The Gilded Rose Kata from a Gherkin Perspective

Ken Pugh

In reviewing a book, I was reminded about the Gilded Rose kata. It revolves around some legacy code that has no tests. You need to make some changes to the code to support a new requirement. The kata was created by Bobby Johnson (http://iamnotmyself.com/) and updated by Emily Bache (http://coding-is-like-cooking.info/)

One can look at this exercise from multiple perspectives. A common approach is to add some automated tests so that the code be refactored prior to making changes. The tests could be unit tests or characterization tests that produce a reference set of input/output values.[i] The unit tests are typically written from the developer's point of view. An alternative is to write Gherkin tests (Cucumber / Specflow) from the customer's point of view. This has the advantage of creating tests that have a closer match to the requirements.

The requirements are listed here: https://github.com/emilybache/GildedRose-Refactoring-Kata/blob/master/GildedRoseRequirements.txt. Here are the relevant domain terms:

> *- All items have a SellIn value which denotes the number of days we have to sell the item*
>
> *- All items have a Quality value which denotes how valuable the item is*
>
> *- At the end of each day our system lowers both values for every item*
>
> There are alternatives for determining how quality changes for specific items.

Since the original code was in C#, it was an easy matter to use SpecFlow as the testing framework. The notes column was abbreviated, so that it would fit into the page width. The notes could correspond to the names given to unit tests. The code and feature files are at https://github.com/atdd-bdd/GildedRose

The values for "Then" were determined from the requirements and then run against the source code. Creating the tests pointed out an unclear requirement (at least one that was unclear to me). Running the tests showed a failure that pointed it out.

A couple of lines in the examples check breakpoints in the requirements where a change in a value causes a different behavior (e.g. Sellin > 10 and Sellin <= 0). Someone with a testing perspective may add these lines or mutation testing may suggest them. Adding a new test involves just adding a new line.

```
Scenario Outline: Quality changes each day
Given item "<name>" with current Quality <cq> and current SellIn <cs>
When a day passes
Then item has revised Quality <rq> and revised Sellin <rs>
Examples:
| name                                    | cq  | cs  | rq  | rs  | notes              |
| Non-specific-item                       | 1   | 1   | 0   | 0   | quality decrease   |
| Non-specific-item                       | 0   | 1   | 0   | 0   | never below 0      |
| Non-specific-item                       | 4   | 0   | 2   | -1  | twice as fast      |
#
| Aged Brie                               | 4   | 1   | 5   | 0   | increases          |
| Aged Brie                               | 4   | 0   | 6   | -1  | increases twice    |
| Aged Brie                               | 4   | -1  | 6   | -2  | increases twice    |
| Aged Brie                               | 49  | -1  | 50  | -2  | 50 limit           |
| Aged Brie                               | 50  | -1  | 50  | -2  | 50 limit           |
#
| Sulfuras, Hand of Ragnaros              | 80  | 1   | 80  | 1   | never changes      |
#
| Backstage passes to a TAFKAL80ETC concert | 1   | 11  | 2   | 10  | increase by 1      |
| Backstage passes to a TAFKAL80ETC concert | 1   | 10  | 3   | 9   | increase by 2      |
| Backstage passes to a TAFKAL80ETC concert | 1   | 6   | 3   | 5   | increase by 2      |
| Backstage passes to a TAFKAL80ETC concert | 1   | 5   | 4   | 4   | increase by 3      |
```

```
| Backstage passes to a TAFKAL80ETC concert  | 1   | 0   | 0   | -1  | after concert     |
| Backstage passes to a TAFKAL80ETC concert  | 50  | 11  | 50  | 10  | 50 limit          |
```

This table could be split between multiple scenarios or separate example tables, one for each item. Alternatively, one could use a step definition table, such as follows. The glue code would require only one step definition and an interface class.

```
Scenario: Quality changes each day for items
# current quality cq and current sellIn cs
# revised quality rq and revised sellIn rs
* Quality changes after a day passes for specific items
| name                                    | cq  | cs  | rq  | rs  | notes             |
| Non-specific-item                       | 1   | 1   | 0   | 0   | quality decrease  |
| Non-specific-item                       | 0   | 1   | 0   | 0   | never below 0     |
…. As above
```

Separation

Writing the tests in this form helped me to better understand the requirements. It made it easier to write tests for the new requirements. It also showed some duplication. The tests for the change in SellIn could be separated from the tests for Quality. SellIn changes in the same way for each passing day except for the "Sulfuras" item. The above scenarios could be split into two, one for Quality and one for SellIn:

```
Scenario: Quality changes each day for items
# Current quality cq and revised quality rq
# Current sellIn cs
* Quality changes after a day passes for items based on SellIn
| name                                  | cq | cs | rq | notes            |
| Non-specific-item                     | 1  | 1  | 0  | quality decrease |
| Non-specific-item                     | 0  | 1  | 0  | never below 0    |
…  and so forth

Scenario: SellIn changes each day except for Sulfuras
# Current sellIn cs and revised sellIn rs
* SellIn changes after a day passes
| name                          | cs   | rs  | notes                       |
| Non-specific-item             | 1    | 0   | SellIn down by 1            |
| Non-specific-item             | -1   | -2  | Regardless of value         |
| Non-specific-item             | 100  | 99  | Is there a maximum SellIn?  |
| Sulfuras, Hand of Ragnaros    | 1    | 1   | SellIn never changes        |
```

