REVIEW FEEDBACK

Cameron Logie 01/12

01 December 2020 / 11:00 AM / Reviewer: Ronald Munodawafa

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

GENERAL FEEDBACK

Feedback: You now have a very solid development process with major improvements across the board in all the areas. I appreciate your effort in ensuring that your requirements-gathering process was well-executed and resolving the structure-first approach you had in your previous review. Congratulations on the progress! You completed the exercise successfully with a very steady process! I understand that you deliberately excluded Git usage to focus on a particular subset of your process. This is valid! I believe the next step for you is to take your current development process and include the use of source control management. With this in your armour, you have a well-rounded development process. I recommend that you continue working on your process. You could see how it fits in your projects. It was a pleasure reviewing you and I wish you the best of luck in your future endeavours!

I CAN TDD ANYTHING – Steady

Feedback: You followed a steady red-green-refactor cycle running your tests before making them pass. The tests themselves were based on the acceptance criteria tying each of your development iterations to meeting the client's needs directly.

You followed an outside-in approach to your test progression allowing you to incrementally meet the client's requirements. This was reflected by the decision to test for different aspects of the system's behaviour at once before one particular feature such as the low-pass filter being complete.

I CAN PROGRAM FLUENTLY – Steady

Feedback: You were comfortable with the use of the Unix utilities to manage your development environment. You were also comfortable using Ruby to compose your tests. With the aid of documentation, you could use the built-in Array class methods. You could use control flow features such as if expressions to support the implementation of your program logic. Your overall implementation flowed logically.

I CAN DEBUG ANYTHING – Steady

Feedback: You read the backtrace messages from your failed tests. You could interpret them to determine what the minimal implementation required to pass the tests would be.

You used put statements to view the runtime state of your last test's variables. This helped you resolve the bug with testing for timing.

You made of effective use of your debugging toolbox.

I CAN MODEL ANYTHING – Steady

Feedback: You designed your system interface to be a class named 'Filter' with methods named 'soundwave' and 'convert'. The names followed the Ruby naming convention of classes having PascalCased noun-like names and methods having snake_cased verb-like names or being named after their return values if they have no side effects.

The use of array mapping was a natural fit for the problem domain resulting in an elegant algorithm.

I CAN REFACTOR ANYTHING –Steady

Feedback: You applied the single-responsibility principle in the refactor phase by introducing private methods ‘nil_frequency’ and ‘incompatible_frequency’ that would help you keep the implementation responsibilities for edge cases out of your ‘convert’ method. This was a brilliant way to clean up your code. I encourage you to start refactoring as early as you can so that your codebase evolves with clean code iteration to iteration.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Steady

Feedback: You had an iterative red-green-refactor cycle that resulted in your overall development process being incremental. Your tests were directed at testing for simpler behaviours such as unmodified soundwaves before taking on the cases involving modification. This not only helped make your development process incremental but it was a logical way of ordering your tests. As for prioritisation, you provided the client with immediate value by focusing on core cases such first before dealing with edge cases such as bad input.

I USE AN AGILE DEVELOPMENT PROCESS – Strong

Feedback: You reified the requirements as they were given to you by the client. You then went through them to double-check with the client on your understanding of them.

You took note of the acceptance criteria deriving them from the information the client had given you. Using the case listing was a great way to specify the behaviour of the system in a way you could use for your tests.

Your overall requirements-gathering process was comprehensive and exhaustive.

I WRITE CODE THAT IS EASY TO CHANGE – Improving

Feedback: You only tested at the system boundaries keeping your tests and implementation decoupled. This made it possible to refactor your implementation or tests without needing to change both at the same time.

Your tests' names were based on the acceptance criteria they addressed. Yoru program entities had names such as 'convert' and 'soundwaves' which were derived from the ubiquitous language. Your naming practice helped make the usage and intention of your code easily discoverable.

You did not commit every working version of your solution. Committing every working version helps ensure that there is a solid reference for your development history that could be used as a basis for future change. You could improve in this regard by keeping a checklist so you do not forget to commit.

I CAN JUSTIFY THE WAY I WORK – Improving

Feedback: You were somewhat quiet during the session with the client. I encourage you to provide the client with justifications for the actions you are taking. To help keep them in the loop, you should consider consistently updating them on the progress of work. Your client's investment in your working process is an investment in its outcome.