

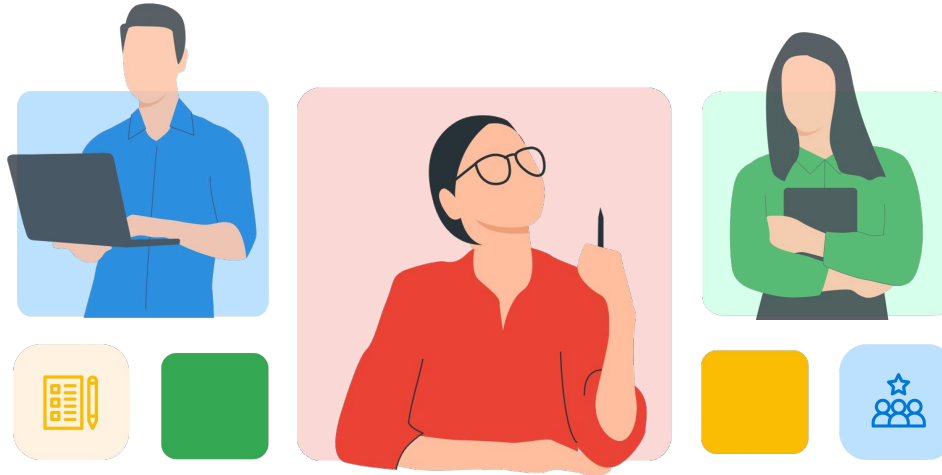


# 79 Bootcamp Digitalization



# Solid Design Principles in Java

---



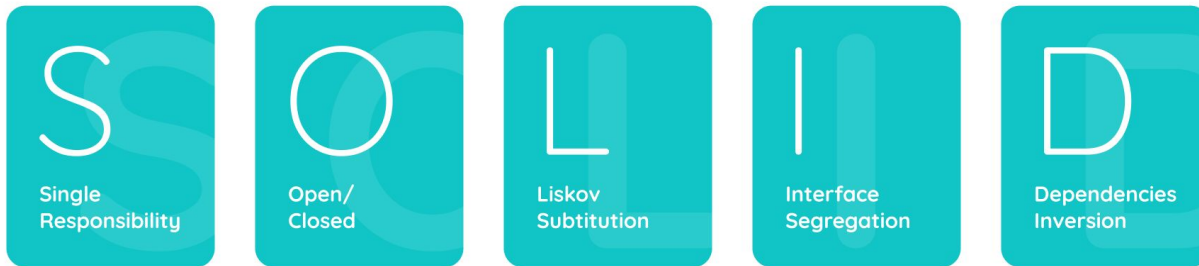
# 1. Apa itu SOLID Principle?

---

# S.O.L.I.D

## Apa itu SOLID

SOLID merupakan kumpulan dari beberapa principle yang diwujudkan oleh engineer-engineer yang ahli dibidangnya. SOLID membantu kita mengembangkan sebuah perangkat lunak dengan tingkat kekukuhan yang tinggi.

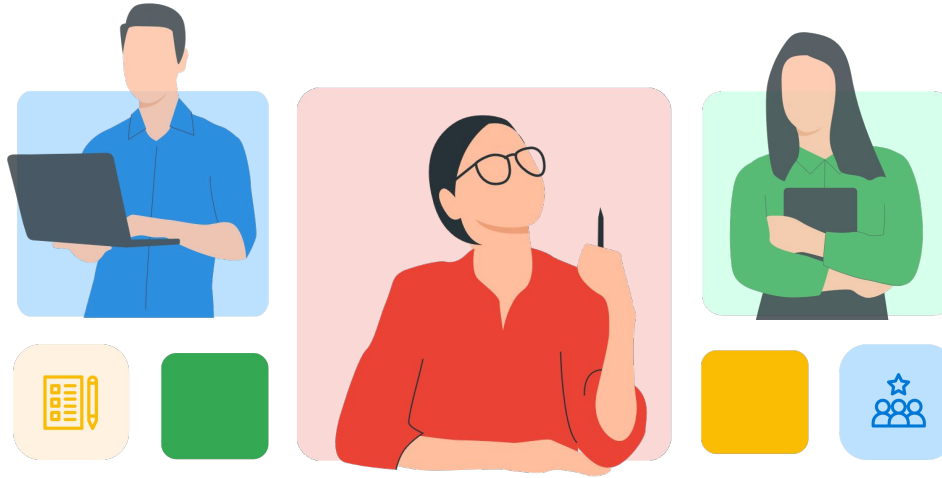


## Tujuan SOLID

Dengan mengikuti prinsip SOLID, kode yang kita buat dapat dengan mudah di ekstensi (extended) dan dipertahankan (maintained).

Sebuah prinsip yang dimaksudkan untuk membantu kita dalam menuliskan kode yang rapi. Bagaimana hal itu dapat diwujudkan? Berikut adalah tujuan dari prinsip SOLID dalam pembuatan struktur mid-level perangkat lunak:

- Toleran terhadap perubahan.
- Mudah dipahami.
- Komponen dasar dapat digunakan kembali dalam bentuk software system lainnya.



## 2. Single Responsibility Principle (SRP)

---

# Single Responsibility Principle



A module should be responsible to one, and only one, actor.

(Robert Cecil Martin, 2017)

Every class in Java should have a single job to do. To be precise, there should only be one reason to change a class.

