DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

Aug 20, 2017

Contents

This is a collection of known design patterns and some sample code how to implement them in PHP. Every pattern has a small list of examples (most of them from Zend Framework, Symfony2 or Doctrine2 as I'm most familiar with this software).

I think the problem with patterns is that often people do know them but don't know when to apply which.

CHAPTER 1

Patterns

The patterns can be structured in roughly three different categories. Please click on **the title of every pattern's page** for a full explanation of the pattern on Wikipedia.

Creational

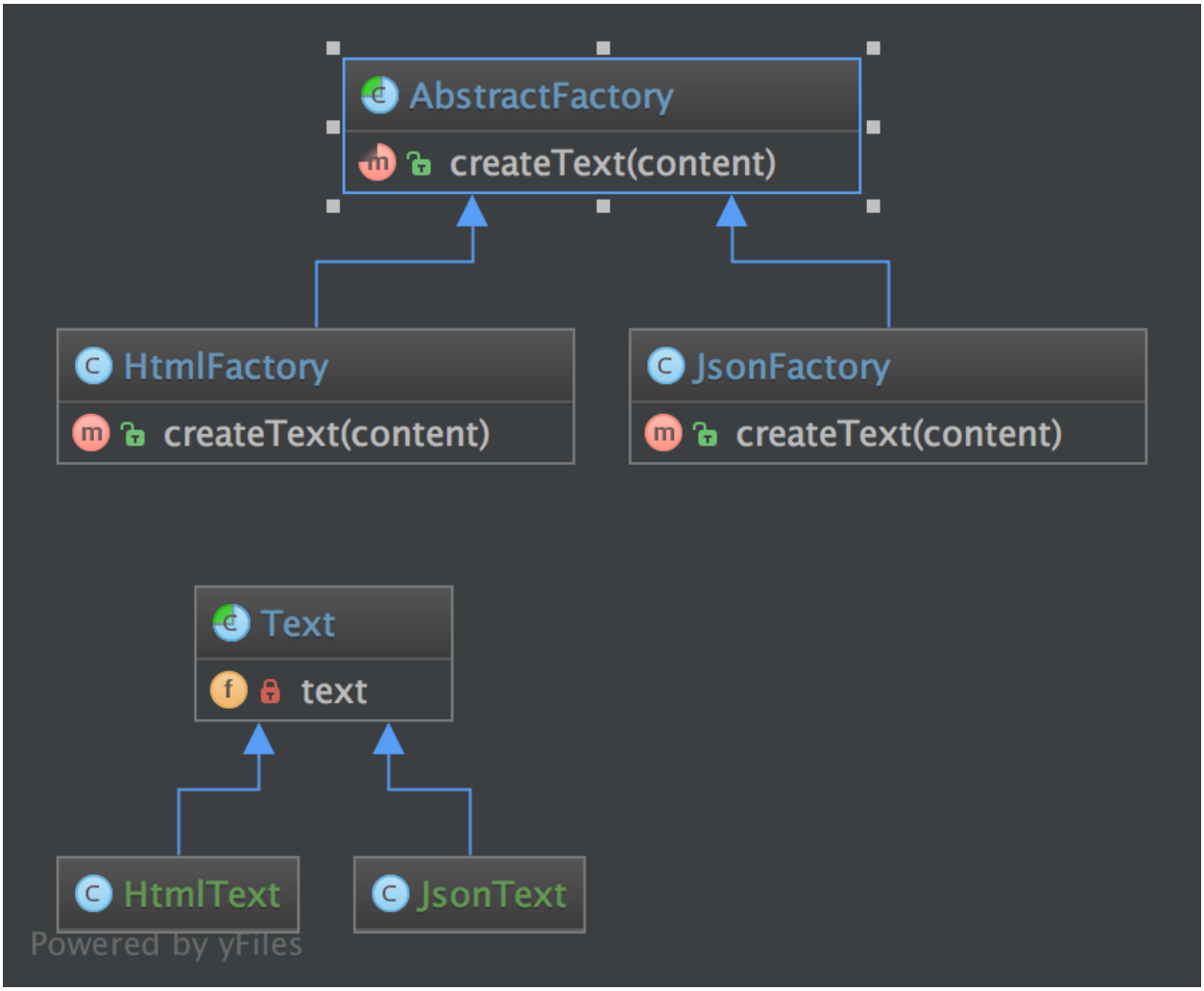
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Abstract Factory

Purpose

To create series of related or dependent objects without specifying their concrete classes. Usually the created classes all implement the same interface. The client of the abstract factory does not care about how these objects are created, he just knows how they go together.

UML Diagram



Code

You can also find this code on GitHub

AbstractFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * In this case, the abstract factory is a contract for creating some components
 * for the web. There are two ways of rendering text: HTML and JSON
 */
abstract class AbstractFactory
{
    abstract public function createText(string $content): Text;
}
```

JsonFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class JsonFactory extends AbstractFactory
{
    public function createText(string $content): Text
    {
        return new JsonText($content);
    }
}
```

HtmlFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class HtmlFactory extends AbstractFactory
{
    public function createText(string $content): Text
    {
        return new HtmlText($content);
    }
}
```

Text.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

abstract class Text
{
    /**
     * @var string
     */
    private $text;

    public function __construct(string $text)
    {
        $this->text = $text;
    }
}
```

JsonText.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class JsonText extends Text
{
    // do something here
}
```

HtmlText.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class HtmlText extends Text
{
    // do something here
}
```

Test

Tests/AbstractFactoryTest.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Tests;

use DesignPatterns\Creational\AbstractFactory\HtmlFactory;
use DesignPatterns\Creational\AbstractFactory\HtmlText;
use DesignPatterns\Creational\AbstractFactory\JsonFactory;
use DesignPatterns\Creational\AbstractFactory\JsonText;
use PHPUnit\Framework\TestCase;

class AbstractFactoryTest extends TestCase
{
    public function testCanCreateHtmlText()
    {
        $factory = new HtmlFactory();
        $text = $factory->createText('foobar');

        $this->assertInstanceOf(HtmlText::class, $text);
    }

    public function testCanCreateJsonText()
    {
        $factory = new JsonFactory();
        $text = $factory->createText('foobar');

        $this->assertInstanceOf(JsonText::class, $text);
    }
}
```

Builder

Purpose

Builder is an interface that build parts of a complex object.

Sometimes, if the builder has a better knowledge of what it builds, this interface could be an abstract class with default methods (aka adapter).

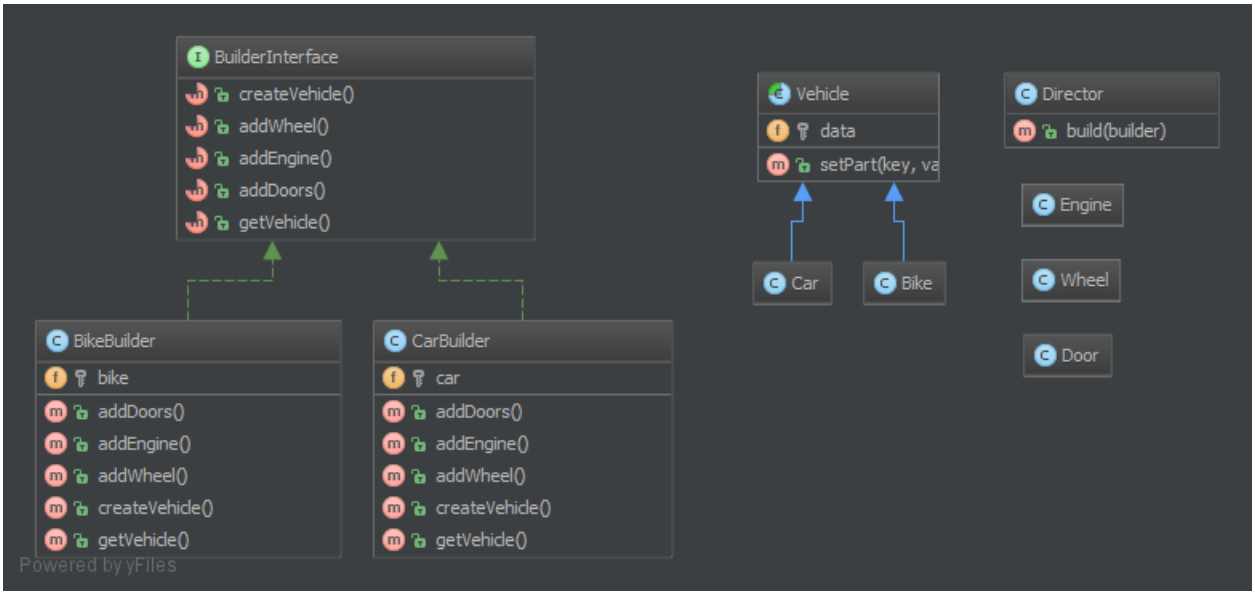
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

Note: Builders have often a fluent interface, see the mock builder of PHPUnit for example.

Examples

- PHPUnit: Mock Builder

UML Diagram



Code

You can also find this code on GitHub

Director.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

/**
 * Director is part of the builder pattern. It knows the interface of the builder
 * and builds a complex object with the help of the builder
 *
 * You can also inject many builders instead of one to build more complex objects
 */
class Director
{
    public function build(BuilderInterface $builder): Vehicle
    {
        $builder->createVehicle();
        $builder->addDoors();
        $builder->addEngine();
        $builder->addWheel();

        return $builder->getVehicle();
```

```
    }
}
```

BuilderInterface.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

interface BuilderInterface
{
    public function createVehicle();

    public function addWheel();

    public function addEngine();

    public function addDoors();

    public function getVehicle(): Vehicle;
}
```

TruckBuilder.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

class TruckBuilder implements BuilderInterface
{
    /**
     * @var Parts\Truck
     */
    private $truck;

    public function addDoors()
    {
        $this->truck->setPart('rightDoor', new Parts\Door());
        $this->truck->setPart('leftDoor', new Parts\Door());
    }

    public function addEngine()
    {
        $this->truck->setPart('truckEngine', new Parts\Engine());
    }

    public function addWheel()
    {
        $this->truck->setPart('wheel1', new Parts\Wheel());
        $this->truck->setPart('wheel2', new Parts\Wheel());
        $this->truck->setPart('wheel3', new Parts\Wheel());
        $this->truck->setPart('wheel4', new Parts\Wheel());
        $this->truck->setPart('wheel5', new Parts\Wheel());
        $this->truck->setPart('wheel6', new Parts\Wheel());
```

```
    }

    public function createVehicle()
    {
        $this->truck = new Parts\Truck();
    }

    public function getVehicle(): Vehicle
    {
        return $this->truck;
    }
}
```

CarBuilder.php

```
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

class CarBuilder implements BuilderInterface
{
    /**
     * @var Parts\Car
     */
    private $car;

    public function addDoors()
    {
        $this->car->setPart('rightDoor', new Parts\Door());
        $this->car->setPart('leftDoor', new Parts\Door());
        $this->car->setPart('trunkLid', new Parts\Door());
    }

    public function addEngine()
    {
        $this->car->setPart('engine', new Parts\Engine());
    }

    public function addWheel()
    {
        $this->car->setPart('wheelLF', new Parts\Wheel());
        $this->car->setPart('wheelRF', new Parts\Wheel());
        $this->car->setPart('wheelLR', new Parts\Wheel());
        $this->car->setPart('wheelRR', new Parts\Wheel());
    }

    public function createVehicle()
    {
        $this->car = new Parts\Car();
    }

    public function getVehicle(): Vehicle
    {
        return $this->car;
    }
}
```

Parts/Vehicle.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

abstract class Vehicle
{
    /**
     * @var object[]
     */
    private $data = [];

    /**
     * @param string $key
     * @param object $value
     */
    public function setPart($key, $value)
    {
        $this->data[$key] = $value;
    }
}
```

Parts/Truck.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Truck extends Vehicle
{
}
```

Parts/Car.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Car extends Vehicle
{
}
```

Parts/Engine.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Engine
{
}
```

Parts/Wheel.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

```

```php
class Wheel
{
}
```

Parts/Door.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

Test

Tests/DirectorTest.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Tests;

use DesignPatterns\Creational\Builder\Parts\Car;
use DesignPatterns\Creational\Builder\Parts\Truck;
use DesignPatterns\Creational\Builder\TruckBuilder;
use DesignPatterns\Creational\Builder\CarBuilder;
use DesignPatterns\Creational\Builder\Director;
use PHPUnit\Framework\TestCase;

class DirectorTest extends TestCase
{
    public function testCanBuildTruck()
    {
        $truckBuilder = new TruckBuilder();
        $newVehicle = (new Director())->build($truckBuilder);

        $this->assertInstanceOf(Truck::class, $newVehicle);
    }

    public function testCanBuildCar()
    {
        $carBuilder = new CarBuilder();
        $newVehicle = (new Director())->build($carBuilder);

        $this->assertInstanceOf(Car::class, $newVehicle);
    }
}
```

Factory Method

Purpose

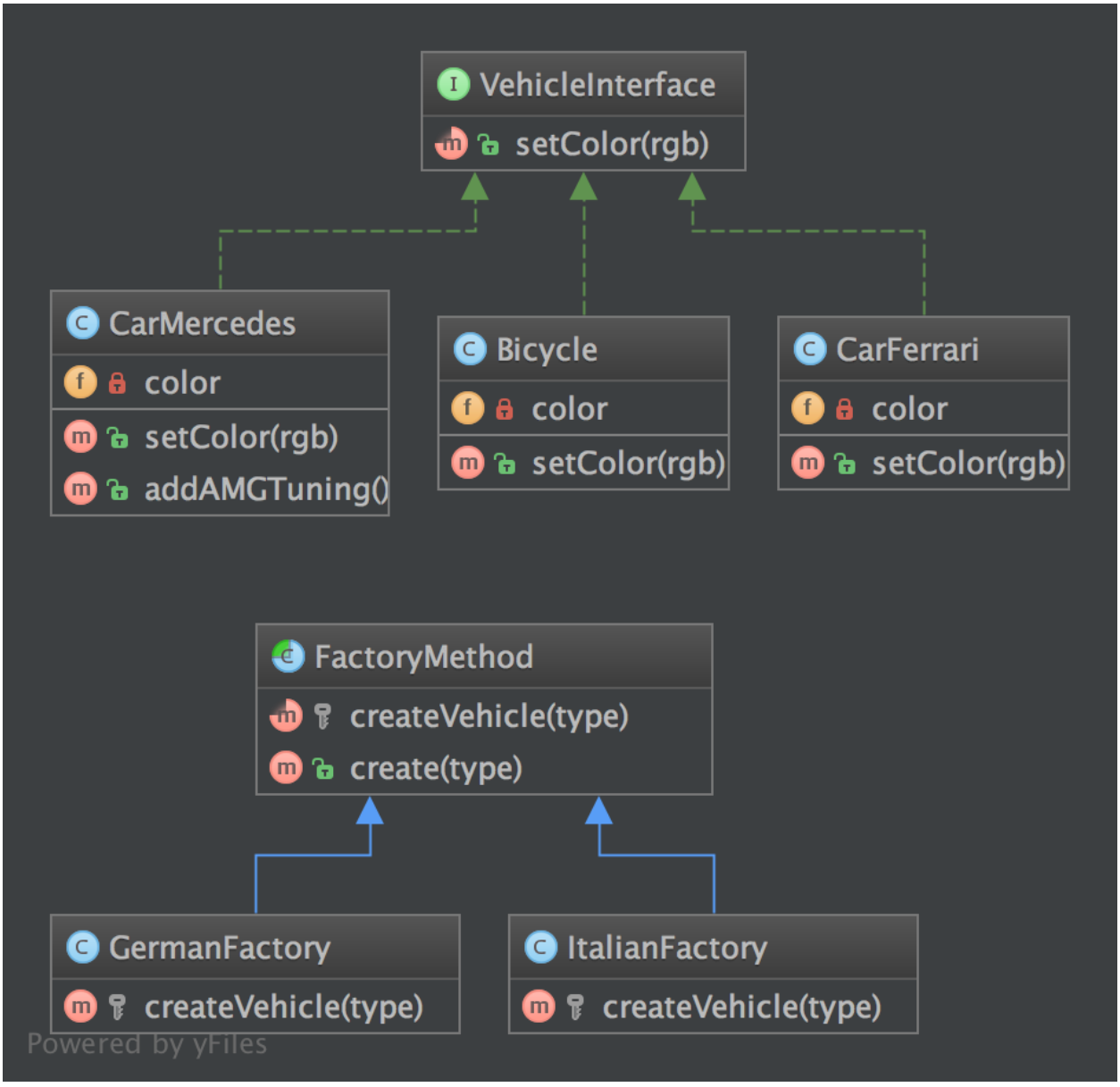
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects

For simple case, this abstract class could be just an interface

This pattern is a "real" Design Pattern because it achieves the "Dependency Inversion Principle" a.k.a the "D" in S.O.L.I.D principles.

It means the FactoryMethod class depends on abstractions, not concrete classes. This is the real trick compared to SimpleFactory or StaticFactory.

UML Diagram



Code

You can also find this code on GitHub

FactoryMethod.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

abstract class FactoryMethod
{
    const CHEAP = 'cheap';
    const FAST = 'fast';

    abstract protected function createVehicle(string $type): VehicleInterface;

    public function create(string $type): VehicleInterface
    {
        $obj = $this->createVehicle($type);
        $obj->setColor('black');

        return $obj;
    }
}
```

ItalianFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class ItalianFactory extends FactoryMethod
{
    protected function createVehicle(string $type): VehicleInterface
    {
        switch ($type) {
            case parent::CHEAP:
                return new Bicycle();
            case parent::FAST:
                return new CarFerrari();
            default:
                throw new \InvalidArgumentException("$type is not a valid vehicle");
        }
    }
}
```

GermanFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class GermanFactory extends FactoryMethod
{
    protected function createVehicle(string $type): VehicleInterface
    {
        switch ($type) {
            case parent::CHEAP:
                return new Bicycle();
            case parent::FAST:
                $carMercedes = new CarMercedes();
                // we can specialize the way we want some concrete Vehicle since we
↪know the class
```

```
                $carMercedes->addAMGTuning();

                return $carMercedes;
            default:
                throw new \InvalidArgumentException("$type is not a valid vehicle");
        }
    }
}
```

VehicleInterface.php

```
<?php

namespace DesignPatterns\Creational\FactoryMethod;

interface VehicleInterface
{
    public function setColor(string $rgb);
}
```

CarMercedes.php

```
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class CarMercedes implements VehicleInterface
{
    /**
     * @var string
     */
    private $color;

    public function setColor(string $rgb)
    {
        $this->color = $rgb;
    }

    public function addAMGTuning()
    {
        // do additional tuning here
    }
}
```

CarFerrari.php

```
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class CarFerrari implements VehicleInterface
{
    /**
     * @var string
     */
    private $color;

    public function setColor(string $rgb)
```

```
    {
        $this->color = $rgb;
    }
}
```

Bicycle.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

class Bicycle implements VehicleInterface
{
    /**
     * @var string
     */
    private $color;

    public function setColor(string $rgb)
    {
        $this->color = $rgb;
    }
}
```

Test

Tests/FactoryMethodTest.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod\Tests;

use DesignPatterns\Creational\FactoryMethod\Bicycle;
use DesignPatterns\Creational\FactoryMethod\CarFerrari;
use DesignPatterns\Creational\FactoryMethod\CarMercedes;
use DesignPatterns\Creational\FactoryMethod\FactoryMethod;
use DesignPatterns\Creational\FactoryMethod\GermanFactory;
use DesignPatterns\Creational\FactoryMethod\ItalianFactory;
use PHPUnit\Framework\TestCase;

class FactoryMethodTest extends TestCase
{
    public function testCanCreateCheapVehicleInGermany()
    {
        $factory = new GermanFactory();
        $result = $factory->create(FactoryMethod::CHEAP);

        $this->assertInstanceOf(Bicycle::class, $result);
    }

    public function testCanCreateFastVehicleInGermany()
    {
        $factory = new GermanFactory();
        $result = $factory->create(FactoryMethod::FAST);

        $this->assertInstanceOf(CarMercedes::class, $result);
```

```
29      }
30
31      public function testCanCreateCheapVehicleInItaly()
32      {
33          $factory = new ItalianFactory();
34          $result = $factory->create(FactoryMethod::CHEAP);
35
36          $this->assertInstanceOf(Bicycle::class, $result);
37      }
38
39      public function testCanCreateFastVehicleInItaly()
40      {
41          $factory = new ItalianFactory();
42          $result = $factory->create(FactoryMethod::FAST);
43
44          $this->assertInstanceOf(CarFerrari::class, $result);
45      }
46
47      /**
48       * @expectedException \InvalidArgumentException
49       * @expectedExceptionMessage spaceship is not a valid vehicle
50       */
51      public function testUnknownType()
52      {
53          (new ItalianFactory())->create('spaceship');
54      }
55  }
```

Multiton

THIS IS CONSIDERED TO BE AN ANTI-PATTERN! FOR BETTER TESTABILITY AND MAINTAINABILITY USE DEPENDENCY INJECTION!

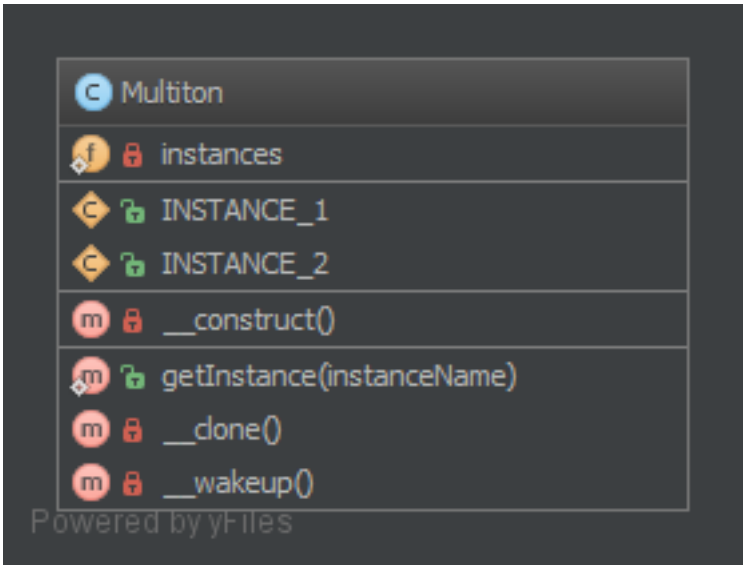
Purpose

To have only a list of named instances that are used, like a singleton but with n instances.

Examples

- 2 DB Connectors, e.g. one for MySQL, the other for SQLite
- multiple Loggers (one for debug messages, one for errors)

UML Diagram



Code

You can also find this code on GitHub

Multiton.php

```php
<?php

namespace DesignPatterns\Creational\Multiton;

final class Multiton
{
    const INSTANCE_1 = '1';
    const INSTANCE_2 = '2';

    /**
     * @var Multiton[]
     */
    private static $instances = [];

