Abstract Factory

Objective

Allow the creation of related or dependent object groups, without specifying their specific classes.

Function

Work with objects from different families so that they do not mix with each other, making transparent the type of specific family being used.

Structure

- Class(client): Represents the person or event that triggers the execution of the pattern.

- AbstractProduct (One, Two): Interfaces that define the structure of objects to create families.

- Product (One, Two): Classes that inherit from AbstractProduct in order to implement families of concrete objects.

- Platform (One, Two): They represent the specific factories that will serve to create the instances of all classes in the family. In this class you must there is a method for creating each of the family's classes.

- AbstractPlatform: Defines the structure of the factories and must provide a method for every class in the family.

The structure that meets this pattern is shown in Figure 1

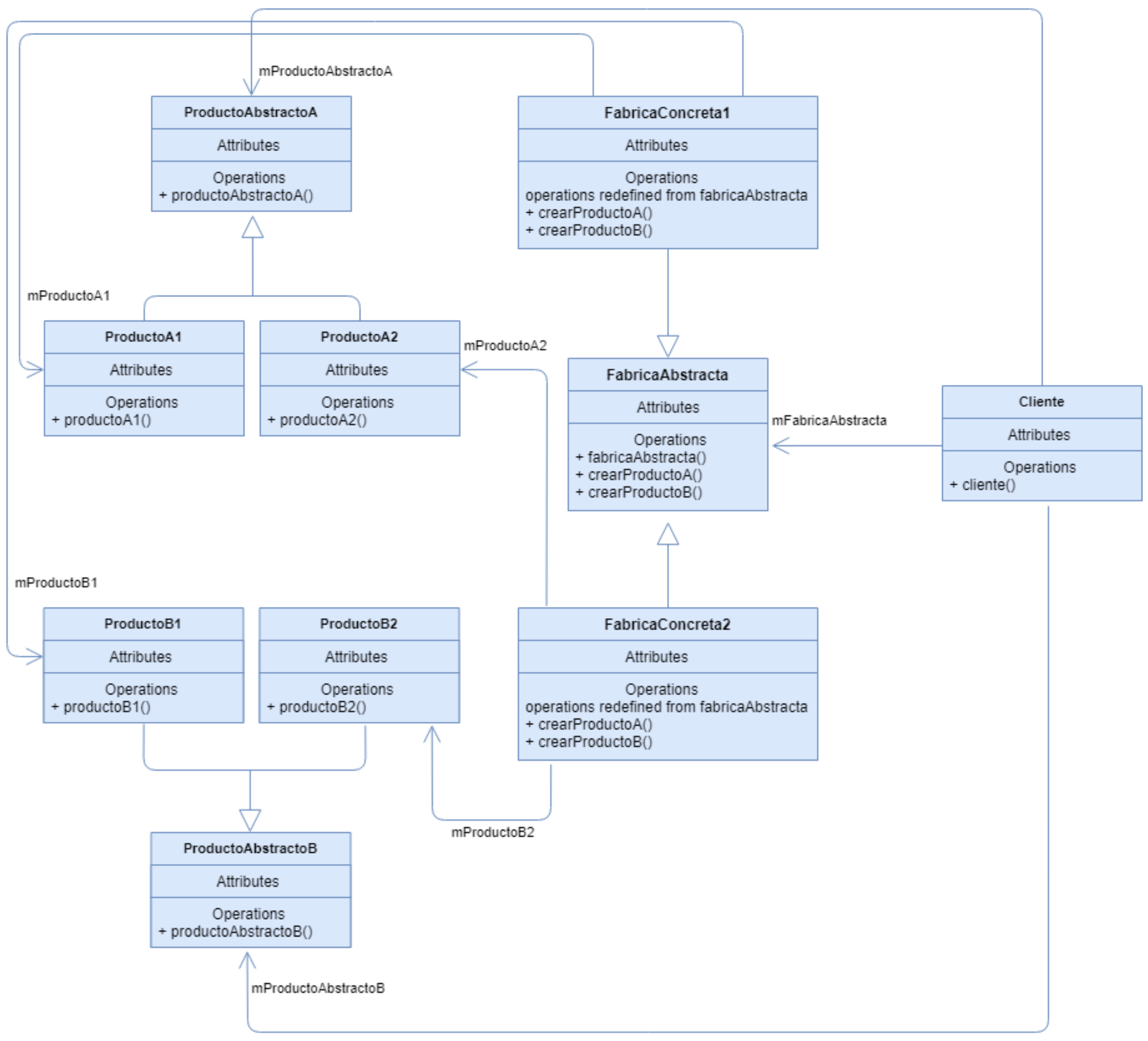

Figure 1: UML Diagram Abstract Factory Pattern

Applications

The use of the Abstract Factory pattern is recommended in the following cases:

- The system to be developed must be independent of the way in which their objects.

- The system must be configured with one or more possible product sets.

- A group of related objects is designed to be used together, and this restriction needs to be reinforced.

Design Patterns Collaborators

- An abstract factory is always a single instance object, corresponding to the Singleton pattern.

- An abstract factory can be implemented as a Factory Method pattern or a Prototype pattern.

