SOLID Principles & Clean Code

Writing PHP that humans can actually read

José María Vázquez - August, 2026

syde

Agenda

01

Why does code quality matter?

02

SOLID – One principle at a time

03

Clean Code fundamentals

04

Putting it all together

Why Does Code Quality Matter?

70%

of dev time is spent reading, not writing code

80%

of bugs are found in the 20% of most complex modules

10x

more expensive to fix a bug in production than in review

01

The SOLID Principles

SOLID at a Glance

S — **Single Responsibility**
One class, one reason to change

O — **Open / Closed**
Open for extension, closed for modification

L — **Liskov Substitution**
Subtypes must be substitutable for their base type

I — **Interface Segregation**
Clients shouldn't depend on methods they don't use

D — **Dependency**
Depend on abstractions, not concretions

S – Single Responsibility Principle

"A class should have only one reason to change" — Robert C. Martin

BEFORE

```php
class OrderService {

    public function save(Order $order) {
        // Writes to DB (wpdb)
        global $wpdb;
        $wpdb->insert('orders', [...]);
    }

    public function sendEmail(Order $order) {
        // Sends email via wp_mail()
        wp_mail($order->email, 'Your order', '...');
    }

    public function generateInvoice(Order $order) {
        // Generates a PDF — different library
        return new TCPDF(...);
    }

    public function validatePayment(array $data) {
        // Validates card data — yet another concern
    }

}
```

AFTER

```php
class OrderRepository {
    public function save(Order $order): void {
        global $wpdb;
        $wpdb->insert('orders', [...]);
    }
}

class OrderMailer {
    public function sendConfirmation(Order $o): void {
        wp_mail($o->email, 'Confirmed', '...');
    }
}

class InvoiceGenerator {
    public function generate(Order $o): string { ... }
}

class PaymentValidator {
    public function validate(array $data): bool { ... }
}
```

Each class has ONE job. Changing the email provider never touches the DB layer.

O – Open / Closed Principle

"Open for extension, closed for modification"

BEFORE

```
function calculate_shipping(string $method, float $weight): float {

    // Every new carrier = editing this function
    if ($method === 'standard') {
        return $weight * 2.5;
    } elseif ($method === 'overnight') {
        return $weight * 8.0;
    }
    // Adding "express"? Must touch this file ❌
    return 0;
}
```

AFTER

```
interface ShippingMethod {
    public function calculate(float $weight): float;
}

class StandardShipping implements ShippingMethod {

    public function calculate(float $weight): float {
        return $weight * 2.5;
    }
}

class OvernightShipping implements ShippingMethod {

    public function calculate(float $weight): float {
        return $weight * 8.0;
    }
}
// Adding Express = just a new class. Zero changes ✅
```

Adding a new carrier = new class only. Existing code is never touched.

L – Liskov Substitution Principle

"Subtypes must be substitutable for their base types without altering correctness"

VIOLATION

```
class Rectangle {

    public function setWidth(int $w): void { $this->w = $w; }
    public function setHeight(int $h): void {$this->h = $h; }
    public function area(): int { return $this->w * $this->h; }
}

class Square extends Rectangle {

    // Breaks Rectangle's contract!
    public function setWidth(int $w): void {
        $this->w = $this->h = $w; // ⚠ surprise side effect
    }

    public function setHeight(int $h): void {
        $this->w = $this->h = $h; // ⚠ surprise side effect
    }
}
```

SOLUTION

```
interface Shape {
    public function area(): int;
}

class Rectangle implements Shape {
   public function __construct(
      private int $width,
      private int $height
    ) {}

   public function area(): int {
      return $this->width * $this->height;
   }
}

class Square implements Shape {
    public function __construct(private int $side) {}
    public function area(): int { return $this->side ** 2; }
}
```

No surprise behaviour. Any Shape can be used wherever Shape is expected.

I – Interface Segregation Principle

"Clients should not be forced to depend on interfaces they do not use"

❌ FAT INTERFACE

```php
interface ContentPublisher {
    public function publish(): void;
    public function schedule(DateTime $date): void;
    public function exportToPdf(): string;
}

// WidgetPublisher only needs publish()
// but is forced to implement everything
class WidgetPublisher implements ContentPublisher {
    public function publish(): void { /* ok */ }

    public function schedule(DateTime $d): void {
        throw new Exception('Not supported'); // ❌
    }

    public function exportToPdf(): string {
        return ''; // ❌ meaningless
    }
}
```

✅ SEGREGATED

```php
interface Publishable {
    public function publish(): void;
}

interface Schedulable {
    public function schedule(DateTime $date): void;
}

interface PdfExportable {
    public function exportToPdf(): string;
}

// Each class implements only what it needs
class WidgetPublisher implements Publishable {
    public function publish(): void { /* ✅ */ }
}

class PagePublisher implements Publishable, Schedulable, PdfExportable {
    // implements all three ✅
}
```

Small, focused interfaces prevent empty implementations and unexpected exceptions.

