BAB 2
LANDASAN TEORI

2.1 Software Quality Assurance

Software Quality Assurance (SQA) adalah *system* urutan tindakan untuk memastikan bahwa item atau produk sesuai dengan persyaratan teknis tertentu. Tujuan kegiatan *SQA* membahas aspek fungsional manajemen dan ekonomi dari pengembangan dan pemeliharaan perangkat lunak. Dalam dunia perangkat lunak, definisi metrik dari "kualitas" dapat disimpulkan bahwa: sejauh mana perangkat lunak sesuai dengan kriteria kualitas.

Tujuan	Atribut	Metrik
Requirement Quality	*Ambiguity*	Jumlah *modifier* yang ambigu
	Completeness	Jumlah dari *TBA*, *TBD*
	Understandability	Jumlah bagan/sub bagan
	Volatility	Jumlah perubahan setiap waktu kebutuhan Ketika perubahan diminta
	Traceability	Jumlah kebutuhan yang tidak bisa di lacak untuk didesain/kode
	Model clarity	Jumlah model *UML* Jumlah halaman deskriptif setiap model
Design Quality	*Architectural integrity*	Ada tidaknya model arsitektural
	Component completeness	Jumlah komponen yang dilacak
Code Quality	*Complexity*	*Cyclomatic complexity*
	Maintainability	Maintainability Index

Tujuan	Atribut	Metrik
	Understandability	Persentase dari komentar internal Konvensi penamaan variabel
	Reusability	Persentase komponen *reusable*
	Documentation	*Readability index*
QC effectiveness	*Resource Allocation*	*Staff hour percentage per activity*
	Completion rate	*Actual vs. budgeted completion time*
	Review effectiveness	*Review metrics*
	Testing effectiveness	*Number of errors found and criticality* *Error required to correct an error* *Origin of error*

2.2 Maintainability

Maturitas dari praktek pengembangan perangkat lunak, *maintainability* telah menjadi salah satu topik utama dalam industri[3]. Menurut Rudolp Frederick Stapelerg, *maintainabaility* adalah kemungkinan suatu produk tidak akan kembali ke kondisi kerja yang efektif dalam jangka waktu tertentu. Dengan demikian dapat juga disimpulkan bahwa *maintainability* adalah kemampuan untuk melakukan pemeliharaan pada sistem selama jangka waktu tertentu ketika sistem mengalami kegagalan [4].

2.3 Maintainability Index

Maintainability Index merupakan sebuah kualitas metrik yang digunakan untuk mengukur tingkat *maintainability* sumber kode perangkat lunak. Metrik ini

dihitung denga menggunakan variabel-variabal seperti *Lines Of Code* (LOC), *Cyclomatic Complexity* (CC), *Halstead's Volume*(HV). Metrik ini menyatakan, semakin tinggi nilai hasil pengukuran *maintainability index*, maka usaha yang dibutuhkan untuk melakukan perawatan sumber kodenya akan semakin besar pula. Berikut salah rumus dari *Maintainability Index*[8]*:*

$$MI = \frac{171 - 5.2 * \log(HV) - 0.23 * CC - 16.2 * \ln(LOC)}{171} * 100$$

Nilai hasil pengukuran yang didapat dari formula sebelumnya, dapat diambil kesimpulan berdasarkan aturan pada tabel 2.1 di bawah:

Tabel 2.1 Rentang nilai *maintainability index*

Maintainability Index	Klasifikasi
MI > 85	Dapat dirawat dengan sangat baik
65 < MI ≤ 85	Cukup dapat dipelihara dengan baik
MI ≤ 65	Kurang dapat dipelihara dengan baik

