BAB 2
LANDASAN TEORI

2.1 *Point of Sales*

Point of Sales (POS) merupakan tempat pembayaran para pelanggan dalam membeli sebuah produk. Biasanya POS berbentuk mesin, *tablet*, *smarpthone*, mesin EDC atau perangkat lainnya yang digunakan untuk transaksi di toko. Di Indonesia mungkin sebutan yang paling umum adalah mesin kasir [11]. Aplikasi POS tidak hanya membantu kinerja POS harian dan mingguan, tetapi juga tingkat inventaris berdasarkan SKU dan lokasi, status pesanan, persentase stok, serta gudang dan penyimpanan stok [12]. Namun pada kenyataannya POS tidak semua sama. Para pelaku usaha tidak menerapkan semua fungsi POS karena kurangnya kemampuan teknis [12].

2.2 Android

Android adalah sebuah sistem operasi *mobile* yang telah hadir selama 15 tahun. Android pada umumnya sebagai basis dari sistem operasi telepon genggam dan *tablet* di seluruh dunia. Pemilik dari sistem operasi Android adalah Google. Bagaimanapun, sistem operasi ini bersifat *open source* yang berarti kode program dari Android bebas diakses oleh siapa pun bahkan untuk penggunaan komersial [13].

Dengan sifatnya yang *open source* ini memudahkan pengembang aplikasi Android untuk menggunakan kode program yang dibuat oleh Google. Kemudahan ini mengakibatkan Android mendapatkan banyak sekali jutaan dukungan aplikasi berbayar ataupun gratis yang dapat diunduh melalui Google Play Store [14].

2.2.1 Versi Android

Sejak rilis pertama Android hingga saat ini, Android sudah bertransformasi secara tampilan, konsep, hingga fungsi. Berikut ini adalah daftar versi android juga sebutan tiap versi rilisnya [15]:

1. 1.0 – 1.1 Tidak memiliki sebutan.
2. 1.5 : Cupcake

3. 1.6 : Donut
4. 2.0 – 2.1 : Eclair
5. 2.2 : Froyo
6. 2.3 : Gingerbread
7. 3.0 – 3.2 : Honeycomb
8. 4.0 : Ice Cream Sandwich
9. 4.1 – 4.3 : Jelly Bean
10. 4.4 : KitKat
11. 5.0 – 5.1 : Lollipop
12. 6.0 : Marshmallow
13. 7.0 – 7.1 : Nougat
14. 8.0 – 8.1 : Oreo
15. 9.0 : Pie
16. 10 : Tidak memiliki sebutan.
17. 11 : Tidak memiliki sebutan.
18. 12 : Tidak memiliki sebutan.

2.3 Bahasa Pemrograman Java

Java merupakan bahasa pemrograman tingkat tinggi, kokoh, berorientasi obyek dan aman [16]. Java pertama kali dirilis oleh Sun Microsystems ditahun 1995. Program java dapat dengan mudah ditemukan dibanyak jenis perangkat dari ponsel pintar hingga komputer *mainframe.* Java tidak dikompilasi ke bahasa *processor* asli tetapi membutuhkan sebuah "*virtual machine*" yang mengerti hasil kompilasi java yang bernama java bytecode. Untuk menjalankan program java memerlukan implementasi "*virtual machine"* [17].

2.4 *Cyclomatic Complexity*

McCabe's Cyclomatic Complexity merupakan alat ukur untuk menghitung *control flow* dari sebuah modul. Jika nilai dari *cyclomatic complexity* semakin tinggi maka modul tersebut akan semakin sulit untuk diuji dan dirawat [18].

Pengukurannya didasarkan pada teori grafik dan juga dihitung menurut karakteristik program seperti yang ditangkap oleh grafik aliran programnya [4].

Dalam perhitungannya, jika sebuah modul tidak memiliki percabangan maka kompleksitasnya satu [19]. Apabila modul memiliki satu atau lebih percabangan, maka perhitungan *Cyclomatic Complexity* dapat mengikuti persamaan rumus berikut:

$$V(g) = E - N + 2$$

Keterangan:

$V(g) = Cyclomatic\ Complerxity$

$E = Jumlah\ edge\ pada\ grafik$

$N = Jumlah\ node\ pada\ grafik$

Pada suatu modul, disarankan memiliki jumlah *Cyclomatic Complexity* pada kurang dari 15. Apabila melebihi, maka modul tersebut memiliki jalur eksekusi yang banyak sehingga semakin sulit untuk diidentifikasi dan diperiksa. Jumlah jalur eksekusi tertinggi dalam sebuah modul disarankan 100 [19]. Rentang nilai *cyclomatic complexity* beserta risiko[20] dapat dilihat pada **Tabel 2-1** Rentang Nilai *Cyclomatic Complexity***.**

Tabel 2-1 Rentang Nilai *Cyclomatic Complexity*

Rentang	Keterangan
1 - 10	Risiko Minimal
11 - 20	Risiko Sedang
21 - 50	Risiko Tinggi
> 50	Risiko Sangat tinggi

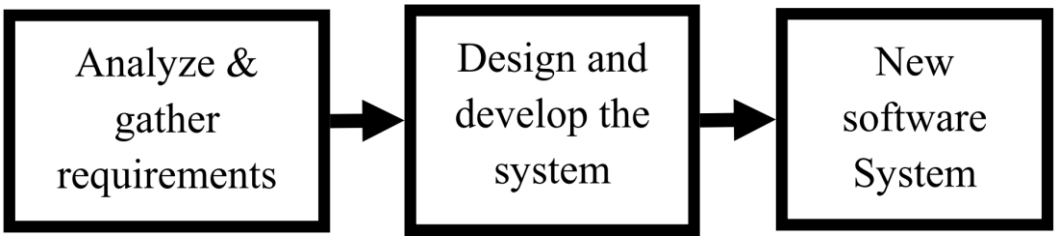
2.5 ButterKnife

ButterKnife adalah pustaka Android yang dikembangkan oleh Jake Wharton untuk membantu meringankan tugas pengembang aplikasi Android karena mampu menyederhanakan penulisan komponen tampilan di Android. Pustaka ButterKnife digunakan ketika mendeklarasikan sebuah komponen tampilan, maka pengembang

aplikasi tidak perlu menghubungkan komponen tampilan tersebut dengan fungsi *findViewById()* [21]. Fungsi tersebut digantikan oleh anotasi *@BindView* oleh ButterKnife [7].

2.6 *Re-engineering* Perangkat Lunak

Proses *re-engineering* dari perangkat lunak adalah proses memodifikasi dan menyusun ulang perangkat lunak supaya lebih mudah untuk dipelihara. Proses ini merupakan bagian dari menulis ulang atau membuat struktur baru untuk keseluruhan sistem lama tanpa mengubah fungsionalitas dari sistem tersebut [22]. Pada dasarnya, proses dari *re-engineering* itu terdiri dari memeriksa kembali, menganalisis isi dari perangkat lunak, memperbaiki isi dari perangkat lunak yang perlu diperbaiki berdasarkan kebutuhan, dipecah-pecah, lalu pada akhirnya digabungkan menjadi bentuk yang baru [23]. Struktur umum dari *re-engineering* perangkat lunak dapat dilihat pada **Gambar 2-1** Struktur Umum dari *Re-engineering* Perangkat Lunak**.**



Gambar 2-1 Struktur Umum dari *Re-engineering* Perangkat Lunak

2.6.1 Taksonomi *Re-engineering* Perangkat Lunak

Berikut ini adalah sub-bagian dari taksonomi dan ruang lingkup domain dari *re-engineering* perangkat lunak:

1. *Forward Engineering*

   Forward Engineering adalah prosedur tradisional yang dilakukan pada suatu perangkat lunak dari abstraksi *high-level* dan *logical design* implementasi hingga implementasi fisik [23].

2. *Reverse Engineering*

   Reverse Engineering adalah proses menganalisis sistem perangkat lunak untuk melihat keterkaitan perangkat lunak dan mekanismenya juga untuk menghasilkan representasi perangkat lunak dari perangkat lunak yang sedang berjalan dalam struktur yang berbeda atau di tingkat abstraksi yang lebih tinggi [23].

3. *Re-documentation*

   Re-documentation adalah sub-area dari *reverse eingineering.* Tujuan dilakukannya *re-documentation* adalah untuk memulihkan dokumentasi yang hilang atau tidak ada dari level abstraksi seperti dokumentasi *data flow*, struktur data, dan *control flow* [24].

4. *Reverse Design or Design Recovery*

   Reverse Design or Design Recovery adalah membuat ulang abstraksi desain yang merupakan gabungan dari desain dokumentasi, kode, pengalaman pribadi, domain aplikasi dan informasi umum masalah [23].

