**The SOLID Principles: Writing Scalable & Maintainable Code**

[[Ryan Lai](https://forreya.medium.com/?source=post_page-----13040ada3bca--------------------------------)](https://forreya.medium.com/?source=post_page-----13040ada3bca--------------------------------)

[Ryan Lai](https://forreya.medium.com/?source=post_page-----13040ada3bca--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F23030b811506&operation=register&redirect=https%3A%2F%2Fforreya.medium.com%2Fthe-solid-principles-writing-scalable-maintainable-code-13040ada3bca&user=Ryan+Lai&userId=23030b811506&source=post_page-23030b811506----13040ada3bca---------------------post_header-----------)

8 min read

·

Jun 28

1.5K

17

Has anyone ever told you that you write *“bad code?”*

Well if you have, there’s really nothing to be ashamed about. We all write flawed code as we learn. The good news is, it’s fairly straightforward to make improvements- but only if you’re willing.

One of the best ways to improve your code is by learning some programming design principles. You can think of programming principles as a general guide to becoming a better programmer- the raw philosophies of code, one could say. Now, there are a whole array of principles out there (one could argue there might even be an overabundance) but I will cover five essential ones which go under the acronym SOLID.

*Note: I will be using Python in my examples but these concepts are easily transferrable to other languages such as Java.*

**1. First off… ‘S’ in SOLID stands for Single Responsibility**



This principle teaches us to:

**Break our code into modules of one responsibility each**.

Let’s take a look at this Person class which performs unrelated tasks such as sending emails and calculating taxes.

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
   
 def send\_email(self, message):  
 # Code to send an email to the person  
 print(f"Sending email to {self.name}: {message}")  
  
 def calculate\_tax(self):  
 # Code to calculate tax for the person  
 tax = self.age \* 100  
 print(f"{self.name}'s tax: {tax}")

According to the **Single Responsibility Principle**, we should split the Person class up into several smaller classes to avoid violating the principle.

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
class EmailSender:  
 def send\_email(person, message):  
 # Code to send an email to the person  
 print(f"Sending email to {person.name}: {message}")  
  
class TaxCalculator:  
 def calculate\_tax(person):  
 # Code to calculate tax for the person  
 tax = person.age \* 100  
 print(f"{person.name}'s tax: {tax}")

It’s more lines- sure- but now we can more easily identify what each section of the code is trying to accomplish, test it more cleanly, and reuse parts of it elsewhere (without worrying about irrelevant methods).

**2. Next up is ‘O’… or the Open/Closed Principle**



This principle suggests that we design our modules to be able to:

**Add new functionality in the future without directly modifying our existing code**.

Once a module is in use, it’s essentially locked, and this reduces the chances of any new additions breaking your code.

This is one of the hardest of the 5 principles to fully grasp due to its contradictory nature, so let’s take a look at an example:

class Shape:  
 def \_\_init\_\_(self, shape\_type, width, height):  
 self.shape\_type = shape\_type  
 self.width = width  
 self.height = height  
  
 def calculate\_area(self):  
 if self.shape\_type == "rectangle":  
 # Calculate and return the area of a rectangle  
 elif self.shape\_type == "triangle":  
 # Calculate and return the area of a triangle

In the example above, the Shape class handles different shape types directly within its calculate\_area() method. This violates the **Open/Closed Principle** because we are modifying the existing code instead of extending it.

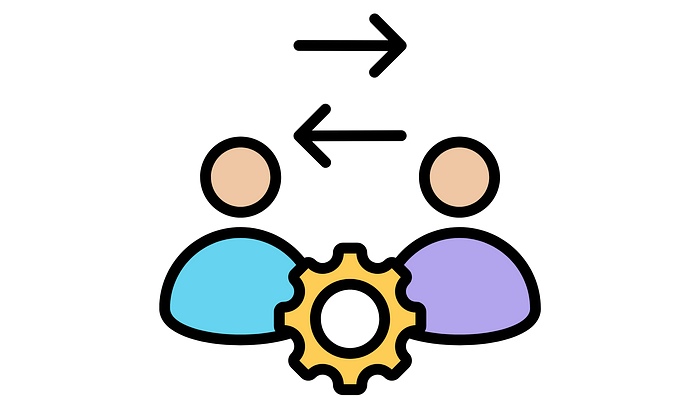
This design is problematic because as more shape types are added, the calculate\_area() method becomes more complex and harder to maintain. It violates the principle of separating responsibilities and makes the code less flexible and extensible. Let’s take a look at one way we could resolve this issue.

class Shape:  
 def \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
 def calculate\_area(self):  
 pass  
  
class Rectangle(Shape):  
 def calculate\_area(self):  
 # Implement the calculate\_area() method for Rectangle  
  
class Triangle(Shape):  
 def calculate\_area(self):  
 # Implement the calculate\_area() method for Triangle

In the example above, we define the base class Shape, whose sole purpose is to allow more specific shape classes to inherit its properties. For example, the Triangle class extends onto the calculate\_area() method to calculate and return the area of a triangle.

