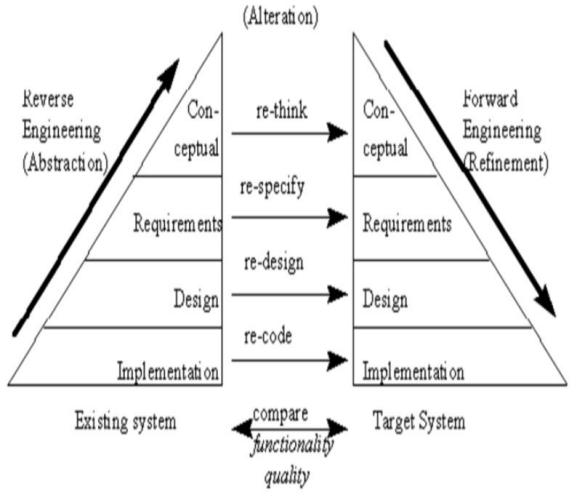
BAB 2
LANDASAN TEORI

2.1 *Reengineering*

Dalam pembangunan ulang perangkat lunak ada konsep yang bisa digunakan, konsep ini terdiri dari beberapa tahapan yang diperlukan untuk membangun ulang suatu perangkat lunak.



Gambar 2-1 - Proses Reengineering[5]

Reengineering merupakan sebuah proses yang melakukan pembangunan ulang terhadap perangkat lunak yang sudah ada atau yang sedang berjalan menjadi suatu perangkat lunak baru dengan adanya peningkatan kualitas. Kunci utama dari *reengineering* adalah meningkatkan atau mengubah perangkat lunak yang ada saat ini sehingga lebih mudah dipahami dalam bentuk perangkat lunak yang baru [5].

2.1.1 Proses *reengineering*

Pada proses *reengineering* ada 2 bagian besar didalamnya, yaitu *forward engineering* dan *reverse engineering.*

2.1.1.1 *Reverse Engineering*

Reverse Engineering merupakan suatu proses menganalisis sistem yang berjalan saat ini untuk mengidentifikasi komponen sistem dan menciptakan

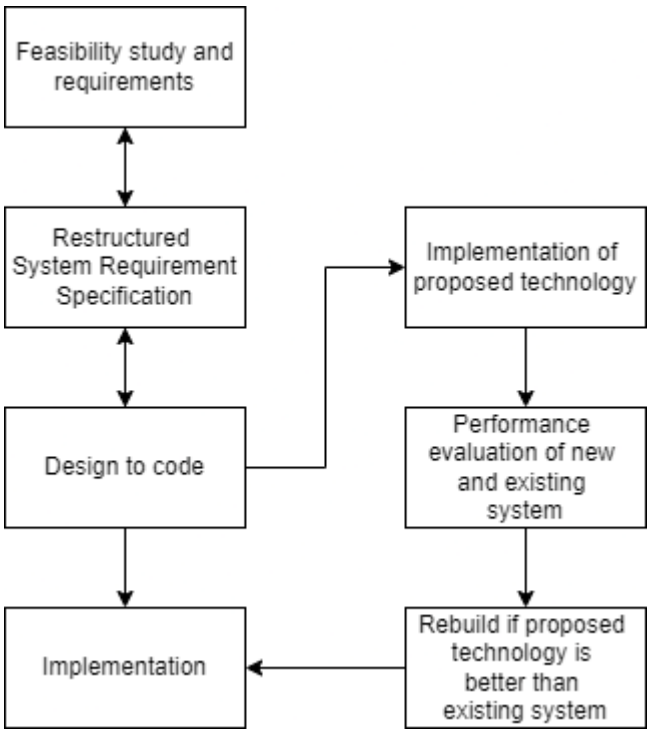
representasi sistem dalam bentuk lain atau pada tingkat abstraksi yang berbeda. Hal tersebut menjelaskan bahwa *reverse engineering* tidak menghasilkan informasi baru terhadap sistem melainkan mendapatkan kembali informasi yang hilang atau belum ditentukan [5].

2.1.1.2 *Forward Engineering*

Proses *forward engineering* bergerak dari atas kebawah pada suatu level abstraksi. Proses ini merupakan proses yang berlawanan dengan proses *reverse engineering* yang bergerak dari bawah keatas pada suatu level abstraksi. *Forward engineering* merupakan proses pengembangan perangkat lunak tradisional yang dimana seluruh informasi kebutuhannya telah diuraikan dan dijelaskan secara rinci sebelumnya. Sehingga pembangunan berjalan lurus kebawah berdasarkan kebutuhan yang telah ditentukan [5].

