REVIEW FEEDBACK

Iain Aitken 15/03

15 March 2021 / 04:00 PM / Reviewer: Pierre Roodman

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

GENERAL FEEDBACK

Feedback: Well done with completing your first review session. You have shown really good problem-solving skills and a good familiarity with Ruby. I would just encourage you to consider using a simpler test progression in order to drive the growth of your algorithm and focus a bit less on the implementation details from the beginning and how you assume that you would like the algorithm to work. You could also just try and gather more information about the program from the client in order to have a better overall understanding of how the program might behave.

I CAN TDD ANYTHING – Improving

Feedback: You were attempting to use the behaviours for your testing which is a great way to have your tests and code decoupled from each other and have your tests client-oriented, but your first test case was a very complex test case. I suggest taking a more iterative approach to your test progression starting from the simplest case that is a valid behaviour as described by the client, hardcoding the return and then breaking the assumption of the previous test by bringing in a new test with a different test case on the same level of complexity. This will bring in a simple if-else logic which you would then refactor if there is any duplication that can be generalised. A possible route for this particular exercise would be a single green grade>> then a single amber grade >> then a single red grade >> then a string of multiple grades (you will already have the logic for differentiating grades and this will introduce the split and count logic) >> then a test that has different capitalisation perhaps. Each test is to be passed in the simplest possible manner and the refactor phase can be used to clean up and generalise the

duplication that you identify. Thereby growing your algorithm naturally without having to think too much about how the implementation logic will look.

I suggest reading the following article about the transformation priority premise:
https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html
The following video is also a good reference to see TDD in action testing for behaviours. The language that is being used is not Ruby, but the principle is quite universal.
https://www.youtube.com/watch?v=QbNhpPQkCBs
You can start watching this video from around the 34-minute mark.

I CAN PROGRAM FLUENTLY – Steady

Feedback: You comfortably used your editor and the terminal to set up, access and navigate your development environment. You are quite familiar with Ruby syntax and language constructs as well as are able to identify which methods are not available to your particular version of Ruby.

You were familiar with Ruby's control flow features, String and Array classes. Your algorithm was also quite logical.

I CAN DEBUG ANYTHING – Steady

Feedback: You were familiar with the common errors and could identify the critical information necessary from the failing tests to narrow your bug search space. When refactoring, you did spend a good portion of your resources trying to get the count methods to work. In a case like this, using IRB in order to check your critical assumptions may have helped you to discover that the array was no longer in existence because you were no longer splitting the string and that you needed to be included in the logic.

I CAN MODEL ANYTHING – Steady

Feedback: You modelled your solution in a class with a single method to start with which I felt was a nice and simple implementation to start with as you could have added methods later on when you were ready to extract methods. A class was perhaps not necessary though as this solution did not necessarily need state in order to be solved and a more functional approach would have been sufficient.

You followed naming convention best-practices and named your method containing a verb. You also stuck to Ruby naming conventions, naming your methods in snake_case and your class in UpperCamelCase. Your algorithm was also making sense and you were well on your way to completing the exercise.

I CAN REFACTOR ANYTHING –Steady

Feedback: You showed a good understanding of how to refactor code in order to make it more readable. You could on some refactor phases also consider your variable naming and if they are relevant to the client's domain. You did not get the concatenation portion of your algorithm completed, but that logic would have been a good candidate for method extraction in order for the main method to adhere to the single-responsibility principle.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Improving

Feedback: You were prioritising core cases over edge cases which provides the client with immediate value.

Your first test was introducing too much complexity at one point in time. I would suggest a simpler test progression as I have laid out in the "I can TDD anything" section of this review. Whilst you are refining your process, perhaps the following checklist may help you:

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 were doing really good research using the Ruby documentation. Research is an important tool for any developer and helps to be sure of how to use specific methods or solutions rather than using resources trying to make code work that they are not sure how to implement.

I USE AN AGILE DEVELOPMENT PROCESS – Improving

Feedback: You did a good job asking questions about the main requirements of the program and what the main behaviours would be. I would suggest that perhaps you spend a little more time considering how an input might vary especially in the case of strings because they are highly mutable. Also, take note of the examples that are provided as they tell a story as well. An example is the single green grade that I shared with you. If you inspect this, you will notice that red and amber grades are not outputted as they do not have a count, therefore are not included in the output string. You could consider common edge cases as well which may arise for the datatype of the input that you are working with.

I would like to encourage you to use an input-output table that can be used to record some examples of your own. This can be used to flesh out some simple test cases that you can use in your tests as well as provide a way of exploring and clarifying behaviours with the client.

I WRITE CODE THAT IS EASY TO CHANGE – Improving

Feedback: 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 they will not break your test suite.

I would have liked to have seen you make use of Git as part of your process. Using Git to commit your work means that it is easy to roll back to an earlier version if something goes wrong thereby making your code easier to change. The commit messages also serve to document the changes made in terms of features completed. This will help a client to be able to keep track of the progress of the development of the program. You should commit on every green phase and refactor phase of the RGR cycle.

I would suggest that the method name “parse” perhaps be a bit more related to the client’s domain. Perhaps something like “generate_report. This makes code easier to read and code that is easier to read is easier to change.

I CAN JUSTIFY THE WAY I WORK – Steady

Feedback: You are very vocal about what you are doing and why you are doing so at a level that made sense and was quite clear. Perhaps a potential refinement to your communication could be giving the client/interviewer updates on the high-level progress of the program. You could confirm when a certain feature has been completed and then make it clear which feature you will be working on next.