
Solid
ERYISSENAL PACK NEL

ORIGIN

The origins of the SOLID principles can be traced back to the work of**Robert C. Martin**(also known as "Uncle Bob"). He first introduced these concepts in his 2000 article "Design Principles and Design Patterns."

A **data**in which the principles were formally grouped under the acronym SOLID was around**2004**, when **Michael Feathers**coined the term to facilitate the memorization and discussion of these important object-oriented design concepts.

The**aim**of the SOLID principles is to guide developers in creating software that is:

- **Easy to understand:**Well-structured code with clear responsibilities is easier to understand.
- **Flexible:**The ability to extend behavior without modifying existing code makes it easier to adapt to new requirements.
- **Maintainable:**Code with low coupling and high cohesion is easier to change and fix over time.
- **Reusable:**Well-defined components with unique responsibilities can be used in different parts of the system or in other projects.
- **Testable:**Smaller, single-task classes are easier to unit test.
- **With loose coupling:**Classes are less dependent on each other, which makes the system more robust to change.

In short, SOLID principles aim to reduce the "fragility" and "rigidity" of code, promoting a cleaner, more modular, and sustainable design in the long run.

ABOUT THE AUTHOR

The ability to program is necessary for every man, regardless of the future. Today, with the help of artificial intelligence, new programs can be created very easily. However, the programmer is necessary because creation involves many steps, and almost everything we have, man has created and will continue to do, that is, inventing with or without the help of artificial intelligence, or putting everything into the world that it does not yet do.

The author has several activities in society, whether as an unethical hacker or programming, many think they have some connection, others try, he has already done it, this work aims to help and improve, not only in relation to solid, in stages of different ways to be able to create, and make new and diverse other software.

SINGLE LIABILITY PRINCIPLE

The**Single Responsibility Principle (SRP)**, the first letter of the acronym SOLID, states that**a class should have only one reason to change**. In other words, a class should have a single responsibility.

Let's break this principle down:

What does "a responsibility" mean?

A responsibility can be understood as a group of related functionalities that change for the same reason. If a class has multiple responsibilities, any change to one of these responsibilities can affect the others, leading to more fragile and difficult-to-maintain code.

What is the purpose of SRP?

The main objective of the SRP is to promote:

- **High cohesion:**A class with a single responsibility tends to have its elements (attributes and methods) highly related to each other.
- **Low coupling:**When a class has a single responsibility, it tends to depend less on other classes, making the system more flexible and resistant to change.
- **Ease of maintenance:**If a class has only one reason to change, it's easier to understand the impact of a change and perform tests focused on that responsibility.
- **Reusability:**Classes with well-defined responsibilities are easier to reuse in different parts of the system or in other projects.
- **Testability:**It is simpler to write unit tests for classes that have a single responsibility, as the scope of the test is well defined.

How to identify if a class violates the SRP?

A class may be violating the SRP if:

- It has many different responsibilities.
- Its name does not clearly reflect a single purpose.
- Changes in different parts of the system often lead to modification of this class.
- It has many unrelated methods.

Example (in pseudocode):

Imagine a classUserReportwhich has two responsibilities: generating a user report and saving that report to a file.

class RelatorioUsuario

{ fun gerarRelatorio(listaDeUsuarios): String { // Logic to generate the report }

fun salvarRelatorio(conteudoDoRelatorio, nomeDoArquivo) { // Logic to save the report to a file } }

This class violates SRP because it has two reasons to change: the report generation logic may change (format, information included) and the way the report is saved may change (file type, storage location).

How to apply the SRP?

To apply the SRP, the class`UserReport`could be refactored into two separate classes:

```
class UserReportGenerator {

fun generateReport(listOfUsers): String {

// Logic to generate the report

   }

}
```

```
class PersistenciaRelatorio {

   fun salvarRelatorio(conteudoDoRelatorio, nomeDoArquivo) {

// Logic to save the report to a file

   }

}
```

Now, each class has a single responsibility and a single reason for change. If the report generation method needs to be changed, only the class`UserReportGenerator`will be modified. If the way you save the report changes, only the class`PersistenciaRelatorio`will be changed.

By following the Single Responsibility Principle, you contribute to more organized, flexible, and maintainable code in the long run.

EM PHP.

Example of SRP Violation in PHP:

```
<?php

class RelatorioUsuario {

   public function gerarRelatorio(array $listaDeUsuarios): string {

// Logic to generate the report (plain text format)

$relatorio = "User Report:\n";

foreach ($listOfUsers as $user) {

$relatorio .= "- Nome: " . $usuario['nome'] . ", Email: " . $usuario['email']
. "\n";

      }

return $relatorio;

   }

   public function salvarRelatorio(string $conteudoDoRelatorio, string
$nomeDoArquivo): void {

// Logic to save the report to a file

      file_put_contents($nomeDoArquivo, $conteudoDoRelatorio);

echo "Report saved in: " . $fileName . "\n";

   }

}
```

```
// Usage:
$users = [
['name' => 'John', 'email' => 'john@example.com'],
['name' => 'Maria', 'email' => 'maria@example.com'],
];

$relatorioUsuario = new RelatorioUsuario();
$relatorio = $relatorioUsuario->generateRelatorio($usuarios);
echo $relatorio;
$relatorioUsuario->saveRelatorio($relatorio, 'relatorio_usuarios.txt');

?>
```

Example of SRP Application in PHP:

```
<?php
class UserReportGenerator {
    public function gerarRelatorio(array $listaDeUsuarios): string {
// Logic to generate the report (plain text format)
$relatorio = "User Report:\n";
foreach ($listOfUsers as $user) {
$relatorio .= "- Nome: " . $usuario['nome'] . ", Email: " . $usuario['email']
. "\n";
```

```
        }
return $relatorio;
    }
}

class PersistenciaRelatorio {
    public function salvarRelatorio(string $conteudoDoRelatorio, string $nomeDoArquivo): void {
// Logic to save the report to a file
        file_put_contents($nomeDoArquivo, $conteudoDoRelatorio);
echo "Report saved in: " . $fileName . "\n";
    }
}

// Usage:
$users = [
['name' => 'John', 'email' => 'john@example.com'],
['name' => 'Maria', 'email' => 'maria@example.com'],
];

$reportgenerator = new UserReportGenerator();
$report = $generatorReport->generateReport($users);
```

```
echo $relatorio;


$persistenciaRelatorio = new PersistenciaRelatorio();

$persistenceReport->saveReport($report, 'report_users.txt');


?>
```

Now, each class has a single responsibility and a single reason for change. If the report generation method needs to be changed, only the class`UserReportGenerator`will be modified. If the way you save the report changes, only the class`PersistenciaRelatorio`will be changed.

