Object Oriented PHP

Junade Ali

Object-Oriented PHP

Writing Resilient & Reusable Code in PHP 7

Junade Ali

This book is for sale at http://leanpub.com/object-orientedphp

This version was published on 2017-06-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

SOLID Design Principles

There are many PHP Developers who will write "OOP" on their resumes, but they will understand little more than how to write classes and instantiate them. So far in this book we've covered some more advanced language features that help us write Object-Oriented code, however you still may be scratching your head wondering what purpose they are for. The concept of interfaces, for example, confuses many developers; why would you write classes which contain no body?

This chapter seeks to go into the fundamental principles behind Object-Oriented Programming in a clear and concise way. These principles are often referred to as the "First Five Principles of Object-Oriented Programming", or the SOLID Principles. The acronym of "SOLID" was coined by Robert C. Martin based on what Michael Feathers described as the "First Five Principles".

Single Responsibility Principle

Robert C. Martin expressed this principle quite simply: "A class should have only one reason to change."

Consider you're making a PHP app to calculate airline miles from flight information. After building this project, what are the possible reasons we'd want to change the program after it's built? Well, maybe we want to change the algorithm for generating the points, or maybe the UI for displaying the calculation. These are fundamentally our "reasons to change" and therefore we consider them to be responsibilities under the SRP (Single Responsibility Principle).

Every module or class should only be responsible of one piece of functionality and it should fully encapsulate that responsibility. Within the context of the SRP we consider a responsibility to be "a reason to change". By combining responsibilities we can cause some nasty issues, such as tight-coupling and making code-reuse harder.

Let's assume a project manager has given us an airline miles calculator app to refactor, and it doesn't match this principle. Let's refactor it so it does. The code have been given contains a single `Miles` class which has both the HTML that contains the User Interface and the business logic that runs the points calculation:

```php
<?php

class Miles
{
    public function render(): string
    {
        if (isset($_POST['distance'])) {
            return $this->getMiles();
        }

        return $this->getForm();
    }

    public function calculateMiles(int $distance, bool $businessClass, bool $fly\
ingClubMember): int
    {
        $multiplier = 1;

        if ($businessClass === true) {
            $multiplier *= 2;
        }

        if ($flyingClubMember === true) {
            $multiplier *= 2;
        }

        return $distance * $multiplier;
    }

    private function getMiles(): string
    {
        $miles = $this->calculateMiles(
            $_POST['distance'],
            isset($_POST['businessclass']),
            isset($_POST['flyingclubmember'])
        );

        return $this->loadPage('<p>You have: <b>' . $miles . ' miles</b>.</p>');
    }

    private function getForm(): string
    {
```

```
        return $this->loadPage('
<form action="" method="POST">
  Distance:
  <input type="number" name="distance" min="0" step="1" />
  <br>
  Business Class Flyer:
  <input type="checkbox" name="businessclass"><br>
  Flying Club Member:
  <input type="checkbox" name="flyingclubmember">
  <br><br>
  <input type="submit" value="Submit">
</form>
        ');
    }

    private function loadPage(string $html): string
    {
        return '
<!DOCTYPE html>
<html>
  <body>
    ' . $html . '
  </body>
</html>
        ';
    }
}
```

Our business logic is horribly intertwined with HTML, the `Miles` class conjoins our calculator logic with the code that renders the web page meaning the class is responsible for both the presentation and calculation of reward miles. By separating presentation from our business logic we can better prevent duplicate code and ensure our class supplies with the Single Responsibility Principle.

We can utilise this class with the following `index.php` file:

```
<?php

require_once('Miles.php');

$calculator = new Miles();
echo $calculator->render();
```

This makes code reuse more difficult, a developer might not want to pull in a bloated class into another part of the application just to do a simple points calculation. When a class does more and

more, as developers are reluctant to add classes to a project, this leads to an Anti-Pattern known as the "God Class". An irrational fear of adding classes to a project converges to there being one "God Class" which has multiple responsibilities throughout the application. Functions with different responsibilities become tightly bound to each other, and the code becomes a ball of mud.