2.1.2 Enhanced Reengineering

Pada *reengineering* ada perkembangan dalam tahapan – tahapan yang dilakukan. Salah satunya perkembangannya dinamakan *enhanced reengineering*.



Gambar 2-2 - Mekanisme proses enhanced reengineering [5]

Enhanced Reengineering merupakan proses *reengineering* yang memanfaatkan beberapa metode dan level abstraksi untuk mengubah perangkat lunak yang sedang berjalan atau yang ada saat ini menjadi perangkat lunak yang baru. Proses ini memanfaatkan *reverse engineering* dan *forward engineering* [5], [7].

2.1.2.1 Feasibility study dan *requirements*

Tahapan pertama merupakan *feasibility study* dan *requirements*. Pada tahapan ini dilakukan pengecekan terhadap konfigurasi dan kompatibilitas sistem komputer. Setelah tahapan *feasibility study* selesai dilakukan kebutuhan sistem akan ditentukan kembali berdasarkan keinginan pengguna. Dokumen resmi yang berisi semua persyaratan akan tertulis pada Software Requirement Specification (SRS) untuk menentukan ulang persyaratan sistem [5].

2.1.2.2 *Restructured system requirements specification*

Tahapan ini melakukan restrukturisasi terhadap SRS yang telah dibangun sebelumnya. Dokumentasi merupakan atribut penting dalam proses pengembangan perangkat lunak karena berfungsi sebagai perencana untuk produk akhir. Pada tahapan ini juga dilakukan perbandingan antara SRS yang sudah ada dengan SRS yang baru. SRS yang baru akan diintegrasikan dengan SRS yang sudah ada [5].

2.1.2.3 *Design to Code*

Tahapan ini akan membahas mengenai rincian tentang *design to code*. Pada tahapan ini akan dilakukan penyesuaian kode berdasarkan *re-design document* yang telah ditentukan sebelumnya. Algoritma dan fungsionalitas yang sudah ada harus ditulis kembali. Hal tersebut diperlukan jika algoritma dan fungsionalitas yang ada saat itu sudah tidak efisien dalam segi waktu dan tidak lagi akurat sehingga diperlukan algoritma baru. Maka dari itu perlu direncanakan untuk direkayasa ulang dengan teknik baru [5].

2.1.2.4 *Comparison of existing* dan *proposed functionalities*

Tahapan ini memberikan rincian tentang proses pengujian ulang. Pengujian ulang dilakukan dengan membandingkan antara perangkat lunak yang lama dengan perangkat lunak yang baru dari sisi kinerja fungsionalitas [5].

2.1.2.5 Implementasi

Tahap ini merupakan tahap terakhir dari *Enhanced Reengineering*. Berdasarkan hasil dari tahapan sebelumnya

2.2 *Maintainability*

Maintainability merupakan sisi kualitas yang menggambarkan kemampuan sistem untuk mengalami perubahan dengan tingkat kemudahan. Perubahan tersebut dapat mencakup komponen, fitur dan antarmuka saat mengubah fungsionalitas atau memperbaiki kesalahan. Sisi kualitas ini merupakan salah satu sisi kualitas yang utama dikarenakan jika suatu perangkat lunak memiliki tingkat *maintainability* yang tinggi dapat menurunkan pengeluaran biaya. Selain itu, perangkat lunak yang memiliki tingkat *maintainability* yang tinggi juga dapat dirawat dengan mudah sehingga waktu dan upaya ketika memperbaiki suatu kesalahan dapat dilakukan seminimum mungkin [6], [8].

2.2.1 *Maintainability Index*

Maintainability Index merupakan *software metric* yang digunakan untuk mengukur suatu perangkat lunak apakah perangkat lunak ini akan mudah atau sulit untuk mengalami perubahan dan perawatan di masa yang akan datang. *Software metric* ini menghitung berdasarkan *Lines of Code (*LOC), *Cyclomatic Complexity* (CC) dan *Halstead Volume* (HV). Persamaan dari *Maintainability Index* dapat dilihat pada Gambar 2-3 [9][10].

$$MI = MAX(0, (171 - 5.2\times\ln(HV) - 0.23\times CC - 16.2\times\ln(LOC))*100/171)$$

Gambar 2-3 - Persamaan Maintainability Index [10]

Dengan keterangan:

a. HV : Halstead Metrics Volumes
b. CC : Cyclomatic Complexity
c. LOC : Lines of Code
d. perCM : Percent line of comment

Lalu hasil dari perhitungan *Maintainability Index* diklasifikasikan menjadi beberapa bagian yang dapat dilihat pada Tabel 2-1 [10].

