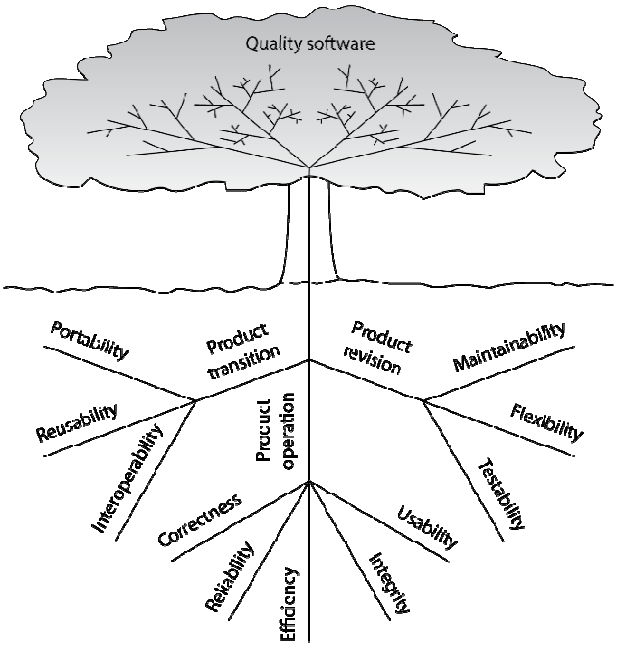
# BAB 2
# LANDASAN TEORI

## 2.1 *Software Quality Assurance*

*Software Quality Assurance* (SQA) merupakan suatu sistem dari serangkaian tindakan untuk meyakinkan suatu barang atau produk sudah sesuai dengan persyaratan teknis yang ditetapkan (IEEE, 1991) [1]. Tujuan dari kegiatan SQA mengacu kepada aspek fungsional, manajerial, dan ekonomi dari pengembangan dan pemeliharaan perangkat lunak. Salah satu model pengujian perangkat lunak adalah model *McCall*, di mana model pengujian ini terdiri dari 11 faktor yang dibagi menjadi 3 kelompok, yaitu *Product Operation (Correctness, Reliability, Efficiency, Integrity, Usability), Product Revision (Maintainability, Flexibility, Testability),* dan *Product Transition (Portability, Reusability, Interoperability)* [2] [1]. Selanjutnya, pembagian tersebut digambarkan sebagai Pohon *McCall* yang dapat dilihat pada Gambar 2.1.




**Gambar 2.1 Pohon *McCall*, Pembagian Faktor Pengujian** [1]

Dari faktor kualitas perangkat lunak yang diperlihatkan pada Gambar 2.1, terdapat sub faktor pengujian yang memperjelas pengujian yang dilakukan oleh setiap faktor. Sub faktor tersebut dapat dilihat pada Tabel 2.1.

**Tabel 2.1 Sub Faktor Dari Setiap Faktor Kualitas Perangkat Lunak**

| Kategori | Faktor Kualitas Perangkat Lunak | Sub Faktor |
|---|---|---|
| *Product Operation* | *Correctness* | *Accuracy* |
| | | *Completeness* |
| | | *Up-to-dateness* |
| | | *Availability (Response Time)* |
| | | *Coding and Documentation Guidelines* |
| | | *Compliance (Consistency)* |
| | *Reliability* | *System Reliability* |
| | | *Application Reliability* |
| | | *Computational Failure Recovery* |
| | | *Hardware Failure Recovery* |
| | *Efficiency* | *Efficiency of Processing* |
| | | *Efficiency of Storage* |
| | | *Efficiency of Communication* |
| | | *Efficiency of Power Usage (for* |

| Kategori | Faktor Kualitas Perangkat Lunak | Sub Faktor |
|---|---|---|
| | | *portable units)* |
| | *Integrity* | *Access Control* |
| | | *Access Audit* |
| | *Usability* | *Operability* |
| | | *Training* |
| *Product Revision* | *Maintainability* | *Simplicity* |
| | | *Modularity* |
| | | *Self-Descriptiveness* |
| | | *Coding and Documentation Guidelines* |
| | | *Compliance (Consistency)* |
| | | *Document Accessibility* |
| | *Flexibility* | *Modularity* |
| | | *Generality* |
| | | *Simplicity* |
| | | *Self-Descriptiveness* |
| | *Testability* | *User Testability* |
| | | *Failure Maintenance Testability* |

| Kategori | Faktor Kualitas Perangkat Lunak | Sub Faktor |
|---|---|---|
| | | *Tracability* |
| *Product Transition* | *Portability* | *Software System Independence* |
| | | *Modularity* |
| | | *Self Descriptive* |
| | *Reusability* | *Modularity* |
| | | *Document Accessibility* |
| | | *Software System Independence* |
| | | *Application Independence* |
| | | *Self Descriptive* |
| | | *Generality* |
| | | *Simplicity* |
| | *Interoperability* | *Commonality* |
| | | *System Compatibility* |
| | | *Software System Independence* |
| | | *Modularity* |

## 2.2 *Maintainability*

*Maintainability* merupakan salah satu faktor pada SQA, di mana *maintainability* berfokus pada kemudahan dari perangkat lunak untuk dipelihara [2]. *Maintainability* menentukan suatu upaya yang dibutuhkan oleh tim yang melakukan pemeliharaan untuk mengidentifikasi penyebab dari suatu kesalahan pada perangkat lunak, memperbaiki kesalahan yang ada, dan melakukan pengujian terhadap koreksi kesalahan yang sudah dilakukan [1]. Faktor ini mengacu pada seberapa modularnya struktur dari suatu perangkat lunak, dokumentasi internal dari program, dan dokumen manual untuk pengembang [1]. Bentuk kebutuhan *maintainability* yang umum di antaranya [1]:

1) Pengembang mengikuti standar dan pedoman penulisan kode dari perusahaan pemilik perangkat lunak.
2) Ukuran dari modul perangkat lunak tidak lebih dari 30 pernyataan

Menurut *McCall*, faktor ini terdiri atas beberapa sub faktor yang dapat dilihat pada Tabel 2.2.

**Tabel 2.2 Sub Faktor dari *Maintainability***

| Faktor Kualitas Perangkat Lunak | Sub Faktor Kualitas Perangkat Lunak | Keterangan |
|---|---|---|
| *Maintainability* | *Simplicity* | Kemudahan dalam memahami perangkat lunak |
| | *Modularity* | Ukuran modul pada perangkat lunak |
| | *Self-Descriptiveness* | Kode sumber yang mudah untuk dipahami |
| | *Coding and Documentation Guidelines* | Ketersediaan petunjuk penulisan kode dan dokumentasi |

| Faktor Kualitas Perangkat Lunak | Sub Faktor Kualitas Perangkat Lunak | Keterangan |
|---|---|---|
| | *Compliance (Consistency)* | Penggunaan desain dan teknik implementasi yang seragam |
| | *Document Accessibility* | Kemudahan pengaksesan dokumen perangkat lunak |

## 2.3 *Maintainability Index*

*Maintainability Index (MI)* merupakan suatu metrik komposit yang menggabungkan sejumlah metrik kode tradisional menjadi satu nilai yang menyatakan nilai relatif dari aspek *maintainability* [7] [8]. *Maintainability Index* terdiri atas metrik *Halstead Volume (HV)*, metrik *Cyclomatic Complexity (CC)*, rata-rata jumlah baris kode per modul (LOC), dan persentase jumlah komentar per modul (COM) [9]. Adapun formula *Maintainability Index* dapat dilihat pada Gambar 2.2.

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) + 50 * \sin(\sqrt{2.4 * COM})$$

**Gambar 2.2 *Maintainability Index Formula* [8]**

Akan tetapi, terdapat banyak varian dalam perhitungan *Maintainability Index*. Seperti yang digunakan oleh Microsoft pada aplikasi Visual Studio yang dapat dilihat pada Gambar 2.3.

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

Di mana $0 \leq MI \leq 100$

**Gambar 2.3 *Maintainability Index Formula* pada *Microsoft Visual Studio*** [10]

Adapun formula *Maintainability Index* yang digunakan pada penelitian ini adalah formula yang digunakan pada Microsoft Visual Studio. Secara umum, nilai dari *Maintainability Index* diukur dari 0 sampai 100, di mana semakin tinggi nilai tersebut menandakan tingginya *maintainability* dari kode sumber yang diukur [10]. Nilai tersebut terbagi menjadi tiga kategori yang dapat dilihat pada Tabel 2.3.

