BAB 2

LANDASAN TEORI

2.1 Bahasa Pemograman PHP

Bahasa pemrograman open source yang dinamakan PHP umumnya digunakan untuk pengembangan website dan dapat disisipkan ke dalam kode HTML [7]. Bahasa pemrograman PHP merupakan salah satu pilihan terbaik untuk pengembangan aplikasi web karena mampu mengakses *database*, memproses data, dan menghasilkan output yang dinamis pada halaman web. PHP merupakan bahasa pemrograman *open source* yang tersedia secara gratis dan dapat diubah sesuai dengan kebutuhan pengguna. PHP juga memiliki berbagai macam *library* tambahan *open source* yang memudahkan pengembangan aplikasi web. Karena biaya rendah dan banyaknya sumber daya yang tersedia, PHP menjadi salah satu pilihan yang populer bagi pengembang aplikasi web.

2.2 Framework Laravel

Laravel merupakan kerangka kerja (*framework*) sumber terbuka (*open-source*) yang ditulis dalam bahasa pemrograman PHP. *Framework* ini menawarkan arsitektur bersih dan modular untuk membangun aplikasi web yang menggunakan pola desain *Model-View-Controller* (MVC). Laravel dilengkapi dengan banyak fitur seperti *routing, middleware, Object-Relational Mapping* (ORM), dan *templating* yang memudahkan dalam pengembangan aplikasi web. Selain itu, dokumentasi Laravel yang lengkap dan komunitas pengembang yang aktif juga membantu para pengembang untuk lebih mudah memahami dan menggunakan *framework* ini. Dengan menggunakan Laravel, pengembang dapat membuat aplikasi web dengan kode yang mudah dipelihara, efisien, dan lebih mudah untuk dikembangkan

2.3 PHPMetrics

PHPMetrics merupakan sebuah alat analisis kualitas kode (*code quality analysis tool*) untuk bahasa pemrograman PHP yang bersifat *open source*. Alat ini dikembangkan oleh Jean-François Lépine dan dapat mengukur beberapa *metrics* seperti tingkat kompleksitas aplikasi, jumlah code atau variabel tertentu, data terkait *object oriented*, maupun tingkat *maintainability*. PHPMetrics mengklasifikasikan metrics pengukuran ke dalam beberapa kategori, antara lain:

a. *Complexity* yang mencakup *Cyclomatic complexity*, *Myer's interval*, dan *Relative system complexity*.
b. *Volume* yang mencakup *Vocabulary*, *Data complexity*, *Lines of code*, dan *Readability*.
c. *Object Oriented* yang mencakup *Lack of cohesion of methods*, *Coupling*, dan *Abstraction*.
d. *Maintainability* yang mencakup *Maintainability index*, *Halstead's metrics*, dan *Effort*.

2.4 *Maintainbilit*y Indeks

Maintainability Index (MI) adalah sebuah ukuran komposit yang menggabungkan beberapa metrik tradisional dalam kode untuk mengevaluasi tingkat *maintainability* suatu aplikasi [11] [12]. MI terdiri dari metrik *Halstead Volume* (V), metrik *Cyclomatic Complexity* (V(g)), rata-rata jumlah baris kode per modul (LoC), dan persentase jumlah komentar per modul (CM). *Maintainbaility index* ini diformulasikan oleh SEI (Software Engineering Institute). MI memberikan sebuah nilai relatif terhadap maintainability suatu aplikasi yang diukur dengan metrik-metrik tersebut [13]. Adapun formula *Maintainability Index* dapat dilihat pada rumus ini

$$\mathrm{MI} = \Big((171 - (5.2 * \ln(\mathrm{V})) - (0.23 * (\mathrm{V(g)}) - 16.2 * \ln(\mathrm{LoC})) * \frac{100}{171} + 50 * \sin\left(\sqrt{2{,}4 * CM}\right) \Big)$$

Keterangan:

$V(g)$ = $Cyclomatic\ Complerxity$

LOC = Line of Code

V = Halstead Metrics

CM = Percent Line of Comment

Secara umum, nilai dari *Maintainability Index* diukur dari 0 sampai 100 lebih, di mana semakin tinggi nilai tersebut maka aplikasi tersebut mudah dipelihara. Nilai tersebut terbagi menjadi tiga kategori yang dapat dilihat pada **Tabel 2.1** [26].

Tabel 0.1 Rentang Penilaian Maintainability Index

Rentang	Keterangan
< 64	low maintainbility, Kode program memiliki masalah serius.
65 – 84	medium maintainability, Kode program mengalami beberapa masalah, tetapi tidak ada yang terlalu serius
> 85	high maintainability, kode tidak memiliki masalah serius

2.5 Cyclomatic Complexity

McCabe's *Cyclomatic Complexity* adalah sebuah metode pengukuran yang digunakan untuk menghitung jumlah kontrol alur dari suatu modul dalam sebuah program. Semakin tinggi nilai *Cyclomatic Complexity*, semakin kompleks kontrol alur dari modul tersebut. Hal ini dapat mempengaruhi kesulitan dalam pengujian

dan perawatan modul tersebut. Jika sebuah modul tidak memiliki percabangan, maka kompleksitasnya dianggap satu. Namun, jika modul memiliki satu atau lebih

percabangan, maka perhitungan dapat dilakukan menggunakan persamaan rumus tertentu.

$$V(g) = E - N + 2$$

Keterangan:

$V(g)$ = $Cyclomatic\ Complerxity$

E = $Jumla$h $edge\ pada\ grafik$

N = $Jumla$h $node\ pada\ grafik$

Sebaiknya modul memiliki tingkat *Cyclomatic Complexity* kurang dari 15, karena nilai yang lebih tinggi akan membuat identifikasi dan inspeksi menjadi lebih sulit akibat adanya banyak jalur eksekusi. Namun, nilai maksimal yang disarankan untuk *Cyclomatic Complexity* adalah 100. Rentang nilai *Cyclomatic Complexity* dan risikonya dapat ditemukan dalam **Tabel 2.2**.

Tabel 0.2 Rentang nilai Cyclomatic Complexity

Rentang	Keterangan
0 – 5	Minimal
6 – 10	Sedang
10+	Tinggi

2.6 Halstead Metrics

Halstead metrics adalah teknik pengukuran kompleksitas modul dalam sebuah program yang didasarkan pada jumlah operator dan operand dalam kode sumber. Terdapat enam komponen kategori dalam *Halstead's metrics*, yaitu:

1. *Length of Program*

Length of Program adalah pengukuran yang menghitung jumlah total operator dan operand yang muncul dalam suatu program. Formula untuk menghitung *Length of Program* dirumuskan dalam persamaan sebagai berikut:

$$N = N1 + N2$$

Keterangan:

N1 = total semua *operator* yang muncul

N2 = total semua *operand* yang muncul

2. *Vocabulary of the program*

Vocabulary of the program adalah suatu penghitungan jumlah operator dan operand yang berbeda yang digunakan dalam sebuah program. Rumus untuk menghitung *Vocabulary of the program* adalah sebagai berikut:

$$n = n1 + n2$$

Keterangan:

n = *Vocabulary of the program*

n1 = jumlah *operator* berbeda

n2 = jumlah *operand* berbeda

3. *Volume of the program*

Halstead's metric menggunakan *Volume* untuk mengukur ukuran program. *Volume* digunakan untuk mengukur jumlah total instruksi yang ada dalam kode program. Persamaan untuk menghitung *Volume* of the program ditentukan sebagai berikut:

$$V = N\ x\ log_2 n$$

Keterangan:

