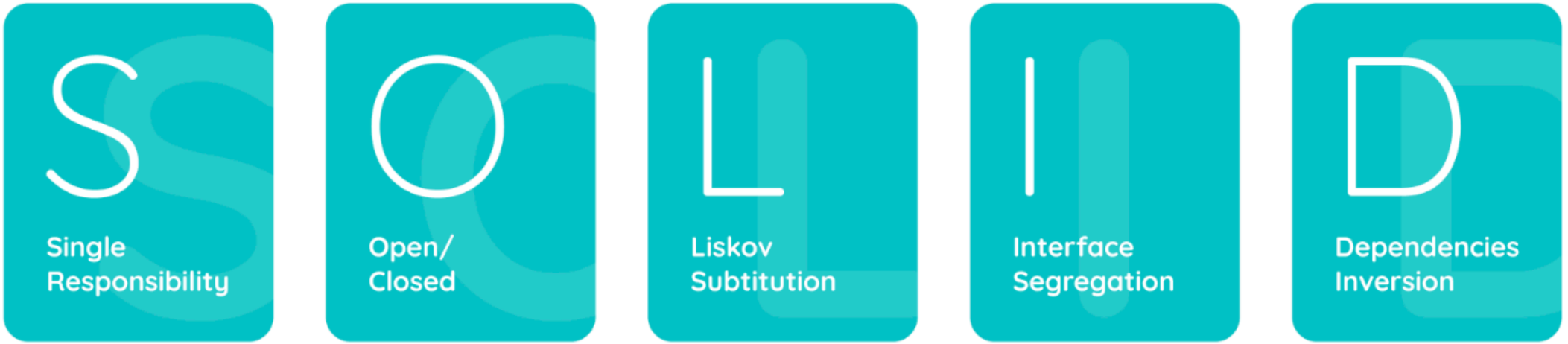
SOLID PRINCIPLE

Presented by Agung N. Aprianto

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.**



Single Responsibility Principle (SRP)

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

Robert Cecil Martin, 2017

Single Responsibility Principle (SRP)

SRP 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* untuk memenuhi kebutuhan dari *actor. Actor* merupakan kumpulan "user" atau "stakeholder" yang menginginkan perubahan pada perangkat lunak kita

Open/Close Principle (OCP)

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

Bertrand Meyer, 1988

Open/Close Principle (OCP)

Pada Tahun 1988, seorang profesor asal Perancis, Bertrand Meyer menulis sebuah buku yang berjudul Object Oriented Software Construction. Di dalamnya terdapat sebuah aturan yang mengatur di mana sebuah artefak perangkat lunak harus terbuka untuk ditambahkan tetapi tertutup untuk dimodifikasi. Aturan tersebut kemudian ditulis lagi pada sebuah artikel yang berjudul The Open-Closed Principle oleh Robert C. Martin pada tahun 1996.

Liskov Substitution Principle (LSP)

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Liskov Substitution Principle (LSP)

Liskov’s 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.

Liskov Substitution Principle (LSP)

Untuk menjadikan sebuah kelas benar-benar menjadi SubClass, kelas tersebut tidak hanya wajib untuk menerapkan fungsi dan properti dari SuperClass, melainkan juga harus memiliki perilaku yang sama dengan SuperClass-nya. Untuk mencapainya, terdapat beberapa aturan yang harus dipatuhi.

Liskov Substitution Principle (LSP)

- Contravariant dan Covariant
- Preconditions dan Postconditions
- Invariant
- Constraint

Contravariant dan Covariant

Contravariant dan Covariant

Contravariant adalah kondisi di mana parameter dari sebuah fungsi yang berada pada SubClass harus memiliki tipe dan jumlah yang sama dengan fungsi yang berada pada SuperClass-nya. Sedangkan Covariant adalah kondisi pengembalian nilai dari fungsi yang berada pada SubClass dan SuperClass.

Preconditions dan Postconditions

Preconditions dan Postconditions

preconditions dan postconditions merupakan tindakan yang harus ada sebelum atau sesudah sebuah proses dijalankan. Contohnya, ketika kita ingin memanggil sebuah fungsi yang digunakan untuk membaca data dari database, terlebih dahulu kita harus memastikan database tersebut dalam keadaan terbuka agar proses dapat dijalankan. Ini disebut sebagai precondition. Sedangkan postcondition, contohnya saat proses baca tulis di dalam database telah selesai, kita harus memastikan database tersebut sudah tertutup.

Invariant

Invariant

Dalam pembuatan sebuah SubClass, SubClass tersebut harus memiliki invariant yang sama dengan SuperClass-nya. Invariant sendiri adalah penjelasan dari kondisi suatu proses yang benar sebelum proses tersebut dimulai dan tetap benar setelahnya.

Constraint

Constraint

Constraint dari sebuah SubClass secara default dapat memiliki fungsi dan properti dari SuperClass-nya. Selain itu, kita juga dapat menambahkan member baru di dalamnya. Constraint di sini adalah pembatasan yang ditentukan oleh SuperClass terhadap perubahan keadaan sebuah obyek. Sebagai contoh misal SuperClass memiliki obyek yang memiliki nilai tetap, maka SubClass tidak diijinkan untuk mengubah keadaan dari nilai obyek tersebut.

Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces that they do not use.

Robert Cecil Martin

Interface Segregation Principle (ISP)

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

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Robert Cecil Martin

Dependency Inversion Principle (DIP)

Prinsip Dependency Inversion hampir sama dengan konsep layering dalam aplikasi, di mana low-level modules bertanggung jawab dengan fungsi yang sangat detail dan high-level modules menggunakan low-level classes untuk mencapai tugas yang lebih besar. Hal ini bisa dicapai dengan bergantung pada sebuah abstraksi, ketika ada ketergantungan antar kelas seperti interface, daripada referensi langsung ke kelas lainnya.

Dependency Inversion Principle (DIP)

Apa yang dimaksud dengan high-level modules dan low-level modules? Agar lebih mudah memahaminya, kita dapat mengkategorikan kelas-kelas menjadi sebuah hirarki. High-level modules adalah kelas-kelas yang berurusan dengan kumpulan-kumpulan fungsionalitas. Pada hirarki tertinggi terdapat kelas-kelas yang mengimplementasikan aturan bisnis sesuai dengan desain yang telah ditentukan. Low-level modules bertanggung jawab pada operasi yang lebih detail. Pada level terendah memungkinkan modul ini untuk bertanggung jawab dalam menulis informasi ke database atau menyampaikan pesan ke sistem operasi.

Don’t Repeat Yourself (DRY)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

by **Andy Hunt** and **Dave Thomas** in their book ***The Pragmatic Programmer***

Don’t Repeat Yourself (DRY)

“Don’t Repeat Yourself” (atau lebih mudahnya disingkat DRY) adalah satu kalimat istilah yang sering diperbincangkan di kalangan teknis pemrograman yaitu untuk tidak mengulangi proses yang sama berulang kali, yang mana sebenarnya proses tersebut dapat dilakukan dengan suatu solusi hanya satu kali saja.

Any Question ?

Thank You