The SOLID Principles

This article introduces the SOLID principles of OOP design.

Need some guidelines for creating well-designed classes? In this article, we will introduce some principles that provide just the solution. The *SOLID Principles* aim to create software that is easy to change, understand, and reuse.

SOLID is an acronym that stands for:

- *Single Responsibility Principle*: Classes should have only one reason to change.
- *Open-Closed Principle*: We should be able to add functionality through the creation of new classes rather than changing existing classes.
- *Liskov Substitution Principle*: We should be able to replace a subclass with its parent class without breaking our codebase.
- *Interface Segregation Principle*: Interfaces should only contain attributes and methods used by all of their subclasses.
- *Dependency Inversion Principle*: Classes should interact with interfaces rather than concrete classes.

Let's take a closer look at each.

Single Responsibility Principle

The Single Responsibility Principle states that modules (a class or a set of functions) should have only one reason to change. Reasons to change include business decisions, database changes, or adjustments to the user experience. When multiple of these factors could impact a class, that class is violating the Single Responsibility Principle. In order to align with the Single Responsibility Principle, the class would be broken up into smaller classes according to these different reasons. This reduces the complexity of our classes and the number of times each will have to change.

Let's say we are creating a tower defense game, and we need to design an `Enemy` entity. We initially come up with an object that handles the following:

- Attacking and taking damage
- Displaying on the screen
- State info such as health and type information

At one level, it seems our class follows the Single Responsibility Principle, if something about the `Enemy` changes, then this class has to change. However, each feature corresponds to completely different areas of the application. Changes to the gameplay mechanics, presentation, or database can each result in changes to this class.

The Single Responsibility principle would have us split the class into something like the following:

- `EnemyBehavior`: Handles attacking and taking damage.
- `EnemyState`: Handles the state information for the enemy.
- `EnemyStorage`: Handles its database storage and retrieval.
- `EnemyDisplay`: Handles displaying the enemy on the screen.

Overall, the Single Responsibility Principle minimizes the impact of change between files.

Open-Closed Principle

The Open-Closed Principle states that software classes should be open for extension but closed for modification. This means we should be able to add functionality to our system by adding new classes without having to change existing ones.

When we have to edit an existing class in order to incorporate new business features, our code likely violates both the Single Responsibility and Open-Closed Principles. The combination of the Single Responsibility and Open-Closed Principles has us focus on separating our classes based on independent behaviors.

Multiple Choice Question
Which would be a side effect of utilizing the Open-Closed and Single Responsibility Principles?
The same files being changed over and over again.
A system consisting of a small number of large classes.
Classes are rarely created to add new functionality.
A system consisting of a large number of small classes
👏
Correct! Applying these principles will increase the number of classes in your system, as each class will have limited responsibilities and extensions will be frequently created.

Liskov Substitution Principle

The Liskov Substitution Principle states that a subclass should be able to be substituted with its parent class. There should be no special behavior that we have to do when dealing with particular subclasses. This allows us to define our systems in terms of many interchangeable parts.

Let's say that we have a `Dog` and a `Cat` and a `Turtle` class, all of which are subclasses of `Pet`. If we are following the Liskov Substitution Principle, we should be able to replace any usage of `Dog`, `Cat`, or `Turtle` in our code with Pet and have our code still work.

This allows us to write any code in terms of `Pet`, rather than using specific subclasses. This code is more extensible because any future `Pets` that follow the Liskov Substitution Principle can be added and will work with our codebase without needing to change any existing code. This also furthers our use of the Open-Closed Principle.

Interface Segregation Principle

If we have written a lot of object-oriented code, we have probably had to implement methods that we don't actually plan on using. We sometimes need to add these methods in order to make a class conform to an *interface*, a description of the inherited properties and methods that need to be implemented by a subclass, we need to use.

When we decide to add methods to an interface that are only used by some of the subclasses, we are violating the Interface Segregation Principle. We should instead define new interfaces that properly abstract the different types of subclasses.

Dependency Inversion Principle

While the Liskov Substitution Principle has us design classes that *can* be substituted for their superclass in client code, the Dependency Inversion Principle has us actually write our client in terms of those superclasses.

The Dependency Inversion Principle states that systems are the most flexible when dependencies (classes calling other classes) refer to abstractions rather than concrete classes. This ensures that we define appropriate high-level

interfaces and write code that is generic to those interfaces, enhancing our use of the Open-Closed Principle as well as the Interface Segregation Principle.

Review

Multiple Choice Question
Which is **NOT** a summary of a SOLID principle?
A system should be built out of a small number of large classes.
Classes should have one reason to change.
A system should be built out of many interchangeable parts.
Classes should not depend on anything they don't use.

Correct! This would likely cause a violation of the Single Responsibility Principle.
In this article, we have introduced the SOLID principles for creating software that is easy to change, understand, and reuse. Rather than being independent guidelines, the SOLID principles support and reinforce each other, resulting in cohesive guidelines for well-designed systems. Recognizing violations of these principles in our codebases and fixing them is an important aspect of improving the health of our system designs.