**Tabel 2.3 Rentang Penilaian *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 |

## 2.4 *Halstead Complexity*

*Halstead Complexity* merupakan salah satu metrik yang digunakan untuk mengukur kompleksitas dengan cara menghitung setiap operator dan operan yang terdapat dalam *class*/modul pada suatu perangkat lunak [11]. Metrik ini biasa digunakan dalam metrik perhitungan *maintainability*. Pengukuran metrik ini terdiri atas:

1. *Operand* dan *Operator*

Operan dan operator merupakan kunci dalam perhitungan *Halstead Complexity*, di mana simbol yang digunakan untuk menggambar kedua hal tersebut adalah sebagai berikut:

$\eta_1 = jumlah\ operator\ unik$

$\eta_2 = jumlah\ operan\ unik$

$N_1 = jumlah\ operator\ keseluruhan$

$N_2 = jumlah\ operan\ keseluruhan$

Operator pada perhitungan terdiri atas operator matematika, kata kunci pada bahasa pemrograman yang digunakan seperti *for*, *if* dan *var*, simbol pemisah

seperti tanda kurung, tanda koma, dan titik koma. Sedangkan operan yang dimaksud berupa variabel, konstanta, angka, maupun kumpulan karakter. Contoh perhitungan dapat dilihat pada Tabel 2.4.

```
const audience = require('../models/keywords');

class Keyword{
    searchKeywords(req, res, next){
        audience.find({keyword : req.body.keyword}, (err, docs)=>{
            res.json({
                status_code : 200,
                messages : "Success get keywords",
                results : docs
            })
        })
    }
}

module.exports = new Keyword();
```

**Gambar 2.4 Sampel Perhitungan Operan Dan Operator Pada Metrik *Halstead***

**Tabel 2.4 Hasil Perhitungan Operan Dan Operator**

| Operator | Operan |
|---|---|
| const<br>require<br>class<br>module<br>new<br>=<br>()<br>{}<br>=><br>,<br>;<br>: | audience<br>Keyword<br>keyword<br>searchKeyword<br>req<br>res<br>next<br>err<br>docs<br>status_code<br>messages<br>results<br>'../models/keywords'<br>'Success get keyword'<br>200 |
| $\eta_1 = 12$ | $\eta_2 = 15$ |
| $N_1 = 30$ *dari Gambar 2.4 | $N_2 = 20$ *dari Gambar 2.4 |

2. *Program Length*

*Program length* merupakan panjang dari program hasil dari penjumlahan total operator dan operan.

$$N = N_1 + N_2$$

3. *Program Vocabulary*

*Program vocabulary* merupakan hasil penjumlahan dari jumlah unik operator dan operan yang terdapat pada program.

$$\eta = \eta_1 + \eta_2$$

4. *Volume*

*Volume* merupakan ukuran dari algoritma yang diimplementasikan pada program. Perhitungan pada volume didasarkan pada jumlah dari operator yang digunakan dan operan yang terpakai pada algoritma, di mana formulanya adalah sebagai berikut:

$$V = N * log_2\eta$$

Suatu fungsi setidaknya memiliki volume antara 20 sampai 1000. Apabila suatu fungsi memiliki volume lebih dari 1000 itu berarti kemungkinan fungsi tersebut memiliki terlalu banyak pekerjaan yang dilakukan.

5. *Difficulty*

Metrik ini digunakan untuk mengukur kerawanan terjadinya kesalahan pada program dengan menghitung proporsi jumlah unik dari operator yang ada pada program, dan proporsi rasio antara total jumlah operan dan jumlah unik operan pada program. Berikut formulanya:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

6. *Effort to Implement*

Metrik ini merupakan metrik untuk usaha untuk mengimplemen atau memahami program yang dihitung berdasarkan nilai *Volume* dan nilai *Difficulty* dari program. Berikut formulanya:

$$E = V * D$$

7. *Time to Implement*

Metrik ini digunakan untuk mengukur waktu untuk mengimplemen atau memahami program dalam satuan detik. Metrik ini menghitung proporsi usaha atau E. Berikut formulanya:

$$T = \frac{E}{18}$$

8. *Number of Delivered Bugs*

Metrik ini digunakan untuk mengukur jumlah *bug* yang muncul. Metrik ini berkorelasi dengan kompleksitas dari program secara keseluruhan. Berikut formulanya:

$$B = \frac{E^{\frac{2}{3}}}{3000}$$

## 2.5 *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 [12]. Apabila memiliki banyak percabangan, maka perhitungannya dapat mengikuti formula berikut:

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

Keterangan:

$M = Cyclomatic\ Complexity$

$E = Jumlah\ edge\ pada\ graf$

$N = Jumlah\ node\ pada\ graf$

$P = Jumlah\ komponen\ yang\ terhubung$

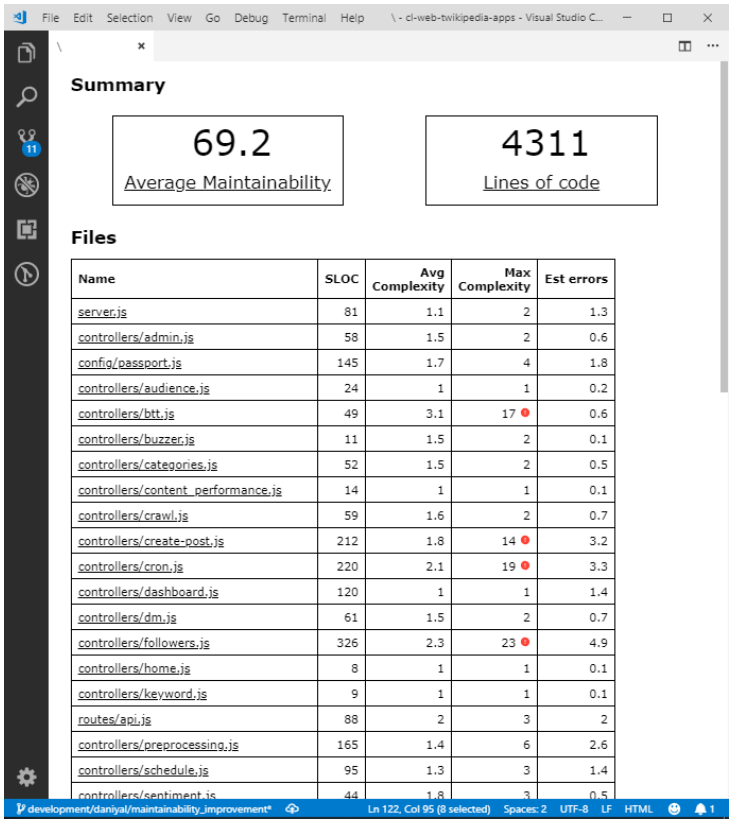
Jumlah *Cyclomatic Complexity* pada suatu fungsi disarankan kurang dari 15. Apabila melebihi 15, artinya terdapat sekitar 15 jalur eksekusi percabangan pada fungsi tersebut. Lebih dari itu, jalur eksekusi akan semakin sulit untuk diidentifikasi dan diperiksa. Adapun batas tertinggi jumlah jalur eksekusi dalam sebuah modul adalah 100 [13].

## 2.6 *JS Complexity Analysis*

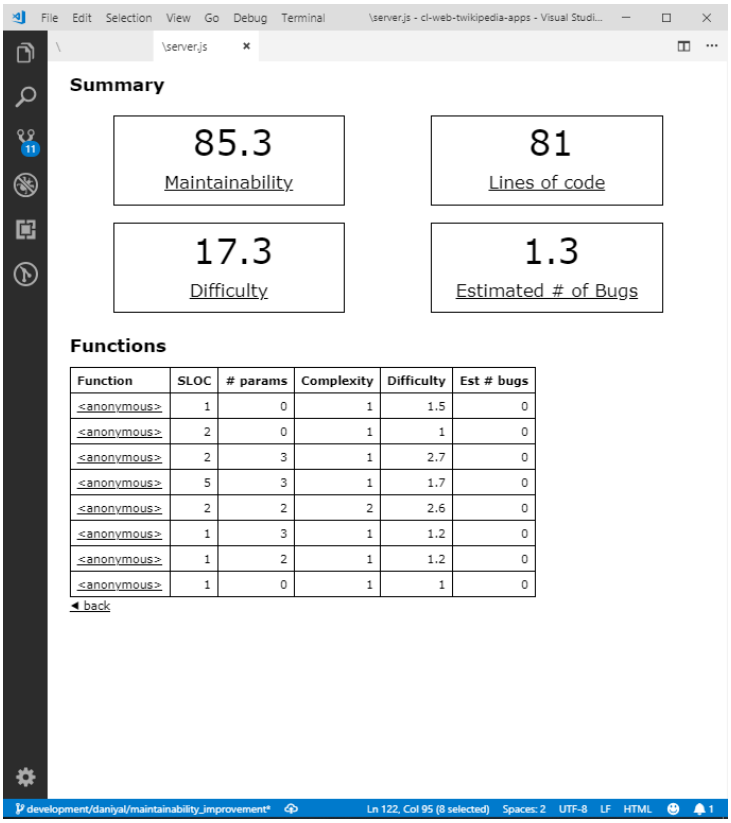
*JS Complexity Analysis* merupakan sebuah *plugin* untuk teks editor *Microsoft Visual Studio Code*. *JS Complexity Analysis* digunakan untuk menghitung kompleksitas kode sumber dari sebuah aplikasi berbasis *Javascript*. Metrik yang dapat dihitung oleh *plugin* ini antara lain:

1) Jumlah baris kode sumber
2) Jumlah parameter dari masing-masing fungsi
3) *Cyclomatic Complexity*
4) *Halstead Metric*
5) *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 2.5 dan Gambar 2.6.




