LAPORAN PRAKTIKUM
PEMROGRAMAN BERORIENTASI OBJEK RB
MODUL 6

Oleh :

Ilham Yoga Pratama (121140081)

Program Studi Teknik Informatika

Institut Teknologi Sumatera

2023

1. Kelas Abstrak

a) Pengertian Kelas Abstrak

Kelas abstrak adalah kelas yang masih dalam bentuk abstrak. Karena bentuknya masih abstrak, dia tidak bisa dibuat langsung menjadi objek. Sebuah kelas agar dapat disebut kelas abstrak setidaknya memiliki satu atau lebih method abstrak. Method abstrak adalah method yang tidak memiliki implementasi atau tidak ada bentuk konkritnya akan tetapi memiliki deklarasi.

b) Implementasi Kelas Abstrak

Secara default, Python tidak menyediakan kelas abstrak. Python hadir dengan modul yang menyediakan basis untuk mendefinisikan Abstract Base Classes (ABC). Fungsi abstrak pada kelas abstrak ditandai dengan memberikan decorator @abstractmethod pada fungsi yang dibuat. Dalam hal ini, kita perlu menggunakan kelas abstrak ketika ingin membuat kerangka kerja untuk kelas turunan yang memiliki fitur-fitur umum. Misalnya, jika kita ingin membuat sebuah game, kita bisa membuat kelas abstrak "Character" yang memiliki fitur-fitur umum seperti health point, damage, dan posisi di peta. Lalu, kita bisa membuat kelas turunan seperti "Player" dan "Enemy" yang memiliki fitur-fitur yang lebih spesifik.

Dengan menggunakan kelas abstrak, kita bisa menghindari duplikasi kode dan mempercepat proses pengembangan, karena fitur-fitur umum sudah diatur di kelas abstrak. Selain itu, penggunaan kelas abstrak juga memungkinkan kita untuk memaksakan kelas turunan untuk mengimplementasikan metode-metode yang dibutuhkan.

Berikut dilampirkan kode dengan mengimplementasikan kelas abstrak dengan modul ABC:

```
from abc import ABC, abstractmethod

class Character(ABC):
    def __init__(self, hp, damage, x, y):
        self.hp = hp
        self.damage = damage
        self.x = x
        self.y = y

    @abstractmethod
    def attack(self, target):
        pass

class Player(Character):
    def __init__(self, hp, damage, x, y):
        super().__init__(hp, damage, x, y)

    def attack(self, target):
        target.hp -= self.damage
        print("Player attacks! Target HP:
{}".format(target.hp))
class Enemy(Character):
    def __init__(self, hp, damage, x, y):
        super().__init__(hp, damage, x, y)

    def attack(self, target):
        target.hp -= self.damage
        print("Enemy attacks! Target HP: {}".format(target.hp))

player = Player(100, 10, 0, 0)
enemy = Enemy(50, 5, 10, 10)

player.attack(enemy)
enemy.attack(player)
```

2. Interface

a) Pengertian Interface

Interface pada Python didefinisikan menggunakan kelas python dan merupakan subclass interface. Interface juga merupakan suatu kontrak atau speseifikasi yang harus dipenuhi oleh sebuah objek atau kelas, tanpa memperhatikan implementasi di dalamnya. Interface berisi metode yang bersifat abstrak. Dalam hal ini interface dapat dibagi menjadi dua macam yaitu :

1. Informal Interface

   Dalam beberapa kasus, kita mungkin tidak memerlukan batasan ketat dari interface Python formal. Karena Python bersifat dinamis, kita dapat membuat interface informal. Interface Python informal adalah kelas yang mendefinisikan metode yang dapat diganti tanpa adanya unsur paksaan. Interface informal, umumnya dikenal sebagai **Protocols or Duck Typing.** Duck typing adalah konsep di mana sebuah objek diterima sebagai

argumen dalam suatu fungsi atau metode berdasarkan perilakunya, bukan tipe datanya. Sebuah objek dapat digunakan sebagai pengganti objek lain jika objek tersebut memiliki atribut dan metode yang dibutuhkan oleh fungsi atau metode tersebut.