Now let's refactor this `Miles` class so we have a `MilesCalculator` class containing the business logic and a `MilesUI` class containing the GUI. Our UI class looks like this:

```
<?php

class MilesUI
{
    private $calculator;

    public function __construct(Calculator $calculator)
    {
        $this->calculator = $calculator;
    }

    public function render(): string
    {
        if (isset($_POST['distance'])) {
            return $this->getMiles();
        }

        return $this->getForm();
    }

    private function getMiles(): string
    {
        $miles = $this->calculator->calculate(
            $_POST['distance'],
            isset($_POST['businessclass']),
            isset($_POST['flyingclubmember'])
        );

        return $this->loadPage('<p>You have: <b>' . $miles . ' miles</b>.</p>');
    }

    private function getForm(): string
    {
        return $this->loadPage('
<form action="" method="POST">
```

```
  Distance:
  <input type="number" name="distance" min="0" step="1" />
  <br>
  Business Class Flyer:
  <input type="checkbox" name="businessclass"><br>
  Flying Club Member:
  <input type="checkbox" name="flyingclubmember">
  <br><br>
  <input type="submit" value="Submit">
</form>
        ');
    }

    private function loadPage(string $html): string
    {
        return '
<!DOCTYPE html>
<html>
  <body>
    ' . $html . '
  </body>
</html>
        ';
    }
}
```

Note that our UI class contains some HTML, this is acceptable when well managed - but it sometimes can be easier to use a template engine to manage HTML, one such engine is Twig[1]. A more optimal solution is simply to have PHP serve an API which a Javascript framework like React or Vue.js can consume. Regardless of how you chose to do it, it's vital the UI class is separate from your business logic, in this example we're ensuring that the UI is managed by a separate class.

When instantiating this UI class, we inject a `Calculator` interface - this interface looks like this:

[1] http://twig.sensiolabs.org/

```php
<?php

interface Calculator
{
    public function calculate(int $distance, bool $businessClass, bool $flyingCl\
ubMember): int;
}
```

This interface is then implemented in our `MilesCalculator` class, this is our concrete implementation of our `Calculator` interface that we can use with our `MilesUI` class:

```php
<?php

class MilesCalculator implements Calculator
{
    public function calculate(int $distance, bool $businessClass, bool $flyingCl\
ubMember): int
    {
        $multiplier = $this->getMultiplier($businessClass, $flyingClubMember);

        return $distance * $multiplier;
    }

    private function getMultiplier(bool $businessClass, bool $flyingClubMember):\
 int
    {
        $multiplier = 1;

        if ($businessClass === true) {
            $multiplier *= 2;
        }

        if ($flyingClubMember === true) {
            $multiplier *= 2;
        }

        return $multiplier;
    }
}
```

Our index.php file looks much the same as it did earlier, except we inject an instance of `MilesCalculator` into `MilesUI`:

