Introduce Refactoring By Example

Jan 6, 2015 Advanced php (/tags/php), ood (/tags/ood), refactoring (/tags/refactoring), design pattern (/tags/design pattern), tdd (/tags/tdd)

Refactoring is the key to clean, well-designed, professional code. If you've ever wondered how those well-designed software projects were actually designed, the answer is usually by refactoring.

In this tutorial, I'll show you how Refactoring can improve your code design. We'll build a project from scratch which will start with a poor design, but then in step-by-step manner we'll improve it by applying some Refactoring techniques on it.

If you're new to it, this tutorial will be as an overview on Refactoring, so don't worry if you find some unclear concepts along the way, because more detailed tutorials will likely come in the future. It's just to whet your appetite for using it. However, if you've been using it before, then treat it as an exercise, and maybe you'll get something new out of it.

Although the example will be in PHP, concepts apply on any other OOP language. So you can read it even if you're not a PHP developer.

What is Refactoring?

Here’s the official definition of refactoring:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

In other words, refactoring is used to improve your software design without changing its behavior (logic). Refactoring is applied after your code is written and tested. I mean you’ll start with a crap design, then you’ll use refactoring to convert it into a great one.

Refactoring and testing

In simple words, refactoring can’t be done without testing. And that’s too obvious because how would we know that we’ve not broken our code after we had changed it. Of course we can’t. So our tests will serve as a feedback to our changes. So they tell us if everything still working after modifications.

And remember Refactoring is the third phase of the TDD cycle (RED, GREEN, REFACTOR). So it’s worth mentioning that we’ll use TDD in the example we’re going to build.

The Example

For me, I really find explaining an idea by examples is much better than throwing some concepts that aren’t well visualized. So we’ll build a simple project, then we’ll apply some of the refactoring techniques on it to see how much far can it improve its design.

We’ll build a String Calculator (it’s different than the popular String Calculator Kata), this calculator can take an expression of type string (like this: “2+3+4”) and then parses it and performs the necessary calculation, after that it returns the result. But this calculator is limited to only one type of operation in each calculation. In other words, you can’t mix different types of operations in the same expression like this: “3+4*2”.

Maybe this is so simple to be refactored, but nevertheless I think it would be a great exercise to learn refactoring. Just treat it as a code kata (http://en.wikipedia.org/wiki/Kata_%28programming%29).

So with that, let’s over engineer our awesome calculator. The source code can be found on github (https://github.com/TahaSh/Refactoring-StringCalculator).

Setup the project

Let’s begin by setting up the project’s directory first. Create a new directory, then inside it create a src/ and tests/ directories.

After that, pull in PHPUnit via composer by running this command:

```
composer require phpunit/phpunit --dev
```

And then of course, we have to specify our configurations. So create a phpunit.xml in your project’s root directory, and put this:

```
<phpunit
  colors="true"
  convertErrorsToExceptions="true"
  convertNoticesToExceptions="true"
  convertWarningsToExceptions="true"
  bootstrap="vendor/autoload.php">

  <testsuites>
     <testsuite name="default">
        <directory>tests</directory>
     </testsuite>
  </testsuites>
</phpunit>
```

Finally, we need to register a PSR-4 autoloader (if you're not familiar with it, check out this tutorial (http://taha-sh.com/blog/havent-you-used-composer-yet)). So in our composer.json, write this:

```
{
    "require-dev": {
        "phpunit/phpunit": "~4.4"
    },
    "autoload": {
        "psr-4": {
            "Acme\\": "src"
        }
    }
}
```

Then run `composer dump-autoload` to let composer know about our PSR-4 autoloader. So clearly, our namespace is Acme and it resides in the src/ directory.

Now let's dive in and write our first feature, addition.

Addition

Since this tutorial isn't about testing (there is another dedicated tutorial (http://taha-sh.com/blog/your-first-unit-tests-with-phpunit) on that) I'll be a little faster here.

Now in your tests/ directory create StringCalculatorTest.php and write the following test:

```
<?php

use Acme\StringCalculator;

class StringCalculatorTest extends PHPUnit_Framework_TestCase {

    /**
     * @test
     */
    function it_adds_numbers()
    {
        $calculator = new StringCalculator;

        $result = $calculator->calculate('2+2+2');
        $this->assertEquals(6, $result);

        $result = $calculator->calculate('2 + 2 + 2');
        $this->assertEquals(6, $result);
    }
}
```

So we can see how our calculator works from our tests. Simply, we create a StringCalculator object and we use its calculate() method which we provide it the expression we want to calculate. I included two assertions to make sure that the calculator can calculate expressions regardless of spaces between numbers.

