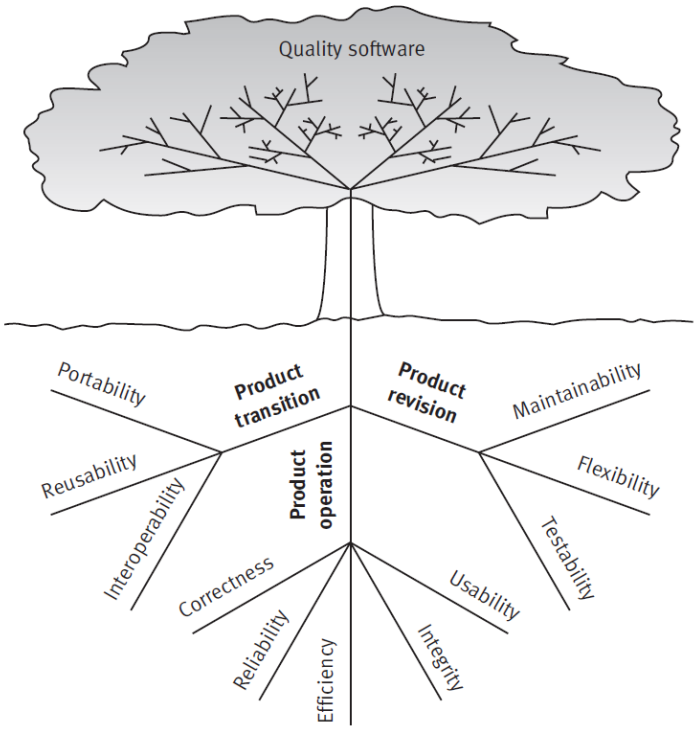
BAB 2
LANDASAN TEORI

2.1 Software Quality Assurance

Software Quality Assurance adalah suatu sistem yang terdiri dari serangkaian proses untuk memastikan bahwa suatu produk atau barang memenuhi spesifikasi teknis yang telah ditentukan (IEEE, 1991)[3]. Perangkat lunak dibuat dengan memperhatikan berbagai macam faktor kualitas. Faktor kualitas tersebut dapat dijelaskan oleh banyak model kualitas, salah satunya adalah Model McCall. Model kualitas McCall membagi kualitas perangkat lunak menjadi 3 bagian utama, yaitu: Product Revision (Maintainability, Flexibility, dan Testability), Product Transition (Portability, Reusability, dan Interoperability), dan Product Operation (Correctness, Reliability, Efficiency, Integrity, dan Usability[3], [4]. Pembagian tersebut dapat dilihat pada gambar berikut.



Gambar 2.1 Pembagian faktor kualitas McCall

Dari masing – masing faktor kualitas yang terlihat pada gambar 2.1. Pembagian faktor kualitas McCall, terdapat beberapa sub faktor yang

memperjelas pengujian oleh setiap faktor, seperti dapat dilihat pada tabel berikut:

Tabel 2.1 Faktor Kualitas McCall

Kategori Model McCall	Faktor Kualitas Perangkat Lunak	Sub Faktor
Product Revision	*Maintainability*	*Simplicity*
		Modularity
		Self-descriptiveness
		Coding and documentation guidelines
		Compliance (consistency)
		Document accessibility
	Flexibility	*Modularity*
		Generality
		Simplicity
		Self-descriptiveness
	Testability	*User testability*
		Failure maintenance testability
		Traceabillity
Product Operation	*Correctness*	*Accuracy*
		Completeness
		Up-to-dateness
		Availability
		Coding and documentation guidelines
		Compliance (consistency)

Kategori Model McCall	Faktor Kualitas Perangkat Lunak	Sub Faktor
	Reliability	*System realiability*
		Application reliability
		Computational failure recovery
		Hardware failure recovery
	Efficiency	*Efficiency of processing*
		Efficiency of storage
		Efficiency of communication
		Efficiency of power usage
	Integrity	*Access control*
		Access audit
	Usability	*Operability*
		Training
Product Transition	*Portability*	*Software system independence*
		Modularity
		Self descriptive
	Reusability	*Modularity*
		Document accessibility
		Software system independence
		Application independence
		Self descriptive
		Generality
		Simplicity

Kategori Model McCall	Faktor Kualitas Perangkat Lunak	Sub Faktor
	Interoperability	*Commonality*
		System compatibility
		Software system independence
		Modularity

2.2 Maintainability

Maintanability adalah salah satu faktor pada SQA yang berfokus pada kemudahan dalam pemeliharaan suatu perangkat lunak. Maintainability menentukan usaha yang dibutuhkan oleh tim developer untuk mengidentifikasi penyebab kesalahan pada software, memperbaiki dan melakukan pengujian terhadap kesalahan yang dilakukan[3]. Faktor ini mengacu pada struktur modul perangkat lunak, dokumentasi program internal dan dokumen manual pengembang[3].

Berdasarkan model kualitas McCall, faktor *maintainability* ini terdiri atas beberapa sub faktor seperti pada tabel berikut:

Tabel 2.2 Faktor Maintainability

Faktor Kualitas Perangkat Lunak	Sub Faktor	Penjelasan
Maintainability	*Simplicity*	Mengacu pada tingkat kemudahan dalam memahami perangkat lunak
	Modularity	Mengacu pada ukuran modul perangkat lunak
	Self-descriptiveness	Mengacu pada kemudahan dalam memahami kode sumber

Faktor Kualitas Perangkat Lunak	Sub Faktor	Penjelasan
	Coding and documentation guidelines	Mengacu pada ketersediaan dokumentasi perangkat lunak dan petunjuk penulisan kode
	Compliance (consistency)	Mengacu pada keseragaman standar desain dan teknik implementasi
	Document accessibility	Mengacu pada kemudahan mendapatkan dokumentasi perangkat lunak

2.3 *Maintainability Index*

Maintainability Index (MI) merupakan suatu metrik yang dapat mengukur seberapa mudah suatu kode untuk dipelihara atau dimaintain[7], [8]. *Maintainability Index* terdiri dari metrik *Cyclomatic Complexity* (CC), metrik *Halstead Volume* (HV), metrik *Lines of Code* (LOC), dan metrik persentase jumlah komentar per modul (COM)[9]. Adapun formula untuk menghitung *Maintainability Index* menurut *Software Engineering Institute* (SEI) dapat diihat pada gambar berikut[8].

$$MI = \left(\left(171 - (5{,}2 * \ln(V)) - (0{,}23 * CC) - (16.2 * \ln(LoC))\right) * \frac{100}{171} + 50 * \sin(\sqrt{2{,}4 * CM}) \right)$$

Namun formula untuk menghitung *Maintainability Index* memiliki banyak varian. Seperti formula yang digunakan oleh Microsoft pada Aplikasi Visual Studio dapat dilihat seperti pada gambar berikut[10].

$$MI = \frac{171 - (5{,}2 * \ln(V)) - (0{,}23 * CC) - (16.2 * \ln(LoC))}{171} * 100$$

Dimana $0 \leq MI \leq 100$

Formula *Maintainability Index* yang akan digunakan pada penelitian ini adalah formula yang dirumuskan oleh SEI. Secara umum, nilai dari *Maintainability Index* diukur dari 0 sampai 100. Semakin tinggi nilai dari *Maintainability Index* tersebut, semakin tinggi tingkat maintainability dari kode sumber yang diukur. Nilai tersebut dibagi menjadi tiga kategori yang dapat dilihat pada tabel berikut[9]

Tabel 2.3 Rentang nilai *Maintainability Index*

Rentang	Keterangan
> 85	High Maintainability
64 < MI < 84	Medium Maintainability
< 64	Low Maintainability

2.4 Cyclomatic Complexity

Cyclomatic Complexity adalah metrik yang digunakan untuk mengukur tingkat kompleksitas pada sebuah fungsi dengan mempertimbangkan grafik kendali. Secara singkat, jika sebuah fungsi tidak memiliki percabangan, maka tingkat kompleksitasnya adalah 1[11], [12]. Formula untuk menghitung Cyclomatic Complexity dapat dilihat pada gambar berikut:

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

Keterangan:

M = Cyclomatic Complexity

E = Jumlah edge pada graf kendali

N = Jumlah node pada graf kendali

P = Jumlah komponen yang terhubung

2.5 Halstead Metrics

Halstead Complexity adalah satu dari beberapa metrik yang digunakan untuk mengukur tingkat kompleksitas perangkat lunak dengan menghitung jumlah operator dan operan yang ada dalam sebuah class atau modul[13]. Pengukuran Halstead Metric terdiri dari beberapa metrik, yaitu:

2.5.1 Operand dan Operator

Operator pada perhitungan terdiri atas operator matematika seperti for, if dan var, simbol tanda kurung, tanda koma, dan titik koma. Sedangkan operan adalah variabel, konstanta, angka ataupun karakter. Berikut merupakan simbol untuk menggambarkan operan dan operator

$$n_1 = jumlah\ operator\ unik$$
$$n_2 = jumlah\ operand\ unik$$
$$N_1 = jumlah\ operator\ keseluruhan$$
$$N_2 = jumlah\ operand\ keseluruhan$$

2.5.2 Program Length

Program Length merupakan total penjumlahan operator dan operand.

$$N = N_1 + N_2$$

2.5.3 Program Vocabulary

Program Vocabulary adalah jumlah dari operator dan operan unik yang ada dalam sebuah program.

$$n = n_1 + n_2$$

2.5.4 Volume

Volume adalah ukuran dari implementasi suatu algoritma dalam sebuah program. Perhitungan volume didasarkan pada jumlah operator yang digunakan dan operan yang dipakai dalam algoritma, dengan rumus sebagai berikut:

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

Jika volume sebuah fungsi melebihi 1000, hal ini menunjukkan bahwa fungsi tersebut mungkin memiliki terlalu banyak tugas yang harus dilakukan.

2.5.5 Difficulty

Difficulty digunakan untuk menentukan tingkat rawan terjadinya kesalahan dalam sebuah program dengan menghitung proporsi operator unik dan rasio antara jumlah total operan dan operan unik dalam program. Berikut adalah rumusnya:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

2.5.6 Effort to Implement

Metrik ini merupakan usaha untuk memahami program berdasarkan nilai Volume dan Difficulty sebelumnya, dengan rumus sebagai berikut:

$$E = V * D$$

2.5.7 Time to Implement

Metrik ini digunakan untuk mengukur waktu dari *Effort to Implement*. Berikut rumusnya:

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

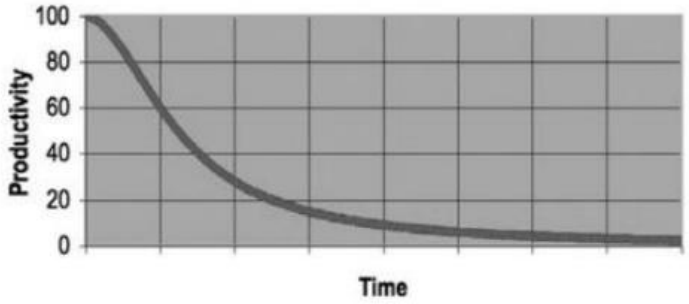
2.5.8 Number of Delivered Bugs

Metrik ini digunakan untuk menguku jumlah bug yang terjadi dengan rumus sebagai berikut:

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

2.6 *Clean code*

Clean code adalah suatu konsep penulisan kode perangkat lunak yang baik, dimana kode tersebut dapat dibaca oleh pengembang lain sehingga memudahkan dalam proses pemeliharaan dan pengembangan. *Clean code* bertujuan untuk mengatasi penurunan produktivitas dari pengembangan perangkat lunak karena kode yang tidak terstandar[14]. Hubungan antara produktivitas dan waktu dapat dilihat pada gambar di bawah.



Gambar 2.2 Pengaruh produktivitas terhadap waktu

Clean code memiliki beberapa kategori yang dijadikan sebagai pedoman penulisan kode, yaitu:

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

2.6.1 Meaningful Names

Meaningful Names atau nama yang bermakna merupakan salah satu prinsip penting dalam *clean code*. Penggunaan nama yang bermakna pada variabel, fungsi, kelas dan metode sangat penting untuk memudahkan pembacaan dan pemeliharaan kode. Berikut merupakan panduan mengenai *meaningful names*:

1. Use Intention-revealing Names

   Nama variabel harus bisa mencerminkan apa yang disimpan di dalamnya. Ini bertujuan agar menghindari kesulitan dalam memahami fungsi dari kode tersebut. Contoh baik dan buruk dapat dilihat pada contoh di bawah.

Tabel 2.4 Contoh Good Code Use Intention-revealing Names

Good Code	$kodeProduk = ‘ghj123’;

Tabel 2.5 Contoh Good Code Use Intention-revealing Names

Bad Code	$data = ‘ghj123’;

2. Use Searchable Names

   Membaca kode lebih sering daripada menulis kode. Oleh karena itu sangat penting untuk membuat nama yang mudah dicari. Ini akan mempercepat proses pemahamaan keseluruhan program, seperti pada contoh berikut.

Tabel 2.6 Contoh Good Code Use Searchable Names

Good Code	class KeranjangBelanja {
	  public function tambah_produk($produk) {
	  // ...
	  }
	}

Tabel 2.7 Contoh Bad Code Use Searchable Names

Bad Code	class A {
	  public function fn() {
	  // ...
	  }
	}

3. Avoid Mental Mapping

   Untuk memudahkan orang lain memahami kode, disarankan untuk menghindari penggunaan singkatan dalam penamaan variabel. Ini menjadi semakin penting ketika bekerja dalam tim atau berkolaborasi dengan orang lain dalam pengembangan proyek. Oleh karena itu, sebaiknya pengembang menggunakan nama variabel yang jelas dan mudah dipahami untuk memastikan bahwa kode yang ditulis mudah dimengerti oleh orang lain. Berikut contoh dari mental mapping:

Tabel 2.8 Contoh Good Code Avoid Mental Mapping

Good Code	$locations = ['Jakarta', 'Bandung', 'Tasikmalaya'];
	
	foreach ($locations as $location) {
	  // ...
	  dispatch($location);
	}

Tabel 2.9 Contoh Bad Code Avoid Mental Mapping

Bad Code	$l = ['Jakarta', 'Bandung', 'Tasikmalaya'];
	
	for ($i = 0; $i < count($l); $i++) {
	  $li = $l[$i];
	  // ...
	  dispatch($li);
	}

4. Avoid Disinformation

Avoid disinformation mengacu pada ide bahwa dalam pemberian nama pada variabel, fungsi, kelas, atau elemen lain dalam kode, sebaiknya menghindari memberikan nama yang menyesatkan atau menimbulkan kesalahpahaman tentang tujuan atau fungsi sebenarnya dari entitas tersebut.

Dalam prinsip ini, penting untuk memilih nama yang mencerminkan dengan jelas tujuan dan fungsionalitas entitas tersebut, sehingga pembaca kode dapat dengan mudah memahaminya tanpa merasa bingung atau tertipu oleh nama yang ambigu atau tidak sesuai. Berikut merupakan contoh *avoid disinformation*.

Tabel 2.10 Contoh Good Code Avoid Disinformation

Good Code	function calculateTotalPrice($items) {
	$totalPrice = 0;
	foreach ($items as $item) {
	$totalPrice += $item['price'];
	}
	return $totalPrice;
	}

Tabel 2.11 Contoh Bad Code Avoid Disinformation

Bad Code	function sum($arr) {
	$result = 0;
	foreach ($arr as $i) {
	$result += $i;
	}
	return $result;
	}

5. Avoid Encoding

Prinsip ini adalah menghindari pengkodean (encoding) informasi tambahan dalam nama variabel, fungsi, atau elemen lain dalam kode. Informasi tambahan seperti tipe data, skala, atau status seharusnya tidak diungkapkan melalui pengkodean dalam nama, tetapi seharusnya melalui pendekatan yang lebih jelas dan eksplisit.

Tujuan dari prinsip ini adalah agar kode lebih mudah dibaca dan dipahami tanpa harus menerjemahkan atau mendekode informasi yang tersembunyi dalam nama variabel atau elemen. Berikut merupakan contoh *avoid encoding*.

Tabel 2.12 Contoh Good Code Avoid Encoding

Good Code	function getActiveUsers($users) {
	$activeUsers = [];
	foreach ($users as $user) {
	if ($user['status'] === 'active') {
	$activeUsers[] = $user;
	}
	}
	return $activeUsers;
	}

Tabel 2.13 Contoh Bad Code Avoid Encoding

Bad Code	function getActiveUsersArray($arrUsers) {
	$activeUsersArray = [];
	foreach ($arrUsers as $usr) {
	if ($usr['s'] === 'active') { // 's' sebagai encoding untuk 'status'
	$activeUsersArray[] = $usr;
	}
	}
	return $activeUsersArray;
	}

6. Use Pronounceable Names

Prinsip ini dalam pemrograman menekankan pentingnya memberikan nama – nama yang dapat diucapkan dengan mudah. Tujuannya adalah untuk memudahkan komunikasi antar pengembang saat membaca atau berbicara tentang kode, serta membantu mencegah kesalahan penulisan atau pelafalan yang dapat muncul akibat nama yang sulit diucapkan.