Scope of action

Applied at the object level.

Problem

To implement object groups you need to specify their exact names in the client class, i.e. if you need to change an object you must also change the class that instances it.

Solution

The Abstract Factory design pattern isolates the "client" classes from the names and definitions of the "product" classes (it supports the implicit creation of objects); so, the only way to get a "product" is through an Abstract Factory class, in this way the set of products can be easily modified without the need to update each client class.

Diagram or Implementation

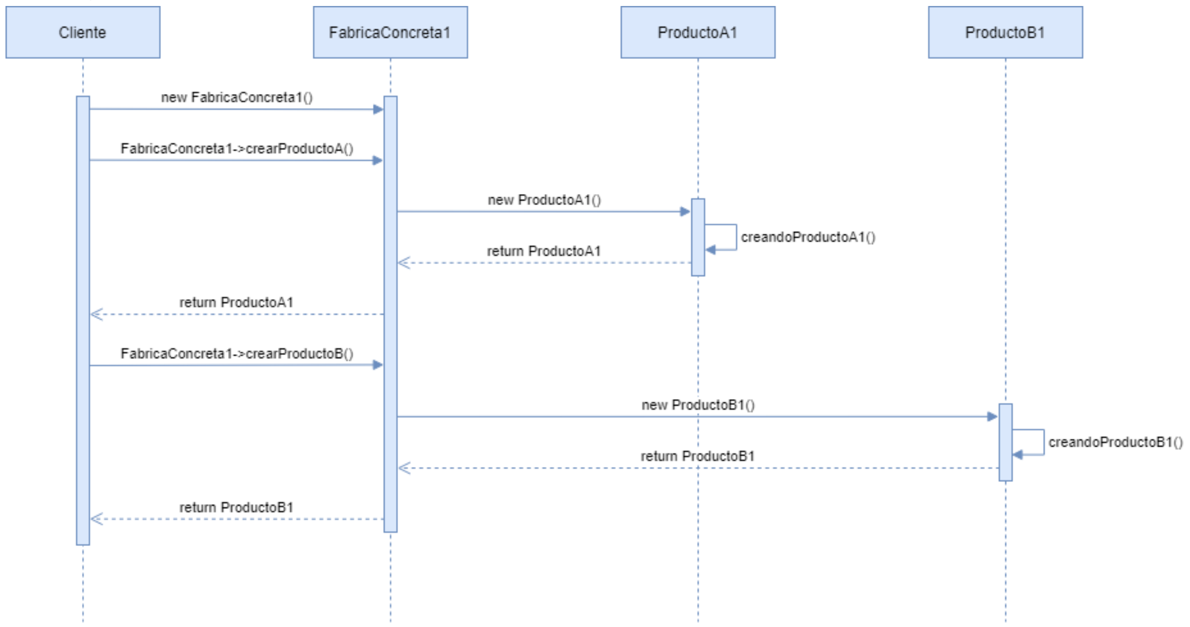

Figure 2: UML Diagram Abstract Factory Pattern

Figure 2 explains the behaviour of the pattern by means of a sequence diagram.

- Client class requests the creation of the FabricaConcreta1 to the AbstractFactory class.

- AbstractFactory class creates an instance of the FabricaConcreta1 and returns it.

- The client requests the FabricaConcreta1 to create a ProductoA.

- The FabricaConcreta1 class creates an instance of ProductoA1 which is part of the family1 and returns it.

- Client class this time requests the creation of the FabricaConcreta2 to the AbstractFactory class.

- The AbstractFactory class creates an instance of the FabricaConcreta2.

- The client requests FabricaConcreta2 to create a ProductoA .

- The FabricaConcreta2 class creates an instance of ProductoA2 which is part of family2 and returns it.