**Gambar 2.5 Hasil Analisis *JS Complexity Analysis* Secara Keseluruhan**



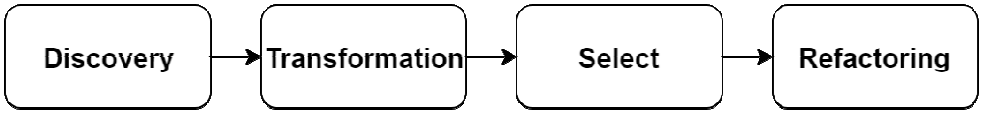

**Gambar 2.6 Hasil Analisis *JS Complexity Analysis* Pada Sebuah Modul**

## 2.7 *Code Readability Metric*

*Code Readability Metric* merupakan metrik yang digunakan untuk menilai suatu tingkat keterbacaan dari suatu kode sumber. Penilaian pada kode sumber dilakukan dengan menguji kode sumber tersebut ke partisipan atau dengan pengembang dari aplikasi yang diteliti [14]. Bobot penilaian disediakan oleh peneliti, sehingga partisipan cukup memberikan nilai yang sesuai berdasarkan rentang bobot nilai yang disediakan. Penggunaan *Code Readability Metric* sendiri dimaksudkan untuk mengumpulkan data latih yang digunakan oleh program yang dibangun untuk menilai *readability* atau tingkat keterbacaan dari suatu kode sumber berbasis Java. Penelitian tentang *Code Readability Metric* ini dikembangkan oleh Raymond P. L. Buse dan Westley R. Weimer. Sehingga pada penelitian ini, penilaian tingkat keterbacaan dari suatu kode sumber mengikuti metode yang sudah mereka lakukan.

## 2.8 *Refactoring*

*Refactoring* merupakan suatu tindakan perubahan terhadap struktur internal dari perangkat lunak agar lebih mudah untuk dipahami dan melakukan modifikasi tanpa mempengaruhi perilakunya (UML Distilled, 2003) [6]. Proses *refactoring* meliputi penyederhanaan logika yang kompleks, penghapusan duplikasi, dan melakukan klarifikasi terhadap kode yang tidak jelas [6]. Adapun tahapan proses yang dilakukan dalam *refactoring* secara umum dapat dilihat pada Gambar 2.7.




**Gambar 2.7 Tahapan *Refactoring* Secara Umum** [15]

1. *Discovery*

*Discovery* merupakan tahap dalam melakukan pencarian kecacatan pada kode sumber yang diperiksa. Kecacatan tersebut dapat berupa duplikasi, struktur kode yang berantakan, atau pun pustaka yang sudah usang *(deprecated)*.

2. *Transformation*

Pada tahap ini, dari hasil pencarian kecacatan yang ditemukan, ditentukan bentuk-bentuk perubahan yang dapat diterapkan untuk memperbaiki kecacatan yang ditemukan.

3. *Select*

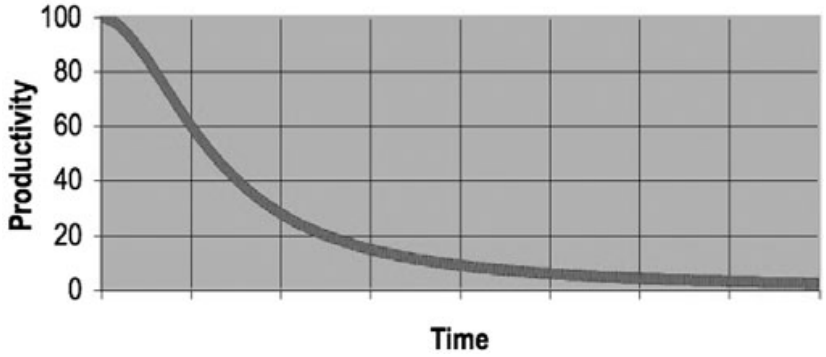
Pada tahap ini dilakukan pemilihan dari bentuk solusi yang terbentuk agar solusi yang diterapkan merupakan solusi terbaik yang dapat diberikan.

4. *Refactoring*

Pada tahap ini dilakukan *refactoring* atau penerapan solusi yang dipilih.

## 2.9 *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 [3]. *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 [3].




**Gambar 2.8 *Productivity vs. Time*** [3]

Adapun hal inti yang dibahas dalam *Clean code* adalah sebagai berikut:

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

### 2.9.1 *Meaningful Name*

Pada konsep ini terdapat petunjuk dalam melakukan pemberian nama terhadap variabel, *method*/fungsi, dan *class*/modul 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. Contoh petunjuk ini dapat dilihat pada Gambar 2.9.

```
Buruk
const yyyymmdstr = moment().format("YYYY/MM/DD");

Baik
const currentDate = moment().format("YYYY/MM/DD");
```

**Gambar 2.9 Contoh Penggunaan Nama Yang Dapat Dieja Dan Memiliki Arti**

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. Contoh petunjuk ini dapat dilihat pada Gambar 2.10.

```
Buruk
setTimeout(blastOff, 86400000);

Baik
const MILLISECONDS_IN_A_DAY = 86400000;
setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

**Gambar 2.10 Contoh Nama Yang Mudah Dicari**

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. Contoh petunjuk ini dapat dilihat pada Gambar 2.11.

```
Buruk
const locations = ["Bandung", "Cimahi", "Sumedang"];
locations.forEach(l => {
  doStuff();
  // ...
  dispatch(l);
});

Baik
const locations = ["Bandung", "Cimahi", "Sumedang"];
locations.forEach(location => {
  doStuff();
  // ...
  dispatch(location);
});
```

**Gambar 2.11 Contoh Menghindari Nama Yang Bersifat *Mental Mapping***

4) Kata benda untuk *class*/modul

Dalam melakukan pemberian nama terhadap *class*/modul, disarankan untuk menggunakan kata benda dan menghindari penggunaan kata kerja. Hal tersebut didasari oleh penggunaan *class*/modul yang biasanya digunakan mewakili suatu entitas. Adapun contohnya seperti penggunaan kata *User*, *AuthController*, *MessageParser*, dan *Controller*.

5) Kata kerja untuk *method*/fungsi

Tidak seperti penamaan *class*/modul, 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*. Contoh petunjuk ini dapat dilihat pada Gambar 2.12.

```
const name = employee.getName();
customer.setName("Mike");
if (paycheck.isPosted()){
    printInvoice();
}
```

**Gambar 2.12 Contoh Kata Kerja Pada Penamaan *Method*/Fungsi**

6) Menghindari penggunaan konteks yang tidak diperlukan

Hal ini biasa terjadi pada suatu *class*/modul atau pada objek, di mana atributnya mengandung nama dari *class*/modul atau objek tempat atribut tersebut berada. Hal tersebut tidak disarankan, karena bersifat repetitif. Contoh petunjuk ini dapat dilihat pada Gambar 2.13.

```
Buruk
let car = {
    carMake: 'Honda',
    carModel: 'Accord',
    carColor: 'Blue'
};

function paintCar(car) {
    car.carColor = 'Red';
}

Baik
let car = {
    make: 'Honda',
    model: 'Accord',
    color: 'Blue'
};

function paintCar(car) {
    car.color = 'Red';
}
```

**Gambar 2.13 Contoh Penamaan Tanpa Konteks Yang Tidak Perlu**

### 2.9.2 *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. Contoh petunjuk ini dapat dilihat pada Gambar 2.14.

```
Buruk
function createMenu(title, body, buttonText, cancellable) {
  // ...
}

Baik
function createMenu({ title, body, buttonText, cancellable }) {
  // ...
}

createMenu({
    title: 'Foo',
    body: 'Bar',
    buttonText: 'Baz',
    cancellable: true
});

// atau

let menuDetail = {
    title: 'Foo',
    body: 'Bar',
    buttonText: 'Baz',
    cancellable: true
};

createMenu(menuDetail);
```

**Gambar 2.14 Contoh Objek Sebagai Parameter Pada *Method*/fungsi**

2) Menghindari penggunaan *flag* sebagai parameter

Penggunaan *flag* sebagai parameter dapat menandakan *method*/fungsi yang dibangun memiliki pekerjaan lebih dari satu. Contoh petunjuk ini dapat dilihat pada Gambar 2.15.

```
Buruk
function createFile(name, temp) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}

Baik
function createFile(name) {
  fs.create(name);
}
function createTempFile(name) {
  createFile(`./temp/${name}`);
}
```

**Gambar 2.15 Contoh Menghindari Penggunaan *Flag* Sebagai Parameter**

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. Contoh petunjuk ini dapat dilihat pada Gambar 2.16.

```
Buruk
function emailClients(clients) {
  clients.forEach(client => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}

Baik
function emailActiveClients(clients) {
  clients.filter(isActiveClient).forEach(email);
}

function isActiveClient(client) {
  const clientRecord = database.lookup(client);
  return clientRecord.isActive();
}
```

**Gambar 2.16 Contoh *Method*/fungsi Yang Melakukan Satu Pekerjaan**

4) Hindari efek samping

Maksud dari menghindari efek samping ini, yaitu menghindari mutasi terhadap variabel global atau parameter yang bersifat referensi oleh *method*/fungsi yang ditulis karena hal tersebut bersifat fatal. Pada bahasa pemrograman yang digunakan pada penelitian ini, yaitu *Javascript*, tipe data primitif dimasukkan sebagai suatu nilai saat dijadikan sebagai argumen, sedangkan larik dan objek dimasukkan sebagai suatu referensi. Contoh petunjuk ini dapat dilihat pada Gambar 2.17 dan Gambar 2.18.

