BAB II
TINJAUAN PUSTAKA

II.1.Perpustakaan Digital

Dengan berkembangnya teknologi informasi saat ini, berimbas pada munculnya perpustakaan yang bisa diakses melalui internet yang biasa disebut sebagai perpustakaan digital. Namun sebelum mempelajari lebih lanjut mengenai perpustakaan digital, kita mesti mengerti dulu apa itu perpustakaan.

II.1.1. Pengertian Perpustakaan

Perpustakaan umumnya dikenal sebagai tempat di mana buku-buku disimpan, memuat informasi, data dan hiburan yang bisa didapatkan pembaca. Namun kadang kala perpustakaan juga menyimpan informasi dalam bentuk lain, seperti rekaman suara dan video untuk kepentingan dokumentasi. Dengan adanya perpustakaan ini, para pembaca bisa membaca buku yang diperlukan tanpa perlu membelinya, dan bisa meminjamnya dengan persyaratan yang berlaku di perpustakaan tersebut. Dengan begini, semua pembaca termasuk pembaca yang tak mampu untuk membeli buku pun bisa mendapatkan informasi ilmu pengetahuan dengan gratis.

Masuknya era modern, ada istilah baru yang disebut dengan perpustakaan modern, yaitu suatu tempat untuk mengakses informasi dalam format apa pun, baik informasi itu disimpan dalam buku maupun bentuk lainnya, contohnya artikel jurnal yang tersimpan di server internet. Karena itu, perpustakaan modern ini, selain memuat informasi dalam bentuk cetak, juga ada yang memuatnya dalam bentuk digital.

II.1.2. Pengertian Perpustakaan Digital

Perpustakaan digital menerapkan teknologi informasi dan komunikasi untuk menyimpan, mendapatkan dan menyebarluaskan informasi-informasi ilmu pengetahuan dalam format digital [2]. Berbeda dengan perpustakaan konvensional yang mengoleksi informasi-informasi dalam bentuk cetak pada sebuah buku, perpustakaan digital identik dengan penyimpanan informasi-informasinya dalam bentuk digital dan berada pada suatu server internet. Dengan begitu, pembaca bisa langsung membaca buku, mendengar arsip suara atau melihat arsip video langsung

yang tersedia di perpustakaan digital kapan dan di mana saja selama bisa terkoneksi dengan perpustakaan digital tersebut melalui internet. Berbeda dengan perpustakaan konvensional yang biasanya harus dikunjungi di satu tempat dan waktu-waktu tertentu sesuai dengan ketentuan yang diberlakukan perpustakaan terkait [2].

Perpustakaan digital ini sendiri tidak bisa berdiri sendiri, artinya perpustakaan digital ini terikat pada sumber-sumber lain dan biasanya terbuka untuk semua pembaca di seluruh dunia. Koleksi perpustakaan digital tidaklah terbatas pada dokumen yang menggantikan bentuk cetak saja, malah ruang lingkupnya sampai pada artefak digital yang tidak bisa digantikan dalam bentuk tercetak. Perpustakaan ini melayani berupa manajer informasi dan pemakaian informasi [2].

II.2.OOP (Object Oriented Programming)

Sebagai sebuah pola dalam pemrograman komputer, OOP sangat berguna dalam pemodelan desain dan pengembangan objek. Pola ini menekankan pada pemodelan objek serta keterhubungan objek-objek yang saling terkait. OOP terdiri dari *class-class* yang dibutuhkan untuk membuat suatu objek sekaligus memanipulasinya [3]. *Class-class* yang ada memiliki peranannya masing-masing sesuai dengan apa yang sudah ditetapkan. Bisa dikatakan class adalah sebuah *blueprint* untuk mendefinisikan objek. Objek ini memiliki *variable*, *prototype*, dan *method* [3].

OOP pada Javascript memiliki beberapa konsep yang sangat berguna di dalamnya, yaitu sebagai berikut.

II.2.1. Constructor

Ada banyak cara untuk menciptakan sebuah objek. Salah satunya adalah dengan *constructor* [4]. *Constructor* adalah suatu fungsi atau *method* yang pertama kali dipanggil oleh *class* saat sebuah objek diinstansiasi/dibuat.

```
1.  class Orang {
2.      constructor(nama, umur) {
3.          this.nama = nama;
4.          this.umur = umur;
5.      }
6.   //..
```

```
}

let raka = new Orang("Raka", 21);
```

Pada contoh di atas, *constructor* memiliki dua buah argumen, yaitu nama dan umur yang kemudian dipakai pada proses yang terjadi pada *constructor*. *Constructor* akan pertama kali dijalankan saat perintah *new* terhadap *class* dijalankan.

II.2.2. Property

Property menjelaskan suatu objek. *Property* bisa dikatakan sebagai *variable* dalam objek, yang membawa suatu nilai tertentu. Seperti contoh di atas, sebuah *class* Orang memiliki sebuah *property* nama dan umur yang ditandai dengan *this*, yang didapat dari *method constructor*.

