Usman Institute of Technology

Department of Computer Science

Course Code: SE308

Course Title: Software Design and Architecture

Fall 2022

Lab 05

OBJECTIVE: To Understand and Implement the SOLID Design Principles

Student Information

Group Members	
Student Name	**Muhammad Abrar Bajwa (20B-017-SE)**
Student Name	**Muhammad Waleed (20B-115-SE)**
Student Name	**Farhan Ali (20B-055-SE)**
Date	**10-11-2022**

Assessment

Marks Obtained	
Remarks	
Signature	

Software Design Principle (SOLID)

Several principles have been identified throughout the literature that help in making component-level design decisions, including (SOLID). These design principles intended to make object-oriented designs more understandable, flexible, and maintainable. The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin [1] also called "Uncle Bob".

1. Single Responsibility Principle
2. The open–closed principle (OCP)
3. The Liskov substitution principle (LSP)
4. The interface segregation principle (ISP)
5. Dependency Inversion Principle

1. Single Responsibility Principle

The single responsibility principle (SRP) states that every class, method, and function should have only one job or one reason to change.

Consider any game application that do gaming activities as well as to track the scores, so the Game class kept the responsibility to keep game playing activities, and the Scorer class got the keep all score as stats section.

*Game.py -- **SRP Violation***

```
Class Game :  def play (self) def saveScore (self) def hardLevel (self)
```

- The Three functions, it declares are certainly functions belonging to a game.
- The *play* and *hardlevel* functions manage the playing activities
- while the *saveScore* keep all score as stats

Rectification : To make the Game class conforms to the single responsibility principle, you'll need to create another class that is in charge of storing game scores in database

Exercise 1: Use the single responsibility principle to separate classes, methods, implement your own scenario.

[1] Martin, Robert C. (2000). "Design Principles and Design Patterns" (PDF). Retrieved 2022-11-10
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

2. *The open–closed principle (OCP)*

The open-closed principle states that a class, method, and function should be open for extension

*Emp.py -- **OCP Violation***

```
Class ITEmployee :  def
        work (self)  def
        Analysis (self) def
        develope (self) def
        test (self)
```

declares

- Consider the four functions, it certainly functions belonging to a ITEmployee.
- What will you do when new types of Employees come?
- This if-elif chain eventually will become hell and very hard to maintain.
- You can create a new class as a subclass of the Employee

Appy SPR first then follow the rectification instruction

Rectification: Now Employee is an abstract class and it has an abstract method called work. All subclasses of this class have to implement a work function. Developer calls its develop method and Tester calls its test method. In the company, all we had to do is calling the work() method of the given employee. If I need to add a new Employee like Maintenance Engineer, all you need to do is implement the work() method

Exercise 2: Use the Open/Close principle implement your own scenario.

3. *The Liskov substitution principle (LSP*

The Liskov substitution principle states that a child class must be substitutable for its parent class. Liskov substitution principle aims to ensure that the child class can assume the place of its parent class without causing any errors.

This principle was coined by Barbar Liskov [1] in her work regarding data abstraction and type theory. by Contract (DBC) by Bertrand Meyer [1].

```
class Employee(ABC):
        def save (self):

class Students (Employee):
        def save (self)

class Teacher (Employee)
        def save (self)

class Manager (Employee)
        def save (self)
```

- Consider the four Class
- Let's modify the code and add one more abstract method salary() in each class.
- Assume that students did not get paid
- Student class will throw exceptions or not work as expected.

Rectification : Remove the salary() method from Student and create a new abstract class Payment

Exercise 3: Use the Liskov Substitute principle and implement it with your own scenario.

4. ***The interface segregation principle (ISP***

   The interface segregation principle states that an interface should be as small a possible in terms of cohesion. In other words, it should do ONE thing

5. ***Dependency Inversion Principle***

   The dependency inversion principle states that, high-level modules should not depend on lowlevel modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions

Exercise 4: Use the Interface Sagregation principle and implement it with the given scenario (Web Resource).

Exercise 5: Use the Dependency Inversion principle and implement it with given scenario (Web Resource).