2.4 Halstead's Metrics

Halstead's Metrics adalah pengukuran yang dibuat untuk mengukur secara langsung ukuran kompleksitas program dari kode sumbernya. Salah satu hipotesis menyarankan bahwa, semakin tinggi usaha yang dilakukan dalam pengembangan sebuah kode sumber, maka akan semakin tinggi pula kemungkinan terdapatnya *software error* dalam kode sumber[9]. Pengukuran *halstead's metrics* bergantung kepada eksekusi program dan pengukurannya, yang dapat dianalisa dari jumlah operator dan operan dari kode sumber. Target pengukuran *Halstead* adalah untuk mengukur kualitas tertentu, misalnya, kosa kata, volume, level, masalah, aktivitas pemrograman, dan waktu pemrograman yang diperlukan[10]. Pengukuran *halstead* didasarkan pada variabel-variabel berikut:

n1 = jumlah operator unik atau berbeda.

n2 = jumlah operan unik atau berbeda.

N1 = jumlah total kemunculan operator.

N2 = jumlah total kemunculan operan.

a. Pengukuran N (*Length of the Program*):

Berapa jumlah dari total jumlah operator dan operan dalam program?

$N=N_1+N_2$

Dimana:

N_1 = Jumlah Operator

N_2 = Jumlah Operan

b. Pengukuran n (*Vocabulary of the Program*):

Berapa jumlah dari total jumlah operator dan operan unik dalam program?

$n=n_1+n_2$

Dimana:

N_1 = Jumlah operator unik

N_2 = Jumlah operan unik

c. Pengukuran V (*Volume of Program*):

Berapa volume dari algoritma yang digunakan dalam kode sumber?

$$V = (N1 + N2)\ log2(n1 + n2)$$

$$V = N\ log2(n)$$

Dimana:

V = *Volume of the program*

N = hasil pengukuran *length of the program*

n = hasil pengukuran *vocabulary of the program*

d. Pengukuran D (*Difficulty of the Program):*

Apakah jumlah operator unik berbanding lurus dengan jumlah dari total penggunaan operan?

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

Dimana:

D = *Difficulty*

N2 = Total semua operan yang ada

n1 = Jumlah total operator unik

n2 = Jumlah total operan unik

e. Pengukuran V* (*Potential or Minimum Volume V**):

$$V^* = (2 + n2)\ log2(2 + n2)$$

Dimana:

N2 = Total semua operan yang ada

n2 = Jumlah total operan unik

f. Pengukuran L (*Implementation Level*):

$$L = \frac{V^*}{V}$$

Dimana:

V^* = Pengukutan *Potential* atau *Minimum Volume* V^*

N2 = Total semua operan yang ada

g. Pengukuran L' (*Program Level Estimator):*

$$L' = \left(\frac{2}{n1}\right) * \left(\frac{n2}{N2}\right)$$

Dimana

n1 = Jumlah total operator unik

n2 = Jumlah total operan unik

N2 = Total semua operan yang ada

h. Pengukuran I (*Intelligent Content)*

$$I = L'V$$

Dimana

L = Pengukuran *Program Level Estimator*)

V = Pengukruran *Volume Of the Program*

i. Pengukuran E (*Effort*) :

Upaya yang diperlukan untuk menerapkan atau memahami program berbanding lurus dengan kesulitan dan volume.

$$E = D * V$$

Dimana:

D = Hasil pengukuran *Diffulty*

V = Volume of the Program

j. Pengukuran B (*Bugs*):

Jumlah bug yang diprediksi dalam program:

Apakah sebanding dengan usaha?

$$B = \frac{E0.667}{3000}$$

Dimana:

E = Hasil pengukuran *Effort*

k. Pengukuran T (*Time to Implement):*

Waktu perkiraan yang dibutuhkan untuk melakukan implementasi program. Metrik ini berbanding lurus denga usaha:

$$T = \frac{E}{S}$$

Dimana:

E = Hasil pengukuran *Effort*

S = 18

2.5 Cyclomatic Complexity

Cyclomatic Complexity adalah metrik buat mengukur kompleksitas dalam sebuah fungsi menggunakan memperhatikan graf kendali, singkatnya jika fungsi nir mempunyai percabangan maka kompleksitasnya[11]. Jika mempunyai *poly* percabangan, maka perhitungannya bisa mengikuti formula berikut:

$$M = E - N + 2P$$

Dimana:

