

Clean Code

Silver - Chapter 1 - Topic 6

**Selamat datang di Chapter 1 Topic 6
online course Back End Java dari
Binar Academy!**

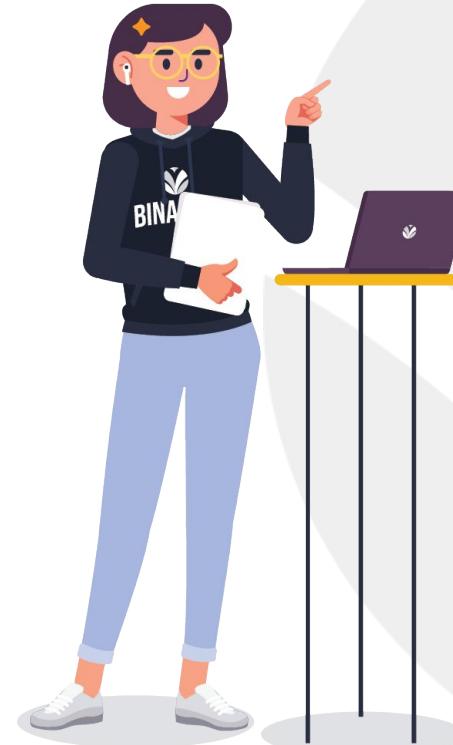


Haloo, kita ketemu lagi! 😺

Pada topik sebelumnya, kita udah belajar tentang Working as A Backend Engineer.

Pada topik ini, kita bakal ngobrolin tentang **Clean Code**. Pembahasan kita nggak akan jauh jauh dari konsep S.O.L.I.D OOP, contoh dari good code dan bad code, sampai cara buat melakukan refactor.

Kalau gitu langsung aja kita buka tirai selanjutnya~



Dari sesi ini, kita bakal bahas hal-hal berikut:

- Konsep S.O.L.I.D OOP
- Contoh good code dan bad code
- Cara melakukan refactor yang baik menurut buku Clean Code



Untuk membuat code yang baik, kita perlu tahu prinsip-prinsip yang biasanya dipakai sama seorang programer.

Nah, si Uncle Bob ngenalin kita sama prinsip **S.O.L.I.D.** buat make it happens.

Kira-kira prinsipnya kayak gimana, yaaa?



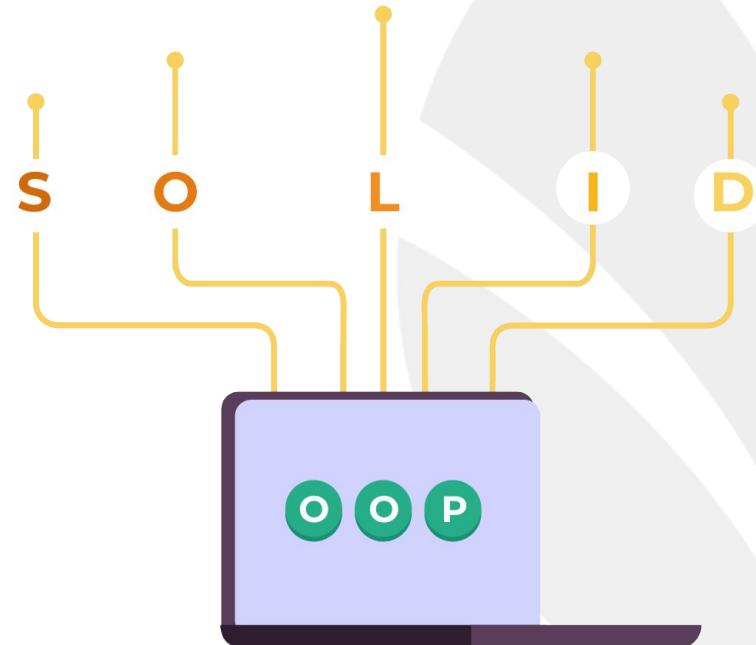
S.O.L.I.D itu apa, sih?

S.O.L.I.D Principle merupakan prinsip-prinsip yang membantu programmer ketika doi menggunakan bahasa pemrograman berbasis OOP.



Kalau menerapkan S.O.L.I.D principle, **kodingan yang ditulis bakal lebih bersih, robust, dan gampang buat dipelihara.**

Dari situ, penting banget untuk seorang programmer yang pakai Java buat memahami prinsip S.O.L.I.D ini.

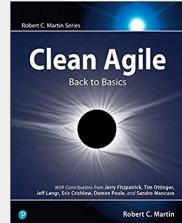
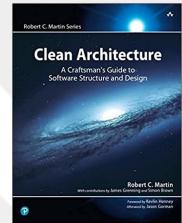
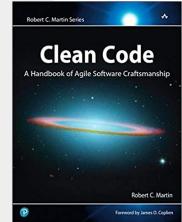


Seseorang yang ada dibalik S.O.L.I.D Principle~

S.O.L.I.D Principle dicetuskan oleh **Robert C. Martin** atau lebih dikenal dengan sebutan **Uncle Bob**.

