# Transcription of original 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, us developers are forced to think about the interfaces, behaviour and responsibilities of our code. It promotes separation of concerns. So, as we write tests for specific behaviours, we tend to create smaller, focused, more defined tests. It aligns with solid, which is kind of a big thing, but naturally leads to design, where each component has a single responsibility. And refactoring is a very important step in this cycle. After writing a test and making it pass, a 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 a 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 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 tests as to what's actually happening. And it describes our system. Tells us what the expected behaviours 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 this cycle work? First, we write the failing test based on the requirements. This test is actually a pilot test. This is a test that we do. We do 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, optimise and clean the code, ensuring all of our tests have passed. So, there are two approaches to test-driven development. We've got inside-out and outside-in. They have different names as well, but I cannot remember them. They're American 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 minimises 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, our fail stage. Which one's better? Neither. We use them when we need to. Outside-in is usually better with complex applications. And... 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. Tests serve 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 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, and we know why it's breaking. Code quality. Quality 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 maintainable 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 for this 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'm replacing things. I'm lower casing, splitting, reversing, joining. A lot cleaner. A lot cleaner and easier to read. We test again. This time I've added another test with spaces and capital letters, which passes again. We refactor once more. So here is what we currently have. And I'm 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're passed. This cycle will repeat for reversing the string and all of that other jazz. So failing. Oops. Passing. And now we've 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's happy. Everything has passed. The end. Whew. Thanks. Questions? No. What does SOLID say? Or is it just a fresh question? Single responsibility. Open ended something, I think. Good code. Good code. Yes. I don't remember all of them. You've come across the term before, have you? No. Oh, that's why you asked me. Well, we actually did learn this in Studio One. OK. Well, I've . Yes. Throughout the degree, where would you see inside out versus outside in used? That's a good one, isn't it? I guess 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 all these things, these tests really helped streamline everything. But halfway through, I got this. And things needed to get done. But. So it was probably inside out, you think, the way you were using it? Yeah. Because that's mainly done by developers. Yes. It's actually inside out. And the outside in, who's doing that thing? Is it developers still, or more testing? So that's still testing. That's the front end, Cypress, end-to-end. I mean, I'm assuming a developer would still use that as well. OK. Yeah. So, yeah. So the same team, probably? Yeah. Probably. This promotes a lot more collaboration with everyone around. Yeah. Yes. What would happen if the test fails when it's post-pass? Would you then just refactor what's ever happening? Or would you then do the full circle again? You'd probably just go back to what you've done. Like, start back from where it was passing. Is that what you mean? And then just- But what if it never passes? Like, you make it, it fails, then you write the code, and it still fails? Then you need to seriously look at what you're doing. Or you're just overcomplicating something, I would say. Yes. What is mocking and stuff? Mocking and all that is like, when you call an API, this is how you call an API. This is how you call an API. That's how you call an API. So, you're just overcomplicating something, I would say. Yeah. Yeah. Well, I mean, I think it's really, really important that you're not just overcomplicating something, and you're just overcomplicating something, and then you're overcomplicating something. So, you're just overcomplicating something, I would say. Yes. What is mocking and stuff? So, 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. Yeah, when something isn't built yet, you're just sort of taking it. Yeah. So, that's really the first thing. Yeah. Then it allows you to easily add your API in, because you know exactly how it's going to work. A couple of seconds to ask a question? Okay, looks good. Thank you very much, well done. Yeah, so, I think that's it. I think that's all the time for the presentation. Yeah. Thank you. Yeah. I think that's it. Yeah. Thank you. No problem. Thank you. Thank you. Thank you. Yeah, so, I think that's all the time we've got for today. Thank you. Thank you. Okay, so, I think we've got time for one more question. So, I think this is the final question. Thank you.