Study Cases

Interface System

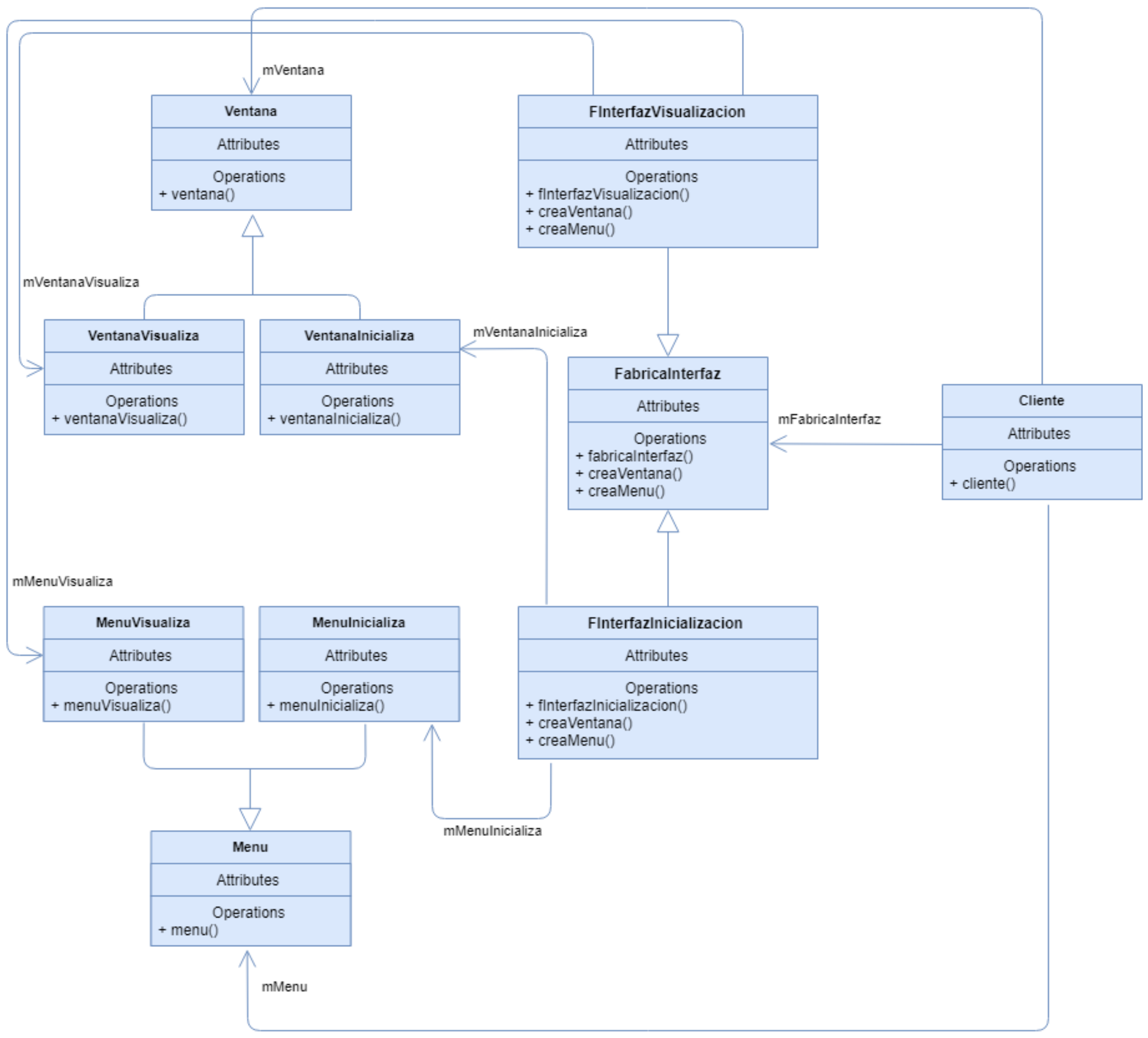

Figure 3: UML Diagram Interface System

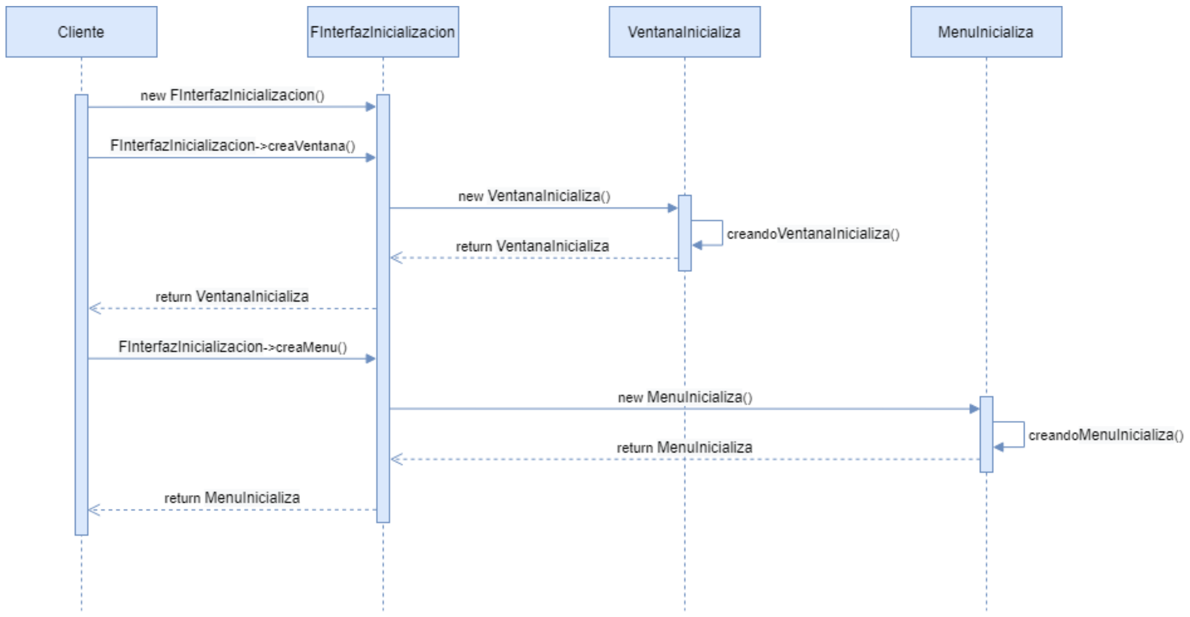

Figure 4: UML Diagram Interface System

Implementation in 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;
```

```php
    }

    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);
```