By following the Single Responsibility Principle, you contribute to more organized, flexible, and maintainable code in the long run.

OPEN AND CLOSED PRINCIPLE

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Applying this to our example, imagine that we now need to generate reports in different formats (e.g., CSV and PDF). If we directly modify the classUserReportGeneratorAdding this new functionality would violate the OCP. The class would be open to extension (new formats), but we would also need to modify it (add logic for the new formats).

The solution to apply OCP would be to use**abstraction**We could create an interface or an abstract class to define the contract of a report generator, and then create concrete classes for each specific format.

```
interface ReportGenerator {

fun generate(listOfUsers): String

}


class CSVReportGenerator : ReportGenerator {

override fun generate(listOfUsers): String {

// Logic to generate report in CSV format

   }

}


class PDFReportGenerator : ReportGenerator {
```

```
override fun generate(listOfUsers): String {
// Logic to generate report in PDF format
    }
}

// The client class now depends on the abstraction (interface)
class ServicoDeRelatorio {
private var generator: ReportGenerator

fun ReportService(generator: ReportGenerator) {
this.generator = generator
    }

fun generateReport(listOfUsers): String {
return generator.generate(listOfUsers)
    }
}

// Usage:
val generatorCSV = GeneratorReportCSV()
val servicoCSV = ReportService(CSVgenerator)
```

val relatorioCSV = servicoCSV.generateRelatorio(usuarios)

val generatorPDF = GeneratorPDFReport()

val servicoPDF = ServicoDeRelatorio(generadorPDF)

val relatorioPDF = servicoPDF.generateRelatorio(usuarios)

In this example, the class`ServicoDeRelatorio`is open to work with different types of report generators (extension), without needing to be modified when a new format is added. Simply create a new class that implements the interface`Report Generator`.

Example of OCP Violation in PHP:

<?php

class UserReportGenerator {

public function gerarRelatorio(array $listaDeUsuarios, string $formato): string {

if ($format === 'text') {

$relatorio = "User Report (Text):\n";

foreach ($listOfUsers as $user) {

$relatorio .= "- Nome: " . $usuario['nome'] . ", Email: " . $usuario['email'] . "\n";

}

return $relatorio;

} elseif ($format === 'csv') {

```
$report = "Name,Email\n";

foreach ($listOfUsers as $user) {

$relatorio .= $usuario['nome'] . "," . $usuario['email'] . "\n";

        }

return $relatorio;

      }

throw new InvalidArgumentException("Unsupported report format: " .
$format);

   }

}


// Usage:

$users = [

['name' => 'John', 'email' => 'john@example.com'],

['name' => 'Maria', 'email' => 'maria@example.com'],

];


$reportgenerator = new UserReportGenerator();

echo $generatorReport->generateReport($users, 'text');

echo "\n";

echo $generatorReport->generateReport($users, 'csv');
```

// To add a new format (e.g. HTML), we would need to MODIFY the GeradorRelatorioUsuario class.

// This violates the Open/Closed Principle.

?>

In this example, the classUserReportGeneratoris responsible for generating reports in different formats. If we need to add a new format (such as HTML), we will have to**modify**the class, adding another blockelseifThis violates the Open/Closed Principle as the class is not closed to modification to extend its functionality.

Example of OCP Application in PHP:

<?php

interface ReportGenerator {

public function generate(array $listaDeUsuarios): string;

}

class TextReportGenerator implements ReportGenerator {

public function generate(array $listOfUsers): string {

$relatorio = "User Report (Text):\n";

foreach ($listOfUsers as $user) {

$relatorio .= "- Nome: " . $usuario['nome'] . ", Email: " . $usuario['email'] . "\n";

```
        }
return $relatorio;
    }
}

class CSVReportGenerator implements ReportGenerator {
public function generate(array $listOfUsers): string {
$report = "Name,Email\n";
foreach ($listOfUsers as $user) {
$relatorio .= $usuario['nome'] . "," . $usuario['email'] . "\n";
        }
return $relatorio;
    }
}

// To add a new format (e.g. HTML), we create a new class
class ReportGeneratorHTML implements ReportGenerator {
public function generate(array $listOfUsers): string {
$relatorio = "<h1>User Report (HTML)</h1><ul>";
foreach ($listOfUsers as $user) {
```

```
        $report .= "<li><strong>Name:</strong> " .
htmlspecialchars($user['name']) . ", <strong>Email:</strong> " .
htmlspecialchars($user['email']) . "</li>";

    }

$relatorio .= "</ul>";

return $relatorio;

  }

}


// Client class that depends on the abstraction (interface)

class ServicoDeRelatorio {

private $generator;


public function __construct(ReportGenerator $generator) {

$this->generator = $generator;

  }


  public function gerarRelatorio(array $listaDeUsuarios): string {

return $this->generator->generate($listOfUsers);

  }

}
```

```
// Usage:
$users = [
['name' => 'John', 'email' => 'john@example.com'],
['name' => 'Maria', 'email' => 'maria@example.com'],
];

$generatorText = new ReportService(new ReportGeneratorText());
echo $generatorText->generateReport($users);
echo "\n";

$geradorCSV = new ServicoDeRelatorio(new GeradorRelatorioCSV());
echo $generatorCSV->generateReport($users);
echo "\n";

$generatorHTML = new ReportService(new ReportGeneratorHTML());
echo $generatorHTML->generateReport($users);
// To add a new format, we just create a new class that implements GeradorRelatorio.
// The ServicoDeRelatorio class does not need to be modified.
?>
```

Liskov Substitution Principle (LSP)

The**Liskov Substitution Principle (LSP)**, the third letter of the acronym SOLID, states that**subtypes must be substitutable for their base types without altering the correctness of the program**.

In simplest terms, if you have a base class (superclass) and a derived class (subclass), you should be able to use any object from the subclass in place of an object from the superclass without the program breaking or behaving unexpectedly.

What does this mean in practice?

- **Inheritance must represent an "is a" relationship:**The subclass should actually be a more specific type of the superclass and behave consistently with it.
- **Contracts must be preserved:**Subclasses should not weaken the preconditions of superclass methods (what must be true before the method is executed) nor strengthen the postconditions (what must be true after the method is executed).
- **Invariants must be maintained:**Superclass invariants (rules that must always be true for superclass objects) must also be true for subclass objects.
- **Exceptions must be consistent:**Subclasses should not throw exceptions that the superclass does not expect, unless those exceptions are subtypes of the exceptions thrown by the superclass.