Tabel 2.14 Contoh Good Code Use Pronounceable Names

Good Code	$ transactionCount = 10;

Tabel 2.15 Contoh Bad Code Use Pronounceable Names

Bad Code	$ txnCnt = 10;

7. Penggunaan kata benda untuk modul/*class*

   Penggunaan kata benda pada sebuah *class*/modul merupakan hal yang disarankan. Hal ini karena suatu class atau modul pada umumnya digunakan untuk merepresentasikan suatu entitas atau objek tertentu, sehingga menggunakan kata benda akan lebih sesuai dan mudah dipahami. Berikut contoh *good code* dan *bad code*.

Tabel 2.16 Contoh Good Code Penggunaan Kata Benda pada Kelas

Good Code	class PersegiPanjang {
	function hitungLuas($panjang, $lebar) {
	return $panjang * $lebar;
	}
	}

Tabel 2.17 Contoh Bad Code Penggunaan Kata Benda pada Kelas

Bad Code	class Hitung {
	function rumus($a, $b) {
	return $a * $b;
	}
	}

8. Penggunaan kata kerja untuk fungsi/*method*

   Dalam *clean code*, disarankan untuk menggunakan kata kerja pada nama fungsi/method, dan menghindari penggunaan kata benda. Hal ini

karena nama fungsi yang mengandung kata kerja akan lebih menjelaskan apa yang dilakukan oleh fungsi tersebut, sehingga membuat kode menjadi lebih mudah dipahami oleh programmer lain. Selain itu, penggunaan kata kerja pada nama fungsi juga akan memudahkan programmer dalam melakukan debugging dan maintenance code. Berikut contoh good code dan bad code.

Tabel 2.18 Contoh Good Code Penggunaan Kata Kerja pada Fungsi

Good Code	class Pengguna {
	function getData() {
	// ...
	}
	}

Tabel 2.19 Contoh Bad Code Penggunaan Kata Kerja pada Fungsi

Bad Code	class Pengguna {
	function ambilData() {
	// ...
	}
	}

9. Make Meaningful Distinctions

Penamaan atribut menggunakan nama class yang terletak pada class tersebut tidak disarankan karena dapat terlihat redundan. Berikut contoh yang dapat dilihat pada gambar di bawah:

Tabel 2.20 Contoh Good Code Make Meaningful Distinctions

Good Code	class Motorcycle
	{
	public $make;
	public $model;
	public $color;
	}

Tabel 2.21 Contoh Bad Code Make Meaningful Distinctions

Bad Code	class Motorcycle
	{
	public $motorcycleMake;
	public $motorcycleModel;
	public $motorcycleColor;
	}

2.6.2 Clean Functions

Dalam buku "Clean Code: A Handbook of Agile Software Craftsmanship" yang ditulis oleh Robert C. Martin, *clean functions* merupakan suatu konsep yang menyatakan bahwa fungsi harus mudah dibaca, dan mudah dipahami. *Clean functions* memiliki beberapa karakteristik atau prinsip yaitu:

1. Function Argument

Tabel 2.22 Contoh Good Code Function Argument

Good Code	function hitung($bilangan_pertama, $bilangan_kedua)
	{
	$hasil = ($bilangan_pertama + $bilangan_kedua) * PI;
	return $hasil;
	}

Tabel 2.23 Contoh Bad Code Function Argument

Bad Code	function hitung($a, $b, $c, $d, $e)
	{
	$hasil = ($a + $b) * $c - $d / $e;
	return $hasil;
	}

Pada contoh kode *bad code*, terdapat banyak parameter yang digunakan dalam fungsi "hitung" yaitu 5 parameter. Jumlah parameter yang

terlalu banyak ini dapat menyulitkan penggunaan fungsi tersebut, terutama saat melakukan pengujian dan *debugging*.

Pada contoh kode *good code*, fungsi "hitung" hanya menggunakan 2 parameter yang menjelaskan jenis data yang diharapkan. Selain itu, fungsi tersebut juga menggunakan konstanta *phi* sebagai nilai tetap, sehingga tidak memerlukan parameter lagi. Dengan demikian, kode tersebut lebih mudah dipahami dan diuji karena hanya menggunakan jumlah parameter yang ideal.

Dalam *clean code*, disarankan untuk meminimalkan jumlah parameter fungsi, agar lebih mudah dipahami dan diuji. Jika terdapat kebutuhan untuk menggunakan lebih dari 2 parameter, maka sebaiknya gunakan objek sebagai parameter pada fungsi tersebut.

2. Do One Thing

Tabel 2.24 Contoh Good Code Do One Thing

Good Code	function hitungJumlah($angka)
	{
	return array_sum($angka);
	}

Tabel 2.25 Contoh Bad Code Do One Thing

Bad Code	function hitungJumlah($angka)
	{
	$total = 0;
	foreach ($angka as $a) {
	$total += $a;
	}
	return $total;
	}

Pada contoh kode *bad code*, fungsi "hitungJumlah" tidak hanya menghitung jumlah nilai dalam array, tetapi juga melakukan perulangan

untuk melakukan penjumlahan. Hal ini dapat membuat fungsi terlihat rumit dan sulit untuk dibaca dan dipahami.

Pada contoh kode *good code*, fungsi "hitungJumlah" hanya memanggil fungsi bawaan PHP yaitu *array_sum* untuk menghitung jumlah nilai dalam array. Dengan menggunakan fungsi bawaan PHP, kode menjadi lebih mudah dipahami dan terlihat lebih sederhana.

3. Small

Prinsip ini menekankan pentingnya menjaga fungsi-fungsi dalam kode agar tetap kecil dan berfokus pada satu tugas atau tanggung jawab tertentu. Fungsi-fungsi yang kecil lebih mudah dipahami, diuji, dikelola, dan digunakan ulang. Hal ini juga membantu menjaga kode tetap terorganisir dan mendorong pemisahan tugas yang jelas antara berbagai bagian kode.

Tabel 2.26 Contoh Good Code Prinsip Small

Good Code	function validateEmail($email) {
	  return filter_var($email, FILTER_VALIDATE_EMAIL);
	}
	
	function generateUserId($userName) {
	  return strtolower(str_replace(" ", "_", $userName));
	}

Tabel 2.27 Contoh Bad Code Prinsip Small

Bad Code	function processUser($userData) {
	  // Performs user data validation, calculates total, and generates user ID
	  $total = 0;
	  $valid = false;
	  $userId = '';
	  // ... perform various actions ...
	  return [
	  'total' => $total,

	'valid' => $valid, 'userId' => $userId,]; }

Pada contoh *bad code* terlihat bahwa fungsi processUser() melakukan beberapa tugas dalam satu fungsi. Hal ini tidak disarankan dalam prinsip *clean functions*.

Kemudian pada contoh *good code* terlihat bahwa setiap fungsi hanya mengerjakan satu tugas spesifik sehingga fungsi menjadi lebih rapi dan kecil.

4. Block and Indenting

Prinsip ini merujuk pada cara kita mengelompokkan kode ke dalam blok yang terstruktur dan menggunakan indentasi yang konsisten untuk meningkatkan keterbacaan dan pemahaman kode. Dengan mengatur kode dalam blok yang jelas dan menggunakan indentasi yang benar, kita membuat struktur kode menjadi lebih mudah diikuti dan dipahami oleh programmer lain. Berikut merupakan contoh *good code* dan *bad code*.

Tabel 2.28 Contoh Good Code Block and Indenting

Good Code	function calculateTotalPrice($items) {
	$totalPrice = 0;
	foreach ($items as $item) {
	$totalPrice += $item['price'];
	}
	return $totalPrice;
	}

Tabel 2.29 Contoh Bad Code Block and Indenting

Bad Code	function calculateTotalPrice($items) {
	$totalPrice = 0;
	foreach ($items as $item) {
	$totalPrice += $item['price'];

	} return $totalPrice; }

5. Stepdown Rule

Prinsip *Stepdown Rule* menjelaskan bagaimana menjaga tingkat abstraksi yang konsisten. Prinsip ini mengacu pada cara mengatur kode sehingga kode menjadi lebih mudah dibaca dan dipahami dimulai dengan tingkat abstraksi yang lebih tinggi ke tingkat abstraksi yang lebih rendah.

