# DesignPatternsPHP Documentation

*Release 1.0*

Dominik Liebler and contributors

сеп 30, 2023

# Съдържание

This is a collection of known design patterns and some sample code how to implement them in PHP. Every pattern has a small list of examples.

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

Глава 1

# Patterns

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

## 1.1 Създаващи

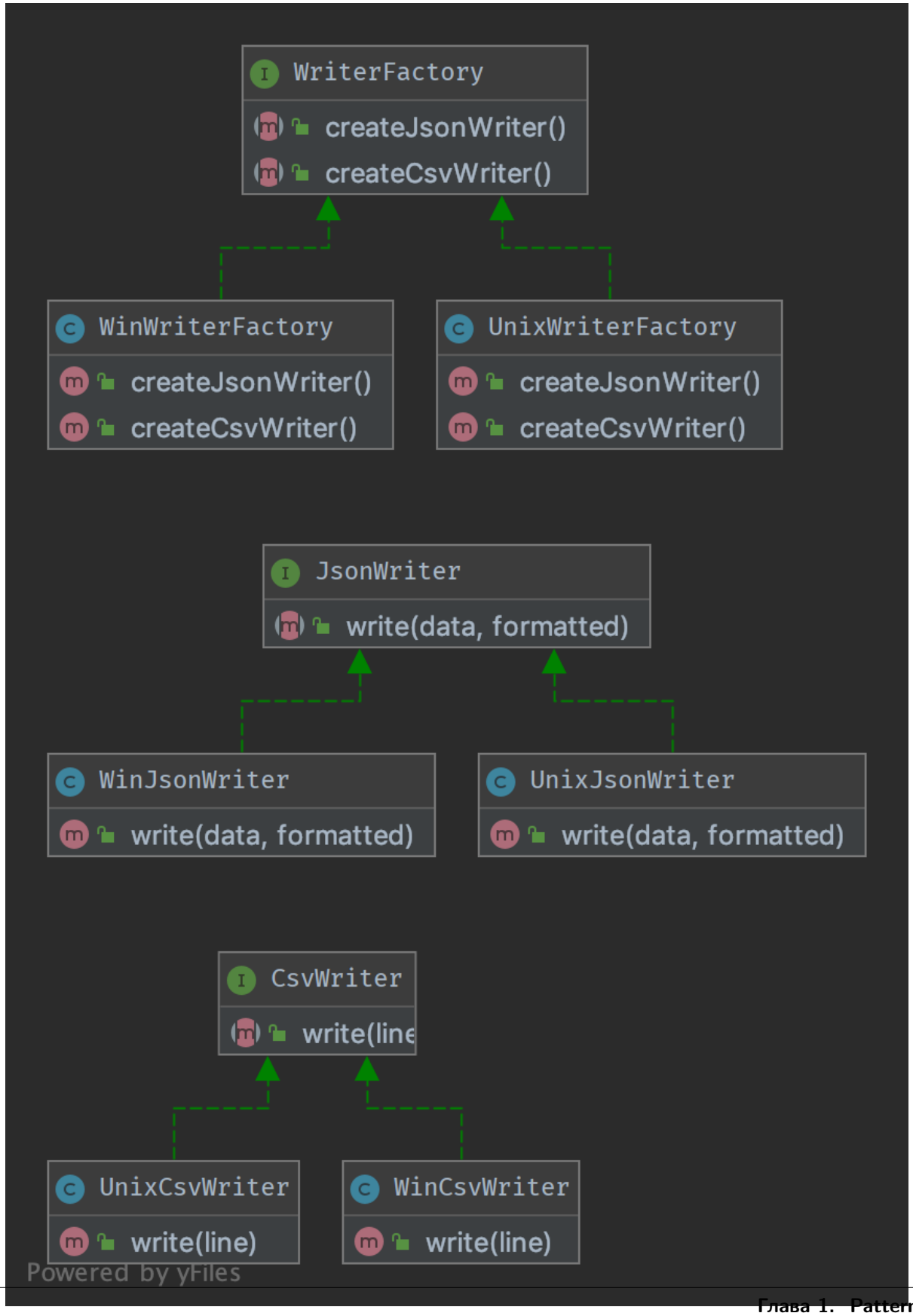
В софтуерното инженерство създаващи шаблони са шаблони, които се занимават с механизми за създаване на обекти, опитвайки се да създадат обекти по начин, подходящ за ситуацията. Основната форма на създаване на обект може да доведе до проблеми с дизайна или да добави сложност към дизайна. Създаващи шаблони решават този проблем чрез някакъв контрол върху създаването на този обект.

### 1.1.1 Абстрактна Фабрика

#### Предназначение

За създаване на поредица от свързани или зависими обекти, без да се посочват техните конкретни класове. Обикновено създадените класове изпълняват един и същ интерфейс. Клиентът на абстрактната фабрика не се интересува от това как се създават тези обекти, той просто знае как вървят заедно.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

WriterFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

CsvWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

JsonWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

UnixCsvWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

UnixJsonWriter.php

```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
<?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
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

WinJsonWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

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

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

WinWriterFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

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

### Тест

Tests/AbstractFactoryTest.php

```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;
```

(continues on next page)

(continued from previous page)

```php
use PHPUnit\Framework\TestCase;

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

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

### 1.1.2 Строител

#### Предназначение

Builder е интерфейс, който изгражда части от сложен обект.

Понякога, ако строител (builder) има по-добри познания за това, което изгражда, този интерфейс може да бъде абстрактен клас с методи по подразбиране (известен още като адаптер).

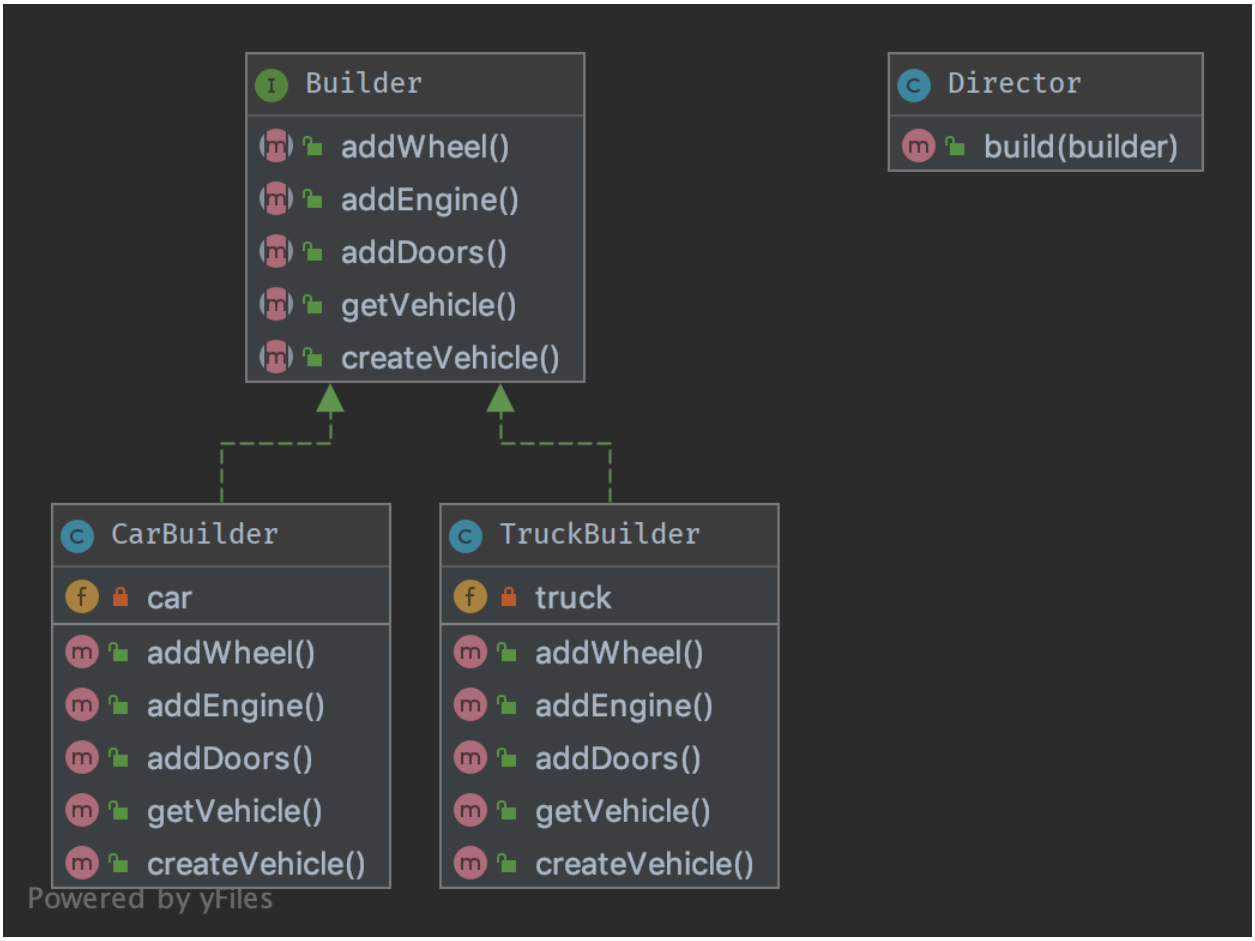
Ако имате сложно дърво за наследяване на обекти, логично е да имате сложно дърво за наследяване и за строители.

Забележка: Конструкторите често имат плавен интерфейс, вижте конструктор на фалшивия обект (mock builder) на PHPUnit например.

