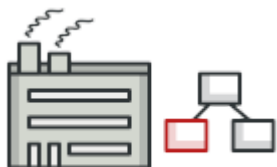




[🏠](#) / [Patrones de diseño](#) / [Factory Method](#) / [PHP](#)



Factory Method en PHP

Factory method es un patrón de diseño creacional que resuelve el problema de crear objetos de producto sin especificar sus clases concretas.

El patrón Factory Method define un método que debe utilizarse para crear objetos, en lugar de una llamada directa al constructor (operador `new`). Las subclases pueden sobrescribir este método para cambiar las clases de los objetos que se crearán.

Si no sabes la diferencia entre varios patrones y conceptos de la fábrica, lee nuestra [Comparación de fábricas](#).

 Aprende más sobre el patrón Factory Method →

Navegación

 [Intro](#)

 [Ejemplo conceptual](#)

 [index](#)

 [Output](#)

 [Ejemplo del mundo real](#)

 [index](#)

 [Output](#)

Complejidad: ★☆☆



Ejemplos de uso: El patrón Factory Method se utiliza mucho en el código PHP. Resulta muy útil cuando necesitas proporcionar un alto nivel de flexibilidad a tu código.

Identificación: Los métodos fábrica pueden ser reconocidos por métodos de creación, que crean objetos de clases concretas, pero los devuelven como objetos del tipo abstracto o interfaz.

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Factory Method** 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\FactoryMethod\Conceptual;
```

```
/**
```

```
 * The Creator class declares the factory method that is supposed to return an
 * object of a Product class. The Creator's subclasses usually provide the
 * implementation of this method.
```

```
*/
```

```
abstract class Creator
```

```
{
```

```
    /**
```

```
     * Note that the Creator may also provide some default implementation of the
     * factory method.
```

```
    */
```

```
    abstract public function factoryMethod(): Product;
```

```
    /**
```

```
     * Also note that, despite its name, the Creator's primary responsibility is
     * not creating products. Usually, it contains some core business logic that
     * relies on Product objects, returned by the factory method. Subclasses can
```



```
*/
public function someOperation(): string
{
    // Call the factory method to create a Product object.
    $product = $this->factoryMethod();
    // Now, use the product.
    $result = "Creator: The same creator's code has just worked with " .
        $product->operation();

    return $result;
}

/**
 * Concrete Creators override the factory method in order to change the
 * resulting product's type.
 */
class ConcreteCreator1 extends Creator
{
    /**
     * Note that the signature of the method still uses the abstract product
     * type, even though the concrete product is actually returned from the
     * method. This way the Creator can stay independent of concrete product
     * classes.
     */
    public function factoryMethod(): Product
    {
        return new ConcreteProduct1();
    }
}

class ConcreteCreator2 extends Creator
{
    public function factoryMethod(): Product
    {
        return new ConcreteProduct2();
    }
}

/**
 * The Product interface declares the operations that all concrete products must
 * implement.
 */
interface Product
{
    public function operation(): string;
}

/**
 * Concrete Products provide various implementations of the Product interface.
```



```
{
    public function operation(): string
    {
        return "{Result of the ConcreteProduct1}";
    }
}

class ConcreteProduct2 implements Product
{
    public function operation(): string
    {
        return "{Result of the ConcreteProduct2}";
    }
}

/**
 * The client code works with an instance of a concrete creator, albeit through
 * its base interface. As long as the client keeps working with the creator via
 * the base interface, you can pass it any creator's subclass.
 */
function clientCode(Creator $creator)
{
    // ...
    echo "Client: I'm not aware of the creator's class, but it still works.\n"
        . $creator->someOperation();
    // ...
}

/**
 * The Application picks a creator's type depending on the configuration or
 * environment.
 */
echo "App: Launched with the ConcreteCreator1.\n";
clientCode(new ConcreteCreator1());
echo "\n\n";

echo "App: Launched with the ConcreteCreator2.\n";
clientCode(new ConcreteCreator2());
```

Output.txt: Resultado de la ejecución

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

App: Launched with the ConcreteCreator2.
```



Ejemplo del mundo real

En este ejemplo, el patrón **Factory Method** proporciona una interfaz para crear conectores en redes sociales, que pueden utilizarse para iniciar sesión en la red, crear publicaciones y, potencialmente, realizar otras actividades; y todo ello sin acoplar el código cliente a clases específicas de la red social particular.

