MODUL 2

SOLID PRINCIPLE

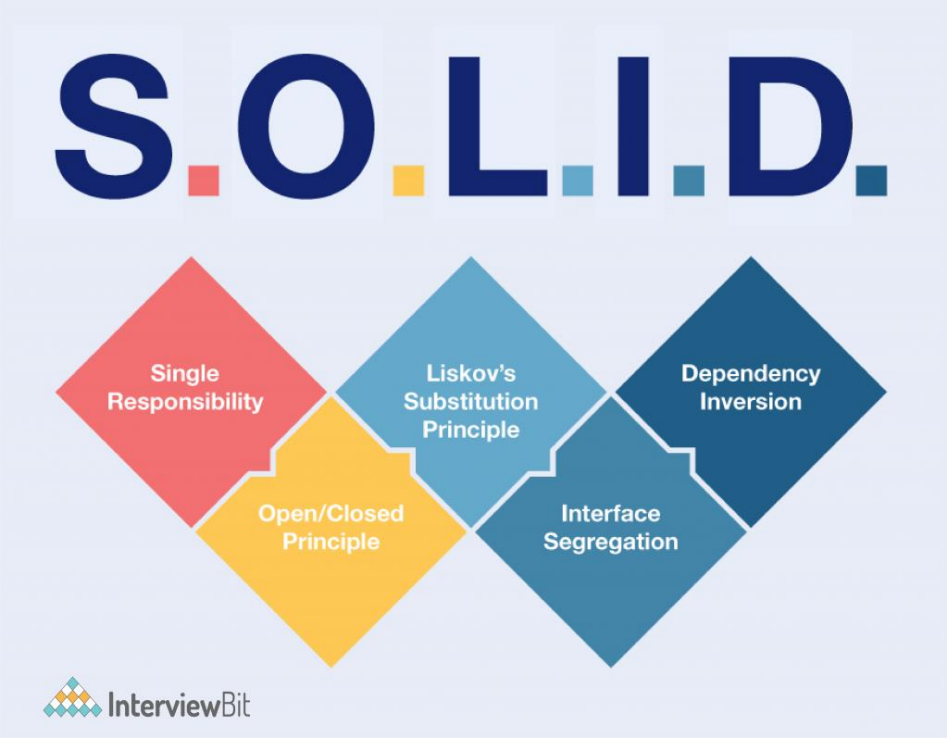
A. TUJUAN

a. Mahasiswa mengenali prinsip SOLID dalam pemrograman perangkat lunak
b. Mahasiswa mampu mengimplementasikan prinsip SOLID dalam bahasa pemrograman Java

B. ALOKASI WAKTU 1 x 50 menit

C. DASAR TEORI

Prinsip SOLID adalah seperangkat prinsip desain perangkat lunak yang telah diusulkan untuk menghasilkan kode yang lebih mudah dipahami, dikelola, dan dipelihara. Prinsip-prinsip ini membantu pengembang perangkat lunak menghasilkan kode yang fleksibel, mudah diubah, dan lebih tahan terhadap perubahan. Prinsip SOLID membentuk panduan untuk membangun desain perangkat lunak yang baik dan menghindari masalah umum seperti ketergantungan berlebihan, kode yang sulit diuji, dan kesulitan dalam melakukan perubahan.



Prinsip-prinsip SOLID dikonsep oleh Robert C. Martin (juga dikenal sebagai Paman Bob) pada tahun 2000 di dalam artikelnya yang berjudul Design Principles and Design Patterns. Melalui prinsip ini, programmer tidak hanya terbantu dalam pembangunan kode yang lebih baik, tetapi juga mendukung konsep-konsep seperti pembuatan pengujian yang lebih baik, modularitas yang lebih baik, dan perubahan yang lebih mudah. Meskipun implementasi dari

prinsip-prinsip ini dapat bervariasi tergantung pada konteks dan bahasa pemrograman yang digunakan, mereka memberikan pedoman yang kuat untuk pengembang perangkat lunak dalam upaya untuk menghasilkan kode yang berkualitas tinggi.

Robert C Martin

Lima prinsip SOLID adalah sebagai berikut:

1. **Single Responsibility Principle (SRP)**: Setiap kelas atau modul dalam kode seharusnya hanya memiliki satu tanggung jawab tunggal. Prinsip ini menggunakan konsep pencegahan adanya kelas yang terlalu kompleks dan sulit dimengerti karena melakukan terlalu banyak hal. Manfaat yang diperoleh melalui prinsip ini yakni:
    a. Pengujian – Kelas dengan satu tanggung jawab akan memiliki kasus pengujian yang jauh lebih sedikit.
    b. Kopling yang lebih rendah – Fungsionalitas yang lebih sedikit dalam satu kelas akan memiliki ketergantungan yang lebih sedikit.
    c. Organisasi – Kelas yang lebih kecil dan terorganisir dengan baik lebih mudah dicari dibandingkan kelas yang monolitik.