If you run the test, it’ll tell you that you don’t have StringCalculator class yet. So the next step is to create it. So in your src/ create StringCalculator.php and write this:

```
<?php

namespace Acme;

class StringCalculator {

    public function calculate($expression)
    {
        $numbers = preg_split("/[^\d\w\s]/", $expression);
        $numbers = array_map("trim", $numbers);

        preg_match("/\d+\s?+([^\w\d\s])/", $expression, $operation);

        if ($operation[1] === '+')
        {
            return array_reduce($numbers, function($carry, $item) {
                return $carry + $item;
            });
        }
    }
}
```

Now if you run your test, it should pass.

The calculate method works as follows, first it parses the expression (using regular expressions) to extract the operation type (e.g. +) and the numbers to perform the operation on. Then it checks the type of the operation and calculates the result accordingly.

Our first refactor

I think now it's the time for our first refactor. If you've noticed from above, I've used the word parses when I was describing how the calculate() method works. So I think it's a sign for an Extract Method refactoring. So create a new private method in your StringCalculator class, and call it parseExpression($expression), then copy the code portion that responsible for the parsing into it, like this:

```
private function parseExpression($expression)
{
    $numbers = preg_split("/[^\d\w\s]/", $expression);
    $numbers = array_map("trim", $numbers);

    preg_match("/\d+\s?+([^\w\d\s])/", $expression, $operation);

    return [$numbers, $operation[1]];
}
```

Then our calculate() method becomes like this:

```
public function calculate($expression)
{
    list($numbers, $operation) = $this->parseExpression($expression);

    if ($operation === '+')
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry + $item;
        });
    }
}
```

Don’t underestimate how important this is, with that move we made our code reads much better. I mean when you comeback to this method later, you don’t have to spend two minutes thinking about what that regular expression does. All you need to care about is that it parses the expression and extracts the operation and the numbers from it.

After that change, run your test to see if it’s still working. If it’s, then that’s great, otherwise check your code and see what causes that failure (your tests usually help you with that).

Multiplication

Let’s support multiplication operations. Again write the test first (TDD). So in your StringCalculatorTest.php add this:

```
/**
 * @test
 */
function it_multiplies_numbers()
{
    $calculator = new StringCalculator;

    $result = $calculator->calculate('2*2*2');
    $this->assertEquals(8, $result);

    $result = $calculator->calculate('2 * 3 * 4');
    $this->assertEquals(24, $result);
}
```

Multiplication isn't implemented yet, so if you run the tests they'll fail. So let's add that feature. In StringCalculator.php add another check to the calculate() method, like this:

```
public function calculate($expression)
{
    list($numbers, $operation) = $this->parseExpression($expression);

    if ($operation === '+')
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry + $item;
        });
    }
    else if ($operation === '*')
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry * $item;
        }, 1);
    }
}
```

With that change, tests should pass, Great!

Another refactor

The calculate() method still seems so messy, so there is a need for another refactoring. And again we'll use the Extract Method refactoring. So extract the part that is responsible for performing the calculation into a performOperation($type, $number) method.

```
private function performOperation($operation, $numbers)
{
    if ($operation === '+')
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry + $item;
        });
    }
    else if ($operation === '*')
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry * $item;
        }, 1);
    }
}
```

Then modify the calculate() method to this:

```
public function calculate($expression)
{
    list($numbers, $operation) = $this->parseExpression($expression);

    return $this->performOperation($operation, $numbers);
}
```

Run your tests.

Another thing you can do, is to convert the if statement to switch statement. Not a big deal, but I think it makes it clearer that we're checking the value of the same variable. You don't have to do it if you want since we'll change it later, but nevertheless it becomes:

```
private function performOperation($operation, $numbers)
{
    switch ($operation) {
        case '+':
            return array_reduce($numbers, function($carry, $item) {
                return $carry + $item;
            });
            break;
        case '*':
            return array_reduce($numbers, function($carry, $item) {
                return $carry * $item;
            }, 1);
            break;
    }
}
```

Replace operations with symbolic constants

Usually, I don't like to keep literal values as they are in my code. So when I see any literal strings or numbers (Magic Numbers (http://en.wikipedia.org/wiki/Magic_number_%28programming%29)) that my logic depends on I tend to hide them behind a symbolic constant. A symbolic constant is just a simple named class constant.