5. *Program comprehension or Program understanding*

   Sebuah terminologi *Program comprehension or Program understanding* erat kaitannya dengan proses *reverse engineering*. *Program comprehension* secara harfiah pemahaman dimulai dengan program sementara *reverse engineering* dapat dimulai pada struktur perangkat lunak yang dapat dieksekusi dan biner atau pada deskripsi desain perangkat lunak yang canggih [23].

6. *Restructuring*

   Restructuring adalah proses mengubah data dari suatu struktur representasi ke struktur yang berbeda pada tingkat abstraksi komparatif

yang sangat mirip sekaligus melindungi semantik dan fungsionalitas sistem perangkat lunak yang ada [23] .

7. *Recode*

   Recode adalah proses yang terdiri dari pengubahan implementasi karakteristik program. Merestrukturisasi *control flow* dan pengubahan translasi bahasa pada program level [23].

8. *Re-design*

   Re-design adalah proses mengubah karakteristik desain. Perubahan kelayakan termasuk peningkatan algoritma, membuat struktur baru arsitektur desain, mengubah model sistem data di basis data atau dalam struktur data [23].

9. *Respecify*

   Respecify adalah proses yang terdiri dari mengubah karakteristik dari sebuah kebutuhan. Perubahan semacan ini mungkin hanya mengubah struktur kebutuhan yang ada saja [23].

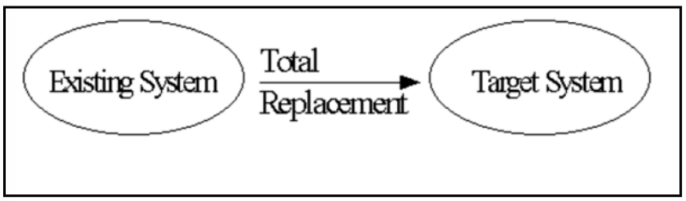
2.6.2 Pendekatan *Re-engineering* Perangkat Lunak

Berikut ini adalah pendekatan-pendekatan *re-engineering* perangkat lunak:

1. Pendekatan *Big Bang*

   Nama lain dari pendekatan ini adalah "*Lump Sum*"; Yang mana keseluruhan perangkat lunak akan diganti. Penggunaan pendekatan ini adalah ketika perangkat lunak memerlukan penyelesaian masalah segera. Contohnya saat perangkat lunak perlu dimigrasikan ke suatu arsitektur yang berbeda. Pendekatan ini terlihat pada **Gambar 2-2** Pendekatan *Big Bang*. Singkatnya, keuntungan dari pendekatan ini adalah perangkat lunak memungkinkan untuk berjalan dilingkungan baru tanpa membutuhkan integrasi *interfaces*. Pendekatan ini juga memiliki kekurangan yaitu tidak cocok digunakan pada sistem yang

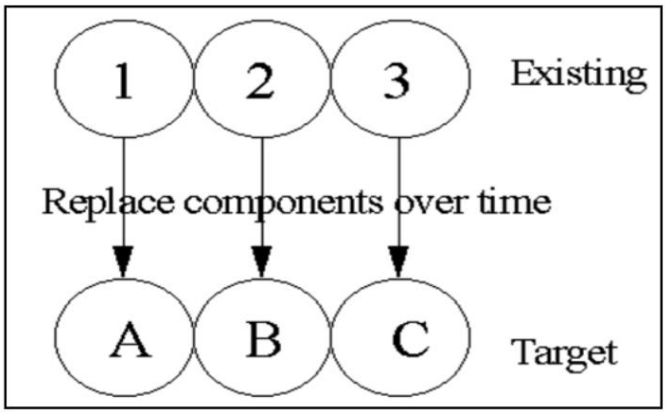
besar karena akan menghabiskan banyak waktu dan sumber daya sebelum sistem target berhasil diterapkan [3].



Gambar 2-2 Pendekatan *Big Bang*

2. Pendekatan *Incremental*

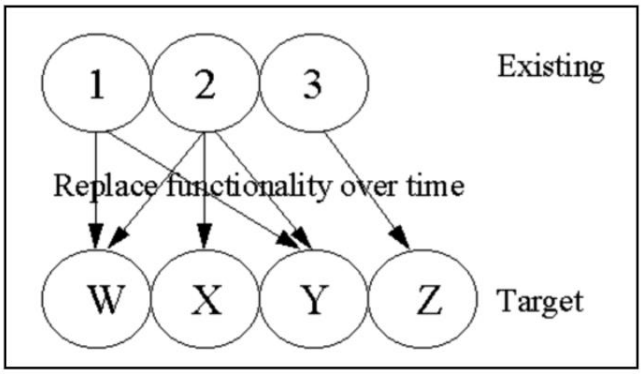
   Nama lain dari pendekatan ini adalah "*Phase-out*" atau "*Additive*". Seperti yang dapat dilihat pada **Gambar 2-3** Pendekatan *Incremental* pada pendekatan ini setiap bagian dari sistem dilakukan *re-engineering* kemudian dilakukan penambahan menjadi versi baru agar kebutuhan utama terpenuhi. Pendekatan ini memiliki keunggulan yang mana bagian – bagian dari sistem yang dilakukan *re-engineering* lebih cepat karena *simplicity* dalam pelacakan kesalahan. Keunggulan ini lah yang menjadikannya rendah risiko dibandingkan pendekatan Big Bang. Pihak *client* juga lebih suka dengan pendekatan ini karena setiap bagian yang selesai bisa secara cepat disampaikan. Namun pendekatan ini juga memiliki kekurangan yakni penyampaian sistem utuh akan membutuhkan waktu yang cukup lama juga perubahan struktur secara keseluruhan selain bagian yang dilakukan *re-engineering* tidak dimungkinkan untuk diubah [3].



Gambar 2-3 Pendekatan *Incremental*

3. Pendekatan *Evolutionary*

   Pendekatan ini serupa dengan pendekatan *incremental* yang mana setiap komponen pada perangkat lunak yang lama diganti dengan sistem yang baru, seperti yang dapat dilihat pada **Gambar 2-4** Pendekatan *Evolutionary*. Namun pergantian dilakukan berdasarkan fungsionalitas bukan dari pengubahan struktur dari perangkat lunak yang lama. Pada pendekatan ini tim pengembang berfokus pada pembuatan komponen *cohesive* [3].



Gambar 2-4 Pendekatan *Evolutionary*

2.7 *Enhanced Re-engineering*

Enhanced Re-engineering adalah mekanisme *re-engineering* perangkat lunak yang memanfaatkan banyak keunggulan dari banyak metode dan level abstraksi

untuk mengubah perangkat lunak yang sudah ada ke perangkat lunak yang baru. *Enhanced Re-engineering* menggunakan kedua teknik dari *forward engineering* dan *reverse engineering* [3].

Berikut adalah penjelasan dari tahap-tahap *Enhanced Re-engineering:*

1. Studi kelayakan dan Kebutuhan

   Pada tahap ini, studi kelayakan dilakukan untuk memeriksa konfigurasi dan kompatibilitas sistem komputer. Setelah menyelesaikan studi kelayakan, kebutuhan sistem ditentukan ulang berdasarkan keinginan pengguna. *Software Requirements Spesification* (SRS) adalah dokumen resmi yang memiliki semua persyaratan dalam tulisan terstruktur. Pada tahap ini untuk menentukan kembali persyaratan sistem, maka sistem perlu dipetakan menggunakan SRS [3].

2. Restrukturisasi Spesifikasi Kebutuhan Sistem

   Tahap ini menggambarkan secara detail proses SRS yang dilakukan penyusunan ulang struktur. Dokumentasi adalah atribut penting dalam proses pengembangan perangkat lunak karena hal tersebut mereproduksi komponen dari proses rekayasa ulang total dan berfungsi sebagai perencana untuk produk akhir. Pada tahap ini sebagai tahap perbandingan antara persyaratan sistem yang sudah ada dengan mekanisme yang baru. SRS digunakan untuk mengintegrasikan kedua SRS yang baru dengan SRS yang sudah ada [3].

3. *Design to Code*

   Tahap ini menawarkan detail tentang desain ke proses kode. Pada tahap ini, sesuai dengan kode. Pada tahap ini disesuaikan dengan kode dokumen yang telah didesain ulang yang dilakukan oleh *programer*. Umumnya, tahap ini adalah penulisan ulang algoritma lama dan fungsi yang sudah diimplementasikan dalam bahasa pengembangan tradisional. Misalnya, jika suatu teknologi sistem membutuhkan waktu yang lama dan ketepatan yang diperlukan sudah tidak dapat digunakan maka membutuhkan algoritma

baru. Maka dari itu sistem harus dilakukan *re-engineering* dengan teknik yang baru [3].