**Single Responsibility Principle** merupakan sebuah principle yang relatif mudah diterapkan dalam pengembangan perangkat lunak. Sederhananya, principle ini digunakan untuk mengatur tanggung jawab dari sebuah entitas yang berada di dalam sebuah proyek dalam hal ini adalah sebuah module/class.

**Tanggung jawab (responsibility)** berarti bahwa jika suatu class punya 2 (dua) fungsionalitas yang **tidak memiliki keterkaitan** untuk melakukan suatu perubahan, maka kita harus **membagi fungsionalitas** yang berbeda tersebut dengan cara memisahnya menjadi dua class yang berbeda.

Contoh:

Employee class melakukan beberapa pekerjaan sekaligus, hal ini melanggar aturan SRP.

```
public class Employee {  
    public void calculatePay() {}  
    public void reportHours() {}  
    public void save() {}  
    public void hireEmployee() {}  
    public void fireEmployee() {}  
    public void getDailyHistoryPayment(){}  
    public void getMonthlyHistoryPayment(){}  
}
```



Dengan menerapkan Konsep atau prinsip SRP, maka beberapa method harus dipecah dan dijadikan class tersendiri. Seperti Berikut ini:

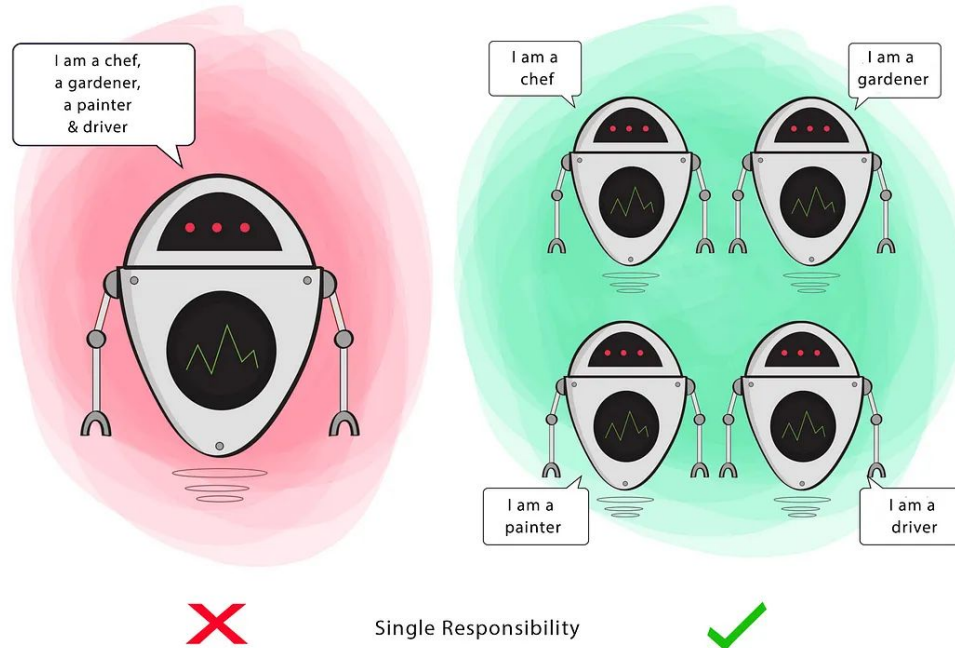
```
public class Employee {  
    public void calculatePay() {}  
    public void reportHours() {}  
    public void save() {}  
}
```

```
public class EmployeeRecruitment {  
    public void hireEmployee() {}  
    public void fireEmployee() {}  
}
```

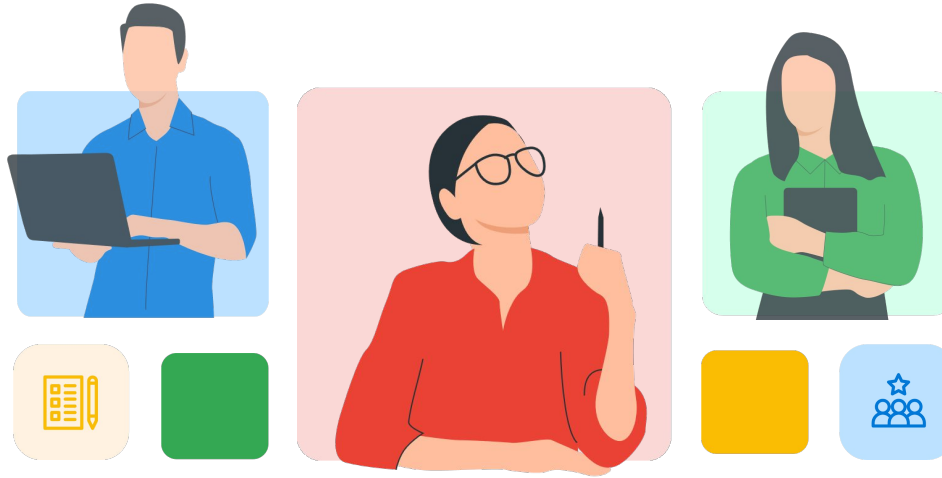
```
public class HistoryPayment {  
    public void getDailyHistoryPayment(){}  
    public void getMonthlyHistoryPayment(){}  
}
```

Keunggulan dari SRP adalah kode yang dibuat akan lebih mudah dikelola dan diidentifikasi. Bayangkan jika seluruh operasi untuk business rules Employee ditumpuk pada class Employee, developer akan terus menambahkan fungsi pada class Employee seiring bertambahnya requirement atau business rules.

