BAB 6

SOLID PRINCIPLE

6.1 Tujuan

1. Dapat memahami konsep SOLID pada Pemrograman Berorientasi Objek
2. Dapat mengimplementasikan konsep SOLID menggunakan Python

6.2 Pengantar

6.2.1 Pengantar SOLID

Konsep SOLID adalah sebuah akronim yang mencakup lima prinsip dasar dalam pemrograman berorientasi objek dan desain. SOLID berasal dari Single Responsibility Principle (SRP), Single Responsibility Principle (SRP), Single Responsibility Principle (SRP), Interface Segregation Principle (ISP) dan Interface Segregation Principle (ISP). Prinsip-prinsip ini ditujukan untuk membuat sistem perangkat lunak yang mudah dipahami, mudah dikelola, dan dapat diperluas seiring berjalannya waktu. Secara umum, tujuan dari prinsip SOLID adalah untuk mengurangi ketergantungan dan kopling dalam kode, membuat kode lebih modular, dan meningkatkan fleksibilitas dan pemanfaatan kembali kode.

Prinsip SOLID bukanlah aturan yang harus selalu diikuti, tetapi lebih ke arah panduan yang dirancang untuk membantu pengembang membuat perangkat lunak yang lebih mudah dikelola dan dipahami. Mengikuti prinsip-prinsip ini biasanya menghasilkan kode yang lebih bersih, lebih mudah diperluas, dan lebih mudah diuji.

Namun, penting untuk diingat bahwa tidak ada aturan yang absolut dalam pemrograman, dan ada situasi di mana mungkin lebih masuk akal untuk melanggar prinsip-prinsip ini. Misalnya, dalam kasus di mana pengembangan cepat lebih penting daripada pemeliharaan jangka panjang, atau di mana kode yang ditulis tidak mungkin perlu diperluas atau dimodifikasi di masa depan, mungkin tidak perlu untuk menerapkan prinsip SOLID secara ketat.

6.2.2 5 Prinsip SOLID

6.2.2.1 Single Responsibility Principle (SRP)

Setiap class harus memiliki tanggung jawab tunggal. Dalam kata lain, setiap class harus memiliki satu alasan untuk berubah. Tujuan dari prinsip ini adalah untuk mengurangi kompleksitas sistem dengan memisahkan fungsionalitas sehingga perubahan dalam

satu bagian sistem tidak mempengaruhi bagian lainnya. Selain itu juga untuk meminimalkan dampak perubahan pada kode. Jika sebuah class atau modul hanya memiliki satu tanggung jawab, maka ada lebih sedikit risiko bahwa perubahan pada bagian lain dari sistem akan mempengaruhi class atau modul tersebut.

Bayangkan seseorang memiliki sebuah restoran dengan satu karyawan yang melakukan segalanya - memasak, membersihkan, melayani pelanggan, mengurus keuangan, dan sebagainya. Jika karyawan tersebut sakit, seluruh operasi restoran akan terganggu. Dengan menerapkan SRP, maka akan ada orang yang berbeda untuk setiap tugas (seorang koki, pelayan, pembersih, dll.), sehingga jika satu orang tidak bisa bekerja, itu tidak akan mengganggu seluruh operasi.

6.2.2.2 Open/Closed Principle (OCP)

"Buka untuk ekstensi, tutup untuk modifikasi". Artinya, entitas perangkat lunak (kelas, modul, fungsi, dll.) harus dibuka untuk ekstensi tetapi ditutup untuk modifikasi. Ini berarti bahwa kita harus dapat menambahkan fitur atau perilaku baru ke entitas tersebut tanpa mengubah kode yang ada. Tujuannya adalah untuk membuat kode yang lebih mudah diperluas. Dengan membuat entitas yang "terbuka untuk ekstensi tetapi tertutup untuk modifikasi", kita dapat menambahkan fitur atau perilaku baru tanpa mengubah kode yang ada.

Bayangkan kita memiliki sebuah mobil yang tidak dapat dimodifikasi. Jika kita ingin menambahkan fitur baru seperti sistem navigasi, kita harus merusak bagian interior mobil untuk memasangnya. Mobil ideal adalah mobil yang memiliki slot yang dapat disesuaikan untuk memasukkan sistem navigasi baru tanpa harus merusak bagian lain dari mobil.

