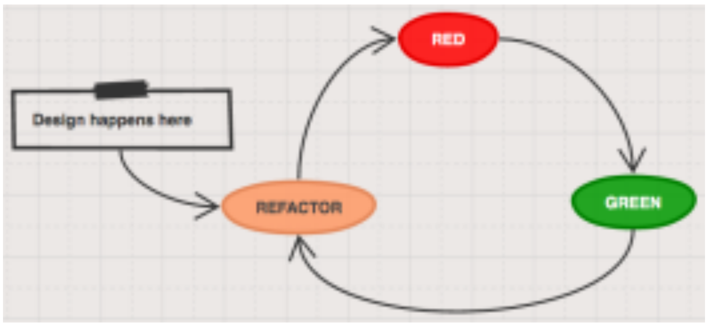
Test first vs Test after...

Test First

- Makes us clarify the acceptance criteria. We have to ask ourselves how we can tell when we're done
- Encourages us to write loosely coupled components, so they can easily be tested in isolation and, at higher levels, combined together
- Adds an executable description of what the code does
- Adds to a complete regression suite

Classicist TDD

- State based test, write a test, get a state and assert on state
- Good for exploration

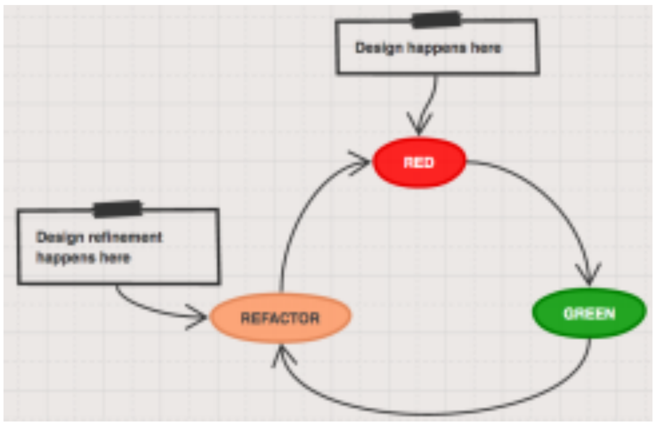


Examples

```
@Test
public void test() {
    assertEquals("cba", Util.reverse(input));
}

@Test
public void test() {
   Component component = new Component();
   component.setData("some data");
   assertEquals("some data", component.getData());
}
```

Outside – IN

- Mocks / test doubles are used at the boundaries of the system

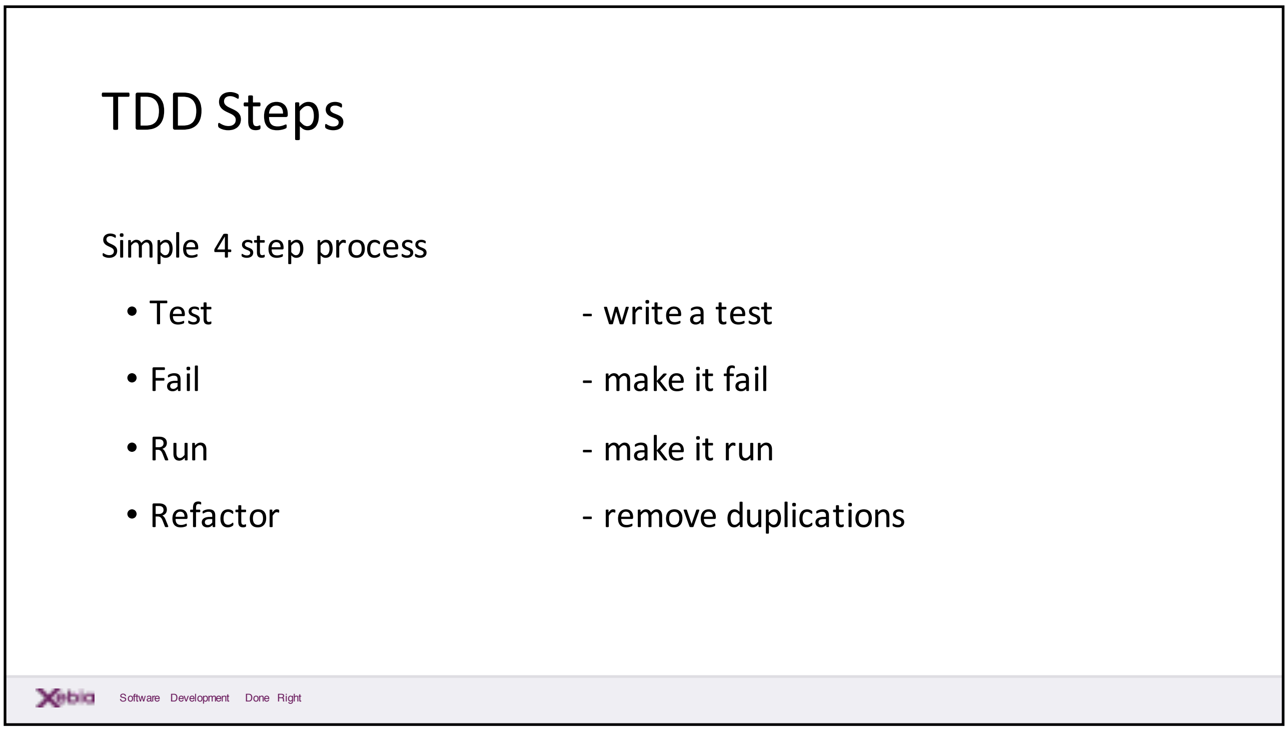


Examples

```
@Test
public void test() {
  // mock creation
  List mockedList  = mock(List.class);

  mockedList.add("one");
  mockedList.clear();

  // testing interaction
  verify(mockedList).add("one");
  verify(mockedList).clear();
}
```