D – Dependency Inversion Principle

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

❌ TIGHT COUPLING

```php
class PostService {
    private wpdb $db;

    public function __construct() {
        // Hardcoded — impossible to test or swap
        global $wpdb;
        $this->db = $wpdb; // ❌ coupled to global
    }

    public function getPublished(): array {
        return $this->db->get_results(
            "SELECT * FROM posts WHERE status='publish'"
        );
    }
}
```

✅ INJECTED

```php
interface PostRepositoryInterface {
    public function getPublished(): array;
}

class PostService {

    // Depends on the interface, not the implementation
    public function __construct(
        private PostRepositoryInterface $repository
    ) {}

    public function getPublished(): array {
        return $this->repository->getPublished();
    }
}

// In production: new PostService(new pdbPostRepository())
// In tests:      new PostService(new MockPostRepository())
```

Depend on the interface. Swap MySQL for a mock in tests — PostService never changes.

syde

02

Clean Code Fundamentals

Meaningful Names

"The name of a variable, function, or class should answer why it exists, what it does, and how it is used."

UNCLEAR

```php
<?php
$d = 30;       // What is d?
$t = time();   // What kind of time?

function get_d(): array {
   // What does this return?
   return get_option('d_data', []);
}

$lst = get_d();
if( count($lst) > 0 ){
   // Process... what exactly?
}
```

CLEAR

```php
<?php
$days_since_last_login = 30;
$current_timestamp     = time();

function get_active_subscribers(): array {
    return get_option('active_subscribers', []);
}

$active_subscribers = get_active_subscribers();
if( count($active_subscribers) > 0 ){
    notify_subscribers($active_subscribers);
}
```

✓ Pronounceable

✓ Searchable

✓ No abbreviations

✓ Self-documenting

Functions: Small & Focused

GOD FUNCTION — does 6 things

```php
<?php
function process_user_registration(array $data): int {
    // 1. Validate
    if( empty($data['email']) || empty($data['pass']) ){
        wp_die('Missing fields');
    }
    // 2. Hash
    $hash = wp_hash_password($data['pass']);
    // 3. Insert
    $user_id = wp_insert_user([
        'user_email' => $data['email'],
        'user_pass'  => $hash,
    ]);
    // 4. Email
    wp_mail($data['email'], 'Welcome!', '...');
    // 5. Meta
    update_user_meta($user_id, 'source', $data['source']);
    // 6. Log
    error_log('User registered: ' . $data['email']);
    return $user_id;
}
```

ONE LEVEL OF ABSTRACTION

```php
<?php
function register_user(array $data): int {
    $validated = validate_registration_data($data);
    $hashed    = hash_user_password($validated);
    $user_id   = save_user_to_database($hashed);
    send_welcome_email($user_id);
    log_registration($user_id);
    return $user_id;
}

// Each helper: small, named, testable
function validate_registration_data(array $d): array {
    if( empty($d['email']) ){
        throw new InvalidArgumentException('Email required');
    }
    return $d;
}

function hash_user_password(array $d): array {
    $d['pass'] = wp_hash_password($d['pass']);
    return $d;
}
```

The rules:

- Max ~20 lines
- Do ONE thing
- No hidden side effects
- Max 2–3 params

Comments: When to Use Them

"Don't comment bad code — rewrite it." — Brian W. Kernighan

NOISE — restates the code

```php
<?php
// Check if user is admin
if( current_user_can('manage_options') ){ ... }

// Loop through posts
foreach( $posts as $post ){ ... }

// Get option
$val = get_option('my_plugin_setting');

// Increment counter
$count++;
```

ADDS VALUE — explains why

```php
<?php
// Bcrypt silently truncates passwords > 72 bytes.
// We truncate here explicitly so behaviour is clear.
$password = mb_substr($password, 0, 72);

// WP_Query 'post__not_in' is slow on large tables.
// Using meta_query instead — see GitHub issue #342.
$query = new WP_Query(['meta_query' => [...]]);

// Transient TTL is intentionally short (5 min):
// data changes frequently during flash sales.
set_transient('sale_price', $price, 5 * MINUTE_IN_SECONDS);
```

Good comments explain WHY. The code itself should explain the WHAT.

Worth writing

- Why a workaround exists
- Non-obvious algorithm constraint
- Performance trade-off reason
- Link to issue / RFC

Delete these

- Restating the code in English
- Commented-out dead code
- Changelog (“// John, 2026”)
- Misleading / outdated comments

DRY & YAGNI

Don't Repeat Yourself

❌ NOISE — restates the code

```
// ❌ Tax rate duplicated 3 times
$tax_a = $price_a * 0.21;
$tax_b = $price_b * 0.21;
$tax_c = $price_c * 0.21;

// ✅ Single source of truth
const TAX_RATE = 0.21;
function apply_tax( float $price ): float {
    return $price * TAX_RATE;
}
```

One change, one place.
Never miss an update.

