Padrões de Projetos PHP (DesignPatternsPHP)

Esta é uma coleção de padrões de projetos conhecidos e alguns códigos de exemplo de como implementá-los em PHP. Todo padrão tem uma pequena lista de exemplos.

Eu acredito que o problema com os padrões é que muitas pessoas os conhecem mas não sabem quando aplicá-los

Padrões

Os padrões podem ser estruturados grosseiramente em tres categorias diferentes. Por favor clique no **no título da página de cada padrão** para uma explicação completa do padrão na Wikipedia.

1. Criacional

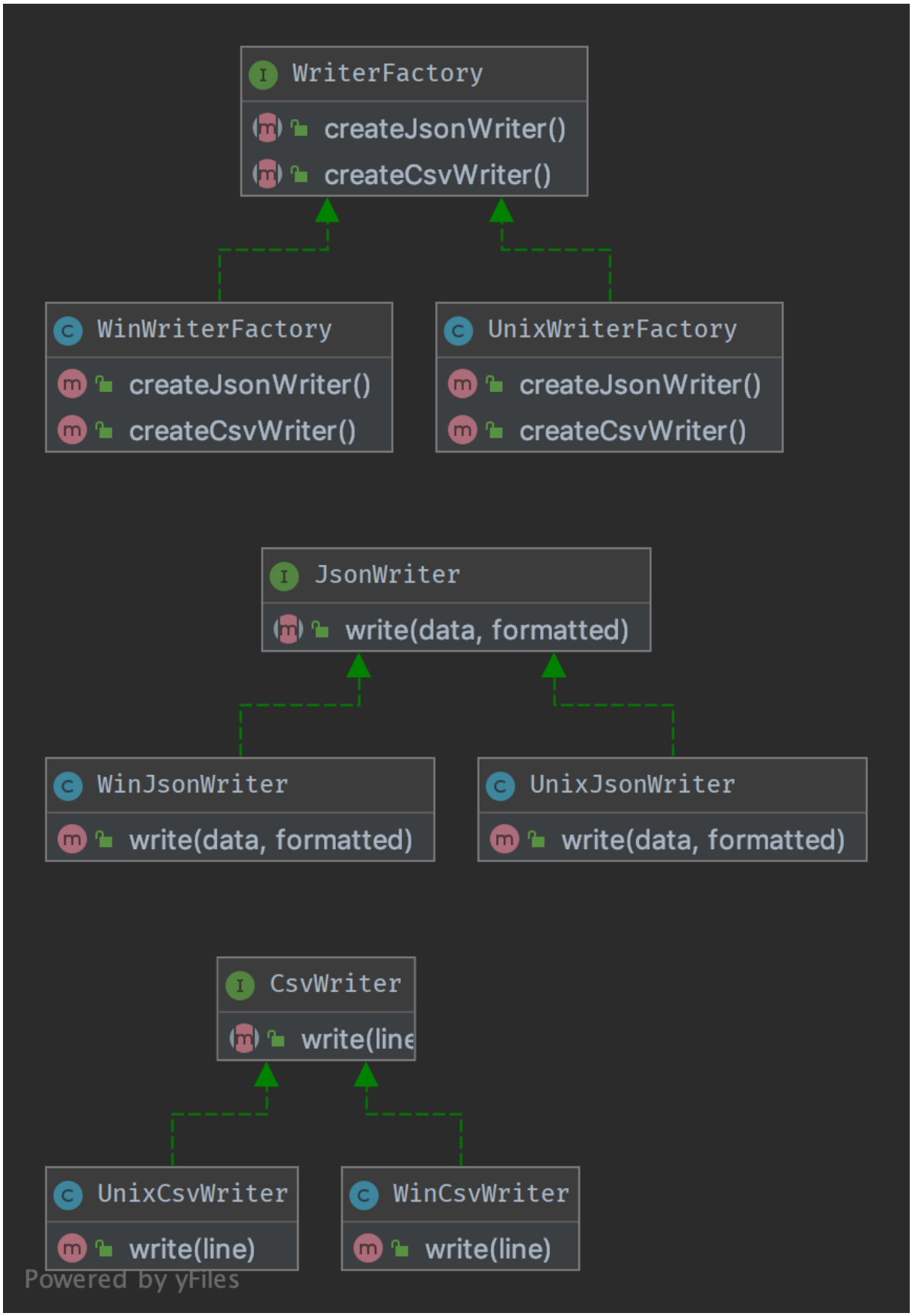
Em engenharia de software, padrões de projeto do tipo criacional são padrões que trabalham com mecanismos de criação de objetos, criando objetos de maneira adequada às situações. A forma básica para criação de objetos pode resultar em problemas de design ou adicionar complexidade ao mesmo. Padrões de Criação resolvem este problema mantendo a criação do objeto sob controle.

1.1. Fábrica de abstração (Abstract Factory)

1.1.1. Objetivo

Criar séries de objetos relacionados ou dependentes sem especificar suas classes concretas. Usualmente todas as classes criadas implementam a mesma interface. O cliente da fábrica de abstração não precisa se preocupar como estes objetos são criados, ele só sabe obtê-los.

1.1.2. Diagrama UML



1.1.3. Código

Você pode encontrar o código no Github

WriterFactory.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

interface WriterFactory
{
    public function createCsvWriter(): CsvWriter;
    public function createJsonWriter(): JsonWriter;
}
```

CsvWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

interface CsvWriter
{
    public function write(array $line): string;
}
```

JsonWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

interface JsonWriter
{
    public function write(array $data, bool $formatted): string;
}
```

UnixCsvWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class UnixCsvWriter implements CsvWriter
{
    public function write(array $line): string
    {
        return join(',', $line) . "\n";
    }
}
```

UnixJsonWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;
```

```
class UnixJsonWriter implements JsonWriter
{
    public function write(array $data, bool $formatted): string
    {
        $options = 0;

        if ($formatted) {
            $options = JSON_PRETTY_PRINT;
        }

        return json_encode($data, $options);
    }
}
```

UnixWriterFactory.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class UnixWriterFactory implements WriterFactory
{
    public function createCsvWriter(): CsvWriter
    {
        return new UnixCsvWriter();
    }

    public function createJsonWriter(): JsonWriter
    {
        return new UnixJsonWriter();
    }
}
```

WinCsvWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class WinCsvWriter implements CsvWriter
{
    public function write(array $line): string
    {
        return join(',', $line) . "\r\n";
    }
}
```

WinJsonWriter.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class WinJsonWriter implements JsonWriter
{
    public function write(array $data, bool $formatted): string
    {


        return json_encode($data, JSON_PRETTY_PRINT);
```

```
    }
}
```

WinWriterFactory.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class WinWriterFactory implements WriterFactory
{
    public function createCsvWriter(): CsvWriter
    {
        return new WinCsvWriter();
    }

    public function createJsonWriter(): JsonWriter
    {
        return new WinJsonWriter();
    }
}
```

1.1.4. Teste

Tests/AbstractFactoryTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\AbstractFactory\Tests;

use DesignPatterns\Creational\AbstractFactory\CsvWriter;
use DesignPatterns\Creational\AbstractFactory\JsonWriter;
use DesignPatterns\Creational\AbstractFactory\UnixWriterFactory;
use DesignPatterns\Creational\AbstractFactory\WinWriterFactory;
use DesignPatterns\Creational\AbstractFactory\WriterFactory;
use PHPUnit\Framework\TestCase;

class AbstractFactoryTest extends TestCase
{
    public function provideFactory()
    {
        return [
            [new UnixWriterFactory()],
            [new WinWriterFactory()]
        ];
    }

    /**
     * @dataProvider provideFactory
     *
     * @param WriterFactory $writerFactory
     */
    public function testCanCreateCsvWriterOnUnix(WriterFactory $writerFactory)
    {
        $this->assertInstanceOf(JsonWriter::class, $writerFactory-
>createJsonWriter());
```

```
        $this->assertInstanceOf(CsvWriter::class, $writerFactory-
>createCsvWriter());
    }
}
```

1.2. Construtor (Builder)

1.2.1. Objetivo

Contrutor é uma interface que constrói partes de objetos complexos.

As vezes, se o construtor tem melhor conhecimento do que ele cria, essa interface pode ser uma classe abstrata com métodos padrão (como o padrão adaptador).

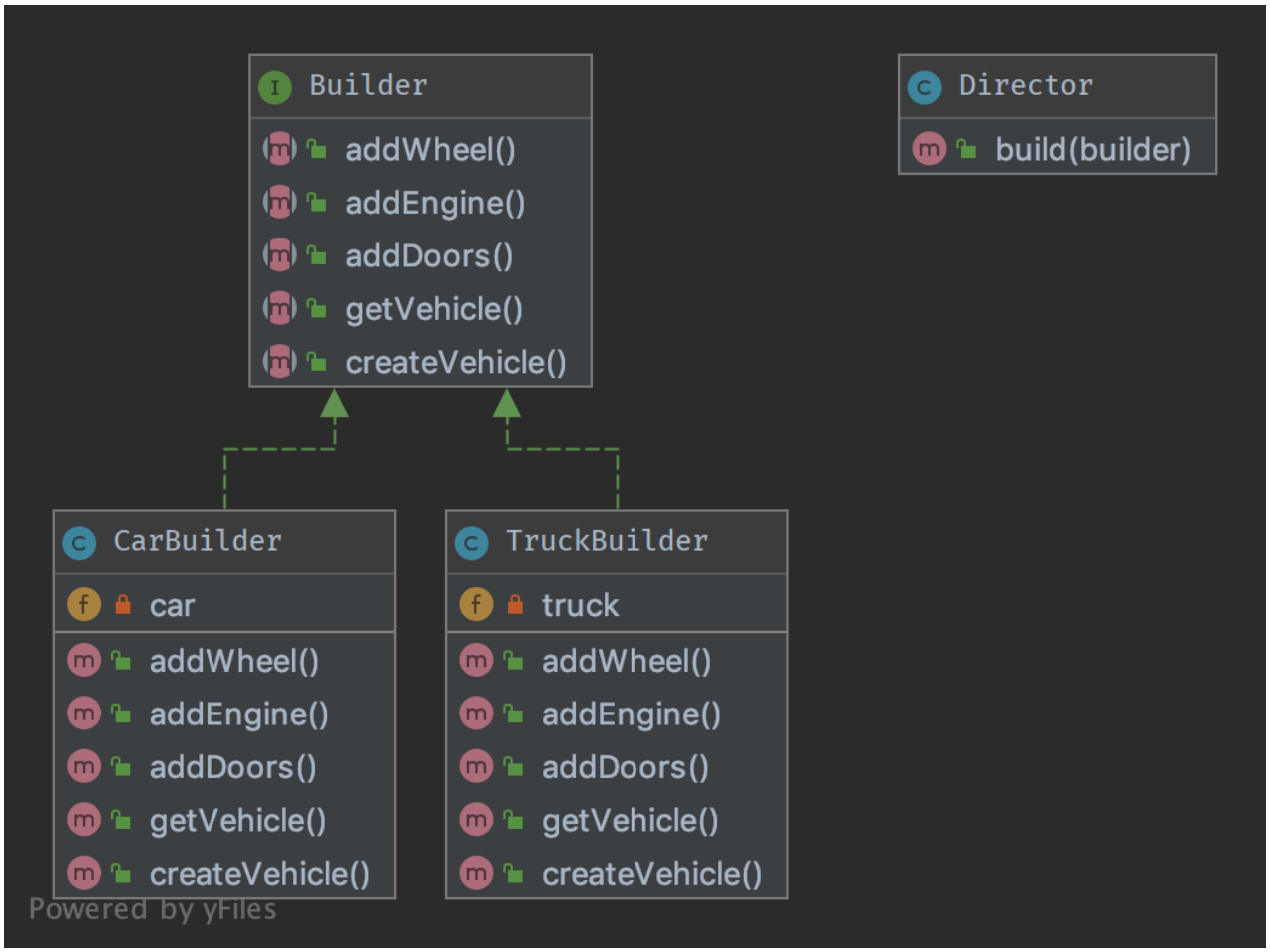
Se você tem uma árvore de herança entre objetos complexa, é lógico que se tenha uma árvore de herança complexa para os construtores também

Nota: Construtores têm frequentemente, uma interface fluente. Veja o construtor mock do PHPUnit, por exemplo.

1.2.2. Exemplos

- PHPUnit: Contrutor Mock

1.2.3. Diagrama UML



1.2.4. Código

Você pode encontrar esse código no Github

Director.php

```
<?php declare(strict_types=1);

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(Builder $builder): Vehicle
    {
        $builder->createVehicle();
        $builder->addDoors();
        $builder->addEngine();
```

```
        $builder->addWheel();

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

Builder.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

interface Builder
{
    public function createVehicle();

    public function addWheel();

    public function addEngine();

    public function addDoors();

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

TruckBuilder.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Door;
use DesignPatterns\Creational\Builder\Parts\Engine;
use DesignPatterns\Creational\Builder\Parts\Wheel;
use DesignPatterns\Creational\Builder\Parts\Truck;
use DesignPatterns\Creational\Builder\Parts\Vehicle;

class TruckBuilder implements Builder
{
    private Truck $truck;

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

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

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

```
    }

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

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

CarBuilder.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Door;
use DesignPatterns\Creational\Builder\Parts\Engine;
use DesignPatterns\Creational\Builder\Parts\Wheel;
use DesignPatterns\Creational\Builder\Parts\Car;
use DesignPatterns\Creational\Builder\Parts\Vehicle;

class CarBuilder implements Builder
{
    private Car $car;

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

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

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

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

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

Parts/Vehicle.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

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

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

Parts/Truck.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Truck extends Vehicle
{
}
```

Parts/Car.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Car extends Vehicle
{
}
```

Parts/Engine.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Engine
{
}
```

Parts/Wheel.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Wheel
{
}
```

Parts/Door.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

1.2.5. Teste

Tests/DirectorTest.php

```php
<?php declare(strict_types=1);

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);
    }
}
```

1.3. Fábrica de Métodos (Factory Method)

1.3.1. Objetivo

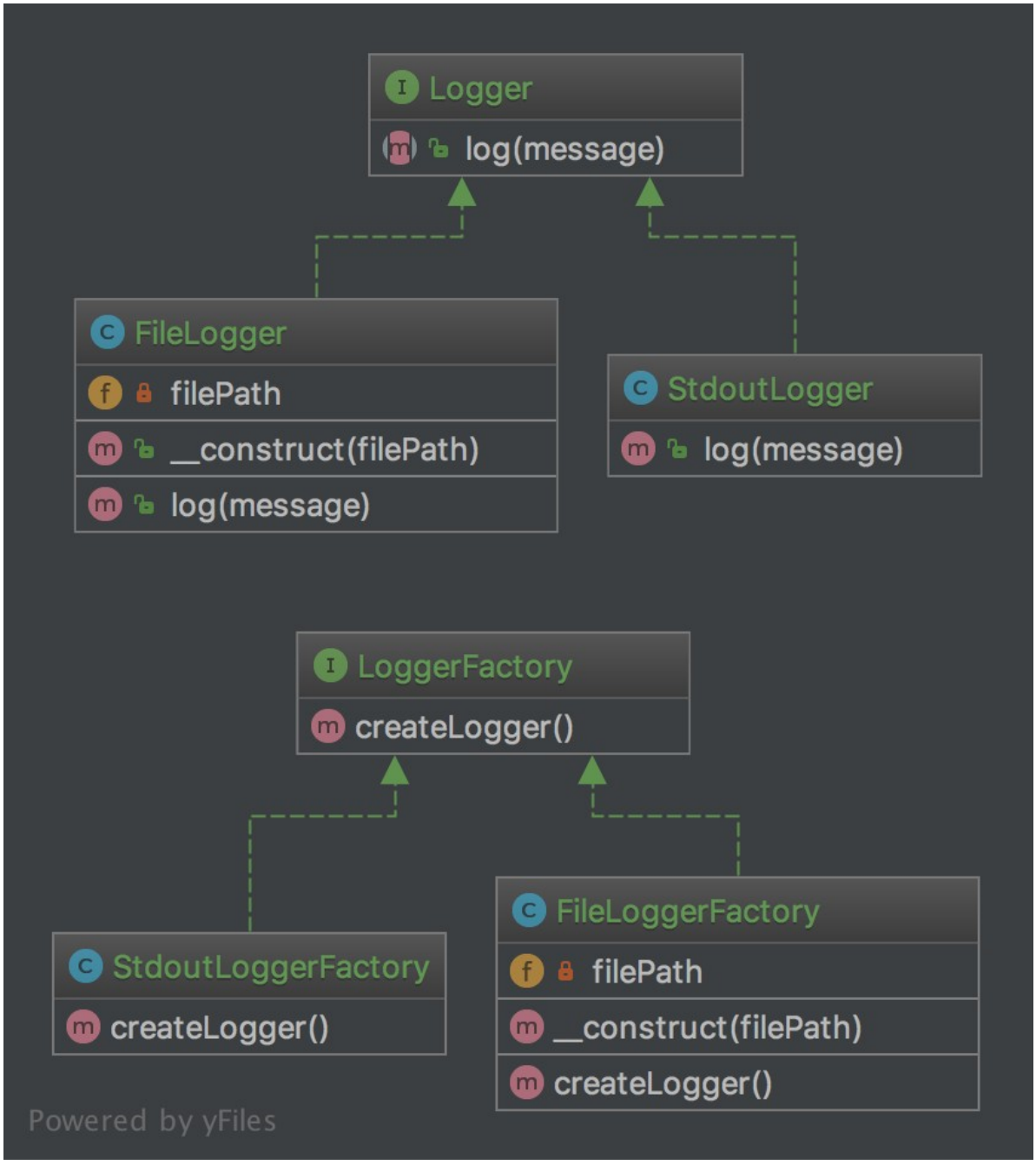
O ponto positivo em relação ao SimpleFactory é que pode-se extender sua implementação de diferentes maneiras para a criação de objetos.

Para casos simples, esta classe abstrata pode ser apenas uma interface.

Este padrão é um padrão de projetos de software “real” já que trata o “Princípio da inversão de dependências” o “D” nos princípios SOLID

Significa que a Fábrica de Método depende de abstrações, não implementação. Este é uma vantagem comparado ao SimpleFactory ou StaticFactory.

1.3.2. Diagrama UML



1.3.3. Código

Você pode encontrar este código no Github

Logger.php

```
<?php declare(strict_types=1);
```

```php
namespace DesignPatterns\Creational\FactoryMethod;

interface Logger
{
    public function log(string $message);
}
```

StdoutLogger.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

class StdoutLogger implements Logger
{
    public function log(string $message)
    {
        echo $message;
    }
}
```

FileLogger.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

class FileLogger implements Logger
{
    private string $filePath;

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

    public function log(string $message)
    {
        file_put_contents($this->filePath, $message . PHP_EOL, FILE_APPEND);
    }
}
```

LoggerFactory.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

interface LoggerFactory
{
    public function createLogger(): Logger;
}
```

StdoutLoggerFactory.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

class StdoutLoggerFactory implements LoggerFactory
```

```
{
    public function createLogger(): Logger
    {
        return new StdoutLogger();
    }
}
```

FileLoggerFactory.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

class FileLoggerFactory implements LoggerFactory
{
    private string $filePath;

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

    public function createLogger(): Logger
    {
        return new FileLogger($this->filePath);
    }
}
```

1.3.4. Teste

Tests/FactoryMethodTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod\Tests;

use DesignPatterns\Creational\FactoryMethod\FileLogger;
use DesignPatterns\Creational\FactoryMethod\FileLoggerFactory;
use DesignPatterns\Creational\FactoryMethod\StdoutLogger;
use DesignPatterns\Creational\FactoryMethod\StdoutLoggerFactory;
use PHPUnit\Framework\TestCase;

class FactoryMethodTest extends TestCase
{
    public function testCanCreateStdoutLogging()
    {
        $loggerFactory = new StdoutLoggerFactory();
        $logger = $loggerFactory->createLogger();

        $this->assertInstanceOf(StdoutLogger::class, $logger);
    }

    public function testCanCreateFileLogging()
    {
        $loggerFactory = new FileLoggerFactory(sys_get_temp_dir());
        $logger = $loggerFactory->createLogger();
```

```
        $this->assertInstanceOf(FileLogger::class, $logger);
    }
}
```

1.4. Agrupamento (Pool)

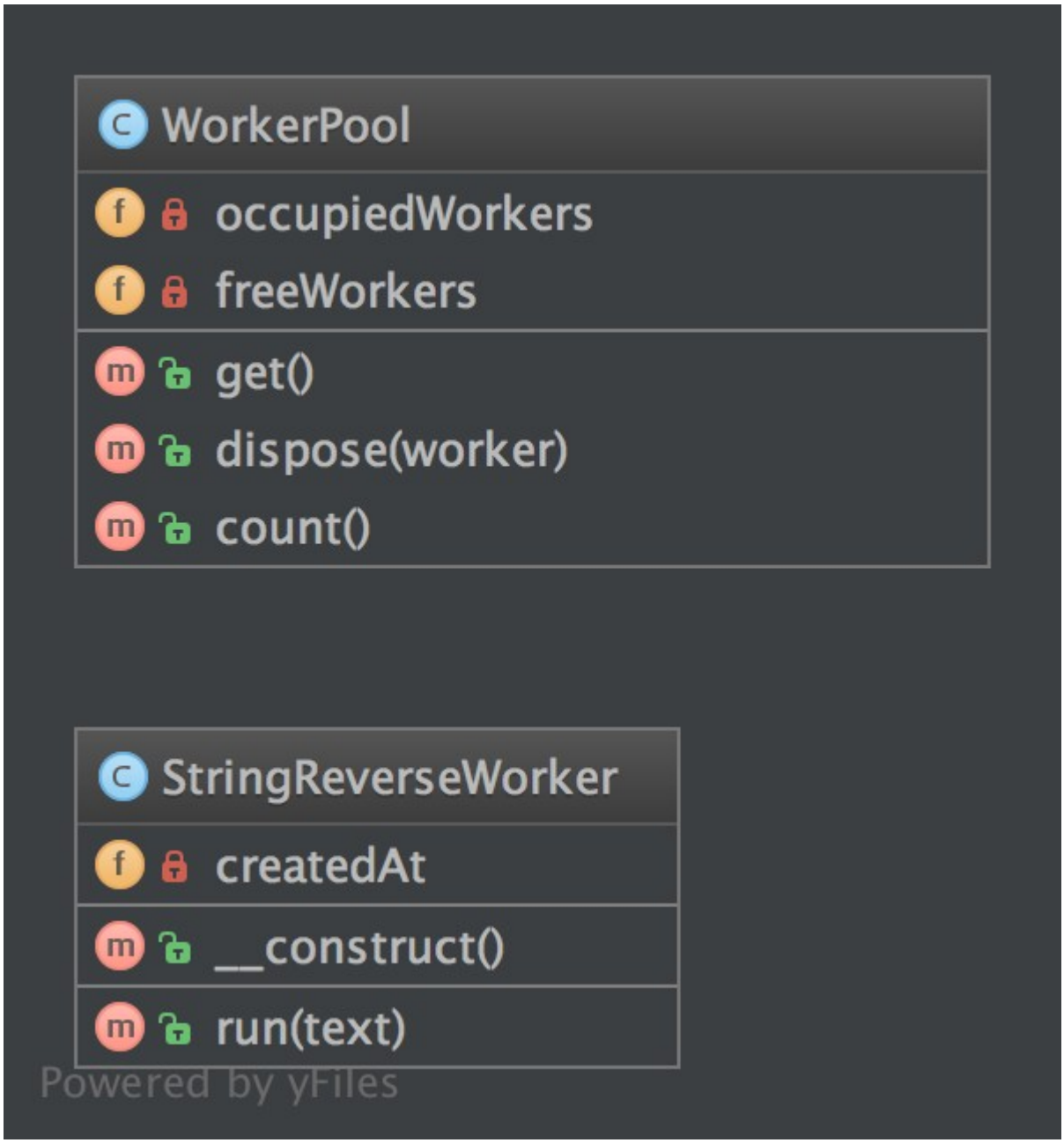
1.4.1. Purpose

O **padrão de conjunto de objetos** é um padrão de design criacional de software que usa um conjunto de objetos inicializados mantidos prontos para usar – um “pool” – em vez de alocá-los e destruí-los sob demanda. Um cliente do agrupamento solicitará um objeto do pool e executará operações no objeto retornado. Quando o cliente termina, retorna o objeto, que é um tipo específico de objeto de fábrica, para o pool, em vez de destruí-lo.