Order and Payment

1) Single Responsibility Principle

DIInterface.py

```python
from abc import ABCMeta, abstractmethod


class Order(metaclass=ABCMeta):
    @abstractmethod
    def add_items(self, item_name: str, price: float, quantity: int)
-> None:
        pass

    @abstractmethod
    def calculate_total(self) -> None:
        pass


class DemoPayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order, code: int) -> None:
        pass


class ExamplePayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order) -> None:
        pass
```

Example.py

```python
from DIInterface import Order as IOrder, ExamplePayment
from abc import abstractmethod, ABCMeta


class Authorize:
    is_valid = False

    def verify(self) -> None:
        self.is_valid = True

    def is_authorized(self) -> bool:
        return self.is_valid


class GenericAuthentication(metaclass=ABCMeta):
    @abstractmethod
    def is_authorized(self) -> bool:
        pass
```

```python
class Order(IOrder):
    items = []
    status = "Not Paid"

    def add_items(self, item_name: str, price: float, quantity:
int):
        item = {
            "name": item_name,
            "price": price,
            "quantity": quantity
        }

        self.items.append(item)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item.get('price') * item.get('quantity')

        return total


class DebitPayment(ExamplePayment):

    def __init__(self, code: int, validator: GenericAuthentication):
        self.code = code
        self.validator = validator

    def pay(self, order: Order):
        if not self.validator.is_authorized():
            raise Exception("User not verified.")
        print(f'Payment is processing using debit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class CreditPayment(ExamplePayment):

    def __init__(self, code: int, validator: GenericAuthentication):
        self.code = code
        self.validator = validator

    def pay(self, order: Order):
        if not self.validator.is_authorized():
            raise Exception("User not verified.")
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')
```

```python
class VisaPayment(ExamplePayment):
    def __init__(self, email: str):
        self.email = email

    def pay(self, order: Order):
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.email}.')
        order.status = 'Paid'
        print(f'Payment Successful')


# Testing.
order = Order()
order.add_items("Bajwa", 100, 2)
order.add_items("Waleed", 500, 2)
order.calculate_total()

# validate
validator = Authorize()

# Payment.
visa_payment = CreditPayment(123, validator)
validator.verify()
visa_payment.pay(order)
```

Output

```
PS D:\UNI\Semester 5\SDA\Labs> & "C:/Users/MUHAMMAD ABRAR BAJWA/AppData/Local/Programs/Python/Python39/python.exe" "d:/UNI/Seme
ster 5/SDA/Labs/Lab 5/Solid Design Principle (Order and Payment)/SingleResponsibility/example.py"
Payment is processing using debit card.
Validating code 123.
Payment Successful
PS D:\UNI\Semester 5\SDA\Labs>
```

2) Open/Close Principle

OCInterface.py

```
from abc import ABCMeta, abstractmethod


class Order(metaclass=ABCMeta):
    @abstractmethod
    def add_items(self, item_name: str, price: float, quantity: int)
-> None:
        pass

    @abstractmethod
    def calculate_total(self) -> None:
        pass


class Payment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order, code: int) -> None:
        pass


class DemoPayment(metaclass=ABCMeta):
    @abstractmethod
    def debit(self, order: Order) -> None:
        pass

    @abstractmethod
    def credit(self, order: Order) -> None:
        pass
```

Example.py

```
from OCInterface import Order as IOrder, Payment


class Order(IOrder):
    items = []
    status = "Not Paid"

    def add_items(self, item_name: str, price: float, quantity:
int):
        item = {
            "name": item_name,
            "price": price,
            "quantity": quantity
        }

        self.items.append(item)

    def calculate_total(self):
```

```
        total = 0
        for item in self.items:
            total += item.get('price') * item.get('quantity')

        return total


class DebitPayment(Payment):
    @staticmethod
    def pay(order: Order, code: int):
        print(f'Payment is processing using debit card.')
        print(f'Validating code {code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class CreditPayment(Payment):
    @staticmethod
    def pay(order: Order, code: int):
        print(f'Payment is processing using credit card.')
        print(f'Validating code {code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


# Testing.
order = Order()
order.add_items("Bajwa", 100, 2)
order.add_items("Waleed", 500, 2)
order.calculate_total()

# Pay debit.
debit_payment = DebitPayment()
debit_payment.pay(order, 123)

# Pay credit.
credit_payment = CreditPayment()
credit_payment.pay(order, 123)
```

