.com software Follow
Dec 13, 2022 · 9 min read · · Listen

Save

Simple Trick to Improve Your Classes in PHP

My secret trick how to code better. See how to make your classes better and avoid having too many responsibilities.

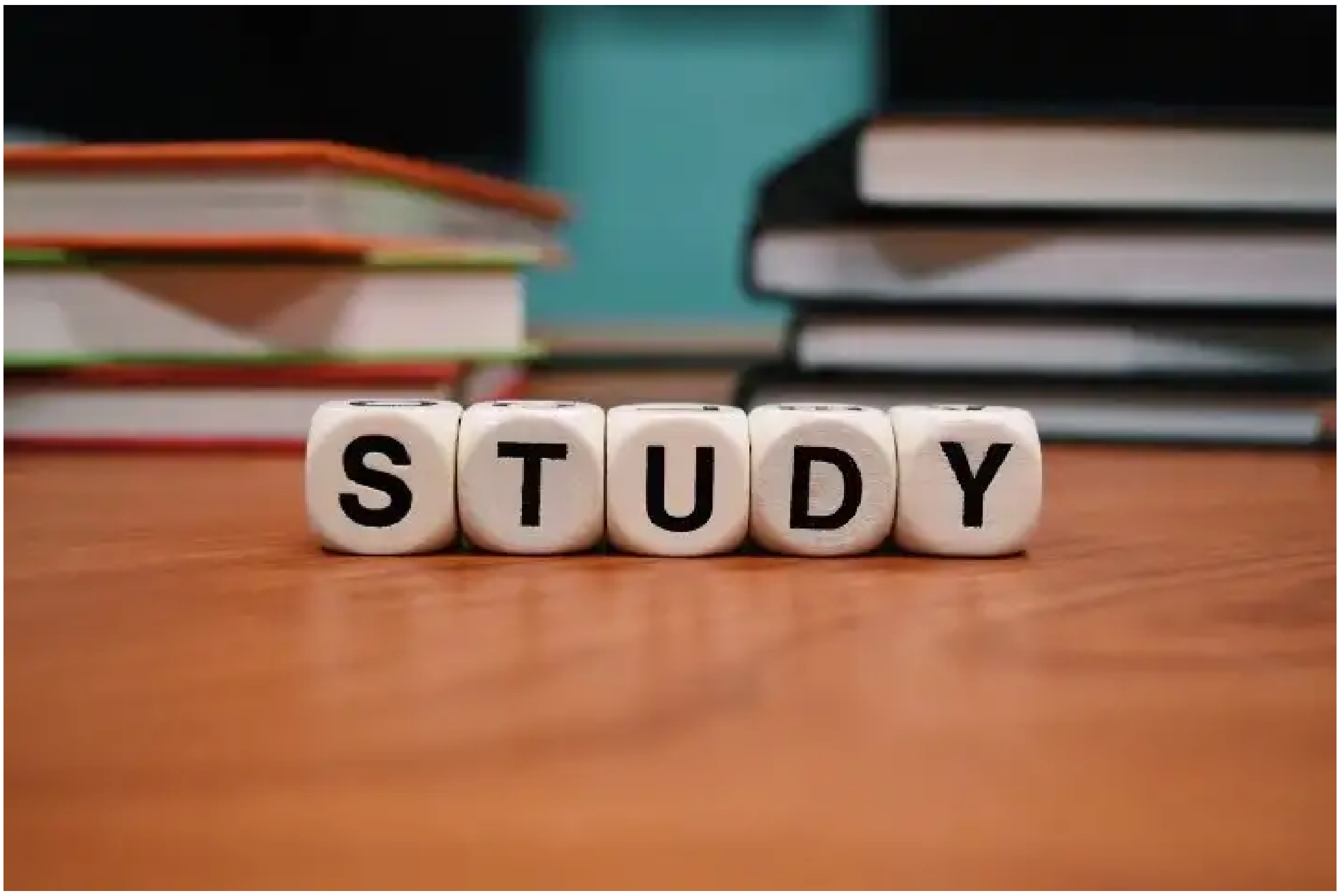
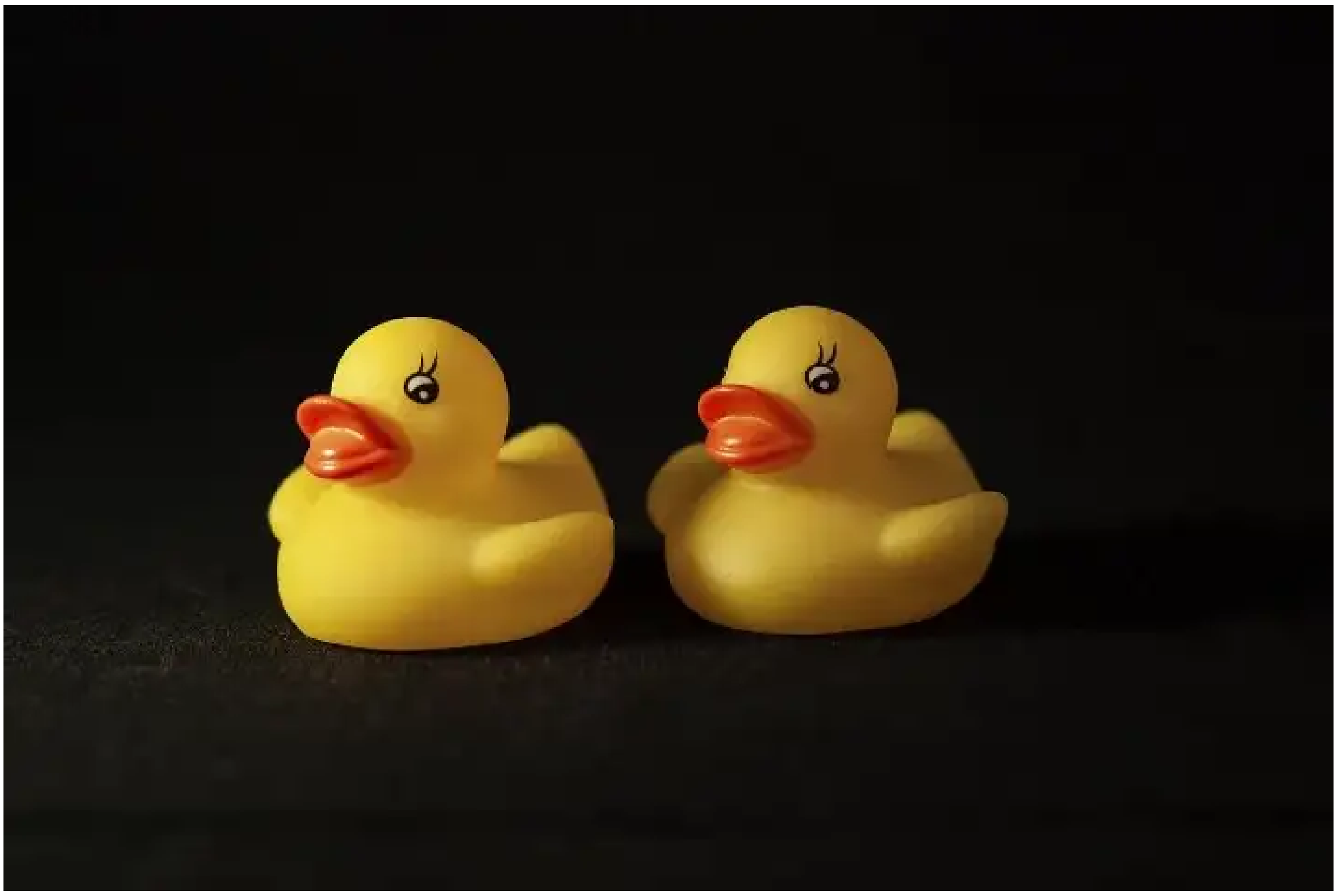