Nilai *Maintainability Index*	Klasifikasi
MI > 19	Highly Maintainable
9 < MI ≤ 19	Moderately Maintainable
MI ≤ 9	Difficult to Maintain

Tabel 2-1 Klasifikasi Nilai Maintainability Index

2.3 *Halstead Metrics*

Halstead Metrics merupakan suatu pengukuran yang digunakan untuk menghitung kompleksitas modul dari suatu program. Pengukuran ini dilakukan dengan cara menentukan ukuran kuantitatif kompleksitas dari operator dan operand. Pada pengukuran ini terdapat 6 jenis yaitu *Length of Program, Vocabulary of The Program, Volume of The Program, Difficulty, Effort* dan *Number of Bugs Expected in The Program* [9].

2.3.1 *Length of Program*

Length of Progam merupakan kalkulasi atau perhitungan mengenai jumlah total operator dan operand yang muncul. Persamaannya dapat dilihat pada gambar 2-4 [9].

$$N = N1 + N2$$

Gambar 2-4 - Perhitungan Length of Program [9]

Dengan keterangan:

a. N : *Length of Program.*
b. N1 : Semua operator yang muncul.

c. N2 : Semua operand yang muncul.

2.3.2 *Vocabulary of The Program*

Vocabulary of The Program merupakan perhitungan mengenai jumlah operator dan operand yang unik dan muncul pada program. Persamaan dari perhitungan ini dapat dilihat pada gambar 2-5 [9].

$$n = n1 + n2$$

Gambar 2-5 - Persamaan Vocabulary of The Program [9]

Dengan keterangan:

a. n : *Vocabulary of The Program.*
b. n1 : Jumlah operator unik.
c. n2 : Jumlah operand unik.

2.3.3 *Volume of The Program*

Volume of The Program merupakan perhitungan mengenai volume progam. Persamaan dari perhitungan ini dapat dilihat pada gambar 2-6 [9].

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

Gambar 2-6 - Persamaan Volume of The Program [9]

Dengan keterangan:

a. V : *Volume of The Program*.
b. N : Nilai kalkulasi dari *Length of The Program.*
c. n : Nilai kalkulasi dari *Vocabulary of The Program*.

2.3.4 *Difficulty*

Difficulty merupakan perhitungan yang digunakan untuk mengetahui kesulitan dan pengembangan program. Persamaan dari perhitungan ini dapat dilihat pada gambar 2-7 [9].

$$D = \frac{n1}{2} \times \frac{N2}{n2}$$

Gambar 2-7 - Perhitungan Difficulty [9]

Dengan keterangan:

a. D : *Difficulty.*
b. n1 : Jumlah operator unik.
c. n2 : Jumlah operand unik.
d. N2 : Jumlah semua operand yang muncul.

2.3.5 *Effort*

Effort merupakan perhitungan yang digunakan untuk mengetahui sumber daya yang digunakan untuk mengembangkan suatu program. Persamaan dari perhitungan ini dapat dilihat pada gambar 2-8 9 [7].

$$E = D \times V$$

Gambar 2-8 – Persamaa Effort [9]

Dengan keterangan:

a. E : *Effort.*
b. D : Nilai dari perhitungan *Difficulty*.
c. V : Nilai dari perhitungan *Volume of The Program.*

2.3.6 *Number of Bugs Expected in The Program*

Number of Bugs Expected in The Program merupakan perhitungan yang digunakan untuk mengetahui prediksi dari bug yang akan terjadi pada program ini. Persamaan dari perhitungan ini dapat dilihat pada gambar 2-9 [9].