II.2.3. Method

Method adalah fungsi yang paling sering ditemui di bahasa pemrograman berkonsep OOP, yang menjelaskan tingkah laku atau pekerjaan yang dilakukan oleh objek. *Method* bisa dikatakan sebagai kata kerja, yang mana tindakan tertentu yang dilakukan oleh objek [4]. *Method* digunakan untuk melakukan kegiatan-kegiatan yang ada pada objek, seperti mengubah informasi, mendapatkan informasi dari objek itu sendiri atau objek lain, dan menampilkan suatu data dan informasi dari objek.

```
class Orang {
    constructor(nama, umur) {
        this.nama = nama;
        this.umur = umur;
    }

    tidur(jam) {
        return `${this.nama} tidur selama ${jam} jam.`;
    }

    makan(porsi) {
        return `${this.nama} makan sebanyak ${porsi} porsi.`;
    }
}

let raka = new Orang("Raka", 21);

console.log(raka.tidur(12));
```

Gambar II-1 Output Method

Seperti contoh di atas, *class* Orang memiliki dua *method*, yaitu tidur dan makan. Saat objek dengan *method* tidur dipanggil, *method* tidur tersebut akan menampilkan sebuah data berupa *string* seperti pada Gambar II-1, sesuai dengan isi perintah yang ada pada *method* tersebut.

II.2.4. Inheritence

Salah satu konsep yang melibatkan dua *class* atau lebih, bertujuan untuk menggunakan sumber kode yang berulang tanpa membuat sumber kode yang baru secara berulang. Dalam hal ini, pembuatan *class* yang baru dilakukan terhadap *class* yang sudah ada, *class* baru ini disebut dengan *subclass*, sementara *class* yang sudah ada disebut dengan *parent class* atau *super class*. Tujuan konsep ini guna pada suatu ketika ada perbaikan atau pembaharuan sumber kode *parent class* ke sebuah *subclass*. Dengan begitu, programmer tidak perlu repot-repot memperbaiki atau memperbarui *subclass* jika perbaikan atau pembaruan itu terjadi hanya pada *parent class* saja [3].

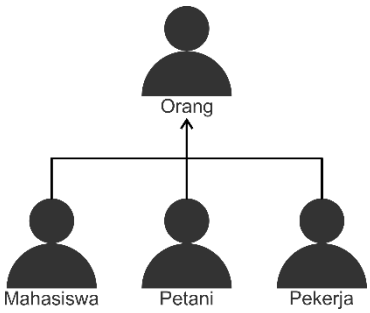
```
1.  class Orang {
2.      constructor(nama, umur) {
3.          this.nama = nama;
4.          this.umur = umur;
5.      }
6.
7.      tidur(jam) {
8.          return `${this.nama} tidur selama ${jam} jam.`;
9.      }
10.
11.     makan(porsi) {
12.         return `${this.nama} makan sebanyak ${porsi} porsi.`;
13.     }
14. }
15.
16. class Mahasiswa extends Orang {
17.     kuliah(jam) {
18.         return `${this.nama} kuliah selama ${jam} jam.`
19.     }
20. }
21.
22. class Petani extends Orang {
23.     bertani(jam) {
24.         return `${this.nama} bertani selama ${jam} jam.`
25.     }
26. }
```

```
27.
28. class Pekerja extends Orang {
29.     bekerja(jam) {
30.         return `${this.nama} bekerja selama ${jam} jam.`
31.     }
32. }
33.
34. let raka = new Mahasiswa("Raka", 21);
35. let yanto = new Petani("Yanto", 22);
36. let juhri = new Pekerja("Juhri", 23);
37.
38. console.log(raka.tidur(12));
39. console.log(raka.kuliah(4.5));
40. console.log(yanto.bertani(5));
41. console.log(juhri.bekerja(6));
```

```
Raka tidur selama 12 jam.                    index.js:38
Raka kuliah selama 4.5 jam.                  index.js:39
Yanto bertani selama 5 jam.                  index.js:40
Juhri bekerja selama 6 jam.                  index.js:41
>
```

Gambar II-2 Output Inheritence

Semisalnya *class* Orang sebagai *parent class*, lalu *class* Mahasiswa, Petani dan Pekerja sebagai *subclass*. Semua *property* dan *method* yang ada pada *parent class* akan diwariskan kepada *subclass* karena *class* Orang sudah ditetapkan sebagai *parent class* dari beberapa *subclass* yang ada. Berikut Gambar II-3 menggambarkan hubungan *inheritance*.



Gambar II-3 Ilustrasi Inheritance

II.2.5. Polymorphism

Bisa dikatakan *polymorphism* adalah suatu *property* yang memiliki berbagai bentuk. Dalam bahasa pemrograman dan khususnya pada OOP, *polymorphism* adalah suatu usaha untuk membentuk suatu bentuk fungsi lain dari fungsi yang sudah ada, dan konsepnya hampir sama dengan *inheritence* yang mana memiliki *parent class* dan *subclass*. Dengan adanya *polymorphism* ini, aplikasi yang

dibangun akan bisa berukuran lebih kecil. Programmer jadi tidak perlu memeriksa objek satu per satu untuk menentukan tipe dan metode apa yang sesuai. *Polymorphism* membebaskan programmer dari ketidaknyamanan ini dan menawarkannya fleksibilitas dalam membangun perangkat lunak [3]. Sifat paling mencolok dari *polymorphism* adalah *overriding* dan *overloading*.