# Ilustrasi Single Responsibility Principle



Improvement



### 3. Open/Close Principle (OCP)





# Open/Close Principle (OCP)



A software artifact should be open for extension but closed for modification.

(Bertrand Meyer, 1988)

The OCP is one of the driving forces behind the architecture of systems. The goal is to make the system easy to extend without incurring a high impact of change. This goal is accomplished by partitioning the system into components, and arranging those components into a dependency hierarchy that protects higher-level components from changes in lower-level components.



Lantas apa yang dimaksud dengan **terbuka untuk ditambahkan** dan **tertutup untuk dimodifikasi**?

Jangan bingung. Terbuka untuk ditambahkan adalah keadaan ketika sebuah sistem dapat ditambahkan dengan spesifikasi baru yang dibutuhkan. Sedangkan tertutup untuk dimodifikasi adalah agar ketika ingin menambahkan spesifikasi baru, kita tidak perlu mengubah atau memodifikasi sistem yang telah ada.

Ketika akan terjadi penambahan fitur pada suatu class, maka seharusnya kita melakukan extends class yang ingin ditambahkan fitur tersebut bukan dengan langsung memodifikasi class tersebut.

Sebagai contoh, Pada kode di bawah, ketika terdapat metode pembayaran baru, maka class di bawah ini harus diubah, sehingga keseluruhan program harus melakukan recompilation atau redeployment.

```
public class PaymentCalculator {  
    public double adminFee(PaymentMethod paymentMethod) {  
        if (paymentMethod instanceof VirtualAccountMethod) {  
            return paymentMethod.price * 0.1;  
        } else if (paymentMethod instanceof CODMethod) {  
            return paymentMethod.price * 0.2;  
        } else {  
            return 0;  
        }  
    }  
}
```

Dengan menerapkan OCP (Open Close Principle), kita dapat mengubah class PaymentCalculator menjadi abstract class PaymentMethod. Di dalamnya terdapat abstract function calculateAdminFee() yang akan mengembalikan nilai admin fee.

Jadi, perhitungan admin fee diletakkan pada masing-masing jenis payment method, dalam hal ini dimodelkan dalam bentuk class CODMethod yang berbeda seperti VirtualAccountMehtod.

```
abstract class PaymentMethod {  
    int price;  
    public Double calculateAdminFee();  
}  
  
public class CODMethod extends PaymentMethod{  
    public Double calculateAdminFee() {  
        return price * 0.2;  
    }  
}  
  
public class VirtualAccountMethod extends PaymentMethod{  
    public Double calculateAdminFee() {  
        return price * 0.1;  
    }  
}
```

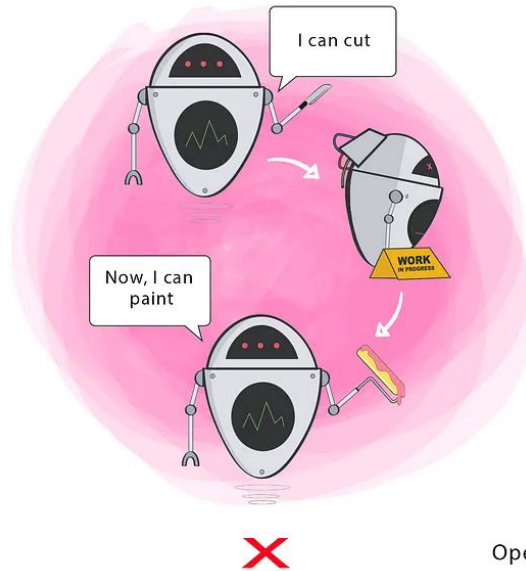
Tentu akan lebih mudah jika kita ingin menambahkan payment method baru, Maka kita tinggal membuat class baru kemudian extends ke class PaymentMethod.

Kemudian ketika kita ingin membuat object Payment Method di Main Class, Maka kita tinggal memanggil method menghitung biaya admin sesuai tipe pembayaran.

```
public class PaymentCalculator {  
    public static void main(String[] args) {  
        PaymentMethod bcaVirtualAccount = new VirtualAccountMethod();  
        System.out.println(bcaVirtualAccount.calculateAdminFee());  
    }  
}
```

Sehingga jika ada metode pembayaran baru, cukup membuat object baru tanpa harus memodifikasi yang sudah ada.