Uncle Bob’s 3 rules for TDD

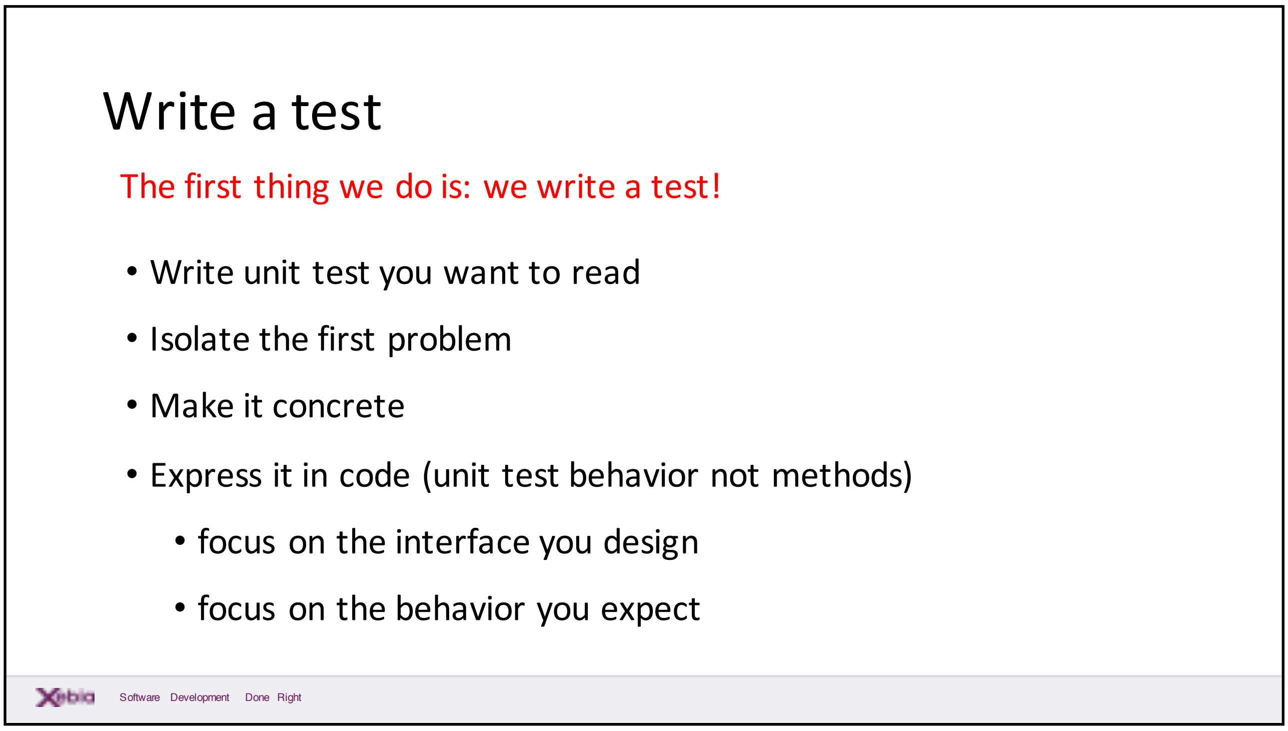
- You cannot write any production code until you have a failing test [Not compiling is failing]
- You are not allowed to write more in a unit test than is sufficient to fail, as soon as the unit test fails or fails to compile, you must stop writing test and start writing production code
- You are not allowed to write more production code than is sufficient to pass currently failing test



TDD Steps

Simple 4 step process

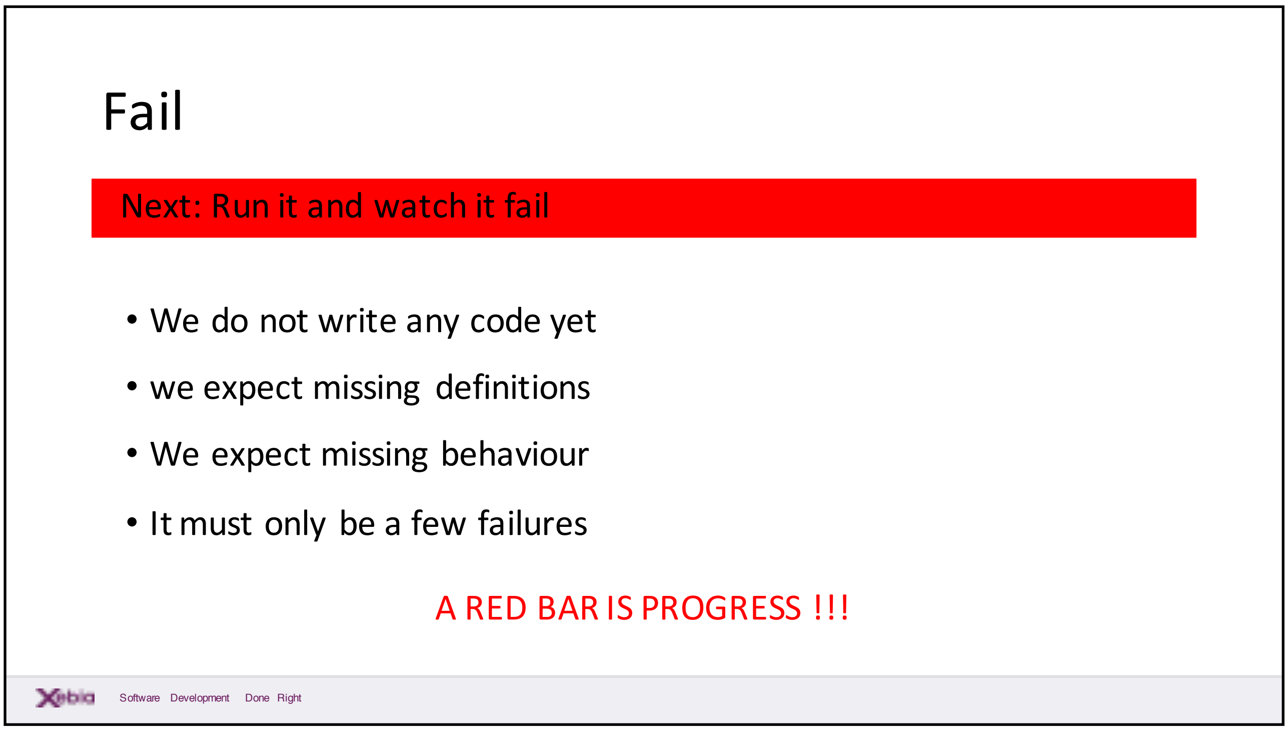
• Test	- write a test
• Fail	- make it fail
• Run	- make it run
• Refactor	- remove duplications