Pada dasarnya, prinsip ini mendorong penggunaan fungsi atau metode yang lebih tinggi untuk mengatur logika, dan kemudian menggabungkannya dengan fungsi atau metode yang lebih rendah.

Tabel 2.30 Contoh Good Code Stepdown Rule

Good Code	function processOrder($order) {
	  validateOrder($order);
	  calculateTotal($order);
	  generateInvoice($order);
	}
	function validateOrder($order) {
	  // Validate order details
	  // ...
	}
	function calculateTotal($order) {
	  // Calculate total price
	  // ...
	}
	function generateInvoice($order) {
	  // Generate invoice
	  // ...
	}

Tabel 2.31 Contoh Bad Code Stepdown Rule

Bad Code	function processOrder($order) {
	// Validate order details
	if ($order['quantity'] <= 0) {
	// Handle invalid order
	// ...
	}
	// Calculate total price
	$totalPrice = $order['quantity'] * $order['price'];
	
	// Generate invoice
	$invoice = [
	'total' => $totalPrice,
	'details' => $order['details']
	];
	}

6. No Side Effects

   Side Effects terjadi ketika fungsi melakukan lebih dari sekadar menghitung atau mengembalikan nilai. *Side effect* bisa berupa perubahan yang mempengaruhi keadaan di luar fungsi, seperti memodifikasi variabel global, mengubah struktur data, atau berinteraksi dengan sumber daya eksternal seperti *database*.

Tabel 2.32 Contoh Good Code No Side Effects

Good Code	function calculateTotal($items) {
	$total = 0;
	foreach ($items as $item) {
	$total += $item['price'];
	}
	return $total;
	}
	}

Tabel 2.33 Contoh Bad Code No Side Effects

Bad Code	$globalTotal = 0;
	
	function addToTotal($amount) {
	  global $globalTotal;
	  $globalTotal += $amount;
	}
	
	function updateDatabase($data) {
	  // ... perform database update ...
	}

2.6.3 Clean Comments

Clean comments adalah konsep dalam menulis komentar pada kode sumber yang bertujuan untuk memberikan informasi tambahan yang efektif dan informatif. Pentingnya penggunaan penamaan yang jelas dan bermakna dapat mengurangi jumlah komentar yang dibutuhkan untuk menjelaskan kode. Berikut adalah beberapa petunjuk dalam penulisan komentar:

1. Explain Yourself in Code

   Prinsip ini merujuk pada komentar dalam kode program seharusnya tidak hanya menjelaskan apa yang dilakukan oleh kode tersebut, tetapi juga mengapa kode tersebut dilakukan dan bagaimana cara kerjanya. Dengan kata lain, komentar harus memberikan pemahaman yang lebih mendalam tentang tujuan dan konteks dari kode tersebut, sehingga orang lain yang membaca kode tersebut dapat dengan mudah memahami alur dan tujuannya. Berikut merupakan contoh kode *explain yourself in code*.

Tabel 2.34 Contoh Good Code Explain Yourself in Code

Good Code	// Calculate the total price including tax
	$subtotal = 100;
	$taxRate = 0.08;
	$taxAmount = $subtotal * $taxRate;
	$totalPrice = $subtotal + $taxAmount;
	
	// Display the total price to the user
	echo "Subtotal: $" . $subtotal . "\n";
	echo "Tax: $" . $taxAmount . "\n";
	echo "Total Price: $" . $totalPrice . "\n";

Tabel 2.35 Contoh Bad Code Explain Yourself in Code

Bad Code	$st = 200; // set subtotal
	$tr = 0.1; // set tax rate
	$ta = $st * $tr; // calculate tax amount
	$tp = $st + $ta; // calculate total price
	
	echo $tp; // display total price

2. Informative Comments

Informative comments mengacu pada penulisan komentar yang memberikan informasi yang berguna dan relevan bagi pembaca kode. Tujuannya adalah untuk memberikan pemahaman lebih lanjut tentang bagaimana kode bekerja.

Tabel 2.36 Contoh Good Code Informative Comments

Good Code	/** * Calculate the total price including tax. * * This function takes a subtotal and a tax rate as input, * calculates the tax amount, and returns the total price.

	*
	* @param float $subtotal The subtotal amount before tax.
	* @param float $taxRate The tax rate as a decimal (e.g., 0.08 for 8%).
	* @return float The total price including tax.
	*/
	function calculateTotalPrice($subtotal, $taxRate) {
	$taxAmount = $subtotal * $taxRate;
	$totalPrice = $subtotal + $taxAmount;
	return $totalPrice;
	}

Tabel 2.37 Contoh Bad Code Informative Comments

Bad Code	// Calculate total
	$t = $s * 0.1; // 10% tax
	$total = $s + $t; // total price
	
	// Display result
	echo $total;

3. Avoid Redundant Comments

   Prinsip *Avoid Redundant Comments* merupakan salah satu aspek penting dalam praktik penulisan komentar yang baik dalam kode. Prinsip ini mendorong pengembang untuk menghindari penulisan komentar yang redundan atau berulang – ulang. Komentar yang redundan hanya mengulang informasi yang sudah jelas terlihat dari kode itu sendiri, dan tidak memberikan nilai tambah atau pemahaman yang lebih baik. Berikut merupakan contoh *good code* dan *bad code* pada prinsip *avoid redundant comments*

Tabel 2.38 Contoh Good Code Avoid Redundant Comments

Good Code	$total = 0;
	
	foreach ($items as $item) {
	  $price = $item['price'];
	  $total += $price;
	}

Tabel 2.39 Contoh Bad Code Avoid Redundant Comments

Bad Code	// Initialize the total to zero
	$total = 0;
	
	foreach ($items as $item) {
	  // Get the item price
	  $price = $item['price'];
	
	  // Add the item price to the total
	  $total += $price;
	}

4. Avoid Misleading Comments

   Prinsip ini mengacu pada pentingnya menghindari penulisan komentar yang dapat menyesatkan atau memberikan informasi yang salah kepada pembaca kode. Komentar yang menyesatkan dapat menyebabkan kebingungan dan mengakibatkan kesalahan dalam pemahaman atau implementasi kode. Berikut merupakan contoh kode pada prinsip *avoid misleading comments*.

Tabel 2.40 Contoh Good Code Avoid Misleading Comments

Good Code	$sum = 0;
	$count = 0;
	
	foreach ($numbers as $number) {
	  $sum += $number;
	  $count++;
	}
	
	$average = $sum / $count;

Tabel 2.41 Contoh Bad Code Avoid Misleading Comments

Bad Code	// Calculate the average of the numbers
	$sum = 0;
	$count = 0;
	
	foreach ($numbers as $number) {
	  $sum += $number;
	  $count++;
	}
	
	$average = $sum / $count; // calculate average

5. Avoid Commented-Out Code

   Prinsip ini mengatur untuk menghindari mengomentari kode yang sebenarnya tidak lagi digunakan atau relevan dalam program. Kode yang dikomentari biasanya tidak memberikan manfaat kepada pengembang dan dapat mengaburkan pemahaman terhadap kode yang aktif. Berikut contoh kode yang dikomentari.

Tabel 2.42 Contoh Bad Code Avoid Commented-out Code

Bad Code	$subtotal = 100;
	// $discount = 20; // Commented out discount
	$total = $subtotal; // Calculate total without discount
	// $total -= $discount; // Subtract discount

2.6.4 *Clean code* Formatting

Clean code formatting adalah konsep dalam menulis kode program yang mengacu pada cara mengatur kode dengan benar sehingga mudah dibaca, dimengerti, dan dipelihara oleh tim pengembang yang berbeda. Beberapa prinsip formatting yang digunakan dalam *clean code* antara lain konsistensi, penggunaan spasi, indentasi, dan pemformatan yang sesuai dengan konvensi bahasa pemrograman yang digunakan.

1. Vertical Openness Between Concepts

   Prinsip ini menjelaskan tentang mengatur jarak vertikal yang baik antara konsep-konsep yang berbeda dalam kode, seperti antara fungsi, metode, dan blok logika, agar kode lebih mudah dibaca dan dipahami.

   Prinsip ini juga mengacu pada penggunaan spasi kosong atau baris kosong untuk memisahkan konsep-konsep yang berbeda dalam kode. Hal ini membantu memisahkan secara visual setiap bagian kode sehingga pengembang dapat dengan mudah mengidentifikasi dan memahami struktur dan alur logika dari kode tersebut.