Contoh kasus informal interface :

```python
from abc import ABC, abstractmethod

class Perhitungan(ABC):
    @abstractmethod
    def hitung_luas(self):
        pass

class Lingkaran(Perhitungan):
    def __init__(self, jari_jari):
        self.jari_jari = jari_jari

    def hitung_luas(self):
        luas = 3.14 * self.jari_jari ** 2
        print("Luas lingkaran dengan jari-jari {} adalah {}".format(self.jari_jari, luas))

class PersegiPanjang(Perhitungan):
    def __init__(self, panjang, lebar):
        self.panjang = panjang
        self.lebar = lebar

    def hitung_luas(self):
        luas = self.panjang * self.lebar
        print("Luas persegi panjang dengan panjang {} dan lebar {} adalah {}".format(self.panjang,
self.lebar, luas))

lingkaran = Lingkaran(7)
lingkaran.hitung_luas()

persegi_panjang = PersegiPanjang(10, 5)
persegi_panjang.hitung_luas()
```

Dalam kasus ini, terdapat interface "Perhitungan" yang mendefinisikan satu metode yaitu "hitung_luas". Metode ini harus diimplementasikan di dalam kelas-kelas yang mengimplementasikan interface tersebut.

Kelas "Lingkaran" dan "PersegiPanjang" adalah kelas-kelas yang mengimplementasikan interface "Perhitungan". Kedua kelas ini mengimplementasikan metode "hitung_luas" yang didefinisikan di dalam interface. Meskipun keduanya memiliki implementasi yang berbeda untuk masing-masing metode, namun keduanya memiliki tanda tangan metode yang sama, sehingga dapat digunakan secara interchangeable di dalam aplikasi kita.

Pada contoh ini, kelas "Lingkaran" dan "PersegiPanjang" memiliki metode "hitung_luas" yang berbeda, namun keduanya tetap mengimplementasikan interface "Perhitungan" karena keduanya memiliki tanda tangan metode "hitung_luas" yang sama. Sehingga, keduanya dapat digunakan secara interchangeable di dalam aplikasi kita.

2. Formal Interface

   Pada formal interface kita bisa menggunakan konsep Abstract Base Classes (ABC). Formal interface kelas abstrak dapat dibangun hanya dengan sedikit baris kode, untuk kemudian diimplementasikan pada kelas konkret.

   Contoh kasus formal interface :