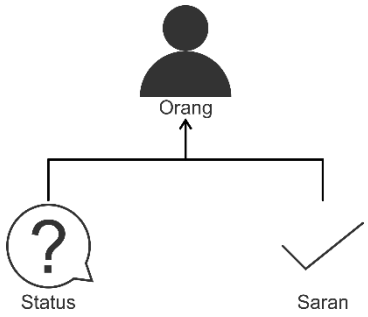
```
1.  class Orang {
2.      constructor(nama, umur) {
3.          this.nama = nama;
4.          this.umur = umur;
5.      }
6.
7.      tidur(jam) {
8.          return `${this.nama} tidur selama ${jam} jam.`;
9.      }
10.
11.     makan(porsi) {
12.         return `${this.nama} makan sebanyak ${porsi} porsi.`;
13.     }
14. }
15.
16. class statusOrang extends Orang {
17.     tidur(jam) {
18.         if(jam >= 7 && jam <= 8) {return `${this.nama} cukup tidur.`}
19.         else if(jam < 7) {return `${this.nama} kurang tidur.`}
20.         else if(jam > 8) {return `${this.nama} kelebihan tidur.`}
21.     }
22.
23.     makan(porsi) {
24.         if(porsi === 1) {return `${this.nama} cukup makan.`}
25.         else if(jam < 1) {return `${this.nama} kurang makan.`}
26.         else if(jam > 1) {return `${this.nama} kelebihan makan.`}
27.     }
28. }
29.
30. class saranOrang extends Orang {
31.     tidur(jam) {
32.        if(jam >= 7 && jam <= 8) {return `${this.nama} sudah memenuhi waktu
    tidur.`}
33.         else if(jam < 7) {return `${this.nama} harus tidur lebih.`}
34.         else if(jam > 8) {return `${this.nama} harus kurangi tidur.`}
35.     }
36.
37.     makan(porsi) {
38.         if(porsi === 1) {return `${this.nama} sudah memenuhi porsi makan.`}

39.         else if(jam < 1) {return `${this.nama} harus makan lebih.`}
40.         else if(jam > 1) {return `${this.nama} harus kurangi makan.`}
41.     }
42. }
43.
44. let raka = new Orang("Raka", 21);
45. let statusRaka = new statusOrang("Raka", 21);
46. let saranRaka = new saranOrang("Raka", 21);
47.
```

```
48. console.log(raka.tidur(12));
49. console.log(statusRaka.tidur(12));
50. console.log(saranRaka.tidur(12));
```

```
Raka tidur selama 12 jam.                    index.js:48
Raka kelebihan tidur.                        index.js:49
Raka harus kurangi tidur.                    index.js:50
>
```

Gambar II-4 Output Polymorphism

Semisal kita memiliki sebuah *parent class* Orang dan beberapa *subclass*. Di mana masing-masing *class* memiliki metode yang berbeda. Saat membuat instansi baru dari *class* Orang yang seperti contoh di atas dibuat pada objek raka, dan kita buat instansi baru lagi dari *subclass*-nya dengan objek statusRaka dan saranRaka dari *class* statusOrang dan saranOrang. Masing-masing *class* memiliki metode yang sama pada metode tidur dan makan namun tugas yang berbeda. Dan saat menginstansiasikan sebuah objek dari *subclass*, selain *method* dari *subclass* itu sendiri yang akan dimiliki, *subclass* juga akan diwarisi metode yang dimiliki *parent class*-nya, sehingga tidak perlu lagi diimplementasikan pada *subclass*-nya. Berikut Gambar II-5 menggambarkan hubungan *polymorphism*.



Gambar II-5 Ilustrasi Polymorphism

II.2.6. Abstraction

Abstraction adalah salah satu konsep yang penting dan sering digunakan pada pembangunan *class library*. *Abstraction* memungkinkan untuk memilih kumpulan besar data untuk hanya menampilkan detail yang relevan dengan objek. *Abstraction* membantu dalam mengurangi kompleksitas dan usaha dalam pemrograman. *Abstract class* pada Javascript harus ditandai dengan `@abstract`.

II.2.7. Encapsulation

Encapsulation memungkinkan untuk melarang pengaksesan data secara langsung, sehingga harus ada objek lain yang bertanggung jawab untuk mengakses data yang dimaksud. *Encapsulation* memungkinkan data yang ada terjaga dan lebih aman.

II.3. Style Programming Javascript

Setiap bahasa pemrograman pasti memiliki *style*/gayanya masing-masing, termasuk juga bahasa Javascript. Dengan *style* programming ini, penulisan kode program bisa lebih konsisten sehingga programmer bisa lebih fokus pada kontennya, karena pada dasarnya *style* penulisan kode setiap programmer memiliki gaya yang berbeda-beda. Berikut ini adalah *style* programming berdasarkan Google Javascript Style yang dirangkum sedemikian rupa untuk mempermudah pembangunan *class library* [5].

II.3.1. Penamaan

Penamaan yang diberikan harus jelas, bermakna dan sesuai alasan. Tidak perlu mengkhawatirkan *space* untuk mengetikkan nama yang panjang, jauh lebih penting untuk bisa memahami kode secepat mungkin dan bisa dipahami oleh pembaca/programmer lain. Jangan menggunakan singkatan yang ambigu atau asing bagi pembaca yang tidak terlibat dengan proyek yang dibangun, dan jangan menyingkat dengan menghapus huruf di bagian kata. Penamaan masing-masing *identifier* bisa dilihat pada Tabel II.1.