M = *Cyclomatic Complexity*

E = Jumlah edge pada graf

N = Jumlah node pada graf
P = Jumlah komponen yang terhubung

Jumlah Cyclomatic Complexity dalam suatu fungsi disarankan kurang berdasarkan 15. Jika melebihi 15, merupakan masih ada kurang lebih 15 *execution flow* percabangan dalam fungsi tersebut. Lebih berdasarkan itu, *execution flow* hukuman akan semakin sulit untuk diidentifikasi & diperiksa. Adapun batas tertinggi *execution flow* pada sebuah modul merupakan 100.

2.6 Refactoring

Menurut taksonomi *Chikofsky* dan *Cross*, restrukturisasi atau *refactoring* didefinisikan sebagai transformasi dari satu bentuk representasi ke bentuk representasi lainnya pada tingkat abstraksi relatif yang sama, sambil mempertahankan perilaku eksternal sistem subjek (fungsionalitas dan semantik)[5]. Transformasi restrukturisasi sering kali merupakan salah satu penampilan, seperti mengubah kode untuk memperbaiki strukturnya dalam pengertian tradisional desain terstruktur. Sementara restrukturisasi menciptakan versi baru yang menerapkan atau mengusulkan perubahan pada sistem subjek, biasanya tidak melibatkan modifikasi karena persyaratan baru. Namun, ini dapat mengarah pada pengamatan yang lebih baik dari sistem subjek yang menyarankan perubahan yang akan meningkatkan aspek sistem[12]

Langkah-langkah proses *refactoring* terdiri dari beberapa aktivitas yang berbeda berikut ini[12]:

1. Menemukan di bagian mana perangkat lunak perlu di lakukan *refactor*
2. Menentukan Langkah *refactoring* apakah yang cocok untuk diterapkan kepada bagian yang ditemukan
3. Memastikan bagian yang di refactor mempertahankan sifat yang sama.
4. Melakukan *refactoring*
5. Menilai efek refactoring pada kualitas karakteristik perangkat lunak (misalnya, kompleksitas, pemahaman, pemeliharaan) atau proses (misalnya, produktivitas, biaya, usaha).

6. Pertahankan konsistensi antara kode program *refactored* dan artefak perangkat lunak lainnya (seperti dokumentasi, dokumen desain, spesifikasi persyaratan, pengujian, dll.).

2.7 Design Pattern

Design pattern adalah sebuah cara atau solusi bagaimana memecahkan permasalahan yang biasa terjadi dalam pengembangan perangkat lunak berbasis objek untuk dapat digunakan terus menerus pada permasalahan yang sama[4].

Design pattern menjelaskan cara komunikasi antar objek dan modul yang sudah disesuaikan dengan kebutuhan untuk memecahkan suatu masalah desain yang umum dalam suatu konteks tertentu. Berdasarkan buku *Learning Javascript* Design Pattern (Addy Osmani, 2012) yang digunakan sebagai referensi penggunaan design pattern pada penelitian ini, ditemukan 11 *design pattern* yang dikelompok menjadi 3 kategori, yaitu *Creational, Structural*, dan *Behavioral*.