    Perhatikan contoh berikut:

```
class Mobil{
    void nyalakanMesin(){
        //...
    }
    void matikanMesin(){
        //...
    }
    void nyalakanAC(){
        //...
    }
    void matikanAC(){
        //...
    }
    void nyalakanRadio(){
        //...
    }
}
```

Pada contoh di atas, kita memiliki kelas mobil dengan berbagai method di dalamnya yang bias akita jumpai di mobil pada umumnya. Namun, hal ini ternyata kurang efektif karena bisa jadi ada beberapa mobil yang tidak memiliki AC dan radio. Jika kita mengubah method pada kelas tersebut, bisa menimbulkan error yang berkelanjutan. Oleh karena itu, dengan menerapkan prinsip single responsibility, kita memisahkan method tersebut agar masuk ke dalam kelas yang berkesesuaian.

```
class Mobil{
    AirConditioner AC;
    Radio radio;

    void nyalakanMesin(){
        //...
    }
    void matikanMesin(){
        //...
    }
}

class AirConditioner{
    void nyalakanAC(){
        //...
    }
    void matikanAC(){
        //...
    }
}

class Radio{
    void nyalakanRadio(){
        //...
    }
}
```

2. **Open/Closed Principle (OCP)**: Prinsip ini menyatakan bahwa kode sumber harus terbuka untuk perluasan tetapi tertutup untuk modifikasi. Dengan kata lain, kita seharusnya bisa menambahkan fungsionalitas baru tanpa mengubah kode yang sudah ada. Hal ini tidak berlaku jika pada awalnya kita memang bermaksud untuk melakukan perubahan pada existing code karena adanya error.
   Perhatikan contoh berikut:

```
class robot{
    void berjalan(){
        //...
    }
    void bersuara(){
        //...
    }
}
```

Kita memiliki sebuah kelas robot yang dapat berjalan dan bersuara. Pada suatu saat ada pengembangan dari robot tersebut sehingga dapat terbang namun tidak perlu bisa berjalan lagi, lalu diubahlah kelas tersebut menjadi seperti berikut

```
class robot{
//    void berjalan(){
//        //...
//    }
    void bersuara(){
        //...
    }
    void terbang(){
        //...
    }
}
```

Sekilas tidak ada yang salah, namun perubahan yang terjadi menyebabkan potensi error jika ada objek dari kelas robot yang menggunakan method berjalan() di dalam sistem kita. Dibanding menambah dan mengurangi method kelas robot, maka lebih baik gunakan extends.

```
class robotku{
    void berjalan(){
        //...
    }
    void bersuara(){
        //...
    }
}
class robotSuper extends robotku{
    void terbang(){
        //...
    }
}
```

3. **Liskov Substitution Principle (LSP)**: Prinsip ini berbicara tentang substitusi kelas turunan (subclass) untuk kelas induk (superclass) tanpa merusak fungsionalitas yang diharapkan. Jika kelas turunan tidak dapat digunakan sebagai pengganti kelas induk tanpa mengubah perilaku yang diharapkan, maka prinsip ini dilanggar. Sederhananya, suatu kelas turunan haruslah memiliki sifat atau perilaku yang menyerupai parentnya.
   Prinsip ini agak rumit dan menarik karena semuanya dirancang berdasarkan konsep pewarisan, jadi mari kita lebih memahaminya dengan sebuah contoh:

```
abstract class Bebek{
    void berenang(){
        //...
    }
    void makan(){
        //logic dikunyah
    }
    void bersuara(){
        //...
    }
}

class Peking extends Bebek{
    @Override
    void makan(){
        //logic langsung telan
    }
}

class BebekKaret extends Bebek{
    @Override
    void makan(){
        //tidak bisa
    }
}
```

Sebuah kelas abstrak bebek memiliki beberapa kelas turunan yang mana melakukan overriding method sesuai dengan spesifikasinya. Sekilas nampak tidak masalah karena bebek karet melakukan overriding meskipun tidak ada logic di dalamnya. Namun hal ini kurang efektif.

Dalam proses testing, method yang tidak menampilkan proses apapun dicurigai terdapat error. Serupa dengan perilaku method makan di bebek karet, tidak menampilkan proses apapun. Hal ini akan dicurigai sebagai error meskipun sesungguhnya tidak.

Daripada membuat ambiguitas seperti itu, maka lebih baik bebek karet dipisahkan dari parent bebek dan dimasukkan dalam parent baru yang lebih sesuai, misal mainan. Kode yang tepat berubah menjadi berikut:

```
abstract class Mainan{
    void dipencet(){
        //...
    }
    void dimainkan(){
        //...
    }
}
class BebekKaret extends Mainan{
    @Override
    void dipencet(){
        //logikanya akan bersuara sama seperti bebek
    }
    @Override
    void dimainkan(){
        //logikanya akan bisa berenang sama seperti bebek
    }
}
```

