Feature Mapping - a simpler path from stories to executable acceptance criteria

John Ferguson Smart

BDD Advisor GCSC

Table of Contents

Introduction

This article is about an important but sometimes-tricky part of agile requirements discovery: writing good acceptance criteria.

In Behaviour Driven Development (BDD), we try to define acceptance criteria as "executable specifications", in a way that lends itself to automation. Each BDD acceptance criteria illustrates a concrete example of a business rule or user interaction with the system. Sometimes we use tables to group several related examples together, to make them easier to compare and contrast.

One common way of expressing acceptance criteria is to use the well-known "Given-When-Then" format:

```
Given <some precondition>
When <something happens>
Then <we expect some outcome>
```

This format is a great way to make sure that we are thinking in terms out the outcomes we want to achieve. After all, the outcomes of an application are where the value lies. Given-When-Then scenarios are also great for automation and producing living documentation, thank to tools like Cucumber and Specflow.

When Given-When-Then gets misused

Unfortunately, the Given-When-Then format can be tricky and is prone to misuse, especially when folk are new to BDD.

For example, suppose we are writing an educational application, where students submit essays to be marked by a teacher. When a teacher reviews an essay, she needs to add a mark of 1 to 10 in each of three categories (spelling, reasoning, relevance). If the student gets 6 or less in any category, the teacher has the option of returning the essay to the student to review and add corrections. If not, the essay goes into the student’s official results.

We might write an acceptance criteria using the "Given-When-Then" format" like the following:

```
Scenario: An essay that scores less than 5 in any area should be returned to the
student for correction
  Given Stuart the student has submitted his Politics 101 essay for review
  And Tess the teacher has chosen the essay to mark
  When Tess gives the essay a mark of 4 in spelling, 6 in reasoning and 6 in relevance
  Then Tess should be able to return the essay to Stuart to be corrected
```

This scenario gives a nice illustration of one particular path through the application workflow.

However, I often also see acceptance criteria that look more like this:

```
Scenario: Reviewing an essay
  Given a teacher opens a students essay to review
  Then she should be able to enter the marks for each category
  And if the 'reasoning' mark is less than 7, then the 'relevance' mark cannot be
above 6
  And the minimum mark in any category should be 0
  And if a mark is below 0, then the user shall get a "Mark not allowed" error message
  And if the marks are 6 or below, then the 'return to student for correction' button
should be enabled
```

As you can see, there's a whole lot of stuff going on in this acceptance criteria, way more than we would want to check with any one automated test. There are certainly some useful business rules buried in the text, but it bundles several rules and flows into a single scenario. This makes the requirement harder to understand, harder to automate and increases the risk of ambiguity or misunderstanding. Which in turn undermines one of the main motivations behind BDD, to build a shared understanding within the team.

When BAs write acceptance criteria like this, it leads to extra upfront work (writing in this format is not always the most natural thing to do for many BAs) and unnecessary rework when it comes to converting them into a format that can be automated. It also creates a disconnect for the product owners and business, between the the acceptance criteria they helped write or review, and the ones that end up being automated and turned into living documentation.

One way to address this problem is to teach BAs to write better Given-When-Then scenarios. While it is great for BAs to be able to express requirements in these terms, there are other more time-efficient ways of getting a story's acceptance criteria nailed down.

Introducing Feature Mapping

In the rest of this article, we look one such technique that can make it easier to go from user stories to acceptance criteria, acceptance that the business can relate to, and that we can automate as part of our automated acceptance test. **Feature Mapping** draws on Jeff Patton's Story Mapping, Matt Wynne's Example Mapping and other techniques to make it easier for teams to find good examples to use as the basis for the BDD acceptance criteria. In many ways Feature Mapping can be thought of as an extension of Example Mapping.

You use Feature Mapping when you need to define the acceptance criteria for a story from the backlog. Or, more precisely, when you need to agree on some acceptance criteria that can form the basis of automated acceptance tests, and become and automated part of the definition of done.

We start with the story

Feature Mapping always starts with a feature or story, typically from your existing product backlog. Feature mapping can be done shortly before sprint planning (to get a better understanding of what a story involves), or just before development starts on that story (more appropriate if you are using a Kanban-style process).

Feature Mapping sessions should typically be fairly short: less than half an hour for a reasonably well-understood story.

Let’s look at an example. Suppose we have the following story for our educational application:

```
Feature: Teachers can return bad essays to be corrected
In order to allow students to learn from their mistakes
As a teacher marking student essays
I want to be able to return an essay to a student for corrections when the marks are
poor
```

Feature Mapping works best as a team activity, with the Three-Amigos trio of a BA or product owner (representing business knowledge), a developer (representing the technical perspective) and a tester (to make sure the requirements are testable). BAs might write some initial acceptance criteria for the story beforehand, and that’s great. But these acceptance criteria can simply be one-liner business-rules, and not fully-blown Given-When-Then statements.