    /**
     * this is private to prevent from creating arbitrary instances
     */
    private function __construct()
    {
    }

    public static function getInstance(string $instanceName): Multiton
    {
        if (!isset(self::$instances[$instanceName])) {
            self::$instances[$instanceName] = new self();
        }

        return self::$instances[$instanceName];
    }
```

```
30
31      /**
32       * prevent instance from being cloned
33       */
34      private function __clone()
35      {
36      }
37
38      /**
39       * prevent instance from being unserialized
40       */
41      private function __wakeup()
42      {
43      }
44  }
```

Test

Pool

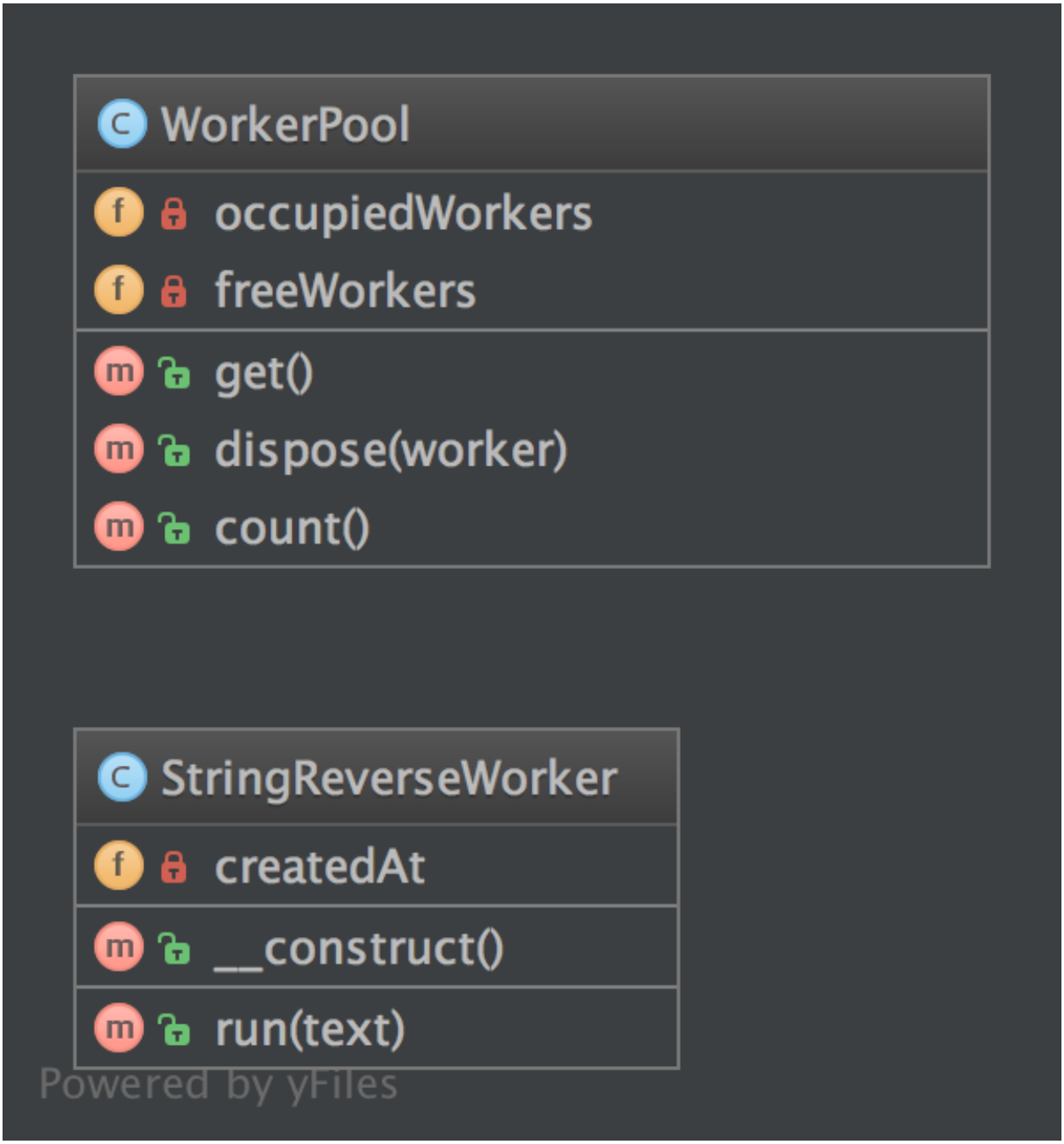
Purpose

The **object pool pattern** is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when creation of the new objects (especially over network) may take variable time.

However these benefits are mostly true for objects that are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps. In certain situations, simple object pooling (that hold no external resources, but only occupy memory) may not be efficient and could decrease performance.

UML Diagram



Code

You can also find this code on GitHub

WorkerPool.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Pool;
```

```php

class WorkerPool implements \Countable
{
    /**
     * @var StringReverseWorker[]
     */
    private $occupiedWorkers = [];

    /**
     * @var StringReverseWorker[]
     */
    private $freeWorkers = [];

    public function get(): StringReverseWorker
    {
        if (count($this->freeWorkers) == 0) {
            $worker = new StringReverseWorker();
        } else {
            $worker = array_pop($this->freeWorkers);
        }

        $this->occupiedWorkers[spl_object_hash($worker)] = $worker;

        return $worker;
    }

    public function dispose(StringReverseWorker $worker)
    {
        $key = spl_object_hash($worker);

        if (isset($this->occupiedWorkers[$key])) {
            unset($this->occupiedWorkers[$key]);
            $this->freeWorkers[$key] = $worker;
        }
    }

    public function count(): int
    {
        return count($this->occupiedWorkers) + count($this->freeWorkers);
    }
}
```

StringReverseWorker.php

```php
<?php

namespace DesignPatterns\Creational\Pool;

class StringReverseWorker
{
    /**
     * @var \DateTime
     */
    private $createdAt;

    public function __construct()
    {
        $this->createdAt = new \DateTime();
```

```
    }

    public function run(string $text)
    {
        return strrev($text);
    }
}
```

Test

Tests/PoolTest.php

```
<?php

namespace DesignPatterns\Creational\Pool\Tests;

use DesignPatterns\Creational\Pool\WorkerPool;
use PHPUnit\Framework\TestCase;

class PoolTest extends TestCase
{
    public function testCanGetNewInstancesWithGet()
    {
        $pool = new WorkerPool();
        $worker1 = $pool->get();
        $worker2 = $pool->get();

        $this->assertCount(2, $pool);
        $this->assertNotSame($worker1, $worker2);
    }

    public function testCanGetSameInstanceTwiceWhenDisposingItFirst()
    {
        $pool = new WorkerPool();
        $worker1 = $pool->get();
        $pool->dispose($worker1);
        $worker2 = $pool->get();

        $this->assertCount(1, $pool);
        $this->assertSame($worker1, $worker2);
    }
}
```

Prototype

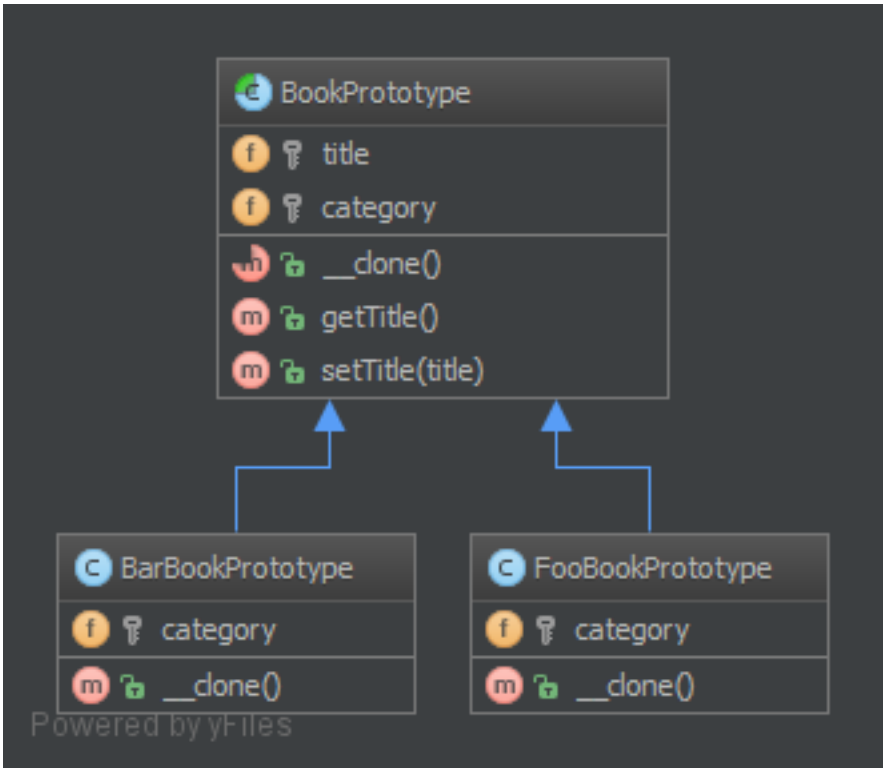
Purpose

To avoid the cost of creating objects the standard way (new Foo()) and instead create a prototype and clone it.

Examples

- Large amounts of data (e.g. create 1,000,000 rows in a database at once via a ORM).

UML Diagram



Code

You can also find this code on GitHub

BookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

abstract class BookPrototype
{
    /**
     * @var string
     */
    protected $title;

    /**
     * @var string
     */
    protected $category;

    abstract public function __clone();

    public function getTitle(): string
    {
        return $this->title;
    }
```

```
23
24      public function setTitle($title)
25      {
26          $this->title = $title;
27      }
28  }
```

BarBookPrototype.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\Prototype;
4
5   class BarBookPrototype extends BookPrototype
6   {
7       /**
8        * @var string
9        */
10      protected $category = 'Bar';
11
12      public function __clone()
13      {
14      }
15  }
```

FooBookPrototype.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\Prototype;
4
5   class FooBookPrototype extends BookPrototype
6   {
7       /**
8        * @var string
9        */
10      protected $category = 'Foo';
11
12      public function __clone()
13      {
14      }
15  }
```

Test

Tests/PrototypeTest.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\Prototype\Tests;
4
5   use DesignPatterns\Creational\Prototype\BarBookPrototype;
6   use DesignPatterns\Creational\Prototype\FooBookPrototype;
7   use PHPUnit\Framework\TestCase;
8
9   class PrototypeTest extends TestCase
```

```
{
    public function testCanGetFooBook()
    {
        $fooPrototype = new FooBookPrototype();
        $barPrototype = new BarBookPrototype();

        for ($i = 0; $i < 10; $i++) {
            $book = clone $fooPrototype;
            $book->setTitle('Foo Book No ' . $i);
            $this->assertInstanceOf(FooBookPrototype::class, $book);
        }

        for ($i = 0; $i < 5; $i++) {
            $book = clone $barPrototype;
            $book->setTitle('Bar Book No ' . $i);
            $this->assertInstanceOf(BarBookPrototype::class, $book);
        }
    }
}
```

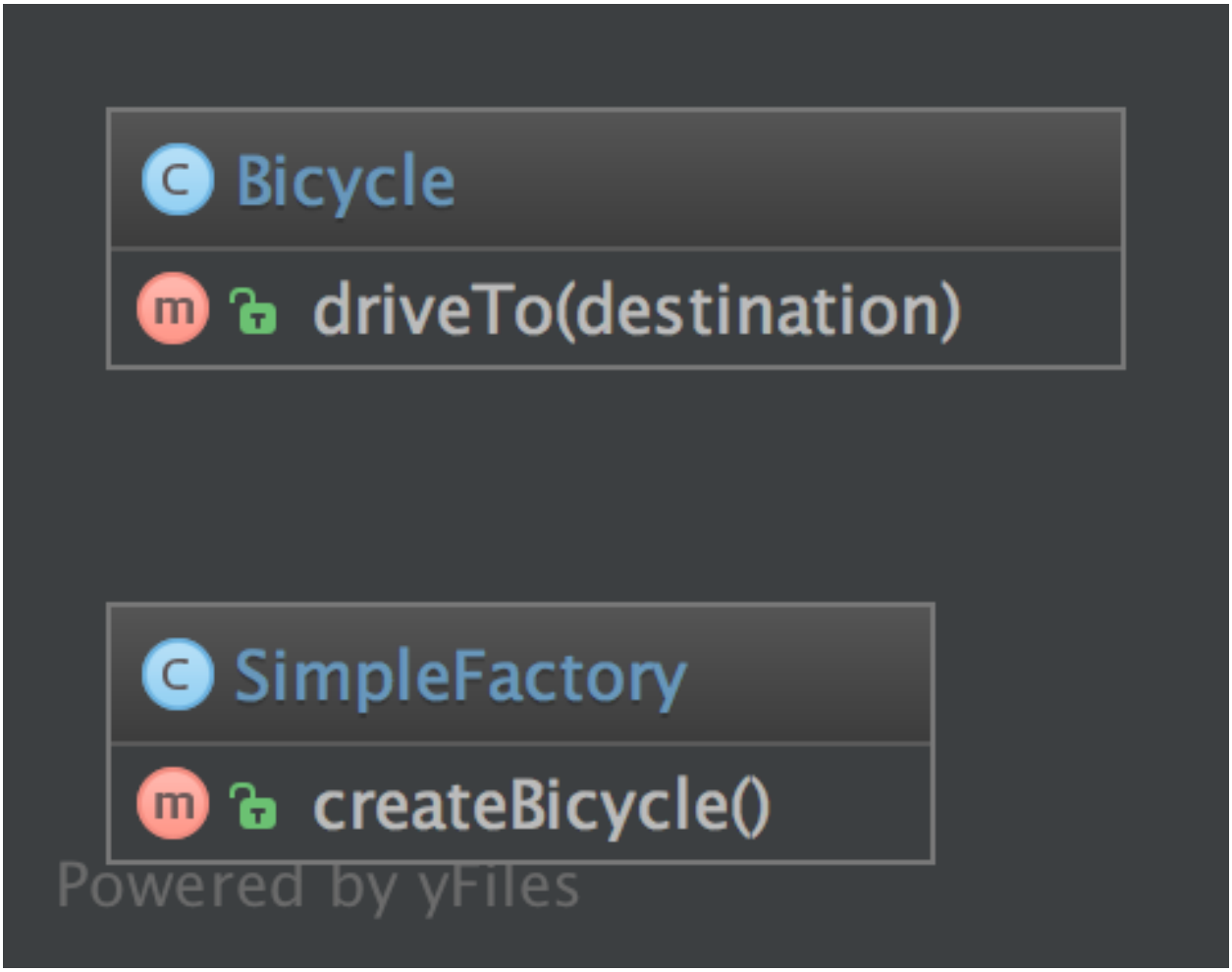
Simple Factory

Purpose

SimpleFactory is a simple factory pattern.

It differs from the static factory because it is not static. Therefore, you can have multiple factories, differently parametrized, you can subclass it and you can mock it. It always should be preferred over a static factory!

UML Diagram



Code

You can also find this code on GitHub

SimpleFactory.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

class SimpleFactory
{
    public function createBicycle(): Bicycle
    {
        return new Bicycle();
    }
}
```

Bicycle.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

class Bicycle
{
    public function driveTo(string $destination)
    {
    }
}
```

Usage

```php
 $factory = new SimpleFactory();
 $bicycle = $factory->createBicycle();
 $bicycle->driveTo('Paris');
```

Test

Tests/SimpleFactoryTest.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory\Tests;

use DesignPatterns\Creational\SimpleFactory\Bicycle;
use DesignPatterns\Creational\SimpleFactory\SimpleFactory;
use PHPUnit\Framework\TestCase;

class SimpleFactoryTest extends TestCase
{
    public function testCanCreateBicycle()
    {
        $bicycle = (new SimpleFactory())->createBicycle();
        $this->assertInstanceOf(Bicycle::class, $bicycle);
    }
}
```

Singleton

THIS IS CONSIDERED TO BE AN ANTI-PATTERN! FOR BETTER TESTABILITY AND MAINTAINABILITY USE DEPENDENCY INJECTION!

Purpose

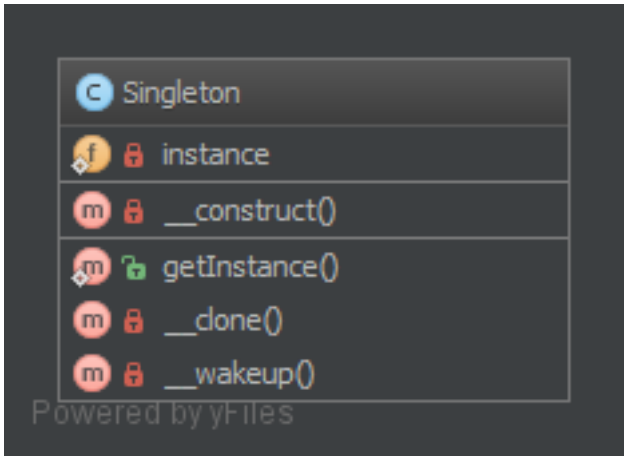
To have only one instance of this object in the application that will handle all calls.

Examples

- DB Connector

- Logger (may also be a Multiton if there are many log files for several purposes)
- Lock file for the application (there is only one in the filesystem ...)

UML Diagram



Code

You can also find this code on GitHub

Singleton.php

```php
<?php

namespace DesignPatterns\Creational\Singleton;

final class Singleton
{
    /**
     * @var Singleton
     */
    private static $instance;

    /**
     * gets the instance via lazy initialization (created on first usage)
     */
    public static function getInstance(): Singleton
    {
        if (null === static::$instance) {
            static::$instance = new static();
        }

        return static::$instance;
    }

    /**
     * is not allowed to call from outside to prevent from creating multiple instances,
     * to use the singleton, you have to obtain the instance from Singleton::getInstance() instead
```

```php
     */
    private function __construct()
    {
    }

    /**
     * prevent the instance from being cloned (which would create a second instance of it)
     */
    private function __clone()
    {
    }

    /**
     * prevent from being unserialized (which would create a second instance of it)
     */
    private function __wakeup()
    {
    }
}
```

Test

Tests/SingletonTest.php

```php
<?php

namespace DesignPatterns\Creational\Singleton\Tests;

use DesignPatterns\Creational\Singleton\Singleton;
use PHPUnit\Framework\TestCase;

class SingletonTest extends TestCase
{
    public function testUniqueness()
    {
        $firstCall = Singleton::getInstance();
        $secondCall = Singleton::getInstance();

        $this->assertInstanceOf(Singleton::class, $firstCall);
        $this->assertSame($firstCall, $secondCall);
    }
}
```

Static Factory

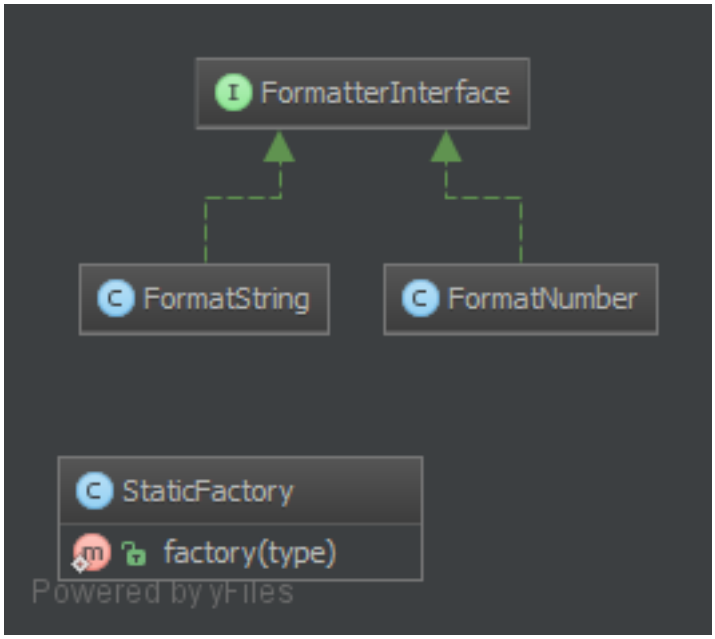
Purpose

Similar to the AbstractFactory, this pattern is used to create series of related or dependent objects. The difference between this and the abstract factory pattern is that the static factory pattern uses just one static method to create all types of objects it can create. It is usually named `factory` or `build`.

Examples

- Zend Framework: `Zend_Cache_Backend` or `_Frontend` use a factory method create cache backends or frontends

UML Diagram



Code

You can also find this code on GitHub

StaticFactory.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

/**
 * Note1: Remember, static means global state which is evil because it can't be mocked for tests
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
final class StaticFactory
{
    /**
     * @param string $type
     *
     * @return FormatterInterface
     */
    public static function factory(string $type): FormatterInterface
    {
        if ($type == 'number') {
            return new FormatNumber();
        }
```

```
21
22          if ($type == 'string') {
23              return new FormatString();
24          }
25
26          throw new \InvalidArgumentException('Unknown format given');
27      }
28  }
```

FormatterInterface.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   interface FormatterInterface
6   {
7   }
```

FormatString.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   class FormatString implements FormatterInterface
6   {
7   }
```

FormatNumber.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   class FormatNumber implements FormatterInterface
6   {
7   }
```

Test

Tests/StaticFactoryTest.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory\Tests;
4
5   use DesignPatterns\Creational\StaticFactory\StaticFactory;
6   use PHPUnit\Framework\TestCase;
7
8   class StaticFactoryTest extends TestCase
9   {
10      public function testCanCreateNumberFormatter()
11      {
12          $this->assertInstanceOf(
13              'DesignPatterns\Creational\StaticFactory\FormatNumber',
```

```
            StaticFactory::factory('number')
        );
    }

    public function testCanCreateStringFormatter()
    {
        $this->assertInstanceOf(
            'DesignPatterns\Creational\StaticFactory\FormatString',
            StaticFactory::factory('string')
        );
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testException()
    {
        StaticFactory::factory('object');
    }
}
```

Structural

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

Adapter / Wrapper

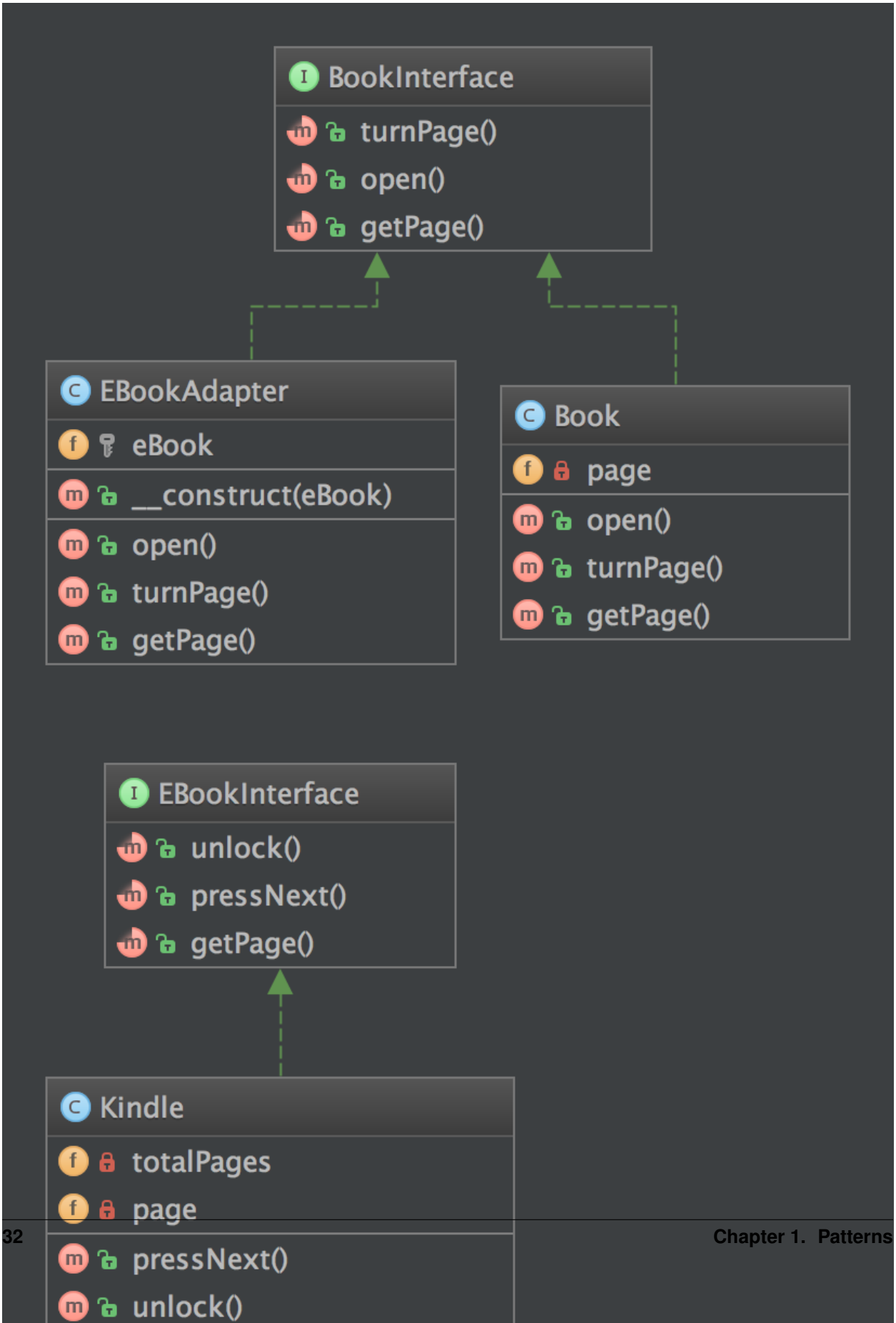
Purpose

To translate one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces by providing its interface to clients while using the original interface.

