BAB 2
LANDASAN TEORI

2.1 *Software Reengineering*

Software Reengineering merupakan proses memodifikasi dan mengorganisasi ulang software yang sudah ada agar dapat lebih mudah dipelihara (*maintain*) [6]. Proses tersebut adalah menulis ulang atau restrukturisasi seluruh sistem tanpa mengubah fungsionalitas yang ada. Tujuan utama dari proses rekayasa ulang adalah untuk memodifikasi atau memodernisasi sistem sebelumnya ke versi yang lebih baru [6].

2.1.1 Taksonomi *Sofware Reengineering*

Berikut ini adalah sub bagian dari taksonomi Software Reengineering:

1. ***Reverse Engineering***

   Reverse Engineering adalah proses menganalisis sistem subjek untuk mengidentifikasi keterkaitan komponen sistem dan membuat representasi dari sistem dalam bentuk lain atau pada level abstraksi yang lebih tinggi [3].

2. ***Forward Engineering***

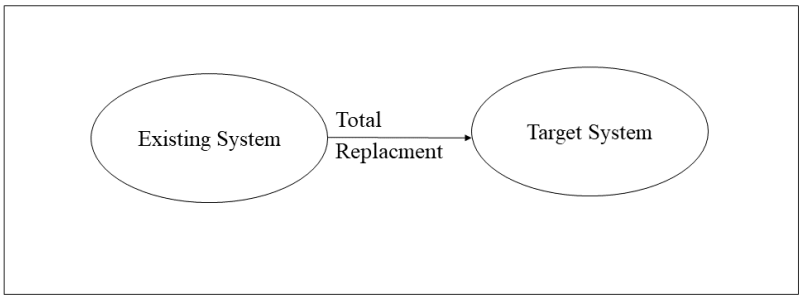
   Forward Engineering adalah kebalikan dari *Reverse Engineering* dengan kata lain *forward engineering* adalah cara tradisional pengembangan perangkat lunak dimana informasi diuraikan, dijelaskan secara lebih rinci sambil beralih dari konseptual desain ke detail desain, hingga implantasi fisik [3].

2.1.2 Pendekatan *Software Reengineering*

1. **Pendekatan *Big Bang***

Pendekatan ini biasa dikenal sebagai “Lump Sum”, dimana di satu waktu keseluruhan sistem diganti. Pendekatan ini digunakan ketika sistem membutuhkan pemecahan masalah sesegara mungkin. Misalnya ketika sistem ingin melakukan migrasi sistem dengan arsitektur yang berbeda. Jika disimpulkan maka kelebihan dari menggunakan pendekatan ini adalah memungkinkan sistem untuk berjalan di

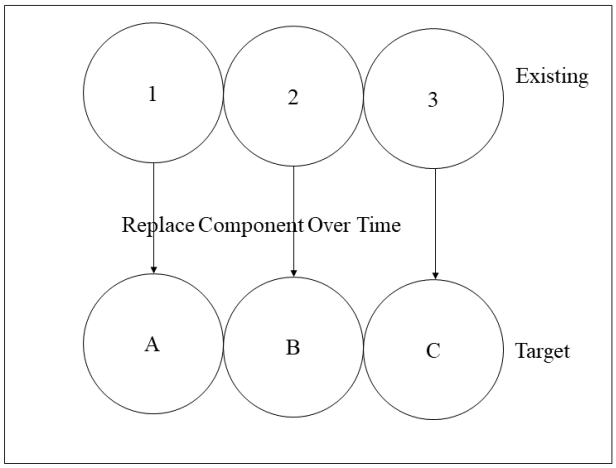
lingkungan yang baru tanpa memerukan integrasi antarmuka. Kekurangan dari pendekatan ini adalah tidak cocok untuk proyek yang besar karena menghabiskan banyak waktu dan sumberdaya sebelum terget sistem ada [3].



Gambar 2.1 Pendekatan *Big Bang*

2. Pendekatan *Incremental*

Pendekatan ini dikenal sebagai "Penghapusan bertahap" atau "Aditif". Dalam pendekatan ini, setiap bagian dari sistem direkayasa ulang dan ditambahkan dalam bentuk versi yang baru agar kebutuhan utama terpenuhi. Pendekatan ini memiliki kelebihan di mana bagian-bagian dari sistem diproduksi lebih cepat dengan mudah dalam pencarian error. Oleh karena itu pendekatan ini lebih rendah risikonya jika dibandingkan dengan pendekatan *big bang.* Adapun kekurangan dari pendekatan ini adalah penyampaian sistem secara lengkap akan membutuhkan waktu yang cukup lama [3].