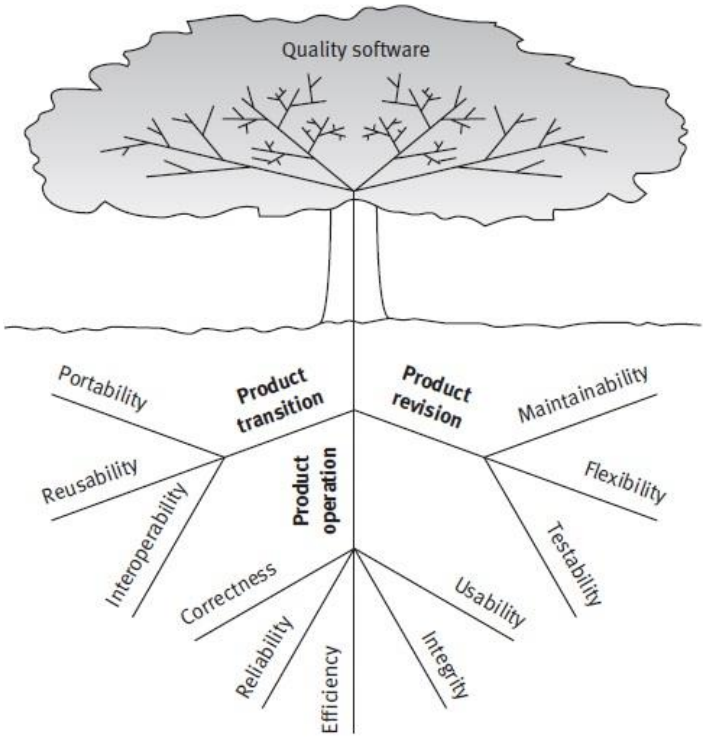
V = *Volume of the program*

N = nilai kalkulasi *length of the program*

n = nilai kalkulasi *vocabulary of the program*

2.7 Metode McCall

Metode McCall merupakan salah satu model yang membahas Faktor Kualitas Perangkat Lunak. Model ini mencakup tiga perspektif utama yaitu *Product Operation* (*Correctness*, *Reliability*, *Efficiency*, *Integrity*, *Usability*), *Product Revision* (*Maintainability*, *Flexibility*, *Testability*), dan Product Transition (*Portability*, *Reusability*, *Interoperability*) [15]. Model ini memiliki kriteria paling lengkap untuk kualitas perangkat lunak, dan karena keseluruhan dan rinciannya, metode ini bermanfaat untuk menguji dan memastikan kualitas perangkat lunak. Model kualitas McCall terdiri dari 11 faktor kualitas, yang digambarkan sebagai pohon McCall yang terlihat pada **Gambar 2.1**.



Gambar 2.1 Pohon McCall

Penjelasan dari masing-masing faktor kualitas tersebut adalah sebagai berikut:

1. *Correctness*

Untuk menilai kebenaran sebuah perangkat lunak, perlu dipenuhi kriteria berikut:

- Mampu menghasilkan output yang benar sesuai dengan masukan pengguna

- Menjalankan proses yang sesuai tanpa kurang atau berlebihan,
- Dapat dibuktikan secara formal melalui pendekatan matematis.

2. *Reliability*

Dalam aspek reliabilitas *(reability)*, fokusnya adalah pada kemungkinan terjadinya masalah selama operasi perangkat lunak dalam jangka waktu tertentu dan di lingkungan tertentu. Dalam pandangan ini, keandalan perangkat lunak tidak tergantung secara langsung pada waktu.

3. *Efficiency*

Cara-cara untuk menggunakan sumber daya seperti waktu pemrosesan (eksekusi), penggunaan media penyimpanan (memori, ruang, dan *bandwidth*).

4. *Integrity*

Model McCall menempatkan lebih banyak fokus pada integritas perangkat lunak, khususnya pada aspek keamanannya. Oleh karena itu, pengembang perangkat lunak harus memperhatikan kebutuhan akses pengguna terhadap perangkat lunak.

5. *Usability*

Faktor ini menitikberatkan pada kemudahan penggunaan dan pembelajaran perangkat lunak. *Usability* mempertimbangkan aspek-aspek akademis seperti psikologi, ergonomi, dan faktor manusia.

6. *Maintainbility*

Maintainability merujuk pada kemudahan perangkat lunak dalam menjalani proses pemeliharaan, termasuk perbaikan masalah, penambahan fitur baru, kemudahan pemeliharaan di masa depan, dan penyesuaian terhadap perubahan lingkungan. Tingkat kemudahan pemeliharaan suatu perangkat lunak dapat diukur dari usaha yang diperlukan untuk memperbaiki bug kecil.

7. *Flexibility*

Kemudahan dalam melakukan modifikasi karena perubahan lingkungan.

8. *Testability*

Testability adalah kemampuan perangkat lunak untuk dites. Selain itu *testability* adalah tingkat yang dimiliki suatu sistem untuk memfasilitasi kriteria pengujian dan melakukan evaluasi atas hasil pengujian untuk menentukan sejauh mana kriteria tersebut terpenuhi.

9. *Portability*

Suatu perangkat lunak dianggap portabel jika biaya yang diperlukan untuk memindahkannya ke lingkungan baru lebih rendah daripada biaya yang diperlukan untuk membuatnya dari awal, termasuk biaya transportasi dan adaptasi.

10. *Reusability*

Reusabilitas *(Reusability)* merupakan suatu karakteristik dari perangkat lunak yang memungkinkan penggunaan kembali perangkat lunak atau bagian-bagiannya dalam sistem yang berbeda. Kualitas reusabilitas suatu perangkat lunak dinilai dari kemampuan modul-modulnya untuk digunakan kembali dalam aplikasi lain.

11. *Interoperability*

Interoperabilitas *(Interoperability)* merupakan kemampuan suatu perangkat lunak untuk berintegrasi dengan perangkat lunak lain secara efektif tanpa mengalami kesulitan atau hambatan dalam proses tersebut.

2.8 Design Pattern

Dalam pengembangan perangkat lunak, design patterns adalah suatu standar solusi yang dapat digunakan untuk mengatasi masalah umum yang sering terjadi dalam merancang software. Design patterns tidak berupa rancangan akhir yang dapat langsung diterapkan ke dalam kode, tetapi berupa deskripsi atau pola yang menunjukkan bagaimana mengatasi masalah tersebut dan dapat diterapkan pada situasi yang berbeda-beda [16].

Dalam buku *Design Patterns: Elements of Reusable Object Oriented Software* mencakup 23 *design patterns* [17]. Dari 23 *design patterns* yang ada terbagi menjadi 3 kategori yaitu :

a. Creational Design Pattern

Creational Design Pattern adalah kategori desain pola yang berfokus pada cara objek dibuat. Pembuatan objek seringkali menjadi sulit karena melibatkan logika dan kondisi yang rumit, sehingga Creational Design Pattern bertujuan untuk mengurangi kompleksitas kode yang dibuat oleh pengembang. Terdapat lima jenis pola desain dalam kategori ini, yaitu Factory Method, Abstract Factory, Builder, Prototype, dan Singleton.

b. Structural Design Pattern

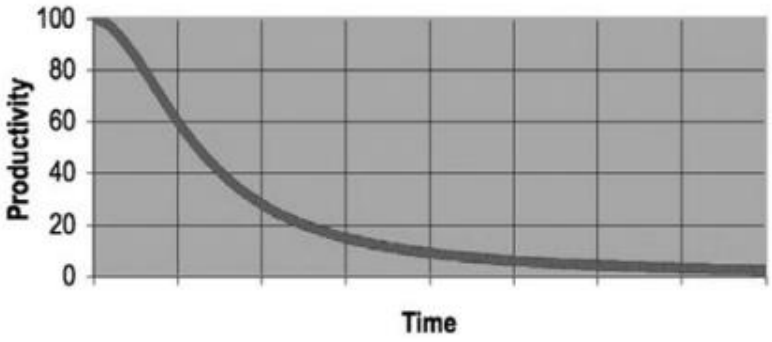
Structural Design Patterns berkaitan dengan hubungan antara elemen dalam kelas dan objek. Pola-pola ini membantu menyederhanakan struktur sistem dengan menentukan keterkaitan antar objek. Beberapa contoh pola yang termasuk dalam kategori ini antara lain *Adapter, Bridge, Composite, Decorator, Façade, Flyweight,* dan *Proxy*.

c. Behavioral Design Pattern

Behavioral Design Patterns berfokus pada cara objek berinteraksi dan berkomunikasi satu sama lain. Pola-pola ini bertujuan untuk mengurangi kompleksitas yang mungkin terjadi saat objek saling berinteraksi. Beberapa jenis desain pola dalam kategori ini termasuk *Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy,* dan *Visitor*.

2.9 *Clean Code*

Clean code adalah prinsip penulisan kode yang mudah dibaca dan dipahami oleh pengembang lainnya. Tujuannya adalah untuk mengurangi dampak negatif kode yang sulit dibaca dan memperbaiki produktivitas dalam pengembangan perangkat lunak seperti yang terlihat pada **Gambar 2.2** [18].