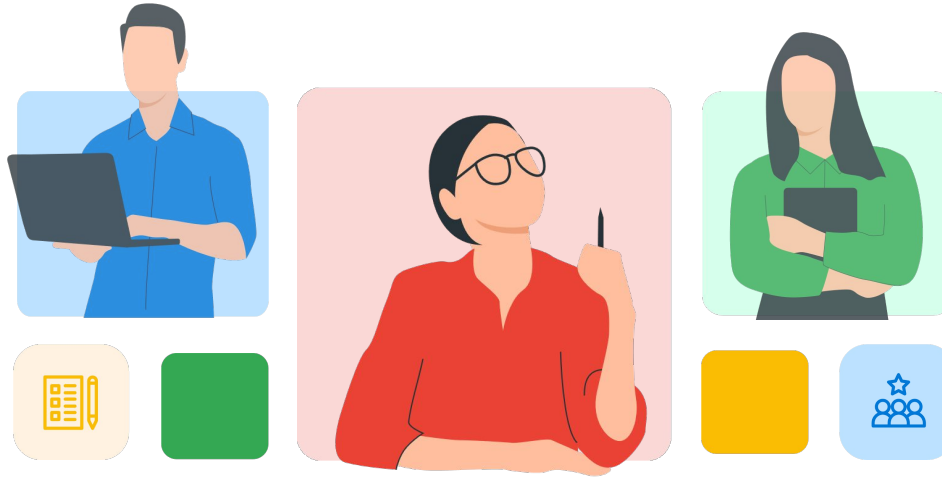
# Ilustrasi Open/Close Principle (OCP)



Open-Closed







## 4. Liskov Substitution Principle

---

# Liskov Substitution Principle (LSP)

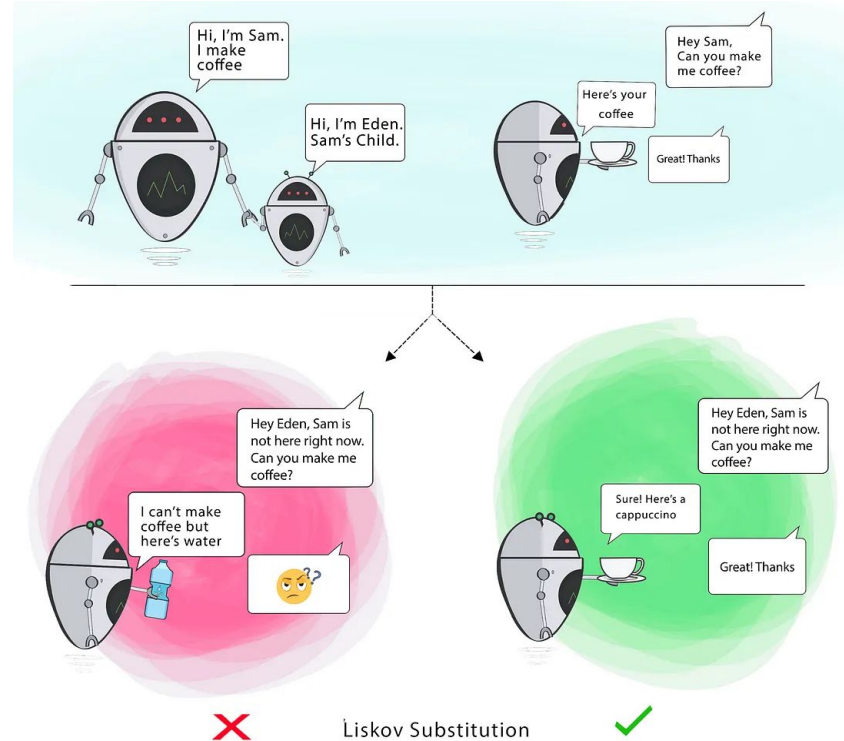
Every subclass or derived class should be substitutable for their base or parent class.

When a child Class cannot perform the same actions as its parent Class, this can cause bugs.

If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a child. The child Class should be able to do everything the parent Class can do. This process is called Inheritance.

The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.

