PHP

SOLID: Part 1 - The Single Responsibility Principle

by Patkos Csaba 13 Dec 2013 25 Comments

15 11

English

Single Responsibility (SRP), Open/Close, Liskov's Substitution, Interface Segregation, and Dependency Inversion. Five agile principles that should guide you every time you write code.

The Definition

> *A class should have only one reason to change.*

Defined by Robert C. Martin in his book Agile Software Development, Principles, Patterns, and Practices and later republished in the C# version of the book Agile Principles, Patterns, and Practices in C#, it is one of the five SOLID agile principles. What it states is very simple, however achieving that simplicity can be very tricky. A class should have only one reason to change.

But why? Why is it so important to have only one reason for change?

In statically typed and compiled languages, several reasons may lead to several, unwanted redeployments. If there are two different reasons to change, it is conceivable that two different teams may work on the same code for two different reasons. Each will have to deploy its solution, which in the case of a compiled language (like C++, C# or Java), may lead to incompatible modules with other teams or other parts of the application.

Even though you may not use a compiled language, you may need to retest the same class or module for different reasons. This means more QA work, time, and effort.

The Audience

Determining the one single responsibility a class or module should have is much more complex

than just looking at a checklist. For example, one clue to find our reasons for change is to analyze the audience for our class. The users of the application or system we develop who are served by a particular module will be the ones requesting changes to it. Those served will ask for change. Here are a couple of modules and their possible audiences.

- **Persistence Module** - Audience include DBAs and software architects.
- **Reporting Module** - Audience include clerks, accountants, and operations.
- **Payment Computation Module for a Payroll System** - Audience may include lawyers, managers, and accountants.
- **Book Search Module for a Library Management System** - Audience may include the librarian and/or the clients themselves.

Roles and Actors

Associating concrete persons to all of these roles may be difficult. In a small company a single person may need to satisfy several roles while in a large company there may be several persons allocated to a single role. So it seems much more reasonable to think about the roles. But roles by themselves are quite difficult to define. What is a role? How do we find it? It is much easier to imagine actors doing those roles and associating our audience with those actors.

So if our audience defines reasons for change, the actors define the audience. This greatly helps us to reduce the concept of concrete persons like "John the architect" to Architecture, or "Mary the referent" to Operations.

> *So a responsibility is a family of functions that serves one particular actor. (Robert C. Martin)*

Source of Change

In the sense of this reasoning, actors become a source of change for the family of functions that serves them. As their needs change, that specific family of functions must also change to accommodate their needs.

> *An actor for a responsibility is the single source of change for that responsibility. (Robert C. Martin)*

Classic Examples

Objects That Can "Print" Themselves

Let's say we have a `Book` class encapsulating the concept of a book and its functionalities.

```
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function printCurrentPage() {
        echo "current page content";
    }
}
```

This may look like a reasonable class. We have book, it can provide its title, author and it can turn the page. Finally, it is also able to print the current page on the screen. But there is a little problem. If we think about the actors involved in operating the `Book` object, who might they be? We can easily think of two different actors here: Book Management (like the librarian) and Data Presentation Mechanism (like the way we want to deliver the content to the user - on-screen, graphical UI, text-only UI, maybe printing). These are two very different actors.

Mixing business logic with presentation is bad because it is against the Single Responsibility Principle (SRP). Take a look at the following code:

```
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

}

interface Printer {

    function printPage($page);
```

```
}

class PlainTextPrinter implements Printer {

    function printPage($page) {
        echo $page;
    }

}

class HtmlPrinter implements Printer {

    function printPage($page) {
        echo '<div style="single-page">' . $page . '</div>';
    }

}
```

Even this very basic example shows how separating presentation from business logic, and respecting SRP, gives great advantages in our design's flexibility.

Objects That Can "Save" Themselves

A similar example to the one above is when an object can save and retrieve itself from presentation.

```
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

    function save() {
        $filename = '/documents/'. $this->getTitle(). ' - ' . $this->getAuthor
        file_put_contents($filename, serialize($this));
    }

}
```