Gambar 2.2 *Productivity vs Time*

Terdapat beberapa hal inti yang dibahas pada *Clean Code* yaitu :

1 *Meaningful Name*
2 *Clean Function*
3 *Clean Comments*
4 *Formatting Code*
5 *Error Handling*
6 *Object and Data Structures*
7 *Clean Class*

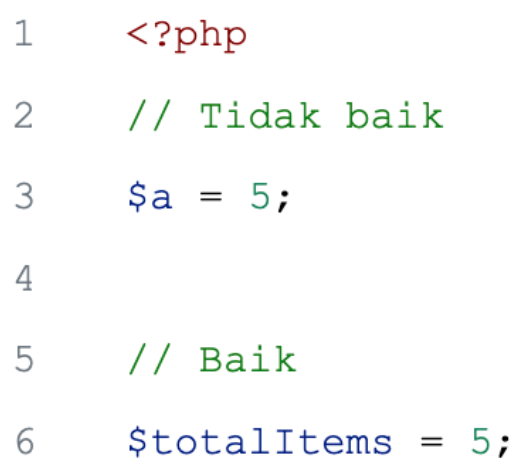
4.2.4 Meaningful Name

Pola penamaan yang bermakna *(meaningful names)* merupakan sebuah konsep dalam pemrograman yang memberikan panduan mengenai cara memberi nama variabel, fungsi/metode, dan kelas agar lebih mudah dipahami. Beberapa panduan yang dapat diikuti dalam memberikan nama adalah sebagai berikut:

1. *Use Intention-Revealing Name*

Use Intention-Revealing Names mengacu pada penggunaan nama-nama yang mengungkapkan niat atau tujuan dari sebuah kode. Dalam pengembangan perangkat lunak, ini berarti memberikan nama yang jelas, deskriptif, dan mudah dimengerti untuk variabel, fungsi, metode, dan elemen-elemen lain dalam kode.

Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.3**.



Gambar 2.3 Contoh *Meaningful Names Use Intention-Revealing Name*

2. *Avoid Disinformation*

Avoid Disinformation menekankan pada pentingnya menghindari penggunaan nama yang dapat menyesatkan atau menimbulkan kebingungan tentang apa yang sebenarnya dilakukan oleh kode tersebut. *Avoid Disinformation* dalam penggunaan *meaningful names* bertujuan untuk mencegah kesalahpahaman, meningkatkan kejelasan, dan memudahkan pemeliharaan code. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.4**.

```php
<?php
// Tidak baik
$getAllData = function() {
    // ...
};

// Baik
$getUserData = function() {
    // ...
};
```

Gambar 2.4 Contoh *Meaningful Names Avoid Disinformation*.

3. *Make Meaningful Distinctions*

Make Meaningful Distinctions dalam prinsip *meaningful names* berarti membuat perbedaan yang bermakna antara elemen-elemen yang memiliki nama serupa dalam kode. Hal ini membantu dalam membedakan dan mengidentifikasi dengan jelas elemen-elemen tersebut berdasarkan tujuan, fungsi, atau karakteristik yang berbeda. Dengan menerapkan *Make Meaningful Distinctions* dalam *meaningful names*, dapat memastikan bahwa elemen-elemen kode yang serupa dikenali dengan jelas dan tidak menimbulkan kebingungan. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.5**.

```php
<?php

// Tidak baik
class Product {
    // ...
}

class ProductData {
    // ...
}

// Baik
class Product {
    // ...
}

class ProductDetails {
    // ...
}
```

Gambar 2.5 Contoh *Meaningful Names Make Meaningful Distinctions*

4. *Use Pronounceable Names*

Use Pronounceable Names berarti menggunakan nama-nama yang mudah diucapkan dan dilafalkan oleh orang-orang ketika membaca atau berbicara tentang code. Nama-nama ini harus mudah dipahami secara lisan dan tidak membingungkan. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.6**.

```php
<?php

// Tidak baik
$qtyLmt = 100;
$plb = calculateProductLabel();

// Baik
$quantityLimit = 100;
$productLabel = calculateProductLabel();

```

Gambar 2.6 Contoh *Meaningful Names Use Pronounceable Names*

5. *Use Searchable Names*

Use Searchable Names dalam prinsip *meaningful names* yaitu menggunakan nama-nama yang mudah dicari atau ditemukan dalam kode. Nama-nama ini harus memungkinkan penggunaan fungsi pencarian dalam editor kode untuk menemukan referensi atau penggunaan elemen code dengan cepat. Bertujuan untuk mempermudah pengembang dalam menemukan referensi atau penggunaan elemen kode tertentu melalui fitur pencarian dalam editor kode. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.7**.

```php
<?php

// Tidak baik
$rv = getProducts($id);

// Baik
$relatedProducts = getRelatedProducts($productId);

```

Gambar 2. 7 Contoh *Meaningful Names Use Searchable Names*

6. *Avoid Encodings*

Avoid Encodings yaitu menghindari penggunaan kode atau encoding yang ambigu atau tidak jelas dalam memberikan nama pada elemen-elemen kode. Penggunaan kode atau *encoding* yang sulit dipahami dapat menyebabkan kebingungan dan mengurangi kejelasan dalam kode. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.8**.

```
1    <?php
2
3    // Tidak baik
4    $amt1 = $qty * $prc;
5
6    // Baik
7    $totalAmount = $quantity * $price;
8
```

Gambar 2.8 Contoh *Meaningful Names Avoid Encodings*.

7. *Avoid Mental Mapping*

Avoid Mental Mapping yaitu menghindari membuat pengguna kode harus melakukan pemetaan mental atau mengingat hubungan antara nama dan artinya. Nama-nama harus jelas dan deskriptif sehingga pengguna kode dapat langsung memahami maknanya tanpa harus melakukan pemetaan mental tambahan. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.9**.

```php
<?php

// Tidak baik
$c = 10; // Jumlah pelanggan
$i = 0; // Indeks

// Baik
$customerCount = 10;
$index = 0;

```

Gambar 2.9 Contoh *Meaningful Names Avoid Mental Mapping*

4.2.5 Clean Function

digu. Berikut merupakan petunjuk yang dapat digunakan dalam menulis *method*/fungsi:

i. *Make it small*

Make it small dalam *clean function* mengacu pada prinsip bahwa sebuah fungsi harus dibuat dengan ukuran yang kecil dan fokus pada tugas yang spesifik. Fungsi yang kecil memiliki tujuan yang jelas, mudah dipahami, dan lebih mudah untuk diuji, dipelihara, dan digunakan ulang. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.10**.

```php
<?php

// Tidak baik
function processOrder($order)
{
    // Logika yang kompleks untuk memproses pesanan
    // ...
}

// Baik
function validateOrder($order)
{
    // Logika validasi pesanan
    // ...
}

function calculateTotal($order)
{
    // Logika perhitungan total pesanan
    // ...
}

function saveOrder($order)
{
    // Logika menyimpan pesanan ke database
    // ...
}

```

Gambar 2.10 Contoh *Clean function Make it small*

i. *Block and Indenting*

Block and Indenting dalam clean function mengacu pada prinsip untuk menggunakan tata letak yang jelas dan indentasi yang konsisten dalam kode. Ini membantu meningkatkan keterbacaan dan memudahkan pengembang dalam memahami struktur dan alur logika program. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.11**.

```php
<?php

// Tidak baik
function calculateTotal($items)
{
if(count($items)>0){
$total = 0;
foreach($items as $item){
$total += $item['price'] * $item['quantity'];
}
return $total;
}
}

// Baik
function calculateTotal($items)
{
    if (count($items) > 0) {
        $total = 0;
        foreach ($items as $item) {
            $total += $item['price'] * $item['quantity'];
        }
        return $total;
    }
}

```

Gambar 2.11 Contoh *Clean function Block and Indenting*

i. *Do One Thing*

Do One Thing dalam clean function mengacu pada prinsip bahwa sebuah fungsi seharusnya hanya melakukan satu tugas atau memiliki satu tanggung jawab. Fungsi yang melakukan satu hal secara terfokus membuat kode lebih mudah dipahami, diuji, dan dipelihara. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.12**.

```php
<?php

// Tidak baik
function validateAndSaveUser($user)
{
    // Validasi data pengguna
    // ...

    // Simpan pengguna ke database
    // ...

    // Kirim email konfirmasi
    // ...

    // Kirim notifikasi ke administrator
    // ...
}

// Baik
function validateUser($user)
{
    // Validasi data pengguna
    // ...
}

```

Gambar 2.12 Contoh *Clean function Do One Thing*

i. *Stepdown Rule*

Stepdown Rule dalam *clean function* mengacu pada prinsip bahwa kode seharusnya ditulis dalam urutan yang terorganisir, di mana setiap level abstraksi diikuti oleh level yang lebih spesifik. Prinsip ini membantu dalam membangun struktur yang jelas dan mempermudah pemahaman alur logika program. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.13**.

```php
<?php

function processOrder($order)
{
    validateOrder($order);
    calculateTotal($order);
    applyDiscount($order);
    generateInvoice($order);
    sendNotification($order);
}

function validateOrder($order)
{
    // Validasi pesanan
    // ...
}

function calculateTotal($order)
{
    // Menghitung total pesanan
    // ...
}


function applyDiscount($order)
{
    // Menerapkan diskon jika berlaku
    // ...
}

function generateInvoice($order)
{
    // Menghasilkan invoice pesanan
    // ...
}

function sendNotification($order)
{
    // Mengirim notifikasi kepada pelanggan
    // ...
}

