# 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 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 refined. Just like we've done. So it's not just a matter of time. 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 behaviors are for each component and the code itself. And it is easier to understand as everything is designed individually everything is more 초explanatory 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 and more user-friendly software. And it dictates how our system should be designed because the 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 development. We've got inside out and outside in. They have different names as well but I can't 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 fail stage. Which one's better? Neither. We 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. One of 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. 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. 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 test 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'm replacing things. I'm lower casing. Splitting. Reversing. Joining. A lot cleaner and easier to read. We run the 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. Whoops. 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. And we're done. Everyone's happy. Everything has passed. The end. Whew. Nice. Questions? No. Solid state. Single responsibility. Open ended. Something. I think. Good code. Good code. Oh, god. Good code. Yes. I don't remember. You've come across the term before, have you? No. Still hasn't. Oh, that's why you're asking. We actually did learn this in Studio One. Okay. Yes. Throughout the degree, where would you have seen inside out versus outside being used? Oh, yeah. I don't remember. I'm not sure. I don't remember. I don't remember. I don't remember. I don't remember. I remember. I don't remember. I don't remember. I don't remember. I don't remember. I don't remember. 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 really sick of it. Things needed to get done. So that was probably inside out, you think? The way you were using it? Yeah. Because that's mainly done by developers. And the outside in? Who's doing that thing? Is it developers still? That's still testing. That's the front end, Cypress, end to end. I'm assuming a developer would still use that as well. So the same team probably. This promotes a lot more collaboration with everyone around. What would happen if the test fails when it's supposed to pass? Do you then just go back there and watch what's happening? Or do you then do the full circle again? You probably just go back to what you've done start back there and go back from where it was passing. Is that what you mean? What if it never passes? Like you make it, it fails, and you grab 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? 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. And when something isn't built yet you're just sort of thinking. Which also then allows you to easily add your API in because you know exactly how it's going to work. A couple of seconds. Okay. Looks good. Thank you very much. Well done.