# Ilustrasi Liskov Substitution Principle (LSP)

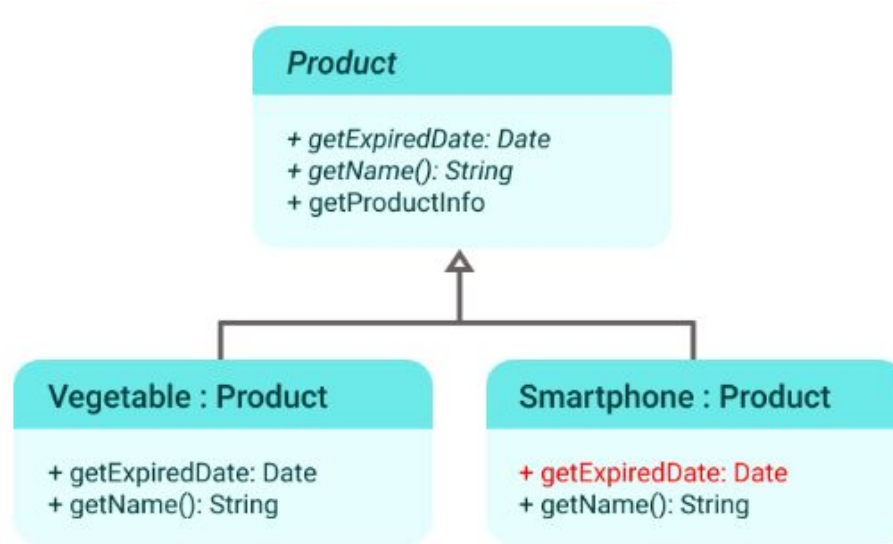


Improvement

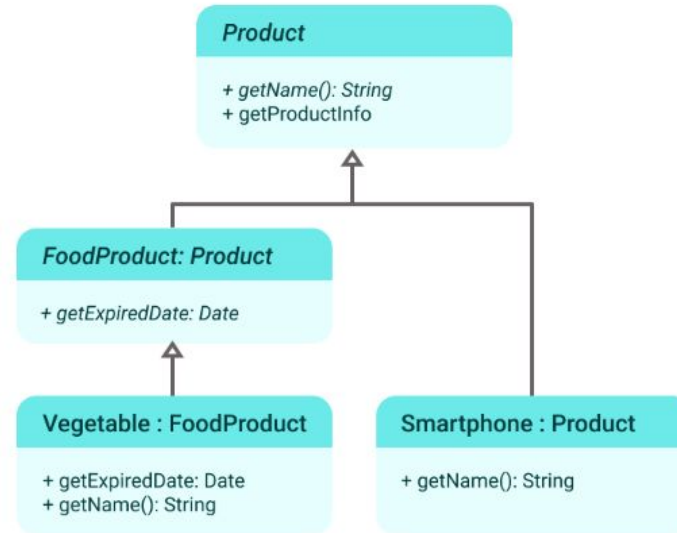
Sederhananya, **Liskov substitution** adalah aturan yang berlaku untuk hirarki pewarisan. Hal ini mengharuskan kita untuk mendesain kelas-kelas yang kita miliki sehingga ketergantungan antar klien dapat disubstitusikan tanpa klien mengetahui tentang perubahan yang ada. Oleh karena itu, seluruh SubClass setidaknya dapat berjalan dengan cara yang sama seperti SuperClass-nya.

LSP adalah tentang delegasi tanggung jawab. Jadi, anak robot harus bisa melakukan apa yang ibunya bisa. Misalnya ibunya bisa buat teh, maka anaknya pun juga harus bisa melakukan hal yang sama. Itulah yang disebut dengan LSP.

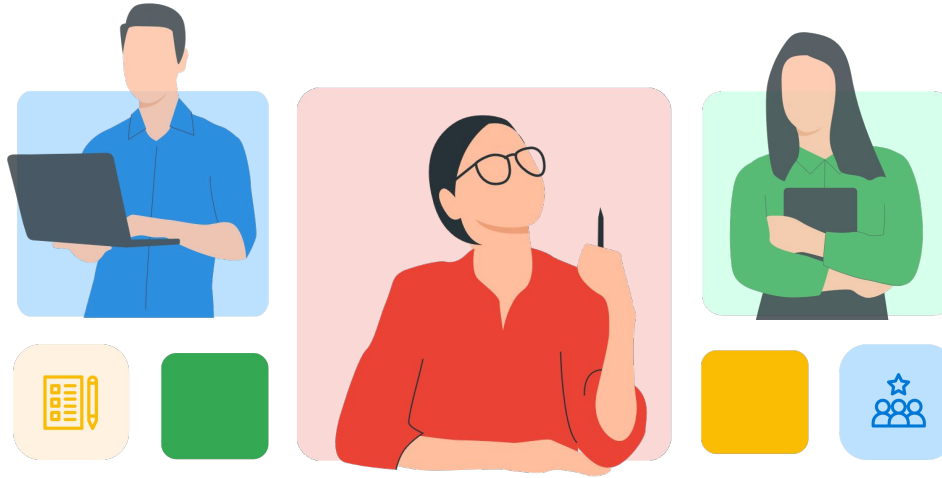
Sebagai Contoh:



Untuk mengatasi kasus di atas, kita perlu melakukan substitusi fungsi yang tidak relevan tersebut ke dalam kelas abstraksi sendiri dan diwariskan pada kelas yang relevan. Namun, Perubahan ini tetap menjadikan kelas Product sebagai SuperClass dari hirarki yang ada saat ini. Kurang lebih perubahannya akan seperti berikut:



Dengan perubahan seperti di atas, kita sudah memenuhi aturan yang ada. Mudah bukan? Liskov Substitution principle merupakan prinsip yang dapat meningkatkan design dari sistem yang kita kembangkan. Sehingga ketergantungan antar klien dapat disubstitusikan tanpa klien tahu perubahan yang ada.



## 5. Interface Segregation Principle (ISP)

---


# Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods that they do not use.

(Robert Cecil Martin)

When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

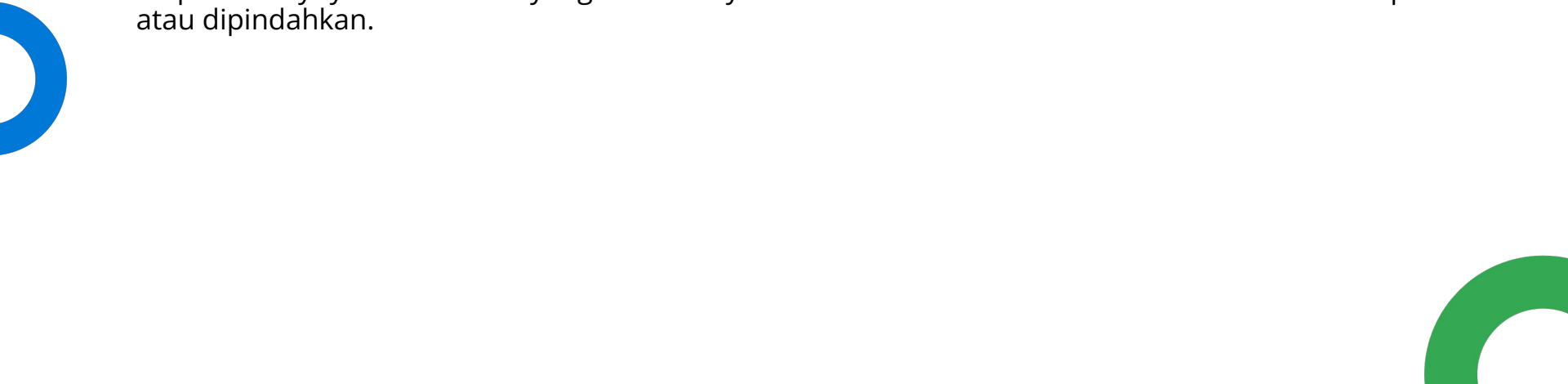
A Class should perform only actions that are needed to fulfil its role. Any other action should be removed completely or moved somewhere else if it might be used by another Class in the future.