So in our case we'd like to represent '+' and '*' with ADDITION and MULTIPLICATION constants respectively. So add the two constants to the StringCalculator class, and replace the literals in the performOperation() method with those constants.

```
class StringCalculator {

    const ADDITION = '+';
    const MULTIPLICATION = '*';

    //...

    private function performOperation($operation, $numbers)
    {
        switch ($operation) {
            case static::ADDITION:
                return array_reduce($numbers, function($carry, $item) {
                    return $carry + $item;
                });
                break;
            case static::MULTIPLICATION:
                return array_reduce($numbers, function($carry, $item) {
                    return $carry * $item;
                }, 1);
                break;
        }
    }
}
```

With that change, we're no longer tied to literal values. And that's so useful, because imagine for any reason if the operation's symbol for multiplication was changed to X instead of the asterisk, then all we have to do is to change the constant value instead of replacing each literal value.

Now run your tests to make sure that everything still works.

Replace Conditional with Polymorphism

One of the greatest refactorings is "Replace Conditional with Polymorphism". Which states that "when you have a conditional that chooses different behavior depending on the type of an object, you should move each leg of the conditional to an overriding method in a subclass, then make the original method abstract".

If this seems confusing to you, don't worry here's a simpler one. When we have a certain behavior (in this case performing the operation) that depends on the type of an object (in this case the

operation's type: ADDITION, MULTIPLICATION), then we should move each behavior to a certain class (each behavior has its own class), and each class extends a superclass that has an overriding method (for example: perform() method). So each class will override that method and fill it in with its own implementation (e.g. the Addition must perform the addition operation). And that superclass should not be instantiated directly (i.e. abstract class).

If it's still confusing, code explains better.

According to the above explanation, we have to create three classes. Operation (abstract), Addition and Multiplication.

Let's create the Operation class first, so in your src/ create a new directory named Operations/ which will hold all classes related to performing operations. Then inside it, create Operation.php which will be the abstract class that other classes extend from. Write this into it:

```
<?php

namespace Acme\Operations;

abstract class Operation {

    abstract function perform($numbers);
}
```

Then create the Addition class. So in your src/Operations/ directory create Addition.php, and put this into it:

```
<?php

namespace Acme\Operations;

class Addition extends Operation {

    function perform($numbers)
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry + $item;
        });
    }
}
```

Because it extends the Operation class, and because there is an abstract method in it, which is perform(), we have to override it with the appropriate implementation, in this case, we put the code of addition into it.

Do the same for multiplication. So create Multiplication.php in src/Operations, and write this:

```
<?php

namespace Acme\Operations;

class Multiplication extends Operation {

    function perform($numbers)
    {
        return array_reduce($numbers, function($carry, $item) {
            return $carry * $item;
        }, 1);
    }
}
```

Now it's time to use them. And obviously, we've done all of this to change the implementation of the performOperation() in the StringCalculator class. So the first step is to remove everything within that method. Then we have to create the correct object that performs the correct operation. But how to decide which one should be created? When you're in a situation like this, go for the factory design pattern. And in this case we'll apply it in its most basic

form. We'll use a static method that takes the operation's type and returns the correct object accordingly. And the best place for this factory is within the Operation abstract class.

So your Operation class becomes like this:

```
abstract class Operation {

    public static function make($type)
    {
        switch ($type) {
            case \Acme\StringCalculator::ADDITION:
                return new Addition;
                break;
            case \Acme\StringCalculator::MULTIPLICATION:
                return new Multiplication;
                break;

            default:
                throw new \Exception('Not supported Operation');
                break;
        }
    }

    abstract function perform($numbers);
}
```

So that make() method uses a switch statement to determine which is the correct object to return. However, if it didn't find anything (which means there's no matching operation), it'll throw an exception telling that the operation isn't supported.