#### Примери

- PHPUnit: Mock Builder

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Director.php

```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
 */
```

(continues on next page)

(continued from previous page)

```php
class Director
{
    public function build(Builder $builder): Vehicle
    {
        $builder->createVehicle();
        $builder->addDoors();
        $builder->addEngine();
        $builder->addWheel();

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

Builder.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

interface Builder
{
    public function createVehicle(): void;

    public function addWheel(): void;

    public function addEngine(): void;

    public function addDoors(): void;

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

TruckBuilder.php

```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
{
```

(continues on next page)

(continued from previous page)

```
    private Truck $truck;

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

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

    public function addWheel(): void
    {
        $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(): void
    {
        $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;

```

(continues on next page)

(continued from previous page)

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

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

    public function addWheel(): void
    {
        $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(): void
    {
        $this->car = new Car();
    }

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

Parts/Vehicle.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

abstract class Vehicle
{
    final public function setPart(string $key, object $value)
    {
    }
}
```

Parts/Truck.php

```
<?php

declare(strict_types=1);

```

(continues on next page)

(continued from previous page)

```
namespace DesignPatterns\Creational\Builder\Parts;

class Truck extends Vehicle
{
}
```

Parts/Car.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Car extends Vehicle
{
}
```

Parts/Engine.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Engine
{
}
```

Parts/Wheel.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Wheel
{
}
```

Parts/Door.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

#### Тест

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);
    }
}
```

### 1.1.3 Метод Фабрика

#### Предназначение

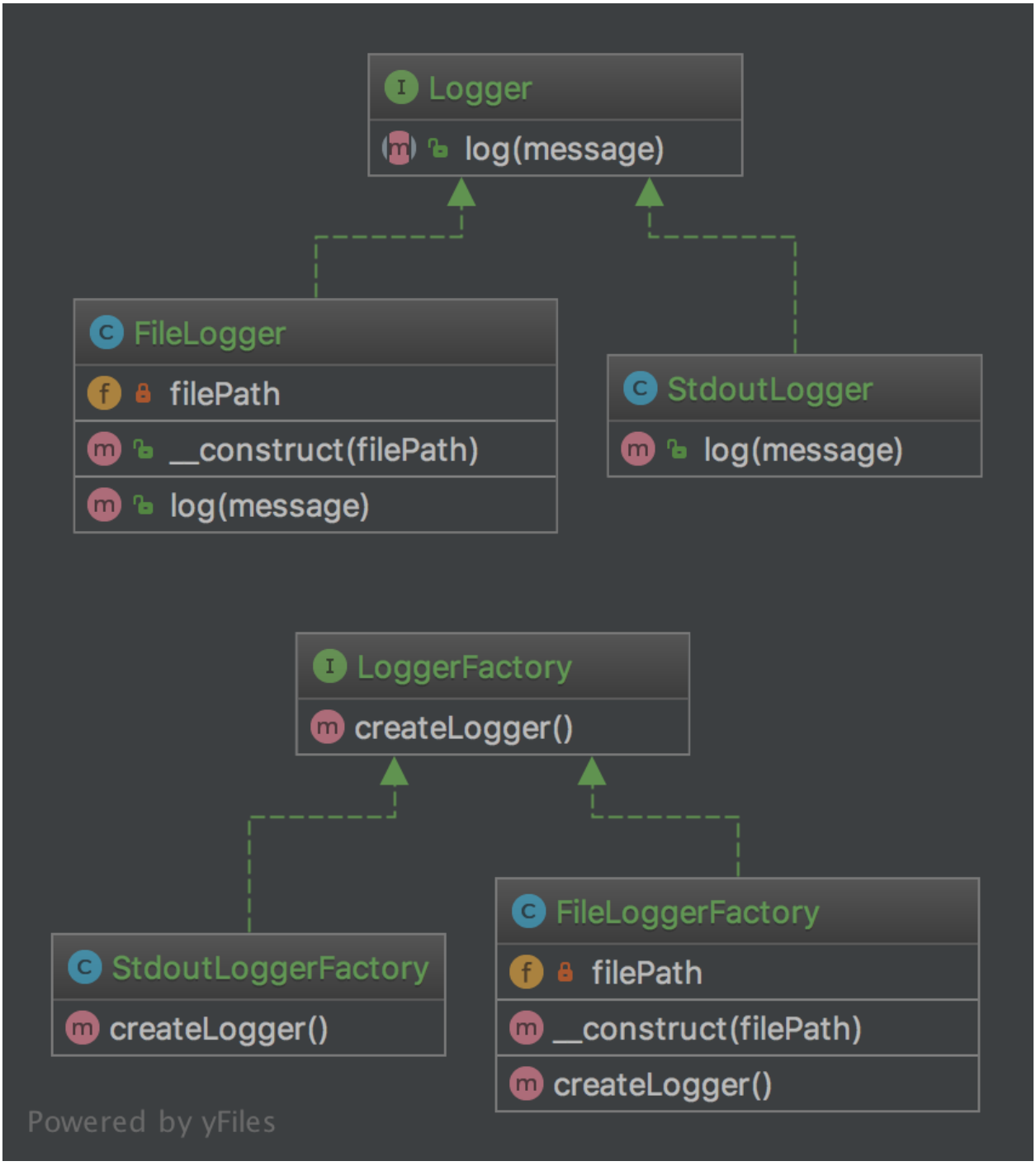
Добрата точка за SimpleFactory е, че можете да го подкласирате, за да реализира различни начини за създаване на обекти.

За прости случаи този абстрактен клас може да бъде просто интерфейс.

Този модел е „истински“ шаблон на проектиране, тъй като постига принципа на инверсия на зависимостите, известен още като „D“ в SOLID принципите.

Това означава, че класът FactoryMethod зависи от абстракциите, а не от конкретни класове. Това е истинският трик в сравнение с SimpleFactory или StaticFactory.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Logger.php

```php
<?php

declare(strict_types=1);

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
{
    public function __construct(private string $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
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\FactoryMethod;

class FileLoggerFactory implements LoggerFactory
{
    public function __construct(private string $filePath)
    {
    }

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

#### Тест

Tests/FactoryMethodTest.php

```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);
    }
}
```

### 1.1.4 Pool

#### Purpose

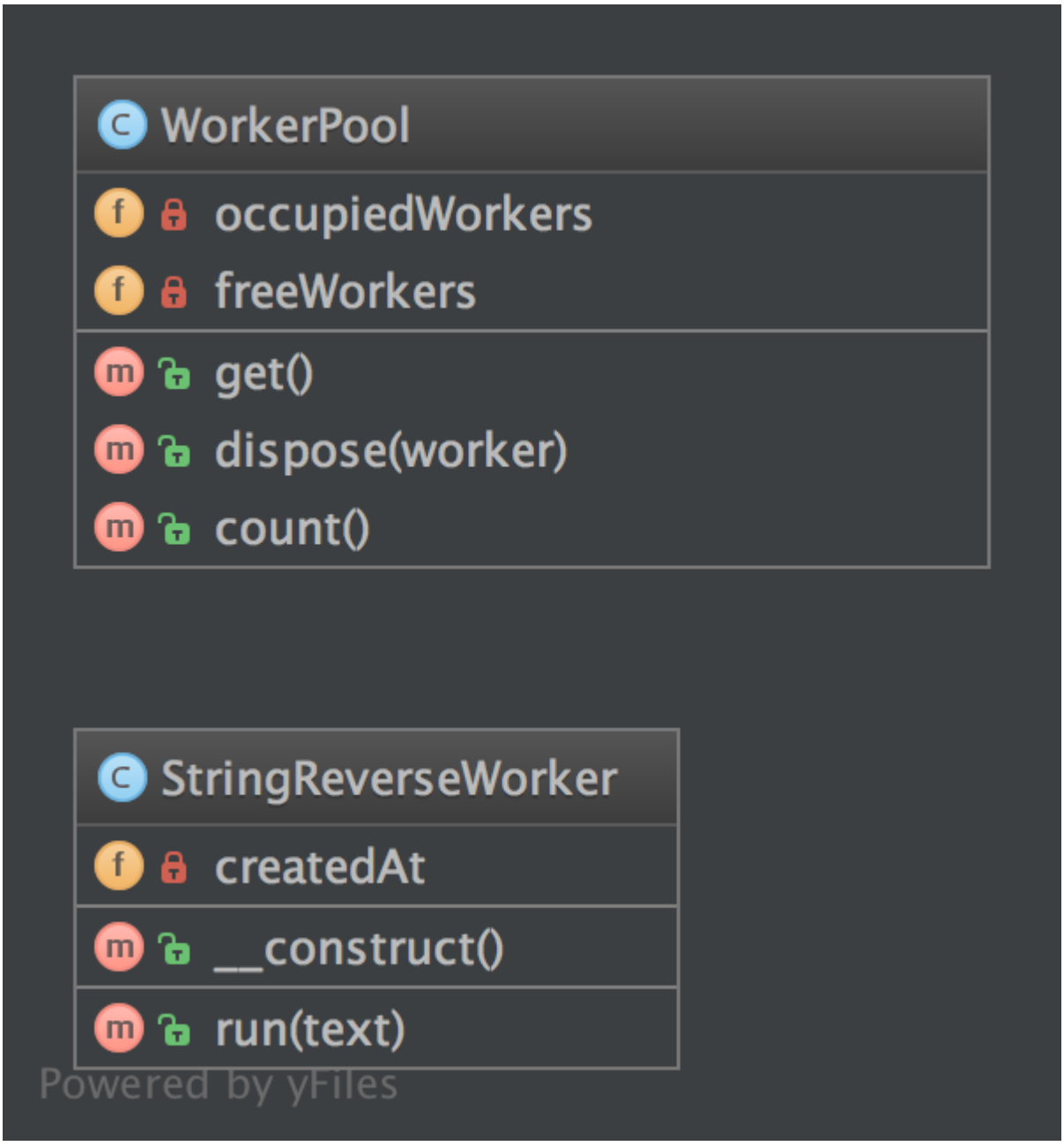
** Шаблонът за обект на обекти ** е софтуерен шаблон за създаване на дизайн, който използва набор от инициализирани обекти, поддържани в готовност за използване - „пул“ - вместо да ги разпределя и унищожава при поискване. Клиент на пула ще поиска обект от пула и ще извърши операции върху върнатия обект. Когато клиентът завърши, той връща обекта, който е специфичен тип фабричен обект, в пула, вместо да го унищожава.

Обединяването на обекти може да предложи значително подобрение на производителността в ситуации, когато цената за инициализиране на екземпляр на клас е висока, скоростта на създаване на екземпляр на клас е висока и броят на използваните екземпляри по всяко време е нисък. Обединеният обект се получава в предвидимо време, когато създаването на новите обекти (особено по мрежа) може да отнеме променливо време.

Тези предимства обаче важат най-вече за обекти, които са скъпи по отношение на времето, като връзки към база данни, връзки на сокети, нишки и големи графични обекти като шрифтове или растерни

изображения. В определени ситуации простото обединяване на обекти (които не съдържат външни ресурси, а само заемат памет) може да не е ефективно и да намали производителността.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

WorkerPool.php

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

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

StringReverseWorker.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Pool;

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

### Тест

Tests/PoolTest.php

