RESPONSI
PEMROGRAMAN BERORIENTASI OBJEK

YOGI PRANOTO

5230411284

PROGRAM STUDI INFORMATIKA

FAKULTAS SAINS & TEKNOLOGI

UNIVERSITAS TEKNOLOGI YOGYAKARTA

YOGYAKARTA

2024

SOAL :

1. Jelaskan perbedaan use case diagram dengan class diagram?
2. Jelaskan jenis-jenis dependasi?
3. Apa perbedaan pemrograman terstruktur dengan berorientasi objek, jelaskan?
4. Jelaskan konsep objek dan beri contohnya?
5. Jelaskan jenis-jenis acces modifier beri contohnya dalam baris pemrograman?
6. Gambarkan contoh pewarisan dalam diagram class?

JAWAB :

1. Perbedaannya yaitu :

    - Use Case Diagram
      Use case diagram berfungsi untuk menggambarkan interaksi antara pengguna (dikenal sebagai aktor) dan sistem. Diagram ini memberikan gambaran tentang fungsionalitas yang ditawarkan oleh sistem dari sudut pandang pengguna. Elemen utama dalam use case diagram meliputi:

        - Aktor: Ini adalah entitas eksternal yang berinteraksi dengan sistem, seperti pengguna akhir atau sistem lain.
        - Use Case: Merupakan representasi dari fungsi atau layanan yang disediakan oleh sistem untuk aktor.
        - Relasi: Menggambarkan hubungan antara aktor dan use case, termasuk asosiasi, inklusi, dan ekstensi.

      Fokus utama dari use case diagram adalah untuk menunjukkan apa yang dilakukan oleh sistem dan bagaimana pengguna berinteraksi dengan fungsionalitas tersebut. Dengan kata lain, diagram ini membantu tim pengembang memahami kebutuhan pengguna dan bagaimana sistem akan memenuhi kebutuhan tersebut.

    - Class Diagram
      Di sisi lain, class diagram berfokus pada struktur statis dari sistem. Diagram ini menggambarkan kelas-kelas yang ada dalam sistem, termasuk atribut (data) dan metode (fungsi) yang dimiliki oleh setiap kelas. Elemen-elemen utama dalam class diagram meliputi:
        - Kelas: Representasi dari objek dalam sistem, yang mencakup informasi tentang data dan perilaku.
        - Relasi: Menggambarkan hubungan antar kelas, seperti asosiasi, pewarisan, serta agregasi atau komposisi.

      Fokus dari class diagram adalah untuk menunjukkan bagaimana data dan fungsionalitas diorganisir di dalam sistem. Dengan diagram ini, pengembang dapat melihat struktur internal sistem dan bagaimana berbagai elemen saling berinteraksi**.**

2. Dependensi dalam konteks pengembangan perangkat lunak merujuk pada hubungan antara berbagai komponen atau elemen dalam sistem. Memahami jenis-jenis dependensi sangat penting untuk merancang sistem yang efisien dan mudah dikelola. Berikut adalah beberapa jenis dependensi yang umum ditemukan:

    a) Dependensi Fungsional

    Dependensi fungsional terjadi ketika satu komponen atau modul dalam sistem bergantung pada fungsi atau layanan yang disediakan oleh komponen lain. Misalnya, jika Modul A memanggil fungsi dari Modul B untuk menjalankan tugas tertentu, maka Modul A memiliki dependensi fungsional terhadap Modul B.

    b) Dependensi Data

    Dependensi data terjadi ketika satu komponen memerlukan data yang dihasilkan atau dikelola oleh komponen lain. Contohnya, jika Modul C membutuhkan akses ke database yang dikelola oleh Modul D, maka ada dependensi data antara Modul C dan Modul D.

    c) Dependensi Temporal

    Dependensi temporal muncul ketika suatu komponen harus dieksekusi sebelum komponen lain. Dalam hal ini, urutan eksekusi menjadi penting. Misalnya, jika Modul E harus menyelesaikan prosesnya sebelum Modul F dapat memulai, maka ada dependensi temporal antara keduanya.