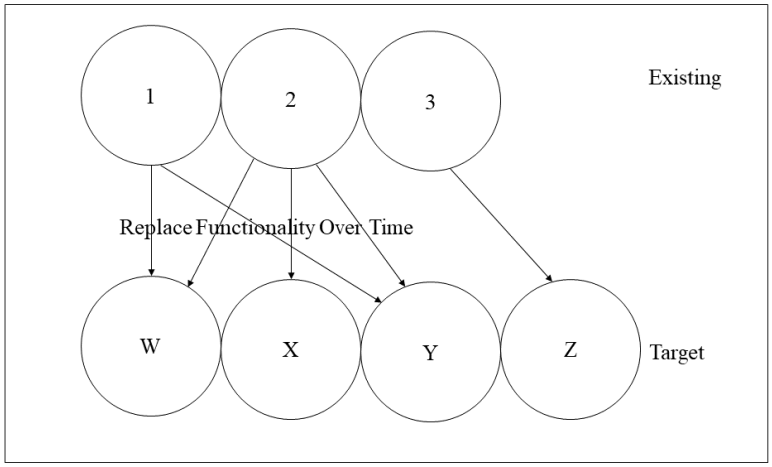


Gambar 2.2 Pendekatan *Incremental*

3. Pendekatan *Evolutaionary*

Pendekatan ini mirip dengan pendekatan *incremental* dimana setiap komponen dari sistem lama direkayasa ulang dan diganti dengan sistem yang baru. Penggantian dilakukan berdasarkan fungsionalitas bukan pada strukur sistem yang lama. Pengembang hanya fokus pada pembuat komponen yang kohesif. Kelebihan pendekatan ini adalah komponen sistemnya digambarkan sangat kohesif. Oleh karena itu, pendekatan ini cocok untuk mengubah sistem saat ini menjadi berorentasi objek. Beberapa masalah antarmuka dan waktu respons mungkin

terjadi. Kekurangan dari pendekatan ini adalah fungsi yang serupa harus disempurnakan sebagai unit fungsional tunggal dalam sistem baru [3].



Gambar 2.3 Pendekatan *Evulationary*

2.2 *Enhanced Reengineering*

Enhanced Reengineering merupakan 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 kebutuhan sistem. Setelah itu pada tahap kedua adalah pemetaan restrukturisasi spesifikasi persyaratan perangkat lunak untuk penyelesaian desain dokumen, hasil pada tahap ini adalah dokumen yang sudah didesain ulang [3].

Pada tahap ketiga, bagian pemrograman 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 performa dari sistem yang sudah ada dengan sistem 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 ini merupakan tahapan *Enhancend Reengineering*:

1. **Studi Kelayakan dan Kebutuhan**

   Pada tahap ini studi kelayakan dilakukan oleh peneliti untuk memeriksa konfigurasi dan kompatibilitas sistem komputer. Setelah menyelesaikan studi kelayakan, kebutuhan sistem ditentukan ulang berdasarkan keinginan pengguna. SRS *(Software Requirements Spesification)* merupakan dokumen resmi yang memiliki semua persyaratan dalam tulisan terstruktur. Pada tahap ini untuk menentukan kembali pesyaratan sistem, maka sistem perlu dipetakan menggunakan SRS [6].

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 [6].

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 direengineering dengan teknik yang baru [6].

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 [6].

5. **Implementasi**

   Tahap ini adalah tahap terakhir dari mekanisme tahap reengineering. Pada tahap implementasi, bagian – bagian tertentu diganti berdasarkan empat tahap sebelumnya [6].

2.3 *Maintainability*

Menurut Rudolp Frederick Stapelberg *maintainability* merupakan probabilitas terhadap item yang gagal akan dikembalikan ke kondisi efektif operasional dalam waktu tertentu. Sehingga dapat disimpulkan juga bahwa *maintainability* adalah kemampuan dalam melakukan perawatan terhadap suatu sistem dengan rentang waktu yang ditentukan ketika sistem tersebut memiliki suatu masalah [7].

