**Kelas Pemrogramman SOLID – Dicoding**

***Software Design Principle*** merupakan serangkaian aturan yang dapat mencegah kita melakukan kebiasaan buruk dalam melakukan pengembangan perangkat lunak.

Prinisip ini diusulkan oleh seorang *Software Enginner* asal Amerika bernama Robert C. Martin, berangkat dengan alasan :

1. Sebuah sistem yang sulit dipelihara karena setiap perubahannya akan mempengaruhi bagian sistem yang sudah ada.
2. Saat melakukan perubahan, sangat mungkin akan merusak sistem yang sudah berjalan.
3. Sistem yang tidak dapat diterapkan pada aplikasi lain karena tidak dapat dipisahkan dari aplikasi saat ini.

Paradigma pemrograman adalah cara kita menulis program.

Prinsip S.O.L.I.D ini adalah panduan bagi kita saat kita akan mengembangkan sebuah sistem dengan pendekatan OOP.

Jika kita menggunakan prinsip ini, kita akan membuat sebuah sistem yang mudah digunakan, dikembangkan, dan kokoh.

Dengan paradigma OOP kita dapat dengan mudah memvisualisasikan kode karena konsepnya yang mirip dengan dunia nyata.

1. **Pilar OOP**
2. **Inheritance**

Saat menggunakan *inheritance*, kita dapat dengan mudah menggunakan kembali, menyesuaikan, dan memelihara kode yang sudah dibuat.

Macam – macam inheritance :

1. *Single Inheritance*

**Class** yang dibuat hanya mewarisi satu **class**

1. *Multi Level Inheritance*

**SubClass** dapat mewarisi **superClass** yang merupakan sebuah **SubClass** dari **superClass** lain.

1. *Multiple Inheritance*

Sebuah **class** dapat mewarisi lebih dari satu **superClass**

1. *Hierarchical* *Inheritance*

**SuperClass** yang diwarisi oleh beberapa **SubClass**

1. *Hybrid* *Inheritance*
2. **Encaptulation**

*Encaptulation* merupakan proses dimana sebuah penanganan data ditempatkan di dalam **class**.

Saat menggunakan *encaptulation*, data dapat diisolasi dan tidak dapat diakses langsung dari luar. Dengan begini, kita cukup menggunakan data tersebut tanpa tahu bagaimana proses yang terjadi.

Kita akan lebih leluasa dalam melakukan perubahan nilai tanpa harus mengakses propertinya secara langsung, cukup gunakan fungsi *setter* dan *getter.*

*Encaptulation* bukan menyembunyikan sebuah data, tetapi *encaptulation* yang menyebabkan data tersebut tersembunyi.

1. **Abstraction**

*Abstraction* adalah mekanisme saat proses dalam sebuah *object* disembunyikan. *Object* tersebut hanya akan menyediakan apa yang benar – benar perlu digunakan.

***Abstraction Layer***

*Abstraction layer* atau *abstraction level* merupakan mekanisme yang memisahkan 2 kompleksitas sebuah sistem.

Dalam proses komputasi, *abstraction layer* atau level merupakan cara menyembunyikan detil implementasi yang kompleks dari serangkaian fungsionalitas tertentu dengan tujuan agar dapat memisahkan masalah seperti introperabilitas.

***Abtraction* dalam OOP**

*Object* dalam bahasa pemrograman dengan paradigma OOP menyediakan sebuah *abstraction* yang digunakan untuk menyembunyikan mekanisme internal.

Kita hanya perlu tahu fungsi apa saja yang tersedia untuk memicu sebuah proses tertentu.

Sebaliknya, kita tidak perlu paham bagaimana proses dan tindakan apa yang harus dilakukan untuk mendapatkan hasil yang sesuai harapan.

**Perbedaan *Encaptulation* dan *Abstraction***

Jika *encaptulation* menyembunyikan data dengan menyediakan fungsi *getter* dan *setter* untuk mengaksesnya, maka *abstraction* menyembunyikan sebuah implementasi dengan memanfaatkan *abstract class, interface,* dan lain sebagainya.