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

Gambar 2-9 - Perhitungan Number of Bugs Expected in The Program

Dengan keterangan:

a. B : *Number of Bugs Expected in The Program*.
b. V : Nilai dari perhitungan *Volume of The Program.*

2.4 *Cyclomatic Complexity*

Cyclomatic Complexity merupakan suatu metric yang menghitung *control flow* dari suatu modul. Jika modul tersebut memiliki kompleksitas yang tinggi maka modul tersebut akan sulit untuk diuji dan dijaga [9]. Untuk mengetahui hasil

Cyclomatic Complexity dari suatu modul dapa menggunakan persamaan yang dapat dilihat pada gambar 2-10 [9].

$$V(g) = e - n + 2$$

Gambar 2-10 - Persamaan Cyclomatic Complexity [9]

Dengan keterangan:

a. V(g) : *Cyclomatic Complexity*.
b. e : Jumlah dari *Edge*.
c. n : Jumlah dari *Node*.

2.5 TypeScript

TypeScript hadir menggantikan kesuksesan JavaScript yang tetap dianggap buruk untuk mengembangkan dan memelihara aplikasi yang besar. TypeScript merupakan *extension* dari Bahasa Pemrograman JavaScript yang dibangun untuk menutupi kekurangan JavaScript. Secara sintaks, TypeScript merupakan *superset* dari ECMAScript 5 sehingga bisa dibilang setiap program JavaScript merupakan program TypeScript juga karena pada akhirnya TypeScript akan di *compile* menjadi JavaScript. TypeScript memperkaya JavaScript dengan menambahkan *module system*, *interfaces* dan *static type system.* Selain itu, TypeScript juga didukung oleh *tooling* dan *IDE Experiences* yang sebelumnya hanya bisa dirasakan oleh beberapa bahasa pemrogaman seperti C dan Java. Hal ini ditujukan untuk membantu para *developer* dalam membangun aplikasi yang besar dengan mudah [11].

2.6 *Desktop Web Apps*

Pendekatan *desktop web apps* merupakan pendekatan yang memungkinkan mengemas sebuah *website* lalu mendistribusikannya ke berbagai *platform*. Sehingga 1 aplikasi atau perangkat lunak yang dibangun dapat digunakan pada berbagai macam *platform* yang berbeda. Selain itu, pendekatan ini juga dapat membuat sebuah *website* mengakses *native application programming interface* (API) atau API dari sebuah sistem operasi layaknya aplikasi berbasis *desktop* seperti *desktop notifications* dan *system tray*. Hal tersebut merupakan suatu kelebihan dari pendekatan ini karena hal tersebut tidak dapat dilakukan oleh *website*

biasa. Namun, karena aplikasi dieksekusi diatas *browser* pendekatan ini berpotensi untuk menimbulkan kinerja yang cukup besar pada perangkat yang digunakan [12].

Pendekatan *desktop web apps* hadir untuk menyelesaikan beberapa masalah yang sedang terjadi seperti [12]:

1. *Developer* sering kesulitan dalam membuat suatu aplikasi dan menerapkannya ke berbagai *platform*.
2. Menggunakan kembali *library* dan *development tools* dalam konteks *desktop applications* sering kali merepotkan.

Saat ini ada banyak *desktop web apps frameworks* yang bisa digunakan. Salah satunya yang populer, masih dipelihara dan dikembangkan adalah Electron. *Framework* ini merupakan sebuah *framework* yang kembangkan oleh GitHub dengan tujuan untuk membangun *cross-platform desktop apps* menggunakan HTML, CSS dan JavaScript. Electron bekerja dengan mengkombinasikan antara Chromium dan Node.js menjadi satu *runtime* [12].

2.7 *Clean Code*

Clean Code adalah suatu konsep atau petunjuk dalam menuliskan struktur kode dengan bersih sehingga kode tersebut dapat dibaca oleh *developer* yang lain dengan mudah. Konsep ini bertujuan untuk mengatasi permasalahan penurunan produktivitas yang disebabkan oleh struktur kode yang berantakan. Konsep ini membahas beberapa hal diantaranya adalah:

1. *Meaningful Names.*
2. *Clean Functions.*
3. *Clean Comment.*
4. *Clean Error Handling.*

2.7.1 *Meaningful Names*

Bagian ini menjelaskan bagaimana cara untuk memberikan nama yang baik pada suatu variabel, fungsi dan metode. Tujuannya dari *meaningful names* ini agar

hal – hal tersebut dapat dibaca dan dipahami dengan baik. Dalam memberikan nama ada beberapa hal yang perlu diperhatikan yaitu:

1. Gunakan nama yang mempunyai arti.
2. Nama yang mudah dicari.
3. Kata kerja untuk fungsi.
4. Hindari penggunaan kata yang tidak diperlukan.