O agrupamento de objetos pode oferecer um aumento significativo de desempenho em situações onde o custo de inicializar uma instância de classe é alto, a taxa de instanciação de uma classe é alta, e o número de instâncias em uso em qualquer momento é baixo. O objeto em pool é obtido em tempo previsível enquanto a criação dos novos objetos (especialmente na rede) pode levar tempo variável.

No entanto, esses benefícios são principalmente verdadeiros para objetos que são caros em relação ao tempo, como conexões de banco de dados, conexões de soquete, encadeamentos e grandes objetos gráficos, como fontes ou bitmaps. Em certas situações, o agrupamento de objetos simples (que não contém recursos externos, mas apenas ocupam memória) pode não ser eficiente e diminuir o desempenho.

1.4.2. Diagrama UML



1.4.3. Código

Você também pode encontrar esse código no GitHub

WorkerPool.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Pool;

use Countable;

class WorkerPool implements Countable
{
```

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

    /**
     * @var StringReverseWorker[]
     */
    private array $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 declare(strict_types=1);

namespace DesignPatterns\Creational\Pool;

use DateTime;

class StringReverseWorker
{
    private DateTime $createdAt;

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

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

1.4.4. Teste

Tests/PoolTest.php

```
<?php declare(strict_types=1);

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);
    }
}
```

1.5. Protótipo

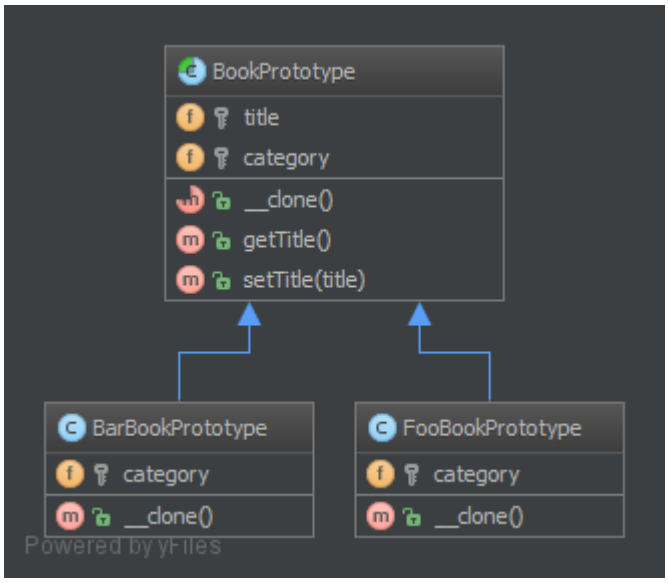
1.5.1. Objetivo

Para evitar o custo da criação tradicional de objetos (new Foo()) e criar um protótipo deste e clonálos

1.5.2. Exemplos

- Uma grande quantidade de dados. Exemplo: a criação de 1,000.000 linhas em um banco de dados via um ORM

1.5.3. Diagrama UML



1.5.4. Código

Você também encontra este código no GitHub

BookPrototype.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Prototype;

abstract class BookPrototype
{
    protected string $title;
    protected string $category;

    abstract public function __clone();

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

    public function setTitle(string $title)
    {
        $this->title = $title;
    }
}
```

BarBookPrototype.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Prototype;

class BarBookPrototype extends BookPrototype
{
```

```
    protected string $category = 'Bar';

    public function __clone()
    {
    }
}
```

FooBookPrototype.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Prototype;

class FooBookPrototype extends BookPrototype
{
    protected string $category = 'Foo';

    public function __clone()
    {
    }
}
```

1.5.5. Teste

Tests/PrototypeTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Prototype\Tests;

use DesignPatterns\Creational\Prototype\BarBookPrototype;
use DesignPatterns\Creational\Prototype\FooBookPrototype;
use PHPUnit\Framework\TestCase;

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);
        }
    }
}
```

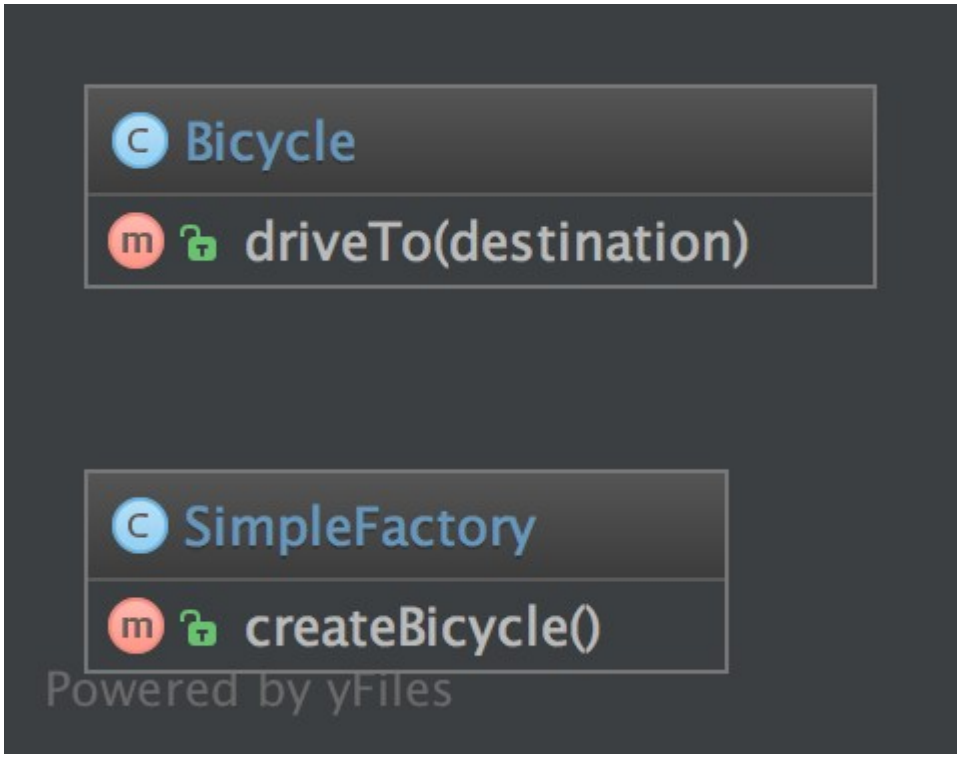
1.6. Fábrica Simples

1.6.1. Objetivo

SimpleFactory (Fábrica Simples) é uma implementação mais simples do padrão fábrica

Diferencia-se do padrão de Fábrica Estática porque não é estático. Assim sendo, você precisa ter múltiplas fábricas, diferentemente parametrizadas, você pode ter subclasses disto e pode simular isto. Este padrão deve ser sempre preferido sobre o padrão de Fábrica Estática!

1.6.2. Diagrama UML



1.6.3. Código

Você também pode encontrar esse código no GitHub

SimpleFactory.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\SimpleFactory;

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

```
}
```

Bicycle.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\SimpleFactory;

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

1.6.4. Uso

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

1.6.5. Teste

Tests/SimpleFactoryTest.php

```
<?php declare(strict_types=1);

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);
    }
}
```

1.7. Singleton

CONSIDERADO UM ANTI-PATTERN! PARA MELHOR TESTABILIDADE E MANUTENIBILIDADE USE INJEÇÃO DEPENDÊNCIAS!

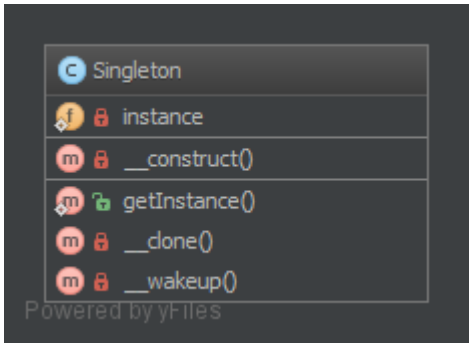
1.7.1. Objetivo

Ter uma única instância da classe na aplicação que será responsável por gerenciar todas as chamadas a ela.

1.7.2. Exemplos

- Conexão ao banco de dados (DB Connector)
- Logger
- Arquivo de lock para a aplicação (existe apenas um em todo o sistema de arquivos …)

1.7.3. Diagrama UML



1.7.4. Código

Você também pode encontrar esse código no GitHub

Singleton.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Singleton;

final class Singleton
{
    private static ?Singleton $instance = null;

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

1.7.5. Teste

Tests/SingletonTest.php

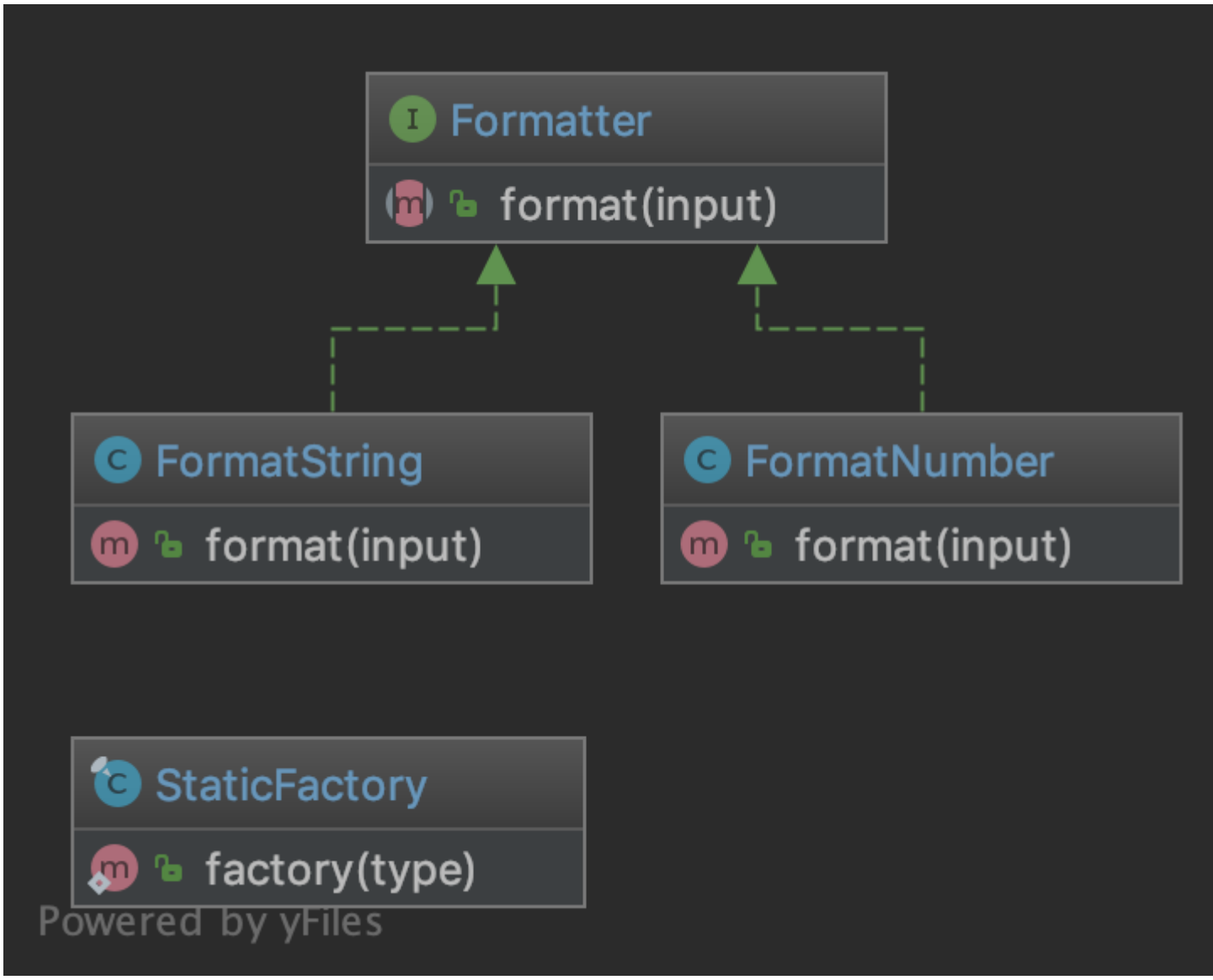
```
<?php declare(strict_types=1);

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);
    }
}
```

1.8. Fábrica Estática

1.8.1. Objetivo

Semelhante à Fábrica Abstrata, este padrão é utilizado para a criação de um conjunto de objetos relacionados ou dependentes. A diferença entre este padrão e o Fábrica Abstrata é que o Fábrica Estática usa apenas um método estático para criar todos os tipos de objetos que ele pode criar. Ele é chamado usualmente de `factory` ou `build`.

1.8.2. Diagrama UML



1.8.3. Código

Você também pode encontrar este código no Github

StaticFactory.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\StaticFactory;

use InvalidArgumentException;

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

        throw new InvalidArgumentException('Unknown format given');
    }
}
```

Formatter.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\StaticFactory;

interface Formatter
{
    public function format(string $input): string;
}
```

FormatString.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\StaticFactory;

class FormatString implements Formatter
{
    public function format(string $input): string
    {
        return $input;
    }
}
```

FormatNumber.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\StaticFactory;

class FormatNumber implements Formatter
{
    public function format(string $input): string
```

```
    {
        return number_format((int) $input);
    }
}
```

1.8.4. Teste

Tests/StaticFactoryTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\StaticFactory\Tests;

use InvalidArgumentException;
use DesignPatterns\Creational\StaticFactory\FormatNumber;
use DesignPatterns\Creational\StaticFactory\FormatString;
use DesignPatterns\Creational\StaticFactory\StaticFactory;
use PHPUnit\Framework\TestCase;

class StaticFactoryTest extends TestCase
{
    public function testCanCreateNumberFormatter()
    {
        $this->assertInstanceOf(FormatNumber::class,
StaticFactory::factory('number'));
    }

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

    public function testException()
    {
        $this->expectException(InvalidArgumentException::class);

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

2. Estrutural

Em Engenharia de Software, Padrões de Design Estrutural são Padrões de Design (Design Patterns) que facilitam o design, identificando uma forma simples de perceber o relacionamentos entre entidades.

2.1. Adaptador (Adapter / Wrapper)

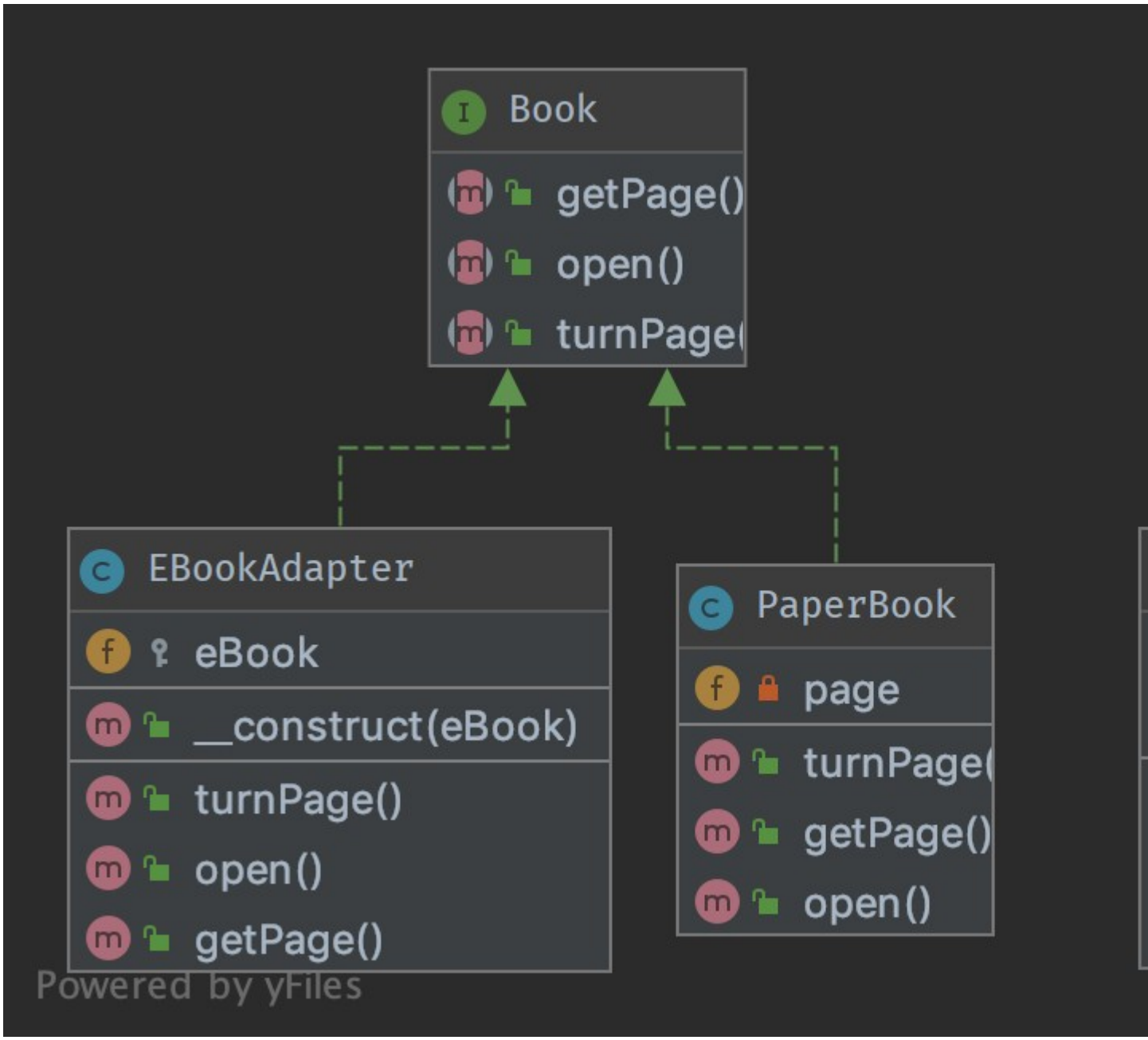
2.1.1. Objetivo

Adaptar (compatibilizar) uma interface de uma classe a outra interface. Um adapter permite que classes trabalhem juntas quando, normalmente isso não seria possível devido imcompatibilidade, provendo sua interface para seus clientes enquanto usa ainterface original

2.1.2. Exemplos

- Adaptador de bibliotecas clientes de banco de dados
- usando diversos webservices e adapters para normalizar os dados de modo que a saída seja sempre a mesma

2.1.3. Diagrama UML



2.1.4. Código

You can also find this code on GitHub

Book.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

interface Book
{
    public function turnPage();
```

```
    public function open();

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

PaperBook.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

class PaperBook implements Book
{
    private int $page;

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

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

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

EBook.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

interface EBook
{
    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;
}
```

EBookAdapter.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

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

```
class EBookAdapter implements Book
{
    protected EBook $eBook;

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

Kindle.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

/**
 * this is the adapted class. In production code, this could be a class from another packa
 * Notice that it uses another naming scheme and the implementation does something similar
 */
class Kindle implements EBook
{
    private int $page = 1;
    private int $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];
    }
}
```

2.1.5. Teste

Tests/AdapterTest.php

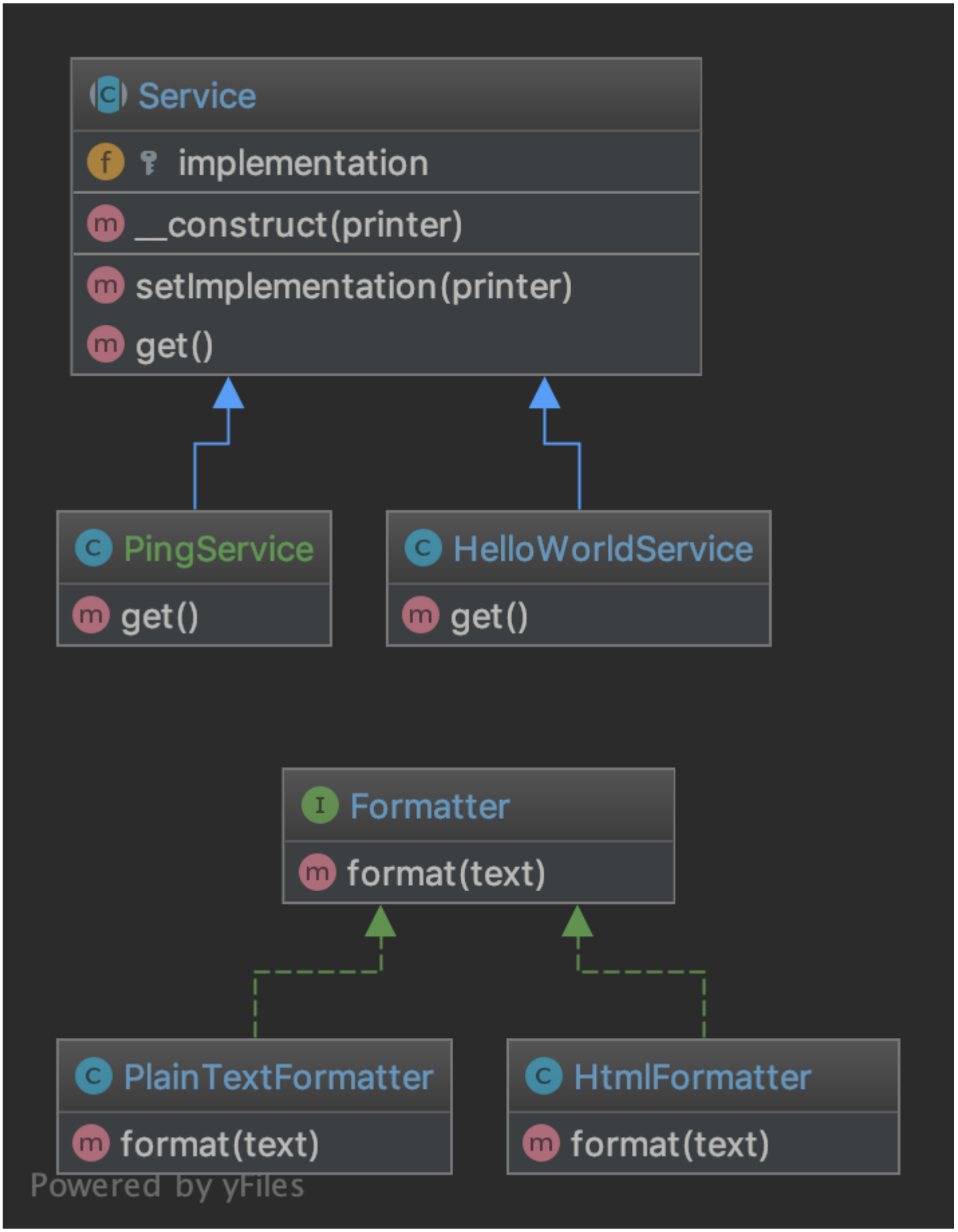
```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter\Tests;

use DesignPatterns\Structural\Adapter\PaperBook;
use DesignPatterns\Structural\Adapter\EBookAdapter;
use DesignPatterns\Structural\Adapter\Kindle;
use PHPUnit\Framework\TestCase;

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

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

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

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

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

2.2. Ponte (Bridge)

2.2.1. Objetivo

Desacoplar uma abstração da sua implementação de modo que as duas possam variar independentemente.

2.2.2. Diagrama UML



2.2.3. Código

Você também pode encontrar este código no GitHub

Formatter.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

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

PlainTextFormatter.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

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

HtmlFormatter.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

class HtmlFormatter implements Formatter
{
    public function format(string $text): string
    {
        return sprintf('<p>%s</p>', $text);
    }
}
```

