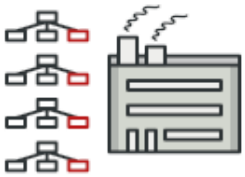

Abstract Factory in PHP

Abstract Factory is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.

Abstract Factory defines an interface for creating all distinct products, but leaves the actual product creation to concrete factory classes. Each factory type corresponds to a certain product variety.

The client code calls creational methods of a factory object instead of creating products directly with a constructor call (`new` operator). Since a factory corresponds to a single products variant, all its products will be compatible.

Client code works with factories and products only through their abstract interfaces. It allows the same client code work with different products. You just create a new concrete factory class and pass it to client code.

- If you can't figure out the difference between various factory patterns and concepts, then read our **Factory Comparison guide**.

Learn more about Abstract Factory →

Usage of the pattern in PHP

Complexity: ★★☆

Popularity: ★★★

Usage examples: The Abstract Factory pattern is pretty common in PHP code. Many frameworks and libraries use it to provide a way to extend and customize their standard components.

Navigation

Conceptual Example

This example illustrates the structure of the **Abstract Factory** design pattern. It focuses on answering these questions:

- What classes does it consists of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

After learning about the pattern's structure it'll be easier for you to grasp the following example, based on a real world PHP use case.

index.php: Conceptual Example

```php
<?php

namespace RefactoringGuru\AbstractFactory\Conceptual;

/**
 * The Abstract Factory interface declares a set of methods that return
 * different abstract products. These products are called a family and are
 * related by a high-level theme or concept. Products of one family are usually
 * able to collaborate among themselves. A family of products may have several
 * variants, but the products of one variant are incompatible with products of
 * another.
 */
interface AbstractFactory
{
    public function createProductA(): AbstractProductA;
```

```
    public function createProductB(): AbstractProductB;
}

/**
 * Concrete Factories produce a family of products that belong to a single
 * variant. The factory guarantees that resulting products are compatible. Note
 * that signatures of the Concrete Factory's methods return an abstract product,
 * while inside the method a concrete product is instantiated.
 */
class ConcreteFactory1 implements AbstractFactory
{
    public function createProductA(): AbstractProductA
    {
        return new ConcreteProductA1;
    }

    public function createProductB(): AbstractProductB
    {
        return new ConcreteProductB1;
    }
}

/**
 * Each Concrete Factory has a corresponding product variant.
 */
class ConcreteFactory2 implements AbstractFactory
{
    public function createProductA(): AbstractProductA
    {
        return new ConcreteProductA2;
    }

    public function createProductB(): AbstractProductB
    {
        return new ConcreteProductB2;
    }
}

/**
 * Each distinct product of a product family should have a base interface. All
 * variants of the product must implement this interface.
 */
interface AbstractProductA
{
    public function usefulFunctionA(): string;
}

/**
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductA1 implements AbstractProductA
{
```

```
    public function usefulFunctionA(): string
    {
        return "The result of the product A1.";
    }
}

class ConcreteProductA2 implements AbstractProductA
{
    public function usefulFunctionA(): string
    {
        return "The result of the product A2.";
    }
}

/**
 * Here's the the base interface of another product. All products can interact
 * with each other, but proper interaction is possible only between products of
 * the same concrete variant.
 */
interface AbstractProductB
{
    /**
     * Product B is able to do its own thing...
     */
    public function usefulFunctionB(): string;

    /**
     * ...but it also can collaborate with the ProductA.
     *
     * The Abstract Factory makes sure that all products it creates are of the
     * same variant and thus, compatible.
     */
    public function anotherUsefulFunctionB(AbstractProductA $collaborator): string;
}

/**
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductB1 implements AbstractProductB
{
    public function usefulFunctionB(): string
    {
        return "The result of the product B1.";
    }

    /**
     * The variant, Product B1, is only able to work correctly with the variant,
     * Product A1. Nevertheless, it accepts any instance of AbstractProductA as
     * an argument.
     */
    public function anotherUsefulFunctionB(AbstractProductA $collaborator): string
    {
```

```
        $result = $collaborator->usefulFunctionA();

        return "The result of the B1 collaborating with the ({$result})";
    }
}

class ConcreteProductB2 implements AbstractProductB
{
    public function usefulFunctionB(): string
    {
        return "The result of the product B2.";
    }

    /**
     * The variant, Product B2, is only able to work correctly with the variant,
     * Product A2. Nevertheless, it accepts any instance of AbstractProductA as
     * an argument.
     */
    public function anotherUsefulFunctionB(AbstractProductA $collaborator): string
    {
        $result = $collaborator->usefulFunctionA();

        return "The result of the B2 collaborating with the ({$result})";
    }
}

/**
 * The client code works with factories and products only through abstract
 * types: AbstractFactory and AbstractProduct. This lets you pass any factory or
 * product subclass to the client code without breaking it.
 */
function clientCode(AbstractFactory $factory)
{
    $productA = $factory->createProductA();
    $productB = $factory->createProductB();

    echo $productB->usefulFunctionB() . "\n";
    echo $productB->anotherUsefulFunctionB($productA) . "\n";
}

/**
 * The client code can work with any concrete factory class.
 */
echo "Client: Testing client code with the first factory type:\n";
clientCode(new ConcreteFactory1);

echo "\n";

echo "Client: Testing the same client code with the second factory type:\n";
clientCode(new ConcreteFactory2);
```

