Hello everyone, my name is Artem Otlyaguzov, and I'm here today to talk about the SOLID principles. These principles are important for modern application design and are something that you will encounter as programmers. I'd like to walk you through each of the principles and explain why they're important.

Let's begin by introducing our new friends: each letter in SOLID stands for one of the five principles: S for Single Responsibility, O for Open/Closed, L for Liskov Substitution, I for Interface Segregation, and D for Dependency Inversion.

First, the Single Responsibility Principle says that each class should have one and only one responsibility. This means that a class should only do one thing, and it should do it well. This helps to make the code easier to understand and maintain.

Next, the Open/Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to add new features to your code without having to change existing code.

The Liskov Substitution Principle says that if you have a base class and a derived class, then objects of the derived class should be substitutable for objects of the base class. This means that the derived class must behave in a way that is consistent with the base class, so that objects of either class can be used interchangeably.

Interface Segregation states that clients should not depend on methods they don't use. This means that interfaces should be designed in such a way that clients only need to know about the methods they use, not the ones they don't.

Finally, the Dependency Inversion Principle suggests that high-level modules should depend on low-level modules, rather than the other way around. This means that low-level code should be abstracted into higher-level components, which makes it easier to maintain and test.

Starting with the letter S and the Single Responsibility Principle (SRP), which states that each class should have one responsibility and one single purpose, it is important to understand what this means. A class should only perform one job and should have only one reason for change. This means avoiding creating classes with multiple responsibilities, as this can lead to confusion and complexity. For example, a library class should not include elements of a book class, such as pages or their content. Instead, let the book handle these aspects of its own.

Moving on to the next principle, the Open/Closed Principle (OCP). This principle states that software entities, such as classes, modules, and functions, should be open for addition of new features, but closed to changes that affect existing functionality. This means that when we create new classes and subclasses, we should do so in a way that allows for the addition of new functionality without altering the existing code.

In the illustrations, we see two cats. Imagine that both exist, one as a two-dimensional representation and the other as a more detailed version. To create these, we did not change the fundamental concept of what a cat is. We simply added more features to the existing concept.

Next, we have the Liskov Substitution Principle (LSP), which is the most complex concept to understand. However, once you delve deeper into it, it becomes quite natural and logical. Let me start with a quote from Barbara Liskov, the author of this principle: "Objects in a program should be replaceable by instances of their subclasses without affecting the correctness of the program." According to this principle, all functions of a superclass should also be available in its subclasses.

For example, let's look at three objects: an ambulance, a race car, and a helicopter. These all belong to the superclass Vehicle, but a helicopter cannot perform all the same functions as a vehicle on the ground, such as traveling on roads. Therefore, we can say that a helicopter is not truly a vehicle. Next, we have the Interface Segregation Principle. This principle states that a class should not have to implement methods that it does not need. It encourages keeping things simple and atomic, to avoid situations where a child class may not implement all the functionality of its parent class (or may have an unnecessary implementation that will never be used).

Finally, the Dependency Inversion Principle states that higher-level modules should communicate with lower-level ones in a way that allows changes in the lower modules without requiring changes in the higher modules. This helps to ensure that changes to the lower levels of the system do not affect the higher levels, and vice versa.

These were just some basic principles of design patterns and the SOLID principles. I hope this introduction has piqued your interest in learning more about these topics.

Thank you for your attention.