Building Collaboration with Visible Tests

Ken Pugh, ken@kenpugh.com, https://kenpugh.com

Customer requirements that have outcomes visible to the user can use a testing approach that is readable to the Customer to decrease effort and increase collaboration. The FizzBuzz Kata is a frequently used exercise to develop TDD skills. We'll show an alternative approach to the kata that focuses the visible output and provides an easy way to add variations from a tester's perspective. This approach can be used for any business rules and calculations that are specified by the customer or subject matter experts.

Requirement

The requirement is:

- Write a program that prints one line for each number from 1 to 100
- For multiples of three print "Fizz" instead of the number
- For the multiples of five print "Buzz" instead of the number
- For numbers which are multiples of both three and five print "FizzBuzz" instead of the number

We're going to defer the first bullet point for now (and discuss that in another article "The Full Story") and focus on the last three.

Understanding the Requirement

The Triad consists of three perspectives – Customer (requirements provider), Developer (requirements implementer), and Tester (critical analyzer of requirements and implementations). Based on the requirements, they come up with a scenario. Using a common form, it might look something like:

```
Scenario Outline: Output based on what is the multiple of 3 and 5
Given input is <Input>
When Fizzbuzz is computed
Then output is <Output>
Examples:
| Input   | Output      | Notes                    |
| 1       | 1           | Not multiple of 3 or 5   |
| 3       | Fizz        | Multiple of 3            |
| 5       | Buzz        | Multiple of 5            |
| 15      | FizzBuzz    | Multiple of 3 and 5      |
```

An alternative way of expressing this scenario looks like this:

```
Scenario: Output based on what is the multiple of 3 and 5
* Given the input, FizzBuzz output should be:
| Input   | Output      | Notes                    |
| 1       | 1           | Not multiple of 3 or 5   |
| 3       | Fizz        | Multiple of 3            |
| 5       | Buzz        | Multiple of 5            |
| 15      | FizzBuzz    | Multiple of 3 and 5      |
```

TDD Cycle

Now the scenario is a check that the requirements are understood correctly. It represents Kent Beck's "to-do list" of tests that need to pass. The developer creates a step definition to connect this scenario to the actual code. Each row represents a test for a portion of the requirement. To use this in the red-green-refactor TDD cycle, you decide which test you want to execute first and comment out all the rest. Make the test red (fail), then make it green (pass), refactor, check that all pass, and then uncomment another row.

```
| Input   | Output       | Notes                   |

| 1       | 1            | Not multiple of 3 or 5 |
#| 3       | Fizz        | Multiple of 3           |
#| 5       | Buzz        | Multiple of 5           |
#| 15      | FizzBuzz    | Multiple of 3 and 5     |
```

Variations

Now additional variations can be added as desired or as they are discovered. For example, the original checks were only for the first multiple. Here are checks for higher multiples:

```
| Input   | Output     | Notes                   |
| 6       | Fizz       | Multiple of 3           |
| 10      | Buzz       | Multiple of 5           |
| 30      | FizzBuzz   | Multiple of 3 and 5     |
```

These variations might be added by any perspective. A tester perspective might want to test the highest limits This scenario checks the values just below and above each significant change and some upper limits. If a tester wanted to run these checks, they could do so without creating any additional step definitions, thus avoiding test maintenance issues. If a single perspective (e.g. tester) creates this scenario, it should be passed by the customer and developer to ensure that there is agreement.

Note that all tests are in a consistent format, making it easier for the entire Triad to collaborate.

```
| Input   | Output     | Notes                             |
| 90      | FizzBuzz   | Highest value producing FizzBuzz  |
| 99      | Fizz       | Highest value producing Fizz      |
| 100     | Buzz       | Highest value producing Buzz      |
```

The Triad can examine out-of-range values and collectively determine what should be the appropriate output. For example:

```
| Input   | Output     | Notes                                  |
| 0       | 0          | Below minimum, is this what is desired?  |
| 101     | 101        | Above maximum, is this what is desired?  |
```

You can rearrange the rows as desired to group the various checks together. There are many ways you can do this. The Triad selects one that is most readable. For example, one might have:

```
| Input   | Output    |
# Not multiple of 3 or 5
| 1       | 1         |
# Multiples of 3
| 3       | Fizz      |
| 6       | Fizz      |
# Multiples of 5
| 5       | Buzz      |
| 10      | Buzz      |
# Multiples of 3 and 5
| 15      | FizzBuzz  |
| 30      | FizzBuzz  |
```

Summary

Using Gherkin allows tests to be created independent of the implementation language. They focus on the way something should behave, not the code which makes it behave that way. This separation of concerns allows all perspectives of the Triad to collaborate.

Questions

What would you term these tests that these scenarios represent?   Are they behavior tests? Are they unit tests?   Are they behavior tests that involve a single code unit (e.g., a method)?   Who should run them?