For the story shown here, a BA might note down the following initial business rules:

- The minimum mark in any category should be 0
- If the 'reasoning' mark is less than 7, then the 'relevance' mark cannot be above 6
- A teacher can return an essay to a student if any mark is 6 or below
- A teacher must return an essay to a student if it is being reviewed for the first time and at least one mark is below 5

In a less ideal world, you might have a story with acceptance criteria that look like the ones we saw earlier. I find that acceptance criteria like these tend to need a fair bit of refactoring, and it is generally easier to consider them as a source or starting point for discussions during the Feature Mapping workshop.

If a story doesn’t have any initial acceptance criteria or business rules to start with, that’s fine too; you will discover them during the Feature Mapping exercise.

Good acceptance criteria generally avoid talking about user interfaces, field formats, buttons and so forth, as the acceptance criteria are rarely a good place to put these. Mockups or wireframes work much better for this.

We understand the actors

Most stories involve at least one, and sometimes several, actors. Understanding what actors are involved helps us reason about the tasks they need to perform to achieve the story goals. In our case, we can identify two actors: the student who submits the essay (let’s call this actor Stu), and the teacher, who marks the essay (Tess).

We break the feature into tasks or steps

Next we break the feature into steps or tasks. How does the actor (or actors) interact with the application to achieve the goal of this story? What tasks do they need to perform? If you were manually demonstrating how this feature worked, or that it worked, what are the steps you would need to do? This is similar to Story Mapping, but we are focusing on a particular feature or story, rather than trying to get a high level view of the application.

In the story shown above ("Teachers can return bad essays to be corrected"), we might identify five main tasks:

- The student submits his essay for marking
- The teacher opens the essay
- The teacher records marks for each category
- The teacher returns the essay to the student for correction
- The teacher saves the final results

When we lay them out horizontally, we get something like this:

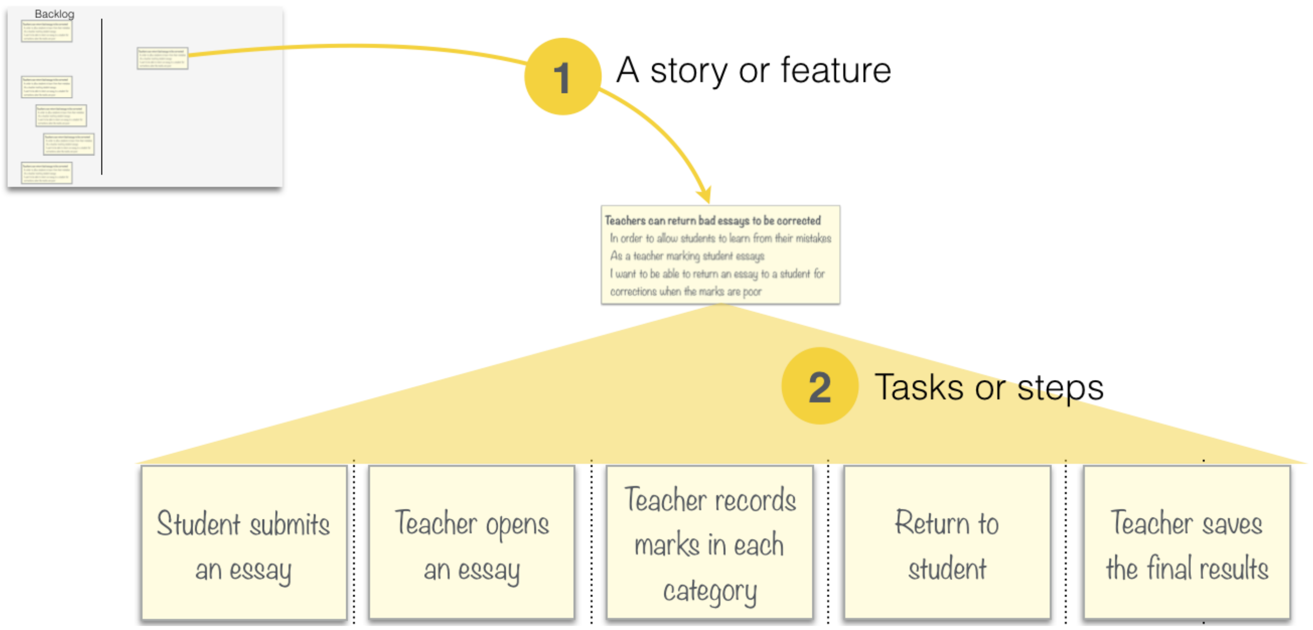
Backlog
1
A story or feature
Teachers can return bad essays to be corrected
In order to allow students to learn from their mistakes
As a teacher marking student essays
I want to be able to return an essay to a student for corrections when the marks are poor
2
Tasks or steps
Student submits an essay
Teacher opens an essay
Teacher records marks in each category
Return to student
Teacher saves the final results