```

Gambar 2.13 Contoh *Clean function Stepdown Rule*

ii. *Function Arguments*

Function Arguments dalam *clean function* mengacu pada parameter atau argumen yang diterima oleh sebuah fungsi. Prinsip ini mendorong penggunaan jumlah dan tipe argumen yang minimal serta menjaga argumen tetap fokus pada tugas yang spesifik. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.14**.

```php
<?php

function calculateCircleArea($radius)
{
    return 3.14 * $radius * $radius;
}

function sendEmail($to, $subject, $message)
{
    // Logika pengiriman email
    // ...
}

function getUser($userId)
{
    // Mendapatkan data pengguna berdasarkan ID
    // ...
}

```

Gambar 2.14 Contoh *Clean function Function Arguments*

iii. *No Side Effect*

No Side Effect dalam *clean function* mengacu pada prinsip bahwa sebuah fungsi seharusnya tidak memiliki efek samping yang tidak diharapkan pada lingkungan atau data di luar fungsi itu sendiri. Fungsi yang tidak memiliki efek samping mempermudah pemahaman, pemeliharaan, dan pengujian kode. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.15**.

```php
<?php

function calculateArea($length, $width)
{
    return $length * $width;
}

function isEven($number)
{
    return $number % 2 === 0;
}

function greet($name)
{
    return "Hello, " . $name;
}
```

Gambar 2. 15 Contoh Clean *function No Side Effect*

4.2.6 Clean Comment

Konsep *clean comment* memberikan petunjuk dalam menulis komentar yang efektif dan informatif agar memberikan informasi tambahan pada kode. Namun, penamaan yang jelas dan bermakna juga sangat penting dalam mengurangi jumlah komentar yang dibutuhkan. Semakin baik penamaan variabel, fungsi, dan modul, semakin sedikit komentar yang dibutuhkan untuk menjelaskan kode. Menurut buku Clean Code karya Robert C. Martin terdapat 2 (dua) jenis komentar, yaitu Good Comment dan Bad Comment. Berikut petunjuk dalam menulis kedua jenis komentar tersebut:

1. *Good Comment*

a. *Explain Yourself in Code*

Explain Yourself in Code dalam *clean comments* mengacu pada prinsip bahwa kode seharusnya self-explanatory dan komentar harus digunakan secara minimal. Komentar hanya diperlukan untuk menjelaskan alasan, keputusan desain, atau pemahaman yang tidak dapat diungkapkan melalui kode itu sendiri. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.16**.

```php
<?php

// Baik
// Periksa untuk melihat apakah karyawan memenuhi syarat untuk tunjangan penuh
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))


// Tidak baik
if (employee.isEligibleForFullBenefits())

```

Gambar 2.16 Contoh *Clean Comment Explain Yourself in Code*

b. *Legal Comments*

Legal Comments dalam *clean comments* mengacu pada prinsip bahwa komentar yang digunakan dalam kode sebaiknya berkaitan dengan aspek hukum, lisensi, hak cipta, atau informasi legal lainnya yang relevan dengan proyek perangkat lunak. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.17**.

```php
<?php

/**
 * Proyek XYZ - Hak Cipta © 2023 oleh Perusahaan ABC.
 * Versi 1.0.0 - Dilarang menggunakan tanpa izin tertulis.
 * Informasi lebih lanjut: www.perusahaanabc.com
 */

// Hak cipta © 2023 oleh Perusahaan ABC. Hak cipta dilindungi oleh hukum.
// Jangan menggunakan code ini tanpa izin tertulis dari pemilik hak cipta.

// Lisensi: MIT License. Informasi lebih lanjut tersedia di https://opensource.org/licenses/MIT.
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))

```

Gambar 2.17 Contoh *Clean Comment Legal Comments*

c. *Informative Comments*

Informative Comments dalam *clean comments* mengacu pada prinsip bahwa komentar yang digunakan dalam kode sebaiknya memberikan informasi yang berguna, menjelaskan kompleksitas atau algoritma tertentu, memberikan

pemahaman tentang niat atau logika di balik kode, atau memberikan panduan penggunaan yang jelas. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.18**.

```php
<?php

// Baik
// format yang cocok kk:mm:ss EEE, MMM dd, yyyy Pola timeMatcher = Pola.compile(
"\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");

// Tidak Baik
// Mengembalikan instansce dari Responder yang sedang diuji.
protected abstract Responder responderInstance();

```

Gambar 2.18 Contoh *Clean Comment Informative Comments*

d. *Clarification*

Clarification dalam *clean comments* mengacu pada prinsip bahwa komentar yang digunakan dalam kode sebaiknya memberikan penjelasan atau klarifikasi tambahan untuk memperjelas maksud, penggunaan, atau perilaku dari elemen-elemen kode yang kompleks atau ambigu. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.19**.

```php
<?php

function testCompareTo() {
    $a = PathParser::parse("PageA");
    $ab = PathParser::parse("PageA.PageB");
    $b = PathParser::parse("PageB");
    $aa = PathParser::parse("PageA.PageA");
    $bb = PathParser::parse("PageB.PageB");
    $ba = PathParser::parse("PageB.PageA");

    assert($a->compareTo($a) == 0); // a == a
    assert($a->compareTo($b) != 0); // a != b
    assert($ab->compareTo($ab) == 0); // ab == ab
    assert($a->compareTo($b) == -1); // a < b
    assert($aa->compareTo($ab) == -1); // aa < ab
    assert($ba->compareTo($bb) == -1); // ba < bb
    assert($b->compareTo($a) == 1); // b > a
    assert($ab->compareTo($aa) == 1); // ab > aa
    assert($bb->compareTo($ba) == 1);  // bb > ba
}

```

Gambar 2.19 Contoh *Clean Comment Clarification*

e. *Warning of Consequences*

Warning of Consequences dalam *clean comments* merujuk pada prinsip bahwa komentar sebaiknya digunakan untuk memberikan peringatan tentang konsekuensi atau efek samping yang mungkin terjadi akibat penggunaan atau perubahan pada elemen kode tertentu. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.20**.

```php
<?php

/**
 * @ignore
 * Takes too long to run
 */
function _testWithReallyBigFile() {
    writeLinesToFile(10000000);
    $response->setBody($testFile);
    $response->readyToSend($this);
    $responseString = $output->toString();
    assert(strpos($responseString, "Content-Length: 1000000000") !== false);
    assertTrue($bytesSent > 1000000000);
}
```

Gambar 2.20 Contoh *Clean Comment Warning of Consequences*

2. *Bad Comment*

a. *Redundant Comments*

Redundant Comments merupakan prinsip yang menyarankan untuk menghindari penggunaan komentar yang redundan atau berulang dalam kode. Hal ini dilakukan untuk mencegah informasi yang sama dituliskan dua kali, baik dalam komentar maupun dalam kode itu sendiri. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.21**.

```php
<?php

// Bad example - redundant comment
// Assign 5 to the variable x
$x = 5;

// Good example - self-explanatory code
$x = 5;

// Bad example - redundant comment
// Loop through the array and print each element
foreach ($array as $element) {
    echo $element;
}

// Good example - self-explanatory code
foreach ($array as $element) {
    echo $element;
}

```

Gambar 2.21 Contoh *Clean Comment Redundant Comments*

b. *Misleading Comments*

Misleading Comments dalam *clean comments* adalah prinsip yang menyarankan untuk menghindari penggunaan komentar yang menyesatkan atau membingungkan dalam kode. Hal ini dilakukan untuk memastikan bahwa komentar yang ditambahkan memberikan informasi yang akurat dan tidak menyesatkan pembaca kode. Berikut contoh penggunaan prinsip ini dalam bahasa PHP, dapat dilhat pada **Gambar 2.22**.

```php
<?php

// Bad example - misleading comment
// Calculate the sum of two numbers
$result = $number1 - $number2;

// Good example - accurate comment
// Calculate the difference between two numbers
$result = $number1 - $number2;

// Bad example - misleading comment
// Validate the user's age
if ($age < 18) {
    // ...
}

// Good example - accurate comment
// Check if the user is underage
if ($age < 18) {
    // ...
}