Uncle Bob merupakan seorang software engineer kelahiran 1952 yang terkenal dengan bukunya yang berjudul **Clean Code**, **Clean Architecture** dan **Clean Agile** yang sering jadi panduan buat programmer OOP.

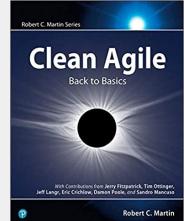
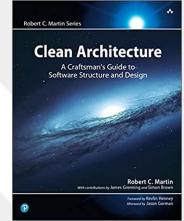
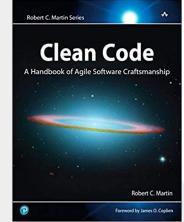
Kamu udah pernah baca bukunya Uncle belum?



Kamu udah pernah baca bukunya Uncle belum?



Selain itu, Uncle Bob juga merupakan salah satu founder Agile manifesto yang jadi pondasi dari Agile methodology.



Single Responsibility

Open for extension & close for
modification principle

Liskov Substitution Principle

Interface Segregation Principle

Dependancy Inversion Principle

**Bukan cuma sekedar huruf,
S.O.L.I.D itu merupakan akronim!**

Yap, sebagaimana gambar disamping, ada arti dibalik setiap huruf S.O.L.I.D.

Kita jelasin lebih detail setelah slide ini, ya!

Single Responsibility Principle (S)

Prinsip ini merupakan sebuah kutipan dari uncle Bob yang berarti **sebuah class cuma diperbolehkan buat punya satu tanggung jawab aja.**

Contohnya, pas kita bikin service buat men-generate file pdf dan service buat mengirimkan email. Keduanya pasti punya tanggung jawab yang berbeda, kan?

Jadi, sebaiknya kedua service tersebut harus terpisah supaya lebih optimal.

“A class should have only one reason to change.” — Uncle Bob



“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”
— Bertrand Meyer

Open For extension, Closed For Modification Principle (O)

Tujuan dari Open Closed Principle adalah untuk **memastikan code existing nggak tersenggol sama code yang baru.**



Dengan melakukan Open Closed Principle, kita nggak perlu melakukan regression testing di modul, atau mengubah unit test di code yang lama.

Dari situ, setiap ada perubahan baru, diusahakan nggak sampai melakukan modifikasi pada class atau method yang existing.

Kunci buat menerapkan prinsip ini adalah dengan **bikin abstraksi yang baik di awal**.

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”
— Bertrand Meyer



Misalnya gini, ada service untuk melakukan generate file.

Code existing cuma bisa menghandle buat generate file berekstensi .pdf aja, sedangkan fitur buat men-generate file berekstensi .xlsx bakal didevelop.

Sebaiknya, dari awal udah dibikin abstract class buat bikin service generate file yang nantinya bisa jadi class-class turunannya, gengs.

Tujuannya, supaya existing class kayak class buat meng-generate file berekstensi .pdf nggak perlu dimodifikasi, tapi bikin class baru.



Liskov Substitution principle (L)

Prinsip ini berkaitan sama inheritance. Jadi, **super class harus bisa diinstance sama subclassnya.**

Prinsip ini merupakan prinsip terusan dari Open Closed Principle.

Buat melakukan implementasi prinsip ini, validasi di masing-masing subclass nggak diperbolehkan dibikin lebih strict daripada superclass. Di samping itu, subclass harus mewariskan rules yang mirip sama superclassnya.



“Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” — Uncle Bob

Interface Segregation Principle (I)

Maksud dari Interface Segregation principle ini, yaitu:

Class yang mengimplementasikan sebuah interface, jangan sampai mengimplementasikan method yang nggak diperlukan.

Apalagi kalau sampai memaksakan kode.



“Clients should not be forced to depend upon interfaces that they do not use.”
– Uncle Bob

Misalnya, ada sebuah class yang meng-override banyak method dari suatu interface, tapi dalam implementasinya di salah satu method, content dari method tersebut nggak dituliskan karena merasa method tersebut nggak perlu digunakan.

Contoh tersebut melanggar Interface Segregation.

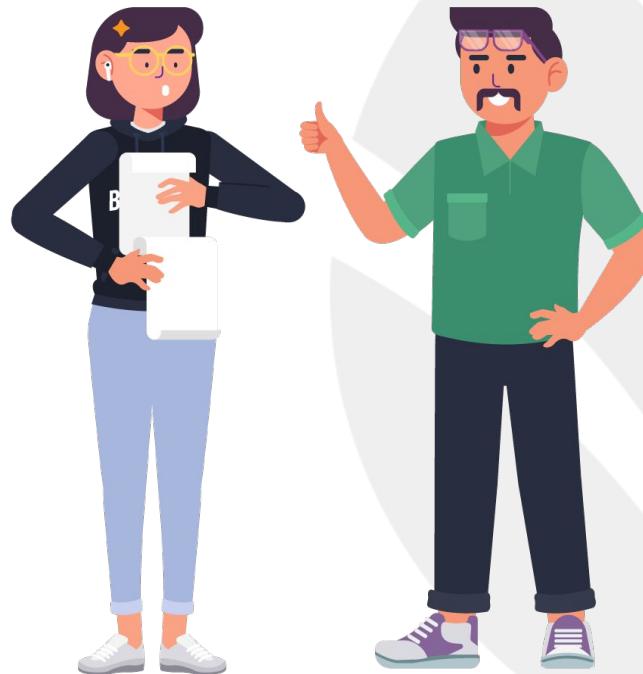


“Clients should not be forced to depend upon interfaces that they do not use.”
– Uncle Bob

“Terus, gimana dong caranya buat menerapkan prinsip ini?”