Examples

- DB Client libraries adapter
- using multiple different webservices and adapters normalize data so that the outcome is the same for all

UML Diagram


BookInterface
turnPage()
open()
getPage()
EBookAdapter
eBook
__construct(eBook)
open()
turnPage()
getPage()
Book
page
open()
turnPage()
getPage()
EBookInterface
unlock()
pressNext()
getPage()
Kindle
totalPages
page
pressNext()
unlock()

Code

You can also find this code on GitHub

BookInterface.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

interface BookInterface
{
    public function turnPage();

    public function open();

    public function getPage(): int;
}
```

Book.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

class Book implements BookInterface
{
    /**
     * @var int
     */
    private $page;

    public function open()
    {
        $this->page = 1;
    }

    public function turnPage()
    {
        $this->page++;
    }

    public function getPage(): int
    {
        return $this->page;
    }
}
```

EBookAdapter.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * This is the adapter here. Notice it implements BookInterface,
 * therefore you don't have to change the code of the client which is using a Book
 */
class EBookAdapter implements BookInterface
```

```
{
    /**
     * @var EBookInterface
     */
    protected $eBook;

    /**
     * @param EBookInterface $eBook
     */
    public function __construct(EBookInterface $eBook)
    {
        $this->eBook = $eBook;
    }

    /**
     * This class makes the proper translation from one interface to another.
     */
    public function open()
    {
        $this->eBook->unlock();
    }

    public function turnPage()
    {
        $this->eBook->pressNext();
    }

    /**
     * notice the adapted behavior here: EBookInterface::getPage() will return two integers, but BookInterface
     * supports only a current page getter, so we adapt the behavior here
     *
     * @return int
     */
    public function getPage(): int
    {
        return $this->eBook->getPage()[0];
    }
}
```

EBookInterface.php

```
<?php

namespace DesignPatterns\Structural\Adapter;

interface EBookInterface
{
    public function unlock();

    public function pressNext();

    /**
     * returns current page and total number of pages, like [10, 100] is page 10 of 100
     *
     * @return int[]
     */
```

```
    public function getPage(): array;
}
```

Kindle.php

```
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * this is the adapted class. In production code, this could be a class from another package, some vendor code.
 * Notice that it uses another naming scheme and the implementation does something similar but in another way
 */
class Kindle implements EBookInterface
{
    /**
     * @var int
     */
    private $page = 1;

    /**
     * @var int
     */
    private $totalPages = 100;

    public function pressNext()
    {
        $this->page++;
    }

    public function unlock()
    {
    }

    /**
     * returns current page and total number of pages, like [10, 100] is page 10 of 100
     *
     * @return int[]
     */
    public function getPage(): array
    {
        return [$this->page, $this->totalPages];
    }
}
```

Test

Tests/AdapterTest.php

```
<?php

namespace DesignPatterns\Structural\Adapter\Tests;

```

```php
use DesignPatterns\Structural\Adapter\Book;
use DesignPatterns\Structural\Adapter\EBookAdapter;
use DesignPatterns\Structural\Adapter\Kindle;
use PHPUnit\Framework\TestCase;

class AdapterTest extends TestCase
{
    public function testCanTurnPageOnBook()
    {
        $book = new Book();
        $book->open();
        $book->turnPage();

        $this->assertEquals(2, $book->getPage());
    }

    public function testCanTurnPageOnKindleLikeInANormalBook()
    {
        $kindle = new Kindle();
        $book = new EBookAdapter($kindle);

        $book->open();
        $book->turnPage();

        $this->assertEquals(2, $book->getPage());
    }
}
```

Bridge

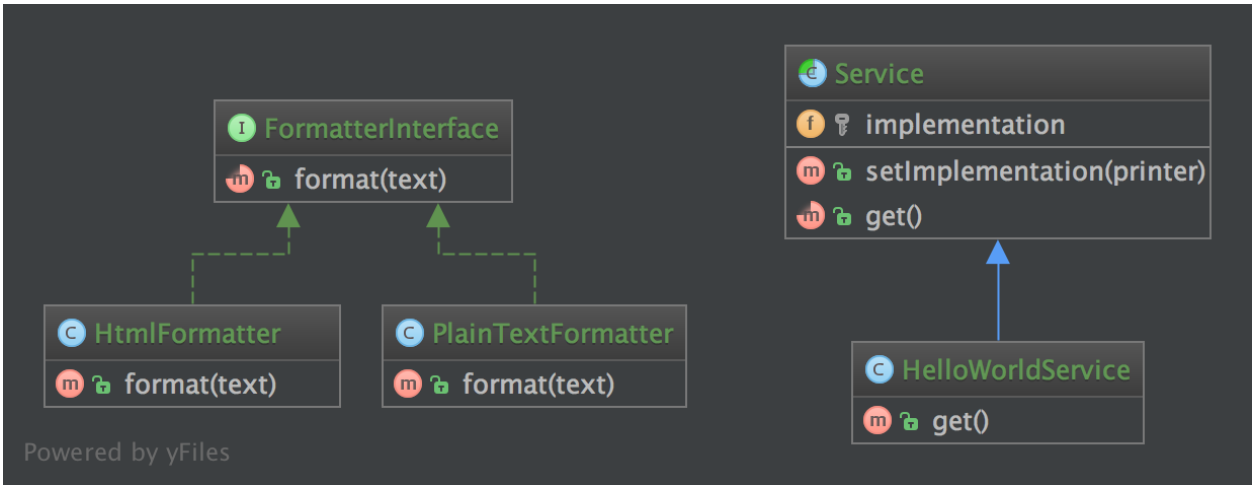
Purpose

Decouple an abstraction from its implementation so that the two can vary independently.

Examples

- Symfony DoctrineBridge

UML Diagram



Code

You can also find this code on GitHub

FormatterInterface.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

interface FormatterInterface
{
    public function format(string $text);
}
```

PlainTextFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

class PlainTextFormatter implements FormatterInterface
{
    public function format(string $text)
    {
        return $text;
    }
}
```

HtmlFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

class HtmlFormatter implements FormatterInterface
{
    public function format(string $text)
```

```
    {
        return sprintf('<p>%s</p>', $text);
    }
}
```

Service.php

```
<?php

namespace DesignPatterns\Structural\Bridge;

abstract class Service
{
    /**
     * @var FormatterInterface
     */
    protected $implementation;

    /**
     * @param FormatterInterface $printer
     */
    public function __construct(FormatterInterface $printer)
    {
        $this->implementation = $printer;
    }

    /**
     * @param FormatterInterface $printer
     */
    public function setImplementation(FormatterInterface $printer)
    {
        $this->implementation = $printer;
    }

    abstract public function get();
}
```

HelloWorldService.php

```
<?php

namespace DesignPatterns\Structural\Bridge;

class HelloWorldService extends Service
{
    public function get()
    {
        return $this->implementation->format('Hello World');
    }
}
```

Test

Tests/BridgeTest.php

```php
<?php

namespace DesignPatterns\Structural\Bridge\Tests;

use DesignPatterns\Structural\Bridge\HelloWorldService;
use DesignPatterns\Structural\Bridge\HtmlFormatter;
use DesignPatterns\Structural\Bridge\PlainTextFormatter;
use PHPUnit\Framework\TestCase;

class BridgeTest extends TestCase
{
    public function testCanPrintUsingThePlainTextPrinter()
    {
        $service = new HelloWorldService(new PlainTextFormatter());
        $this->assertEquals('Hello World', $service->get());

        // now change the implementation and use the HtmlFormatter instead
        $service->setImplementation(new HtmlFormatter());
        $this->assertEquals('<p>Hello World</p>', $service->get());
    }
}
```

Composite

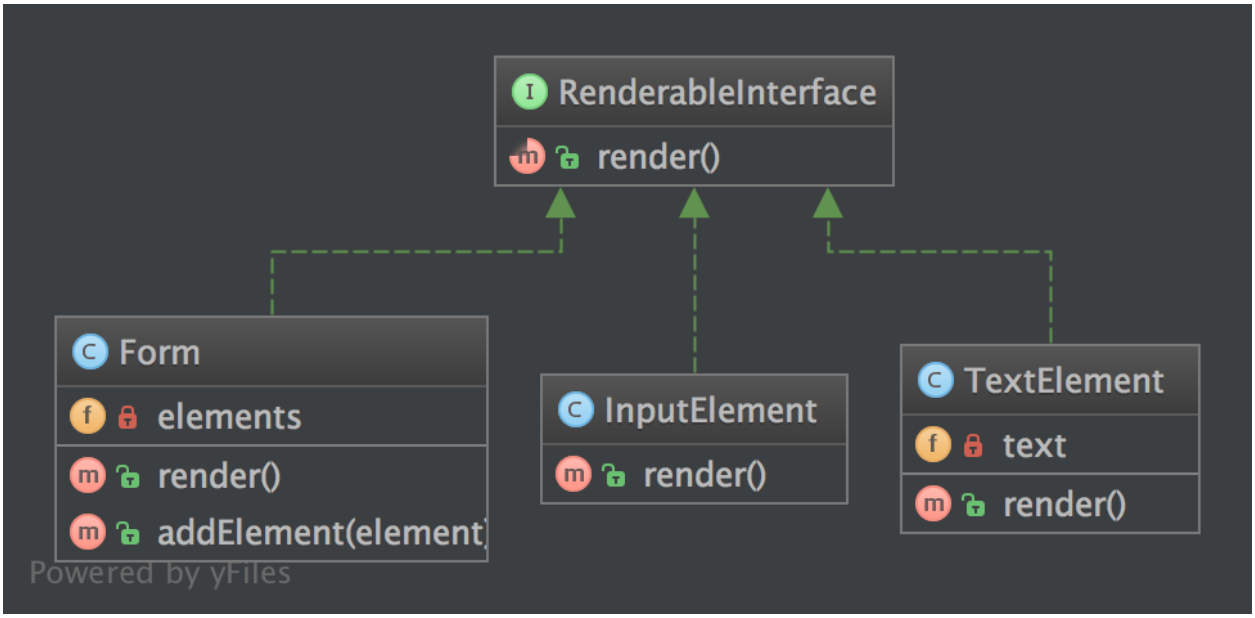
Purpose

To treat a group of objects the same way as a single instance of the object.

Examples

- a form class instance handles all its form elements like a single instance of the form, when `render()` is called, it subsequently runs through all its child elements and calls `render()` on them
- `Zend_Config`: a tree of configuration options, each one is a `Zend_Config` object itself

UML Diagram



Code

You can also find this code on GitHub

RenderableInterface.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

interface RenderableInterface
{
    public function render(): string;
}
```

Form.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

/**
 * The composite node MUST extend the component contract. This is mandatory for␣
↪building
 * a tree of components.
 */
class Form implements RenderableInterface
{
    /**
     * @var RenderableInterface[]
     */
    private $elements;

```

```php
    /**
     * runs through all elements and calls render() on them, then returns the complete representation
     * of the form.
     *
     * from the outside, one will not see this and the form will act like a single object instance
     *
     * @return string
     */
    public function render(): string
    {
        $formCode = '<form>';

        foreach ($this->elements as $element) {
            $formCode .= $element->render();
        }

        $formCode .= '</form>';

        return $formCode;
    }

    /**
     * @param RenderableInterface $element
     */
    public function addElement(RenderableInterface $element)
    {
        $this->elements[] = $element;
    }
}
```

InputElement.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

class InputElement implements RenderableInterface
{
    public function render(): string
    {
        return '<input type="text" />';
    }
}
```

TextElement.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

class TextElement implements RenderableInterface
{
    /**
     * @var string
     */
    private $text;
```

```php

    public function __construct(string $text)
    {
        $this->text = $text;
    }

    public function render(): string
    {
        return $this->text;
    }
}
```

Test

Tests/CompositeTest.php

```php
<?php

namespace DesignPatterns\Structural\Composite\Tests;

use DesignPatterns\Structural\Composite;
use PHPUnit\Framework\TestCase;

class CompositeTest extends TestCase
{
    public function testRender()
    {
        $form = new Composite\Form();
        $form->addElement(new Composite\TextElement('Email:'));
        $form->addElement(new Composite\InputElement());
        $embed = new Composite\Form();
        $embed->addElement(new Composite\TextElement('Password:'));
        $embed->addElement(new Composite\InputElement());
        $form->addElement($embed);

        // This is just an example, in a real world scenario it is important to␣
↪remember that web browsers do not
        // currently support nested forms

        $this->assertEquals(
            '<form>Email:<input type="text" /><form>Password:<input type="text" /></
↪form></form>',
            $form->render()
        );
    }
}
```

Data Mapper

Purpose

A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent of each other and the data mapper

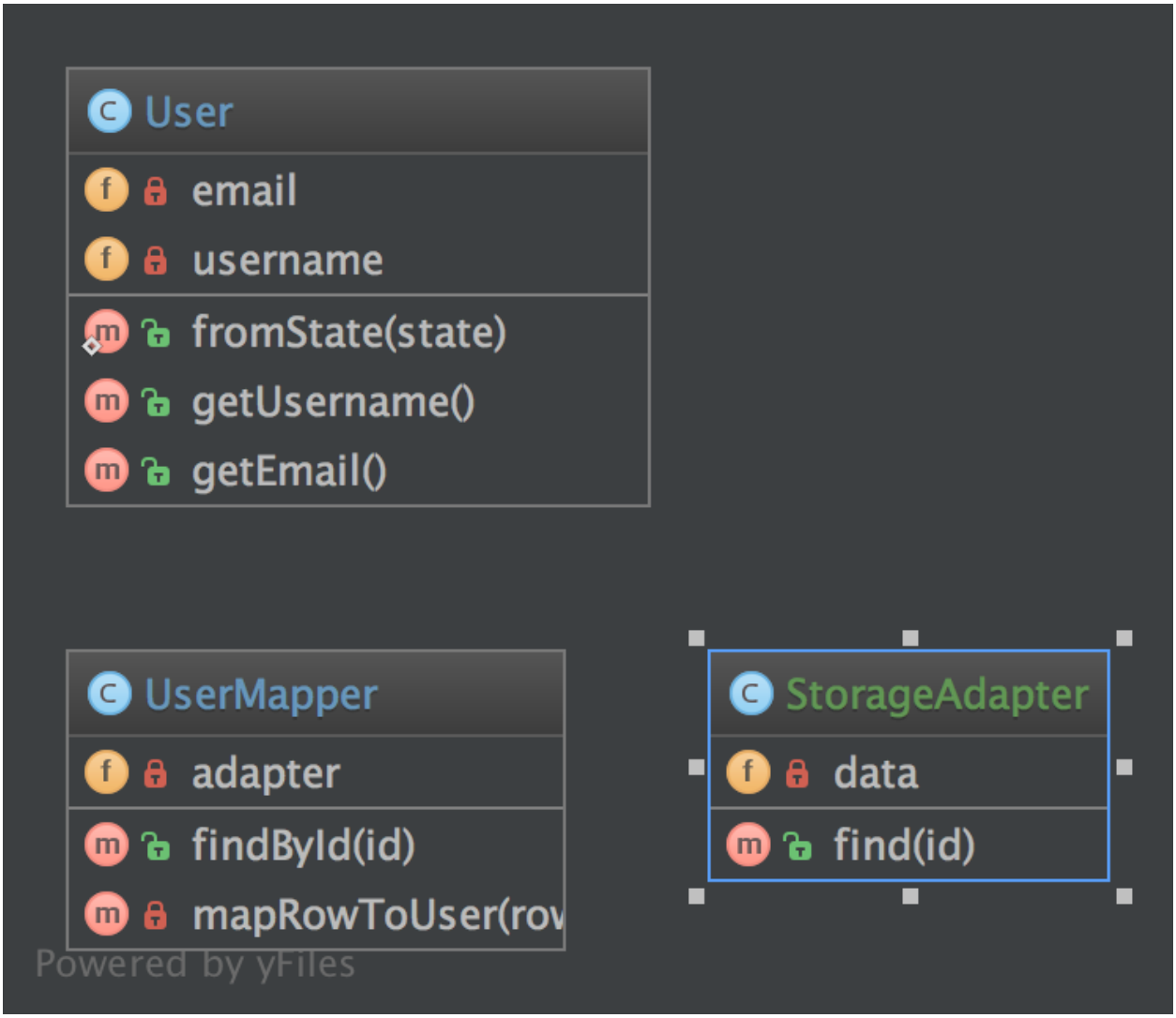
itself. The layer is composed of one or more mappers (or Data Access Objects), performing the data transfer. Mapper implementations vary in scope. Generic mappers will handle many different domain entity types, dedicated mappers will handle one or a few.

The key point of this pattern is, unlike Active Record pattern, the data model follows Single Responsibility Principle.

Examples

- DB Object Relational Mapper (ORM) : Doctrine2 uses DAO named as "EntityRepository"

UML Diagram



Code

You can also find this code on GitHub

User.php

```php
<?php

namespace DesignPatterns\Structural\DataMapper;

class User
{
    /**
     * @var string
     */
    private $username;

    /**
     * @var string
     */
    private $email;

    public static function fromState(array $state): User
    {
        // validate state before accessing keys!

        return new self(
            $state['username'],
            $state['email']
        );
    }

    public function __construct(string $username, string $email)
    {
        // validate parameters before setting them!

        $this->username = $username;
        $this->email = $email;
    }

    /**
     * @return string
     */
    public function getUsername()
    {
        return $this->username;
    }

    /**
     * @return string
     */
    public function getEmail()
    {
        return $this->email;
    }
}
```

UserMapper.php

```php
<?php

namespace DesignPatterns\Structural\DataMapper;

class UserMapper
```

```
{
    /**
     * @var StorageAdapter
     */
    private $adapter;

    /**
     * @param StorageAdapter $storage
     */
    public function __construct(StorageAdapter $storage)
    {
        $this->adapter = $storage;
    }

    /**
     * finds a user from storage based on ID and returns a User object located
     * in memory. Normally this kind of logic will be implemented using the Repository pattern.
     * However the important part is in mapRowToUser() below, that will create a business object from the
     * data fetched from storage
     *
     * @param int $id
     *
     * @return User
     */
    public function findById(int $id): User
    {
        $result = $this->adapter->find($id);

        if ($result === null) {
            throw new \InvalidArgumentException("User #$id not found");
        }

        return $this->mapRowToUser($result);
    }

    private function mapRowToUser(array $row): User
    {
        return User::fromState($row);
    }
}
```

StorageAdapter.php

```
<?php

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
{
    /**
     * @var array
     */
    private $data = [];

    public function __construct(array $data)
    {
```

```
        $this->data = $data;
    }

    /**
     * @param int $id
     *
     * @return array|null
     */
    public function find(int $id)
    {
        if (isset($this->data[$id])) {
            return $this->data[$id];
        }

        return null;
    }
}
```

Test

Tests/DataMapperTest.php

```
<?php

namespace DesignPatterns\Structural\DataMapper\Tests;

use DesignPatterns\Structural\DataMapper\StorageAdapter;
use DesignPatterns\Structural\DataMapper\User;
use DesignPatterns\Structural\DataMapper\UserMapper;
use PHPUnit\Framework\TestCase;

class DataMapperTest extends TestCase
{
    public function testCanMapUserFromStorage()
    {
        $storage = new StorageAdapter([1 => ['username' => 'domnikl', 'email' =>
↪'liebler.dominik@gmail.com']]);
        $mapper = new UserMapper($storage);

        $user = $mapper->findById(1);

        $this->assertInstanceOf(User::class, $user);
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testWillNotMapInvalidData()
    {
        $storage = new StorageAdapter([]);
        $mapper = new UserMapper($storage);

        $mapper->findById(1);
    }
}
```

Decorator

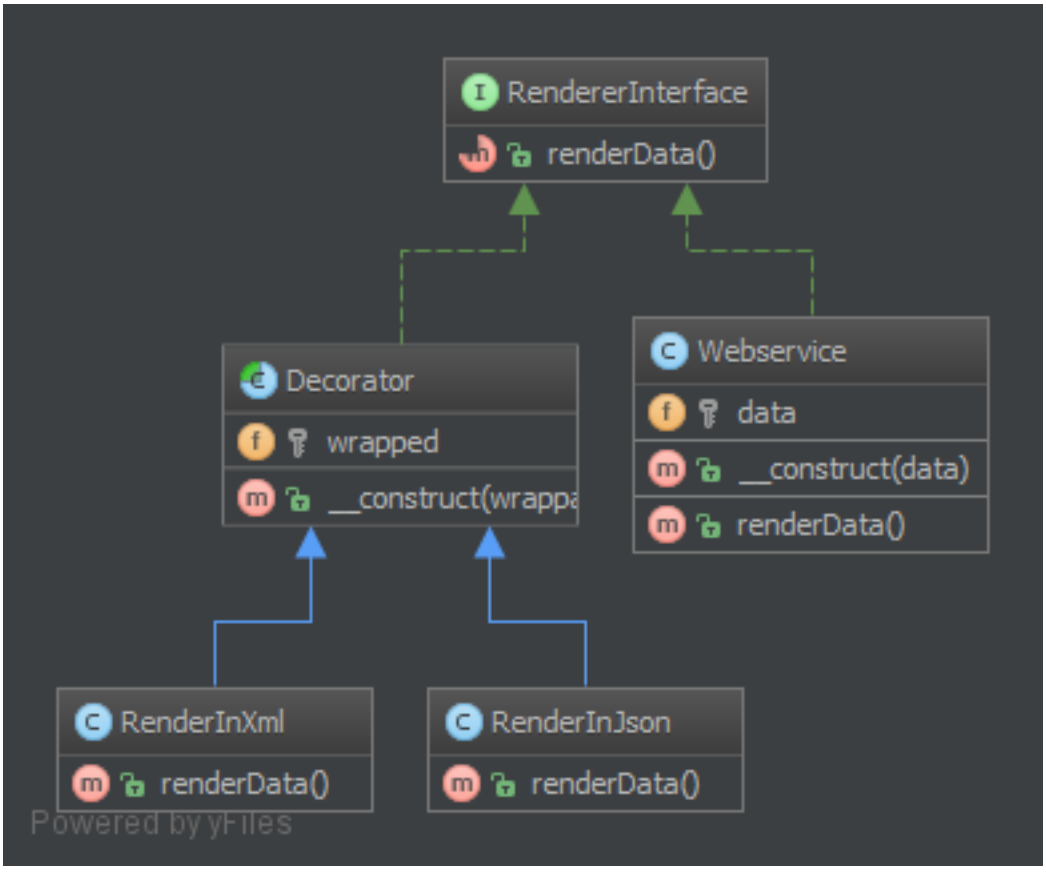
Purpose

To dynamically add new functionality to class instances.

Examples

- Zend Framework: decorators for `Zend_Form_Element` instances
- Web Service Layer: Decorators JSON and XML for a REST service (in this case, only one of these should be allowed of course)

UML Diagram



Code

You can also find this code on GitHub

RenderableInterface.php

```
<?php

namespace DesignPatterns\Structural\Decorator;

```

```php
interface RenderableInterface
{
    public function renderData(): string;
}
```

Webservice.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

class Webservice implements RenderableInterface
{
    /**
     * @var string
     */
    private $data;

    public function __construct(string $data)
    {
        $this->data = $data;
    }

    public function renderData(): string
    {
        return $this->data;
    }
}
```

RendererDecorator.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * the Decorator MUST implement the RenderableInterface contract, this is the key-
↪feature
 * of this design pattern. If not, this is no longer a Decorator but just a dumb
 * wrapper.
 */
abstract class RendererDecorator implements RenderableInterface
{
    /**
     * @var RenderableInterface
     */
    protected $wrapped;

    /**
     * @param RenderableInterface $renderer
     */
    public function __construct(RenderableInterface $renderer)
    {
        $this->wrapped = $renderer;
    }
}
```

XmlRenderer.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

class XmlRenderer extends RendererDecorator
{
    public function renderData(): string
    {
        $doc = new \DOMDocument();
        $data = $this->wrapped->renderData();
        $doc->appendChild($doc->createElement('content', $data));

        return $doc->saveXML();
    }
}
```

JsonRenderer.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

class JsonRenderer extends RendererDecorator
{
    public function renderData(): string
    {
        return json_encode($this->wrapped->renderData());
    }
}
```

Test

Tests/DecoratorTest.php

```php
<?php

namespace DesignPatterns\Structural\Decorator\Tests;

use DesignPatterns\Structural\Decorator;
use PHPUnit\Framework\TestCase;

class DecoratorTest extends TestCase
{
    /**
     * @var Decorator\Webservice
     */
    private $service;

    protected function setUp()
    {
        $this->service = new Decorator\Webservice('foobar');
    }

    public function testJsonDecorator()
    {
        $service = new Decorator\JsonRenderer($this->service);
```

```
23
24          $this->assertEquals('"foobar"', $service->renderData());
25      }
26
27      public function testXmlDecorator()
28      {
29          $service = new Decorator\XmlRenderer($this->service);
30
31          $this->assertXmlStringEqualsXmlString('<?xml version="1.0"?><content>foobar</
    ↪content>', $service->renderData());
32      }
33  }
```

Dependency Injection

Purpose

To implement a loosely coupled architecture in order to get better testable, maintainable and extendable code.