4. Evaluasi Performa

   Tahap ini menghasilkan detail mengenai proses pengujian ulang. Untuk pengujian ulang, perangkat lunak yang sudah ada diambil kemudian dibandingkan kinerja fungsionalitasnya dengan fungsionalitas perangkat lunak yang baru [3] .

5. Implementasi

   Tahap ini adalah tahap terakhir dari mekanisme tahap *re-engineering*. Pada tahap implementasi, bagian – bagian tertentu diganti berdasarkan empat tahap sebelumnya [3].

2.8 *Clean Architecture*

Robert C. Martin mengusulkan *Clean Architecture* sebagai alternatif untuk menggabungkan ide-ide dari model arsitektur berlapis lainnya, seperti *Hexagonal Architecture* dan *Onion Architecture* [25]. Tujuan arsitektur adalah pemisahan masalah dengan memisahkan perangkat lunak menjadi lapisan, memisahkan aturan bisnis dan antarmuka [10].

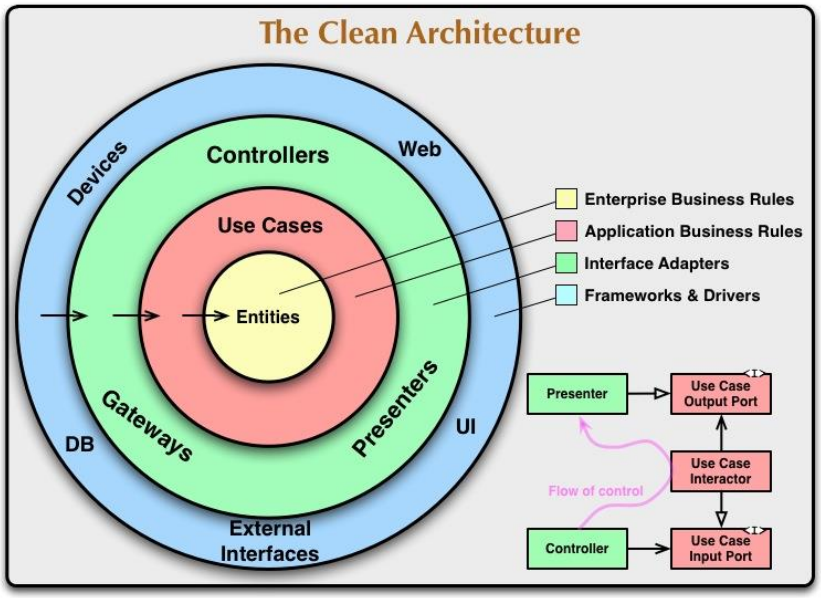
Masing-masing arsitektur tersebut menghasilkan sistem yang memiliki karakteristik sebagai berikut:

1. Independen dari *framework*. Arsitektur tidak bergantung pada keberadaan beberapa perpustakaan perangkat lunak yang sarat fitur. Ini memungkinkan *programer* untuk menggunakan *framework* tersebut sebagai alat, daripada memaksa *programer* untuk memaksakan perangkat lunak ke dalam batasannya yang terbatas [10].
2. Dapat diuji. Aturan bisnis dapat diuji tanpa antarmuka, *database*, *server web*, atau elemen eksternal lainnya [10].
3. Independen dari antarmuka. Antarmuka dapat berubah dengan mudah, tanpa mengubah bagian dari perangkat lunak lainnya. Antarmuka *web*

dapat diganti dengan antarmuka berbasis teks misalnya, tanpa mengubah aturan bisnis [10].

4. Independen dari *database*. *Programer* dapat menukar Oracle atau SQLServer ke Mongo, BigTable, CouchDB, atau yang lainnya. Aturan bisnis tidak terikat ke *database* [10].
5. Independen dari agensi eksternal mana pun. Faktanya, aturan bisnis tidak tahu apa-apa tentang antarmuka ke dunia luar [10].

Pada **Gambar 2-5** Bagan *Clean Architecture* dapat dilihat bagan dari *Clean Architecture* yang diusulkan oleh Robert C. Martin .



Gambar 2-5 Bagan *Clean Architecture*

2.8.1 *The Dependency Rule*

Lingkaran konsentris mewakili area perangkat lunak yang berbeda. Secara umum, semakin jauh *programer* melangkah, semakin tinggi level perangkat lunaknya. Lingkaran luar adalah mekanisme. Lingkaran dalam adalah kebijakan [10].

Aturan utama yang membuat arsitektur ini berfungsi adalah *Dependency Rule*. Aturan ini mengatakan bahwa dependensi kode sumber hanya dapat mengarah ke dalam. Tidak ada di lingkaran dalam yang bisa tahu apa-apa tentang sesuatu di lingkaran luar. Secara khusus, sesuatu yang dinyatakan di lingkaran luar

tidak boleh disebutkan dengan kode di lingkaran dalam. Itu termasuk, fungsi, kelas. variabel, atau entitas perangkat lunak lain yang bernama [10].

Dengan cara yang sama, format data yang digunakan di lingkaran luar tidak boleh digunakan oleh lingkaran dalam, terutama jika format tersebut dihasilkan oleh kerangka kerja di lingkaran luar [10].

2.8.2 *Entities or Enterprise Business Rule*

Entities (entitas) merangkum aturan bisnis di seluruh perusahaan. Entitas dapat berupa obyek dengan metode, atau dapat berupa kumpulan struktur dan fungsi data. Tidak masalah selama entitas dapat digunakan oleh banyak aplikasi berbeda di perusahaan [10].

2.8.3 *Use Cases or Application Business Rules*

Perangkat lunak di lapisan ini berisi aturan bisnis khusus aplikasi. Ini merangkum dan mengimplementasikan semua kasus penggunaan perangkat lunak. Kasus penggunaan ini mengatur aliran data ke dan dari entitas, dan mengarahkan entitas tersebut untuk menggunakan aturan bisnis di seluruh perusahaan mereka untuk mencapai tujuan kasus penggunaan [10].

2.8.4 *Interface Adapters*

Perangkat lunak di lapisan ini adalah seperangkat adaptor yang mengubah data dari format yang paling sesuai untuk kasus penggunaan dan entitas, ke format yang paling sesuai untuk beberapa agensi eksternal seperti basis data atau Web. Lapisan inilah, misalnya, yang sepenuhnya berisi arsitektur MVC dari GUI. Penyaji, Tampilan, dan Pengontrol semuanya ada di sini. Model kemungkinan hanya struktur data yang diteruskan dari pengontrol ke kasus penggunaan, dan kemudian kembali dari kasus penggunaan ke penyaji dan tampilan [10].

Demikian pula, data diubah, di lapisan ini, dari bentuk yang paling nyaman untuk entitas dan kasus penggunaan, ke dalam bentuk yang paling nyaman untuk kerangka *persistensi* apa pun yang digunakan. yaitu basis data. Tidak ada kode di dalam lingkaran ini yang harus tahu apa-apa tentang basis data. Jika basis data

adalah basis data SQL, maka semua SQL harus dibatasi pada lapisan ini, dan khususnya pada bagian lapisan ini yang berhubungan dengan basis data [10].

Juga di lapisan ini adalah adaptor lain yang diperlukan untuk mengonversi data dari beberapa bentuk eksternal, seperti layanan eksternal, ke bentuk internal yang digunakan oleh kasus penggunaan dan entitas [10].

2.8.5 *Frameworks and Drivers*

Lapisan terluar umumnya terdiri dari kerangka kerja dan alat-alat seperti basis data, *framework web*, dll. Umumnya, tidak perlu menulis banyak kode di lapisan ini selain kode *glue* yang berkomunikasi ke lingkaran berikutnya ke dalam. Lapisan ini adalah tempat semua detail pergi. Web adalah detail. Basis data adalah detail [10].

2.9 *Design Patterns*

Dalam rekayasa perangkat lunak, *design patterns* adalah solusi umum yang dapat diulang untuk masalah yang umum terjadi dalam desain perangkat lunak. *Design patterns* bukanlah desain akhir yang dapat diubah langsung menjadi kode. Ini adalah deskripsi atau *template* untuk bagaimana memecahkan masalah yang dapat digunakan dalam banyak situasi yang berbeda [26].

