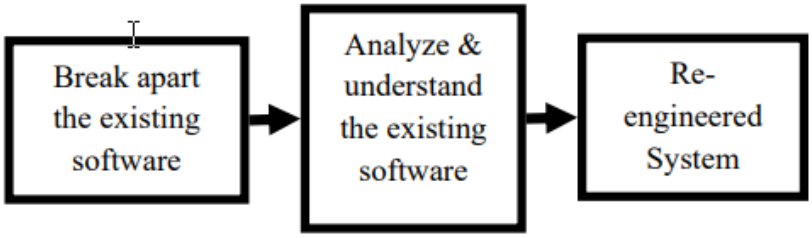
BAB 2
LANDASAN TEORI

2.1 *Software Reengineering*

Proses *Reengineering* software adalah proses memodifikasi dan merombak software yang sudah ada agar dapat lebih terpelihara [7]. Proses ini adalah menulis ulang atau restruktrurisasi keseluruhan sistem tanpa merubah fungsionalitas dari suatu sistem. Pada dasarnya *reengineering* adalah proses memeriksa kembali dan menganalisis, memperbaiki konten yang perlu diperbaiki berdasarkan kebutuhan kemudian dipecah-pecah yang pada akhirnya digabungkan kembali menjadi bentuk yang baru [8]. Alur penerapan dari software reengineering dapat dilihat pada



Gambar 2-1 Software Reengineering

2.1.1 Taksonomi *Software Reengineering*

Berikut adalah sub bagian dari taksonomi dan ruang lingkup domain dari software Reengineering

1. ***Forward Reengineering***

   Forward Engineering adalah prosedur tradisional yang dilakukan pada suatu sistem dari abstraksi high-level dan *logical* desain implementasi kepada implementasi fisik [9].

2. ***Reverse Engineering***

   Reverse Engineering adalah proses menganalisis sistem subjek untuk mengidentifikasi komponen sistem dan keterhubungannya dan membuat representasi sistem dalam bentuk lain atau pada tangkat abstraksi yang lebih tinggi [10] .

3. ***Redocumentation***

   Redocumentation adalah subarea dari *reverse engineering* dimana tujuannya adalah untuk memulihkan hilang atau tidak adanya dokumentasi dari level abstraksi seperti dokumentasi *data flow*, struktur data, dan *control flow*[11] .

4. ***Reverse Design or Design Recovery***

   Reverse Design or Design Recovery adalah menciptakan kembali abstraksi desain dari campuran dari desain dokumentasi, kode, *personal experience*, domain aplikasi dan informasi umum masalah [8].

5. ***Restructuring***

   Restructuring adalah proses mengubah data dari suatu representasi struktur ke struktur yang berbeda [8]

6. ***Recode***

   Recode adalah proses yang terdiri dari pengubahan karakteristik pengubahan program. Merestrukturisasi *control flow* dan pengubahan pada translasi pada program level [12].

7. ***Redesign***

   Redesign adalah proses memodifikasi karakteristik desain. Perubahan Kelayakan berisi peningkatkan algoritma, merestrukturisasi arsitektur desain, mengubah model sistem data [13].

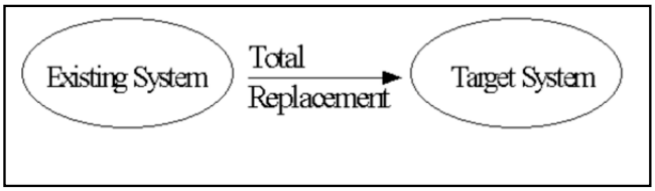
8. ***Respecify***

   Respecify adalah proses yang terdiri dari memodifikasi karakteristik *requirements*. Perubahan *requirements* memungkinkan untuk perubahan pada *requirements* yang ada saja [8].

2.1.2 Pendekatan *Software Reengineering*

1. Pendekatan *Big Bang*

Pendekatan *Big Bang* atau dikenal dengan istilah *"Lump Sum"* . Yang mana keseluruhan sistem akan diganti. Pendekatan ini digunakan ketika sistem membutuhkan penyelesaian masalah segera. Misalnya ketika sistem akan dimigrasi ke suatu arsitektur yang berbeda. Pendekatan ini terlihat pada Gambar 2-2. Intinya keuntungan dari pendekatan ini adalah sistem dimungkinkan untuk berjalan dilingkungan baru tanpa mempengaruhi integrasi *interfaces.* Namun kekurangan dari pendekatan ini adalah tidak cocok dengan sistem yang besar karena akan menghabiskan banyak waktu dan sumber daya sebelum sistem target berhasil diterapkan [3] .

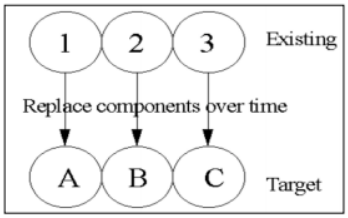


Gambar 2-2 Pendekatan *Big Bang*

2. Pendekatan *Incremental*

Pendekatan ini dikenal sebagai *"Phase Out"* atau *"Additive"*. Dalam pendekatan ini, setiap bagian dari sistem di*reengineering* kemudian ditambahkan menjadi versi baru agar kebutuhan utama terpenuhi. Pendekatan ini dapat dilihat pada **Error! Reference source not found.** yang mana memiliki keunggulan dimana bagian – bagian dari sistem yang di*reengineering* lebih cepat karena *simplicity* dalam pelacakan *errors*.

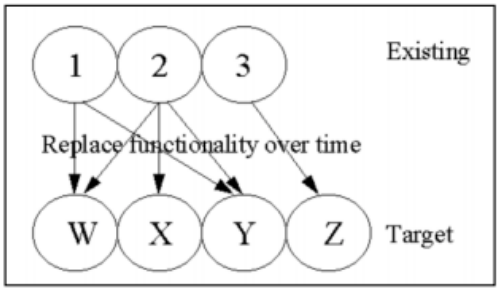
Maka dari itu pendekatan ini lebih rendah resikonya dibandingkan pendekatan *Big Bang*. Dan juga biasanya client lebih suka dengan pendekatan ini karena setiap part yang selesai bisa secara cepat disampaikan. Namun kekurangnya adalah penyampaian sistem secara lengkap akan membutuhkan waktu yang cukup lama. Dan kerugian lainnya adalah perubahan struktur secara keseluruhan selain bagian yang di*reengineering* tidak dimungkinkan untuk dirubah [3].



Gambar 2-3 Pendekatan Incremental

3. **Pendekatan *Evolutionary***

   Pendekatan ini mirip dengan pendekatan *incremental* dimana setiap komponen pada sistem yang lama diganti dengan sistem yang baru , seperti yang ditunjukan pada **Error! Reference source not found.** Namun pergantian dilakukan berdasarkan fungsionalitas bukan dari pengubahan struktur dari sistem yang lama. Pada pendekatan ini tim pengembang berfokus pada pembuatan komponen *cohesive* [3].



