Julián Rodrigo Gamba López

Actividad 2 - Implementación de principios SOLID y GRASP

Link Repositorio:
https://github.com/jgambalo/SOLID-GRASP---ACTIVIDAD2

S: Principio de responsabilidad única

Se crea la clase con atributos y métodos de sólo parametrización de Número de pisos, nombre y Número de apartamentos. Además, posee un método para calcular los habitantes del edificio.

```python
# Attributes y Methods
class Building:
    def __init__(self, floors, name, apartments):
        self.floors = floors
        self.name = name
        self.apartments = apartments

    def calculateHabitants(self):
        print ("Working calculateHabitants")
```

Se crea solamente con las operaciones que van a afectar la base de datos, recibiendo como atributo un objeto de la clase Building. Con funciones de guardar y eliminar un registro

```python
# DB Operations
class BuildingDB:
    def __init__(self, build):
        self.floors = build.floors
        self.name = build.name
        self.apartments = build.apartments

    def saveBuilding(self):
        print ("Working saveBuilding", self.name)

    def deleteBuilding(self):
        print ("Working deleteBuilding", self.name)
```

Julián Rodrigo Gamba López

Pruebas de testeo del programa

```
#Test
edificio = Building(10, "Bacata", 40)
BuildingDB(edificio).saveBuilding()
BuildingDB(edificio).deleteBuilding()
```

Resultados

```
PS D:\Documentos\GEC\Universidad\Actividad 2> python "Ejemplo S.py"
Working saveBuilding Bacata
Working deleteBuilding Bacata
```

Como se puede observar, se implementó satisfactoriamente el Principio de responsabilidad única, con el cual cada clase tienen una sola responsabilidad y alguna modificación no alterará otras funciones del código. Además, nuestro código está mucho más encapsulado

O: Principio abierto/cerrado y L: Principio de sustitución de Liskov

Con el siguiente código se ejemplificará la aplicación del Principio abierto/cerrado y Principio de Sustitución de Liskov. Iniciamos con la creación de la clase abstracta City con sólo un método extensivo de población

```
import abc
from abc import ABC, abstractmethod

# Abstract Class
class City(ABC):
    @abc.abstractproperty
    def population(self):
        return 0
```

Julián Rodrigo Gamba López

Se crean las clases extensivas de diferentes ciudades (New York, Los Angeles y Dallas) retornando los valores de población de cada una

```
#Extended Classes
class newYork(City):
    @property
    def population(self):
        return 18000

class LA(City):
    @property
    def population(self):
        return 5000

class Dallas(City):
    @property
    def population(self):
        return 1000
```

Pruebas de testeo del programa

```
#Test
cityArray = []

cityArray.append (newYork())
cityArray.append (LA())
cityArray.append (Dallas())

for obj in cityArray:
    print(obj.population)
```

Resultado

```
PS D:\Documentos\GEC\Universidad\Actividad 2> python "Ejemplo O y L.py"
18000
5000
1000
```

Cómo se puede observar cada clase de cada ciudad extiende la clase City e implementa el método population. Por lo tanto, si añadimos una nueva ciudad no se tendrá que alterar la clase principal de City.

Por otra parte, se logra evidenciar el cumplimiento del Principio de sustitución de Liskov, donde con la implementación del método extensivo se varía la subclase de cada ciudad más no la Superclase City

I: Principio de segregación de interfaz

Para llevar a cabo el Principio de Segregación de Interfaz, inicialmente se crean las interfaces necesarias para no implementar métodos innecesarios en cada una de las clases

```python
import abc
from abc import ABC, abstractmethod

# Specific Interfaces
class iAnimal(ABC):
    @abc.abstractmethod
    def eat(self):
        pass

class iFlyAnimal(ABC):
    @abc.abstractmethod
    def fly(self):
        pass

class iWalkAnimal(ABC):
    @abc.abstractmethod
    def walk(self):
        pass
```

Julián Rodrigo Gamba López

Luego se crean las clases de cada uno de los animales, en este punto implementamos sólo las interfaces necesarias para el funcionamiento de esta clase

```
# Animals Classes
class bird(iAnimal, iFlyAnimal):
    @classmethod
    def eat(self):
        print("Eat")

    @classmethod
    def fly(self):
        print("Fly")

class feline(iAnimal,iWalkAnimal):
    @classmethod
    def eat(self):
        print("Eat")

    @classmethod
    def walk(self):
        print("Walk")
```

Ahora procedemos a realizar una prueba de funcionamiento

```
#Test
print("---Dove---")
dove = bird()
dove.eat()
dove.fly()
print("---Tiger---")
tiger = feline()
tiger.eat()
tiger.walk()
```

Resultado

```
PS D:\Documentos\GEC\Universidad\Actividad 2> python "Ejemplo I.py"
---Dove---
Eat
Fly
---Tiger---
Eat
Walk
```

Cómo se observa, al implementar este principio se logra que cada uno de los objetos creados posea solamente los métodos necesarios para su funcionamiento, lo que conlleva a un entendimiento y mejores prácticas a la hora decodificar

D: Principio de inversión de dependencias

Para implementar este principio se creará una interfaz con los métodos más simple para la administración de datos ya sea en un servicio de base de datos o una API.

```
# Connection Interface
class connection(ABC):
    @abc.abstractmethod
    def getData(self):
        pass

    @abc.abstractmethod
    def setData(self):
        pass
```

Luego vamos a implementar esta interfaz en nuestras clases, ya sea de Base de datos o API.

```
#DB Class
class DatabaseService(connection):

    @classmethod
    def getData(self):
        print("Getting Data From DB")

    @classmethod
    def setData(self):
        print("Setting Data From DB")

#API Class
class APIService(connection):

    @classmethod
    def getData(self):
        print("Getting Data From API")

    @classmethod
    def setData(self):
        print("Setting Data From API")
```

Ahora procedemos a realizar una prueba de funcionamiento

```
#Test
DB = DatabaseService()
DB.getData()
DB.setData()

API = APIService()
API.getData()
API.setData()
```

Resultado

```
PS D:\Documentos\GEC\Universidad\Actividad 2> python "Ejemplo D.py"
Getting Data From DB
Setting Data From DB
Getting Data From API
Setting Data From API
```

Cómo podemos observar, con este principio se logra generalizar algunas tareas donde en caso de alguna variación en la infraestructura de toma de datos, solamente es necesario modificar puntualmente la clase e implementar la interfaz genérica de esta. Lo que ayuda potencialmente a realizar modificaciones en pequeña escala al código en caso de estos cambios.