Caranya, yaitu nggak bikin interface dengan method yang terlalu banyak.

Dengan begitu, class yang mengimplementasikan interface tersebut hanya menggunakan method yang diperlukan aja.



Dependency Inversion Principle (D)

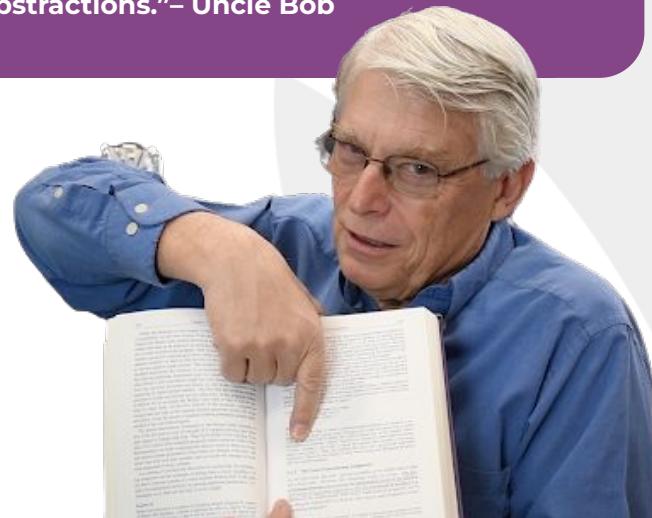
Satu patokan dalam Dependency Inversion, yaitu:

Modul level tinggi nggak boleh bergantung pada modul level rendah.

Yang dimaksud modul level-tinggi (high-level) adalah interface atau abstraksi. Sedangkan yang dimaksud modul level-rendah (low-level) adalah modul turunan atau yang membantu modul high level.

Tujuan dari larangan ini supaya dependency inversion jadi modul independen.

“Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.” – Uncle Bob



Setelah kenalan sama Uncle Bob, kita bakal praktik implementasi kode.

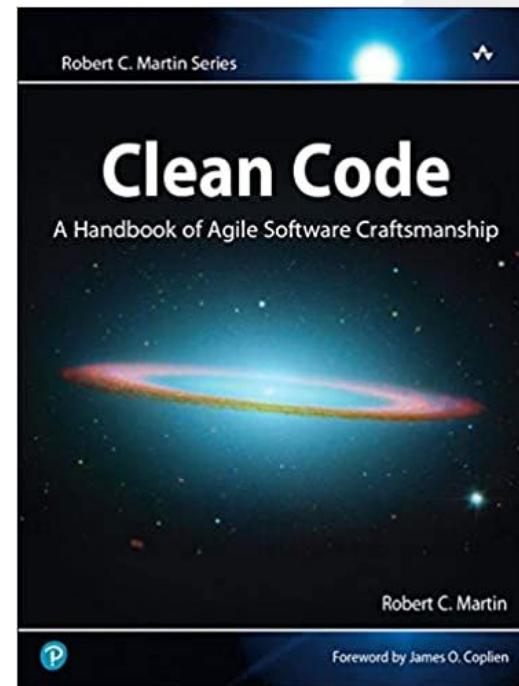
So, selamat datang di materi Java Best Practice. Lezgow~



Langsung aja! Ayo kita kupas tuntas buku Clean Code ini!

Di samping adalah buku dari Uncle Bob yang berjudul Clean Code.

Kita bakal cari tahu gimana code yang buruk dan gimana cara bikin code ideal berdasarkan buku Clean Code-nya Uncle Bob.



Kalau menurut Uncle Bob, **Clean Code** adalah **kumpulan code yang mudah dibaca, mudah dimengerti, dan mudah di-maintenance**. Baik dipahami oleh developernya sendiri ataupun yang lain.

Bukunya udah mirip kayak cara buat memahami perempuan aja, ya 😊



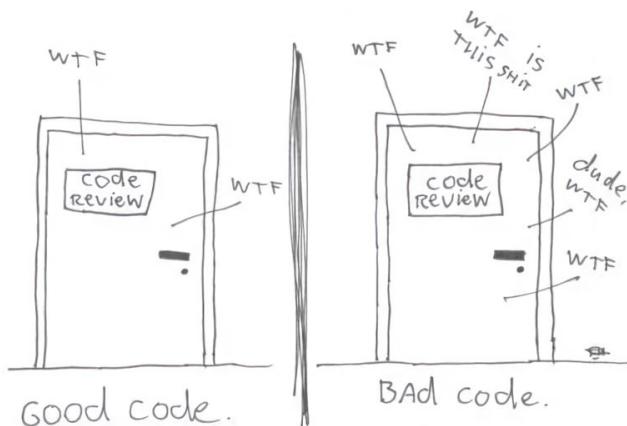
Kalau definisi di bukunya tertulis kayak gini, nih:

Clean Code adalah kode di dalam perangkat lunak (software) yang formatnya udah benar dan disusun dengan baik dan rapi.

