Make Your Project SOLID

Symfony Live 2012 Berlin

Tobias Schlitt (@tobySen)

2012-11-22

About me

- Degree in computer sience
- Proessional PHP since 2000
- Open source enthusiast
- Passion for
  - Software Design
  - Automated Testing

Co-founder of

Helping people to create high quality web applications.

`http://qafoo.com`

- Expert consulting
- Individual training

From 2013 on incorporating Doctrine 2 & Symfony2 expertise!

And there was . . .

S	Singleton (Static)
T	Tight Coupling
U	Untestable
P	Premature Optimization
I	Indescriptive Naming
D	Duplication

`http://bit.ly/dont_be_stupid`

S.O.L.I.D

- 5 essential principles of object oriented design
- Introduced by Robert C. Martin (Uncle Bob)
  - not the inventor of the principles
- Have proven to lead to better code
- Scientific background (partly)

By Example

A weather loader component

- Fetch weather for a city
- Relevant data:
  - Condition
  - Temperature
  - Wind
- Be service-agnostic
  - Weather service come and go
  - Data licenses may change
- Log service failures
- Make it possible to add service fallbacks later

Single Responsibility Principle

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

The Issue

```php
<?php

class GoogleWeatherService
{
    public function getWeatherForLocation( Location $location )
    {
        $xml     = $this->getData( $location );
        $weather = $this->extractWeather( $xml );
        return $weather;
    }

    protected function extractWeather( $xml )
    {
        $weather = new Weather();
        $weather->conditions = $this->parseConditions( $xml );
        // ...
        $weather->windSpeed = $this->convertMilesToKilometer(
            $this->parseWindSpeed( $xml )
        );
        return $weather;
    }

    /* ... */
}
```

The Fix

```php
<?php

class GoogleWeatherService
{
    public function __construct(
        HttpClient $client, GoogleDataParser $parser )
    { /* ... */ }

    public function getWeatherForLocation( Location $location )
    {
        $xml = $this->client->get( sprintf(
            'http://.../?city=%s',
            $location->city
        ) );
        return $this->parser->parseWeather( $xml );
    }
}
```

Single Responsibility Principle

- One responsibility per class
- Separation of concerns
- Responsibilities hard to detect

Open/Close Principle

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

The Wrong Way

```
3   class WeatherLoader
4   {
5       public function __construct( $service )
6       { /* ... */ }
7
8       public function getWeatherForLocation( Struct\Location $location )
9       {
10          // ...
11          switch( get_class( $this->service ) )
12          {
13              case 'GoogleWeatherService':
14                  return $this->service->getWeather( $location );
15
16              case 'WetterComWeatherService':
17                  return $this->service->retrieveWeather(
18                      $location->city, $location->country
19                  );
20              // ...
21          }
22      }
23  }
```

The Right Way

```
3   class WeatherLoader
4   {
5       public function __construct( WeatherService $service )
6       { /* ... */ }
7
8       public function getWeatherForLocation( Struct\Location $location )
9       {
10          // ...
11          return $this->service->getWeatherForLocation( $location );
12      }
13  }
```

Open/Close Principle

- Changes introduce errors
  - Especially cascading changes
- Ideally: Write once, change never!
- Extend software only by new code
  - New interface implementations
  - Inheritance
  - Aggregation

Liskov Substitution Principle

“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

A Simple Class

```php
<?php

class DistanceConverter
{
    const FACTOR = 0.6214;

    public function milesToKilometers( $miles )
    {
        return $miles / self::FACTOR;
    }
}
```

