Strategy — the power of interface

Published March 12, 2015

Have you ever wondered how other programmers solve the same programmatic problems that you toil endlessly upon? Or, better yet, if there is an already established solution that may help you to write the code?

There is, in fact, and it's been around for quite a while. It's done through the use of **design patterns**, which are programmatic solutions to common problems that you may confront on a daily basis. There exist a few dozen design patterns, and one of the most beneficial for the PHP programmer is the **strategy design pattern**.

*We consider use of the strategy pattern when we need to choose between similar classes that are different only in their implementation.*

We consider use of the **strategy design pattern** when we need to choose between similar classes that are different only in their implementation. For these cases, the **strategy pattern** instructs us to create an [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces) that the classes can implement, while the choice of which of the classes to implement is done during the program runtime.

What are the strategy design pattern characteristics?

There are three main code sections that are the hallmarks of the **strategy pattern**:

1. Several classes that implement the same [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces).
2. A code that chooses from which of the classes to create an object.
3. A client class that is fed with the object, and performs the task that the program is meant to do.

When to consider the use of the strategy design pattern?

The strategy pattern is a good solution for those cases in which we need the program to select which code alternative, of several similar code alternatives, to implement at runtime (i.e. when the program is running).

*The strategy pattern is a good solution for those cases in which we need the program to select which code alternative to implement at runtime.*

We all know what code alternatives are. Consider the following plain function with the name of couponGenerator that generates different coupons for different car types. To those who are interested in buying a BMW, it offers a different coupon than to those who are interested in buying a Mercedes.

// Generate different coupons for different car types.

function cupounGenerator($car)

{

  if($car == "bmw")

  {

    $cupoun = "Get 5% off the price of your new BMW";

  }

  else if($car == "mercedes")

  {

    $cupoun = "Get 7% off the price of your new Mercedes";

  }

  return $cupoun;

}

The coupon is the discount rate that is offered to the customers interested in purchasing a car. It can take a number of factors into account for weighting the discount rate. For example, we can weigh into the decision of the discount rate the following factors:

* We might want to offer a discount during a downturn in the purchase of cars.
* We might want to offer a discount when the stock of cars for sale is too large.

Let's rewrite the function couponGenerator to take into account these factors. For this purpose, we add the following variables:

* $isHighSeason to indicate if it is a period of record sales.
* $bigStock to indicate if there are a large number of cars in stock.

The rewritten code now includes the new variables that affect the discount rate:

// Add to the discount during the downturn or if the stock of cars is too large.

function cupounGenerator($car)

{

  $discount = 0;

  $isHighSeason = false;

  $bigStock = true;

  if($car == "bmw")

  {

    if(!$isHighSeason) {$discount += 5;}

    if($bigStock) {$discount += 7;}

  }

  else if($car == "mercedes")

  {

    if(!$isHighSeason) {$discount += 4;}

    if($bigStock) {$discount += 10;}

  }

  return $cupoun = "Get {$discount}% off the price of your new car.";

}

echo cupounGenerator("bmw");

**Result:**  
Get 12% off the price of your new car

From procedural code to object-based code

Up till now, the requirements were pretty basic and, therefore, we could settle for a procedural code which relies on a single function that gives a discount by manufacturer. In reality, however, the considerations can be much more complex, which require us to write a number of functions for determining the discount rate.

According to the considerations that we have just seen, we need to create the two following functions:

* A function that adds a discount when sales take a downturn. Let's call it: addSeasonDiscount.
* A function that adds a discount when the inventory is too large that we'll name accordingly: addStockDiscount.

In those cases where we want the different types of cars to be treated differently, we may convene the functions under separate classes, having a class for each type of car. In our case, we assign one class to generate coupons for BMW and a second class to generate coupons for Mercedes.

* Let's call the class that generates coupons for BMW: bmwCouponGenerator.
* And the class that generates coupons for Mercedes: mercedesCouponGenerator.

The interface

In fact, we need to write two classes that have methods that share the same names. In this situation, we should write an [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces) that will oblige every class that implements it, including those that we are just going to write and those that we would like to add in the future.

*The first stractural feature of the starategy pattern is that we have several classes that implement the same interface.*

Let's give the interface the name of carCouponGenerator and let it contain the abstract methods that we would like to implement in the concrete classes.

// interface to commit the classes that generate coupons:

// 1. to add downturn discount

// 2. to add stock discount

interface carCouponGenerator {

  function addSeasonDiscount();

  function addStockDiscount();

}

The classes that implement the interface

Once we have written the interface, we can write the classes that implement the interface.