Berikut ini contoh penulisan yang benar.

```
1.  errorCount          // Tidak ada singkatan.
2.  dnsConnectionIndex  // Kebanyakan orang tahu kepanjangan dari “DNS”.
3.  referrerUrl         // Pengarah "URL".
4.  customerId          // “Id” selalu dipakai dan tak mungkin disalahpahami.
```

Lalu berikut ini adalah contoh penulisan yang salah.

```
1.  n                   // Tidak memiliki arti.
2.  nErr                // Penyingkatan yang ambigu.
3.  nCompConns          // Penyingkatan yang ambigu juga.
4.  wgcConnections      // Hanya orang terlibat dalam proyek yang mengerti.
5.  pcReader            // Banyak yang bisa disingkat menjadi “pc”.
6.  cstmrId             // Menghapus huruf-huruf.
```

```
7.  kSecondsPerDay      // Jangan menggunakan Hungarian Notation.
```

Tabel II.1 Penamaan Identifier di Javascript

Identifier	Penulisan	Contoh
Package	lowerCamelCase	my.exampleCode.deepSpace
Class	UpperCamelCase	VisibilityMode
Method	lowerCamelCase	sendMessage
Enum	UpperCamelCase	BackgroundColor
Constant	CONSTANT_CASE	MAX_COUNT
Non-constant	lowerCamelCase	computedValues
Parameter	lowerCamelCase	maxId
Local Variable	lowerCamelCase	lastQuery
Template	SATUHURUF	TYPE

II.3.2. Kurung Kurawal

Kurung kurawal diperlukan untuk semua struktur kontrol (contoh *if*, *else*, *for*, *do*, *while*, serta yang lainnya), bahkan jika badan pernyataannya hanya memiliki satu pernyataan. Pernyataan pertama dari blok yang tidak kosong harus dimulai dari barisnya sendiri. Pernyataan sederhana bisa sepenuhnya masuk pada satu baris tanpa pembungkus (di dalam kurung kurawal) dan dapat diletakkan pada satu baris dengan tanpa kurung kurawal. Ini adalah satu-satunya kasus di mana struktur kontrol dapat menghilangkan kurung kurawal dan baris baru. Contoh seperti di bawah ini.

```
if (shortCondition()) foo();
```

Dan berikut ini adalah contoh penulisan yang salah.

```
1.  if (someVeryLongCondition())
2.    doSomething();
3.
4.  if (anotherCondition()) {
5.    doSomething();
6.  }
```

Penulisan kurung kurawal mengikuti *style* Kernighan dan Ritchie (*Egyptian brackets*), yaitu sebagai berikut.

a. Tidak ada garis baru sebelum kurung pembuka.
b. Garis baru setelah kurung kurawal pembuka.
c. Garis baru sebelum kurung kurawal penutup.
d. Garis baru setelah kurung kurawal penutup jika kurung kurawal itu mengakhiri pernyataan, badan *function*, pernyataan *class*, atau *class method*. Tidak ada garis baru setelah kurung kurawal jika diikuti *else*, *catch*, *while*, koma, titik koma, atau tanda kurung kanan.

Contoh:

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo) {
    if (condition(foo)) {
      try {
        // Note: mungkin akan gagal.
        something();
      } catch (err) {
        recover();
      }
    }
  }
}
```

Blok kosong bisa saja langsung ditutup setelah kurung kurawal ditulis dan langsung ditutup dengan kurung kurawal kanan, tanpa karakter, spasi, atau garis baru di dalamnya (yaitu seperti {}), kecuali jika itu adalah bagian dari pernyataan multi-blok (yang secara langsung mengandung beberapa blok: *if*/*else* atau *try*/*catch*/*finally*). Contohnya seperti di bawah ini.

```
function doNothing() {}
```

Lalu berikut ini adalah contoh penulisan yang salah.

```
if (condition) {
  // …
} else if (otherCondition) {} else {
  // …
}

try {
```

```
    // …
} catch (e) {}
```

II.3.3. Indentasi

Setiap kali blok baru dibuat, pemberian indentasi diberikan dengan dua spasi. Ketika blok berakhir, indentasi kembali ke level indentasi sebelumnya. Indentasi berlaku untuk kode dan komentar di seluruh blok.

Setiap literal array dapat secara opsional diformat seolah-olah itu adalah blok, atau bisa juga dengan membungkus baris/line-wrapping. Line-wrapping bisa dilihat pada II.3.5. Sebagai contoh, dua contoh di bawah ini adalah contoh yang valid.

```
const a = [
  0,
  1,
  2,
];

const b =
    [0, 1, 2];
```

```
const c = [0, 1, 2];

someMethod(foo, [
  0, 1, 2,
], bar);
```

Setiap objek literal secara opsional dapat dibentuk sebagai blok. Sebagai contoh, dua contoh di bawah ini adalah contoh yang valid.

```
const a = {
  a: 0,
  b: 1,
};

const b =
    {a: 0, b: 1};
```

```
const c = {a: 0, b: 1};

someMethod(foo, {
  a: 0, b: 1,
}, bar);
```

Class literal (baik deklarasi atau ekspresi) diindentasi sebagai blok. Jangan menambahkan titik koma setelah *method* (setelah tutup kurung kurawal) ataupun pada *class* deklarasi, kecuali *class* ekspresi. Berikut ini adalah contoh *class* deklarasi.

```
class Foo {
  constructor() {
    /** @type {number} */
    this.x = 42;
  }

  /** @return {number} */
  method() {
    return this.x;
  }
}
```

```
Foo.Empty = class {};
```

Dan berikut ini adalah contoh *class* ekspresi dan *class* deklarasi.

