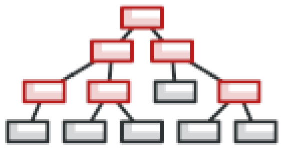
REBAJA DE VERANO



Composite en PHP

Composite es un patrón de diseño estructural que permite componer objetos en una estructura en forma de árbol y trabajar con ella como si fuera un objeto único.

El patrón Composite se convirtió en una solución muy popular para la mayoría de problemas que requieren la creación de una estructura de árbol. La gran característica del Composite es la capacidad para ejecutar métodos de forma recursiva por toda la estructura de árbol y recapitular los resultados.

Aprende más sobre el patrón Composite →

Navegación

Intro

Ejemplo conceptual
index
Output

Ejemplo del mundo real
index
Output

Complejidad: ★★☆

Popularidad: ★★☆

Ejemplos de uso: El patrón Composite se utiliza habitualmente al trabajar con árboles de objetos. El ejemplo más sencillo sería aplicar el patrón a elementos del árbol DOM, trabajando

Identificación: El Composite es fácil de reconocer por los métodos de comportamiento que toman una instancia del mismo tipo abstracto/interfaz y lo hacen una estructura de árbol.

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Composite** 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
<?php

namespace RefactoringGuru\Composite\Conceptual;

/**
 * The base Component class declares common operations for both simple and
 * complex objects of a composition.
 */
abstract class Component
{
    /**
     * @var Component|null
     */
    protected $parent;

    /**
     * Optionally, the base Component can declare an interface for setting and
     * accessing a parent of the component in a tree structure. It can also
     * provide some default implementation for these methods.
     */
    public function setParent(?Component $parent)
    {
        $this->parent = $parent;
    }
```

```
        return $this->parent;
    }

    /**
     * In some cases, it would be beneficial to define the child-management
     * operations right in the base Component class. This way, you won't need to
     * expose any concrete component classes to the client code, even during the
     * object tree assembly. The downside is that these methods will be empty
     * for the leaf-level components.
     */
    public function add(Component $component): void { }

    public function remove(Component $component): void { }

    /**
     * You can provide a method that lets the client code figure out whether a
     * component can bear children.
     */
    public function isComposite(): bool
    {
        return false;
    }

    /**
     * The base Component may implement some default behavior or leave it to
     * concrete classes (by declaring the method containing the behavior as
     * "abstract").
     */
    abstract public function operation(): string;
}

/**
 * The Leaf class represents the end objects of a composition. A leaf can't have
 * any children.
 *
 * Usually, it's the Leaf objects that do the actual work, whereas Composite
 * objects only delegate to their sub-components.
 */
class Leaf extends Component
{
    public function operation(): string
    {
        return "Leaf";
    }
}

/**
 * The Composite class represents the complex components that may have children.
 * Usually, the Composite objects delegate the actual work to their children and
 * then "sum-up" the result.
```

REBAJA DE VERANO

```php
{
    /**
     * @var \SplObjectStorage
     */
    protected $children;

    public function __construct()
    {
        $this->children = new \SplObjectStorage();
    }

    /**
     * A composite object can add or remove other components (both simple or
     * complex) to or from its child list.
     */
    public function add(Component $component): void
    {
        $this->children->attach($component);
        $component->setParent($this);
    }

    public function remove(Component $component): void
    {
        $this->children->detach($component);
        $component->setParent(null);
    }

    public function isComposite(): bool
    {
        return true;
    }

    /**
     * The Composite executes its primary logic in a particular way. It
     * traverses recursively through all its children, collecting and summing
     * their results. Since the composite's children pass these calls to their
     * children and so forth, the whole object tree is traversed as a result.
     */
    public function operation(): string
    {
        $results = [];
        foreach ($this->children as $child) {
            $results[] = $child->operation();
        }

        return "Branch(" . implode("+", $results) . ")";
    }
}

/**
```

```php
function clientCode(Component $component)
{
    // ...

    echo "RESULT: " . $component->operation();

    // ...
}

/**
 * This way the client code can support the simple leaf components...
 */
$simple = new Leaf();
echo "Client: I've got a simple component:\n";
clientCode($simple);
echo "\n\n";

/**
 * ...as well as the complex composites.
 */
$tree = new Composite();
$branch1 = new Composite();
$branch1->add(new Leaf());
$branch1->add(new Leaf());
$branch2 = new Composite();
$branch2->add(new Leaf());
$tree->add($branch1);
$tree->add($branch2);
echo "Client: Now I've got a composite tree:\n";
clientCode($tree);
echo "\n\n";

/**
 * Thanks to the fact that the child-management operations are declared in the
 * base Component class, the client code can work with any component, simple or
 * complex, without depending on their concrete classes.
 */
function clientCode2(Component $component1, Component $component2)
{
    // ...

    if ($component1->isComposite()) {
        $component1->add($component2);
    }
    echo "RESULT: " . $component1->operation();

    // ...
}
```

Output.txt: Resultado de la ejecución

```
Client: I get a simple component:
RESULT: Leaf

Client: Now I get a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree::
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```

Ejemplo del mundo real

El patrón **Composite** puede agilizar el trabajo con cualquier estructura recursiva con forma de árbol. El árbol DOM HTML es un ejemplo de dicha estructura. Por ejemplo, mientras varios elementos de entrada pueden actuar como hojas, los elementos complejos como formas y grupos de campo (fieldsets) juegan el papel de compuestos.

Con esto en mente, puedes utilizar el patrón Composite para aplicar varios comportamientos a todo el árbol HTML, del mismo modo que a sus elementos internos, sin acoplar tu código a clases concretas del árbol DOM. Ejemplos de estos comportamientos serían representar los elementos DOM, exportarlos a varios formatos, validar sus partes, etc.