Tabel 2.43 Contoh Good Code Vertical Openness Between Concepts

Good Code	class UserManager {
	
	public function createUser($username, $password) {
	$user = new User($username);
	$user->setPassword($password);
	
	return $user;
	}

Good Code	public function updateUser($user, $newData) {
	// Logic to update user data
	
	return $user;
	}
	}

Tabel 2.44 Contoh Bad Code Vertical Openness Between Concepts9

Bad Code	class UserManager {
	public function createUser($username, $password) {
	$user = new User($username);
	$user->setPassword($password);
	// Some additional logic for user creation
	return $user;
	}
	public function updateUser($user, $newData) {
	// Logic to update user data
	return $user;
	}
	// ... other methods ...
	}

2. Vertical Density

Vertical density menjelaskan untuk menjaga jarak kerapatan antar bagian kode yang berkaitan. Terlalu banyak baris kosong antara bagian-bagian kode yang saling terkait dapat mengakibatkan kode terlihat terlalu terpisah dan menyebabkan pengembang kesulitan dalam mengikuti alur logika. Meskipun pemisahan visual yang baik antara konsep-konsep penting untuk keterbacaan, akan tetapi terlalu banyak baris kosong dapat mengurangi kepadatan vertikal kode.

Tabel 2.45 Contoh Good Code Vertical Density

Good Code	class Calculator {
	  public function add($a, $b) {
	  return $a + $b;
	  }
	
	  public function subtract($a, $b) {
	  return $a - $b;
	  }
	
	  public function multiply($a, $b) {
	  return $a * $b;
	  }
	
	  public function divide($a, $b) {
	  if ($b != 0) {
	  return $a / $b;
	  } else {
	  throw new Exception("Division by zero.");
	  }
	  }
	}

Tabel 2.46 Contoh Bad Code Vertical Density

Bad Code	class Calculator {
	  public function add($a, $b) {
	  return $a + $b;
	  }

```
    public function subtract($a, $b) {
        return $a - $b;
    }




    public function multiply($a, $b) {
        return $a * $b;
    }
    }
```

3. Vertical Distance

Vertical distance menjelaskan untuk menghindari penggunaan baris kode yang terlalu banyak dalam satu blok atau metode. Prinsip ini mendorong untuk membatasi ukuran blok-blok kode atau metode agar tidak terlalu panjang vertikalnya.

Tabel 2.47 Contoh Good Code Vertical Distance

Good Code

```
class OrderManager {
    public function processOrder($order) {
        $this->applyDiscounts($order);
        $this->calculateTotal($order);
        $this->placeOrder($order);
    }
    private function applyDiscounts($order) {
        // Applying applicable discounts
    }
    private function calculateTotal($order) {
        // Calculating the total cost
    }
}
```

Tabel 2.48 Contoh Bad Code Vertical Distance

Bad Code	class OrderManager {
	public function processOrder($order) {
	// Validating the order details
	// Applying applicable discounts
	// Calculating the total cost
	// Placing the order and notifying the customer
	}
	}

4. Horizontal Openness and Density

Prinsip ini mengatur pada penggunaan spasi dan tata letak yang baik di dalam baris kode secara horizontal. Prinsip ini menekankan pentingnya penggunaan spasi untuk memisahkan elemen-elemen seperti operator, variabel, dan simbol lainnya, sehingga kode menjadi lebih mudah dibaca.

Tabel 2.49 Contoh Good dan Bad Code Horizontal Openness

Good Code	$totalPrice = $subtotal + ($subtotal * $taxRate);
Bad Code	$totalPrice=$subtotal+($subtotal*$taxRate);

Kemudian untuk *horizontal density* berkaitan dengan pembatasan panjang baris kode secara horizontal. Baris kode yang terlalu panjang akan mengakibatkan kode sulit dipahami.

Tabel 2.50 Contoh Good dan Bad Code Horizontal Density

Good Code	$result = someFunction($param1, $param2, $param3);
Bad Code	$result = someFunction($param1, $param2, $param3, $param4, $param5, $param6, $param7, $param8);

5. Horizontal Alignment and Indentation

Prinsip ini mengatur agar kode memiliki indentasi yang konsisten dan menyusun blok – blok kode agar terlihat perbedaanya.

Tabel 2.51 Contoh Good dan Bad Code Horizontal Alignment

Good Code	if ($condition) {
	  if ($nestedCondition) {
	  }
	}

Bad Code	if ($condition) {
	// some code here
	if ($nestedCondition) {
	}
	}

2.6.5 Clean Error Handling

Clean Error Handling adalah sebuah konsep dalam *clean code* yang berfokus pada cara menangani kesalahan (error) pada kode agar lebih mudah dipahami dan diatur. Error handling yang baik dapat membantu dalam pemecahan masalah dan memperbaiki kesalahan dengan lebih efisien. Konsep clean error handling, seperti yang dijelaskan oleh Robert Martin dalam bukunya "*Clean code*", menekankan pentingnya menangani kesalahan dengan cara yang efektif dan menginformasikan pesan kesalahan dengan jelas. Beberapa prinsip dalam konsep clean error handling di antaranya adalah:

1. Jangan mengabaikan kesalahan: Ketika sebuah kesalahan terjadi, jangan mengabaikannya atau hanya mencatatnya dalam log. Kesalahan harus ditangani dengan baik dan memberikan informasi yang jelas tentang kesalahan tersebut.
2. Gunakan exception handling: Gunakan exception handling untuk menangani kesalahan. Dalam PHP, exception handling dapat diimplementasikan menggunakan try-catch statement. Ketika terjadi kesalahan, exception akan dilempar (thrown) dan kemudian ditangkap (caught) menggunakan try-catch statement.
3. Berikan pesan kesalahan yang jelas: Pesan kesalahan harus memberikan informasi yang jelas tentang kesalahan yang terjadi. Pesan kesalahan

sebaiknya mencakup jenis kesalahan, lokasi kesalahan, dan informasi yang berguna untuk memperbaiki kesalahan tersebut.

Berikut merupakan contoh *good code* dan *bad code* pada prinsip *clean error handling*.

Tabel 2.52 Contoh Good Code Clean Error Handling

Good Code	function divideNumbers($dividend, $divisor) {
	  if ($divisor == 0) {
	  throw new Exception("Divisor cannot be zero.");
	  }
	  return $dividend / $divisor;
	}
	try {
	  $result = divideNumbers(10, 0);
	  echo "Result: " . $result;
	} catch (Exception $e) {
	  echo "Error: " . $e->getMessage();
	}

Tabel 2.53 Contoh Bad Code Clean Error Handling

Bad Code	function divideNumbers($dividend, $divisor) {
	  if ($divisor == 0) {
	  return "Divisor cannot be zero.";
	  }
	  return $dividend / $divisor;
	}
	$result = divideNumbers(10, 0);
	  if (is_string($result)) {
	  echo "Error: " . $result;
	  } else {
	  echo "Result: " . $result;

Perbedaan antara *good code* dan *bad code* tersebut adalah pada *good code* menggunakan *exception handling* yang jelas dan konsisten untuk

menangani kesahalan, sementara *bad code* hanya mengembalikan pesan kesalahan sebagai *string* dan menggunakan pengecekan tipe secara manual untuk menentukan apakah terjadi *error*. *Good code* memberikan pesan *error* yang informatif dan memisahkan logika penanganan *error* dari logika bisnis utama, sehingga memudahkan pemahaman dan *debugging* kode.

2.6.6 Clean Object and Data Structures

Pada prinsip Object and Data Structures, terdapat dua konsep yang berbeda. Objek adalah konsep dalam pemrograman berorientasi objek yang terdiri dari atribut (data) dan perilaku (fungsi). Objek digunakan untuk mewakili entitas nyata atau abstrak pada program. Sementara itu, struktur data adalah cara penyimpanan, pengorganisasi, dan pengaksesan data. Berikut adalah beberapa contoh praktik *clean code* yang sesuai dengan aturan ini.

1. Data Abstraction

   Abstraksi data (*data abstraction*) dalam *Clean Object and Data Structures* merujuk pada konsep menyembunyikan detail implementasi dan memisahkan antara inti atau esensi data dengan cara data diakses dan dimanipulasi.

