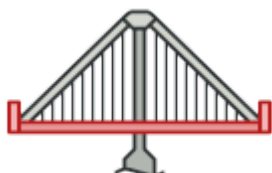




[🏠](#) / [Patrones de diseño](#) / [Bridge](#) / [PHP](#)



# Bridge en PHP

**Bridge** es un patrón de diseño estructural que divide la lógica de negocio o una clase muy grande en jerarquías de clases separadas que se pueden desarrollar independientemente.

Una de estas jerarquías (a menudo denominada Abstracción) obtendrá una referencia a un objeto de la segunda jerarquía (Implementación). La abstracción podrá delegar algunas (en ocasiones, la mayoría) de sus llamadas al objeto de las implementaciones. Como todas las implementaciones tendrán una interfaz común, serán intercambiables dentro de la abstracción.

 Aprende más sobre el patrón Bridge →

## Navegación

 [Intro](#)

 [Ejemplo conceptual](#)

 [index](#)

 [Output](#)

 [Ejemplo del mundo real](#)

 [index](#)

 [Output](#)

Complejidad: ★★ ★

Popularidad: ★ ☆ ☆

**Ejemplos de uso:** El patrón Bridge es de especial utilidad para soportar varios tipos de servidores de bases de datos o trabajar con varios proveedores de API de cierto tipo (por



**Identificación:** El patrón Bridge se puede reconocer por una distinción clara entre alguna entidad controladora y varias plataformas diferentes en las que se basa.

## Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Bridge** y se centra en las siguientes preguntas:

- ¿De qué clases se compone?
- ¿Qué papeles juegan esas clases?
- ¿De qué forma se relacionan los elementos del patrón?

Después de conocer la estructura del patrón, será más fácil comprender el siguiente ejemplo basado en un caso de uso real de PHP.

### index.php: Ejemplo conceptual

```
<?php
```

```
namespace RefactoringGuru\Bridge\Conceptual;
```

```
/**
```

```
 * The Abstraction defines the interface for the "control" part of the two class
 * hierarchies. It maintains a reference to an object of the Implementation
 * hierarchy and delegates all of the real work to this object.
 */
```

```
*/
```

```
class Abstraction
```

```
{
```

```
    /**
```

```
     * @var Implementation
```

```
     */
```

```
    protected $implementation;
```

```
    public function __construct(Implementation $implementation)
```

```
    {
```

```
        $this->implementation = $implementation;
```

```
    }
```

```
    public function operation(): string
```

```
    {
```

```
        return "Abstraction: Base operation with:\n" .
```

```
            $this->implementation->operationImplementation();
```



```
/**
 * You can extend the Abstraction without changing the Implementation classes.
 */
class ExtendedAbstraction extends Abstraction
{
    public function operation(): string
    {
        return "ExtendedAbstraction: Extended operation with:\n" .
            $this->implementation->operationImplementation();
    }
}

/**
 * The Implementation defines the interface for all implementation classes. It
 * doesn't have to match the Abstraction's interface. In fact, the two
 * interfaces can be entirely different. Typically the Implementation interface
 * provides only primitive operations, while the Abstraction defines higher-
 * level operations based on those primitives.
 */
interface Implementation
{
    public function operationImplementation(): string;
}

/**
 * Each Concrete Implementation corresponds to a specific platform and
 * implements the Implementation interface using that platform's API.
 */
class ConcreteImplementationA implements Implementation
{
    public function operationImplementation(): string
    {
        return "ConcreteImplementationA: Here's the result on the platform A.\n";
    }
}

class ConcreteImplementationB implements Implementation
{
    public function operationImplementation(): string
    {
        return "ConcreteImplementationB: Here's the result on the platform B.\n";
    }
}

/**
 * Except for the initialization phase, where an Abstraction object gets linked
 * with a specific Implementation object, the client code should only depend on
 * the Abstraction class. This way the client code can support any abstraction-
 * implementation combination.
 */
```



```
{  
    // ...  
  
    echo $abstraction->operation();  
  
    // ...  
}  
  
/**  
 * The client code should be able to work with any pre-configured abstraction-  
 * implementation combination.  
 */  
$implementation = new ConcreteImplementationA();  
$abstraction = new Abstraction($implementation);  
clientCode($abstraction);  
  
echo "\n";  
  
$implementation = new ConcreteImplementationB();  
$abstraction = new ExtendedAbstraction($implementation);  
clientCode($abstraction);
```