```
Buruk
let name = "Ryan McDermott";

function splitIntoFirstAndLastName() {
  name = name.split(" ");
}

splitIntoFirstAndLastName();

console.log(name); // ['Ryan', 'McDermott'];


Baik
function splitIntoFirstAndLastName(name) {
  return name.split(" ");
}

const name = "Ryan McDermott";
const newName = splitIntoFirstAndLastName(name);

console.log(name); // 'Ryan McDermott';
console.log(newName); // ['Ryan', 'McDermott'];
```

**Gambar 2.17 Contoh Menghindari Efek Samping Dari Mutasi Global**

```
Buruk
const addItemToCart = (cart, item) => {
  cart.push({ item, date: Date.now() });
};

Baik
const addItemToCart = (cart, item) => {
  return [...cart, { item, date: Date.now() }];
};
```

**Gambar 2.18 Contoh Menghindari Efek Dari Mutasi Parameter**

### 2.9.3 *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 *class*/modul semakin sedikit pula komentar yang harus diberikan. Berikut merupakan petunjuk dalam melakukan penulisan komentar:

1) Komentar hanya untuk hal-hal yang memiliki kompleksitas logika

Penulisan kode sumber yang baik dapat menghasilkan kode yang mudah dipahami sehingga penggunaan komentar dapat dikurangi. Akan tetapi, tidak semua kode yang baik dapat mengurangi kompleksitas logika yang rumit, sehingga pengembang membutuhkan petunjuk dalam memahami kode sumber tersebut. Contoh petunjuk ini dapat dilihat pada Gambar 2.19.

```
Buruk
function hashIt(data) {
  // ini hash
  let hash = 0;

  // panjang dari sebuah string
  const length = data.length;

  // Loop setiap karakter dalam sebuah data
  for (let i = 0; i < length; i++) {
    // mendapatkan karakter di kode
    const char = data.charCodeAt(i);
    // Membuah hash
    hash = ((hash << 5) - hash) + char;
    // mengonversi ke 32-bit integer
    hash &= hash;
  }
}

Baik
function hashIt(data) {
  let hash = 0;
  const length = data.length;

  for (let i = 0; i < length; i++) {
    const char = data.charCodeAt(i);
    hash = ((hash << 5) - hash) + char;

    // Mengonversi ke 32-bit integer
    hash &= hash;
  }
}
```

**Gambar 2.19 Contoh Pemberian Komentar Pada Bagian Yang Penting**

2) Komentar yang informatif dan memberikan klarifikasi

Komentar yang baik dapat menjelaskan informasi mendasar dari kode sumber yang ditulis, seperti menjelaskan keluaran dari *method*/fungsi yang dibuat atau menjelaskan format yang tidak bisa dibaca oleh manusia (*Regular Expression*). Upaya tersebut dilakukan untuk memberikan klarifikasi terhadap pengembang yang sedang mencoba untuk memahami kode sumber yang ada. Contoh petunjuk ini dapat dilihat pada Gambar 2.20 dan Gambar 2.21.

```
// Mengembalikan instance Responder yang sedang diuji.
function responderInstance();
```

**Gambar 2.20 Contoh Penjelasan Keluaran Dari Suatu *Method*/fungsi**

```
// pencocokan format dengan kk:mm:ss EEE, MMM dd, yyyy
let timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*,
\\d*");
```

**Gambar 2.21 Contoh Penjelasan Format *RegEx* Agar Dapat Dibaca**

3) Menghindari kode yang dikomentari pada basis kode

Suatu kode yang terpakai dan dikomentari tidak disarankan untuk dibiarkan begitu saja. Karena akan merumitkan kode sumber dan membuat pengembang menerka-nerka kegunaan dari kode tersebut. Simpan kode tersebut ke dalam riwayat dengan memanfaatkan *version control* seperti *Git*, sehingga kode tersebut akan tetap aman dan basis kode menjadi lebih bersih.

### 2.9.4 *Clean Code Formatting*

Pada konsep ini terdapat petunjuk dalam melakukan penulisan kode sumber agar mudah untuk dibaca dengan memanfaatkan format penulisan, hal ini meliputi penggunaan *indentation*, spasi, tab dan penempatan kode berdasarkan penggunaannya. Penentuan format penulisan dapat mengikuti *code convention/code styling* dari bahasa pemrograman yang digunakan atau mengikuti kesepakatan dari tim pengembang. Berikut merupakan petunjuk penulisan kode sumber dengan format penulisan yang baik:

1) Konsisten

Konsistensi terhadap format penulisan perlu diperhatikan agar kode yang ditulis tidak berantakan dan dapat mempermudah pengembang lain dalam memahami kode tersebut. Meskipun kode yang ditulis tidak sesuai dengan *code convention/code styling* dari bahasa pemrograman yang digunakan tidak menjadi kendala selama format penulisan yang digunakan konsisten dan mudah untuk dibaca. Contoh petunjuk ini dapat dilihat pada Gambar 2.22.

**Buruk**

```
// Penamaan Konstanta
const DAYS_IN_WEEK = 7;
const daysInMonth = 30;

const songs = ["Back In Black", "Stairway to Heaven", "Hey
Jude"];
const Artists = ["ACDC", "Led Zeppelin", "The Beatles"];

// Penamaan method/fungsi dan class/module
function eraseDatabase() {}
function restore_database() {}

class animal {}
class Alpaca {}
```

**Baik**

```
// Penamaan Konstanta
const DAYS_IN_WEEK = 7;
const DAYS_IN_MONTH = 30;

const SONGS = ["Back In Black", "Stairway to Heaven", "Hey
Jude"];
const ARTISTS = ["ACDC", "Led Zeppelin", "The Beatles"];


// Penamaan method/fungsi dan class/module
function eraseDatabase() {}
function restoreDatabase() {}

class Animal {}
class Alpaca {}
```

**Gambar 2.22 Contoh Konsistensi Pada Pemberian Nama**

*2) Indentation*

Penggunaan *indentation* pada penulisan kode dapat mempermudah pengembang dalam membedakan setiap blok logika yang terdapat pada kode tersebut. Contoh petunjuk ini dapat dilihat pada Gambar 2.23.

**Buruk**

```
function getTimeline(a,b){const c=new
Twit({consumer_key:process.env.TWITTER_KEY,consumer_secret:process
.env.TWITTER_SECRET,access_token:a.body.access_token,access_token_
secret:a.body.token_secret});c.get("statuses/home_timeline",(a,c)=
>{a&&console.log(a),b.json({status_code:200,message:"Success get
feed timeline",data:c})})}
```

**Baik**

```
function getTimeline(req, res, next){
    const T = new Twit({
      consumer_key: process.env.TWITTER_KEY,
      consumer_secret: process.env.TWITTER_SECRET,
      access_token: req.body.access_token,
      access_token_secret: req.body.token_secret,
    });
    T.get('statuses/home_timeline', (err, data, response)=>{
        if(err)
          console.log(err)

        res.json({
          status_code : 200,
          message : "Success get feed timeline",
          data : data
        });
    });
}
```

**Gambar 2.23 Contoh *Indentation* Pada Suatu *Method*/fungsi**

3) Penempatan *method*/fungsi yang saling berkaitan

Penempatan *method*/fungsi yang dipanggil dan *method*/fungsi yang memanggil disarankan untuk saling berdekatan agar penelusuran logika dan kesalahan dapat dilakukan dengan mudah. Contoh petunjuk ini dapat dilihat pada Gambar 2.24.

**Buruk**

```
class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }

  lookupPeers() {
```

```
    return db.lookup(this.employee, "peers");
  }

  lookupManager() {
    return db.lookup(this.employee, "manager");
  }

  getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  perfReview() {
    this.getPeerReviews();
    this.getManagerReview();
  }

  getManagerReview() {
    const manager = this.lookupManager();
  }
}

const review = new PerformanceReview(employee);
review.perfReview();
```

**Baik**

```
class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }

  perfReview() {
    this.getPeerReviews();
    this.getManagerReview();
  }

  getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  lookupPeers() {
    return db.lookup(this.employee, "peers");
  }

  getManagerReview() {
    const manager = this.lookupManager();
  }

  lookupManager() {
    return db.lookup(this.employee, "manager");
  }
}

const review = new PerformanceReview(employee);
review.perfReview();
```

**Gambar 2.24 Contoh Penempatan Method/fungsi Yang Baik**

### 2.9.5 *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. Contoh petunjuk ini dapat dilihat pada Gambar 2.25.

```
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);
}
```

**Gambar 2.25 Contoh Penanganan Kesalahan Dengan Notifikasi**

### 2.9.6 *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) Membuat objek memiliki *private member*

Agar struktur data yang digunakan memiliki abstraksi, *class*/modul yang digunakan harus memiliki *private member* di dalamnya. Pada bahasa pemrograman yang digunakan pada penelitian ini, *private member* pada suatu *class*/modul tidak secara eksplisit dapat dilakukan, akan tetapi hal tersebut dapat dilakukan dengan memanfaatkan teknik *Closure.* Contoh petunjuk ini dapat dilihat pada Gambar 2.26.

```
Buruk
const Employee = function(name) {
  this.name = name;
};

Employee.prototype.getName = function getName() {
  return this.name;
};

const employee = new Employee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Nama employee:
John Doe
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Nama employee:
undefined

Baik
function makeEmployee(name) {
  return {
    getName() {
      return name;
    },
  };
}

const employee = makeEmployee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Nama employee:
John Doe
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Nama employee:
John Doe
```

**Gambar 2.26 Contoh Implementasi *Private Member***

*2) 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. Contoh petunjuk ini dapat dilihat pada Gambar 2.27.