Service.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

abstract class Service
{
    protected Formatter $implementation;

    public function __construct(Formatter $printer)
    {
        $this->implementation = $printer;
    }
```

```
    public function setImplementation(Formatter $printer)
    {
        $this->implementation = $printer;
    }

    abstract public function get(): string;
}
```

HelloWorldService.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

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

PingService.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

class PingService extends Service
{
    public function get(): string
    {
        return $this->implementation->format('pong');
    }
}
```

2.2.4. Teste

Tests/BridgeTest.php

```
<?php declare(strict_types=1);

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 testCanPrintUsingThePlainTextFormatter()
    {
        $service = new HelloWorldService(new PlainTextFormatter());
```

```
        $this->assertSame('Hello World', $service->get());
    }

    public function testCanPrintUsingTheHtmlFormatter()
    {
        $service = new HelloWorldService(new HtmlFormatter());

        $this->assertSame('<p>Hello World</p>', $service->get());
    }
}
```

2.3. Composto (Composite)

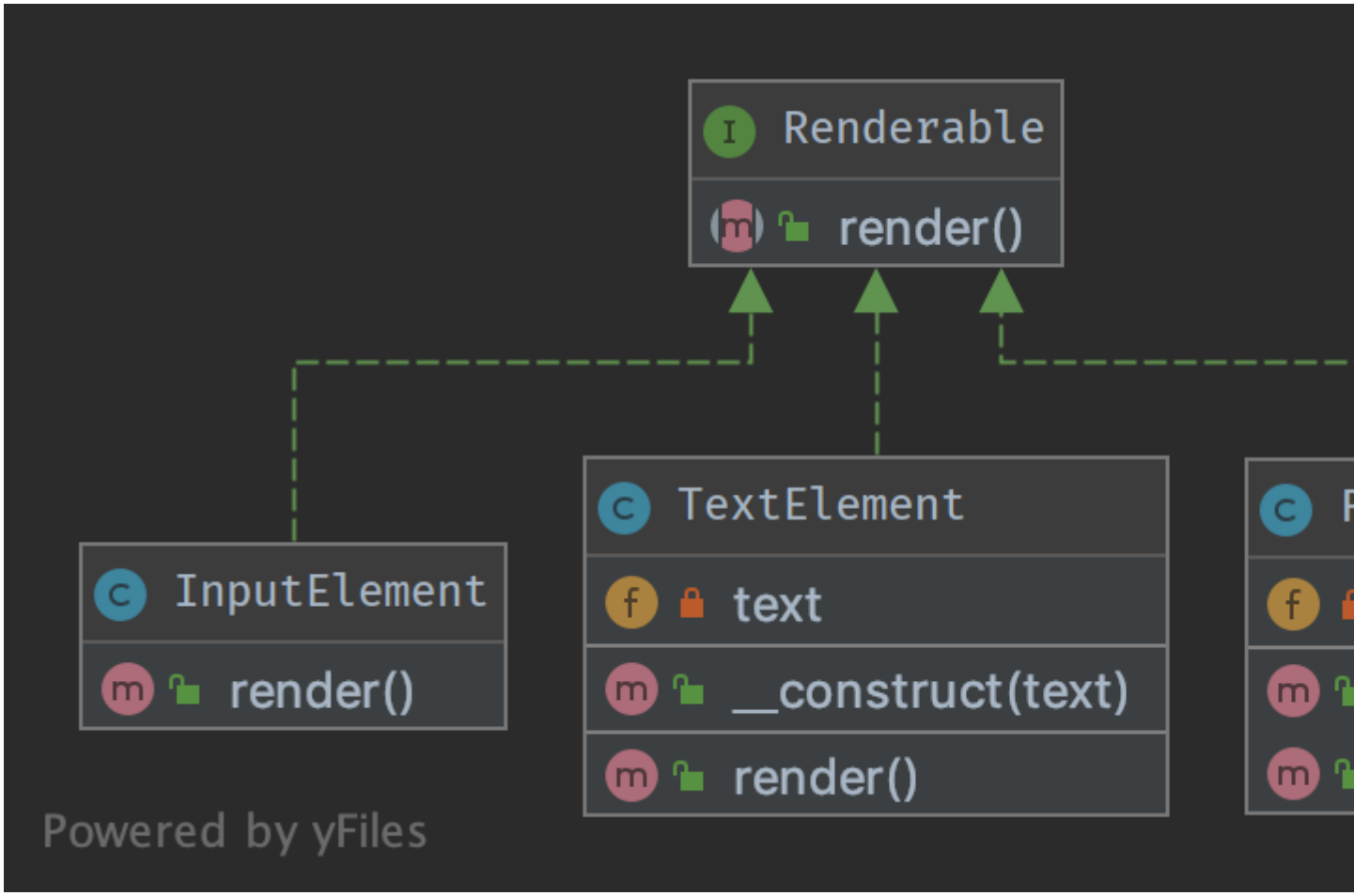
2.3.1. Objetivo

Tratar um grupo de objetos da mesma forma como uma única instância do objeto.

2.3.2. Exemplos

- uma instância de classe de formulário manipula todas os seus elementos de formulário como uma simples instância do formulário, quando `render()` é chamado, ele subsequentemente roda através de seus elementos filhos e chama `render()` neles.

2.3.3. Diagrama UML



2.3.4. Código

Você também pode encontrar este código no GitHub

Renderable.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

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

Form.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

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

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

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

        $formCode .= '</form>';

        return $formCode;
    }

    public function addElement(Renderable $element)
    {
        $this->elements[] = $element;
    }
}
```

InputElement.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

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

TextElement.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

class TextElement implements Renderable
{
    private string $text;

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

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

2.3.5. Teste

Tests/CompositeTest.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Composite\Tests;

use DesignPatterns\Structural\Composite\Form;
use DesignPatterns\Structural\Composite\TextElement;
use DesignPatterns\Structural\Composite\InputElement;
use PHPUnit\Framework\TestCase;

class CompositeTest extends TestCase
{
    public function testRender()
    {
        $form = new Form();
        $form->addElement(new TextElement('Email:'));
        $form->addElement(new InputElement());
        $embed = new Form();
        $embed->addElement(new TextElement('Password:'));
        $embed->addElement(new 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->assertSame(
            '<form>Email:<input type="text" /><form>Password:<input
type="text" /></form></form>',
            $form->render()
        );
    }
}
```

2.4. Mapeador de dados (Data Mapper)

2.4.1. Objetivo

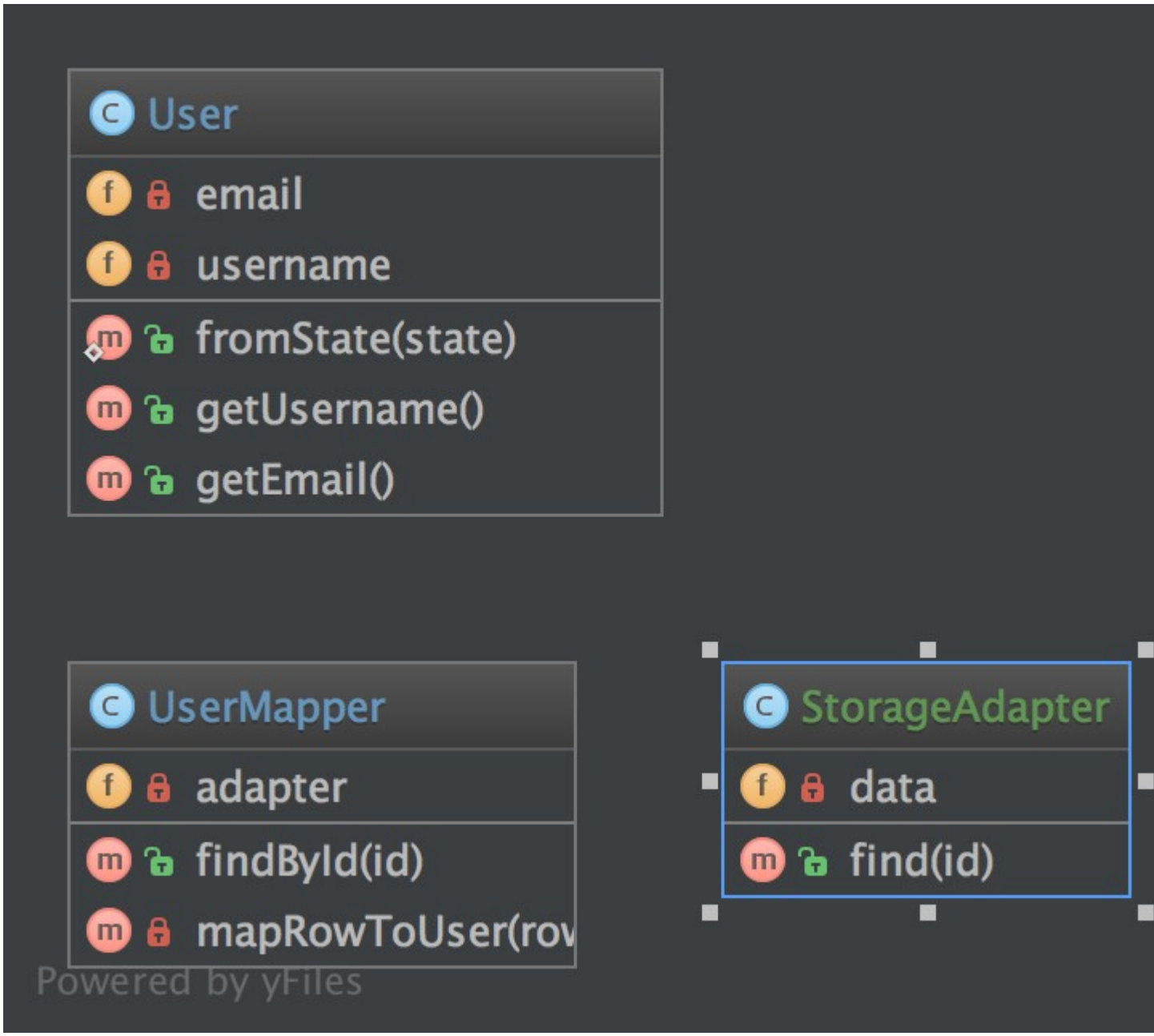
Um mapeador de dados é uma camada de acesso à dados que realiza transferências bidirecionais de dados entre um armazenamento de dados persistente (frequentemente um banco de dados relacional) e uma representação em memória dos dados (a camada de domínio). O objetivo do padrão é manter a representação em memória e o armazenamento de dados persistente independente um do outro e do próprio mapeador de dados. A camada é composta de um ou mais mapeadores (ou Objetos de Acesso à Dados - Data Access Objects), realizando a tranferência dos dados. Implementações de mapeadores variam em escopo. Mapeadores genéricos irão manipular muitos tipos de entidades de domínio diferentes e mapeadores dedicados irão manipular um ou alguns.

O ponto-chave deste padrão é que, diferente do padrão de registro ativo (Active Record pattern), os modelos de dados seguem o princípio de responsabilidade simples (Single Responsibility Principle).

2.4.2. Exemplos

- Mapeamento objeto-relacional do bando de dados (ORM - Object Relational Mapper) : Doctrine2 usa um objeto de acesso a dados (DAO) nomeado como “EntityRepository”

2.4.3. Diagrama UML



2.4.4. Código

Você também pode encontrar este código no GitHub

User.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper;

class User
{
    private string $username;
    private string $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;
    }

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

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

UserMapper.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper;

use InvalidArgumentException;

class UserMapper
{
    private StorageAdapter $adapter;

    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 pat
     * However the important part is in mapRowToUser() below, that will create a business
     * data fetched from storage
     */
    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 declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
{
    private array $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;
    }
}
```

2.4.5. Teste

Tests/DataMapperTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper\Tests;

use InvalidArgumentException;
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);
    }

    public function testWillNotMapInvalidData()
    {
        $this->expectException(InvalidArgumentException::class);

        $storage = new StorageAdapter([]);
        $mapper = new UserMapper($storage);

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

2.5. Decorador (Decorator)

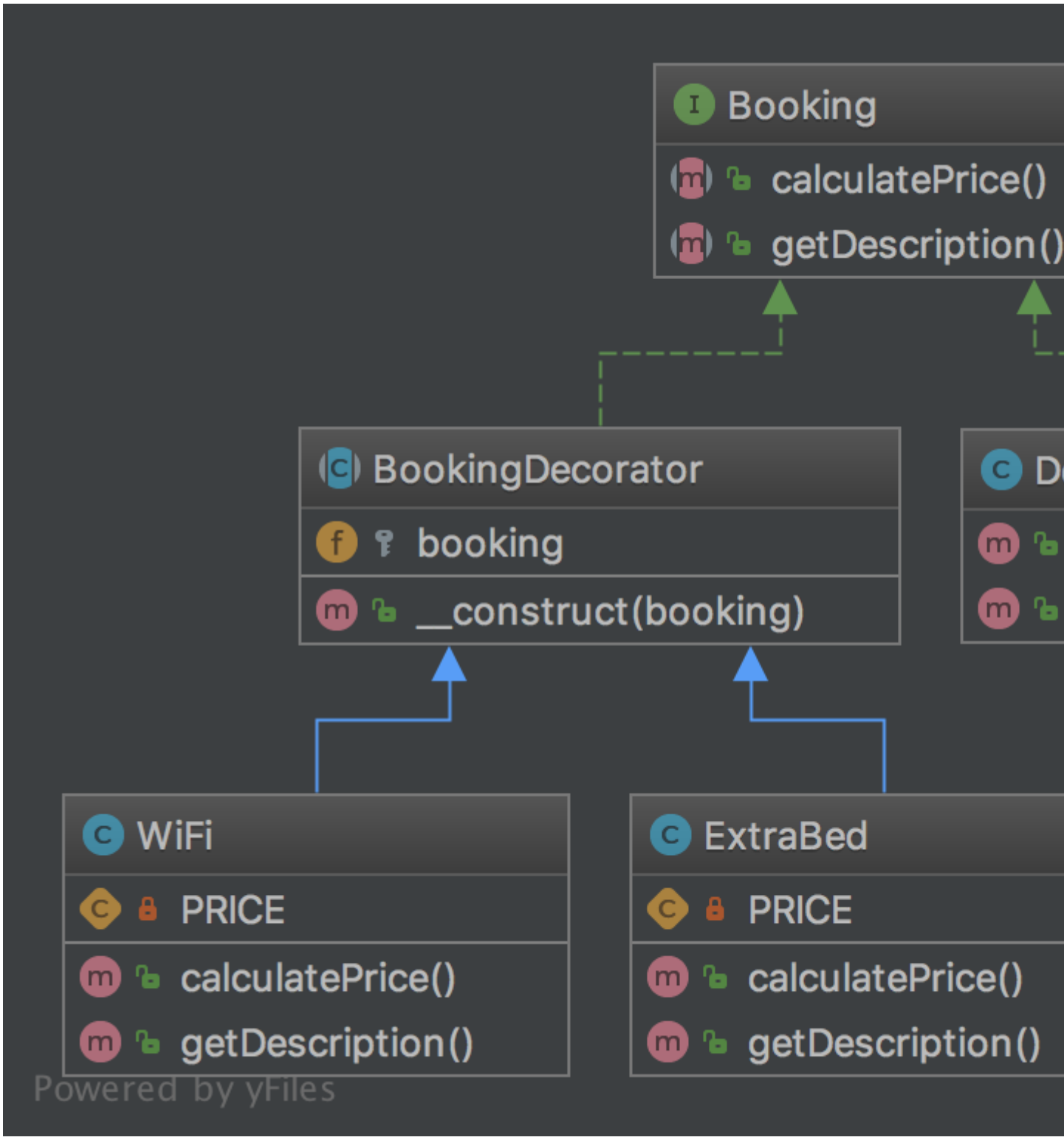
2.5.1. Objetivo

Adicionar dinamicamente novas funcionalidades para as instâncias de uma classe.

2.5.2. Exemplos

- Camada de serviço da web: Decoradores JSON e XML para um serviço REST (neste caso, apenas um destes deve estar permitido.

2.5.3. Diagrama UML



2.5.4. Código

Você também pode encontrar este código no GitHub

Booking.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator;

interface Booking
{
    public function calculatePrice(): int;

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

BookingDecorator.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator;

abstract class BookingDecorator implements Booking
{
    protected Booking $booking;

    public function __construct(Booking $booking)
    {
        $this->booking = $booking;
    }
}
```

DoubleRoomBooking.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator;

class DoubleRoomBooking implements Booking
{
    public function calculatePrice(): int
    {
        return 40;
    }

    public function getDescription(): string
    {
        return 'double room';
    }
}
```

ExtraBed.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator;

class ExtraBed extends BookingDecorator
{
    private const PRICE = 30;

    public function calculatePrice(): int
    {
        return $this->booking->calculatePrice() + self::PRICE;
    }
```

```
    public function getDescription(): string
    {
        return $this->booking->getDescription() . ' with extra bed';
    }
}
```

WiFi.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator;

class WiFi extends BookingDecorator
{
    private const PRICE = 2;

    public function calculatePrice(): int
    {
        return $this->booking->calculatePrice() + self::PRICE;
    }

    public function getDescription(): string
    {
        return $this->booking->getDescription() . ' with wifi';
    }
}
```

2.5.5. Teste

Tests/DecoratorTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Decorator\Tests;

use DesignPatterns\Structural\Decorator\DoubleRoomBooking;
use DesignPatterns\Structural\Decorator\ExtraBed;
use DesignPatterns\Structural\Decorator\WiFi;
use PHPUnit\Framework\TestCase;

class DecoratorTest extends TestCase
{
    public function testCanCalculatePriceForBasicDoubleRoomBooking()
    {
        $booking = new DoubleRoomBooking();

        $this->assertSame(40, $booking->calculatePrice());
        $this->assertSame('double room', $booking->getDescription());
    }

    public function testCanCalculatePriceForDoubleRoomBookingWithWiFi()
    {
        $booking = new DoubleRoomBooking();
        $booking = new WiFi($booking);

        $this->assertSame(42, $booking->calculatePrice());
```

```
        $this->assertSame('double room with wifi', $booking->getDescription());
    }

    public function
testCanCalculatePriceForDoubleRoomBookingWithWiFiAndExtraBed()
    {
        $booking = new DoubleRoomBooking();
        $booking = new WiFi($booking);
        $booking = new ExtraBed($booking);

        $this->assertSame(72, $booking->calculatePrice());
        $this->assertSame('double room with wifi with extra bed', $booking-
>getDescription());
    }
}
```

2.6. Injeção de dependência (Dependency Injection)

2.6.1. Objetivo

Implementar uma arquitetura menos acoplada com o objetivo de obter um código mais fácil de testar, de melhor manutenibilidade e mais extensível.

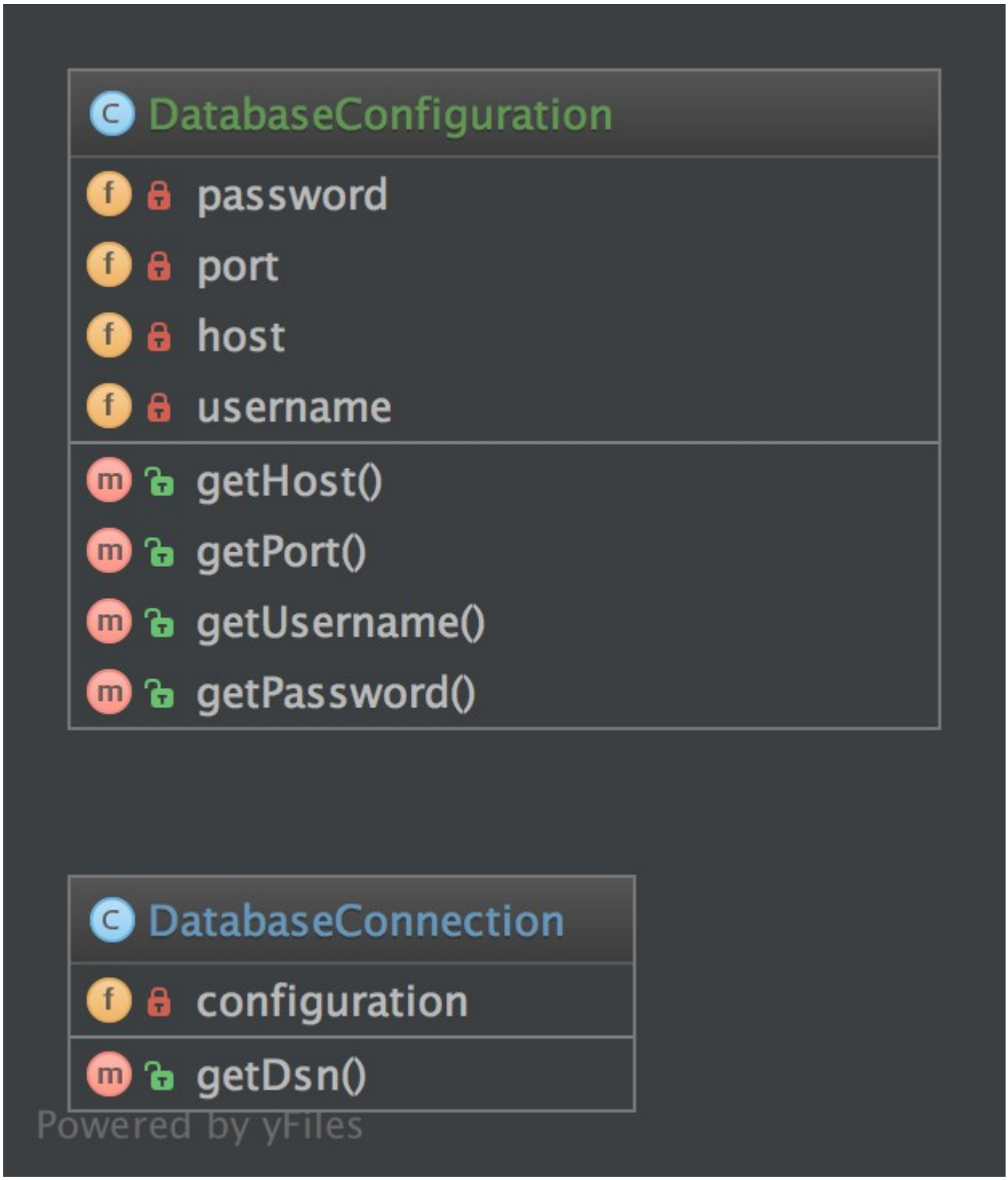
2.6.2. Uso

`DatabaseConfiguration` é injetado e `DatabaseConnection` irá receber tudo o que é necessário de `$config`. Sem a Injeção de Dependência, a configuração seria criada diretamente em `DatabaseConnection`, o que não é muito bom para testar e extender isto.