2.4 *Maintainability Index*

Maintainability Index (MI) merupakan *software metric* yang megukur sebuah perangkat lunak mudah atau sulit untuk dilakukan perawatan atau perubahan di masa yang akan datang. *Maintainability Index* dihitung berdasarkan nilai formula dari *Halstead's Volume* (HV), *Cyclomatic Complexity (CC)* dan *Lines of Code (LOC)* [8]. Semakin tinggi nilai dari kalkulasi *maintainability index* mengindikasikan bahwa kode program memiliki tingkat *maintainability* yang baik, dimana hal ini akan membuat *source code* lebih mudah dipahami dan dirawat sehingga akan lebih mudah dalam pencarian dan perbaikan kecacatan *(bugs).* Berikut ini adalah formula dari *Maintainability Index (MI) :*

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

Keterangan:

HV = Halstead Metrics Volumes

CC = Cyclomatic Complexity

LOC = Lines of Code

PerCM = Percent line of comment

Tabel 2.1 Klasifikasi Maintainability Index

Nilai *Maintainability Index*	Klasifikasi
MI > 85	Highly Maintainable
65 < MI ≤ 85	Moderately Maintainable
MI ≤ 65	Difficult to Maintain

2.5 *Halstead Metrics*

Halstead Metric merupakan pengukuran yang dikembangkan untuk mengukur kompleksitas dalam suatu program langsung dari *source code*. Pengukuran dilakukan dengan menentukan ukuran kuantitatif kompleksitas dari operator dan operand dalam suatu modul sistem [8]. Pada *Halstead's metric* terdapat enam jenis komponen yaitu :

1. *Length of program*

   Length of program adalah kalkulasi jumlah total operator dan operand yang muncul. Persamaan *Length of the program* dirumuskan pada persamaan di bawah ini:

   $$N = N1 + N2$$

   Keterangan:

   N1 = Jumlah Operator

   N2 = Jumlah Operand

2. *Vocabulary of the program*

   Vocabulary of the program adalah kalkulasi jumlah operator unik dan operand unik yang muncul dalam program. Persamaan *Vocabulary of the program* dirumuskan pada persamaan di bawah ini:

$$n = n1 + n2$$

Keterangan:

n = *Vocabulary of program*

n1 = Jumlah Operator Unik

n2 = Jumlah Operand Unik

3. *Volume of the program*

   Volume of the program adalah volume dalam *halstead metric* yang digunakan untuk mengetahui volume program. Persamaan *volume of the program* dirumuskan pada persamaan di bawah ini:

$$V = N \times log2\, n$$

   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* digunakan untuk mengetahui kesulitan dan pengembangan program. Persamaan *Difficulty* dirumuskan pada persamaan di bawah ini:

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

   Keterangan:

   D = *Difficulty*

   N2 = Total semua operan yang muncul

   n1 = Jumlah operator unik

   n2 = Jumlah operan unik

5. *Effort*

   Effort dalam *Halstead metric* digunakan untuk mengetahui sumber daya yang digunakan untuk pengembangan program. Persamaan *Effort* dirumuskan pada persamaan di bawah ini :

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

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* digunakan untuk mengetahui prediksi *bug* pada program. Persamaan *Number of bugs expected in the program* dirumuskan pada persamaan di bawah ini :

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

Keterangan:

B = *Number of bugs expected in the program*

V = Dari kalkulasi *Volume of the program*

2.6 *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 [9]. 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.

2.7 *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 [10].

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 [10].

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*

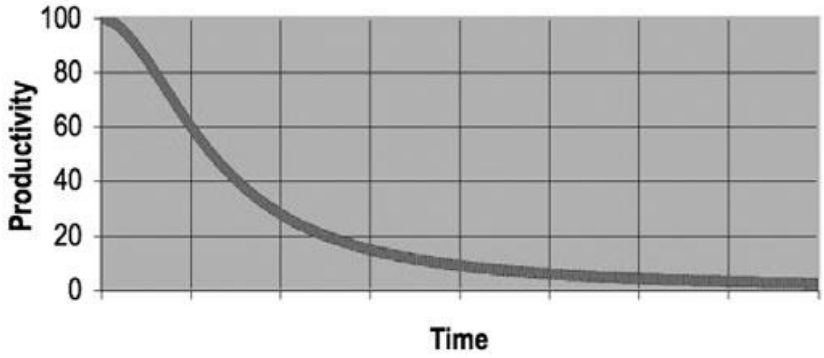
3. Behavioral Design Patterns

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*

2.8 *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 2.4** di mana waktu dapat mempengaruhi tingkat produktivitas [11]. 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 2.4 ***Productivity vs Time***

2.8.1 *Meaningful Name*

Meaningful Names merupakan aturan aturan untuk melakukan penamaan terhadap kelas,variabel dan fungsi agar mudah untuk dipahami oleh developer lain. Berikut adalah beberapa aturan dalam *meaningfulnames.*

1. ***Use Intention-Revealing Names***

   Use Intention-Revealing Names maksudnya adalah penamaan dalam suatu variable, fungsi, kelas, itu harus mengungkapkan suatu tujuan, apabila nama tersebut masih membutuhkan komentar maka nama tersebut tidak mempresentasikan makna atau tujuan dari nama tersebut.