Image by Pixabay

At the moment of writing, there are about 6500 articles on Medium mentioning the *Single Responsibility Principle*. Written mostly by people who don't know what this rule is exactly about.

There are already some **best articles** on the topic, written by the most influential people in the coding industry. I will reference these articles at the bottom of this publication. Today though, I will do my best to avoid the theory and stick to real-life example.



Rubber ducks by Karen Laårk Boshoff

Are you familiar with the Rubberducking technique? In theory, by articulating the problem to a duck (be it imaginary or real if you insist) you often come up with the solution in the process of speaking everything aloud.

This is the trick I use when designing new functionalities. I ask the duck on my desk questions about the code I'm trying to write. **The main question that assists me in writing decoupled code is**:

If was to write more implementations of the following contract, are there any common parts that all classes will have to share?

If the answer to the question is “yes,” then **that common part is the code that has to be decoupled** — extracted to another class.

Let me show you a bunch of authentic examples of how this **helps me write better code!**

> *Examples presented below are in PHP. However, the trick applies to any object-oriented language: Java, Python, you name it.*

Scenario #1 Find or create a user model

We need a service that will always return a user. Is this service/repository interface correct? I’ve seen such implementation in many projects.

```
interface UserRepositoryInterface
{
    public function findByEmailOrCreate(string $email): User;
}
```

Let’s imagine implementations of the contract. I would expect:

- `DoctrineUserRepository`
- `InMemoryUserRepository`
- `EloquentUserRepository`

Each implementation would require to do roughly the same:

```
public function findByEmailOrCreate(string $email): User
{
    $user = $this->databaseSpecificQueryToReconstituteTheObject([
        'email' => $email,
    ]);

    if ($user) {
        return $user;
    }

    // common part of the method that is required
    // to be repeated in case of every implementation
    $user = $this->userFactory->create($email);

    $this->databaseSpecificQueryToPersistTheObject($user);

    return $user;
}
```

The common part is the call to the `UserFactory` to create a User model. This is a clear indicator that this part of the code should not belong here and the class does too much. This means that the contract itself (the interface) is not correct.

Let’s design a better interface:

```
interface UserRepositoryInterface
{
    public function findByEmail(string $email): ?User;

    public function add(User $user): void;
}
```

Implementations of this contract can focus on vendor-specific code to persist and reconstitute the object, without any code duplication.

You can still have a “*provider*” that will do what you need, but will do better as it will be free of any infrastructure related details:

```
final class UserByEmailProvider
{
    public function __construct(
        private UserRepositoryInterface $userRepository,
        private UserFactoryInterface $userFactory,
    ) {}

    public function __invoke(string $email): User
    {
        $user = $this->userRepository->findByEmail($email);

        if (null === $user) {
            $user = $this->userFactory->create($email);
            $this->userRepository->add($user);
        }

        return $user;
    }
}
```

Scenario #2 Order invoice PDF generator

We're developing code to generate PDF invoices from orders submitted on our e-commerce website. Most people would simply jump into the implementation.

Let's do the same and create a simple proof-of-concept service that is going to be used in a controller:

```
class OrderPdfInvoiceGenerator
{
    public function __construct(
        private OrderRepositoryInterface $orderRepository,
        private ViewRenderer $viewRenderer,
        private SnappyPdf $pdfGenerator,
    ) {}

    public function __invoke(string $orderId): \SplFileInfo
    {
        $order = $this->orderRepository->getById($orderId);

        $html = $this->viewRenderer->render(
            'Orders/pdf.template',
            $this->orderToArray($order),
        );

        $invoiceFilename = vsprintf('%s/invoice-%s.pdf', [
            sys_get_temp_dir(),
            $orderId,
        ]);

        $this->pdfGenerator->generateFromHtml($html, $invoiceFilename);

        return new \SplFileInfo($invoiceFilename);
    }

    private function orderToArray(Order $order): array
    {
        return [
            'order_id'              => $order->getId(),
            'order_number'          => $order->getNumber(),
            'order_total'           => $order->getTotal(),
            'order_shipping_total' => $order->getShippingTotal(),
            'created_at'            => $order->getCreatedAt(),
        ];
    }
}
```

I guess the code is legit. In my past days, I've created countless numbers of services like this one 😊 . Let's ask ourselves yet another question:

What other implementations we might ever need?

