BE **SOLID**

Five basic programming principles

A NEWCRAFT tech meetup

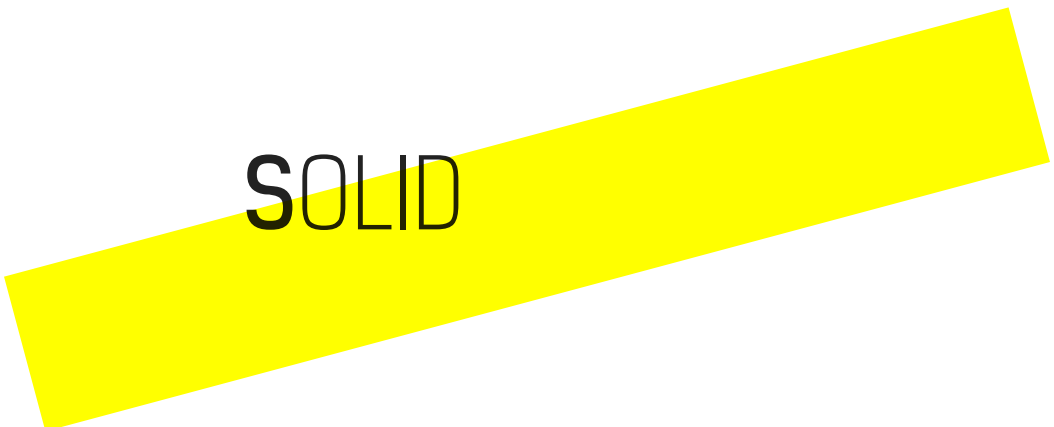
S.O.L.I.D.

SOLID is an acronym defining
five basic programming principles

SOLID principles were defined by
Robert *"Uncle Bob"* Martin
in the early 2000s

WHAT DOES **SOLID** MEANS?

Single Responsability Principle
Open/Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle


SOLID

SINGLE RESPONSABILITY PRINCIPLE*

There should **never** be more than **one reason** for a class to change.

* also known as SRP

DON'T DO GOD CLASSES

```
class OrderEmailHandler
{
    public function __construct(array $data) { /* ... */ }
    public function createMessage() { /* ... */ }
    public function send($email) { /* ... */ }
    public function displayWebVersion() { /* ... */ }
    private function validateEmail() { /* ... */ }
    private function checkUserPermission() { /* ... */ }
}
```

BUT RATHER **SMALL** CLASSES (WHICH WORK TOGETHER)

```
// namespace MyApp\Controller\Order;
class OrderMessageController {}

// namespace MyApp\Email\Order;
class OrderMessageCreator {}
class OrderMessageSender {}

// namespace MyApp\Validator
class EmailValidator {}
```

ONE RESPONSABILITY PER CLASS

- One reason to change
- Stay focused on a single concern
- Simplify maintenance
- Simplify testing
- Can be reused (DRY ❤)

ONE CLASS PER RESPONSABILITY

- Always try to give every class its **own** responsibility
- Define for every responsibility a class
- When the responsability changes, **only one class** has to be modified

DEMO

EmailAndPdfHtmlProcessor

Responsible for:

- creating the HTML used to render the PDF
- creating the HTML used for the email body
- creating the Swift_Message object

TOO MANY RESPONSABILITIES

TIPS

If your class name contains a separator like **And** or **Or**
The class has probably more than one goal

Write straightforward code and comments
Conditions like `if / elseif / else` and `switch` are an indicator.
If you start using many conditions you are going on the wrong direction


SOLID

OPEN/CLOSED PRINCIPLE*

Software entities should be **open for extension**,
but **closed for modification**.

* also known as OCP

IN OTHER WORDS

When a new functionality is needed,
we should **not modify our existing code**
but rather **write new code** that will be **used by existing code**.