```

Gambar 2.22 Contoh *Clean Comment Misleading Comments*

c. *Journal Comments*

Journal Comments dalam *clean comments* adalah prinsip yang menyarankan untuk menghindari penggunaan komentar jurnal atau log dalam kode. Komentar jurnal adalah komentar yang mencatat kegiatan atau perubahan terakhir yang dilakukan pada kode. Prinsip ini bertujuan untuk memastikan bahwa komentar yang ditambahkan fokus pada memberikan pemahaman yang relevan dan berguna tentang kode, bukan catatan internal pengembang. Berikut contoh *Avoid Journal Comments*, dapat dilhat pada **Gambar 2.23**.

```
<?php
/**
 * * Changes (from 11-Oct-2001)
 * --------------------------
 * 11-Oct-2001 : *
 * 05-Nov-2001 : *
 * 12-Nov-2001 : *
 *
 *
 * 05-Dec-2001 :
 * 29-May-2002 : *
 * 27-Aug-2002 :
 * 03-Oct-2002 :
 * 13-Mar-2003 :
 * 29-May-2003 :
 * 04-Sep-2003 :
 * 05-Jan-2005 :
Re-organised the class and moved it to new package com.jrefinery.date (DG);
Added a getDescription() method, and eliminated NotableDate class (DG);
IBD requires setDescription() method, now that NotableDate class is gone (DG); Changed getPreviousDayOfWeek(),
getFollowingDayOfWeek() and getNearestDayOfWeek() to correct bugs (DG);
Fixed bug in SpreadsheetDate class (DG);
Moved the month constants into a separate interface (MonthConstants) (DG);
Fixed bug in addMonths() method, thanks to N???levka Petr (DG); Fixed errors reported by Checkstyle (DG);
Implemented Serializable (DG);
Fixed bug in addMonths method (DG);
Implemented Comparable. Updated the isInRange javadocs (DG); Fixed bug in addYears() method (1096282) (DG);
 *
```

Gambar 2.23 Contoh *Clean Comment Journal Comments*

d. *Commented-Out Code*

Commented Out merupakan prinsip dalam clean comments yang menganjurkan untuk menghindari penggunaan kode yang di-*comment out* (dihapus dengan tanda komentar). Kode yang di-*comment out* sering kali digunakan untuk sementara atau sebagai cadangan, dan tidak seharusnya ada dalam versi final kode yang diproduksi. Penggunaan kode yang di-*comment out* dapat membingungkan dan mempersulit pemahaman kode. Dalam prinsip "*clean comments*", disarankan untuk tidak meng-*comment out* kode yang tidak diperlukan. Berikut contoh *Commented-Out Code* dilhat pada **Gambar 2.24**.

```
48      public function addUser(Request $reqeust)
49      {
50          // // $validator = Validator::make($reqeust->all(), [
51          // //     'nama' => 'required|string|max:255',
52          // //     'username' => 'required|string|max:255|unique:users',
53          // //     'password' => 'required|string|min:8|confirmed',
54          // //     'hak' => 'required|string|max:255',
55          // // ]);
56
57          // if ($validator->fails()) {
58          //     return response()->json([
59          //         'status' => 'error',
60          //         'message' => $validator->errors()->first(),
61          //     ], 400);
62          // }
```

Gambar 2.24 Contoh *Clean Comment Commented-Out Code*

4.2.7 Formatting Code

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. Berikut merupakan petunjuk penulisan kode sumber dengan format penulisan yang baik:

1. *Vertical Openness Between Concepts*

Vertical Openness Between Concepts adalah prinsip dalam *formatting code* yang menekankan pentingnya memberikan ruang vertikal yang memadai antara konsep-konsep yang berbeda dalam kode. Ini berarti menggunakan baris kosong atau spasi tambahan untuk memisahkan blok-blok logis dalam kode. Berikut contoh *Vertical Openness Between Concepts* dilhat pada **Gambar 2.25**.

```php
<?php
function calculateSum($numbers) {
    $sum = 0;

    foreach ($numbers as $number) {
        $sum += $number;
    }

    return $sum;
}

function calculateAverage($numbers) {
    $count = count($numbers);

    if ($count === 0) {
        return 0;
    }

    $sum = calculateSum($numbers);
    $average = $sum / $count;

    return $average;
}

$myNumbers = [1, 2, 3, 4, 5];

$result = calculateAverage($myNumbers);

```

Gambar 2.25 Contoh *Formatting Code Vertical Openness Between Concepts*

2. *Vertical Density*

Vertical Density dalam *formatting code* merujuk pada penggunaan ruang vertikal secara efisien untuk menyusun konsep-konsep terkait dalam satu blok kode. Prinsip ini menekankan pada pengurangan ruang kosong yang tidak perlu dan penempatan informasi yang relevan secara berdekatan. Berikut contoh *Vertical Density* dilhat pada **Gambar 2.26.**

```php
<?php
function calculateSum($numbers) {
    $sum = 0;
    foreach ($numbers as $number) {
        $sum += $number;
    }
    return $sum;
}

function calculateAverage($numbers) {
    $count = count($numbers);
    if ($count === 0) {
        return 0;
    }
    $sum = calculateSum($numbers);
    $average = $sum / $count;
    return $average;
}

$myNumbers = [1, 2, 3, 4, 5];
$result = calculateAverage($myNumbers);
echo "Average: " . $result;

```

Gambar 2.26 Contoh *Formatting Code Vertical Density*

3. *Vertical Distance*

Vertical Distance dalam *formatting code* mengacu pada pemisahan visual antara konsep-konsep yang berbeda dalam satu blok kode. Prinsip ini mendorong penggunaan baris kosong untuk memisahkan logika atau elemen-elemen yang berbeda secara konseptual, sehingga meningkatkan kejelasan dan pemahaman kode. Berikut contoh *Vertical Distance* dilhat pada **Gambar 2.27**.

```php
<?php
function calculateSum($numbers) {
    $sum = 0;
    foreach ($numbers as $number) {
        $sum += $number;
    }
    return $sum;
}

function calculateAverage($numbers) {
    $count = count($numbers);

    if ($count === 0) {
        return 0;
    }

    $sum = calculateSum($numbers);
    $average = $sum / $count;

    return $average;
}

$myNumbers = [1, 2, 3, 4, 5];
$result = calculateAverage($myNumbers);
echo "Average: " . $result;

```

Gambar 2.27 Contoh *Formatting Code Vertical Distance*

4. *Horizontal Openness and Density*

Horizontal Openness and Density dalam *formatting code* mengacu pada pengaturan ruang *horizontal* di antara elemen-elemen kode. Prinsip ini bertujuan untuk meningkatkan kejelasan dan pemahaman kode dengan menggunakan spasi yang tepat antara operator, argumen, parameter, dan elemen-elemen kode lainnya. Berikut contoh *Horizontal Openness and Density* dilihat pada **Gambar 2.28**.

```php
<?php

function calculateDiscount($price, $discountPercentage) {
    $discountedPrice = $price - ($price * $discountPercentage / 100);
    return $discountedPrice;
}

$originalPrice = 1000;
$discountPercentage = 20;
$discountedPrice = calculateDiscount($originalPrice, $discountPercentage);

echo "Original Price: " . $originalPrice . "\n";
echo "Discount Percentage: " . $discountPercentage . "%\n";
echo "Discounted Price: " . $discountedPrice;

```

Gambar 2.28 ***Formatting Code Horizontal Openness and Density***

5. *Horizontal Alignment and Indentation*

Horizontal Alignment and Indentation dalam *formatting code* mengacu pada cara menyusun dan mengatur indentasi kode secara horizontal untuk meningkatkan keterbacaan dan kekonsistenan. Berikut contoh *Horizontal Alignment and Indentation* dilhat pada **Gambar 2.29**.

```
3   class FitNesseExpediter implements ResponseSender
4   {
5       private $socket;
6       private $input;
7       private $output;
8       private $request;
9       private $response;
10      private $context;
11      private $requestParsingTimeLimit;
12      private $requestProgress;
13      private $requestParsingDeadline;
14      private $hasError;
15
16      public function __construct($s, $context)
17      {
18          $this->context = $context;
19          $this->socket = $s;
20          $this->input = $s->getInputStream();
21          $this->output = $s->getOutputStream();
22          $this->requestParsingTimeLimit = 10000;
23      }
24
25  }
```

Gambar 2.29 Contoh *Formatting Code Horizontal Alignment and Indentation*

4.2.8 Error Handling

Pada konsep *Error Handling* terdapat berbagai cara yang efisien untuk menampilkan pesan kesalahan dengan mudah. Salah satu cara yang dianjurkan adalah dengan tidak mengabaikan kesalahan yang terjadi. Seringkali, kesalahan hanya dicatat dalam log tanpa tindakan lebih lanjut sehingga pengguna tidak tahu kesalahan apa yang terjadi. Oleh karena itu, diperlukan tindakan tambahan agar pesan kesalahan tidak hanya dicatat, tetapi juga dapat memberikan notifikasi baik kepada pengguna maupun pengembang tentang kesalahan yang terjadi. Berikut merupakan petunjuk bagaimana penanganan *error* yang dianjurkan menurut buku "*Clean Code*" Karya Robert C. Martin.