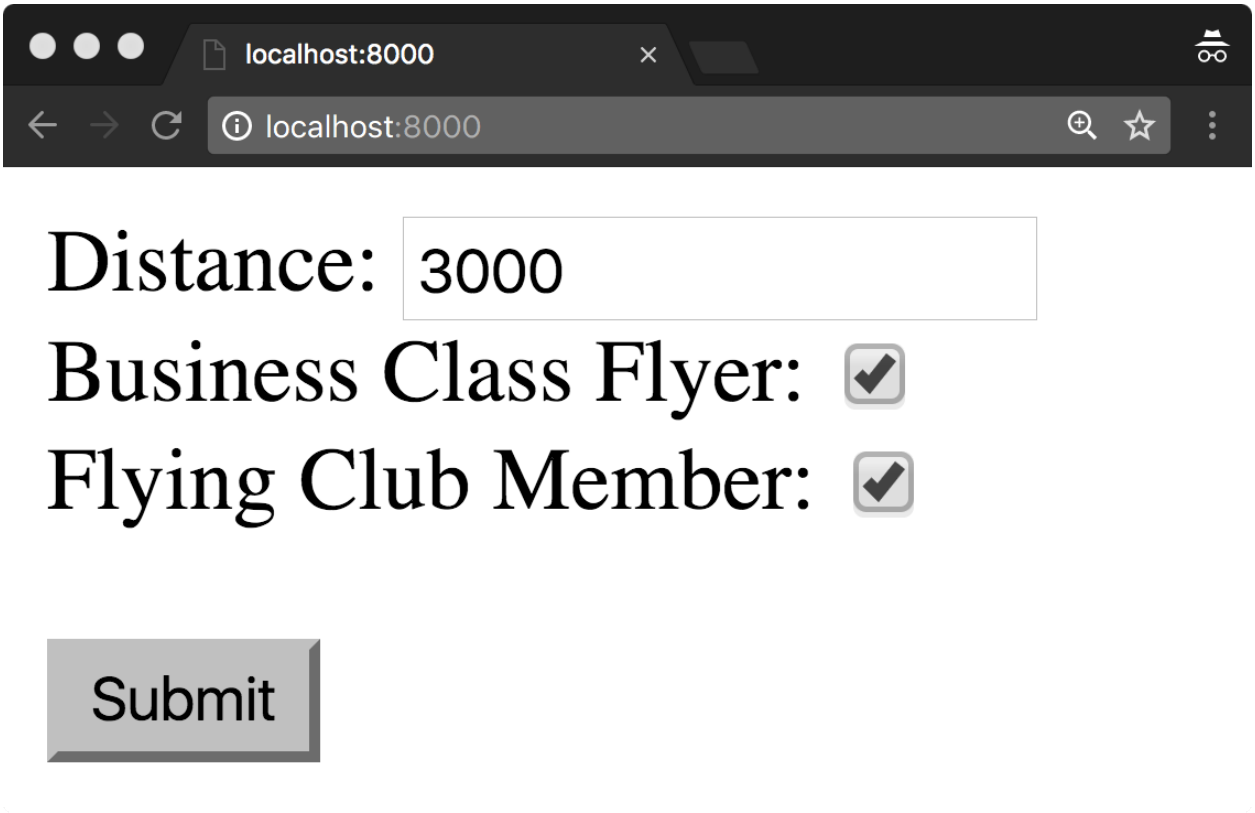
```php
<?php

require_once('MilesUI.php');
require_once('Calculator.php');
require_once('MilesCalculator.php');

$calculator = new MilesCalculator();

$ui = new MilesUI($calculator);
echo $ui->render();
```

Notice that so far in this book, we've been testing code snippets by running them from the terminal? In this example let's do things slightly differently and use the in-built development server in PHP by running this command from the terminal:

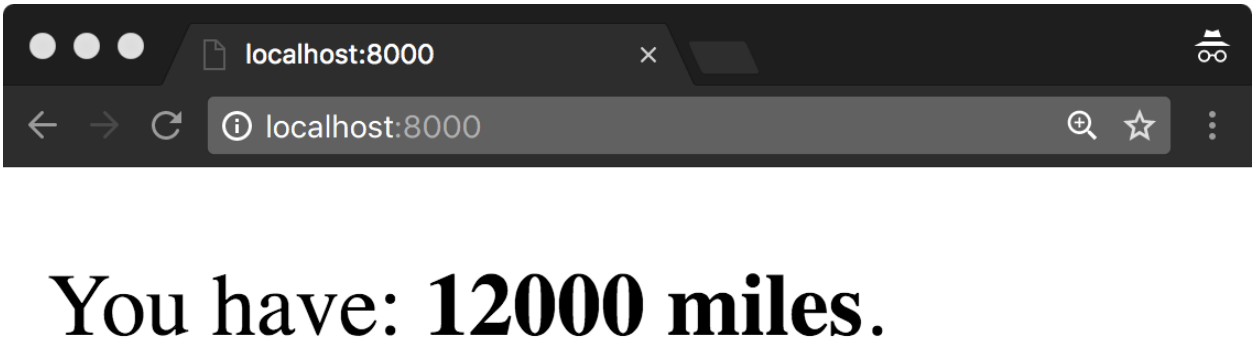
```
php -S localhost:8000
```

The server will then spin into action and we can access webpage at http://localhost:8000:

Let's fill out the form with a 3000 mile business class flight with Flying Club membership:



With the form filled out, we should then get a result of 12000 reward miles:



The Single Responsibility Principle can be violated in many other ways; if we had a system for employee data with the underlying data being stored in JSON, we should avoid creating a class which simultaneously contains business logic for the employee data whilst also having the class hold the logic for dealing with the persistence layer (storing and managing the JSON in the file system).

The Single Responsibility Principle is one of the most core principles to writing Object-Oriented code and it should be something you start to do naturally by considering the axis for change for each class.

Open/Closed Principle

The Open/Closed Principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification".

Simply, this means that we should be able to extended a given piece of software without needing to modify it's source code. In the context of OOP, this means we should be able to extend a classes behaviour without needing to modifying it. This fundamentally comes down to two core attributes:

- "Open for Extension" - we can make the class behave in new ways as the requirements for what the class needs to do evolves
- "Closed for Modification" - you cannot change the source code of the class itself, it is considered inviolable

This principle is said to be originally coined by the French software engineer, Bertrand Meyer, in his 1988 book *Object Oriented Software Construction*.

Whilst these principles may seem at odds with each other, they aren't. Let me demonstrate how we can extend a class using the Object Decorator Pattern. Imagine our `MilesCalculator` class from earlier needs to be extended for a new MilesPlus program. With a MilesPlus credit card, Flying Club members in Business class get a 1500 mile bonus for each flight - if these criteria aren't met the standard calculator applies.

Our `Calculator` interface is the same as before:

```
1  <?php
2
3  interface Calculator
4  {
5      public function calculate(int $distance, bool $businessClass, bool $flyingCl\
6  ubMember): int;
7  }
```

The same goes for our `MilesCalculator` class from before:

```php
<?php

class MilesCalculator implements Calculator
{
    public function calculate(int $distance, bool $businessClass, bool $flyingCl\
ubMember): int
    {
        $multiplier = $this->getMultiplier($businessClass, $flyingClubMember);

        return $distance * $multiplier;
    }

    private function getMultiplier(bool $businessClass, bool $flyingClubMember):\
 int
    {
        $multiplier = 1;

        if ($businessClass === true) {
            $multiplier *= 2;
        }

        if ($flyingClubMember === true) {
            $multiplier *= 2;
        }

        return $multiplier;
    }
}
```

Our `MilesPlusCalculator` is where things start to get interesting, we are treating this like the `MilesCalculator` by implementing the `Calculator` interface:

```php
<?php

class MilesPlusCalculator implements Calculator
{
    private $milesCalculator;

    public function __construct()
    {
        $this->milesCalculator = new MilesCalculator();
    }
```

```
11
12      public function calculate(int $distance, bool $businessClass, bool $flyingCl\
13  ubMember): int
14      {
15          $miles = $this->milesCalculator->calculate($distance, $businessClass, $f\
16  lyingClubMember);
17
18          if ($businessClass === true && $flyingClubMember === true) {
19              $miles += 1500;
20          }
21
22          return $miles;
23      }
24  }
```

When the new class is instantiated, it creates a private instance of the `MilesCalculator` class and stores it in `$milesCalculator` for when it needs it. When the calculation actually takes place, the `calculate()` method uses the `$milesCalculator` instance to find what the old value should be then if the criteria for the bonus miles are met - the method adds them too.

