cucumber school

Lesson 5 Answers

Loops

1 - Reflection questions

1.1 - Removing tests

In this episode we removed two tests. What were our reasons for deleting each one? What other reasons can you think of to delete a test?

We removed the scenario 'Listener hears a message' because it was a duplicate of the 'Listener is within range' scenario.

We removed the scenario 'Listener hears a different message' because we judged that the code was very unlikely to regress in this way. The risk of such a regression was not significant enough to justify keeping the test.

Every automated test carries a cost with it: there's the cost of waiting for it to run, the cost of maintaining it, the cost of navigating past it each when you're trying to find something else nearby it. For a single test these costs are almost insignificant, but they add up. So each test must justify its existence in your suite. Never be afraid to delete a test that's no longer giving you the benefit you need from it.

Other reasons to delete a test might be:

- the behaviour is already guaranteed by another test, perhaps in a unit test.
- the behaviour being tested has been extracted out into a library or other 3rd party component.
- the test has become unreliable, failing randomly. You've tried to fix it but the effort will outweigh the benefit of it.

1.2 - The BDD/TDD loops

We mentioned two loops in the whole BDD process. Can you explain what they are and the steps they're composed of?

The outer loop is an acceptance test, where you decide what you want to build, describing the desired behaviour in collaboration with your customer or user representative.

When you have written a failing acceptance test, you then start to drive out behaviour in your solution, using unit tests to guide the individual changes you make to classes within your system.

You write a unit test, watch it fail, then make a change to the system to make the test pass. Finally, you clean up your code to make it easier for others to work on in the future.

Finally, once your unit tests are passing, you can run the acceptance test again. If that's passing, you're done! If not, you can dive back in and make more test-driven modifications to the system until the acceptance test passes.

1.3 - Outside-in development

Outside-in development is often mentioned in the world of BDD. With what you learned so far, do you have any idea of what it is about? How does it compare to other, more traditional development approaches?

Outside-in means starting from "why?" and using that to help decide "what" and finally "how" to solve the user's problem. Asking "why" helps us to understand the value our user or stakeholder will get from the work we're doing.

This works at every level on a project. From prioritising big features, right down to programming individual edge-cases.

1.4 - Cleaning up

We said one of the important activities of TDD is cleaning up. It's called refactoring. Why should you care about that? How would you convince your boss (who's not into technicalities) that refactoring is not optional? Try to think about the business benefits of this practice.

Refactoring means improving the design of existing code without changing its behaviour. This is an investment for the future. If you're 100% certain that nobody will ever touch this code again then there's no point in refactoring it. You can walk away from the kitchen and leave it to rot, because nobody will ever need to go back in there.

However most code - particularly on successful business applications - spends a lot longer being changed than it does being written in the first place. So generally this is a worthwhile investment.

Code that isn't being refactored will gradually become harder and harder to change, until the cost of an individual change might become higher than the business value that it will deliver. At this point companies start thinking about re-writing their business applications. This can get very expensive.

Some people use the metaphor of technical debt for this. If you don't pay back the loan, you'll pay higher and higher maintenance (interest) costs.

1.5 - It takes too much time

Newcomers to TDD and BDD often worry that working this way will have a negative impact on their productivity, adding a lot of overhead and therefore slowing down development. What do you think about those claims?

It's true, and worth admitting, that learning TDD takes time. The time to climb that learning curve will undoubtedly have a short-term impact on your productivity.

Once you have the hang of it, you'll find that you spend a little bit more time up-front, figuring out what test to write, and perhaps going back to your stakeholders to clarify things that the tests are forcing you to figure out. The payoff comes once you've done your implementation. You'll find the old circus of bouncing back and forth between testers and developers just doesn't happen anymore. This demoralising 'rework' cycle just disappears.

If you're evaluating a team-wide adoption of TDD/BDD, It's worth measuring how much time you spend right now in bugfixing. Now try applying TDD/BDD to one of your user stories, and measure how that changes.