Katalog GoF yang ada pada buku *Design Patterns: Elements of Reusable Object Oriented Software* mencakup 23 *design patterns* [27]. Buku tersebut ditulis oleh Erich Gamma, Richard Helm, Ralph Johnson, dan John Vlissides. Dari 23 *design patterns* yang ada terbagi menjadi 3 kategori yaitu *creational pattern*, *structural pattern*, *behavioral patterns*. Berikut adalah penjelasan dari setiap kategori tersebut:

1) *Creational Design Pattern*

   Creational Design Pattern membahas tentang bagaimana suatu obyek dibuat. Kadang kala penciptaan suatu obyek tidaklah mudah. Dikarenakan mungkin penciptaan suatu obyek melibatkan beberapa logika dan kondisi. *Creational Design Pattern* ditujukan untuk menghilangkan kompleksitas

dari kode yang dibuat oleh pengembang. Ada lima *design pattern* dalam kategori ini yaitu:

- *Factory Method*
- *Abstract Factory*
- *Builder*
- *Prototype*
- *Singleton*

2) *Structural Design Patterns*

Structural Design Patterns berkaitan dengan isi dari kelas dan obyek. Pola ini menyimplifikasi struktur suatu sistem dengan mengidentifikasi hubungan antar obyek. Berikut adalah pola pada kategori ini:

- *Adapter*
- *Bridge*
- *Composite*
- *Decorator*
- *Façade*
- *Flyweight*
- *Proxy*

3) *Behavioral Design Patterns*

Behavioral Design Patterns berkaitan dengan bagaimana antar obyek berinteraksi dan berkomunikasi. Pola ini mengupayakan untuk mengurangi kerumitan yang mungkin terjadi ketika obyek saling berkomunikasi. Berikut adalah *design pattern* pada kategori ini:

- *Interpreter*
- *Template Method*
- *Chain of Responsibility*
- *Command*
- *Iterator*
- *Mediator*

- *Memento*
- *Observer*
- *State*
- *Strategy*
- *Visitor*

2.10 *Maintainability Index*

Maintainability adalah kemudahan dari perangkat lunak untuk dipelihara [28] seperti :

1. Memperbaiki kerusakan
2. Menemukan kebutuhan baru
3. Membuat pemeliharaan selanjutnya lebih mudah
4. Mengatasi lingkungan yang berubah.

Tingkatan seberapa mudah aplikasi dapat dilakukan pemeliharaan dapat diukur menggunakan *Maintainability Index* (MI). Rentang nilai dari MI berada di antara 0 sampai 100. Jika nilai semakin mendekati 100 maka semakin mudah juga suatu aplikasi dapat dipelihara [29]. Coleman menggunakan pendekatan 50 *regression model* untuk mengidentifikasi model yang sederhana dan akurat yang bisa diterapkan pada aplikasi pada umumnya. Berikut adalah formula dari *Maintainability Index* (MI):

$$MI = 171 - 5.2\ x\ log_2(V) - 0.23 * V(g) - 16.2\ x\ \ log_2(LOC) + (50\ x \sin(\sqrt{2.46\ x\ perCM}))$$

Di mana:

$V = Halstead\ Metrics\ Volumes$

$V(g) = Cyclomatic\ Complexity$

$LOC = Lines\ of\ Code$

$perCM = \ Percent\ Line\ of\ Comment$

Hasil dari formula MI di atas memiliki rentang negatif tak hingga sampai 171 $(-\infty, 171)$, jika suatu nilai MI mendekati 0 maka dapat diartikan aplikasi sulit untuk dipelihara dan jika nilai negatif maka aplikasi tidak dapat dibedakan dengan negatif lainnya. Klasifikasi dari MI dapat dilihat pada **Tabel 2-2** Klasifikasi *Maintainability Index***.** Supaya nilai tetap pada rentang 0 sampai 100 maka rumus yang direkomendasikan [29] adalah sebagai berikut :

$$MI = \max\{0, (171 - 5.2\ x\ log_2(V) - 0.23 * V(g) - 16.2\ x\ log_2(LOC))x\ 100/\ 171\}$$

Tabel 2-2 Klasifikasi *Maintainability Index*

Nilai MI	Klasifikasi
0 - 9	*Tough to Maintain*
10 - 19	*Moderate Level of Difficulty to Maintain*
20 – 100	*Easy to Maintain*

2.11 *Halstead Metrics*

Halstead metrics adalah pengukuran yang dikembangkan untuk mengukur kompleksitas modul suatu program langsung dari kode sumber. Pengukuran dilakukan dengan menentukan ukuran kuantitatif kompleksitas dari *operator* dan *operand* dalam modul sistem [30]. Pada *Halstead's metrics* terdapat beberapa enam jenis komponen yaitu:

1. *Length of program*

   Length of program adalah kalkulasi jumlah total operator dan operan yang muncul. Persamaan *Length of program* dirumuskan pada persamaan sebagai berikut:

   $$N = N1 + N2$$

   Keterangan:

   N1 = total semua *operator* yang muncul

   N2 = total semua *operand* yang muncul

2. *Vocabulary of the program*

Vocabulary of the program adalah kalkulasi jumlah *operator* dan *operand* berbeda yang muncul dalam program. Persamaan *Vocabulary of the program* dirumuskan pada persamaan sebagai berikut:

$$n = n1 + n2$$

Keterangan:

n = *Vocabulary of the program*

n1 = jumlah *operator* berbeda

n2 = jumlah *operand* berbeda

3. *Volume of the program*

   Volume dalam *Halstead metric* di gunakan untuk mengetahui volume program. Persamaan *Volume of the program* dirumuskan pada persamaan sebagai berikut:

$$V = N \times \log_2 n$$

   Keterangan:

   V = *Volume of the program*

   N = nilai kalkulasi *length of the program*

   n = nilai kalkulasi *vocabulary of the program*

4. *Difficulty*

   Difficulty dalam *Halstead's metrics* di gunakan untuk mengetahui kesulitan dan pengembangan program. Persamaan *Difficulty* dirumuskan pada persamaan sebagai berikut:

$$D = \frac{n1}{2} \; x \; \frac{N2}{n2}$$

   Keterangan:

   D = *Difficulty*

   N2 = total semua *operand* yang muncul

   n1 = jumlah operator berbeda

   n2 = jumlah operan berbeda

5. *Effort*

   Effort dalam *Halstead's metrics* di gunakan untuk mengetahui sumber daya yang digunakan untuk pengembangan program. Persamaan *Effort* dirumuskan pada persamaan sebagai berikut :

   $$E = D \; x \; V$$

   Keterangan:

   E = *Effort*

   D = nilai dari *Difficulty*

   V = nilai *Volume of the program*

6. *Number of bugs expected in the program*

   Number of bugs expected in the program dalam *Halstead's metrics* digunakan untuk mengetahui prediksi bug pada program. Persamaan *Number of bugs expected in the program* dirumuskan pada persamaan sebagai berikut:

   $$B = \frac{V}{3000}$$

   Keterangan:

   B = *Number of bugs expected in the program*

   V = nilai *Volume of the program*

2.12 *Refactoring*

Terdapat dua definisi mengenai *Refactoring* menurut Martin Flower yang mana tergantungkonteks. Definisi pertama yaitu *refactoring* adalah struktur internal aplikasi yang diubah agar aplikasi tersebut lebih mudah dipahami dan lebih mudah untuk dilakukan modifikasi tanpa mengubah tingkah laku dari aplikasi. Definisi kedua adalah *refactoring* adalah membuat struktur ulang suatu aplikasi dengan melakukan serangkaian *refactoring* tanpa mengubah tingkah laku dari aplikasi [31].

Beberapa alasan kode program pada suatu aplikasi harus dilakukan *refactoring* menurut Martin Flower adalah sebagai berikut:

1. Meningkatkan suatu desain aplikasi

   Jika *refactoring* desain pada suatu aplikasi tidak dilaksanakan, aplikasi akan mengalami kerusakan. Hal tersebut sangat mungkin untuk terjadi karena *programer* melakukan perubahan pada *code* bukan untuk tujuan jangka panjang atau perubahan pada *code* dilakukan tanpa memahami terlebih dahulu *design code* pada suatu aplikasi. Hal tersebut bisa menjadi efek berkepanjangan sehingga mengakibatkan *design code* menjadi sangat susah dibaca.

2. Membuat Aplikasi menjadi lebih mudah dipahami

   Dalam menulis kode program, kode tersebut tidak boleh komputer saja yang paham apa perintah yang diberikan oleh *programer*. Namun *programer* lain juga harus dapat membaca kode program tersebut dan bagaimana cara memodifikasikannya. *Readability code* dapat mempengaruhi waktu dalam *programer* untuk melakukan suatu perubahan. *Readability code* yang rendah dapat berakibat pada lamanya proses pengubahan. *Refactoring* adalah salah satu cara untuk membuat *code* menjadi lebih mudah untuk dibaca.