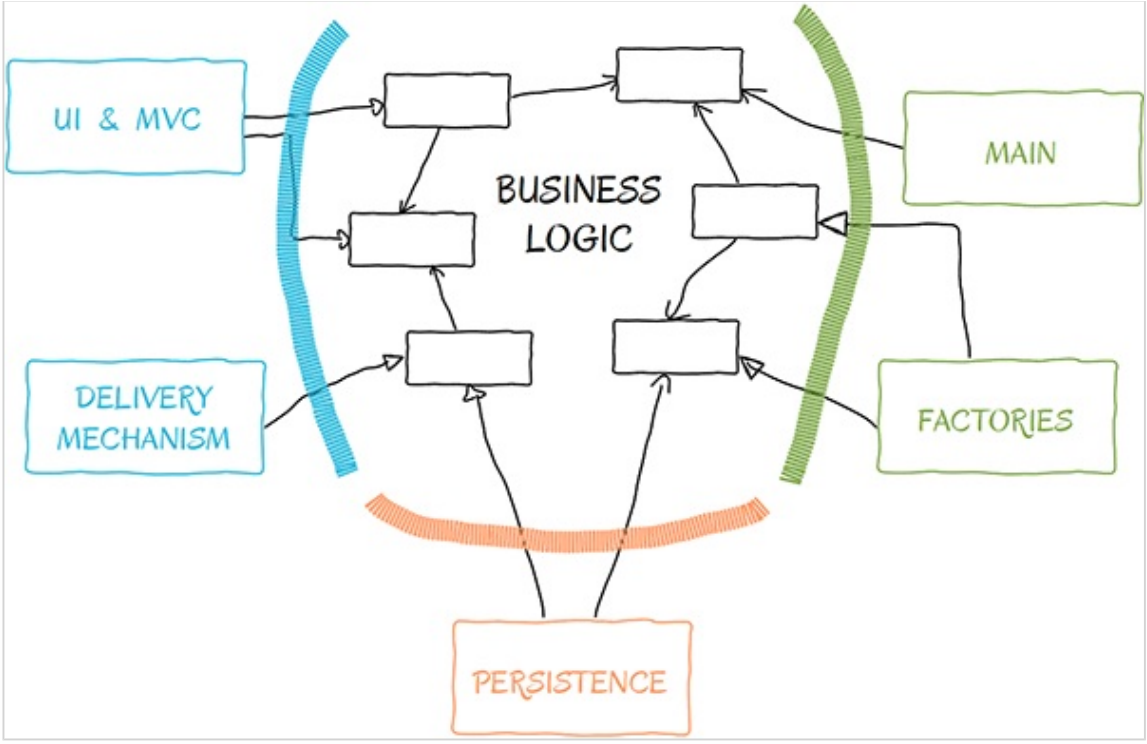
We can, again identify several actors like Book Management System and Persistence. Whenever we want to change persistence, we need to change this class. Whenever we want to change how we get from one page to the next, we have to modify this class. There are several axis of change here.

```php
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

}

class SimpleFilePersistence {

    function save(Book $book) {
        $filename = '/documents/' . $book->getTitle() . ' - ' . $book->getAutho
        file_put_contents($filename, serialize($book));
    }

}
```

Moving the persistence operation to another class will clearly separate the responsibilities and we will be free to exchange persistence methods without affecting our `Book` class. For example implementing a `DatabasePersistence` class would be trivial and our business logic built around operations with books will not change.

A Higher Level View

In my previous articles I frequently mentioned and presented the high level architectural schema that can be seen below.



If we analyze this schema, you can see how the Single Responsibility Principle is respected. Object creation is separated on the right in Factories and the main entry point of our application, one actor one responsibility. Persistence is also taken care of at the bottom. A separate module for the separate responsibility. Finally, on the left, we have presentation or the delivery mechanism if you wish, in the form of an MVC or any other type of UI. SRP respected again. All that remains is to figure out what to do inside of our business logic.

Software Design Considerations

When we think about the software that we need to write, we can analyze many different aspects. For example, several requirements affecting the same class may represent an axis of change. This axes of change may be a clue for a single responsibility. There is a high probability that groups of requirements that are affecting the same group of functions will have reasons to change or be specified in the first place.

The primary value of software is ease of change. The secondary is functionality, in the sense of satisfying as much requirements as possible, meeting the user's needs. However, in order to achieve a high secondary value, a primary value is mandatory. To keep our primary value high, we must have a design that is easy to change, to extend, to accommodate new functionalities and to ensure that SRP is respected.

We can reason in a step by step manner:

1. High primary value leads in time to high secondary value.

2. Secondary value means needs of the users.
3. Needs of the users means needs of the actors.
4. Needs of the actors determines the needs of changes of these actors.
5. Needs of change of actors defines our responsibilities.

So when we design our software we should:

1. Find and define the actors.
2. Identify the responsibilities that serve those actors.
3. Group our functions and classes so that each has only one allocated responsibility.

Discover Tanzania

flash-safaris.com

We make your dream come true with a tailor made trip!

Advertisement

A Less Obvious Example

```
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

    function getLocation() {
        // returns the position in the library
        // ie. shelf number & room number
    }

}
```

Now this may appear perfectly reasonable. We have no method dealing with persistence, or presentation. We have our `turnPage()` functionality and a few methods to provide different information about the book. However, we may have a problem. To find out, we might want to analyze our application. The function `getLocation()` may be the problem.

All of the methods of the `Book` class are about business logic. So our perspective must be from the business's point of view. If our application is written to be used by real librarians who are searching for books and giving us a physical book, then SRP might be violated.