6.2.2.3 Liskov Substitution Principle (LSP)

Subtypes harus dapat menggantikan tipe dasarnya. Dalam kata lain, jika sebuah program dirancang untuk menggunakan objek tipe tertentu, maka objek dari subtype tersebut harus dapat bekerja dengan program tersebut tanpa perlu modifikasi apa pun pada program tersebut. Tujuannya adalah untuk memastikan bahwa penggantian

tipe dasar dengan subtype-nya tidak akan mempengaruhi perilaku program. Hal ini membuat kode lebih mudah dipahami dan dikelola, karena kita dapat mempercayai bahwa semua objek yang mengimplementasikan tipe atau antarmuka tertentu akan berperilaku dengan cara yang sama.

Bayangkan memiliki burung yang bisa terbang. Kita merancang kandang dengan tinggi tertentu karena burung tersebut bisa terbang. Sekarang, kita memutuskan untuk menambahkan pinguin (yang juga merupakan burung) ke dalam kandang tersebut. Pinguin tidak bisa terbang sehingga kandang tersebut tidak cocok untuk pinguin. Dengan kata lain, meskipun pinguin adalah burung, ia tidak bisa digunakan sebagai pengganti burung yang bisa terbang.

6.2.2.4 Interface Segregation Principle (ISP)

Klien tidak boleh dipaksa untuk bergantung pada antarmuka yang mereka tidak gunakan. Class tidak harus mengimplementasikan metode yang tidak mereka butuhkan. Sebaliknya, jika suatu kelas membutuhkan hanya sebagian dari metode dalam suatu antarmuka, maka antarmuka tersebut sebaiknya dibagi menjadi antarmuka yang lebih kecil yang lebih spesifik.

Tujuannya adalah untuk mengurangi ketergantungan yang tidak perlu antara modul. Dengan memastikan bahwa klien hanya bergantung pada antarmuka yang mereka butuhkan, kita membuat sistem yang lebih modular dan lebih mudah dikelola.

Bayangkan remote kontrol TV yang juga memiliki tombol untuk mengontrol AC, mesin cuci, dan lemari es. Meskipun ini mungkin terdengar efisien, ini bisa menjadi masalah jika kita hanya ingin mengontrol TV. Dengan memisahkan remote menjadi lebih spesifik (satu untuk TV, satu untuk AC, dll.), Kita membuat interaksi menjadi lebih mudah dan intuitif.

6.2.2.5 Dependency Inversion Principle (DIP)

Class tingkat tinggi tidak harus bergantung pada kelas tingkat rendah. Kedua jenis class tersebut harus bergantung pada abstraksi. Dengan demikian detail implementasi harus bergantung pada kebijakan dan bukan sebaliknya.Tujuannya adalah untuk

mengurangi ketergantungan antara modul tingkat tinggi dan modul tingkat rendah, yang membuat sistem lebih mudah dikelola dan diperluas.

Bayangkan lampu yang bisa dinyalakan dengan menekan saklar. Jika lampu tersebut dihubungkan langsung ke saklar, maka kita tidak akan bisa mengubah cara menghidupkannya tanpa merusak koneksi tersebut. Namun, jika lampu dan saklar sama-sama tergantung pada sistem listrik (abstraksi), kita bisa dengan mudah mengganti saklar dengan pengendali suara atau sensor gerak tanpa merusak lampu atau membutuhkan perubahan signifikan pada sistem.

6.3 Kegiatan Praktikum

6.3.1 Kegiatan 1 : Single Responsibility Principle (SRP)

1. Misalkan kita memiliki kelas UserManager yang bertanggung jawab untuk mengelola data pengguna dan juga mencetak laporan pengguna. Ini melanggar SRP karena UserManager memiliki lebih dari satu tanggung jawab.

```
1. class UserManager:
2.     def __init__(self):
3.         self.users = []
4.
5.     def add_user(self, user):
6.         self.users.append(user)
7.
8.     def print_user_report(self):
9.         for user in self.users:
10.            print(f'User: {user.name}, Email: {user.email}')
```

