FizzBuzz

Difficulty: Novice

Estimated Duration: 20 minutes

Introduction

This kata, taken from a popular children's maths game (or student drinking game), is the starting point on the TDD track. It's designed to be a semi-guided first stop for learning TDD from scratch.

We'll emphasise the following:

- Start by writing a failing test for the simplest behaviour.
- Implement the simplest amount of code needed to make the test pass.
- As you add more tests, refactor to make the code more generic and more suitable.

Instructions

Write a function that takes positive integers and outputs their string representation.

Your function should comply with the following additional rules:

- If the number is a multiple of three, return the string "Fizz".
- If the number is a multiple of five, return the string "Buzz".
- If the number is a multiple of both three and five, return the string "FizzBuzz".

For example, given the numbers from 1 to 15 in order, the function would return:

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

Starting Off

As this is (most likely) your first ever attempt at classical TDD, this page will start you off. We will run through the first few cycles of TDD, showing the various steps, and try to explain the underlying rationale for the decisions we make.

Initially, we have no tests, no code, but the 'system' can be considered Green because we have no failing tests. The very first action is to convert to a state of Red (at least one test is failing)

FIRST CYCLE

Red: Write the first failing test

To write the first test, we'll try to find the simplest unit of functionality that we can. In this case, recall that the program takes positive integers as input. The lowest positive integer is 1, so we will start with a test for that.

```
class FizzBuzzShould {
    @Test
    void convert_1_to_1() {
        assertEquals("1", new FizzBuzz().convert(1));
    }
}
```

It is worth briefly considering general advice around naming tests. Consider how you name your tests, as maintaining your tests and making changes to them will become a 'cost' to your development. Please read the section Why do you name the tests the way you do?

This will fail to compile because the `FizzBuzz` class doesn't exist. Now that we have a failing test (compilation failure is still a failure!), we can write just enough production code to make the test compile:

```
public class FizzBuzz {
    public String convert(int number) {
        throw new UnsupportedOperationException("implement me!");
    }
}
```

The test now fails instead with:

```
java.lang.UnsupportedOperationException: implement me!

        at com.codurance.fizzbuzz.FizzBuzz.convert(FizzBuzz.java:5)
        at FizzBuzzShould.convert_1_to_1(FizzBuzzShould.java:9)
        ....
```

Green: Write code to make the test pass

Here we will apply the first principle of TDD:

You are not allowed to write any production code unless it is for making a failing unit test pass

What is the simplest code that will make the test go green? It's just to return a constant:

```
    public String convert(int number) {
        return "1";
    }
```

This is known as "Faking it". Don't be concerned too much by this.

Some of the guiding principles for TDD are *"KISS" (or "keep it simple stupid")*, and *"YAGNI" ("you ain't going to need it")*. Both of these principles are subtly different but they both agree that in the context of TDD, the focus should be to just write enough code to pass the test, using the simplest solution to do so. This helps keep the design as clean and clear as possible.

One of the seemingly magical qualities of TDD is that even though some of the code seems counterintuitive to start with, by concentrating on the simplest solution possible an elegant solution will *emerge* as you add tests. Stick with the simplest solution possible and only introduce additional complexity as and when it is required.

Many developers feel the urge to predict where the changes to the code should go - resist this urge. The beauty of TDD is that as you go along you will start to introduce the necessary features for the desired solution, but you will be guided by the tests, and the tests will confirm those changes once they are in place.

Even Kent Beck, the pioneer of TDD, has suggested a principle of his own: *"fake it until you make it"*. This is what we've just done.

Refactor step

There's no duplication in the code that can be removed, so there's no refactoring to do yet.

Congratulations! You've completed one full cycle of classical TDD. Now we repeat the process.

SECOND CYCLE

Red step

Trying to decide what to write as the next test is where the *driving* of test-driven development comes in.

What particular behaviour in the specification are we driving towards?