Figure 1: We break a story into tasks or steps

Examples, Rules and Questions

Finding examples

Once we have an idea of the tasks or steps involved in achieving the story goal, we talk through concrete examples of these steps. Each example illustrates a different flow through the steps. For this story, we might start with a simple example of a student (let's call him Stu) who gets good marks, and a teacher (Tess) who records them and saves them in the final results of the exam records.

```
Example: Stu got 9 in every category so his essay is saved in the final results
```

If we map this out into the various tasks we identified earlier, we would get something like this:

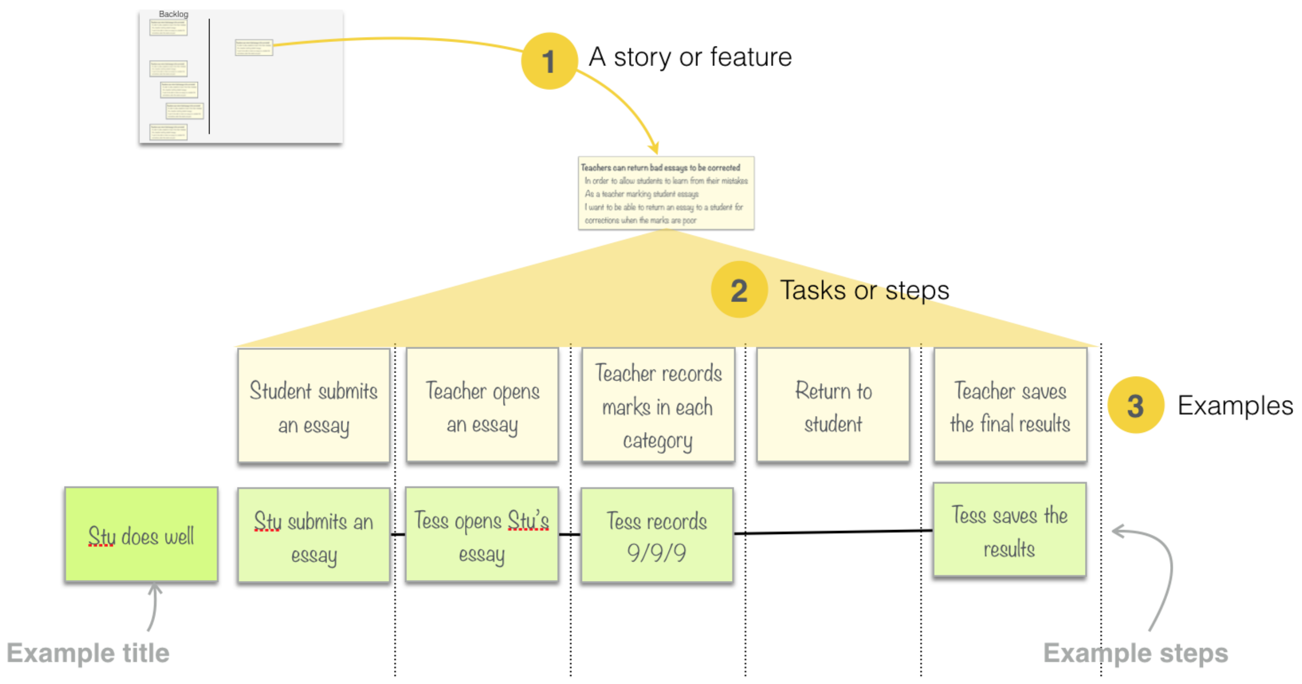
Backlog
1 A story or feature
Teachers can return bad essays to be corrected
In order to allow students to learn from their mistakes
As a teacher marking student essays
I want to be able to return an essay to a student for corrections when the marks are poor
2 Tasks or steps
3 Examples
Student submits an essay
Teacher opens an essay
Teacher records marks in each category
Return to student
Teacher saves the final results
Stu does well
Stu submits an essay
Tess opens Stu's essay
Tess records 9/9/9
Tess saves the results
Example title
Example steps

Figure 2: Mapping examples

This example might be called a "happy path", but we don't stop here. We look for other examples that illustrate different flows through the story. At each step or task, we can ask questions like: "what else could happen here?" and "what other inputs would change the outcomes?"