1. Creational Design Pattern

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 :

a. Factory Method
b. Abstract Factory
c. Builder
d. Prototype
e. Singleton

2. Structural Design Pattern

Struktural *Pattern* ini menyederhanakan struktur suatu sistem menggunakan identifikasi interaksi antar objek. Berikut merupakan *pattern* dalam kategori ini :

a. *Adapter*
b. *Bridge*

c. *Composite*
d. *Decorator*
e. *Façade*
f. *Flyweight*
a. *Proxy*

3. *Behavioural Design Pattern*

Behavioral Design Pattern herbi hubungan & komunikasi antara banyak sekali objek. Hal ini berupaya mengurangi kerumitan yang mungkin terjadi waktu objek saling berkomunikasi. Berikut merupakan design pattern dalam kategori ini :

a) *Interpreter*
b) *Template Method*
c) *Chain of Responsibility*
d) *Command*
e) *Iterator*
f) *Mediator*
g) *Memento*
h) *Observer*
i) *State*
j) *Strategy*
k) *Visitor*

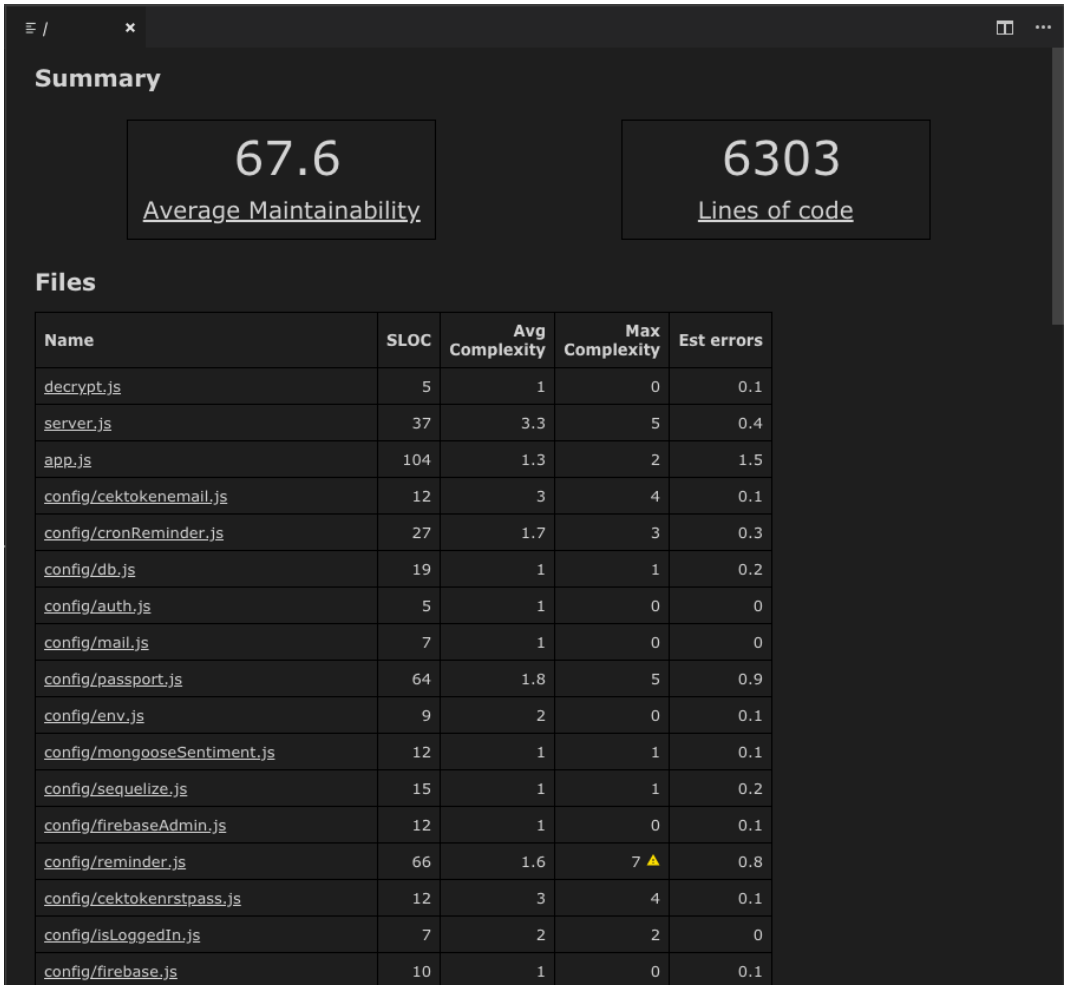
2.8 JS Complexity Analysis

JS Complexity Analysis merupakan sebuah *plugin* untuk *teks editor Microsoft* Visual Studio Code. *JS Complexity Analysis* digunakan untuk menghitung kompleksitas kode sumber dari sebuah perangkat lunak berbasis *javascript*. Metrik yang dapat dihitung oleh plugin ini antara lain:

a) Jumlah baris kode sumber
b) Jumlah parameter dari masing-masing
c) Cyclomatic Complexity
d) Halstead's Metric

e) Maintainability

Dalam menghitung *maintainability*, *plugin* ini menggunakan formula *maintainability index* yang digunakan pada Microsoft Visual Studio. Adapun hasil analisis dari plugin ini dapat dilihat pada Gambar 2.1 & Gambar 2.2:



Summary

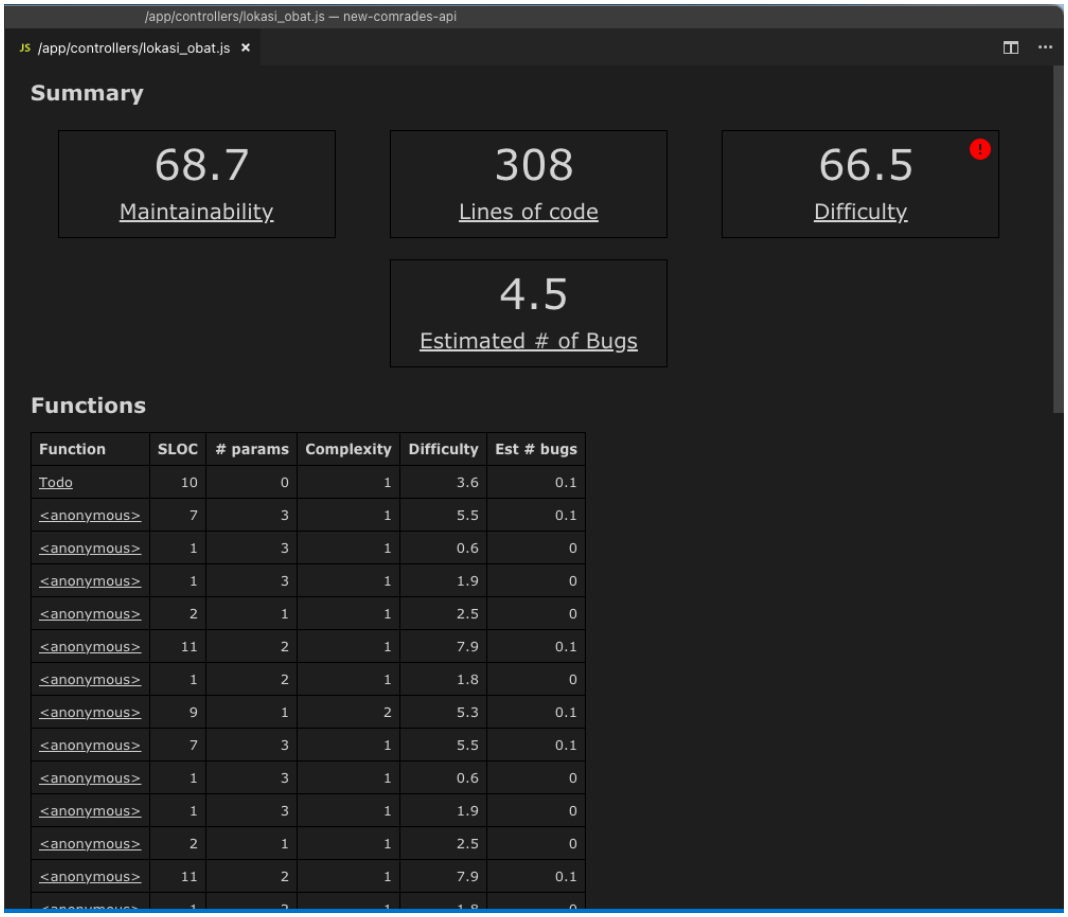
67.6
Average Maintainability

6303
Lines of code

Files

Name	SLOC	Avg Complexity	Max Complexity	Est errors
decrypt.js	5	1	0	0.1
server.js	37	3.3	5	0.4
app.js	104	1.3	2	1.5
config/cektokenemail.js	12	3	4	0.1
config/cronReminder.js	27	1.7	3	0.3
config/db.js	19	1	1	0.2
config/auth.js	5	1	0	0
config/mail.js	7	1	0	0
config/passport.js	64	1.8	5	0.9
config/env.js	9	2	0	0.1
config/mongooseSentiment.js	12	1	1	0.1
config/sequelize.js	15	1	1	0.2
config/firebaseAdmin.js	12	1	0	0.1
config/reminder.js	66	1.6	7 ▲	0.8
config/cektokenrstpass.js	12	3	4	0.1
config/isLoggedIn.js	7	2	2	0
config/firebase.js	10	1	0	0.1

Gambar 2.1 JS Complexity I

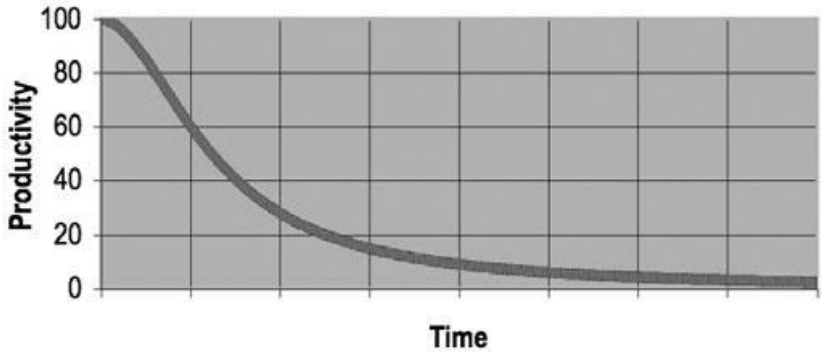


Gambar 2.2 JS Complexity II

2.9 Clean Code

Clean Code merupakan suatu konsep yang menyatakan bahwa, kode yang baik adalah kode yang mudah dimengerti dan dikembangkan oleh pengembang lain sehingga dapat mempercepat proses pengembangan dan perawatannya sendiri [5]..