In our opinion, it's in the first line:

Write a function that takes positive integers and outputs their string representation

Our code has made the first step towards implementing this feature, but it's not correct yet: `convert(2)` will return `"1"`.

Our next test can target this:

```
@Test
void convert_2_to_2() {
    assertEquals("2", new FizzBuzz().convert(2));
}
```

Green step

The simplest thing to make this pass is adding simple `if` statement:

```
public String convert(int number) {
    if (number == 2) return "2";
    return "1";
}
```

This is the next implementation strategy up from *"Faking It"* which is *"Obvious Implementation"* - basically if the implementation is obvious, code it in.

Be Patient. To an experienced developer, these steps seem trivial and unnecessary. And for developers who are very experienced in TDD, they sometimes skip these steps. But any discipline is about learning to do it properly, and practising in this way until it becomes internalised and second-nature. When you're learning to play the piano, you need to practice your scales and chopsticks. It's the same with TDD.

Refactor step

There is now some duplication in both the implementation and the tests. However, we don't want to try to refactor quite yet.

While normally code should conform to the *Don't Repeat Yourself (a.k.a. DRY) principle*, that doesn't mean you need to remorselessly refactor out duplication as soon as you see it. There is a more subtle guiding principle that is worth considering, which is about deferring decision making. Experienced designers know that "the best decisions are made with the maximum possible information". Therefore pragmatic designers try to defer design decisions for as long as they can.

This is not about ignoring duplication, but about tolerating it for a short time, as a trade-off to get a clearer picture of any emerging patterns in your code. Sometimes if you remove a pattern of duplication immediately it leads to less optimal designs. By waiting for the duplication to happen three times, you allow the possibility of a larger more subtle pattern to emerge that requires a different design decision.

So, until we see three cases of obvious redundancy, we will defer refactoring it out. This is known as the *Rule of Three*.

THIRD CYCLE

Red

We are still driving towards a function that outputs the string representation of positive integers: this is our first "feature", however small. We prefer not to add more "features" (e.g. printing Fizz instead of 3) until the current one is fully implemented.

Besides, in this case, the other rules are special cases over a base case. So it makes sense to finish off the base case before moving to the next bit of behaviour.

So we choose another regular number, avoiding what we know will become `Fizz`, for our next test:

```
@Test
void convert_4_to_4() {
    assertEquals("4", new FizzBuzz().convert(4));
}
```

Green

The simplest solution to make the test pass is still to add another if statement:

```
public String convert(int number) {
    if (number == 4) return "4";
    if (number == 2) return "2";
    return "1";
}
```

Refactor

We can easily spot a rule-of-three violation in the implementation now. This is a chance for us to make a leap from a specific solution to a more generic one:

```
public String convert(int number) {
    return String.valueOf(number);
}
```

It's also important to keep duplication as low as possible in tests, so we convert our three methods to a parameterized test case:

```
@ParameterizedTest
@CsvSource({ "1,1", "2,2", "4,4" })
void convert_number_to_FizzBuzz_string(int input, String expectedOutput) {
    assertEquals(expectedOutput, new FizzBuzz().convert(input));
}
```

FOURTH CYCLE

In the first to third cycle, we were dealing with the most generic requirement of the function: that it takes positive integers and outputs their string representation.

The refactoring to `String.valueOf()` we made in the last cycle has had a great effect: by induction we can see that implementation holds for all positive integers that aren't multiples of 3 or 5, not just the three test cases that we have.

This is fantastic: our tests have driven us to a simple generic algorithm!

However, if we look back at the specification, we'll see that we have three more unimplemented behaviours:

- Return `Fizz` for multiples of 3
- Return `Buzz` for multiples of 5
- Return `FizzBuzz` for multiples of 3 and 5

At this point, we have an algorithm that is incomplete in its implementation but covers the largest (generic) use case. We now try to use the third implementation strategy of *triangulation*. We do this by creating a specific test case, that forces the behaviour of your code to change.

