
Understanding and Using the PHP SplSubject/SplObserver Classes

Peter Lavin

Revised 2012-11-22

Table of Contents

1. The SplSubject Interface	1
2. SplObserver Interface	1
3. The Observer Class	2
4. The Subject Class	2
5. Main	3
6. Using in a Real Application	4
7. Code Appendix	4
8. About the Author	7

This article looks at the Observer/Subject design pattern using interfaces bundled with PHP version 5.1 and higher. This pattern is sometimes known as a Listener pattern and is useful for facilitating communication between objects—it's a way of informing interested parties when relevant changes have occurred. A simple example of this pattern is implemented to help understand its use.

1. The SplSubject Interface

As implemented in PHP, the Observer/Subject pattern incorporates the SplSubject and SplObserver interfaces. The `spl` prefix standing for "Standard PHP Library". Three methods are defined in the SplSubject interface. They are as follows:

- `public attach (SplObserver $observer)`
- `public detach (SplObserver $observer)`
- `public notify ()`

Since the SplSubject is an interface, all its methods are public and must be defined by any class that implements it.

The purpose of the `attach` and `detach` methods is transparent; they are used to associate observers with the subject or to remove them. Note that these methods use type hinting to ensure that only an observer object can be added or removed.

Through the `attach` method, the subject holds a reference to the observer. The subject can then use the `notify` method to inform observers when there is a significant event or change of state in the subject.

As always with an interface, the details of implementation are left up to the individual programmer.

2. SplObserver Interface

The SplObserver interface is much simpler and defines only one method:

- `public update (SplSubject $subject)`

Again, type hinting ensures that only a class that implements the subject interface can be passed in. The `update` method will be invoked whenever there is an event that is of interest to observers. Since the observer has a reference to the subject, he can query the subject for any change in state.

3. The Observer Class

The observer class is the simpler of the subject/observer pair. In the example code in this article, two observers are created, a doctor and a derived class, the neurosurgeon. Using two observer classes gives better insight into how this design pattern can be used.

3.1. The Doctor Class

The Doctor class represents a general practitioner. It is made up of one data member, the doctor's name, the required observer `update` method and two "magic" methods. If you aren't familiar with PHP's magic methods, they are methods that are invoked indirectly. The `__construct` method is called when an object is created and the `__toString` method when an object is output. The need for a constructor is readily apparent, though it is worth noting that, unlike some other object-oriented languages, a PHP constructor does not bear the name of the class.

While constructors are usually a requirement, the `__toString` method need not have been implemented. It's there as a convenience so that we can easily output the class and keep track of what is occurring. The subject class also implements a `__toString` method for exactly the same reason.

The `update` method is used to determine the patient's health and take appropriate action.

To view the code see [Figure 1, "The SplObserver Class – Doctor" \[4\]](#).

3.2. Neurosurgeon Class

The Neurosurgeon class is even simpler than the Doctor class. This class has one method, the `update` method. Since the Neurosurgeon is a specialized form of the Doctor class it is not surprising that its implementation of the `update` method overrides the implementation of this method in the parent class.

As with any derived class, the behaviour of the derived class is similar but slightly different from the parent class. A Neurosurgeon won't be interested when the patient's state has no bearing on its speciality, so unlike the general practitioner, does not react when the patient gets a cold.

The neurosurgeon has all the attributes of the Doctor class, but overrides the Doctor's implementation of the `update` method. The neurosurgeon only reacts when the patient gets a headache.

To view the code see [Figure 2, "The Derived SplObserver Class – Neurosurgeon" \[5\]](#).

4. The Subject Class

The subject is the more interesting interface given that it must implement three methods. It must implement `attach` and `detach` methods with parameters type hinted to the SplObserver class. It must also have a `notify` method used to inform observers of a change of state.

Given that patients often have more than one doctor, it makes sense to store a reference to each observer in an associative array, declared as a class level variable, in this case the `$observers` variable.

This class is discussed in detail in the following sections but for a complete code listing see [Figure 3, "The SplSubject Class – Patient \(part one\)" \[5\]](#) and [Figure 4, "The SplSubject Class – Patient \(part two\)" \[6\]](#).

4.1. Attaching an Observer

As noted, type hinting the parameter to the `attach` method ensures that an instance of the SplObserver class is passed in. Attempt to pass in a class that does not implement SplObserver and a fatal error will be thrown.

At first glance the way to attach an observer seems quite simple. To ensure that the same observer isn't added twice you might think that the following code would suffice:

```
if(!in_array($o, $this->observers)){
    $this->observers[] = $o;
}
```

Let's see why this approach is inadequate.