Write a test

The first thing we do is: we write a test!

- Write unit test you want to read
- Isolate the first problem
- Make it concrete
- Express it in code (unit test behavior not methods)
  - focus on the interface you design
  - focus on the behavior you expect

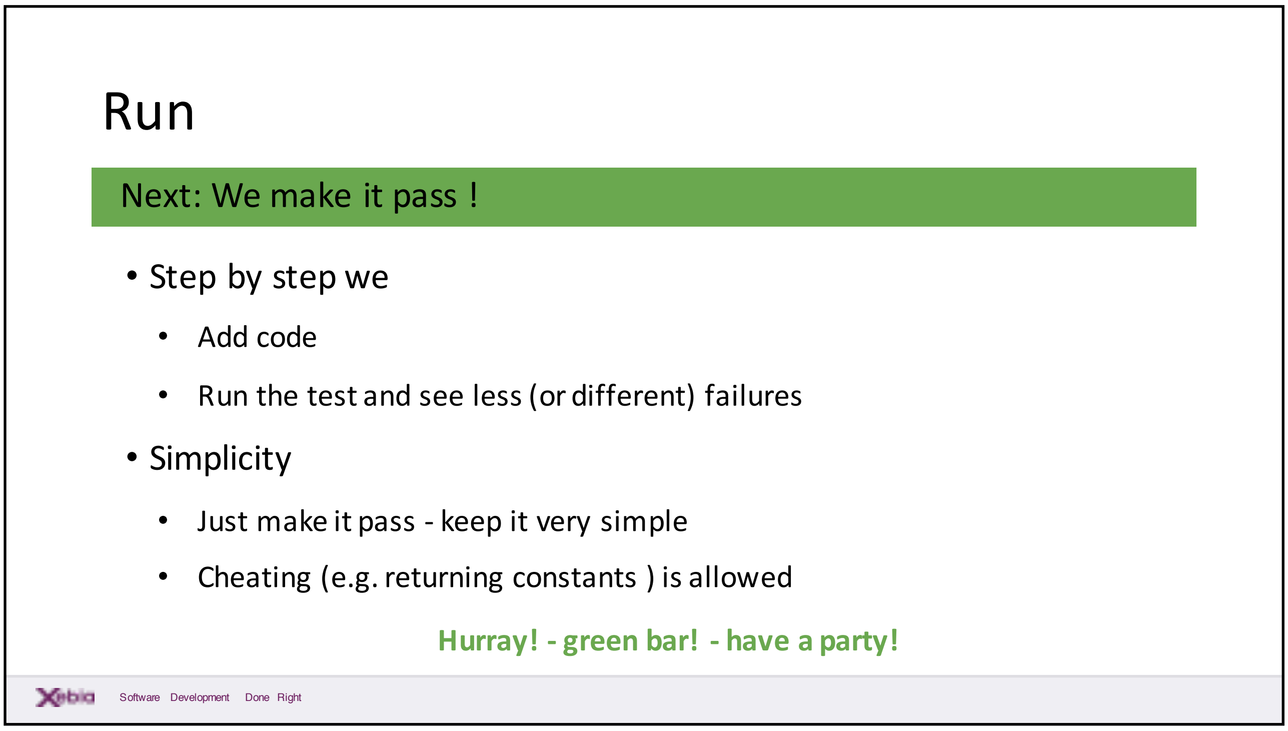


Fail

Next: Run it and watch it fail

- We do not write any code yet
- we expect missing definitions
- We expect missing behaviour
- It must only be a few failures

A RED BAR IS PROGRESS !!!