3. Memudahkan untuk mencari *Bugs*

   Salah satu pernyataan dari Kent Beck yaitu "*Refactoring helps me be much more effective at writing robust code*". Maksud dari kutipan ini adalah dengan melakukan *refactoring*, *programer* akan memahami maksud dari *code* yang lama kemudian akan menambahkan pemahaman baru pada *code* tersebut. Dengan memperbaiki struktur aplikasi, maka *programer* yang melakukan *refactor* akan menemukan asumsi-asumsi pada *code* sebelumnya yang dapat mengakibatkan *bug*.

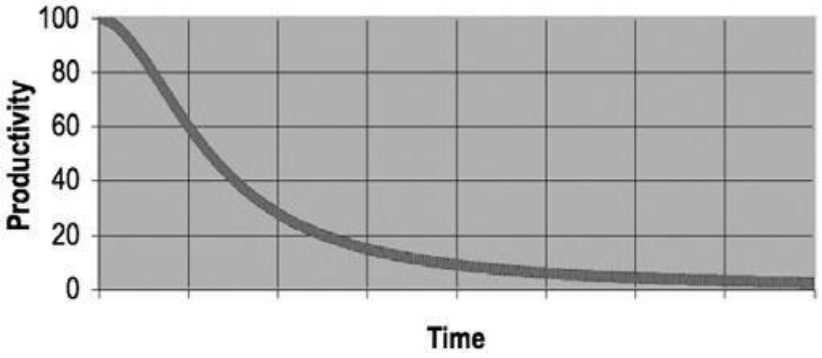
4. Meningkatkan Perfoma Aplikasi

Refactoring biasanya berbicara tentang meningkatkan desain, meningkatkan readability, mengurangi *bugs*. Hal tersebut berbicara mengenai kualitas namun pada logika dasarnya bahwa hal tersebut akan mengakibatkan lamanya proses pembangunan suatu aplikasi. Aplikasi tanpadesain yang bagus biasanya akan cepat pada proses pembangunan namun karena desain yang kurang baik maka dapat dipastikan aplikasi tersebut pada waktu mendatang akan mengalami perlambatan dari proses pembuatanatau performa aplikasi. Proses *Refactoring* akan berfokus pada meningkatkan performa dan mencari *bugs* daripada menambah suatu fitur baru.

2.13 *Clean Code*

Sesuai namanya, *clean code* merujuk pada penulisan struktur kode yang bersih, dan harus dapat dibaca dan diubah dengan mudah oleh pengembang lain. Tujuan dari *clean code* adalah untuk mengatasi penurunan tingkat produktivitas pengembangan perangkat lunak akibat dari struktur kode yangberantakan seperti yang dapat dilihat pada **Gambar 2-6** *Productivity vs Time* di mana waktu dapat mempengaruhi tingkat produktivitas [32]. Berikut ini adalah beberapa hal inti yang dibahas pada *Clean Code:*

1. *Meaningful Names*
2. *Clean Functions*
3. *Clean Comments*
4. *Clean Code Formatting*
5. *Clean Error Handling*
6. *Clean Object and Data Structure*
7. *Clean Classes*



Gambar 2-6 *Productivity vs Time*

2.13.1 *Meaningful Names*

Konsep *meaningful names* merupakan petunjuk mengenai pemberian nama terhadap variabel, *method*/fungsi, dan kelas agar lebih mudah untuk dipahami. Berikut ini adalah petunjuk yang dapat digunakan dalam melakukan pemberian nama:

1) Nama yang Dapat Dieja dan Memiliki Arti

   Penggunaan nama yang dapat dieja dan secara jelas menerangkan kegunaan dari kode yang ditulis dapat memudahkan *programer* lain dalam memahaminya. Sehingga *programer* tidak perlu mengalami kesulitan hanya untuk memahami kegunaan kode tersebut. Berikut ini adalah contoh dari nama yang dapat dieja dan memiliki arti:

   Baik

```
1. int date;
2. String month;
3. public ArrayList<Food>
```

   Buruk

```
1. int j;
2. String sm; //selected month
3. public ArrayList<plc>
```

2) Nama yang Mudah untuk Dicari

   Petunjuk ini digunakan untuk memudahkan *programer* dalam pencarian kode. Keuntungan dari penggunaan petunjuk ini, *programer* tidak perlu

memahami program secara keseluruhan. Hal ini biasa diterapkan untuk menamai suatu konstanta. Berikut ini adalah contoh dari kode mengenai penamaan yang mudah untuk dicari:

Baik

```
const int BASE_SALARY_PER_HOUR = 45;
int totalSalary = 0;
for (int i = 0; i = employee.getSize(); i++) {
    int workInHour = employee.get(i).getWorkInHour();
    int salary = BASE_SALARY_PER_HOUR * workInHour;
    employee.get(i).setSalary(salary);
    totalSalary += salary;
}
```

Buruk

```
int k = 0;
while (k < 40) {
     m += (p[k]*2)/10;
}
```

3) Menghindari *Mental Mapping*

Penggunaan singkatan seperti i dan n untuk penamaan variabel iterasi dan jumlah, atau penggunaan singkatan lain yang tidak diketahui oleh orang pada umumnya dapat mempersulit proses pemahaman dari kode yang ditulis. Sehingga pengembang disarankan untuk menghindari penggunaan singkatan tersebut. Berikut ini adalah contoh kode untuk menghindari *mental mapping*:

Baik

```
Luigi player = new Luigi();
String role = player.getRole();
```

Buruk

```
Data player = new Data();
String role = player.getRole();
```

4) Kata benda untuk kelas

   Dalam melakukan pemberian nama terhadap kelas, disarankan untuk menggunakan kata benda dan menghindari penggunaan kata kerja. Hal tersebut didasari oleh penggunaan kelas yang biasanya digunakan mewakili suatu entitas. Contoh nama kelas yang tidak disarankan adalah Manager, Processor, Data dan Info.

5) Kata kerja untuk *method*/fungsi

   Tidak seperti penamaan *kelas*, penamaan pada *method*/fungsi disarankan untukmenggunakan kata kerja. Hal tersebut dilakukan untuk mendeskripsikan tujuan dari pekerjaan yang dilakukan oleh *method*/fungsi yang ditulis secara eksplisit.*Method*/fungsi *accessors*, *mutators*, dan *predicates* biasanya memiliki prefiks *set*, *get*, dan *is*. Berikut ini merupakan contoh dari kode dalam kata kerja untuk *method*/fungsi:

Baik

```
1. class Luigi() {
2.     String role;
3.     public void setRole(String role) {}
4. }
```

Buruk

```
1. class Luigi() {
2.     String role;
3.     public void role(String role) {}
4. }
```

6) Menghindari penggunaan konteks yang tidak diperlukan

Hal ini biasa terjadi pada suatu kelas atau pada obyek, di mana atributnya mengandung nama dari kelas atau obyek tempat atribut tersebut berada. Hal tersebut tidak disarankan, karena bersifat repetitif. Berikut ini merupakan contoh kode untuk menghindari penggunaan konteks yang tidak diperlukan:

Baik

```
1.void setRole(HashMap player) {
2.    player["role"] = "Enemy";
3.}
```

Buruk

```
1. void setRole(HashMap player) {
2.     player["PlayerRole"] = "Enemy";
3. }
```

2.13.2 *Clean Function*

Konsep ini merupakan pembahasan dalam menulis *method*/fungsi yang bersih agar lebih mudah untuk dipahami. Berikut merupakan petunjuk yang dapat digunakan dalam menulis *method*/fungsi:

1) Jumlah parameter

   Jumlah parameter yang baik dalam sebuah *method*/fungsi adalah maksimal sebanyak 2 parameter. Jumlah parameter yang terbatas tersebut dimaksudkan untuk menghindari banyaknya variasi pengujian terhadap *method*/fungsi dalam kode. Jika dibutuhkan parameter yang digunakan lebih dari jumlah yang baik, obyek dapat digunakan sebagai parameter dari *method*/fungsi tersebut. Berikut ini merupakan contoh dari jumlah parameter yang baik dalam sebuah *method*/fungsi:

   Baik

```
1. void placeButton(Coordinate coordinate, int fontSize) {
2.     // ...
3. }
```

   Buruk

```
void placeButton(int x, int y, int z, int fontSize) {
    // ...
}
```