2. Untuk mematuhi SRP, kita bisa memindahkan tanggung jawab mencetak laporan ke kelas lain, seperti UserReport.

```
1. class User:
2.     def __init__(self, name, email):
3.         self.name = name
4.         self.email = email
5.
6.
7. class UserManager:
8.     def __init__(self):
```

```
        self.users = []

    def add_user(self, user):
        self.users.append(user)


class UserReport:
    @staticmethod
    def print_user_report(users):
        for user in users:
            print(f'User: {user.name}, Email: {user.email}')


# Membuat objek User
user1 = User('Alice', 'alice@example.com')
user2 = User('Bob', 'bob@example.com')

# Membuat objek UserManager dan menambahkan User
user_manager = UserManager()
user_manager.add_user(user1)
user_manager.add_user(user2)

# Membuat objek UserReport dan mencetak laporan
user_report = UserReport()
user_report.print_user_report(user_manager.users)
```

6.3.2 Kegiatan 2 : Open/Closed Principle (OCP)

1. Misalkan kita memiliki program yang mencetak pesan selamat datang dalam berbagai bahasa. Dalam contoh ini, jika kita ingin menambahkan bahasa baru, kita harus memodifikasi metode greet pada kelas Greeter. Ini melanggar Prinsip Open/Closed Principle (OCP) karena kita harus memodifikasi kelas yang ada untuk menambahkan fungsi baru.

```
1. class Greeter:
2.     def __init__(self, language):
3.         self.language = language
4.
5.     def greet(self):
6.         if self.language == 'english':
7.             return 'Hello!'
8.         elif self.language == 'spanish':
9.             return '¡Hola!'
10.        elif self.language == 'french':
11.            return 'Bonjour!'
12.        else:
13.            return 'Language not supported.'
```

2. Untuk mematuhi Prinsip Open/Closed Principle (OCP), kita bisa merancang kelas Greeter agar dapat menerima objek "Greeting" yang dapat menyapa dalam bahasa apa pun, seperti ini:

```
1. class Greeter:
2.     def __init__(self, greeter):
3.         self.greeter = greeter
4.
5.     def greet(self):
6.         return self.greeter.greet()
7.
8.
9. class EnglishGreeter:
10.    def greet(self):
11.        return 'Hello!'
12.
13.
14.class SpanishGreeter:
15.    def greet(self):
16.        return '¡Hola!'
17.
18.
```

```
class FrenchGreeter:
    def greet(self):
        return 'Bonjour!'


# Membuat objek greeter dalam berbagai bahasa
english_greeter = EnglishGreeter()
spanish_greeter = SpanishGreeter()
french_greeter = FrenchGreeter()

# Membuat objek Greeter dan menggreet dalam berbagai bahasa
greeter = Greeter(english_greeter)
print(greeter.greet())  # Output: Hello!

greeter = Greeter(spanish_greeter)
print(greeter.greet())  # Output: ¡Hola!

greeter = Greeter(french_greeter)
print(greeter.greet())  # Output: Bonjour!
```

6.3.3 Kegiatan 3 : Open/Closed Principle (OCP)

1. Perhatikan contoh kode berikut, Ostrich adalah turunan dari Bird, namun Ostrich tidak dapat terbang seperti Bird lainnya. Ini melanggar Liskov Substitution Principle karena kita tidak bisa menggantikan Bird dengan Ostrich dalam konteks terbang.

```
class Bird:
    def fly(self):
        return "I can fly!"

class Duck(Bird):
    pass

class Ostrich(Bird):
    def fly(self):
        return "I can't fly!"
```