Gambar 2-4 Pendekatan *Evolutionary*

2.2 *Enchanced Reengineering*

Enhanced Reengineering adalah proses software *Reengineering* yang memanfaatkan banyak keunggulan dari banyak metode dan level abstraksi untuk mengubah software sistem yang sudah ada ke software sistem yang baru. Pada hal ini sistem menggunakan kedua Teknik dari *forward engineering* dan *reverse engineering.* Pada tahap pertama yang dilakukan adalah melakukan studi kelayakan untuk menguji kompatibilitas sistem kemudian membangun komponen yang diperlukan. Kemudian mengumpulkan *requirements*. Setelah itu pada tahap kedua adalah pemetaan restrukturisasi spesifikasi persyaratan software (SRS) untuk penyelesaian desain dokumen, hasil pada tahap ini adalah dokumen yang sudah

didesain ulang. Pada tahap ketiga, bagian pemograman disesuaikan berdasarkan perubahan yang telah dilakukan pada dokumen yang sudah didesain ulang, pada tahap ini bisa kembali pada tahap 2. Kemudian tahap ini melakukan retesting dan *reintegration* dari modul yang sudah ada kepada fungsi tertentu. Tahap ini, sistem membandingkan performance dari software yang sudah ada dengan software yang baru. Sebagai hasil dari keseluruhan tahap, algoritma yang lebih baik akan mengantikan algoritma dari sistem yang sudah ada. Setelah integrasi berbagai unit telah selesai maka sistem yang dimodifikasi harus diimplementasikan untuk mendapatkan sistem target yang dibutuhkan oleh pengguna [3].

Berikut adalah penjelasan dari tahap – tahap *Enchanced Reengineering* :

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.

   SRS *(Software Requirements Spesification)* adalah dokumen resmi yang memiliki semua persyaratan dalam tulisan terstruktur. Pada tahap ini untuk menentukan kembali pesyaratan sistem, maka sistem perlu dipetakan menggunakan SRS [7].

2. **Restrukturisasi Spesifikasi Kebutuhan Sistem**

   Tahap ini menggambarkan secara detail proses SRS yang direkstrukturisasi. 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 [7].

3. Design to Code

Tahap ini merekomendasikan detail tentang *design* ke proses kode. Pada tahap ini, sesuai dengan kode. Pada tahap ini disesuaikan dengan kode dokumen *redesign* yang dilakukan oleh programmer. 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 di*reengineering* dengan teknik yang baru [7].

4. Evaluasi Performa sistem baru dengan sistem yang sudah ada

   Tahap ini memberikan rincian tentang proses pengujian ulang. Untuk pengujian ulang, perangkat lunak yang sudah ada diambil kemudian dibandingkan kinerja fungsionalitasnya dengan fungsionalitas perangkat lunak yang baru [7] .

5. Implementasi

   Tahap ini adalah tahap terakhir dari mekanisme tahap *reengineering.* Pada tahap implementasi, bagian – bagian tertentu diganti berdasarkan empat tahap sebelumnya [7].

2.3 *Design Pattern*

Design Pattern mengacu pada suatu rencana, *blueprint,* atau layout yang menjadi dasar utama dari suatu software. Sebagai bagian dari pengembangan software, developer diharuskan untuk memecahkan beberapa masalah. Dari perspektif dunia nyata developer diharuskan menyelesaikan *business problem.* Dan dari perspektif pengembangan software, developer harus menyelesaikan masalah dari suatu software. Masalah software pada dasarnya adalah tugas yang ingin developer capai. Contoh masalah software adalah membuat objek dan mengisinya

dengan data dari database. Masalah yang dihadapi biasanya banyak dan seringkali berulang dan begitu pula dengan solusinya.

Selama bertahun-tahun, industri software telah mengumpulkan *wisdom* yang dapat digunakan untuk menyelesaikan masalah yang biasa terjadi diperangkat lunak. *Wisdom* ini membimbing para developer dalam pengembangan suatu aplikasi. *Design Patterns* adalah bagian penting dari hasil pengumpulan wisdom ini. Sederhananya, seiring berjalannya waktu *Design Pattern* adalah solusi yang terbukti dapat menyelesaikan suatu desain masalah yang sudah diketahui. Alih-alih menghabiskan waktu untuk menemukan solusi baru developer bisa menggunakan *Design Pattern* yang sudah digunakan dan diuji oleh developer dunia. Dengan cara ini Anda yakin bahwa Anda Pendekatan adalah pendekatan terbaik yang mungkin dalam konteks tertentu. Jika developer menemukan masalah yang sama sekali baru dan belum pernah ditangani sebelumnya, kemungkinan besar adalah bahwa tidak akan ada *design pattern* yang dapat menyelesaikan masalah itu. Untungnya, selama bertahun-tahun industri software telah mengumpulkan berbagai pola yang mencakup sebagian besar masalah yang akan developer hadapi sebagai pengembang software [14].

Dalam buku *Design Patterns: Elements of Reusable Object Oriented Software, penulis Erich Gamma, Richard Helm, Ralph Johnson,* dan *John Vlissides* telah membuat set katalog design pattern. Dan sekarang ini katalog mereka dianggap sebagai salah satu sumber informasi paling populer tentang *design pattern*. Oleh karena itu katalog tersebut yang didokumentasikan oleh empat penulis disebut dengan *Gang of Four,* atau *GoF*, *design pattern* [14].

Katalog *GoF* mencakup 23 *design pattern.* Terbagi menjadi 3 kategori yaitu *creational pattern, structural pattern, behavioral patterns.* Berikut adalah penjelasan dari setiap kategori tersebut :

1. ***Creational Design Patterns***

   Creational Design Pattern berhubungan dengan bagaimana suatu objek dibuat. Terkadang untuk melakukan instantiasi suatu objek tidaklah

mudah. Hal tersebut mungkin melibatkan beberapa logika dan kondisi. *Creational Design Pattern* dimaksudkan untuk menghilangkan kompleksitas dari *code* yang tuliskan developer. Ada lima *design pattern* dalam kategori ini yaitu :

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

2. Structural Design Patterns

Struktural Design Patterns berhubungan dengan komposisi kelas dan objek. *Patterns* ini menyederhanakan struktur suatu sistem dengan mengidentifikasi hubungan antar objek. Berikut adalah pattern pada kategori ini :

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

3. Behavioral Design Patterns

Behavioral Design Pattern berhubungan dengan interaksi dan komunikasi antara berbagai objek. Hal ini berupaya mengurangi kerumitan yang mungkin terjadi ketika objek saling berkomunikasi. Berikut adalah design pattern pada kategori ini :

- *Interpreter*
- *Template Method*

- *Chain of Responsibility*
- *Command*
- *Iterator*
- *Mediator*
- *Memento*
- *Observer*
- *State*
- *Strategy*
- *Visitor*

2.4 *Maintainability Index*

Maintainability adalah kemudahan dari perangkat lunak untuk dipelihara [15] seperti :

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

Tingkat keterpeliharaan aplikasi dapat diukur menggunakan *Maintainability Index(MI). MI* merepresentasikan kemudahan relatif dari suatu keterpeliharaan dari suatu aplikasi. Rentang nilai dari nilai *MI* berada diantara 0 sampai 100 yang mana semakin tinggi nilai maka semakin mudah pula suatu aplikasi dapat dipelihara [16]. 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) :*