```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);
    }
}
```

### 1.1.5 Прототип

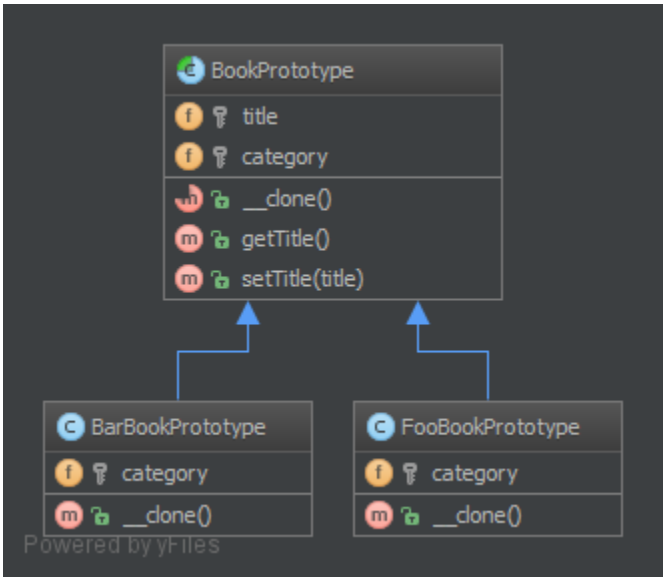
#### Предназначение

За да избегнете разходите за създаване на обекти по стандартния начин (new Foo()) и вместо това да създадете прототип и да го клонирате.

#### Примери

- Големи количества данни (напр. Създаване на 1 000 000 реда в база данни наведнъж чрез ORM).

#### UML Диаграма




#### Код

Можете също да намерите този код в GitHub

BookPrototype.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Prototype;

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

    abstract public function __clone();

```

(continues on next page)

(continued from previous page)

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

    final public function setTitle(string $title): void
    {
        $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()
    {
    }
}
```

#### Тест

Tests/PrototypeTest.php

```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);
        }
    }
}
```

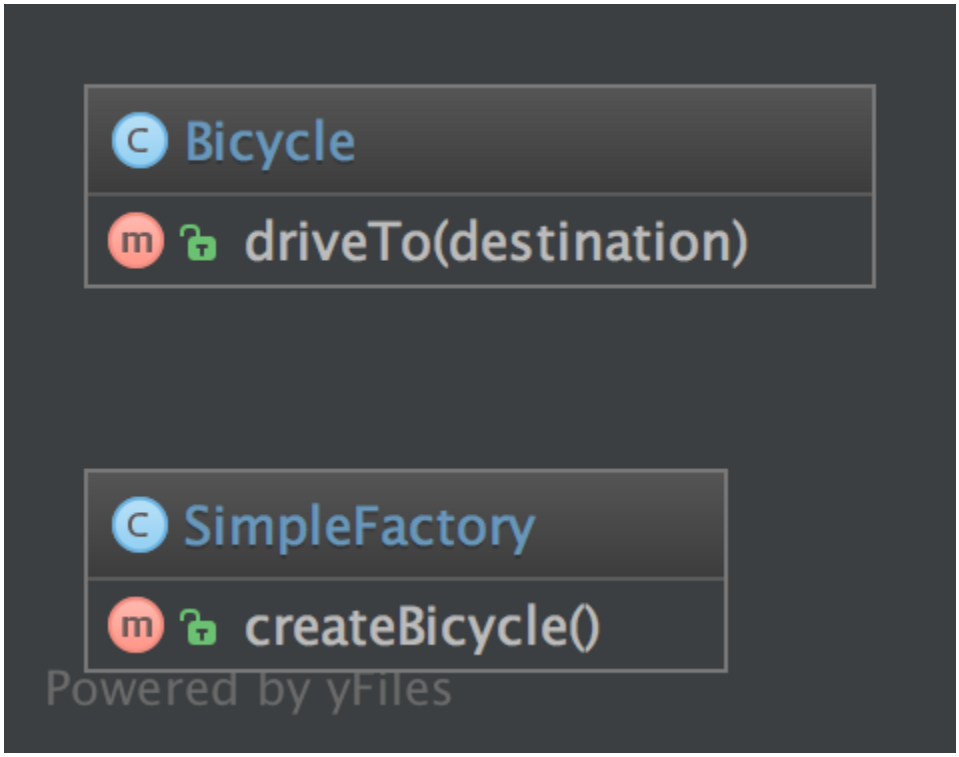
### 1.1.6 Simple Factory

#### Предназначение

SimpleFactory е прост фабричен шаблон.

Тя се различава от статичната фабрика, защото не е статична. Следователно можете да имате множество фабрики, параметризирани по различен начин, можете да го наследявате и да го макетирате (you can mock it). Винаги трябва да се предпочита пред статична фабрика!

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

SimpleFactory.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\SimpleFactory;

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

Bicycle.php

```php
<?php

declare(strict_types=1);

```

(continues on next page)

(continued from previous page)

```php
namespace DesignPatterns\Creational\SimpleFactory;

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

#### Usage

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

#### Тест

Tests/SimpleFactoryTest.php

```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);
    }
}
```

### 1.1.7 Сек

** ТОВА СЧИТА СЕ ЗА АНТИ-ШАБЛОН (anti-pattern)! ЗА ПО-ДОБРА ПРОВЕРИМОСТ И ПОДДЪРЖАЕМОСТ ИЗПОЛЗВАЙТЕ ИНЖЕКЦИЯ НА ЗАВИСИМОСТ! **

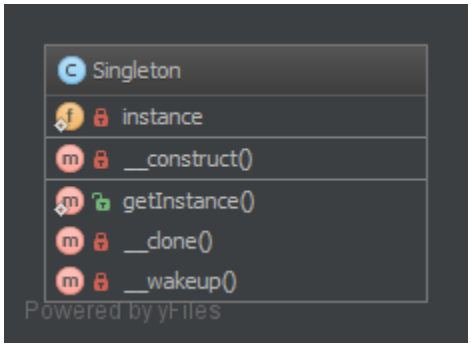
### Предназначение

Да има само един екземпляр на този обект в приложението, който да обработва всички повиквания.

### Примери

- DB конектор
- Регистратор (logger)
- Config Manager
- Threads Handling
- Lock file за приложението (във файловата система има само един . . . )

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Singleton.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Creational\Singleton;

use Exception;

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

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

(continues on next page)

(continued from previous page)

```
            self::$instance = new self();
        }

        return self::$instance;
    }

    /**
     * is not allowed to call from outside to prevent from creating multiple instances,
     * to use the singleton, you have to obtain the instance from␣
↪Singleton::getInstance() instead
     */
    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)
     */
    public function __wakeup()
    {
        throw new Exception("Cannot unserialize singleton");
    }
}
```