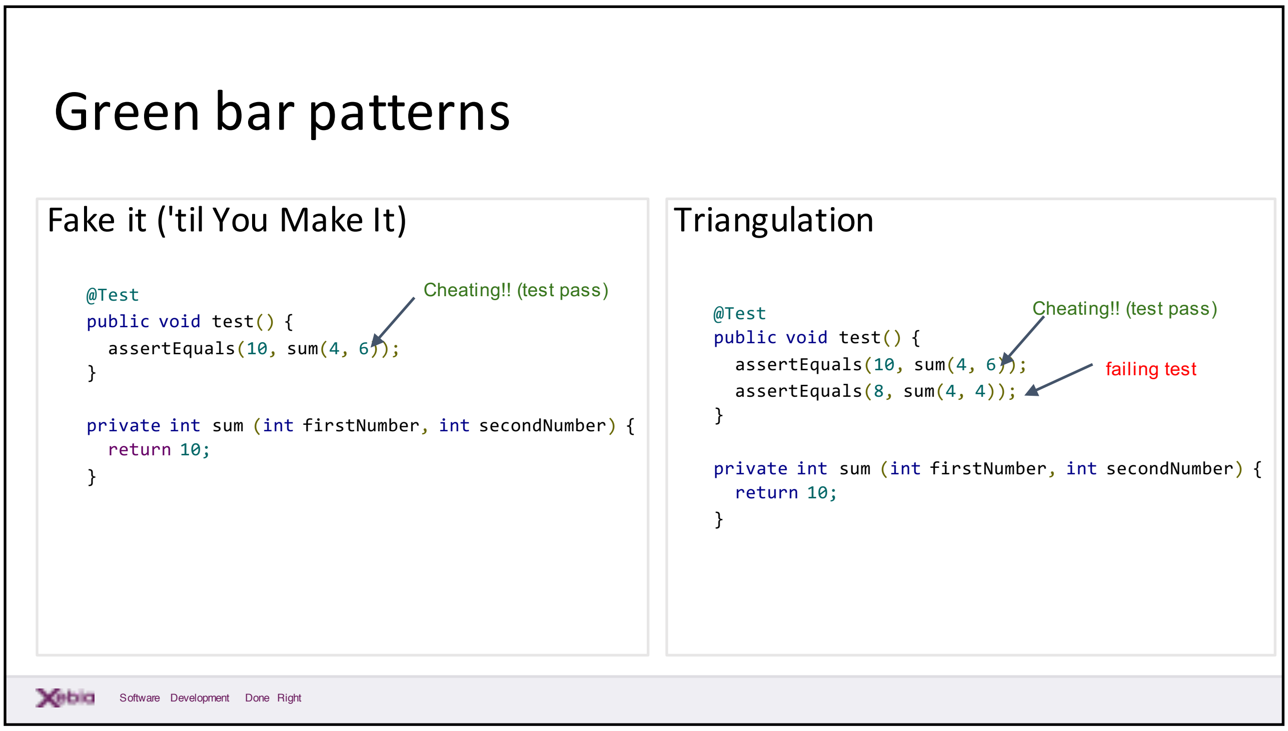


Run

Next: We make it pass !

- Step by step we
  - Add code
  - Run the test and see less (or different) failures
- Simplicity
  - Just make it pass - keep it very simple
  - Cheating (e.g. returning constants) is allowed

Hurray! - green bar! - have a party!



Green bar patterns

Fake it ('til You Make It)

```
@Test
public void test() {
  assertEquals(10, sum(4, 6));
}

private int sum (int firstNumber, int secondNumber) {
  return 10;
}
```

Cheating!! (test pass)

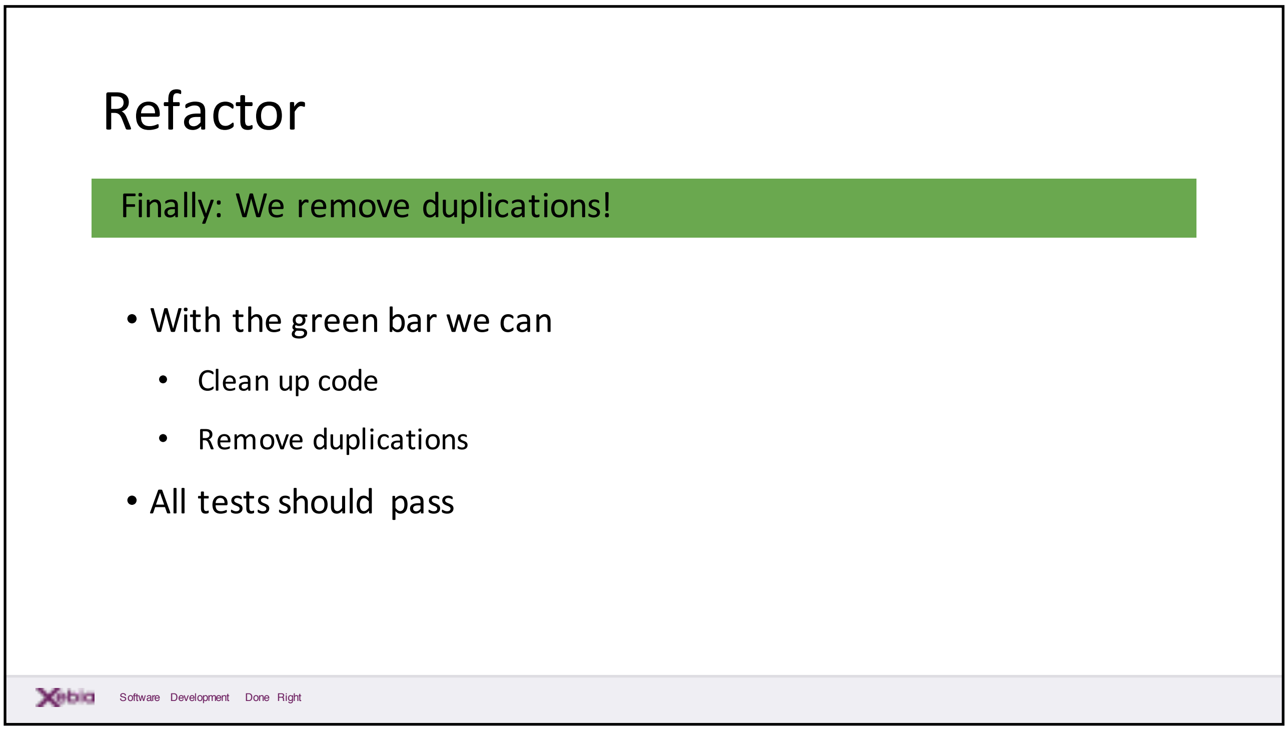
Triangulation

```
@Test
public void test() {
  assertEquals(10, sum(4, 6));
  assertEquals(8, sum(4, 4));
}

private int sum (int firstNumber, int secondNumber) {
  return 10;
}
```