EXAMPLE

Calculating price

CONTEXT

We sell subscriptions for a big phone operator

```
public class Subscription
{
    private $price;
    private $discount;
    private $optionPrice;

    // getters...
}
```

```
class PriceCalculator
{
    public function calculate(array $subscriptions)
    {
        $price = 0;
        foreach ($subscriptions as $subscription) {
            $price += $subscription->getPrice() -
                $subscription->getDiscount() +
                $subscription->optionPrice;
        }

        return $price;
    }
}
```

NEW FEATURE

A new product `Phone` comes into play
... and its price is calculated differently

```
public class Phone
{
    private $quantity;
    private $price;
    private $discount;

    // getters...
}
```

HOW TO CALCULATE THE NEW PRICE?

```
class PriceCalculator
{
    public function calculate(array $products)
    {
        $price = 0;
        foreach ($products as $product) {
            if ($product instanceof Subscription) {
                $price += $product->getPrice() -
                    $product->getDiscount() +
                    $product->getOptionPrice();
            } else {
                $price += $quantity *
                    ($product->getPrice() - $product->getDiscount());
            }
        }

        return $price;
    }
}
```

It works **but**...

- `PriceCalculator` is **not closed for modification**
  as we need to change it in order to extend it.
- We'll quickly have an **ugly** and **unmaintenable** peace of code

HOW TO BE **OCP** COMPLIANT?

One solution among others

```
interface Orderable
{
    public function calculatePrice();
}
```

```
public class Subscription implements Orderable
{
    public function calculatePrice()
    {
        return $this->price - $this->discount + $product->optionPrice;
    }
}
```

```
public class Phone implements Orderable
{
    public function calculatePrice()
    {
        return $quantity * ($this->getPrice() - $this->getDiscount());
    }
}
```

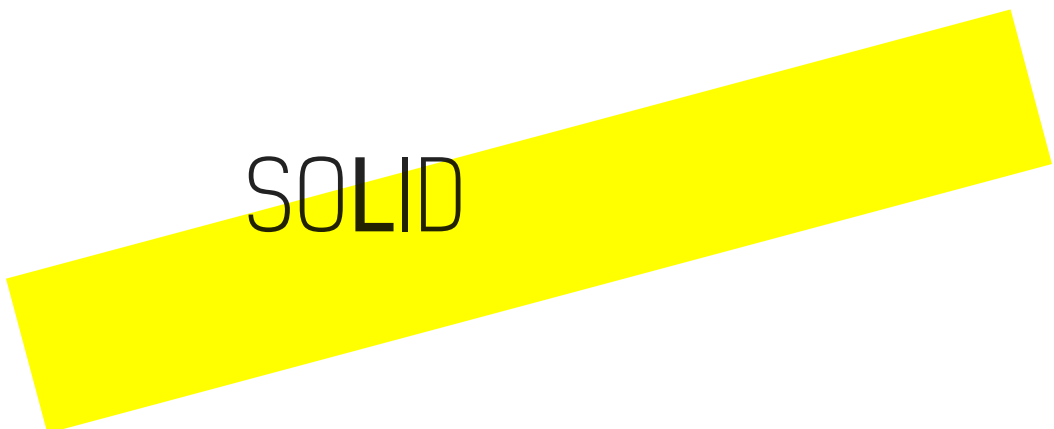
```php
class PriceCalculator
{
    /**
     * @param Orderable[] $products
     */
    public function calculate(array $products)
    {
        $price = 0;
        foreach ($products as $product) {
            $price += $product->calculatePrice();
        }

        return $price;
    }
}
```

We'll never modify `PriceCalculator` again,
it's closed for modification.

When we need a new class, it has to implement `Orderable`
and calculate its price alone.

We reduce the risk of introducing bugs in an old functionality, as the `PriceCalculator::calculate()` method is already unit tested


SOLID

LISKOV SUBSTITUTION PRINCIPLE*

Functions that use pointers or references to base classes must be able **to use objects of derivated classes without knowing it**

* also known as LSP

IN OTHER WORDS

Objects in a program **should be replaceable with instances of their subtypes** without altering the correctness of the program.