For example, what happens if Stu submits a poor essay? We could write another example, like the following:

```
Example: Stu got 4 in spelling and his essay is returned for correction
```

If we map this on our feature map, we might get something like this:

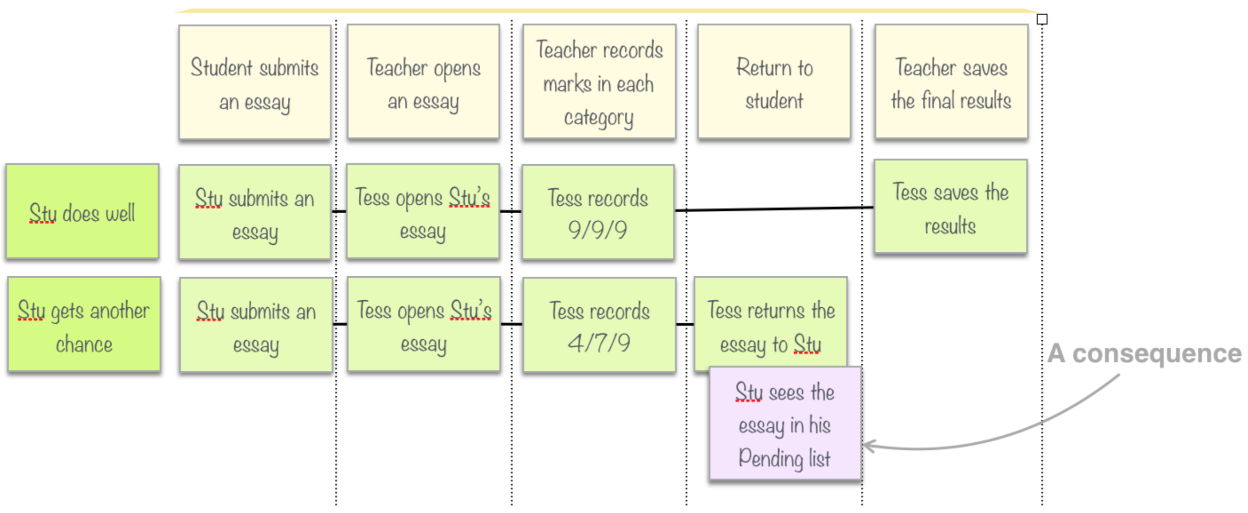

Figure 3: Adding a counter-example

In this map I’ve added a second row in the examples, to represent the case where Stu submits an essay but forgot to use the spell checker. There is something interesting happening in the 'Return to Student' column. I have added a card called "Tess returns the essay to Stu", but this seems a little inconclusive. The real goal is that Stu receives the essay to correct, not just that Tess has sent it. How do we know that Stu actually received it? To make sure this is the case, and to emphasise the actual outcome we want, I’ve added a *Consequence* card (in mauve to make it easy to distinguish from ordinary example steps). You don’t always need *Consequence* cards, but often they are handy when you want to communicate the expected outcomes more clearly.

This conversation might lead to other examples. Stu got a 4 in spelling, but good marks in other areas, so Tess gave him a chance to correct his work. What would happen if he scored 4 in every category? Would Tess still give him a second chance? What if this was the second time he submitted his essay? And so on. Each of these might become an example; some (such as the question of what happens when he submits his paper several times) might even be refactored as acceptance criteria for a different story.

You may notice the way I have used lines to connect the tasks, to show the branches and variations in the flow. I find this makes the map clearer and easier to understand. However you could also just repeat the tasks, and have a full row of tasks for each example if you find this expresses your intent better.

Rules explain the examples, examples illustrate the rules

In Example Mapping, we say that examples illustrate business rules, and business rules explain (or give context to) the examples. Both of the previous examples illustrate the same business rule: *A teacher can return an essay to a student if any mark is 6 or below*. To make this clearer, we can add a card to represent this rule at over the example title cards, as we do with Example Mapping:

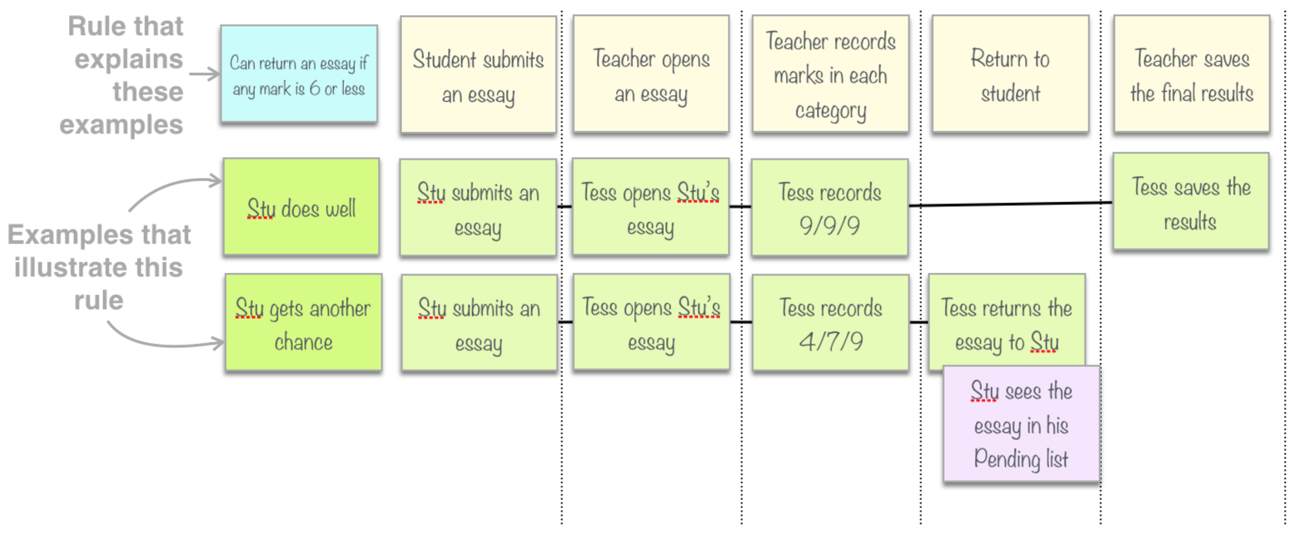
Rule that explains these examples
Can return an essay if any mark is 6 or less
Student submits an essay
Teacher opens an essay
Teacher records marks in each category
Return to student
Teacher saves the final results
Examples that illustrate this rule
Stu does well
Stu submits an essay
Tess opens Stu's essay
Tess records 9/9/9
Tess saves the results
Stu gets another chance
Stu submits an essay
Tess opens Stu's essay
Tess records 4/7/9
Tess returns the essay to Stu
Stu sees the essay in his Pending list

Figure 4: Rules explain examples

This lets us explore the scenarios in two dimensions. If we already have an idea of the rules (or some of the rules), we can take a rule and walk through some concrete examples that illustrate this rule. This helps us explore our understanding of the rules by looking for examples and counter-examples. Or we can continue to work through examples ("what else could happen in this task? How would that affect the outcome"), and add rules to explain the new examples as we need them.

Questions highlight uncertainty

The third element of Example Mapping comes into play when you discover something that no one knows. Suppose you know that some subjects marked on a scale of 1 to 10, whereas others from 0 to 10. The BA is aware of this rule, but doesn't know what subjects it relates to, or whether it is in scope for this story. So she writes it down on a *Question* card (a pink one, using the Examples Mapping conventions). I like to place these cards under the step they relate to, to give them some context and help identify complexity: for example, a step or task with a lot of questions associated with it indicates a lot of uncertainty might need to be refactored into a story of it's own.

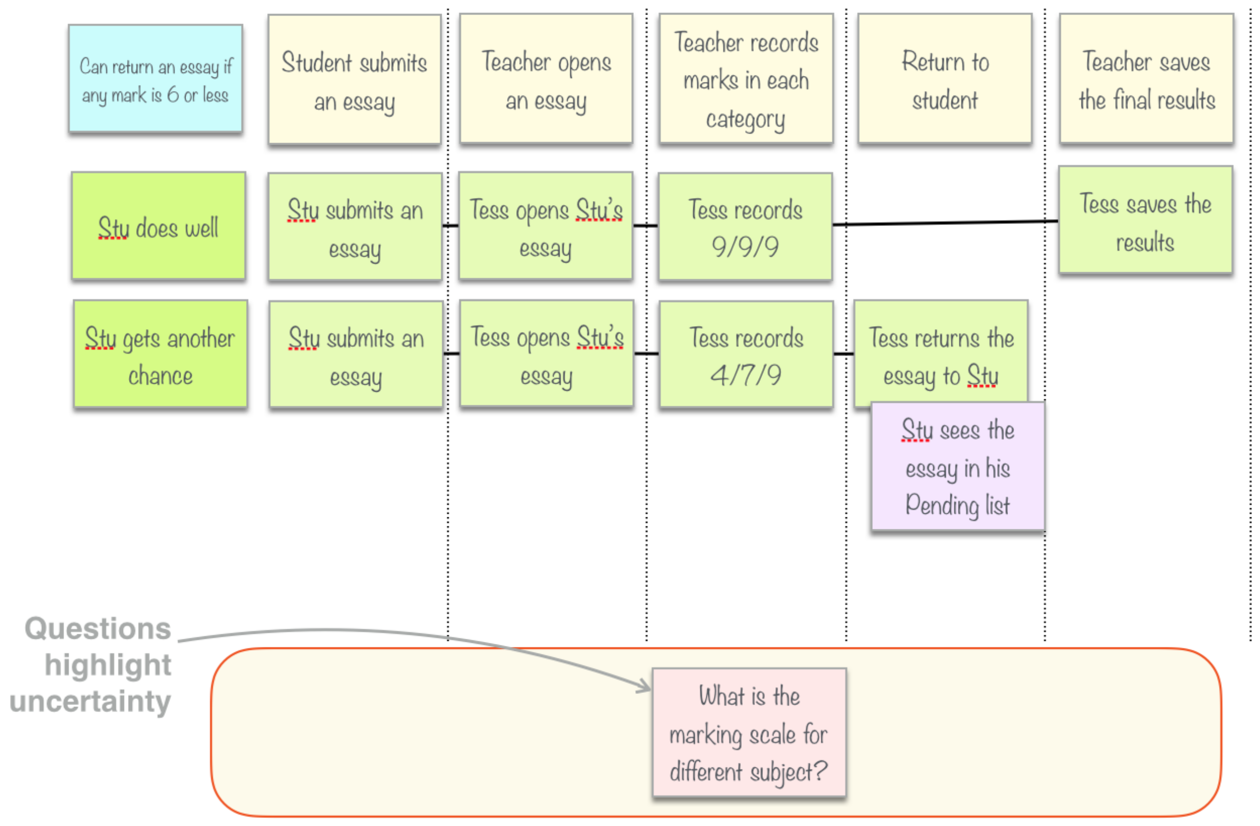
Can return an essay if any mark is 6 or less
Student submits an essay
Teacher opens an essay
Teacher records marks in each category
Return to student
Teacher saves the final results
Stu does well
Stu submits an essay
Tess opens Stu's essay
Tess records 9/9/9
Tess saves the results
Stu gets another chance
Stu submits an essay
Tess opens Stu's essay
Tess records 4/7/9
Tess returns the essay to Stu
Stu sees the essay in his Pending list
Questions highlight uncertainty
What is the marking scale for different subject?