2) Menghindari penggunaan *flag* sebagai parameter

Penggunaan *flag* sebagai parameter memberikan arti bahwa da;am *method*/fungsi yang dibangun memiliki pekerjaan lebih dari satu. Berikut merupakan contoh penggunaan *flag* sebagai parameter:

Baik

```
class Log {
    void printWarning(String log) {
        System.printWarning(log);
    }

    void printDebug(String log) {
        System.printDebug(log);
    }
}
```

Buruk

```
class Log {
    void print(bool isWarning, String log) {
        if (isWarning)
            System.printWarning(log);
        else
            System.printDebug(log);
    }
}
```

3) Jumlah pekerjaan

Jumlah pekerjaan yang dilakukan oleh suatu *method*/fungsi disarankan hanya satu pekerjaan saja. Jika jumlah pekerjaan dalam suatu *method*/fungsi lebih dari satu pekerjaan maka akan sulit untuk dibuat, diuji, dan dipahami. Apabila *method*/fungsi yang ditemukan melakukan lebih dari satu pekerjaan, pisahkan pekerjaan-pekerjaan tersebut ke dalam *method*/fungsi

yang berbeda. Berikut ini merupakan contoh kode mengenai jumlah pekerjaan yang baik:

Baik

```
1. class Presence {
2.     String name;
3.     long timestamp;
4.
5.     void setName(String name) {
6.         this.name = name;
7.     }
8.
9.     void setPresence() {
10.        this.timestamp = Calendar.getInstance().getTimeInMilis();
11.    }
12. }
```

Buruk

```
1. class Presence {
2.     String name;
3.     long timestamp;
4.
5.     void setName(String name) {
6.         this.name = name;
7.         this.timestamp = Calendar.getInstance().getTimeInMilis();
8. }
```

2.13.3 *Clean Comment*

Konsep ini membahas mengenai pemberian komentar yang baik dan efisien dalam kode. Komentar pada kode diperlukan untuk memberikan informasi yang tidak tersampaikan oleh kode yang telah ditulis. Namun, pemanfaatan *clean comment* berhubungan dengan pemanfaatan *meaningful names*, di mana semakin jelas penamaan dari suatu variabel, *method*/fungsi, maupun kelas semakin sedikit pula komentar yang harus diberikan. Berikut merupakan petunjuk dalam melakukan penulisan komentar:

1) Komentar yang informatif

   Penulisan komentar sebaiknya memberikan penjelasan yang informatif mengenai kode. Informatif yang dimaksud adalah pemberian informasi

kepada orang non teknis supaya memahami kode tersebut. Berikut ini merupakan contoh komentar yang informatif dalam kode program:

Baik

```
1. // format matched kk:mm:ss EEE, MMM dd, yyyy
2. Pattern timeMatcher =
3.     Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Buruk

```
1.// this is for formatting time
2.Pattern timeMatcher =
3.    Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

2) Menggunakan Klarifikasi

Jika *programer* menggunakan pustaka yang tidak dapat diubah, *programer* disarankan untuk memberikan komentar untuk memberikan klarifikasi supaya *programer* lainnya dapat memahami maksud dari suatu kode tersebut. Berikut ini merupakan contoh komentar untuk memberikan klarifikasi pada kode:

```
1. public void testCompareTo() throws Exception {
2.     assertTrue(a.compareTo(a) == 0); // a == a
3.     assertTrue(a.compareTo(b) != 0); // a != b
4.     assertTrue(ab.compareTo(ab) == 0); // ab == ab
5.     assertTrue(a.compareTo(b) == -1); // a < b
6. }
```

3) Menggunakan Komentar Peringatan

Terdapat beberapa kode yang jika dieksekusi dapat berakibat buruk pada suatu aplikasi namun dibutuhkan ketika kondisi tertentu. Komentar disarankan untuk memberikan peringatan jika kode tersebut memiliki dapat berakibat buruk. Berikut ini merupakan contoh kode yang diberikan komentar peringatan:

```
1. // Don't run unless you
2. // have some time to spend.
3. public void _testUploadBigFile() {
4.     writeLinesToFile(10000000);
5.     apiService.setBody(testFile);
```

```
6.      Response apiResponse = apiService.send(this);
7.      String responseString = apiResponse.toString();
8.  }
```

4) Menggunakan Komentar Catatan *Todo*

Terdapat kemungkinan *programer* tidak sempat menyelesaikan aplikasi secara keseluruhan. Untuk kode-kode yang belum terselesaikan disarankan untuk *mengunakan* catatan *todo*. Berikut ini merupakan contoh kode yang diberikan komentar catatan *todo*:

```
1.  // TODO-MdM these are not needed
2.  // We expect this to go away when we do the checkout model
3.  protected VersionInfo makeVersion() throws Exception
4.  {
5.  return null;
6.  }
```

5) Hindari Komentar pada kode yang sudah jelas

Sering ditemukan *programer* memberikan komentar pada kode yang sudah jelas penggunaannya. Hal ini akan memberikan informasi yang berulang. Berikut ini adalah contoh penggunaan komentar pada kode yang sudah jelas:

```
1.  /*
2.  memberikan nilai bilangan bulat 32 pada ukuran ikan
3.  */
4.  int fishSize = 32;
```

2.13.4 *Clean Error Handling*

Petunjuk pada konsep penganganan kesalahan (*Error Handling*) ini bertujuan agar pesan kesalahan dapat digunakan secara efisien dan dapat ditampilkan dengan mudah dengan menerapkan penanganan kesalahan yang bersih yang mana tidak mengabaikan kesalahan yang tertangkap.

Penanganan kesalahan pada umumnya hanya ditanggulangi dengan cara menampilkan ke *logger* tanpa tindakan lebih lanjut sehingga pengguna tidak

mengetahui kesalahan apa yang terjadi pada saat itu. Tindakan lebih lanjut diperlukan agar pesan kesalahan tidak hanya ditampilkan ke *logger*, tetapi pengguna atau pengembang juga mendapatkan pesan berupa notifikasi terkait kesalahan yang terjadi. Berikut ini adalah contoh kode dari penanganan kesalahan yang baik:

Baik

```
try {
    functionThatMightThrow();
} catch (error) {
    // Satu opsi (lebih sesuai untuk menampilkan kesalahan daripada console.log):
    console.error(error);
    // opsi lainnya:
    notifyUserOfError(error);
    // opsi lainnya:
    reportErrorToService(error);
}
```

Buruk

```
try {
    functionThatMightThrow();
} catch (error) {
    console.log(error);
}
```

2.13.5 *Clean Object and Data Structure*

Konsep ini membahas petunjuk dalam membuat struktur data yang bersih. Struktur data yang bersih memiliki abstraksi sehingga tidak dapat diakses langsung secara publik. Berikut merupakan petunjuk dalam membuat struktur data yang bersih:

1) Penggunaan Data *Abstraction*

   Abstraction disarankan ketika pembuatan struktur data. *Abstraction* membantu *programer* agar aplikasi dapat mudah dipelihara dengan leluasa untuk memodifikasi tanpa perlu takut merusak struktur aplikasi.

Berikut ini merupakan contoh kode struktur data yang menggunakan *abstraction*:

Baik

```
1.public interface Point {
2.    double getX();
3.    double getY();
4.    void setCartesian(double x, double y);
5.    double getR();
6.    double getTheta();
7.    void setPolar(double r, double theta);
8.}
```

Buruk

```
1. public class Point {
2.     public double x;
3.     public double y;
4. }
```

2) Membuat obyek memiliki *private member*

Penggunaan *private member* disarankan untuk membuat struktur data yang baik. Hal ini membantu *programer* untuk *hiding implementation* yang mana *programer* tidak perlu mengetahui bagaimana proses di dalamnya. Berikut ini merupakan contoh kode program yang mana obyek memiliki *private member*:

```
1. public RectangleView(){
2.     private int getLuas(){
3.         return RectangleViewModel.getLuas();
4.     }
5.
6.     private int getKeliling(){
7.         return RectangelViewModel.getKeliling();
8.     }
9.
10.    public void showRectangleResult(){
11.        System.out.println("Luas : "+ getLuas());
12.        System.out.println("Keliling : "+ getKeliling());
13.    }
```

```
14. }
```

3) *Getter* dan *Setter*

Struktur data yang bersifat *private* perlu diakses menggunakan *getter* dan *setter*. Tujuan dari penggunaan *getter* dan *setter* adalah untuk menghindari berubahnya nilai pada struktur data yang dilakukan sengaja maupun tidak. Berikut ini merupakan contoh kode program yang menggunakan *getter* dan *setter*:

```
1.  public Employee(){
2.      String name;
3.
4.      public void setName(String name){
5.          this.name = name;
6.      }
7.
8.      public String getName(){
9.          return this.name;
10.     }
11. }
```

2.13.6 *Clean Class*

Konsep *clean class* ini terdapat petunjuk dalam membuat kelas yang lebih bersih dan terorganisir. Berikut merupakan petunjuk yang dapat dalam membuat kelas:

1) *Class Organization*

Dalam pembuatan kelas, *code convention*/*code styling* dari bahasa pemrograman yang digunakan dapat diikuti untuk mendapatkan *class organization* yang baik.