Example of LSP violation (in pseudocode):

Imagine a base classRectanglewith methods to define width and height, and calculate area.

```
class Rectangle {

var width: Integer

var height: Integer


fun setWidth(width: Integer) {

this.width = width

    }


fun setHeight(height: Integer) {

this.height = height

    }


fun calculateArea(): Integer {

return width * height

    }

}
```

Now, imagine a subclassSquarethat wounds ofRectangle, because a square "is a" rectangle with equal sides.

```
class Square : Rectangle {
override fun setWidth(side: Integer) {
super.defineLength(side)
super.setHeight(side)
    }

override fun setHeight(side: Integer) {
super.defineLength(side)
super.setHeight(side)
    }
}
```

Consider the following client code that expects to work with aRectangle:

```
fun increaseWidth(rectangle: Rectangle) {
rectangle.setWidth(rectangle.width + 5)
// Width is expected to increase, but height may remain the same
}

val ret = Rectangle()
ret.defineWidth(10)
right.defineAltura(5)
increaseWidth(ret)
```

```
print(ret.calculateArea()) // Output: 75 (expected)


val square = Square()

square.setWidth(10)

square.setHeight(10)

increaseWidth(square)

print(square.calculateArea()) // Output: 225 (unexpected - height also changed)
```

In this example, when passing an objectSquarefor the functionincreaseWidth, the expected behavior for aRectangle(where only the width increases) is not maintained. TheSquarechanges both the width and height, violating the LSP. TheSquareis not a perfect substitute forRectangle in this context.

How to apply LSP?

To apply LSP, the relationship betweenRectangleandSquarecould be rethought. One possible solution would be to not use inheritance in this case, or perhaps create a different abstraction that represents the idea of a geometric shape with sides that can be modified independently.

Another approach would be to ensure that subclass methods behave consistently with the superclass's expectations. In the case ofSquare, perhaps it would not be appropriate to have separate methods for setting width and height, but rather one method for setting the "side".

Following the Liskov Substitution Principle is crucial to creating robust and extensible systems, where inheritance is used correctly and different types can be treated polymorphically without causing unexpected side effects.

Example of LSP Violation in PHP:

```
<?php


class Rectangle {
    protected $largura;
    protected $altura;


    public function __construct(int $largura, int $altura) {
$this->width = $width;
$this->height = $height;
    }


    public function definirLargura(int $largura): void {
$this->width = $width;
    }


    public function definirAltura(int $altura): void {
$this->height = $height;
    }
```

```
    public function obterLargura(): int {
        return $this->largura;
    }

    public function obterAltura(): int {
        return $this->height;
    }

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

class Square extends Rectangle {
    public function __construct(int $lado) {
parent::__construct($side, $side);
    }

    public function definirLargura(int $lado): void {
parent::setWidth($side);
parent::setHeight($side);
```

```
    }

    public function definirAltura(int $lado): void {
parent::setWidth($side);
parent::setHeight($side);
    }
}

function increaseWidth(Rectangle $rectangle): void {
$originalwidth = $rectangle->getWidth();
$rectangle->setWidth($originalWidth + 5);
echo "Width increased to: " . $rectangle->getWidth() . ", Area is now: " . $rectangle->calculateArea() . "\n";
// Width is expected to increase, but height may remain the same
}

// Testing with Rectangle
$ret = new Rectangle(10, 5);
echo "Original rectangle: Width = " . $ret->getWidth() . ", Height = " . $ret->getHeight() . ", Area = " . $ret->calculateArea() . "\n";
increaseWidth($ret); // Expected output: Width increased to: 15, Area is now: 75
```

```
// Testing with Square (violating LSP)

$square = new Square(10);

echo "Original square: Side = " . $square->getWidth() . ", Area = " .
$square->calculateArea() . "\n";

increaseWidth($square); // Unexpected output: Width increased to: 15,
Area is now: 225 (height also changed)


?>
```

In this PHP example, the function`increaseWidth`expects to receive an object of type`Rectangle`and only increase its width. However, when an object of the subclass`Square`is passed, changing the width also affects the height, leading to unexpected behavior and violating the LSP. The`Square`does not behave in a manner consistent with the expectations of the superclass`Rectangle`in this specific context.

How to Apply LSP in PHP (Refactoring):

One way to implement LSP would be to rethink the inheritance relationship or introduce a different abstraction. A possible solution would be to have an interface for geometric shapes that have a calculable area, and perhaps treat rectangles and squares differently if their manipulation properties are fundamentally different.

```
<?php

interface GeometricForm {
    public function calcularArea(): int;
}

class Retangulo implements FormaGeometrica {
    protected $largura;
    protected $altura;

    public function __construct(int $largura, int $altura) {
$this->width = $width;
$this->height = $height;
    }

    public function definirLargura(int $largura): void {
$this->width = $width;
    }

    public function definirAltura(int $altura): void {
$this->height = $height;
```

```
    }

    public function obterLargura(): int {
        return $this->largura;
    }

    public function obterAltura(): int {
        return $this->height;
    }

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

class Quadrado implements FormaGeometrica {
protected $side;

    public function __construct(int $lado) {
$this->side = $side;
    }
```

```
    public function setSide(int $side): void {
$this->side = $side;
    }

    public function obterLado(): int {
        return $this->lado;
    }

    public function calcularArea(): int {
return $this->side * $this->side;
    }
}

function imprimirArea(FormaGeometrica $forma): void {
echo "The area of the shape is: " . $shape->calculateArea() . "\n";
}

// Testing with Rectangle
$ret = new Rectangle(10, 5);
printArea($ret); // Output: The area of the shape is: 50
```

```
// Testing with Square

$square = new Square(10);

printArea($square); // Output: The area of the shape is: 100


// The increaseWidth function no longer makes sense in this context,

// because the manipulation of a Square is different.


?>
```

In this refactoring,RectangleandSquareimplement a common interfaceGeometric Shapethat defines the contract for calculating the area. The manipulation of dimensions (width, height, side) is done independently in each class, respecting its own rules. The functionimprimirAreanow works with anyGeometric Shapewithout making assumptions about how their dimensions are defined, avoiding LSP violation.

This PHP example illustrates how misuse of inheritance can lead to violation of the Liskov Substitution Principle, and how refactoring, possibly using interfaces and composition instead of direct inheritance, can help adhere to this important principle of object-oriented design.

Interface Segregation Principle (ISP):

Now, let's address the**Interface Segregation Principle (ISP)**:

Interface Segregation Principle (ISP): No customer should be forced to rely on methods they do not use.