Konsep *clean code* bertujuan untuk mencegah penurunan produktivitas kerja tim dalam sebuah pengembangan perangkat lunak. Kode sumber yang berantakan, akan menyebabkan penurunan tingkat produktivitas tim pengembang. Hal ini dapat diilustrasikan oleh Gambar 2.3._Beberapa hal yang biasa dibahas mengenai *clean code* terdiri dari *Meaningful Names, Clean Functions, Clean Comments, Clean Code Formatting, Clean Error Handling, Clean Object and Data Structure, Clean Classes.*



Gambar 2.3 Productivity vs Time

a. Meaningful Name

Meaningful Name merupakan aturan penamaan terhadap *class, variable, function* dkk., agar lebih mudah dipahami oleh pengembang lain.

Berikut ini akan dipaparkan bagian-bagian sering menjadi aturan dalam mewujudkan *meaningful name:*

1. Use Intention-Revealing Names

Penamaan harus dapat mewakilkan tujuan, maksud dari sebuah variable atau fungsi, gunakan penamaan yang eksplisit. Jika nama tersebut masih membutuhkan komentar, itu berarti nama itu belum mepresentasikan apapun.

2. *Avoid Disinformation*

Hindari penamaan yang merepresantasikan hal yang ambigu sehingga menimbulkan disinformasi.