Output

```
PS D:\UNI\Semester 5\SDA\Labs> & "C:/Users/MUHAMMAD ABRAR BAJWA/AppData/Local/Programs/Python/Python39/python.exe" "d:/UNI/Seme
ster 5/SDA/Labs/Lab 5/Solid Design Principle (Order and Payment)/OpenClose/example.py"
Payment is processing using debit card.
Validating code 123.
Payment Successful
Payment is processing using credit card.
Validating code 123.
Payment Successful
PS D:\UNI\Semester 5\SDA\Labs>
```

3) Liskov Substitute Principle

LSInterface.py

```
from abc import ABCMeta, abstractmethod


class Order(metaclass=ABCMeta):
    @abstractmethod
    def add_items(self, item_name: str, price: float, quantity: int)
-> None:
        pass

    @abstractmethod
    def calculate_total(self) -> None:
        pass


class DemoPayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order, code: int) -> None:
        pass


class Payment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order) -> None:
        pass
```

Example.py

```
from LSInterface import Order as IOrder, Payment


class Order(IOrder):
    items = []
    status = "Not Paid"

    def add_items(self, item_name: str, price: float, quantity:
int):
        item = {
            "name": item_name,
            "price": price,
            "quantity": quantity
        }

        self.items.append(item)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item.get('price') * item.get('quantity')

        return total
```

```python
class DebitPayment(Payment):
    def __init__(self, code: int):
        self.code = code

    def pay(self, order: Order):
        print(f'Payment is processing using debit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class CreditPayment(Payment):
    def __init__(self, code: int):
        self.code = code

    def pay(self, order: Order):
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class VisaPayment(Payment):
    def __init__(self, email: str):
        self.email = email

    def pay(self, order: Order):
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.email}.')
        order.status = 'Paid'
        print(f'Payment Successful')


# Testing.
order = Order()
order.add_items("Bajwa", 100, 2)
order.add_items("Waleed", 500, 2)
order.calculate_total()

# Payment.
visa_payment = VisaPayment('test@gmail.com')
visa_payment.pay(order)
```

Output

```
PS D:\UNI\Semester 5\SDA\Labs> & "C:/Users/MUHAMMAD ABRAR BAJWA/AppData/Local/Programs/Python/Python39/python.exe" "d:/UNI/Semester 5/SDA/Labs/Lab 5/Solid Design Principle (Order and Payment)/LiskovSubstitution/example.py"
Payment is processing using credit card.
Validating code test@gmail.com.
Payment Successful
PS D:\UNI\Semester 5\SDA\Labs>
```

4) Interface Segregation Principle

ISInterface.py

```
from abc import ABCMeta, abstractmethod


class Order(metaclass=ABCMeta):
    @abstractmethod
    def add_items(self, item_name: str, price: float, quantity: int)
-> None:
        pass

    @abstractmethod
    def calculate_total(self) -> None:
        pass


class DemoPayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order, code: int) -> None:
        pass


class ExamplePayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order) -> None:
        pass
```

Example.py

```
from abc import abstractmethod
from ISInterface import Order as IOrder, DemoPayment, ExamplePayment


class PaymentWithAuth(DemoPayment):

    @abstractmethod
    def two_factor_auth(self) -> None:
        pass


class Order(IOrder):
    items = []
    status = "Not Paid"

    def add_items(self, item_name: str, price: float, quantity:
int):
        item = {
            "name": item_name,
            "price": price,
            "quantity": quantity
        }
```

```
        self.items.append(item)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item.get('price') * item.get('quantity')

        return total


class DebitPayment(PaymentWithAuth):

    def __init__(self, code: int):
        self.is_verified = False
        self.code = code

    def pay(self, order: Order):
        if not self.is_verified:
            raise Exception("User not verified.")
        print(f'Payment is processing using debit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')

    def two_factor_auth(self):
        self.is_verified = True


class CreditPayment(PaymentWithAuth):
    def __init__(self, code: int):
        self.code = code
        self.is_verified = False

    def pay(self, order: Order):
        if not self.is_verified:
            raise Exception("User not verified.")
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')

    def two_factor_auth(self):
        self.is_verified = True


class VisaPayment(ExamplePayment):
    def __init__(self, email: str):
        self.email = email

    def pay(self, order: Order):
        print(f'Payment is processing using credit card.')
```

```
        print(f'Validating code {self.email}.')
        order.status = 'Paid'
        print(f'Payment Successful')


# Testing.
order = Order()
order.add_items("Bajwa", 100, 2)
order.add_items("Waleed", 500, 2)
order.calculate_total()

# Payment.
visa_payment = VisaPayment('test@gmail.com')
visa_payment.pay(order)
```

