SOLID pill

3/4/2020

Overview

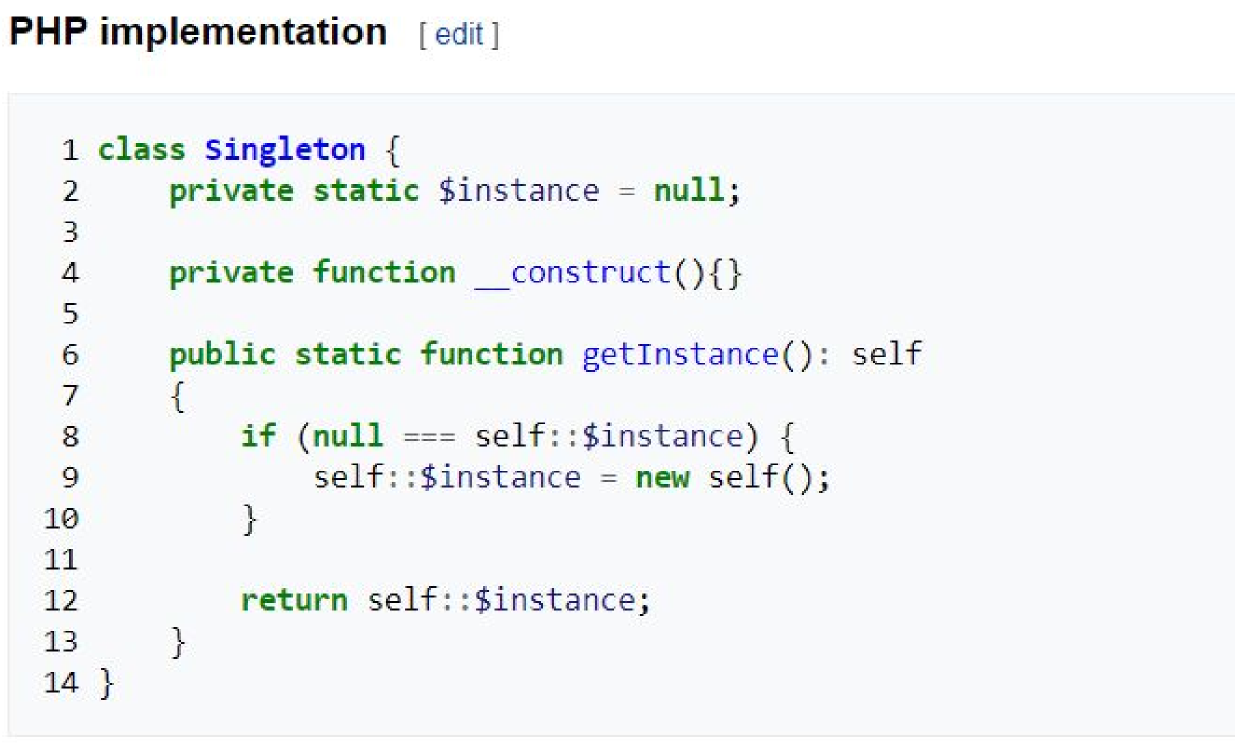
This pill is designed to improve our understanding of object oriented programing and specifically the SOLID principles. To clarify all the detailed concepts of OOP the following questions are answered:

STUPID (why to avoid it in programming)

- What is S-Singleton ?
  - In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. The term comes from the mathematical concept of a singleton.

    The singleton design pattern solves problems like:
    - How can it be ensured that a class has only one instance?
    - How can the sole instance of a class be accessed easily?
    - How can a class control its instantiation?
    - How can the number of instances of a class be restricted?

    The singleton design pattern describes how to solve such problems:
    - Hide the constructor of the class.
    - Define a public static operation (getInstance()) that returns the sole instance of the class.



PHP implementation [edit]

```php
class Singleton {
    private static $instance = null;

    private function __construct(){}

    public static function getInstance(): self
    {
        if (null === self::$instance) {
            self::$instance = new self();
        }

        return self::$instance;
    }
}
```

- What is T-tight coupling ?
  - Tight coupling, also known as strong coupling, is a generalization of the Singleton issue. Basically, you should reduce coupling between your modules. Coupling is the degree to which each program module relies on each one of the other modules.
    If making a change in one module in your application requires you to change another module, then coupling exists. For instance, you instantiate objects in your constructor's class instead of passing instances as arguments. That is bad because it doesn't allow further changes such as replacing the instance by an instance of a sub-class, a mock or whatever.