Untuk apa disusun rapi? Supaya kode programnya mudah dibaca, dimengerti, ditelusuri, dan di-customize sama developer lain kalau nantinya ada perubahan.



The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE

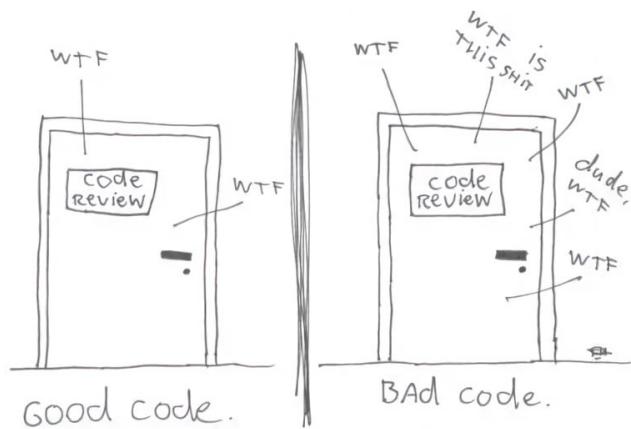


(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

Parameter dari kualitas sebuah kode dibilang baik atau belum adalah seberapa banyak developer nemu **code yang membingungkan**.

Buat menguji kualitas suatu code, kita bisa melakukan **code review**.

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

Gambar di samping adalah joke di kalangan developer tentang cara mengukur kualitas kode menggunakan WTF/minutes (WTFPM), yang merupakan singkatan dari "**Works that Frustrate**".

Cara kerjanya adalah berapa banyak code yang dapat dibaca developer per menitnya.

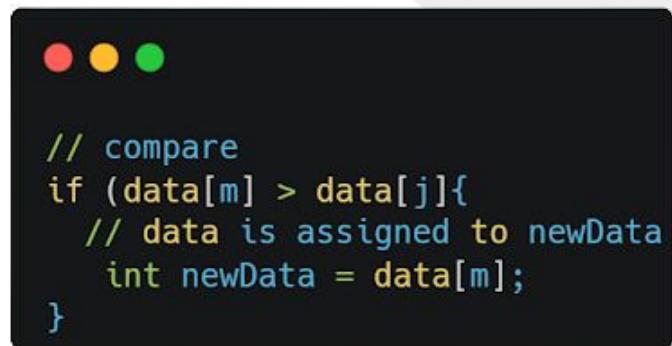
Berikut adalah penerapan dalam coding yang bikin sebuah code jadi buruk dan cara mengatasinya!

- **Menulis Comment yang Berantakan dan nggak jelas**

Contohnya ada di samping, ya.

Sebaiknya **jangan memberi comment untuk logika yang simple**.

Comment untuk memberikan penjelasan bisa dilakukan per method aja atau di bagian yang sulit aja. Jadi, nggak perlu bikin comment di setiap line. Selain itu buat menjelaskan cara run aplikasi juga bisa dijelaskan pakai file README.



```
// compare
if (data[m] > data[j]){
    // data is assigned to newData
    int newData = data[m];
}
```

- **Mengikuti Teknologi yang Sedang Hype**

Kita nggak perlu selalu pakai teknologi yang terbaru atau kekinian karena mengikuti hype.

Selalu pakai teknologi terbaru bisa bikin orang kagum sih, tapi sebenarnya dalam penerapannya nggak bakal produktif.

Kan percuma~



- **Melakukan Semuanya Sendirian**

Cara melakukan software development secara lambat adalah melakukan segalanya secara sendirian, Sob.

Kamu harus tahu nih kalau partner itu termasuk aset kita, lho.

Jadi, kalau saling berkolaborasi, kita bisa sekaligus mempelajari sesuatu dengan lebih mudah.



#BAHANRENUNGAN

**“Ngoding itu susah.
Apalagi sendirian.
Udah, gitu aja.”**

- **Menulis Method Terlalu Panjang**

Menulis method content yang terlalu panjang dan kompleks bakal bikin developer lain sulit buat memahami isi method tersebut, Gengs.

Jadi, yang singkat-singkat aja, ya.



