REVIEW FEEDBACK

Lou Reade 17/05

17 May 2021 / 10:00 AM / Reviewer: Ronald Munodawafa

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

GENERAL FEEDBACK

Feedback: Hi Yellow Beedrill 7,

You have improved incredibly since your last review. You incorporated all the feedback that Pierre shared with you. Well done for that!

There are still some areas you need to pay attention to. I encourage you to spend time developing a systematic debugging process. But over and above, you're doing well.

Due to a misperception, I erroneously informed you that you had introduced multiple tests in a single red phase. However, upon reviewing the video, I found this not to be true. I apologise for the false claim.

Having said that, I encourage you to continue practising and incorporating the feedback from these reviews. It will help reinforce the skills you are developing.

Kind regards,
Ronald

I CAN TDD ANYTHING – Steady

Feedback: You derived your tests from your input-output table, which oriented them towards the behaviour you and the client agreed upon.

Your overall test-driven development was fine. However, you may need to consider varying your tests. You used subarrays of [10,20,30] for all your tests. Varying your tests on input cases where you are not making assumptions prevents biasing your implementation towards an incorrect implementation. That said, your tests incrementally led to developments in your code.

You also might need to practise testing for exceptions to gain better familiarity with the matchers available to you. Overall, you were aware of the testing semantics.

I CAN PROGRAM FLUENTLY – Strong

Feedback: You used the terminal to set up your project folder. You can speed your setup by using RSpec's init option, which you can learn about here https://relishapp.com/rspec/rspec-core/docs/command-line/init-option. You were comfortable with the terminal as part of your development workflow with Git and the Unix utilities.

You were familiar with Ruby's syntax and constructs, such as classes and methods. You were familiar with array operations such as pushing and indexing. You were able to utilise the higher-order functions defined as methods on the Array class quite well.

I CAN DEBUG ANYTHING – Improving

Feedback: You encountered a bug while testing the passing of the second test. The bug involved a subtle typo. You rushed over the backtrace message, which is a common developer mistake. I encourage you to read the backtrace messages carefully and slowly. That said, you used print statements to gain visibility into your program's runtime state, which contributed to the bug's resolution.

You encountered another bug: with your test for the exception. Due to unfamiliarity with the testing syntax, you quickly assumed that the bug was with your implementation and started unjustified changes. I recommend considering tests as code that can also be buggy. You can use print statements

as you did before in a feedback loop to determine your hypotheses' validity. Only when you've verified your assumptions are you better informed to make changes. That said, you quickly recognised that the braces were missing and could proceed with the client's requirements.

Please remember to keep IRB as part of your debugging toolkit, given that it helps you form a feedback loop in which you can make changes to verify the overall behaviour of program entities whose behaviour you are uncertain about.

I CAN MODEL ANYTHING – Steady

Feedback: You constructed your solution as a class with a stateless public method. You named your class 'BandPassFilter' and your method 'filter', which adhered to naming best practices and Ruby's naming conventions. You could have named the method 'apply' or given it a different name to reduce the redundancy, although it's a verb in this instance, which helps with comprehension of your interface.

You used mapping for your algorithm, which was a natural fit for the problem. As a result, your algorithm was elegantly composed.

I CAN REFACTOR ANYTHING –Strong

Feedback: You introduced contexts after you were done testing with single-frequency arrays. The contexts were grouped your tests by functionality, which helped improve the maintainability of your test suite.

You applied the single-responsibility principle in two instances - validating the input soundwave and updating the soundwave. The extraction of these into private methods adhered to the interface segregation principle and helped your public method read declaratively.

Your refactorings were regular, and you performed them in the refactor phases, which helped keep your code clean.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Strong

Feedback: You ran your tests after every change, utilising them as a measure of progress. Your tests always justified your code, resulting in a strict red-green relationship. You also included the refactor phases, thus following a brilliant red-green-refactor cycle.

Your test progression was sensible, and you dealt with the core cases first, providing the client with immediate value and logical development direction. Your overall development workflow was methodical.

I USE AN AGILE DEVELOPMENT PROCESS – Strong

Feedback: You took notes as the client spoke. You asked for input. You asked for the requirements surrounding the representation of frequencies and soundwaves. You captured most of the main requirements.

You used your input-output table to determine the acceptance criteria and plan for your tests. You managed to cover finer details and edge cases such as empty arrays. Your consideration of the input's variation resulted in a comprehensive specification of the requirements.

I WRITE CODE THAT IS EASY TO CHANGE – Steady

Feedback: You committed after every green phase, but for your refactorings, you waited until you had multiple of them. You also committed your test groupings along with the code for the following green phase. A good rule of thumb is to commit every new working version of your repository. That means committing after every green phase and again after every single refactoring. It helps with more granular reversions.

Your commit messages were mostly helpful. However, you could improve on them by only stating the implementation work you've done for the commit. You do not need to specify any information about the tests unless all the work you've

done for the commit involves tests, e.g. when you grouped your tests. That said, the capitalisation of your commits improved their readability.

Your names were part of the domain, which clarified the intent behind your code. Your tests described the expected behaviour, providing a meaningful specification of the system under test.

A refactoring that you may have had to think twice about was the change from if-else expression to the ternary conditional expression, given that it did not improve readability. Please remember that there are instances that if-else expressions are clearer. A good measure to keep in mind is your code's changeability without comments.

The refactoring you engaged in was enabled by the flexibility brought on by the decoupled relationship between your tests and code. You achieved this decoupling by not removing tests that passed without a green phase and writing end-to-end tests. In the end, your code was changeable.

I CAN JUSTIFY THE WAY I WORK – Strong

Feedback: You communicated with the client very well. You used asterisks in your input-output table to indicate that your implementation passed a given acceptance criterion. This was a brilliant way to keep track of your progress and signal it to the client.

Your overall justifications were sound, and your workflow was logical.