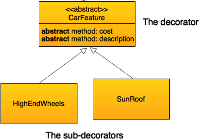
The decorator pattern in PHP and convertibles

Published August 05, 2015

We use the **decorator design pattern** to add new optional features to our code without changing the existing classes. The new features are added by creating new classes that belong to the same type as the existing classes.

*The new features are added by creating new classes that belong to the same type as the existing classes.*



The basic classes

In the example given below, an auto manufacturing company uses an [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces) to dictate to all of the implementing classes that they need to have price and description methods.

interface Car {

function cost();

function description();

}

The company manufactures different types of cars, including: compact, sedan, SUV and luxury.

That's how the interface is implemented for the SUV type:

class Suv implements Car {

function cost()

{

return 30000;

}

function description ()

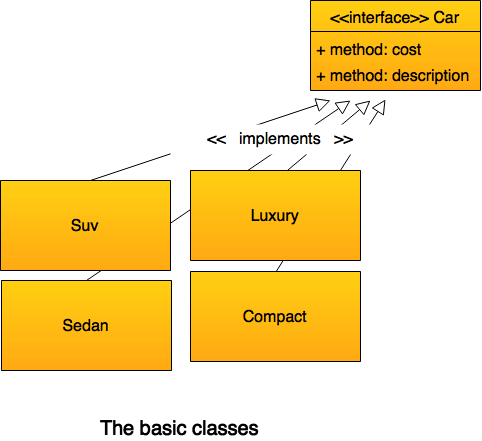
{

return "Suv";

}

}

The following diagram may help us in understanding the code:



The problem

The problem starts when customers are given the choice to add optional features to their car, like high end wheels, a car rear spoiler, or a sun roof. We wouldn't like to change the existing classes to include optional features, so we need a better solution.

The solution

When we want to add optional features to our code we use the **decorator pattern**, which instructs us to add to the basic classes that implement the interface, an [abstract class](https://phpenthusiast.com/object-oriented-php-tutorials/abstract-classes-and-methods) that also implements the same interface. The abstract class is used as a super-class that the features classes [inherit](https://phpenthusiast.com/object-oriented-php-tutorials/inheritance-in-object-oriented-php) from.

The **decorator pattern** thus has the following structure:

* A **decorator** which is an [abstract](https://phpenthusiast.com/object-oriented-php-tutorials/abstract-classes-and-methods) class that implements the interface.
* **Sub-classes** that inherit the decorator class.

The decorator

The abstract class CarFeature implements the Car interface while, at the same time, it is used as the super-class that the concrete features classes [inherit](https://phpenthusiast.com/object-oriented-php-tutorials/inheritance-in-object-oriented-php) from.

abstract class CarFeature implements Car {

protected $car;

function \_\_construct(Car $car)

{

$this->car = $car;

}

abstract function cost();

abstract function description();

}

The abstract class CarFeature implements the Car interface in an unusual way by re-defining the interface methods as [abstract methods](https://phpenthusiast.com/object-oriented-php-tutorials/abstract-classes-and-methods). The abstract class also holds a reference to an object that was created from one of the basic classes.

The sub-classes that inherit the decorator

Now, let's add the concrete class SunRoof which extends the CarFeature decorator.

class SunRoof extends CarFeature {

function cost ()

{

return $this->car->cost() + 1500;

}

function description()

{

return $this->car->description() . ", sunroof";

}

}

In pretty much the same way, we can write a HighEndWheels class that can give a customer the choice to add high end wheels to their car.

class HighEndWheels extends CarFeature {

function cost ()

{

return $this->car->cost() + 2000;

}

function description()

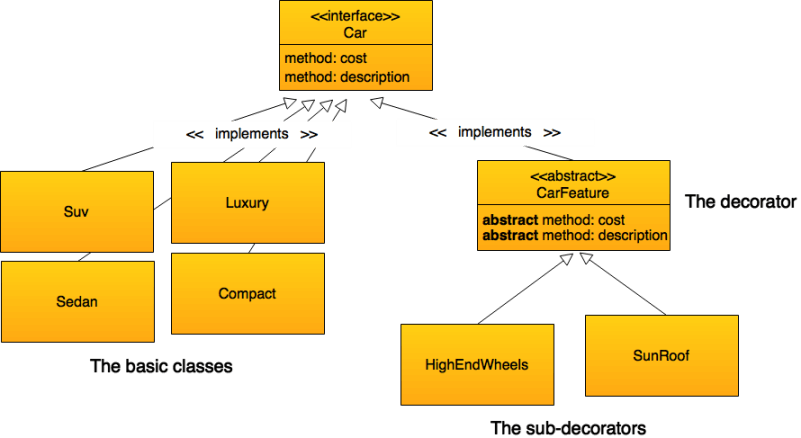
{

return $this->car->description() . ", high end wheels";

}

}

The following diagram can help us understand the code structure and the connection between the classes:



Let's test the code

In order to implement the code, we need to:

1. First, create an object from one of the basic classes (in our example, it is the Suv class).
2. Pass the object that was created from the basic class as a parameter to the class that adds the first feature (i.e., the SunRoof class).
3. Pass the object that was created from the first feature class to the second feature class, and so on until we finish adding all the optional features.
4. Run the methods on the last object that we created in the process of decoration.

The code below equips the basic Suv with a SunRoof.

// Create an object from one of the basic classes.

$basicCar = new Suv();

// Pass the object from the basic class as a parameter to the first feature class.

$carWithSunRoof = new SunRoof($basicCar);

Once we finish adding the features to the basic class, we are able to run the cost and description methods to give us the details of the Suv with a sunroof object that we have just created.

// Run the methods on the last object that was created.

echo $carWithSunRoof -> description();

echo " costs ";

echo $carWithSunRoof -> cost();

**Result:**  
*Suv, sunroof costs 31500*

Another example involves Suv that we'd like to equip with both a sunroof as well as high end wheels.

// 1. Create an object from one of the basic classes.

$basicCar = new Suv();

// 2. Pass the object from the basic class as a parameter to the first feature class.

$carWithSunRoof = new SunRoof($basicCar);

// 3. Pass the object from the first feature class as a parameter to the second feature class.

$carWithSunRoofAndHighEndWheels = new HighEndWheels($carWithSunRoof);

// 4. Run the methods on the last object that was created.

echo $carWithSunRoofAndHighEndWheels -> description();

echo " costs ";

echo $carWithSunRoofAndHighEndWheels -> cost();

**And the result:**  
*Suv, sunroof, high end wheels costs 33500*

In conclusion

The usefulness of the **decorator design pattern** lies in the fact that we can add new optional classes to existing basic classes without having to change the codes of the latter, thereby helping us to avoid imposing additional responsibilities on classes that were already tested.