



REBAJA DE VERANO



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



Builder en PHP

Builder es un patrón de diseño creacional que permite construir objetos complejos paso a paso.

Al contrario que otros patrones creacionales, Builder no necesita que los productos tengan una interfaz común. Esto hace posible crear distintos productos utilizando el mismo proceso de construcción.

 [Aprende más sobre el patrón Builder →](#)

Navegación

 [Intro](#)

 [Ejemplo conceptual](#)

 [index](#)

 [Output](#)

 [Ejemplo del mundo real](#)

 [index](#)

 [Output](#)

Complejidad: ★★☆☆

Popularidad: ★★★

Ejemplos de uso: El patrón Builder es muy conocido en el mundo PHP. Resulta especialmente útil cuando debes crear un objeto con muchas opciones posibles de configuración.



de creación y varios métodos para configurar el objeto resultante. A menudo, los métodos del Builder soportan el encadenamiento (por ejemplo, `algúnBuilder->establecerValorA(1)->establecerValorB(2)->crear()`).

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Builder** 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\Builder\Conceptual;
```

```
/**
```

```
 * The Builder interface specifies methods for creating the different parts of  
 * the Product objects.
```

```
*/
```

```
interface Builder
```

```
{
```

```
    public function producePartA(): void;
```

```
    public function producePartB(): void;
```

```
    public function producePartC(): void;
```

```
}
```

```
/**
```

```
 * The Concrete Builder classes follow the Builder interface and provide  
 * specific implementations of the building steps. Your program may have several  
 * variations of Builders, implemented differently.
```

```
*/
```

```
class ConcreteBuilder1 implements Builder
```

```
{
```

```
    private $product;
```



```
* A fresh builder instance should contain a blank product object, which is
* used in further assembly.
```

```
*/
```

```
public function __construct()
{
    $this->reset();
}
```

```
public function reset(): void
{
    $this->product = new Product1();
}
```

```
/**
```

```
 * All production steps work with the same product instance.
```

```
*/
```

```
public function producePartA(): void
{
    $this->product->parts[] = "PartA1";
}
```

```
public function producePartB(): void
{
    $this->product->parts[] = "PartB1";
}
```

```
public function producePartC(): void
{
    $this->product->parts[] = "PartC1";
}
```

```
/**
```

```
 * Concrete Builders are supposed to provide their own methods for
 * retrieving results. That's because various types of builders may create
 * entirely different products that don't follow the same interface.
 * Therefore, such methods cannot be declared in the base Builder interface
 * (at least in a statically typed programming language). Note that PHP is a
 * dynamically typed language and this method CAN be in the base interface.
 * However, we won't declare it there for the sake of clarity.
```

```
 *
```

```
 * Usually, after returning the end result to the client, a builder instance
 * is expected to be ready to start producing another product. That's why
 * it's a usual practice to call the reset method at the end of the
 * `getProduct` method body. However, this behavior is not mandatory, and
 * you can make your builders wait for an explicit reset call from the
 * client code before disposing of the previous result.
```

```
*/
```

```
public function getProduct(): Product1
{
    $result = $this->product;
```



```
        return $result;
    }
}

/**
 * It makes sense to use the Builder pattern only when your products are quite
 * complex and require extensive configuration.
 *
 * Unlike in other creational patterns, different concrete builders can produce
 * unrelated products. In other words, results of various builders may not
 * always follow the same interface.
 */
class Product1
{
    public $parts = [];

    public function listParts(): void
    {
        echo "Product parts: " . implode(', ', $this->parts) . "\n\n";
    }
}

/**
 * The Director is only responsible for executing the building steps in a
 * particular sequence. It is helpful when producing products according to a
 * specific order or configuration. Strictly speaking, the Director class is
 * optional, since the client can control builders directly.
 */
class Director
{
    /**
     * @var Builder
     */
    private $builder;

    /**
     * The Director works with any builder instance that the client code passes
     * to it. This way, the client code may alter the final type of the newly
     * assembled product.
     */
    public function setBuilder(Builder $builder): void
    {
        $this->builder = $builder;
    }

    /**
     * The Director can construct several product variations using the same
     * building steps.
     */
    public function buildMinimalViableProduct(): void
```



```
}

public function buildFullFeaturedProduct(): void
{
    $this->builder->producePartA();
    $this->builder->producePartB();
    $this->builder->producePartC();
}

}

/**
 * The client code creates a builder object, passes it to the director and then
 * initiates the construction process. The end result is retrieved from the
 * builder object.
 */
function clientCode(Director $director)
{
    $builder = new ConcreteBuilder1();
    $director->setBuilder($builder);

    echo "Standard basic product:\n";
    $director->buildMinimalViableProduct();
    $builder->getProduct()->listParts();

    echo "Standard full featured product:\n";
    $director->buildFullFeaturedProduct();
    $builder->getProduct()->listParts();

    // Remember, the Builder pattern can be used without a Director class.
    echo "Custom product:\n";
    $builder->producePartA();
    $builder->producePartC();
    $builder->getProduct()->listParts();
}