You Ain't Gonna Need It

✅ ADDS VALUE — explains why

```
// ❌ Speculative features
class ProductRepository {
    public function importFromCsv() {}  // no req
    public function importFromXml() {}  // no req
    public function importFromApi() {}  // no req
}

// ✅ Build what you need today
class ProductRepository {
    public function save(Product $p): void { ... }
    public function findById(int $id): Product { ... }
}
```

Build now, extend later
when the need is real.

Error Handling

SWALLOWED ERROR — caller gets null

```php
<?php
function get_post_data(int $post_id): ?array {
   try {
      global $wpdb;
      $result = $wpdb->get_row(
       $wpdb->prepare('SELECT * FROM posts WHERE ID=%d', $post_id),
       ARRAY_A
      );
      return $result;
   }catch( Exception $e ){
      error_log($e->getMessage()); // silently swallowed
      return null; // caller has no idea what went wrong
   }
}
// Caller: if( $data === null ){ ??? }
```

- Use typed exceptions
- Don’t catch what you can’t handle

TYPED EXCEPTIONS — caller knows why

```php
<?php
class PostNotFoundException extends RuntimeException {}
class DatabaseException extends RuntimeException {}

function get_post_data( int $post_id ): array {
   global $wpdb;
   $result = $wpdb->get_row(
       $wpdb->prepare('SELECT * FROM posts WHERE ID=%d', $post_id),
       ARRAY_A
   );
   if( $wpdb->last_error ){
      throw new DatabaseException($wpdb->last_error);
   }
   if( $result === null ){
      throw new PostNotFoundException("Post {$post_id} not found");
   }
   return $result;
}
// Caller handles PostNotFound and DatabaseError differently
```

- Include context in messages
- Fail fast and visible

03

Putting it ALL together

Full Refactor: Before & After

A real WordPress admin_post handler, before and after

BEFORE — one function, 6 jobs

```php
<?php
add_action('admin_post_update_post', function() {
    // 1. Nonce check
    if( !wp_verify_nonce($_POST['_wpnonce'], 'update_post') ){
        wp_die('Unauthorized');
    }
    // 2. Permission check
    if( !current_user_can('edit_posts') ){
        wp_die('No access');
    }
    // 3. Sanitize
    $title   = sanitize_text_field($_POST['title']);
    $content = wp_kses_post($_POST['content']);
    // 4. Update DB
    wp_update_post(['ID' => $_POST['id'], 'post_title' => $title]);
    // 5. Notify
    wp_mail(get_option('admin_email'), 'Post updated', $title);
    // 6. Redirect
    wp_redirect(admin_url('edit.php'));
});
```

AFTER — SRP + DI + Clean Code

```php
<?php
class PostUpdateHandler {
    public function __construct(
        private FormValidator       $validator,
        private PostUpdater         $updater,
        private NotificationService $notifier
    ) {}

    public function handle(): void {
        $data    = $this->validator->validate($_POST);
        $post    = $this->updater->update($data);
        $this->notifier->postUpdated($post);
        wp_redirect(admin_url('edit.php'));
    }
}

// Register the hook cleanly:
$handler = new PostUpdateHandler(
    new FormValidator(),
    new PostUpdater(),
    new NotificationService(new EmailNotifier())
);
add_action('admin_post_update_post', [$handler, 'handle']);
```

Testable | Extensible | Readable | Swappable notifier

Quick Reference

S	**Single Responsability**	One class, one reason to change
O	**Open / Closed**	Extend behaviour, never modify existing code
L	**Liskov Substitution**	Subtypes must honor their parent's contract
I	**Interface Segregation**	Small, focused interfaces only
D	**Dependency Inversion**	Inject abstractions, not concretions

–	**Meaningful Names**	Reveal intent - no $d, $t or $tmp
–	**Small Functions**	One level of abstraction, max ~20 lines
–	**DRY**	One place for every piece of knowledge (Don't Repeat Yourself)
–	**YAGNI**	Build only what you need right now (You Ain't Gonna Need It)
–	**Error Handling**	Typed exceptions, context, fail fast

Code is read more than it's written.

Write for the engineer who reads it at 2 AM.

→ Apply SOLID to keep your plugin classes focused and extensible

→ Name things so a stranger understands without reading the body

→ Small functions, typed exceptions, no duplication

→ Start today: pick ONE file and apply one principle

syde

Thank You!

GOLD AGENCY PARTNER WORDPRESS VIP

woo PRO Partner

BEST WORDPRESS AGENCY DESIGNRUSH.com 2019