index.php: Ejemplo del mundo real

```
<?php
```

```
namespace RefactoringGuru\FactoryMethod\RealWorld;
```

```
/**
```

```
 * The Creator declares a factory method that can be used as a substitution for  
 * the direct constructor calls of products, for instance:
```

```
 *
```

```
 * - Before: $p = new FacebookConnector();
```

```
 * - After: $p = $this->getSocialNetwork();
```

```
 *
```

```
 * This allows changing the type of the product being created by
```

```
 * SocialNetworkPoster's subclasses.
```

```
 */
```

```
abstract class SocialNetworkPoster
```

```
{
```

```
    /**
```

```
     * The actual factory method. Note that it returns the abstract connector.
```

```
     * This lets subclasses return any concrete connectors without breaking the
```

```
     * superclass' contract.
```

```
     */
```

```
    abstract public function getSocialNetwork(): SocialNetworkConnector;
```

```
    /**
```

```
     * When the factory method is used inside the Creator's business logic, the
```

```
     * subclasses may alter the logic indirectly by returning different types of
```

```
     * the connector from the factory method.
```

```
     */
```

```
    public function post($content): void
```

```
    {
```

```
        // Call the factory method to create a Product object...
```

```
        $network = $this->getSocialNetwork();
```

```
        // ...then use it as you will.
```

```
        $network->login();
```

```
        $network->createPost($content);
```



```
}

/**
 * This Concrete Creator supports Facebook. Remember that this class also
 * inherits the 'post' method from the parent class. Concrete Creators are the
 * classes that the Client actually uses.
 */
class FacebookPoster extends SocialNetworkPoster
{
    private $login, $password;

    public function __construct(string $login, string $password)
    {
        $this->login = $login;
        $this->password = $password;
    }

    public function getSocialNetwork(): SocialNetworkConnector
    {
        return new FacebookConnector($this->login, $this->password);
    }
}

/**
 * This Concrete Creator supports LinkedIn.
 */
class LinkedInPoster extends SocialNetworkPoster
{
    private $email, $password;

    public function __construct(string $email, string $password)
    {
        $this->email = $email;
        $this->password = $password;
    }

    public function getSocialNetwork(): SocialNetworkConnector
    {
        return new LinkedInConnector($this->email, $this->password);
    }
}

/**
 * The Product interface declares behaviors of various types of products.
 */
interface SocialNetworkConnector
{
    public function logIn(): void;

    public function logOut(): void;
}
```



```
}

/**
 * This Concrete Product implements the Facebook API.
 */
class FacebookConnector implements SocialNetworkConnector
{
    private $login, $password;

    public function __construct(string $login, string $password)
    {
        $this->login = $login;
        $this->password = $password;
    }

    public function logIn(): void
    {
        echo "Send HTTP API request to log in user $this->login with " .
            "password $this->password\n";
    }

    public function logOut(): void
    {
        echo "Send HTTP API request to log out user $this->login\n";
    }

    public function createPost($content): void
    {
        echo "Send HTTP API requests to create a post in Facebook timeline.\n";
    }
}

/**
 * This Concrete Product implements the LinkedIn API.
 */
class LinkedInConnector implements SocialNetworkConnector
{
    private $email, $password;

    public function __construct(string $email, string $password)
    {
        $this->email = $email;
        $this->password = $password;
    }

    public function logIn(): void
    {
        echo "Send HTTP API request to log in user $this->email with " .
            "password $this->password\n";
    }
}
```



```
{
    echo "Send HTTP API request to log out user $this->email\n";
}

public function createPost($content): void
{
    echo "Send HTTP API requests to create a post in LinkedIn timeline.\n";
}
}

/**
 * The client code can work with any subclass of SocialNetworkPoster since it
 * doesn't depend on concrete classes.
 */
function clientCode(SocialNetworkPoster $creator)
{
    // ...
    $creator->post("Hello world!");
    $creator->post("I had a large hamburger this morning!");
    // ...
}

/**
 * During the initialization phase, the app can decide which social network it
 * wants to work with, create an object of the proper subclass, and pass it to
 * the client code.
 */
echo "Testing ConcreteCreator1:\n";
clientCode(new FacebookPoster("john_smith", "*****"));
echo "\n\n";

echo "Testing ConcreteCreator2:\n";
clientCode(new LinkedInPoster("john_smith@example.com", "*****"));
```

Output.txt: Resultado de la ejecución

Testing ConcreteCreator1:

```
Send HTTP API request to log in user john_smith with password *****
Send HTTP API requests to create a post in Facebook timeline.
Send HTTP API request to log out user john_smith
Send HTTP API request to log in user john_smith with password *****
Send HTTP API requests to create a post in Facebook timeline.
Send HTTP API request to log out user john_smith
```

Testing ConcreteCreator2:

```
Send HTTP API request to log in user john_smith@example.com with password *****
```




REBAJA DE VERANO



Send HTTP API request to log in user john_smith@example.com with password *****
Send HTTP API requests to create a post in LinkedIn timeline.
Send HTTP API request to log out user john_smith@example.com

Ejemplo conceptual

Ejemplo del mundo real

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

Spanish office:

 Oleksandr Shvets

 Avda Pamplona 63, 4b

Pamplona, Spain, 31010

 Email: spain@refactoring.guru

Términos y condiciones

Política de privacidad

Política de uso de contenido

About us