1. *Use Exceptions Rather than Return Codes*

Use Exceptions Rather than Return Codes adalah prinsip dalam penanganan kesalahan yang menyarankan penggunaan pengecualian (*exceptions*) daripada kode pengembalian (return codes) untuk menandakan kegagalan atau kondisi yang tidak

normal dalam sebuah fungsi atau metode. Berikut merupakan contoh sebelum dan sesudah implementasi *Exceptions* pada kode. Dapat dilihat pada **Gambar 2.30** **Gambar 2.31**.

```php
<?php

class DeviceController {
    public function sendShutDown() {
        $handle = $this->getHandle(DEV1); // Check the state of the device

        if ($handle != DeviceHandle::INVALID) {
            // Save the device status to the record field
            $this->retrieveDeviceRecord($handle);

            // If not suspended, shut down
            if ($this->record->getStatus() != DEVICE_SUSPENDED) {
                $this->pauseDevice($handle);
                $this->clearDeviceWorkQueue($handle);
                $this->closeDevice($handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " . DEV1.toString());
        }
    }

    // Other methods and class members...
}
```

Gambar 2.30 Contoh kode sebelum *implementasi exceptions*

```php
<?php

class DeviceController {
    public function sendShutDown() {
        try {
            $this->tryToShutDown();
        } catch (DeviceShutDownError $e) {
            logger.log($e);
        }
    }

    private function tryToShutDown() {
        $handle = $this->getHandle(DEV1);
        $record = $this->retrieveDeviceRecord($handle);
        $this->pauseDevice($handle);
        $this->clearDeviceWorkQueue($handle);
        $this->closeDevice($handle);
    }

    private function getHandle($id) {
        // ...
        throw new DeviceShutDownError("Invalid handle for: " . $id->toString());
        // ...
    }

    // Other methods and class members...
}
```

Gambar 2.31 Contoh kode sesudah *implementasi exceptions*

2. *Don't Return Null*

"*Don't return null*" merupakan prinsip dalam penanganan kesalahan yang menyarankan untuk menghindari pengembalian nilai null sebagai indikator kesalahan. Sebagai gantinya, gunakan mekanisme lain, seperti pengecualian (*exceptions*), untuk melaporkan dan menangani kesalahan. Berikut merupakan contoh sebelum dan sesudah implementasi *Exceptions* pada kode. Dapat dilihat pada **Gambar 2.32** dan **Gambar 2.33.**

```php
<?php

class Database {
    public function getConnection() {
        // Membuat koneksi ke database
        if ($connectionError) {
            return null;
        }

        return $connection;
    }
}

$db = new Database();
$connection = $db->getConnection();

if ($connection === null) {
    echo "Failed to connect to the database.";
} else {
    // Melakukan operasi database
}
```

Gambar 2.32 Contoh kode penggunaan *return null*

```php
<?php

class Database {
    public function getConnection() {
        // Membuat koneksi ke database
        if ($connectionError) {
            throw new DatabaseConnectionException("Failed to connect to the database.");
        }

        return $connection;
    }
}

try {
    $db = new Database();
    $connection = $db->getConnection();
    // Melakukan operasi database
} catch (DatabaseConnectionException $e) {
    echo "Error: " . $e->getMessage();
}
```

Gambar 2.33 Contoh kode penggunaan *return* tidak *null*

4.2.9 Object and Data Structures

Pada konsep Object and Data Structures, terdapat 2 konsep yang berbeda dalam pemrograman yang berkaitan dengan cara representasi dan interaksi antara data. Object Pada paradigma pemrograman berorientasi objek, objek merupakan sebuah entitas yang memiliki atribut (data) dan perilaku (metode). Objek digunakan untuk mewakili suatu konsep nyata atau abstrak dalam sistem. Sedangkan data *structure* (struktur data) merupakan cara penyimpanan, organisasi, dan pengaksesan data dalam sebuah program. Terdapat petunjuk penulisan atau implementasi kode sumber pendekatan *Object and Data Structures* yang dianjurkan.

2.13 *Data Abstraction*

Data Abstraction adalah konsep dalam pemrograman yang mengacu pada penyembunyian detail implementasi dan memberikan representasi yang lebih tinggi tingkat pada data atau objek. Ini berarti hanya mengekspos informasi yang penting atau relevan dan menyembunyikan detail yang kompleks atau tidak perlu. Dapat dilihat pada **Gambar 2.34.**

```php
<?php

abstract class Shape {
    abstract public function calculateArea();
    abstract public function calculatePerimeter();
}

class Circle extends Shape {
    private $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }

    public function calculateArea() {
        return pi() * $this->radius * $this->radius;
    }

    public function calculatePerimeter() {
        return 2 * pi() * $this->radius;
    }
}

class Rectangle extends Shape {
    private $length;
    private $width;

    public function __construct($length, $width) {
        $this->length = $length;
        $this->width = $width;
    }

    public function calculateArea() {
        return $this->length * $this->width;
    }

    public function calculatePerimeter() {
        return 2 * ($this->length + $this->width);
    }
}

$circle = new Circle(5.0);
echo "Circle Area: " . $circle->calculateArea() . "\n";
echo "Circle Perimeter: " . $circle->calculatePerimeter() . "\n";

$rectangle = new Rectangle(4.0, 6.0);
echo "Rectangle Area: " . $rectangle->calculateArea() . "\n";
echo "Rectangle Perimeter: " . $rectangle->calculatePerimeter() . "\n";
```

Gambar 2.34 Contoh implementasi *Data Abstraction*

2.14 *Data/Object Anti-Symmetry*

Data/Object Anti-Symmetry adalah konsep dalam paradigma pemrograman yang menekankan perbedaan peran antara objek dan data. Konsep ini menyatakan bahwa objek harus memiliki tingkat abstraksi dan kemampuan untuk melakukan tindakan, sementara data seharusnya hanya berfungsi sebagai penyimpan nilai dan tidak memiliki kemampuan untuk melakukan tindakan. Dapat dilihat pada **Gambar 2.35.**

```php
<?php

// Contoh data (Data)
class User {
  public $id;
  public $name;
  public $email;
}

// Contoh objek (Object)
class UserManager {
  public function createUser($name, $email) {
    // Logika bisnis untuk membuat user baru
    // ...

    // Membuat objek User
    $user = new User();
    $user->id = uniqid(); // Menghasilkan ID unik
    $user->name = $name;
    $user->email = $email;

    $this->saveUser($user);

    return $user;
  }

  private function saveUser(User $user) {
    // Logika untuk menyimpan data user ke dalam database
    // ...
  }
}

// Penggunaan objek UserManager
$userManager = new UserManager();
$newUser = $userManager->createUser("John Doe", "john@example.com");
echo "User created with ID: " . $newUser->id;

```

Gambar 2.35 Contoh implementasi *Data/Object Anti-Symmetry*

4.2.10 Clean Class

Clean Class adalah sebuah konsep dalam pengembangan perangkat lunak yang mengacu pada prinsip-prinsip pemrograman yang menjaga kelas-kelas agar tetap bersih, terorganisir, dan mudah dipahami. Tujuan utama *Clean Class* adalah menciptakan kelas-kelas yang memiliki tanggung jawab yang terfokus, kohesi yang tinggi, serta ketergantungan yang rendah antar kelas. Terdapat beberapa prinsip yang menciptakan desain yang lebih modular, fleksibel, mudah dipahami, dan mudah dipelihara.