By following the **Open/Closed Principle**, we can add new shapes without modifying the existing Shape class. This allows us to extend the functionality of the code without the need to change its core implementation.

**3. Now for ‘L’… the Liskov Substitution Principle (LSP)**



In this principle, Liskovis basically trying to tell us the following:

**Subclasses should be able to be used interchangeably with their superclasses without breaking the functionality of the program.**

Now what does this actually mean? Let’s consider a Vehicle class with a method called start\_engine().

class Vehicle:  
 def start\_engine(self):  
 pass  
  
class Car(Vehicle):  
 def start\_engine(self):  
 # Start the car engine  
 print("Car engine started.")  
  
class Motorcycle(Vehicle):  
 def start\_engine(self):  
 # Start the motorcycle engine  
 print("Motorcycle engine started.")

According to the **Liskov Substitution Principle,** any subclass of Vehicle should also be able to start the engine without any issues.

But if, for example, we added a Bicycle class. We obviously would no longer be able to start the engine because bicycles don’t have engines. Below demonstrates the incorrect way to resolve this issue.

class Bicycle(Vehicle):  
 def ride(self):  
 # Rides the bike  
 print("Riding the bike.")  
  
 def start\_engine(self):  
 # Raises an error  
 raise NotImplementedError("Bicycle does not have an engine.")

To properly adhere to the LSP, we could take two routes. Let’s take a look at the first one.

**Solution 1:** Bicycle becomes its own class (without inheritance) to ensure that all Vehicle subclasses behave consistently with their superclass.

class Vehicle:  
 def start\_engine(self):  
 pass  
  
class Car(Vehicle):  
 def start\_engine(self):  
 # Start the car engine  
 print("Car engine started.")  
  
class Motorcycle(Vehicle):  
 def start\_engine(self):  
 # Start the motorcycle engine  
 print("Motorcycle engine started.")  
  
class Bicycle():  
 def ride(self):  
 # Rides the bike  
 print("Riding the bike.")

**Solution 2:**The superclass Vehicle is split into two, one for vehicles with engines and another one for the latter. All subclasses can then be used interchangeably with their superclass without altering the expected behavior or introducing exceptions.

class VehicleWithEngines:  
 def start\_engine(self):  
 pass  
  
class VehicleWithoutEngines:  
 def ride(self):  
 pass  
  
class Car(VehicleWithEngines):  
 def start\_engine(self):  
 # Start the car engine  
 print("Car engine started.")  
  
class Motorcycle(VehicleWithEngines):  
 def start\_engine(self):  
 # Start the motorcycle engine  
 print("Motorcycle engine started.")  
  
class Bicycle(VehicleWithoutEngines):  
 def ride(self):  
 # Rides the bike  
 print("Riding the bike.")

**4. Next in line… ‘I’ for Interface Segregation**



The general definition states that our modules shouldn’t be forced to worry about functionalities that they don’t use. That’s a bit ambiguous though. Let’s transform this obscure sentence into a more concrete set of instructions:

Client-specific interfaces are better than general-purpose ones. This means that classes should not be forced to depend on interfaces they don’t use. Instead, they should rely on smaller, more specific interfaces.

Let’s say we have an Animal interface with methods like walk(), swim(), and fly().

class Animal:  
 def walk(self):  
 pass  
  
 def swim(self):  
 pass  
  
 def fly(self):  
 pass

The thing is, not all animals can perform all these actions.

**For example:**Dogs can’t swim or fly and therefore both these methods which are inherited from the Animal interface are made redundant.

