

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:

</> PHP



```
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:

`</> PHP`



```
// HTML, SQL, validation, business logic, and output all mixed together
```

Better:

`</> PHP`



```
$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.

`</> PHP`



```
$users = $userClient->findActiveUsersByRole('admin');
```

The caller does not need to manually build:

`</> SQL`



```
SELECT * FROM users WHERE status = ? AND role = ?
```

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;
    public function commit(): void;
    public function rollback(): void;
}
```

Some clients may only read. Some may only write. Some may not handle transactions.

Better:

 PHP



```
interface SelectClient
{
    public function select(array $params): array;
}

interface InsertClient
{
    public function insert(array $params): int;
}

interface UpdateClient
{
    public function update(array $params): int;
}

interface TransactionalClient
{
    public function beginTransaction(): void;
    public function commit(): void;
    public function rollback(): void;
}
```

Then a class implements only what it actually supports.

This is very relevant to PHP API design because PHP applications often grow organically. A giant interface becomes painful fast.

5. Dependency Inversion Principle

High-level code should depend on abstractions, not concrete low-level classes.

Bad:

</> PHP



```
class UserService
{
    private PDO $pdo;

    public function __construct()
    {
        $this->pdo = new PDO('mysql:host=localhost;dbname=app', 'user', 'pass');
    }
}
```

The service creates its own PDO connection. That makes testing harder and locks it to one implementation.

Better:

</> PHP



```
interface UserRepository
{
    public function findById(int $id): ?User;
}

class PdoUserRepository implements UserRepository
{
    public function __construct(
        private PDO $pdo
    ) {}

    public function findById(int $id): ?User
    {
        // PDO query here
    }
}

class UserService
{
    public function __construct(
        private UserRepository $users
    ) {}

    public function getUser(int $id): ?User
    {
        return $this->users->findById($id);
    }
}
```

Now `UserService` does not care whether the data comes from:

```
MySQL
SQLite
API
cache
test fake
mock object
```



Applied to your system, this is extremely important. If your API is designed so that another data layer can be plugged in later without changing the calling application code, that is dependency inversion.

How SOLID applies specifically to PHP

PHP lets you write very messy procedural code, but it also lets you write disciplined object-oriented code.

Modern PHP supports:

 PHP



```
declare(strict_types=1);

interface QueryExecutor
{
    public function execute(Query $query): QueryResult;
}

final class PdoQueryExecutor implements QueryExecutor
{
    public function __construct(
        private PDO $pdo
    ) {}

    public function execute(Query $query): QueryResult
    {
        // execute safely
    }
}
```

Good PHP development often means using the language in a disciplined subset:

- Use strict types where possible.
- Use namespaces.
- Use Composer autoloading.
- Use interfaces for swappable components.
- Use final classes where inheritance is not intended.
- Use value objects for meaningful domain concepts.
- Use exceptions intentionally.
- Avoid global state.
- Avoid mixing HTML, SQL, and business logic.
- Avoid giant arrays with unclear shapes unless documented or wrapped.



PHP does not force this discipline on you. That is part of why it is quirky. It gives you a lot of freedom, including freedom to create chaos.

Why PHP is quirky

PHP is quirky because it evolved from a practical web templating/tooling language into a full general-purpose programming language.

Some quirks include:

Function naming inconsistency

Examples:

 </> PHP

```
str_contains()  
array_key_exists()  
htmlspecialchars()  
mysqli_fetch_assoc()
```

Some functions use underscores. Some do not. Some parameter orders are inconsistent.

Loose typing history

Older PHP code often relies on implicit conversions:

 </> PHP

```
"5" == 5      // true  
"0" == false  // true
```

Modern PHP allows stricter coding, but the legacy behavior is still part of the language.

Prefer:

 </> PHP

```
declare(strict_types=1);  
  
if ($value === 5) {  
    // strict comparison  
}
```

Arrays do too much

PHP arrays are both lists and maps:

`</> PHP`



```
$list = ['a', 'b', 'c'];

$map = [
    'name' => 'Leonard',
    'role' => 'admin',
];
```

This is convenient but can become messy when data structures get complicated.

Web-first execution model

PHP was built around request/response web pages. That makes it excellent for many web applications, but it also shaped its culture and design.

Legacy baggage

PHP has improved dramatically, but it still carries compatibility decisions from older versions.

But PHP is also good

PHP deserves credit. It is not a joke language.

Modern PHP is capable of excellent software architecture. It powers serious systems. It is fast enough for a huge number of applications. Deployment is straightforward. Hosting support is everywhere. The standard web workflow is practical.

PHP is especially good for:

```
server-rendered websites
custom web applications
admin panels
e-commerce systems
content systems
database-driven applications
API backends
small-to-medium business software
```



Its biggest weakness is that it lets developers write bad code easily.

Its biggest strength is that a disciplined developer can build productive, maintainable systems quickly.

The real hierarchy

SOLID is important, but it is not the whole foundation.

A better hierarchy is:

1. Correctness
2. Security
3. Clarity
4. Maintainability
5. Separation of concerns
6. Testability
7. Reusability
8. Performance
9. SOLID principles
10. Design patterns



SOLID is a tool for achieving maintainability, extensibility, and testability. It is not a religion.

Sometimes a simple function is better than five classes and three interfaces.

For example:

</> PHP



```
function secondsFromIsoDuration(string $duration): int
{
    $interval = new DateInterval($duration);

    return ($interval->h * 3600)
        + ($interval->i * 60)
        + $interval->s;
}
```

That does not need a full object model unless it becomes part of a larger reusable system.

But for your database API, SOLID is highly relevant because you are building reusable infrastructure. Infrastructure code benefits greatly from:

small classes
clear responsibilities
interfaces
swappable components
value objects
factories
dependency injection
careful result mapping



That is exactly the kind of software where SOLID pays off.



...

Sources