Output.txt: Execution result

```
Client: Testing client code with the first factory type:
The result of the product B1.
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:
The result of the product B2.
The result of the B2 collaborating with the (The result of the product A2.)
```

Real World Example

In this example, the **Abstract Factory** pattern provides an infrastructure for creating various types of templates for different elements of a web page.

A web application can support different rendering engines at the same time, but only if its classes are independent of the concrete classes of rendering engines. Hence, the application's objects must communicate with template objects only via their abstract interfaces. Your code shouldn't create the template objects directly, but delegate their creation to special factory objects. Finally, your code shouldn't depend on the factory objects either but, instead, should work with them via the abstract factory interface.

As a result, you will be able to provide the app with the factory object that corresponds to one of the rendering engines. All templates, created in the app, will be created by that factory and their type will match the type of the factory. If you decide to change the rendering engine, you'll be able to pass a new factory to the client code, without breaking any existing code.

index.php: Real World Example

```php
<?php

namespace RefactoringGuru\AbstractFactory\RealWorld;

/**
 * The Abstract Factory interface declares creation methods for each distinct
 * product type.
 */
interface TemplateFactory
{
    public function createTitleTemplate(): TitleTemplate;

    public function createPageTemplate(): PageTemplate;

    public function getRenderer(): TemplateRenderer;
```

```php
}

/**
 * Each Concrete Factory corresponds to a specific variant (or family) of
 * products.
 *
 * This Concrete Factory creates Twig templates.
 */
class TwigTemplateFactory implements TemplateFactory
{
    public function createTitleTemplate(): TitleTemplate
    {
        return new TwigTitleTemplate;
    }

    public function createPageTemplate(): PageTemplate
    {
        return new TwigPageTemplate($this->createTitleTemplate());
    }

    public function getRenderer(): TemplateRenderer
    {
        return new TwigRenderer();
    }
}

/**
 * And this Concrete Factory creates PHPTemplate templates.
 */
class PHPTemplateFactory implements TemplateFactory
{
    public function createTitleTemplate(): TitleTemplate
    {
        return new PHPTemplateTitleTemplate;
    }

    public function createPageTemplate(): PageTemplate
    {
        return new PHPTemplatePageTemplate($this->createTitleTemplate());
    }

    public function getRenderer(): TemplateRenderer
    {
        return new PHPTemplateRenderer();
    }
}

/**
 * Each distinct product type should have a separate interface. All variants of
 * the product must follow the same interface.
 *
 * For instance, this Abstract Product interface describes the behavior of page
```

```php
 * title templates.
 */
interface TitleTemplate
{
    public function getTemplateString(): string;
}

/**
 * This Concrete Product provides Twig page title templates.
 */
class TwigTitleTemplate implements TitleTemplate
{
    public function getTemplateString(): string
    {
        return "<h1>{{ title }}</h1>";
    }
}

/**
 * And this Concrete Product provides PHPTemplate page title templates.
 */
class PHPTemplateTitleTemplate implements TitleTemplate
{
    public function getTemplateString(): string
    {
        return "<h1><?= \$title; ?></h1>";
    }
}

/**
 * This is another Abstract Product type, which describes whole page templates.
 */
interface PageTemplate
{
    public function getTemplateString(): string;
}

/**
 * The page template uses the title sub-template, so we have to provide the way