Imagine we added a new feature to our interfaceReport Generator: the ability to preview the report on the screen.

```
interface ReportGenerator {

fun generate(listOfUsers): String

fun preview(): void // New functionality

}

class CSVReportGenerator : ReportGenerator {

override fun generate(listOfUsers): String {

      // ...

   }

override fun preview() {

// It may not make sense to preview a CSV directly

      throw UnsupportedOperationException()

   }

}

class PDFReportGenerator : ReportGenerator {

override fun generate(listOfUsers): String {
```

```
        // ...
    }
override fun preview() {
// Logic to preview the PDF
    }
}
```

EM PHP

```
<?php

interface ReportGenerator {
public function generate(array $listaDeUsuarios): string;
}

class TextReportGenerator implements ReportGenerator {
public function generate(array $listOfUsers): string {
$relatorio = "User Report (Text):\n";
foreach ($listOfUsers as $user) {
$relatorio .= "- Nome: " . $usuario['nome'] . ", Email: " . $usuario['email']
. "\n";
        }
return $relatorio;
    }
}

class CSVReportGenerator implements ReportGenerator {
```

```
public function generate(array $listOfUsers): string {
$report = "Name,Email\n";
foreach ($listOfUsers as $user) {
$relatorio .= $usuario['nome'] . "," . $usuario['email'] . "\n";
      }
return $relatorio;
   }
}

// To add a new format (e.g. HTML), we create a new class
class ReportGeneratorHTML implements ReportGenerator {
public function generate(array $listOfUsers): string {
$relatorio = "<h1>User Report (HTML)</h1><ul>";
foreach ($listOfUsers as $user) {
         $report .= "<li><strong>Name:</strong> " .
htmlspecialchars($user['name']) . ", <strong>Email:</strong> " .
htmlspecialchars($user['email']) . "</li>";
      }
$relatorio .= "</ul>";
return $relatorio;
   }
}

// Client class that depends on the abstraction (interface)
class ServicoDeRelatorio {
private $generator;

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

   public function gerarRelatorio(array $listaDeUsuarios): string {
return $this->generator->generate($listOfUsers);
```

```
    }
}

// Usage:
$users = [
['name' => 'John', 'email' => 'john@example.com'],
['name' => 'Maria', 'email' => 'maria@example.com'],
];

$generatorText = new ReportService(new ReportGeneratorText());
echo $generatorText->generateReport($users);
echo "\n";

$geradorCSV = new ServicoDeRelatorio(new GeradorRelatorioCSV());
echo $generatorCSV->generateReport($users);
echo "\n";

$generatorHTML = new ReportService(new ReportGeneratorHTML());
echo $generatorHTML->generateReport($users);

// To add a new format, we just create a new class that implements
GeradorRelatorio.
// The ServicoDeRelatorio class does not need to be modified.

?>
```

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP). We'll explore it with explanation, pseudocode, and PHP examples.

Dependency Inversion Principle (DIP):

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions. [1]

In essence, DIP advocates that high-level classes (which contain business logic) should not directly depend on low-level classes (which implement details like database access, file systems, etc.). Instead, both should depend on interfaces or abstract classes.

DIP Objectives:

- **Reduce coupling:**By relying on abstractions rather than concrete implementations, classes become less dependent on each other. This makes the system more flexible and resilient to change.
- **Increase reuse:**Well-defined abstractions can be implemented in different ways and reused in different parts of the system.
- **Improve testability:**It's easier to test high-level classes when their dependencies are abstractions, because we can use mocks or stubs to simulate the behavior of low-level dependencies.

Pseudocode Example (DIP Violation):

```
class RepositorioDeClientes {

fun getClientById(id: Integer): Client {
```

```
// Specific logic for accessing the MySQL database
// to search for the client by ID
    }
}


class ServicoDeCliente {
var repository: ClientRepository

    fun ServicoDeCliente() {
this.repositorio = new RepositorioDeClientes() // Direct dependency on the concrete implementation
    }

fun buscarCliente(id: Integer): Cliente {
return repository.getClientById(id)
    }
}

// High-level module (ClientService) depends directly on the low-level module (ClientRepository)
// If we want to change the database to PostgreSQL, we would need to modify ServicoDeCliente.
```

In this example,Customer Service(high level) depends directly on the concrete implementationRepositorioDeClientes(low-level) that is specific to a MySQL database. If we decide to change the database, we will have to modify the classCustomer Service, violating the DIP.

Pseudocode Example (DIP Application):

```
interface IRepositorioDeClientes {
fun getClientById(id: Integer): Client
}

class RepositorioDeClientesMySQL : IRepositorioDeClientes {
fun getClientById(id: Integer): Client {
// Specific logic for accessing the MySQL database
   }
}

class RepositorioDeClientesPostgreSQL : IRepositorioDeClientes {
fun getClientById(id: Integer): Client {
// Specific logic for accessing the PostgreSQL database
   }
}

class ServicoDeCliente {
var repository: IClientRepository

// The dependency is now on the abstraction (interface)
fun CustomerService(repository: ICustomerRepository) {
this.repository = repository
   }
```

```
fun buscarCliente(id: Integer): Cliente {
return repository.getClientById(id)
   }
}

// Usage:
var mysqlRepo = new RepositorioDeClientesMySQL()
var clienteService1 = new ServicoDeCliente(mysqlRepo)
clientService1.searchClient(123)

var postgresRepo = new RepositorioDeClientesPostgreSQL()
var clienteService2 = new ServicoDeCliente(postgresRepo)
clientService2.searchClient(456)

// ServicoDeCliente doesn't care about the specific implementation of the repository.
// We can easily change the implementation without modifying ServicoDeCliente.
```

Now, both the high-level module (Customer Service) and low-level modules (RepositorioDeClientesMySQL, RepositorioDeClientesPostgreSQL) depend on abstractionCustomerRepository. The classCustomer Servicedoesn't care which specific repository implementation is being used. This makes the system more flexible.

PHP Example (DIP Violation):

```
<?php

class MySQLClienteRepository {
    public function getClientePorId(int $id): array {
// Simulation of access to the MySQL database
        return ['id' => $id, 'nome' => 'Cliente do MySQL'];
    }
}

class ClienteService {
    private $clienteRepository;

    public function __construct() {
$this->clienteRepository = new MySQLClienteRepository(); // Direct dependency
    }

    public function buscarCliente(int $id): array {
        return $this->clienteRepository->getClientePorId($id);
    }
```

```
}


// Usage:
$clienteService = new ClienteService();
$client = $clientService->searchClient(1);
print_r($client);


// If we want to use another database, we need to modify
ClienteService.
```

