Nama : Richard Arya Winarta

NIM : 121140035

Kelas : RB

Tugas 5 Praktikum Pemrograman Berbasis Objek

Abstraksi

Abstraksi dalam konsep PBO adalah model konsep yang dibuat untuk memperlihatkan atribut esensial yang penting untuk diketahui, dan menyembunyikan detail-detail kompleks yang tidak penting untuk mengurangi kompleksitas.

Ilustrasi : Jika kita ingin memodelkan sebuah mobil dalam program Python, kita dapat membuat kelas mobil yang memiliki atribut seperti warna, model, dan tahun, serta metode seperti mengemudi, mengisi bahan bakar, dan berhenti. Dalam hal ini, kelas mobil adalah abstraksi dari objek nyata dan kita dapat memanipulasi objek mobil dalam program tanpa harus memikirkan detail kompleksitas bagaimana kelas mobil bekerja.

Contoh :

```
class Mobil:
    def __init__(self, warna, model, tahun, km):
        self.warna = warna
        self.model = model
        self.tahun = tahun
        self.km = km

    def mengemudi(self):
        self.km = self.km + 40
        return self.km

    def isi_bahan_bakar(self):
        print("Mobil dengan model", self.model, "sedang diisi bahan bakar")

    def berhenti(self):
        print("Mobil dengan model", self.model, "telah berhenti")

honda = Mobil('merah', 'HRV', 2019, 0)
honda.mengemudi()
```

Penjelasan : Dalam kode ini, proses pembuatan kelas mobil merupakan abstraksi dari objek honda. Dengan menggunakan abstraksi, kita dapat memanipulasi objek honda tanpa harus tahu bagaimana mobil bekerja. Ketika kita memanggil method mengemudi, kita hanya mengetahui bahwa objek honda akan melaksanakan perintah mengemudi tanpa melihat proses yang terjadi di belakang layar.

Enkapsulasi

Enkapsulasi dalam konsep PBO adalah model konsep yang dibuat untuk melindungi, menyembunyikan data atau informasi dari akses langsung di luar program atau kelas. Untuk membatasi akses terhadap property / method suatu kelas, terdapat 3 access modifier yaitu :

- Public access modifier (default) : atribut / method dalam suatu kelas dapat diakses secara bebas dari luar kelas.
- Protected access modifier : atribut / method dalam suatu kelas hanya dapat diakses oleh kelas turunannnya (cara mendeklarasikannya dengan menambahkan 1 underscore (_) sebelum nama variabel / method).
- Private access modifier : atribut / method dalam suatu kelas hanya dapat diakses oleh kelas itu sendiri (cara mendeklarasikannya dengan menambahkan 2 underscore (_) sebelum nama variabel / method).

Catatan : Python tidak memiliki mekanisme apa pun yang secara efektif untuk membatasi akses ke atribut ataupun metode private

- Akses langsung atribut / method private secara paksa
  Syntax : object._class__attribute atau object._class__method
- Setter dan Getter, setter adalah sebuah method yang digunakan untuk mengatur sebuah property yang ada di dalam suatu kelas/objek. Sedangkan Getter adalah sebuah method yang digunakan untuk mengambil nilai dari suatu property. Setter dan Getter dapat menggunakan atau tidak menggunakan decorator pada saat pembuatan.

Contoh :

```
class Pekerja:

    def __init__(self, nama, umur, gaji):
        self.nama = nama # public attribute
        self._umur = umur # protected attribute
        self.__gaji = gaji # private attribute

    def tampilkan_nama(self): # public method
        print(self.nama)

    def _tampilkan_umur(self): # protected method
        print(self._umur)

    def __tampilkan_gaji(self): # private method
        print(self.__gaji)


    def get_gaji(self): # getter method
        return self.__gaji

    def set_gaji(self, gaji_baru): # setter method
        self.__gaji = gaji_baru

    @property # decorator
    def gaji(self): # bentuk getter dengan decorator
        return self.__gaji

    @gaji.setter # decorator
    def gaji(self, gaji_baru): # bentuk setter dengan decorator
        self.__gaji = gaji_baru
```

Penjelasan :

```
print(pekerja1.nama) # public attribute bisa diakses di luar kelas
print(pekerja1._umur) # protected attribute bisa diakses di luar kelas
#print(pekerja1.__gaji) # private attribute tidak bisa diakses di luar kelas
print(pekerja1._Pekerja__gaji) # private attribute bisa diakses secara paksa dengan menambahkan _class
print(pekerja1.get_gaji()) # private attribute bisa diakses di luar kelas menggunakan getter method (public method)
print()

pekerja1.tampilkan_nama() # public method bisa di akses di luar kelas
pekerja1._tampilkan_umur() # protected method bisa di akses di luar kelas
#pekerja1.__tampilkan_gaji() # private method tidak bisa di akses di luar kelas
pekerja1._Pekerja__tampilkan_gaji() # private method bisa diakses secara paksa dengan menambahkan _class
pekerja1.tampilkan_gaji_public() # private method dapat dibungkus dengan public method agar bisa digunakan di luar kelas
print()
```

Inheritance

Inheritance dalam konsep PBO adalah model konsep dimana kita dapat membuat kelas baru yang memanfaatkan dan berbagi atribut atau metode yang ada pada kelas yang lain.