This is a strategy performed as a small step, where the implementation for the specific tests can start to reveal the underlying patterns that will give clues to the eventual generic solution.

So in order to start implementing the `Fizz` behaviour, we can add a test for a specific case: the number 3. Adding this test and implementing the code to satisfy will start to reveal the change in the behaviour.

Red

We add a new test:

```
@Test
void convert_3_to_Fizz() {
    assertEquals("Fizz", new FizzBuzz().convert(3));
}
```

We expect this to fail - it is expecting `Fizz` but our code will give it `3`. Indeed, it does fail:

```
org.opentest4j.AssertionFailedError:
Expected :Fizz
Actual   :3
```

Green

The simplest way to implement this is with a simple `if` condition:

```
String convert(int i) {
    if (i == 3) return "Fizz";
    return String.valueOf(i);
}
```

Note that once again we have implemented only the code that is necessary to make the test pass. We might have a good idea of where the algorithm is *going*, but it's important not to jump ahead: let the tests lead the way.

Refactor

There's no obvious duplication here, so there's nothing to refactor.

You're next!

Try to implement the rest of the exercise on your own:

- Finish the implementation of 'Fizz' by expanding the current functionality to multiples of 3
- Implement the `Buzz` behaviour for numbers that are multiples of 5
- Finally, the rules for `FizzBuzz`

Frequently Asked Questions

Isn't this unnecessarily complicated?

It might seem it for now, because this kata is simple enough that a full solution will immediately suggest itself to most developers. However, consider this: you will often be designing algorithms that are too big to reason about in one go. When that happens, writing the simplest thing first and building it up in baby steps starts to look like a much more attractive proposal. In other words, TDD scales very well.

What about invalid inputs?

When doing katas, we like to focus on the Happy Path. There is certainly a time and a place for protection from bad input (e.g. in user interfaces!), but that's not what we're practising here. Besides, good code design would suggest separating the responsibility of checking input validity from the responsibility of executing the algorithm. So if this were a real system, the `FizzBuzz` class might be wrapped by a `FizzBuzzInputChecker` class which ensured that `FizzBuzz` was only ever fed valid numbers.

Why do you name your tests the way you do?

One of the pitfalls of TDD is that people belatedly recognise that the tests are as much a part of the code base as the actual code. Initially, people tend to make the mistake of treating tests as second-class code that is somehow inferior to the code that implements the software. Almost everyone comes to realise that the tests are equally as important as the implementation code.

One of the great advantages of tests is that when they break they can give the developers fast feedback of where the problem has occurred. The way a test is named can either help or hinder this process.

When naming a test, it is recommended to refrain from:

- Giving your test technical names, for example, "TotalSalesAggregatorReturnsInteger"
- Naming tests in a way that reveals implementation details, "ClientLookupReturnsTrue" or "SalesPriceDecoratorTest"

Coupling your test names with the names of the classes the test is based is also not advisable. This is because your code will be constantly changing - classes can be renamed or deprecated, or have some of their functions moved into another class. Every time you change the configuration of classes in your code, you leave yourself open to breaking the associated tests.

The best practice for naming tests is to use names that describe the business functionality or feature. This allows you to refactor your code such that, as long as the business behaviour remains the same, you should not have to change your test name.

Optional Extension

One version of the FizzBuzz game in real life adds an extra Fizz or Buzz whenever one of the digits ('3' or '5') appears in the number itself (for example, see Dr Mike's Math Games for Kids).

So, '3' would become `FizzFizz`, '5' would become `BuzzBuzz`, '15' would be `FizzBuzzBuzz`.

Modify your program to reflect this requirement while maintaining your discipline in using red-green-refactor cycles.

baby steps | red-green-refactor | classical TDD | TDD starter

Psst!

Katalyst is in beta and we'd love your feedback.

Email us at katalyst@codurance.com and let us know what you think.