4.2. Equality and Identity

The problem with using the `in_array` function is that two objects of the same class with the same attributes will be treated as equal. In terms of the classes used in this example, two doctors with the same name would be treated as the same object. The reason for this is that the `in_array` function tests for equality of objects not for identity, hence the need for a more long-winded approach using the `array_keys` function.

The `array_keys` function returns an array of keys of attached observers. If two doctors with the same attributes have been attached to a patient, it will return both of them. The identity operator, `'==='`, is able to distinguish between objects of the same class with the same attributes.

The `in_array` function might suffice if you were sure that "equal" objects would never be added. However, using `array_keys` and the identity operator is perhaps the more robust implementation.

Identical logic is used for detaching an observer so the `detach` method need not be discussed.

4.3. Other Methods and Attributes of the Patient

As an `SplSubject` the `Patient` class must implement the `notify` method. This method iterates over the array of observers and calls the `update` method of each observer, passing a reference to the patient as an argument. Each observer is guaranteed to have an `update` method because only `SplObservers` can be added to the observers array and any class that implements the `SplObserver` interface must implement this method.

Apart from the magic methods—which function in the same way as the magic methods of the `Doctor` class—the only other methods of the `Patient` class set and get the state of the patient. A serious implementation of a patient class would probably want to use a more robust way of changing the patient's state, perhaps using a `setState` method that stored the patient's condition in an array. However, the two set and get methods used here serve well for simple illustrative purposes.

5. Main

The main portion of the code (see [Figure 5, “Main” \[7\]](#)) shows how these three classes interact. A doctor, a neurosurgeon and a patient are created. The doctor and neurosurgeon are attached to the patient. An attempt to attach the doctor twice outputs a message indicating that the doctor is already an observer, indicating that the `attach` method won't allow the same observer to be attached twice. When the patient gets a cold, the doctor reacts by telling the patient to rest in bed. Since this is not a neurosurgeon's area of expertise, the neurosurgeon has no response. However, when the patient gets a headache, both the doctor and the neurosurgeon respond. Since the neurosurgeon jumps to the questionable conclusion that a mere headache requires "brain surgery", the patient, Ivy, drops him as an observer. When the patient next gets a headache the lack of response from the neurosurgeon shows that he is no longer an observer.

The output is displayed below:

```
Creating Doctor Hardcastle.
Creating Neurosurgeon Cranium.
Creating Patient Ivy.
```


Figure 4. The SplSubject Class – Patient (part two)

```
/**
 * Implement SplSubject detach method
 */
public function detach(SplObserver $o){
    //use the same logic to detach
    $elements = array_keys($this->observers, $o);
    foreach($elements as $value){
        //objects with the same attributes can be distinguished
        //using identity operator
        //but derived class with same attributes will be different
        if($o === $this->observers[$value]){
            unset($this->observers[$value]);
            echo "$o is no longer an observer. <br />";
            break;
        }
    }
}

/**
 * Implement SplSubject method
 */
public function notify(){
    foreach ($this->observers as $value){
        $value->update($this);
    }
}

/**
 * Patient-specific set/get methods
 */
public function setHeadache($bool){
    $this->headache = $bool;
    if ($bool == true){
        echo "$this has a headache.<br />";
    }
    $this->notify();
}
public function getHeadache(){
    return $this->headache;
}
public function setCold($bool){
    $this->cold = $bool;
    if ($bool == true){
        echo "$this has a cold.<br />";
    }
    $this->notify();
}
public function getCold(){
    return $this->cold;
}
}
?>
```

Figure 5. Main

```
<?php
include 'Patient.php';
include 'Doctor.php';
include 'Neurosurgeon.php';
$doctor = new Doctor("Hardcastle");
echo "Creating $doctor.<br />";
$neurosurgeon = new Neurosurgeon("Cranium");
echo "Creating $neurosurgeon.<br />";
$patient = new Patient("Ivy");
echo "Creating $patient.<br />";
$patient->attach($doctor);
//can't attach the same doctor twice
$patient->attach($doctor);
$patient->attach($neurosurgeon);
$patient->setCold(true);
$patient->setCold(false);
//patient gets a headache
$patient->setHeadache(true);
//headache gone
$patient->setHeadache(false);
//get rid of neurosurgeon
$patient->detach($neurosurgeon);
//next headache, only the doctor responds
$patient->setHeadache(true);
?>
```

8. About the Author

Peter Lavin is a technical writer who has been published in a number of print and online magazines. He is the author of [Object Oriented PHP](#), published by No Starch Press and a contributor to [PHP Hacks](#) by O'Reilly Media.

Please do not reproduce this article in whole or part, in any form, without obtaining written permission.