### Тест

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();
```

(continues on next page)

(continued from previous page)

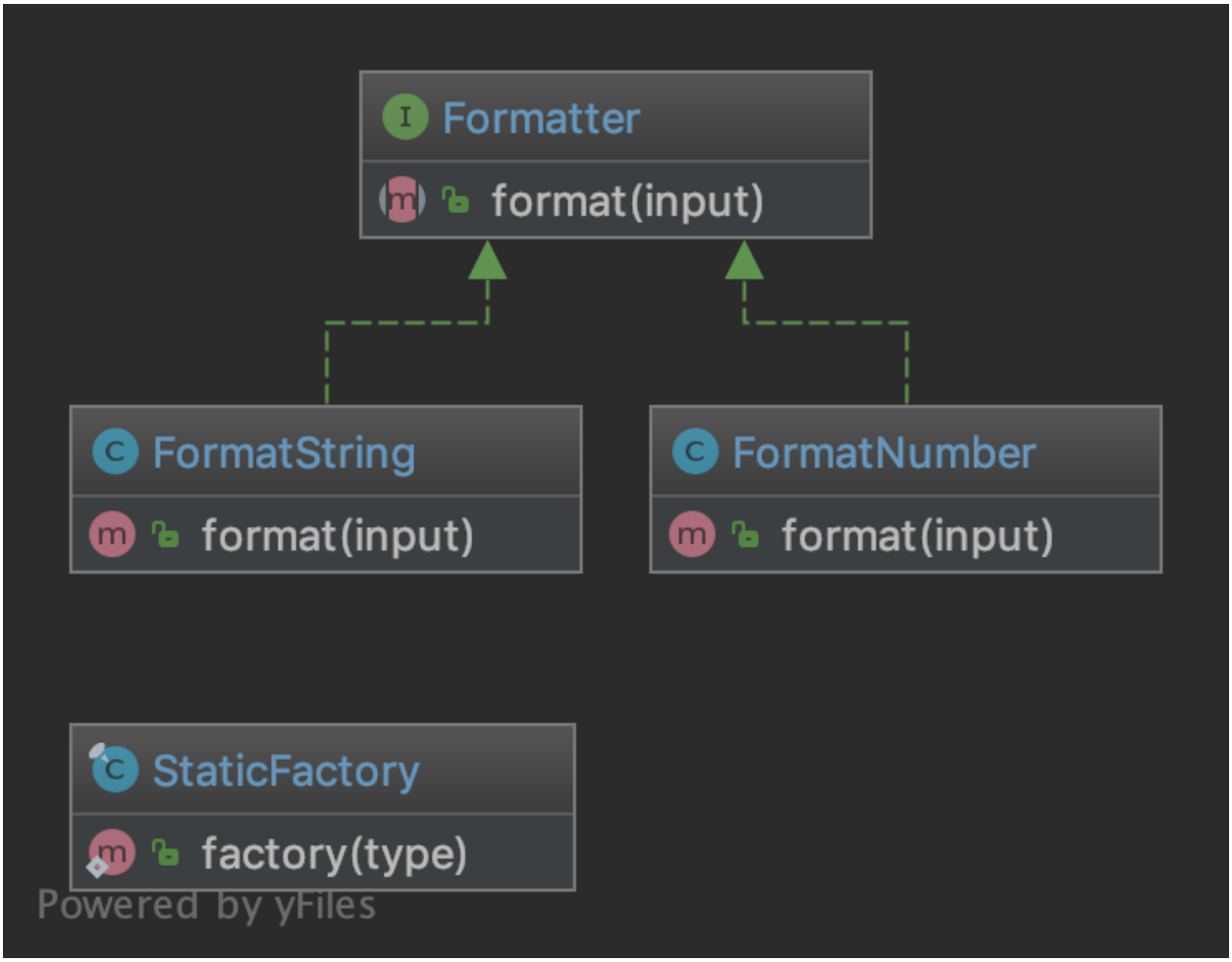
```
16
17          $this->assertInstanceOf(Singleton::class, $firstCall);
18          $this->assertSame($firstCall, $secondCall);
19      }
20  }
```

### 1.1.8 Статична фабрика

#### Предназначение

Подобно на AbstractFactory, този модел се използва за създаване на поредица от свързани или зависими обекти. Разликата между този и абстрактния фабричен модел е, че статичният фабричен модел използва само един статичен метод, за да създаде всички видове обекти, които може да създаде. Обикновено се нарича `factory` или `build`.

#### UML Диаграма



### Код

Можете също да намерите този код в 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 tests
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
final class StaticFactory
{
    public static function factory(string $type): Formatter
    {
        return match ($type) {
            'number' => new FormatNumber(),
            'string' => new FormatString(),
            default => 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
{
```

(continues on next page)

(continued from previous page)

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

FormatNumber.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);
    }
}
```

### Тест

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()
```

(continues on next page)

(continued from previous page)

```
26      {
27          $this->expectException(InvalidArgumentException::class);
28
29          StaticFactory::factory('object');
30      }
31  }
```

## 1.2 Структурни

В софтуерното инженерство, структурните шаблони са шаблони, които улесняват дизайна, като идентифицират прост начин за реализиране на връзките между обектите.

### 1.2.1 Адаптер / Обвивка

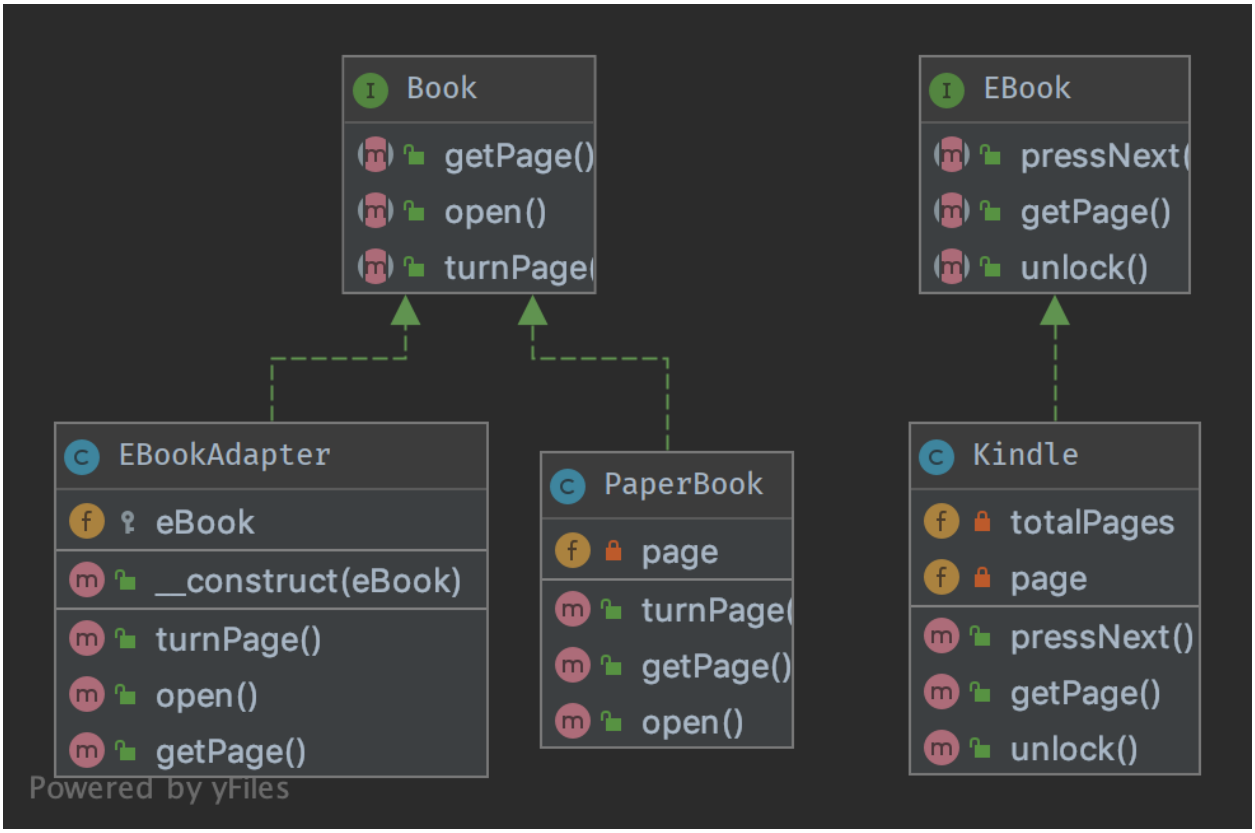
#### Предназначение

За да преведете един интерфейс за клас в съвместим интерфейс. Адаптерът позволява на класовете да работят заедно, което обикновено не може поради несъвместими интерфейси, като предоставя своя интерфейс на клиенти, докато използва оригиналния интерфейс.

#### Примери

- DB Client адаптер за библиотеки
- използване на множество различни уеб услуги (web services) и адаптери нормализират данните, така че резултатът да е еднакъв за всички

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Book.php

```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
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter;

class PaperBook implements Book
{
    private int $page;

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

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

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

EBook.php

```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
<?php

declare(strict_types=1);
```

(continues on next page)

(continued from previous page)

```php

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

Kindle.php

```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 package, some vendor code.
 * Notice that it uses another naming scheme and the implementation does something similar but in another way
 */
class Kindle implements EBook
```

(continues on next page)

(continued from previous page)

```
{
    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];
    }
}
```

### Тест

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()
```

(continues on next page)

(continued from previous page)

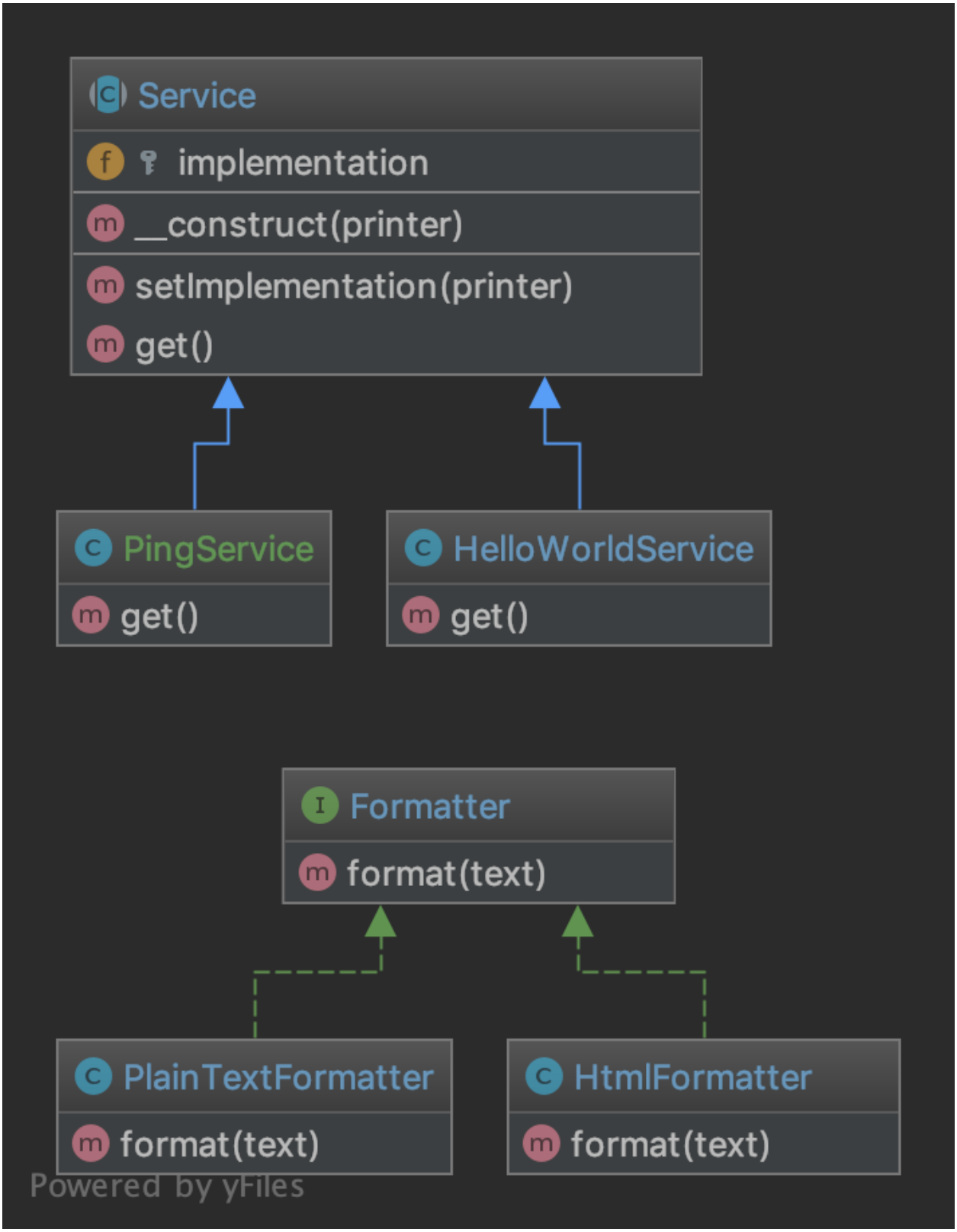
```
24      {
25          $kindle = new Kindle();
26          $book = new EBookAdapter($kindle);
27
28          $book->open();
29          $book->turnPage();
30
31          $this->assertSame(2, $book->getPage());
32      }
33  }
```

### 1.2.2 Мост

#### Предназначение

Отделете абстракцията от нейното изпълнение, така че двете да могат да варират независимо.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Formatter.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Bridge;

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

PlainTextFormatter.php