Prinsip ini sendiri bertujuan untuk mengurangi jumlah ketergantungan sebuah class terhadap interface class yang tidak dibutuhkan.

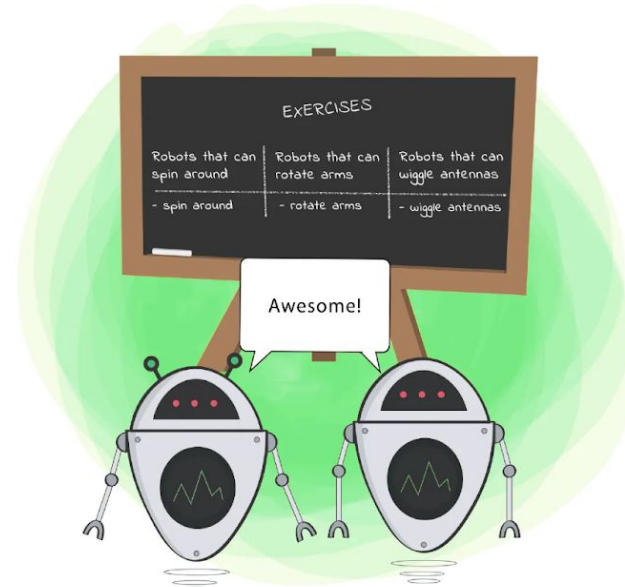
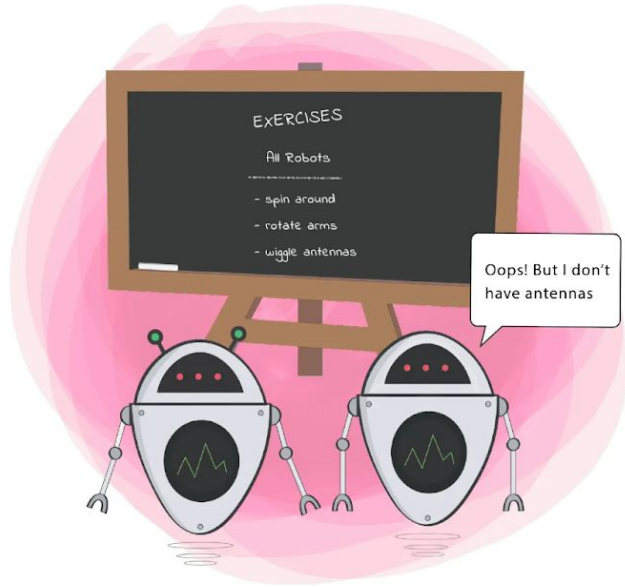
Faktanya, class memiliki ketergantungan terhadap class lainnya. Jumlah ketergantungan dari fungsi pada sebuah interface class yang dapat diakses oleh class tersebut harus dioptimalkan atau dikurangi. Mengapa penting?

Sebuah class hanya boleh melakukan action yang hanya diperlukan untuk memenuhi responsibilitynya. Action lain yang seharusnya tidak dilakukan oleh class tersebut harus dihapus atau dipindahkan.





# Ilustrasi Interface Segregation Principle (ISP)



Interface Segregation

Improvement

Sebagai Contoh:

```
interface VehicleInterface {  
    void drive();  
    void stop();  
    void refuel();  
    void openDoors();  
}  
  
public abstract class Vehicle implements VehicleInterface {  
  
}
```

```
public class Motorcycle extends Vehicle {  
  
    // Can be implemented  
    @Override  
    public void drive() { }  
  
    @Override  
    public void stop() { }  
  
    @Override  
    public void refuel() { }  
  
    // Can not be implemented  
    @Override  
    public void openDoors() { }  
}
```

```
public class Car extends Vehicle {  
  
    // Can be implemented  
    @Override  
    public void drive() { }  
  
    @Override  
    public void stop() { }  
  
    @Override  
    public void refuel() { }  
  
    // Can implemented  
    @Override  
    public void openDoors() { }  
}
```

