SOLID Principles with Unity

The SOLID principles are a set of design principles used in object-oriented programming to make software more maintainable, scalable, and reusable. They stand for Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.

Here are some examples of how to teach each of these principles using Unity

Single Responsibility Principle (SRP):

This principle states that a class should have only one responsibility. In Unity, this could mean creating separate classes for different aspects of a game object.

For example, you could have a class for handling movement, another for handling collisions, and another for handling visual effects. Each of these classes would have a single responsibility and would be easier to maintain and modify.

Open-Closed Principle (OCP):

This principle states that a class should be open for extension but closed for modification. In Unity, this could mean using inheritance and polymorphism to extend the behavior of existing classes without modifying them directly.

For example, you could create a base class for enemy AI behavior and then create specific subclasses for different types of enemies that inherit from this base class.

Liskov Substitution Principle (LSP):

This principle states that subclasses should be substitutable for their base classes without altering the correctness of the program. In Unity, this could mean ensuring that subclasses of game objects can be used interchangeably with their parent class.

For example, if you have a base class for projectiles, you should be able to use any subclass of projectile without affecting the overall behavior of the game.

Interface Segregation Principle (ISP):

This principle states that classes should not be forced to depend on methods they do not use. In Unity, this could mean creating smaller, more focused interfaces for specific aspects of a game object.

For example, you could create separate interfaces for handling movement, collisions, and visual effects, each with only the methods needed for that particular aspect.

Dependency Inversion Principle (DIP):

This principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In Unity, this could mean using interfaces to decouple higher-level game systems from lower-level implementation details.

For example, you could create an interface for handling player input and then have multiple implementations of this interface that use different input methods (e.g., keyboard, gamepad, touch screen).

Later through the Module, we will be talking about **Design Patterns**. When we use them, we will be revisiting these principles and show how these are used in design patterns.