- **Pakai Nama Variable dan Method yang Nggak Punya Arti**

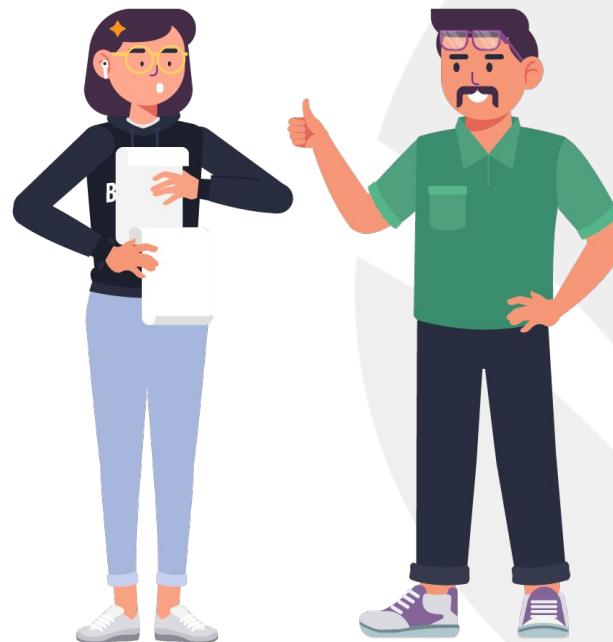
Coba deh kamu lihat contoh code di samping. Ada **int q** dan **int r yang gak punya arti** alias cuma developer yang buat codenya aja yang paham. Hal ini bakal membingungkan orang lain yang baca codenya.



```
int q = getData1();  
int r = getData2();  
double t = q / r;  
saveData3(t);
```

Ketika membuat nama variable dan method, itu harus clear, nih. Karena hal ini berpengaruh ke thread.

Kalau nama variable dan method nggak clear dan dipakai berulang jadinya nggak bagus, karena bakal makan memory.



- **Membuat Semua Variable Jadi Global Variable**

Hal ini bakal bikin variable jadi sulit buat di-debug.

- **Null Sebagai Return Value**

Nggak memberikan return atau response yang jelas pas terjadi error, tapi menggantinya dengan null aja bakal bikin user jadi bingung.

So, jangan sampai kayak gitu, ya.



- **Selalu Meninggalkan Jejak**

Melakukan testing, file dummy atau variable dummy yang nggak dibersihkan bakal membahayakan code di production, lho.

Ibaratnya kayak kamu makan pisang tapi kulitnya nggak dibuang di tempat sampah, pasti nanti bakal membahayakan orang lain yang lewat.



- **Tidak Menuliskan Unit Test**

Ibarat sebuah tanda, menuliskan unit test bisa jadi tanda buat kita kalau code tersebut udah di-test oleh developer atau belum.

Jadi, jangan lupa buat menuliskan unit test-nya ya, gengs.



- **Bikin Banyak Repository pada Git Buat Tujuan yang Sama.**

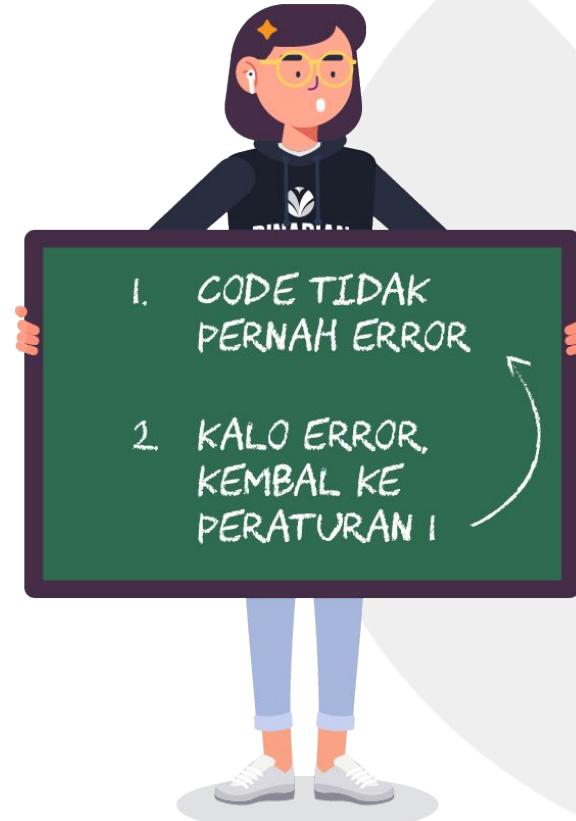
Selain membuang resource, hal ini juga bakal membingungkan developer lain yang berkolaborasi.



- **Selalu Percaya Bahwa Suatu Code Nggak Punya Error**

Sebaiknya developer harus selalu skeptis pada suatu code.

Kayak pepatah yang bilang “nggak ada yang sempurna di dunia ini”, begitu juga sama code yang pasti punya sebuah error pas dijalankan.