```
/** Class ekspresi */
foo.Bar = class extends Foo {
  method() {
    return super.method() / 2;
  }
};

/** Class deklarasi */
class Frobnicator {
  frobnicate(message) {}
}
```

Ketika mendeklarasikan *anonymous function* di dalam argumen untuk pemanggilan *function*, isi dari fungsi tersebut diberi indentasi dua spasi lagi dari indentasi sebelumnya. Contoh sebagai berikut.

```
prefix.something.reallyLongFunctionName('whatever', (a1, a2) => {
  // Diberi indentasi dua kali dari pernyataan prefix satu baris di atas.
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

some.reallyLongFunctionCall(arg1, arg2, arg3)
    // Line-wrapping bisa dilihat pada II.3.5
    .thatsWrapped()
    .then((result) => {
      // Diberi indentasi dua kali dari pemanggilan '.then()' di atas.
      if (result) {
        result.use();
      }
    });
```

Kemudian pada *switch*, sama seperti blok lainnya, isi blok *switch* juga diberikan dua kali spasi. Setelah label *switch*, garis baru diberikan dan indentasi bertambah dua kali spasi persis seperti jika blok dibuka. Baris kosong bersifat opsional antara *break* dan *case* selanjutnya. Contoh sebagai berikut.

```
switch (animal) {
  case Animal.BANDERSNATCH:
    handleBandersnatch();
    break;

  case Animal.JABBERWOCK:
    handleJabberwock();
```

```
    break;

  default:
    throw new Error('Binatang tidak dikenal');
}
```

II.3.4. Pernyataan (Statements)

Setiap pernyataan diikuti oleh satu baris. Setiap pernyataan harus diakhiri dengan titik koma, seperti di bawah ini.

```
let nama = "Raka";
let umur = "21;
```

Mengandalkan *Automatic Semicolon Insertion* (ASI) seperti di bawah ini dilarang.

```
let nama = "Raka"
let umur = "21
```

II.3.5. Line-Wrapping

Line-wrapping adalah teknik memecah potongan kode menjadi beberapa baris untuk mematuhi batas kolom yang mana setiap baris harus tidak lebih dari 80 kolom per baris. Tidak ada formula khusus untuk teknik *line-wrapping* ini. Sangat banyak cara yang valid untuk *line-wrapping* potongan kode pada satu baris ke beberapa baris.

Arahan utama dari *line-wrapping* adalah lebih memilih untuk memberikan garis baru pada tingkat sintaksis yang lebih tinggi.

Bagus:

```
currentEstimate =
    calc(currentEstimate + x * currentEstimate) /
        2.0;
```

Kurang bagus:

```
currentEstimate = calc(currentEstimate + x *
    currentEstimate) / 2.0;
```

Pada contoh sebelumnya, level sintaksis dari tertinggi ke terendah adalah *assignment*, pembagian, pemanggilan *function*, parameter, *number constant*.

Operator dibungkus sebagai berikut.

a. Ketika baris terputus di operator, garis baru diberikan setelah simbol. Ini tidak berlaku pada titik “.” yang sebenarnya bukan operator.
b. Method atau nama constructor tetap melekan pada tanda kurung pembuka “(“ yang mengikutinya.
c. Tanda koma “,” tetap melekat pada token yang mendahuluinya. Contoh “estimate,”.

Ketika *line-wrapping*, setiap baris setelah yang pertama (baris lanjutan hasil *line-wrapping*) diberi indentasi empat kali spasi.

II.3.6. Komentar

Blok komentar diindentasi pada tingkat yang sama dengan kode di sekitarnya. Kode itu bisa berada dalam /* ... */ atau //. Untuk komentar /* ... */ yang berbaris banyak, baris selanjutnya harus dimulai dengan * sejajar dengan * di baris sebelumnya.

