REVIEW FEEDBACK

Peter Allen 19/02

19 February 2021 / 11:00 AM / Reviewer: Pierre Roodman

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

GENERAL FEEDBACK

Feedback: It was a pleasure conducting your first code review. I can see that you are a good problem solver. I suggest just refining your process especially refactoring on every refactor phase as I have laid out in this review by practising on some code challenges using challenges from sites like CodeWars and employing TDD on those challenges. I look forward to conducting another review session with you and encourage you to book some more reviews.

I CAN TDD ANYTHING – Improving

Feedback: You have for the most part tested for behaviours only based on the acceptance criteria, which is great for maintaining flexibility in your implementation code as you can make changes without breaking the tests and means that your tests are client-oriented as well since they drive the development of the acceptance criteria as laid out by the client. Your first test, however, was testing if the method returned a string which is not quite necessary because when you start doing behaviour based tests, a critical assumption of those tests is that the returned output is a string. You could probably do without the first test and the second test that you created would have been a great test to start with.
You added count logic in your implementation when your tests were still single grade tests. This means that your implementation logic was not informed by the test, but rather you were adding logic that you had predetermined to be the solution to the problem. You were therefore following a bit of a mix of an inside-out approach and an outside-in approach. An inside-out approach may work well when you have a very strong understanding of what the final solution will be and therefore focuses a bit more on the implementation logic and how

to gradually create logic for specific parts of the algorithm and tie them up together later. An outside-in approach is concerned only with the high-level behaviour of the logic and makes no assumption of how the final implementation will be, but gradually builds the algorithm through hardcoding to pass tests and introducing small transformations of the logic through refactor phases and new tests on the red phases. The outside-in approach builds up the code in a very natural way with small increments at a time which has the side-effect of clean code development as well as code that is less likely to introduce complex bugs. The following resource may give you a better understanding of the 2 different approaches.
https://8thlight.com/blog/georgina-mcfadyen/2016/06/27/inside-out-tdd-vs-outside-in.html

You did not closely adhere to the RGR cycle and left refactoring for later on in the development cycle. This meant that you experienced a period during the process where you were trying to refactor, but the code at that point made the refactoring considerably difficult and after a while, you abandoned refactoring the code. Sticking to a more strict RGR cycle refactoring at each refactor phase may help you with this. I will discuss this more in the “I have a methodical approach to solving problems” section of the review.

Your tests for multiple grades were testing for commas without spaces which were not how you recorded the acceptance criteria from the examples that were provided. This was actually an acceptable input, but this was never confirmed with the client before starting to code. This led you to adjust all of your tests and changing implementation etc. when you discovered that the spaces were actually meant to be included and would actually be the most common input. I suggest not changing the tests. If your tests are testing a valid requirement, they are still relevant and are actually recording the progress of the development of the algorithm. Changing tests can also introduce invisible bugs as you no longer have tests that will test particular cases.

I CAN PROGRAM FLUENTLY – Improving

Feedback: You are quite familiar with the terminal and were able to set up your environment, run tests and use IRB quite smoothly. You are reasonably

familiar with Ruby syntax and language constructs and are familiar with array and string methods as well. I do, however, suggest a bit of practice with similar problems on a website like CodeWars in order to just get more used to when and how to use these built-in methods. Your algorithm was growing quite well and I believe given more time, you would eventually have developed the full solution.

I CAN DEBUG ANYTHING – Improving

Feedback: You are familiar with some common errors and are able to get those resolved quite easily. You know what the failing code of a new test on the Red phase should look like as well. You also used IRB in order to test your assumptions and create a feedback loop when making potential corrections to the code.

When you ran into errors whilst refactoring your code, you found yourself making what looked like random changes to the code in order to fix the bugs. I would suggest taking a step back in order to come up with a well-investigated hypothesis as to why the bug is occurring and then make changes that you believe will fix the problem. You could also use Google to research an issue that you are not certain about. I believe that a bit of practice with CodeWar challenges might sharpen this skill as well.

I CAN MODEL ANYTHING – Steady

Feedback: You modelled your solution in a single method which I felt was a nice and simple implementation and provided a good place to start since this exercise did not require state in order in the solution. This also left your program open to adding methods later on if required to do so perhaps during refactoring in order to adhere to the single-responsibility principle.

You also stuck to Ruby naming conventions, naming your methods in snake_case and your class in UpperCamelCase.

The algorithm that you were developing was definitely making logical sense and you were quite far into creating a solution that took care of all of the core

requirements. I believe that some method extraction could have been completed at this stage though as your main method did have more than one responsibility i.e. splitting the grades, counting the grades and returning the result.

I CAN REFACTOR ANYTHING –Improving

Feedback: You left your refactoring for a point where you had developed code that took care of the requirements that you were aware of at the time. This meant that you had complex refactoring to do which was introducing complex bugs as well each time you tried to generalise code or extract methods which became quite a task to complete and eventually led you to abandon the refactoring of the code. This was a symptom of not sticking to the RGR cycle more closely which would have done refactoring more incrementally leading to code that is cleanly developed on each RGR cycle.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Improving

Feedback: You have prioritised core cases over the edge cases which meant that you were able to provide immediate value to the client as you developed the logic for each core case.

You did not follow a strict RGR cycle. You introduced some tests that were already going to pass rather than tests that required a transformation to the implementation logic in order to make them pass, thus fulfilling the requirement of what a red phase is used for in TDD.

You also skipped refactoring phases which meant that you ran into the complex code that was hard to refactor instead of refactoring on each refactor cycle in order to iteratively clean up code during the process.2

While you are still refining your RGR process, perhaps you could follow 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

You can think of each new test and refactor phases as the introduction of simple transformations to your algorithm. You may find the following 2 blog posts quite interesting concerning the process and transformations to the algorithm
https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html
http://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html

I USE AN AGILE DEVELOPMENT PROCESS – Steady

Feedback: You reified the main requirements quite well and were able to develop an input-output table that represented the core requirements of the program. Your information gathering process could be a bit refined, as it appeared that you got a skeleton idea of how the program should work and ran with that. I would suggest investigating variations of the input that might be edge cases as strings are highly mutable, so it is important to determine how to handle possible variations such as strings with no spaces, empty strings etc. This will help you to be more informed on what development decisions to make as you will have a broader understanding of the program as a whole.

I WRITE CODE THAT IS EASY TO CHANGE – Improving

Feedback: I was pleased that 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 as test suites will not break due to changes in implementation logic or refactoring of the code.

Ideally, you should commit to Git when all your current tests pass - at the end of the green and refactor step of your red-green-refactor cycle. This ensures that the latest working version of your code is always available which helps with being able to change your code because if the program runs into problems, you can roll it back to a previous working version. This also means that your commit messages document the context of the changes made keeping the client updated with the progress of the program.

You chose a sensible name for the method which was informed by the client's domain making it easy to understand what it does. The parameter and subsequent variable for your input string were based on the type i.e. "string". This is not indicative of the data that is stored in this variable which is not informative to somebody who wants to make changes to the code. I suggest using a more descriptive name eg. grade_input.

I CAN JUSTIFY THE WAY I WORK – Steady

Feedback: You were communicating to me the fact that you completed a specific feature of the program every time you were able to complete said features which is a great way to keep a client updated to your progress. You also kept me updated on your thinking process and what you were attempting to do which is a great habit for technical interviews.

When deviating from the process, such as when you skipped refactoring phases, it is very important to be able to communicate and justify why you are doing so, as there are cases where one could deviate if there is a good enough reason to do so. This will help to keep you accountable for the process as well as you would need to have a sound reason to deviate from the process.