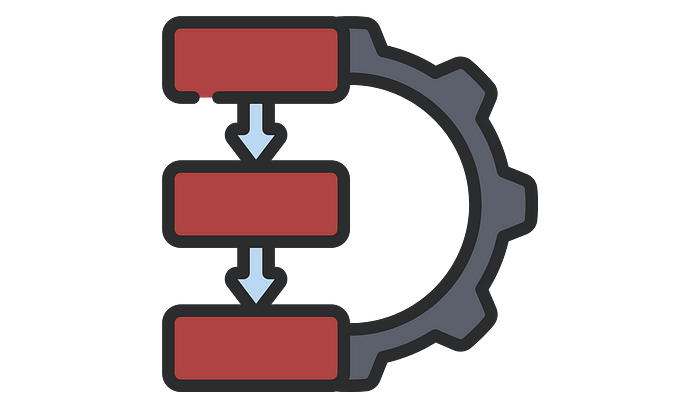
class Dog(Animal):  
 # Dogs can only walk  
 def walk(self):  
 print("Dog is walking.")  
  
class Fish(Animal):  
 # Fishes can only swim  
 def swim(self):  
 print("Fish is swimming.")  
  
class Bird(Animal):  
 # Birds cannot swim  
 def walk(self):  
 print("Bird is walking.")  
  
 def fly(self):  
 print("Bird is flying.")

We need to break our Animal interface down into smaller, more specific sub-categories, which we can then use to compose an exact set of functionality that each animal requires.

class Walkable:  
 def walk(self):  
 pass  
  
class Swimmable:  
 def swim(self):  
 pass  
  
class Flyable:  
 def fly(self):  
 pass  
  
class Dog(Walkable):  
 def walk(self):  
 print("Dog is walking.")  
  
class Fish(Swimmable):  
 def swim(self):  
 print("Fish is swimming.")  
  
class Bird(Walkable, Flyable):  
 def walk(self):  
 print("Bird is walking.")  
  
 def fly(self):  
 print("Bird is flying.")

By doing this, we achieve a design where classes only rely on the interfaces they need,**reducing unnecessary dependencies**. This becomes especially useful when testing, as it allows us to mock out only the functionality that each module requires.

**5. Which brings us to… ‘D’ for Dependency Inversion**



This one is pretty straightforward to explain, it states:

High-level modules should not rely directly on low-level modules. Instead, both should rely on abstractions (interfaces or abstract classes).

Once again, let’s look at an example. Suppose we have a ReportGenerator class that, naturally, generates reports. To perform this action, it needs to fetch data from a database first.

class SQLDatabase:  
 def fetch\_data(self):  
 # Fetch data from a SQL database  
 print("Fetching data from SQL database...")  
  
class ReportGenerator:  
 def \_\_init\_\_(self, database: SQLDatabase):  
 self.database = database  
  
 def generate\_report(self):  
 data = self.database.fetch\_data()  
 # Generate report using the fetched data  
 print("Generating report...")

In this example, the ReportGenerator class directly depends on the concrete SQLDatabase class.

This works fine for now, but what if we wanted to switch to a different database such as MongoDB? This tight coupling would make it difficult to swap out the database implementation without modifying the ReportGenerator class.

To adhere to the Dependency Inversion Principle, we would introduce an abstraction (or interface) that both the SQLDatabase and MongoDatabase classes can depend on.

class Database():  
 def fetch\_data(self):  
 pass  
  
class SQLDatabase(Database):  
 def fetch\_data(self):  
 # Fetch data from a SQL database  
 print("Fetching data from SQL database...")  
  
class MongoDatabase(Database):  
 def fetch\_data(self):  
 # Fetch data from a Mongo database  
 print("Fetching data from Mongo database...")

Note that the ReportGenerator class would now also depend on the new Database interface through its constructor.

class ReportGenerator:  
 def \_\_init\_\_(self, database: Database):  
 self.database = database  
  
 def generate\_report(self):  
 data = self.database.fetch\_data()  
 # Generate report using the fetched data  
 print("Generating report...")

The high-level module (ReportGenerator) now does not depend on low-level modules (SQLDatabase or MongoDatabase) directly. Instead, they both depend on the interface (Database).

**Dependency Inversion** means our modules won’t need to know what implementation they are getting- only that they will receive certain inputs and return certain outputs.

**Conclusion**



Nowadays I see a lot of discussion online about the SOLID design principles and whether they have withstood the test of time. In this modern world of multi-paradigm programming, cloud computing, and machine learning… **is SOLID still relevant?**

Personally, I believe that the SOLID principles will always be the basis of good code design. Sometimes the benefits of these principles may not be obvious when working with small applications, but once you start to work on larger-scale projects, the difference in code quality is way worth the effort of learning them. The modularity that SOLID promotes still makes these principles the foundation for modern software architecture, and personally, I don’t think that’s going to change anytime soon.

Thank you for reading.