A Dollar Kata

Ken Pugh (atdd@kenpugh.com)

This kata revolves around a common domain term – money. Money appears in many applications, so the code from this kata might be adapted to those applications. Although it uses a dollar, it's easily changeable to the currency of your choice by just replacing the currency symbol. Since requirements /tests written in Gherkin are implementation independent, you can implement this in any language. Some languages may be easier than other since they have more extensive libraries. Programmers can try multiple implementations and then have discussions as to the relative merits of each one.

The Context

Sam wants to have a system which keeps track of a total amount of money. You enter an amount and it updates the total and a count of how many times the total has been updated. Here's the first scenario from the user's point of view.

```
Scenario: Adding money adds to total and increments count
Given current state is:
| Total   | Count  |
| $10.00  | 1      |
When user enters
| Amount  |
| $5.00   |
Then state is now
| Total   | Count  |
| $15.00  | 2      |
```

The Details

Now Sam is really concerned about the way the user enters information into the system. He wants it to check for incorrectly entered values. So he got the Triad together (he as customer and the two people having the perspectives of developer and tester) to come up with how the dollar entry should behave. They came to a common understanding of the various ways an amount could be entered and which entries would be invalid. Entries can have the currency symbol (if it's in the right place), commas (if in the right places) and a decimal part (if two digits). Negative values can use either the minus sign or parentheses, but not both. These have been broken down into parts which you might develop iteratively (or skip, you want a shorter kata).

```
Scenario: Domain Term Dollar
* Dollar Validation
| Input          | Valid  | Notes                                                |
| 1              | Yes    |                                                      |
| $1             | Yes    |                                                      |
| $-1            | Yes    | Negative                                             |
| -$1            | Yes    | Negative                                             |
| A              | No     | Must be digit, $, or -                               |
# Decimal
| 1.00           | Yes    |                                                      |
| 1.1            | No     | Either zero or two digits                            |
| $12345A67.80   | No     | Contains non-digit                                   |
# Parentheses
| (1)            | Yes    | Negative                                             |
| ($1)           | Yes    | Negative                                             |
| ($1            | No     | Parentheses must match                               |
| -($1)          | No     | Double negative                                      |
| (-$1)          | No     | Double negative                                      |
| 123$456().80   | No     | Characters in incorrect order                        |
# Commas
| 1000.01        | Yes    | Thousands                                            |
| 1,000.01       | Yes    | Comma okay                                           |
| 10,00.00       | No     | Commas must be in proper places (every three digits) |
```

```
| 1,234,567.89  | Yes    | Commas in proper places                          |
| 1234567.89    | Yes    | Without commas                                   |
| 1,234567.89   | No     | Must have all commas or no commas                |
```

The scenario lists the cases that the Triad has agreed upon. These are representative of valid and invalid entries. If you have a testing perspective, you may find more cases which you can add.

A companion to the first scenario shows the desired behavior when the entry is invalid:

```
Scenario: Entering invalid amount does not change total and shows a generic message
Given current state is:
| Total   | Count  |
| $10.00  | 1      |
When customer enters
| Amount |
| -($1)  |
Then state is now
| Total   | Count  |
| $10.00  | 1      |
And result shows
| Message         |
| Invalid format  |
```

Here's a scenario that shows the entry amount can be negative:

```
Scenario: Adding a negative amount of money changes total and count
Given current state is:
| Total   | Count  |
| $10.00  | 1      |
When user enters
| Amount    |
| -$15.00   |
Then state is now
| Total   | Count  |
| -$5.00  | 2      |
```

More Details

Sam really would like more information on the type of error to be shown to the user, so they can easily identify what is the issue. (What would you do if the compiler just returned "Compile Error" for any error?) For each invalid input, a specific message should be in the result.

```
Scenario: Business Rule Dollar Input Errors
* Messages to show for format errors
| Input          | Message                          |
| 1.1            | Must be either zero or two digits  |
| A              | Invalid character                |
| $12345A67.80   | Invalid character                |
| ($1            | Parentheses must match           |
| -($1)          | Double negative                  |
| (-$1)          | Double negative                  |
| 123$456().80   | Characters in incorrect order    |
| 10,00.00       | Commas must be every three digits  |
| 1,234567.89    | Commas must be every three digits  |
```

The scenario in the previous section would be replaced by:

```
Scenario: Entering invalid amount does not change total and generates message
Given current state is:
| Total   | Count  |
| $10.00  | 1      |
When customer enters
```

```
| Amount |
| -($1)  |
Then state is now
| Total   | Count  |
| $10.00  | 1      |
And result shows
| Message          |
| Double negative |
```

Approach

This application is about functionality, not appearance. You don't need to create a graphical user interface. You can use the scenarios as an interface. But if you want to try out "the real application" the user interface could be as simple as a command program which asks for an entry, prints the new state and the result, and then loops back for another entry. If you want a little more challenge, the interface could be a command line which takes the entry as an argument, prints the new state and the result, and then exits.

You get to decide in what sequence you'd like to implement this problem. Sam is not going to release it until the specific error messages are shown. But you might want to create something earlier to get quicker feedback from your users on the interface. The users can be anyone.

You can use Cucumber/SpecFlow to run the scenarios in your language. You can comment out all but one line of data in the table to act as a single test. As you go to green, you can uncomment out another line. You get to choose in which sequence you uncomment. It doesn't have to be in the order listed. Alternatively, you can convert the scenarios into the language specific testing framework (JUnit, Jasmine, etc.). The scenarios for validity and the error messages were separated to make smaller tables. You might find it easier to combine them.

Extensions

You could code the application using a graphical user interface and use a graphical testing tool to check the implementation. You could code it as a microservice and test it with your favorite testing tool. The same validation code could be used to validate data coming from a file, a database or a message.

Length

The reason for this long kata is that it is representative of some of the more difficult problems you may face. The validation is more complicated than just a simple parse. There are multiple ways you can implement it. You may come out with a useful value object that might be incorporated into your applications, especially if you discuss with your Triad what formats, validations, and messages there should be.

This can be a short kata by using only some of the variations of the input format (e.g. only minus signs for numbers, no commas, no decimal part). It can be shortened to display just the generic error message. So, you could start with these two variations and then call it done or expand it. The first expansion might be displaying specific error messages. The next would be adding more input formats and validations.

Another Extension

Once you are done with Sam's original problem, there are a couple of extensions that Sam would like. First, it would be helpful if the position at which the error appeared is shown along with the message. (A compiler tells you what line number the error is on).

```
Scenario: Business Rule Dollar Input Errors With Position
* Messages to show for format errors
| Input          | Message                          | Position |
| A              | Invalid character                | 1        |
| $12345A67.80   | Invalid character                | 7        |

Scenario: Entering invalid amount shows position of error
When customer enters
| Amount       |
| $12345A67.80 |
And result shows
| Message              | Position |
```

```
| Invalid character    | 7         |
```

The second is that he wants to use the system to total up the averages of a set of inputs, rather than just adding a single amount each time. He'd like the user to be able to enter from one to ten numbers, have them averaged, and then added to the total.

```
Scenario: Adding money adds to total and increments count
Given current state is:
| Total    | Count   |
| $10.00   | 1       |
When user enters
| Amount   |
| $5.00    |
| $5.01    |
| $5.02    |
Then average is
| Average |
| $5.01   |
Then state is now
| Total    | Count   |
| $15.01   | 2       |
```

Have a friend play the part of Sam and have a discussion on all the ramifications of how to compute the average. Then write a scenario for the decision and implement it.