# Transcription of processed audio:

Hello, today I'm going to be talking about test driven development. It is a methodology of testing. We'll be comparing it to our traditional water flow testing, as that's what we mostly do here at the degree. So what is it? It's more than a test approach. Test driven development is more about design than anything else. Of course, the testing is so important. It encourages incremental and iterative design. By writing our test first. We can then use it to develop our test. Us developers are forced to think about the interfaces, behavior and responsibilities of our code. It promotes separation of concerns. So as we write tests for specific behaviors, we tend to create smaller, focused, more defined tests. It aligns with solid, which is kind of a big thing. It naturally leads to design where each component has a single responsibility. And refactoring is a very important step in the cycle. After writing a test and making it pass, the developer will then go back, refactor, clean up the code, run the test again and repeat that cycle until they're happy with it. It identifies issues early. So this feedback loop is extremely helpful. If the test is hard to write or the code becomes difficult to change, we know we've done something wrong. And it indicates that our design needs to be addressed early. So it's a very important step in our design. And it is a very important step in our development. So it's very important to keep track of what's going on in the code that we're writing, and it's a big key. And really important to keep track of how we're going to implement it. The third step is the ability to adapt. So we've got a lot of tests. If the test is hard to write or the curve becomes difficult to change, we know we've done something wrong. And it indicates that our design needs to be improved. And of course, the immediate feedback encourages better design from the start. It also serves as a living documentation for our code itself. So as we continue and it grows larger and larger, we refer to the test as to what's actually happening. And it describes our system. It tells us what the expected behaviors are for each component and the code itself. And it is easier to understand. As everything is designed individually, everything is more self-explanatory as a whole. So we focus on interface and implementation, design over anything else. Writing our test first forces us to think in the smallest way possible. And it dictates how our system should be interacted with. This could lead to better and more thoughtful API design. Because a test will guide us to our end goal. And there's a small quote here from Robert C. Martin. He was the founder of Clean Coders and author of multiple books and is known for his advocacy of solid and object-oriented programming. So how does the cycle work? First we write the failing test based on the requirements. This test is actually a positive rather than a negative. As it says it's failed, we know we have an end goal in mind. We then write the minimum amount of code to pass that test. This is the green set. And finally, we refactor, optimize, and clean the code ensuring all of our tests have passed. So there are two approaches to test driven testing. The first is the test driven test. And the second is the test driven test. For development, we've got inside out and outside in. They have different names as well, but I cannot remember them. American or States or something. So for inside out, we test at the smallest unit level. So all of our functions and things like that. This is far easier to learn. And it's considered what beginners should start with. It minimizes mocking and stubbing and helps prevent over-engineering. And design happens at our refactor stage. For outside in, this test at the outermost level. So what the user is seeing. It's harder to learn, but ensures that code aligns with the overall business needs. It has a heavy reliance on external dependencies. So a lot of mocking and stubbing will be required. And the design happens at the red stage. So at beta stage. Which one's better? Neither. We'll use them when we need to. Outside in is usually better with complex applications. They have a lot of rapid changing dependencies. Smaller monolithic applications are often better suited for the inside out approach. So outside works a lot better with front end using things like cypress and end-to-end testing. What are the benefits? We catch issues before they become a problem. Test serves as documentation of what the code should do. It encourages small, modular and easily testable code. And we can ensure that as we write more and more, our existing functionality doesn't break. And of course, it allows the stakeholders to better understand our project. So, comparing that to the waterfall. For debugging. Traditional often occurs after a large amount of testing. After a large amount of code has been written. Which can cause a lot of errors. For test driven, it's more straightforward since tests are written first. We know it's going to break. We know why it's breaking. Code quality. It can vary. Even if there are tests, or none at all. Traditional kind of gets in the way sometimes. And test driven, it enforces a higher standard of code quality. Leads to cleaner, more maintained code. Speed. Traditional often seen as quicker in the start. You're straight into it. Things are getting done. And test driven is very slow at the start. Because you are starting with your tests. So how does this all work? We write a test for a function that does not yet exist. So I am using Jest for this. So our describe is basically a test suite. With a specific function. We then use our it or test. To basically say this test is going to run this. And we're going to check is it a palindrome. So we run the test and it fails. That's good. So we write enough code to pass the test. I wrote the most ridiculous thing I could. Just to show it. We run the test and it passes. Now we refactor. So here I am replacing things. I am lower casing. Splitting, reversing, joining. A lot cleaner and easier to read. We test again. This time I have added another test. With spaces and capital letters. Which passes again. We refactor once more. So here is what we currently have. And I am going to take this first one. This clean string. And turn that into its own reusable function. So we make a test case for that. We fail it. And we basically write our own function for that. Run the test again. And we pass. This cycle will repeat for reversing the string. And all of that other jazz. So failing. Oops. Passing. And now we have got an entire test suite. Full of a bunch of little functions. Our code is a lot easier to read. A lot more maintainable. Everyone is happy. Everything has passed. The end. Whew. Nice. Questions? No. Solid state. Simple responsibility. Open ended something. I think. Good code. Good code. Yes. I don't remember. You have come across the term before, have you? No. That's why you are asking. We actually did learn this in Studio One. Okay. It's a bit of a long time. Yeah. Yeah. Yes. Throughout the degree, where would you have seen Inside Out versus Outside In used? That's a good one. I guess in Studio is really the only option. So I have used test-driven development for the first half of my advanced app project. Using Firebase and I've used it for the first half of my advanced project. Using Firebase and all these things, these tests really helped streamline everything. But halfway through, I got really sick of it. Things needed to get done. But. So that was probably Inside Out, you think, the way you were using it? Yeah. Because that's mainly done by developers. Yes. Inside Out. And the Outside In, who's doing that thing? Is it developers still or more? So that's still testing. That's the development. So that's still testing. That's the development. Yeah. So that's the development. Yeah. So that's the development. Yeah. So that's the development. Yeah. So that's the development. Yeah. So that's the development. Yeah. So it's the front-end, Cypress, end-to-end. I mean, I'm assuming a developer would still use that as well. So the same team, probably. Yeah. This promotes a lot more collaboration with everyone around. Yeah. Yes? What would happen if the test fields in this first part do the averages with the frame? Yeah. Yeah. It would be a lot more efficient. Yeah. So I'm going to say that's the current state of the problem. Yeah. These are the test fields that are being tested. So for the test fields, we're not really going to be using them for a lot of the test fields. They're just going to be the test fields. Yeah. Does it pass, fills when it's supposed to pass? Do you then just go back there, whatever's happening, or do you then do the full circle again? You probably just go back to what you've done, start back from where it was passing. Is that what you mean? Or what if it never passes? Like you make it, it fills, and you write the code, and it still fills? You need to seriously look at what you're doing. Or you're just overcomplicating something, I would say. Mocking and all that is like, when you call an API, this is how I see it anyway, and you get your data, mocking would just be like having an object or something that you're calling, rather than calling the API itself. And when something isn't built yet, you're just sort of faking it. Which also then allows you to easily add your API in, because you know exactly how it's going to work. Okay, looks good. Thank you very much. Well done. Thank you.