A solid description for the SOLID principles

1 Introduction

The SOLID principles are a set of guidelines in object-oriented programming that encourage code readability, scalability, and maintainability. SOLID stands for:

- S: Single Responsibility Principle (SRP)
- O: Open/Closed Principle (OCP)
- L: Liskov Substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

Let's consider the example of a simple "shape" application to illustrate the SOLID principles.

2 Single Responsibility Principle (SRP)

2.1 Concept

A class should have only one reason to change.

2.2 Example

Let's say we have different shapes, and each shape should be responsible for calculating its own area.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle:
    def __init__(self, radius):
```

```
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius
```

In this example, the `Rectangle` and `Circle` classes each have a single responsibility: to calculate their respective areas.

3 Open/Closed Principle (OCP)

3.1 Concept

Software entities should be open for extension but closed for modification.

3.2 Example

To adhere to OCP, we should be able to add new shapes without modifying the existing code.

```
class Shape:
    def area(self):
        pass

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius
```

Now, we can add more shapes without modifying the existing shape classes.

4 Liskov Substitution Principle (LSP)

4.1 Concept

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

4.2 Example

We can use a derived class object wherever a base class object is expected.

```
def calculate_area(shape):
    return shape.area()

rectangle = Rectangle(2, 3)
circle = Circle(5)

print(calculate_area(rectangle))  # 6
print(calculate_area(circle))  # 78.53975
```

5 Interface Segregation Principle (ISP)

5.1 Concept

Clients should not be forced to depend on interfaces they do not use.

5.2 Example

Here, each shape adheres to the simplified `Shape` interface, which only requires an `area` method. A shape class doesn't need to implement any other methods it doesn't use.

```
class Shape:
    def area(self):
        pass
```

6 Dependency Inversion Principle (DIP)

6.1 Concept

High-level modules should not depend on low-level modules. Both should depend on abstractions.

6.2 Example

The `calculate_area` function is a high-level module. It doesn't depend on the specific shape classes (`Rectangle`, `Circle`); instead, it depends on the `Shape` abstraction.

```
def calculate_area(shape):
    return shape.area()
```

7 Conclusion

By following the SOLID principles in this manner, we create a more maintainable, extensible, and robust application. Each principle targets a specific aspect of good object-oriented design and, when combined, offers a comprehensive approach to creating clean code.