## Output.txt: Resultado de la ejecución

```
Abstraction: Base operation with:  
ConcreteImplementationA: Here's the result on the platform A.  
  
ExtendedAbstraction: Extended operation with:  
ConcreteImplementationB: Here's the result on the platform B.
```

## Ejemplo del mundo real

En este ejemplo, la jerarquía `Página` actúa como **Abstracción**, y la jerarquía `Procesador` actúa como **Implementación**. Los objetos de la clase `Página` pueden ensamblar páginas web de un tipo particular utilizando elementos básicos proporcionados por un objeto `Procesador` adjunto a esa página. Al estar ambas jerarquías de clases separadas, puedes añadir una nueva clase `Procesador` sin cambiar ninguna de las clases `Página` y viceversa.

## index.php: Ejemplo del mundo real



```
namespace RefactoringGuru\Bridge\RealWorld;

/**
 * The Abstraction.
 */
abstract class Page
{
    /**
     * @var Renderer
     */
    protected $renderer;

    /**
     * The Abstraction is usually initialized with one of the Implementation
     * objects.
     */
    public function __construct(Renderer $renderer)
    {
        $this->renderer = $renderer;
    }

    /**
     * The Bridge pattern allows replacing the attached Implementation object
     * dynamically.
     */
    public function changeRenderer(Renderer $renderer): void
    {
        $this->renderer = $renderer;
    }

    /**
     * The "view" behavior stays abstract since it can only be provided by
     * Concrete Abstraction classes.
     */
    abstract public function view(): string;
}

/**
 * This Concrete Abstraction represents a simple page.
 */
class SimplePage extends Page
{
    protected $title;
    protected $content;

    public function __construct(Renderer $renderer, string $title, string $content)
    {
        parent::__construct($renderer);
        $this->title = $title;
    }
}
```



```
public function view(): string
{
    return $this->renderer->renderParts([
        $this->renderer->renderHeader(),
        $this->renderer->renderTitle($this->title),
        $this->renderer->renderTextBlock($this->content),
        $this->renderer->renderFooter()
    ]);
}

/**
 * This Concrete Abstraction represents a more complex page.
 */
class ProductPage extends Page
{
    protected $product;

    public function __construct(Renderer $renderer, Product $product)
    {
        parent::__construct($renderer);
        $this->product = $product;
    }

    public function view(): string
    {
        return $this->renderer->renderParts([
            $this->renderer->renderHeader(),
            $this->renderer->renderTitle($this->product->getTitle()),
            $this->renderer->renderTextBlock($this->product->getDescription()),
            $this->renderer->renderImage($this->product->getImage()),
            $this->renderer->renderLink("/cart/add/" . $this->product->getId(), "Add to c
            $this->renderer->renderFooter()
        ]);
    }
}

/**
 * A helper class for the ProductPage class.
 */
class Product
{
    private $id, $title, $description, $image, $price;

    public function __construct(
        string $id,
        string $title,
        string $description,
        string $image,
```



# REBAJA DE VERANO



```
$this->id = $id;
$this->title = $title;
$this->description = $description;
$this->image = $image;
$this->price = $price;
}

public function getId(): string { return $this->id; }

public function getTitle(): string { return $this->title; }

public function getDescription(): string { return $this->description; }

public function getImage(): string { return $this->image; }

public function getPrice(): float { return $this->price; }
}

/**
 * The Implementation declares a set of "real", "under-the-hood", "platform"
 * methods.
 *
 * In this case, the Implementation lists rendering methods that can be used to
 * compose any web page. Different Abstractions may use different methods of the
 * Implementation.
 */
interface Renderer
{
    public function renderTitle(string $title): string;

    public function renderTextBlock(string $text): string;

    public function renderImage(string $url): string;

    public function renderLink(string $url, string $title): string;

    public function renderHeader(): string;

    public function renderFooter(): string;

    public function renderParts(array $parts): string;
}

/**
 * This Concrete Implementation renders a web page as HTML.
 */
class HTMLRenderer implements Renderer
{
    public function renderTitle(string $title): string
```



```
}

public function renderTextBlock(string $text): string
{
    return "<div class='text'>$text</div>";
}

public function renderImage(string $url): string
{
    return "<img src='$url'>";
}

public function renderLink(string $url, string $title): string
{
    return "<a href='$url'>$title</a>";
}

public function renderHeader(): string
{
    return "<html><body>";
}

public function renderFooter(): string
{
    return "</body></html>";
}

public function renderParts(array $parts): string
{
    return implode("\n", $parts);
}
}

/**
 * This Concrete Implementation renders a web page as JSON strings.
 */
class JsonRenderer implements Renderer
{
    public function renderTitle(string $title): string
    {
        return '"title": "' . $title . '"';
    }

    public function renderTextBlock(string $text): string
    {
        return '"text": "' . $text . '"';
    }

    public function renderImage(string $url): string
    {