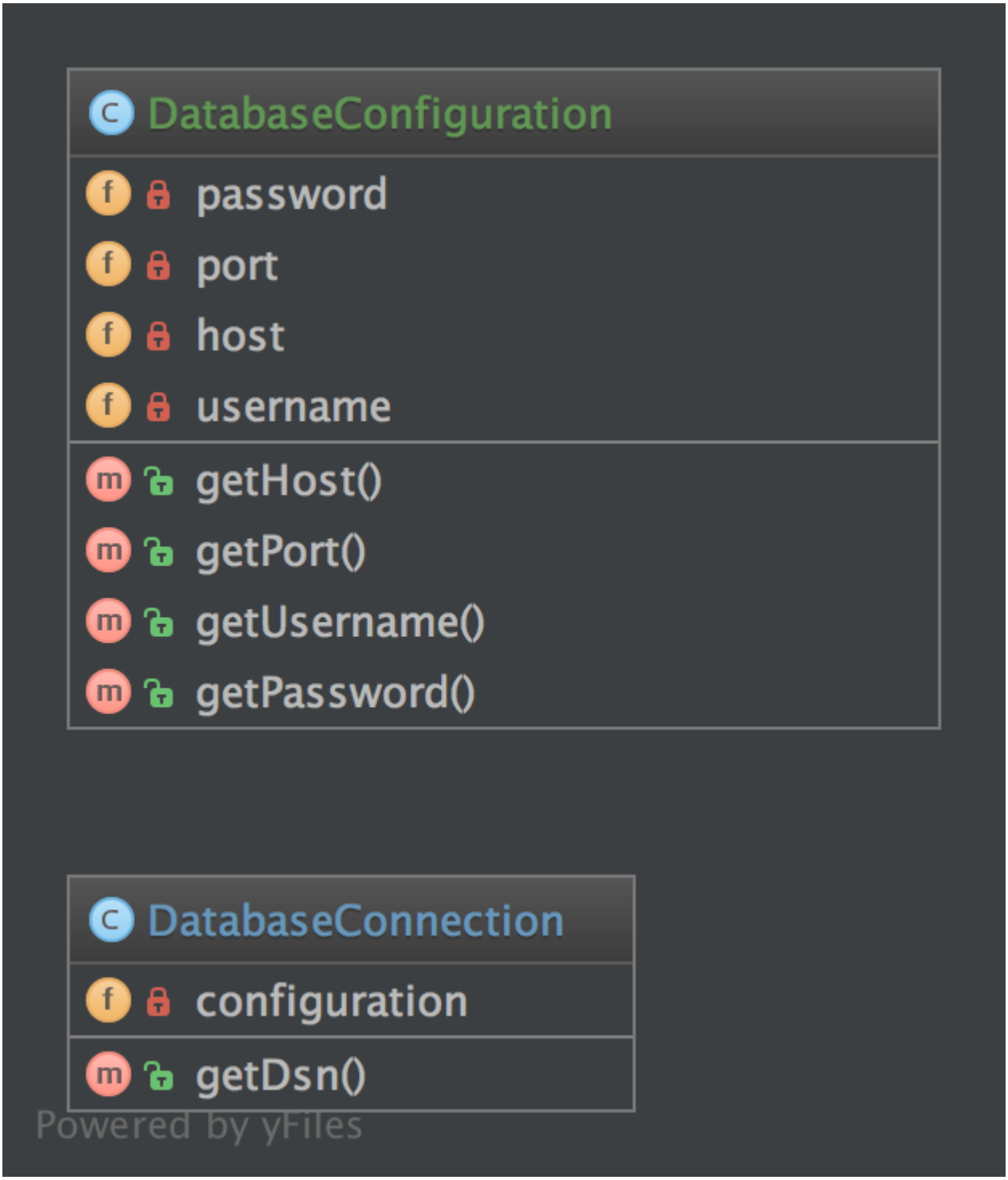
Usage

DatabaseConfiguration gets injected and `DatabaseConnection` will get all that it needs from `$config`. Without DI, the configuration would be created directly in `DatabaseConnection`, which is not very good for testing and extending it.

Examples

- The Doctrine2 ORM uses dependency injection e.g. for configuration that is injected into a `Connection` object. For testing purposes, one can easily create a mock object of the configuration and inject that into the `Connection` object
- Symfony and Zend Framework 2 already have containers for DI that create objects via a configuration array and inject them where needed (i.e. in Controllers)

UML Diagram



Code

You can also find this code on GitHub

DatabaseConfiguration.php

```php
<?php

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConfiguration
{
    /**
     * @var string
     */
    private $host;

    /**
     * @var int
     */
    private $port;

    /**
     * @var string
     */
    private $username;

    /**
     * @var string
     */
    private $password;

    public function __construct(string $host, int $port, string $username, string
↪$password)
    {
        $this->host = $host;
        $this->port = $port;
        $this->username = $username;
        $this->password = $password;
    }

    public function getHost(): string
    {
        return $this->host;
    }

    public function getPort(): int
    {
        return $this->port;
    }

    public function getUsername(): string
    {
        return $this->username;
    }

    public function getPassword(): string
    {
        return $this->password;
    }
}
```

DatabaseConnection.php

```php
<?php

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConnection
{
    /**
     * @var DatabaseConfiguration
     */
    private $configuration;

    /**
     * @param DatabaseConfiguration $config
     */
    public function __construct(DatabaseConfiguration $config)
    {
        $this->configuration = $config;
    }

    public function getDsn(): string
    {
        // this is just for the sake of demonstration, not a real DSN
        // notice that only the injected config is used here, so there is
        // a real separation of concerns here

        return sprintf(
            '%s:%s@%s:%d',
            $this->configuration->getUsername(),
            $this->configuration->getPassword(),
            $this->configuration->getHost(),
            $this->configuration->getPort()
        );
    }
}
```

Test

Tests/DependencyInjectionTest.php

```php
<?php

namespace DesignPatterns\Structural\DependencyInjection\Tests;

use DesignPatterns\Structural\DependencyInjection\DatabaseConfiguration;
use DesignPatterns\Structural\DependencyInjection\DatabaseConnection;
use PHPUnit\Framework\TestCase;

class DependencyInjectionTest extends TestCase
{
    public function testDependencyInjection()
    {
        $config = new DatabaseConfiguration('localhost', 3306, 'domnikl', '1234');
        $connection = new DatabaseConnection($config);

        $this->assertEquals('domnikl:1234@localhost:3306', $connection->getDsn());
    }
}
```

Facade

Purpose

The primary goal of a Facade Pattern is not to avoid you to read the manual of a complex API. It's only a side-effect. The first goal is to reduce coupling and follow the Law of Demeter.

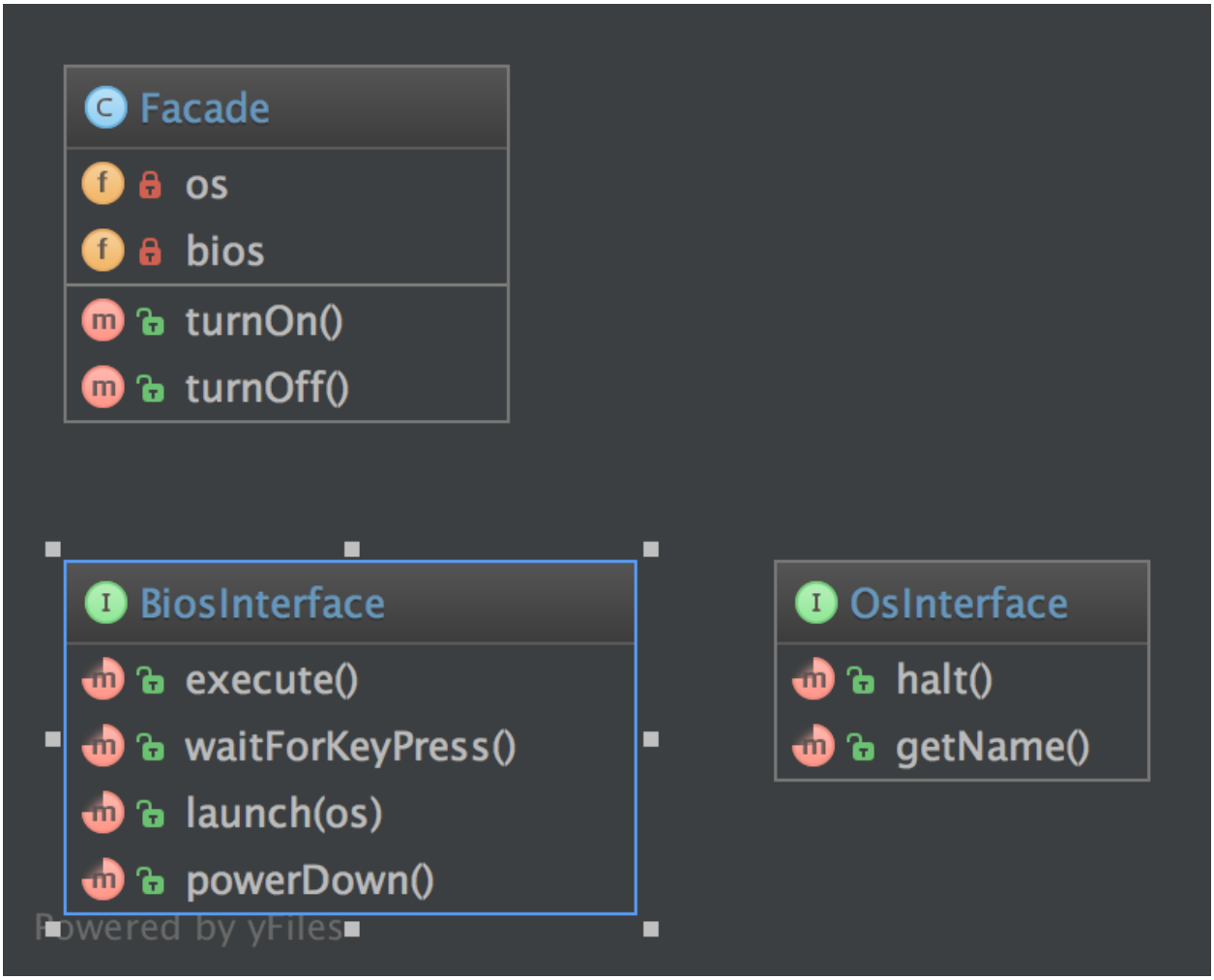
A Facade is meant to decouple a client and a sub-system by embedding many (but sometimes just one) interface, and of course to reduce complexity.

- A facade does not forbid you the access to the sub-system
- You can (you should) have multiple facades for one sub-system

That's why a good facade has no `new` in it. If there are multiple creations for each method, it is not a Facade, it's a Builder or a [Abstract|Static|Simple] Factory [Method].

The best facade has no `new` and a constructor with interface-type-hinted parameters. If you need creation of new instances, use a Factory as argument.

UML Diagram



Code

You can also find this code on GitHub

Facade.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

class Facade
{
    /**
     * @var OsInterface
     */
    private $os;

    /**
     * @var BiosInterface
     */
```

```
    private $bios;

    /**
     * @param BiosInterface $bios
     * @param OsInterface   $os
     */
    public function __construct(BiosInterface $bios, OsInterface $os)
    {
        $this->bios = $bios;
        $this->os = $os;
    }

    public function turnOn()
    {
        $this->bios->execute();
        $this->bios->waitForKeyPress();
        $this->bios->launch($this->os);
    }

    public function turnOff()
    {
        $this->os->halt();
        $this->bios->powerDown();
    }
}
```

OsInterface.php

```
<?php

namespace DesignPatterns\Structural\Facade;

interface OsInterface
{
    public function halt();

    public function getName(): string;
}
```

BiosInterface.php

```
<?php

namespace DesignPatterns\Structural\Facade;

interface BiosInterface
{
    public function execute();

    public function waitForKeyPress();

    public function launch(OsInterface $os);

    public function powerDown();
}
```

Test

Tests/FacadeTest.php

```php
<?php

namespace DesignPatterns\Structural\Facade\Tests;

use DesignPatterns\Structural\Facade\Facade;
use DesignPatterns\Structural\Facade\OsInterface;
use PHPUnit\Framework\TestCase;

class FacadeTest extends TestCase
{
    public function testComputerOn()
    {
        /** @var OsInterface|\PHPUnit_Framework_MockObject_MockObject $os */
        $os = $this->createMock('DesignPatterns\Structural\Facade\OsInterface');

        $os->method('getName')
            ->will($this->returnValue('Linux'));

        $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\BiosInterface
↪')
            ->setMethods(['launch', 'execute', 'waitForKeyPress'])
            ->disableAutoload()
            ->getMock();

        $bios->expects($this->once())
            ->method('launch')
            ->with($os);

        $facade = new Facade($bios, $os);

        // the facade interface is simple
        $facade->turnOn();

        // but you can also access the underlying components
        $this->assertEquals('Linux', $os->getName());
    }
}
```

Fluent Interface

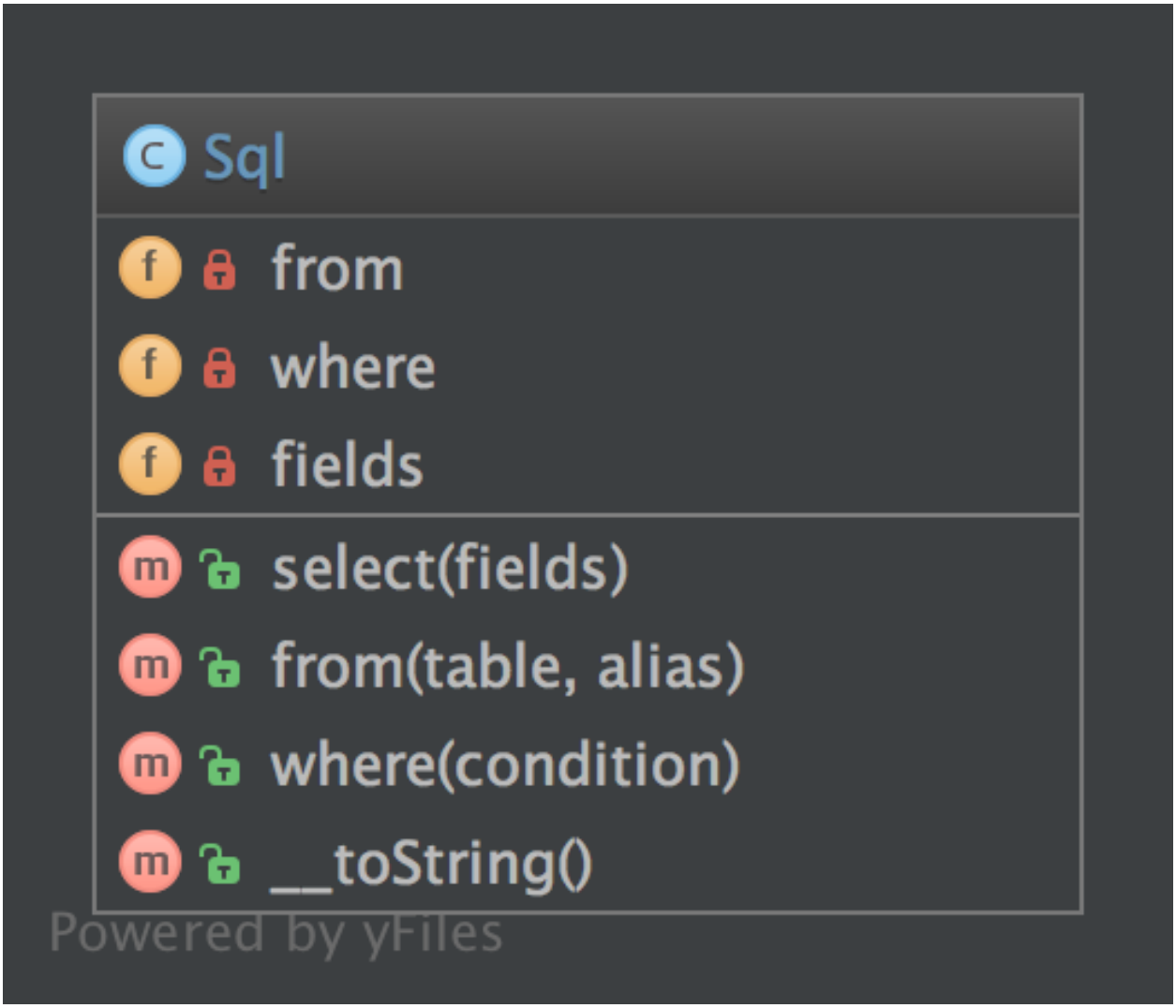
Purpose

To write code that is easy readable just like sentences in a natural language (like English).

Examples

- Doctrine2's QueryBuilder works something like that example class below
- PHPUnit uses fluent interfaces to build mock objects
- Yii Framework: CDbCommand and CActiveRecord use this pattern, too

UML Diagram



Code

You can also find this code on GitHub

Sql.php

```php
<?php

namespace DesignPatterns\Structural\FluentInterface;

class Sql
{
    /**
     * @var array
     */
    private $fields = [];

    /**
```

```
     * @var array
     */
    private $from = [];

    /**
     * @var array
     */
    private $where = [];

    public function select(array $fields): Sql
    {
        $this->fields = $fields;

        return $this;
    }

    public function from(string $table, string $alias): Sql
    {
        $this->from[] = $table.' AS '.$alias;

        return $this;
    }

    public function where(string $condition): Sql
    {
        $this->where[] = $condition;

        return $this;
    }

    public function __toString(): string
    {
        return sprintf(
            'SELECT %s FROM %s WHERE %s',
            join(', ', $this->fields),
            join(', ', $this->from),
            join(' AND ', $this->where)
        );
    }
}
```

Test

Tests/FluentInterfaceTest.php

```
<?php

namespace DesignPatterns\Structural\FluentInterface\Tests;

use DesignPatterns\Structural\FluentInterface\Sql;
use PHPUnit\Framework\TestCase;

class FluentInterfaceTest extends TestCase
{
    public function testBuildSQL()
    {
        $query = (new Sql())
```

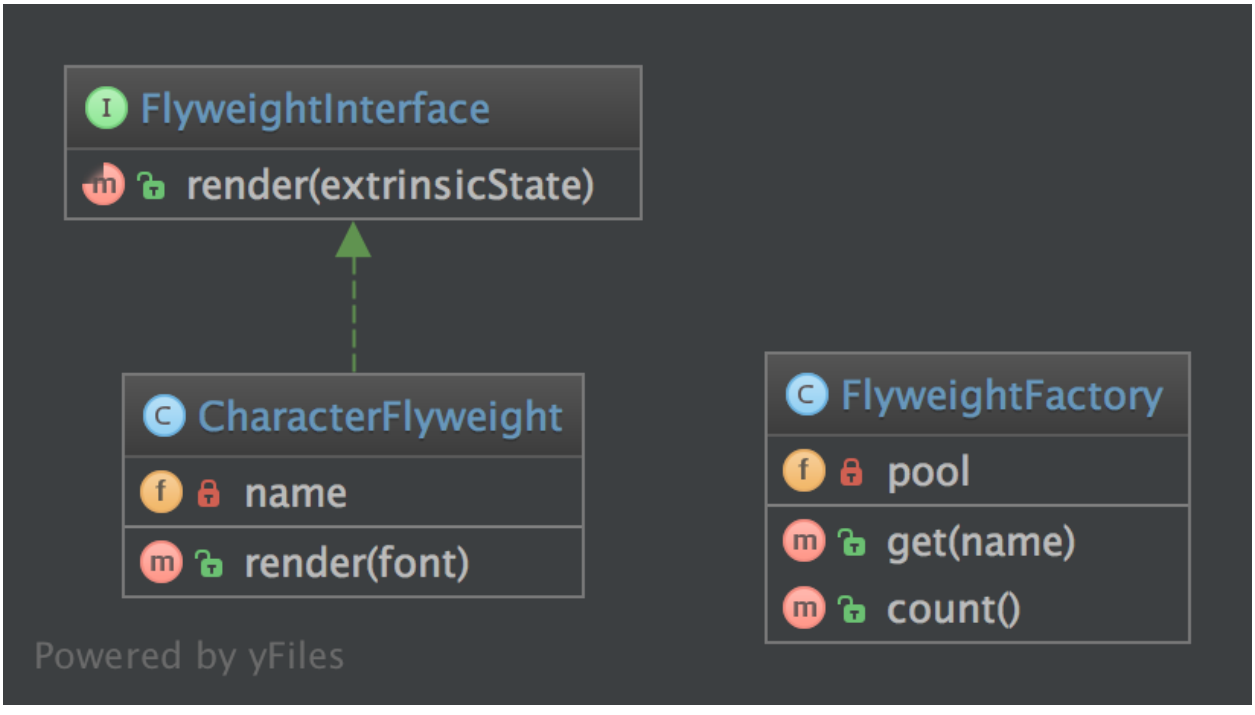
```
                ->select(['foo', 'bar'])
                ->from('foobar', 'f')
                ->where('f.bar = ?');

        $this->assertEquals('SELECT foo, bar FROM foobar AS f WHERE f.bar = ?', (string) $query);
    }
}
```

Flyweight

Purpose

To minimise memory usage, a Flyweight shares as much as possible memory with similar objects. It is needed when a large amount of objects is used that don't differ much in state. A common practice is to hold state in external data structures and pass them to the flyweight object when needed.

UML Diagram



Code

You can also find this code on GitHub

FlyweightInterface.php

```
<?php

namespace DesignPatterns\Structural\Flyweight;

```

```php
interface FlyweightInterface
{
    public function render(string $extrinsicState): string;
}
```

CharacterFlyweight.php

```php
<?php

namespace DesignPatterns\Structural\Flyweight;

/**
 * Implements the flyweight interface and adds storage for intrinsic state, if any.
 * Instances of concrete flyweights are shared by means of a factory.
 */
class CharacterFlyweight implements FlyweightInterface
{
    /**
     * Any state stored by the concrete flyweight must be independent of its context.
     * For flyweights representing characters, this is usually the corresponding character code.
     *
     * @var string
     */
    private $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function render(string $font): string
    {
        // Clients supply the context-dependent information that the flyweight needs to draw itself
        // For flyweights representing characters, extrinsic state usually contains e.g. the font.

        return sprintf('Character %s with font %s', $this->name, $font);
    }
}
```

FlyweightFactory.php

```php
<?php

namespace DesignPatterns\Structural\Flyweight;

/**
 * A factory manages shared flyweights. Clients should not instantiate them directly,
 * but let the factory take care of returning existing objects or creating new ones.
 */
class FlyweightFactory implements \Countable
{
    /**
     * @var CharacterFlyweight[]
     */
    private $pool = [];
```

```php

    public function get(string $name): CharacterFlyweight
    {
        if (!isset($this->pool[$name])) {
            $this->pool[$name] = new CharacterFlyweight($name);
        }

        return $this->pool[$name];
    }

    public function count(): int
    {
        return count($this->pool);
    }
}
```

Test

Tests/FlyweightTest.php

```php
<?php

namespace DesignPatterns\Structural\Flyweight\Tests;

use DesignPatterns\Structural\Flyweight\FlyweightFactory;
use PHPUnit\Framework\TestCase;

class FlyweightTest extends TestCase
{
    private $characters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
        'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'];
    private $fonts = ['Arial', 'Times New Roman', 'Verdana', 'Helvetica'];

    public function testFlyweight()
    {
        $factory = new FlyweightFactory();

        foreach ($this->characters as $char) {
            foreach ($this->fonts as $font) {
                $flyweight = $factory->get($char);
                $rendered = $flyweight->render($font);

                $this->assertEquals(sprintf('Character %s with font %s', $char,
↪$font), $rendered);
            }
        }

        // Flyweight pattern ensures that instances are shared
        // instead of having hundreds of thousands of individual objects
        // there must be one instance for every char that has been reused for␣
↪displaying in different fonts
        $this->assertCount(count($this->characters), $factory);
    }
}
```

Proxy

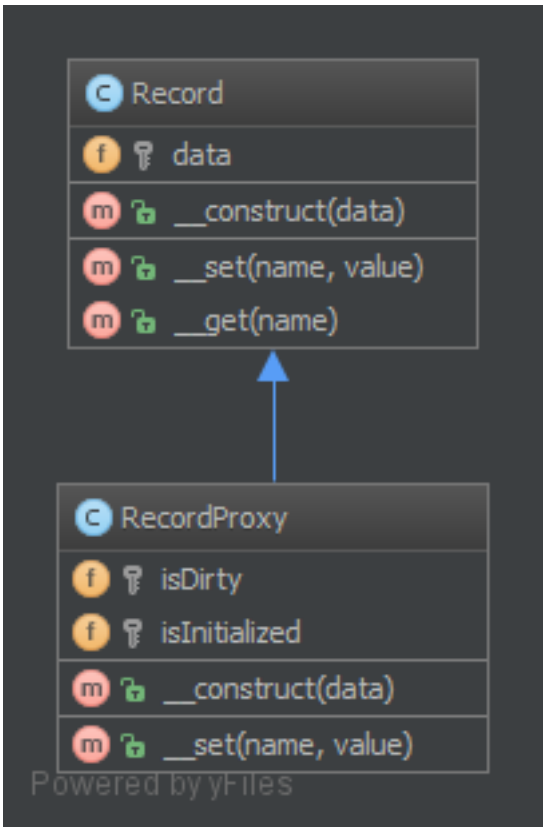
Purpose

To interface to anything that is expensive or impossible to duplicate.

Examples

- Doctrine2 uses proxies to implement framework magic (e.g. lazy initialization) in them, while the user still works with his own entity classes and will never use nor touch the proxies

UML Diagram



Code

You can also find this code on GitHub

Record.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

/**
 * @property string username
 */
```

```php
class Record
{
    /**
     * @var string[]
     */
    private $data;

    /**
     * @param string[] $data
     */
    public function __construct(array $data = [])
    {
        $this->data = $data;
    }

    /**
     * @param string $name
     * @param string  $value
     */
    public function __set(string $name, string $value)
    {
        $this->data[$name] = $value;
    }

    public function __get(string $name): string
    {
        if (!isset($this->data[$name])) {
            throw new \OutOfRangeException('Invalid name given');
        }

        return $this->data[$name];
    }
}
```

