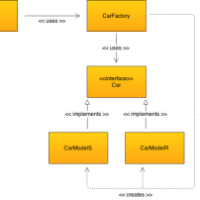
The factory design pattern in PHP explained

Published June 05, 2015

We consider the use of the **factory pattern** in those cases when we want the main part of our code (a.k.a. business logic) to address only the management of objects rather than their making. For these cases, the **factory pattern** instructs us to separate the making of objects into a factory class while leaving the main part of the app to handle the management of the objects.

Let's demonstrate some scenarios in which the factory design pattern can be beneficial:

1. If a class manages a shopping cart, it should deal only with shopping cart management (e.g., adding or removing products from the cart) and not be concerned with the manufacturing of those products.
2. A warehouse management class follows the quantities of items entering and leaving the warehouse but has no interest in making those items.
3. A car buying agency should simply track orders and not have to also deal with the details of how the cars are made.



We consider the use of the factory pattern in those cases when we want the main part of our code to address only the management of objects rather than their making.

What are the characteristics of the factory design pattern?

The factory design pattern has two main sections:

* The main part of the app that manages objects but does not make them.
* The factory class that makes the objects.

When to consider the use of the factory design pattern?

The **factory pattern** is a good solution for those cases in which the main part of the app needs to manage the objects and not create them.

Let's demonstrate this with a CarOrder class that handles cars orders. Each time a new car is ordered, we call the order method and pass the car type as a parameter ('r' or 's' model). The order method then makes a new car object (based on the parameter) and adds the newly created car object to the $carOrders array that stores the list of cars that were ordered.



In this textbook example, it does not seem like a bad idea to mix between the code that makes objects and the code that manages them. However, most of the code in real life is much more complex. For example, different car models may very well need different chassis; the higher-end model may have extra comfortable seats or be a convertible, etc. It's too much responsibility for a method whose sole purpose is to order cars to handle their manufacturing as well.

The factory method

Once we establish that it is better not to place the responsibility for the production of cars on the order method, we can try to solve the problem by moving the code responsible for the production of cars to a separate make method whose sole purpose would be to produce cars.

Let's look at the code after we've separated the class into the following two methods:

1. The order method manages the order of cars.
2. The make method makes the car objects.

class CarOrder {

  protected $carOrders = array();

  protected $car;

  // Order the car.

  public function order($model=null)

  {

    $car = $this->make($model);

    $this->carOrders[] = $car->getModel();

  }

  // The actual making of the car is now separated into a method of its own.

  protected function make($model=null)

  {

    if(strtolower($model) == 'r')

      return $this->car = new CarModelR();

    return $this->car = new CarModelS();

  }

  public function getCarOrders()

  {

    return $this->carOrders;

  }

}

The code above demonstrates the making of objects in a dedicated make method. This approach is also known as the **factory method**. Using the factory method is legitimate and can certainly help in some cases but not in all of them, due to the following reasons:

1. The more methods we add to the cars' manufacturing and order processes, the more we bloat the class and make it less readable and maintainable.
2. We still make cars within the CarOrder class, which should engage only in orders and not in automobile manufacturing.

In order to solve these problems, we use the factory pattern that instructs us to move the making of objects to a separate factory class.

The Factory class

Let's create a dedicated CarFactory class with the make method that creates the car objects on demand:

class CarFactory {

  protected $car;

  // Determine which model to manufacture, and instantiate

  // the concrete classes that make each model.

  public function make($model=null)

  {

    if(strtolower($model) == 'r')

      return $this->car = new CarModelR();

    return $this->car = new CarModelS();

  }

}

Now that we have a dedicated CarFactory class, we need to create an object out of it in the constructor of the CarOrder class so we can use it's make() method.

class CarOrder {

  protected $carOrders = array();

  protected $car;

  // First, create the carFactory object in the constructor.

  public function \_\_construct()

  {

    $this->car = new CarFactory();

  }

  public function order($model=null)

  {

    // Use the make() method from the carFactory.

    $car = $this->car->make($model);

    $this->carOrders[]=$car->getModel();

  }

  public function getCarOrders()

  {

    return $this->carOrders;

  }

}

The main part of the app needs to pass parameters to the factory class (in our example, it is a short string that tells the class which car object to create) and, in return, it gets the objects. The advantage is obvious: the main part is now free to do its job without having to delve deeply into the technical details of how the objects (e.g., the dependencies that such an object may have) are made.

Understanding the example can lead us to formulate the most important characteristic of the factory pattern:

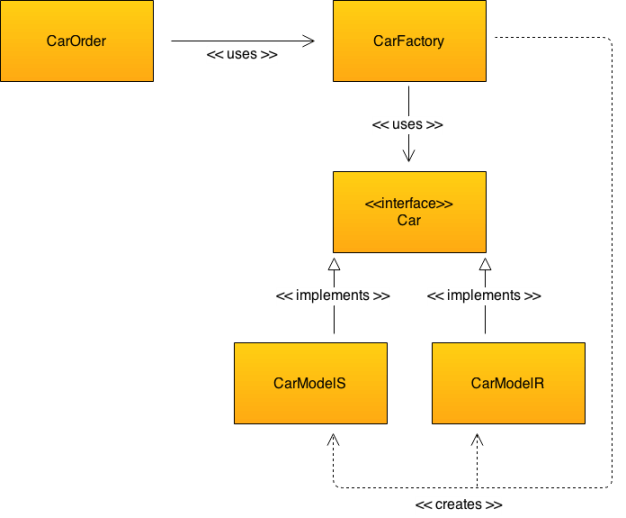
*When we use the factory pattern, we separate the making of objects into a dedicated class whose main responsibility is the making of objects.*

The car classes

But what about the actual classes that make the car objects? For this purpose, we use a Car [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces), and implement it with concrete classes that make the r and s models. Let's start with the interface which tells us what features all the car objects need to have:



The following UML diagram may help us in understanding the code structure and the connection between the classes:



Let's test the code

Let's use the following code in order to test everything that we have written:

$carOrder = new CarOrder;

var\_dump($carOrder->getCarOrders());

$carOrder->order('r');

var\_dump($carOrder->getCarOrders());

$carOrder->order('s');

var\_dump($carOrder->getCarOrders());

And the result:

**array**(size=0) empty**array**(size=1) 0 => string 'r' (length=1)**array**(size=2) 0 => string 'r' (length=1) 1 => string 's' (length=1)

In conclusion

We should consider the use of the factory design pattern in those cases where we need to free a class from making the objects that it manages. For this aim, we make the objects in a dedicated factory class.