Audio Transcription

All right, 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 approach, 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, but naturally leads to a 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 is difficult to change, we know we've done something wrong, and it indicates that our design needs to be improved. 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, 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 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 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, it 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 state. 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. What are the benefits? We catch issues before they become a problem. Test service 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, 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. In 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 gist 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 it's 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 lowercasing, splitting, reversing, joining, a lot cleaner and easier to read. We test again, but 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. we've 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 TIS 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. Nice. Questions? No. What does solid state? A single responsibility. Open ended something, I think. Good code, good code. Yes. I don't remember. We actually did learn this in Studio One. Yes. 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 rid of some of it. And things needed to get done. So that was probably inside it, you think? The way you were using it? Yeah. Because that's mainly done by developers those were the musoe data and the outside in who's using that then? Is it developer still? It's still testing it's the front end, Cypress, IntEnd I'm assuming a developer would still use that as well So the same team? This promotes a lot more collaboration with everyone around What would happen if the test fails when it's supposed to pass? Do you just refactor what's happening or do you do the full circle again? You probably just go back to what you've done like start back from where it was passing is that what you mean like and then you need to seriously look at what you're doing or you're just you're just over complicating something I would say What is mocking? 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 faking it? Which also then allows you to easily add your API end because you know exactly how it's going to work Okay, that's it. Thank you very much ■■■■■■