2. ***Avoid Disinformation***

   Avoid Disinformation maksudnya adalah hindari penggunakan nama sebuah platform atau varian unix. Hindari juga penamaan yang bervariaso dalam hal kecil dan juga penggunaan huruf L kecil dan O besar sebagai nama variabel, terutama dalam kombinasi.

3. ***Use Pronounceable Names***

   Use Pronounceable Names maksudnya adalah menggunakan nama yang dapat diucapkan oleh manusia, sehingga manusia dapat mengerti maksud dari nama tersebut.

4. ***Use Searchable Name***

   Use Searchable Name maksudnya adalah untuk menggunakan nama yang mudah dicari. Nama huruf tunggal dan konstanta numerik memiliki masalah tertentu karena tidak mudah untuk ditemukan di seluruh badan teks,

penggunaan huruf e untuk variable juga pilihan yang buruk, karena e adalah kata yang paling umum dalam bahasa inggris. Pastikan bahwa penamaan bukan nama yang sangat berguna, tapi setidaknya dapat dicari.

5. ***Avoid Mental Mapping***

   Avoid Mental Mapping maksudnya adalah masalah dalam penamaan variable satu huruf. Dalam looping biasanya menggunakan huruf i, j atau k. apabila skalanya kecil hal tersebut tidak jadi masalah. Dalam hal lain penamaan satu huruf adalah pilihan yang sangat buruk.

6. ***Class Names***

   Class Names maksudnya adalah nama kelas dan objek harus memiliki nama frasa kata benda atau kata benda seperti *Customer*, *WikiPage* dan *Account*. Hindari kata-kata seperti *Manager, Processor, Data* atau *Info* untuk penamaan sebuah kelas. Penamaan kelas tidak boleh menggunakan kata kerja dan huruf pertama kelas harus huruf capital.

7. ***Method Names***

   Method Names maksudnya adalah nama metode harus memiliki nama dari kata kerja atau frasa kata kerja seperti *postpayment, deletePage*.

8. ***Don't be Cute***

   Don't be Cute maksudnya adalah jangan menggunakan kata-kata yang menurut kita lucu, tetapi gunakan kata apa yang kita maksud, apa yang kita maksud yaitu apa yang kita katakan.

9. ***Don't Pun***

   Don't Pun maksudnya adalah kita sebagai *developer* harus menggunakan kata yang mempunyai tujuan yang sama tetapi memiliki sintaks yang berbeda. Contohnya adalah penggunaan kata *add* yang sudah terlalu banyak dalam sebuah *class,* maka kita dapat menggunakan sintaks lain yang memiliki arti yang sama seperti *insert* atau *append*, yang dimana penulisannya sangat berbeda tetapi secara arti mempunyai tujuan yang sama dengan *add*.

2.8.2 *Clean Function*

Pada konsep ini terdapat petunjuk dalam menulis method/fungsi yang bersih agar lebih mudah untuk dipahami. Berikut adalah beberapa aturan dalam *clean fuction*:

1. ***Small***

   Small maksudnya adalah penamaan fungsi, pernyataan dalam fungsi dan baris yang dimana sebisa mungkin di dalam suatu fungsi itu harus singkat dan jelas. contohnya itu baris pernyataan sebisa mungkin tidak lebih 20 sampai 30 dan akan lebih baik apabila hanya 3 sampai dengan 4 baris saja.

2. ***Do One Thing***

   Do One Thing maksudnya adalah fungsi hanya melakukan satu hal, tidak melakukan lebih dari satu hal, kecuali fungsi tersebut tidak dapat di pecah menjadi beberapa bagian karena saling berhubungan.

3. ***One Level of Abstraction per Function***

   One Level of Abstraction per *function* maksudnya adalah untuk memastikan fungsi kita melakukan "satu hal", kita perlu memastikan bahwa pernyataan dalam fungsi kita berada pada tingkat abstraksi yang sama. Pembacaan code fungsi dibaca dari atas kebawah.

4. ***Use Descriptive Names***

   Use Descriptive Names maksudnya adalah memilih nama yang baik untuk fungsi kecil yang melakukan satu hal. Semakin kecil dan focus suatu fungsi, semakin mudah untuk memilih nama yang deskriptif. Nama yang panjang lebih baik daripada nama yang pendek tetapi membingungkan.