1. **Polymorphism**

*Polymorphism* merupakan kemampuan *object*, variabel, atau fungsi yang dapat memiliki berbagai bentuk. Secara umum, *polymorphism* dalam OOP terjadi ketika suatu **superClass** direfrensikan ke dalam **SubClass**. Alhasil, kita dapat mengembangkan sebuah program secara umum, bukan spesifik.

*Polymorphism* terbagi menjadi 2, yaitu :

***Compile time polymorphim***

*Compile time polymorphism* adalah sebuah proses dimana sebuah *method* atau fungsi dipanggil saat kompilasi. Ini dapat terjadi karena sebuah konsep bernama *method overloading. Method overloading* merupakan kondisi dimana kita bisa membuat 2 atau lebih fungsi yang memiliki jumlah, tipe, dan urutan parameter yang berbeda didalam sebuah **class.**

***Runtime Polymorphism***

*Runtime polymorphism* adalah proses dimana sebuah fungsi dipanggil saat *runtime.* Contoh dari *runtime polymorphism* adalah *method overriding*, yaitu sebuah **class** yang memiliki fungsi dengan nama yang sama dengan fungsi yang ada pada **class** induknya.

Sederhananya, *method overriding* adalah sebuah fitur pada bahasa pemrograman yang memungkinkan *SubClass* mewarisi sebuah implementasi yang spesifik dari sebuah fungsi yang ada pada *parentClass.* Implementasi pada *subClass* akan menimpa implementasi pada *parentClass.*

Lalu kenapa *method overriding* dipanggil saat *runtime* ? Jawabannya, ketika proses kompilasi berlangsung, kompiler tidak tahu fungsi mana yang dipanggil. Apakah yang ada pada *parentClass* atau *SubClass,* karena keduannya memiliki nama yang sama. Sehingga, JVM ( Java *Virtual Machine* ) yang akan berperan menentukan fungsi mana yang akan dipanggil ketika *runtime.*

1. ***Relasi pada OOP***

Tujuannya adalah agar kita dapat menulis kode dengan mudah, bersih, dan bisa digunakan kembali ( *reuasable ),* serta untuk mengurangi kompleksitas sistem yang kita kembangkan.

Jika kita paham hubungan antar *object,* kita dapat lebih mudah memahami dan mengurangi kompleksitas sistem yang akan kita kembangkan.

Dalam hubungan antar objek*,* terdapat 3 kategori :

* *Association*, memiliki hubungan “has-a” yang menyatakan bahwa sebuah **class** memiliki hubungan dengan **class** lainnya.
* *Dependency,* memiliki arti bahwa sebuah **class** memiliki ketergantungan terhadap **class** lain.
* *Generalitazion,* memiliki hubungan “is-a” dari sebuah **class** yang spesifik ke **class** yang lebih umum.

Detail lebih lanjut :

1. **Asosiasi**

Asosiasi didefinisikan sebagai hubungan yang terstruktur, yang secara konsep memiliki arti bahwa dua komponen saling terhubung satu sama lain. Setiap objek memiliki siklus hidupnya sendiri dan tidak memiliki *ownership* antara kedua objek tersebut.

**Kardinalitas**

Hubungan asosiasi sendiri memiliki hubungan *one-to-one, one-to-many, many-to-one,* dan *many-to-many* atau yang disebut sebagai kardinalitas. Kardinalitas adalah hubungan maksimum yang terjadi dari himpunan entitas yang satu ke himpunan entitas yang lain, demikian sebaliknya.

* *One-to-one relationship*

Hubungan satu ke satu terjadi ketika satu objek A memiliki refrensi dari satu objek B, begitupula sebaliknya.

* *One-to-many relationship*

Hubungan satu ke banyak ini diartikan dengan hubungan antara objek A dan objek B, dimana anggota objek A terhubung dengan lebih dari satu anggota objek B, tetapi anggota dari objek B hanya terhubung dengan satu anggota objek A.

