cucumber school

Lesson 7 Answers

Details

1 - Reflection questions

1.1 - The *premium account* feature

Can you remember the different criticisms we made of the original version of the premium account feature? What did we do about each one?

1. There was only one scenario
2. The scenario was very long
3. There were several When steps
4. No rules were documented in the feature's description
5. It contained incidental details

We gradually refactored the scenario to remove the incidental details and distill it down to its essence. As we did so it became clear that multiple rules were being tested. We spoke with our domain expert to clarify the rules, then documented them. We split the long scenario so that each rule was tested separately.

1.2 - Naming scenarios

Which TV series inspired our trick for naming scenarios? More importantly: how is it useful?

We talk about naming scenarios like they named episodes of *Friends*. This helps you to give a headline summary of what happens in the scenario.

1.3 - Pushing down the details

When we pushed the details down into the step definitions, we changed the content of the messages. For example, the message "Come and buy a coffee" became "a message containing the word buy". Why did we do that? Were there any risks in doing this?

We decided that the detail of the exact words of the message was *incidental* to the scenario, and actually distracted the reader from understanding the *essence* of the scenario.

Changing the words was a choice, again to remove incidental details. The old message, mentioning coffee, could have confused someone reading the code that the word "coffee" might also have some significance. Once details go out of sight, our experience is that it's better to deliberately make them boring so it's obvious they're not meant to be seen.

The risk of doing this was that, unknown to us, there *was* some significance in the message other than the word buy. By changing the words in the message we were changing the test. It's possible, if we didn't fully understand the consequences, that we might have broken the test.

1.4 - @todo

Tamsin's bug scenario was tagged with `@todo`. How did we make use of that tag during the lesson?

We excluded the tag from our test run using `~@todo` so that we only ran the scenarios that we expected to pass during our refactoring.

1.5 - When have you gone too far?

When teams get obsessed with removing detail from their scenarios, they can go too far. Taken to an absurd level, they would write one scenario for their whole system:

```
Given the system is running
When I use it
Then it should work, perfectly
```

What are the disadvantages of making your scenarios too abstract? How would you know when you've gone too far?

When a scenario has too little detail in it, the reader is forced to place more trust in the authors of the step definition code beneath. Imagine how much code you'd need in the step definition for `Then it should work, perfectly.`

Scenarios need enough detail in them to be vivid and interesting to read, but only just enough - too much and the detail becomes distracting. We can't tell you where to draw this line - different people on your team will probably disagree on it. The important thing is to find a compromise that you're all happy with. The process of reaching this agreement

will teach you a lot about one another's perspective, and perhaps bring you closer as a team.