PHP Example (DIP Application):

```
<?php

interface ClienteRepositoryInterface {
   public function getClientePorId(int $id): array;
}

class MySQLClienteRepository implements ClienteRepositoryInterface {
   public function getClientePorId(int $id): array {
// Simulation of access to the MySQL database
      return ['id' => $id, 'nome' => 'Cliente do MySQL'];
   }
}

class PostgreSQLClienteRepository implements
ClienteRepositoryInterface {
   public function getClientePorId(int $id): array {
// Simulation of access to the PostgreSQL database
      return ['id' => $id, 'nome' => 'Cliente do PostgreSQL'];
   }
}

class ClienteService {
   private $clienteRepository;

// Dependency injection through the constructor
   public function __construct(ClienteRepositoryInterface
$clienteRepository) {
      $this->clienteRepository = $clienteRepository;
   }

   public function buscarCliente(int $id): array {
```

```
        return $this->clienteRepository->getClientePorId($id);
    }
}

// Usage:
$mysqlRepo = new MySQLClienteRepository();
$clienteService1 = new ClienteService($mysqlRepo);
$client1 = $clientService1->searchClient(1);
print_r($cliente1);

$postgresRepo = new PostgreSQLClienteRepository();
$clienteService2 = new ClienteService($postgresRepo);
$client2 = $clientService2->searchClient(2);
print_r($cliente2);

// ClientService doesn't care about the specific repository
implementation.
```

In the PHP example that applies the DIP, the class`ClienteService`depends on an interface`ClienteRepositoryInterface`instead of a concrete implementation as`MySQLClienteRepository`. The repository-specific implementation is injected into the`ClienteService`through the constructor. This allows us to easily change the repository implementation without modifying the class`ClienteService`, adhering to the Dependency Inversion Principle.

I hope this explanation and the pseudocode and PHP examples of the Dependency Inversion Principle were clear and helpful! With this, we've covered all five SOLID principles.

GANG'S SINISTER RELATIONSHIP WITH SOLID

CREATIONAL GANG

Regarding the Gang of Four (GoF) "Creational Patterns", there are five design patterns defined in this category:

1. Factory Method: Defines an interface for creating an object, but allows subclasses to change the type of objects that will be created.
2. Abstract Factory: Provides an interface for creating families of related or dependent objects, without specifying their concrete classes.
3. Builder: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
4. Prototype: Specifies the types of objects to be created using a prototypical instance and creates new objects by copying this prototype.
5. Singleton: Ensures that a class has only one instance and provides a global access point to it.

THE CREATIVE GANG WITH SOLID

Of the five creational patterns of the GoF, Factory Method, Abstract Factory, and Builder are the ones that have the most direct and significant relationship with the SOLID principles, especially with SRP (Single Responsibility) and OCP (Open/Closed), as discussed previously.

Let's briefly recap how each relates to SOLID:

- Factory Method e Abstract Factory:
    - SRP: Delegate the responsibility of creating objects to separate factory classes, isolating this responsibility from the main business classes.
    - OCP: Allows the introduction of new types of objects (or families of objects) through the creation of new subclasses of factories and products, without the need to modify existing client code that depends on the factory interfaces.
    - DIP: Promotes reliance on abstractions (factory and product interfaces) rather than concrete implementations.
- Builder:
    - SRP: Separates the responsibility for the complex construction of an object from its representation and the class that represents it. Each ConcreteBuilder is responsible for building a specific representation.
    - OCP: Makes it easy to add new ways to construct an object by creating new ConcreteBuilders without changing the client code that uses the Builder interface.
    - DIP: By relying on Builder interfaces, client code becomes less coupled to the concrete implementations of the build.

The other two creational patterns have a less direct relationship to the principles we mentioned in the context of creational patterns:

- Prototype: Prototype's main focus is creating objects by cloning existing instances. While it can help reduce coupling to the direct creation of concrete classes, its relationship to SRP and OCP is not as central as that of the Factory and Builder patterns.

- Singleton: The Singleton ensures that a class has only one instance. While useful in certain scenarios, it can actually violate the SRP if the Singleton class takes on too many responsibilities beyond ensuring its single instance. It can also hinder testability and introduce global dependencies, which can violate the DIP in some cases. Its relationship to the OCP is not a primary focus.

Therefore, of the five creational patterns of the GoF, the three that relate most strongly to the SOLID principles (especially SRP and OCP, in the context of our earlier discussion of creational patterns) are: Factory Method, Abstract Factory, and Builder.

Creational Design Patterns: Factory Method and Abstract Factory in PHP

The standards**Factory Method**and**Abstract Factory**are creational design patterns that provide ways to **abstract the process of creating objects**. They help make code more flexible, extensible, and less coupled by delegating the responsibility of instantiating concrete classes to subclasses or another factory.

1. Factory Method

Purpose: It defines an interface for creating an object, but allows subclasses to change the type of objects created. In other words, a class delegates the responsibility of instantiating objects to its subclasses.

Structure:

- **Product:**Defines the interface of the object that the factory creates.
- **Concrete Product:**Implements the Product interface.
- **Creator:**Declares the factory method that returns a Product object. The Creator may contain some default business logic that depends on the Product objects it creates.
- **Concrete Creator:**Override the factory method to return an instance of a specific Concrete Product.

PHP example:

Let's imagine we have different types of transportation (Car and Truck) and we want a way to create them without directly depending on their concrete classes.

StandardsCreational Project: Factory Method and Abstract Factory in PHP

The standards**Factory Method**and**Abstract Factory**are creational design patterns that provide ways to**abstract the process of creating objects**. They help make code more flexible, extensible, and less coupled by delegating the responsibility of instantiating concrete classes to subclasses or another factory.

1. Factory Method

Purpose: It defines an interface for creating an object, but allows subclasses to change the type of objects created. In other words, a class delegates the responsibility of instantiating objects to its subclasses.

Structure:

- **Product:**Defines the interface of the object that the factory creates.
- **Concrete Product:**Implements the Product interface.
- **Creator:**Declares the factory method that returns a Product object. The Creator may contain some default business logic that depends on the Product objects it creates.
- **Concrete Creator:**Override the factory method to return an instance of a specific Concrete Product.

PHP example:

Let's imagine we have different types of transportation (Car and Truck) and we want a way to create them without directly depending on their concrete classes.

PHP

```
<?php



// Product Interface

interface Transport {
```

```
    public function entregar(): string;
}

// Concrete Products
class Car implements Transport {
    public function entregar(): string {
return "Delivering by car.";
    }
}

class Caminhao implements Transporte {
    public function entregar(): string {
return "Delivering by truck.";
    }
}

// Creator
abstract class Logistica {
    abstract public function createTransport(): Transport;

    public function planejarEntrega(): string {
$transport = $this->createTransporte();
return "Planning delivery." . $transport->deliver();
```

```
    }
}

// Concrete Creators
class LogisticaRodoviaria extends Logistica {
    public function createTransport(): Transport {
return new Car();
    }
}

class LogisticaMaritima extends Logistica {
    public function createTransport(): Transport {
return new Caminhao();
    }
}

// Client
function executarEntrega(Logistica $logistica) {
echo $logistics->deliveryplan() . "\n";
}


// Usage
```

executeDelivery(new LogisticaRodoviaria()); // Output: Planning delivery. Delivering by car.

executeDelivery(new LogisticaMaritima()); // Output: Planning delivery. Delivering by truck.

