SOLID Principles

SOLID is an acronym for five design principles that help software developers write clean, maintainable, and scalable code. These principles were introduced by Robert C. Martin (Uncle Bob) and aim to make systems more flexible and easier to refactor and maintain.

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

Explanation: If a class has more than one responsibility, it becomes more difficult to maintain and change. By separating concerns, you make the system easier to understand and modify.

Bad Example:

In this bad example, a single class handles both order processing and invoice printing, violating SRP.

Code snippet:

```
class OrderProcessor:
    def process_order(self, order):
        # Order processing logic here
        pass
    def print_invoice(self, order):
        # Invoice printing logic here
        pass
```

A better design would be to separate these responsibilities into two classes: one for order processing and one for invoice printing.

Good Example:

Code snippet:

```
class OrderProcessor:
    def process_order(self, order):
        # Order processing logic here
        pass

class InvoicePrinter:
    def print_invoice(self, order):
        # Invoice printing logic here
        pass
```

2. Open/Closed Principle (OCP)

Definition: A class should be open for extension but closed for modification.

Explanation: You should be able to add new functionality to a class without altering its existing code. This can be achieved through polymorphism or abstraction.

Bad Example:

In this bad example, we modify the existing 'PaymentProcessor' class to add new payment types, violating OCP.

Code snippet:

```
class PaymentProcessor:
    def process_payment(self, payment, payment_type):
        if payment_type == 'credit_card':
            # Credit card payment logic here
            pass
        elif payment_type == 'paypal':
            # PayPal payment logic here
            pass
```

Good Example:

In this good example, we extend the payment processing functionality using inheritance without modifying existing code.

Code snippet:

```
class PaymentProcessor:
    def process_payment(self, payment):
        raise NotImplementedError("Subclasses should implement this method.")
class CreditCardPayment(PaymentProcessor):
    def process_payment(self, payment):
        # Credit card payment logic here
        pass
class PayPalPayment(PaymentProcessor):
    def process_payment(self, payment):
        # PayPal payment logic here
        pass
```

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Explanation: A subclass should extend a superclass without changing its behavior. If a subclass cannot be used in place of its superclass, then the subclass violates the Liskov Substitution Principle.

Bad Example:

In this bad example, the subclass 'Penguin' violates LSP by changing the expected behavior of 'fly' method.

Code snippet:

```
class Bird:
    def fly(self):
        return "Flying"
class Penguin(Bird):
    def fly(self):
        # Penguin can't fly, so this breaks LSP
        raise Exception("Penguin can't fly")
```

Good Example:

In this good example, subclasses extend the behavior of the base class without changing the expected behavior.

Code snippet:

```
class Bird:
    def fly(self):
        return "Flying"

class Sparrow(Bird):
    def fly(self):
        return "Sparrow flying"

class Penguin(Bird):
    def fly(self):
        return "Penguin can't fly"
```

4. Interface Segregation Principle (ISP)

Definition: A client should not be forced to implement interfaces it doesn't use.

Explanation: Instead of having a large, monolithic interface, split it into smaller, more specific interfaces. This reduces the impact of changes and makes the system easier to maintain.

Bad Example:

In this bad example, a single interface forces clients to implement methods they don't need, violating ISP.

Code snippet:

```
class Worker:
    def work(self):
        pass
    def eat(self):
        pass


class Robot(Worker):
    def work(self):
        pass
    # Robot doesn't need to eat, but is forced to implement the eat method.
```

Good Example:

In this good example, we separate the interface into smaller, more specific ones, so clients are not forced to implement methods they don't use.

Code snippet:

```
class Worker:
    def work(self):
        pass
```

```
class Eater:
    def eat(self):
        pass


class Robot(Worker):
    def work(self):
        pass


class Human(Worker, Eater):
    def work(self):
        pass
    def eat(self):
        pass
```

5. Dependency Inversion Principle (DIP)

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

Explanation: In dependency inversion, the high-level class should not directly depend on low-level classes, but both should depend on abstractions such as interfaces or abstract classes.

Bad Example:

In this bad example, high-level modules depend directly on low-level modules, violating DIP.

Code snippet:

```
class UserManager:
    def __init__(self):
        self.storage = DatabaseManager()
```

```
    def save_user(self, user):
        self.storage.save(user)
```

Good Example:

In this good example, high-level modules depend on abstractions instead of low-level modules.

Code snippet:

```
class DataStorage:
    def save(self, data):
        pass

class DatabaseManager(DataStorage):
    def save(self, data):
        # Save to database
        pass

class UserManager:
    def __init__(self, storage: DataStorage):
        self.storage = storage

    def save_user(self, user):
        self.storage.save(user)
```