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def get_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def draw(self):
        print("Menggambar persegi
panjang")
    def get_area(self):
        return self.length * self.width
```

   Pada contoh program di atas, kita menggunakan module ABC dari Python untuk membuat sebuah interface formal bernama "Shape". Interface formal ini memiliki dua metode yang harus diimplementasikan oleh setiap kelas yang mengimplementasikan interface ini, yaitu metode "draw()" dan "get_area()". Kedua metode ini didekorasi dengan decorator "@abstractmethod", yang menandakan bahwa metode tersebut hanya merupakan definisi dan harus diimplementasikan oleh kelas yang mengimplementasikan interface ini.

Selanjutnya, kita membuat sebuah kelas "Rectangle" yang mengimplementasikan interface formal "Shape". Kita mengimplementasikan kedua metode "draw()" dan "get_area()" sesuai dengan definisi yang telah didefinisikan di dalam interface formal "Shape".

Dalam Python, ketika kita membuat sebuah kelas yang mengimplementasikan sebuah interface formal, kita harus mengimplementasikan semua metode yang terdapat pada interface formal tersebut. Jika kita tidak mengimplementasikan semua metode tersebut, maka akan terjadi error saat program dijalankan.

b) Implementasi Interface

Pada dasarnya, bahasa pemrograman Python tidak memiliki konsep interface seperti yang dimiliki oleh bahasa pemrograman Java. Namun, dalam Python kita dapat menggunakan modul abc (abstract base class) untuk membuat sebuah interface formal. Meskipun tidak diperlukan, namun penggunaan interface formal pada Python dapat memberikan beberapa keuntungan, antara lain:

a. Modularitas: Dengan menggunakan interface formal, kita dapat memisahkan antarmuka dengan implementasi. Hal ini dapat membantu kita dalam membangun kode yang lebih modular dan reusable.
b. Polimorfisme: Interface formal memungkinkan kita untuk membuat sebuah kelas yang memiliki beberapa implementasi yang berbeda. Dalam Python, polimorfisme dapat membantu kita untuk membuat kode yang lebih fleksibel dan mudah untuk diubah.
c. Dokumentasi: Interface formal dapat digunakan sebagai dokumentasi untuk menunjukkan bagaimana sebuah kelas seharusnya digunakan. Hal ini dapat membantu dalam mempermudah pemahaman terhadap sebuah kelas atau modul.

Kita dapat menggunakan interface formal pada Python ketika kita ingin memisahkan antarmuka dengan implementasi, dan ingin memastikan bahwa sebuah kelas mengimplementasikan semua metode yang diperlukan. Namun, dalam kebanyakan kasus, penggunaan interface formal pada Python tidak selalu diperlukan. Kita dapat menggunakan tipe data abstrak seperti tuple, list, atau dictionary sebagai pengganti dari interface formal dalam Python.

3. Kelas Konkret

a) Pengertian Kelas Konkret

Kelas konkret adalah kelas biasa yang telah diimplementasikan secara lengkap dan dapat langsung digunakan tanpa perlu dilakukan pengembangan lebih lanjut. Kelas ini memiliki definisi lengkap untuk semua metodenya dan tidak ada metode yang perlu diimplementasikan lagi.

b) Implementasi Kelas Konkret

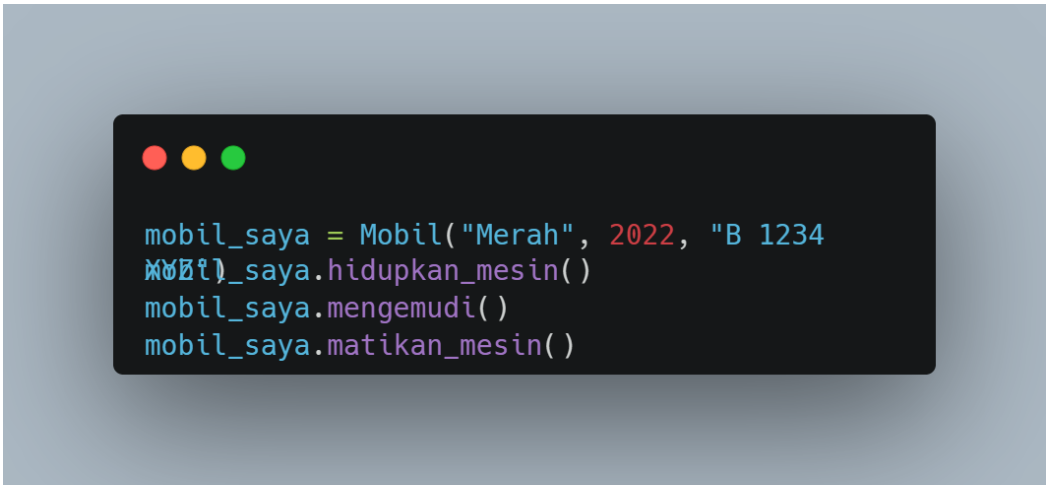
Kita perlu menggunakan kelas konkret ketika kita ingin membuat sebuah objek yang memiliki fungsi-fungsi tertentu secara lengkap dan siap digunakan. Kita dapat membuat objek dari kelas konkret tersebut dan langsung menggunakannya tanpa perlu melakukan pengembangan lebih lanjut. Jika kita ingin menambahkan fungsi baru atau mengubah implementasi dari sebuah kelas konkret, kita dapat melakukannya dengan mengubah kode di dalam kelas tersebut.

Contoh kasus kelas konkret :

