THE UNIVERSITY OF ZAMBIA

SCHOOL OF NATURAL SCIENCES

DEPARTMENT OF COMPUTER SCIENCES

NAME: Lukundo nakamba

COMPUTER NUMBER:2022077385

COURSE CODE:3600

COURSE NAME: SOFTWARE ENGINEERING

LAB 5

SOLID Principles

SOLID is an acronym representing five fundamental principles of object-oriented programming and design that help create maintainable, scalable, and flexible software. These principles are a set of design principles that aim to promote cleaner, more robust and maintainable code. They are essential guidelines for writing clean code.

Types of SOLID principles

1. Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should have only one job or responsibility.When a class handles multiple responsibilities, changes to one functionality can affect others. SRP helps separate concerns and makes code easier to maintain.

Example: A User class should handle user data, while a UserRepository class handles data storage and retrieval.

```
class User:
    def _init_(self, name, email):
        self.name = name
        self.email = email

class UserRepository:
    def save_user(self, user):
        # Save user to database
        Pass
```

Benefits:

- Easier debugging (issues are isolated)
- Simplified testing
- More modular code

2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions) should be open for extension but closed for modification. You should be able to add new functionality without changing existing code, typically through inheritance or composition.

Example: A PaymentGateway class can be extended to support different payment methods without modifying its core logic.

```
from abc import ABC, abstractmethod


class PaymentGateway(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass


class PayPalPaymentGateway(PaymentGateway):
    def process_payment(self, amount):
        # Process PayPal payment
        pass


class StripePaymentGateway(PaymentGateway):
    def process_payment(self, amount):
        # Process Stripe payment
        pass
```

Benefits:

- Reduces risk of introducing bugs in existing code
- More stable code base
- Easier to add new features

3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. Subclasses should extend the behavior of their parent class without changing its fundamental contract.

Example: A Rectangle class and a Square class can both inherit from a Shape class without affecting the correctness of the program.

```
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def area(self):
        pass


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

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


class Square(Shape):
    def _init_(self, side):
        self.side = side

    def area(self):
        return self.side ** 2
```

Benefits:

- More reliable inheritance hierarchies
- Better code reuse
- Fewer surprises when using polymorphism

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they don't use. Instead of one large interface, create multiple smaller, more specific interfaces.

Example: Instead of a large, general-purpose Printable interface, create separate interfaces for Printable, Scannable, and Faxable.

```
from abc import ABC, abstractmethod


class Printable(ABC):
    @abstractmethod
    def print(self):
        pass


class Scannable(ABC):
    @abstractmethod
    def scan(self):
        pass


class MultifunctionalDevice(Printable, Scannable):
    def print(self):
        # Print document
        pass


    def scan(self):
        # Scan document
        pass
```

Benefits:

- Reduces side effects of changes
- More cohesive interfaces
- Avoids dummy implementations

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions. Depends on interfaces or abstract classes rather than concrete implementations.

Example: A NotificationService class depends on an abstraction (e.g., Notifier interface) rather than a specific implementation (e.g., EmailNotifier class).

```
from abc import ABC, abstractmethod


class Notifier(ABC):
    @abstractmethod
    def notify(self, message):
        pass


class EmailNotifier(Notifier):
    def notify(self, message):
        # Send email notification
        pass


class NotificationService:
    def _init_(self, notifier: Notifier):
        self.notifier = notifier


    def send_notification(self, message):
        self.notifier.notify(message)
```

Benefits:

- More flexible and reusable code
- Easier testing (can inject mock dependencies)
- Reduced module coupling
- Practical Benefits of SOLID Principles