2.6.3. Exemplos

- O ORM Doctrine 2 usa injeção de dependência, por exemplo, para a configuração que é injetada no objeto `Connection`. Para propósitos de teste, algúem pode facilmente criar um objeto simulado na configuração e injetá-lo no objeto `Connection`.
- many frameworks already have containers for DI that create objects via a configuration array and inject them where needed (i.e. in Controllers)

2.6.4. Diagrama UML



2.6.5. Código

Você pode também ver este código no GitHub

DatabaseConfiguration.php

```
<?php declare(strict_types=1);
```

```
namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConfiguration
{
    private string $host;
    private int $port;
    private string $username;
    private string $password;

    public function __construct(string $host, int $port, string $username, string $passwor
    {
        $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 declare(strict_types=1);

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConnection
{
    private DatabaseConfiguration $configuration;

    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()
        );
    }
}
```

2.6.6. Teste

Tests/DependencyInjectionTest.php

```
<?php declare(strict_types=1);

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

2.7. Facade (Fachada)

2.7.1. Objetivo

The primary goal of a Facade Pattern is not to avoid you having 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.

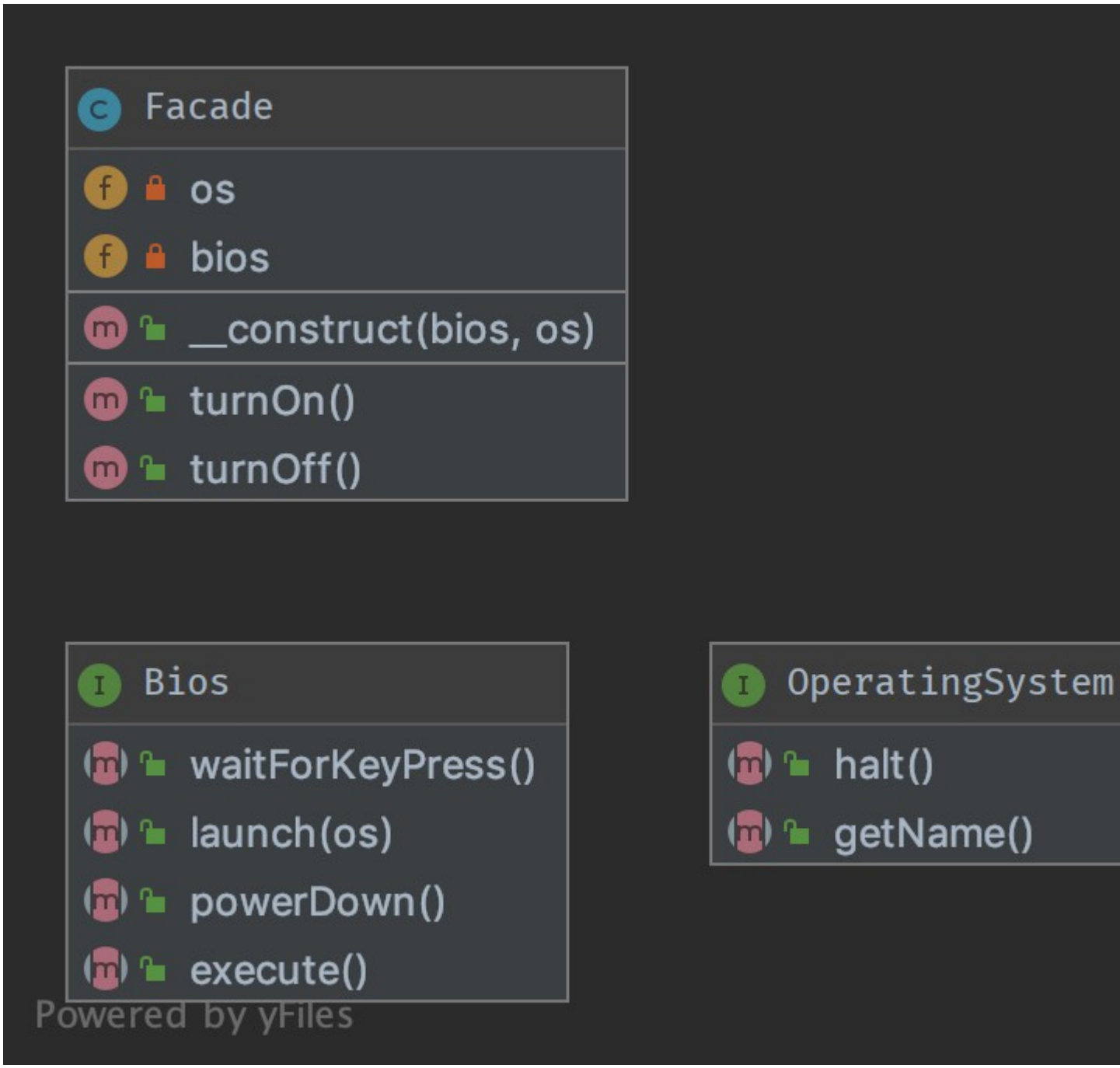
Uma Facade destina-se a dissociar um cliente e um subsistema, incorporando muitas (porém as vezes apenas uma) interface e, é claro, para reduzir a complexidade.

- Uma Facade não proibe você de acessar o sub-sistema
- Você pode (você deve) ter múltiplas facades para um sub-sistema

É por isso que uma boa facade não tem nenhum `new` nela. Se há multiplas criações para cada método, ela não é uma facade, ela é uma Builder (Construtora) ou uma [Abstract|Static|Simple] Factory (Fábrica) [Method].

A melhor facade não tem nenhum `new` e um construtor com parâmetros interface-type-hinted. Se você precisa da criação de novas instâncias, use uma Factory (Fábrica) como argumento.

2.7.2. Diagrama UML



2.7.3. Código

Você também pode encontrar este código no GitHub

Facade.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Facade;

class Facade
```

```
{
    private OperatingSystem $os;
    private Bios $bios;

    public function __construct(Bios $bios, OperatingSystem $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();
    }
}
```

OperatingSystem.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Facade;

interface OperatingSystem
{
    public function halt();

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

Bios.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Facade;

interface Bios
{
    public function execute();

    public function waitForKeyPress();

    public function launch(OperatingSystem $os);

    public function powerDown();
}
```

2.7.4. Teste

Tests/FacadeTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Facade\Tests;

use DesignPatterns\Structural\Facade\Bios;
use DesignPatterns\Structural\Facade\Facade;
use DesignPatterns\Structural\Facade\OperatingSystem;
use PHPUnit\Framework\TestCase;

class FacadeTest extends TestCase
{
    public function testComputerOn()
    {
        $os = $this->createMock(OperatingSystem::class);

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

        $bios = $this->createMock(Bios::class);

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

        /** @noinspection PhpParamsInspection */
        $facade = new Facade($bios, $os);
        $facade->turnOn();

        $this->assertSame('Linux', $os->getName());
    }
}
```

2.8. Interface Fluente (Fluent Interface)

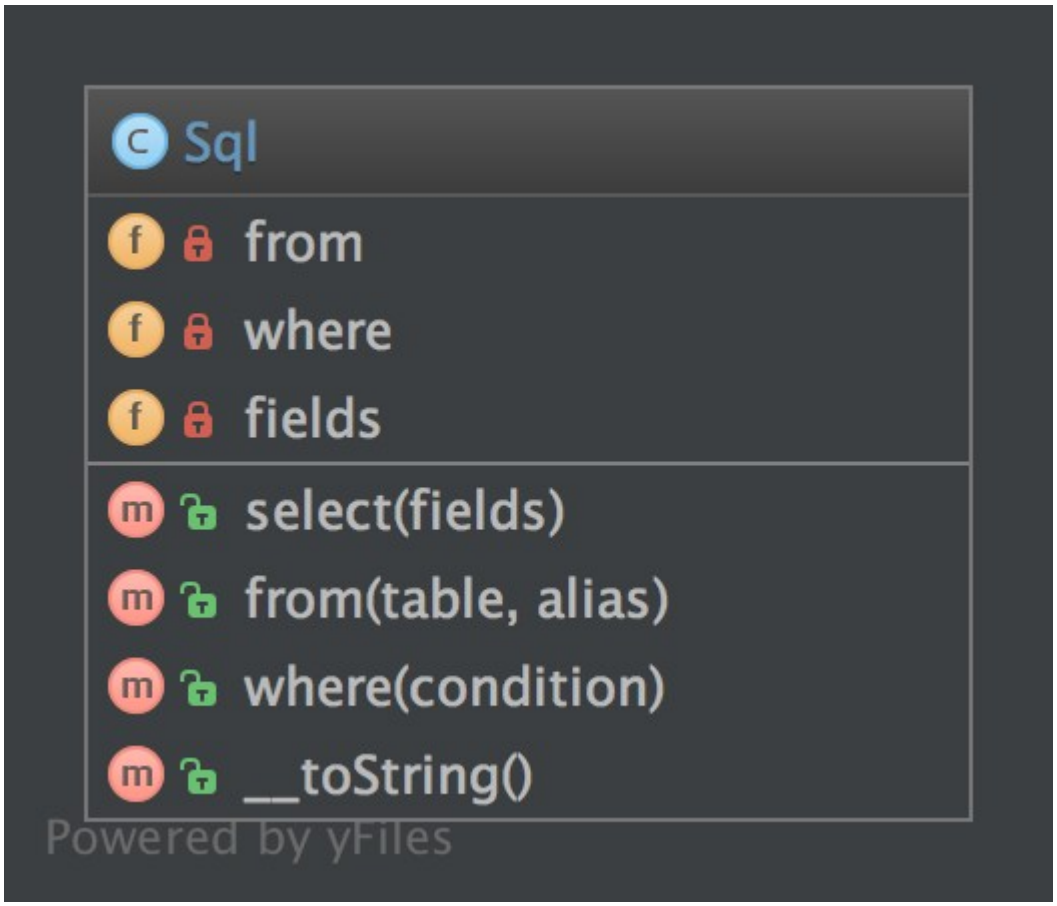
2.8.1. Objetivo

Escrever código que é facilmente legível como sentenças em uma linguagem natural (como o Português).

2.8.2. Exemplos

- O QueryBuilder do Doctrine2 funciona de alguma forma parecida com o exemplo abaixo
- PHPUnit usa interfaces fluentes para construir objetos simulados (mock objects)

2.8.3. Diagrama UML



2.8.4. Código

Você também pode encontrar este código no GitHub

Sql.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\FluentInterface;

class Sql
{
    private array $fields = [];
    private array $from = [];
    private array $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)
        );
    }
}
```

2.8.5. Teste

Tests/FluentInterfaceTest.php

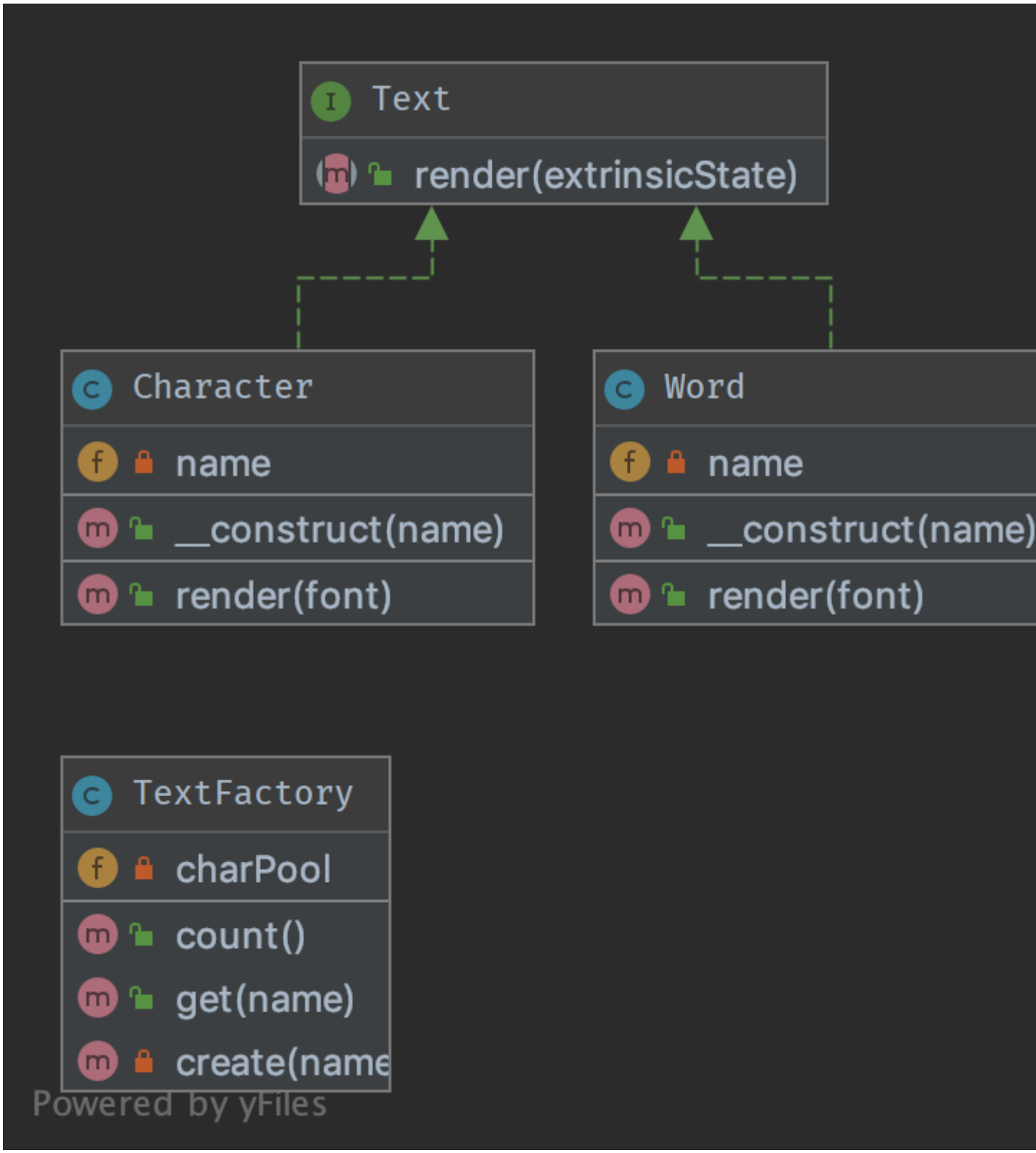
```
<?php declare(strict_types=1);

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

2.9. Flyweight (Mosca)

2.9.1. Objetivo

Minimizar o uso de memória, um Flyweight compartilha memória o quanto for possível com objetos similares. Ele é necessário quando um grande número de objetos são utilizados de forma que não diferem muito em estado. Uma prática comum é manter o estado nas estruturas de dados externos e passá-los para o objeto Flyweight quando necessário.

2.9.2. Diagrama UML



2.9.3. Código

Você também pode encontrar este código no GitHub

Text.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Flyweight;

/**
 * This is the interface that all flyweights need to implement
 */
interface Text
{
    public function render(string $extrinsicState): string;
}
```

Word.php

```
<?php

namespace DesignPatterns\Structural\Flyweight;

class Word implements Text
{
    private string $name;

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

    public function render(string $font): string
    {
        return sprintf('Word %s with font %s', $this->name, $font);
    }
}
```

Character.php

```
<?php declare(strict_types=1);

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 Character implements Text
{
    /**
     * Any state stored by the concrete flyweight must be independent of its context.
     * For flyweights representing characters, this is usually the corresponding character
     */
    private string $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 d
         // For flyweights representing characters, extrinsic state usually contains e.g.

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

TextFactory.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Flyweight;

use Countable;

/**
 * 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 TextFactory implements Countable
{
    /**
     * @var Text[]
     */
    private array $charPool = [];

    public function get(string $name): Text
    {
        if (!isset($this->charPool[$name])) {
            $this->charPool[$name] = $this->create($name);
        }

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

    private function create(string $name): Text
    {
        if (strlen($name) == 1) {
            return new Character($name);
        } else {
            return new Word($name);
        }
    }

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

2.9.4. Teste

Tests/FlyweightTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Flyweight\Tests;

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

class FlyweightTest extends TestCase
{
    private array $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 array $fonts = ['Arial', 'Times New Roman', 'Verdana', 'Helvetica'];

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

        for ($i = 0; $i <= 10; $i++) {
            foreach ($this->characters as $char) {
                foreach ($this->fonts as $font) {
                    $flyweight = $factory->get($char);
                    $rendered = $flyweight->render($font);

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

        foreach ($this->fonts as $word) {
            $flyweight = $factory->get($word);
            $rendered = $flyweight->render('foobar');

            $this->assertSame(sprintf('Word %s with font foobar', $word),
$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) + count($this->fonts),
$factory);
    }
}
```

2.10. Proxy

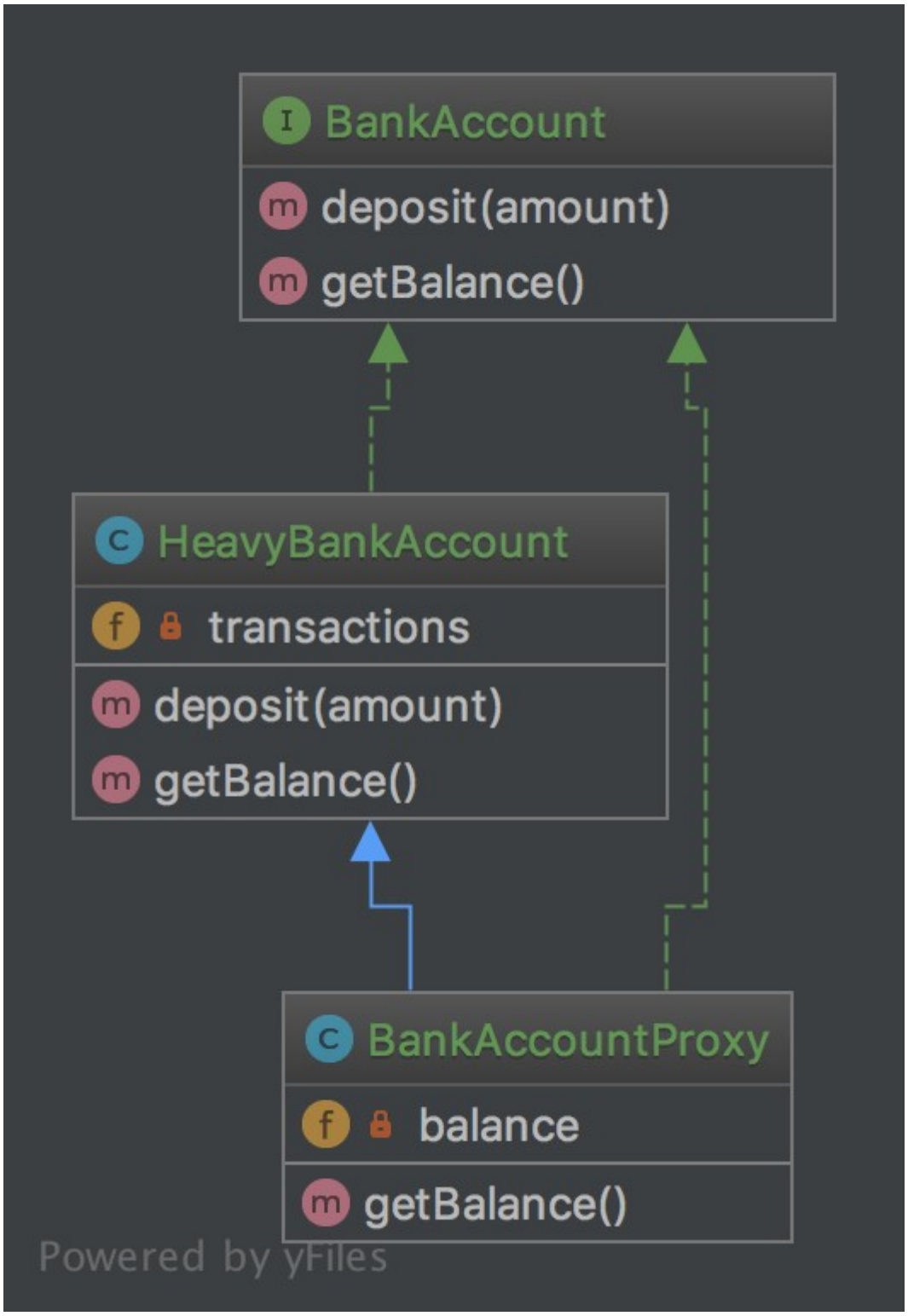
2.10.1. Objetivo

Servir de interface para qualquer coisa que é custosa ou impossível de duplicar.

2.10.2. Exemplos

- Doctrine2 usa proxies para implementar nele a mágica do framework (p.e. lazy initialization), enquanto o usuário ainda trabalha com suas próprias classes de entidade e nunca usará ou tocará os proxies

2.10.3. Diagrama UML



2.10.4. Código

Você também pode encontrar este código no GitHub

BankAccount.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Proxy;

interface BankAccount
{
    public function deposit(int $amount);

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

HeavyBankAccount.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Proxy;

class HeavyBankAccount implements BankAccount
{
    /**
     * @var int[]
     */
    private array $transactions = [];

    public function deposit(int $amount)
    {
        $this->transactions[] = $amount;
    }

    public function getBalance(): int
    {
        // this is the heavy part, imagine all the transactions even from
        // years and decades ago must be fetched from a database or web service
        // and the balance must be calculated from it

        return (int) array_sum($this->transactions);
    }
}
```