    Tightly coupled modules are difficult to reuse, and also hard to test.

    ```php
    class House {
      public function __construct() {
        $this->door   = new Door;
        $this->window = new Window;
      }
    }
    ```

    Flexible coupling

    ```php
    class House {
      public function __construct(Door $door, Window $window) { // Door, Window are interfaces
        $this->door   = $door;
        $this->window = $window;
    ```

```
    }
}
```

- What is an U-untestabiliity ?
    - Unit testing is important. If you don't test your code, you are bound to ship broken code. But still most people don't properly cover their code with tests. Why? Mostly because their code is hard to test. What makes code hard to test? Mainly the previous point: Tight coupling. Unit testing - it might seem obvious - tests units of code (usually classes). But how can you test individual classes if they are tightly coupled? You probably can somehow, with even more dirty hacks. But people usually don't do those efforts and just leave their code untested and broken.

      Whenever you don't write unit tests because you "don't have time" the real cause probably is that your code is bad. If your code is good you can test it in no time. Only with bad code unit testing becomes a burden.

- What is an P-premature optimization ?
    - Don't waste your time trying to optimize parts of the codes that do not improve the overall software performance.

      ```
      if (strlen($frm['title_german']) == strcspn($frm['title_german'], '<>')) {
          // ...
      }
      ```

      Unintelligible code above (too much optimization)

      ```
      if (preg_match('(<|>)', $frm['title_german'])) {
          // ...
      }
      ```
- What is I-Indescriptive Naming ?
    - Please, name your classes, methods, variables properly, so that people actually know what you mean. I'm not arguing about variables like $i, those are short, but still self-explanatory.
- What is Duplication ?
    - Copy and paste may seem easy but truly it makes the code ugly. You have a long codebase full of redundant codes.

SOLID-principles (and why to follow)

- What is S - Principle of single responsibility (SRP)?
    - A class, method or function should have only one responsibility i.e it should carry out only one function. A function that checks if a user exists before logging the user in does not follow SRP. That is because it carries out two functions, checks if the user already exists and logging the user. These two actions should be

separated into different functions, one for checking if the user exists and the other functions for logging the user in. That way it abides by SRP.

    - Using layers in your application helps a lot. Split big classes in smaller ones, and avoid god classes. Last but not least, write straightforward comments.

- What is meant by O - Open / closed principle (OCP) ?
    - A class or method should be open to extension but closed to modification. This is achievable through inheritance or composition where the base class is extended to any subclass including its methods without modifying the base class. Methods can also be overridden in subclasses ensuring polymorphism. In functional programming, this is achievable using higher-order functions. You should make all member variables private by default. Write getters and setters only when you need them.
- What is L - Liskov Substitution Principle (LSP) ?
    - Liskov Substitution Principle or LSP states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program.
    - Let's take an example. A rectangle is a plane figure with four right angles. It has a width, and a height. Now, take a look at the following pseudo-code:

```
rect = new Rectangle();

rect.width  = 10;
rect.height = 20;

assert 10 == rect.width
assert 20 == rect.height
```

    - We simply set a width and a height on a Rectangle instance, and then we assert that both properties are correct. So far, so good.
      Now we can improve our definition by saying that a rectangle with four sides of equal length is called a square. A square is a rectangle so we can create a Square class that extends the Rectangle one, and replace the first line above by the one below:

```
rect = new Square();
```

    - According to the definition of a square, its width is equal to its height. Can you spot the problem? The first assertion will fail because we had to change the behavior of the setters in the Square class to fit the definition. This is a violation of the Liskov Substitution Principle.
- What is I - Principle of interface segregation (ISP)?
    - Clients should not be forced to depend on methods it does not use. Do not add functionality to an existing interface by adding new methods but rather create many client-specific interfaces that contain only what the client needs for its specific use case.
    - Interface Segregation Principle or ISP states that many client-specific interfaces are better than one general-purpose interface. In other words, you should not

have to implement methods that you don't use. Enforcing ISP gives you low coupling, and high cohesion.