   Dalam konteks ini, abstraksi data berfokus pada penggunaan antarmuka yang abstrak atau kontrak untuk berinteraksi dengan data, daripada mengungkapkan rincian implementasi data secara langsung. Dengan menggunakan abstraksi data, pengguna hanya perlu mengetahui dan menggunakan antarmuka yang didefinisikan, tanpa perlu tahu bagaimana data diimplementasikan secara spesifik. Berikut contoh penggunaan *data abstraction.*

Tabel 2.54 Contoh Good Code Data Abstraction

Good Code	class Customer {
	private $name;
	
	public function __construct($name) {
	$this->name = $name;
	}

```
        public function getName() {
            return $this->name;
        }
    }
    class CustomerService {
        public function createCustomer(Customer $customer) {
            // Logika untuk membuat pelanggan baru.
        }
}
```

Tabel 2.55 Contoh Bad Code Data Abstraction

Bad Code	class CustomerService {
	public function createCustomer($name) {
	// Logika untuk membuat pelanggan baru
	}
	}

Pada *good code* terdapat pemisahan antara kelas customer sebagai data abstraksi dan kelas CustomerService sebagai objek abstraksi. Kelas customer hanya memiliki data dan metode-metode akses untuk data tersebut. Ini menjaga integritas data dan mencegah akses langsung ke data dari luar kelas. CustomerService hanya berinteraksi dengan objek Customer tanpa harus tahu tentang detail dari objek customer.

Sedangkan pada *bad code* fungsi createCustomer() dan menerima parameter sebagai gantinya objek yang sesuai. Hal ini membuat kelas CustomerService bergantung pada struktur data yang spesifik (urutan parameter) dan tidak memisahkan data dari logika bisnis. Kode tersebut lebih sulit untuk dikelola dan rentan terhadap kesalahan, karena tidak ada cara yang jelas untuk memastikan bahwa parameter yang diteruskan sesuai dengan data pelanggan dengan benar.

2. The Law of Demeter

The law of demeter adalah sebuah prinsip dalam pemrograman berorientasi objek yang menyatakan bahwa sebuah objek seharusnya berinteraksi hanya dengan objek-objek yang berada dalam jarak dekat dengannya. Dalam kata lain, objek tersebut tidak boleh memiliki pengetahuan yang terlalu dalam tentang struktur internal dari objek-objek lain.

Prinsip ini bertujuan untuk mengurangi ketergantungan yang erat antara objek-objek dan mencegah terjadinya ketergantungan berantai yang kompleks. Berikut adalah contoh penerapan *the law of demeter*.

Tabel 2.56 Contoh Good Code The Law of Demeter

Good Code	class Wallet {
	  private $balance;
	
	  public function __construct($balance) {
	  $this->balance = $balance;
	  }
	  public function getBalance() {
	  return $this->balance;
	  }
	  public function deductBalance($amount) {
	  if ($this->balance >= $amount) {
	  $this->balance -= $amount;
	  return true;
	  }
	  return false;
	  }
	}

Good Code	class Person {
	  private $wallet;
	
	  public function __construct($wallet) {
	  $this->wallet = $wallet;
	  }
	  public function spendMoney($amount) {
	  if ($this->wallet->deductBalance($amount)) {
	  echo "Money spent: $amount\n";
	  } else {
	  echo "Insufficient balance.\n";
	  }
	  }
	}

Tabel 2.57 Contoh Bad Code The Law of Demeter

Bad Code	class Wallet {
	  private $balance;
	
	  public function __construct($balance) {
	  $this->balance = $balance;
	  }
	  public function getBalance() {
	  return $this->balance;
	  }
	}
	class Person {
	  private $wallet;

Bad Code	public function __construct($wallet) {
	$this->wallet = $wallet;
	}
	public function spendMoney($amount) {
	// Melanggar Hukum Demeter: Person seharusnya tidak perlu mengetahui tentang Wallet.
	$currentBalance = $this->wallet->getBalance();
	if ($currentBalance >= $amount) {
	$this->wallet->deductBalance($amount);
	echo "Money spent: $amount\n";
	} else {
	echo "Insufficient balance.\n";
	}
	}
	}

Pada contoh *good code*, kelas Person tidak mengakses metode getBalance() secara langsung pada objek Wallet, melainkan hanya berinteraksi dengan metode deductBalance() yang sesuai. Ini menjaga pemisahan antarobjek dengan lebih baik.

Pada contoh *bad code*, kelas Person seharusnya tidak perlu mengetahui tentang struktur objek Wallet. Namun, kelas Person secara langsung mengakses metode getBalance() pada objek Wallet.

3. Data Transfer Object

Data Transfer Object (DTO) adalah sebuah pola desain (design pattern) yang digunakan dalam pemrograman untuk mengirim data antara lapisan aplikasi. DTO bertindak sebagai wadah sederhana yang membawa data dari satu lapisan ke lapisan lainnya tanpa memiliki logika bisnis yang kompleks.

DTO dirancang untuk mengoptimalkan komunikasi dan transfer data antara objek dan komponen dalam aplikasi. Biasanya, DTO

digunakan ketika data perlu dipindahkan dari objek domain ke lapisan presentasi atau ketika data perlu dikirim melalui jaringan atau antarmuka layanan.

2.6.7 Clean Classes

1. Single Responsibility Principle

Single Responsibility Principle (SRP) adalah sebuah prinsip dalam pemrograman yang menyatakan bahwa setiap class atau modul harus hanya memiliki satu tanggung jawab atau fungsi tertentu[15]. Dengan kata lain, sebuah class atau modul sebaiknya hanya bertanggung jawab dalam melakukan satu hal saja dan tidak lebih dari itu. Hal ini membantu dalam memperjelas fungsionalitas dari setiap class atau modul sehingga lebih mudah dipahami, diuji, dan dikembangkan. Dengan menerapkan SRP, kode akan menjadi lebih bersih, mudah dipelihara, dan dapat dikembangkan secara terpisah tanpa memengaruhi kode lain.

Tabel 2.58 Contoh Good Code Single Responsibility Principle

Good Code	class EmailSender {
	  public function sendEmail($to, $subject, $message) {
	  // Logika untuk mengirim email
	  }
	}
	
	class UserManager {
	  public function registerUser($username, $password) {
	  // Logika untuk mendaftarkan pengguna baru
	  }
	}

Tabel 2.59 Contoh Bad Code Single Responsibility Principle

Bad Code	class User {
	public function registerUser($username, $password) {
	// Logika untuk mendaftarkan pengguna baru
	}
	public function sendEmail($to, $subject, $message) {
	// Logika untuk mengirim email
	}
	}

Pada *good code*, kelas EmailSender bertanggung jawab hanya untuk mengirim email, sedangkan kelas UserManager bertanggung jawab hanya untuk mendaftarkan pengguna baru. Setiap kelas memiliki tanggung jawab tunggal sesuai dengan SRP.

Pada *bad code*, kelas User memiliki tanggung jawab ganda, yaitu mendaftarkan pengguna baru dan mengirim email. Hal ini melanggar SRP karena satu kelas seharusnya hanya memiliki satu tanggung jawab tunggal.

2. Open-closed Principle (OCP)

Prinsip ini menyatakan bahwa entitas perangkat lunak (kelas, modul, atau fungsi) seharusnya terbuka untuk perluasan tetapi tertutup untuk modifikasi[15]. Ini berarti kita harus dapat menambahkan fitur baru ke sistem tanpa mengubah kode yang sudah ada. Dengan menerapkan OCP, kita mendorong penggunaan polimorfisme, abstraksi, dan pewarisan untuk mencapai fleksibilitas dan keberlanjutan kode.

Tabel 2.60 Contoh Good Code Open-closed Principle

Good Code	interface Shape {
	public function calculateArea();
	}
	class Circle implements Shape {
	public function calculateArea() {
	// Logika untuk menghitung luas lingkaran

Good Code	}
	}
	class Rectangle implements Shape {
	public function calculateArea() {
	// Logika untuk menghitung luas persegi panjang
	}
	}

Tabel 2.61 Contoh Bad Code Open-closed Principle

Bad Code	class Circle {
	public function calculateArea() {
	// Logika untuk menghitung luas lingkaran
	}
	}
	class Rectangle {
	public function calculateArea() {
	// Logika untuk menghitung luas persegi panjang
	}
	}

Pada *good code* menggunakan konsep interface dan implementasi dari interface Shape. Sehingga dapat menambahkan bentuk baru dengan mengimplementasikan interface Shape tanpa mengubah kode yang sudah ada.

Pada *bad code* tidak menggunakan konsep interface, sehingga saat menambahkan bentuk baru seperti segitiga, harus mengubah kode di kedua kelas Circle dan Rectangle. Hal ini melanggar OCP karena kelas-kelas tersebut tidak tertutup untuk modifikasi.