?>

Advantages of the Factory Method:

- **Flexibility:**Makes it easy to introduce new product types without changing existing client code. Simply create a new Concrete Creator.
- **Low Coupling:**The client code works with an interface (Creator and Product), without depending on the specific product classes.
- **Personalization:**Creator subclasses can customize the type of product that is created.

2. Abstract Factory

Purpose: Provides an interface for creating families of related or dependent objects without specifying their specific classes. Use an abstract factory when families of related products need to be used together.

Structure:

- **Abstract Factory:**Declares methods for creating each of the distinct products in a family.
- **Concrete Factory (ConcreteFactory):**Implements abstract factory methods to create instances of specific concrete products in a family.
- **Abstract Product:**Declares the interface for a product type.

- **Concrete Product:**Implements the abstract product interface and belongs to a specific concrete factory.
- **Customer:**It works with the abstract factory and the interfaces of abstract products, without knowing the concrete classes that are being created.

PHP example:

Let's imagine that we need to create graphical interface elements (Buttons and Checkboxes) for different operating systems (Windows and macOS).

```
<?php

// Abstract Products
interface Button {
   public function renderizar(): string;
}

interface Checkbox {
   public function renderizar(): string;
}

// Concrete Products for Windows
class WindowsBotao implements Botao {
   public function renderizar(): string {
return "Rendering Windows Button.";
   }
}

class WindowsCheckbox implements Checkbox {
   public function renderizar(): string {
      return "Renderizando Checkbox do Windows.";
```

```
    }
}

// Concrete Products for macOS
class MacOSBotao implements Botao {
    public function renderizar(): string {
return "Rendering macOS Button.";
    }
}

class MacOSCheckbox implements Checkbox {
    public function renderizar(): string {
        return "Renderizando Checkbox do macOS.";
    }
}

// Abstract Factory
interface GUIFactory {
public function createButton(): Button;
    public function criarCheckbox(): Checkbox;
}

// Concrete Factories
class WindowsFactory implements GUIFactory {
    public function createButton(): Boot {
        return new WindowsBotao();
    }

    public function criarCheckbox(): Checkbox {
        return new WindowsCheckbox();
    }
}
```

```
class MacOSFactory implements GUIFactory {
    public function createButton(): Boot {
        return new MacOSBotao();
    }

    public function criarCheckbox(): Checkbox {
        return new MacOSCheckbox();
    }
}

// Client
function configurarAplicacao(GUIFactory $factory) {
$button = $factory->createButton();
    $checkbox = $factory->criarCheckbox();
echo $button->render() . "\n";
echo $checkbox->renderize() . "\n";
}

// Usage
echo "Configuring for Windows:\n";
configurarAplicacao(new WindowsFactory());
// Exit:
// Rendering Windows Button.
// Renderizando Checkbox do Windows.

echo "\nConfiguring for macOS:\n";
configurarAplicacao(new MacOSFactory());
// Exit:
// Rendering macOS Button.
// Rendering macOS Checkbox.

?>
```

Advantages of Abstract Factory:

- **Consistency:**Ensures that products in a family are compatible with each other.
- **Insulation:**The client code does not depend on the concrete product classes, only on the interfaces.
- **Facility of Family Exchange:**Exchanging one product family for another is simple, just provide a different concrete factory.

When to use:

- When the system needs to create families of related or dependent objects.
- When it is important that objects created together are compatible.
- When you want to abstract how object families are created.

In summary:

- **Factory Method:**Useful when a class doesn't know which object subclass it needs to create and delegates that decision to its subclasses. Focused on creating a single type of product.
- **Abstract Factory:**Useful when you need to create families of related objects and ensure they work together. Focused on creating a set of related products.

Both standards promote the principle of**Dependency Inversion (DIP)**by making client code depend on abstractions (interfaces or abstract classes) rather than concrete implementations. They can also contribute to the principle**Open/Closed (OCP)**, because adding new product types or product families usually involves creating new subclasses of factories and products, without modifying existing client code.

BUILDER

Yes, the design pattern**Builder**also correlates with the SOLID principles, especially with the**SRP (Single Responsibility Principle)** and the **OCP (Open/Closed Principle)**, although the correlation with OCP is a bit more indirect in some implementations.

SRP (Single Responsibility Principle) and Builder:

- **Separation of Construction from Representation:**The main goal of the Builder pattern is to separate the construction of a complex object from its representation. The "Director" class (optional) or the client itself orchestrates the construction steps, while the "ConcreteBuilder" classes are responsible for building the different parts of the object. This means that the object's construction logic is isolated from the classes that actually represent the final object.
- **Focus on Construction:**Each ConcreteBuilder has the sole responsibility of constructing a specific representation of the object. For example, if you have an objectDocumentthat can be built in PDF or HTML format, you would have aPdfDocumentBuilderand oneHtmlDocumentBuilder, each with the responsibility of building the specific representation. This adheres to the SRP, as each builder has a unique reason to change (changes in the way a specific format is built).

OCP (Open/Closed Principle) and Builder:

- **Extent of Construction:**The Builder pattern makes it easy to add new ways to construct an object without changing the client code that uses the builder. If you need to construct theDocumentin a new format (e.g. TXT), you can simply create a newTxtDocumentBuilderwithout modifying Director or the client code that orchestrates the build. The system is open to extension (new build methods) without needing to modify existing code.
- **Closed for Modification (Indirectly):**Client code that uses the Director (if any) or interacts with the Builder interface generally doesn't need to be modified when new ConcreteBuilders are added. It works with an abstraction (the Builder interface or the Director) that remains stable. Changes to how specific objects are constructed are isolated to the ConcreteBuilder classes.

Other SOLID Principles:

- **LSP (Liskov Substitution Principle):**Typically, the different representations constructed by ConcreteBuilders should be substitutable subtypes of the final object type (if a common interface exists). However, the Builder's primary focus is not inheritance or substitution polymorphism, but rather the separation of construction.
- **ISP (Interface Segregation Principle):**The Builder pattern can lead to more specific builder interfaces for different types of complex objects, which can indirectly

align with the ISP by preventing clients from relying on methods they don't use.
- **DIP (Dependency Inversion Principle):**By relying on Builder interfaces rather than concrete implementations in client code or the Director, the Builder pattern can promote DIP, making code more decoupled and testable.

