REVIEW FEEDBACK

Holly Duckett 26/02

26 February 2021 / 09:00 AM / Reviewer: Pierre Roodman

Steady - You credibly demonstrated this in the session.
Improving - You did not credibly demonstrate this yet.

GENERAL FEEDBACK

Feedback: This was a good first review and you showed a reasonable understanding of what a TDD process should be. There are just a few refinements that you can make. The main one being not to add production code if you do not have a failing test for it yet.

I CAN TDD ANYTHING – Improving

Feedback: Your input-output table was serving well as you were able to base your tests on the behaviours as was agreed upon with the client. Your tests were therefore client-oriented and were able to provide immediate value to the client as each test passed.

You started with an outside-in approach which meant that you started with hard-coding and then wanted to iteratively build your algorithm using the RGR cycles to add transformations to your code. You did, however, stray from this approach and started focusing on how you wanted to add future test cases and wrote that code without creating a failing test that would introduce that logic, thereby going against the first law of TDD “You are not allowed to write any production code unless it is to make a failing unit test pass.”
I recommend that you read the following article about the 3 laws of TDD.
http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

I CAN PROGRAM FLUENTLY – Steady

Feedback: You are quite familiar with the terminal and how to set up your development environment and testing environment. You are aware of Ruby syntax and language constructs as well. You are familiar with array methods and string methods as you were making use of them, but I believe that you could perhaps practise using them more on a website like CodeWars in order to use them in a way where you will not have to debug too often.

You were developing a logical program and you were able to get as far as starting to create the code for multiple grade strings with various grades.

I CAN DEBUG ANYTHING – Strong

Feedback: You are familiar with common errors and read the error messages very well. You made excellent use of IRB in order to test your assumptions in your production code and this created a solid feedback loop that helped you to resolve the bugs that were introduced with the split and count methods without making random changes to the code. Well done.

I CAN MODEL ANYTHING – Steady

Feedback: You modelled your solution as a single method. I felt that this was a nice and simple place to begin and a good fit for this particular exercise as it did not require state and you could add methods later on when refactoring and extracting methods from your main method.

Your method name “report” follows the Ruby convention of naming methods in snake_case and was based on the client’s domain.

Your algorithm was growing nicely and after a few more RGR cycles it would have become more generic and took on more varied tests cases.

I CAN REFACTOR ANYTHING –Improving

Feedback: You chose to refactor the hardcoding from your first test, without there being any duplication in the code at that stage to generalise. This led you to add logic for future tests without having written the failing tests to justify that logic. Your refactoring should only be done for current

production code and tests and not for future tests as you will run the risk of overengineering and introducing bugs that are untested.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Improving

Feedback: You have prioritised core cases over test cases which is an approach that provides immediate value to the client.

You were not following a strict RGR cycle as you did not refactor after your second test. I suggest taking advantage of the benefits of refactoring after every green phase. This may help you to write clean code throughout the development process and leaves little room for introducing bugs during complex refactoring when it is left for later on. While you are still refining the TDD process and the RGR cycle, I suggest using the following checklist.
Write a failing test.
Did you run the test?
Did it fail?
Did it fail because of an assertion?
Did it fail because of the last assertion?
Make all tests pass by doing the simplest thing that could possibly work.
Consider the resulting code. Can it be improved through refactoring? If so, do it, but make sure that all tests still pass.
Ask yourself the question, ‘what is my code currently assuming’ and think of the next simple test that will break that assumption and introduce a new failing test.
Repeat

The following videos could help to visualise this process for you:
https://vimeo.com/43734265
https://www.youtube.com/watch?v=QbNhpPQkCBs

I USE AN AGILE DEVELOPMENT PROCESS – Improving

Feedback: You were asking relevant questions which meant that you were able to clarify the main requirements of the program relatively quickly and you were able to flesh out a really good input-output table with your own examples for test cases. This was great as you were able to have a good place to keep track

of the requirements that needed to be met and could just simply copy and paste the inputs and outputs into your tests.

I would encourage you to spend a little time considering any edge cases which may occur as a string is highly mutable and some common edge cases would be empty strings, incorrectly spelt grades, upper and lower case etc.

I WRITE CODE THAT IS EASY TO CHANGE – Strong

Feedback: You made regular use of Git as part of your process, committing whenever tests passed. This meant that the latest working version of your code was always available to be rolled back to in the event of adding program breaking code, this means that your code is more changeable as a result. Your commit message documented the context of the changes made meaning that it was easy for anybody looking at the commit history to see the progress of your program.

You had your test suite properly decoupled from your implementation by making sure the tests were based solely on acceptance criteria, and not reliant on the current implementation. This makes changes to the code much easier to change because refactoring the code or changing the implementation altogether will not break the test suite as the tests will always be relevant so long as the acceptance criteria are relevant.

You had sensible name choices for your variable and function names that were relevant to the client's domain such as "grade_array" and "report". Descriptive naming makes code easy to read and code that is easy to read and understand is easier to make changes to.

I CAN JUSTIFY THE WAY I WORK – Steady

Feedback: You were very vocal about what you were doing and why you were doing it. I was never in doubt as to what your reasoning was. I would just advise you to justify any reasons for straying from the process. Examples of such deviations are refactoring when there is no duplication and skipping a refactor phase. This will help you to determine if your reasoning is sound and hold you accountable to the process.