```
1.  /*
2.   * Ini
3.   * bagus.
4.   */
5.
6.  // Ini juga
7.  // bagus.
8.
9.  /* Mantap. */
```

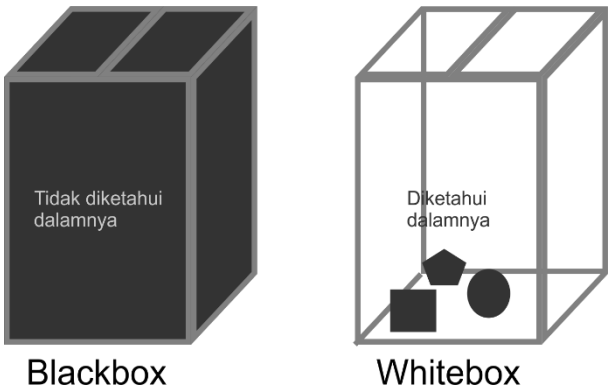
II.4. Class Library

Class library sering muncul pada konsep pemrograman berorientasi objek/OOP. *Class library* adalah kumpulan *class* yang sudah dirancang sedemikian rupa sebelumnya untuk digunakan saat membangun sebuah aplikasi perangkat lunak baru. Aplikasi berskala besar yang memiliki ratusan bahkan ribuan baris kode program membuatnya susah untuk diimplementasikan. Jika terdapat suatu kesalahan pada bagian tertentu, akan sulit jika misalnya bagian kesalahan itu terhubung dengan bagian lain dan perbaikan kode memakan waktu yang lama [6].

Untuk menggunakan sumber kode yang sudah ada ke dalam program baru, bahasa pemrograman harus bisa menyediakan kompilasi bagian program yang

terpisah. Dengan seperti itu, sumber kode yang sering dibutuhkan dapat dibuat dan disimpan pada sebuah *library* (pustaka untuk pemrograman perangkat lunak) [6]. Penggunaan *class library* sebagai contoh misalnya pada pembangunan aplikasi baru, dan aplikasi tersebut belum memiliki fungsional *repository*, sementara tersedia suatu *library* yang memfasilitasi fungsional tersebut. Kemudian programmer mengimporkan *library* yang tersedia lalu menerapkannya pada aplikasi tersebut. Dengan begitu, programmer pun tak perlu membuat ulang fungsionalitas-fungsionalitas terkait dan tinggal menyesuaikan apa saja yang dibutuhkan untuk mengimplementasikannya.

Berdasarkan karakteristik kustomisasi *framework* berorientasi objek, terdapat dua bentuk dari sebuah *class library* dan/atau *framework*, yaitu *whitebox* dan *blackbox*. *Whitebox* pada dasarnya adalah sebuah arsitektur yang diketahui komponennya oleh programmer saat membangun aplikasi. Desain lengkap harus didokumentasikan karena ini diperlukan untuk diadaptasikan ke aplikasi yang konkret. Mekanisme yang digunakan untuk memberikan fleksibilitas dari *whitebox* biasanya hanya terbatas pada *inheritance*. Sementara *blackbox* kebalikannya, menyembunyikan struktur internalnya. *Blackbox* memungkinkan pengguna hanya mengetahui hotspot dan deskripsi umumnya tentang penggunaan *framework*/*class library* daripada arsitekturnya [7]. Gambar II-6 mengilustrasikan perbandingan antara *whitebox* dan *blackbox class library*.



Gambar II-6 Blackbox vs Whitebox

II.5.Pengujian Perangkat Lunak

Pengujian perangkat lunak perlu dilakukan guna menemukan kesalahan/*bugs* atau kekurangan terhadap fungsional-fungsional pada aplikasi yang dibangun. Pengujian perangkat lunak pada penelitian ini terbagi menjadi pengujian unit dan pengujian integrasi.

II.5.1. Pengujian Unit

Pengujian unit dilakukan dengan pengujian pada unit program, di mana istilah unit ini diasumsikan dengan arti yang berbeda pada ruang lingkup tertentu. Unit yang dimaksud adalah sebuah prosedur, *module*, atau fungsional dari sebuah aplikasi [8]. Pengujian unit dilakukan dengan menguji kode dari prosedur, *module*, dan fungsional yang sudah ditulis apakah sudah sesuai harapan atau tidak. Dengan melakukan pengujian unit ini, programmer bisa lebih menghemat waktu saat melakukan perbaikan kode (*debugging*).

II.5.2. Pengujian Integrasi

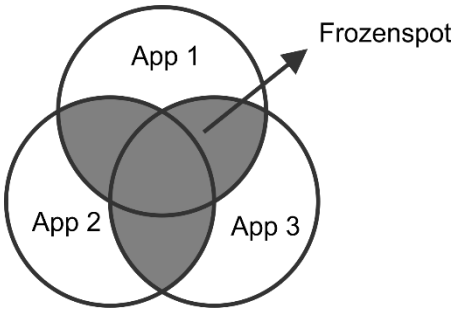
Pengujian integrasi dilakukan dengan melihat interaksi dari berbagai unit dan *module* yang menyusun suatu sistem [8]. Pengujian integrasi dilakukan setelah pengujian unit, dengan bertujuan untuk mencari kesalahan dan masalah yang terjadi pada perangkat lunak yang dibangun, dan dilakukan oleh programmer yang sudah ahli di bidangnya.

II.6.Analisis Domain

Analisis domain dilakukan dengan menganalisis beberapa aplikasi sejenis dari domain yang sama dengan tujuan untuk menentukan fungsionalitas yang sering ada pada domain aplikasi yang dianalisis [9]. Tujuan dari analisis domain ini adalah untuk menentukan fokus utama domain, serta informasi yang relevan [10]. Hal ini adalah tahap awal yang dibutuhkan untuk membangun *class library*. Untuk melakukan analisis domain, dibutuhkan setidaknya dua sampai lima aplikasi berdomain sama. Setelah analisis domain dilakukan, dilanjutkan dengan tahapan selanjutnya, yaitu *frozenspot*, *hotspot*, serta mendefinisikan kartu *hotspot*-nya.