```
Buruk
function makeBankAccount() {
  // ...

  return {
    balance: 0,
    // ...
  };
}

const account = makeBankAccount();
account.balance = 100;

Baik
function makeBankAccount() {
  // atribut privat
  let balance = 0;

  // sebuah "getter", dibuat publik dengan
  // memasukkanya ke dalam objek yang akan dikembalikan
  function getBalance() {
    return balance;
  }

  // sebuah "setter", dibuat publik dengan
  // memasukkanya ke dalam objek yang akan dikembalikan
  function setBalance(amount) {
    // ... validasi sebelum memperbarui balance
    balance = amount;
  }

  // Objek yang dikembalikan
  return {
    // ...
    getBalance,
    setBalance,
  };
}

const account = makeBankAccount();
account.setBalance(100);
```

**Gambar 2.27 Contoh Implementasi *Getter* Dan *Setter***

### 2.9.7 *Clean Class*

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

*1) 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. Contoh penggunaan ES2015/ES6 dalam pembuatan *class*/modul dapat dilihat pada Gambar 2.28.

**Bad**

```
const Animal = function(age) {
  if (!(this instanceof Animal)) {
    throw new Error("Instantiate Animal with `new`");
  }

  this.age = age;
};

Animal.prototype.move = function move() {};

const Mammal = function(age, furColor) {
  if (!(this instanceof Mammal)) {
    throw new Error("Instantiate Mammal with `new`");
  }

  Animal.call(this, age);
  this.furColor = furColor;
};

Mammal.prototype = Object.create(Animal.prototype);
Mammal.prototype.constructor = Mammal;
Mammal.prototype.liveBirth = function liveBirth() {};

const Human = function(age, furColor, languageSpoken) {
  if (!(this instanceof Human)) {
    throw new Error("Instantiate Human with `new`");
  }

  Mammal.call(this, age, furColor);
  this.languageSpoken = languageSpoken;
};

Human.prototype = Object.create(Mammal.prototype);
Human.prototype.constructor = Human;
Human.prototype.speak = function speak() {};
```

**Good**

```
class Animal {
  constructor(age) {
    this.age = age;
  }

  move() {
    /* ... */
  }
}

class Mammal extends Animal {
  constructor(age, furColor) {
    super(age);
    this.furColor = furColor;
  }

  liveBirth() {
    /* ... */
  }
}

class Human extends Mammal {
  constructor(age, furColor, languageSpoken) {
    super(age, furColor);
    this.languageSpoken = languageSpoken;
  }

  speak() {
    /* ... */
  }
}
```

**Gambar 2.28 Contoh ES2015/ES6 Dalam Pembuatan *Class*/Modul**

*2) 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. Contoh petunjuk ini dapat dilihat pada Gambar 2.29.

```
Buruk
class UserSettings {
  constructor(user) {
    this.user = user;
  }

  changeSettings(settings) {
    if (this.verifyCredentials()) {
      // ...
    }
  }

  verifyCredentials() {
    // ...
  }
}

Baik
class UserAuth {
  constructor(user) {
    this.user = user;
  }

  verifyCredentials() {
    // ...
  }
}

class UserSettings {
  constructor(user) {
    this.user = user;
    this.auth = new UserAuth(user);
  }

  changeSettings(settings) {
    if (this.auth.verifyCredentials()) {
      // ...
    }
  }
}
```

**Gambar 2.29 Contoh Penggunaan *Single Responsibility Principle***

## 2.10 *Design Pattern*

*Design pattern* merupakan suatu bentuk solusi berupa desain yang digunakan untuk memecahkan permasalahan yang biasa terjadi dalam desain perangkat lunak berorientasi objek, sehingga solusi yang sudah dibuat dapat digunakan kembali untuk memecahkan masalah yang serupa [16]. *Design pattern* tidak berbicara tentang desain seperti struktur data buatan yang dapat diubah menjadi suatu modul *(class)* dan digunakan kembali apa adanya, bukan juga tentang desain khusus suatu domain *(domain-specific design)* untuk sebuah aplikasi secara keseluruhan atau berupa sub sistem saja. Akan tetapi, *design pattern* menerangkan bentuk komunikasi antar objek dan modul *(class)* yang sudah disesuaikan untuk memecahkan suatu masalah desain yang umum dalam suatu konteks tertentu [16].

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* [4] [16]. Adapun kategori dan contoh implementasi dari *design pattern* tersebut dapat dilihat pada Tabel 2.5.

**Tabel 2.5 Kategori *Design Pattern* Dan Contoh Implementasi** [16]

| **Kategori *Design Pattern*** | ***Design Pattern*** | **Kegunaan** |
|---|---|---|
| *Creational* | *Constructor* | *Constructor* digunakan untuk membuat objek dengan tipe tertentu di mana objek yang diinstansiasi dapat digunakan sebagai parameter dan dapat menerima parameter yang digunakan *constructor* untuk menyetel atribut dan *method* saat objek pertama kali dibuat.<br><br>Contoh implementasi :<br>`function Car(model, year, miles) {` |

| Kategori *Design Pattern* | *Design Pattern* | Kegunaan |
|---|---|---|
| | | `this.model = model;`<br>`this.year = year; this.miles = miles;`<br>`this.toString = function () { return this.model + " has done " + this.miles + " miles"; };`<br>`}`<br>`var civic = new Car("Honda Civic", 2009, 20000);`<br>`var 42ondeo = new Car("Ford Mondeo", 2010, 5000);`<br><br>`console.log(civic.toString());`<br>`console.log(42ondeo.toString());` |
| | *Singleton* | *Singleton* digunakan agar penggunaan *class*/modul dapat dilakukan tanpa melakukan instansiasi. *Singleton* akan membuat instansiasi terhadap dirinya sendiri.<br><br>Contoh implementasi:<br>`var Singleton = (function () {`<br>`var instantiated;`<br><br>`function init() {`<br>`return {`<br>`publicMethod: function () {`<br>`console.log('hello world');`<br>`},`<br>`publicProperty: 'test' };  }`<br>`return {`<br>`getInstance: function () {`<br>`if (!instantiated) {`<br>`instantiated = init();`<br>`}`<br>`return instantiated;`<br>`}`<br>`};`<br>`}) ();`<br><br>`// pemanggilan singleton`<br>`Singleton.getInstance().publicMethod();` |
| | *Prototype* | Berdasarkan penjelasan *Gang of Four*, *prototype* digunakan untuk membuat objek dengan cara melakukan kloning terhadap objek yang sudah sebagai *template*. Sehingga objek yang terbentuk akan memiliki struktur data dan nilai yang sama dengan objek yang dikloning. |

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td>Contoh implementasi:<br><pre>// Objek kloning
var someCar = {
  drive: function () {
    /* ... */
  },
  name: 'Mazda 3'
};
// Penggunaan Object.create untuk menghasil objek baru
var anotherCar = Object.create(someCar);

console.log(anotherCar.name); // 'Mazda 3'</pre></td></tr>
<tr><td></td><td>Factory</td><td>Factory digunakan untuk membuat objek di mana sub kelas yang digunakan berhak memutuskan class/modul mana yang akan diinstansiasi. Factory mengatasi permasalahan dari pendefinisian method/fungsi yang terpisah untuk pembuatan objek dan suatu sub kelas sehingga dapat menentukan jenis objek yang dapat diproduksi oleh Factory.<br><br>Contoh implementasi:<br><pre>function VehicleFactory() {}

// Menggunakan objek Cars yang terdapat pada
// contoh Constructor Pattern
VehicleFactory.prototype.vehicleClass = Car;
VehicleFactory.prototype.getVehicle = function (options) {
  return new this.vehicleClass(options);
};

var carFactory = new VehicleFactory();
var car = carFactory.getVehicle({ color: "yellow", turbo: true });

console.log(car instanceof Car); // => true</pre></td></tr>
<tr><td>Structural</td><td>Module</td><td>Module digunakan untuk menyediakan fitur enkapsulasi (publik dan privat) dalam suatu</td></tr>
</table>

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td>class/modul pada rekayasa perangkat lunak konvensional. Pada Javascript sendiri, Module digunakan “meniru” konsep class agar pengembang dapat memasukkan method/fungsi dan variabel yang bersifat privat dan publik ke dalam suatu objek.<br><br>Contoh implementasi:<br><pre>// Pemanfaatan Module Pattern
// Dengan menggunakan pendekatan IIFE (Immediately Invoked Function Expression)
var testModule = (function () {
  var counter = 0;

  return {
    incrementCounter: function () {
      return counter++;
    },
    resetCounter: function () {
      console.log('counter value prior to reset:' + counter);
      counter = 0;
    }
  };
})();

testModule.incrementCounter();
testModule.resetCounter();
console.log(testModule.counter); // Undefined</pre></td></tr>
<tr><td></td><td>Facade</td><td>Facade digunakan untuk menyediakan interface yang lebih tinggi dari suatu kode yang rumit sehingga pengembang cukup menggunakan class/modul tersebut dengan mudah tanpa harus memikirkan kerumitan dibaliknya.<br><br>Contoh implementasi:<br><pre>// Penambah Event-Listener
// untuk Cross-Browser
var addMyEvent = function( el,ev,fn ){
  if(el.addEventListener){
    el.addEventListener( ev,fn, false );
  }
  else if(el.attachEvent){</pre></td></tr>
</table>

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td><pre>
    el.attachEvent( 'on'+ ev, fn );
  }
  else{
    el['on' + ev] = fn;
  }
};
</pre></td></tr>
<tr><td></td><td>Decorator</td><td>Decorator digunakan untuk melakukan modifikasi terhadap sistem yang akan diberi fitur tambahan tanpa mengubah kode yang sudah ada.<br><br>Contoh implementasi:<pre>
var Person =  function( firstName , lastName ){
  this.firstName = firstName;
  this.lastName =  lastName;
  this.gender = 'male'
}

// instansiasi dari 'Person'
var clark = new Person( "Clark" , "Kent" );

// Membuat subclass constructor untuk 'Superhero':
var Superhero = function( firstName, lastName , powers ){

  // Panggil superclass constructor pada objek baru
  //  lalu gunakan .call() untuk memanggil constructor sebagai sebuah method
  // dari object yang diinisialisasi.
  Person.call(this, firstName, lastName);

  // Terakhir, masukkan powers, sebuah array dari kumpulan sifat
  // Tidak tersedia di 'Person'
  this.powers = powers;
}

SuperHero.prototype = Object.create(Person.prototype);
var superman = new Superhero( "Clark" ,"Kent" , ['flight','heat-vision'] );

console.log(superman); // menampilkan propertis dari superhero beserta gendernya
</pre></td></tr>
<tr><td>Behavioral</td><td>Observer</td><td>Observer digunakan untuk mengawasi perubahan nilai atau state pada suatu objek. Observer memperbolehkan suatu objek (subscriber) untuk mengamati objek lainnya (publisher) dengan cara</td></tr>
</table>

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td>mendaftarkan diri sebagai pengamat terhadap publisher sehingga apabila nanti publisher mengalami perubahan, publisher akan mengirimkan pemberitahuan kepada pengamat yang terdaftar.<br><br>Contoh implementasi dengan memanfaatkan JQuery:<br><pre>// Publish
// jQuery: $(obj).trigger(“channel”, [arg1, arg2, arg3]);
$(el).trigger(“/login”, [{username:”test”, userData:”test”}]);

// Subscribe
// jQuery: $(obj).on(“channel”, [data], fn);
$(el).on(“/login”, function(event){
  /* ... */
});

// Unsubscribe
// jQuery: $(obj).off(“channel”);
$(el).off(“/login”);</pre></td></tr>
<tr><td></td><td>Mediator</td><td>Mediator digunakan untuk menjembatani komunikasi antar objek yang memiliki hubungan secara langsung. Mediator menawarkan loose coupling antar objek yang berhubungan dengan cara mengendalikan interaksi dari relasi tersebut. Mediator merupakan shared object pada Observer.<br><br>Contoh implementasi:<br><pre>var mediator = (function(){
  // Storage for our topics/events
  var channels = {};

  // Subscribe to an event, supply a callback to be executed
  // when that event is broadcast
  var subscribe = function(channel, fn){
    if (!channels[channel]) channels[channel] = [];
      channels[channel].push({ context: this, callback: fn });
      return this;
    };</pre></td></tr>
</table>

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td>