What if there is a need to output the invoice in the `html` format? Maybe in the future, we would need additional formats. It is a good time to introduce an interface for different invoice format generators. We can extract interface candidate from the previous implementation:

```
interface GeneratesInvoiceFromOrder
{
    public function __invoke(string $orderId): \SplFileInfo;
}
```

Now we can have as many invoice generators as we desire:

```
final class PdfInvoiceGenerator implements GeneratesInvoiceFromOrder {}
final class HtmlInvoiceGenerator implements GeneratesInvoiceFromOrder {}
final class JsonInvoiceGenerator implements GeneratesInvoiceFromOrder {}
```

And again, each implementation would require to do roughly the same:

```
final class SomeInvoiceGenerator implements GeneratesInvoiceFromOrder
{
    public function __invoke(string $orderId): \SplFileInfo
    {
        // common part: retrieve order from the database
        $order = $this->orderRepository->getById($orderId);

        // common part: convert order to array
        $orderPayload = $this->orderToArray($order);

        // common part: generate temporary file
        $invoiceFilename = vsprintf('%s/invoice-%s.format', [
            sys_get_temp_dir(),
            $orderId,
        ]);

        $invoiceContents = $this->generateFormatSpecificInvoiceContent(
            $orderPayload
        );

        // common part: store invoice to a temporary file
        file_put_contents($invoiceFilename, $invoiceContents);

        // common part: return the result
        return new \SplFileInfo($invoiceFilename);
    }
}
```

How about we get rid of all the common parts of the code? All we ever need is the order payload only and a `stringable` result to store it somewhere. Let's rewrite the interface to:

```
interface GeneratesInvoiceFromOrder
{
    /** @param array<string, mixed> $orderPayload */
    public function __invoke(array $orderPayload): string;
}
```

Our invoice generators are now much leaner, focused, easier to maintain, to unit test, to work with:

```
final class HtmlInvoiceGenerator implements GeneratesInvoiceFromOrder
{
    public function __construct(
        private ViewRenderer $viewRenderer,
    ) {}

    /** {@inheritDoc} */
    public function __invoke(array $orderPayload): string
    {
        return $this->viewRenderer->render(
            'Orders/pdf.template',
            $orderPayload,
        );
    }
}
```

For the `pdf` generator we could use the generator from above to reduce code duplication yet further:

```
final class PdfInvoiceGenerator implements GeneratesInvoiceFromOrder
{
    public function __construct(
        private HtmlInvoiceGenerator $htmlGenerator,
        private SnappyPdf $pdfGenerator,
    ) {}

    /** {@inheritDoc} */
    public function __invoke(array $orderPayload): string
    {
        $html = ($this->htmlGenerator)($orderPayload);
```

Open in app ↗ Sign up Sign In

Another example for `json` format:

```
final class JsonInvoiceGenerator implements GeneratesInvoiceFromOrder
{
    /** {@inheritDoc} */
    public function __invoke(array $orderPayload): string
    {
        return json_encode($orderPayload, JSON_THROW_ON_ERROR);
    }
}
```

As you can see we managed to reduce code duplication substantially. We still need some code for the controller to fetch the `Order` entity, create the payload, etc.

A factory would be nice to handle different formats:

```
final class OrderInvoiceGeneratorFactory
{
    /** @var array<string, GeneratesInvoiceFromOrder> */
    private array $generators;

    public function __construct(
        private HtmlInvoiceGenerator $htmlGenerator,
        private PdfInvoiceGenerator $pdfGenerator,
        private JsonInvoiceGenerator $jsonGenerator,
    ) {
        $this->generators = [
            'html' => $htmlGenerator,
            'pdf' => $pdfGenerator,
            'json' => $jsonGenerator,
        ];
    }

    public function __invoke(string $format): GeneratesInvoiceFromOrder
    {
        if (false === isset($this->generators[$format])) {
            throw new UnsupportedFormatException($format);
        }

        return $this->generato
    }
}
```

And the final service for the controller:

```
final class DefaultInvoiceGenerator
{
    public function __construct(
        private OrderRepositoryInterface $orderRepository,
        private OrderInvoiceGeneratorFactory $orderInvoiceGeneratorFactory,
    ) {}

    /**
     * @throws OrderNotFoundException
     * @throws UnsupportedFormatException
     */
    public function __invoke(string $orderId, string $format): \SplFileInfo
    {
        $generator = ($this->orderInvoiceGeneratorFactory)($format);

        $orderPayload = $this->orderToArray(
            $this->orderRepository->getById($orderId),
        );

        $invoiceFilename = vsprintf('%s/invoice-%s.%s', [
            sys_get_temp_dir(),
            $orderId,
            $format,
        ]);

        file_put_contents(
            $invoiceFilename,
            $generator($orderPayload),
        );

        return new \SplFileInfo($invoiceFilename);
    }

    private function orderToArray(Order $order): array
    {
        return [
            'order_id'             => $order->getId(),
            'order_number'         => $order->getNumber(),
            'order_total'          => $order->getTotal(),
            'order_shipping_total' => $order->getShippingTotal(),
            'created_at'           => $order->getCreatedAt(),
        ];
    }
}
```