* *Many-to-many relationship*

Hubungan antara objek A dan B, dimana setiap anggota dari objek A maupun objek B memiliki hubungan lebih dari satu objek A dan B

Dalam hubungan asosiasi antara dua objek, terdapat dua bentuk relasi, yaitu agregasi dan komposisi. Kedua bentuk hubungan asosiasi ini dibedakan berdasarkan aturan khusus yang berlaku pada bentuk hubungan tersebut.

* Agregasi

Meskipun sama – sama berupa hubungan yang “memiliki”, perbedaan dengan bentuk komposisi adalah bentuk hubungan ini tidak terikat, yang berarti setiap *class* dapat berdiri sendiri.

* Komposisi

Sebuah hubungan dapat dikatakan komposisi jika sebuah *class* “memiliki” *class*

lainnya. Aturan yang berlaku untuk hubungan ini adalah ketika *class* yang

“memil iki” *class* lainnya dihancurkan, maka *class* yang “dimiliki” oleh *class* tersebut akan hancur juga atau dengan kata lain tidak dapat digunakan.

Bentuk hubungan komposisi lebih kuat dibanding dengan agregasi. Kedua bentuk hubungan ini merupakan bentuk hubungan antar objek yaitu asosiasi.

Hubungan asosiasi dapat dikatakan komposisi jika hubungan tersebut adalah hubungan “memiliki”, dan dikatakan hubungan agregasi jika terdapat objek yang “menggunakan” objek lainnya.

1. **Dependensi**

Hubungan dependensi diartikan sebagai hubungan antara dua buah *class,* dimana satu *class* memiliki ketergantungan dengan *class* lainnya tetapi *class* lainnya tidak/mungin memiliki ketergantungan terhadap *class* pertama tadi.

Hubungan dependensi terjadi apabila sebuah fungsi pada *class* A membutuhkan *class* B sebagai parameter, atau fungsi pada *class* A memiliki nilai kembalian berupa *class* B, atau *class* A menggunakan *class* B tetapi *class* B bukan merupakan sebuah atribut.

Sebenarnya hubungan asosiasi sudah pasti memiliki hubungan dependensi.

Hubungan asosiasi dapat menjadi hubungan dependensi jika *class* yang dibutukan bukan merupakan atribut dari *class* yang membutuhkannya.

1. **Generalisasi**

Dalam hubungan generalisasi, terjadi proses memisahkan karakteristik dari dua atau lebih *class* dan menggabungkannya menjadi satu *class* yang lebih umum, atau biasa disebut **SuperClass**.

1. **Spesialisasi**

Berbanding terbalik dengan generalisasi, hubungan spesialisasi berarti membuat sebuah SubClass dari *class* yang sudah ada. Cara termudah untuk mengenali bahwa kita dapat membuat *class* baru dari *class* yang sudah ada adalah ketika atribut, asosiasi, atau fungsi pada sebuah *class* hanya mengarah pada satu objek di *class* tersebut. Dengan ini kita dapat membuat sebuah *class* tersendiri untuk objek tersebut.

1. **Realisasi dan Implementasi**

Hubungan realisasi atau implementasi adalah hubungan abstraksi khusus antara dua *class*, satu mewakili *class* yang direalisasikan ( *supplier* ) dan yang lainnya mewakili *class* implementasinya ( *client* ).

Realisasi dapat digunakan dalam beberapa hal seperti, untuk menyempurnakan sebuah *class*, optimasi, transformasi, *template*, model sintetis, komposisi kerangka kerja, dll.

*Interface* *realitation* adalah hubungan realisasi khusus antara *classifier* dan *interface*. Hubungan ini menandakan bahwa *classifier* mengimplementasikan satu atau beberapa *class interface. Classifier* mendukung serangkaian fitur yang dimiliki oleh kelas *interface.* Selain mendukung fitur, *class classifier* harus mematuhi batasan yang dimiliki oleh *class interface*.