BankAccountProxy.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Proxy;

class BankAccountProxy extends HeavyBankAccount implements BankAccount
{
    private ?int $balance = null;

    public function getBalance(): int
    {
        // because calculating balance is so expensive,
        // the usage of BankAccount::getBalance() is delayed until it really is needed
        // and will not be calculated again for this instance

        if ($this->balance === null) {
            $this->balance = parent::getBalance();
        }

        return $this->balance;
    }
```

```
}
```

2.10.5. Teste

ProxyTest.php

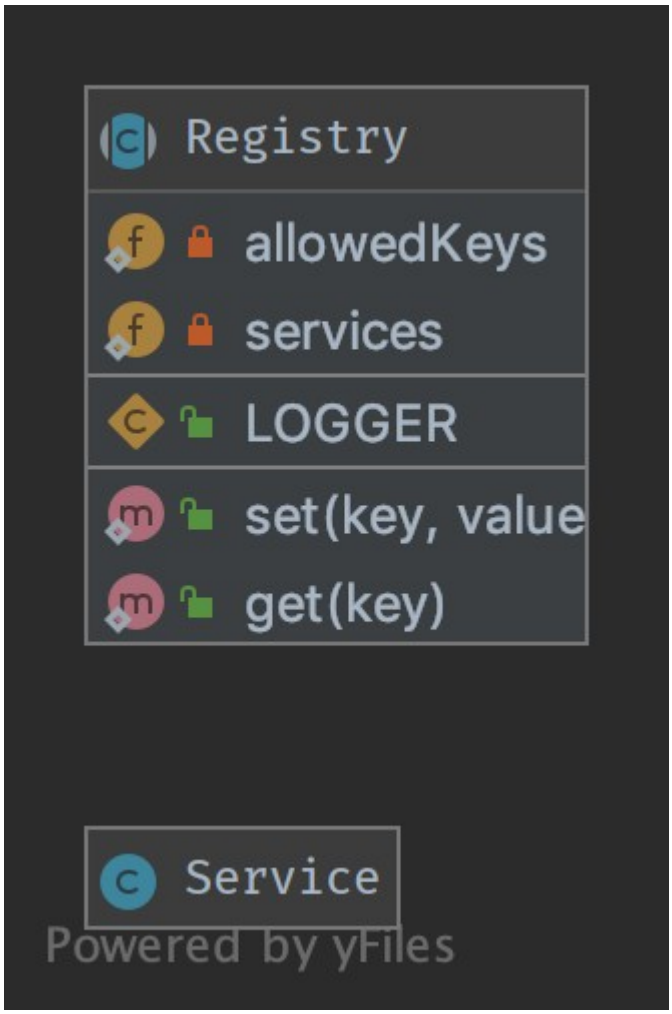
```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Proxy\Tests;

use DesignPatterns\Structural\Proxy\BankAccountProxy;
use PHPUnit\Framework\TestCase;

class ProxyTest extends TestCase
{
    public function testProxyWillOnlyExecuteExpensiveGetBalanceOnce()
    {
        $bankAccount = new BankAccountProxy();
        $bankAccount->deposit(30);

        // this time balance is being calculated
        $this->assertSame(30, $bankAccount->getBalance());

        // inheritance allows for BankAccountProxy to behave to an outsider
exactly like ServerBankAccount
        $bankAccount->deposit(50);

        // this time the previously calculated balance is returned again without
re-calculating it
        $this->assertSame(30, $bankAccount->getBalance());
    }
}
```

2.11. Registry (Registro)

2.11.1. Objetivo

Implementar um armazenamento centralizado para objetos frequentemente usados em toda a aplicação, é tipicamente implementado usando uma classe abstrata com apenas métodos estáticos (ou usando o padrão Singleton). Lembre que isto introduz em estado global, o qual deve ser evitado sempre! Em vez disso, implemente isto usando a Injeção de Dependência (Dependency Injection)!

2.11.2. Diagrama UML



2.11.3. Código

Você também pode encontrar este código no GitHub

Registry.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Registry;

use InvalidArgumentException;

abstract class Registry
{
    const LOGGER = 'logger';

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

```
    private static array $allowedKeys = [
        self::LOGGER,
    ];

    public static function set(string $key, Service $value)
    {
        if (!in_array($key, self::$allowedKeys)) {
            throw new InvalidArgumentException('Invalid key given');
        }

        self::$services[$key] = $value;
    }

    public static function get(string $key): Service
    {
        if (!in_array($key, self::$allowedKeys) || !isset(self::$services[$key])) {
            throw new InvalidArgumentException('Invalid key given');
        }

        return self::$services[$key];
    }
}
```

Service.php

```
<?php

namespace DesignPatterns\Structural\Registry;

class Service
{

}
```

2.11.4. Teste

Tests/RegistryTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Registry\Tests;

use InvalidArgumentException;
use DesignPatterns\Structural\Registry\Registry;
use DesignPatterns\Structural\Registry\Service;
use PHPUnit\Framework\MockObject\MockObject;
use PHPUnit\Framework\TestCase;

class RegistryTest extends TestCase
{
    /**
     * @var Service
     */
    private MockObject $service;
```

```
    protected function setUp(): void
    {
        $this->service = $this->getMockBuilder(Service::class)->getMock();
    }

    public function testSetAndGetLogger()
    {
        Registry::set(Registry::LOGGER, $this->service);

        $this->assertSame($this->service, Registry::get(Registry::LOGGER));
    }

    public function testThrowsExceptionWhenTryingToSetInvalidKey()
    {
        $this->expectException(InvalidArgumentException::class);

        Registry::set('foobar', $this->service);
    }

    /**
     * notice @runInSeparateProcess here: without it, a previous test might have
set it already and
     * testing would not be possible. That's why you should implement Dependency
Injection where an
     * injected class may easily be replaced by a mockup
     *
     * @runInSeparateProcess
     */
    public function testThrowsExceptionWhenTryingToGetNotSetKey()
    {
        $this->expectException(InvalidArgumentException::class);

        Registry::get(Registry::LOGGER);
    }
}
```

3. Comportamental

Em Engenharia de Software, Padrões de Design Comportamentais são Padrões de Design (Design Patterns) que identificam padrões de comunicação comuns entre objetos e realizam estes padrões. Ao fazerem isto, estes padrões melhoram a flexibilidade na realização desta comunicação.

3.1. Cadeia de Responsabilidades (Chain Of Responsibilities)

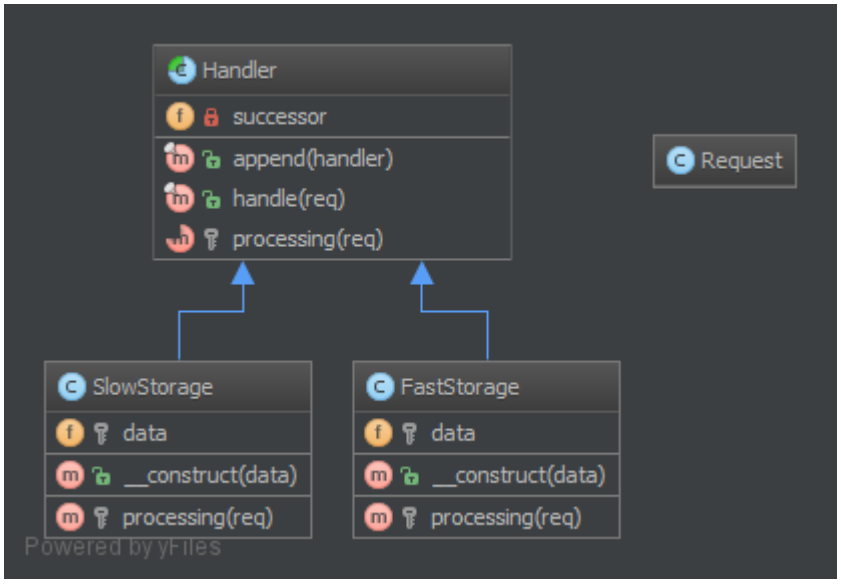
3.1.1. Objetivo

Construir uma cadeia de objetos para manipular uma chamada em ordem sequencial. Se um objeto não pode manipular a chamada, ele delega a chamada para o próximo na cadeia e assim por diante.

3.1.2. Exemplos

- estrutura de log, onde cada elemento da cadeia decide autonomamente o que fazer com a mensagem de log
- um filtro de Spam
- Cache: o primeiro objeto é uma instância de p.e. uma interface Memcached, se essa "falta" é delegada a chamada à interface de banco de dados.

3.1.3. Diagrama UML



3.1.4. Código

Você também pode encontrar este código no GitHub

Handler.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

use Psr\Http\Message\RequestInterface;

abstract class Handler
```

```
{
    private ?Handler $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
     */
    final public function handle(RequestInterface $request): ?string
    {
        $processed = $this->processing($request);

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

        return $processed;
    }

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

Responsible/FastStorage.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

class HttpInMemoryCacheHandler extends Handler
{
    private array $data;

    public function __construct(array $data, ?Handler $successor = null)
    {
        parent::__construct($successor);

        $this->data = $data;
    }

    protected function processing(RequestInterface $request): ?string
    {
        $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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

class SlowDatabaseHandler extends Handler
{
    protected function processing(RequestInterface $request): ?string
    {
        // this is a mockup, in production code you would ask a slow (compared to in-memor

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

3.1.5. Teste

Tests/ChainTest.php

```
<?php declare(strict_types=1);

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;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\UriInterface;

class ChainTest extends TestCase
{
    private Handler $chain;

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

    public function testCanRequestKeyInFastStorage()
    {
        $uri = $this->createMock(UriInterface::class);
        $uri->method('getPath')->willReturn('/foo/bar');
        $uri->method('getQuery')->willReturn('index=1');
```

```
        $request = $this->createMock(RequestInterface::class);
        $request->method('getMethod')
            ->willReturn('GET');
        $request->method('getUri')->willReturn($uri);

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

    public function testCanRequestKeyInSlowStorage()
    {
        $uri = $this->createMock(UriInterface::class);
        $uri->method('getPath')->willReturn('/foo/baz');
        $uri->method('getQuery')->willReturn('');

        $request = $this->createMock(RequestInterface::class);
        $request->method('getMethod')
            ->willReturn('GET');
        $request->method('getUri')->willReturn($uri);

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

3.2. Comando (Command)

3.2.1. Objetivo

Encapsular invocação e desacoplamento.

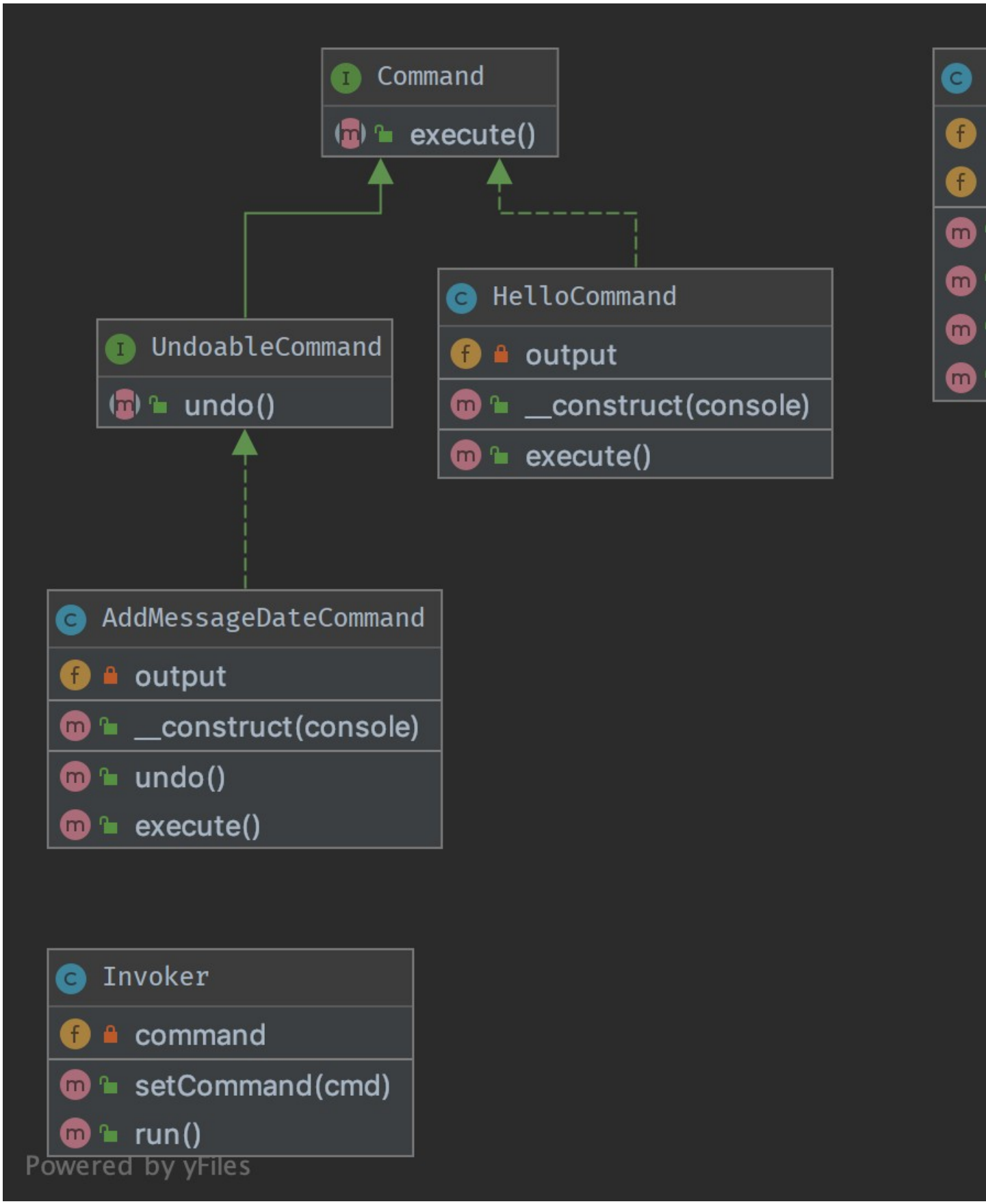
Nós temos um Invocador e um Receptor. Este padrão usa um “Comando” para delegar a chamada do método contra o Receptor e apresenta o mesmo método “execute”. Portanto, o Invocador sabe apenas chamar “execute” para processar o Comando do cliente. O Receptor é desacoplado do Invocador.

O segundo aspecto deste padrão é o desfazer(), o qual desfaz o método execute(). O comando pode ser também agregado para combinar mais comandos complexos com o mínimo copiar-colar e confiando na composição sobre herança.

3.2.2. Exemplos

- A text editor : all events are commands which can be undone, stacked and saved.
- grandes ferramentas CLI usam subcomandos para distribuir várias tarefas e empacotá-las em “módulos”, cada um deles pode ser implementado com o padrão Comando (p.e. vagrant)

3.2.3. Diagrama UML



3.2.4. Código

Você também pode encontrar este código no GitHub

Command.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

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

UndoableCommand.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

interface UndoableCommand extends Command
{
    /**
     * This method is used to undo change made by command execution
     */
    public function undo();
}
```

HelloCommand.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

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

    /**
     * Each concrete command is built with different receivers.
     * There can be one, many or completely no receivers, but there can be other commands
     */
    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');
    }
}
```

AddMessageDateCommand.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

/**
 * This concrete command tweaks receiver to add current date to messages
 * invoker just knows that it can call "execute"
 */
class AddMessageDateCommand implements UndoableCommand
{
    private Receiver $output;

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

    /**
     * Execute and make receiver to enable displaying messages date.
     */
    public function execute()
    {
        // sometimes, there is no receiver and this is the command which
        // does all the work
        $this->output->enableDate();
    }

    /**
     * Undo the command and make receiver to disable displaying messages date.
     */
    public function undo()
    {
        // sometimes, there is no receiver and this is the command which
        // does all the work
        $this->output->disableDate();
    }
}
```

Receiver.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

/**
 * Receiver is a specific service with its own contract and can be only concrete.
 */
class Receiver
{
```

```
    private bool $enableDate = false;

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

    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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

/**
 * Invoker is using the command given to it.
 * Example : an Application in SF2.
 */
class Invoker
{
    private Command $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 ...
     */
    public function setCommand(Command $cmd)
    {
        $this->command = $cmd;
    }

    /**
```

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

3.2.5. Teste

Tests/CommandTest.php

```
<?php declare(strict_types=1);

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->assertSame('Hello World', $receiver->getOutput());
    }
}
```

Tests/UndoableCommandTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command\Tests;

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

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

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

```
        $messageDateCommand = new AddMessageDateCommand($receiver);
        $messageDateCommand->execute();

        $invoker->run();
        $this->assertSame("Hello World\nHello World [".date('Y-m-d').']',
$receiver->getOutput());

        $messageDateCommand->undo();

        $invoker->run();
        $this->assertSame("Hello World\nHello World [".date('Y-m-d')."]\nHello
World", $receiver->getOutput());
    }
}
```

3.3. Iterator (Iterador)

3.3.1. Objetivo

Tornar um objeto iterável e fazê-lo aparecer como uma coleção de objetos.

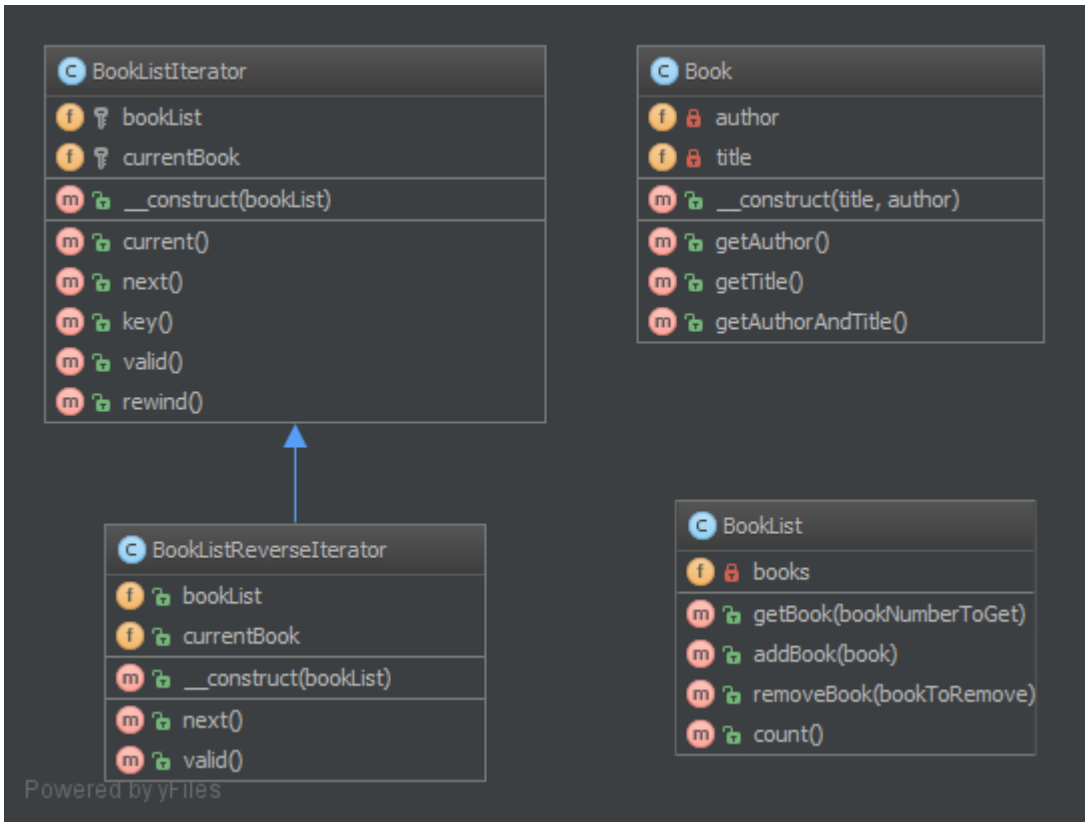
3.3.2. Exemplos

- processar um arquivo linha por linha somente passando por todas as linhas (as quais tenham uma representação em objeto) do arquivo (que, é claro, um objeto também)

3.3.3. Nota

A Standard PHP Library (SPL) define um Iterator de interface que é mais apropriado para isto! Muitas vezes você gostaria de implementar a interface Countable também, para permitir `count($object)` no seu objeto iterável

3.3.4. Diagrama UML



3.3.5. Código

Você também pode encontrar este código no GitHub

Book.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    private string $author;
    private string $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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Iterator;

use Countable;
use Iterator;

class BookList implements Countable, Iterator
{
    /**
     * @var Book[]
     */
    private array $books = [];
    private int $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]);
    }
}
```

3.3.6. Teste

Tests/IteratorTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Iterator\Tests;

use DesignPatterns\Behavioral\Iterator\Book;
use DesignPatterns\Behavioral\Iterator\BookList;
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->assertSame(
            [
                '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->assertSame(
            ['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);
    }
}
```

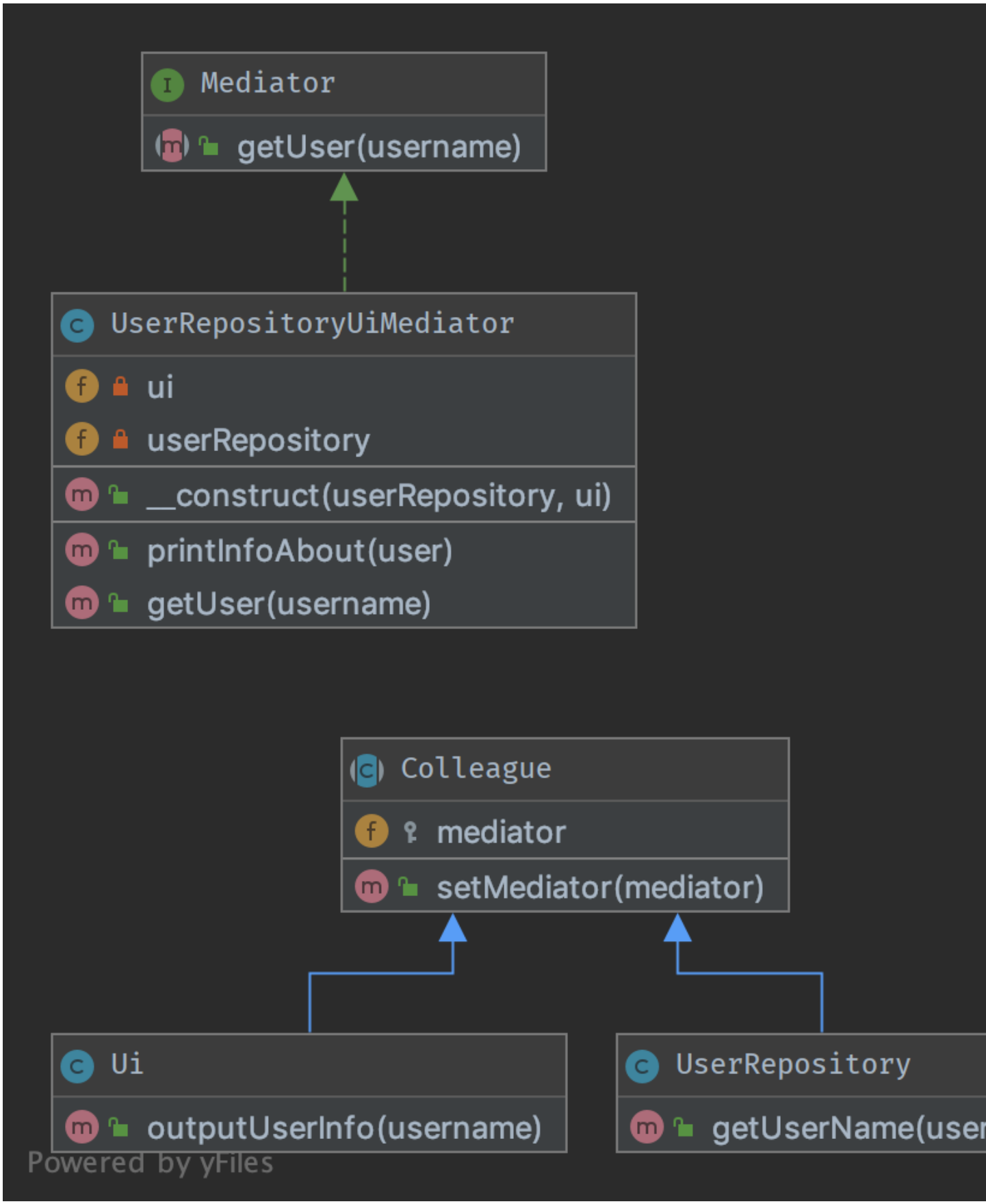
3.4. Mediator (Mediador)

3.4.1. Objetivo

Este padrão provê uma forma fácil para desacoplar muitos componentes trabalhando juntos. Ele é uma boa alternativa ao Observer SE você tem uma “inteligência central”, como um controlador (mas não no sentido do MVC).

Todos componentes (chamados Colleague - em inglês, Colega) são acoplados apenas ao Mediator e é uma coisa boa porque, em POO (Programação Orientada a Objetos), uma boa amiga é melhor que muitas. Esta é a característica-chave deste padrão.

3.4.2. Diagrama UML



3.4.3. Código

Você também pode encontrar este código no GitHub

Mediator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

interface Mediator
{
    public function getUser(string $username): string;
}
```