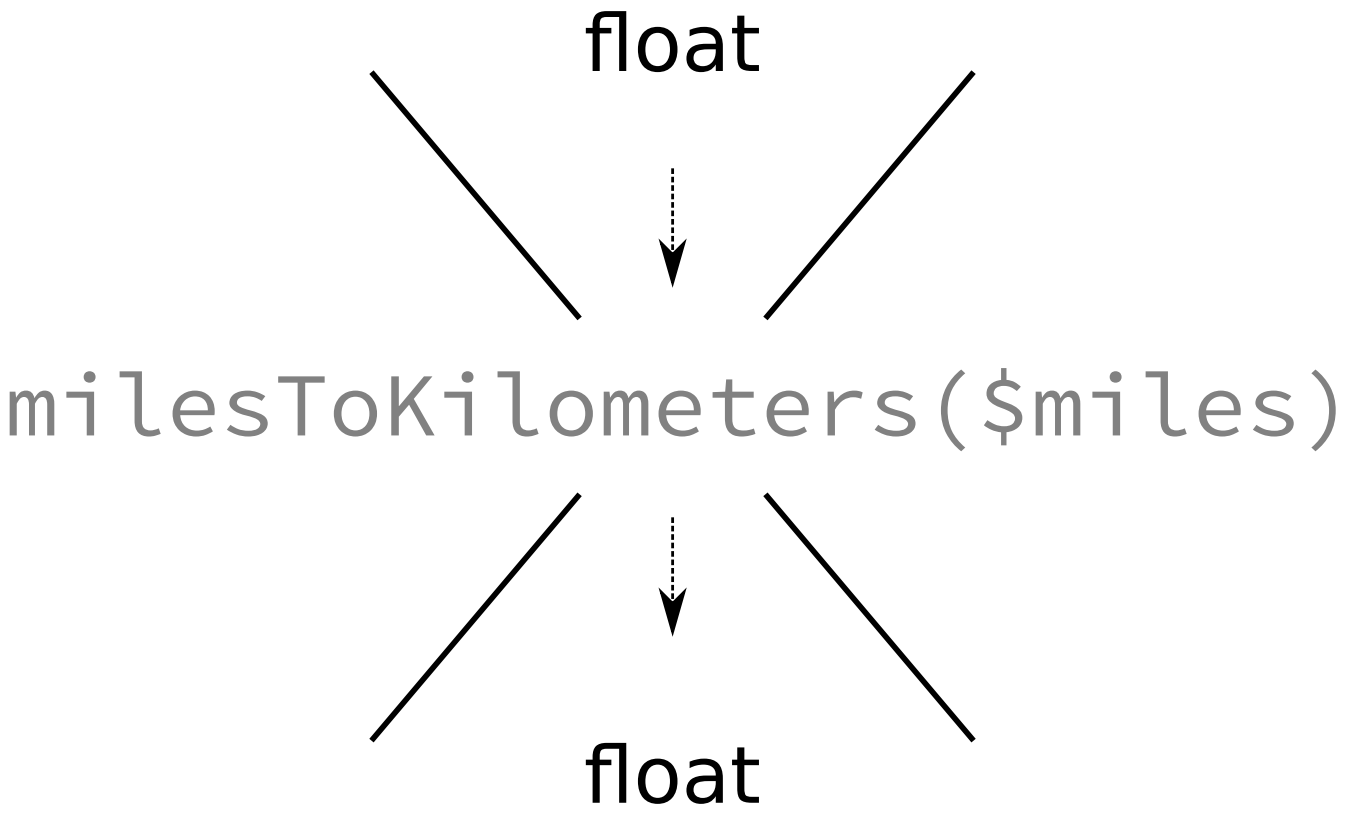
Getting into Trouble

```php
<?php

class FormattingDistanceConverter extends DistanceConverer
{
    public function milesToKilometers( $miles )
    {
        if ( $miles < 0 )
        {
            throw new InvalidArgumentException();
        }
        return sprintf(
            '%01.2f km', parent::milesToKilometers( $miles )
        );
    }
}
```