```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
<?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
<?php

declare(strict_types=1);
```

(continues on next page)

(continued from previous page)

```
4
5   namespace DesignPatterns\Structural\Bridge;
6
7   abstract class Service
8   {
9       public function __construct(protected Formatter $implementation)
10      {
11      }
12
13      final public function setImplementation(Formatter $printer)
14      {
15          $this->implementation = $printer;
16      }
17
18      abstract public function get(): string;
19  }
```

HelloWorldService.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Structural\Bridge;
6
7   class HelloWorldService extends Service
8   {
9       public function get(): string
10      {
11          return $this->implementation->format('Hello World');
12      }
13  }
```

PingService.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Structural\Bridge;
6
7   class PingService extends Service
8   {
9       public function get(): string
10      {
11          return $this->implementation->format('pong');
12      }
13  }
```

#### Тест

Tests/BridgeTest.php

```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());
    }
}
```

### 1.2.3 Композиция

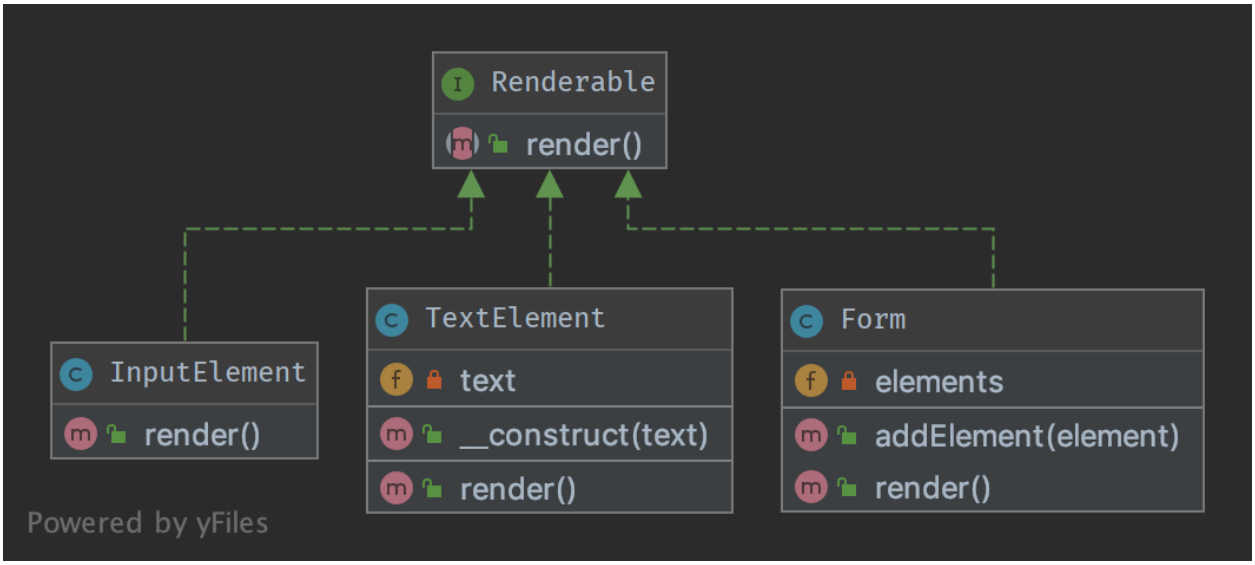
#### Предназначение

Да се третира група обекти по същия начин като единичен екземпляр на обекта.

### Примери

- екземпляр на клас на форма обработва всички негови елементи като единичен екземпляр, когато се извика `render()`, той впоследствие преминава през всички свои дъщерни елементи и извиква `render()` върху тях

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Renderable.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

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

Form.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Composite;

/**
```

(continues on next page)

(continued from previous page)

```php
 * 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 representation
     * of the form.
     *
     * from the outside, one will not see this and the form will act like a single object instance
     */
    public function render(): string
    {
        $formCode = '<form>';

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

        return $formCode . '</form>';
    }

    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
{
    public function __construct(private string $text)
    {
    }

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

### Тест

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>',
```

(continues on next page)

(continued from previous page)

```
            $form->render()
        );
    }
}
```

### 1.2.4 Data Mapper

#### Предназначение

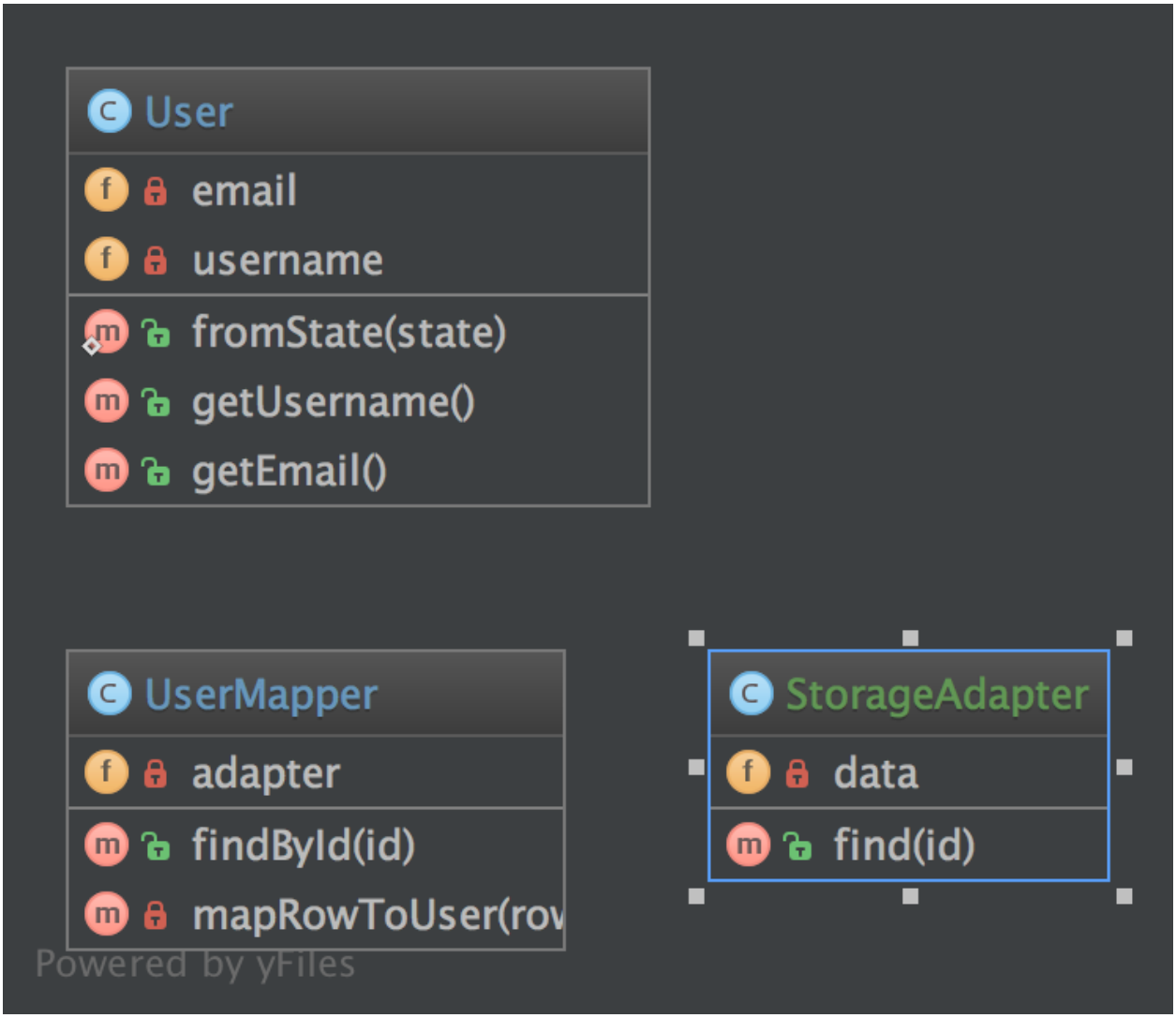
Data Mapper е слой за достъп до данни, който извършва двупосочен трансфер на данни между постоянното хранилище на данни (често релационна база данни) и представяне на данни в паметта (домейн слой). Целта на шаблона е да запази представянето в паметта и постоянното съхранение на данни независимо един от друг и самия картограф (mapper) на данни. Слоят се състои от един или повече картографи (mapper) (или обекти за достъп до данни), извършващи прехвърлянето на данни. Изпълнения на Mapper се различават по обхват. Общите картографи (mappers) ще обработват много различни типове обект на домейн, специалните картографи (mappers) ще работят с един или няколко.