EXAMPLE

That is **NOT** related to rectangles and squares

But you can find the famous rectangle example HERE

```php
class DeviceCollection
{
    protected $collection;

    public function add(Device $device)
    {
        $this->collection[$device->getId()] = $device;
    }

    public function getAll()
    {
        return $this->collection;
    }
}
```

```php
class OrderedDeviceCollection extends DeviceCollection
{
    public function add(Device $device)
    {
        parent::add($device);

        usort($this->collection, function($a, $b) {
            if ($a->getPrice() === $b->getPrice()) {
                return 0;
            }

            return ($a->getPrice() < $b->getPrice()) ? -1 : 1;
        });
    }
}
```

WHY IS THE LSP VIOLATED?

```
public function testAddToCollection()
{
    $collection = new DeviceCollection();
    $collection->add(new Device(1, 9.99));
    $collection->add(new Device(2, 199));
    $collection->add(new Device(3, 59));

    $this->assertEqual(
        [new Device(1, 9.99), new Device(2, 199), new Device(3, 59)],
        $collection->getAll()
    );
    // true
}
```

Works! *But wait a minute...*

What happens if we use

`$collection = new OrderedDeviceCollection();`?

We can't replace `DeviceCollection` by a class of its subtypes

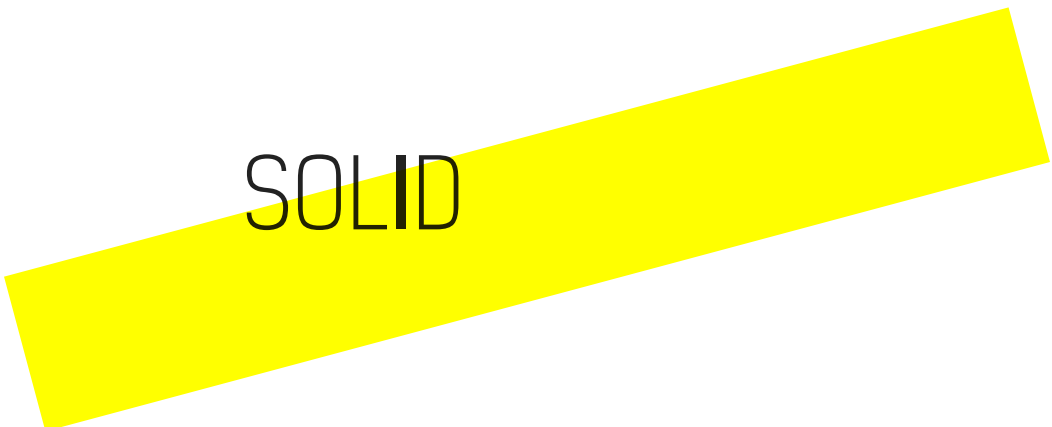
LSP is violated

A LSP VIOLATION IN OUR CODE

```php
class CellSubscription extends Product
{
    public function setMinutes($minutes)
    {
        $this->minutes = $minutes;
    }
}
```

```php
class ConfiguredCellSubscription extends CellSubscription
{
    public function setMinutes($minutes)
    {
        throw new \BadMethodCallException('Cannot set minutes for configured cell subsc
    }
}
```

In that case, using COMPOSITION would have been a better solution than INHERITANCE


SOLID

INTERFACE SEGREGATION PRINCIPLE*

Clients **should not** be forced to depend upon interfaces that they do not use

* also known as ISP

IN OTHER WORDS

Many client specific interfaces are better than
one general-purpose interface

The bigger the interface is, the more likely it is that all its features won't be used by the classes that implement it