3. *Use Pronounceable Names*

Gunakan nama yang bisa dieja, sehingga mudah diingat dan diucapkan.

4. *Use Searchable Names*

Gunakan nama yang mudah dicari. Denga cara seperti ini, pengembang tidak perlu harus memahami seluruh bagian kode secara kesuluruhan.

5. *Avoid Mental Mapping*

Hindari penamaan hanya denga menggunakan satu huruf. Hal ini biasa terjadi dalam sebuah looping (variable I, j, k).

b. *Clean Function*

Aturan ini berfokus kepada bagaimana membuat fungsi yang bersih agar lebih mudah dipahami. Beberapa diantaranya adalah:

1. Small

Penamaan fungsi, pernyataan maupun jumlah *line of code* sebuah fungsi, diusahakan dibuat lebih singkat agar muda dikelola dan dipahami.

2. Do One Thing

Hindari fungsi yang memilki aktivitas yang lebih dari satu dalam satu fungsi. Jika ada, maka disarankan harus dipecah menjadi beberapa fungsi yang lebih kecil denga tujuan masing-masing berbeda.

3. Use Descriptive Names

Nama fungsi diharuskan mewakili apa yang akan dilakukan oleh fungsi tersebut. Usahakan tulis fungsi yang kecil sehingga lebih mudah memilih nama yang deskriptif.

4. Function Arguments

Fungsi yang ideal adalah fungsi yang memiliki 0 argument. Semakin sedikit argument sebuah fungsi, maka akan semakin mudah dikelola dan dipahami.

c. *Clean Comment*

Ada petunjuk dalam ide ini untuk menulis komentar yang efektif dan efisien, sehingga komentar yang dibuat dapat memberikan informasi yang secara eksplisit tidak disediakan oleh kode sumber. Penggunaan *clean code*, di sisi lain, terkait dengan penggunaan nama yang bermakna, yang memerlukan peningkatan penamaan variabel, metode/fungsi, atau kelas perbaikan. Berikut adalah beberapa panduan untuk membuat komentar yang bersih:

1) Good Comment

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

a. *Legal Comment* yaitu komentar yang terpaksa harus ditulis karena alasan hukum 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 perangkat lunak, komentar termasuk dalam *bad comment*. Komentar yang diberikan terkadang seharusnya tidak usah diberikan, tetapi developer memasukkan 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.

d. *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. 23 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 perangkat lunak. Diperlukan tindakan lebih lanjut agar pesan kesalahan tidak hanya ditampilkan ke *logger*, tetapi perangkat lunak dapat memberikan pesan berupa notifikasi, baik terhadap pengguna maupun pengembang, tentang kesalahan yang terjadi.

e. Clean Classes

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

1) Single Responsibility Principle

Suatu class/modul disarankan untuk berukuran tidak terlalu besar, selain banyak jumlah baris yang ditulis, pengembang akan kerepotan dalam memahami *class*/modul tersebut. Dengan menjaga ukuran *class*/modul untuk tidak terlalu besar, dapat mempermudah proses modifikasi. Ukuran suatu *class*/modul diukur berdasarkan jumlah *responsibility* yang ditanggung oleh *class*/modul tersebut. Class/modul yang bersih merupakan *class*/modul yang hanya memiliki satu *responsibility* saja. Hampir sama dengan *clean function*, apabila *class*/modul tersebut memiliki lebih dari satu *responsibility*, pisahkan *responsibility* ke dalam *class*/modul yang berbeda.

2) *Class Organization*

Pembuatan *class*/modul yang baik dapat dilakukan dengan mengikuti *code convetion*/*code styling* dari bahasa pemrograman yang digunakan. Adapun pada

penelitian ini, bahasa pemrograman yang digunakan merupakan Javascript, sehingga pembuatan *class*/modul mengikut aturan yang ada pada standar bahasa tersebut, yaitu ES2015/ES6.