We can reason that the actor operations are the ones interested in the methods `getTitle()`, `getAuthor()` and `getLocation()`. The clients may also have access to the application to select a book and read the first few pages to get an idea about the book and decide if they want it or not. So the actor readers may be interested in all the methods except `getLocations()`. An ordinary client doesn't care where the book is kept in the library. The book will be handed over to the client by the librarian. So, we do indeed have a violation of SRP.

```
01  class Book {
02
03      function getTitle() {
04          return "A Great Book";
05      }
06
07      function getAuthor() {
08          return "John Doe";
09      }
10
11      function turnPage() {
12          // pointer to next page
13      }
14
15      function getCurrentPage() {
16          return "current page content";
17      }
18
19  }
20
21  class BookLocator {
22
23      function locate(Book $book) {
24          // returns the position in the library
25          // ie. shelf number & room number
26          $libraryMap->findBookBy($book->getTitle(), $book->getAuthor());
27      }
28
29  }
```

Introducing the `BookLocator`, the librarian will be interested in the `BookLocator`. The client will be interested in the `Book` only. Of course, there are several ways to implement a `BookLocator`. It can use the author and title or a book object and get the required information from the `Book`. It always depends on our business. What is important is that if the library is changed, and the librarian will have to find books in a differently organized library, the `Book` object will not be

affected. In the same way, if we decide to provide a pre-compiled summary to the readers instead of letting them browse the pages, that will not affect the librarian nor the process of finding the shelf the books sits on.

However, if our business is to eliminate the librarian and create a self-service mechanism in our library, then we may consider that SRP is respected in our first example. The readers are our librarians also, they need to go and find the book themselves and then check it out at the automated system. This is also a possibility. What is important to remember here is that you must always consider your business carefully.

Final Thoughts

The Single Responsibility Principle should always be considered when we write code. Class and module design is highly affected by it and it leads to a low coupled design with less and lighter dependencies. But as any coin, it has two faces. It is tempting to design from the beginning of our application with SRP in mind. It is also tempting to identify as many actors as we want or need. But this is actually dangerous - from a design point of view - to try and think of all the parties from the very beginning. Excessive SRP consideration can easily lead to premature optimization and instead of a better design, it may lead to a scattered one where the clear responsibilities of classes or modules may be hard to understand.

So, whenever you observe that a class or module starts to change for different reasons, don't hesitate, take the necessary steps to respect SRP, however don't overdue it because premature optimization can easily trick you.

Advertisement

Difficulty:
Beginner
Length:

Quick

Tagged with:

PHP | Web Development | SOLID

Translations Available:

Português

About Patkos Csaba

I had my first contact with computers in the mid-80s when I visited my father at work. Probably it was an important moment for what I am doing now. I am a proud member of an agile team working for a company called Syneto. Through my carrier I programmed in several programming languages and I had the chance to learn and use daily all the major Agile techniques from Scrum to Lean and from TDD to DDD. Since August 2012 I am sharing my knowledge with the Nettuts+ readers by articles, tutorials and premium courses, all about programming.

+ Expand Bio

Advertisement

Recommended for you

24 Ja...

code.tutsplus.com

Android...

code.tutsplus.com

How to...

code.tutsplus.com

10 Mo...

code.tutsplus.com

Related Posts

Refactoring Legacy Code: Part 7 - Identifying the Presentation Layer

1 day ago • Learn techniques for how to deal with complex and complicated unknown legacy code, how to underst...

Single Page ToDo Application With Backbone.js

20 days ago • Join me as I walk you through creating models, collections, views, events, and a router to build ...

SOLID: Part 4 - The Dependency Inversion Principle

13 Feb 2014 • The Single Responsibility (SRP), Open/Closed (OCP), Liskov Substitution, Interface Segregation, a...

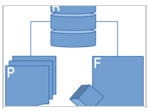
SOLID: Part 3 - Liskov Substitution & Interface Segregation Principles

24 Jan 2014 • The Single Responsibility (SRP), Open/Closed (OCP), Liskov Substitution, Interface Segregation, a...

SOLID: Part 2 - The Open/Closed Principle

20 Jan 2014 • Single Responsibility (SRP), Open/Closed (OCP), Liskov's Substitution, Interface Segregation, and...

The Repository Design Pattern



25 Nov 2013 • The Repository Design Pattern, defined by Eric Evens in his Domain Driven Design book, is one of ...

Advertisement

tuts+

Teaching skills to millions worldwide.

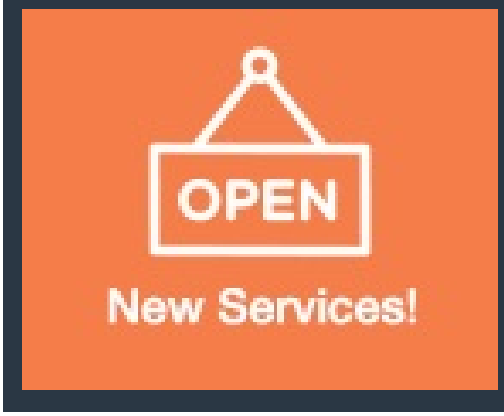
- Tutorials
- Courses
- eBooks
- Jobs
- Blog

Follow Us

- Subscribe to Blog
- Follow us on Twitter
- Be a fan on Facebook
- Circle us on Google+
- Tutorials
- Courses
- eBooks

Add more features to your website such as user profiles, payment gateways, image galleries and more.

Browse WordPress Plugins



Microlancer is now Envato Studio! Custom digital services like logo design, WordPress installaton, video production and more.

Check out Envato Studio

© 2014 Envato Pty Ltd.

envato