Con el patrón Composite, no tienes que comprobar si se trata del tipo de elemento simple o complejo antes de ejecutar el comportamiento. Dependiendo del tipo de elemento, se ejecuta directamente o se pasa a todos los hijos del elemento.

index.php: Ejemplo del mundo real

```php
<?php

namespace RefactoringGuru\Composite\RealWorld;

/**
 * The base Component class declares an interface for all concrete components,
 * both simple and complex.
 *
 * In our example, we'll be focusing on the rendering behavior of DOM elements.
```

```
{
    /**
     * We can anticipate that all DOM elements require these 3 fields.
     */
    protected $name;
    protected $title;
    protected $data;

    public function __construct(string $name, string $title)
    {
        $this->name = $name;
        $this->title = $title;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setData($data): void
    {
        $this->data = $data;
    }

    public function getData(): array
    {
        return $this->data;
    }

    /**
     * Each concrete DOM element must provide its rendering implementation, but
     * we can safely assume that all of them are returning strings.
     */
    abstract public function render(): string;
}

/**
 * This is a Leaf component. Like all the Leaves, it can't have any children.
 */
class Input extends FormElement
{
    private $type;

    public function __construct(string $name, string $title, string $type)
    {
        parent::__construct($name, $title);
        $this->type = $type;
    }

    /**
```

```
     * lifting within the Composite pattern.
     */
    public function render(): string
    {
        return "<label for=\"{$this->name}\">{$this->title}</label>\n" .
            "<input name=\"{$this->name}\" type=\"{$this->type}\" value=\"{$this->data}\"
    }
}

/**
 * The base Composite class implements the infrastructure for managing child
 * objects, reused by all Concrete Composites.
 */
abstract class FieldComposite extends FormElement
{
    /**
     * @var FormElement[]
     */
    protected $fields = [];

    /**
     * The methods for adding/removing sub-objects.
     */
    public function add(FormElement $field): void
    {
        $name = $field->getName();
        $this->fields[$name] = $field;
    }

    public function remove(FormElement $component): void
    {
        $this->fields = array_filter($this->fields, function ($child) use ($component) {
            return $child != $component;
        });
    }

    /**
     * Whereas a Leaf's method just does the job, the Composite's method almost
     * always has to take its sub-objects into account.
     *
     * In this case, the composite can accept structured data.
     *
     * @param array $data
     */
    public function setData($data): void
    {
        foreach ($this->fields as $name => $field) {
            if (isset($data[$name])) {
                $field->setData($data[$name]);
            }
```

```php
    /**
     * The same logic applies to the getter. It returns the structured data of
     * the composite itself (if any) and all the children data.
     */
    public function getData(): array
    {
        $data = [];

        foreach ($this->fields as $name => $field) {
            $data[$name] = $field->getData();
        }

        return $data;
    }

    /**
     * The base implementation of the Composite's rendering simply combines
     * results of all children. Concrete Composites will be able to reuse this
     * implementation in their real rendering implementations.
     */
    public function render(): string
    {
        $output = "";

        foreach ($this->fields as $name => $field) {
            $output .= $field->render();
        }

        return $output;
    }
}

/**
 * The fieldset element is a Concrete Composite.
 */
class Fieldset extends FieldComposite
{
    public function render(): string
    {
        // Note how the combined rendering result of children is incorporated
        // into the fieldset tag.
        $output = parent::render();

        return "<fieldset><legend>{$this->title}</legend>\n$output</fieldset>\n";
    }
}

/**
 * And so is the form element.
```

```
{
    protected $url;

    public function __construct(string $name, string $title, string $url)
    {
        parent::__construct($name, $title);
        $this->url = $url;
    }

    public function render(): string
    {
        $output = parent::render();
        return "<form action=\"{$this->url}\">\n<h3>{$this->title}</h3>\n$output</form>\r
    }
}

/**
 * The client code gets a convenient interface for building complex tree
 * structures.
 */
function getProductForm(): FormElement
{
    $form = new Form('product', "Add product", "/product/add");
    $form->add(new Input('name', "Name", 'text'));
    $form->add(new Input('description', "Description", 'text'));

    $picture = new Fieldset('photo', "Product photo");
    $picture->add(new Input('caption', "Caption", 'text'));
    $picture->add(new Input('image', "Image", 'file'));
    $form->add($picture);

    return $form;
}

/**
 * The form structure can be filled with data from various sources. The Client
 * doesn't have to traverse through all form fields to assign data to various
 * fields since the form itself can handle that.
 */
function loadProductData(FormElement $form)
{
    $data = [
        'name' => 'Apple MacBook',
        'description' => 'A decent laptop.',
        'photo' => [
            'caption' => 'Front photo.',
            'image' => 'photo1.png',
        ],
    ];
```

```php
/**
 * The client code can work with form elements using the abstract interface.
 * This way, it doesn't matter whether the client works with a simple component
 * or a complex composite tree.
 */
function renderProduct(FormElement $form)
{
    // ..

    echo $form->render();

    // ..
}

$form = getProductForm();
loadProductData($form);
renderProduct($form);
```

Output.txt: Resultado de la ejecución

```
<form action="/product/add">
<h3>Add product</h3>
<label for="name">Name</label>
<input name="name" type="text" value="Apple MacBook">
<label for="description">Description</label>
<input name="description" type="text" value="A decent laptop.">
<fieldset><legend>Product photo</legend>
<label for="caption">Caption</label>
<input name="caption" type="text" value="Front photo.">
<label for="image">Image</label>
<input name="image" type="file" value="photo1.png">
</fieldset>
</form>
```

Ejemplo conceptual   Ejemplo del mundo real

LEER SIGUIENTE

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