

SOLID Principles (Menulis Kode yang Bersih dan Maintainable)

SOLID adalah akronim yang diperkenalkan oleh Robert C. Martin (https://en.wikipedia.org/wiki/Robert_C._Martin).

Ini adalah lima prinsip panduan untuk memastikan perangkat lunak yang Anda bangun mudah dipahami, fleksibel, dan mudah dipelihara.

1. S - Single Responsibility Principle (SRP)

"Satu kelas, satu alasan untuk berubah."

Prinsip ini menyatakan bahwa sebuah kelas hanya boleh memiliki **satu tanggung jawab utama**. Jangan membuat "God Object" atau kelas Palugada (Apa Lu Mau Gue Ada) yang melakukan segalanya.

Contoh Kasus: Kelas Laporan

Salah (Melanggar SRP):

Kelas ini melakukan dua hal: menyimpan data laporan DAN mencetak laporan. Jika format cetak berubah, kelas ini harus diedit. Jika logika data berubah, kelas ini juga diedit.

```
class Laporan {  
    void hitungData() {  
        // Logika bisnis menghitung data  
    }  
  
    void cetakLaporan() {  
        // Logika formatting output (HTML/PDF/Text)  
        System.out.println("Mencetak Laporan...");  
    }  
}
```

Benar (Menerapkan SRP):

Pecah menjadi dua kelas yang fokus.

```
// Kelas 1: Fokus pada data  
class Laporan {  
    void hitungData() {  
        // Logika bisnis murni  
    }  
}  
  
// Kelas 2: Fokus pada tampilan/output
```

```
class PencetakLaporan {  
    void cetak(Laporan laporan) {  
        // Logika formatting  
    }  
}
```

2. O - Open/Closed Principle (OCP)

"Terbuka untuk ekstensi, tertutup untuk modifikasi."

Anda harus bisa menambahkan fitur baru **tanpa** mengacak-acak kode lama yang sudah berjalan dengan baik. Kita menggunakan *Inheritance* atau *Interface* untuk mencapai ini.

Contoh Kasus: Diskon Pelanggan

Salah (Melanggar OCP):

Setiap ada tipe pelanggan baru, kita harus membongkar kode hitungDiskon dan menambah if-else. Ini berisiko merusak logika lama.

```
class HitungDiskon {  
    int hitung(String tipe, int harga) {  
        if (tipe.equals("Regular")) {  
            return harga;  
        } else if (tipe.equals("VIP")) {  
            return harga - 10000;  
        }  
        // Jika ada tipe "SuperVIP", kita harus edit file ini lagi!  
        return harga;  
    }  
}
```

Benar (Menerapkan OCP):

Gunakan polimorfisme. Jika ada tipe baru, cukup buat kelas baru. Kode lama tidak disentuh.

```
interface SkemaDiskon {  
    int hitung(int harga);  
}
```

```
class DiskonRegular implements SkemaDiskon {  
    public int hitung(int harga) { return harga; }  
}  
  
class DiskonVIP implements SkemaDiskon {  
    public int hitung(int harga) { return harga - 10000; }
```

```
}
```

```
// Mau tambah SuperVIP? Tinggal buat kelas baru implements SkemaDiskon.  
// Tidak perlu menyentuh kelas-kelas di atas.
```

3. L - Liskov Substitution Principle (LSP)

"Subkelas harus bisa menggantikan Superkelasnya tanpa merusak program."

Jika B adalah anak dari A, maka kita harus bisa menggunakan B di mana pun A digunakan, tanpa memunculkan error aneh. Hati-hati saat melakukan *inheritance* hanya karena "mirip".

Contoh Kasus: Burung dan Penguin

Salah (Melanggar LSP):

Penguin adalah Burung, tapi dia tidak bisa terbang. Jika kita paksa, program akan error.

```
class Burung {  
    void terbang() {  
        System.out.println("Burung terbang...");  
    }  
}
```

```
class Penguin extends Burung {  
    @Override  
    void terbang() {  
        // Melanggar prinsip! Mengubah perilaku dasar dengan Error.  
        throw new UnsupportedOperationException("Penguin tidak bisa terbang!");  
    }  
}
```

Benar (Menerapkan LSP):

Pisahkan hierarkinya. Jangan asumsikan semua burung bisa terbang.

```
class Burung {  
    void makan() { ... }  
}
```

```
class BurungTerbang extends Burung {  
    void terbang() { ... }  
}
```

```
class Elang extends BurungTerbang { ... } // Aman  
class Penguin extends Burung { ... } // Aman, tidak punya metode terbang
```

4. I - Interface Segregation Principle (ISP)

"Banyak interface spesifik lebih baik daripada satu interface raksasa."

Jangan memaksa sebuah kelas untuk mengimplementasikan metode yang tidak ia butuhkan. Ini sering terjadi pada *interface* yang terlalu gemuk.

Contoh Kasus: Pekerja Pabrik

Salah (Melanggar ISP):

Robot dipaksa mengimplementasikan metode makan(), padahal robot tidak makan.

```
interface Pekerja {
```

```
    void kerja();
```

```
    void makan();
```

```
}
```

```
class Manusia implements Pekerja {
```

```
    public void kerja() { ... }
```

```
    public void makan() { ... }
```

```
}
```

```
class Robot implements Pekerja {
```

```
    public void kerja() { ... }
```

```
    public void makan() {
```

```
        // Robot tidak butuh ini, tapi terpaksa ada karena interface
```

```
        // return null;
```

```
}
```

```
}
```

Benar (Menerapkan ISP):

Pecah interface menjadi bagian-bagian kecil (role-based).

```
interface BisaKerja {
```

```
    void kerja();
```

```
}
```

```
interface BisaMakan {
```

```
    void makan();
```

```
}
```

```
class Manusia implements BisaKerja, BisaMakan { ... }
```

```
class Robot implements BisaKerja { ... } // Robot tidak perlu implement BisaMakan
```

5. D - Dependency Inversion Principle (DIP)

"Bergantunglah pada abstraksi, bukan pada konkretisasi."

Modul tingkat tinggi (logika bisnis utama) tidak boleh bergantung langsung pada modul tingkat rendah (detail teknis seperti database, sensor, dll). Keduanya harus bergantung pada *Interface*.

Contoh Kasus: Saklar Lampu

Salah (Melanggar DIP):

Saklar terikat mati dengan Lampu. Saklar ini tidak bisa dipakai untuk menyalakan Kipas Angin.

```
class Lampu {  
    void nyala() { ... }  
    void mati() { ... }  
}
```

```
class Saklar {  
    Lampu lampu; // Ketergantungan langsung (Hard dependency)  
  
    public Saklar() {  
        this.lampu = new Lampu(); // Saklar menciptakan Lampu  
    }  
  
    void tekan() {  
        lampu.nyala();  
    }  
}
```

Benar (Menerapkan DIP):

Saklar bergantung pada "Barang Elektronik" (*Interface*). Sekarang Saklar bisa menyalakan apa saja!

```
interface BarangElektronik {  
    void nyala();  
    void mati();  
}  
  
class Lampu implements BarangElektronik { ... }  
class KipasAngin implements BarangElektronik { ... }  
  
class Saklar {
```

```
BarangElektronik barang; // Bergantung pada Abstraksi

// Dependency Injection via Constructor
public Saklar(BarangElektronik barang) {
    this.barang = barang;
}

void tekan() {
    barang.nyala();
}
}
```