Colleague.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

abstract class Colleague
{
    protected Mediator $mediator;

    public function setMediator(Mediator $mediator)
    {
        $this->mediator = $mediator;
    }
}
```

Ui.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

class Ui extends Colleague
{
    public function outputUserInfo(string $username)
    {
        echo $this->mediator->getUser($username);
    }
}
```

UserRepository.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

class UserRepository extends Colleague
{
    public function getUserName(string $user): string
    {
        return 'User: ' . $user;
    }
```

```
}
```

UserRepositoryUiMediator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

class UserRepositoryUiMediator implements Mediator
{
    private UserRepository $userRepository;
    private Ui $ui;

    public function __construct(UserRepository $userRepository, Ui $ui)
    {
        $this->userRepository = $userRepository;
        $this->ui = $ui;

        $this->userRepository->setMediator($this);
        $this->ui->setMediator($this);
    }

    public function printInfoAbout(string $user)
    {
        $this->ui->outputUserInfo($user);
    }

    public function getUser(string $username): string
    {
        return $this->userRepository->getUserName($username);
    }
}
```

3.4.4. Teste

Tests/MediatorTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Ui;
use DesignPatterns\Behavioral\Mediator\UserRepository;
use DesignPatterns\Behavioral\Mediator\UserRepositoryUiMediator;
use PHPUnit\Framework\TestCase;

class MediatorTest extends TestCase
{
    public function testOutputHelloWorld()
    {
        $mediator = new UserRepositoryUiMediator(new UserRepository(), new
Ui());

        $this->expectOutputString('User: Dominik');
        $mediator->printInfoAbout('Dominik');
    }
```

```
}
```

3.5. Memento (Lembrança)

3.5.1. Objetivo

Ele provê a habilidade de restaurar um objeto para seu estado anterior (desfazer via rollback) ou ganhar acesso ao estado do objeto sem revelar sua implementação (p.e. o objeto não é obrigado a ter uma funcionalidade para retornar ao estado atual).

O padrão Memento é implementado com três objetos: o Originator, um Caretaker e um Memento.

Memento - um objeto que *contém um snapshot único e concreto do estado* de qualquer objeto ou recurso: string, número, array, uma instance de classe e assim por diante. A singularidade, neste caso, não implica a proibição da existência de estados semelhantes em diferentes snapshots. Isso significa que o estado pode ser extraído como o clone independente. Qualquer objeto armazenado no Memento deve ser *uma cópia completa do objeto original em vez de uma referência* para o objeto original. O objeto Memento é um “objeto opaco” (o objeto que ninguém pode ou deve mudar).

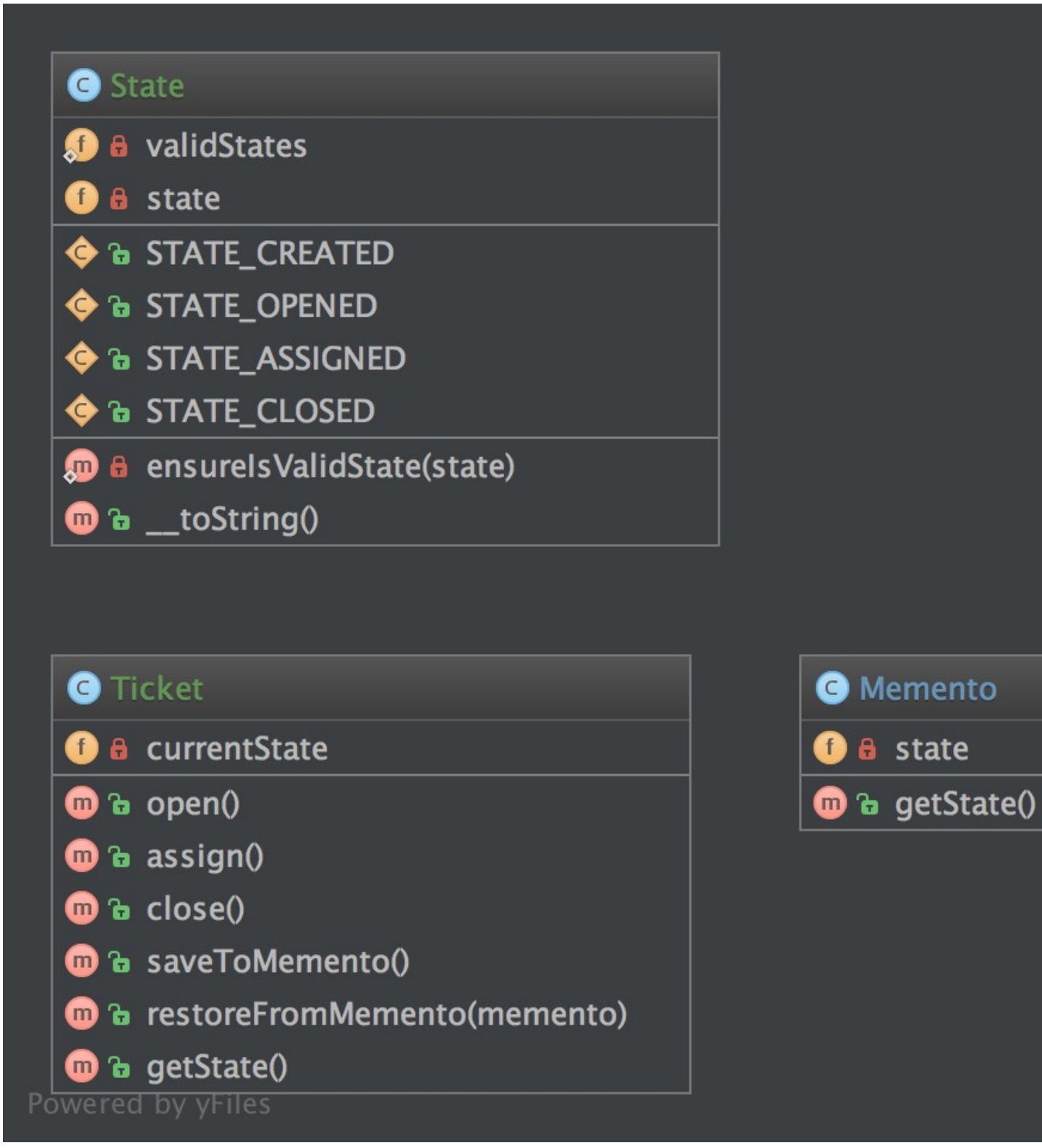
Originator - é um objeto que contém o *estado atual de um objeto externo é estritamente o tipo especificado*. Originator é capaz de criar uma cópia única deste estado e devolvê-lo envolto em um Memento. O O Originator não conhece a história das mudanças. Você pode definir um estado concreto ao Originator do lado de fora, que será considerado como atual. O Originator deve certificar-se de que determinado estado corresponde ao tipo permitido de objeto. Originator pode (mas não deve) ter quaisquer métodos, mas eles *não podem fazer alterações no estado do objeto salvo*.

Caretaker *controla a história dos estados*. Ele pode fazer alterações em um objeto; tomar uma decisão para salvar o estado de um objeto externo no Originator; pedir a partir do snapshot do Originator do estado atual ou definir o estado do Originator para equivalência com algum snapshot do histórico.

3.5.2. Exemplos

- A semente de um gerador de números pseudo-aleatórios
- O estado em uma máquina de estados finitos
- Controle para estados intermediários de ORM Model antes de salvar

3.5.3. Diagrama UML



3.5.4. Código

Você também pode encontrar este código no GitHub

Memento.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

class Memento
{
    private State $state;

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

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

State.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

use InvalidArgumentException;

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

    private string $state;

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

    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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

/**
 * Ticket is the "Originator" in this implementation
 */
class Ticket
{
    private State $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;
    }
}
```

3.5.5. Teste

Tests/MementoTest.php

```
<?php declare(strict_types=1);

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->assertSame(State::STATE_OPENED, (string) $ticket->getState());

        $memento = $ticket->saveToMemento();

        // assign the ticket
        $ticket->assign();
        $this->assertSame(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->assertSame(State::STATE_OPENED, (string) $ticket->getState());
        $this->assertNotSame($openedState, $ticket->getState());
    }
}
```

3.6. Objeto Nulo (Null Object)

3.6.1. Objetivo

Objeto Nulo (Null Object) não é um padrão de projeto GoF, mas um esquema o qual aparece com frequencia suficiente para ser considerado um padrão. Ele tem os seguintes benefícios:

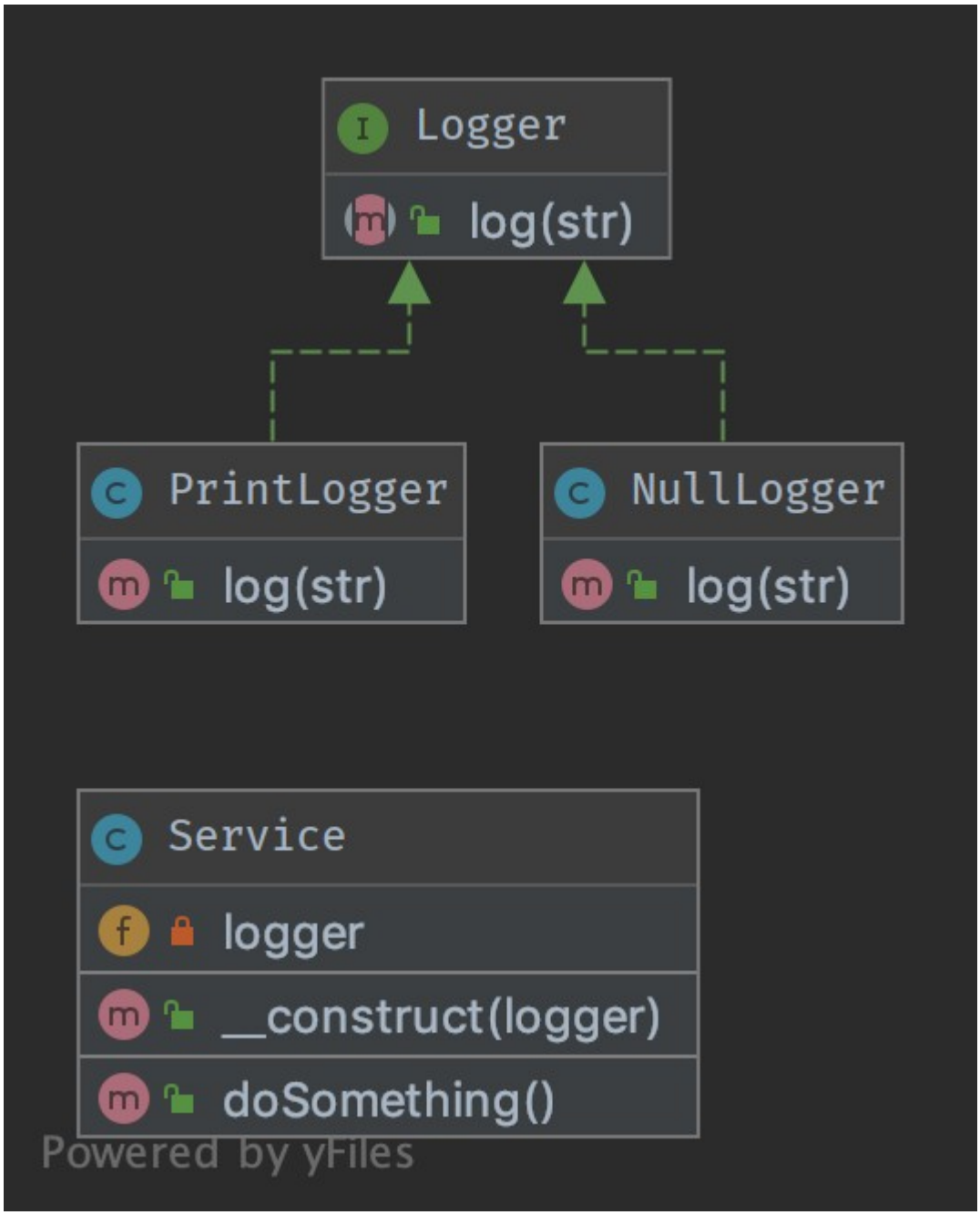
- Código do cliente é simplificado
- Reduz a chance de exceções de ponto nulo (null pointer exceptions)
- Menores condicionais requerem menos casos de teste

Métodos que retornam um objeto ou nulo devem ao invés disso, retornar um objeto ou `NullObject`. `NullObject`é simplesmente código boilerplate como `if (! is_null($obj)) { $obj->callSomething(); }` para apenas `$obj->callSomething();` eliminando então a checagem condicional no código cliente.

3.6.2. Exemplos

- Null logger or null output to preserve a standard way of interaction between objects, even if the shouldn’t do anything
- manipulador nulo em um padrão de Cadeia de Responsabilidades (Chain of Responsibilities)
- commando nulo em um padrão Comando (Command)

3.6.3. Diagrama UML



3.6.4. Código

Você também pode encontrar este código no GitHub

Service.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    private Logger $logger;

    public function __construct(Logger $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(
        $this->logger->log('We are in '.__METHOD__);
    }
}
```

Logger.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

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

PrintLogger.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

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

NullLogger.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

class NullLogger implements Logger
```

```
{
    public function log(string $str)
    {
        // do nothing
    }
}
```

3.6.5. Teste

Tests/LoggerTest.php

```
<?php declare(strict_types=1);

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();
    }
}
```

3.7. Observador (Observer)

3.7.1. Objetivo

Implementar um comportamento de publicação/subscrição para um objeto, sempre que um objeto “Sujeito” altera o seu estado, o “Observador” anexo será notificado. Ele é usado para ordenar a massa de objetos combinados e usa baixo acoplamento em seu lugar.

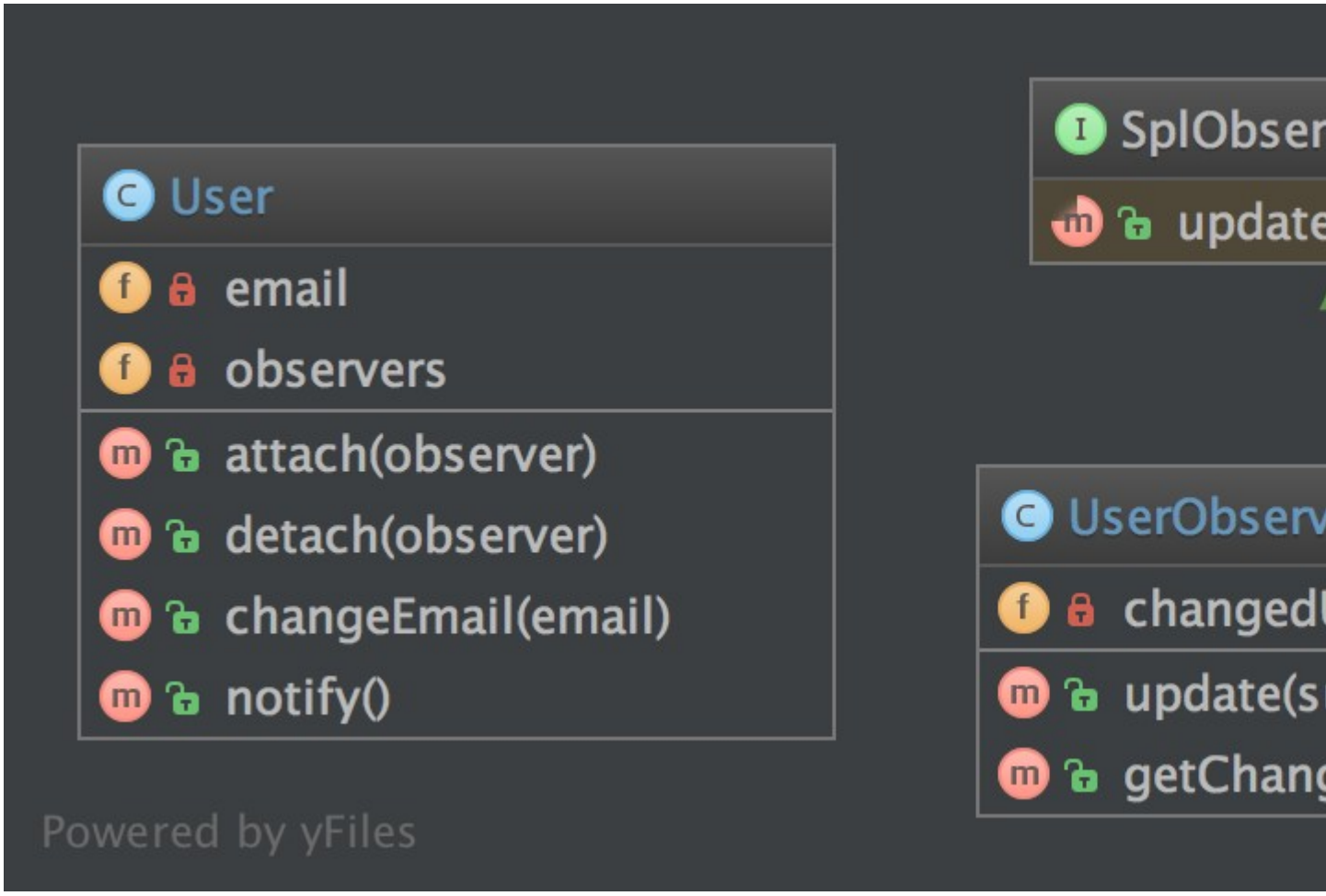
3.7.2. Exemplos

- um sistema de fila de mensagem é observado para apresentar o progresso de um trabalho na interface gráfica do usuário (GUI)

3.7.3. Nota

O PHP já define duas interfaces que podem ajudar a implementar este padrão: SplObserver e SplSubject.

3.7.4. Diagrama UML



3.7.5. Código

Você também pode encontrar este código no GitHub

User.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Observer;

use SplSubject;
use SplObjectStorage;
use SplObserver;

/**
 * User implements the observed object (called Subject), it maintains a list of observers
 * them in case changes are made on the User object
 */
```

```
class User implements SplSubject
{
    private string $email;
    private SplObjectStorage $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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Observer;

use SplObserver;
use SplSubject;

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

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

    /**
     * @return SplSubject[]
     */
```

```
    public function getChangedUsers(): array
    {
        return $this->changedUsers;
    }
}
```

3.7.6. Teste

Tests/ObserverTest.php

```
<?php declare(strict_types=1);

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());
    }
}
```

3.8. Especificação

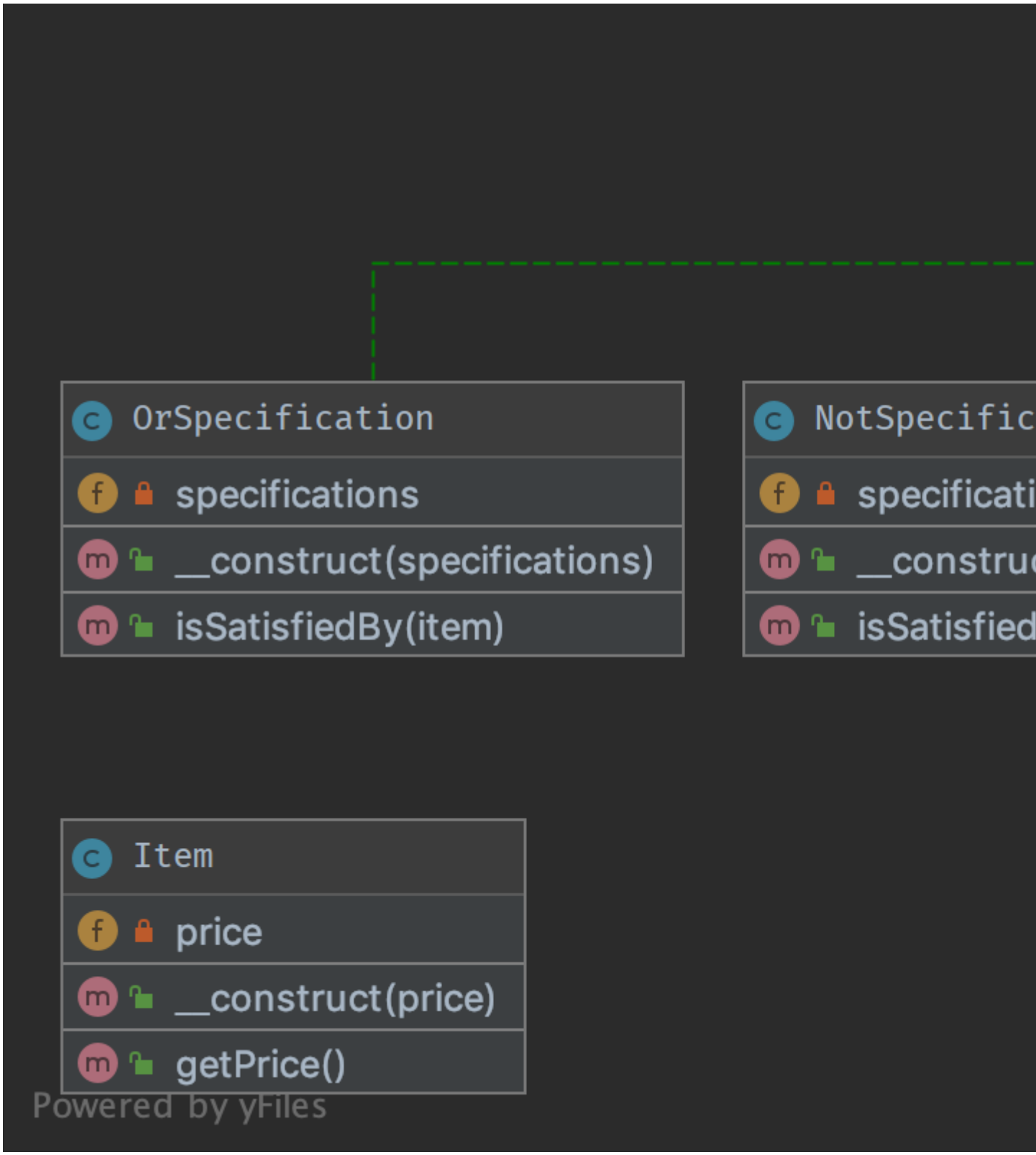
3.8.1. Objetivo

Constroe uma clara especificação das regras de negócio, com a qual os objetos podem ser validados. A classe de especificação composta tem um método chamado `isSatisfiedBy` que retorna verdadeiro ou falso dependendo se um dado objeto satisfaz a especificação.

3.8.2. Exemplos

- RulerZ

3.8.3. Diagrama UML



3.8.4. Código

Você também pode encontrar este código no GitHub

Item.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

class Item
{
    private float $price;

    public function __construct(float $price)
    {
        $this->price = $price;
    }

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

Specification.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

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

OrSpecification.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

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

    /**
     * @param Specification[] $specifications
     */
    public function __construct(Specification ...$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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

class PriceSpecification implements Specification
{
    private ?float $maxPrice;
    private ?float $minPrice;

    public function __construct(?float $minPrice, ?float $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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

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

    /**
     * @param Specification[] $specifications
     */
    public function __construct(Specification ...$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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

class NotSpecification implements Specification
{
    private Specification $specification;

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

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

3.8.5. Teste

Tests/SpecificationTest.php