Liskov Substitution Principle

In it's simplified form: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it".

Fundamentally this principle means that any class should be substitutable for it's base class or interface; if we have an interface called `Shape` and a class called `Square`, we should be able to replace any instance of `Square` with another object that meets the `Shape` interface and it should work as normal. This definition of a subtype relation was named after Barbara Liskov in a 1987 conference keynote address titled *Data abstraction and hierarchy*.

Suppose we have an abstract class called `Staff` (this could just as easily be an interface or a normal class), to this we're going to add two concrete implementations of this in `Manager` and `Executive`:

```php
<?php

abstract class Staff
{
    protected $baseSalary;
    private $paid;

    public function __construct(double $baseSalary)
    {
        $this->baseSalary = $baseSalary;
    }

    public abstract function getWeeklyHours(): double;

    public abstract function getSalary(): double;

    public function pay(double $bonus): bool
    {
        $this->paid += $this->getSalary() + $bonus;

        return true;
    }
}
```

The first concrete implementation we'll build will be `Executive`:

```php
<?php

class Executive extends Staff
{
    public function getWeeklyHours(): double
    {
        return 37.5;
    }

    public function getSalary(): double
    {
        return $this->baseSalary;
    }
}
```

Similarly, we can build another concrete implementation called `Manager`:

```php
<?php

class Manager extends Staff
{
    public function getWeeklyHours(): double
    {
        return 40;
    }

    public function getSalary(): double
    {
        return $this->baseSalary * 1.2;
    }
}
```

However; now suppose we want to add an `UnpaidIntern` class - we will quickly find that the `pay()` would end up needing to be overwritten to do absolutely nothing:

```php
<?php

class UnpaidIntern
{
    public function getWeeklyHours(): double
    {
        return 35;
    }

    public function getSalary(): double
    {
        return 0;
    }

    public function pay(double $bonus): bool
    {
        return false;
    }
}
```

Our `UnpaidIntern` class is extending our `Staff` class but changing the behaviour of the underlying `Staff` abstract method so it breeches this principle. Suppose we then add an additional method to our `UnpaidIntern` class, this poses another breech of the principle:

```
public function makeCoffee(): bool
{
    return true;
}
```

As we have added a public method which doesn't exist in the original interface, we introduce a situation whereby instances of the `UnpaidIntern` class cannot be substituted for other instances of the `Staff` abstract class.

Another more hypothetical example that has been coined before is the example of a `Rectangle` class which mandates a `setWidth()` and a `setHeight()` method. When we extend this class to a `Square` class mandating both a `setWidth()` and a `setHeight()` method no longer makes sense. You change one property, and the other must change - all sides must be equal in a square. It is a mathematical impossibility to independently change the `$width` and the `$height` property when we talk about a square. Therefore the `Square` inheriting a `Rectangle` class fails the Liskov Substitution Test as you cannot ensure they're both the same without having a check after the property has been changed to make sure both the `$width` and the `$height` property are the same.

Interface Segregation Principle

The Interface Segregation Principle states that no client should be forced to depend on methods it does not use.

This principle essentially outlines that we should favour small, specific interfaces over large bloated ones. All classes should only have to implement the methods they need - this helps keep our system decoupled. I'll demonstrate this with a simple example of a `CardReader` interface:

```
<?php

interface BadCardReader
{
    public function __construct(string $cardNumber, string $expiry, string $pin);

    public function withdraw(double $amount): bool;

    public function deposit(double $amount): bool;
}
```

In order for a `ATM` to implement this interface, it has to have a `deposit()` method, even if it might only be used for withdrawing money from accounts with the `withdraw()` method. Similarly a `DepositKiosk` class might only need the `deposit()` method but it has to contain the `withdraw()` method.