3. Liskov Substitution Principle (LSP)

Prinsip ini menyatakan bahwa objek dari kelas turunan harus dapat digunakan sebagai pengganti objek kelas dasarnya tanpa mengganggu kebenaran dan kegunaan program[15]. Dalam hal ini, hubungan antar kelas turunan harus mematuhi kontrak yang ada pada kelas dasar. Dengan menerapkan LSP, kita dapat menghindari masalah yang muncul saat menggunakan pewarisan yang salah atau melanggar kontrak.

Tabel 2.62 Contoh Good Code Liskov Substitution Principle

Good Code	class Animal {
	  public function makeSound() {
	  // Logika untuk menghasilkan suara hewan
	
	  }
	}
	
	class Dog extends Animal {
	  public function makeSound() {
	  // Logika untuk menghasilkan suara anjing
	
	  }
	}
	
	class Cat extends Animal {
	  public function makeSound() {
	  // Logika untuk menghasilkan suara kucing
	
	  }
	}

Tabel 2.63 Contoh Good Code Liskov Substitution Principle

Bad Code	class Animal {
	  public function makeSound() {
	  // Logika untuk menghasilkan suara hewan
	  }
	}
	
	class Dog extends Animal {
	  public function makeSound() {
	  throw new Exception("Dogs cannot make sound.");
	  }
	}
	
	class Cat extends Animal {
	  public function makeSound() {
	  // Logika untuk menghasilkan suara kucing
	  }
	}

Pada *good code*, kelas Dog dan Cat adalah turunan dari kelas Animal, dan setiap kelas memiliki implementasi yang sesuai dengan fungsinya. Semua kelas turunan dapat digunakan sebagai pengganti kelas Animal tanpa mempengaruhi kebenaran program.

Pada *bad code*, kelas Dog melanggar kontrak LSP karena menghasilkan exception saat memanggil metode makeSound(). Hal ini membuat kelas Dog tidak dapat digunakan sebagai pengganti kelas Animal seperti yang diharapkan.

4. Interface Segregation Principle (ISP)

Prinsip ini menyatakan bahwa klien tidak boleh dipaksa untuk mengimplementasikan metode yang tidak mereka butuhkan[15]. Interface atau kontrak antara kelas-kelas harus cukup spesifik dan sesuai dengan kebutuhan setiap klien. Dengan menerapkan ISP, kita dapat menghindari dependensi yang tidak perlu dan membuat kode lebih fleksibel.

Tabel 2.64 Contoh Good Code Interface Segregation Principle

Good Code	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) {
	// Logika untuk mengirim email
	}
	
	public function sendSms($to, $message) {
	// Logika untuk mengirim SMS
	}
	}

Tabel 2.65 Contoh Bad Code Interface Segregation Principle

Bad Code	interface NotificationSender {
	public function sendEmail($to, $subject, $message);
	public function sendSms($to, $message);
	}

Bad Code	interface NotificationSender {
	  public function sendEmail($to, $subject, $message);
	  public function sendSms($to, $message);
	}
	
	class NotificationManager implements NotificationSender {
	  public function sendEmail($to, $subject, $message) {
	  // Logika untuk mengirim email
	  }
	
	  public function sendSms($to, $message) {
	  // Logika untuk mengirim SMS
	  }
	}

Pada *good code* menggunakan dua kelas interface terpisah, yaitu EmailSender dan SmsSender, yang mendefinisikan metode yang sesuai dengan fungsinya. Kelas NotificationManager mengimplementasikan kedua interface ini sesuai dengan kebutuhan.

Pada *bad code*, hanya menggunakan satu interface yang mencakup semua metode, NotificationSender, yang mengharuskan implementasi semua metode tersebut oleh kelas NotificationManager. Hal ini melanggar ISP karena kelas NotificationManager harus mengimplementasikan metode yang tidak diperlukan atau relevan.

5. Dependency Inversion Principle (DIP)

Prinsip ini menyatakan bahwa modul yang tingkatannya lebih tinggi tidak boleh bergantung pada modul yang tingkatannya lebih rendah[15]. Keduanya seharusnya bergantung pada abstraksi. Prinsip ini juga mengusulkan penggunaan injeksi ketergantungan untuk membalikkan aliran dependensi. Dengan menerapkan DIP, kita dapat mencapai

ketergantungan yang lemah, pengujian yang mudah, dan komponen yang dapat digunakan ulang.

Tabel 2.66 Contoh Good Code Dependency Inversion Principle

Good Code	interface DatabaseConnection {
	public function connect();
	}
	class MySqlConnection implements DatabaseConnection {
	public function connect() {
	// Logika untuk menghubungkan ke database MySQL
	}
	}
	
	class PostgreSqlConnection implements DatabaseConnection {
	public function connect() {
	// Logika untuk menghubungkan ke database PostgreSQL
	}
	}
	
	class DatabaseManager {
	private $database;
	
	public function __construct(DatabaseConnection $database) {
	$this->database = $database;
	}
	public function performQuery($query) {
	$this->database->connect();
	// Logika untuk menjalankan query
	}
	}

Bad Code	class MySqlConnection {
	  public function connect() {
	  // Logika untuk menghubungkan ke database MySQL
	  }
	}
	
	class PostgreSqlConnection {
	  public function connect() {
	  // Logika untuk menghubungkan ke database PostgreSQL
	  }
	}
	
	class DatabaseManager {
	  private $database;
	
	  public function __construct($databaseType) {
	  if ($databaseType === 'mysql') {
	  $this->database = new MySqlConnection();
	  } elseif ($databaseType === 'postgresql') {
	  $this->database = new PostgreSqlConnection();
	  }
	  }
	
	  public function performQuery($query) {
	  $this->database->connect();
	  // Logika untuk menjalankan query
	  }
	}

Good code menggunakan konsep injeksi ketergantungan melalui konstruktor pada kelas DatabaseManager. Ini memungkinkan penggunaan berbagai implementasi DatabaseConnection, seperti MySqlConnection dan PostgreSqlConnection, tanpa mengunci kelas DatabaseManager pada implementasi tertentu.

Pada *bad code*, kelas DatabaseManager bergantung langsung pada kelas konkret MySqlConnection dan PostgreSqlConnection, hal ini melanggar prinsip *dependency inversion principle* karena modul tingkat tinggi (DatabaseManager) bergantung pada modul tingkat rendah (implementasi database spesifik) daripada pada abstraksi.

2.7 Bahasa Pemrograman PHP

PHP adalah bahasa pemrograman server-side scripting yang digunakan untuk membangun aplikasi web dan dapat terintegrasi dengan HTML. PHP saat ini adalah salah satu bahasa pemrograman paling populer yang digunakan di komunitas *open-source* dan industri untuk membangun aplikasi berfokus web dan kerangka kerja aplikasi yang besar[16]. Hal ini menunjukkan bahwa PHP memiliki daya tarik yang kuat sebagai bahasa pemrograman yang fleksibel dan dapat digunakan dalam berbagai macam aplikasi web. PHP biasanya digunakan untuk memproses data formulir, mengakses basis data, dan menghasilkan konten dinamis pada halaman web. PHP dikembangkan pada tahun 1994 oleh Rasmus Lerdorf, dan saat ini dikelola oleh The PHP Group.

PHP adalah sebuah bahasa pemrograman open-source yang umumnya digunakan untuk pengembangan website. Bahasa pemrograman ini dapat disisipkan ke dalam kode HTML dan merupakan salah satu pilihan terbaik untuk pengembangan aplikasi web. PHP mampu mengakses database, memproses data, dan menghasilkan output yang dinamis pada halaman web. Keuntungan menggunakan PHP adalah karena bahasa pemrograman ini tersedia secara gratis, mudah dimodifikasi sesuai dengan kebutuhan pengguna, dan memiliki berbagai macam library tambahan *open-source* yang memudahkan pengembangan aplikasi web. Dengan biaya rendah dan

banyaknya sumber daya yang tersedia, PHP menjadi salah satu pilihan yang populer bagi pengembang aplikasi web.

2.8 Laravel

Laravel adalah suatu *framework* atau kerangka kerja yang bersifat open-source yang ditulis dalam bahasa pemrograman PHP. Framework ini menyediakan arsitektur modular yang rapi untuk membangun aplikasi web yang menggunakan pola desain Model-View-Controller (MVC). Terdapat banyak fitur yang disediakan oleh Laravel seperti routing, middleware, Object-Relational Mapping (ORM), dan templating yang memudahkan pengembangan aplikasi web. Selain itu, dokumentasi yang lengkap dan didukung oleh komunitas yang aktif, membantu para pengembang memahami dan menggunakan framework ini dengan lebih mudah. Dengan menggunakan Laravel, pengembang dapat membuat aplikasi web dengan kode yang mudah dipelihara, efisien, dan lebih mudah untuk dikembangkan.