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

* V : *Rata-rata Halstead*

** V(g)* : *Rata-rata Cyclomatic Complexity*

** LOC : Rata-rata Lines of code*

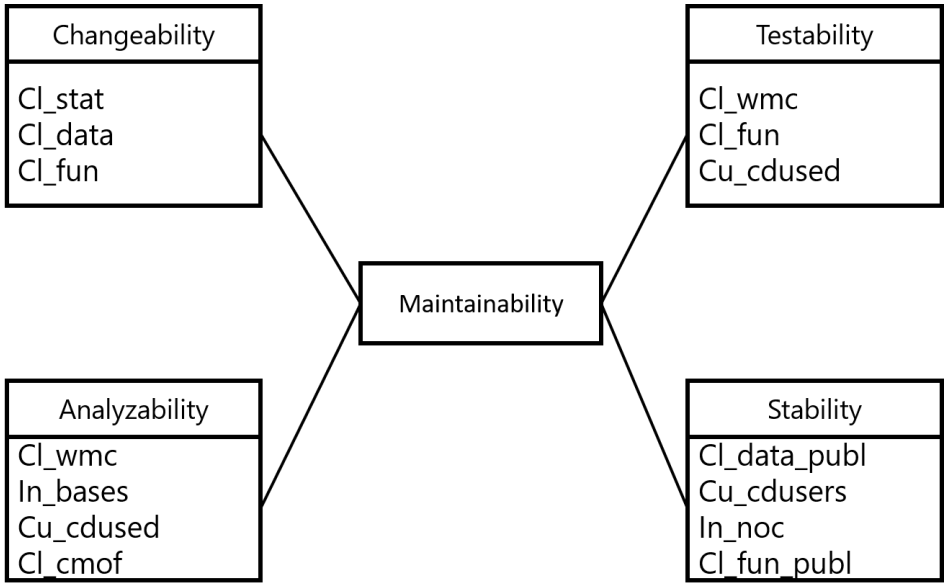
** perCM : Rata-rata jumlah comment lines*

Dari formula *MI* memiliki rentang (−∞, 171), jika suatu nilai *MI* mendekati 0 maka aplikasi merepresentasikan susah dipelihara dan jika nilai negative maka aplikasi tidak dapat dibedakan dengan negative lainnya. Agar nilai terjaga pada rentang nilai 0 sampai 100 maka formula yang direkomendasikan [16] adalah sebagai berikut :

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

Banyak penelitian untuk menghitung *maintainability* dengan *maintainability Index (MI)*[17]. Namun Ahmad A. Safian dan Areej Al-Rabadi melakukan suatu penelitian khusus untuk mendapatkan suatu formula baru untuk mengukur *maintainability* aplikasi android yang lebih baik [17]. Hal itu disebabkan karena penelitian sebelumnya pada penelian Ahmad dan Areej mengatakan bahwa harus ada formula baru yang lebih relevan untuk mendapatkan *maintainability*.

Ada empat faktor yang mempengaruhi *maintainability* dari suatu software. Faktor tersebut adalah *analyzability* yang mengukur kemampuan untuk menentukan kesalahan atau kegagalan dalam aplikasi, *changeability* yang mengukur kapabilitas untuk memodifikasi aplikasi, *stability* yang mengukur kapabilitas untuk menghidari efek yang tidak terduga dari perubahan bentuk dari aplikasi, dan *testability* yang mengukur kapabilitas untuk diuji dan *validate* dari perubahan aplikasi. **Gambar 2-5** menunjukan metrik-metrik yang mempengaruhi faktor-faktor yang mempengaruhi *maintainability*.



Gambar 2-5 Software Metrics *Maintaintainability Factor*

Penjelasan dari metrik-metrik pada faktor-faktor yang mempengaruhi *maintainability* pada **Gambar 2-5** dapat dilihat pada **Tabel 2-1**.

Tabel 2-1 Penjelasan metrik-metrik yang mempengaruhi *maintainability*

Metrik	Definisi	Dampak
Cl_wmc	(*Weighted method perclass*) mengukur kompleksitas pada fungsi / *method*	*Complexity, understandability* dan *cohesion*
In_bases	Jumlah kelas dari kelas yang mewarisi langsung atau tidak	*Complexity* dan *understandability*
Cu_cdused	Jumlah kelas yang dipakai pada kelas ini.	*Coupling, complexity* dan *reuse of methods*
Cl_cmof	Rasio antara jumlah baris komentar dan total jumlah baris dalam kode.	Ukuran Kelas
Cl_stat	Jumlah dari *executable statements*.	Ukuran Kelas
Cl_data	Jumlah total dari atribut yang terdapat pada kelas.	*Complexity* dan Ukuran Kelas

Metrik	Definisi	Dampak
Cl_fun	Jumlah total dari fungsi yang terdapat pada kelas.	*Complexity* dan *cohesion*
Cl_fun_publ	Jumlah total dari fungsi *public* yang terdapat pada kelas.	*Coupling* dan *polymorphism*
Cu_cdusers	Jumlah kelas yang menggunakan kelas ini.	*Coupling*, *changeability* dan *reuse of methods*.
In_noc	Jumlah anak kelas atau *sub class* pada kelas ini.	*Reuse of methods* dan *testing*
Cl_data_publ	Jumlah total dari atribut *public* yang terdapat pada kelas ini.	*Encapsulation*

Metrik-metrik pada **Tabel 2-1** memiliki rentang yang dapat diterima agar aplikasi *maintainability*nya baik. Rentang nilai metrik agar aplikasi memiliki maintainability yang baik bisa dilihat di **Tabel 2-2**.

Tabel 2-2 Rentang nilai metrik yang dapat diterima untuk *maintainability*

Metrik	Rentang (min-maks)
Cl_wmc	0-11
Cl_fun	0-9
Cu_edused	0-6
Cl_stat	0-7
Cl_data	0-25
Cl_data_publ	0-7
Cu_cdusers	0-3
In_noc	0-5
Cl_fun_publ	0-7
In_bases	0-4
Cl_cmof	0-100

Formula baru untuk *maintainability index (MI)* yang dapat diterapkan pada aplikasi android adalah sebagai berikut :

$$\textbf{\textit{Maintainability Index}} = NOP + IN\_NOC + IN\_BASES + 2 * CL\_FUN + CL\_CMOF + CL\_DATA + CL\_STAT + CL\_FUN\_PUB + CU\_CDUSERS + CL\_DATA\_PUB + 2*CU\_CDUSED + 2*CL\_WMC$$

** NOP adalah jumlah permissions yang diminta aplikasi agar aplikasi itu dapat berjalan (<uses-permission>)*

Maintainability Index mempunyai rentang nilai yang menandakan tingkat *maintainability* dari suatu aplikasi. Semakin tinggi suatu nilai maka semakin baik pula tingkat *maintainability*nya [16]. Rentang nilai untuk maintainability index dapat dilihat pada **Tabel 2-3**.

Tabel 2-3 Rentang nilai *maintainability index*

Rentang	Keterangan
$20 \leq MI \leq 100$	Dapat dipelihara dengan baik
$10 \leq MI < 20$	Cukup untuk dapat dipelihara
$0 \leq MI < 10$	Sulit untuk dapat dipelihara

2.5 *Halstead Metrics*

Halstead metric 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[18]. Pada Halstead's metric 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 the program* dirumuskan pada persamaan sebagai berikut :

   $$N = N1 + N2$$

   Keterangan:
   N1 = total semua operator yang muncul
   N2 = total semua operan yang muncul

2. *Vocabulary of the program*

   Vocabulary of the program adalah kalkulasi jumlah operator dan operand unik 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 unik
   n2 = jumlah operand unik

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 metric* 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 operan yang muncul

   n1 = jumlah operator unik

   n2 = jumlah operan unik

5. *Effort*

   Effort dalam *Halstead metric* 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 kalkulasi Difficulty

   V = nilai kalkulasi Volume of the program

6. *Number of bugs expected in the program*

   Number of bugs expected in the program dalam *Halstead metric* di gunakan 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 = *dari kalkulasi Volume of the program*

2.6 *Cohesion* dan *Coupling*

Pada pembuatan aplikasi dengan menggunakan paradigma berbasis objek, aplikasi harus mempunyai kelas desain yang baik karena hal tersebut akan berguna pada saat implementasi *software architecture* yang mana hal itu akan membantu pada solusi bisnis aplikasi [2]. Ada empat tipe kelas desain yang mana setiap tipe merepresentasikan layer yang berbeda pada *design architecture*. Tipe-tipe *design classes* adalah *user interface classes, business classes, process classes, persistent classes, system classes*.