    d) Dependensi Konfigurasi

    Dependensi konfigurasi terjadi ketika sebuah komponen memerlukan pengaturan atau konfigurasi tertentu yang disediakan oleh komponen lain. Misalnya, jika Modul G memerlukan file konfigurasi yang dihasilkan oleh Modul H, maka ada dependensi konfigurasi antara keduanya.

    e) Dependensi Versi

    Dependensi versi muncul ketika suatu komponen bergantung pada versi tertentu dari komponen lain. Hal ini sering terjadi dalam pengelolaan paket perangkat lunak, di mana suatu aplikasi memerlukan versi spesifik dari library atau framework untuk berfungsi dengan baik.

    f) Dependensi Lingkungan

    Dependensi lingkungan terjadi ketika suatu komponen bergantung pada kondisi lingkungan tertentu, seperti sistem operasi, perangkat keras, atau pengaturan jaringan. Misalnya, jika sebuah aplikasi hanya dapat berjalan di sistem operasi tertentu, maka ada dependensi lingkungan yang harus diperhatikan.

g) Dependensi Antarmuka

Dependensi antarmuka terjadi ketika satu komponen bergantung pada antarmuka yang disediakan oleh komponen lain. Dalam konteks pemrograman berorientasi objek, ini sering kali melibatkan implementasi antarmuka atau kelas abstrak yang harus diikuti oleh kelas lain.

3. Perbedaannya adalah:

a. Pemrograman Terstruktur

- Definisi: Paradigma yang menekankan penggunaan struktur kontrol (seperti urutan, percabangan, dan perulangan) untuk mengorganisir kode.
- Pendekatan: Memecah program menjadi fungsi atau prosedur yang terpisah. Data dan fungsi biasanya tidak terikat.
- Contoh Bahasa: C, Pascal, Fortran.
- Kelemahan: Sulit dikelola pada program besar karena pemisahan antara data dan fungsi.

b. Pemrograman Berorientasi Objek (OOP)
- Definisi: Paradigma yang menggunakan objek sebagai unit dasar, menggabungkan data dan perilaku.
- Pendekatan: Menggunakan kelas dan objek, mendukung konsep pewarisan dan polimorfisme.
- Contoh Bahasa: Java, C++, Python, Ruby.
- Keuntungan: Memudahkan pengembangan yang modular, lebih mudah dipelihara dan diperluas.

4. Komponen Objek
- Atribut:
  Atribut adalah karakteristik atau properti dari objek. Ini bisa berupa data yang menyimpan informasi tentang objek tersebut.
  Contoh: Dalam objek "Mobil", atributnya bisa mencakup warna, merk, model, dan tahun.
- Metode:
  Metode adalah fungsi atau perilaku yang dapat dilakukan oleh objek. Metode mendefinisikan apa yang dapat dilakukan oleh objek tersebut.
  Contoh: Dalam objek "Mobil", metode bisa mencakup berjalan(), berhenti(), dan berbelok().

- Contoh objek

  - Kelas mobil (Blueprint untuk objek)

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

    def berjalan(self):
```

```
        print(f"{self.merk}    {self.model}    sedang
berjalan.")

    def berhenti(self):
        print(f"{self.merk} {self.model} berhenti.")

    def berbelok(self, arah):
        print(f"{self.merk} {self.model} berbelok ke
{arah}.")
```

- Objek dari kelas mobil

```
# Membuat objek dari kelas Mobil
mobil_saya = Mobil("Toyota", "Camry", 2020, "Merah")