Cheating!! (test pass)

failing test

Software Development Done Right

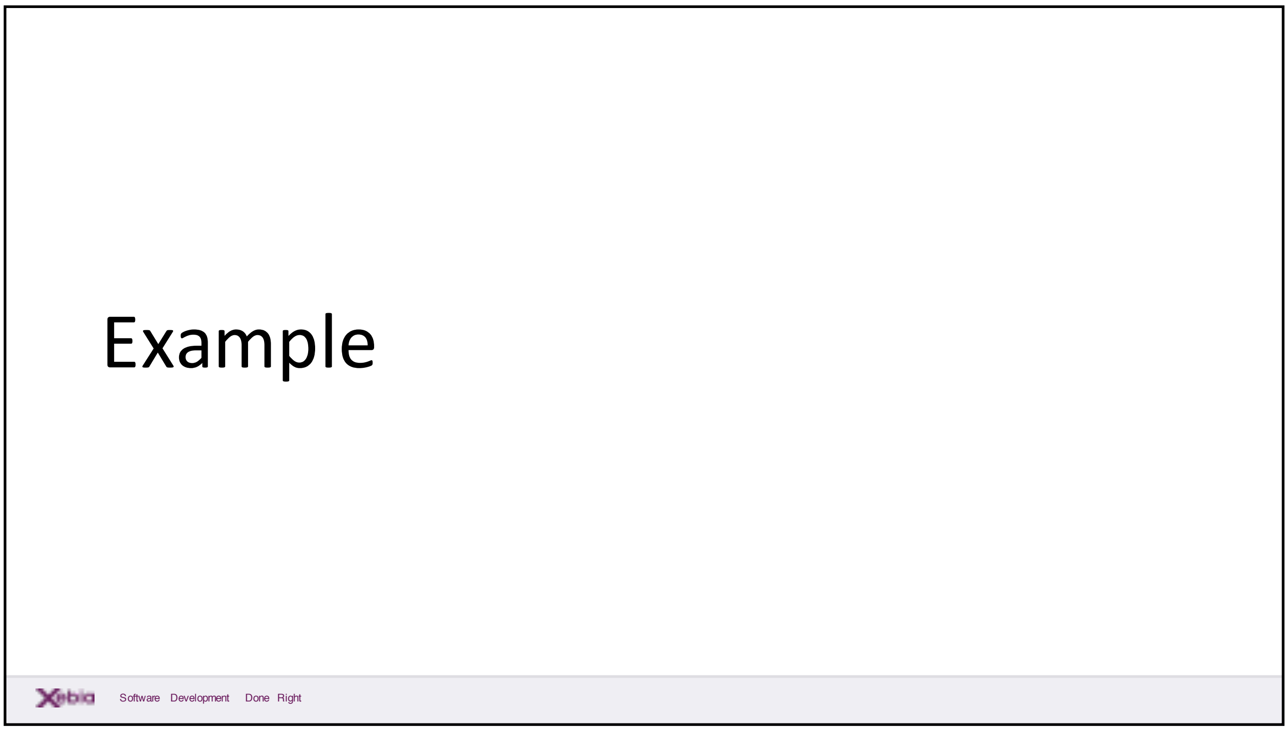


Refactor

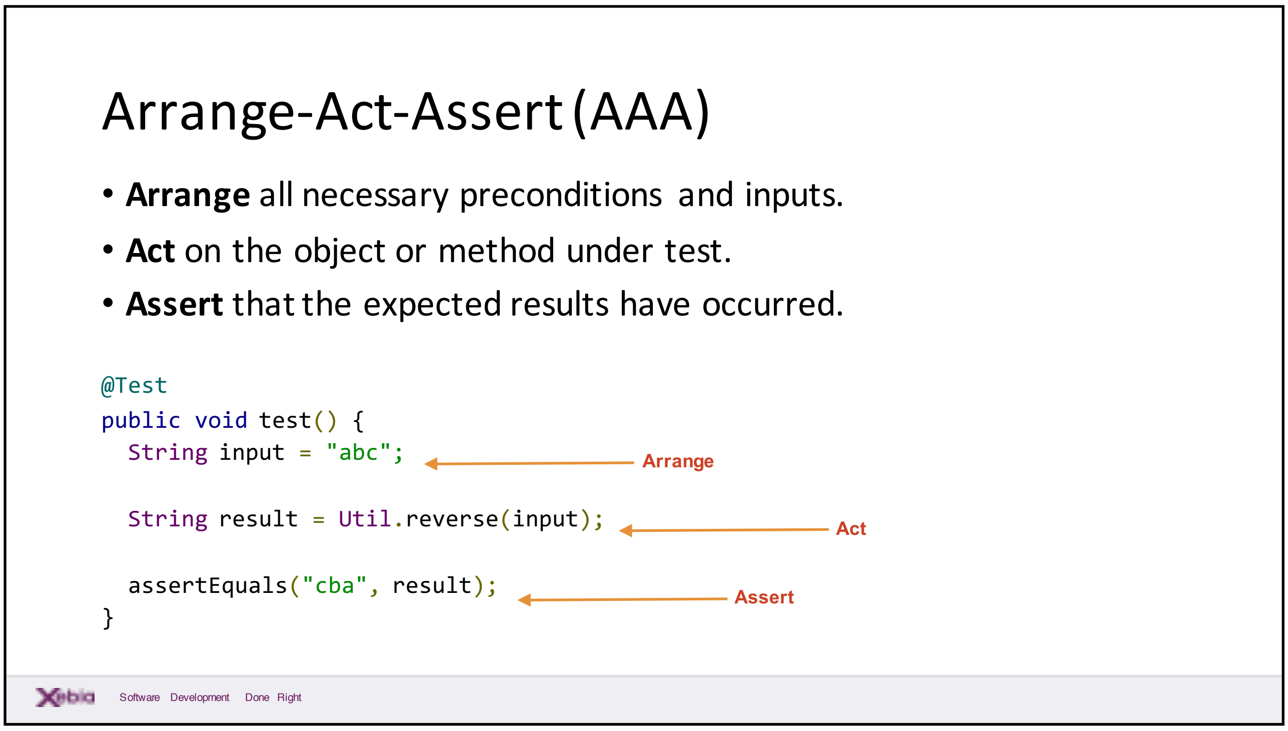
Finally: We remove duplications!

- With the green bar we can
  - Clean up code
  - Remove duplications
- All tests should pass

Software Development Done Right



Example



Arrange-Act-Assert (AAA)

- **Arrange** all necessary preconditions and inputs.
- **Act** on the object or method under test.
- **Assert** that the expected results have occurred.

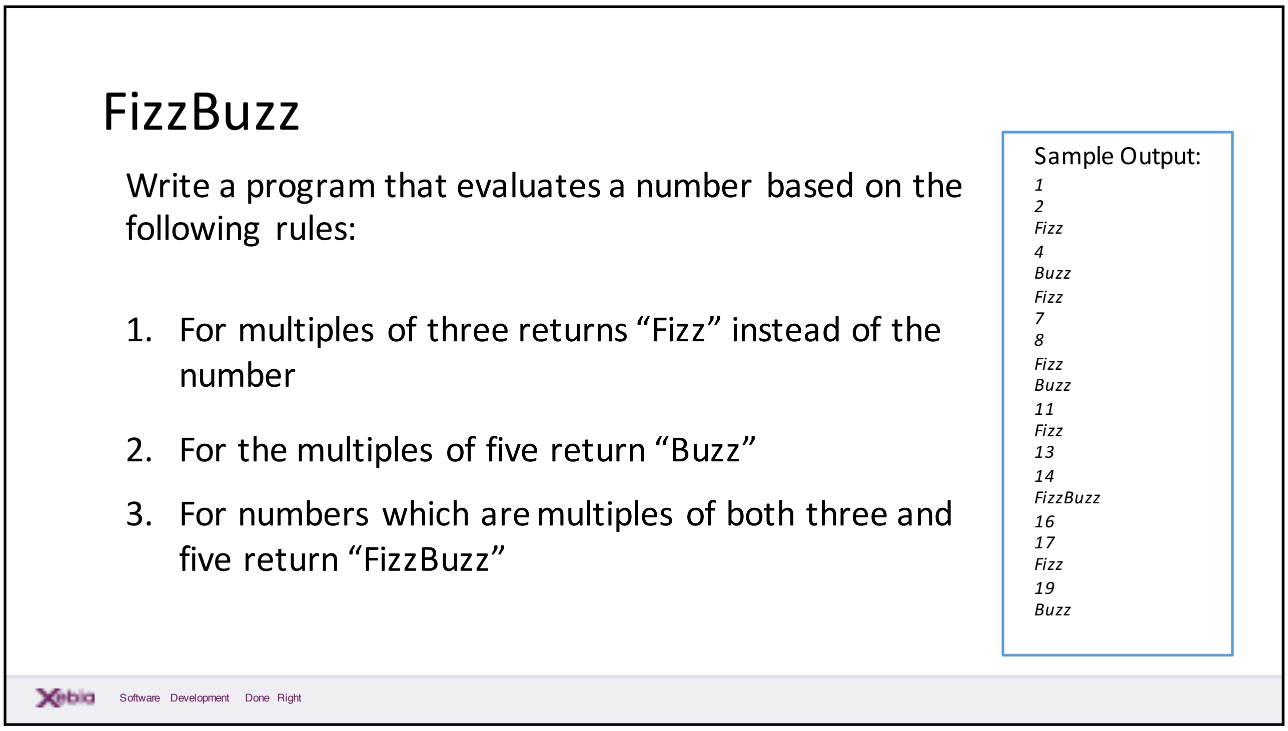
```
@Test
public void test() {
  String input = "abc";                ←——————— Arrange

  String result = Util.reverse(input);   ←——————— Act

  assertEquals("cba", result);     ←——————— Assert
}
```

Discipline - once again

- One thing at a time
- Solve bigger problems by solving fewer problems at once
- Subdivide, so that integrating the separate solutions will be a smaller problem than just solving them together

Principles

- DRY (Don’t repeat yourself)
  - Reduce repetition of information of all kinds (even in tests)
- YAGNI (You aren't gonna need it)
  - Some capability we presume our software needs in the future should not be built now because "you aren't gonna need it”
- KISS (Keep it simple, stupid)
  - The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided



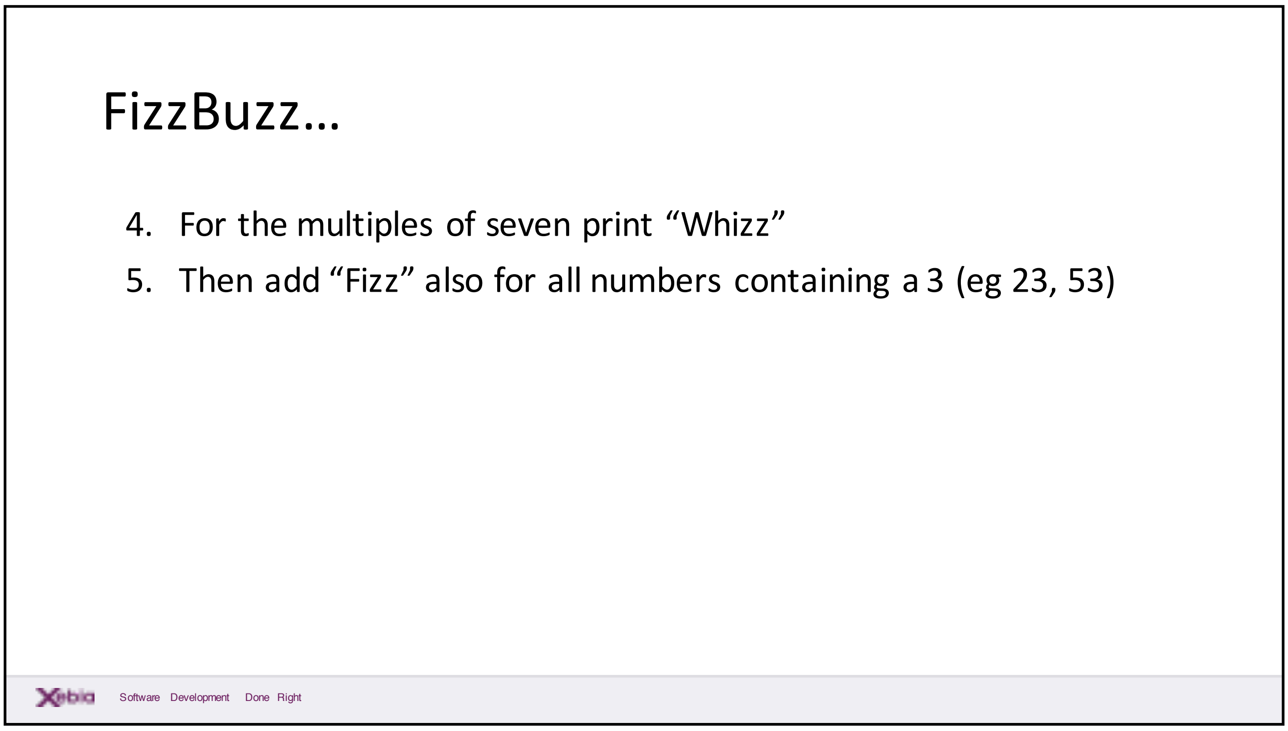
FizzBuzz

Write a program that evaluates a number based on the following rules:

1. For multiples of three returns “Fizz” instead of the number
2. For the multiples of five return “Buzz”
3. For numbers which are multiples of both three and five return “FizzBuzz”

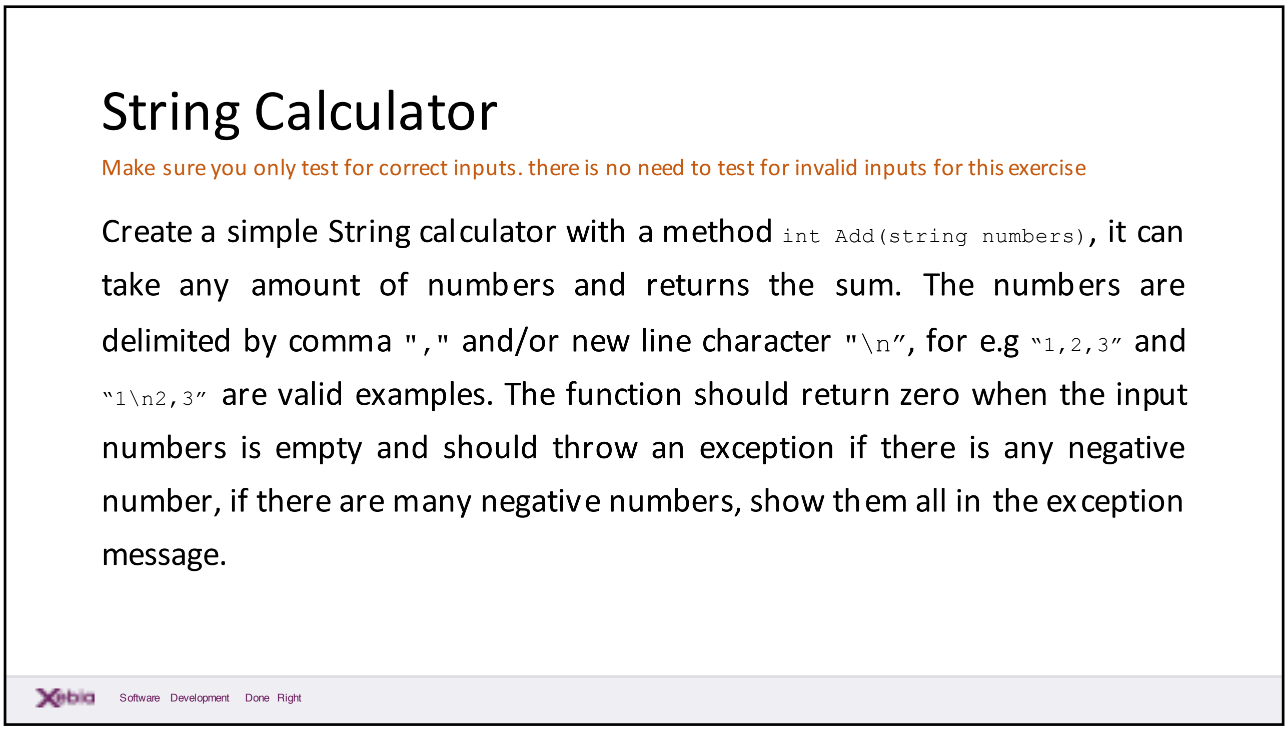
Sample Output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```