RecordProxy.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

class RecordProxy extends Record
{
    /**
     * @var bool
     */
    private $isDirty = false;

    /**
     * @var bool
     */
    private $isInitialized = false;

    /**
     * @param array $data
     */
    public function __construct(array $data)
    {
        parent::__construct($data);
```

```php

        // when the record has data, mark it as initialized
        // since Record will hold our business logic, we don't want to
        // implement this behaviour there, but instead in a new proxy class
        // that extends the Record class
        if (count($data) > 0) {
            $this->isInitialized = true;
            $this->isDirty = true;
        }
    }

    /**
     * @param string $name
     * @param string  $value
     */
    public function __set(string $name, string $value)
    {
        $this->isDirty = true;

        parent::__set($name, $value);
    }

    public function isDirty(): bool
    {
        return $this->isDirty;
    }
}
```

Test

Registry

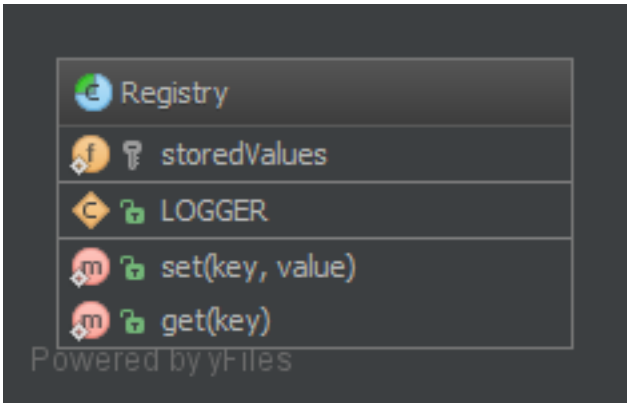
Purpose

To implement a central storage for objects often used throughout the application, is typically implemented using an abstract class with only static methods (or using the Singleton pattern). Remember that this introduces global state, which should be avoided at all times! Instead implement it using Dependency Injection!

Examples

- Zend Framework 1: `Zend_Registry` holds the application's logger object, front controller etc.
- Yii Framework: `CWebApplication` holds all the application components, such as `CWebUser`, `CUrlManager`, etc.

UML Diagram



Code

You can also find this code on GitHub

Registry.php

```php
<?php

namespace DesignPatterns\Structural\Registry;

abstract class Registry
{
    const LOGGER = 'logger';

    /**
     * this introduces global state in your application which can not be mocked up for testing
     * and is therefor considered an anti-pattern! Use dependency injection instead!
     *
     * @var array
     */
    private static $storedValues = [];

    /**
     * @var array
     */
    private static $allowedKeys = [
        self::LOGGER,
    ];

    /**
     * @param string $key
     * @param mixed  $value
     *
     * @return void
     */
    public static function set(string $key, $value)
    {
        if (!in_array($key, self::$allowedKeys)) {
            throw new \InvalidArgumentException('Invalid key given');
        }
```

```
35
36          self::$storedValues[$key] = $value;
37      }
38
39      /**
40       * @param string $key
41       *
42       * @return mixed
43       */
44      public static function get(string $key)
45      {
46          if (!in_array($key, self::$allowedKeys) || !isset(self::$storedValues[$key]))
    ↪{
47              throw new \InvalidArgumentException('Invalid key given');
48          }
49
50          return self::$storedValues[$key];
51      }
52  }
```

Test

Tests/RegistryTest.php

```
1   <?php
2
3   namespace DesignPatterns\Structural\Registry\Tests;
4
5   use DesignPatterns\Structural\Registry\Registry;
6   use stdClass;
7   use PHPUnit\Framework\TestCase;
8
9   class RegistryTest extends TestCase
10  {
11      public function testSetAndGetLogger()
12      {
13          $key = Registry::LOGGER;
14          $logger = new stdClass();
15
16          Registry::set($key, $logger);
17          $storedLogger = Registry::get($key);
18
19          $this->assertSame($logger, $storedLogger);
20          $this->assertInstanceOf(stdClass::class, $storedLogger);
21      }
22
23      /**
24       * @expectedException \InvalidArgumentException
25       */
26      public function testThrowsExceptionWhenTryingToSetInvalidKey()
27      {
28          Registry::set('foobar', new stdClass());
29      }
30
31      /**
32       * notice @runInSeparateProcess here: without it, a previous test might have set␣
    ↪it already and
```

```
33      * testing would not be possible. That's why you should implement Dependency␣
↪Injection where an
34      * injected class may easily be replaced by a mockup
35      *
36      * @runInSeparateProcess
37      * @expectedException \InvalidArgumentException
38      */
39     public function testThrowsExceptionWhenTryingToGetNotSetKey()
40     {
41         Registry::get(Registry::LOGGER);
42     }
43 }
```

Behavioral

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Chain Of Responsibilities

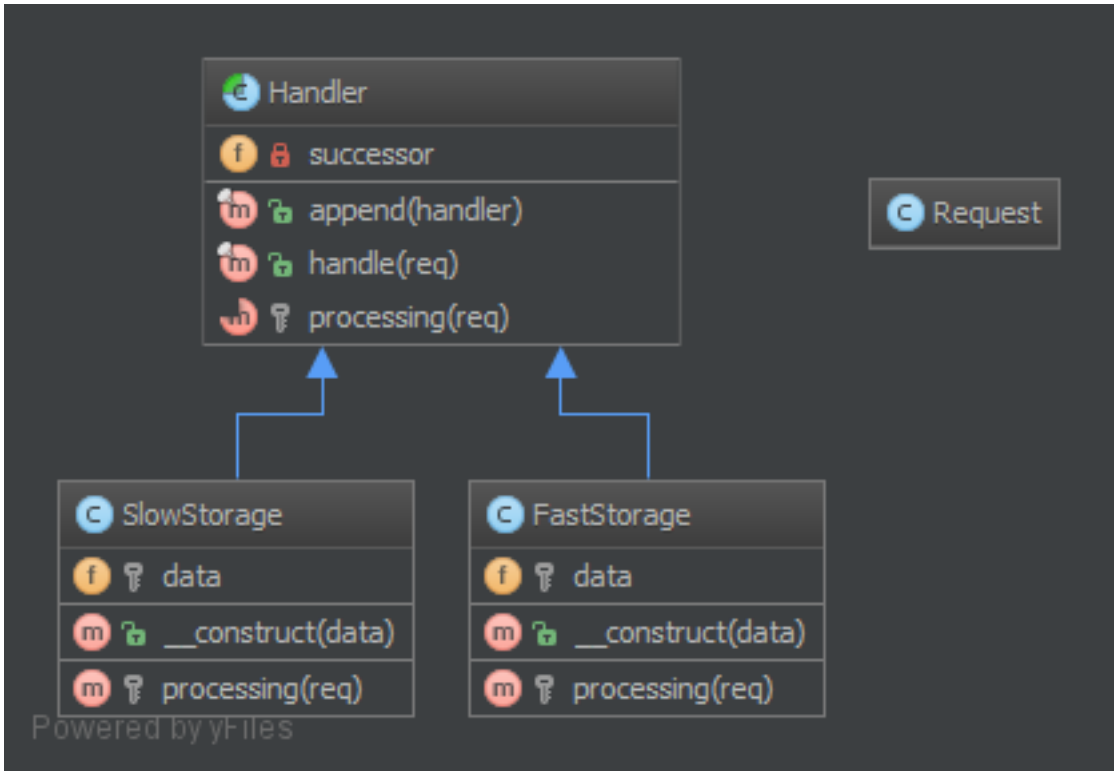
Purpose

To build a chain of objects to handle a call in sequential order. If one object cannot handle a call, it delegates the call to the next in the chain and so forth.

Examples

- logging framework, where each chain element decides autonomously what to do with a log message
- a Spam filter
- Caching: first object is an instance of e.g. a Memcached Interface, if that "misses" it delegates the call to the database interface
- Yii Framework: CFilterChain is a chain of controller action filters. the executing point is passed from one filter to the next along the chain, and only if all filters say "yes", the action can be invoked at last.

UML Diagram



Code

You can also find this code on GitHub

Handler.php

```php
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

abstract class Handler
{
    /**
     * @var Handler|null
     */
    private $successor = null;

    public function __construct(Handler $handler = null)
    {
        $this->successor = $handler;
    }

    /**
     * This approach by using a template method pattern ensures you that
     * each subclass will not forget to call the successor
```

```
     *
     * @param RequestInterface $request
     *
     * @return string|null
     */
    final public function handle(RequestInterface $request)
    {
        $processed = $this->processing($request);

        if ($processed === null) {
            // the request has not been processed by this handler => see the next
            if ($this->successor !== null) {
                $processed = $this->successor->handle($request);
            }
        }

        return $processed;
    }

    abstract protected function processing(RequestInterface $request);
}
```

Responsible/FastStorage.php

```
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
use Psr\Http\Message\RequestInterface;

class HttpInMemoryCacheHandler extends Handler
{
    /**
     * @var array
     */
    private $data;

    /**
     * @param array $data
     * @param Handler|null $successor
     */
    public function __construct(array $data, Handler $successor = null)
    {
        parent::__construct($successor);

        $this->data = $data;
    }

    /**
     * @param RequestInterface $request
     *
     * @return string|null
     */
    protected function processing(RequestInterface $request)
    {
        $key = sprintf(
            '%s?%s',
```

```
            $request->getUri()->getPath(),
            $request->getUri()->getQuery()
        );

        if ($request->getMethod() == 'GET' && isset($this->data[$key])) {
            return $this->data[$key];
        }

        return null;
    }
}
```

Responsible/SlowStorage.php

```
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
use Psr\Http\Message\RequestInterface;

class SlowDatabaseHandler extends Handler
{
    /**
     * @param RequestInterface $request
     *
     * @return string|null
     */
    protected function processing(RequestInterface $request)
    {
        // this is a mockup, in production code you would ask a slow (compared to in-memory) DB for the results

        return 'Hello World!';
    }
}
```

Test

Tests/ChainTest.php

```
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\HttpInMemoryCacheHandler;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowDatabaseHandler;
use PHPUnit\Framework\TestCase;

class ChainTest extends TestCase
{
    /**
     * @var Handler
```

```
     */
    private $chain;

    protected function setUp()
    {
        $this->chain = new HttpInMemoryCacheHandler(
            ['/foo/bar?index=1' => 'Hello In Memory!'],
            new SlowDatabaseHandler()
        );
    }

    public function testCanRequestKeyInFastStorage()
    {
        $uri = $this->createMock('Psr\Http\Message\UriInterface');
        $uri->method('getPath')->willReturn('/foo/bar');
        $uri->method('getQuery')->willReturn('index=1');

        $request = $this->createMock('Psr\Http\Message\RequestInterface');
        $request->method('getMethod')
            ->willReturn('GET');
        $request->method('getUri')->willReturn($uri);

        $this->assertEquals('Hello In Memory!', $this->chain->handle($request));
    }

    public function testCanRequestKeyInSlowStorage()
    {
        $uri = $this->createMock('Psr\Http\Message\UriInterface');
        $uri->method('getPath')->willReturn('/foo/baz');
        $uri->method('getQuery')->willReturn('');

        $request = $this->createMock('Psr\Http\Message\RequestInterface');
        $request->method('getMethod')
            ->willReturn('GET');
        $request->method('getUri')->willReturn($uri);

        $this->assertEquals('Hello World!', $this->chain->handle($request));
    }
}
```

Command

Purpose

To encapsulate invocation and decoupling.

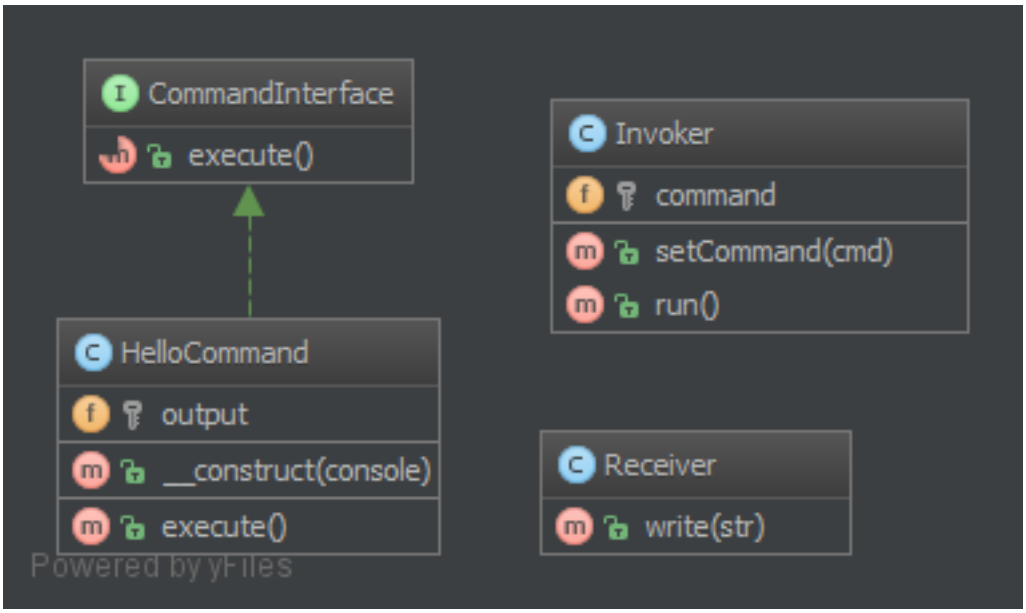
We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

The second aspect of this pattern is the undo(), which undoes the method execute(). Command can also be aggregated to combine more complex commands with minimum copy-paste and relying on composition over inheritance.

Examples

- A text editor : all events are Command which can be undone, stacked and saved.
- Symfony2: SF2 Commands that can be run from the CLI are built with just the Command pattern in mind
- big CLI tools use subcommands to distribute various tasks and pack them in "modules", each of these can be implemented with the Command pattern (e.g. vagrant)

UML Diagram



Code

You can also find this code on GitHub

CommandInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

interface CommandInterface
{
    /**
     * this is the most important method in the Command pattern,
     * The Receiver goes in the constructor.
     */
    public function execute();
}
```

HelloCommand.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;
```

```php

/**
 * This concrete command calls "print" on the Receiver, but an external
 * invoker just knows that it can call "execute"
 */
class HelloCommand implements CommandInterface
{
    /**
     * @var Receiver
     */
    private $output;

    /**
     * Each concrete command is built with different receivers.
     * There can be one, many or completely no receivers, but there can be other commands in the parameters
     *
     * @param Receiver $console
     */
    public function __construct(Receiver $console)
    {
        $this->output = $console;
    }

    /**
     * execute and output "Hello World".
     */
    public function execute()
    {
        // sometimes, there is no receiver and this is the command which does all the work
        $this->output->write('Hello World');
    }
}
```

Receiver.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * Receiver is specific service with its own contract and can be only concrete.
 */
class Receiver
{
    /**
     * @var bool
     */
    private $enableDate = false;

    /**
     * @var string[]
     */
    private $output = [];

    /**
     * @param string $str
```

```
     */
    public function write(string $str)
    {
        if ($this->enableDate) {
            $str .= ' ['.date('Y-m-d').']';
        }

        $this->output[] = $str;
    }

    public function getOutput(): string
    {
        return join("\n", $this->output);
    }

    /**
     * Enable receiver to display message date
     */
    public function enableDate()
    {
        $this->enableDate = true;
    }

    /**
     * Disable receiver to display message date
     */
    public function disableDate()
    {
        $this->enableDate = false;
    }
}
```

Invoker.php

```
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * Invoker is using the command given to it.
 * Example : an Application in SF2.
 */
class Invoker
{
    /**
     * @var CommandInterface
     */
    private $command;

    /**
     * in the invoker we find this kind of method for subscribing the command
     * There can be also a stack, a list, a fixed set ...
     *
     * @param CommandInterface $cmd
     */
    public function setCommand(CommandInterface $cmd)
    {
        $this->command = $cmd;
```

```
    }

    /**
     * executes the command; the invoker is the same whatever is the command
     */
    public function run()
    {
        $this->command->execute();
    }
}
```

Test

Tests/CommandTest.php

```
<?php

namespace DesignPatterns\Behavioral\Command\Tests;

use DesignPatterns\Behavioral\Command\HelloCommand;
use DesignPatterns\Behavioral\Command\Invoker;
use DesignPatterns\Behavioral\Command\Receiver;
use PHPUnit\Framework\TestCase;

class CommandTest extends TestCase
{
    public function testInvocation()
    {
        $invoker = new Invoker();
        $receiver = new Receiver();

        $invoker->setCommand(new HelloCommand($receiver));
        $invoker->run();
        $this->assertEquals('Hello World', $receiver->getOutput());
    }
}
```

Iterator

Purpose

To make an object iterable and to make it appear like a collection of objects.

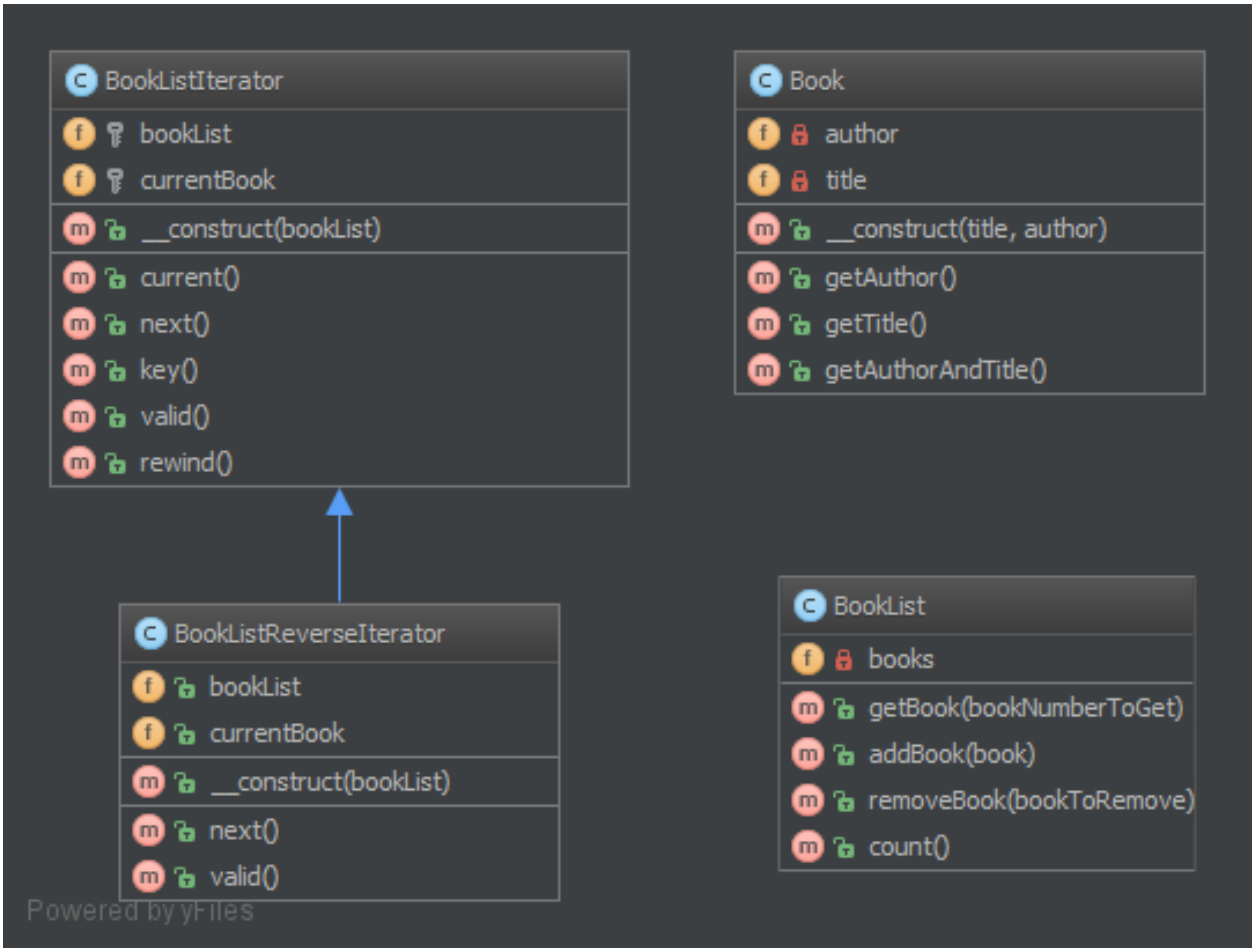
Examples

- to process a file line by line by just running over all lines (which have an object representation) for a file (which of course is an object, too)

Note

Standard PHP Library (SPL) defines an interface Iterator which is best suited for this! Often you would want to implement the Countable interface too, to allow `count($object)` on your iterable object

UML Diagram



Code

You can also find this code on GitHub

Book.php

```php
<?php

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    /**
     * @var string
     */
    private $author;

    /**
     * @var string
     */
    private $title;

```

```
    public function __construct(string $title, string $author)
    {
        $this->author = $author;
        $this->title = $title;
    }

    public function getAuthor(): string
    {
        return $this->author;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function getAuthorAndTitle(): string
    {
        return $this->getTitle().' by '.$this->getAuthor();
    }
}
```

BookList.php

```
<?php

namespace DesignPatterns\Behavioral\Iterator;

class BookList implements \Countable, \Iterator
{
    /**
     * @var Book[]
     */
    private $books = [];

    /**
     * @var int
     */
    private $currentIndex = 0;

    public function addBook(Book $book)
    {
        $this->books[] = $book;
    }

    public function removeBook(Book $bookToRemove)
    {
        foreach ($this->books as $key => $book) {
            if ($book->getAuthorAndTitle() === $bookToRemove->getAuthorAndTitle()) {
                unset($this->books[$key]);
            }
        }

        $this->books = array_values($this->books);
    }

    public function count(): int
    {
```

```
        return count($this->books);
    }

    public function current(): Book
    {
        return $this->books[$this->currentIndex];
    }

    public function key(): int
    {
        return $this->currentIndex;
    }

    public function next()
    {
        $this->currentIndex++;
    }

    public function rewind()
    {
        $this->currentIndex = 0;
    }

    public function valid(): bool
    {
        return isset($this->books[$this->currentIndex]);
    }
}
```

Test

Tests/IteratorTest.php

```
<?php

namespace DesignPatterns\Behavioral\Iterator\Tests;

use DesignPatterns\Behavioral\Iterator\Book;
use DesignPatterns\Behavioral\Iterator\BookList;
use DesignPatterns\Behavioral\Iterator\BookListIterator;
use DesignPatterns\Behavioral\Iterator\BookListReverseIterator;
use PHPUnit\Framework\TestCase;

class IteratorTest extends TestCase
{
    public function testCanIterateOverBookList()
    {
        $bookList = new BookList();
        $bookList->addBook(new Book('Learning PHP Design Patterns', 'William Sanders
↪'));
        $bookList->addBook(new Book('Professional Php Design Patterns', 'Aaron Saray
↪'));
        $bookList->addBook(new Book('Clean Code', 'Robert C. Martin'));

        $books = [];

        foreach ($bookList as $book) {
```

```
            $books[] = $book->getAuthorAndTitle();
        }

        $this->assertEquals(
            [
                'Learning PHP Design Patterns by William Sanders',
                'Professional Php Design Patterns by Aaron Saray',
                'Clean Code by Robert C. Martin',
            ],
            $books
        );
    }

    public function testCanIterateOverBookListAfterRemovingBook()
    {
        $book = new Book('Clean Code', 'Robert C. Martin');
        $book2 = new Book('Professional Php Design Patterns', 'Aaron Saray');

        $bookList = new BookList();
        $bookList->addBook($book);
        $bookList->addBook($book2);
        $bookList->removeBook($book);

        $books = [];
        foreach ($bookList as $book) {
            $books[] = $book->getAuthorAndTitle();
        }

        $this->assertEquals(
            ['Professional Php Design Patterns by Aaron Saray'],
            $books
        );
    }

    public function testCanAddBookToList()
    {
        $book = new Book('Clean Code', 'Robert C. Martin');

        $bookList = new BookList();
        $bookList->addBook($book);

        $this->assertCount(1, $bookList);
    }

    public function testCanRemoveBookFromList()
    {
        $book = new Book('Clean Code', 'Robert C. Martin');

        $bookList = new BookList();
        $bookList->addBook($book);
        $bookList->removeBook($book);

        $this->assertCount(0, $bookList);
    }
}
```

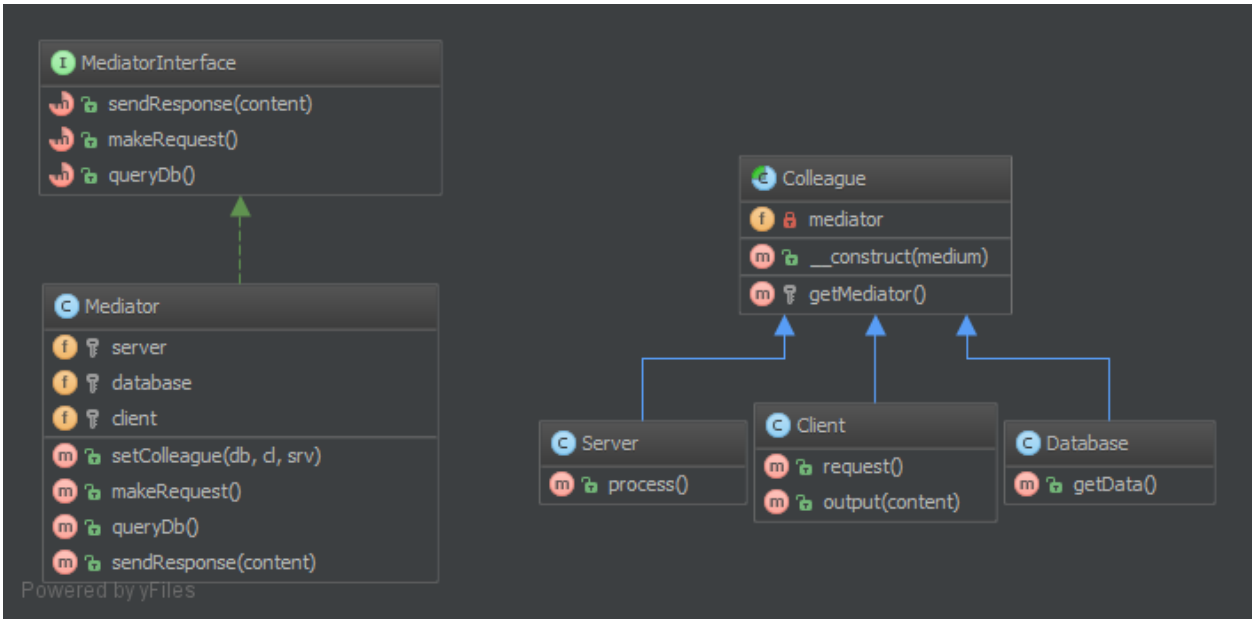
Mediator

Purpose

This pattern provides an easy way to decouple many components working together. It is a good alternative to Observer IF you have a "central intelligence", like a controller (but not in the sense of the MVC).