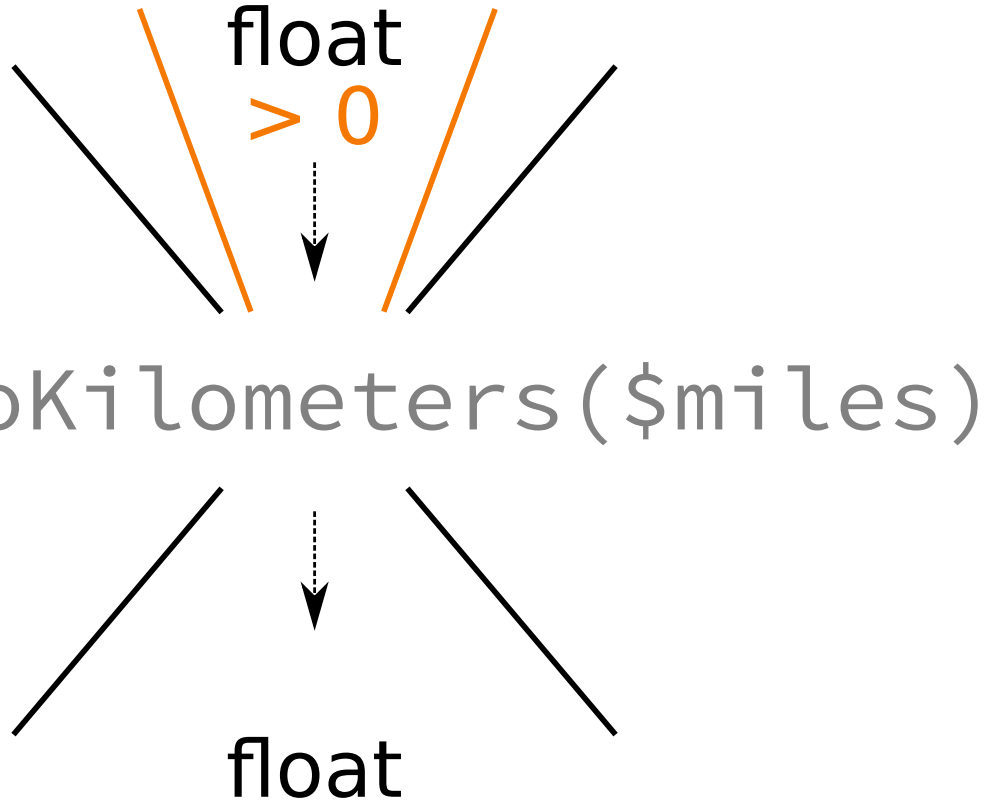
Liskov Substitution Principle

- Be less strict on input
- Be more strict on output



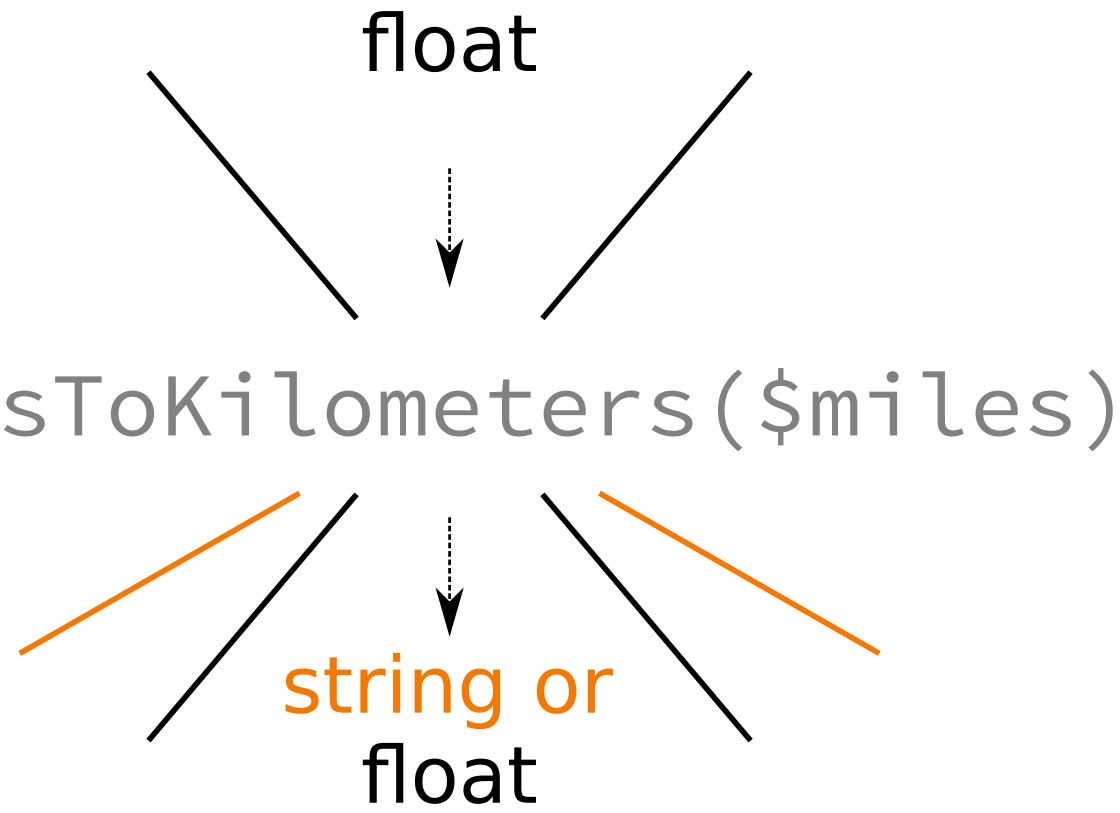
Liskov Substitution Principle

- Be less strict on input
- Be more strict on output



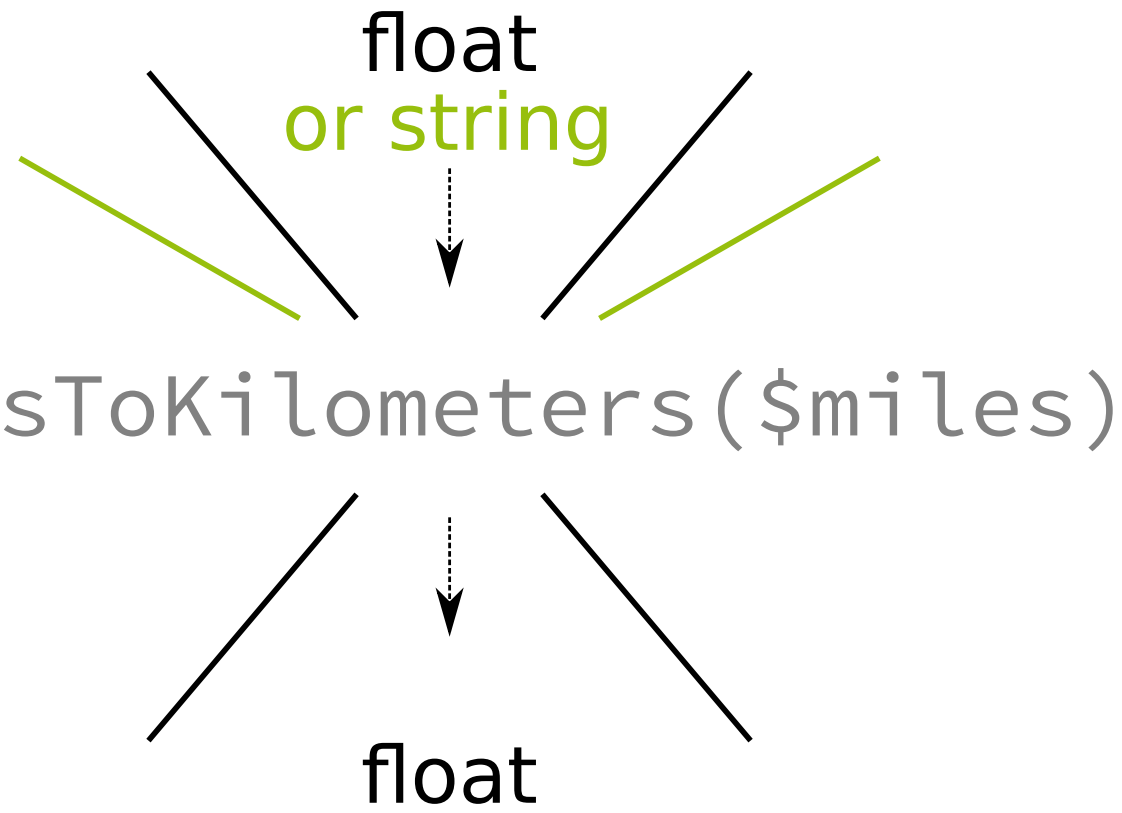
Liskov Substitution Principle

- Be less strict on input
- Be more strict on output



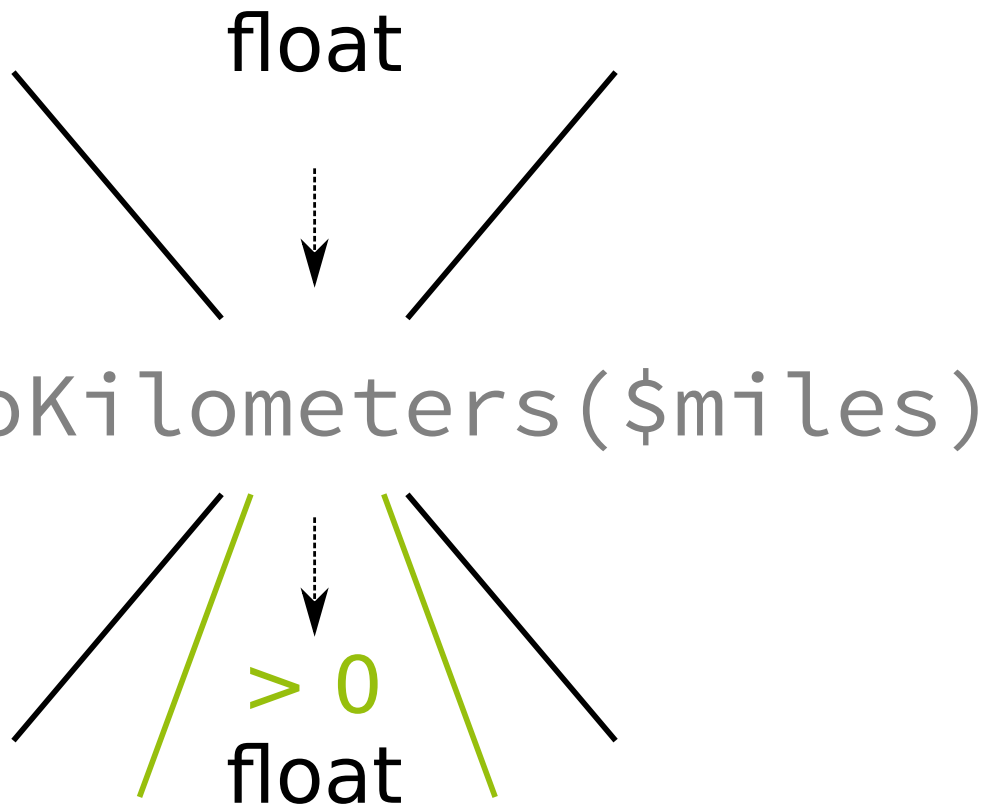
Liskov Substitution Principle

- Be less strict on input
- Be more strict on output



Liskov Substitution Principle

- Be less strict on input
- Be more strict on output



Liskov Substitution Principle

- Do not change contracts by inheritance
- Methods must work as expected in derived classes
- Users must not distinguish between super- and subclass
- Subtype polymorphism

Dependency Inversion Principle

“A. High-level modules should not depend on low level modules. Both should depend on abstractions.”

“B. Abstractions should not depend upon details. Details should depend upon abstractions.”

The Issue

```php
<?php

class WeatherLoader
{
    public function __construct(
        GoogleWeatherService $weatherService, FileLogger $logger )
    {
        $this->weatherService = $weatherService;
        $this->logger         = $logger;
    }