FizzBuzz…

4. For the multiples of seven print “Whizz”
5. Then add “Fizz” also for all numbers containing a 3 (eg 23, 53)

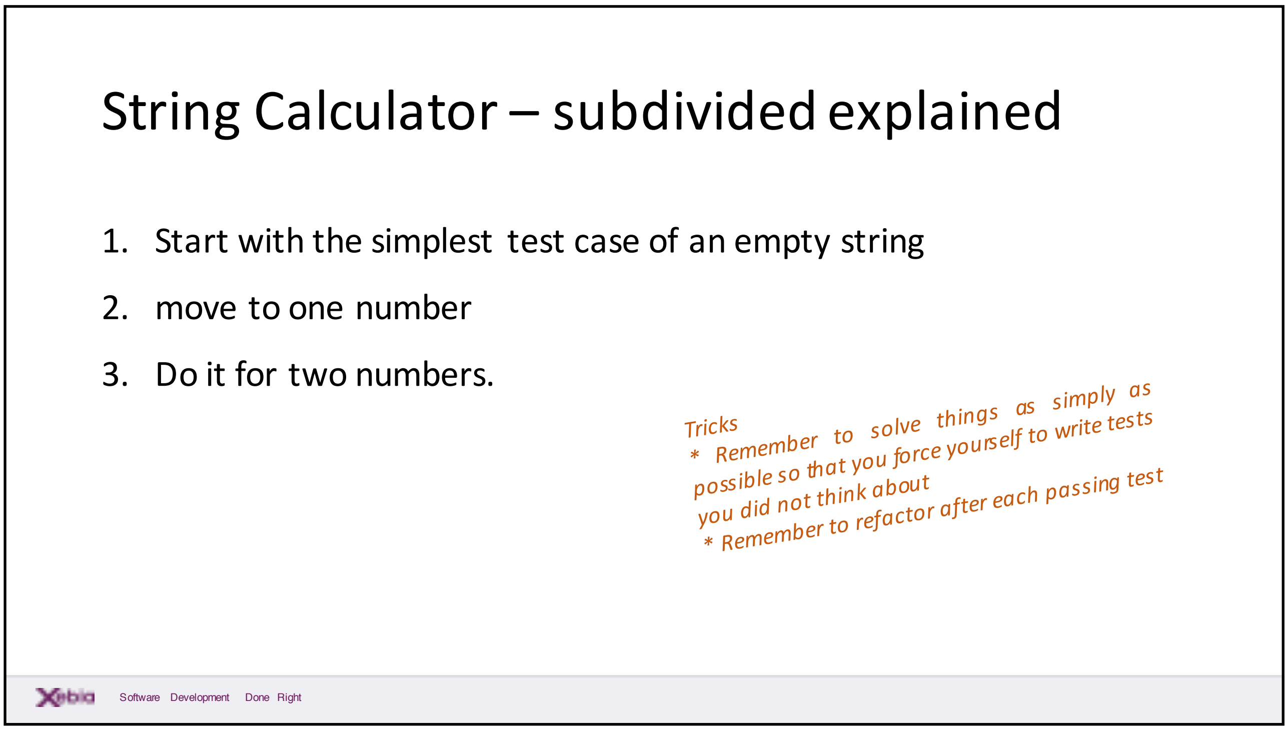


String Calculator

Make sure you only test for correct inputs. there is no need to test for invalid inputs for this exercise

Create a simple String calculator with a method `int Add(string numbers)`, it can take any amount of numbers and returns the sum. The numbers are delimited by comma `","` and/or new line character `"\n"`, for e.g `"1,2,3"` and `"1\n2,3"` are valid examples. The function should return zero when the input numbers is empty and should throw an exception if there is any negative number, if there are many negative numbers, show them all in the exception message.

Software Development Done Right



String Calculator – subdivided explained

1. Start with the simplest test case of an empty string
2. move to one number
3. Do it for two numbers.

Tricks
** Remember to solve things as simply as possible so that you force yourself to write tests you did not think about*
** Remember to refactor after each passing test*

Software Development Done Right

String Calculator – subdivided explained

2. Allow the Add method to handle an unknown amount of numbers
3. Allow the Add method to handle new lines between numbers (instead of commas). the following input is ok: `1\n2` (will equal 3)
4. Calling Add with a negative number will throw an exception `negatives not allowed` - and the negative that was passed. if there are multiple negatives, show all of them in the exception message

Bonus Step

- Numbers bigger than 1000 should be ignored, so adding 2 + 1001 = 2