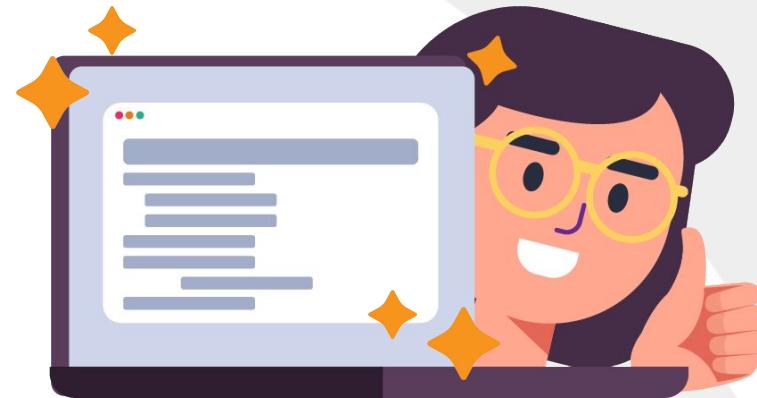


- **Nggak Memperhatikan Estetika**

Estetika yang dimaksud disini bukan tentang keindahan dalam karya seni lho ya, tapi berkaitan sama indentation.

Jadi, **dengan memperhatikan estetika, readability dari suatu code bisa meningkat.**

Selain itu, kalau pakai indentation yang konsisten, kita bisa bagi line code jadi suatu paragraf atau grouping (memisahkan satu space antara line) buat mempercantik code.



- **Nggak Bikin Dokumentasi**

Kalau nggak bikin dokumentasi, waktu bakal terbuang buat mengecek ulang code terlebih dahulu buat cari tahu gimana cara kerja code tersebut.

Jadi, selalu bikin dokumentasi supaya nggak kerja dua kali, ya.



- **Ketidakkonsistenan Terhadap Rule**

Nggak konsisten ini misalnya tuh:

1. Nggak konsisten terhadap versi dan tools yang dipakai,
2. Nggak konsisten terhadap kesepakatan dalam coding yang udah ditetapkan di tim.

Kalau nggak konsisten, bikin ketidakseragaman yang akhirnya menambah kompleksitas dari sebuah code. So, jangan labil kayak orang yang lagi PDKT, ya.



Kalau tadi udah cerita tentang Uncle Bob dan bukunya, sekarang kita bakal bahas tokoh lainnya yang punya buku juga.

Namanya Martin Fowler yang punya buku berjudul **Refactor**.

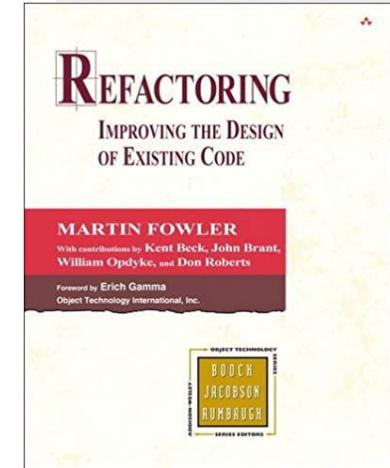
Kalau gitu, yuk kita bahas!



Martin Fowler dan bukunya yang berjudul Refactoring~

Menurut buku Martin Fowler, refactoring adalah proses mengubah sistem dari suatu software tanpa mengubah fungsi dari segala fitur software tersebut.

Dengan kata lain, refactoring adalah sebuah **teknik untuk meningkatkan desain code kita supaya code menjadi mudah dibaca dan di-maintain**.



Refactoring juga bisa dibilang sebagai langkah buat **“membersihkan” code** supaya bisa meminimalisir munculnya bug di software yang di-develop.

Secara umum, tujuan dari refactor adalah buat meningkatkan kualitas desain dari perangkat lunak kita, gengs.



Me trying refactor my code

Hati-hati, code yang buruk bisa jadi bom waktu di kemudian hari, sob!

Biasanya karena proses development yang terburu-buru, code yang ditulis bakal jadi jelek dan jadi **technical debt**.



"Technical debt? apaan tuh?"

Technical debt adalah 'hutang' yang muncul ketika developer memilih untuk memprioritaskan proses rilis sebuah proyek (telah berfungsi sebagian besar) daripada menyempurnakan proses coding-nya.

Code yang nggak sempurna di dalam system yang udah dibikin ini nantinya bakal jadi rumit banget.



“Emang apa sih akibat dari Technical debt?”

Pas system udah jadi sangat besar, developer bakal butuh waktu lebih lama dalam melakukan development karena harus memahami banyak proses yang rumit.

Soalnya gini, kalau ada fitur yang bisa dikerjakan dalam waktu 5 hari, eh karena ada code yang kompleks atau salah malah bisa jadi 3 minggu 😞



Dalam buku **The Pragmatic Programmer**, ada satu teori yang dibahas, yaitu **Broken Window Theory!**

Penjelasannya gini, pada suatu gedung ada satu jendela pecah. Terus karena cuma satu yang pecah jadi nggak ada satupun yang peduli.

Begitu terus sampai semua jendela yang ada di gedung tersebut pecah satu persatu dan nggak ada satu pun yang memperdulikan.



Kalau dikaitkan sama software development, bisa jadi seorang developer melihat suatu implementasi code yang nggak punya unit test. Dari situ developer ikutan tuh bikin code yang nggak punya unit test.

Dia praktekin untuk code yang lain, sampai akhirnya nggak ada satupun yang bikin unit test.

Berlaku pula dengan code yang jelek, bisa jadi menumpuk karena nggak ada yang peduli.