Sintaks dasar Inheritance :

```
class Parent:
    pass

class Child(Parent):
    pass
```

Multiple inheritance : konsep inheritance yang mendukung pewarisan kelas lebih dari 1. Dalam multiple inheritance, method yang dicari pertama kali adalah method di kelas objek. Apabila tidak ditemukan, maka akan berlanjut ke superclass dengan urutan pencarian kelas paling kiri hingga kelas paling kanan. Sintaks dasar Multiple Inheritance :

```
class Parent_1:
    pass

class Parent_2:
    pass

class Child(Parent_1, Parent_2):
    pass
```

Inheritance identik : pewarisan yang menambahkan constructor pada class child sehingga class child memiliki constructornya sendiri tanpa menghilangkan constructor pada class parentnya. Constructor yang digunakan adalah super(). Sintaks dasar Inheritance identik :

```
class Parent:
    def __init__(self):
        pass

class Child(Parent):
    def __init__(self):
        super().__init__()
```

Contoh Inheritance :

```
class Hewan:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur

class Anjing(Hewan):
    def suara(self):
        return "Guk guk"

class Kucing(Hewan):
    def suara(self):
        return "Meong"
```

Penjelasan :

```
# Membuat objek Anjing dan Kucing
anjing = Anjing("Spike", 2)
kucing = Kucing("Tom", 1)

# Mengakses atribut dan metode dari kelas parent melalui objek yang dibentuk dari kelas child
print(anjing.nama)     # Output: Spike
print(kucing.umur)     # Output: 1
print(anjing.suara())  # Output: Guk guk
print(kucing.suara())  # Output: Meong
```

Terdapat base class “Hewan” dan terdapat class “Anjing” dan “Kucing” yang merupakan turunan dari class “Hewan”. ini berarti class “Anjing” dan “Kucing” memiliki atribut dan method yang sama dengan class “Hewan”. Dan objek “Anjing” dan “Kucing” dapat menggantikan objek “Hewan” dalam pembentukan objek.

Contoh Multiple Inheritance:

```
class Parent1:
    def method1(self):
        print("Ini parent 1")

class Parent2:
    def method2(self):
        print("Ini parent 2")

class Child(Parent1, Parent2):
    def method3(self):
        print("Ini child")
```

Penjelasan :

```
c = Child()
c.method1() # output = "Ini parent 1"
c.method2() # output = "Ini parent 2"
c.method3() # output = "Ini child"
```

Karena kelas “Child” mewarisi method dan atribut kelas “Parent1” dan “Parent2”, maka objek “c” dapat mengakses seluruh method di kelas yang diwarisi ke kelas “Child”.

Contoh Inheritance identik :

```
class Makanan:
    def __init__(self, nama):
        self.nama = nama

    def makan(self):
        print("Hewan makan apa ya")

class Kambing(Makanan):
    def __init__(self, nama, umur):
        self.umur = umur
        super().__init__(nama)

    def makan(self):
        print(self.nama)
        print("Kambing makan rumput")
```

Penjelasan : karena kelas “Makanan” telah memiliki konstruktor berupa “nama” maka agar kelas turunan “Kambing” bisa memiliki konstruktor sendiri yaitu “umur” diperlukan sebuah konstruktor khusus yaitu super()

Polimorfisme

Polimorfisme dalam konsep PBO adalah model konsep yang dibuat untuk memungkinkannya sebuah interface identik namun memiliki proses eksekusi yang berbeda. Polimorfisme memungkinkan untuk mengambil tindakan yang sama pada objek yang berbeda. Polimorfisme dalam konsep PBO dikenal sebagai method overloading, namun bahasa python tidak menangani kasus overloading secara khusus karena python bersifat dynamic typing.

Konsep Overriding dan overloading

- Overriding : menimpa suatu metode yang ada pada parent class dengan mendefinisikan kembali method dengan nama yang sama pada child class
- Overloading : sebuah method atau fungsi dapat memiliki beberapa definisi dengan nama yang sama, tetapi memiliki parameter yang berbeda.

Contoh Polimorfisme Overriding :

```
class Hewan:
    def bersuara(self):
        print("Hewan bersuara ...")

class Cat(Hewan):
    def bersuara(self):
        print("Meow")

class Dog(Hewan):
    def bersuara(self):
        print("Bark")
```

Penjelasan : Kelas “Cat” dan “Dog” adalah kelas turunan dari kelas “Hewan”. Kedua kelas tersebut meng-override method “bersuara” dari kelas “Hewan” sehingga masing-masing memiliki implementasi yang berbeda-beda.

```
animals = [Cat(), Dog()]

animals[0].bersuara() # Output "Meow"
animals[1].bersuara() # Output "Bark"
```

Contoh Polimorfisme Overloading:

```
class BangunDatar:
    def hitung_luas(self):
        pass

class Rectangle(BangunDatar):
    def hitung_luas(self, panjang, lebar):
        return panjang * lebar

class Circle(BangunDatar):
    def hitung_luas(self, jari_jari):
        return 3.14 * jari_jari * jari_jari

class Triangle(BangunDatar):
    def hitung_luas(self, alas, tinggi):
        return 0.5 * alas * tinggi
```

Penjelasan : Kelas “Rectangle”, “Circle”, dan “Triangle” adalah kelas turunan dari kelas “BangunDatar”. Ketiga kelas tersebut memiliki method yang sama yaitu hitung_luas namun dengan parameter dan implementasi masing-masing.

```
r = Rectangle()
print("Luas persegi panjang: ", r.hitung_luas(5, 10)) # Output 50

c = Circle()
print("Luas lingkaran: ", c.hitung_luas(7)) # Output 153.86

t = Triangle()
print("Luas segitiga: ", t.hitung_luas(6, 8)) # Output 24
```