5. ***Function Arguments***

   Function Arguments maksudnya adalah argument yang ideal untuk suatu fungsi adalah nol, apabila tidak bisa maka minimal satu argument dan maksimal dua argument. Apabila sampai harus tiga argument, maka harus dihindari jika memungkinkan. Lebih dari tiga argument lebih baik untuktidak digunakan. Ketika membaca sebuah *code* yang diceritakan dalam suatu modul, "includeSetupPage ()" lebih mudah dipahami daripada "includeSetupPageInfo(newPageContent)".

6. ***Command Query Separation***

   Command Query Separation maksudnya adalah fungsi harus melakukan sesuatu atau menjawab sesuatu, tidak bisa keduanya. Melakukan dua pekerjaan sering menyebabkan kebingungan dalam pembacaan code.

7. ***Don't Repeat Yourself***

   Don't Repeat Yourself maksudnya adalah jangan melakukan duplikasi fungsi, hal tersebut dapat menyebabkan dalam melakukan kelalaian dalam melakukan perawatan, karena developer membaca fungsi yang mirip sehingga sulit mana yang harus diperbaiki.

2.8.3 *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 adalah beberapa aturan dalam *clean comment:*

1. ***Good Comment***

   Dalam *good comment*, komentar diperlukan atau mempunya manfaat, tetapi satu-satunya komentar yang benar-benar baik adalah komentar yang kita tidak harus tulis. Apabila kita membutuhkan komentar, ada beberapa komentar yang dapat digunakan yaitu :

   a. Legal Comment yaitu komentar yang terpaksa harus ditulis karena alasan hokum seperti komentar *copyright*.

   b. Informative Comments yaitu komentar yang berguna untuk menyediakan informasi.

   c. Explanation of Intent yaitu komentar yang melampaui informasi bermanfaat tentang implementasi code. Ketika kita membandingkan dua objek, penulis memutuskan bahwa dia ingin mengurutkan objek kelasnya lebih tinggi dari objek yang lain.

d. Clarification yaitu komentar yang digunakan untuk membantu menerjemahkan arti dari beberapa argument yang tidak jelas.

e. Warning of Consequences yaitu komentar yang berguna untuk memperingati developer lain tentang konsekuensi apabila kode tersebut berubah.

2. *Bad Comment*

Sebagian besar dalam suatu aplikasi komentar termasuk dalam bad comment. Komentar yang diberikan terkadang seharusnya tidak usah diberikan, tetapi developer memasukan komentar terhadap suatu code yang berarti code tersebut tidak ekspresif. Beberapa komentar yang selalu diberikan oleh developer antara lain :

a. Numbling yaitu komentar yang hanya karena kita harus melakukannya atau karena proses mengharuskannya. Kita harus memastikan bahwa komentar yang kita berikan adalah komentar yang baik atau tidak.

b. Redudant Comments yaitu komentar yang diberikan secara berulang. Hal ini dapat membuat program saat dieksekusi akan terasa sangat lambat.

c. Journal Comments yaitu komentar yang ditambahkan oleh developer ke awal modul setiap kali developer tersebut melakukan perubahan kode. Sehingga komentar-komentar ini terakumulasi sebagai semacam jurnal atau log dari setiap perubahan yang pernah dilakukan.

d. Noise Comments yaitu komentar yang menyatakan kembali kode yang sudah jelas tanpa memberikan informasi yang baru.

2.8.4 *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.

2.8.5 *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.
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.
3. ***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.

2.8.6 *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*:

2.9 *Model View Controller (MVC)*

Model-View-Controller (MVC) adalah pola arsitektur yang memisahkan aplikasi menjadi tiga bagian: *model, view,* dan *controller*. Masing-masing bagian ini dibangun untuk menangani aspek yang spesifik dari suatu aplikasi. MVC adalah salah satu kerangka kerja pengembangan web standar industri yang paling sering digunakan untuk membuat proyek yang dapat diskalakan dan diperluas [12]. Penjelasannya sebagai berikut:

1. ***Model***

   Komponen Model biasanya bertugas untuk mengatur atau memanipulasi data dari database berdasarkan perintah dari *controller*.

2. ***View***

   Komponen View bertugas menampilkan informasi yang mudah dimengerti kepada pengguna sesuai denga perintah dari controller

3. ***Controller***

   Komponen Controller bertugas untuk mengatur apa yang akan dilakukan oleh model, dan menentukan view atau tampilan mana yang akan ditampilkan ke pengguna.