All components (called Colleague) are only coupled to the MediatorInterface and it is a good thing because in OOP, one good friend is better than many. This is the key-feature of this pattern.

UML Diagram



Code

You can also find this code on GitHub

MediatorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Mediator;

/**
 * MediatorInterface is a contract for the Mediator
 * This interface is not mandatory but it is better for Liskov substitution principle concerns.
 */
interface MediatorInterface
{
    /**
     * sends the response.
     *
     * @param string $content
     */
```

```
    public function sendResponse($content);

    /**
     * makes a request
     */
    public function makeRequest();

    /**
     * queries the DB
     */
    public function queryDb();
}
```

Mediator.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator;

/**
 * Mediator is the concrete Mediator for this design pattern
 *
 * In this example, I have made a "Hello World" with the Mediator Pattern
 */
class Mediator implements MediatorInterface
{
    /**
     * @var Subsystem\Server
     */
    private $server;

    /**
     * @var Subsystem\Database
     */
    private $database;

    /**
     * @var Subsystem\Client
     */
    private $client;

    /**
     * @param Subsystem\Database $database
     * @param Subsystem\Client $client
     * @param Subsystem\Server $server
     */
    public function __construct(Subsystem\Database $database, Subsystem\Client
↪$client, Subsystem\Server $server)
    {
        $this->database = $database;
        $this->server = $server;
        $this->client = $client;

        $this->database->setMediator($this);
        $this->server->setMediator($this);
        $this->client->setMediator($this);
    }

```

```
    public function makeRequest()
    {
        $this->server->process();
    }

    public function queryDb(): string
    {
        return $this->database->getData();
    }

    /**
     * @param string $content
     */
    public function sendResponse($content)
    {
        $this->client->output($content);
    }
}
```

Colleague.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator;

/**
 * Colleague is an abstract colleague who works together but he only knows
 * the Mediator, not other colleagues
 */
abstract class Colleague
{
    /**
     * this ensures no change in subclasses.
     *
     * @var MediatorInterface
     */
    protected $mediator;

    /**
     * @param MediatorInterface $mediator
     */
    public function setMediator(MediatorInterface $mediator)
    {
        $this->mediator = $mediator;
    }
}
```

Subsystem/Client.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

/**
 * Client is a client that makes requests and gets the response.
 */
```

```php
class Client extends Colleague
{
    public function request()
    {
        $this->mediator->makeRequest();
    }

    public function output(string $content)
    {
        echo $content;
    }
}
```

Subsystem/Database.php

```php
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

class Database extends Colleague
{
    public function getData(): string
    {
        return 'World';
    }
}
```

Subsystem/Server.php

```php
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

class Server extends Colleague
{
    public function process()
    {
        $data = $this->mediator->queryDb();
        $this->mediator->sendResponse(sprintf("Hello %s", $data));
    }
}
```

Test

Tests/MediatorTest.php

```php
<?php

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Mediator;
use DesignPatterns\Behavioral\Mediator\Subsystem\Client;
```

```php
use DesignPatterns\Behavioral\Mediator\Subsystem\Database;
use DesignPatterns\Behavioral\Mediator\Subsystem\Server;
use PHPUnit\Framework\TestCase;

class MediatorTest extends TestCase
{
    public function testOutputHelloWorld()
    {
        $client = new Client();
        new Mediator(new Database(), $client, new Server());

        $this->expectOutputString('Hello World');
        $client->request();
    }
}
```

Memento

Purpose

It provides the ability to restore an object to it's previous state (undo via rollback) or to gain access to state of the object, without revealing it's implementation (i.e., the object is not required to have a function to return the current state).

The memento pattern is implemented with three objects: the Originator, a Caretaker and a Memento.

Memento – an object that *contains a concrete unique snapshot of state* of any object or resource: string, number, array, an instance of class and so on. The uniqueness in this case does not imply the prohibition existence of similar states in different snapshots. That means the state can be extracted as the independent clone. Any object stored in the Memento should be *a full copy of the original object rather than a reference* to the original object. The Memento object is a "opaque object" (the object that no one can or should change).

Originator – it is an object that contains the *actual state of an external object is strictly specified type*. Originator is able to create a unique copy of this state and return it wrapped in a Memento. The Originator does not know the history of changes. You can set a concrete state to Originator from the outside, which will be considered as actual. The Originator must make sure that given state corresponds the allowed type of object. Originator may (but not should) have any methods, but they *they can't make changes to the saved object state*.

Caretaker *controls the states history*. He may make changes to an object; take a decision to save the state of an external object in the Originator; ask from the Originator snapshot of the current state; or set the Originator state to equivalence with some snapshot from history.

Examples

- The seed of a pseudorandom number generator
- The state in a finite state machine
- Control for intermediate states of ORM Model before saving

UML Diagram

Code

You can also find this code on GitHub

Memento.php

```
<?php

namespace DesignPatterns\Behavioral\Memento;

class Memento
{
```

```
    /**
     * @var State
     */
    private $state;

    /**
     * @param State $stateToSave
     */
    public function __construct(State $stateToSave)
    {
        $this->state = $stateToSave;
    }

    /**
     * @return State
     */
    public function getState()
    {
        return $this->state;
    }
}
```

State.php

```
<?php

namespace DesignPatterns\Behavioral\Memento;

class State
{
    const STATE_CREATED = 'created';
    const STATE_OPENED = 'opened';
    const STATE_ASSIGNED = 'assigned';
    const STATE_CLOSED = 'closed';

    /**
     * @var string
     */
    private $state;

    /**
     * @var string[]
     */
    private static $validStates = [
        self::STATE_CREATED,
        self::STATE_OPENED,
        self::STATE_ASSIGNED,
        self::STATE_CLOSED,
    ];

    /**
     * @param string $state
     */
    public function __construct(string $state)
    {
        self::ensureIsValidState($state);

        $this->state = $state;
```

```
    }

    private static function ensureIsValidState(string $state)
    {
        if (!in_array($state, self::$validStates)) {
            throw new \InvalidArgumentException('Invalid state given');
        }
    }

    public function __toString(): string
    {
        return $this->state;
    }
}
```

Ticket.php

```
<?php

namespace DesignPatterns\Behavioral\Memento;

/**
 * Ticket is the "Originator" in this implementation
 */
class Ticket
{
    /**
     * @var State
     */
    private $currentState;

    public function __construct()
    {
        $this->currentState = new State(State::STATE_CREATED);
    }

    public function open()
    {
        $this->currentState = new State(State::STATE_OPENED);
    }

    public function assign()
    {
        $this->currentState = new State(State::STATE_ASSIGNED);
    }

    public function close()
    {
        $this->currentState = new State(State::STATE_CLOSED);
    }

    public function saveToMemento(): Memento
    {
        return new Memento(clone $this->currentState);
    }

    public function restoreFromMemento(Memento $memento)
    {
```

```
        $this->currentState = $memento->getState();
    }

    public function getState(): State
    {
        return $this->currentState;
    }
}
```

Test

Tests/MementoTest.php

```
<?php

namespace DesignPatterns\Behavioral\Memento\Tests;

use DesignPatterns\Behavioral\Memento\State;
use DesignPatterns\Behavioral\Memento\Ticket;
use PHPUnit\Framework\TestCase;

class MementoTest extends TestCase
{
    public function testOpenTicketAssignAndSetBackToOpen()
    {
        $ticket = new Ticket();

        // open the ticket
        $ticket->open();
        $openedState = $ticket->getState();
        $this->assertEquals(State::STATE_OPENED, (string) $ticket->getState());

        $memento = $ticket->saveToMemento();

        // assign the ticket
        $ticket->assign();
        $this->assertEquals(State::STATE_ASSIGNED, (string) $ticket->getState());

        // now restore to the opened state, but verify that the state object has been␣
↪cloned for the memento
        $ticket->restoreFromMemento($memento);

        $this->assertEquals(State::STATE_OPENED, (string) $ticket->getState());
        $this->assertNotSame($openedState, $ticket->getState());
    }
}
```

Null Object

Purpose

NullObject is not a GoF design pattern but a schema which appears frequently enough to be considered a pattern. It has the following benefits:

- Client code is simplified

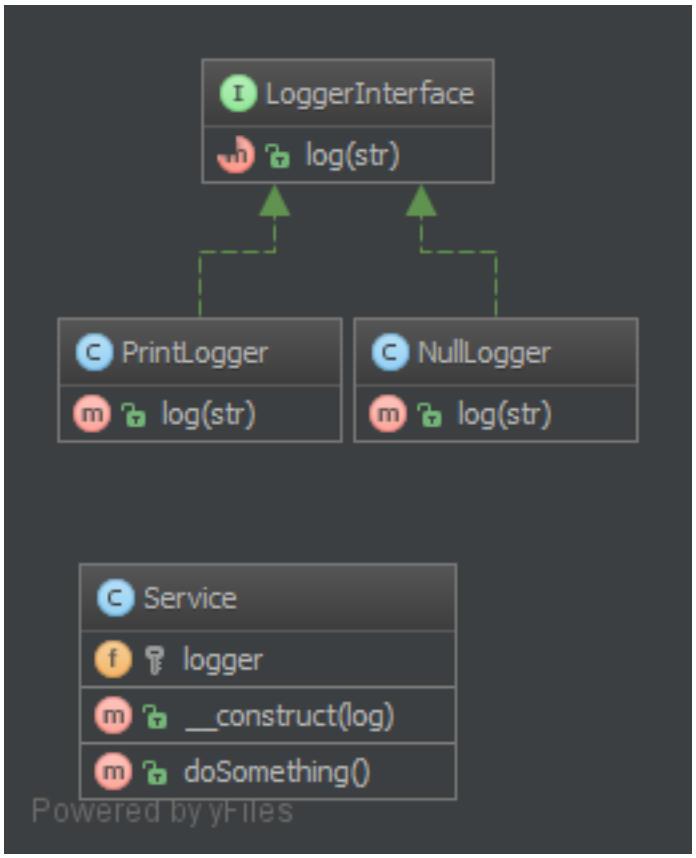
- Reduces the chance of null pointer exceptions
- Fewer conditionals require less test cases

Methods that return an object or null should instead return an object or `NullObject`. `NullObjects` simplify boilerplate code such as `if (!is_null($obj)) { $obj->callSomething(); }` to just `$obj->callSomething();` by eliminating the conditional check in client code.

Examples

- Symfony2: null logger of profiler
- Symfony2: null output in Symfony/Console
- null handler in a Chain of Responsibilities pattern
- null command in a Command pattern

UML Diagram



Code

You can also find this code on GitHub

Service.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    /**
     * @var LoggerInterface
     */
    private $logger;

    /**
     * @param LoggerInterface $logger
     */
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    /**
     * do something ...
     */
    public function doSomething()
    {
        // notice here that you don't have to check if the logger is set with eg. is_null(), instead just use it
        $this->logger->log('We are in '.__METHOD__);
    }
}
```

LoggerInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * Key feature: NullLogger must inherit from this interface like any other loggers
 */
interface LoggerInterface
{
    public function log(string $str);
}
```

PrintLogger.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class PrintLogger implements LoggerInterface
{
    public function log(string $str)
    {
        echo $str;
    }
}
```

NullLogger.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class NullLogger implements LoggerInterface
{
    public function log(string $str)
    {
        // do nothing
    }
}
```

Test

Tests/LoggerTest.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject\Tests;

use DesignPatterns\Behavioral\NullObject\NullLogger;
use DesignPatterns\Behavioral\NullObject\PrintLogger;
use DesignPatterns\Behavioral\NullObject\Service;
use PHPUnit\Framework\TestCase;

class LoggerTest extends TestCase
{
    public function testNullObject()
    {
        $service = new Service(new NullLogger());
        $this->expectOutputString('');
        $service->doSomething();
    }

    public function testStandardLogger()
    {
        $service = new Service(new PrintLogger());
        $this->expectOutputString('We are in␣
↪DesignPatterns\Behavioral\NullObject\Service::doSomething');
        $service->doSomething();
    }
}
```

Observer

Purpose

To implement a publish/subscribe behaviour to an object, whenever a “Subject” object changes its state, the attached “Observers” will be notified. It is used to shorten the amount of coupled objects and uses loose coupling instead.

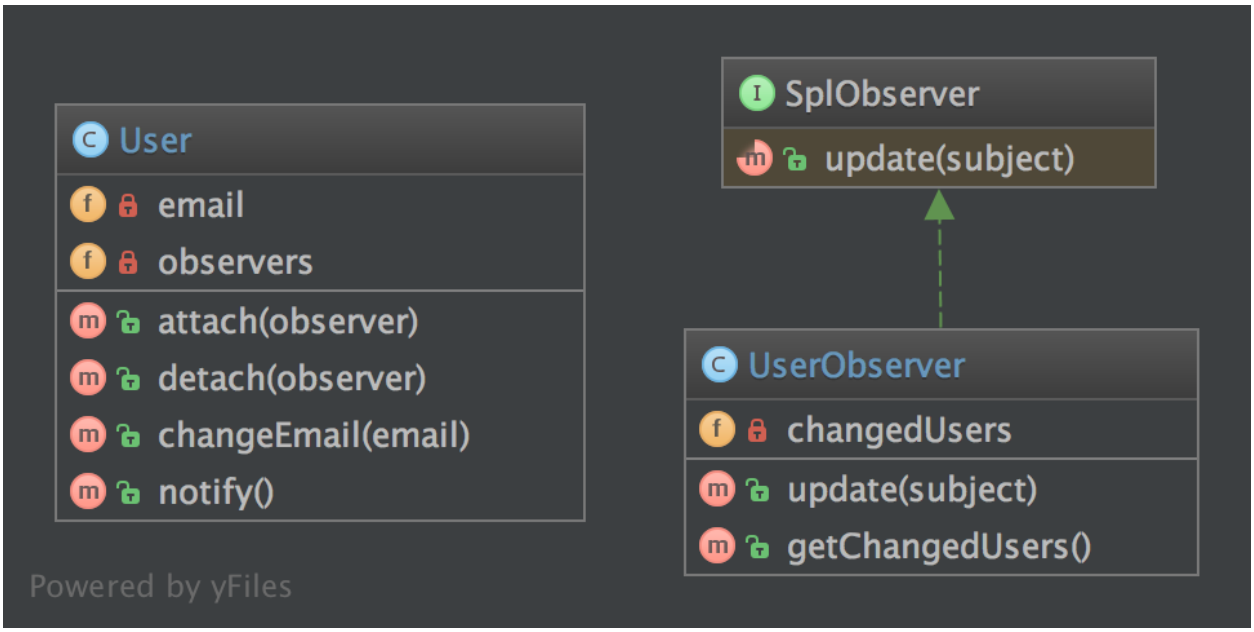
Examples

- a message queue system is observed to show the progress of a job in a GUI

Note

PHP already defines two interfaces that can help to implement this pattern: SplObserver and SplSubject.

UML Diagram



Code

You can also find this code on GitHub

User.php

```php
<?php

namespace DesignPatterns\Behavioral\Observer;

/**
 * User implements the observed object (called Subject), it maintains a list of observers and sends notifications to
 * them in case changes are made on the User object
 */
class User implements \SplSubject
{
    /**
     * @var string
     */
    private $email;

    /**
```

```
     * @var \SplObjectStorage
     */
    private $observers;

    public function __construct()
    {
        $this->observers = new \SplObjectStorage();
    }

    public function attach(\SplObserver $observer)
    {
        $this->observers->attach($observer);
    }

    public function detach(\SplObserver $observer)
    {
        $this->observers->detach($observer);
    }

    public function changeEmail(string $email)
    {
        $this->email = $email;
        $this->notify();
    }

    public function notify()
    {
        /** @var \SplObserver $observer */
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }
}
```

UserObserver.php

```
<?php

namespace DesignPatterns\Behavioral\Observer;

class UserObserver implements \SplObserver
{
    /**
     * @var User[]
     */
    private $changedUsers = [];

    /**
     * It is called by the Subject, usually by SplSubject::notify()
     *
     * @param \SplSubject $subject
     */
    public function update(\SplSubject $subject)
    {
        $this->changedUsers[] = clone $subject;
    }

    /**
```

```
     * @return User[]
     */
    public function getChangedUsers(): array
    {
        return $this->changedUsers;
    }
}
```

Test

Tests/ObserverTest.php

```
<?php

namespace DesignPatterns\Behavioral\Observer\Tests;

use DesignPatterns\Behavioral\Observer\User;
use DesignPatterns\Behavioral\Observer\UserObserver;
use PHPUnit\Framework\TestCase;

class ObserverTest extends TestCase
{
    public function testChangeInUserLeadsToUserObserverBeingNotified()
    {
        $observer = new UserObserver();

        $user = new User();
        $user->attach($observer);

        $user->changeEmail('foo@bar.com');
        $this->assertCount(1, $observer->getChangedUsers());
    }
}
```

Specification

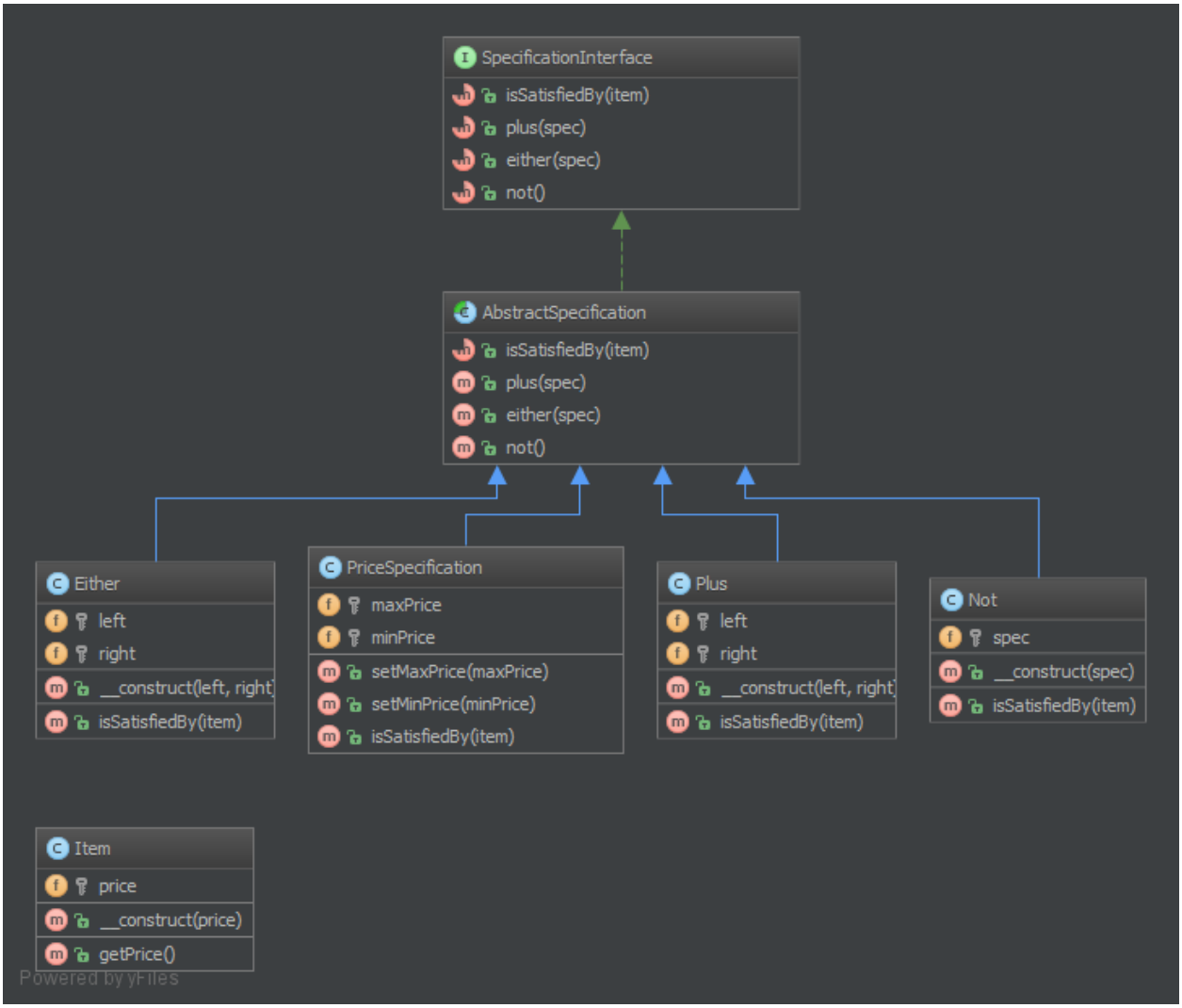
Purpose

Builds a clear specification of business rules, where objects can be checked against. The composite specification class has one method called `isSatisfiedBy` that returns either true or false depending on whether the given object satisfies the specification.

Examples

- RulerZ

UML Diagram



Code

You can also find this code on GitHub

Item.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

class Item
{
    /**
     * @var float
     */
    private $price;

    public function __construct(float $price)
```

```
    {
        $this->price = $price;
    }

    public function getPrice(): float
    {
        return $this->price;
    }
}
```

SpecificationInterface.php

```
<?php

namespace DesignPatterns\Behavioral\Specification;

interface SpecificationInterface
{
    public function isSatisfiedBy(Item $item): bool;
}
```

OrSpecification.php

```
<?php

namespace DesignPatterns\Behavioral\Specification;

class OrSpecification implements SpecificationInterface
{
    /**
     * @var SpecificationInterface[]
     */
    private $specifications;

    /**
     * @param SpecificationInterface[] ...$specifications
     */
    public function __construct(SpecificationInterface ...$specifications)
    {
        $this->specifications = $specifications;
    }

    /**
     * if at least one specification is true, return true, else return false
     */
    public function isSatisfiedBy(Item $item): bool
    {
        foreach ($this->specifications as $specification) {
            if ($specification->isSatisfiedBy($item)) {
                return true;
            }
        }
        return false;
    }
}
```

PriceSpecification.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

class PriceSpecification implements SpecificationInterface
{
    /**
     * @var float|null
     */
    private $maxPrice;

    /**
     * @var float|null
     */
    private $minPrice;