4. **Interface Segregation Principle (ISP)**: Prinsip ini mengemukakan bahwa klien tidak seharusnya dipaksa untuk mengimplementasikan metode yang tidak mereka butuhkan. Prinsip ini mirip dengan Single Responsibility dengan menghindari membuat interface yang terlalu besar dan kompleks. Interface yang terlalu besar akan mengakibatkan kode menjadi tidak rapi, berbelit-belit, dan sulit untuk dilakukan pengujian dengan unit testing.
   Perhatikan kode berikut. Kita memiliki sebuah kelas dengan nama Hero abstrak yang mengimplementasikan interface HeroAbility. Asumsinya disini adalah setiap hero dari job apapun bisa menggunakan ability yang ada di interface, jika tidak bisa cukup dibuat kosong saja.

```
interface HeroAbility{
    void heal();
    void castMagic();
    void assassinate();
}

abstract class Hero implements HeroAbility{
    abstract void attack();
}

class Thief extends Hero{
    @Override
    public void heal(){
        //kosong
    }
    @Override
    public void castMagic(){
        //kosong
    }
    @Override
    public void assassinate(){
        //logika assassinate
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

```
class WhiteMage extends Hero{
    @Override
    public void heal(){
        //logika heal
    }
    @Override
    public void castMagic(){
        //logika cast magic
    }
    @Override
    public void assassinate(){
        //kosong
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

```
class DarkMage extends Hero{
    @Override
    public void heal(){
        //kosong
    }
    @Override
    public void castMagic(){
        //logika cast magic
    }
    @Override
    public void assassinate(){
        //kosong
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

Kelas thief tidak dapat melakukan heal dan cast magic, sementara tipe mage bisa namun tidak dapat melakukan assassinate. Perbedaan antara white mage dan dark mage adalah kemampuan heal. Jika kita menambah job lain, maka ada kemungkinan beberapa method interface banyak yang kosong. Sementara prinsip dari interface adalah wajib implementasi semua methodnya.
Hal ini bisa diatasi dengan memisahkan interface sesuai dengan spesifikasi masing-masing job. Perhatikan contoh yang benar berikut.

```
abstract class Hero{
    abstract void attack();
}

interface healer{
    void heal();
}

interface caster{
    void castMagic();
}

interface stealer{
    void assassinate();
}
```

```
class Thief extends Hero implements stealer{
    @Override
    public void assassinate(){
        //logika assassinate
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

```
class WhiteMage extends Hero implements healer, caster{
    @Override
    public void heal(){
        //logika heal
    }
    @Override
    public void castMagic(){
        //logika cast magic
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

```
class DarkMage extends Hero implements caster{
    @Override
    public void castMagic(){
        //logika cast magic
    }
    @Override
    void attack(){
        //logika attack
    }
}
```

5. **Dependency Inversion Principle (DIP)**: Prinsip yang terakhir ini berbicara tentang ketergantungan programmer yang lebih pada abstraksi daripada implementasi. Ini berarti modul yang lebih tinggi seharusnya tidak bergantung langsung pada modul yang lebih rendah, tetapi keduanya harus bergantung pada abstraksi yang lebih tinggi. Perhatikan contoh berikut:

```
class User{
    //definisi user
}

class UserManager{
    void saveUser(User user){
        //konseksi ke database
        //jalankan query database
    }
}
```

Pada modul penyimpanan data user, sebenarnya sah sah saja jika kita langsung menghubungkannya dengan database yang kita pilih lalu memasukkan data tersebut dengan query database yang disepakati. Namun hal ini menjadi cukup kaku apabila terdapat potensi perubahan konfigurasi penggunaan database lain. Misalkan awalnya MySQL lalu ingin menjadi local storage, maka logika dari method saveUser yang tergolong high level akan banyak sekali perubahan dan ini dapat menimbulkan error.

Solusi yang bisa dilakukan adalah menggunakan abstraksi interface data storage yang diimplementasi oleh low level module yakni kelas MySQL dan LocalStorage. Masing-masing low level module akan memiliki rincian konfigurasinya sendiri sehingga tidak akan mengganggu perubahan yang berpotensi error di high level module

```
class User{
    //definisi user
}

//high level
class UserManager{
    final DataStorage datanya;

    UserManager(this.datanya);

    void saveUser(User user){
        datanya.saveData(user);
    }
}

interface DataStorage{
    void saveData(User user);
}

//low level
class MySQL implements DataStorage{
    @Override
    public void saveData(User user){
        //konek mySQL
        //jalankan query
    }
}
class LocalStorage implements DataStorage{
    @Override
    public void saveData(User user){
        //konek local storage
        //jalankan query
    }
}
```

D. PRAKTIKUM :

- Buat sebuah aplikasi yang mengimplementasikan **minimal 2 prinsip SOLID.**
- Berikan keterangan mengenai **tujuan aplikasi** tersebut (hitung karakter, kalkulator, dll)
- Berikan **keterangan bagian mana** yang terdapat prinsip tersebut lalu **jelaskan**.
- Simpan dengan nama aplikasi sederhana tersebut beserta dokumentasinya dengan nama **praktikum2.zip**