$director = new Director();
clientCode($director);
```

Output.txt: Resultado de la ejecución

Standard basic product:
Product parts: PartA1

Standard full featured product:
Product parts: PartA1, PartB1, PartC1



Ejemplo del mundo real

Una de las mejores aplicaciones del patrón **Builder** es un constructor de una consulta SQL. La interfaz del constructor define los pasos comunes necesarios para construir una consulta SQL genérica. Por otro lado, los constructores concretos, que se corresponden con distintos dialectos SQL, implementan estos pasos devolviendo partes de consultas SQL que se pueden ejecutar en un motor de base de datos particular.

index.php: Ejemplo del mundo real

```
<?php
```

```
namespace RefactoringGuru\Builder\RealWorld;
```

```
/**
```

```
 * The Builder interface declares a set of methods to assemble an SQL query.
```

```
 *
```

```
 * All of the construction steps are returning the current builder object to
```

```
 * allow chaining: $builder->select(...)->where(...)
```

```
 */
```

```
interface SQLQueryBuilder
```

```
{
```

```
    public function select(string $table, array $fields): SQLQueryBuilder;
```

```
    public function where(string $field, string $value, string $operator = '='): SQLQuery
```

```
    public function limit(int $start, int $offset): SQLQueryBuilder;
```

```
    // +100 other SQL syntax methods...
```

```
    public function getSQL(): string;
```

```
}
```

```
/**
```

```
 * Each Concrete Builder corresponds to a specific SQL dialect and may implement
```

```
 * the builder steps a little bit differently from the others.
```

```
 *
```

```
 * This Concrete Builder can build SQL queries compatible with MySQL.
```

```
 */
```

```
class MySQLQueryBuilder implements SQLQueryBuilder
```

```
{
```

```
    protected $query;
```

```
    protected function reset(): void
```



```
}

/**
 * Build a base SELECT query.
 */
public function select(string $table, array $fields): SQLQueryBuilder
{
    $this->reset();
    $this->query->base = "SELECT " . implode(", ", $fields) . " FROM " . $table;
    $this->query->type = 'select';

    return $this;
}

/**
 * Add a WHERE condition.
 */
public function where(string $field, string $value, string $operator = '='): SQLQuery
{
    if (!in_array($this->query->type, ['select', 'update', 'delete'])) {
        throw new \Exception("WHERE can only be added to SELECT, UPDATE OR DELETE");
    }
    $this->query->where[] = "$field $operator '$value'";

    return $this;
}

/**
 * Add a LIMIT constraint.
 */
public function limit(int $start, int $offset): SQLQueryBuilder
{
    if (!in_array($this->query->type, ['select'])) {
        throw new \Exception("LIMIT can only be added to SELECT");
    }
    $this->query->limit = " LIMIT " . $start . ", " . $offset;

    return $this;
}

/**
 * Get the final query string.
 */
public function getSQL(): string
{
    $query = $this->query;
    $sql = $query->base;
    if (!empty($query->where)) {
        $sql .= " WHERE " . implode(' AND ', $query->where);
    }
}
```



```
}
$sql .= ";";
return $sql;
}

/**
 * This Concrete Builder is compatible with PostgreSQL. While Postgres is very
 * similar to Mysql, it still has several differences. To reuse the common code,
 * we extend it from the MySQL builder, while overriding some of the building
 * steps.
 */
class PostgresQueryBuilder extends MysqlQueryBuilder
{
    /**
     * Among other things, PostgreSQL has slightly different LIMIT syntax.
     */
    public function limit(int $start, int $offset): SQLQueryBuilder
    {
        parent::limit($start, $offset);

        $this->query->limit = " LIMIT " . $start . " OFFSET " . $offset;

        return $this;
    }

    // + tons of other overrides...
}

/**
 * Note that the client code uses the builder object directly. A designated
 * Director class is not necessary in this case, because the client code needs
 * different queries almost every time, so the sequence of the construction
 * steps cannot be easily reused.
 *
 * Since all our query builders create products of the same type (which is a
 * string), we can interact with all builders using their common interface.
 * Later, if we implement a new Builder class, we will be able to pass its
 * instance to the existing client code without breaking it thanks to the
 * SQLQueryBuilder interface.
 */
function clientCode(SQLQueryBuilder $queryBuilder)
{
    // ...

    $query = $queryBuilder
        ->select("users", ["name", "email", "password"])
        ->where("age", 18, ">")
        ->where("age", 30, "<")
}
```




```
echo $query;

// ...
}

/**
 * The application selects the proper query builder type depending on a current
 * configuration or the environment settings.
 */
// if ($_ENV['database_type'] == 'postgres') {
//     $builder = new PostgresQueryBuilder(); } else {
//     $builder = new MysqlQueryBuilder(); }
//
// clientCode($builder);

echo "Testing MySQL query builder:\n";
clientCode(new MysqlQueryBuilder());

echo "\n\n";

echo "Testing PostgreSQL query builder:\n";
clientCode(new PostgresQueryBuilder());
```

Output.txt: Resultado de la ejecución

Testing MySQL query builder:

```
SELECT name, email, password FROM users WHERE age > '18' AND age < '30' LIMIT 10, 20;
```

Testing PostgreSQL query builder:

```
SELECT name, email, password FROM users WHERE age > '18' AND age < '30' LIMIT 10 OFFSET 2
```



¡ EED CICIENTE

Inicio



Refactorización





REBAJA DE VERANO




[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](#)