Ключовата точка на този шаблон е, за разлика от Active Record, моделът на данни следва Принципа на единична отговорност.

#### Примери

- DB Object Relational Mapper (ORM): Doctrine2 използва DAO, наречен „EntityRepository“

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

User.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper;

class User
{
    public static function fromState(array $state): User
    {
```

(continues on next page)

(continued from previous page)

```
        // validate state before accessing keys!

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

    public function __construct(private string $username, private string $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
{
    public function __construct(private StorageAdapter $adapter)
    {
    }

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

        if ($result === null) {
```

(continues on next page)

(continued from previous page)

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

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

        return null;
    }
}
```

### Тест

Tests/DataMapperTest.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\DataMapper\Tests;

use InvalidArgumentException;
use DesignPatterns\Structural\DataMapper\StorageAdapter;
```

(continues on next page)

(continued from previous page)

```php
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' => 'someone', 'email' => 'someone@example.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);
    }
}
```

### 1.2.5 Декоратор

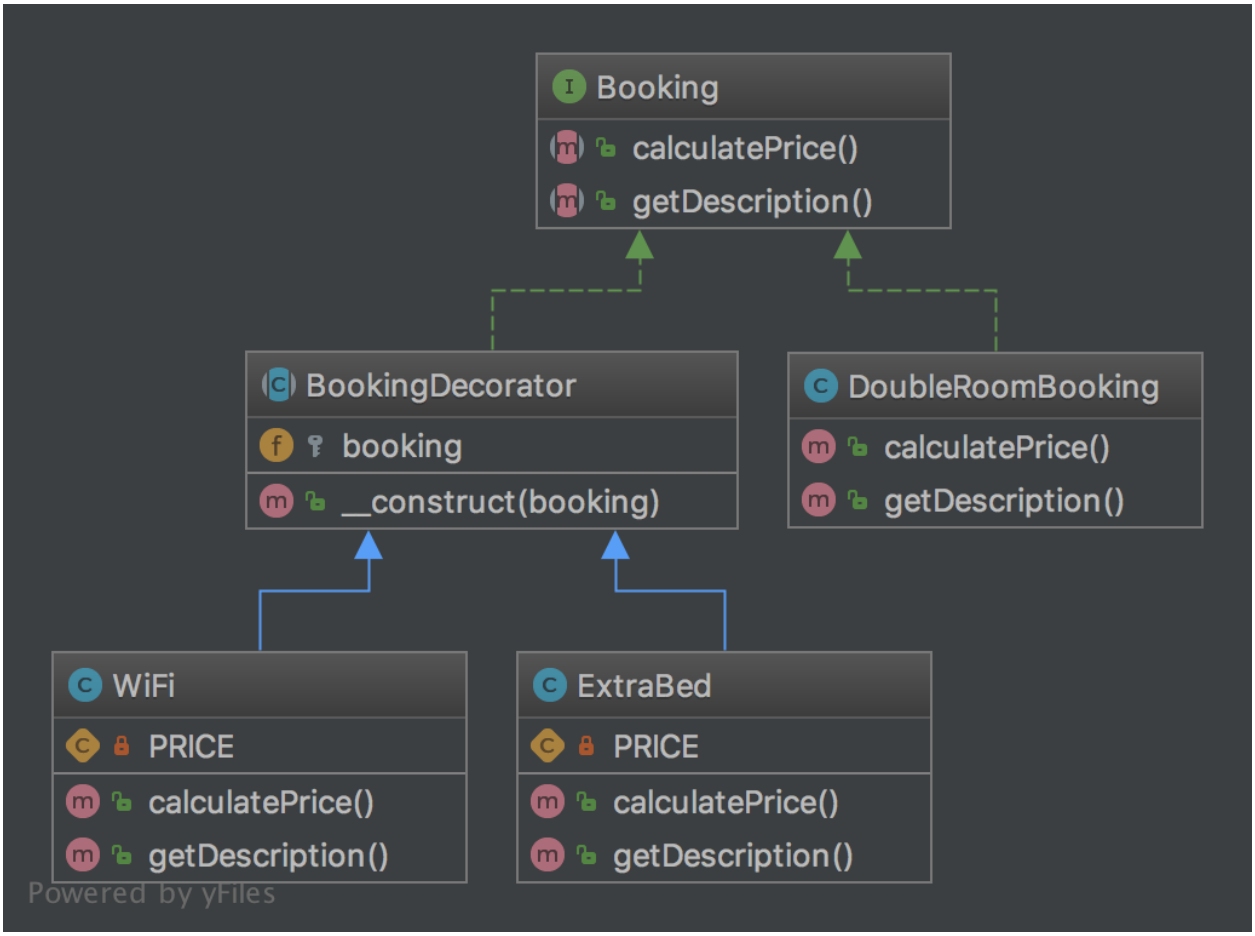
#### Предназначение

Да добавя динамично нова функционалност към екземплярите на класа.

#### Примери

- Слой на уеб услугата: Декоратори JSON и XML за REST услуга (web service) (в този случай, само един от тях трябва да бъде разрешен, разбира се)

### UML Диаграма




### Код

Можете също да намерите този код в 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
{
    public function __construct(protected 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
    {
```

(continues on next page)

(continued from previous page)

```
        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';
    }
}
```

### Тест

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());
    }
```

(continues on next page)

(continued from previous page)

```
21
22      public function testCanCalculatePriceForDoubleRoomBookingWithWiFi()
23      {
24          $booking = new DoubleRoomBooking();
25          $booking = new WiFi($booking);
26
27          $this->assertSame(42, $booking->calculatePrice());
28          $this->assertSame('double room with wifi', $booking->getDescription());
29      }
30
31      public function testCanCalculatePriceForDoubleRoomBookingWithWiFiAndExtraBed()
32      {
33          $booking = new DoubleRoomBooking();
34          $booking = new WiFi($booking);
35          $booking = new ExtraBed($booking);
36
37          $this->assertSame(72, $booking->calculatePrice());
38          $this->assertSame('double room with wifi with extra bed', $booking->
    ↪getDescription());
39      }
40  }
```

### 1.2.6 Инжектиране на зависимости

#### Предназначение

Да приложите свободно свързана архитектура, за да получите по-добре тестваем, поддържаем и разширяем код.

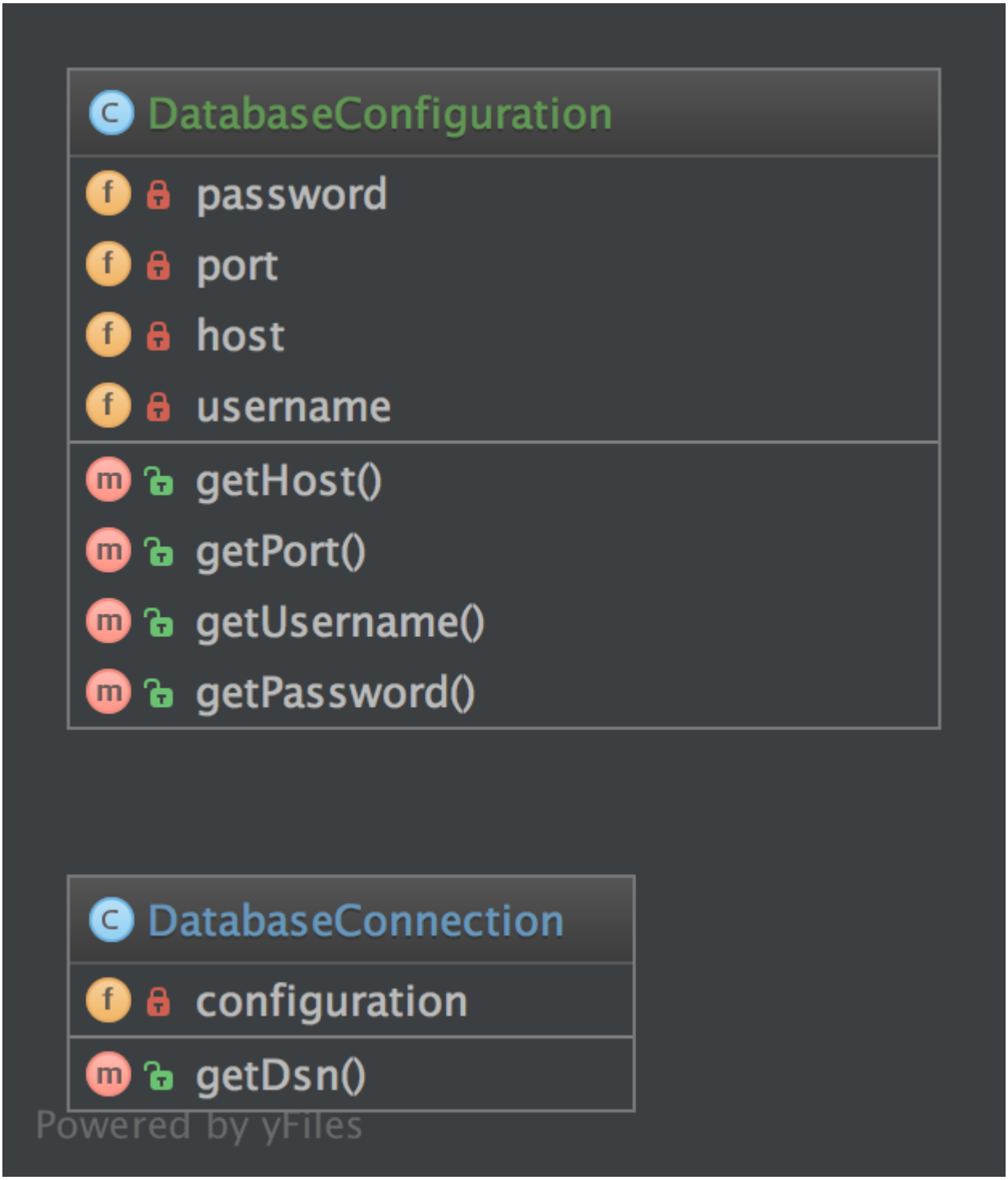
#### Използване

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

#### Примери

- Doctrine2 ORM използва инжекция на зависимост, напр. за конфигурация, която се инжектира в обект `Connection`. За целите на тестването, можете лесно да създадете фалшив обект от конфигурацията и да го инжектирате в обекта `Connection`
- много фреймуърки вече имат контейнери за DI, които създават обекти чрез конфигурационен масив и ги инжектират там, където е необходимо (т.е. в контролери)

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

DatabaseConfiguration.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConfiguration
{
    public function __construct(
        private string $host,
        private int $port,
        private string $username,
        private string $password
    ) {
    }

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

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

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

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

DatabaseConnection.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConnection
{
    public function __construct(private DatabaseConfiguration $configuration)
```