```
class Mobil:
    def __init__(self, warna, tahun_produksi, nomor_plat):
        self.warna = warna
        self.tahun_produksi = tahun_produksi
        self.nomor_plat = nomor_plat
        self.mesin_hidup = False

    def hidupkan_mesin(self):
        if not self.mesin_hidup:
            print("Mesin dihidupkan")
            self.mesin_hidup = True
        else:
            print("Mesin sudah hidup")

    def matikan_mesin(self):
        if self.mesin_hidup:
            print("Mesin dimatikan")
            self.mesin_hidup = False
        else:
            print("Mesin sudah mati")

    def mengemudi(self):
        if self.mesin_hidup:
            print("Mobil sedang dikemudikan")
        else:
            print("Hidupkan mesin terlebih dahulu")
```

Kelas konkret ‘Mobil’ ini memiliki atribut warna, tahun produksi, dan nomor plat, serta metode ‘hidupkan_mesin’, ‘matikan_mesin’, dan ‘mengemudi’. Di dalam metode ‘__init__’, atribut-atribut tersebut diinisialisasi dengan nilai-nilai yang diberikan ketika objek dibuat.

Kelas konkret ‘Mobil’ ini dapat digunakan dengan membuat objek dari kelas tersebut. Contohnya:



```
mobil_saya = Mobil("Merah", 2022, "B 1234 XYZ")
mobil_saya.hidupkan_mesin()
mobil_saya.mengemudi()
mobil_saya.matikan_mesin()
```

Pada contoh di atas, objek ‘mobil_saya’ dibuat dengan atribut warna "Merah", tahun produksi 2022, dan nomor plat "B 1234 XYZ". Kemudian, metode ‘hidupkan_mesin’ dipanggil untuk menghidupkan mesin mobil, diikuti dengan metode ‘mengemudi’ untuk mengemudikan mobil, dan akhirnya metode ‘matikan_mesin’ dipanggil untuk mematikan mesin mobil.

Letak dari kelas konkret ‘Mobil’ berada di dalam blok program Python. Sebagai contoh, kita dapat meletakkannya di dalam sebuah file Python dengan nama ‘mobil.py’. Jika kita ingin menggunakannya di dalam file Python lain, kita dapat mengimpor modul ‘mobil’ dan membuat objek dari kelas ‘Mobil’. Contohnya:

```
from mobil import Mobil

mobil_saya = Mobil("Merah", 2022, "B 1234 XYZ")
mobil_saya.hidupkan_mesin()
mobil_saya.mengemudi()
mobil_saya.matikan_mesin()
```

4. Metaclass

a) Pengertian Metaclass

Di Python, metaclass adalah kelas yang digunakan untuk membuat kelas lainnya. Dalam bahasa pemrograman Python, setiap kelas sebenarnya dibuat dari sebuah metaclass. Metaclass

memungkinkan kita untuk mengontrol cara pembuatan kelas, seperti bagaimana atribut dan metode didefinisikan, bagaimana objek dari kelas tersebut diinisialisasi, dan sebagainya.

b) Implementasi Metaclass

Kita dapat menggunakan metaclass di Python ketika kita ingin melakukan modifikasi atau pengaturan yang spesifik pada saat pembuatan kelas. Berikut adalah beberapa contoh situasi di mana kita mungkin perlu menggunakan metaclass:

1. Ketika kita ingin menambahkan atau mengubah atribut atau metode pada kelas sebelum kelas itu dibuat.
2. Ketika kita ingin memodifikasi perilaku konstruktor kelas.
3. Ketika kita ingin melakukan validasi pada kelas sebelum kelas itu dibuat.
4. Ketika kita ingin membuat kelas baru secara dinamis dengan menggabungkan fitur-fitur dari beberapa kelas yang sudah ada.
5. Ketika kita ingin mengotomatisasi pembuatan kode berulang dengan menggunakan metaprogramming.

Namun, penggunaan metaclass juga harus dipertimbangkan secara hati-hati, karena dapat membuat kode menjadi lebih kompleks dan sulit untuk dipahami. Oleh karena itu, metaclass sebaiknya hanya digunakan dalam situasi-situasi yang memang membutuhkan fitur-fitur khusus yang tidak dapat dicapai dengan cara-cara standar yang tersedia di Python.

Berikut adalah contoh sederhana penggunaan metaclass dalam Python:

```python
class MetaKelas(type):
    def __new__(cls, name, bases, attrs):
        attrs["jumlah_instance"] = 0
        return super().__new__(cls, name, bases,
attrs)
class KelasContoh(metaclass=MetaKelas):
    def __init__(self):
        self.increment_instance()

    @classmethod
    def increment_instance(cls):
        cls.jumlah_instance += 1

    @classmethod
    def get_jumlah_instance(cls):
        return cls.jumlah_instance

class ObjekContoh(KelasContoh):
    pass

objek1 = ObjekContoh()
objek2 = ObjekContoh()

print(ObjekContoh.get_jumlah_instance()) # Output: 2
```

Pada contoh di atas, kita membuat sebuah metaclass ‘MetaKelas’ yang akan digunakan untuk membuat kelas lainnya. Metaclass ini akan menambahkan atribut ‘jumlah_instance’ ke dalam kelas yang dibuat. Kemudian, kita membuat kelas ‘KelasContoh’ dengan menggunakan metaclass ‘MetaKelas’. Kelas ‘KelasContoh’ memiliki metode ‘increment_instance’ dan ‘get_jumlah_instance’ yang digunakan untuk mengakses atribut ‘jumlah_instance’.

Terakhir, kita membuat kelas ‘ObjekContoh’ yang merupakan turunan dari ‘KelasContoh’. Ketika objek ‘objek1’ dan ‘objek2’ dibuat, metode ‘__init__’ pada ‘KelasContoh’ akan dipanggil dan akan memanggil metode ‘increment_instance’ untuk menambahkan jumlah instance kelas tersebut.

Dalam contoh di atas, metaclass ‘MetaKelas’ digunakan untuk menambahkan atribut ‘jumlah_instance’ ke dalam kelas yang dibuat. Dengan menggunakan metaclass ini, kita dapat memastikan bahwa setiap kelas yang dibuat dari ‘MetaKelas’ akan memiliki atribut ‘jumlah_instance’.

5. Kesimpulan

Dalam bahasa pemrograman Python, terdapat beberapa konsep yang penting untuk dipahami, seperti kelas abstrak, interface, kelas konkret, dan metaclass.

Kelas abstrak adalah kelas yang tidak dapat diinstansiasi dan hanya digunakan sebagai kerangka kerja untuk kelas-kelas turunannya.

Interface adalah kumpulan metode yang dideklarasikan tetapi tidak diimplementasikan. Tujuan dari interface adalah untuk memberikan sebuah kontrak atau persyaratan yang harus dipenuhi oleh kelas-kelas yang mengimplementasikannya.

Kelas konkret adalah kelas yang dapat diinstansiasi dan memiliki implementasi lengkap dari metode-metodenya.

Metaclass adalah kelas khusus yang digunakan untuk mengontrol pembuatan kelas di Python. Metaclass dapat digunakan untuk melakukan modifikasi atau pengaturan yang spesifik pada saat pembuatan kelas.

Pemilihan konsep yang tepat dapat membantu kita dalam mengembangkan kode yang lebih mudah dipelihara, diuji, dan diperluas. Namun, penggunaan konsep tersebut harus dilakukan dengan hati-hati dan hanya jika memang membutuhkan fitur-fitur khusus yang tidak dapat dicapai dengan cara-cara standar yang tersedia di Python.

References

https://www.geeksforgeeks.org/abstract-classes-in-python/

https://www.petanikode.com/java-oop-abstract/

https://www.geeksforgeeks.org/python-interface-module/

https://pythonguides.com/python-interface/

https://learnpython101.com/interfaces

https://www.scaler.com/topics/interface-in-python/

https://qastack.id/programming/100003/what-are-metaclasses-in-python

https://www.geeksforgeeks.org/python-metaclasses/