설계패턴 2주차 과제

컴퓨터공학과 202001818 송강규

Lab 1)

1)

class Memo:

    def \_\_init\_\_(self):

        self.memo\_list = []

    def addMemo(self, memo): #메모를 리스트로 추가하는 기능

        self.memo\_list.append(memo)

    def printMemo(self): #모든 메모를 콘솔로 출력하는 기능

        for i in range(len(self.memo\_list)):

            print(f'{i + 1}. {self.memo\_list[i]}')

    def saveMemo(self, memo): #메모를 파일에 저장하는 기능

        with open(memo, "w") as file:

            for memos in self.memo\_list:

                file.write(memos + "\n")

    def getMemo(self, memo): #파일로부터 메모를 불러오는 기능

        with open(memo, "r") as file:

            contents = file.read()

2) SOLID 원칙에 따르면 위 코드는 Single Responsibility Principle을 위반했다. 즉 모든 클래스는 하나의 책임만 가져야 하지만 작성한 코드는 Memo 클래스 안에 메모를 추가, 출력, 저장, 불러오기까지 총 4개의 기능을 가지고 있으므로 수정사항이 발생하면 모든 클래스를 수정해야하는 상황이 발생할 수 있다.

class Memo:

    def \_\_init\_\_(self):

        self.memo\_list = []

    def addMemo(self, memo):

        self.memo\_list.append(memo)

    def printMemo(self):

        for i in range(len(self.memo\_list)):

            print(f'{i + 1}. {self.memo\_list[i]}')

    def saveMemo(self, file\_path):

        memo\_data = '\n'.join(self.memo\_list)

        file\_writer = FileWriter()

        file\_writer.write(file\_path, memo\_data)

    def loadMemo(self, file\_path):

        file\_reader = FileReader()

        self.memo\_list = file\_reader.read(file\_path)

class FileWriter:

    def write(self, file\_path, data):

        with open(file\_path, "w") as file:

            file.write(data)

class FileReader:

    def read(self, file\_path):

        with open(file\_path, "r") as file:

            return file.readlines()

위의 코드는 Single Responsibility Principle을 따르면 코드를 재구성하여 Memo 클래스는 메모 관리 기능에만 책임을 다하고 파일 입출력은 FileWriter 클래스와 FileReader 클래스로 분리하여 각 클래스가 하나의 책임만 가지게 했다.

Lab 2)

1) 직원들의 급여를 계산하는 코드는 SOLID 원칙에 따르면 Single Responsibility 원칙을 위반했다. 그 이유는 Employee 클래스에서 직원의 정보와 급여계산을 모두 처리하고 있기 때문이다.

2) 1의 문제점을 해결하기 위해 급여 계산 기능을 따로 클래스로 분리하면 된다. 그렇게 되면 Employee 클래스에서는 직원의 정보만 관리하고 급여 계산 클래스에서 급여를 계산하면 되기 때문에 직급이 추가 혹은 삭제됐을 때 Employee 클래스는 수정할 필요가 없이 새로운 직급에 대해서 클래스만 추가하면 된다.

class Employee:

    def \_\_init\_\_(self, type\_of\_employee, base\_salary):

        self.type\_of\_employee = type\_of\_employee

        self.base\_salary = base\_salary

class SalaryCalculator:

    def calculate\_salary(employee):

        base\_salary = employee.base\_salary

        if employee.type\_of\_employee == "permanent":

            return base\_salary + 1000

        elif employee.type\_of\_employee == "temporary":

            return base\_salary + 500

        elif employee.type\_of\_employee == "intern":

            return base\_salary + 200

permanent\_employee = Employee("permanent", 3000)

print(SalaryCalculator.calculate\_salary(permanent\_employee))

temporary\_employee = Employee("temporary", 3000)

print(SalaryCalculator.calculate\_salary(temporary\_employee))

intern\_employee = Employee("intern", 3000)

print(SalaryCalculator.calculate\_salary(intern\_employee))

Lab3)

1)

Light  
Fan

Swith

Turn\_light\_on  
turn\_light\_off  
turn\_fan\_on  
turn\_fan\_off

Fan

*turn\_on  
turn\_off*

Turn\_off

Light

*turn\_on  
turn\_off*

Turn\_off

2) Lab 3 코드의 SOLID 윈칙 중 지켜지지 않은 원칙은 Dependency Inversion Principle로 상위 계층 클래스가 하위 계층 클래스에 독립되어 있지 않고 의존하고 있다. 즉 Switch 클래스가 Light 클래스와 Fan 클래스 구현에 의존하고 있기 때문에 위반했다고 할 수 있다.

3) 위의 문제를 고치기 위해서 Switchable 클래스를 통해 의존 관계가 역전 시켜 Switch 클래스가 Light 클래스와 Fan 클래스로부터 독립될 수 있다.

from abc import ABC, abstractmethod

class Switchable(ABC):

    @abstractmethod

    def turn\_on(self):

        pass

    @abstractmethod

    def turn\_off(self):

        pass

class Light(Switchable):

    def turn\_on(self):

        print("Light turned on")

    def turn\_off(self):

        print("Light turned off")

class Fan(Switchable):

    def turn\_on(self):

        print("Fan turned on")

    def turn\_off(self):

        print("Fan turned off")

class Switch:

    def \_\_init\_\_(self, device):

        self.device = device

    def turn\_on(self):

        self.device.turn\_on()

    def turn\_off(self):

        self.device.turn\_off()

*turn\_on  
turn\_off*

Turn\_off

Fan

*turn\_on  
turn\_off*

Turn\_off

Switchable

Light

*turn\_on  
turn\_off*

Turn\_off

*turn\_on  
turn\_off*

Turn\_off

Fan

Lab 4)

1) 집과 차의 기능을 같이 가지고 있는 RV (캠핑카)를 예시로 들었다

House

Car

CarHouse Interface

Sleep()

Cook()

Bath()

Drive()

Travel()

Sleep()

Cook()

Bath()

Drive()

Travel()

RV

Sleep()

Cook()

Bath()

~~Drive()~~

~~Travel()~~

~~Sleep()~~

~~Cook()~~

~~Bath()~~

Drive()

Travel()

2) 아래 다이어그램처럼 재구성하면 Interface Segregation을 위반하지 않는다!

Sleep()

Cook()

Bath()

Drive()

Travel()

House

Sleep()

Cook()

Bath()

Drive()

Travel()

RV

Car

Sleep()

Cook()

Bath()

House interface

Car interface

Drive()

Travel()