Refactoring Gherkin

One looks at the first column in the original table and see multiple rows with the same name. The change in quality is tied to the name. The triad (Customer, Developer, Tester) might recognize that the same quality changes could apply to multiple items. Or they might simply decide when putting the information on a whiteboard that the long name was too much effort to write. The table could be refactored into two parts, tied together by a type which indicates how the quality changes.

```
Scenario: Quality changes each day for items (Type Scenario)
* Quality change type for specific items
| name                                  | type     |
| Non-specific-item                     | NORMAL   |
| Aged Brie                             | BRIE     |
| Sulfuras, Hand of Ragnaros            | LEGACY   |
| Backstage passes to a TAFKAL80ETC concert | PASS |
```

The type in this scenario could be represented by a named string, a constant int value, an enumeration, or something else. It is used in the next two scenarios in place of the name.

```
Scenario: Quality changes each day for items (Change Scenario)
* Quality changes after a day passes based on item type
| type       | cq | cs | rq | notes            |
| NORMAL     | 1  | 1  | 0  | quality decrease |
| NORMAL     | 0  | 1  | 0  | never below 0    |
| NORMAL     | 4  | 0  | 2  | twice as fast    |
#
| BRIE       | 4  | 1  | 5  | increases        |
| BRIE       | 4  | 0  | 6  | increases twice  |
| BRIE       | 4  | -1 | 6  | increases twice  |
| BRIE       | 49 | -1 | 50 | 50 limit         |
| BRIE       | 50 | -1 | 50 | 50 limit         |
#
| LEGACY     | 80 | 1  | 80 | never changes    |
#
| PASS       | 1  | 11 | 2  | increase by 1    |
| PASS       | 1  | 10 | 3  | increase by 2    |
| PASS       | 1  | 6  | 3  | increase by 2    |
| PASS       | 1  | 5  | 4  | increase by 3    |
| PASS       | 1  | 0  | 0  | after concert    |
| PASS       | 50 | 11 | 50 | 50 limit         |

Scenario: SellIn changes each day (SellIn Scenario)
* SellIn changes after a day passes except for LEGACY type
| name       | cs  | rs | notes                                      |
| NORMAL     | 1   | 0  | SellIn down by 1 for anything but LEGACY   |
| NORMAL     | -1  | -2 | Regardless of value                        |
| NORMAL     | 100 | 99 | Is there a maximum SellIn?                 |
| LEGACY     | 1   | 1  | SellIn never changes
```

The implementation could use a switch statement, a hierarchy, delegation, or something else to represent the type and perform different quality changes based on that type. Note that these tests are independent of the implementation (an aspect that Kent Beck refers to structure insensitive at https://medium.com/@kentbeck_7670/test-desiderata-94150638a4b3);

These scenarios represent tests for pieces of the functionality. You should have one or two cases that represent the integration of the separate pieces.

```
Scenario: Quality changes each day for items - combines individual scenarios
* Quality changes and sellin changes after a day passes for specific items
| name                                | cq | cs | rq | rs | notes            |
| Non-specific-item                   | 1  | 1  | 0  | 0  | quality decrease |
| Aged Brie                           | 4  | 1  | 5  | 0  | increases        |
```

Code Notes

The code was refactored to allow the preceding tests to run. The refactorings were pretty standard, except for replacing string comparisons to an enumeration comparison. The resulting code was easier to read since it was shorter and less complex. It still needs refactoring into a switch statement with calls to individual methods for updating the quality for each type. That's left as an exercise for the reader.

Since the SpecFlow tests did not use any instance variables, Visual Studio suggested that the tests could be made static. Visual Studio also suggested checking for nulls passed as parameters, so a check for that was made at the appropriate places.

New Requirement

Here's the new requirement:

> *We have recently signed a supplier of conjured items. This requires an update to our system:*
>
> *- "Conjured" items degrade in Quality twice as fast as normal items*

This means that there is a new type of quality change (call it DOUBLE), which gets added to the Change Scenario.

```
Scenario: Quality changes each day for items (Change Scenario)
* Quality changes after a day passes based on item type
| type       | cq | cs | rq | notes            |
| DOUBLE     | 2  | 1  | 0  | quality decrease |
| DOUBLE     | 0  | 1  | 0  | never below 0    |
| DOUBLE     | 4  | 0  | 0  | twice as fast    |
```

The requirement needs a little more detail. Is the Conjured attribute part of the name or a separate attribute for an item? The answer affects how Type Scenario is determined. It does not affect the Change Scenario. The first one below shows Conjured as part of the name, the second one, as a separate attribute.

```
Scenario: Quality changes each day for items (Type Scenario)
* Quality change type for specific items
| name                                  | type      |
| Non-specific-item                     | NORMAL    |
| Non-specific-item Conjured            | DOUBLE    |
…

Or

Scenario: Quality changes each day for items (Type Scenario)
* Quality change type for specific items
| name                      | Conjured | type   |
| Non-specific-item         |          | NORMAL |
| Non-specific-item Conjured | Yes      | DOUBLE |
```

The notes accompanying the kata stated that Item cannot be changed. So, the scenario would be the first one. The implementation is constraining the requirements.

Summary

Creating implementation-independent Gherkin scenarios (tests) helps in understanding the requirements. Since the tests match the requirements, it can make it easier to introduce a change to the requirements.

Contact

Ken Pugh

ken@kenpugh.com

https://kenpugh.com

[i] Another approach (along the lines of Michael Feathers) is to spawn a method or wrapper a method to handle the new requirement. This could be done without adding unit tests except for the new functionality.