Buat mengatasi technical debt tersebut, refactoring adalah jalan keluarnya!

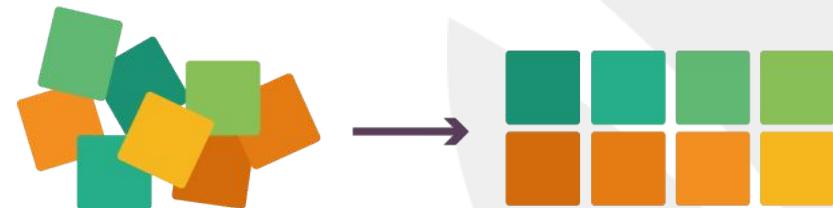
“Di mana sih refactoring diperlukan?”

Biasanya, **refactoring dilakukan di legacy code.**



“Terus, apa iya code tersebut harus berupa legacy code?”

Nggak harus kok, tapi legacy code yang buruk biasanya harus dilakukan refactor secara besar-besaran.



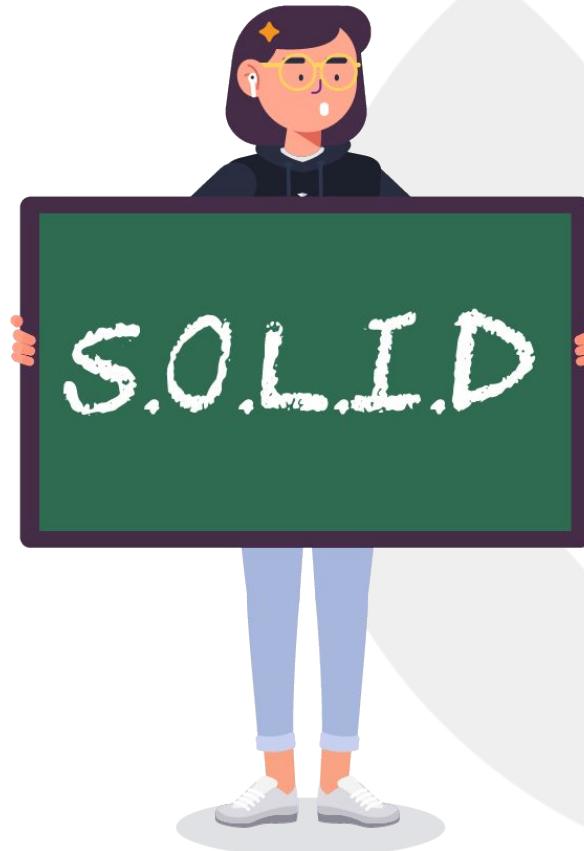
Sebagai junior programmer, kita bisa berupaya menghasilkan code yang berkualitas nih, Sob!

Ada beberapa hal yang bisa dilakukan untuk meningkatkan kualitas dari code Java yang dibuat, yaitu:

- Menginstall [**Sonarlint**](#) buat melihat code smell dengan mengupayakan supaya code smell seminimal mungkin.



- Mempelajari penggunaan design pattern
- Mempelajari prinsip S.O.L.I.D yaitu dengan menerapkan dan memahami prinsip S.O.L.I.D.
- Menerapkan clean code buat memastikan penggunaan best practice pas melakukan coding.

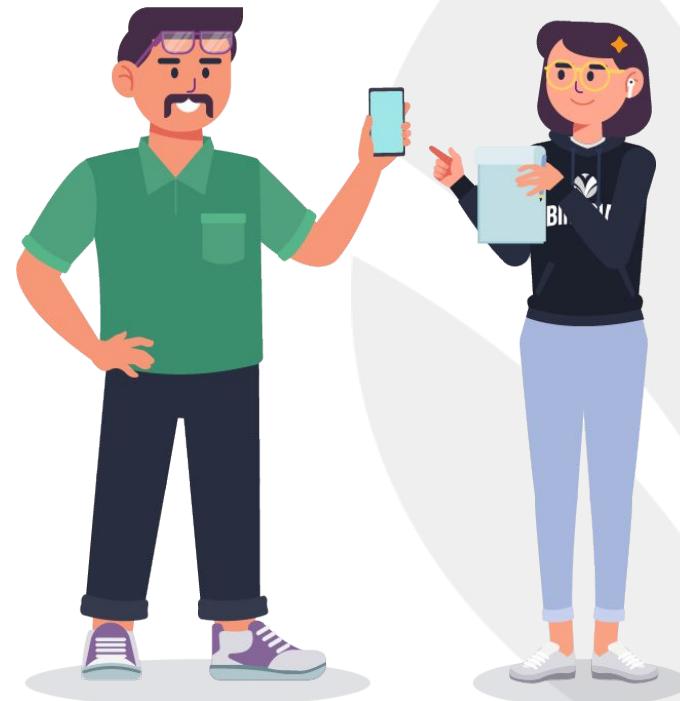


- Membudayakan code review

Meminta review teman ataupun senior terhadap code yang udah dibuat dan meminta feedback.