```
    // Publish/broadcast an event to the rest of the
application
    var publish = function(channel){
      if (!channels[channel]) return false;
      var                    args                    =
Array.prototype.slice.call(arguments, 1);
      for (var I = 0, l = channels[channel].length;
I < l; i++) {
        var subscription = channels[channel][i];

subscription.callback.apply(subscription.context,
args);
      }

      return this;
    };

    return {
      publish: publish,
      subscribe: subscribe,
      installTo: function(obj){
        obj.subscribe = subscribe;
        obj.publish = publish;
      }
    };
}());
```

```
(function( m ){

  // Set a default value for 'person'
  var person = "Luke";

  // Subscribe to a topic/event called 'nameChange'
with
  // a callback function which will log the original
  // person's name and (if everything works) the
incoming
  // name
  m.subscribe('nameChange', function( arg ){
    console.log( person ); // Luke
    person = arg;
    console.log( person ); // David
  });

  // Publish the 'nameChange' topic/event with the
new data
  m.publish( 'nameChange', 'David' );

})( mediator )
```

</td></tr>
<tr><td></td><td>Command</td><td>Command digunakan untuk melakukan enkapsulasi terhadap pemanggilan method, request, atau operasi ke dalam satu objek tunggal sehingga pengembang dapat</td></tr>
</table>

<table>
<tr><th>Kategori Design Pattern</th><th>Design Pattern</th><th>Kegunaan</th></tr>
<tr><td></td><td></td><td>melakukan parameterisasi dan meneruskan pemanggilan method agar dapat dieksekusi kapan saja. Command memungkinkan untuk memisahkan objek yang memanggil suatu aksi dari objek yang mengimplementasikannya.<br><br>Contoh implementasi:<br><pre>(function(){
  var CarManager = {
    // request information
    requestInfo: function( model, id ){
      return 'The information for ' + model + ' with ID ' + id + ' is foobar';
    },

    // purchase the car
    buyVehicle: function( model, id ){
      return 'You have successfully purchased Item ' + id + ', a ' + model;
    },

    // arrange a viewing
    arrangeViewing: function( model, id ){
      return 'You have successfully booked a viewing of ' + model + ' ( ' + id + ' ) ';
    }
  };
})();</pre><br><pre>// Command Pattern
CarManager.execute = function (name) {
  return CarManager[name] &&
CarManager[name].apply(CarManager,
[].slice.call(arguments, 1));
};

// Testing
CarManager.execute("arrangeViewing", "Ferrari",
"14523");
CarManager.execute("requestInfo", "Ford Mondeo",
"54323");
CarManager.execute("requestInfo", "Ford Escort",
"34232");
CarManager.execute("buyVehicle", "Ford Escort",
"34232");</pre></td></tr>
<tr><td></td><td>Flyweight</td><td>Flyweight digunakan untuk optimasi pada data layer, di mana pada Flyweight terdapat konsep informasi,</td></tr>
</table>

| **Kategori *Design Pattern*** | ***Design Pattern*** | **Kegunaan** |
|---|---|---|
| | | yaitu intrinsik dan ekstrinsik. Informasi intrinsik merupakan informasi yang dibutuhkan oleh *method*/fungsi internal dari objek tempat informasi itu berada, dan apabila tidak tersedia akan menyebabkan *method*/fungsi tidak dapat berfungsi. Sedangkan informasi ekstrinsik merupakan informasi yang dapat dihapus maupun disimpan secara eksternal tidak mempengaruhi kinerja objek tempat informasi intrinsik berada.<br><br>Contoh implementasi:<br>`// Not optimized`<br>`var Book = function( id, title, author, genre, pageCount,publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate, availability){`<br>`  this.id = id;`<br>`  this.title = title;`<br>`  this.author = author;`<br>`  this.genre = genre;`<br>`  this.pageCount = pageCount;`<br>`  this.publisherID = publisherID;`<br>`  this.ISBN = ISBN;`<br>`  this.checkoutDate = checkoutDate;`<br>`  this.checkoutMember = checkoutMember;`<br>`  this.dueReturnDate = dueReturnDate;`<br>`  this.availability = availability;`<br>`}`<br><br>`Book.prototype = {`<br>`  getTitle:function(){`<br>`  return this.title;`<br>`  },`<br>`  getAuthor: function(){`<br>`  return this.author;`<br>`  },`<br>`  getISBN: function(){`<br>`  return this.ISBN;`<br>`  },`<br><br>`  // other getters not shown for brevity`<br>`  updateCheckoutStatus: function(bookID, newStatus, checkoutDate,checkoutMember, newReturnDate){`<br>`  this.id = bookID;`<br>`  this.availability = newStatus;`<br>`  this.checkoutDate = checkoutDate;`<br>`  this.checkoutMember = checkoutMember;`<br>`  this.dueReturnDate = newReturnDate;` |

| Kategori *Design Pattern* | *Design Pattern* | Kegunaan |
|---|---|---|
| | | `},`<br>`extendCheckoutPeriod: function(bookID, newReturnDate){`<br>`this.id = bookID;`<br>`this.dueReturnDate = newReturnDate;`<br>`},`<br>`isPastDue: function(bookID){`<br>`var currentDate = new Date();`<br>`return currentDate.getTime() > Date.parse(this.dueReturnDate);`<br>`}`<br>`};` |
| | | `// flyweight optimized version`<br>`var Book = function (title, author, genre, pageCount, publisherID, ISBN) {`<br>`this.title = title;`<br>`this.author = author;`<br>`this.genre = genre;`<br>`this.pageCount = pageCount;`<br>`this.publisherID = publisherID;`<br>`this.ISBN = ISBN;`<br>`};` |
| | | `// Book Factory singleton`<br>`var BookFactory = (function () {`<br>`var existingBooks = {};`<br><br>`return {`<br>`createBook: function (title, author, genre, pageCount, publisherID, ISBN) {`<br><br>`// Find out if a particular book meta-data combination has been created before`<br>`var existingBook = existingBooks[ISBN];`<br>`if (existingBook) {`<br>`return existingBook;`<br>`}`<br>`else {`<br>`// if not, let's create a new instance of it and store it`<br>`var book = new Book(title, author, genre, pageCount, publisherID, ISBN);`<br>`existingBooks[ISBN] = book;`<br>`return book;`<br>`}`<br>`}`<br>`}`<br>`});` |
| | | `// BookRecordManager singleton`<br>`var BookRecordManager = (function () {`<br>`var bookRecordDatabase = {};`<br>`return {`<br><br>`// add a new book into the library system`<br>`addBookRecord: function (id, title, author,` |

| Kategori *Design Pattern* | *Design Pattern* | Kegunaan |
|---|---|---|
| | | genre, pageCount, publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate, availability){<br>var book = bookFactory.createBook(title, author, genre, pageCount, publisherID, ISBN);<br>bookRecordDatabase[id] = {<br>checkoutMember: checkoutMember,<br>checkoutDate: checkoutDate,<br>dueReturnDate: dueReturnDate,<br>availability: availability,<br>book: book<br>}<br>},<br>updateCheckoutStatus: function (bookID, newStatus, checkoutDate, checkoutMember, newReturnDate) {<br>var record = bookRecordDatabase[bookID];<br>record.availability = newStatus;<br>record.checkoutDate = checkoutDate;<br>record.checkoutMember = checkoutMember;<br>record.dueReturnDate = newReturnDate;<br>},<br>extendCheckoutPeriod: function (bookID, newReturnDate) {<br>bookRecordDatabase[bookID].dueReturnDate = newReturnDate;<br>},<br>isPastDue: function (bookID) {<br>var currentDate = new Date();<br>return currentDate.getTime() > Date.parse(bookRecordDatabase[bookID].dueReturnDate);<br>}<br>};<br>}) |

## 2.11 Analisis dan Desain 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 [17].

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 [17].

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*) [17].

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*) [17].

Dokumentasi UML menyediakan 10 macam diagram 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 [18].

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 [18].

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* [18].

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 [18].

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 [18].

## 2.12 *Javascript*

*Javascript* merupakan bahasa pemrograman tingkat tinggi dan dinamis yang digunakan untuk membangun suatu perangkat lunak berbasis web [19]. *Javascript* merupakan salah satu bahasa pemrograman yang memanfaatkan interpreter sebagai mesin untuk menjalankan perintahnya sehingga dapat secara langsung dijalankan tanpa melalui proses kompilasi [19]. Pada umumnya, *Javascript* hanya dijalankan pada *client-side* (Peramban web) saja, akan tetapi dengan menggunakan *NodeJS*, *Javascript* juga dapat dijalankan pada *server-side* [19].

*Javascript* menggunakan banyak paradigma dalam penggunaannya. *Javascript* dapat diimplementasikan dengan pendekatan fungsional maupun berorientasi objek (berbasis prototipe) [19]. Sehingga *Javascript* dapat diimplementasikan dengan berbagai cara. Adapun pada penelitian, paradigma

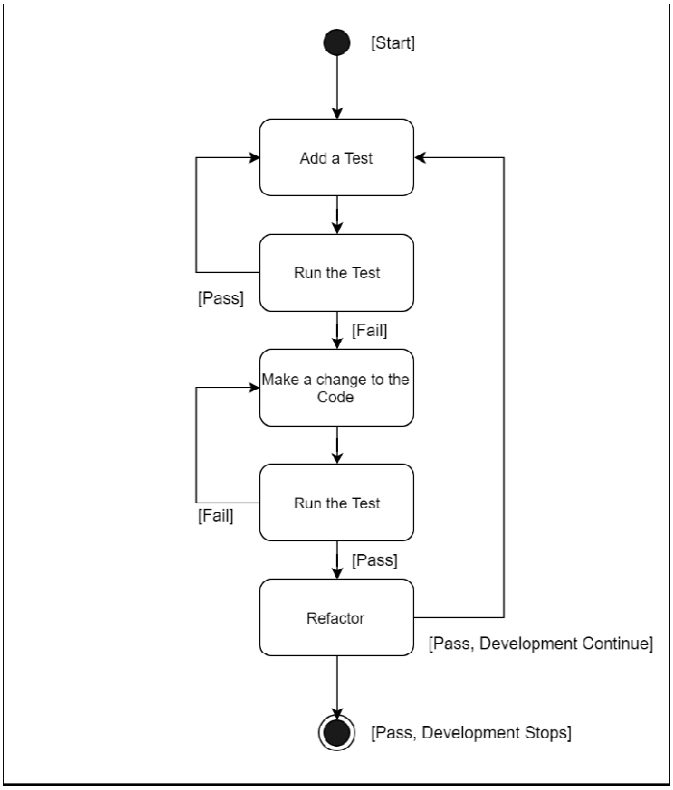
yang digunakan yaitu paradigma berorientasi objek (berbasis prototipe) yang disesuaikan dengan bahasa pemrograman *Javascript*.

## 2.13 *Test-Driven Development*

*Test-Driven Development (TDD)* merupakan suatu pendekatan dalam membangun perangkat lunak yang diawali dengan penulisan suatu *Unit Test* terlebih dahulu sebelum menuliskan kode dari perangkat lunak yang akan dibangun [20]. *Unit Test* berisi skenario pengujian untuk *class* atau modul yang akan dibuat, dan tidak bergantung terhadap *Unit Test* yang lain. Contoh *Unit Test* dapat dilihat pada Gambar 2.30. Adapun proses yang dilakukan pada *Test-Driven Development* dapat dilihat pada Gambar 2.31.

