A Behavior Perspective on Development

Ken Pugh (ken@kenpugh.com)

I like to try multiple different approaches to solving a problem to see how they compare. I might learn something from one approach that I could apply to another. As an example, I created an alternative approach to calculations with fractions. The fraction idea comes from J.B. Rainsberger's portion of his class on Test Driven Development at https://online-training.jbrains.ca/courses/8788/lectures/133485. It is a great demonstration of TDD.

In this example, I'm going to use Gherkin as the specification language for the customer facing behavior For applications where the external behavior and its component parts are specified by the customer and/or stakeholders, having examples in Gherkin can dramatically decrease the communication impedance between the customer and the developer.

External Behavior

I'm going to alter the example context slightly. Let's start with a story that has a solution requiring fraction calculation. The Triad (Customer, Developer, Tester) have created this scenario:

```
Scenario: Fraction Calculator
Given fraction calculator is active
When user enters "3/4 + 5/8 + 3"
Then the total is shown as "35/8"
```

We'll need fraction calculations in order to implement this story. Let's write this calculation in a slightly different form. This represents the external behavior that the user sees:

```
Scenario:  Fraction Calculation External Behavior
* This is a sample of the expressions to be calculated
| Expression     | Result  |
| 3/4 + 5/8 + 3  | 35/8    |
```

Fraction calculations might include subtraction, multiplication, and other operations. To keep this article short, I'll leave those extensions as an exercise to the reader.

Understanding the Detailed Behavior

Let's breakdown the calculation into its component parts. The Triad (augmented if necessary by a Subject Matter Expert in fraction calculation) comes up with these parts:

```
Scenario:  Expressions to Calculate
* This is a sample of the expressions to be calculated
| Expression     | Result  |
| 3/4 + 5/8      | 11/8    |
| 11/8 + 3       | 35/8    |
```

For this simple calculation, the order does not matter. However, in other instances it might. Having examples allows discussion on what the results of a calculation or algorithm should be.

Let's get into more details of the calculation. In this example, there is an obvious domain term – Fraction – that contains a Numerator and a Denominator. Domain terms typically have an internal representation for computation and an external textual representation so that they can be input and

output. Let's start with specifying the textual representation and the internal representation for the values in the external behavior test.

```
Scenario: Business Rule Display to Fraction
* Convert Display to Fraction
| Display  | Numerator  | Denominator  | Notes                    |
| 3/4      | 3          | 4            | Part of expression test  |
| 5/8      | 5          | 8            | Part of expression test  |
| 3        | 3          | 1            | Whole number             |
```

Here is the output representation for a fraction.

```
Scenario: Business Rule Fraction to Display
* Convert Fraction to Display
| Numerator  | Denominator  | Display  | Notes                    |
| 35         | 8            | 35/8     | Part of expression test  |
```

The expression in this scenario is a textual representation of addition.

```
Scenario:  Expressions to Calculate
| Expression     | Result  |
| 3/4 + 5/8      | 11/8    |
```

Let's break this textual expression into its individual components:

```
Scenario:  Display to Components
| Expression  | In1  | Op  | In2  |
| 3/4 + 5/8   | 3/4  | +   | 5/8  |
```

Now we could show the operations using those individual components.

```
Scenario: Business Rule Operations
These are expressed as component values
* Input Fractions (In1 and In2), Operator (Op), Result
| In1    | Op  | In2    | Result  | Notes                          |
| 3/4    | +   | 5/8    | 11/8    | Part of Expression Test        |
| 11/8   | +   | 3      | 35/8    | Part of Expression Test        |
```

Now we're walking a fine line between external behavior and internal behavior. These components represent concepts of shared understanding of the Triad of the problem space and how the solution would be reached even without a computer. For more complex algorithms, understanding how something is calculated prior to implementing it can dramatically reduce rework.

The example contained just addition. If there were more operators, then they could be listed separately:

```
Scenario: Domain Term Operators
* Supported Operators
| Symbol | Meaning         | Notes                |
| +      | Addition        |                      |
| *      | Multiplication  | To do next iteration |
```

Design and Implementation

You may note that the addition behavior does not specify how the implementation will be designed. We pretty much know there will be a Fraction class, since it's a domain term. There are at least three ways to add operations to Fractions. (I usually follow Jerry Weinberg's Rule of Three – if you can't come up with three solutions to a problem, you don't understand the problem). The operations could be:

- member functions (e.g. aFraction.add(anotherFraction))
- static member functions (e.g. Fraction.add (aFraction, anotherFraction))
- static functions in another class (Calculation.add(aFraction, anotherFraction).

The choice is up to the developers based on the code quality of the solution. The developers can decide to implement it one way and then change the implementation without affecting the scenarios that have been written. That is an advantage of writing behavior tests in Gherkin. The test is decoupled from the implementation.

Now you can start with one test at some level, watch it fail, and then implement just enough to make it pass. Again, there are at least three ways to start – with the external behavior, the lowest level behavior, or somewhere in-between. There are "schools" of practice that suggest starting high or starting low. That's a discussion you can research by searching for "London versus Chicago TDD" or "statist versus mockist TDD".

