Hello everybody, welcome back once again to my final installment of the SOLID principles. This week we are covering the Dependency Inversion Principle.

Let's start with some background. The dependency inversion principle was first introduced by Robert C. Martin, also known as Uncle Bob. Though he had implemented the concept earlier on, it was introduced over a series of publications in the 90s that gave it the official name. In this principle, there are two main rules to be followed. The first is ”High-level modules should not depend on low-level modules. Both should depend on abstractions.” And the second is “Abstractions should not depend upon details. Details should depend upon abstractions.” Without context, those two rules are hard to follow so this is best understood through some examples.

First is Rule A. In this example let's start with a typical top-down design architecture, where high-level modules depend on low-level modules. First we have the car module, which serves the purpose of being a car with all of its intricacies. To break down those intricacies, we give the car some lower-level modules being the brake, engine, and radio subsystems. They serve the purpose of handling their respective sub-tasks for the larger system. Below them are the individual low-level modules that serve singular purposes for their upper-level modules. Now imagine the very likely scenario that I want to tweak some code on one of those lower-level modules. Maybe I don't like how the slow-down function is calculated within my brake subsystem. I made those changes, but the recompilation of my low-level module has now forced the Brake module to recompile due to its dependency. And to make matters worse, the highest-level module must now recompile because the brake module was recompiled. Though this example isn't the largest architecture, if this was a huge codebase with tens of thousands of lines of code, these trickle-up recompilations can get incredibly expensive and time consuming for what should be simple small changes. So what is the solution to this predicament? It is to follow the idea that high-level modules don't depend on low-level modules, but rather that both depend on the abstractions between them! With this new approach, I've inserted abstract interfaces between each higher-level module and their lower-level counterparts. By doing so, I'm able to give Brake no information on its low-level modules but rather an interface that describes the behavior of those low-level modules. Now when I make that same slow-down functionality change, the recompilation only affects the specific module I was in because nothing depends on it! The compilation dependency has been completely removed while still having that needed run-time dependency. Similarly, I can make changes to my higher-level modules without affecting any other modules either. Now those small changes, stay small. There's no unnecessary trickle-up effects that can be expensive.

Now the second is rule B, which describes the relationship between abstractions and details. This rule is a zoomed in look on each module-to-interface-to-module relationship in the hierarchy. As shown in this example, whenever the Car module needs to use the slow-down function from the Brakes module, it is only referencing an assumed behavior through the interface, it doesn't actually know anything about the functions details. Then when that assumed function is called, the Brakes module adheres to the abstract behavior, and calls its specific function details to be returned back up to the Car module. By doing this, we have completely

separated the Car and Brakes modules, while still allowing a run-time dependency between them.

By combining the two concepts together, we've created a flexible and decoupled architecture, where the car module can have access to any of its low-level functionalities through the layers of abstraction, without actually knowing what they do. In this example the car module calls the stop-car abstract function which is passed down to Brakes where the details are. Within those details is the slow-down abstract function which is passed down even further to the lower-level modules to return those details. Together all the details get pulled back up to the car module without it knowing what any of those functions actually do. This abstraction holds true for any module and their lower-level functionalities.

Thank you.