Output

```
PS D:\UNI\Semester 5\SDA\Labs> & "C:/Users/MUHAMMAD ABRAR BAJWA/AppData/Local/Programs/Python/Python39/python.exe" "d:/UNI/Seme
ster 5/SDA/Labs/Lab 5/Solid Design Principle (Order and Payment)/InterfaceSegregation/example.py"
Payment is processing using credit card.
Validating code test@gmail.com.
Payment Successful
PS D:\UNI\Semester 5\SDA\Labs>
```

5) Dependency Inversion Principle

DIInterface.py

```
from abc import ABCMeta, abstractmethod


class Order(metaclass=ABCMeta):
    @abstractmethod
    def add_items(self, item_name: str, price: float, quantity: int)
-> None:
        pass

    @abstractmethod
    def calculate_total(self) -> None:
        pass


class DemoPayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order, code: int) -> None:
        pass


class ExamplePayment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, order: Order) -> None:
        pass
```

Example.py

```
from DIInterface import Order as IOrder, ExamplePayment
from abc import abstractmethod, ABCMeta


class Authorize:
    is_valid = False

    def verify(self) -> None:
        self.is_valid = True

    def is_authorized(self) -> bool:
        return self.is_valid


class GenericAuthentication(metaclass=ABCMeta):
    @abstractmethod
    def is_authorized(self) -> bool:
        pass


class Order(IOrder):
    items = []
    status = "Not Paid"

    def add_items(self, item_name: str, price: float, quantity:
int):
        item = {
            "name": item_name,
            "price": price,
            "quantity": quantity
        }

        self.items.append(item)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item.get('price') * item.get('quantity')

        return total


class DebitPayment(ExamplePayment):

    def __init__(self, code: int, validator: GenericAuthentication):
        self.code = code
        self.validator = validator

    def pay(self, order: Order):
        if not self.validator.is_authorized():
            raise Exception("User not verified.")
```

```
        print(f'Payment is processing using debit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class CreditPayment(ExamplePayment):

    def __init__(self, code: int, validator: GenericAuthentication):
        self.code = code
        self.validator = validator

    def pay(self, order: Order):
        if not self.validator.is_authorized():
            raise Exception("User not verified.")
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.code}.')
        order.status = 'Paid'
        print(f'Payment Successful')


class VisaPayment(ExamplePayment):
    def __init__(self, email: str):
        self.email = email

    def pay(self, order: Order):
        print(f'Payment is processing using credit card.')
        print(f'Validating code {self.email}.')
        order.status = 'Paid'
        print(f'Payment Successful')


# Testing.
order = Order()
order.add_items("Bajwa", 100, 2)
order.add_items("Waleed", 500, 2)
order.calculate_total()

# validate
validator = Authorize()

# Payment.
visa_payment = CreditPayment(123, validator)
validator.verify()
visa_payment.pay(order)
```

Output

```
PS D:\UNI\Semester 5\SDA\Labs> & "C:/Users/MUHAMMAD ABRAR BAJWA/AppData/Local/Programs/Python/Python39/python.exe" "d:/UNI/Semester 5/SDA/Labs/Lab 5/Solid Design Principle (Order and Payment)/DependencyInversion/example.py"
Payment is processing using credit card.
Validating code 123.
Payment Successful
PS D:\UNI\Semester 5\SDA\Labs>
```