Practical PHP Refactoring: Introduce Null Object

In the scenario of today, we see repeated checks for an object's equality to null, false or another scalar value without behavior. These checks take a form like !== null and !== false in PHP. These multiple checks are the sign that a relevant case is not modelled with an object: all the logic pertaining to a null value for that role is being spread between its client objects.

Introducing a Null Object specifies clearly a point where to gather all this logic.

Null Object? null isn't an object

The Null Object pattern provides a way for making a null value and a real object indistinguishable: by making them support the same method calls.

You are already using Null Objects all the time. But they aren't objects:

```
$sum = 0;

foreach ($rows as $row) {

      $sum += $row['amount'];

}
```

$sum is a Null Object that supports the sum (null usually can in case of integers due to implicit casts, but not in all languages).

Without it, you would write continuous checks for nullity:

```
$sum = null;

foreach ($rows as $row) {

      if ($sum) {

            $sum += $row['amount'];

      } else {

            $sum = $row['amount'];

      }

}
```

```
$list = array();

foreach ($list as $element) {

      $this->doSomethingWith($element);

}
```

The logic of the Null Object pattern is the same, but **extended to your own types instead of the interpreter's one**. Your classes will have "special case" objects to pass around, instead of plain old arrays having a special case in the empty instance.

Refactoring towards a Null Object is a particular case of refactoring towards polymorphism: providing a new implementation of the contract of a class. This particular case features special semantics, like the **absence of multiple instances of Null Objects**.

A scenario in which this refactoring is badly needed is when the checks for nullity become repeated in different places in the codebase. Null Objects can be a subclass of a concrete class, or an alternate implementation; it's difficult to break the Liskov Substitution Principle with a Null Object. I'll stick with the subclass in the example as it is less invasive.

Steps

1. **Create a subclass** of your concrete class.

2. **Add an isNull() method**, which returns true in the case of the original class and false in its overridden version.

3. When a null or false is returned, **return a new instance of the Null Object.** Usually a Flyweight implementation is enough: all Null Objects of a certain concrete class are equal (otherwise they become real objects, just a different example of polymorphism.)

4. **Find all comparisons and use isNull() instead** of null checks. My alternative to this procedure introduce by Fowler is to use *instanceof NullObjectSubclass* as a rudimental isNull() when you're sure it will be gone by the end of the refactoring. It's not really useful to replace the *=== null* with an equivalent boolean method which will be replaced again shortly.

5. Look for cases where an **operation is invoked only on a real object**, and **move** that code **into the original class**.

6. Look for cases where an **operation is invoked when the object is null**, and **move** it **into the Null Object**. Pass as parameters the depenencies you find.

When you have moved all the code containing operations to execute only in one of the two cases, you can actually remove the conditionals. The refactoring has ended.