```
<?php declare(strict_types=1);

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;

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

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

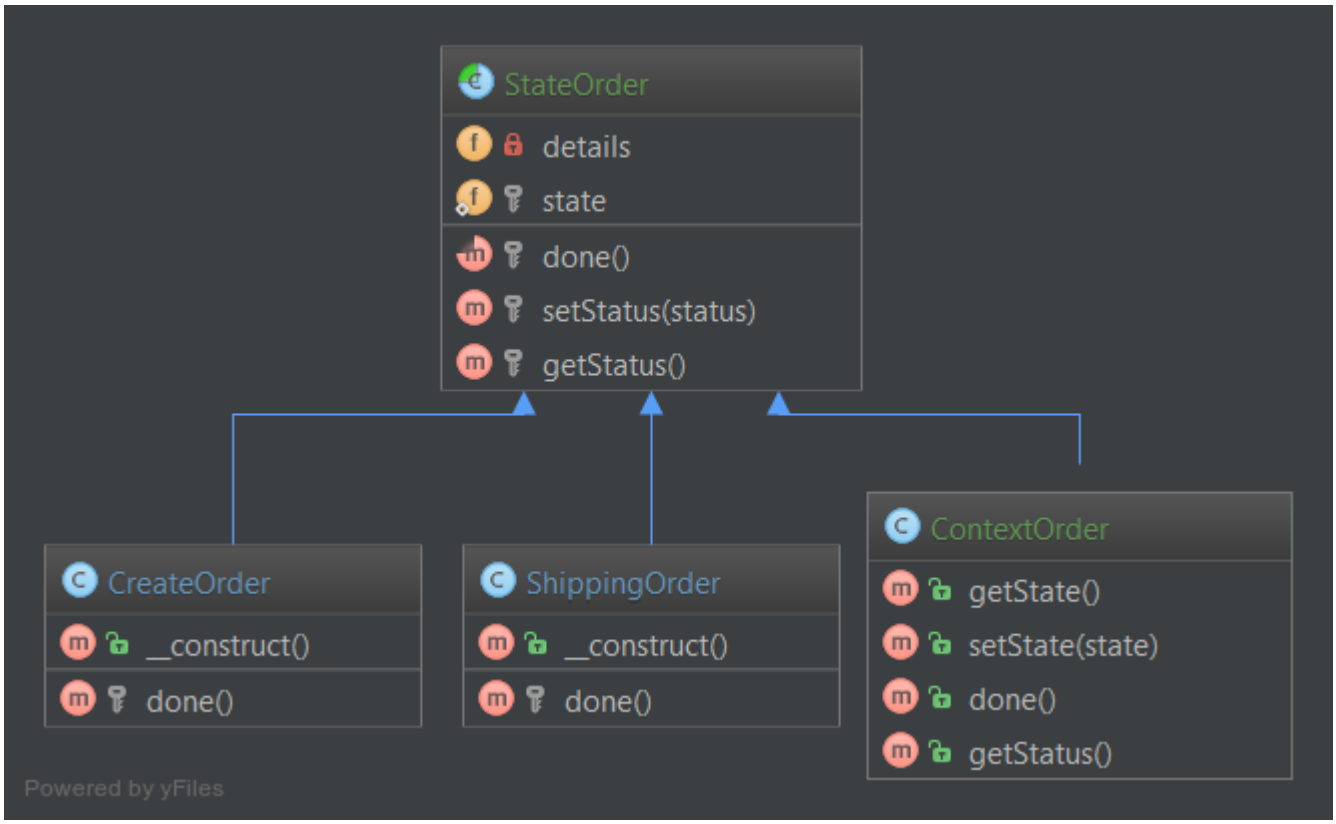
```
        $this->assertFalse($orSpec->isSatisfiedBy(new Item(100)));
        $this->assertTrue($orSpec->isSatisfiedBy(new Item(51)));
        $this->assertTrue($orSpec->isSatisfiedBy(new Item(150)));
    }

    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)));
    }
}
```

3.9. Estado

3.9.1. Objetivo

Encapsular comportamentos diversos para a mesma rotina baseado no estado de um objeto. Este pode ser uma forma limpa para um objeto alterar o seu comportamento em tempo de execução sem recorrer a grandes declarações condicionais monolíticas.

3.9.2. Diagrama UML



3.9.3. Código

Você também pode encontrar este código no GitHub

OrderContext.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class OrderContext
{
    private State $state;

    public static function create(): OrderContext
    {
        $order = new self();
        $order->state = new StateCreated();

        return $order;
    }

    public function setState(State $state)
    {
        $this->state = $state;
    }

    public function proceedToNext()
    {
```

```
        $this->state->proceedToNext($this);
    }

    public function toString()
    {
        return $this->state->toString();
    }
}
```

State.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

interface State
{
    public function proceedToNext(OrderContext $context);

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

StateCreated.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class StateCreated implements State
{
    public function proceedToNext(OrderContext $context)
    {
        $context->setState(new StateShipped());
    }

    public function toString(): string
    {
        return 'created';
    }
}
```

StateShipped.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class StateShipped implements State
{
    public function proceedToNext(OrderContext $context)
    {
        $context->setState(new StateDone());
    }

    public function toString(): string
    {
        return 'shipped';
    }
}
```

StateDone.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class StateDone implements State
{
    public function proceedToNext(OrderContext $context)
    {
        // there is nothing more to do
    }

    public function toString(): string
    {
        return 'done';
    }
}
```

3.9.4. Teste

Tests/StateTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\State\Tests;

use DesignPatterns\Behavioral\State\OrderContext;
use PHPUnit\Framework\TestCase;

class StateTest extends TestCase
{
    public function testIsCreatedWithStateCreated()
    {
        $orderContext = OrderContext::create();

        $this->assertSame('created', $orderContext->toString());
    }

    public function testCanProceedToStateShipped()
    {
        $contextOrder = OrderContext::create();
        $contextOrder->proceedToNext();

        $this->assertSame('shipped', $contextOrder->toString());
    }

    public function testCanProceedToStateDone()
    {
        $contextOrder = OrderContext::create();
        $contextOrder->proceedToNext();
        $contextOrder->proceedToNext();

        $this->assertSame('done', $contextOrder->toString());
    }

    public function testStateDoneIsTheLastPossibleState()
```

```
    {
        $contextOrder = OrderContext::create();
        $contextOrder->proceedToNext();
        $contextOrder->proceedToNext();
        $contextOrder->proceedToNext();

        $this->assertSame('done', $contextOrder->toString());
    }
}
```

3.10. Estratégia

3.10.1. Terminologia:

- Contexto
- Estratégia
- Estratégia Concreta

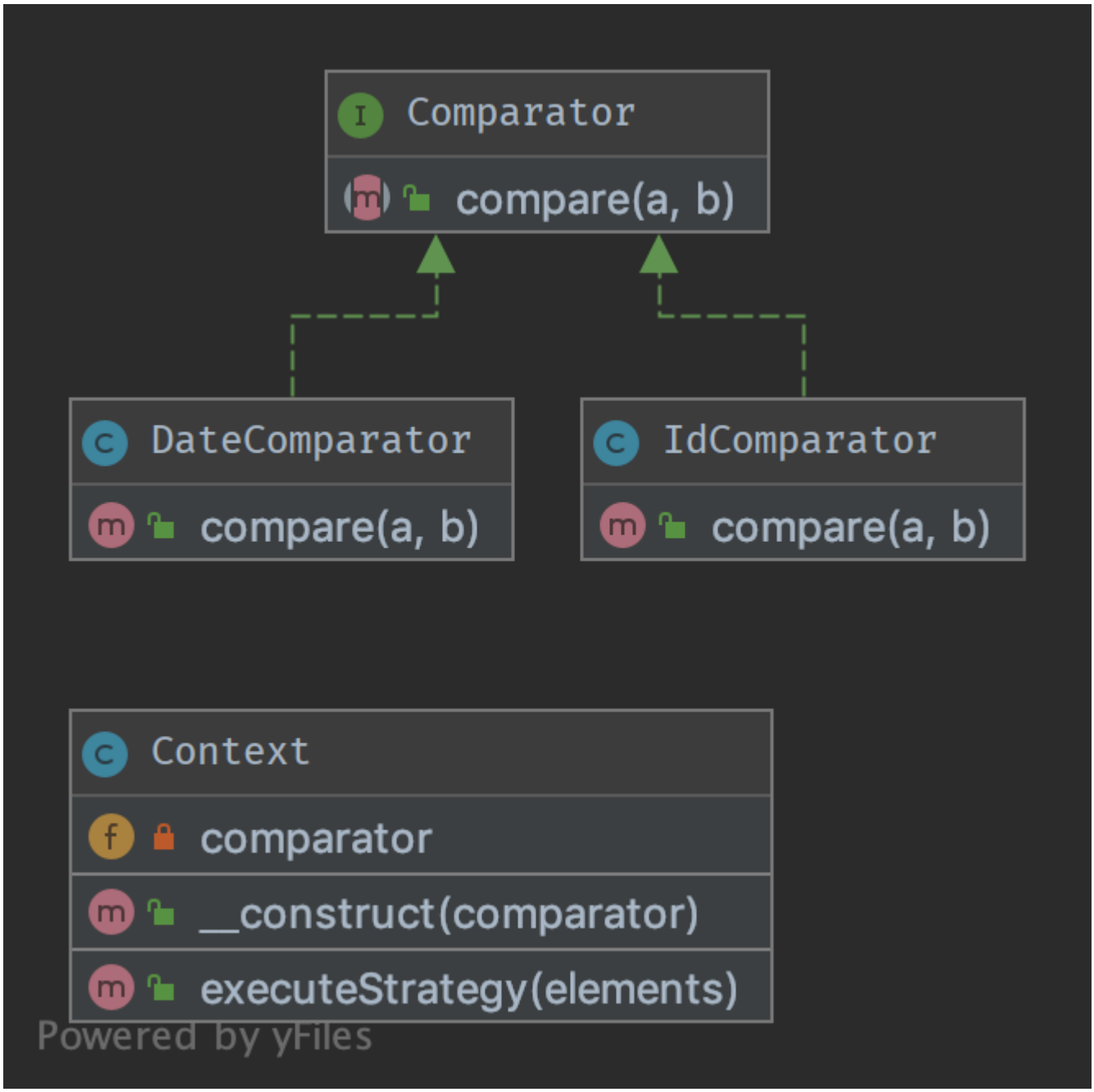
3.10.2. Objetivo

Separar estratégias e habilitar a troca rápida entre elas. Além disso, este padrão é uma boa alternativa à herença (ao invés de ter uma classe abstrata que é extendida).

3.10.3. Exemplos

- ordenando uma lista de objetos, uma estratégia por data, a outra por id
- simplifica o teste unitário: p.e. alternando entre armazenamento em memória ou em arquivo

3.10.4. Diagrama UML



3.10.5. Código

Você também pode encontrar este código no GitHub

Context.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;

class Context
{
    private Comparator $comparator;
```

```
    public function __construct(Comparator $comparator)
    {
        $this->comparator = $comparator;
    }

    public function executeStrategy(array $elements): array
    {
        uasort($elements, [$this->comparator, 'compare']);

        return $elements;
    }
}
```

Comparator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;

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

DateComparator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;

use DateTime;

class DateComparator implements Comparator
{
    public function compare($a, $b): int
    {
        $aDate = new DateTime($a['date']);
        $bDate = new DateTime($b['date']);

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

IdComparator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;

class IdComparator implements Comparator
{
    public function compare($a, $b): int
    {
```

```
        return $a['id'] <=> $b['id'];
    }
}
```

3.10.6. Teste

Tests/StrategyTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy\Tests;

use DesignPatterns\Behavioral\Strategy\Context;
use DesignPatterns\Behavioral\Strategy\DateComparator;
use DesignPatterns\Behavioral\Strategy\IdComparator;
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 Context(new IdComparator());
        $elements = $obj->executeStrategy($collection);

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

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

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

3.11. Método Modelo (Template Method)

3.11.1. Objetivo

Método Modelo é um Padrão de Projeto Comportamental.

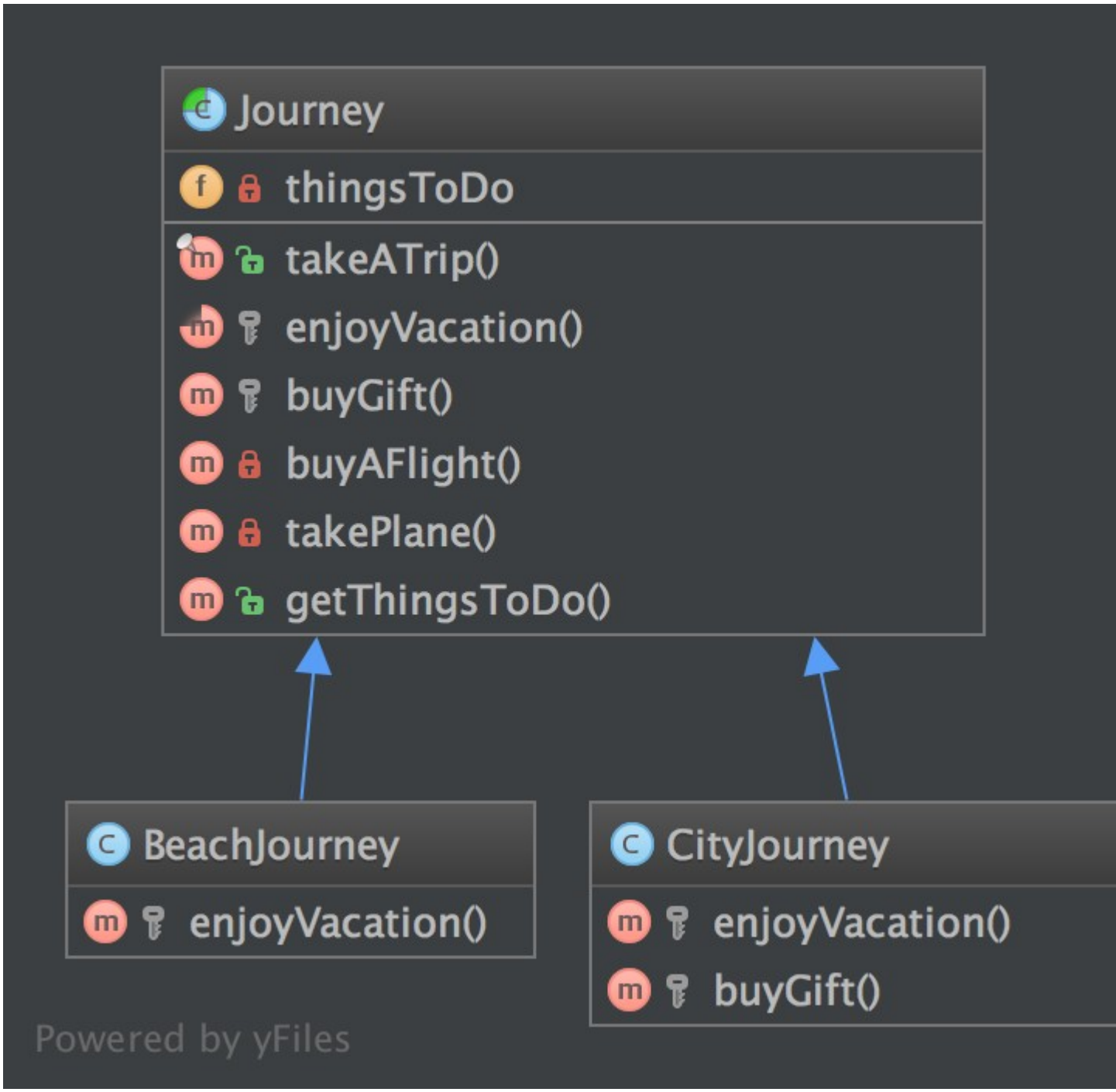
Talvez você já tenha encontrado ele muitas vezes. A ideia é deixar subclasses deste modelo abstrato "terminar" o comportamento de um algoritmo.

Também conhecido como o "Princípio de Hollywood": "Não nos chame, nós chamamos você". Esta classe não é chamada pelas subclasses, mas o inverso. Como? Com abstração do curso.

Em outras palavras, este é um esqueleto de um algoritmo, bem adequado às bibliotecas do framework. O usuário que apenas que implementar um método e a superclasse faz o trabalho.

Ele é uma forma fácil de desacoplar classes concretas e reduzir o copia-cola, este é o motivo de encontrar ele em todo lugar.

3.11.2. Diagrama UML



3.11.3. Código

Você também pode encontrar este código no GitHub

Journey.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\TemplateMethod;
```

```
abstract class Journey
{
    /**
     * @var string[]
     */
    private array $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
     */
    protected function buyGift(): ?string
    {
        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
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\TemplateMethod;

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

CityJourney.php

```php
<?php declare(strict_types=1);

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";
    }
}
```

3.11.4. Teste

Tests/JourneyTest.php

```php
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

use DesignPatterns\Behavioral\TemplateMethod\BeachJourney;
use DesignPatterns\Behavioral\TemplateMethod\CityJourney;
use PHPUnit\Framework\TestCase;

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

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

```
    public function testCanGetOnAJourneyToACity()
    {
        $cityJourney = new CityJourney();
        $cityJourney->takeATrip();

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

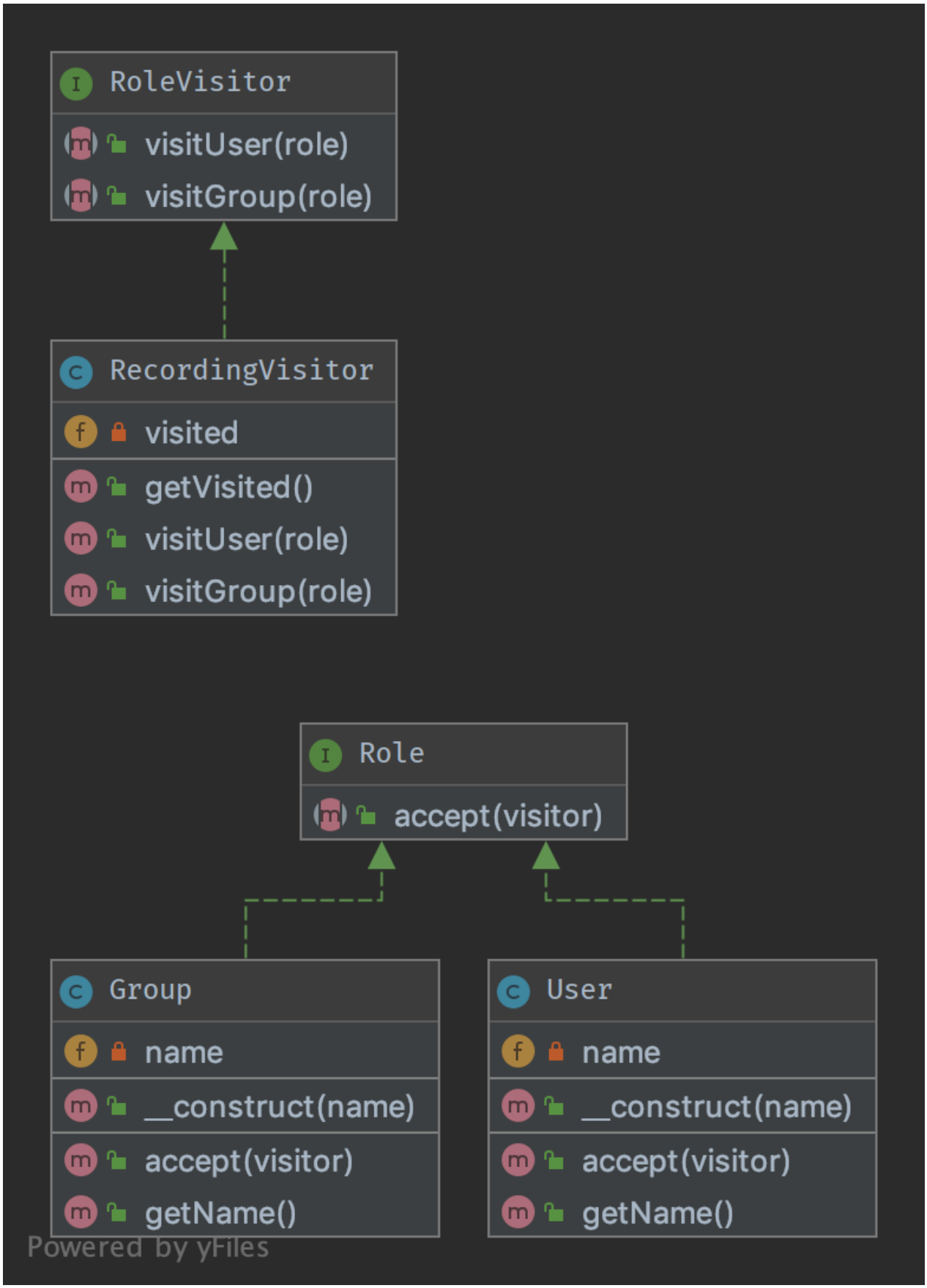
3.12. Visitante (Visitor)

3.12.1. Objetivo

O Padrão Visitante deixa as suas operações terceirizadas em objetos para outros objetos. A principal razão para fazer isto é manter a separação de preocupações. Porém, classes precisam definir um contrato para permitir visitantes (o método `Role::accept` no exemplo).

O contrato é uma classe abstrata, mas você pode ter também uma interface limpa. Neste caso, cada visitante tem que escolher a si mesmo qual método invocar no visitante.

3.12.2. Diagrama UML



3.12.3. Código

Você também pode encontrar este código no GitHub

RoleVisitor.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

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

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

RecordingVisitor.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

class RecordingVisitor implements RoleVisitor
{
    /**
     * @var Role[]
     */
    private array $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 declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;
```

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

User.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

class User implements Role
{
    private string $name;

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

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

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

Group.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

class Group implements Role
{
    private string $name;

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

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

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

3.12.4. Teste

Tests/VisitorTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\Tests\Visitor\Tests;

use DesignPatterns\Behavioral\Visitor\RecordingVisitor;
use DesignPatterns\Behavioral\Visitor\User;
use DesignPatterns\Behavioral\Visitor\Group;
use DesignPatterns\Behavioral\Visitor\Role;
use DesignPatterns\Behavioral\Visitor;
use PHPUnit\Framework\TestCase;

class VisitorTest extends TestCase
{
    private RecordingVisitor $visitor;

    protected function setUp(): void
    {
        $this->visitor = new RecordingVisitor();
    }