Pada *system classes* menurut Arlow dan Neustadt untuk memastikan kelas terbentuk dengan baik maka salah satu faktornya adalah *high cohesion* dan *low coupling*. *High cohesion* adalah desain kelas yang kohesif memiliki serangkaian tanggung-jawab yang kecil dan terfokus, serta menerapkan atribut dan metode secara tunggal untuk melaksanakan tugas tersebut. Dan *low coupling* merupakan ukuran untuk tingkat kolaborasi kelas yang mempunyai rentang nilai yang dapat diterima.

Nilai *coupling* bisa didapatkan dengan menggunakan CBO *(Coupling Between Objects)* yaitu dengan menghitung jumlah instance object yang dilakukan pada suatu kelas. Rentang nilai CBO berdasarkan hasil penelitian dari *plugin* CodeMR dapat dilihat pada **Tabel 2-4**.

Tabel 2-4 Rentang Nilai *Coupling Between Objects*

Rentang	Keterangan
<= 5	Rendah

Rentang	Keterangan
6 – 10	Rendah-Sedang
11 – 20	Sedang-Tinggi
21 – 30	Tinggi
> 30	Sangat Tinggi

Sedangkan untuk mengukur cohesion pada penelitian ini menggunakan $CAMC_s$ (*Cohesion Among Methods)* dengan formula sebagai berikut [19] :

$$\boldsymbol{CAMC}\ (\boldsymbol{C}) = \frac{\sigma + k}{k(l+1)}$$

Keterangan:

σ = Jumlah seluruh parameter unik pada kelas

k = Jumlah *method*

l = Jumlah parameter unik

Karena rentang nilai *cohesion* yang dapat diterima adalah 0-1 [20]. Dan untuk mempermudah penelitian maka semakin tinggi nilai yang didapat pada *cohesion*

maka semakin buruk *cohesion (Lack of Cohesion)*, jadi formula dimodifikasi menjadi berikut **LCAM = 1 – CAM**. Namun formula **CAM** *(Cohesion Among Methods)* tidak dapat diterapkan pada bahasa pemograman *Dynamically Typed* seperti PHP karena *Dynamically Typed* tidak menyertakan tipe data pada parameternya sehingga ada formula lain untuk menghitung *cohesion* yaitu LCOM3*(Lack of cohesion of methods)* [21] yaitu sebagai berikut:

$$\mathbf{LCOM3} = (\text{m-sum(mA)/a}) \;/\; (\text{m-1})$$

Keterangan:
m = jumlah metode pada kelas
a = jumlah variabel pada kelas
mA= jumlah metode yang mengakses suatu variabel
sum(mA) = jumlah mA keseluruhan pada kelas

Rentang nilai *cohesion* yang dapat diterima berdasarkan hasil penelitian plugin CodeMR dapat dilihat pada **Tabel 2-5**.

Tabel 2-5 Rentang *Nilai Lack of Cohesion*

Rentang	Keterangan
<= 0.6	Rendah
0.61 – 0.7	Rendah-Sedang
0.71 – 0.8	Sedang-Tinggi
0.81 – 0.9	Tinggi
> 0.9	Sangat Tinggi

2.7 *Refactoring*

Refactoring menurut Martin Flower mempunyai dua definisi yang tergantung konteks. Definisi pertama yaitu *refactoring* adalah suatu pengubahan pada struktur internal aplikasi agar dapat aplikasi tersebut lebih mudah dipahami dan mudah dalam modifikasi tanpa mengubah *behavior* dari aplikasi. Definisi kedua adalah refactoring adalah restrukturisasi suatu aplikasi dengan melakukan serangkaian *refactoring* tanpa mengubah behavior aplikasi [22].

Alasan Refactoring

Menurut Martin Flower ada beberapa alasan kenapa *code* pada suatu aplikasi harus di*refactor* yaitu sebagai berikut :

1. **Meningkatkan suatu desain aplikasi**

   Tanpa adanya *refactoring* desain pada suatu aplikasi akan mengalami kerusakan. Hal tersebut bisa terjadi karena programmer melakukan perubahan pada *code* untuk tujuan jangka pendek atau perubahan pada *code* dilakukan tanpa adanya pemahaman pada *design code* pada suatu aplikasi. Hal tersebut bisa menjadi efek kumulatif sehingga mengakibatkan *design code* menjadi sangat susah dibaca.

2. **Membuat Aplikasi menjadi lebih mudah dipahami**

   Programming bukan hanya tentang bagaimana komputer melakukan apapun yang ditulis oleh programmer. Tapi bagaimana juga programmer lain dapat membaca *code* dan bagaimana programmer lain melakukan suatu perubahan. *Readability code* dapat mempengaruhi lama proses programmer untuk melakukan suatu perubahan. *Readability code* yang kurang baik dapat mengakibatkan lama proses pengubahan. *Refactoring* adalah salah satu cara untuk membuat *code* menjadi *readable.*

3. Memudahkan untuk mencari *Bugs*

Menurut pernyataan dari Kent Beck yaitu *"Refactoring helps me be much more effective at writing robust code"*. Hal ini punya maksud bahwa dengan *refactoring*, programmer akan memahami secara mendalam maksud dari *code* sebelumnya kemudian akan menambahkan pemahaman baru pada code tersebut. Dengan mengklarifikasi struktur aplikasi, maka programmer pelaku *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 tanpa desain 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 pembuatan atau performa aplikasi. Proses *Refactoring* akan berfokus pada meningkatkan performa dan mencari *bugs* daripada menambah suatu fitur baru.

2.8 *Understandability*

Understandability adalah tingkat di mana arti dari sistem atau modul harus jelas kepada pengguna atau pengembang. *Software Understandability factors* merupakan pembahasan pada COCOMO II *(The Constructive Cost Model II)* [23]. Skala *Software Understandability Factors* dapat dilihat pada **Tabel 2-6**.

Tabel 2-6 *Software Understandability Factors*

Factor	***Very Low***	***Low***	***Nominal***	***High***	***Very High***
Structure	*Very low cohesion, high coupling, spaghetti code*	*Moderately low cohesion, high coupling*	Cukup terstuktur namun ada beberapa *area* yang masih lemah	*High cohesion, low coupling*	Modularitas yang kuat, menerapkan *information hiding in data/control structures*
Application clarity	Tidak ada kecocokan antara program dan	Beberapa korelasi antara program dan aplikasi	*Moderate* Korelasi antara program dan aplikasi	*Good* Korelasi antara program dan aplikasi	*Clear match* antara program dan environment aplikasi

	Environment aplikasi				
Self - descriptiveness	Kode yang tidak jelas*;* Tidak ada dokumentasi,	Beberapa kode memeliki kode komentar dan memiliki header yang berguna	*Moderate level* Kode Komentar, *headers, documentation*	*Good* Kode komentar dan memiliki *headers;* Memiliki dokumentasi yang berguna tapi masih ada kelemahan di beberapa area	*Self-descriptive code;* Dokumentasi yang terbarui, Dibangun dengan struktur yang baik

2.9 Android

Android adalah sistem operasi yang dirancang oleh Google dengan basis kernel Linux untuk mendukung kinerja perangkat elektronik layar sentuh, seperti tablet atau smartphone. Jadi, android digunakan dengan sentuhan, gesekan ataupun ketukan pada layar gadget [24].