# Mengakses atribut
print(mobil_saya.merk)  # Output: Toyota
print(mobil_saya.warna)  # Output: Merah


# Memanggil metode
mobil_saya.berjalan()  # Output: Toyota Camry sedang
berjalan.
mobil_saya.berhenti()      #  Output:  Toyota  Camry
berhenti.
mobil_saya.berbelok("kiri")  # Output: Toyota Camry
berbelok ke kiri.
```

Kesimpulan

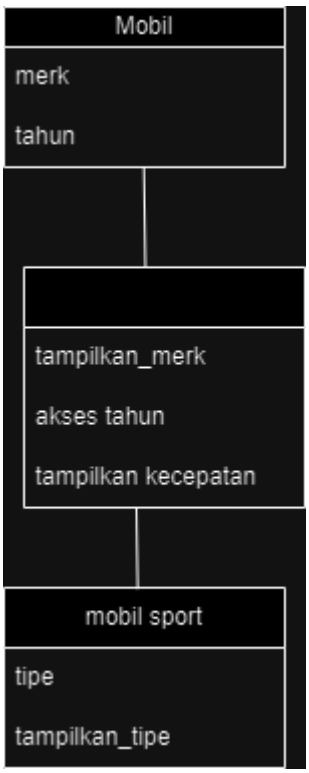
Dalam contoh di atas, "Mobil" adalah kelas yang mendefinisikan atribut dan metode. **mobil_saya** adalah objek yang merupakan instansi dari kelas "Mobil". Dengan menggunakan konsep objek, kita dapat mengorganisir kode dengan cara yang lebih terstruktur dan mudah dipahami, serta memungkinkan penggunaan kembali kode melalui pewarisan dan polimorfisme .

5. Penjelasan dan Perbedaan beserta contohnya:
   - Public: Atribut merk dan metode tampilkan_merk dapat diakses dari mana saja.
   - Private: Atribut __tahun hanya dapat diakses dari dalam kelas.
   - Protected: Atribut _kecepatan dan metode _tampilkan_kecepatan dapat diakses dari dalam kelas dan oleh subclass (meskipun tidak ada subclass di sini).

- Contoh implementasi

```python
class Mobil:
    def __init__(self, merk, tahun):
        self.merk = merk          # Atribut public
        self.__tahun = tahun      # Atribut private
        self._kecepatan = 0       # Atribut protected

    def tampilkan_merk(self):     # Metode public
        print(f"Merk mobil: {self.merk}")

    def _tampilkan_kecepatan(self):  # Metode protected
        print(f"Kecepatan mobil: {self._kecepatan} km/jam")

    def akses_tahun(self):        # Metode public
        print(f"Tahun mobil: {self.__tahun}")  # Mengakses metode private

# Penggunaan kelas Mobil
mobil_saya = Mobil("Toyota", 2020)
mobil_saya.tampilkan_merk()       # Output: Merk mobil: Toyota
mobil_saya.akses_tahun()          # Output: Tahun mobil: 2020

# Mengakses atribut protected
mobil_saya._kecepatan = 100
mobil_saya._tampilkan_kecepatan() # Output: Kecepatan mobil: 100 km/jam
```

6. Gambar diagram kelas



←Kelas induk(Super class)

←Attribut

←Metode public
←Metode public

←Metode Protected

←Kelas anak(Subclass)

←Attribut tambahan
←metode public

- Penjelasan Diagram:
  1. Kelas Induk (Mobil):
     - Memiliki atribut merk (public) dan tahun (private).
     - Memilikmetode tampilkan_merk() dan akses_tahun() (public) serta _tampilkan_kecepatan() (protected).

2. Kelas Anak (MobilSport):
    - Mewarisi semua atribut dan metode dari kelas Mobil.
    - Menambahkan atribut baru tipe (public).
    - Menambahkan metode baru tampilkan_tipe() (public).
      Konsep Pewarisan
    - Kelas MobilSport mewarisi semua atribut dan metode dari kelas Mobil, yang memungkinkan MobilSport untuk menggunakan dan mengakses metode serta atribut yang ada di kelas Mobil.
    - Kelas MobilSport dapat memperluas fungsionalitas dari kelas Mobil dengan menambahkan atribut dan metode baru.
      Diagram ini memberikan gambaran visual tentang bagaimana pewarisan bekerja dalam pemrograman berorientasi objek. Jika Anda menggunakan alat pemodelan seperti UML (Unified Modeling Language), diagram ini dapat dibuat dengan lebih terstruktur dan formal.