Setiap mendapat feedback, segerakan refactor sebelum code naik ke production.

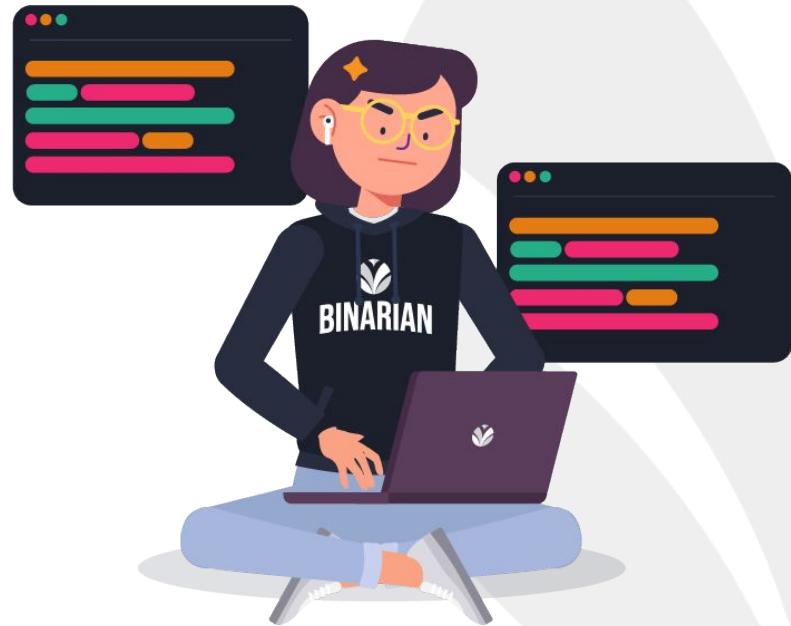
Selain meminta review, kita juga harus membiasakan diri buat melakukan code review punya rekan satu tim. Hal ini bakal meningkatkan awareness kita terhadap bad practice dan menambah kamus kita buat menerapkan good practice.



Pas mau melakukan refactor sebuah code, ada beberapa hal yang perlu diperhatikan nih!

Hal-hal tersebut, yaitu:

- Perhatikan code yang nggak baik, yang harus dilakukan refactor, pastikan fungsi-fungsi dari code sebelum di-refactor.
- Perhatikan effort yang diperlukan buat melakukan refactor, tetapkan juga waktu penggeraan buat melakukan refactor.



- Terapkan clean code yang udah dimengerti.
- Membereskan code smell.
- Bikin unit test dari code yang udah diperbaiki.
- Pastikan code yang diperbaiki nggak mengubah business logic-nya.



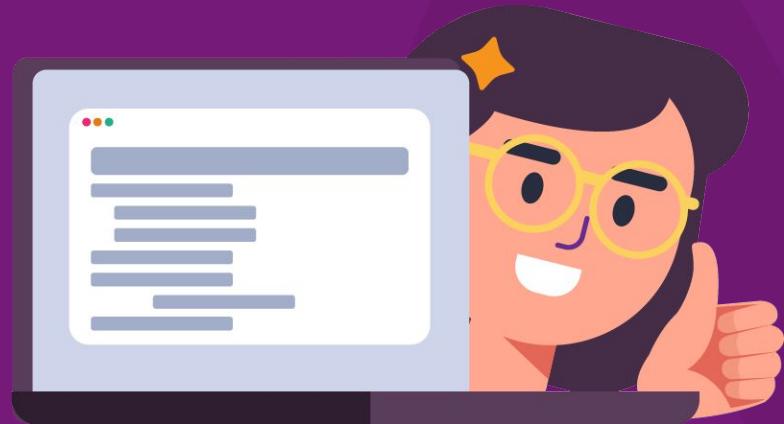
Terusss...

- Lakukan code review.
- Lakukan regression testing terhadap modul-modul yang terkena impact.
- Buat dokumentasi setelah di-refactor.



Yeay! Seperti itulah tips dan trik untuk bisa membuat code yang gak cuma yang penting bisa dijalankan, tapi juga cantik dan juga gak bikin orang lain bingung bacanya.

Coba deh kamu bayangin, selain dari sisi developer, siapa lagi sih pihak yang akan juga akan dirugikan dengan adanya technical debt?



Referensi dan bacaan lebih lanjut~

1. Martin, Robert C. 2008. Clean Code. Massachusetts: Pearson Education
2. Martin, Robert C. 2017. Clean Architecture. Massachusetts: Pearson Education
3. Martin, Robert C. 2019. Clean Agile. Massachusetts: Pearson Education
4. Fowler, Martin, Kent Beck. 1999. Refactoring. Massachusetts: Pearson Education
5. Thomas, David, Andrew Hunt. The Pragmatic Programmer. 2000. Massachusetts: Addison-Wesley Professional



Nah, selesai sudah pembahasan kita di Chapter 1 Topic 6 ini.

Selanjutnya, kita bakal bahas tentang Version Control.

Penasaran kayak gimana? Cus langsung ke topik selanjutnya~