In summary:

The Builder pattern correlates strongly with the**SRP**by separating the responsibility for construction from the representation of the object. It also aligns with the**OCP**by allowing the extension of how objects are constructed by adding new ConcreteBuilders without modifying the core client code. Although the correlation with other SOLID principles is less direct, proper use of the Builder can contribute to a more robust design that adheres to the general principles of good object-oriented design.

Builder in PHP (Similar to the Factory and Abstract Factory examples)

The standard**Builder**is also a creational pattern, like Factory Method and Abstract Factory, but its focus is**separate the construction of a complex object from its representation**. Instead of creating objects directly or delegating the creation of different types to subclasses (Factory Method) or families of objects (Abstract Factory), the Builder uses a separate object (the "builder") to construct the object step by step.

Purpose:

- Build complex objects with many optional parameters or settings.
- Provide a more readable and fluid way to create complex objects.
- Allow different representations of the object being built using the same build process.

Structure:

- **Product:**Represents the complex object being constructed.
- **Builder (Interface):**Defines an abstract interface for all possible ways of constructing the parts of the Product.
- **ConcreteBuilder(s):**They implement the Builder interface to build and assemble specific parts of the Product. Each ConcreteBuilder creates a different representation of the Product.
- **Director (Director) - Optional:**Constructs the object using the Builder interface. It defines the order of the construction steps. The client can use the Director or interact directly with ConcreteBuilder.

PHP Example (Constructing an "Email" object):

Let's imagine that we need to build an objectEmailwhich can have multiple optional attributes such as subject, body, CC recipients, blind carbon copy recipients, and attachments. Using a constructor with too many optional parameters can become confusing. The Builder offers a more elegant solution.

```
<?php


// Product

class Email {

private string $recipient;

private ?string $assunto = null;

    private ?string $corpo = null;

private array $copyTo = [];

    private array $copiaOcultaPara = [];

private array $annexos = [];


    public function __construct(string $destinatario) {

$this->recipient = $recipient;

    }


    public function setAssunto(string $assunto): self {

$this->subject = $subject;
```

```
        return $this; // Permite method chaining
    }

    public function setCorpo(string $corpo): self {
$this->body = $body;
        return $this;
    }

    public function adicionarCopiaPara(string $email): self {
$this->copyTo[] = $email;
        return $this;
    }

    public function adicionarCopiaOcultaPara(string $email): self {
        $this->copiaOcultaPara[] = $email;
        return $this;
    }

public function attachFile(string $filepath): self {
$this->attachments[] = $filepath;
        return $this;
```

```
    }

    public function enviar(): void {
echo "Sending email to: " . $this->recipient . "\n";
if ($this->subject) echo "Subject: " . $this->subject . "\n";
        if ($this->corpo) echo "Corpo: " . $this->corpo . "\n";
        if (!empty($this->copiaPara)) echo "CC: " . implode(', ', $this->copiaPara) . "\n";
        if (!empty($this->copiaOcultaPara)) echo "BCC: " . implode(', ', $this->copiaOcultaPara) . "\n";
        if (!empty($this->anexos)) echo "Anexos: " . implode(', ', $this->anexos) . "\n";
echo "Email sent!\n\n";
    }
}

// Builder (Implicit interface through the Email class with construction methods)
// In this simple example, the Email class itself acts as a "ConcreteBuilder"
// due to the use of method chaining.

// Client using the Builder directly
```

```
$email1 = (new Email('destinatario1@example.com'))
->setSubject('Project Meeting')
->setCorpo('Hello, I would like to schedule a meeting...')
->addCopyTo('cc1@example.com')
->addCopyTo('cc2@example.com');
$email1->send();

$email2 = (new Email('destinatario2@example.com'))
->setCorpo('Just a quick message.')
->attachFile('/path/to/document.pdf');
$email2->send();

// Example with a Director (optional)
class EmailDirector {
   public function construirEmailDeBoasVindas(string $destinatario, string $nomeUsuario): Email {
       return (new Email($recipient))
->setSubject("Welcome, " . $username . "!")
->setCorpo("Hello " . $username . ",\n\nWelcome to our platform!");
   }
}
```

```
$director = new EmailDirector();

$emailWelcome =
$director->buildWelcomeEmail('new_user@example.com', 'New User');

$emailWelcome->send();


?>
```

Analogy with Factory and Abstract Factory:

- **Factory Method:**Instead of having a classEmailFactorywith methods such ascreateSimpleEmail(), createEmailWithAttachment(), the Builder provides a more flexible way to build different "types" of emails (with different combinations of attributes) using the same "factory" (the classEmailwith their construction methods).

- **Abstract Factory:**If we had different email formats (e.g. plain text and HTML), we could have different ConcreteBuilders (PlainTextEmailBuilder, HTMLEmailBuilder) that would implement an interfaceEmailBuilder. Uma Abstract Factory (EmailFactory) could then provide instances of these Builders, allowing you to create families of email components (text or HTML body, attachments of different types, etc.) in a consistent way.

Advantages of Builder (in this context):

- **Readability:**Creating complex objects becomes more readable and intuitive (especially with method chaining).
- **Flexibility:**Allows you to build different representations of the object with different combinations of attributes.

- **Hides Complexity:**The client code does not need to worry about the construction order or initializing many parameters in the constructor.

While the Builder implementation may seem different from factory patterns, the underlying goal of abstracting the object creation process and making it more flexible and manageable is similar. The Builder excels when construction involves multiple steps and different possible configurations for the same type of object.

Observem as semelhanças dos códigos, aqui está visível;

Análise Comparativa: SRP e Factory Pattern

Embora um código trate de **formatação** e o outro de **criação**, a lógica de organização em termos de **SOLID** é a mesma. Observar a semelhança entre eles nos ajuda a entender o princípio de **delegação** e **indireção** que é fundamental para a arquitetura de software flexível.

Código 1: Refatoração para SRP (Separação de Responsabilidade)

O objetivo é separar a lógica de negócio (Hero) da lógica de representação (HeroFormatter).

PHP

```
<?php

// ... código anterior da classe Hero com getName(), getHealth(), etc. ...


class HeroFormatter{

    // A responsabilidade de formatar a saída (HTML) é isolada aqui.

    public static function toHtml(Hero $pHero)

    {

        $str = "<p>";

        $str .= "<span class='refactored'>Refactored</span><br/>";

        $str .= "<span class='name'>Name</span>:
{$pHero->getName()}<br/>";

        $str .= "<span class='health'>Health</span>:
{$pHero->getHealth()}<br/>";

        $str .= "<span class='power'>Power</span>:
{$pHero->getPower()}<br/>";
```

```
        $str .= "</p>";
        return $str;
    }
}

// Cliente utiliza o Formatter, injetando o objeto Hero no método.
$hulk = new Hero(1, 'Hulk', 100, 200);
print HeroFormatter::toHtml($hulk);
```

Código 2: Factory Pattern (Encapsulamento da Criação)

O objetivo é separar a lógica de negócio (Automobile) da lógica de criação (AutomobileFactory).