Android bersifat open source atau bebas digunakan, dimodifikasi, diperbaiki dan didistribusikan oleh para pembuat ataupun pengembang perangkat lunak. Dengan sifat open source perusahaan teknologi bebas menggunakan OS ini diperangkatnya tanpa lisensi alias gratis[25] .

Begitupun dengan para developer, mereka bebas membuat aplikasi dengan kode-kode sumber yang dikeluarkan google. Dengan seperti itu android memiliki jutaan support aplikasi gratis/berbayar yang dapat diunduh melalui google play [24].

2.9.1 *Android Version*

Dari rilis perdananya hingga saat ini, Android telah berubah secara visual, konseptual dan fungsional. Berikut adalah perkembangan dari android [26] :

1. 1.0 – 1.1
2. Cupcake: 1.5
3. Donut: 1.6
4. Éclair: 2.0 – 2.1
5. Froyo: 2.2
6. Gingerbread: 2.3
7. Honeycomb: 3.0 – 3.2
8. Ice Cream Sandwich: 4.0
9. Jelly Bean: 4.1 – 4.3
10. Kitkat: 4.4
11. Lollipop: 5.0 – 5.1
12. Marshmallow: 6.0

13. Nougat: 7.0 – 7.1
14. Oreo: 8.0 dan 8.1
15. Pie: 9
16. 10

2.9.2 *Android Market*

Pangsa smartphone Android berkisar sekitar 87% dengan sedikit peningkatan di seluruh perkiraan. Pada tahun 2019, pangsa OS akan meningkat menjadi 87,0% dari 85,1% pada tahun 2018 sebagian besar karena peluncuran beberapa peluncuran 5G dan pembersihan persediaan yang dipercepat dari perangkat yang lebih tua. Volume diperkirakan akan tumbuh pada tingkat pertumbuhan tahunan gabungan lima tahun (CAGR) sebesar 1,7% dengan pengiriman 1,30 miliar pada tahun 2023. Harga jual rata-rata Android (ASP) diperkirakan tumbuh sebesar 3,2% pada tahun 2019 menjadi US $ 263, naik dari US $ 254 pada 2018.

Di Indonesia pada beberapa tahun terakhir pangsa pasar selalu berada diatas 90% dan pada tahun 2019 pangsa android mencapai 93.22%. Pangsa tersebut sangat tidak berimbang dengan pangsa sistem operasi smartphone lainnya di Indonesia seperti rival dari android yaitu iOS pada tahun 2019 hanya mampu mencapai 6.38% [27].

2.9.3 *Android Native Application Architectures*

Pengembangan *Mobile App* sekarang ini menjadi bagian penting dari industri software dengan android sebagai ekosistem terbesar saat ini. Pada pembuatan aplikasi android terdapat beberapa arsitektur software yang biasa digunakan yaitu MVC, MVP, dan MVVM [28]. Setiap arsitektur tersebut memiliki *understanding* dan *experience* yang berbeda setiap developernya. Dengan demikian pilihan dan penerapan arsitektur software tersebut bervariasi antara developer ke developer lainnya. Sejauh ini tidak ada suatu penelitian yang memberikan pandangan komprehensif tentang pengunaan arsitektur-arsitektur tersebut [29].

Menurut penelitian dari jurnal *"An Exploratory Study of MVC-based Architectural Patterns in Android Apps"*, MVC adalah arsitektur paling populer

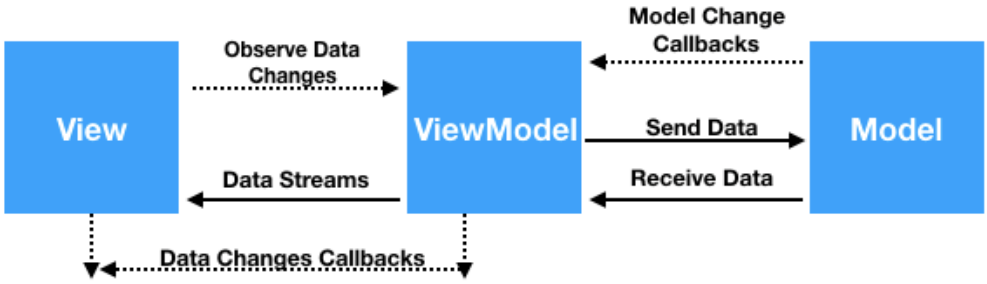
yang mana mendapat 57% dari 5480 aplikasi yang diunduh dari google play store, sedangkan MVP merupakan arsitektur paling rendah pemakaiannya yaitu 8% [29]. Namun setelah dilakukan komparasi berdasarkan *testability*, *modifiability*, dan *performance* ternyata untuk *testability* MVVM adalah paling baik yang mana membutuhkan *test cases* paling sedikit diantara lainnya. Sedangkan untuk *modifiability* dan *performance* MVP dan MVVM lebih baik dari MVP karena MVP dan MVVM memiliki tingkat Coupling dan pemakaian memori yang rendah [28].

2.10 *Model View ViewModel (MVVM)*

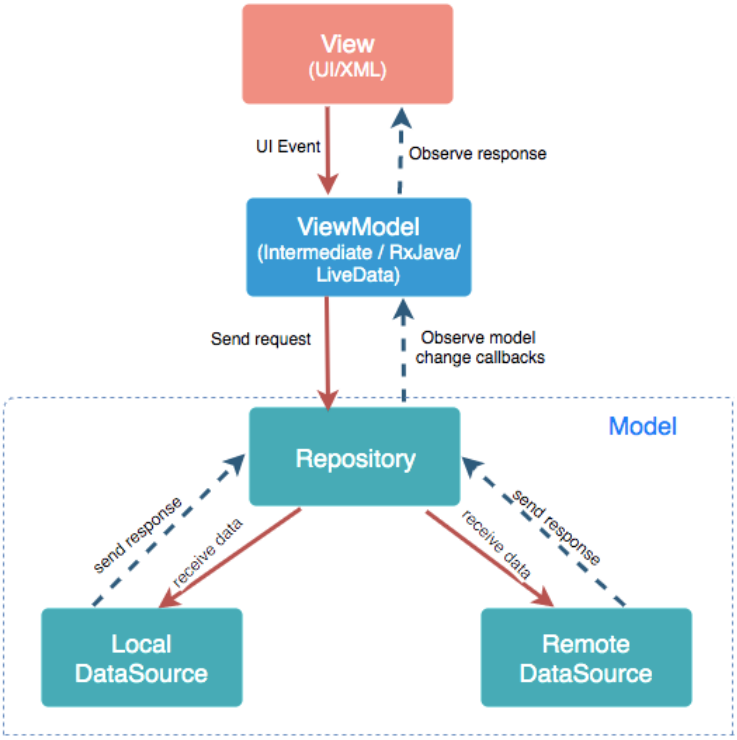
Arsitektur Model-View-ViewModel (MVVM) pertama kali dikenalkan oleh Jhon Grossman pada blognya. Arsitektur ini awalnya digunakan untuk Microsoft Silverlight dan WPF yang mana arsitektur tersebut juga berbasiskan pada arsitektur MVC.

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 mengkolaborasikan *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* [28].

Secara umum hubungan antara *Model*, *View*, dan *ViewModel* bisa dilihat pada **Gambar 2-6**. Berdasarkan pengalaman yang dilalui oleh peneliti tentang hubungan antara MVVM biasanya pada *Model* terdapat suatu *pattern* bernama *Repository Pattern* yang diperlihatkan pada **Error! Reference source not found.**. *Repository pattern* mengatur data berdasarkan data sumbernya sebelumnya dialirkan ke ViewModel.