Here's one way to approach it. Start with the domain terms, i.e., these scenarios:

```
Scenario: Business Rule Display to Fraction
Scenario: Business Rule Fraction to Display
```

Once again, one could consider at least three ways to do this, such as:

- Constructor that takes a string (e.g. Fraction(aString))
- Static method (e.g. Fraction.toFraction(aString))
- Input method (e.g. Input.toFraction(aString))

Next do the lowest level:

```
Scenario: Business Rule Operations
```

And then work your way up. At each level, you get to decide whether to use a fake implementation or an actual one. For example, should the add operator just be coded to return two addition results or should it be coded to compute them? Are you working on the communication between methods or on the functionality of each method?

```
Scenario: Business Rule Operations
| In1    | Op  | In2    | Result  | Notes                          |
| 3/4    | +   | 5/8    | 11/8    | Part of Expression Test        |
| 11/8   | +   | 3      | 35/8    | Part of Expression Test        |
```

It's Not Done

There was only one example of external behavior in the original scenario:

```
| Expression     | Result  |
| 3/4 + 5/8 + 3  | 35/8    |
```

You shouldn't specify all possible combinations of values in external behavior. The number could be staggering. As a behavior is split into components, the variations of values can be applied to those

smaller behaviors. Let's take a look at the previous scenarios with added variations. Here is Fraction input:

```
Scenario: Business Rule Display to Fraction

* Convert Display to Fraction
| Display  | Numerator  | Denominator  | Notes                  | Question                 |
| -3       | -3         | 1            | Negative number        |                          |
| 1/3      | 1          | 3            | Fraction               |                          |
| 2/4      | 1          | 2            | Fraction reduced       | Should it be reduced?    |
| 1 1/3    | 4          | 3            | Whole and fraction     |                          |
```

The question is whether an input text value should be reduced to the smallest denominator (e.g. 2/4 to 1 / 2). Will there be a case where the expression consists of a single fraction with no operation and there might be confusion if the result is displayed differently?

Here is Fraction output:

```
Scenario: Business Rule Fraction to Display
* Convert Fraction to Display
| Numerator  | Denominator  | Display  | Notes                  | Questions              |
| 3          | 1            | 3        | Whole Number           |                        |
| -3         | 1            | -3       | Negative Numerator     |                        |
| 1          | 3            | 1/3      | Fraction               |                        |
| 3          | 3            | 1        | Whole Number           | Should it be fraction? |
| 3          | -1           | 3/-1     | Negative Denominator   | Or should it be -3/1   |
| 3          | 2            | 3/2      | Whole plus fraction    | or should it be 1 1/2  |
```

The questions are how whole numbers or negative numbers should be displayed. The questions in these scenarios are ones that those who specify the external behavior (e.g. the customer) should answer.

Here are variations for the addition operator. This table looks similar to the examples written on J.B.'s "to do" list in his TDD video.

```
Scenario: Business Rule Operations
* Inputs (In1 and In2), Operator (Op), Result
| In1    | Op  | In2    | Result  | Notes                           |
| 3      | +   | 2      | 5       | Whole Number                    |
| 1/3    | +   | 1/3    | 2/3     | Two fractions, same denominator |
| 1/3    | +   | 2/3    | 1       | Whole number result             |
| 1/3    | +   | 1/2    | 5/6     | Different denominators          |
| 1/6    | +   | 1/8    | 7/24    | Fraction and reduce             |
| 1/4    | +   | 1/4    | 1/2     | Fraction and reduce             |
| 1 1/3  | +   | 1 2/3  | 3       | Two whole and fractions         |
| 3      | +   | 1/3    | 10/3    | Whole number and fraction       |
| 1 1/4  | +   | 2 1/8  | 3 3/8   | Different whole numbers         |
```

You can apply these tests one at a time by commenting out all the tests and uncommenting the one you are working on.

If the scenarios are automated, then additional examples/tests can be easily created. Anyone on the Triad can add tests for minimum values, maximum values, non-numeric values, and so forth. What about a denominator whose value is 0?

How Far?

There are a couple of concepts that are used to reduce fractions – Great Common Multiple and Lowest Common Denominator. Here are some scenarios that specify some examples of how they work. You might not specify these with Gherkin. However, if these concepts were important to a stakeholder (user, customer, subject matter expert), you might want to have these readable scenarios. They document the shared understanding.

```
Scenario: Business Rule Greatest Common Multiple
GCM used to reduce fractions to LCD (lowest common denominator)
* GCM is the great common multiple of D1 and D2
| D1  | D2  | GCM  |
| 4   | 8   | 4    |
| 12  | 8   | 4    |
| 12  | 9   | 3    |
| 37  | 74  | 37   |

Scenario: Business Rule Reduce by GCM
* Converting fraction (InNum/InDenom) to LCD (ResNum/ResDenom)
| InNum  | InDenom  | ResNum  | ResDenom  | Notes               |
| 4      | 8        | 1       | 2         |                     |
| 8      | 4        | 2       | 1         | Previous reversed   |
| 12     | 8        | 3       | 2         | Whole and fraction  |
| 37     | 74       | 1       | 2         | Large GSM           |
| 12     | 9        | 4       | 3         | Just another        |
```

Summary

This article has approached a problem by using readable descriptions of the behavior needed to solve that problem. The approach started at the external behavior and then split it into sub-behaviors required to produce the external behavior. The tests for the sub-behaviors are independent of the language and the design. This can be beneficial if other stakeholders need to be in the loop on that behavior.