(continues on next page)

(continued from previous page)

```
    {
    }

    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()
        );
    }
}
```

### Тест

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, 'user', '1234');
        $connection = new DatabaseConnection($config);

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

### 1.2.7 Фасада

#### Предназначение

Основната цел на фасаден шаблон не е да не ви се налага да четете ръководството на сложен API. Това е само страничен ефект. Първата цел е да се намали връзката и да се следва закона на Деметра (Law of Demeter).

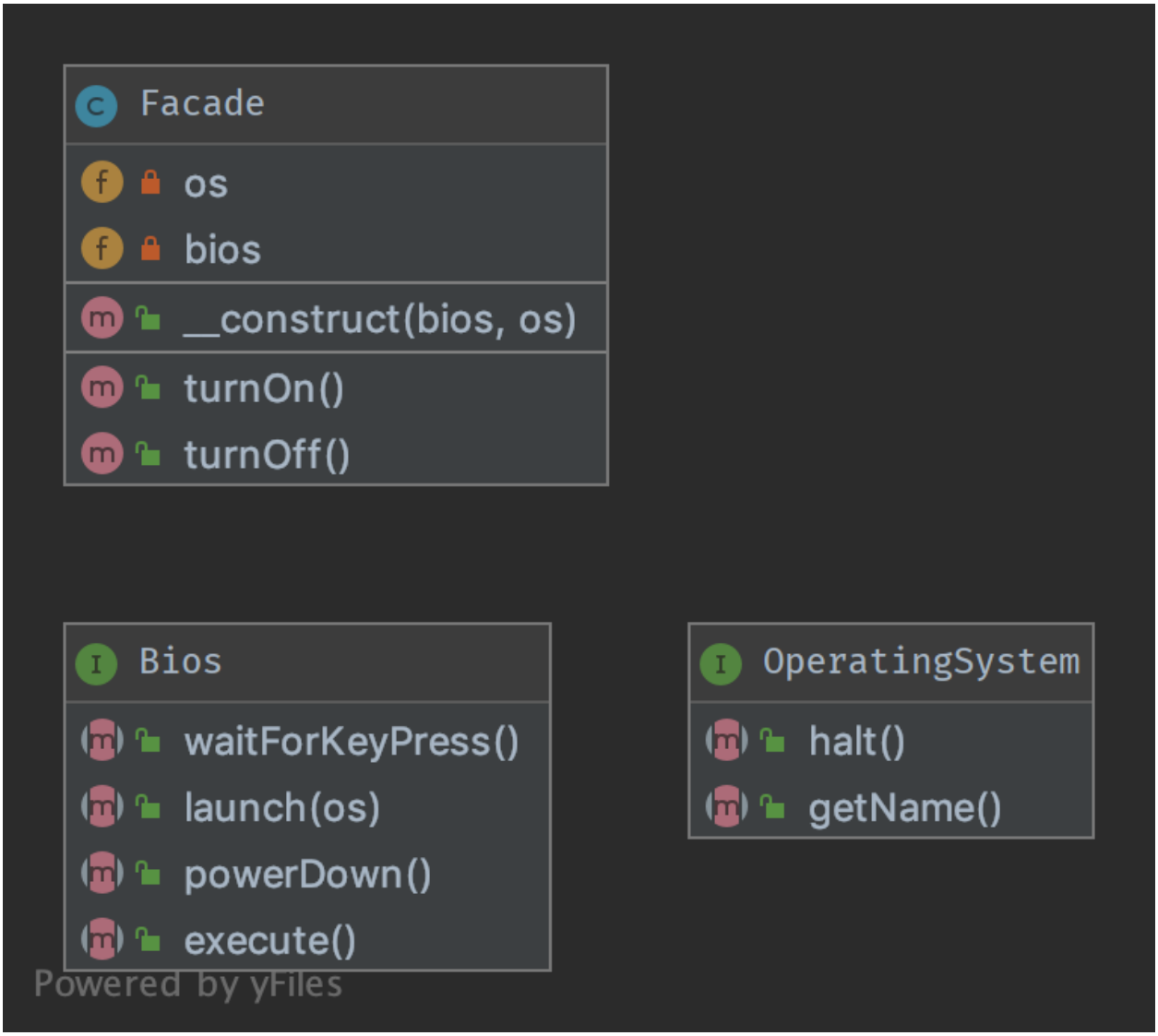
Фасадата има за цел да отдели клиент и подсистема чрез вграждане на много (но понякога само един) интерфейс и, разбира се, да намали сложността.

- Фасадата не ви забранява достъпа до подсистемата
- Можете (трябва) да имате множество фасади за една подсистема

Ето защо добрата фасада няма `new` в себе си. Ако има множество творения за всеки метод, това не е Facade, това е Builder или [Abstract|Static|Simple] Factory [Method].

Най-добрата фасада няма `new` и конструктор с interface-type-hinted параметри. Ако се нуждаете от създаване на нови копия, използвайте Factory като аргумент.

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Facade.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Facade;

class Facade
{
    public function __construct(private Bios $bios, private OperatingSystem $os)
```

(continues on next page)

(continued from previous page)

```
    {
    }

    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();
}
```

#### Тест

Tests/FacadeTest.php

```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());
    }
}
```

### 1.2.8 Fluent Interface

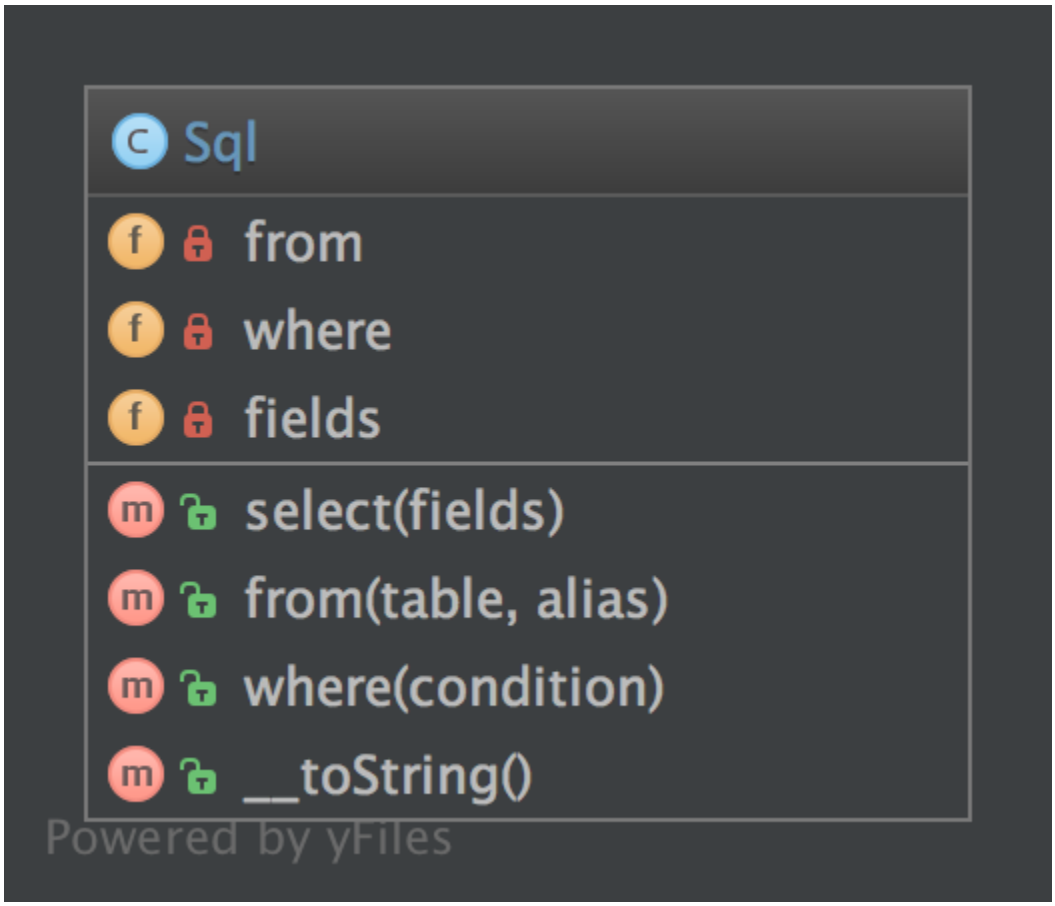
#### Предназначение

Да пишете лесно четим код като изреченията на естествен език (като английски).

### Примери

- QueryBuilder на Doctrine2 работи подобно на този примерен клас по-долу
- PHPUnit използва fluent интерфейси за изграждане на фалшиви обекти

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Sql.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\FluentInterface;

class Sql implements \Stringable
{
    private array $fields = [];
```

(continues on next page)