Gambar 2-6 Hubungan antar Komponen pada MVVM



Gambar 2-7 Hubungan antar Komponen pada MVVM (2)

2.11 *Design Principles*

Design Principles pada software biasanya diidentikan dengan *SOLID principles*. *SOLID principles* adalah suatu bagaimana cara untuk mengatur fungsi-fungsi dan struktur data menjadi suatu class. Dan juga mengatur bagaimana cara class-class saling berkorelasi. Penggunaan kata “class” pada SOLID tidak hanya mengacu pada suatu software 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 software.

Berikut adalah penjelasan dari SOLID [30]:

- **SRP*: The Single Responsibility Principle***

  Single Responsibility Principle (SRP) adalah prinsip pemrograman komputer yang menyatakan bahwa setiap modul atau kelas harus memiliki tanggung jawab atas satu 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 sebagai, " *A class should have only one reason to change*" yang mana setiap kelas hanya harus punya satu alas an untuk berubah.

- **OCP*: The 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.

- **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.

- **ISP*: The Interface Segregation Principle***

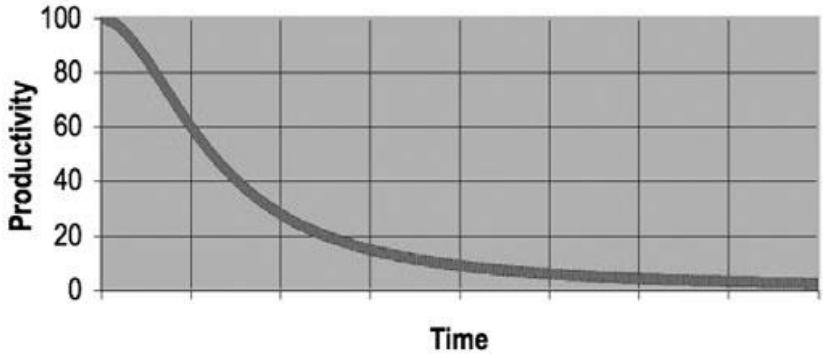
  Prinsip ini menyarankan perancang perangkat lunak untuk menghindari tergantung pada hal-hal yang mereka jangan gunakan.

- **DIP*: The 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.

2.12 *Clean Code*

Clean code merupakan suatu konsep atau petunjuk penulisan struktur kode yang bersih *(Clean)*, di mana kode yang ditulis dapat dibaca oleh pengembang lain dan dapat diubah dengan mudah. *Clean code* bertujuan untuk mengatasi penurunan tingkat produktivitas pengembangan perangkat lunak akibat dari struktur kode yang berantakan seperti yang dapat dilihat pada **Gambar 2-8** di mana waktu dapat mempengaruhi tingkat produktivitas [31]. Adapun beberapa hal inti yang dibahas pada *Clean Code* yaitu *Meaningful Names, Clean Functions, Clean Comments, Clean Code Formatting, Clean Error Handling, Clean Object and Data Structure, Clean Classes.*



Gambar 2-8 *Productivity vs Time*

2.12.1 *Meaningful Name*

Pada konsep ini terdapat petunjuk dalam melakukan pemberian nama terhadap variabel, *method*/fungsi, dan *kelas* agar lebih mudah untuk dipahami. Berikut merupakan petunjuk yang dapat digunakan dalam melakukan pemberian nama:

1) Nama yang dapat dieja dan memiliki arti

Penggunaan nama yang dapat dieja dan secara eksplisit menerangkan kegunaan dari kode yang ditulis dapat memudahkan pengembang lain dalam memahaminya. Sehingga pengembang tidak perlu mengeluarkan usaha yang banyak hanya untuk memahami kegunaan kode tersebut. Berikut contoh nama yang dapat dieja dan memiliki arti:

Buruk

```
1. int i;
2. String dd; //ini hari
3. public list<mhs>
```

Baik

```
1. Int maxArraySize;
2. String day;
3. public list<Mahasiswa>
```

2) Nama yang mudah untuk dicari

Petunjuk ini digunakan untuk memudahkan pencarian kode. Dengan menggunakan penamaan yang mudah dicari, pengembang tidak perlu memahami program secara keseluruhan. Hal ini biasa diterapkan untuk menamai suatu konstanta. Berikut contoh nama yang mudah untuk dicari:

Buruk

```
1. for (int j=0; j<34; j++){
2.     s += (t[j]*4)/5;
3. }
```

Baik

```
1. int realDaysPerIdealDay = 4;
2. const int WORK_DAYS_PER_WEEK = 5;
3. int sum = 0;
4. for (int j=0; j < NUMBER_OF_TASKS; j++) {
5.      int realTaskDays = taskEstimate[j] * realDaysPerI
   dealDay;
6.      int realTaskWeeks = (realdays / WORK_DAYS_PER_WEE
   K);
7.      sum += realTaskWeeks;
8. }
```

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 contoh kode dengan menghindari *mental mapping*:

Buruk

```
1. Data user = Data();
2. String nama = user.getNama();
```

Baik

```
1. Mahasiswa user = Mahasiswa();
2. String nama = user.getNama();
```

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 untuk menggunakan 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 contoh kode kerja yang buruk dan baik:

Buruk

```
1. class Mahasiswa(){
2.    String nama;
3.
4.    public void nama(){}
5.
6. }
7.
```

Baik

```
1. class Mahasiswa(){
2.    String nama;
3.
4.    public void setNama(){}
5.
6. }
7.
```

6) Menghindari penggunaan konteks yang tidak diperlukan

Hal ini biasa terjadi pada suatu *kelas* atau pada objek, di mana atributnya mengandung nama dari *kelas* atau objek tempat atribut tersebut berada. Hal

tersebut tidak disarankan, karena bersifat repetitif. Berikut adalah contih dari penggunaan konteks yang tidak diperlukan :

Buruk

```
1. void setMajor(HashMap student) {
2.     student[“StudentAlamamater”] = 'UNISBA';
3. }
```

Baik

```
1. void setMajor(HashMap student) {
2.     student[“alamamater”] = 'UNISBA';
3. }
```

2.12.2 *Clean Function*

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

1) Jumlah parameter

Dalam menulis *method*/fungsi yang memiliki parameter, jumlah parameter ideal dari suatu fungsi maksimal sebanyak 2 parameter. Hal tersebut dimaksudkan untuk menghindari banyaknya varian pengujian terhadap *method*/fungsi yang ditulis. Apabila terdapat kebutuhan di mana parameter yang digunakan lebih dari jumlah ideal, cukup menggunakan objek sebagai parameter dari *method*/fungsi tersebut. Berikut adalah contoh penggunaan parameter yang baik :

Buruk

```
1. void makeShape(x,y,z,density){
2.    //..
3. }
```

Baik

```
1. void makeShape(coordinate, density){
2.    //..
3. }
```

2) Menghindari penggunaan *flag* sebagai parameter

Penggunaan *flag* sebagai parameter dapat menandakan *method*/fungsi yang dibangun memiliki pekerjaan lebih dari satu. Berikut adalah contoh penggunaan flag sebagai parameter:

Buruk

```
1. class File {
2.   void createFile(name, temp) {
3.       if (temp) {
4.          fs.create(`./temp/${name}`);
5.       } else {
6.          fs.create(name);
7.       }
8.   }
9. }
```

Baik

```
1. class File {
2.   void createFile(name) {
3.      fs.create(name);
4.   }
5.
6.   void createTempFile(name) {
7.      createFile(`./temp/${name}`);
8.   }
9. }
```