Dalam mengembangkan sebuah perangkat lunak, kita membutuhkan arsitektur dan desain.

Arsitektur berada pada tingkat tertinggi atau yang dikenal dengan istilah *high-level*, dimana definisinya merupakan sebuah pola arsitektur yang menentukan bentuk dan struktur keseluruhan perangkat lunak.

Desain memiliki posisi di tingkat rendah atau istilahnya adalah *low-level*, di bawah arsitektur. Desain juga memiliki definisi yaitu interkoneksi antara modul dan beberapa entitas perangkat lunak seperti *packages, components,* dan *classes*.

*Software Design Principle* merupakan sebuah pedoman yang dapat digunakan untuk menghindari desainyang buruk saat mengembangkan perangkat lunak.

Menurut Robert C. Martin, terdapat tiga karakteristik penting dari desain yang buruk yang perlu kita perhatikan dan sebaiknya dihindari.

* *Rigidity*

*Rigidity* ( kekakuan ) adalah kondisi suatu sistem yang sulit diubah, bahkan untuk perubahan yang paling sederhana. Saat kita ingin melakukan perubahan, perubahan tersebut menyebabkan ketergantungan untuk mengubah item lain di dalam suatu modul. Alhasil, perubahan yang seharusnya dapat dilakukan dalam waktu yang singkat, malah sebaliknya.

* *Fragility*

Fragility adalah kecenderungan perangkat lunak yang salah di beberapa bagian setiap kali melakukan perubahan. Seringkali kesalahan terjadi di area yang tidak memiliki hubungan konseptual dengan area yang diubah. Sehingga jika melakukan perbaikan, kadang takut timbul kesalahan dengan cara yang tidak terduga.

Ketika fragility ada di dalam suatu perangkat lunak, kemungkinan kesalahan akan meningkat seiring berjalannya waktu. Perangkat lunak semacam itu tak hanya sulit dipelihara, bahkan sulit juga dipertahankan. Saat melakukan perbaikan, alih-alih memperbaiki kesalahan yang ada, sebuah sistem akan menjadi lebih buruk dan menimbulkan lebih banyak masalah.

* *Immobility*

Sebuah ketidakmampuan untuk menggunakan kembali perangkat lunak dari proyek lain atau bagian-bagian dari proyek yang sama. Seorang enginer tentu akan mengalami kondisi di mana ia membutuhkan modul atau sistem yang sebelumnya sudah pernah ditulis atau dibuat oleh enginer lain. Namun, sering juga terjadi bahwa modul yang dibutuhkan tersebut memiliki terlalu banyak bobot yang bergantung padanya. Setelah mencoba untuk memisahkan, para enginer menemukan bahwa pekerjaan dan risiko yang diperlukan untuk memisahkan bagian yang diinginkan (dari bagian yang tidak diinginkan), terlalu besar untuk ditoleransi. Sehingga alih-alih menulis ulang, sang enginer akan menggunakannya kembali untuk projek lain.

**Apa itu S.O.L.I.D**

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

Paradigma sendiri bukanlah sebuah *principle* yang mengajarkan tentang bagaimana sebuah tanggung jawab suatu entitas yang berada di dalam sebuah perangkat lunak.

**Tujuan S.O.L.I.D**

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

1. ***Single Responsibility Principle* (SRP)**

Prinsip ini digunakan untuk mengatur tanggung jawab dari sebuah entitas yang berada di dalam sebuah proyek dalam hal ini adalah sebuah *class*.

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

**Tujuan menggunakan SRP**

Ketika kita ingin melakukan perubahan pada sebuah *class* yang memiliki tanggung jawab yang banyak, perubahan yang akan dilakukan berpotensi untuk mempengaruhi fungsionalitas dan tanggung jawab lain yang saling berkaitan di dalam *class* tersebut.

SRP merupakan cara yang baik untuk mengidentifikasi *class* selama fase desain aplikasi, dan mengingatkan kita untuk memikirkan semua cara agar *class* dapat dikembangkan tanpa adanya masalah berarti.