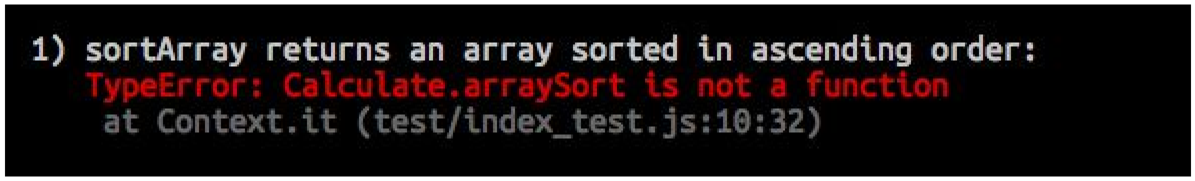
- What is D - Principle of dependency investment (DIP)?
    - Abstractions should not depend on details. Details should depend on the abstractions. High-level modules should not depend on low-level modules. They should both depend on the abstraction.
    - This principle could be rephrased as use the same level of abstraction at a given level. Interfaces should depend on other interfaces. Don't add concrete classes to method signatures of an interface. However, use interfaces in your class methods

Think-Red-Green-Refactor

- What is Red-Green-Refactor?
    - The red phase is always the starting point of the red, green, refactor cycle. The purpose of this phase is to write a test that informs the implementation of a feature. The test will only pass when its expectations are met.

      For example, imagine you want to create a function called sortArray that sorts the numerical values of an array in ascending order. You may start by writing a test that checks the following input and output:

      Input: [2, 4, 1]
      Output: [1, 2, 4]
      When you run this test, you may see an error message like:



```
1) sortArray returns an array sorted in ascending order:
   TypeError: Calculate.arraySort is not a function
    at Context.it (test/index_test.js:10:32)
```

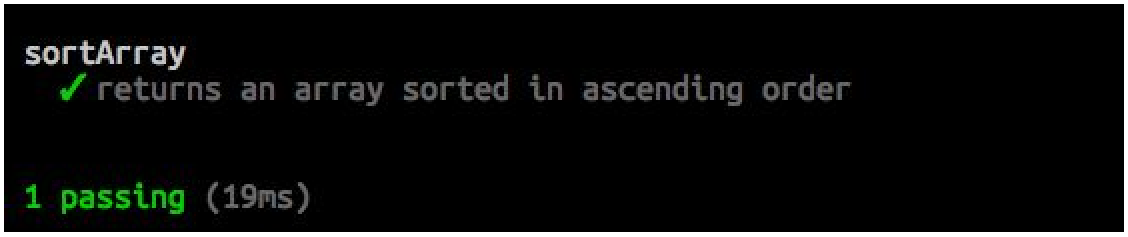
      As you can see, the purpose of this phase was to define what you want to implement. The resulting error message from this test informs your approach to implementation. At this point, you are considered "in the red".

    - The green phase is where you implement code to make your test pass. The goal is to find a solution, without worrying about optimizing your implementation.

      In our sortArray example, the goal is to accept an array like [2, 4, 1] and return [1, 2, 4].

      I decide to approach the problem by writing a loop that iterates over the array, and moves the current number over if it is larger than the number to the right. I nest

this loop inside of a loop that repeats until all of the numbers are sorted (the length of the array). Sorting an array with [3, 4, 5, 6, 2, 1] would look like:
After I implement this, I should see a passing message that looks like:



```
sortArray
  ✓ returns an array sorted in ascending order


1 passing (19ms)
```

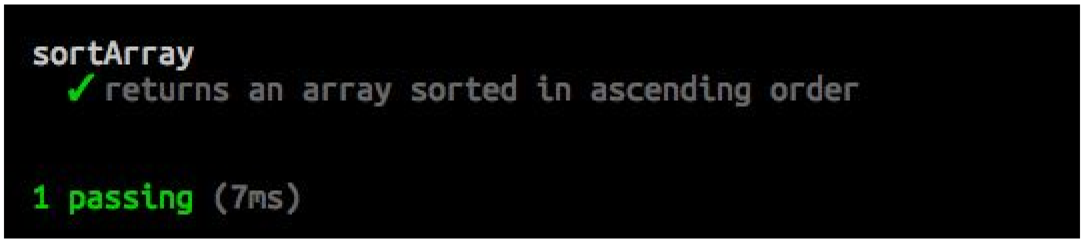
At the end of this phase, you are consider "in the green." You can begin thinking about optimizing your codebase, while having a descriptive test if you do something wrong.