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

    /**
     * @dataProvider provideRoles
     */
    public function testVisitSomeRole(Role $role)
    {
        $role->accept($this->visitor);
        $this->assertSame($role, $this->visitor->getVisited()[0]);
    }
}
```

4. Outros

4.1. Localizador de Serviço

ESTE É CONSIDERADO UM ANTI-PATTERN!

O Localizador de Serviço (Service Locator) é considerado por muitas pessoas como um anti-padrão. Ele viola o princípio da inversão de dependência. O padrão Localizador de Serviço oculta as dependências da classe ao invés de expô-las, como você poderia fazer usando Injeção de Dependências (Dependency Injection). Em caso de mudanças destas dependências, você corre o risco de quebrar a funcionalidade das classes que estão as usando, fazendo com que seu sistema seja difícil de manter.

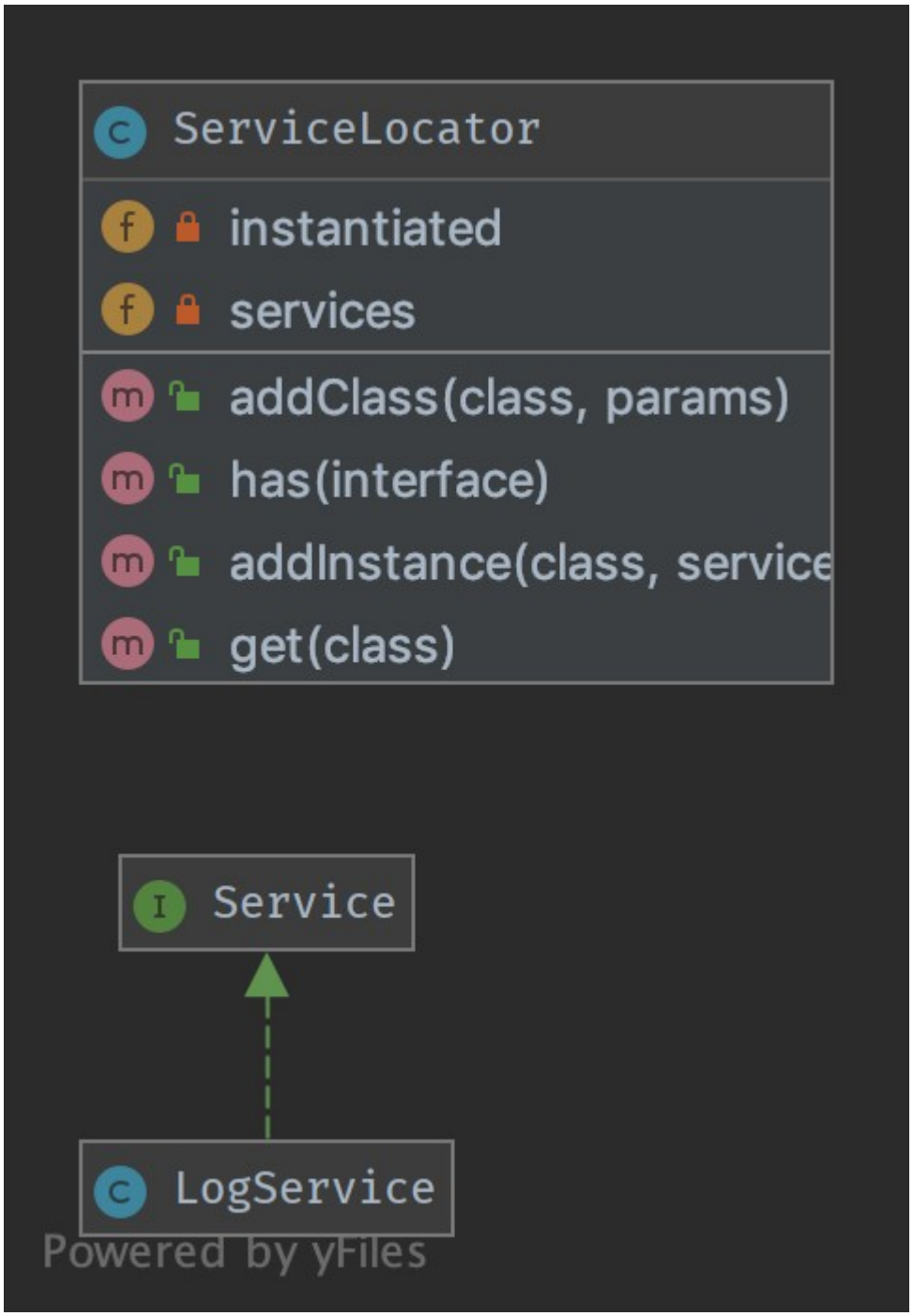
4.1.1. Objetivo

Oferece uma arquitetura desacoplada garantindo testabilidade manutenibilidade e extensão de um código. Injeção de dependências e Localizador de Serviços são implementações do padrão de Inversão de dependências.

4.1.2. Uso

Com `Localizador de Serviço` é possível registrar um serviço para uma determinada interface. Usando esta interface é possível obter esse serviço e usá-la dentro de outras classes da aplicação sem conhecimento de sua implementação. É possível configurar e injetar a instancia de um Localizador de Serviço no _bootstrap_.

4.1.3. Diagrama UML



4.1.4. Código

Você também pode encontrar este código no Github

Service.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;
```

```
interface Service
{

}
```

ServiceLocator.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\ServiceLocator;

use OutOfRangeException;
use InvalidArgumentException;

class ServiceLocator
{
    /**
     * @var string[][]
     */
    private array $services = [];

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

    public function addInstance(string $class, Service $service)
    {
        $this->instantiated[$class] = $service;
    }

    public function addClass(string $class, array $params)
    {
        $this->services[$class] = $params;
    }

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

    public function get(string $class): Service
    {
        if (isset($this->instantiated[$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 (!$object instanceof Service) {
            throw new InvalidArgumentException('Could not register service: is no instance
        }

        $this->instantiated[$class] = $object;

        return $object;
    }
}
```

LogService.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\ServiceLocator;

class LogService implements Service
{

}
```

4.1.5. Teste

Tests/ServiceLocatorTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\ServiceLocator\Tests;

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

class ServiceLocatorTest extends TestCase
{
    private ServiceLocator $serviceLocator;

    public function setUp(): void
    {
        $this->serviceLocator = new ServiceLocator();
    }

    public function testHasServices()
    {
        $this->serviceLocator->addInstance(LogService::class, new LogService());

        $this->assertTrue($this->serviceLocator->has(LogService::class));
        $this->assertFalse($this->serviceLocator->has(self::class));
    }

    public function
testGetWillInstantiateLogServiceIfNoInstanceHasBeenCreatedYet()
```

```
    {
        $this->serviceLocator->addClass(LogService::class, []);
        $logger = $this->serviceLocator->get(LogService::class);

        $this->assertInstanceOf(LogService::class, $logger);
    }
}
```

4.2. Repositório

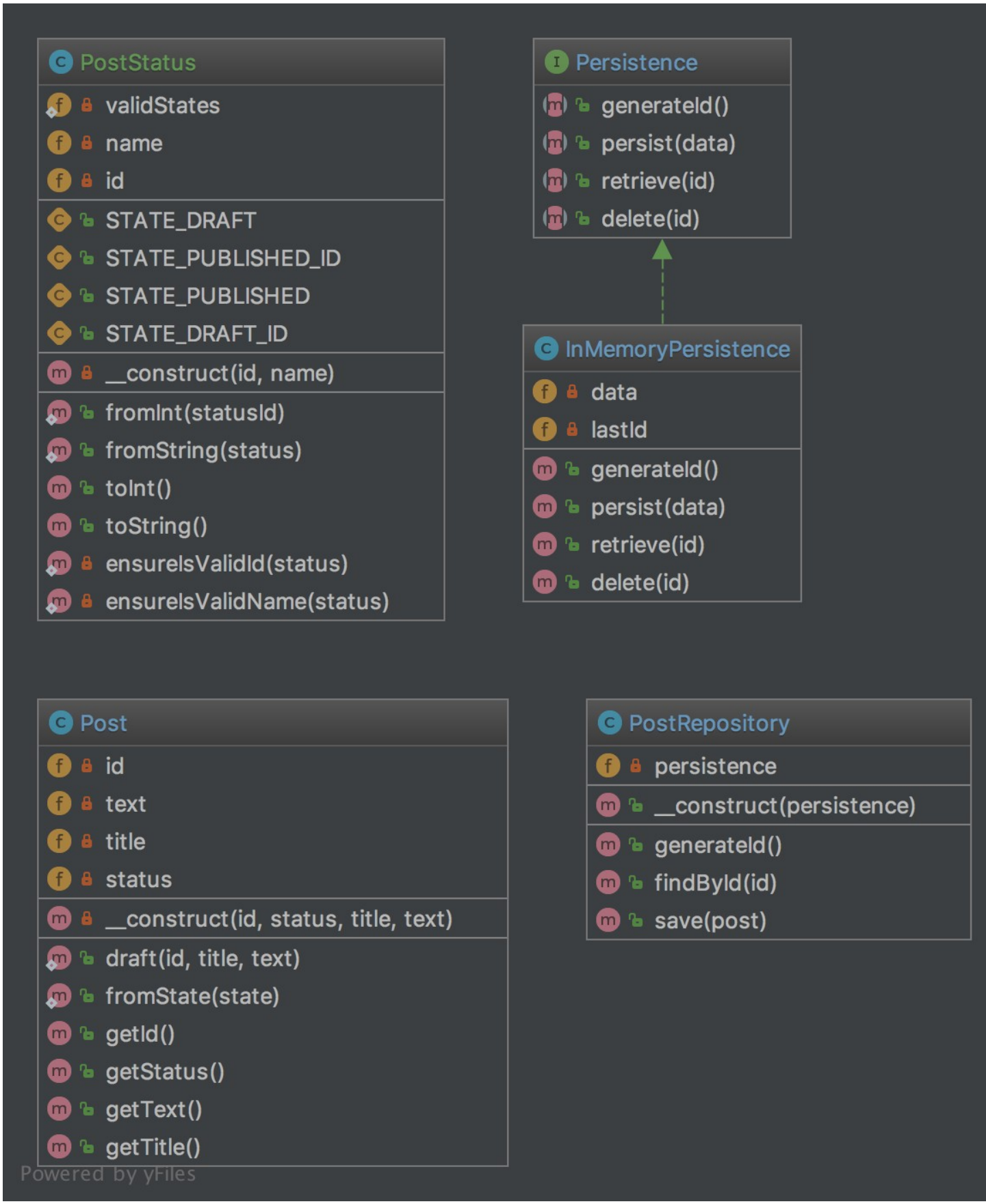
4.2.1. Objetivo

Faz a mediação entre o domínio e as camadas de mapeamento de dados usando uma coleção de interfaces para acessar os objetos de domínio. Repositóros encapsulam um conjunto de objetos persistidos em um data store e as operações feitas sobre eles, provendo uma visão mais orientada a objetos da camada de persistência. Repositorio também apóia o objetivo de alcançar uma separação limpa e uma dependencia unidirecional entre o domínio e as camadas de mapeamento de dados.

4.2.2. Exemplos

- Doctrine 2 ORM: existe um Repositório que faz a mediação entre Entity e DBAL contendo métodos para recuperação de objetos
- Framework Laravel

4.2.3. Diagrama UML



4.2.4. Código

Você também pode encontrar esse código no GitHub

Post.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository\Domain;

class Post
{
    private PostId $id;
    private PostStatus $status;
    private string $title;
    private string $text;

    public static function draft(PostId $id, string $title, string $text): Post
    {
        return new self(
            $id,
            PostStatus::fromString(PostStatus::STATE_DRAFT),
            $title,
            $text
        );
    }

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

    private function __construct(PostId $id, PostStatus $status, string $title, string $te
    {
        $this->id = $id;
        $this->status = $status;
        $this->text = $text;
        $this->title = $title;
    }

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

    public function getStatus(): PostStatus
    {
        return $this->status;
    }

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

    public function getTitle(): string
```

```
    {
        return $this->title;
    }
}
```

PostId.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;

/**
 * This is a perfect example of a value object that is identifiable by it's value alone an
 * is guaranteed to be valid each time an instance is created. Another important property
 * is immutability.
 *
 * Notice also the use of a named constructor (fromInt) which adds a little context when c
 */
class PostId
{
    private int $id;

    public static function fromInt(int $id): PostId
    {
        self::ensureIsValid($id);

        return new self($id);
    }

    private function __construct(int $id)
    {
        $this->id = $id;
    }

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

    private static function ensureIsValid(int $id)
    {
        if ($id <= 0) {
            throw new InvalidArgumentException('Invalid PostId given');
        }
    }
}
```

PostStatus.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;

/**
 * Like PostId, this is a value object which holds the value of the current status of a Po
 * either from a string or int and is able to validate itself. An instance can then be con
 */
class PostStatus
```

```
{
    const STATE_DRAFT_ID = 1;
    const STATE_PUBLISHED_ID = 2;

    const STATE_DRAFT = 'draft';
    const STATE_PUBLISHED = 'published';

    private static array $validStates = [
        self::STATE_DRAFT_ID => self::STATE_DRAFT,
        self::STATE_PUBLISHED_ID => self::STATE_PUBLISHED,
    ];

    private int $id;
    private string $name;

    public static function fromInt(int $statusId)
    {
        self::ensureIsValidId($statusId);

        return new self($statusId, self::$validStates[$statusId]);
    }

    public static function fromString(string $status)
    {
        self::ensureIsValidName($status);
        $state = array_search($status, self::$validStates);

        if ($state === false) {
            throw new InvalidArgumentException('Invalid state given!');
        }

        return new self($state, $status);
    }

    private function __construct(int $id, string $name)
    {
        $this->id = $id;
        $this->name = $name;
    }

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

    /**
     * there is a reason that I avoid using __toString() as it operates outside of the sta
     * and is therefor not able to operate well with exceptions
     */
    public function toString(): string
    {
        return $this->name;
    }

    private static function ensureIsValidId(int $status)
    {
        if (!in_array($status, array_keys(self::$validStates), true)) {
            throw new InvalidArgumentException('Invalid status id given');
        }
    }


    private static function ensureIsValidName(string $status)
```

```
    {
        if (!in_array($status, self::$validStates, true)) {
            throw new InvalidArgumentException('Invalid status name given');
        }
    }
}
```

PostRepository.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository;

use OutOfBoundsException;
use DesignPatterns\More\Repository\Domain\Post;
use DesignPatterns\More\Repository\Domain\PostId;

/**
 * This class is situated between Entity layer (class Post) and access object layer (Persi
 *
 * Repository encapsulates the set of objects persisted in a data store and the operations
 * providing a more object-oriented view of the persistence layer
 *
 * Repository also supports the objective of achieving a clean separation and one-way depe
 * between the domain and data mapping layers
 */
class PostRepository
{
    private Persistence $persistence;

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

    public function generateId(): PostId
    {
        return PostId::fromInt($this->persistence->generateId());
    }

    public function findById(PostId $id): Post
    {
        try {
            $arrayData = $this->persistence->retrieve($id->toInt());
        } catch (OutOfBoundsException $e) {
            throw new OutOfBoundsException(sprintf('Post with id %d does not exist', $id->
        }

        return Post::fromState($arrayData);
    }

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

Persistence.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository;

interface Persistence
{
    public function generateId(): int;

    public function persist(array $data);

    public function retrieve(int $id): array;

    public function delete(int $id);
}
```

InMemoryPersistence.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository;

use OutOfBoundsException;

class InMemoryPersistence implements Persistence
{
    private array $data = [];
    private int $lastId = 0;

    public function generateId(): int
    {
        $this->lastId++;

        return $this->lastId;
    }

    public function persist(array $data)
    {
        $this->data[$this->lastId] = $data;
    }

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

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

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

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

4.2.5. Teste

Tests/PostRepositoryTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\Repository\Tests;

use OutOfBoundsException;
use DesignPatterns\More\Repository\Domain\PostId;
use DesignPatterns\More\Repository\Domain\PostStatus;
use DesignPatterns\More\Repository\InMemoryPersistence;
use DesignPatterns\More\Repository\Domain\Post;
use DesignPatterns\More\Repository\PostRepository;
use PHPUnit\Framework\TestCase;

class PostRepositoryTest extends TestCase
{
    private PostRepository $repository;

    protected function setUp(): void
    {
        $this->repository = new PostRepository(new InMemoryPersistence());
    }

    public function testCanGenerateId()
    {
        $this->assertEquals(1, $this->repository->generateId()->toInt());
    }

    public function testThrowsExceptionWhenTryingToFindPostWhichDoesNotExist()
    {
        $this->expectException(OutOfBoundsException::class);
        $this->expectExceptionMessage('Post with id 42 does not exist');

        $this->repository->findById(PostId::fromInt(42));
    }

    public function testCanPersistPostDraft()
    {
        $postId = $this->repository->generateId();
        $post = Post::draft($postId, 'Repository Pattern', 'Design Patterns
PHP');
        $this->repository->save($post);

        $this->repository->findById($postId);

        $this->assertEquals($postId, $this->repository->findById($postId)-
>getId());
        $this->assertEquals(PostStatus::STATE_DRAFT, $post->getStatus()-
>toString());
    }
}
```

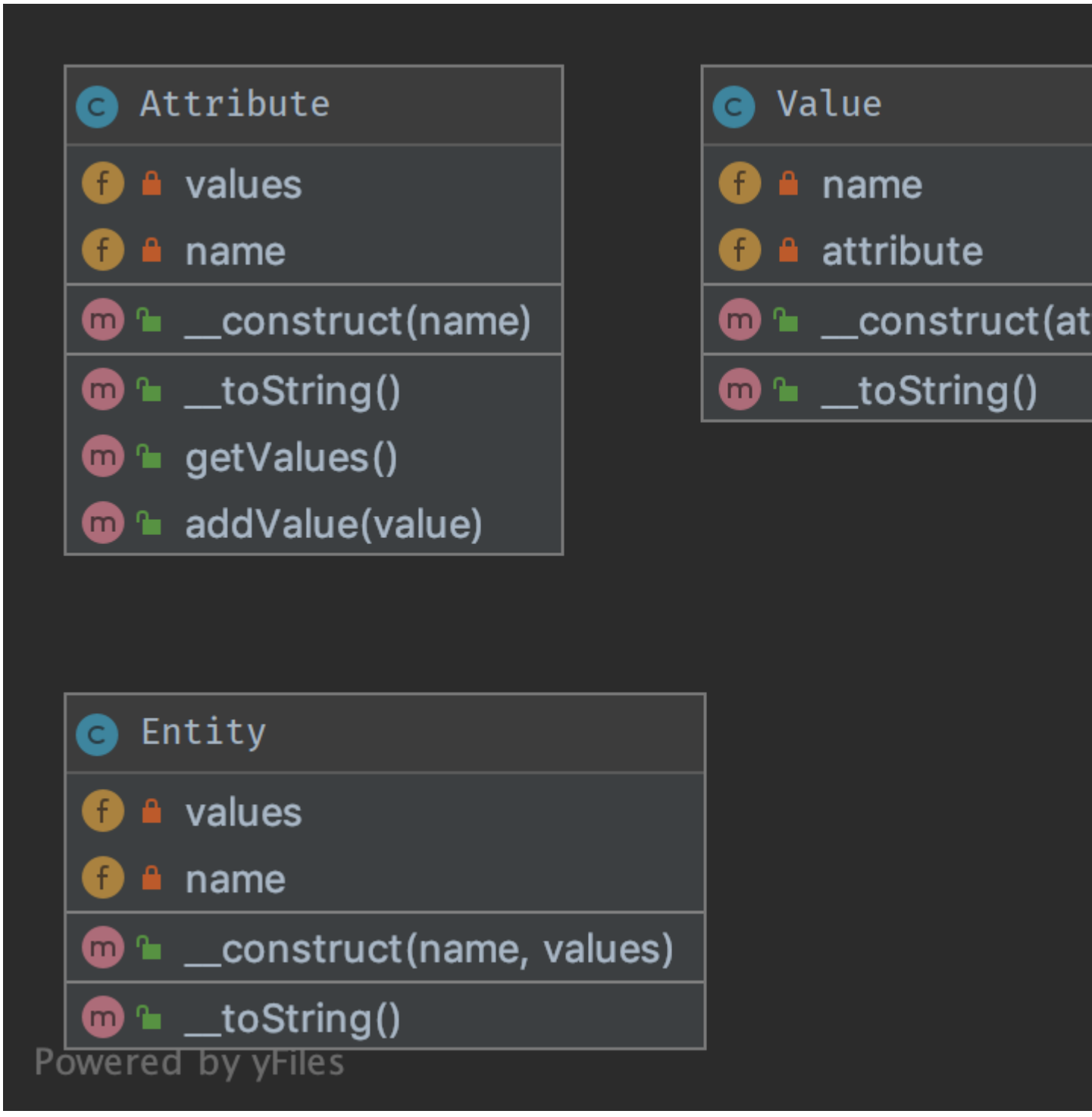
4.3. Entity-Attribute-Value (EAV)

O padrão Entidade–Atributo–Valor (EAV), a fim de implementar o modelo EAV com o PHP.

4.3.1. Objetivo

O modelo Entidade–atributo–valor (EAV) é um modelo de dados para descrever entidades onde o número de atributos (propriedades, parâmetros) que podem ser usados para descrevê-lo é potencialmente vasto, mas o número que realmente se aplicará para uma dada entidade é relativamente modesta.

4.3.2. Diagrama UML



4.3.3. Código

Você pode encontrar o código no Github

Entity.php

```
<?php declare(strict_types=1);
```

```
namespace DesignPatterns\More\EAV;

use SplObjectStorage;

class Entity
{
    /**
     * @var SplObjectStorage<Value,Value>
     */
    private $values;

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

    /**
     * @param string $name
     * @param Value[] $values
     */
    public function __construct(string $name, $values)
    {
        /** @var SplObjectStorage<Value,Value> 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 declare(strict_types=1);

namespace DesignPatterns\More\EAV;

use SplObjectStorage;

class Attribute
{
    private SplObjectStorage $values;
    private string $name;

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

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

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

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

Value.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\EAV;

class Value
{
    private Attribute $attribute;
    private string $name;

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

        $attribute->addValue($this);
    }

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

4.3.4. Teste

Tests/EAVTest.php

```
<?php declare(strict_types=1);

namespace DesignPatterns\More\EAV\Tests;

use DesignPatterns\More\EAV\Attribute;
use DesignPatterns\More\EAV\Entity;
use DesignPatterns\More\EAV\Value;
use PHPUnit\Framework\TestCase;

class EAVTest extends TestCase
{
```

```
    public function testCanAddAttributeToEntity()
    {
        $colorAttribute = new Attribute('color');
        $colorSilver = new Value($colorAttribute, 'silver');
        $colorBlack = new Value($colorAttribute, 'black');

        $memoryAttribute = new Attribute('memory');
        $memory8Gb = new Value($memoryAttribute, '8GB');

        $entity = new Entity('MacBook Pro', [$colorSilver, $colorBlack,
$memory8Gb]);

        $this->assertEquals('MacBook Pro, color: silver, color: black, memory:
8GB', (string) $entity);
    }
}
```

https://designpatternsphp.readthedocs.io/pt_BR/latest/README.html