2. Untuk mematuhi Liskov Substitution Principle, kita bisa mendefinisikan ulang hirarki kelas kita. Kita bisa membuat kelas FlyingBird dan NonFlyingBird yang merupakan turunan dari Bird, dan memindahkan metode fly ke FlyingBird.

```
1. class Bird:
2.     def __init__(self, name):
3.         self.name = name
4.
5.     def introduce(self):
6.         return f"Hello, I'm a {self.name}."
7.
8.
9. class FlyingBird(Bird):
10.     def fly(self):
11.         return "I can fly!"
12.
13.
14.class NonFlyingBird(Bird):
15.     def walk(self):
16.         return "I can walk!"
17.
18.
19.class Duck(FlyingBird):
20.     def quack(self):
21.         return "Quack!"
22.
23.
24.class Ostrich(NonFlyingBird):
25.     def run(self):
26.         return "I can run fast!"
27.
28.
29.# Membuat objek Duck dan Ostrich
30.duck = Duck("Duck")
31.ostrich = Ostrich("Ostrich")
32.
33.# Duck bisa terbang dan berbunyi "quack"
34.print(duck.introduce())  # Output: Hello, I'm a Duck.
35.print(duck.fly())  # Output: I can fly!
36.print(duck.quack())  # Output: Quack!
37.
38.# Ostrich tidak bisa terbang, dan bisa berjalan dan berlari cepat
39.print(ostrich.introduce())  # Output: Hello, I'm an Ostrich.
40.print(ostrich.walk())  # Output: I can walk!
41.print(ostrich.run())  # Output: I can run fast!
```

6.4 Tugas

Anda diberikan skenario berikut: Anda adalah seorang pengembang di sebuah perusahaan yang membuat perangkat lunak untuk mengelola data pegawai dan laporan gajinya. Anda diminta untuk merancang dan mengimplementasikan sistem ini dengan menggunakan prinsip SOLID. Berikut adalah spesifikasi tugas yang harus dilakukan:

1. Class Employee yang memiliki atribut name, position, dan salary.
2. Class EmployeeManager yang bertanggung jawab untuk menambahkan, menghapus, dan mencari pegawai.
3. Class EmployeeReport yang bertanggung jawab untuk menghasilkan laporan gaji pegawai.
4. Pastikan bahwa setiap Class memiliki satu tanggung jawab (Single Responsibility Principle).
5. Pastikan bahwa aplikasi dapat menambahkan jenis laporan gaji baru tanpa mengubah kelas EmployeeReport (Open/Closed Principle).
6. Pastikan bahwa semua metode pada Class EmployeeManager dapat diterapkan pada kelas turunannya (Liskov Substitution Principle).
7. Pastikan bahwa EmployeeReport tidak bergantung langsung pada EmployeeManager, tetapi hanya pada abstraksi dari EmployeeManager (Dependency Inversion Principle).

Berikut ini merupakan kode dari Tugas diatas, tuliskan analisis Anda tentang bagaimana prinsip SOLID diterapkan pada kode berikut

```
1. class Employee:
2.     def __init__(self, name, position, salary):
3.         self.name = name
4.         self.position = position
5.         self.salary = salary
6.
7. class EmployeeManager:
8.     def __init__(self):
9.         self.employees = []
10.
```

```
    def add_employee(self, employee):
        self.employees.append(employee)

    def remove_employee(self, employee):
        self.employees.remove(employee)

    def find_employee(self, name):
        for employee in self.employees:
            if employee.name == name:
                return employee
        return None

class EmployeeReport:
    def generate_report(self, employee):
        return f"Employee: {employee.name}, Position: {employee.position}, Salary: {employee.salary}"

class SalaryReport(EmployeeReport):
    def generate_report(self, employee):
        return f"Salary Report: {employee.name} earns {employee.salary}"

class EmployeeManagerAbstraction:
    def get_employee(self, name):
        pass

class EmployeeManagerAdapter(EmployeeManagerAbstraction):
    def __init__(self, manager):
        self.manager = manager

    def get_employee(self, name):
        return self.manager.find_employee(name)

class ReportGenerator:
    def __init__(self, manager, report):
        self.manager = manager
        self.report = report

    def generate_report(self, name):
        employee = self.manager.get_employee(name)
        if employee:
            return self.report.generate_report(employee)
        return f"Employee {name} not found"

# Mencoba implementasi
manager = EmployeeManager()
```

```
manager.add_employee(Employee("John Doe", "Developer", 5000))
manager.add_employee(Employee("Jane Smith", "Manager", 6000))

adapter = EmployeeManagerAdapter(manager)

generator = ReportGenerator(adapter, SalaryReport())
print(generator.generate_report("John Doe"))  # Output: Salary Report: John Doe earns 5000
print(generator.generate_report("Jane Smith"))  # Output: Salary Report: Jane Smith earns 6000
```