- In the refactor phase, you are still "in the green." You can begin thinking about how to implement your code better or more efficiently. If you are thinking about refactoring your test suite, you should always consider the characteristics of a good test, MC-FIRE. If you want to think about refactoring your implementation code, you can think about how to accomplish the same output with more descriptive or faster code.

  Let's take a moment to think about how I would approach refactoring my sortArray function. After doing some research of sorting methods, I find that I implemented the bubble sort method, which, it turns out, is not a particularly fast sorting method.

  I choose to change the method I use in sortArray() to the merge sort algorithm, because it has a faster average sorting speed than bubble sort.

  As I refactor sortArray(), I have the safety net of my test to notify me if my implementation goes off the tracks, and returns the wrong answer. When I complete my refactor and run my suite again, I receive the following output:



```
sortArray
  ✓ returns an array sorted in ascending order


1 passing (7ms)
```

  The new test code ran faster than before. The (7ms) next to 1 passing is shorthand for seven milliseconds, the amount of time it takes to run the test. At the end of the green phase, our test suite took nineteen milliseconds. Although a twelve millisecond difference is trivial here, the amount of time it takes to run a test suite increases considerably as your codebase grows.

You may not always need to refactor your codebase. However when you're in the green, it's important to ask a few questions before putting yourself back "in the red".

Objectives of project

- Learn what SOLID principles are to apply them in the development of all your projects.
- Learn what is meant by the term STUPID in the world of software development
- Learn what is meant by the term Red - Green - Refactor in the world of software development.

Table of Contents

Project requirements

- You must use GIT
- Create a clear and orderly directory structure

- Both the code and the comments must be written in English
- Use the camelCase code style for defining variables and functions
- In the case of using HTML, never use online styles
- In the case of using different programming languages always define the implementation in separate terms
- Remember that it is important to divide the tasks into several sub-tasks so that in this way you can associate each particular step of the construction with a specific commitment
- You should try as much as possible that the commits and the planned tasks are the same
- Delete terms that are not used or are not necessary to evaluate the project

Risk management

Every project has risks. These risks must be taken into account to improve the workflow of the project. Managing these risks is important for due completion of the project. Unchecked risks could not only lead to hamper of project deliverance but also a badly executed project. The risks hence associated with the project are documented as follows:

Risk	Risk level
Unfamiliarity with OOP	Medium to Low
Health and computer issues	High
Delivering in time	Medium
Completing all the project requirements	Medium

Task Management

The following are the tasks which needed to be accomplished for the completion of the project. The tasks have been divided according to the project specifics. Each task has been clearly defined and their priority as well as their difficulty and time needed. Difficulty level is explained on a scale of 1 to 5 (5 being the most difficult). Priority is explained on the level of 1 to 5 (again 5 the highest parameter being the most prioritized work).

Task	Priority	Description	Difficulty	Time
Read the description of the project	5	This task involves complete understanding of requirements for the project	1	30 mins
Create git repo	5	Creating of git repository for project execution and delivery	1	2 mins

Create the directory structure	4	Creating the necessary files, folders and subfolders for this project.	1	2 mins
SOLID, STUPID and RGB theoretical	4	Understand completely the overview of OOP	2	1-2 hrs
Documentatio n	1	Write the documentation for this project	1	3-4 hr

Defining this part is crucial to the development of the project. It is important to make a good analysis of the situation to organize the project in a good way.

Chronogram

This shows the time-line it took for the project to finish and also the tasks that were achieved during those timeline.

The tasks have been mentioned in the left axis

Tasks	Fri 1000 hrs	Fri 1200 hrs	Fri 1300 hrs
Read project description	**X**		
Git Repo	**X**		
Create files	**X**		
OOP theory	**X**	**X**	**X**
Documentatio n		**X**	**X**

Git Workflow

For this project, all commits we'll be pushed directly to the Master branch. All commits will use a descriptive message, so that the admin and other internet user can follow through the processes.
For more information go to :
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Incidents

- None

Technologies used

For this project, we will use the following technologies:

- PhP
- OOP