II.6.1. Frozenspot

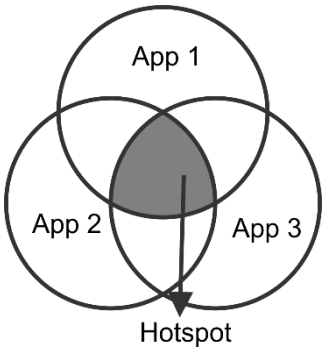
Frozenspot bisa dikatakan sebagai irisan fungsi dari beberapa aplikasi di domain yang sama [9]. Irisan fungsi ini bisa berupa sebagian atau keseluruhan fungsional yang didapat dari irisan tersebut. Hasil dari *frozenspot* kemudian digunakan untuk dilakukannya tahap analisis selanjutnya, yaitu *hotspot*. Gambar II-7 mengilustrasikan sebuah *frozenspot* dengan 3 aplikasi berdomain sama, arsiran abu-abu merupakan irisan yang didapat.



Gambar II-7 Frozenspot

II.6.2. Hotspot

Hotspot merupakan titik fleksibilitas dari domain aplikasi yang diteliti [9]. *Hotspot* dirancang sedemikian rupa dan disesuaikan untuk kebutuhan domain [11]. Selain itu, untuk mendefinisikan *hotspot* ada dua metode, yaitu *blackbox* dan *whitebox*. *Blackbox* adalah cara menentukan *hotspot* dengan melalui pendefinisian dari *frozenspot* yang sudah dianalisis, sementara *whitebox* adalah cara menentukan *hotspot* dari sumber kode yang sudah ada [12]. Gambar II-8 mengilustrasikan sebuah *hotspot*, arsiran abu-abu merupakan *hotspot* yang didapat.



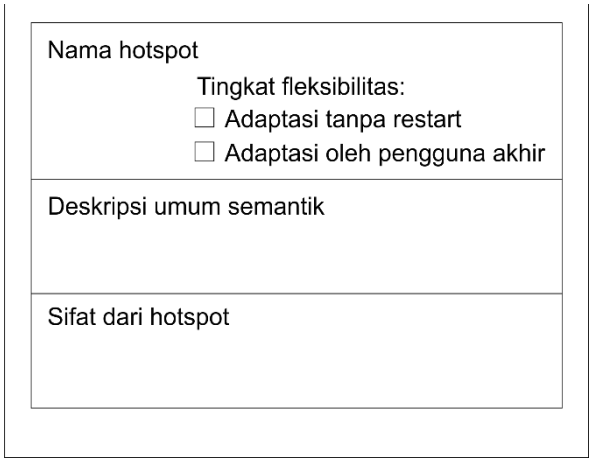
Gambar II-8 Hotspot

II.6.3. Kartu Hotspot

Kartu *hotspot* dimaksudkan untuk mendapatkan kebutuhan fleksibilitas dari aplikasi yang dianalisis. Kartu *hotspot* ini digunakan untuk mendeskripsikan sebuah fungsionalitas-fungsionalitas dari setiap *hotspot* yang didapat [13]. Kartu *hotspot* memiliki beberapa bagian, yaitu sebagai berikut [13].

1. Nama hotspot, yang biasanya berupa nama dari fungsionalitasnya.
2. Tingkat fleksibilitas, yang mana terdapat dua buah pilihan, yaitu *adaptasi tanpa restart* dan *adaptasi oleh pengguna akhir*. *Adaptasi tanpa restart* adalah adaptasi dari hotspot yang bisa langsung digunakan, sementara *adaptasi oleh pengguna akhir* adalah hotspot yang harus diadaptasi oleh pengguna akhir (programmer) sebelum digunakan.
3. Deskripsi umum semantik, merupakan penjelasan dari fungsional dari hotspot secara semantik.
4. Sifat dari hotspot, yang mana setidaknya terdapat minimal dua buah situasi yang spesifik.

Gambar II-9 adalah gambaran daripada kartu *hotspot*.



Gambar II-9 Kartu Hotspot

Method di dalam suatu *class* dikategorikan menjadi *hook method* dan *template method*. *Hook method* bisa dikatakan sebagai *placholder hotspot* fleksibel yang digunakan oleh *method* yang lebih kompleks. *Method* kompleks yang dimaksud adalah *template method*. *Template method* mendefinisikan aliran kontrol umum atau interaksi antara objek [13].

Ahli domain harus tahu bahwa *hotspot* yang dianalisis meminta fleksibilitas *run-time* (adaptasi tanpa *restart*) dan/atau berkemungkinan untuk perlunya adaptasi yang dilakukan oleh pengguna akhir/programmer. Jadi pilihan ini harus dibuat dengan sengaja [13].

Fungsi *hotspot* berhubungan erat dengan *hook method* dan *hook class*. Pada dasarnya, fungsi *hotspot* dengan rincian yang benar menjelaskan bahwa *hook method* atau kumpulan *hook method* harus ditambahkan/ada, baik disatukan di dalam *class* yang mana *template method* berada atau dipisahkan. Sebuah *class* yang memiliki *hook method* disebut *hook class*, sementara *class* yang memiliki *template method* di sebut *template class*. Pemisahan *template class* dan *hook class* membentuk prasyarat adaptasi *run-time*/adaptasi tanpa *restart*/tanpa perlu me-*restart* aplikasi. Berikut Tabel II.2 menjelaskan bagaimana cara transformasi objek model berdasarkan informasi fleksibilitas dari *hotspot* [13].

Tabel II.2 Aturan Transformasi Objek Model terhadap Kartu Hotspot

Adaptasi tanpa restart	Adaptasi oleh pengguna akhir	Transformasi objek model
✓	✓	Dibuat hook method di class terpisah serta alat konfigurasi.
✓	-	Dibuat hook method di class terpisah.
-	✓	Dibuat hook method serta alat konfigurasi.
-	-	Dibuat hook method.

II.7.Pemodelan