Let's simplify the interface down so that the new `CardReader` interface contains the constructor and a method which generates the security auth tokens required to check balances, perform withdrawals and add deposits. We can leave the logic which actually does that behaviour in a separate class which satisfies a separate interface. In short we are closely aligning our `CardReader` interface to a single responsibility and in doing so we ensure that classes implementing it don't have to depend on methods they don't use:

```
<?php

interface CardReader
{
    public function __construct(string $cardNumber, string $expiry, string $auth\
Code);

    public function getAuthCode(): double;
}
```

We can then build an interface for `CashDispenser` which our `ATM` class will later be able to use:

```
<?php

interface CashDispenser
{
    public function __construct(CardReader $reader);

    public function withdraw(double $amount): bool;
}
```

By injecting an instance of `CardReader` into the `CashDispenser` we are able to use the `getAuthCode()` method from the `CashDispenser()` method. In a similar light, we are are also able to build an interface for our `DepositKiosk` using the `CashDepository` interface:

```
<?php

interface CashDepository
{
    public function __construct(CardReader $reader);

    public function deposit(double $amount): bool;
}
```

We now have two specific interfaces, meaning our `CashDispenser` interface isn't bloated and ensuring our classes are decoupled from each other. Interface Bloat is an anti-pattern by which interfaces become so large that code which has different responsibilities becomes tied together in one interface. By abiding by the Interface Segregation Principle we help ensure Interface Bloat is prevented and our code remains decoupled well.

Dependency-Inversion Principle

The Dependency-Inversion Principle is stated in two parts:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

When developing software it is common for us to have high-level classes which need to depend on low-level classes. We can have a `Users` class which depends upon a `Database` class. In Chapter 3 we actually looked at an example of using a class to read from a JSON file, I'll use a truncated versioned of that example here. We start with a JSON file that looks like this:

```
[
  {
    "text": "junade.com",
    "type": "link"
  },
  {
    "text": "It's raining today.",
    "type": "text"
  },
  {
    "text": "icyapril.com",
    "type": "link"
  },
  {
    "text": "Hello world!",
    "type": "text"
  }
]
```

Accordingly we have a simple class to read the JSON file (our low-level module):

```php
<?php

class JSON
{
    private $data;

    public function __construct(string $file)
    {
        $this->processFile($file);
    }

    private function processFile(string $file)
    {
        $contents     = file_get_contents($file);
        $array        = json_decode($contents);
        $arrayReverse = array_reverse($array);
        $this->data   = $arrayReverse;
    }

    public function getData(): array
    {
        return $this->data;
    }

    public function getDataByType(string $type): array
    {
        $result = [];
        $data   = $this->getData();

        foreach ($data as $entry) {
            if ($entry->type === $type) {
                array_push($result, $entry);
            }
        }

        return $result;
    }
}
```

This low-level module is in turn consumed by our high-level module (our Link class):

```php
<?php

class Link
{
    private $data;

    public function __construct(JSON $data)
    {
        $this->data = $data;
    }

    public function getContent(): Generator
    {
        $links = $this->data->getDataByType('link');

        foreach ($links as $link) {
            yield $link->text;
        }
    }
}
```

There is, however a fairly major issue in this `Link` class - what if we want to read from an XML file instead of a JSON file? Our current `Link` class makes explicitly clear that we are injecting a concrete `JSON` class instead of a generic `File` interface. In order to comply with the Dependency-Inversion Principle we need to build a `File` interface for our `JSON` file:

```php
<?php

interface File
{
    public function __construct(string $file);

    public function getData(): array;

    public function getDataByType(string $type): array;
}
```

Let's go ahead and amend our `JSON` class to implement this:

```
<?php

class JSON implements File
...
```