```
var assert = require('assert');
var loginController =  require('../../controllers/login.controller');

describe('LoginController', function () {

  describe('isValidUserId', function(){

    it('should return true if valid user id', function(){
      var isValid = loginController.isValidUserId(['abc123','xyz321'],
'abc123')
      assert.equal(isValid, true);
    });

    it('should return false if invalid user id', function(){
      var isValid =
loginController.isValidUserId(['abc123','xyz321'],'abc1234')
      assert.equal(isValid, false);
    });

  });

});
```

**Gambar 2.30 Contoh *Unit Test***




**Gambar 2.31** ***Test-Driven Development Process*** [20]

*1. Add a Test*

Penggunaan *Test-Driven Development* diawali dengan membuat suatu *Unit Test* yang sederhana dan fokus terhadap perilaku dari suatu method/fungsi ,serta berisi skenario pengujian dari suatu *method*/fungsi. *Unit Test* hanya berisi pemanggilan dari suatu *method*/fungsi yang dibangun, tidak berisi logika yang dapat mengubah fungsional dari *method*/fungsi tersebut. *Unit Test* yang baru tidak boleh memiliki tugas *(assertion)* yang sama dengan yang sudah berjalan sebelumnya.

*2. Run the Tests*

Setelah *Unit Test* dibuat, *Unit Test* dijalankan untuk memastikan skenario pengujian dapat berjalan dengan benar. Dalam pertama kali menjalankan suatu *Unit Test*, *Unit Test* harus menghasilkan kesalahan dari skenario pengujian yang

sudah disiapkan. Apabila *Unit Test* tidak menghasilkan kesalahan, maka *Unit Test* harus dibuat ulang. Karena *Unit Test* sendiri dibuat untuk memastikan *method*/fungsi yang diuji dapat berjalan dengan berbagai skenario yang sudah disiapkan.

*3. Make a Change to the Code*

Setelah *Unit Test* sudah dijalankan dan menghasilkan kesalahan tertentu berdasarkan skenario yang ada, selanjutnya pengembang melakukan perubahan sederhana terhadap kode dari *method*/fungsi yang diuji agar dapat berhasil saat diuji kembali. Meskipun hanya bersifat *hard-code* ataupun berantakan, yang terpenting dapat membuat skenario pengujian yang dibuat terpenuhi.

*4. Run the Tests*

Setelah method/fungsi sudah dimodifikasi sedemikian cara agar dapat memenuhi skenario yang ada, *Unit Test* yang berkaitan dijalankan untuk memastikan *method*/fungsi yang sudah dimodifikasi dapat memenuhi skenario tersebut. Jika hasil pengujian berhasil, maka lakukan *refactoring* terhadap *method*/fungsi yang diuji. Apabila tidak, lakukan perubahan terhadap *method*/fungsi dan jalankan *Unit Test* kembali.

*5. Refactoring*

Apabila seluruh skenario pengujian telah berhasil dilakukan, langkah selanjutnya yaitu melakukan improvisasi terhadap kode *method*/fungsi yang diuji, seperti penghapusan duplikasi kode, perbaikan struktur kode, dan membuat kode lebih bersih *(Clean Code).*

*6. Development Stops and Continue*

Apabila *refactoring* selesai, dan sudah tidak terdapat duplikasi atau sudah tidak terdapat improvisasi yang harus diterapkan lagi, menandakan proses sudah selesai. Adapun pengembangan dilanjutkan dengan melakukan tugas baru, seperti membuat *Unit Test* dengan skenario yang berbeda atau menguji modul yang lain

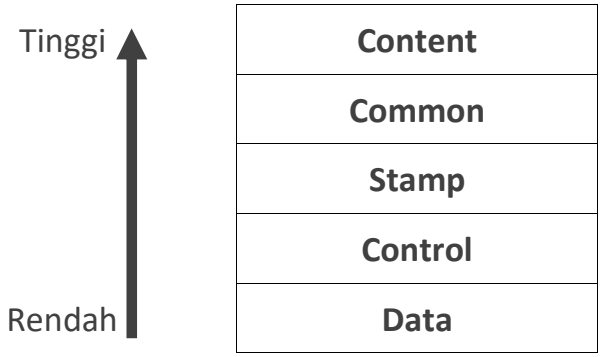
dengan melakukan proses *Test-Driven Development* kembali sampai kebutuhan pengembangan terpenuhi.

## 2.14 *Coupling* dan *Cohesion*

*Coupling* dan *Cohesion* merupakan suatu istilah yang digunakan untuk menilai kegunaan dan ukuran sebuah modul dari suatu sistem berorientasi objek [21]. *Coupling* menilai ketergantungan dari sebuah *class*/modul terhadap *class*/modul lainnya. Sedangkan *Cohesion* menilai keterikatan antar variabel, serta *method*/fungsi yang ada pada *class*/modul. Secara umum, sistem yang memiliki desain yang baik adalah sistem yang memiliki nilai *coupling* yang rendah dan *cohesion* yang tinggi. Hal tersebut menandakan sistem yang dimaksud memiliki *modularity* yang tinggi dan independen.

### 2.14.1 *Coupling*

*Coupling* dapat didefinisikan sebagai tingkat saling ketergantungan yang ada antara *class*/modul dan seberapa dekat mereka saling terhubung satu sama lain. Dalam arti lain, *coupling* menunjukkan seberapa kuat keterkaitan antar *class*/modul. Jadi, saat *coupling* tinggi, artinya *class*/modul tersebut saling bergantung, yang menunjukkan *class*/modul tidak dapat berfungsi tanpa *class*/modul yang lain. Sehingga semakin rendah *coupling*, desain dari perangkat lunak semakin baik. *Coupling* terbagi ke dalam beberapa tipe, pembagian tersebut dapat dilihat pada Gambar 2.32.