Pemisahan tanggung jawab yang baik dilakukan hanya ketika sebuah gambaran lengkap aplikasi secara keseluruhan tentang bagaimana aplikasi itu dapat bekerja telah dibuat dan dipahami dengan baik, sehingga kita dapat memisahkannya dengan rinci.

1. **Open/Close Principle (OCP)**

OCP adalah aturan yang mengatur sebuah perangkat lunak harus terbuka untuk ditambahkan tetapi tertutup untuk dimodifikasi.

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.

**Tujuan menggunakan OCP**

Saat menerapkan *open*/*close* *principle* ke dalam project, kita bisa membatasi kebutuhan untuk mengubah kode yang telah ditulis, diuji, dan di-debug. Tujuannya untuk menghindari resiko atau kelemahan sistem yang bisa saja terjadi. Selain itu, kita bisa menghindari ketergantungan dan meningkatkan fleksibilitas sistem. Tentunya ini akan meringankan proses skalabilitas dari sisi pengembangan perangkat lunak.

1. **Liskov Substitution Principle (LSP)**

Liskov’s substitution adalah aturan yang berlaku untuk hirarki pewarisan. Hal ini mengharuskan kita untuk mendesain *class - class* 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.

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.

***Contravariant* dan *Covariant***

Aturan pertama, SubClass harus memiliki sifat 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***

Selanjutnya adalah aturan *preconditions* dan *postconditions*. Ini merupakan tindakan yang harus ada sebelum atau sesudah sebuah proses dijalankan.

**Invariant**

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

**Constraint**

Terakhir, aturan tentang *constraint* dari sebuah SubClass. Secara default, SubClass 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 objek. Sebagai contoh, semisal SuperClass memiliki objek yang memiliki nilai tetap, maka SubClass tidak diijinkan untuk mengubah keadaan dari nilai objek tersebut.

*Liskov’s Substitution principle* merupakan prinsip yang dapat meningkatkan desain dari sistem yang kita kembangkan. Sehingga ketergantungan antar klien dapat disubstitusikan tanpa klien tahu perubahan yang ada.

1. **Interface Segregation Principle (ISP)**

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.

Ketika prinsip *interface segregation* diterapkan, setiap *class - class* akan mengimplementasikan beberapa *interface class* yang lebih kecil sesuai dengan fungsi-fungsi yang dibutuhkan *class - class* tersebut.

Hal ini berarti bahwa *class - class* yang saling bergantung dapat berkomunikasi dengan menggunakan *interface* yang lebih kecil, mengurangi ketergantungan pada fungsi-fungsi yang tidak digunakan dan mengurangi *coupling*. Dengan menggunakan *interface* yang lebih kecil akan memudahkan dalam implementasi, meningkatkan fleksibilitas, dan juga kemungkinan untuk digunakan kembali (reuse).

**Tujuan menggunakan ISP**

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

**Manfaat menggunakan ISP**

Dengan menerapkan *Interface Segregation Principle* dapat membantu kita untuk mengembangkan sistem yang kukuh dan mudah dipelihara. Kita dapat mencegah pembuatan *interface* yang memiliki banyak fungsi untuk kepentingan yang berbeda-beda.

1. **Dependency Inversion Principle (DIP)**

Pada prinsip *Dependency Inversion,* terdapat dua pernyataan atau aturan yang perlu kita ketahui, yang pertama adalah *high-level module* tidak diperbolehkan untuk bergantung pada *low-level module*. Keduanya harus bergantung pada *abstraction*. Pernyataan yang kedua, abstraksi tidak diperbolehkan untuk bergantung pada detail. Detail harus bergantung pada abstraksi.

**Apa yang dimaksud dengan high-level modules dan low-level modules ?**

Agar lebih mudah memahaminya, kita dapat mengkategorikan *class - class* menjadi sebuah hirarki. *High-level modules* adalah *class – class* yang berurusan dengan kumpulan-kumpulan fungsionalitas. Pada hirarki tertinggi terdapat *class – class* 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.