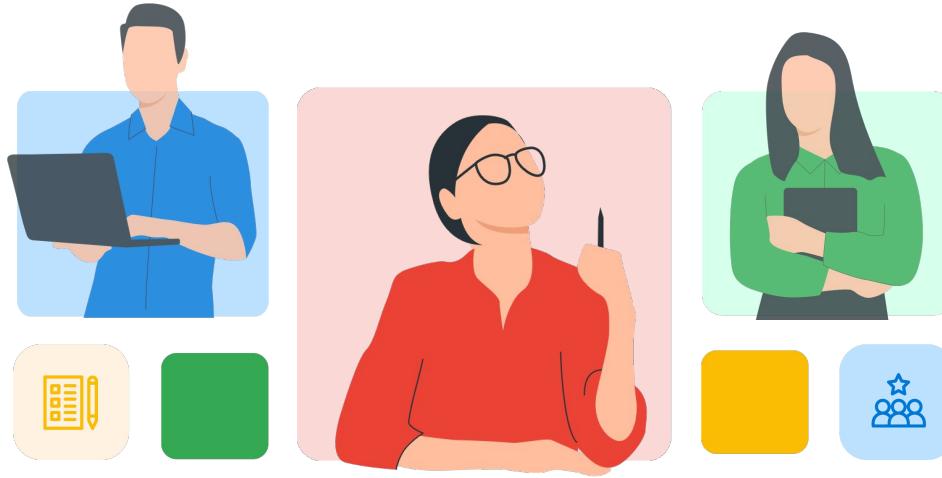
Dengan menerapkan ISP, maka kita harus menyesuaikan interface yang digunakan. Dengan cara membuat sebuah interface baru yang lebih spesifik yaitu DoorInterface.

```
interface VehicleInterface {  
    void drive();  
    void stop();  
    void refuel();  
}  
  
public abstract class Vehicle implements VehicleInterface {  
  
}
```

```
interface DoorInterface {  
    void openDors();  
}
```

```
public class Motorcycle extends Vehicle {  
  
    // Can be implemented  
    @Override  
    public void drive() { }  
  
    @Override  
    public void stop() { }  
  
    @Override  
    public void refuel() { }  
  
}
```

```
public class Car extends Vehicle implements DoorInterface {  
  
    // Can be implemented  
    @Override  
    public void drive() { }  
  
    @Override  
    public void stop() { }  
  
    @Override  
    public void refuel() { }  
  
    // Can be implemented  
    @Override  
    public void openDoors() { }  
  
}
```



## 6. Dependency Inversion Principle (DIP)

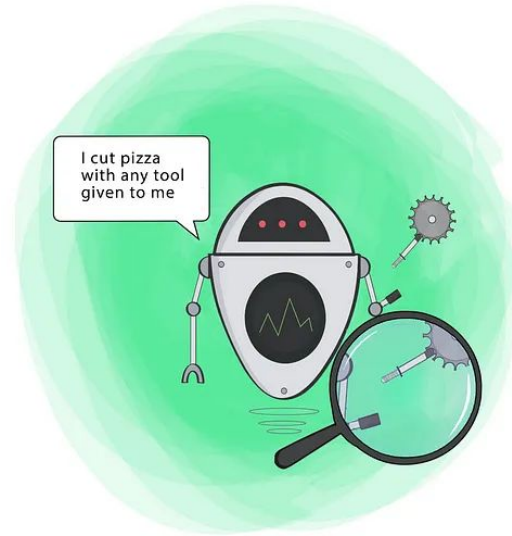
---



# Dependency Inversion Principle (DIP)

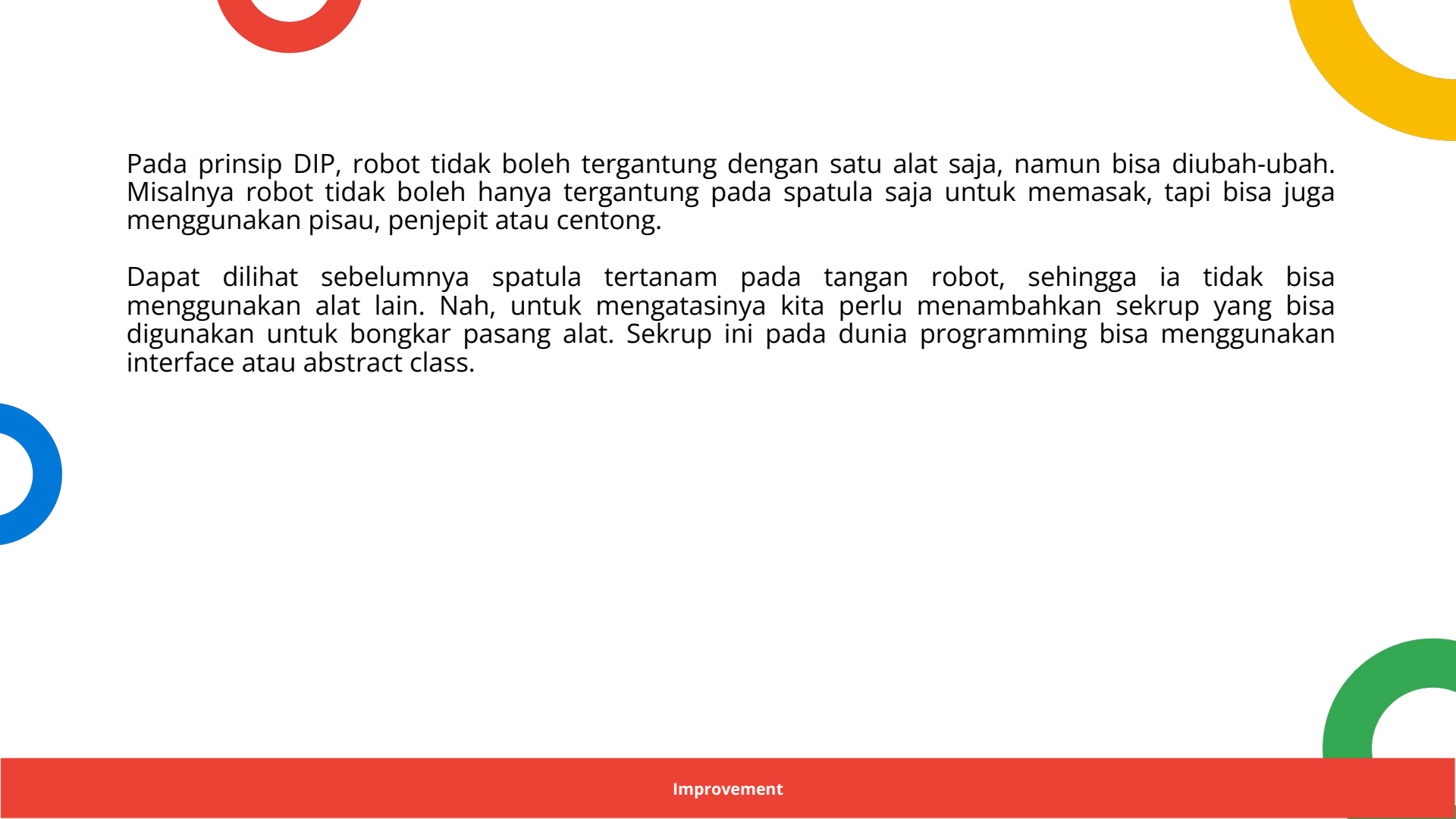
The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