```





```
public function renderLink(string $url, string $title): string
{
    return '"link": {"href": "' . $url . "', "title": "' . $title . '"}';
}

public function renderHeader(): string
{
    return '';
}

public function renderFooter(): string
{
    return '';
}

public function renderParts(array $parts): string
{
    return "{\n" . implode(",\n", array_filter($parts)) . "\n}";
}

}

/**
 * The client code usually deals only with the Abstraction objects.
 */
function clientCode(Page $page)
{
    // ...

    echo $page->view();

    // ...
}

/**
 * The client code can be executed with any pre-configured combination of the
 * Abstraction+Implementation.
 */
$HTMLRenderer = new HTMLRenderer();
$JSONRenderer = new JsonRenderer();

$page = new SimplePage($HTMLRenderer, "Home", "Welcome to our website!");
echo "HTML view of a simple content page:\n";
clientCode($page);
echo "\n\n";

/**
 * The Abstraction can change the linked Implementation at runtime if needed.
 */
$page->changeRenderer($JSONRenderer);
```



```
echo "\n\n";

$product = new Product("123", "Star Wars, episode1",
    "A long time ago in a galaxy far, far away...",
    "/images/star-wars.jpeg", 39.95);

$page = new ProductPage($HTMLRenderer, $product);
echo "HTML view of a product page, same client code:\n";
clientCode($page);
echo "\n\n";

$page->changeRenderer($JSONRenderer);
echo "JSON view of a simple content page, with the same client code:\n";
clientCode($page);
```

## Output.txt: Resultado de la ejecución

HTML view of a simple content page:

```
<html><body>
<h1>Home</h1>
<div class='text'>Welcome to our website!</div>
</body></html>
```

JSON view of a simple content page, rendered with the same client code:

```
{
  "title": "Home",
  "text": "Welcome to our website!"
}
```

HTML view of a product page, same client code:

```
<html><body>
<h1>Star Wars, episode1</h1>
<div class='text'>A long time ago in a galaxy far, far away...</div>
<img src='/images/star-wars.jpeg'>
<a href='/cart/add/123'>Add to cart</a>
</body></html>
```

JSON view of a simple content page, with the same client code:

```
{
  "title": "Star Wars, episode1",
  "text": "A long time ago in a galaxy far, far away...",
  "img": "/images/star-wars.jpeg",
  "link": {"href": "/cart/add/123", "title": "Add to cart"}
}
```



# REBAJA DE VERANO



[LEER SIGUIENTE](#)

[Inicio](#)

[Refactorización](#)

[Patrones de diseño](#)

[Contenido Premium](#)

[Foro](#)


[Contáctanos](#)




© 2014-2023 Refactoring.Guru. Todos los derechos reservados

 Ilustraciones por Dmitry Zhart

#### **Ukrainian office:**


 FOP Olga Skobeleva

 Abolmasova 7  
Kyiv, Ukraine, 02002

 Email: [support@refactoring.guru](mailto:support@refactoring.guru)

#### **Spanish office:**

 Oleksandr Shvets

 Avda Pamplona 63, 4b  
Pamplona, Spain, 31010

 Email: [spain@refactoring.guru](mailto:spain@refactoring.guru)

[Términos y condiciones](#)

[Política de privacidad](#)

[Política de uso de contenido](#)

[About us](#)