Pemodelan dilakukan untuk memodelkan konsep-konsep dan rancangan yang nantinya akan diimplementasikan. Dan alat pemodelan yang digunakan untuk penelitian ini adalah UML.

II.7.1. UML

UML adalah singkatan dari *Unified Model Language* (bahasa pemodelan terpadu). Ketika kita membuat suatu konsep menggunakan UML, ada aturan-aturan yang harus diikuti. Dalam UML terdapat beberapa alat yang berguna untuk penelitian ini, yaitu *use case* dan *class diagram*. Selain itu, UML juga menceritakan bagaimana konteksnya, komunikasi yang terjadi, juga permasalahan yang nantinya akan diselesaikan [14].

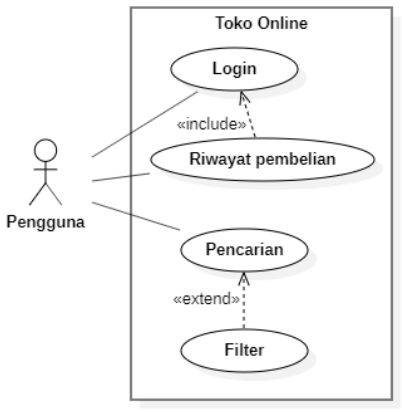
1. Use Case

Use case adalah sebuah *case*/kasus/situasi dari suatu sistem untuk memenuhi salah satu kebutuhan pengguna atau lebih. *Use case* menggambarkan potongan fungsi dari sistem. Use case merupakan suatu titik awal yang baik untuk pengembangan sistem dengan konsep OOP, juga untuk desain, pengujian dan dokumentasi [12].

Selain itu, *use case* juga berguna dalam membangun sebuah pengujian pada sistem. *Use case* menyediakan titik awal yang baik dalam membangun kasus dan prosedur pengujian karena *use case* secara tepat dapat menggambarkan suatu persyaratan pengguna dan kriteria kesuksesan [15].

Saat membangun *use case diagram*, terdapat *actor* (pelaku), *use case* (kegiatan), dan garis penghubung (baik itu garis komunikasi, *include*, maupun *extend*). *Actor* digambarkan sebagai *stickman* dengan label yang menggambarkan *actor* itu sendiri pada bagian bawahnya, sementara *use case* digambarkan dengan bentuk oval dengan nama atau deskripsi yang menjelaskan kegiatan yang terjadi [15]. Untuk garis penghubung, juga terdapat *include* dan *extend* yang mana kedua penghubung ini memiliki fungsi yang hampir sama. *Include* memberitahu sebuah *use case* menggunakan ulang semua langkah-langkah terhadap *use case* lain yang ditunjuk, dan tidak bisa berdiri sendiri. Yang membedakannya dengan *extend* yakni untuk menggunakan ulang semua langkah-langkahnya menjadi opsional, sehingga bisa dikatakan *use case* yang dipilih *extend* terhadap *use case* yang dituju bersifat opsional dalam hal penggunaan ulang langkah dan dependensi.

Use case diagram juga memiliki satu notasi lain yang disebut dengan *system boundary*/area sistem. Dengan area sistem, kita bisa mengetahui entitas mana saja yang berada di dalam sistem. Tentunya *actor* berada pada luar sistem sehingga *actor* diletakkan di luar area sistem. Gambar II-10 menunjukkan contoh *use case* untuk contoh kasus pada sebagian sistem toko online.



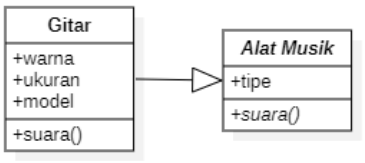
Gambar II-10 Contoh Use Case

Pengguna sebagai *actor*. Login, riwayat pembelian, pencarian dan filter sebagai *use case*. Dan keempat *use case* ini berada pada area sistem yang bernama toko online.

2. Class Diagram

Class diagram sangat berguna dalam semua pemrograman berkonsep OOP, digunakan untuk mengilustrasikan suatu hubungan dan dependensi dari sumber kode antara *class*. *Class diagram* merepresentasikan kumpulan *class* yang diatur dalam suatu kelompok yang membawa karakteristiknya [16].

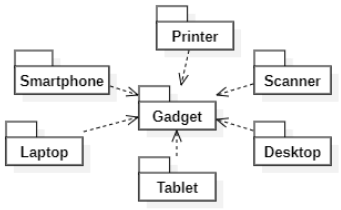
Pada dasarnya, suatu *class* dalam *class diagram* memiliki tiga bagian, bagian atas, tengah dan bawah. Bagian atas menunjukkan nama dari *class*, bagian tengah menunjukkan kumpulan atribut dan informasi dari *class*, sementara bagian bawah menunjukkan kumpulan operasi atau tingkah laku yang menggambarkan *class* itu sendiri. Gambar II-11 menunjukkan contoh *class diagram* untuk contoh kasus gitar.



Gambar II-11 Contoh Class Diagram

3. Package Diagram

Di dalam UML, kumpulan-kumpulan *class* dimodelkan dalam sebuah *package* (paket). Sebagian besar OOP, memiliki paket UML analog untuk mengatur dan menghindari tabrakannya antara nama *class*. *Package diagram* digunakan untuk melihat dependensi antar *package* [15]. Dan setiap *package* yang ada bisa dikatakan sebagai *class* itu sendiri jika berada pada sistem. Gambar II-12 menunjukkan contoh *package diagram* untuk contoh kasus gadget.



Gambar II-12 Contoh Package Diagram