2) *Single Responsibility Principle*

Ukuran suatu kelas disarankan untuk tidak terlalu besar. Jika terlalu besar, *programer* akan kesulitan dalam memahami kelas tersebut belum lagi karena banyaknya jumlah baris yang ditulis. Ukuran kelas yang kecil, dapat mempermudah proses modifikasi. Ukuran suatu kelas dapat

diukur berdasarkan jumlah *responsibility* atau tanggung jawab yang dikerjakan oleh kelas tersebut. *Responsibility* dalam sebuah kelas yang bersih hanya memiliki satu *responsibility* saja. Hampir sama dengan *clean function*, apabila kelas tersebut memiliki lebih dari satu *responsibility*, pisahkan *responsibility* ke dalam kelas yang berbeda. Berikut ini merupakan contoh dari kode yang menerapkan konsep *Single Responsibility Principle*:

Baik

```
1.public class Version(){
2.    public int getMajorVersionNumber();
3.    public int getMinorVersionNumber();
4.    public int getBuildNumber();
5.}
```

Buruk

```
1. class SuperDashboard extends JFrame implements MetaDataUser {
2.     public Comment getLastFocusedComponent();
3.     public void setLastFocused(Component lastFocused);
4.     public int getMajorVersionNumber();
5.     public int getMinorVersionNumber();
6.     public int getBuildNumber();
7. }
```

2.14 Analisis dan Desain Berorientasi Obyek

Dalam memikirkan suatu masalah dengan menggunakan model, pendekatan Analisis dan Desain Berorientasi Obyek (*Object Oriented Analysis and Design*) dapat digunakan. Dasar pembuatan dari analisis ini adalah kombinasi antar obyek, struktur data dan perilaku dalam satu entitas. Penggunaan metode berorientasi obyek dilandasi alasan perangkat lunak yang bersifat dinamis, di mana hal ini disebabkan karena kebutuhan pengguna berubah dengan cepat [33].

Pendekatan ini juga bertujuan untuk menurunkan kompleksitas transisi antar tahap pada pengembangan perangkat lunak, karena notasi yang digunakan pada tahap analisis perancangan dan implementasi relatif sama. Berbeda dengan pendekatan konvensional yang dikarenakan notasi yang digunakan pada tahap

analisanya berbeda-beda. Hal itu menyebabkan transisi antar tahap pengembangan menjadi kompleks.

Selain itu, pendekatan berorientasi obyek membawa pengguna kepada abstraksi atau lebih dekat dengan dunia nyata. Pada dunia nyata yang umunya pengguna lihat adalah obyeknya bukan fungsinya. Beda dengan pendekatan terstruktur yang hanya mendukung abstraksi pada level fungsional. Adapun dalam pemrograman berorientasi obyek menekankan berbagai konsep seperti [33]:

1. *Class*
2. *Object*
3. *Abstract*
4. *Encapsulation*
5. *Polymorphism*
6. *Inheritance*
7. UML (*Unified Modeling Language*).

Penggunaan UML dalam Analisis dan Desain Berorientasi Obyek sebagai alat bantu yang dapat digunakan dalam bahasa pemrograman berorientasi obyek. UML juga merupakan *standard modeling language* yang terdiri dari kumpulan-kumpulan diagram, dikembangkan untuk membantu para pengembang sistem dan perangkat lunak agar bisa menyelesaikan tugas-tugas seperti:

1. Spesifikasi
2. Visualisasi
3. Desain Arsitektur
4. Konstruksi
5. Simulasi dan Testing

Kesimpulan yang ditarik dari UML adalah sebuah bahasa yang berdasarkan grafik atau gambar untuk memvisualisasikan, melakukan spesifikasi, membangun dan pendokumentasian dari sebuah sistem pengembangan perangkat lunak berbasis obyek (*Object Oriented Programming*).

Terdapat 10 macam diagram dalam Dokumentasi UML [9] untuk membuat model aplikasi berorientasi obyek, empat di antaranya sebagai berikut:

1. *Use Case Diagram*

   Diagram ini digunakan untuk menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Penekanan dari *use case diagram* ini terdapat pada apa yang sistem perbuat dan bagaimana sebuah sistem itu bekerja. Sebuah *use case* merepresentasikan sebuah interaksi antara *actor* dengan sistem. *Use case* merupakan bentuk dari sebuah pekerjaan tertentu, misalnya *login* ke dalam sistem, *cetak document* dan sebagainya, sedangkan seorang *actor* adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

2. *Use Case Scenario*

   Sebuah diagram yang menunjukkan *use case* dan aktor mungkin menjadi titikawal yang bagus, tetapi tidak memberikan detail yang cukup untuk desainer sistem untuk benar-benar memahami persis bagaimana sistem dapat terpenuhi. Cara terbaik untuk mengungkapkan informasi penting ini adalah dalam bentukpenggunaan *use case scenario* berbasis teks per *use case*-nya.

3. *Activity Diagram*

   Activity Diagram adalah sebuah tahapan yang lebih fokus kepada menggambarkan proses bisnis dan urutan aktivitas dalam sebuah proses. Di mana biasanya dipakai pada bisnis modeling untuk memperlihatkan urutan aktivitas proses bisnis. *Activity diagram* ini sendiri memiliki struktur yang mirip dengan *flowchart* atau data *flow* diagram pada perancangan terstruktur. *Activity diagram* dibuat berdasarkan sebuah atau beberapa *use case* pada *use case diagram*.

4. *Sequence Diagram*

   Sequence diagram digunakan untuk menggambarkan perilaku pada sebuah *scenario*. Diagram jenis ini memberikan kejelasan sejumlah obyek dan pesan- pesan yang diletakkan di antaranya di dalam sebuah *use case*. Komponen utamanya adalah obyek yang digambarkan dengan kotak segi empat atau bulat,*message* yang digambarkan dengan gari putus dan waktu yang ditunjukkan dengan *progress vertical*. Manfaat dari *sequence diagram* adalah memberikangambaran detail dari setiap interaksi yang terjadi pada *use case diagram* yang dibuat sebelumnya.

5. *Class Diagram*

   Class diagram adalah sebuah *class* yang menggambarkan struktur dan penjelasan *class*, paket dan obyek serta hubungan satu sama lain. *Class diagram*juga menjelaskan hubungan antar *class* secara keseluruhan di dalam sebuah sistem yang sedang dibuat dan bagaimana caranya agar mereka saling berkolaborasi untuk mencapai sebuah tujuan.

2.15 *Design Principles*

Design Principles pada perangkat lunak pada umumnya identik dengan *SOLID principles*. *SOLID principles* adalah prinsip yang membahas bagaimana cara untuk mengatur fungsi-fungsi dan struktur data menjadi suatu *class*. Dan juga mengatur bagaimana cara *class-class* saling berkaitan. Penggunaan kata "*class*" pada SOLID tidak hanya mengacu pada suatu perangkat lunak yang pembangunannya berorientasi objek. Tujuan dari SOLID adalah untuk *Tolerate Change, Are easy to understand,* dan membuat suatu basis komponen yang dapat digunakan pada banyak sistem perangkat lunak. Berikut ini adalah penjelasan dari akronim SOLID [10]:

1) SRP: *Single Responsibility Principle*

Single Responsibility Principle (SRP) adalah prinsip yang menyatakan bahwa setiap modul atau kelas harus memiliki satu tanggung jawab atas bagian dari fungsi yang disediakan oleh perangkat lunak, dan bahwa tanggung jawab harus sepenuhnya dienkapsulasi oleh kelas, modul atau fungsi. Semua layanannya harus selaras dengan tanggung jawab itu. Robert C. Martin menyatakan prinsip ini sebagai, " *A class should have only one reason to change*" yang mana setiap kelas hanya harus punya satu alasan untuk berubah.

2) OCP: *Open-Closed Principle*

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification", Bertrand Meyer membuat prinsip ini terkenal pada 1980-an. Intinya adalah untuk itu sistem perangkat lunak agar mudah diubah, mereka harus dirancang untuk memungkinkan perilaku dari sistem yang akan diubah dengan menambahkan kode baru, daripada mengubah kode yang sudah ada.