 * to set it in the sub-template object. The abstract factory will link the page
 * template with a title template of the same variant.
 */
abstract class BasePageTemplate implements PageTemplate
{
    protected $titleTemplate;

    public function __construct(TitleTemplate $titleTemplate)
    {
        $this->titleTemplate = $titleTemplate;
    }
}
```

```php
/**
 * The Twig variant of the whole page templates.
 */
class TwigPageTemplate extends BasePageTemplate
{
    public function getTemplateString(): string
    {
        $renderedTitle = $this->titleTemplate->getTemplateString();

        return <<<HTML
        <div class="page">
            $renderedTitle
            <article class="content">{{ content }}</article>
        </div>
        HTML;
    }
}

/**
 * The PHPTemplate variant of the whole page templates.
 */
class PHPTemplatePageTemplate extends BasePageTemplate
{
    public function getTemplateString(): string
    {
        $renderedTitle = $this->titleTemplate->getTemplateString();

        return <<<HTML
        <div class="page">
            $renderedTitle
            <article class="content"><?= \$content; ?></article>
        </div>
        HTML;
    }
}

/**
 * The renderer is responsible for converting a template string into the actual
 * HTML code. Each renderer behaves differently and expects its own type of
 * template strings passed to it. Baking templates with the factory let you pass
 * proper types of templates to proper renders.
 */
interface TemplateRenderer
{
    public function render(string $templateString, array $arguments = []): string;
}

/**
 * The renderer for Twig templates.
 */
class TwigRenderer implements TemplateRenderer
{
```

```
    public function render(string $templateString, array $arguments = []): string
    {
        return \Twig::render($templateString, $arguments);
    }
}

/**
 * The renderer for PHPTemplate templates. Note that this implementation is very
 * basic, if not crude. Using the `eval` function has many security
 * implications, so use it with caution in real projects.
 */
class PHPTemplateRenderer implements TemplateRenderer
{
    public function render(string $templateString, array $arguments = []): string
    {
        extract($arguments);

        ob_start();
        eval(' ?>' . $templateString . '<?php ');
        $result = ob_get_contents();
        ob_end_clean();

        return $result;
    }
}

/**
 * The client code. Note that it accepts the Abstract Factory class as the
 * parameter, which allows the client to work with any concrete factory type.
 */
class Page
{

    public $title;

    public $content;

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

    // Here's how would you use the template further in real life. Note that the
    // page class does not depend on any concrete template classes.
    public function render(TemplateFactory $factory): string
    {
        $pageTemplate = $factory->createPageTemplate();

        $renderer = $factory->getRenderer();

        return $renderer->render($pageTemplate->getTemplateString(), [
```

```
            'title' => $this->title,
            'content' => $this->content
        ]);
    }
}

/**
 * Now, in other parts of the app, the client code can accept factory objects of
 * any type.
 */
$page = new Page('Sample page', 'This it the body.');

echo "Testing actual rendering with the PHPTemplate factory:\n";
echo $page->render(new PHPTemplateFactory);


// Uncomment the following if you have Twig installed.

// echo "Testing rendering with the Twig factory:\n"; echo $page->render(new
// TwigTemplateFactory);
```

Output.txt: Execution result

```
Testing actual rendering with the PHPTemplate factory:
<div class="page">
    <h1>Sample page</h1>
    <article class="content">This it the body.</article>
</div>
```

RETURN

← Factory Method in PHP

Abstract Factory in Other Languages