VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

DAA LAB 6

231071052

BATCH – C

LABORATORY 6

SOLID PRINCIPLES:

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**: Each class or module should focus on a single functionality. If a class has multiple responsibilities, changes in one area might impact others, leading to fragile code.
- **Example**: A User class that only manages user data, separate from a UserRepository class that handles database operations related to users.

```
# Violation of SRP
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def save_to_db(self):
        # Code to save user data to the database
        pass

# Following SRP
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:
    def save_to_db(self, user):
        # Code to save user data to the database
        pass
```

In this example, the User class only handles user-related data, while the UserRepository class is responsible for database operations.

2. Open/Closed Principle (OCP)

- **Definition**: Software entities should be open for extension but closed for modification.
- **Explanation**: You should be able to add new features or extend functionality without altering existing code, which helps prevent introducing new bugs in tested code.
- **Example**: Adding a new type of payment method by creating a new class rather than modifying an existing Payment class.

```python
# Violation of OCP
class PaymentProcessor:
    def process_payment(self, payment_type):
        if payment_type == "credit":
            # process credit payment
            pass
        elif payment_type == "paypal":
            # process PayPal payment
            pass

# Following OCP
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def process(self):
        pass

class CreditPayment(PaymentMethod):
    def process(self):
        # process credit payment
        pass

class PaypalPayment(PaymentMethod):
    def process(self):
        # process PayPal payment
        pass

# Usage
def process_payment(payment_method: PaymentMethod):
    payment_method.process()
```

Now, you can add new payment methods by creating new subclasses of PaymentMethod without modifying existing code.

3. Liskov Substitution Principle (LSP)

- **Definition**: Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.
- **Explanation**: Subclasses should extend the functionality of a parent class without changing its original behavior. This principle ensures that a derived class can stand in for its base class without causing errors.
- **Example**: If you have a Bird class, the subclass Penguin should not break the functionality of methods expecting a Bird (e.g., Fly() method should be handled carefully since penguins cannot fly).

```
# Violation of LSP
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        # Sparrow flying logic
        pass

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Penguins can't fly")  # Violates LSP

# Following LSP
class Bird:
    def move(self):
        pass

class FlyingBird(Bird):
    def move(self):
        # logic for flying
        pass

class Penguin(Bird):
    def move(self):
        # logic for swimming
        pass
```

In this version, Penguin doesn't override a fly() method that it cannot fulfill, adhering to the Liskov Substitution Principle.

4. Interface Segregation Principle (ISP)

- **Definition**: A client should not be forced to implement an interface it doesn't use.
- **Explanation**: Instead of one large, complex interface, multiple smaller, specific interfaces are better. This allows classes to implement only what they need.
- **Example**: Instead of a Worker interface with methods like work() and manage(), create separate Worker and Manager interfaces, so each class implements only what's necessary.

```
# Violation of ISP
zclass Worker:
    def work(self):
        pass

    def manage(self):
        pass

class Developer(Worker):
    def work(self):
        # coding work
        pass

    def manage(self):
        raise NotImplementedError("Developers don't manage")  # Violates
ISP

# Following ISP
class Workable:
    def work(self):
        pass

class Manageable:
    def manage(self):
        pass

class Developer(Workable):
    def work(self):
        # coding work
        pass

class Manager(Workable, Manageable):
    def work(self):
        # management work
        pass
```

```
    def manage(self):
        # manage team
        pass
```

5. Dependency Inversion Principle (DIP)

- **Definition**: High-level modules should not depend on low-level modules. Both should depend on abstractions.
- **Explanation**: Instead of high-level modules relying on concrete implementations of lower-level modules, both should depend on abstract classes or interfaces. This reduces tight coupling and enhances flexibility.
- **Example**: In a notification system, instead of a User class depending on a specific EmailService, it should depend on an INotificationService interface. This way, the type of notification can be swapped (e.g., email, SMS) without modifying the User class.

```
# Violation of DIP
class EmailService:
    def send(self, message):
        # logic to send email
        pass

class Notification:
    def __init__(self):
        self.email_service = EmailService()

    def send_notification(self, message):
        self.email_service.send(message)

# Following DIP
from abc import ABC, abstractmethod

class NotificationService(ABC):
    @abstractmethod
    def send(self, message):
        pass

class EmailService(NotificationService):
    def send(self, message):
        # logic to send email
```

```
        pass

class SMSService(NotificationService):
    def send(self, message):
        # logic to send SMS
        pass

class Notification:
    def __init__(self, service: NotificationService):
        self.service = service

    def send_notification(self, message):
        self.service.send(message)

# Usage
email_service = EmailService()
notification = Notification(email_service)
notification.send_notification("Hello World")
```

Here, `Notification` depends on the abstraction `NotificationService` instead of a specific implementation, making it easy to swap `EmailService` with `SMSService`.