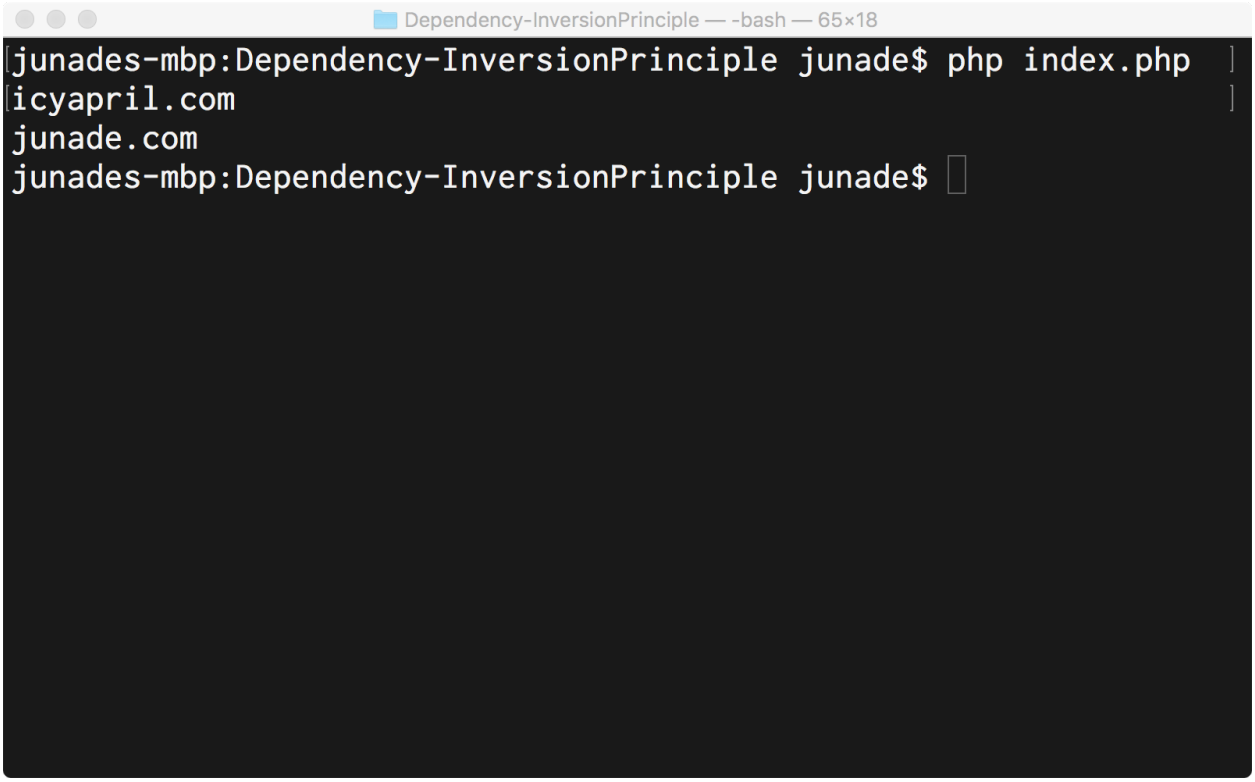
With this in place our constructor for our `Link` class can be refactored so it type hints for the `File` interface instead of the concrete `JSON` class.

```
public function __construct(File $data)
```

Finally, our `index.php` file demonstrates how this can all be put together:

```
<?php

require_once('File.php');
require_once('JSON.php');
require_once('Link.php');

$data = new JSON('data.json');
$link = new Link($data);

foreach ($link->getContent() as $content) {
    echo $content. "\n";
}
```

The script outputs the links we desired; the added benefit is now that should we want to replace our low-level `JSON` class, we can do that so long as the `File` interface is met:



Conclusion

In this chapter we've discussed the SOLID Design Principles and how they can help make sure you write great code; ensuring your code is decoupled, extendable and will be easy to test later down the road. There are some things here which go unsaid; you still need to ensure your code is descriptively named and that you aren't repeating code.

In the next chapter we'll start to discuss Design Patterns which provide effective but reusable solutions to common programming problems.