REVIEW FEEDBACK

Cameron Logie 13/11

13 November 2020 / 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: Well done on completing your first review. You demonstrated that you have a very methodical approach to solving your problems. Your justification for your process overall is sound. I would encourage you to work on following a test-first approach and refer to your requirements to yield the benefits of following an agile development methodology. It was an absolute pleasure to review you. I recommend booking another review so you can get feedback on your process having considered the feedback I've shared with you.

I CAN TDD ANYTHING – Improving

Feedback: You started designing the system before you went into your test-driven development process. You determined the data structures you would use already. However, these design decisions are meant to be driven by your tests. The structure of your solution is also supposed to be defined by your tests. Having your implementation worked out first is leading the tests rather than the other way around.

Your tests, however, needed to be directly concerned with the requirements that had been specified only checking the inputs and outputs rather than testing the hash, an implementation detail removed the clients' requirements. This would have helped provide immediate value to the client.

That said, you have a good understanding of how to compose tests. You just need to reverse the phase order of your cycles to ensure that your process is test-driven.

I CAN PROGRAM FLUENTLY – Steady

Feedback: You were very comfortable with using Ruby, Git and RSpec. You were able to define a class and methods to meet the client’s requirements. You were comfortable with string and hash manipulation. As a result, you were getting close to meeting the client’s requirements.

I CAN DEBUG ANYTHING – Steady

Feedback: You were able to use your editor’s syntax colouring to identify incorrect syntax and resolve the typo bug you encountered with your first test. You read the backtrace messages and could understand what information they provided and relate it to your code to resolve the other typo bugs you encountered.

I CAN MODEL ANYTHING – Steady

Feedback: You defined your interface as a class named ‘Report’ with two methods ‘parse_raw_grade_data’ and ‘generate_report’. The class had a PascalCased noun-like name; the methods had snake_cased verb-like names. This followed Ruby’s and object-oriented programming’s naming conventions.

The use of a hash gave you the basis for an extensible algorithm for the problem.

I CAN REFACTOR ANYTHING –Nothing here

Feedback: You did not have an opportunity to refactor.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Steady

Feedback: You followed a consistent order between your implementation and writing of your tests. This allowed for a systematic approach to your development. You could yield the benefits of test-driven development in controlling the complexity of the interfaces we develop by following a red-green-refactor cycle. Having derived your tests from the acceptance criteria, you would have had a better measure for your development process. Following an implementation-first approach risks biasing your tests away from the client’s requirements. That said, you had a methodical approach to your problem-solving as indicated good prioritisation of the general case of correct grades before the special cases of uncounted grades and empty input.

I USE AN AGILE DEVELOPMENT PROCESS – Steady

Feedback: You asked about the structure of the input and output and were able to uncover finer details relating to capitalisation and edge cases concerning uncounted grades. This was an excellent information gathering process with plenty of questions that allowed you to have a comprehensive understanding of the client’s requirements.

I WRITE CODE THAT IS EASY TO CHANGE – Improving

Feedback: You did not commit working versions of your code. Committing working versions of your code helps ensure that you have a reliable reference for any changes that may be necessary.

Your first test was concerned with whether the hash had the correct results when the data type specified by the client’s requirements was a string. Constructing an input-output table in your information-gathering and basing your tests on it would allow you to ensure that you only test at the specified system boundaries. If you were to try and refactor the hash to a completely different data structure, your tests would break and thus make your solution brittle in the face of change. Making sure you only test the inputs and outputs opens up room for refactoring.

Having said that, the names you chose for your program entities were derived from the domain’s vocabulary. Names such as ‘generate_report’ instantly communicate what the intention behind the code is beyond the machine-interpretable level. This made it easy to reason about your code and made it more likely to be changeable.

I CAN JUSTIFY THE WAY I WORK – Strong

Feedback: In your interaction with the client, you shared the strategy you were going to employ to develop the product. You also made justifications for the technical decisions you made. These included connecting to GitHub to protect the client’s work in case your computer crashed. You also gave the client regular updates about the progress of work. Your justification of your work process was sound!