    /**
     * @param float $minPrice
     * @param float $maxPrice
     */
    public function __construct($minPrice, $maxPrice)
    {
        $this->minPrice = $minPrice;
        $this->maxPrice = $maxPrice;
    }

    public function isSatisfiedBy(Item $item): bool
    {
        if ($this->maxPrice !== null && $item->getPrice() > $this->maxPrice) {
            return false;
        }

        if ($this->minPrice !== null && $item->getPrice() < $this->minPrice) {
            return false;
        }

        return true;
    }
}
```

AndSpecification.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

class AndSpecification implements SpecificationInterface
{
    /**
     * @var SpecificationInterface[]
     */
    private $specifications;

    /**
     * @param SpecificationInterface[] ...$specifications
     */
    public function __construct(SpecificationInterface ...$specifications)
    {
```

```
        $this->specifications = $specifications;
    }

    /**
     * if at least one specification is false, return false, else return true.
     */
    public function isSatisfiedBy(Item $item): bool
    {
        foreach ($this->specifications as $specification) {
            if (!$specification->isSatisfiedBy($item)) {
                return false;
            }
        }

        return true;
    }
}
```

NotSpecification.php

```
<?php

namespace DesignPatterns\Behavioral\Specification;

class NotSpecification implements SpecificationInterface
{
    /**
     * @var SpecificationInterface
     */
    private $specification;

    public function __construct(SpecificationInterface $specification)
    {
        $this->specification = $specification;
    }

    public function isSatisfiedBy(Item $item): bool
    {
        return !$this->specification->isSatisfiedBy($item);
    }
}
```

Test

Tests/SpecificationTest.php

```
<?php

namespace DesignPatterns\Behavioral\Specification\Tests;

use DesignPatterns\Behavioral\Specification\Item;
use DesignPatterns\Behavioral\Specification\NotSpecification;
use DesignPatterns\Behavioral\Specification\OrSpecification;
use DesignPatterns\Behavioral\Specification\AndSpecification;
use DesignPatterns\Behavioral\Specification\PriceSpecification;
use PHPUnit\Framework\TestCase;
```

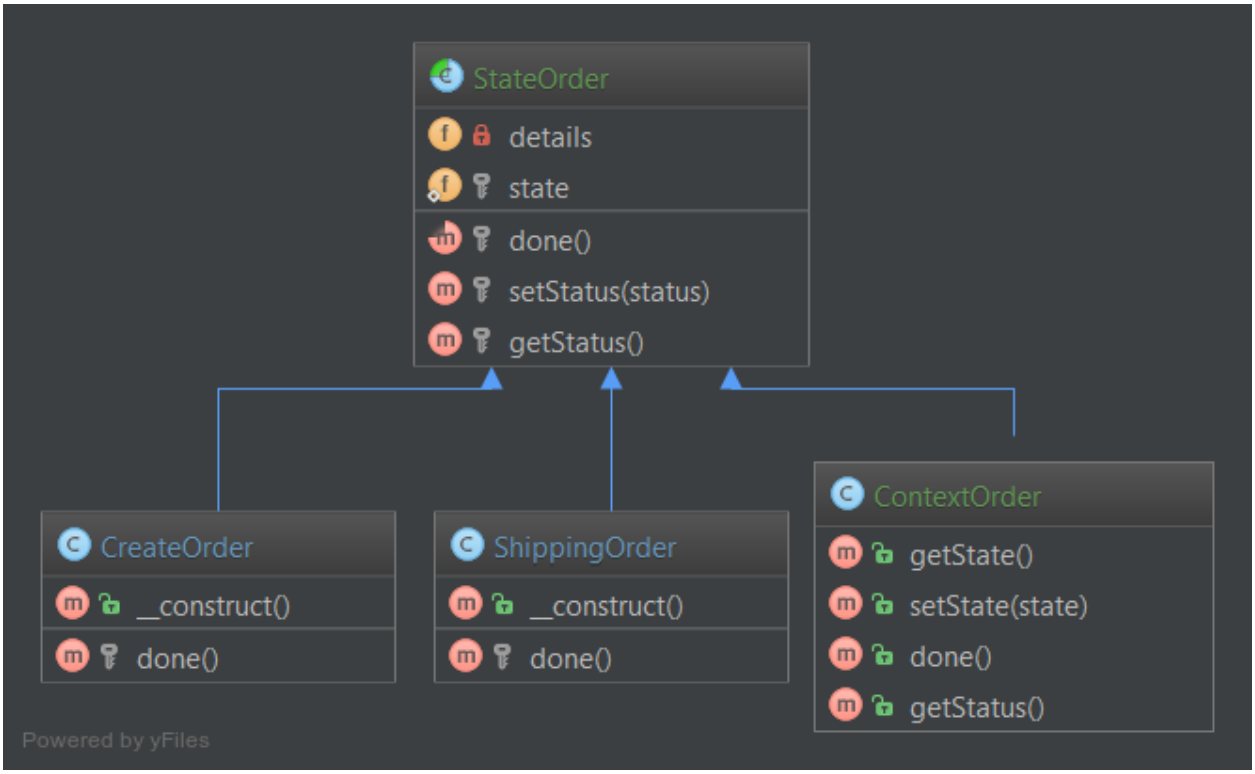
```php

class SpecificationTest extends TestCase
{
    public function testCanOr()
    {
        $spec1 = new PriceSpecification(50, 99);
        $spec2 = new PriceSpecification(101, 200);

        $orSpec = new OrSpecification($spec1, $spec2);

        $this->assertFalse($orSpec->isSatisfiedBy(new Item(100)));
        $this->assertTrue($orSpec->isSatisfiedBy(new Item(51)));
        $this->assertTrue($orSpec->isSatisfiedBy(new Item(150)));
    }

    public function testCanAnd()
    {
        $spec1 = new PriceSpecification(50, 100);
        $spec2 = new PriceSpecification(80, 200);

        $andSpec = new AndSpecification($spec1, $spec2);

        $this->assertFalse($andSpec->isSatisfiedBy(new Item(150)));
        $this->assertFalse($andSpec->isSatisfiedBy(new Item(1)));
        $this->assertFalse($andSpec->isSatisfiedBy(new Item(51)));
        $this->assertTrue($andSpec->isSatisfiedBy(new Item(100)));
    }

    public function testCanNot()
    {
        $spec1 = new PriceSpecification(50, 100);
        $notSpec = new NotSpecification($spec1);

        $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));
        $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));
    }
}
```

State

Purpose

Encapsulate varying behavior for the same routine based on an object's state. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements.

UML Diagram



Code

You can also find this code on GitHub

ContextOrder.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

class ContextOrder extends StateOrder
{
    public function getState():StateOrder
    {
        return static::$state;
    }

    public function setState(StateOrder $state)
    {
        static::$state = $state;
    }

    public function done()
    {
        static::$state->done();
    }

    public function getStatus(): string
```

```
    {
        return static::$state->getStatus();
    }
}
```

StateOrder.php

```
<?php

namespace DesignPatterns\Behavioral\State;

abstract class StateOrder
{
    /**
     * @var array
     */
    private $details;

    /**
     * @var StateOrder $state
     */
    protected static $state;

    /**
     * @return mixed
     */
    abstract protected function done();

    protected function setStatus(string $status)
    {
        $this->details['status'] = $status;
        $this->details['updatedTime'] = time();
    }

    protected function getStatus(): string
    {
        return $this->details['status'];
    }
}
```

ShippingOrder.php

```
<?php

namespace DesignPatterns\Behavioral\State;

class ShippingOrder extends StateOrder
{
    public function __construct()
    {
        $this->setStatus('shipping');
    }

    protected function done()
    {
        $this->setStatus('completed');
    }
}
```

CreateOrder.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

class CreateOrder extends StateOrder
{
    public function __construct()
    {
        $this->setStatus('created');
    }

    protected function done()
    {
        static::$state = new ShippingOrder();
    }
}
```

Test

Tests/StateTest.php

```php
<?php

namespace DesignPatterns\Behavioral\State\Tests;

use DesignPatterns\Behavioral\State\ContextOrder;
use DesignPatterns\Behavioral\State\CreateOrder;
use PHPUnit\Framework\TestCase;

class StateTest extends TestCase
{
    public function testCanShipCreatedOrder()
    {
        $order = new CreateOrder();
        $contextOrder = new ContextOrder();
        $contextOrder->setState($order);
        $contextOrder->done();

        $this->assertEquals('shipping', $contextOrder->getStatus());
    }

    public function testCanCompleteShippedOrder()
    {
        $order = new CreateOrder();
        $contextOrder = new ContextOrder();
        $contextOrder->setState($order);
        $contextOrder->done();
        $contextOrder->done();

        $this->assertEquals('completed', $contextOrder->getStatus());
    }
}
```

Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy

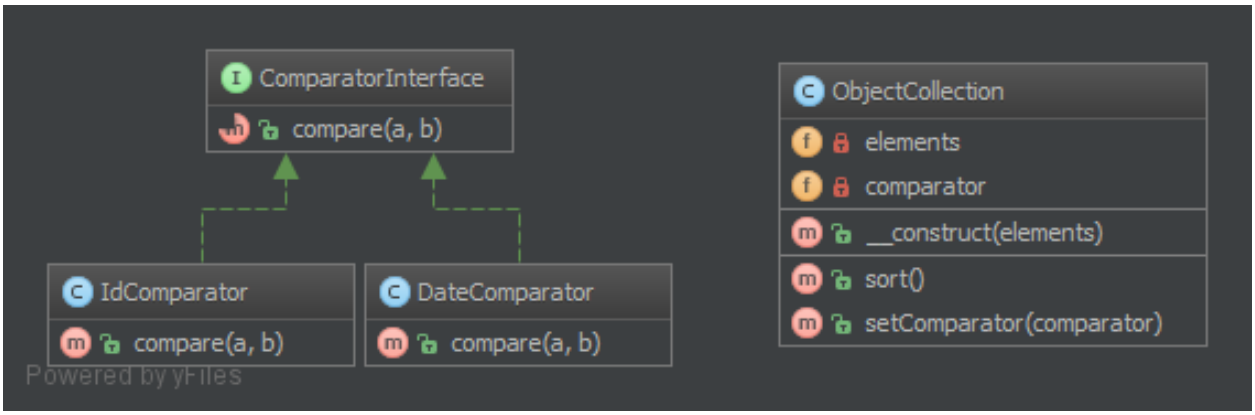
Purpose

To separate strategies and to enable fast switching between them. Also this pattern is a good alternative to inheritance (instead of having an abstract class that is extended).

Examples

- sorting a list of objects, one strategy by date, the other by id
- simplify unit testing: e.g. switching between file and in-memory storage

UML Diagram



Code

You can also find this code on GitHub

ObjectCollection.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

class ObjectCollection
{
    /**
     * @var array
     */
    private $elements;

```

```php
    /**
     * @var ComparatorInterface
     */
    private $comparator;

    /**
     * @param array $elements
     */
    public function __construct(array $elements = [])
    {
        $this->elements = $elements;
    }

    public function sort(): array
    {
        if (!$this->comparator) {
            throw new \LogicException('Comparator is not set');
        }

        uasort($this->elements, [$this->comparator, 'compare']);

        return $this->elements;
    }

    /**
     * @param ComparatorInterface $comparator
     */
    public function setComparator(ComparatorInterface $comparator)
    {
        $this->comparator = $comparator;
    }
}
```

ComparatorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

interface ComparatorInterface
{
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return int
     */
    public function compare($a, $b): int;
}
```

DateComparator.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

class DateComparator implements ComparatorInterface
{
```

```
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return int
     */
    public function compare($a, $b): int
    {
        $aDate = new \DateTime($a['date']);
        $bDate = new \DateTime($b['date']);

        return $aDate <=> $bDate;
    }
}
```

IdComparator.php

```
<?php

namespace DesignPatterns\Behavioral\Strategy;

class IdComparator implements ComparatorInterface
{
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return int
     */
    public function compare($a, $b): int
    {
        return $a['id'] <=> $b['id'];
    }
}
```

Test

Tests/StrategyTest.php

```
<?php

namespace DesignPatterns\Behavioral\Strategy\Tests;

use DesignPatterns\Behavioral\Strategy\DateComparator;
use DesignPatterns\Behavioral\Strategy\IdComparator;
use DesignPatterns\Behavioral\Strategy\ObjectCollection;
use PHPUnit\Framework\TestCase;

class StrategyTest extends TestCase
{
    public function provideIntegers()
    {
        return [
            [
                [['id' => 2], ['id' => 1], ['id' => 3]],
                ['id' => 1],
```

```
            ],
            [
                [['id' => 3], ['id' => 2], ['id' => 1]],
                ['id' => 1],
            ],
        ];
    }

    public function provideDates()
    {
        return [
            [
                [['date' => '2014-03-03'], ['date' => '2015-03-02'], ['date' => '2013-03-01']],
                ['date' => '2013-03-01'],
            ],
            [
                [['date' => '2014-02-03'], ['date' => '2013-02-01'], ['date' => '2015-02-02']],
                ['date' => '2013-02-01'],
            ],
        ];
    }

    /**
     * @dataProvider provideIntegers
     *
     * @param array $collection
     * @param array $expected
     */
    public function testIdComparator($collection, $expected)
    {
        $obj = new ObjectCollection($collection);
        $obj->setComparator(new IdComparator());
        $elements = $obj->sort();

        $firstElement = array_shift($elements);
        $this->assertEquals($expected, $firstElement);
    }

    /**
     * @dataProvider provideDates
     *
     * @param array $collection
     * @param array $expected
     */
    public function testDateComparator($collection, $expected)
    {
        $obj = new ObjectCollection($collection);
        $obj->setComparator(new DateComparator());
        $elements = $obj->sort();

        $firstElement = array_shift($elements);
        $this->assertEquals($expected, $firstElement);
    }
}
```

Template Method

Purpose

Template Method is a behavioral design pattern.

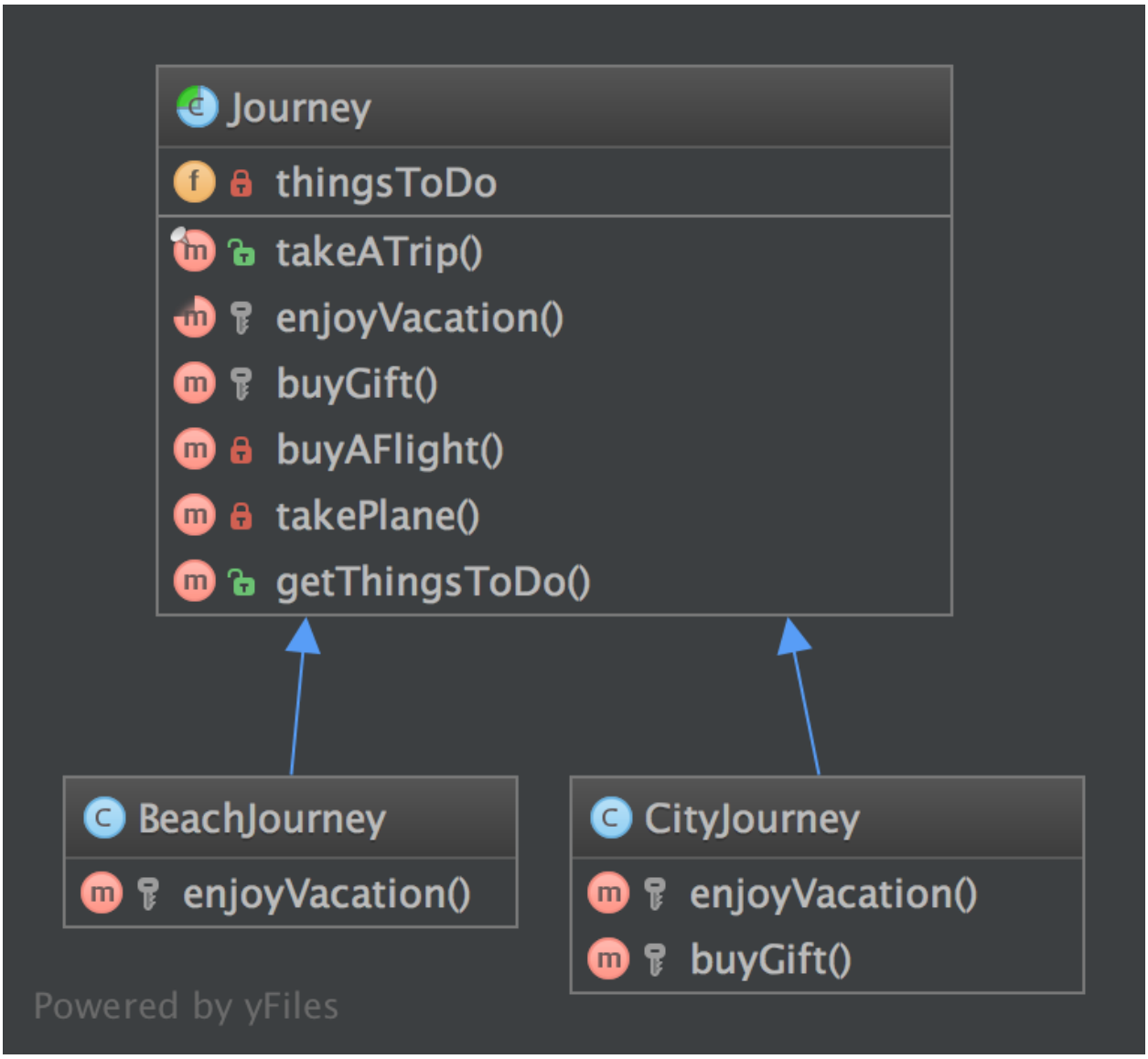
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template "finish" the behavior of an algorithm.

A.k.a the "Hollywood principle": "Don't call us, we call you." This class is not called by subclasses but the inverse. How? With abstraction of course.

In other words, this is a skeleton of algorithm, well-suited for framework libraries. The user has just to implement one method and the superclass do the job.

It is an easy way to decouple concrete classes and reduce copy-paste, that's why you'll find it everywhere.

UML Diagram



Code

You can also find this code on GitHub

Journey.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

abstract class Journey
{
    /**
     * @var string[]
     */
    private $thingsToDo = [];

    /**
     * This is the public service provided by this class and its subclasses.
     * Notice it is final to "freeze" the global behavior of algorithm.
     * If you want to override this contract, make an interface with only takeATrip()
     * and subclass it.
     */
    final public function takeATrip()
    {
        $this->thingsToDo[] = $this->buyAFlight();
        $this->thingsToDo[] = $this->takePlane();
        $this->thingsToDo[] = $this->enjoyVacation();
        $buyGift = $this->buyGift();

        if ($buyGift !== null) {
            $this->thingsToDo[] = $buyGift;
        }

        $this->thingsToDo[] = $this->takePlane();
    }

    /**
     * This method must be implemented, this is the key-feature of this pattern.
     */
    abstract protected function enjoyVacation(): string;

    /**
     * This method is also part of the algorithm but it is optional.
     * You can override it only if you need to
     *
     * @return null|string
     */
    protected function buyGift()
    {
        return null;
    }

    private function buyAFlight(): string
    {
        return 'Buy a flight ticket';
    }

    private function takePlane(): string
```

```
    {
        return 'Taking the plane';
    }

    /**
     * @return string[]
     */
    public function getThingsToDo(): array
    {
        return $this->thingsToDo;
    }
}
```

BeachJourney.php

```
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

class BeachJourney extends Journey
{
    protected function enjoyVacation(): string
    {
        return "Swimming and sun-bathing";
    }
}
```

CityJourney.php

```
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

class CityJourney extends Journey
{
    protected function enjoyVacation(): string
    {
        return "Eat, drink, take photos and sleep";
    }

    protected function buyGift(): string
    {
        return "Buy a gift";
    }
}
```

Test

Tests/JourneyTest.php

```
<?php

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

use DesignPatterns\Behavioral\TemplateMethod;
use PHPUnit\Framework\TestCase;
```

```php

class JourneyTest extends TestCase
{
    public function testCanGetOnVacationOnTheBeach()
    {
        $beachJourney = new TemplateMethod\BeachJourney();
        $beachJourney->takeATrip();

        $this->assertEquals(
            ['Buy a flight ticket', 'Taking the plane', 'Swimming and sun-bathing',
↪'Taking the plane'],
            $beachJourney->getThingsToDo()
        );
    }

    public function testCanGetOnAJourneyToACity()
    {
        $beachJourney = new TemplateMethod\CityJourney();
        $beachJourney->takeATrip();

        $this->assertEquals(
            [
                'Buy a flight ticket',
                'Taking the plane',
                'Eat, drink, take photos and sleep',
                'Buy a gift',
                'Taking the plane'
            ],
            $beachJourney->getThingsToDo()
        );
    }
}
```

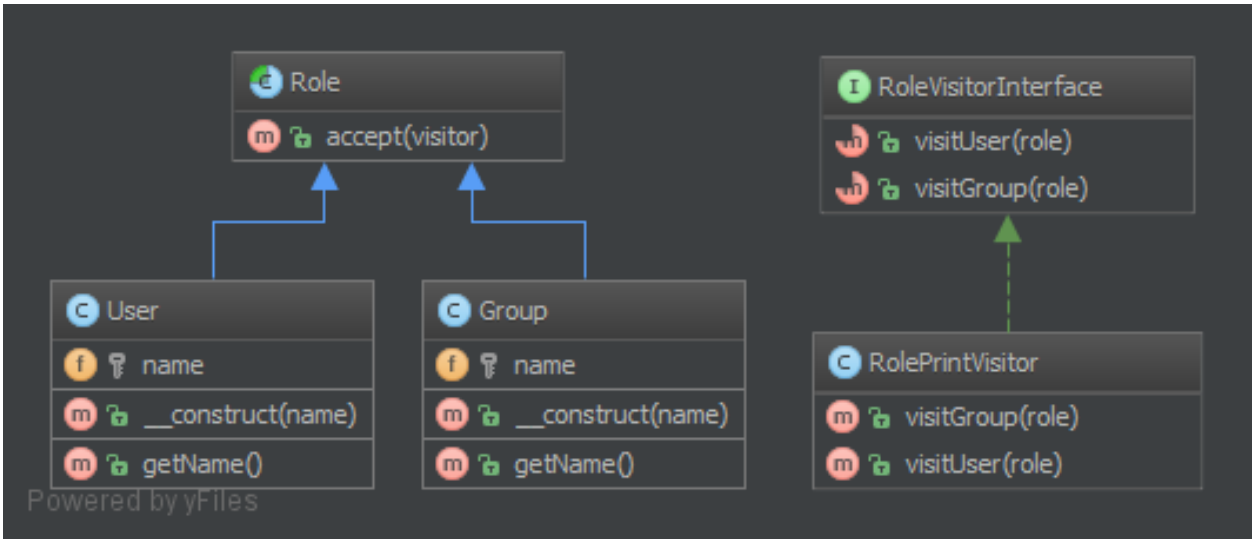
Visitor

Purpose

The Visitor Pattern lets you outsource operations on objects to other objects. The main reason to do this is to keep a separation of concerns. But classes have to define a contract to allow visitors (the `Role::accept` method in the example).

The contract is an abstract class but you can have also a clean interface. In that case, each Visitor has to choose itself which method to invoke on the visitor.