Figure 5: Questions highlight uncertainty

Working with negative cases

The examples we've seen so far could be described as "happy-day cases", smooth flows through the story. But our acceptance criteria should also describe negative scenarios, especially if they are important to the business. Negative scenarios help understand the positive scenarios better, and help flush out incorrect assumptions or missing details.

Acceptance criteria should record both positive and negative scenarios. However, not *all* negative scenarios are useful. Field validation rules make for useful examples if the rules relate to business requirements, but a scenario checking that a badly-formatted date cannot be saved would normally be reserved for unit testing.

Suppose, after some investigation, we learn that the History Department has a police where History exams are marked from 1 to 10, and that our system needs to cater for history essays.

To illustrate this rule properly, we really need two examples: one that shows that you can't enter a 0 mark for a History exam, and another to show that we *can* enter a 0 mark for some other subject.

```
Example 1: Tess records 0/1/2 for Stu's English essay and it is returned for
correction
Example 2: Tess records 0/1/2 for Stu's English essay but she is not allowed save the
marks
```

We could add these examples to the feature map as shown here:

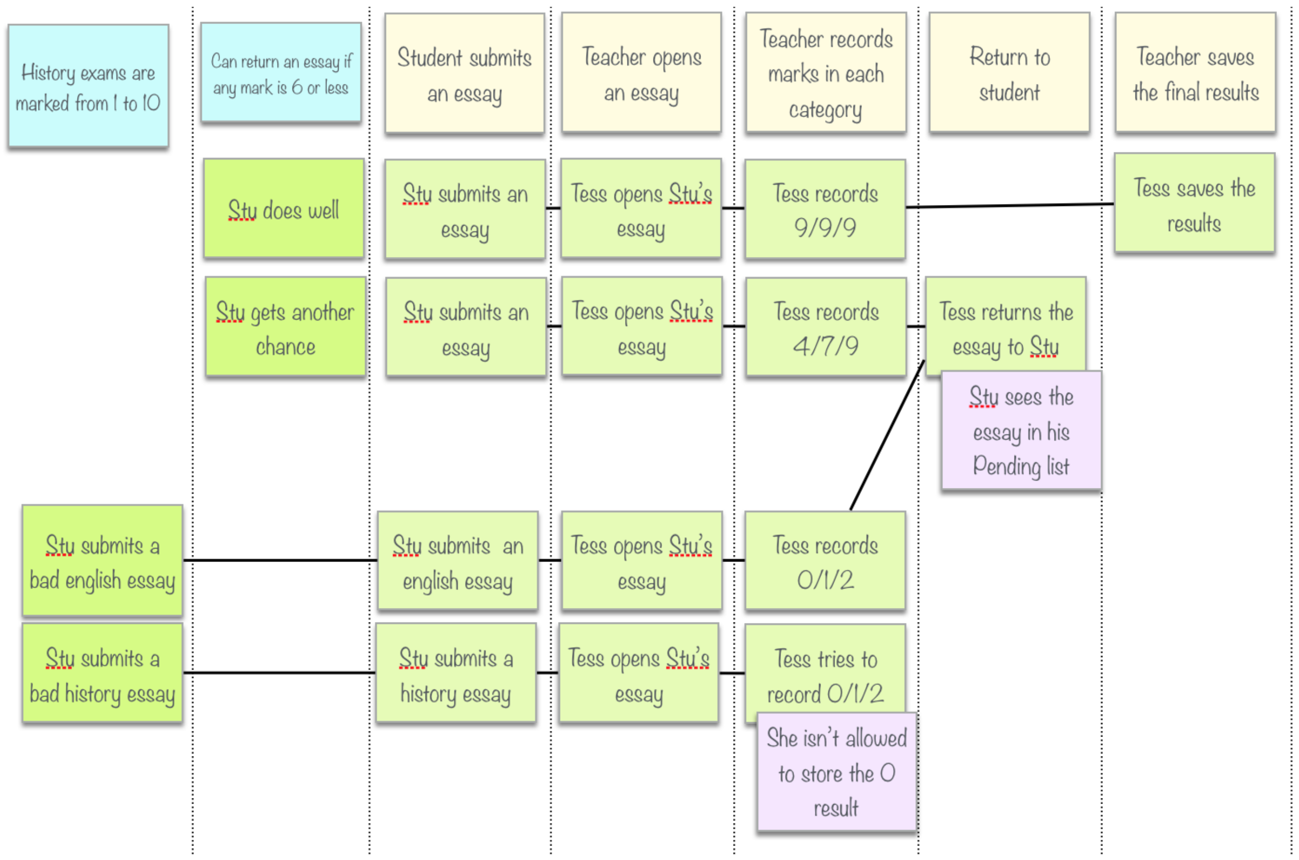
History exams are marked from 1 to 10
Can return an essay if any mark is 6 or less
Student submits an essay
Teacher opens an essay
Teacher records marks in each category
Return to student
Teacher saves the final results
Stu does well
Stu submits an essay
Tess opens Stu's essay
Tess records 9/9/9
Tess saves the results
Stu gets another chance
Stu submits an essay
Tess opens Stu's essay
Tess records 4/7/9
Tess returns the essay to Stu
Stu sees the essay in his Pending list
Stu submits a bad english essay
Stu submits an english essay
Tess opens Stu's essay
Tess records 0/1/2
Stu submits a bad history essay
Stu submits a history essay
Tess opens Stu's essay
Tess tries to record 0/1/2
She isn't allowed to store the 0 result

Figure 6: Representing negative scenarios

Note how we have described the negative outcome - just place a Consequence card to describe this outcome underneath the task where it happened.

Adding extra details