2.7.2 *Clean Function*

Bagian ini menjelaskan tentang bagaimana cara menulis fungsi yang baik agar lebih mudah dipahami dan dibaca oleh para *developer* yang lain. Dalam menuliskan *clean function* ada beberapa hal yang perlu diperhatikan yaitu:

1. Hindari penulisan fungsi yang panjang.
2. Hindari penggunaan flag sebagai parameten.

2.7.3 *Clean Comment*

Bagian ini menjelaskan tentang bagaimana cara menulis komentar pada *source code* dengan baik dan efisien. Dalam menuliskan *clean comment* ada beberapa hal yang perlu diperhatikan yaitu:

1. Gunakan komentar untuk hal yang meliki kompleksitas logika.
2. Komentar klarifikasi yang informatif.

2.7.4 *Clean Formatting*

Bagian ini menjelaskan tentang bagaimana cara menulis format *source code* dengan baik seperti penggunaan *space, tab, identation* dan penempatan kode sesuai kegunaannya. Dalam menuliskan *clean formatting* ada beberahap hal yang perlu diperhatikan yaitu:

1. Konsisten.
2. Identasi yang baik.

2.7.5 *Clean Error Handling*

Bagian ini menjelaskan tentang petunjuk untuk menangani kesalahan yang terjadi dengan tepat, penanganan kesalahan ditujukan untuk menampilkan

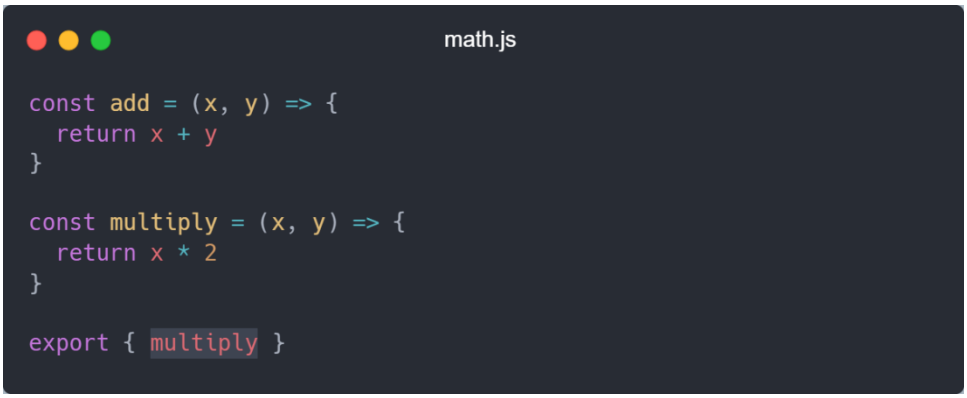
kesalahan dengan mudah dan mudah dipahami sehingga kesalahan dapat ditinjau dengan baik.

2.8 *Design Pattern*

Design Pattern merupakan bagian mendasar dari pengembangan perangkat lunak, karena memberikan solusi untuk masalah – masalah yang sering muncul dalam desain perangkat lunak. Selama beberapa tahun terakhir, ekosistem dari *web development* telah berubah dengan cepat dan beberapa *design pattern* telah berevolusi untuk memecahkan masalah modern dengan teknologi terbaru [13].

2.8.1 *Module Pattern*

Module Pattern merupakan pola yang menerapkan pemecahan kode menjadi bagian – bagian yang lebih kecil dan dapat digunakan kembali. Saat aplikasi yang dibangun berkembang sangat pesat menjadi semakin penting untuk dapat menjaga kode agar dapat dijaga di dipecah menjadi beberapa bagian. Selain dapat memecah kode menjadi bagian – bagian kecil pola ini juga dapat membantu untuk menyimpan nilai secara *private* dalam suatu file atau modul. Secara *default* jika nilai tersebut tidak di *export* maka nilai tersebut tidak akan tersedia di luar modul tersebut. Hal ini dapat mencegah dan mengurasi resiko penamaan nilai yang sama pada bagian lain. Contoh dari pola ini dapat dilihat pada Gambar 2-11 [13].



Gambar 2-11 - Contoh Module Pattern [13]