To get everything back to work, the final step is to replace the implementation of the performOperation() method with this:

```
private function performOperation($operation, $numbers)
{
    $operation = \Acme\Operations\Operation::make($operation);
    return $operation->perform($numbers);
}
```

Look at how simple it is now! Believe it or not, from now on, we don't have to touch this method again even if we want to add a new operation. Thanks to Polymorphism

(http://en.wikipedia.org/wiki/Polymorphism_%28computer_science%29).

If everything was done right, your tests should pass.

Move the operations constants to their right place

Another thing you'll usually notice while you're refactoring your code, is that there are some data (fields or constants) used by methods in other class more than the ones it's actually in. Sometimes it's called the Feature Envy code smell. In our case we can find this with the operations constants (i.e. ADDITION, MULTIPLICATION). Because notice that they're not referenced anymore in their current class StringCalculator, however they're used by the Operation class, so we conclude that it's better to move them to that class.

So move the constants from the StringCalculator class to the Operation class, then update the references in the make() static method. So your Operation class becomes like this:

```
<?php

namespace Acme\Operations;

abstract class Operation {

    const ADDITION = '+';
    const MULTIPLICATION = '*';

    public static function make($type)
    {
        switch ($type) {
            case static::ADDITION:
                return new Addition;
                break;
            case static::MULTIPLICATION:
                return new Multiplication;
                break;

            default:
                throw new \Exception('Not supported Operation');
                break;
        }
    }

    abstract function perform($numbers);
}
```

Run your tests.

Use a custom exception

Usually in my applications I like to use custom exceptions instead of the general exception with message (i.e. \Excepition). Because creating custom exceptions for each error case is always considered a best practice, and the big advantage is that you'll have a named exception that is specific for a certain error, so we're no longer tied to the error message which is likely to change. And even though in most cases all you have to do is to just extend that general exception, you'll also have the ability to add any features you want to it.

So create a new directory called Exceptions/ within src/ which will hold all of our custom exceptions. Our exception will be called UnsupportedOperationException so in that Exceptions/ directory

create UnsupportedOperationException.php, and put this into it:

```
<?php

namespace Acme\Exceptions;

class UnsupportedOperationException extends \Exception {}
```

Notice all we've done is just extending the general Exception class, and that's enough for our case.

Now to use it, all you have to do is to replace the general exception in the Operation class with this:

```
default:
    throw new UnsupportedOperationException;
    break;
```

Of course don't forget to use it at the top of the file:

```
use Acme\Exceptions\UnsupportedOperationException;
```

Run your tests, everything should pass.

Now I think we have something to write a test for. And that's the exception, because I think it's important to check that this exception would be thrown if the operation isn't supported. So in your StringCalculatorTest.php add this test:

```
/**
 * @test
 * @expectedException Acme\Exceptions\UnsupportedOperationException
 */
function it_disallows_unsupported_operations()
{
    $calculator = new StringCalculator;

    $result = $calculator->calculate('3%3');
}
```

Notice we've used annotations to test the expected exception. Run your tests, they should pass (notice we haven't used TDD in this

case, I've done that on purpose to tell you that it's okay to write the implementation first in some cases, especially when you know what your feature is about).

Extract a parser class

If you view your StringCalculator class, you'll notice that we are violating the single responsibility principle (http://en.wikipedia.org/wiki/Single_responsibility_principle), which states that each class should have only one reason to change. And our class has two responsibilities one for performing the operation and the other for parsing. And it's clear that parsing isn't its responsibility, so it's better to have a dedicated class for that. Let's call it ExpressionParser. So in your src/ directory create a file named ExpressionParser.php, then put this into it:

```php
<?php

namespace Acme;

class ExpressionParser {

    private $operation;
    private $numbers;

    public function parse($expression)
    {
        $this->operation = $this->extractOperation($expression);

        $this->numbers = $this->extractNumbers($expression);

        return $this;
    }

    public function getOperation()
    {
        return $this->operation;
    }

    public function getNumbers()
    {
        return $this->numbers;
    }

    private function extractOperation($expression)
    {
        preg_match("/\d+\s?+([^\w\d\s])/", $expression, $operation);
        return $operation[1];
    }

    private function extractNumbers($expression)
    {
        $numbers = preg_split("/[^\d\w\s]/", $expression);
        return array_map("trim", $numbers);
    }
}
```

Then inject it into the constructor of the StringCalculator class. So in your StringCalculator, create a constructor like this:

```
class StringCalculator {

private $parser;

function __construct(ExpressionParser $parser)
{
    $this->parser = $parser;
}
//...
```