Let's start with the bmwCouponGenerator that generates coupons for BMW:

// implements the interface for BMW.

class bmwCouponGenerator implements carCouponGenerator {

  private $discount = 0;

  private $isHighSeason = false;

  private $bigStock = true;

  public function addSeasonDiscount()

  {

    if(!$this->isHighSeason) return $this->discount += 5;

    return $this->discount +=0;

  }

  public function addStockDiscount()

  {

    if($this->bigStock) return $this->discount += 7;

    return $this->discount +=0;

  }

}

Also, let's write the class mercedesCouponGenerator that generates coupons for Mercedes.

// implements the interface for Mercedes.

class mercedesCouponGenerator implements carCouponGenerator {

  private $discount = 0;

  private $isHighSeason = false;

  private $bigStock = true;

  public function addSeasonDiscount()

  {

    if(!$this->isHighSeason) return $this->discount += 4;

    return $this->discount +=0;

  }

  public function addStockDiscount()

  {

    if($this->bigStock) return $this->discount += 10;

    return $this->discount +=0;

  }

}

So far, we have seen the most important feature of the strategy pattern which is the use of an interface to oblige the classes that implement it to the code that they should contain.

The selector code

The second feature of the strategy pattern is a selector code that decides from which class that implements the interface to create the object, and then creates the object accordingly.

*The second feature of the strategy pattern is a selector code that decides from which class that implements the interface to create the object, and then creates the object accordingly.*

The role of the selector code is going to be fulfilled by the function couponObjectGenerator that chooses which of the objects to create (the BMW or the Mercedes), and then creates it accordingly. We pass as a parameter to the function the specific vehicle type using the variable $car, and it chooses accordingly whether to create an object from the bmwCouponGenerator class or from the mercedesCouponGenerator class.

// Create object from the alternative classes.

function couponObjectGenerator($car)

{

  if($car == "bmw")

  {

    $carObj = new bmwCouponGenerator;

  }

  else if($car == "mercedes")

  {

    $carObj = new mercedesCouponGenerator;

  }

  return $carObj;

}

The code that selects which object to create is such an important feature in the strategy design pattern because of the fact that it allows the selection of which object to create during the runtime of the program.

The client class

We have seen how the selector code selects from which of the classes that implement the interface to create an object, and this object can now be used to issue the coupon. To do this, we write the class couponGenerator which receives the object that was created by the selector code, and then returns a coupon through the method getCoupon.

*The third structural feature of the strategy pattern is a client class that puts the code into practice.*

We pass the selected object into the class through a constructor method and, in order to be sure that we pass the right kind of object, we use [type hinting](https://phpenthusiast.com/object-oriented-php-tutorials/type-hinting). The use of [type hinting](https://phpenthusiast.com/object-oriented-php-tutorials/type-hinting) here is extremely beneficial because it forces the constructor to get only objects that implement the interface.

Once we pass the object into the class, we assign it to the private property $carCoupon.

Now, the getCoupon method can generate the coupon. It does so by using the methods it receives from the object that we have just assigned to the $carCoupon variable.

Let's see the code:

// The client class generates coupon from the object of choice.

class couponGenerator {

  private $carCoupon;

  // Get only objects that belong to the interface.

  public function \_\_construct(carCouponGenerator $carCoupon)

  {

    $this->carCoupon = $carCoupon;

  }

  // Use the object's methods to generate the coupon.

  public function getCoupon()

  {

    $discount = $this->carCoupon->addSeasonDiscount();

    $discount = $this->carCoupon->addStockDiscount();

    return $coupon = "Get {$discount}% off the price of your new car.";

  }

}

Let's test the code we have just written

The code given below tests everything that we have just written. The test code is written using the following two steps:

1. Decide from which of the classes that implement the interface to create an object, and pass your decision to the couponObjectGenerator that creates the object of choice.
2. Pass the object that we have created in the first step to the couponGenerator class that generates coupons through the use of the getCoupon method.

1 2 3 4 5 6 7 8 9101112// Test the code.

$car = "bmw";

$carObj = couponObjectGenerator($car);

$couponGenerator = new couponGenerator($carObj);

echo $couponGenerator->getCoupon();

echo "<hr />";

$car = "mercedes";

$carObj = couponObjectGenerator($car);

$couponGenerator = new couponGenerator($carObj);

echo $couponGenerator->getCoupon();

**Result:**  
Get 12% off the price of your new car.  
Get 14% off the price of your new car.

In conclusion

We should consider the use of the **strategy design pattern** in those cases where we want to choose which object to create from several similar classes during the runtime of the program. The classes are similar in that they can use the same name for their methods, but they are different in the way that they apply the methods. Therefore, the strategy pattern uses an interface that the classes can implement.