Sometimes we want to add some extra information. We might want to add the error message that should be displayed if an invalid mark is entered. Or we might want to add a table of test data describing different variations of the same scenario (for example, different mark weightings for different subjects). For situations like this, I generally just write the detailed message or the data table on the back of the task card, or sometimes on a card underneath the main task card for more visibility, so that we can refer to it later when we automate.

In this example our story is quite complete, and contains several distinct flows (in particular, the different flows for when an essay is returned to the student or saved). This is intentional, to make the example a little more interesting. However, you might decide that this feature would be easier to deliver if it were broken up into smaller stories, such as one for when the essay is returned to the student, and one for when it is definitively saved.

From Feature Mapping to Test Automation

One of the nice things about this approach is that we can start automation immediately. Each example maps to a clear sequence of business-level tasks, which in turn are easy to automate. In Cucumber, for example, we can map the steps more-or-less directly to steps in the Cucumber scenario:

```
Scenario: Returning an essay to the student for correction
  A teacher can return an essay to the student to be corrected if any mark is 6 or
less

  Given that Stuart has submitted an essay on 'Politics 101' to be marked
  And that Tess has opened the essay
  When Tess records the following marks:
    | Spelling | Reasoning | Relevance |
    | 6        | 6         | 6         |
  And Tess returns the essay to Stuart to be corrected
  Then Stuart should the 'Politics 101' essay in his Pending Correction list
```

Using the Screenplay pattern in Java, with either JUnit or Cucumber, we could automate these steps using code along the following lines:

```
Actor tess = Actor.named("Tess").whoCan(MarkPapers);
Actor stuart = Actor.named("Stuart").whoCan(SubmitPapers);

givenThat(stuart).wasAbleTo(Submit.anEssayAbout("Politics 101"));
andThat(tess).wasAbleTo(ReviewTheEssay.from(stuart).about("Politics 101"));

when(tess).attemptsTo(
    RecordMarks(of(6).in(Spelling), of(6)).in(Reasoning), of(6).in(Relevance)),
    ReturnTheEssay.forCorrections()
);

then(stuart).should(
    seeThat(HisEssays.thatAre(PendingCorrection), contains("Politics 101"))
)
```

Notice how cleanly the tasks map to steps in Cucumber and Java. The automated tests map closely to the business flow, both externally and internally, and this makes both the reporting clearer and the code easier to understand.