Then remove the parseExpression() method from it. Then modify the calculate() method to use our parser class, like this:

```
public function calculate($expression)
{
    $operation = $this->parser->parse($expression)->getOperation();
    $numbers = $this->parser->parse($expression)->getNumbers();

    return $this->performOperation($operation, $numbers);
}
```

Then update your tests to use that parser. But before you do that, it's better to move the instantiation of the StringCalculator class to the setup() method (which is executed before each test case) first, then inject the parser into it. So your StringCalculatorTest.php becomes like this:

```
<?php

use Acme\StringCalculator;
use Acme\ExpressionParser;

class StringCalculatorTest extends PHPUnit_Framework_TestCase {

    function setup()
    {
        $this->calculator = new StringCalculator(new ExpressionParser);
    }

    /**
     * @test
     */
    function it_adds_numbers()
    {
        $result = $this->calculator->calculate('2+2+2');
        $this->assertEquals(6, $result);

        $result = $this->calculator->calculate('2 + 2 + 2');
        $this->assertEquals(6, $result);
    }

    /**
     * @test
     */
    function it_multiplies_numbers()
    {
        $result = $this->calculator->calculate('2*2*2');
        $this->assertEquals(8, $result);

        $result = $this->calculator->calculate('2 * 3 * 4');
        $this->assertEquals(24, $result);
    }

    /**
     * @test
     * @expectedException Acme\Exceptions\UnsupportedOperationException
     */
    function it_disallows_unsupported_operations()
    {
        $result = $this->calculator->calculate('3%3');
    }

}
```

If everything is OK, your tests should pass.

The result

A good question to ask, “is what we’ve done considered a good design?” Well, we can answer it by measuring its goodness by some well-known metrics. For example, I always like to use the SOLID (http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29) principles as metrics to my design. For example, does our design follow the single responsibility principle? We can see that it does by looking at each class and method we have, everything has only one responsibility (one reason to change).

Also if you know the open-closed principle, you can see that our design is open for extension and closed for modification, we can see that when we want to support a new operation. And to prove that, let’s add subtraction to our calculator.

Another feature: Subtraction

As usual start with the test, so add this into your StringCalculatorTest.php:

```
/**
 * @test
 */
function it_subtracts_numbers()
{
    $result = $this->calculator->calculate('5 - 3 - 1');
    $this->assertEquals(1, $result);
}
```

Run it, it’ll fail. To make it pass, we have to do the following steps:

1. Create a new operation named Subtraction in your Operations/ directory.
2. Make it extends the Operation abstract class, then implement the perform() method.
3. Add a constant in the Operation class to represent it.
4. Add the Subtraction class to the factory list.

So let’s tackle each step. Create Subtraction.php in your Operations/ directory, then write this:

```
<?php

namespace Acme\Operations;

class Subtraction extends Operation {

    function perform($numbers)
    {
        $initial = array_shift($numbers);

        return array_reduce($numbers, function($carry, $item) {
            return $carry - $item;
        }, $initial);
    }
}
```

Then in your Operation class, add this constant: const SUBTRACTION = ‘-’ ;, then add it to the factory list, so your Operation class should look like this:

```
<?php

namespace Acme\Operations;

use Acme\Exceptions\UnsupportedOperationException;

abstract class Operation {

    const ADDITION = '+';
    const MULTIPLICATION = '*';
    const SUBTRACTION = '-';

    public static function make($type)
    {
        switch ($type) {
            case static::ADDITION:
                return new Addition;
                break;
            case static::MULTIPLICATION:
                return new Multiplication;
                break;
            case static::SUBTRACTION:
                return new Subtraction;
                break;

            default:
                throw new UnsupportedOperationException;
                break;
        }
    }

    abstract function perform($numbers);
}
```

Run your tests, everything should pass.

With that, we're finished with our refactoring journey. Congratulations!

Conclusion

I hope you can now see how powerful Refactoring is. With Refactoring we can convert a bad designed (legacy) code, into a well-designed professional code. Notice how easy it becomes to deal with our code after refactoring. Also our code reads much better than before. So evidently, Refactoring is a great thing to master.

As I’ve mentioned at the beginning, this tutorial is intended to give you an overview of Refactoring and how