Gambar 2.32 Pembagian Tipe *Coupling*

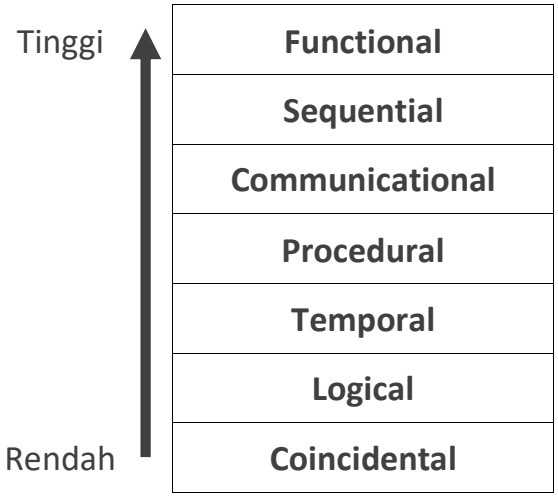
Adapun penjelasan dari Gambar 2.32 dapat dilihat pada Tabel 2.6.

**Tabel 2.6 Tipe *Coupling***

| No | Tipe *Coupling* | Keterangan |
|---|---|---|
| 1 | *Content* | Tipe ini merupakan tipe *coupling* di mana terdapat suatu *class*/modul tertentu yang dapat mengakses dan memodifikasi isi konten dari class/modul lain. |
| 2 | *Common* | Tipe ini merupakan tipe *coupling* di mana terdapat banyak *class*/modul yang dapat melakukan mutasi terhadap suatu variabel global yang sama (*Shared Global Object*). |
| 3 | *Stamp* | Tipe ini merupakan tipe *coupling* di mana terdapat penggunaan suatu struktur data (*Model Class*) yang digunakan untuk mengirimkan informasi kepada komponen lain yang berada pada sistem. |
| 4 | *Control* | Tipe ini merupakan tipe *coupling* di mana terdapat suatu *class*/modul yang dapat mengubah alur eksekusi dari *class*/modul yang lain. |
| 5 | *Data* | Tipe ini merupakan tipe *coupling* di mana *class*/modul saling berinteraksi dengan menukar data sebagai parameter. |

### 2.14.2 *Cohesion*

*Cohesion* menandakan tingkat ketergantungan antar elemen pada suatu *class*/modul. Sehingga *cohesion*, dengan kata lain, digunakan untuk mengukur tanggung jawab sebuah *class*/modul sebagai unit yang bermakna. *Cohesion* terbagi menjadi beberapa tipe yang dapat dilihat pada Gambar 2.33.




**Gambar 2.33 Pembagian Tipe *Cohesion***

Adapun penjelasan dari Gambar 2.33 dapat dilihat pada Tabel 2.7.

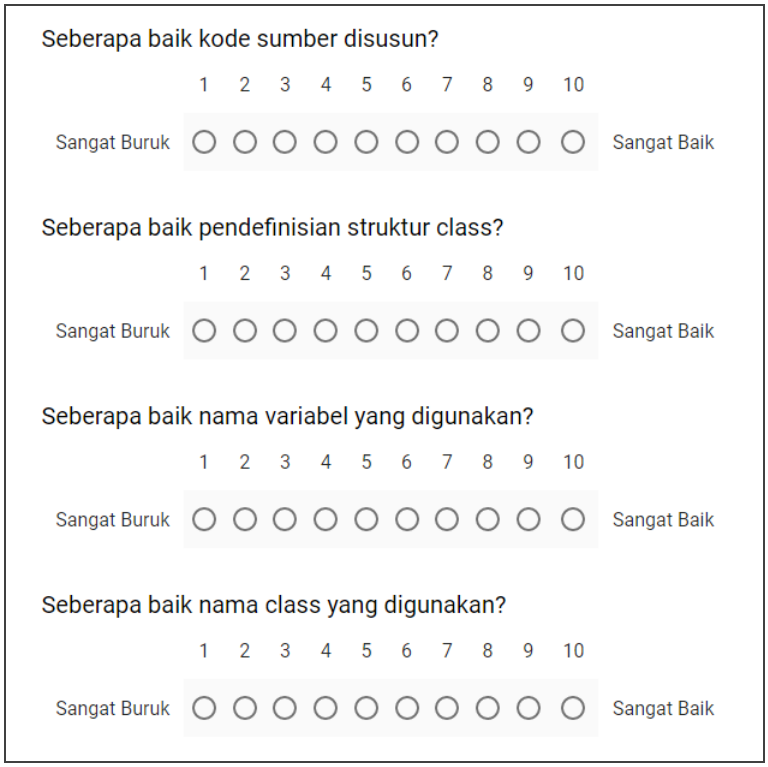
**Tabel 2.7 Tipe *Cohesion***

| No | Tipe *Cohesion* | Keterangan |
|---|---|---|
| 1 | *Functional* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul secara fungsional dikelompokkan menjadi unik logis dan bekerja sama sebagai unit logis, sehingga memiliki fleksibilitas dan *reusability* yang tinggi. |
| 2 | *Sequential* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul dikelompokkan sedemikian rupa sehingga *output* dari kelompok elemen tersebut menjadi *input* untuk kelompok lainnya yang dijalankan secara berurutan. |
| 3 | *Communicational* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul secara logis dikelompokkan |

| No | Tipe *Cohesion* | Keterangan |
|---|---|---|
| | | berdasarkan cara elemen tersebut digunakan secara berurutan dan data yang mereka olah. |
| 4 | *Procedural* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul dikelompokkan sedemikian rupa agar kelompok elemen tersebut dapat dijalankan secara berurutan. |
| 5 | *Temporal* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul dikelompokkan sedemikian rupa agar elemen yang ada di dalamnya dapat dieksekusi secara bersamaan pada suatu waktu. Contohnya kelompok yang berisi inisialisasi sekumpulan objek. |
| 6 | *Logical* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul yang terkait secara logis ditempatkan di komponen yang sama. |
| 7 | *Coincidental* | Tipe ini merupakan tipe *cohesion* di mana elemen pada sebuah *class*/modul ditempatkan acak akibat dari proses pemecahan *class*/modul menjadi *class*/modul yang lebih kecil. |

## 2.15 *Human-Assessment Software Maintainability*

Penilaian *maintainability* perangkat lunak oleh manusia atau secara manual dilakukan untuk menilai *maintainability* dari sisi pengembang yang bersangkutan dengan menilai kesulitan yang dialami pengembang dalam memelihara perangkat lunak. Penilaian dilakukan dengan memberikan sejumlah tugas kepada pengembang untuk melakukan pemeliharaan terhadap perangkat lunak yang akan dinilai. Tugas yang diberikan berupa tugas untuk melakukan pemeliharaan, seperti melakukan modifikasi, penambahan fungsional, maupun perbaikan *bug* terhadap perangkat lunak [22]. Setelah tugas yang diberikan selesai untuk dikerjakan, pengembang akan menilai tingkat kesulitan dari setiap tugas yang diberikan dengan rentang penilaian 1 sampai 5, di mana 1 menandakan tugas sangat mudah untuk dilakukan dan 5 menandakan tugas sangat sulit untuk dilakukan. Selain itu, pengembang juga diminta mengisi kuesioner yang berisi pertanyaan tentang *maintainability* pada perangkat lunak yang dinilai. Pengembang mengisi pertanyaan tersebut dengan memberikan nilai dengan rentang 1 sampai 10, di mana 1 yang berarti sangat buruk, dan 10 yang berarti sangat baik. Pertanyaan-pertanyaan yang digunakan dalam kuesioner diambil dari COCOMO II [22]. Adapun contoh pertanyaan dapat dilihat pada Gambar 2.34.




**Gambar 2.34 Contoh Pertanyaan *Human-Assessment Software Maintainability***