1. *Single Responbility Principle* (SRP)

Single Responbility Principle (SRP) menyatakan bahwa sebuah kelas seharusnya hanya memiliki satu alasan untuk berubah. Artinya, setiap kelas seharusnya memiliki tanggung jawab tunggal yang terfokus. Dengan menerapkan SRP, kelas-kelas dapat menjadi lebih terorganisir, mudah dipahami, dan lebih mudah dipelihara. Prinsip ini membantu mengurangi ketergantungan antar kelas dan memastikan perubahan pada satu tanggung jawab tidak mempengaruhi tanggung jawab lainnya. Dapat dilihat pada **Gambar 2.36**.

```php
<?php

class SuperDashboard extends JFrame implements MetaDataUser {
    public function getCustomizerLanguagePath() {}
    public function setSystemConfigPath($systemConfigPath) {}
    public function getSystemConfigDocument() {}
    public function setSystemConfigDocument($systemConfigDocument) {}

    // ... metode-metode lainnya ...

    // Responsibilitas yang dapat diekstrak ke kelas terpisah
    public function showProgress($s) {}
    public function getLastFocusedComponent() {}
    public function setLastFocused($lastFocused) {}

    // ... metode-metode lainnya ...

}

class Version {
    public function isMetadataDirty() {}
    public function setIsMetadataDirty($isMetadataDirty) {}
    public function getProgramMetadata() {}

    // ... metode-metode lainnya ...
}
```

Gambar 2.36 Contoh *class* yang memenuhi prinsip SRP

Dalam contoh di atas, metode-metode yang berhubungan dengan informasi versi telah dipisahkan ke dalam *class* Version. Hal ini membantu memperjelas tanggung jawab masing-masing kelas dan memungkinkan penggunaan kembali Version dalam aplikasi lain jika diperlukan. Dengan memisahkan tanggung jawab dengan baik, serta dapat menciptakan abstraksi yang lebih jelas dan memudahkan pemeliharaan dan pengembangan kode.

2. *Open-Closed Principle* (OCP)

Prinsip OCP menyatakan bahwa sebuah entitas (kelas, modul, fungsi, dll.) seharusnya terbuka untuk perluasan (*open for extension*) tetapi tertutup untuk

modifikasi (*closed for modification*). Dalam konteks OCP, "terbuka" berarti entitas dapat diperluas dengan menambahkan fitur baru tanpa mengubah kode yang sudah ada. Prinsip ini mendorong penggunaan polimorfisme, warisan, dan abstraksi untuk mencapai fleksibilitas dalam perubahan fitur tanpa mengganggu kode yang sudah ada. Dapat dilihat pada **Gambar 2.37**.

```php
<?php

interface VersionProvider {
    public function getVersion();
}

class DefaultVersionProvider implements VersionProvider {
    public function getVersion() {}
}

class CustomVersionProvider implements VersionProvider {
    public function getVersion() {}
}

class SuperDashboard {
    private $versionProvider;

    public function __construct(VersionProvider $versionProvider) {
        $this->versionProvider = $versionProvider;
    }

    // Implementasikan metode-metode lain yang ada dalam SuperDashboard
    // dengan menggunakan $this->versionProvider untuk mendapatkan versi

    public function showProgress($s) {}
    public function getLastFocusedComponent() {}
    public function setLastFocused($lastFocused) {}
}
```

Gambar 2.37 Contoh *class* yang memenuhi prinsip OCP

Dalam contoh di atas, *class* SuperDashboard menggunakan VersionProvider sebagai dependensi. Ini memungkinkan untuk menyediakan implementasi yang berbeda dari VersionProvider seperti DefaultVersionProvider atau CustomVersionProvider tanpa harus mengubah kode sumber SuperDashboard. Oleh karena itu pengembang dapat mudah menambahkan implementasi lainnya

untuk mendapatkan versi yang berbeda tanpa memodifikasi SuperDashboard itu sendiri.

3. *Liskov Substitution Principle* (LSP)

Prinsip LSP menyatakan bahwa objek dari kelas turunan seharusnya dapat digunakan sebagai pengganti objek dari kelas induk tanpa mempengaruhi kebenaran dan konsistensi program. Artinya, jika ada hubungan warisan antara kelas-kelas, maka objek kelas turunan seharusnya dapat digunakan di mana pun objek kelas induk digunakan tanpa memerlukan perubahan dalam perilaku yang diharapkan. Prinsip ini menjamin bahwa polimorfisme dapat diterapkan dengan benar dan kelas-kelas turunan dapat mengikuti kontrak yang ditetapkan oleh kelas induk. Dapat dilihat pada **Gambar 2.38**.

```php
<?php

class SuperDashboard {
    public function showObject(MetaObject $object) {}
}

class SubDashboard extends SuperDashboard {
    // Implementasi khusus untuk SubDashboard
}

// Menggunakan objek SuperDashboard
$dashboard = new SuperDashboard();
$object = new MetaObject();
$dashboard->showObject($object);

// Menggunakan objek SubDashboard sebagai pengganti objek SuperDashboard
$subDashboard = new SubDashboard();
$subObject = new MetaObject();
$subDashboard->showObject($subObject);

```

Gambar 2.38 Contoh *class* yang memenuhi prinsip LSP

Dalam contoh di atas, terdapat kelas SuperDashboard yang memiliki metode showObject() untuk menampilkan objek MetaObject. Kemudian, dengan membuat kelas turunan SubDashboard yang menggantikan kelas SuperDashboard. Prinsip LSP diterapkan dengan menggunakan objek SubDashboard sebagai pengganti objek SuperDashboard dalam memanggil metode showObject(). Dalam hal ini, objek SubDashboard harus bisa menggantikan objek SuperDashboard tanpa mempengaruhi kebenaran program. Penerapan prinsip LSP memastikan bahwa objek SubDashboard tetap mematuhi kontrak atau perilaku yang didefinisikan oleh kelas SuperDashboard. Dengan demikian, penggunaan objek SubDashboard sebagai pengganti objek SuperDashboard tidak akan mempengaruhi fungsionalitas yang diharapkan dari metode showObject().

4. *Interface Segregation Principle* (ISP)

Prinsip ini menyatakan bahwa klien tidak boleh dipaksa untuk mengimplementasikan metode yang tidak mereka butuhkan [18]. Interface atau kontrak antara kelas-kelas harus cukup spesifik dan sesuai dengan kebutuhan setiap klien. Dengan menerapkan ISP, maka dapat menghindari dependensi yang tidak perlu dan membuat kode lebih fleksibel, Dapat dilihat pada **Gambar 2.39**.

```php
<?php

interface EmailSender {
    public function sendEmail($to, $subject, $message);
}

interface SmsSender {
    public function sendSms($to, $message);
}

class NotificationManager implements EmailSender, SmsSender {
    public function sendEmail($to, $subject, $message) {
        public function sendEmail($to, $subject, $message) {
            // Logika untuk mengirim email
        }
        public function sendSms($to, $message) {
            // Logika untuk mengirim SMS
        }
    }
}

```

Gambar 2.39 Contoh *class* yang memenuhi prinsip ISP

Dalam contoh diatas, dengan menggunakan dua interface terpisah, EmailSender dan SmsSender, yang mendefinisikan metode yang sesuai dengan fungsinya. Kelas NotificationManager mengimplementasikan kedua interface ini sesuai dengan kebutuhan.

5. *Dependency Inversion Principle* (DIP)

Prinsip DIP berfokus pada pemisahan ketergantungan antara modul dan kelas-kelas yang bergantung padanya. Prinsip ini menyatakan bahwa kelas-kelas seharusnya bergantung pada abstraksi, bukan implementasi. Dengan menerapkan DIP dapat menciptakan desain yang fleksibel dan mudah diubah, karena ketergantungan ditentukan oleh abstraksi yang stabil, bukan oleh implementasi yang berubah-ubah. Dapat dilihat pada **Gambar 2.40**.

```php
<?php

interface GUIFramework {
    public function show();
    public function setComponentSize(Dimension $dim);
}

interface MetadataUser {
    public function isMetadataDirty();
    public function setIsMetadataDirty($isMetadataDirty);
    public function getProgramMetadata();
}

class SuperDashboard {
    private $guiFramework;
    private $metadataUser;

    public function __construct(GUIFramework $guiFramework, MetadataUser $metadataUser) {
        $this->guiFramework = $guiFramework;
        $this->metadataUser = $metadataUser;
    }

    public function showProgress($s) {}

    public function getLastFocusedComponent() {}

    public function setLastFocused($lastFocused) {}
}
```

Gambar 2. 40 Contoh *class* yang memenuhi prinsip DIP

Dalam contoh di atas, kelas SuperDashboard bergantung pada dua abstraksi, yaitu GUIFramework dan MetadataUser. Dengan menggunakan abstraksi tersebut, kelas SuperDashboard tidak lagi bergantung pada implementasi spesifik seperti JFrame dan MetaDataUser. Ini memungkinkan fleksibilitas dalam penggunaan berbagai framework GUI dan implementasi MetadataUser yang berbeda.

2.10 Analisis dan Design Berorientasi Objek

OOAD *(Object Oriented Analysis and Design)* adalah metode baru untuk memecahkan masalah dengan menggunakan model berdasarkan konsep objek. Konsep objek sendiri menggabungkan struktur data dan perilaku menjadi satu kesatuan [19].

