Dealing With Bugs Using Impossible Tests.

13 April 2020 · automation, bug, selection randomness, testing

“Muggins! 2 for 15. My points!” My son shouts in my face.

I frown and re-examine my cards, sure enough, I’ve missed a card combination.

I scramble for a response… “Err… ACC Rule 10 sub-section 1 part (b) states I have to be informed prior to the game that the rule is in operation!”

He’s not impressed, But sounding confident is my only hope here. I think the inclusion of the rule details has made him think twice about arguing. (It’s the only rule I know off by heart, and it’s a lifesaver. I might even get it tattooed on my arm.)

For the uninitiated, we were playing Cribbage. An old English card game, now popular around the world. Cribbage has an interesting system of scoring, that you progressively learn the more you play.

The Muggins rule my son referred to is an optional rule whereby if you underscore your hand the opponent can claim those points. As such it’s become a small obsession of mine to learn all the scoring rules and never have to get that muggins rule tattoo.

To help me learn the rules of Cribbage, I wrote code to calculate the scores for me. The development and testing of that library helped me grok and be able to calculate cribbage scores more quickly. Having learned the rules, I am now better able to identify mistakes made by myself and others.

> I didn’t have to manually work out test data that might cause a failure.

The library has a collection of unit tests that helped check my code during initial development using TDD and also during updates and refactoring. These tests are very useful, but they are all essentially scripted by me or based on the websites that I have read. This narrow source of test data makes them not as comprehensive as they could be. They might highlight correct examples of scoring, but would they find unexpected failures?

To provide tests or at least test data that are ‘off the beaten path’ and potentially highlighting new issues, I developed a simple test that randomly creates test data and checks the result. I can take advantage of a situation that should never occur in cribbage. In Cribbage, you can’t score 19. No card combination adds up to a score of 19 (or 25, 26 or 27 for that matter)

My test did not have to re-implement the library’s logic to check if the score is correct. It can use a short-cut: if the score returned by the library was ever one of these impossible scores

then the code had a bug. Run this test often enough, and you have another reasonable check of the scoring engine.

The beauty of this kind of test is that it scales with the time available. How many tests would we like to run? Well, how long do you want the tests to take? How confident do you want to be?

All this can be done, with essentially one parameterized test and one fixture, minimising the reams of brittle tests that often grow up around complex apps.

You can check out the python code yourself, on GitHub. (https://github.com/phoughton/cribbage_scorer/blob/73e57939180fcfe5e65d1f5140e5ea1103c69723/tests/show/show_scorer__impossible_score_test.py#L1)

I actually implemented this test last and ran the test after having ‘finished’ the initial development and testing. Sure enough, despite having a comprehensive set of existing tests for each feature of the scoring, there was a bug. The test correctly highlighted some impossible scores were being produced. Once I knew about the bug and the test data to cause it, it was trivial to identify and fix the issue.

I didn’t have to re-implement the logic to check the results and I didn’t have to manually work out test data that might cause a failure. The technique aims to implement [and therefore maintain] a minimum set of app logic.

We can find a host of possible arithmetic and logic errors just by checking if certain impossible ranges are infringed upon. This might be as simple as just checking if the result is negative (when they should always be positive) or checking for certain error responses that should never occur. As long as the data you are randomly choosing is valid for your app, and you know those responses should not occur, the process is fairly trivial to implement.

You will probably find the biggest barrier to acceptance is some people’s desire to define the data in advance. The commonly felt need to define explicitly what you put in and what you should get out. A test case.