EXAMPLE: PRODUCT

```
interface ProductInterface {
    public function getTitle();
    public function getDescription();
    public function getPrice();
    public function getDeliveryMethods();
    public function getEstimatedDeliveryDate();
}
```

```
class Product implements ProductInterface {
    public function getTitle() {
        //...
    }
    public function getDescription {
        //...
    }
    public function getPrice() {
        //...
    }
    public function getDeliveryMethods() {
        //...
    }
    public function getEstimatedDeliveryDate() {
        //...
    }
}
```

EXAMPLE: DIGITAL PRODUCTS ARE INTRODUCED

We don't need to deliver digital products
but the `ProductInterface` forces us to
implement delivery methods

```
class DigitalProduct implements ProductInterface {
    // ...

    public function getDeliveryMethods() {
        throw new NotImplementedException();
    }
    public function getEstimatedDeliveryDate() {
        throw new NotImplementedException();
    }
}
```

NOT A GOOD SOLUTION

HOW TO IMPROVE THAT?

Split the interface

```
interface ProductInterface {
    public function getTitle();
    public function getDescription();
    public function getPrice();
}

interface DeliverableInterface {
    public function getDeliveryMethods();
    public function getEstimatedDeliveryDate();
}
```

`Product` implements `ProductInterface` AND `DeliverableInterface`

```
class Product implements ProductInterface, DeliverableInterface {
    public function getTitle() {
        //...
    }
    public function getDescription {
        //...
    }
    public function getPrice() {
        //...
    }
    public function getDeliveryMethods() {
        //...
    }
    public function getEstimatedDeliveryDate() {
        //...
    }
}
```

While `DigitalProduct`
ONLY implements `ProductInterface`

Unneeded methods are not implemented

```
class DigitalProduct implements ProductInterface {
    public function getTitle() {
        //...
    }
    public function getDescription {
        //...
    }
    public function getPrice() {
        //...
    }
}
```


SOLID

DEPENDENCY INVERSION PRINCIPLE*

1. High level modules **should not** depend upon low level modules.
   Both should depend upon abstractions.
2. Abstractions **should not** depend upon details.
   Details should depend upon abstractions.

So, **Details** should depend upon **Abstration**
Abstractions should depend upon **Abstration**

* also known as DIP

DEPENDENCY INJECTION IS NOT THE SAME AS THE DEPENDENCY INVERSION PRINCIPLE

WHAT IS THE BENEFIT?

Remove tight COUPLING

So, we are able to replace a piece of the system without having to change more than that individual piece

When the wheel of your car is bursted,
you don't want to change the axletree, do you?

EXAMPLE (ADAPTED FROM LOSTECHIES)

Let's imagine a service containing a encryption algorythm

```
class EncryptionService
{
    public function encrypt($sourceFileName, $targetFileName)
    {
        // Read content
        $content = file_get_contents($sourceFileName);

        // encrypt
        $encryptedContent = $this->doEncryption($content);

        // write encrypted content
        file_put_contents($targetFileName, $encryptedContent);
    }

    private function doEncryption($content)
    {
        // put here your encryption algorithm...
        return $encryptedContent;
    }
}
```

Highly coupled to files as input and output
Can't be used in another context

NEW TRY FOLLOWING DIP

```
class EncryptionService
{
    public function encrypt(ReaderInterface $reader, WriterInterface $writer)
    {
        // Read content
        $content = $reader->getContent();

        // encrypt
        $encryptedContent = $this->doEncryption($content);

        // write encrypted content
        $writer->write($encryptedContent);
    }

    // doEncryption method...
}
```

```
interface ReaderInterface
{
    public function getContent();
}
```

```
interface WriterInterface
{
    public function write($content);
}
```

WHAT IS BETTER?

- `EncryptionService` is not dependent of hardcoded reader and writer
- New readers and writers can be created without modifying `EncryptionService` (OCP ❤)
- `EncryptionService` can be easily tested

THANKS

QUESTIONS?

 github.com/antfroger

 @__tooni__