How Do We Know We're Done? One Way to End the Story

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

"How Do We Know We're Done?" is a question often asked by agile teams. I'd like to show a narrative of how this question might be answered. As the example, I'll used one from James Shore's new edition of *Art of Agile Development*, Chapter 9, Customer Examples. He describes a meeting between the product owner and the developers to gather information on a story[1] The section ends with the developer's taking a picture of a whiteboard of a business rule and heading off to develop. Here's a possible narrative for the rest of the story of "How Do We Know We're Done".

The meeting is attended by all perspectives of the Triad – customer, developer, tester. These are perspectives, not roles, so a developer might take a tester perspective. In BDD/ATDD, the Triad agrees on a shared understanding of the behavior. They document the shared understanding with readable scenarios. The scenarios become a test of whether that behavior has been implemented correctly. So let's see how this would work in the book's context.

The First Example

The first story is "For an invoice hasn't been sent, an account rep can delete the invoice, and if it has been sent, they can't." The rule can be stated as:

```
Scenario: Business Rule Admin Can Delete an Unsent Invoice
* Account Rep can delete unsent invoice
| User Role    | Invoice Sent   | Can Delete   |
| Account Rep  | N              | Y            |
| Account Rep  | Y              | N            |
```

In developing scenarios, you want to use the same domain terms that related scenarios use. User Role is a previously defined domain term that applies to many scenarios. There is a second term – Invoice Sent – which needs to be understood. The Triad agrees on:

```
Scenario: Domain Term Invoice Sent
* Invoice is sent when it is
| Emailed            |
| Printed            |
| Exported as PDF    |
| Exported to URL    |
```

[1] Here's an on-line reference to the chapter: https://www.oreilly.com/library/view/the-art-of/9781492080688/ch09s06.html#idm45110131119424

These two scenarios are good. The Triad also comes up with how this story is going to be demonstrated at the end of development. The demonstration shows the behavior with production good. A simple demo might be:

```
Scenario: Account Rep Can Delete an Unsent Invoice
Given an invoice exists
| Status      |
| Submitted   |
And the user is
| User Role     |
| Account Rep   |
When the user attempts to delete the invoice
Then the invoice is deleted
```

The person doing the demo will show a submitted invoice and then execute whatever is necessary to delete the invoice. Note that the scenario does not specify the clicks – the display behavior[2], but the functional behavior. The Triad agrees that this is the correct behavior of a portion of the requirement. The developer should check that this test passes prior to the demo. It's a higher-level test that ensures the behavior incorporates the business rule.

This check for this behavior can be automated for regression purposes. It also can be used in user documentation. This one document serves multiple purposes.

There will also be a scenario for the other half of the rule, which will also be demo'd.

```
Scenario: Account Rep Cannot Delete Sent Invoice
Given an invoice exists
| Status    |
| Emailed   |
And the user is
| User Role     |
| Account Rep   |
When the user attempts to deletes the invoice
Then the invoice is NOT deleted
```

A More Complicated Scenario

The later part of the section in the book shows a more complicated business rule. It says that there are other User Roles that can delete sent invoices. If the invoice has been sent and then deleted, an audit trail must be created. The example in the book used this table to show who could and could not delete invoices:

```
Scenario: Business Rule Delete an Unsent Invoice
* Given User Role Can They Delete Invoice
| User Role        | Invoice Sent   | Can Delete   | Audit   |
```

[2] If display behavior is important, then the scenario can be written to include that.

```
| Account Rep      | N               | Y             |          |
| Account Rep      | Y               | N             |          |
| CSR              | N               | Y             |          |
| CSR              | Y               | N             |          |
| Manager          | Y               | Y             | Y        |
| CSR Supervisor   | Y               | N             |          |
| Admin            | Y               | Y             | Y        |
```

Just like before, the triad creates a scenario that shows how this business rule is incorporated in the flow. Here is a demo for a Manager who requires an audit if they delete a sent invoice.

```
Scenario: Manager Can Delete Sent Invoice
Given an invoice exists
| Invoice ID   | Status   |
| I671         | Emailed  |
And the user is
| ID   | User Role |
| Sam  | Manager   |
When the user attempts to deletes the invoice
Then the invoice is deleted
And an audit entry is made
| Invoice ID   | Status    | ID   | Role      | Action   | Date Time
|
| I671         | Emailed   | Sam  | Manager   | Delete   | CURRENT
|
```

The triad can specify what they expect should be on an audit entry. Since an audit is a business operation, the customer ought to be involved.

Refactoring Business Rules

This business rule appears fairly simple. But what if there were a number of roles that could delete invoices? The rule can be refactored into two rules by separating concerns of who has a privilege and what is done with that privilege. This refactoring eliminates redundancy at the behavior specification level, which may eliminate some in the code. Here's the original example:

```
Scenario: Business Rule Delete an Unsent Invoice
* Given User Type Can They Delete Invoice
| User Role        | Invoice Sent    | Can Delete    | Audit    |
| Account Rep      | N               | Y             |          |
| Account Rep      | Y               | N             |          |
| CSR              | N               | Y             |          |
| CSR Y            | y               | N             |          |
| Manager          | Y               | Y             | Y        |
| CSR Supervisor   | Y               | N             |          |
| Admin            | Y               | Y             | Y        |
```

The triad recognizes that there is a Delete Invoice Privilege, which User Roles haves. This becomes a domain term in the ubiquitous language. There are two versions – Delete Unsent Invoice and Delete Sent Invoices. So, the scenario is separated into one that shows what that privilege does and who has what privilege.

```
Scenario: Business Rule Delete Invoice Privilege to Delete
Invoice
* Delete Invoice Privilege to Delete Invoice
| Delete Invoice Privilege  | Invoice Sent  | Can Delete  |
Audit |
| Delete Unsent Invoices    | N             | Y           |
|
| Delete Unsent Invoices    | Y             | N           |
|
| Delete Sent Invoices      | N             | Y           |
|
| Delete Sent Invoices      | Y             | Y           | Y
|
```

The other scenario shows which roles have what privilege. Any role not listed does not have the privilege.

```
Scenario: Business Rule Delete Invoice Privilege for User Roles
* Delete Invoice Privilege for User Roles
# All other roles have no delete invoice privilege
| User Role    | Delete Invoice Privilege  |
| Account Rep  | Delete Unsent Invoices     |
| CSR          | Delete Unsent Invoices    |
| Admin        | Delete Sent Invoices      |
| Manager      | Delete Unsent Invoices    |
```

Someone with a testing perspective would add additional scenarios, such as one to check that users without the privilege really cannot delete an invoice.

End Notes

When the Triad uses BDD/ATDD to define the required behavior, the result is both clear definitions of business rules and the flow in which the business rules execute. The behavior of the flow forms the demonstration that the requirement has been understood and implemented correctly, so that "We Know We're Done".[3] BDD/ATDD can considerably reduce the rework caused by unclear requirements. The scenarios are executable documentation. The scenarios represent the requirement. If a requirement changes, then the corresponding scenario changes.

[3] There may be other testing, such as exploratory and usability, that may discover defects not checked by these tests.