3) Jumlah pekerjaan

Suatu *method*/fungsi disarankan untuk melakukan hanya satu pekerjaan saja. Suatu *method*/fungsi yang melakukan lebih dari satu pekerjaan akan sulit untuk dibuat, diuji, dan dipahami. *Method*/fungsi yang lebih kecil akan lebih mudah untuk dimodifikasi. Apabila *method*/fungsi yang ditemukan melakukan lebih dari satu pekerjaan, pecah pekerjaan-pekerjaan tersebut ke dalam *method*/fungsi yang berbeda. Berikut contoh kode jumlah pekerjaan yang baik:

Buruk

```
1. class Absensi(){
```

```
2.    String nama;
3.    Date waktuDatang;
4.
5.    void setNama(String nama){
6.       this.nama = nama
7.       this.waktuDatang = Date().getDate();
8.    }
9. }
```

Baik

```
1. class Absensi(){
2.    String nama;
3.    Date waktuDatang;
4.
5.    void setNama(String nama){
6.       this.nama = nama;
7.    }
8.
9.    void setWaktuDatang(){
10.          this.date = Date().getDate()
11.       }
12.    }
```

2.12.3 *Clean Comment*

Pada konsep ini terdapat petunjuk dalam melakukan pemberian komentar yang baik dan efisien, agar komentar yang ditulis dapat memberikan informasi yang tidak tersampaikan oleh kode sumber yang sudah ditulis. Akan tetapi, 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

Pada penulisan komentar disarankan memberikan informasi yang informatif. Maksud dari informatif adalah memberikan informasi sebaiknya agar orang non

teknis bisa membaca komentar tersebut. Berikut contoh kode komentar yang informatif :

Buruk

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

Baik

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

2) **Menggunakan Klarifikasi**

Jika suatu kode menggunakan library yang tidak dapat diubah oleh programmer. Maka disarankan untuk memberikan komentar untuk memberikan klarifikasi agar programmer lainnya dapat memahami maksud dari suatu kode. Contoh kode komentar klarifikasi adalah sebagai berikut:

```
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**

Ada beberapa kode yang mempunyai konsekuensi buruk pada suatu aplikasi namun diperlukan untuk kondisi-kondisi tertentu. Disarankan untuk memberikan komentar peringatan jika kode tersebut memiliki konsekuensi buruk. Contoh komentar peringatan adalah sebagai berikut:

```
1.// Don't run unless you
2.// have some time to kill.
```

```
3.public void _testWithReallyBigFile() {
4.    writeLinesToFile(10000000);
5.    response.setBody(testFile);
6.    response.readyToSend(this);
7.    String responseString = output.toString();
8.}
```

4) **Menggunakan Komentar Catatan *Todo***

Ada kemungkinan programmer tidak sempat menyelesaikan aplikasi secara keseluruhan. Untuk kode-kode yang belum terselesaikan disarankan untuk mengunakan catatan todo. Contoh komentar catatan *todo* adalah sebagai berikut:

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

5) **Hindari Komentar pada kode yang sudah jelas**

Seringkali programmer memberikan komentar pada kode yang sudah jelas pembacaannya. Hal ini akan memberikan informasi yang kurang berguna. Berikut contoh komentar pada kode yang sudah jelas:

```
1. /*
2. set the value of the age integer to 32
3. */
4. int age = 32;
```

2.12.4 *Clean Error Handling*

Pada konsep ini terdapat petunjuk dalam menggunakan penanganan kesalahan *(Error Handling)* agar dapat digunakan secara efisien dan pesan kesalahan dapat ditampilkan dengan mudah. Petunjuk sederhana dalam menerapkan penanganan kesalahan yang bersih yaitu dengan tidak mengabaikan kesalahan yang tertangkap.

Kesalahan yang terjadi biasanya hanya ditanggulangi dengan cara menampilkannya ke *logger* tanpa tindakan lebih lanjut sehingga pengguna tidak mengetahui kesalahan apa yang terjadi saat menggunakan aplikasi. Diperlukan tindakan lebih lanjut agar pesan kesalahan tidak hanya ditampilkan ke *logger*, tetapi aplikasi dapat memberikan pesan berupa notifikasi, baik terhadap pengguna maupun pengembang, tentang kesalahan yang terjadi. Berikut adalah contoh *clean erro handling* adalah sebagai berikut:

Buruk

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

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);
  }
```

2.12.5 *Clean Object and Data Structure*

Pada konsep ini terdapat petunjuk dalam membuat struktur data yang bersih di mana struktur data memiliki abstraksi sehingga tidak dapat diakses langsung secara publik. Berikut merupakan petunjuk dalam membuat struktur data yang bersih:

1) **Penggunaan *Data Abstraction***

   Pada pembuatan struktur data disarankan menggunakan *hiding implementation (abstraction)*. Hal ini disarankan agar aplikasi dapat mudah dipelihara karena

programmer dapat leluasa untuk memodifikasi tanpa ketakutan merusak struktur aplikasi. Berikut contoh penggunaan *data abstraction* di struktur data:

Buruk

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

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.}
```

2) **Membuat objek memiliki *private member***

Struktur data yang baik disarankan menggunakan private member. Hal ini berguna untuk *hiding implementation* yang mana programmer tidak perlu mengetahui bagaimana proses didalamnya. Berikut adalah contoh penggunaan 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 : "+ getKelilin
   g());
13.       }
```

```
14.     }
```

1) ***Getter* dan *Setter***

Getter dan *Setter* digunakan untuk mengakses struktur data yang bersifat *private*. *Getter* dan S*etter* bertujuan untuk menghindari mutasinya nilai pada struktur data yang dilakukan sengaja maupun tidak. Berikut adalah contoh penggunaan *getter* dan *setter*:

```
1. public Mahasiswa(){
2.    String nama;
3.
4.    public void setNama(String nama){
5.       this.nama = nama;
6.    }
7.
8.    public String getNama(){
9.       return this.nama;
10.        }
11.    }
```

2.12.6 *Clean Class*

Pada konsep ini terdapat petunjuk dalam membuat *kelas* yang lebih bersih dan terorganisir. Berikut merupakan petunjuk yang dapat dalam membuat *kelas*:

1) Class Organization

Pembuatan *kelas* yang baik dapat dilakukan dengan mengikuti *code convetion/code styling* dari bahasa pemrograman yang digunakan.

2) Single Responsibility Principle

Suatu *kelas* disarankan untuk berukuran tidak terlalu besar, selain banyak jumlah baris yang ditulis, pengembang akan kerepotan dalam memahami *kelas* tersebut. Dengan menjaga ukuran kelas untuk tidak terlalu besar, dapat mempermudah proses modifikasi. Ukuran suatu kelas diukur berdasarkan jumlah *responsibility* yang ditanggung oleh kelas tersebut. Kelas yang bersih merupakan *kelas* yang 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 adalah contoh penggunaan *single responsibility*:

Buruk

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

Baik

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

2.13 *Cyclomatic Complexity*

Cyclomatic Complexity merupakan metrik untuk mengukur kompleksitas pada sebuah fungsi dengan memperhatikan graf kendali, singkatnya apabila fungsi tidak memiliki percabangan maka kompleksitasnya 1 [32]. Apabila memiliki banyak percabangan, maka perhitungannya dapat mengikuti formula berikut:

$$M = E - N + 2$$

Keterangan:

$M = Cyclomatic\ Complexity$

$E = Jumlah\ edge\ pada\ graf$

$N = Jumlah\ node\ pada\ graf$

Jumlah *Cyclomatic Complexity* pada suatu fungsi disarankan kurang dari 15. Apabila melebihi 15, artinya terdapat sekitar 15 jalur eksekusi percabangan pada fungsi tersebut. Lebih dari itu, jalur eksekusi akan semakin sulit untuk diidentifikasi dan diperiksa. Adapun batas tertinggi jumlah jalur eksekusi dalam sebuah modul adalah 100 [32].