2.9 PHPMetrics

PhpMetrics adalah sebuah alat analisis kode statis (static code analysis) untuk bahasa pemrograman PHP dan dibuat oleh Jean-François Lépine[17]. Alat ini dapat membantu dalam mengevaluasi kualitas kode, mengidentifikasi masalah kinerja dan kompleksitas, serta memberikan metrik yang dapat membantu pengembang dalam mengambil keputusan yang lebih baik dalam pengembangan perangkat lunak.

PhpMetrics menyediakan berbagai jenis metrik, termasuk kompleksitas kode, jumlah baris kode, jumlah kode yang diulang, pengukuran ketergantungan, dan banyak lagi. Metrik-metrik ini dapat membantu pengembang dalam memahami kualitas dan kompleksitas kode pada proyek PHP, dan memberikan pandangan yang lebih baik tentang bagaimana mengoptimalkan kode untuk kinerja yang lebih baik.

PhpMetrics juga menawarkan visualisasi data dalam bentuk grafik dan diagram, yang memudahkan dalam memahami dan menganalisis hasil analisis kode. Dengan begitu, pengembang dapat melihat metrik-metrik ini dalam

bentuk yang lebih intuitif, dan memudahkan dalam mengambil keputusan yang tepat dalam perbaikan kode.

2.10 Design Pattern

Design pattern adalah sebuah solusi dalam bentuk desain yang digunakan untuk mengatasi masalah umum yang sering terjadi dalam pengembangan perangkat lunak berorientasi objek. Solusi ini dirancang untuk dapat digunakan kembali untuk memecahkan masalah yang serupa di masa depan[18]. Design pattern bukan hanya sekadar mengubah struktur data menjadi modul (class) yang dapat digunakan kembali, juga bukan tentang desain khusus untuk suatu domain atau aplikasi secara keseluruhan atau sub sistem saja. Design pattern menjelaskan bagaimana objek dan modul (class) dapat berkomunikasi dalam suatu konteks tertentu untuk memecahkan masalah desain yang umum[18].

Ada tiga kategori dari 23 design patterns yang dijelaskan dalam buku "Design Patterns: Elements of Reusable Object-Oriented Software". Ketiga kategori tersebut adalah:

1. Creational Patterns: Kategori ini fokus pada cara pembuatan objek. Creational Patterns memberikan cara untuk membuat objek yang fleksibel, tidak tergantung pada kelas konkret, dan menyederhanakan proses pembuatan objek. Contoh dari Creational Patterns adalah *Factory Method, Abstract Factory, Singleton,* dan *Builder*.
2. Structural Patterns: Kategori ini fokus pada cara menyusun objek-objek yang berbeda menjadi struktur yang lebih besar dan lebih kompleks. Structural Patterns menyediakan cara untuk menyusun objek-objek tersebut agar dapat bekerja bersama-sama dengan cara yang optimal dan memastikan fleksibilitas dan ekstensibilitas dalam kode. Contoh dari Structural Patterns adalah *Adapter, Bridge, Composite, Decorator, Facade* dan *Proxy*.
3. Behavioral Patterns: Kategori ini fokus pada cara objek-objek berinteraksi dan membagikan tanggung jawab. Behavioral Patterns membantu dalam memecahkan masalah desain pada proses interaksi antara objek dan

bagaimana objek memenuhi tanggung jawabnya dengan baik. Contoh dari Behavioral Patterns adalah *Template Method*, *Observer*, *Strategy*, dan *Chain of Responsibility*.

2.11 Analisis dan Desain Berorientasi Objek

Metode Analisis dan Desain Berorientasi Objek (Object Oriented Analysis and Design) merupakan pendekatan baru dalam memecahkan masalah dengan membangun model sesuai dengan konsep objek[19]. Model ini didasarkan pada objek yang merupakan kombinasi dari struktur data dan perilaku dalam satu entitas. Alasan penggunaan metode ini adalah karena sifat perangkat lunak yang dinamis, di mana kebutuhan pengguna dapat berubah dengan cepat. Dengan menggunakan pendekatan berorientasi objek, perangkat lunak dapat dirancang agar lebih mudah dipelihara dan dimodifikasi untuk memenuhi kebutuhan yang berubah-ubah.

Pendekatan berorientasi objek dalam pengembangan perangkat lunak bertujuan untuk mengurangi kompleksitas transisi antar tahap. Hal ini disebabkan karena notasi yang digunakan pada tahap analisis, perancangan, dan implementasi relatif sama, berbeda dengan pendekatan konvensional yang menggunakan notasi yang berbeda pada setiap tahap, sehingga membuat proses transisi menjadi lebih kompleks[20].

Dalam pendekatan berorientasi objek, pengguna diarahkan pada pemahaman konsep yang lebih dekat dengan dunia nyata melalui abstraksi atau istilah yang digunakan. Dalam dunia nyata, objek adalah yang sering dilihat oleh pengguna, bukan fungsinya. Hal ini berbeda dengan pendekatan terstruktur yang hanya memungkinkan abstraksi pada level fungsional. Dalam pemrograman berorientasi objek, penekanan diberikan pada beberapa konsep seperti Class, Object, Abstract, Encapsulation, Polymorphism, Inheritance, dan tentunya UML (Unified Modeling Language)[20], [21].

UML (Unified Modeling Language) merupakan sebuah alat bantu dan standar dalam pemrograman berorientasi objek yang terdiri dari kumpulan diagram. UML digunakan untuk membantu para pengembang sistem dan perangkat lunak dalam melakukan tugas seperti spesifikasi, visualisasi, desain

arsitektur, konstruksi, simulasi, dan testing. UML menggunakan grafik atau gambar sebagai basisnya untuk memvisualisasikan, melakukan spesifikasi, membangun, dan mendokumentasikan sistem pengembangan perangkat lunak berbasis objek[21].

Berikut beberapa macam diagram untuk membuat model berorientasi objek antara lain:

1. *Use Case Diagram*

   Use case diagram adalah pemodelan yang berkaitan dengan *behaviour* atau kelakukan dari sistem. Diagram ini menggambarkan fungsi-fungsi yang diharapkan dari sebuah sistem, dengan fokus pada aktivitas sistem dan cara kerjanya. Diagram ini merepresentasikan interaksi antara actor dan sistem, dimana setiap use case mewakili tugas tertentu, sementara actor dapat berupa entitas manusia atau mesin yang berinteraksi dengan sistem untuk menyelesaikan tugas-tugas tertentu[21].

2. *Use Case Scenario*

   Setiap *use case* dilengkapi oleh skenario. *Use Case Scenario* adalah alur jalannya proses use case dari actor dan sistem. *Use Case Scenario* dibuat per *use case* terkecil[21].

3. *Activity Diagram*

   Activity Diagram adalah suatu tahapan yang fokus pada menggambarkan proses bisnis dan urutan aktivitas atau *workflow* dalam proses tersebut[21]. Diagram ini sering digunakan dalam pemodelan bisnis untuk menunjukkan urutan aktivitas dalam suatu proses bisnis. Activity Diagram memiliki struktur yang mirip dengan flowchart atau data flow diagram dalam perancangan terstruktur. Activity Diagram dibuat berdasarkan satu atau beberapa use case pada diagram use case.

4. *Sequence Diagram*

   Sequence diagram berfungsi untuk menggambarkan interaksi antara objek-objek dalam sebuah scenario, dengan menyajikan urutan pesan-pesan yang dikirimkan di antara objek-

objek tersebut. Komponen utama dari diagram ini adalah objek yang direpresentasikan dengan kotak segi empat atau bulat, pesan-pesan yang direpresentasikan dengan garis putus-putus, dan waktu yang ditunjukkan dengan urutan vertikal. Sequence diagram memberikan kejelasan mengenai setiap interaksi yang terjadi pada use case diagram yang telah dibuat sebelumnya, dan berguna untuk memberikan gambaran detail tentang perilaku sistem pada sebuah scenario[21].

5. *Class Diagram*

Class diagram adalah sebuah diagram yang memperlihatkan struktur dan penjelasan tentang class, objek, dan paket, serta hubungan antara mereka dalam sistem. Diagram ini memberikan gambaran keseluruhan tentang hubungan antar class dalam sebuah sistem dan cara mereka berinteraksi untuk mencapai tujuan[21].