At this point, you can automate minimal implementations for your tasks, and you will have a pending executable specification which will act as a starting point for your test automation efforts.

Conclusion

Feature Mapping is a simple process that draws on Story Mapping and Example Mapping to define a path from reasonably well-understood stories to executable specifications. In summary, the overall process goes something like this:

1. Define a feature or story (or pick one from the backlog)
2. Understand what actors are involved in the story
3. Break the feature into tasks to identify the main flows, and lay the tasks horizontally
4. Identify examples that illustrates a principle or variant flow. Ask questions like "But what if…", "what else could lead to this outcome", "what other outcomes might happen", and use the answers to create new examples. Use rules to explain and give context to your examples
5. Rinse and repeat for other rules and examples
6. Create Executable Specifications: automate the main high-level flows with pending steps

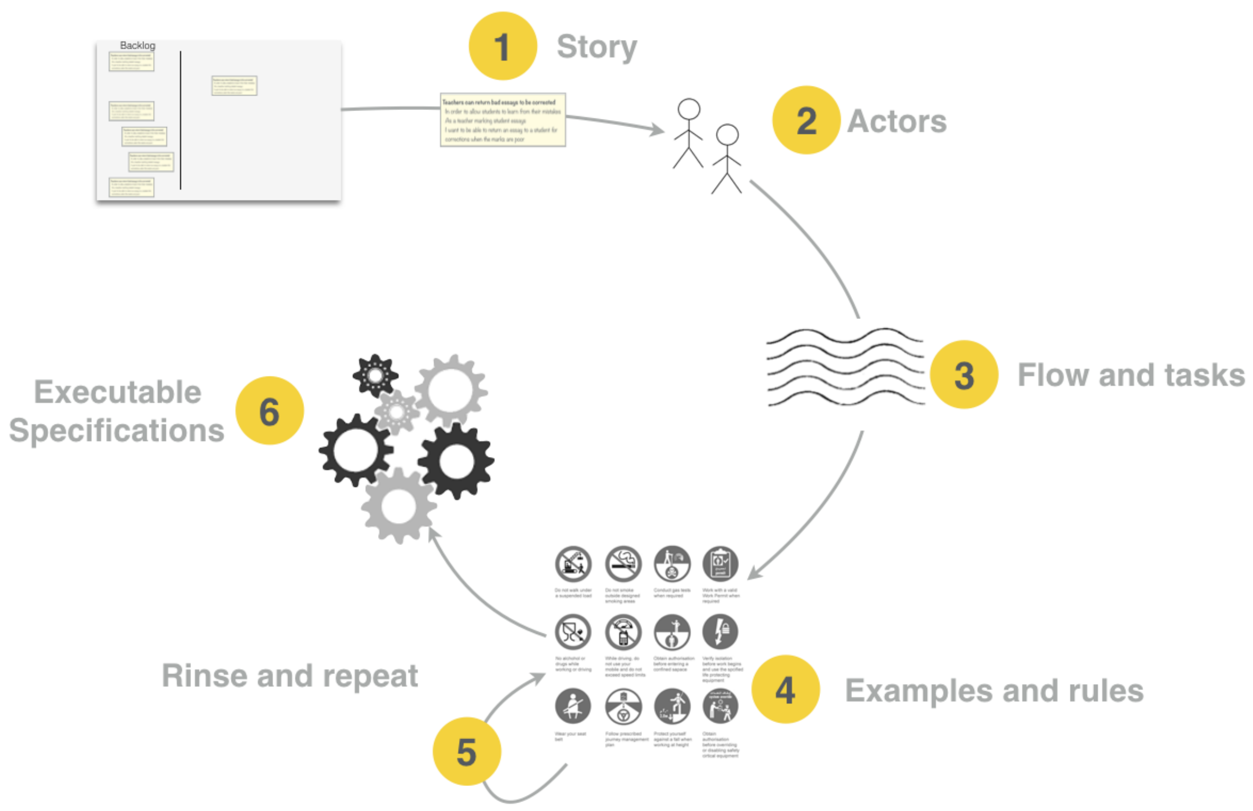

Figure 7: The Feature Mapping process

I have found this process extremely useful for many different types of projects, and breaking down the examples into tasks or steps is a great way to generate useful conversations around the finer points of the requirements. Like example mapping, it may be overkill for very simple stories, and can unveil complexity in larger ones. The process should be easy and flow naturally. If it doesn't, be sure to ask yourself why, and adjust course accordingly.

About the author

John Ferguson Smart (https://www.johnfergusonsmart.com) is an international speaker, consultant, author, and a well-regarded expert in areas such as BDD, TDD, test automation, software craftsmanship and team collaboration, and author of *BDD in Action*, *Jenkins: The Definitive Guide* and *Java Power Tools*. Contact John at john.f.smart@hsbc.com or reachme@johnfergusonsmart.com.