PHP

```
<?php
class Automobile{
    private $vehicleMake;
    private $vehicleModel;

    // A responsabilidade é APENAS representar o objeto.
    public function __construct($make, $model)
    {
        $this->vehicleMake = $make;
        $this->vehicleModel = $model;
    }
```

```
    // ...
}

class AutomobileFactory{
    // A responsabilidade de instanciar (`new`) é isolada aqui.
    public static function create($make, $model)
    {
        // Encapsula a lógica exata de construção
        return new Automobile($make, $model);
    }
}

// Cliente utiliza a Factory, delegando a responsabilidade de criação.
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');
print_r($veyron->getMakeAndModel());
```

Semelhanças Semânticas e a Relação com SOLID

A semelhança crucial entre os dois exemplos reside na técnica de **Indireção** e **Delegação**, o coração do SRP e do OCP:

1. Relação com o SRP (Princípio da Responsabilidade Única)

Em ambos os casos, a classe principal que contém os dados (o **Produto**) fica livre de uma preocupação secundária:

Classe Principal	**Preocupação Removida**	**Classe/Padrão que Assumiu**
Hero	Como devo me formatar?	HeroFormatter (SRP)
Automobile	Como devo ser criado?	AutomobileFactory (Factory)

A semântica é idêntica: isolar uma responsabilidade volátil em uma classe dedicada, garantindo que Hero e Automobile tenham agora **uma única razão para mudar** (seus atributos internos).

2. Relação com o OCP (Princípio Aberto/Fechado)

Ambos os padrões estabelecem um ponto de entrada estável (a API) para tarefas que podem mudar.

- **No Exemplo do SRP:** Se decidirmos que queremos formatar o Hero como JSON, criamos um HeroJsonFormatter (aberto para extensão). O código que chama HeroFormatter::toHtml() pode ser mantido, apenas trocando o objeto do formatador (fechado para modificação).

- **No Exemplo da Factory:** Se adicionarmos uma nova classe de produto, como Truck, o código cliente que chama AutomobileFactory::create(...) idealmente não precisa mudar, pois a Factory lida com a lógica do new Truck(...) internamente (aberto para extensão).

A **delegação do controle** para classes auxiliares é a ferramenta semântica que permite ao seu código cumprir o SRP e o OCP simultaneamente.

A Escalada da Abstração: Do Simple Factory ao Abstract Factory

O que chamamos de **Simple Factory** (como o seu exemplo da AutomobileFactory) é o primeiro passo para o SRP. Ele encapsula o new em um método estático.

O **Factory Method** e, especialmente, o **Abstract Factory** levam essa ideia ao extremo, garantindo que o seu código esteja totalmente aberto à extensão, em total conformidade com o **OCP** (Princípio Aberto/Fechado) e o **DIP** (Princípio da Inversão de Dependência).

A Semelhança Central

A semelhança central entre todos os padrões de Fábrica (Simple, Factory Method, Abstract) e a refatoração SRP (HeroFormatter) é a **delegação da responsabilidade volátil**:

Padrão	**O que é Delegado**	**Princípio Chave**
Simple Factory	O *conhecimento* de qual new chamar.	**SRP** (Isolar a criação)
Abstract Factory	O *tipo* de família de objetos a ser criada.	**OCP** & **DIP** (Abstrair a criação)

Demonstração do Abstract Factory

O **Abstract Factory Pattern** não cria apenas um objeto, mas sim uma **família de objetos relacionados** (ou dependentes).

Imagine um sistema que precisa renderizar interfaces para diferentes plataformas (Web, Desktop). Uma WebFactory criaria WebButton e WebForm, enquanto uma DesktopFactory criaria DesktopButton e DesktopForm. O código cliente só precisa da *interface* da fábrica, não da implementação concreta.

Código Abstract Factory (Exemplo Simplificado de Interface Gráfica)

PHP

```
<?php

// 1. Interfaces dos Produtos (Família A e B)

interface Button {

    public function render(): string;

}

interface Form {

    public function display(): string;

}


// 2. Família de Produtos "Desktop"

class DesktopButton implements Button {

    public function render(): string { return "Botão do Windows/Desktop"; }

}

class DesktopForm implements Form {
```

```
    public function display(): string { return "Formulário de Aplicação de Janela"; }
}

// 3. Interface da Fábrica Abstrata (Define o Contrato)
interface UIFactory {
    public function createButton(): Button;
    public function createForm(): Form;
}

// 4. Fábrica Concreta "Desktop"
class DesktopUIFactory implements UIFactory {
    public function createButton(): Button {
        return new DesktopButton(); // ONDE O 'NEW' ESTÁ ENCAPSULADO
    }
    public function createForm(): Form {
        return new DesktopForm(); // ONDE O 'NEW' ESTÁ ENCAPSULADO
    }
}

// 5. Cliente (Totalmente Desacoplado da Implementação)
class Application {
```

```
    // O Cliente só conhece a Interface UIFactory
    public function renderUI(UIFactory $factory) {
        $button = $factory->createButton();
        $form = $factory->createForm();

        echo $button->render() . "\n";
        echo $form->display() . "\n";
    }
}

// Uso: O sistema pode mudar a família de produtos com uma única linha!
echo "--- Renderizando Desktop ---\n";
$app = new Application();
$app->renderUI(new DesktopUIFactory());
// Se adicionarmos a 'WebUIFactory', só precisamos mudar a linha acima.
```

Relação Direta com OCP e DIP

No exemplo acima, o **OCP é totalmente garantido**:

1. **Aberto para Extensão:** Para adicionar o tema "Web", basta criar WebButton, WebForm e WebUIFactory.
2. **Fechado para Modificação:** A classe Application (o cliente) não precisa ser modificada. Ela sempre recebe e interage com a interface UIFactory, **sem nunca conhecer as classes concretas** (DesktopUIFactory, DesktopButton, etc.).

Isso cumpre o **DIP (Princípio da Inversão de Dependência)**, pois o módulo de alto nível (Application) não depende de módulos de baixo nível (DesktopUIFactory), mas sim de **abstrações** (UIFactory, Button, Form). É a mais pura forma de flexibilidade SOLID.

THE WORK CONTINUES. GEMINI WAS USED. ONLY EBOOK BUT PUT IN ORDER AND AFTER MY YOUTUBA EXAMPLE CODES AND
A DOCKER CLOUD. GEMINI GENERATED BUT THE RELATIONSHIP WITH SOLID I KNOW