Cyclomatic Complexity mempunyai rentang nilai yang menunjukan kompleksitas suatu aplikasi. Nilai *Cyclomatic Complexity* yang dibawah 10 masih dalam rentang yang bisa diterima [33]. Rentang nilai *cyclomatic complexity* yang dapat diterima dapat dilihat pada **Tabel 2-7.**

Tabel 2-7 Rentang Nilai *Cyclomatic Complexity* dapat diterima

Rentang	Keterangan
1 – 10	Resiko Minimal
11 - 20	Resiko Sedang
21 – 50	Resiko Tinggi
50 <	Resiko Sangat Tinggi

2.14 *S-Health SDK*

Samsung Digital Health SDK merupakan pustaka atau library yang dibuat oleh Samsung untuk memudahkan para pengembang untuk sinkronisasi data kesehatan dengan aplikasi S Health versi 4.0 ke atas dan untuk membuat aplikasi kesehatan yang bermanfaat dengan memanfaatkan data kesehatan tersebut. Penyimpanan data kesehatan dapat diintegrasikan pada data kesehatan di S Health yang dapat berbagi dengan rekan aplikasi S Health lainnya berdasarkan minat pengguna. S Health hanya dapat mendukung perangkat Android dengan sistem operasi KitKat 4.4 termasuk perangkat Samsung. Adapun arsitektur untuk kerangka data dengan menggunakan library ini adalah sebagai berikut [34].

2.15 Analisis dan Design Berorientasi Objek

Analisis dan Desain Berorientasi Objek (*Object Oriented Analysis and Design*) adalah cara baru dalam memikirkan suatu masalah dengan menggunakan model yang dibuat menurut konsep. Dasar pembuatannya sendiri adalah objek yang merupakan kombinasi antara struktur data dan perilaku dalam satu entitas. Alasan mengapa harus memakai metode berorientasi objek yaitu karena perangkat lunak itu sendiri yang bersifat dinamis, di mana hal ini disebabkan karena kebutuhan pengguna berubah dengan cepat [35].

Selain itu bertujuan untuk menghilangkan kompleksitas transisi antar tahap pada pengembangan perangkat lunak, karena pada pendekatan berorientasi objek, notasi yang digunakan pada tahap analisis perancangan dan implementasi relatif sama tidak seperti pendekatan konvensional yang dikarenakan notasi yang digunakan pada tahap analisanya berbeda-beda. Hal itu menyebabkan transisi antar tahap pengembangan menjadi kompleks.

Di samping itu dengan pendekatan berorientasi objek membawa pengguna kepada abstraksi atau istilah yang lebih dekat dengan dunia nyata, karena di dunia nyata itu sendiri yang sering pengguna lihat adalah objeknya bukan fungsinya. Beda ceritanya dengan pendekatan terstruktur yang hanya mendukung abstraksi pada

level fungsional. Adapun dalam pemrograman berorientasi objek menekankan berbagai konsep seperti: *Class, Object, Abstract, Encapsulation, Polymorphism, Inheritance* dan tentunya UML (*Unified Modeling Language*)[36].

UML (*Unified Modeling Language*) sendiri merupakan salah satu alat bantu yang dapat digunakan dalam Bahasa pemrograman berorientasi objek. Selain itu UML 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: Spesifikasi, Visualisasi, Desain Arsitektur, Konstruksi, Simulasi dan Testing. Dapat disimpulkan bahwa UML (*Unified Modeling Language*) adalah sebuah Bahasa yang berdasarkan grafik atau gambar untuk memvisualisasikan, melakukan spesifikasi, membangun dan pendokumentasian dari sebuah sistem pengembangan perangkat lunak berbasis objek (*Object Oriented Programming*).

Dokumentasi UML menyediakan 10 macam diagram [37] untuk membuat model aplikasi berorientasi objek yang 4 di antaranya sebagai berikut:

1. ***Use Case Diagram***

   Use case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Di dalam *use case diagram* ini sendiri lebih ditekankan kepada apa yang diperbuat sistem 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 titik awal 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 bentuk penggunaan *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 objek dan pesan-pesan yang diletakkan di antaranya di dalam sebuah *use case*. Komponen utamanya adalah objek 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 memberikan gambaran 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 objek 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.

2.16 *Pittsburg Sleep Quality Index*

Pittsburg Sleep Quality Index (PSQI) dibuat oleh Dr. Daniel J. Buysee dan teman satu pekerjaan lainnya di University of Pittsburgh's Western Psychiatric Institute and Clinic pada tahun 1980-an. PSQI dibuat setelah melakukan observasi bahwa kebanyakan pasien orang dewasa dari usia 19 – 83 tahun memiliki gangguan tidur dengan mengisi kuesioner. Kuesioner memiliki sembilan belas pertanyaan yang masing-masing item memiliki skor dan dikelompokkan menjadi 7 kategori skor penilaian. Tujuh kategori penilaian tersebut di antaranya, subjektif kualitas tidur, latensi tidur, durasi tidur, efisiensi kebiasaan tidur, gangguan tidur, penggunaan obat tidur, dan disfungsi siang. Setiap kategori skor komponen penilaian akan memilliki nilai dari rentang 0 sampai 3 di akhir penghitungan yang berarti semakin kecil atau 0 semakin bagus dan semakin besar atau 3 maka semakin jelek [38].

2.17 Spotify SDK Android

Spotify merupakan salah satu aplikasi layanan musik yang menyediakan API untuk dapat dimanfaatkan oleh pengembangnya. Terdapat Spotify Web API, Spotify iOS SDK, dan juga terdapat Spotify Android SDK. SDK untuk Android ini masih dalam versi beta namun sudah dapat dimanfaatkan untuk menambahkan audio streaming, authentification pengguna dan fitur Spotify lainnya [39].

2.18 Android Studio

Android Studio merupakan lingkungan pengembangan terpadu yang digunakan untuk membuat atau pengembangan sebuah aplikasi berbasiskan Android. Android Studio menawarkan fitur yang lebih banyak untuk meningkatkan produktivitas saat membuat aplikasi Android [40], seperti:

a. Sistem pembuatan berbasis Gradle yang fleksibel.
b. Emulator yang cepat dan kaya fitur.
c. Lingkungan yang menyatu untuk pengembangan semua perangkat Android
d. Instant Run untuk mendorong perubahan ke aplikasi yang berjalan tanpa membuat APK baru.
e. Dukungan bawaan untuk Google Cloud Platform, mempermudah pengintegrasian Google Cloud Messaging dan App Engine.
f. Dukungan C++ dan NDK.
g. Alat penguji dan perangkat kerja yang ekstensif.
h. Alat Lint untuk meningkatkan kinerja, kegunaan, kompatibilitas versi, dan masalah-masalah lain.

2.19 *Android Jetpack*

Jetpack adalah serangkaian *library, tools,* dan panduan untuk membantu developer membangun aplikasi berkualitas tinggi dengan lebih mudah. Komponen-komponen ini membantu developer *best pratices*, membebaskan developer dari kode boilerplate, dan menyederhanakan tugas-tugas kompleks, sehingga developer dapat fokus pada kode yang dibangung selain dari *framework* android [41].

Jetpack terdiri dari beberapa paket library androidx. Semua Paket library tidak diikat oleh platform API. Dengan demikian hal tersebut menawarkan kompatibilitas *backward* dan diperbarui lebih sering daripada platform Android.