UML Diagram



Code

You can also find this code on GitHub

RoleVisitorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

/**
 * Note: the visitor must not choose itself which method to
 * invoke, it is the Visitee that make this decision
 */
interface RoleVisitorInterface
{
    public function visitUser(User $role);

    public function visitGroup(Group $role);
}
```

RoleVisitor.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

class RoleVisitor implements RoleVisitorInterface
{
    /**
     * @var Role[]
     */
    private $visited = [];

    public function visitGroup(Group $role)
    {
```

```
        $this->visited[] = $role;
    }

    public function visitUser(User $role)
    {
        $this->visited[] = $role;
    }

    /**
     * @return Role[]
     */
    public function getVisited(): array
    {
        return $this->visited;
    }
}
```

Role.php

```
<?php

namespace DesignPatterns\Behavioral\Visitor;

interface Role
{
    public function accept(RoleVisitorInterface $visitor);
}
```

User.php

```
<?php

namespace DesignPatterns\Behavioral\Visitor;

class User implements Role
{
    /**
     * @var string
     */
    private $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function getName(): string
    {
        return sprintf('User %s', $this->name);
    }

    public function accept(RoleVisitorInterface $visitor)
    {
        $visitor->visitUser($this);
    }
}
```

Group.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

class Group implements Role
{
    /**
     * @var string
     */
    private $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function getName(): string
    {
        return sprintf('Group: %s', $this->name);
    }

    public function accept(RoleVisitorInterface $visitor)
    {
        $visitor->visitGroup($this);
    }
}
```

Test

Tests/VisitorTest.php

```php
<?php

namespace DesignPatterns\Tests\Visitor\Tests;

use DesignPatterns\Behavioral\Visitor;
use PHPUnit\Framework\TestCase;

class VisitorTest extends TestCase
{
    /**
     * @var Visitor\RoleVisitor
     */
    private $visitor;

    protected function setUp()
    {
        $this->visitor = new Visitor\RoleVisitor();
    }

    public function provideRoles()
    {
        return [
            [new Visitor\User('Dominik')],
            [new Visitor\Group('Administrators')],
        ];
    }
```

```
27
28      /**
29       * @dataProvider provideRoles
30       *
31       * @param Visitor\Role $role
32       */
33      public function testVisitSomeRole(Visitor\Role $role)
34      {
35          $role->accept($this->visitor);
36          $this->assertSame($role, $this->visitor->getVisited()[0]);
37      }
38  }
```

More

Delegation

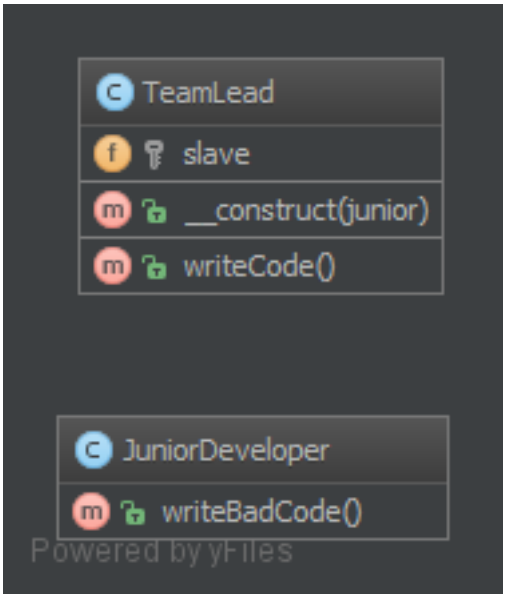
Purpose

Demonstrate the Delegator pattern, where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. In this case TeamLead professes to writeCode and Usage uses this, while TeamLead delegates writeCode to JuniorDeveloper's writeBadCode function. This inverts the responsibility so that Usage is unknowingly executing writeBadCode.

Examples

Please review JuniorDeveloper.php, TeamLead.php, and then Usage.php to see it all tied together.

UML Diagram



Code

You can also find this code on GitHub

TeamLead.php

```php
<?php

namespace DesignPatterns\More\Delegation;

class TeamLead
{
    /**
     * @var JuniorDeveloper
     */
    private $junior;

    /**
     * @param JuniorDeveloper $junior
     */
    public function __construct(JuniorDeveloper $junior)
    {
        $this->junior = $junior;
    }

    public function writeCode(): string
    {
        return $this->junior->writeBadCode();
    }
}
```

JuniorDeveloper.php

```php
<?php

namespace DesignPatterns\More\Delegation;

class JuniorDeveloper
{
    public function writeBadCode(): string
    {
        return 'Some junior developer generated code...';
    }
}
```

Test

Tests/DelegationTest.php

```php
<?php

namespace DesignPatterns\More\Delegation\Tests;

use DesignPatterns\More\Delegation;
use PHPUnit\Framework\TestCase;

class DelegationTest extends TestCase
```

```
{
    public function testHowTeamLeadWriteCode()
    {
        $junior = new Delegation\JuniorDeveloper();
        $teamLead = new Delegation\TeamLead($junior);

        $this->assertEquals($junior->writeBadCode(), $teamLead->writeCode());
    }
}
```

Service Locator

THIS IS CONSIDERED TO BE AN ANTI-PATTERN!

Service Locator is considered for some people an anti-pattern. It violates the Dependency Inversion principle. Service Locator hides class' dependencies instead of exposing them as you would do using the Dependency Injection. In case of changes of those dependencies you risk to break the functionality of classes which are using them, making your system difficult to maintain.

Purpose

To implement a loosely coupled architecture in order to get better testable, maintainable and extendable code. DI pattern and Service Locator pattern are an implementation of the Inverse of Control pattern.

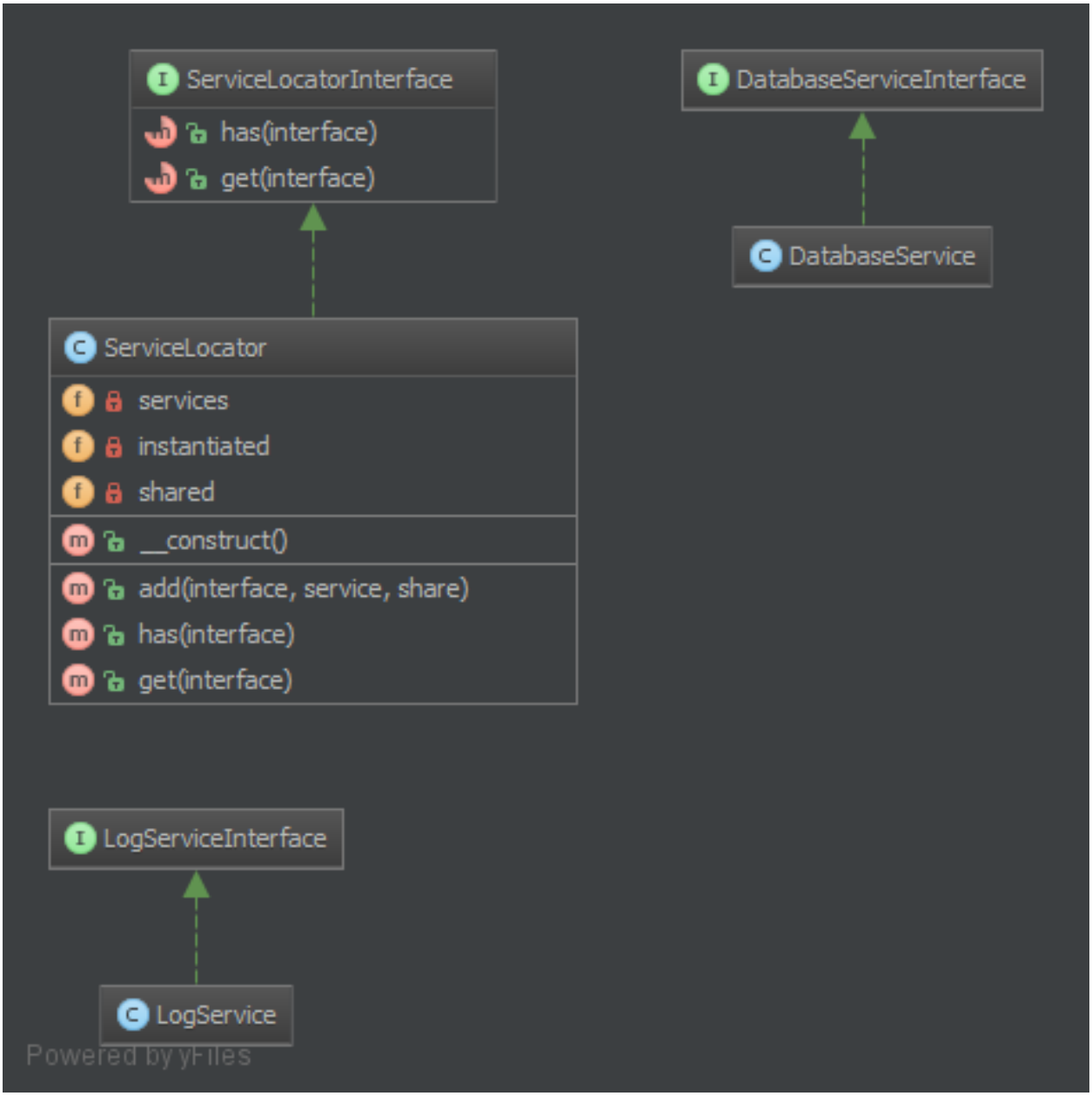
Usage

With `ServiceLocator` you can register a service for a given interface. By using the interface you can retrieve the service and use it in the classes of the application without knowing its implementation. You can configure and inject the Service Locator object on bootstrap.

Examples

- Zend Framework 2 uses Service Locator to create and share services used in the framework(i.e. EventManager, ModuleManager, all custom user services provided by modules, etc...)

UML Diagram



Code

You can also find this code on GitHub

ServiceLocator.php

```php
<?php

namespace DesignPatterns\More\ServiceLocator;

class ServiceLocator
{
```

```php
    /**
     * @var array
     */
    private $services = [];

    /**
     * @var array
     */
    private $instantiated = [];

    /**
     * @var array
     */
    private $shared = [];

    /**
     * instead of supplying a class here, you could also store a service for an interface
     *
     * @param string $class
     * @param object $service
     * @param bool $share
     */
    public function addInstance(string $class, $service, bool $share = true)
    {
        $this->services[$class] = $service;
        $this->instantiated[$class] = $service;
        $this->shared[$class] = $share;
    }

    /**
     * instead of supplying a class here, you could also store a service for an interface
     *
     * @param string $class
     * @param array $params
     * @param bool $share
     */
    public function addClass(string $class, array $params, bool $share = true)
    {
        $this->services[$class] = $params;
        $this->shared[$class] = $share;
    }

    public function has(string $interface): bool
    {
        return isset($this->services[$interface]) || isset($this->instantiated[$interface]);
    }

    /**
     * @param string $class
     *
     * @return object
     */
    public function get(string $class)
    {
        if (isset($this->instantiated[$class]) && $this->shared[$class]) {
```

```
            return $this->instantiated[$class];
        }

        $args = $this->services[$class];

        switch (count($args)) {
            case 0:
                $object = new $class();
                break;
            case 1:
                $object = new $class($args[0]);
                break;
            case 2:
                $object = new $class($args[0], $args[1]);
                break;
            case 3:
                $object = new $class($args[0], $args[1], $args[2]);
                break;
            default:
                throw new \OutOfRangeException('Too many arguments given');
        }

        if ($this->shared[$class]) {
            $this->instantiated[$class] = $object;
        }

        return $object;
    }
}
```

LogService.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

class LogService
{
}
```

Test

Tests/ServiceLocatorTest.php

```
<?php

namespace DesignPatterns\More\ServiceLocator\Tests;

use DesignPatterns\More\ServiceLocator\LogService;
use DesignPatterns\More\ServiceLocator\ServiceLocator;
use PHPUnit\Framework\TestCase;

class ServiceLocatorTest extends TestCase
{
    /**
     * @var ServiceLocator
```

```
13          */
14      private $serviceLocator;
15
16      public function setUp()
17      {
18          $this->serviceLocator = new ServiceLocator();
19      }
20
21      public function testHasServices()
22      {
23          $this->serviceLocator->addInstance(LogService::class, new LogService());
24
25          $this->assertTrue($this->serviceLocator->has(LogService::class));
26          $this->assertFalse($this->serviceLocator->has(self::class));
27      }
28
29      public function testGetWillInstantiateLogServiceIfNoInstanceHasBeenCreatedYet()
30      {
31          $this->serviceLocator->addClass(LogService::class, []);
32          $logger = $this->serviceLocator->get(LogService::class);
33
34          $this->assertInstanceOf(LogService::class, $logger);
35      }
36  }
```

Repository

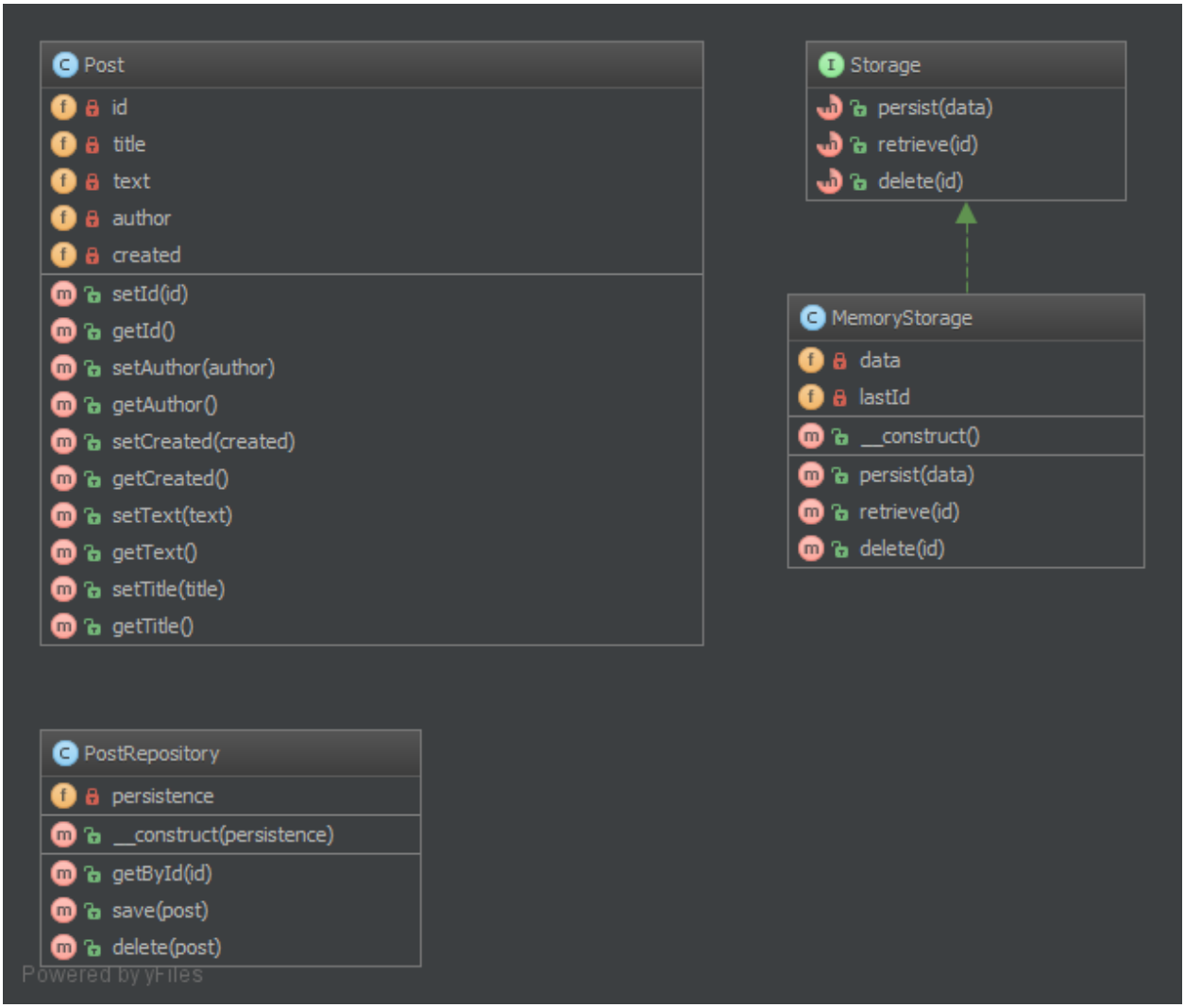
Purpose

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

Examples

- Doctrine 2 ORM: there is Repository that mediates between Entity and DBAL and contains methods to retrieve objects
- Laravel Framework

UML Diagram



Code

You can also find this code on GitHub

Post.php

```php
<?php

namespace DesignPatterns\More\Repository;

class Post
{
    /**
     * @var int|null
     */
    private $id;

    /**
```

```
     * @var string
     */
    private $title;

    /**
     * @var string
     */
    private $text;

    public static function fromState(array $state): Post
    {
        return new self(
            $state['id'],
            $state['title'],
            $state['text']
        );
    }

    /**
     * @param int|null $id
     * @param string $text
     * @param string $title
     */
    public function __construct($id, string $title, string $text)
    {
        $this->id = $id;
        $this->text = $text;
        $this->title = $title;
    }

    public function setId(int $id)
    {
        $this->id = $id;
    }

    public function getId(): int
    {
        return $this->id;
    }

    public function getText(): string
    {
        return $this->text;
    }

    public function getTitle(): string
    {
        return $this->title;
    }
}
```

PostRepository.php

```
<?php

namespace DesignPatterns\More\Repository;

/**
```

```php
 * This class is situated between Entity layer (class Post) and access object layer (MemoryStorage).
 *
 * Repository encapsulates the set of objects persisted in a data store and the operations performed over them
 * providing a more object-oriented view of the persistence layer
 *
 * Repository also supports the objective of achieving a clean separation and one-way dependency
 * between the domain and data mapping layers
 */
class PostRepository
{
    /**
     * @var MemoryStorage
     */
    private $persistence;

    public function __construct(MemoryStorage $persistence)
    {
        $this->persistence = $persistence;
    }

    public function findById(int $id): Post
    {
        $arrayData = $this->persistence->retrieve($id);

        if (is_null($arrayData)) {
            throw new \InvalidArgumentException(sprintf('Post with ID %d does not exist', $id));
        }

        return Post::fromState($arrayData);
    }

    public function save(Post $post)
    {
        $id = $this->persistence->persist([
            'text' => $post->getText(),
            'title' => $post->getTitle(),
        ]);

        $post->setId($id);
    }
}
```

MemoryStorage.php

```php
<?php

namespace DesignPatterns\More\Repository;

class MemoryStorage
{
    /**
     * @var array
     */
    private $data = [];
```

```php

    /**
     * @var int
     */
    private $lastId = 0;

    public function persist(array $data): int
    {
        $this->lastId++;

        $data['id'] = $this->lastId;
        $this->data[$this->lastId] = $data;

        return $this->lastId;
    }

    public function retrieve(int $id): array
    {
        if (!isset($this->data[$id])) {
            throw new \OutOfRangeException(sprintf('No data found for ID %d', $id));
        }

        return $this->data[$id];
    }

    public function delete(int $id)
    {
        if (!isset($this->data[$id])) {
            throw new \OutOfRangeException(sprintf('No data found for ID %d', $id));
        }

        unset($this->data[$id]);
    }
}
```

Test

Tests/RepositoryTest.php

```php
<?php

namespace DesignPatterns\More\Repository\Tests;

use DesignPatterns\More\Repository\MemoryStorage;
use DesignPatterns\More\Repository\Post;
use DesignPatterns\More\Repository\PostRepository;
use PHPUnit\Framework\TestCase;

class RepositoryTest extends TestCase
{
    public function testCanPersistAndFindPost()
    {
        $repository = new PostRepository(new MemoryStorage());
        $post = new Post(null, 'Repository Pattern', 'Design Patterns PHP');

        $repository->save($post);

```

```
19          $this->assertEquals(1, $post->getId());
20          $this->assertEquals($post->getId(), $repository->findById(1)->getId());
21      }
22  }
```

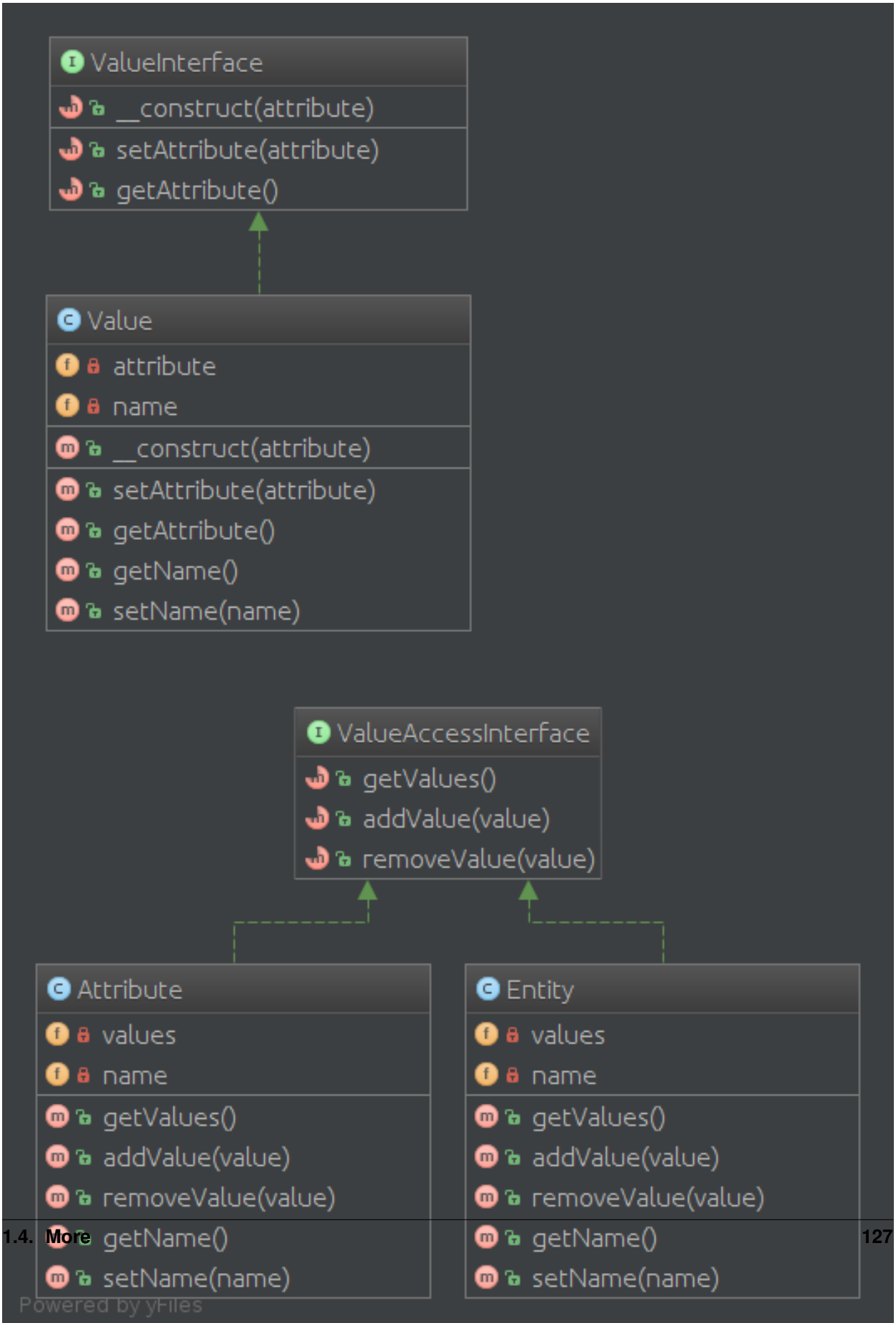
Entity-Attribute-Value (EAV)

The Entity–attribute–value (EAV) pattern in order to implement EAV model with PHP.

Purpose

The Entity–attribute–value (EAV) model is a data model to describe entities where the number of attributes (properties, parameters) that can be used to describe them is potentially vast, but the number that will actually apply to a given entity is relatively modest.

UML Diagram


ValueInterface
__construct(attribute)
setAttribute(attribute)
getAttribute()
Value
attribute
name
__construct(attribute)
setAttribute(attribute)
getAttribute()
getName()
setName(name)
ValueAccessInterface
getValues()
addValue(value)
removeValue(value)
Attribute
values
name
getValues()
addValue(value)
removeValue(value)
getName()
setName(name)
Entity
values
name
getValues()
addValue(value)
removeValue(value)
getName()
setName(name)
Powered by yFiles

Code

You can also find this code on GitHub

Entity.php

```php
<?php

namespace DesignPatterns\More\EAV;

class Entity
{
    /**
     * @var \SplObjectStorage
     */
    private $values;

    /**
     * @var string
     */
    private $name;

    /**
     * @param string $name
     * @param Value[] $values
     */
    public function __construct(string $name, $values)
    {
        $this->values = new \SplObjectStorage();
        $this->name = $name;

        foreach ($values as $value) {
            $this->values->attach($value);
        }
    }

    public function __toString(): string
    {
        $text = [$this->name];

        foreach ($this->values as $value) {
            $text[] = (string) $value;
        }

        return join(', ', $text);
    }
}
```

Attribute.php

```php
<?php

namespace DesignPatterns\More\EAV;

class Attribute
{
    /**
     * @var \SplObjectStorage
     */
```

```
    private $values;

    /**
     * @var string
     */
    private $name;

    public function __construct(string $name)
    {
        $this->values = new \SplObjectStorage();
        $this->name = $name;
    }

    public function addValue(Value $value)
    {
        $this->values->attach($value);
    }

    /**
     * @return \SplObjectStorage
     */
    public function getValues(): \SplObjectStorage
    {
        return $this->values;
    }

    public function __toString(): string
    {
        return $this->name;
    }
}
```

Value.php

```
<?php

namespace DesignPatterns\More\EAV;

class Value
{
    /**
     * @var Attribute
     */
    private $attribute;

    /**
     * @var string
     */
    private $name;

    public function __construct(Attribute $attribute, string $name)
    {
        $this->name = $name;
        $this->attribute = $attribute;

        $attribute->addValue($this);
    }

```

```
25      public function __toString(): string
26      {
27          return sprintf('%s: %s', $this->attribute, $this->name);
28      }
29  }
```

Test

Tests/EAVTest.php

CHAPTER 2

Contribute

If you encounter any bugs or missing translations, please feel free to fork and send a pull request with your changes. To establish a consistent code quality, please check your code using PHP CodeSniffer against PSR2 standard using `./vendor/bin/phpcs -p --standard=PSR2 --ignore=vendor .`.

CHAPTER 3

License

(The MIT License)