3) LSP: The Liskov Substitution Principle

Definisi *Liskov Substitution Principle* yang terkenal, dari tahun 1988. Singkatnya, prinsip ini mengatakan bahwa untuk membangun sistem perangkat lunak dari bagian yang dapat dipertukarkan, bagian itu harus mematuhi kontrak yang memungkinkan bagian-bagian itu diganti satu sama lain.

4) ISP: *Interface Segregation Principle*

Prinsip ini menyarankan perancang perangkat lunak untuk menghindari tergantung pada hal-hal yang mereka jangan gunakan.

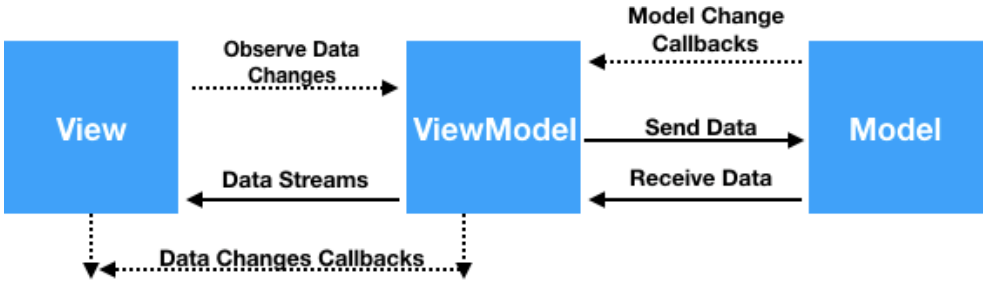
5) DIP: *Dependency Inversion Principle*

Kode yang menerapkan kebijakan *high-level* tidak boleh bergantung pada kode itu mengimplementasikan detail *low-level*. Sebaliknya, detail harus bergantung pada kebijakan.

2.16 *Model View ViewModel (MVVM)*

Arsitektur *Model-View-ViewModel* (MVVM) pertama kali dikenalkan oleh John Gossman pada blognya. Arsitektur ini awalnya digunakan untuk Microsoft Silverlight dan WPF yang mana arsitektur tersebut juga berbasiskan pada arsitektur MVC [34].

Arsitektur MVVM mempunyai tiga komponen yaitu *Model*, *View* dan *ViewModel*. Komponen View bertanggung jawab terhadap User Interface (UI) dari suatu aplikasi. Model adalah komponen yang merepresentasikan data. *ViewModel* bertugas untuk membuat kolaborasi antara *Model* dengan *View*. *ViewModel* bertanggung jawab mengalirkan data dan operasi-operasi ke *View*. Dan juga *ViewModel* bertanggung jawab untuk mengelola logika dan *behavior* pada *View* [35]. Secara umum hubungan antara *Model*, *View*, dan *ViewModel* bisa dilihat pada **Gambar 2-7** Hubungan antar Komponen pada *MVVM*.



Gambar 2-7 Hubungan antar Komponen pada *MVVM*

2.17 Android Studio

Android Studio adalah *Integrated Development Environment* (IDE) resmi untuk pengembangan aplikasi Android, yang didasarkan pada IntelliJ IDEA [36]. Selain sebagai editor kode dengan fitur developer IntelliJ IDEA yang sangat menunjang produktivitas, Android Studio menawarkan banyak fitur yang meningkatkan produktivitas dalam membuat aplikasi Android, seperti:

- Sistem *build* berbasis Gradle yang fleksibel

- Emulator yang cepat dan kaya fitur
- Lingkungan terpadu tempat Anda bisa mengembangkan aplikasi untuk semua perangkat Android
- Terapkan Perubahan untuk melakukan *push* pada perubahan kode dan *resource* ke aplikasi yang sedang berjalan tanpa memulai ulang aplikasi
- *Template* kode dan integrasi GitHub untuk membantu Anda membuat fitur aplikasi umum dan mengimpor kode sampel
- *Framework* dan alat pengujian yang lengkap
- Alat *lint* untuk merekam performa, kegunaan, kompatibilitas versi, dan masalah lainnya
- Dukungan C++ dan NDK
- Dukungan bawaan untuk Google Cloud Platform, yang memudahkan integrasi Google Cloud Messaging dan App Engine

2.18 *Code Readability*

Secara bahasa, *readability* memiliki arti keterbacaan. Keterbacaan ini menjadi hal yang penting dalam menulis kode program. Penting karena bagian substansial dari biaya perangkat lunak dalam daur hidupnya bukanlah ketika pengembangan dilaksanakan, tetapi ketika pemeliharaan (*maintenance*) [1]. Penilaian dari *code readability* dilakukan dengan melakukan wawancara kepada pengembang aplikasi. Berikut ini adalah tahapan yang dilakukan pada saat wawancara kepada pengembang aplikasi [1]:

1. Penguji menceritakan terlebih dahulu tujuan, kegunaan, cerita atau fungsionalitas dari aplikasi. Penguji tidak perlu menjelaskan kode.
2. Penguji menunjukkan berkas kode mana yang digunakan. Dimulai dengan *test cases* yang akan membantu pengembang aplikasi memahami bagaimana fungsi dari kode digunakan.
3. Pengembang aplikasi membaca dan menjelaskan kode dengan suara. Jika kode yang dibaca kurang dapat dipahami, pengembang aplikasi membuat spekulasi kegunaan dari kode tersebut.

4. Penguji tidak menjawab apa pun ketika pengembang aplikasi sedang memikirkan atau bertanya. Penguji dapat menulis catatan terkait bagian yang dibingungkan oleh pengembang aplikasi.
5. Pada tahap akhir, penguji mengkonfirmasi kepada pengembang aplikasi apakah pemahaman mengenai kodenya sesuai. Penguji juga bertanya kepada pengembang aplikasi mengenai pertanyaan yang sempat terucap dan bertanya mengenai pengalaman. Setelah itu penguji meminta penilaian kepada pengembang aplikasi dengan rentang satu sampai empat. Satu artinya sangat sulit dibaca. Dua artinya sulit dibaca. Tiga artinya mudah dibaca. Empat artinya sangat mudah dibaca.
6. Penguji dan pengembang aplikasi berdiskusi mengenai bagaimana cara memperbaiki kode.

Setelah melakukan wawancara, nilai yang telah diperoleh kemudian dikelompokkan dengan klasifikasi [1] yang dapat dilihat pada **Tabel 2-3** Klasifikasi *Code Readability*.

Tabel 2-3 Klasifikasi *Code Readability*

Nilai	Keterangan	Klasifikasi
1	Sangat sulit dibaca	Kode tidak dapat dibaca
2	Sulit dibaca	Kode tidak dapat dibaca
3	Mudah dibaca	Kode dapat dibaca
4	Sangat mudah dibaca	Kode dapat dibaca

2.19 *Integration Testing*

Integration testing adalah teknik sistematis untuk membangun arsitektur perangkat lunak sementara pada saat yang sama melakukan pengujian untuk mengungkap kesalahan yang ada kaitannya dengan antarmuka [4]. Tujuan dari *integration testing* adalah untuk mencari kesalahan dalam interaksi antar unit yang terintegrasi. Setelah semua modul telah diuji unit, pengujian integrasi dilakukan [37].

Terdapat empat pendekatan yang dapat dilakukan dalam *integration testing* di antaranya adalah sebagai berikut [38]:

1. *Big Bang*

   Pendekatan ini untuk *integration testing* di mana semua atau sebagian besar unit digabungkan bersama dan diuji sekaligus. Pendekatan ini diambil ketika tim pengujian menerima seluruh perangkat lunak dalam satu paket.

2. *Top Down*

   Pendekatan ini untuk *integration testing* di mana unit tingkat atas diuji terlebih dahulu dan unit tingkat bawah diuji langkah demi langkah setelah itu. Pendekatan ini diambil ketika pengembangan aplikasi juga menggunakan pendekatan *top-down*. *Test Stubs* diperlukan untuk menyimulasikan unit tingkat yang lebih rendah yang mungkin tidak tersedia selama fase awal.

3. *Bottom Up*

   Pendekatan ini untuk *integration testing* di mana unit tingkat bawah diuji terlebih dahulu dan unit tingkat atas langkah demi langkah setelah itu. Pendekatan ini diambil ketika pengembangan aplikasi juga menggunakan pendekatan *bottom-up*. *Test Driver* diperlukan untuk menyimulasikan unit tingkat yang lebih tinggi yang mungkin tidak tersedia selama fase awal.

4. *Sandwich/Hibrida*

   Pendekatan *integration testing* ini merupakan gabungan dari pendekatan *Top Down* dan *Bottom Up*.