

What are the basics of software development? The SOLID principles would seem to be key to developing good software. How does this apply to php? PHP is a quirky language- yes or no?

Thought for a couple of seconds >

Yes: **PHP is a quirky language**. It is powerful, practical, widely deployed, and much better designed today than older PHP was, but it still carries historical quirks from its long evolution as a web-scripting language.

The basics of software development are not mainly about memorizing principles. They are about building systems that are **correct, understandable, maintainable, testable, reusable, secure, and efficient enough for their purpose**.

## The basics of good software development

At the core, good software usually comes down to these things:

### 1. Correctness

The software should do what it is supposed to do.

That sounds obvious, but it is the first law. Elegant code that gives the wrong answer is bad code. A clunky function that reliably does the right thing may be better than a beautifully abstracted one that introduces subtle bugs.

### 2. Clarity

Code should be understandable.

A future version of you, or another developer, should be able to look at the code and understand:

```
What does this do?  
What data does it need?  
What does it return?
```

What can go wrong?  
Where should I change it?

Clarity often matters more than cleverness.

### 3. Separation of concerns

Different parts of the system should have different jobs.

For example, in a PHP application:

```
Controller / page script: receives the request
Service class: decides what needs to happen
Repository / client: talks to the database
Entity / DTO: carries structured data
View/template: displays output
```



When everything is mixed together, the software becomes harder to change.

Bad:

```
// HTML controller: registers a user, and renders a view about it
// PHP
```



Better:

```
$user = $userService->register($registrationData);
$template->render('success.php', ['user' => $user]);
```

### 4. Encapsulation

Objects and functions should hide unnecessary details.

Your database API is a good example of this. The caller should not need to know every step required to construct a safe SQL query. They should be able to use a clean interface that protects them from low-level mistakes.

```
$user = $userService->register($registrationData);
$template->render('success.php', ['user' => $user]);
```



The caller does not need to manually build:

```
SELECT * FROM ... WHERE ... AND ...
```

```
... SQL
```

The API handles that.

## 5. Reuse without over-abstraction

Reusable code is valuable, but abstraction can become harmful if it is premature or overly complex.

Good abstraction removes repeated pain.

Bad abstraction hides simple things behind unnecessary machinery.

The balance is:

```
Abstract repeated patterns.  
Do not abstract one-off code too early.
```

## 6. Testing and verification

Good software needs ways to verify that it works.

That can mean:

```
manual tests  
unit tests  
integration tests  
benchmark tests  
debug output  
logging  
assertions  
type checks
```

In PHP, even a simple set of repeatable test scripts can be extremely valuable.

## 7. Security

PHP is commonly used for web apps, so security is foundational.

The big basics are:

```
Use prepared statements.  
Validate input.
```

Escape output.  
Protect sessions.  
Hash passwords properly.  
Use CSRF protection.  
Avoid exposing sensitive errors.  
Keep dependencies updated.

For database work, the most important rule is: **never concatenate untrusted values into SQL**. Use bound parameters.

## 8. Performance appropriate to the use case

Performance matters, but the correct question is not “is this the fastest possible?”

The correct question is:

Is this fast enough, and is the performance cost justified by the maintainability bene 

Your database API is exactly in that tradeoff zone: it spends more PHP lines of execution to buy safety, flexibility, query reuse, nested logic, and maintainability. That is valid if the measured cost is acceptable.

---

## SOLID principles

SOLID is a set of object-oriented design principles. It is especially relevant to PHP because modern PHP supports serious object-oriented programming: classes, interfaces, traits, typed properties, union types, readonly properties, enums, namespaces, autoloading, exceptions, and dependency injection.

SOLID stands for:

S - Single Responsibility Principle  
O - Open/Closed Principle  
L - Liskov Substitution Principle  
I - Interface Segregation Principle  
D - Dependency Inversion Principle



### 1. Single Responsibility Principle

A class should have one main reason to change.

This does **not** mean a class can only have one method. It means the class should represent one coherent responsibility.

Bad PHP example:

 PHP



```
class UserManager
{
    public function validateUserData(array $data): bool {}
    public function insertUserIntoDatabase(array $data): void {}
    public function sendWelcomeEmail(array $data): void {}
    public function renderUserHtml(array $data): string {}
}
```

This class does validation, persistence, email, and rendering. It has too many reasons to change.

Better:

 PHP



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

class UserRepository
{
    public function save(User $user): void {}
}

class WelcomeEmailSender
{
    public function send(User $user): void {}
}

class UserViewRenderer
{
    public function render(User $user): string {}
}
```

Applied to your kind of PHP database API:

```
ColumnName class handles column names.
TableName class handles table names.
```



WhereClause class handles WHERE logic.  
ParameterBinder handles bound values.  
QueryExecutor executes the query.  
ResultMapper maps results.

That is strong SRP thinking.

## 2. Open/Closed Principle

Software should be open for extension but closed for modification.

Meaning: you should be able to add new behavior without constantly editing old, working code.

Bad:

</> PHP



```
class PaymentProcessor
{
    public function pay(string $type, float $amount): void
    {
        if ($type === 'paypal') {
            // PayPal logic
        } elseif ($type === 'stripe') {
            // Stripe logic
        } elseif ($type === 'authorize_net') {
            // Authorize.net logic
        }
    }
}
```

Every new payment method forces you to edit the same class.

Better:

</> PHP



```
interface PaymentGateway
{
    public function charge(float $amount): void;
}

class PayPalGateway implements PaymentGateway
{
    public function charge(float $amount): void
    {
```

```

        // PayPal logic
    }
}

class AuthorizeNetGateway implements PaymentGateway
{
    public function charge(float $amount): void
    {
        // Authorize.net logic
    }
}

class PaymentProcessor
{
    public function __construct(
        private PaymentGateway $gateway
    )

```

```

    {
        public function pay(float $amount): void
        {
            $this->gateway->charge($amount);
        }
    }
}

```

Now the processor does not care which gateway is used.

Applied to your API:

A query builder should not need to be rewritten every time you add a new query type.  
 A result mapper should allow new mapping strategies.  
 A factory should allow custom clients to be registered.  
 A cache layer should be swappable.

That is the Open/Closed Principle.

### 3. Liskov Substitution Principle

A child class or implementation should be usable anywhere the parent/interface is expected.

In plain English: if code expects a `PaymentGateway`, any `PaymentGateway` implementation should work without surprising behavior.

Bad:

</> PHP



```
interface Cache
{
    public function get(string $key): mixed;
    public function set(string $key, mixed $value): void;
}

class ReadOnlyCache implements Cache
{
    public function get(string $key): mixed
    {
        return null;
    }

    public function set(string $key, mixed $value): void
    {
        throw new Exception('Cannot write to read-only cache.');
```

This violates expectations. The interface says the cache can `set()`, but this implementation cannot.

Better:

</> PHP



```
interface CacheReader
{
    public function get(string $key): mixed;
}

interface CacheWriter
{
    public function set(string $key, mixed $value): void;
}
```

Now read-only cache can implement only the reader interface.

Applied to PHP:

This matters because PHP lets you type against interfaces:

</> PHP





```
function loadProduct(CacheReader $cache): Product
{
    // any CacheReader should work here
}
```

Your substitutions need to be behaviorally compatible, not just syntactically compatible.

## 4. Interface Segregation Principle

Do not force classes to implement methods they do not need.

Bad:

 PHP



```
interface DatabaseClient
{
    public function select(array $params): array;
    public function insert(array $params): int;
    public function update(array $params): int;
    public function delete(array $params): int;
    public function beginTransaction(): void;
```