In overall, I'm pretty satisfied with the final result. There's still some room for improvement in this class, but it's outside the scope of this publication:

- use a serializer library to convert the order to the array,
- use a filesystem abstraction library to avoid handling physical files.

Scenario #3 Article scraper

We want our e-commerce solution to display article excerpts from several websites. We might want to create an interface:

```
interface ArticleScraperInterface
{
    public function __invoke(): void;
}
```

To have multiple implementations that would import articles to the database, for every possible website. For instance:

```
final class BloombergArticleScraper implements ArticleScraperInterface
{
    public function __construct(
        private GuzzleClient $httpClient,
        private ArticleFactoryInterface $articleFactory,
        private ArticleRepositoryInterface $articleRepository,
    ) {}

    public function __invoke(): void
    {
        $response = $this->httpClient->get('https://www.bloomberg.com');

        $content = $response->getContents();

        $articleItems = $this->doSomeParsing($content);

        foreach ($articleItems as $articleItem) {
            $article = $this->articleFactory->fromArray($articleItem);

            $this->articleRepository->add($article);
        }
    }

    private function doSomeParsing(string $content): array
    {
        // skipped for brevity
    }
}
```

among others:

```
    final class CnnArticleScraper implements ArticleScraperInterface {}
    final class BbcArticleScraper implements ArticleScraperInterface {}
    final class CnbcArticleScraper implements ArticleScraperInterface {}
```

We could execute all scrapers inside a console command using CRON:

```
    final class ScrapeCronCommand extends Command
    {
        /** @param iterable<ArticleScraperInterface> $scrapers */
        public function __construct(
            private iterable $scrapers,
        ) {}

        public function execute(
            InputInterface $input,
            OutputInterface $output,
        ): int {
            foreach ($this->scrapers as $scraper) {
                $scraper();
                $output->writeln('Complete: ' . get_class($scraper));
            }

            return 0;
        }
    }
```

At the first glance, the implementation of the scraper looks nice, right? It loads a website, does some parsing and stores articles.

However, if we dive deep and study possible implementations, we can spot lots of potentially *repeating parts of the code*:

```
public function __invoke(): void
{
    // common part: fetch contents of the remote website
    $response = $this->httpClient->get('https://...');

    // common part: getting contents from the response object
    $content = $response->getContents();

    $articleItems = $this->doSomeParsing($content);

    // common part: iterating over parsed items
    foreach ($articleItems as $articleItem) {
        // common part: build the article model
        $article = $this->articleFactory->fromString(
            $articleItem['url'],
            $articleItem['title'],
            $articleItem['excerpt'],
        );

        // common part: persist the model
        $this->articleRepository->add($article);
    }
}
```

It seems like the only unique code that matters is the actual HTML parsing of the response. We could rewrite the interface to the following:

```
interface ArticleScraperInterface
{
    public function getUrl(): string;

    /**
     * @return iterable<array{url: string, title: string, excerpt: string}>
     */
    public function getArticleItems(string $htmlResponse): iterable;
}
```

Now let’s rewrite the implementation:

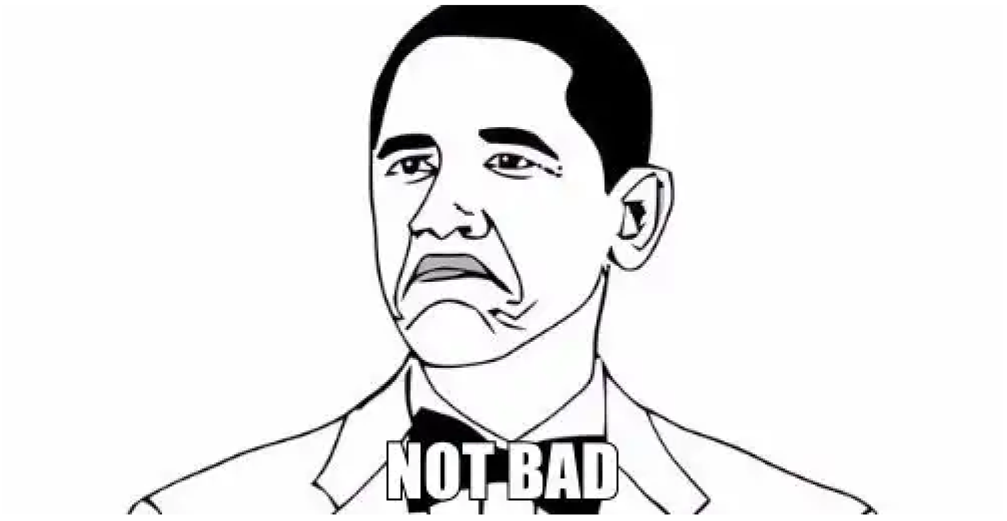
```
final class BloombergArticleScraper implements ArticleScraperInterface
{
    public function getUrl(): string
    {
        return 'https://www.bloomberg.com';
    }

    /** {@inheritDoc} */
    public function __invoke(string $htmlResponse): iterable
    {
        foreach ($this->doSomeParsing($htmlResponse) as $articleItem) {
            yield $articleItem;
        }
    }
}
```

The new implementation is yet again much cleaner, easier to unit test and maintain! We've managed to reduce a lot of duplicated code. We still need some kind of a service to orchestrate the scraping process:

```
final class ArticleScrapingOrchestrator
{
    /** @param iterable<ArticleScraperInterface> $scrapers */
    public function __construct(
        private iterable $scrapers,
        private GuzzleClient $httpClient,
        private ArticleFactoryInterface $articleFactory,
        private ArticleRepositoryInterface $articleRepository,
    ) {}

    /**
     * @return iterable<string>
     */
    public function __invoke(): iterable
    {
        foreach ($this->scrapers as $scraper) {
            $response = $this->httpClient->get($scraper->getUrl());

            $articleItems = $scraper->getArticleItems(
                $response->getContents()
            );

            foreach ($articleItems as $articleItem) {
                $article = $this->articleFactory->fromString(
                    $articleItem['url'],
                    $articleItem['title'],
                    $articleItem['excerpt'],
                );

                $this->articleRepository->add($article);
            }

            yield $scraper->getUrl();
        }
    }
}
```

The new command will get simplified as well as it does not need to deal with individual scrapers anymore, but simply call the orchestrator:

```
final class ScrapeCronCommand extends Command
{
    public function __construct(
        private ArticleScrapingOrchestrator $scrapingOrchestrator,
    ) {}

    public function execute(
        InputInterface $input,
        OutputInterface $output,
    ): int {
        foreach (($this->scrapingOrchestrator)() as $scrapedUrl) {
            $output->writeln('Complete: ' . $scrapedUrl);
        }

        return 0;
    }
}
```



Meme via Meme-Arsenal.com

Summary

We were able to analyze and refine three *real-life* code examples. With **one simple trick**, we were able to remove duplicated noise from the code and permanently improve its quality.

Well-designed code is maintainable code. Such code is also much easier to unit test. New implementations can focus on what is truly important to them without going into **unnecessary details**.

I hope you enjoyed this publication as much as I did while writing it. Developing good code is not trivial by any means. Bad decisions tend to backfire in the future.

It is much less painful to spend some time on the proper design before actually jumping to your favorite editor.

Best Books about Refactoring Everyone Should Read

Refactoring: Improving the Design of Existing Code

As the application of object technology--particularly the Java programming language--has become commonplace, a new...

amzn.to

Refactoring to Patterns

Book annotation not available for this title.Title: Refactoring to PatternsAuthor: Kerievsky, JoshuaPublisher: Prentice...

amzn.to

References

- The Rubber Duck Technique
- About “Single Responsibility Principle” by Uncle Bob
- Why I prefer the “__invoke” method
- Why I love interfaces

PHP Programming Languages Php Developers Laravel Development