    public function getWeatherForLocation( Location $location )
    {
        // ...
        $this->logger->log( 'Some log message.' );
        // ...
        $this->logger->writeToFile();
    }
}
```

Doing it Right

```php
<?php

class WeatherLoader
{
    public function __construct(
        WeatherService $weatherService, Logger $logger )
    {
        $this->weatherService = $weatherService;
        $this->logger         = $logger;
    }
    public function getWeatherForLocation( Location $location )
    {
        // ...
        $this->logger->log( 'Some log message.' );
        // ...
    }
}
```

Dependency Inversion Principle

- Use abstraction to encapsulate low level modules
- Abstractions are the APIs
- Abstractions hide implementation details
- Depend on interfaces, not realizations
- Define interfaces from a usage point of view
- Finding abstractions is not easy
- Dependency Injection, anyone?

Interface Segregation Principle

“Clients should not be forced to depend upon interfaces that they do not use.”

Suboptimal

```
3   class Loader
4   {
5       public function __construct( WeatherService $weatherService, Logger $logger )
6       { /* ... */ }
7
8       public function getWeatherForLocation( Location $location )
9       { /* ... */ }
10  }
```

```
3   abstract class WeatherService
4   {
5       abstract public function getWeatherForLocation( Location $location );
6
7       abstract public function getForecastForLocation( Location $location, $weekDay );
8   }
```

The Fix

```php
interface LocationWeatherProvider
{
    function getWeatherForLocation( Location $location );
}
```

```php
abstract class WeatherService implements LocationWeatherProvider
{
    abstract public function getWeatherForLocation( Location $location );

    abstract public function getForecastForLocation( Location $location, $weekDay );
}
```

```php
class Loader
{
    public function __construct( LocationWeatherProvider $provider, Logger $logger )
    { /* ... */ }

    public function getWeatherForLocation( Location $location )
    { /* ... */ }
}
```

Interface Segregation Principle

- Avoid not needed dependencies
- Design interfaces from a usage point of view
- Do not let unnecessary functionality float in

Motivationals

Find motivational posters ”SOLID in pictures” here:

`http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/`

Thanks for Listening

Rate this talk: `https://joind.in/7565`

Stay in touch

- Tobias Schlitt
- toby@qafoo.com
- @tobySen / @qafoo

Rent a PHP quality expert:
`http://qafoo.com`