Dalam pengembangan perangkat lunak, penggunaan pendekatan berorientasi objek dapat membantu mengurangi kesulitan dalam melakukan transisi antar tahap. Hal ini dikarenakan notasi yang digunakan pada tahap analisis dan desain serta implementasi hampir sama, yang berbeda dengan pendekatan konvensional yang menggunakan notasi yang berbeda pada setiap tahap, sehingga membuat proses transisi menjadi lebih kompleks [19].

Menggunakan pendekatan berorientasi objek membantu pengguna untuk memahami konsep yang lebih dekat dengan dunia nyata, karena pada dunia nyata yang dilihat adalah objek, bukan fungsinya. Dalam pemrograman berorientasi objek, penekanan diberikan pada beberapa konsep seperti *Class, Object, Abstract, Encapsulation, Polymorphism, Inheritance*, dan UML *(Unified Modeling Language)* [19].

UML *(Unified Modeling Language)* adalah bahasa yang menggunakan grafik atau gambar untuk memvisualisasikan, menentukan spesifikasi, membangun, dan mendokumentasikan sebuah sistem pengembangan perangkat lunak berorientasi objek. [19]. Dokumentasi UML menyediakan 10 macam diagram untuk membuat model aplikasi berorientasi objek yang 4 di antaranya sebagai berikut:

1. Use Case Diagram

Diagram *Use Case* digunakan untuk menggambarkan fungsi yang diharapkan dari sebuah sistem, dengan fokus pada operasi sistem dan apa yang dapat dilakukannya. *Use Case* merepresentasikan interaksi antara sistem dan aktor, yang dapat berupa manusia atau mesin. *Use Case* ini merepresentasikan tugas tertentu seperti masuk ke sistem, mencetak dokumen, dan sejenisnya. [20].

2. Use Case Scenario

Walaupun diagram *Use Case* dan Aktor dapat menjadi awal yang baik dalam memahami sistem, namun hal tersebut belum memberikan informasi yang detail bagi desainer sistem. Oleh karena itu, cara terbaik untuk menggambarkan detail sistem secara rinci adalah dengan menggunakan skenario berbasis teks untuk setiap Use Case. [20].

3. Activity Diagram

Activity Diagram *(*Diagram Aktivitas*)* merupakan suatu representasi visual yang memusatkan perhatian pada proses bisnis dan urutan tindakan dalam suatu proses. Diagram ini sering digunakan untuk memodelkan bisnis dan menunjukkan urutan tindakan dalam proses bisnis. Diagram aktivitas mirip dengan aliran *chart* atau diagram alir data dalam pendekatan terstruktur dan dibuat berdasarkan satu atau beberapa kasus penggunaan pada diagram use case. [20].

4. Sequence Diagram

Sequence diagram menampilkan perilaku yang terjadi dalam sebuah skenario. Diagram ini memberikan gambaran yang jelas tentang objek dan pesan yang terlibat dalam sebuah use case. Komponen utama dari sequence diagram adalah objek yang digambarkan dengan bentuk kotak segitiga atau lingkaran, pesan yang digambarkan dengan garis putus-putus, dan waktu yang ditunjukkan dengan garis vertikal. Sequence diagram bermanfaat karena memberikan gambaran rinci dari setiap interaksi yang terjadi dalam use case diagram yang sebelumnya telah dibuat. [20].

5. *Class* Diagram

Class Diagram (Diagram Kelas) adalah diagram yang menggambarkan struktur dan deskripsi dari kelas, paket, dan objek, serta hubungan hubungannya. Ini juga menjelaskan interaksi antar kelas secara keseluruhan dalam sistem yang sedang dikembangkan untuk mencapai tujuan. [20].

2.11 *Readability Indeks*

Readability Indeks merupakan sebuah metrik yang digunakan untuk mengukur sejauh mana sebuah teks atau dokumen tertentu dapat dengan mudah dibaca dan dipahami. *Readbility Indeks* juga dapat memiliki relevansi dengan keterbacaan kode (*code readability*). *Code readability* memiliki aspek yang krusial dalam menulis kode program karena memiliki dampak penting pada biaya pemeliharaan perangkat lunak (*maintaince*) [1]. Penilaian keterbacaan kode dilakukan melalui wawancara dengan pengembang untuk mengevaluasi struktur,

logika, dan pemahaman kode yang dibuat. Berikut merupakan tahapan pada saat dilakukan wawancara kepada pengembang:

1. Penguji menceritakan terlebih dahulu tujuan, kegunaan dari fungsionalitas sistem AuroraViews. Penguji tidak menjelaskan kode.
2. Penguji memberikan *source code* mana yang digunakan serta memberikan form untuk meminta penilaian kepada pengembang dengan rentang satu sampai empat. Satu artinya sangat sulit dibaca. Dua artinya sulit dibaca. Tiga artinya mudah dibaca. Empat artinya sangat mudah dibaca.
3. Penguji tidak menjawab apa pun ketika pengembang sedang memikirkan atau bertanya.
4. Pada tahap akhir penguji merekap hasil penilaian yang telah diberikan melalui form.

Setelah wawancara, hasil penilaian keterbacaan kode dikelompokkan berdasarkan klasifikasi yang tercantum dalam **Tabel 2.3** [1].

Tabel 0.3 Klasifikasi *Readability*

Nilai	Keterangan	Klasifikasi
1	Sangat sulit dibaca	Kode tidak dapat dibaca
2	Sulit dibaca	Kode tidak dapat dibaca
3	Mudah dibaca	Kode dapat dibaca
4	Sangat mudah dibaca	Kode dapat dibaca

2.12 Integration Testing

Integration Testing adalah metode sistematis yang digunakan dalam pengembangan perangkat lunak untuk membangun arsitektur sementara sambil secara simultan melakukan pengujian dengan tujuan mengidentifikasi potensi kesalahan yang terkait dengan antarmuka perangkat lunak [27].

Integration Testing bertujuan untuk mengidentifikasi masalah atau bug yang muncul ketika komponen yang berbeda digabungkan dan berinteraksi satu sama lain. *Integration Testing* dilkakukan setelah pengujian unit dan sebelum penguhian sistem [28]. Terdapat empat jenis pendekatan *Integration Testing* yaitu sebagai berikut:

1. *Big-Bang Integration Testing*

Integration Testing Big-Bang merupakan pendekatan di mana semua komponen atau modul aplikasi perangkat lunak digabungkan dan diuji sekaligus. Pendekatan ini biasanya digunakan ketika komponen perangkat lunak memiliki tingkat ketergantungan yang rendah atau ketika ada kendala dalam lingkungan pengembangan yang mencegah pengujian komponen individual. Tujuan dari pengujian integrasi big-bang adalah untuk memverifikasi fungsionalitas sistem secara keseluruhan dan untuk mengidentifikasi masalah integrasi yang muncul ketika komponen-komponen tersebut digabungkan [28].

2. *Bottom-Up Integration Testing*

Bottom-Up Integration Testing merupakan pendekatan dimana setiap modul pada tingkat yang lebih rendah diuji dengan modul yang lebih tinggi sampai semua modul diuji. Tujuan utama dari pengujian integrasi ini adalah setiap subsistem menguji antarmuka di antara berbagai modul yang membentuk subsistem. Pengujian integrasi ini menggunakan driver pengujian untuk menggerakkan dan meneruskan data yang sesuai ke modul tingkat yang lebih rendah [28].

3. *Top-Down Integration Testing*

Top-Down Integration Testing digunakan untuk mensimulasikan perilaku modul-modul tingkat bawah yang belum terintegrasi. Dalam pengujian integrasi ini, pengujian dilakukan dari atas ke bawah. Pertama, modul tingkat tinggi diuji dan kemudian modul tingkat rendah dan akhirnya mengintegrasikan modul tingkat rendah ke tingkat tinggi untuk memastikan sistem bekerja sebagaimana mestinya [28].

4. *Mixed Integration Testing*

Mixed Integration Testing dapat disebut pengujian integrasi terjepit, dikarenakan mengikuti kombinasi pendekatan pengujian dari atas ke bawah dan dari bawah ke atas. Dalam pendekatan *Top-Down Integration Testing*, pengujian dapat dimulai hanya setelah modul tingkat atas dikodekan dan diuji unit. Dalam pendekatan

Bottom-Up Integration Testing, pengujian dapat dimulai hanya setelah modul tingkat bawah siap. Pendekatan sandwich atau campuran ini mengatasi kekurangan dari pendekatan *Top-Down Integration Testing* dan *Bottom-Up Integration Testing* [28].