# Dependency Inversion Principle (DIP)



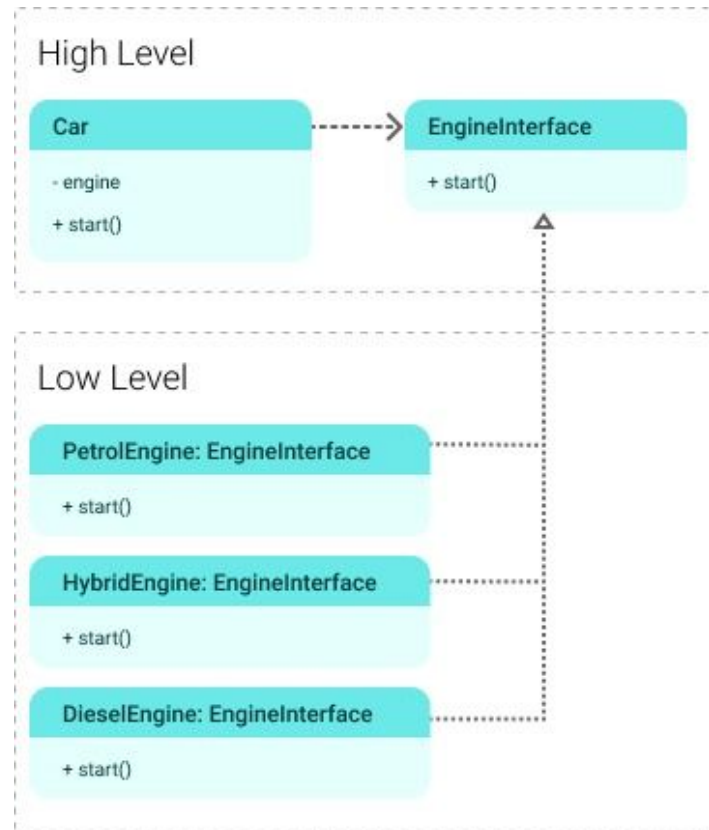
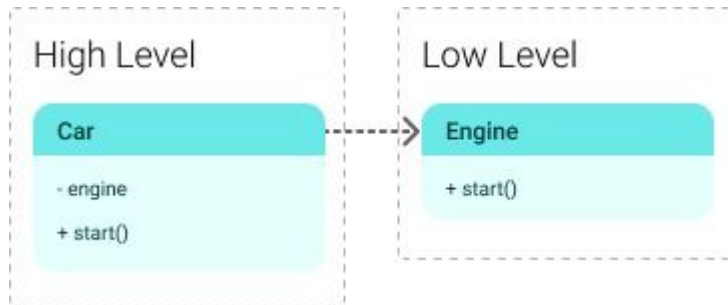
Dependency Inversion

Improvement



Pada prinsip DIP, robot tidak boleh tergantung dengan satu alat saja, namun bisa diubah-ubah. Misalnya robot tidak boleh hanya tergantung pada spatula saja untuk memasak, tapi bisa juga menggunakan pisau, penjepit atau centong.

Dapat dilihat sebelumnya spatula tertanam pada tangan robot, sehingga ia tidak bisa menggunakan alat lain. Nah, untuk mengatasinya kita perlu menambahkan sekrup yang bisa digunakan untuk bongkar pasang alat. Sekrup ini pada dunia programming bisa menggunakan interface atau abstract class.





## Referensi

R. C. Martin, Clean Architecture a Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall, 2018.

U. Thelma, "The S.O.L.I.D principles in pictures," Medium,  
<https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898> (accessed May 14, 2023).



**Tujuh  
Sembilan**  
Always Improving You

# Sekian, Terima Kasih