(continued from previous page)

```
    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)
        );
    }
}
```

### Тест

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()
```

(continues on next page)

(continued from previous page)

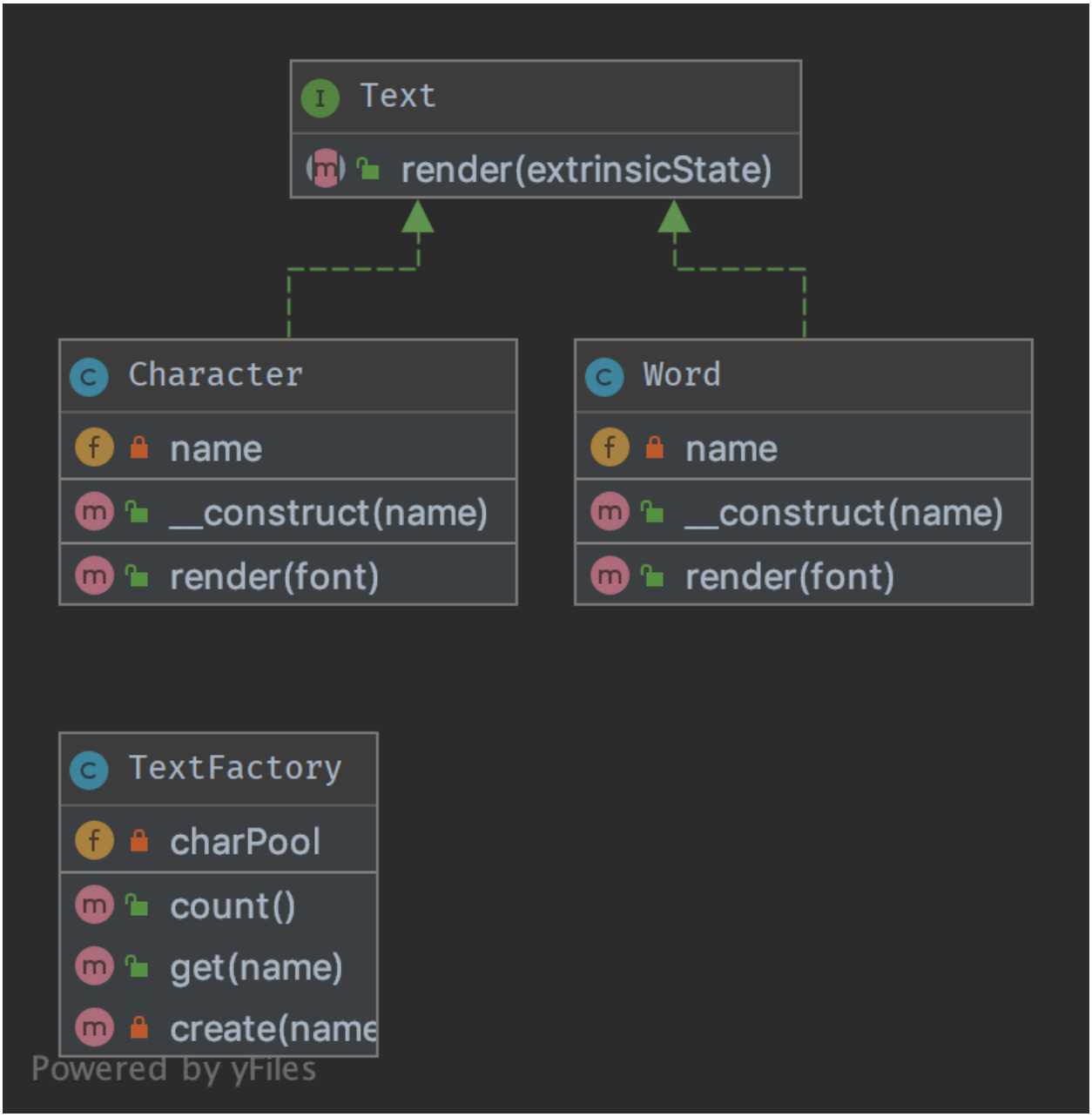
```
    {
        $query = (new Sql())
                ->select(['foo', 'bar'])
                ->from('foobar', 'f')
                ->where('f.bar = ?');

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

### 1.2.9 Миниобект

#### Предназначение

За да сведе до минимум използването на памет, миниобект споделя възможно най-много памет с подобни обекти. Това е необходимо, когато се използва голямо количество обекти, които не се различават особено по състояние. Често срещана практика е да се задържа състояние във външни структури от данни и да се предават на миниобекта, когато е необходимо.

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Text.php

```
<?php

declare(strict_types=1);

```

(continues on next page)

(continued from previous page)

```php
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
<?php

namespace DesignPatterns\Structural\Flyweight;

class Word implements Text
{
    public function __construct(private string $name)
    {
    }

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

Character.php

```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 code.
     */
    public function __construct(private string $name)
    {
    }

    public function render(string $extrinsicState): string
```

(continues on next page)

(continued from previous page)

```
    {
        // Clients supply the context-dependent information that the flyweight needs to draw itself
        // For flyweights representing characters, extrinsic state usually contains e.g. the font.

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

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($name);
        }
        return new Word($name);
    }

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

(continues on next page)

(continued from previous page)

```
}
```

### Тест

Tests/FlyweightTest.php

```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);
    }
```

(continues on next page)

(continued from previous page)

```
44 }
```

### 1.2.10 Пълномощно

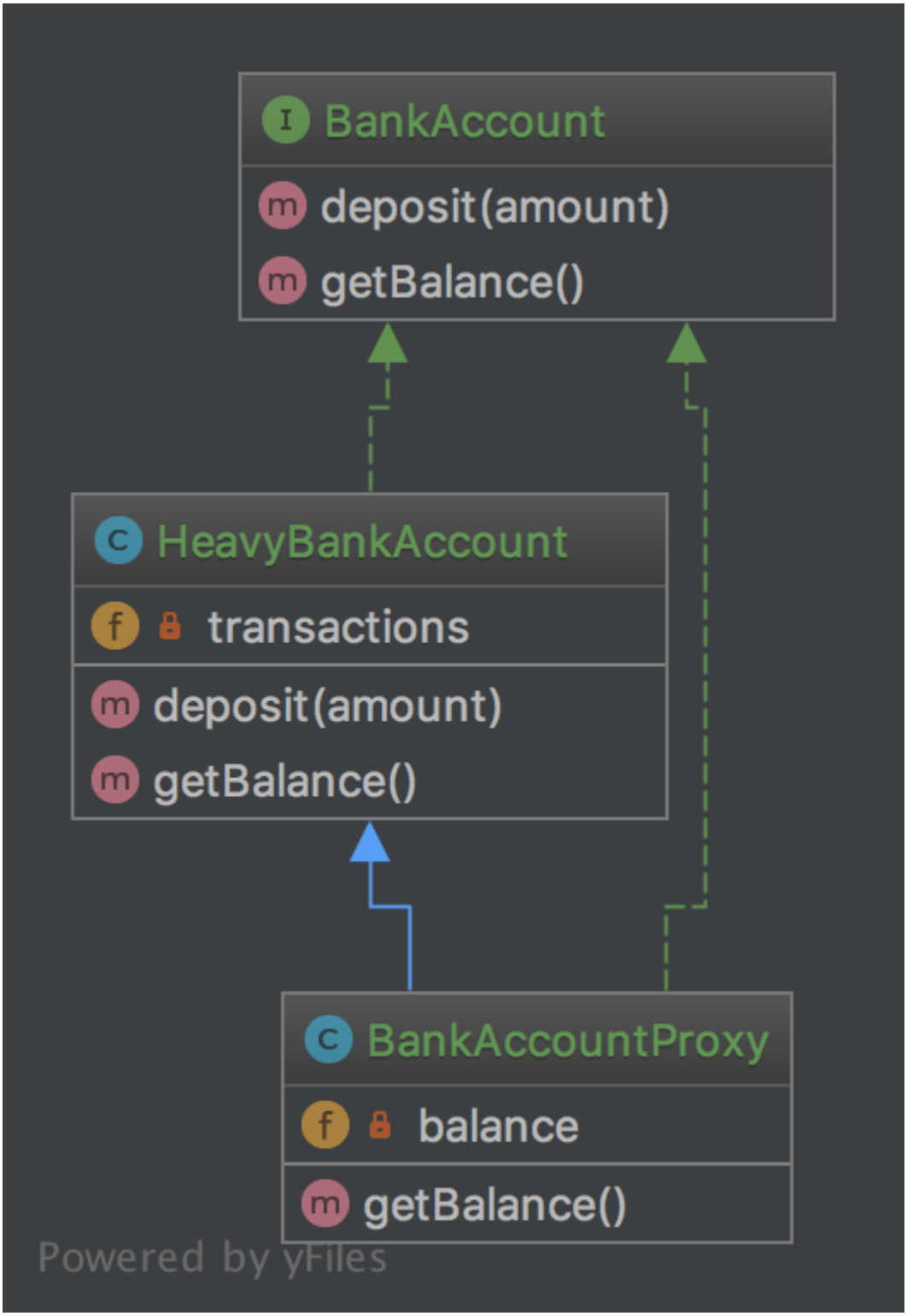
#### Предназначение

Да се свързва с всичко, което е скъпо или невъзможно да се дублира.

#### Примери

- Doctrine2 използва прокси, за да внедри в тях магия на фреймуърка (напр. Мързелива инициализация), докато потребителят все още работи със собствените си класове на субекти и никога няма да използва или докосва прокситата

### UML Диаграма



### Код

Можете също да намерите този код в 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 array_sum($this->transactions);
    }
}
```

BankAccountProxy.php

```php
<?php

declare(strict_types=1);
```

(continues on next page)

(continued from previous page)

```php

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;
    }
}
```

### Тест

ProxyTest.php

```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());
```

(continues on next page)

(continued from previous page)

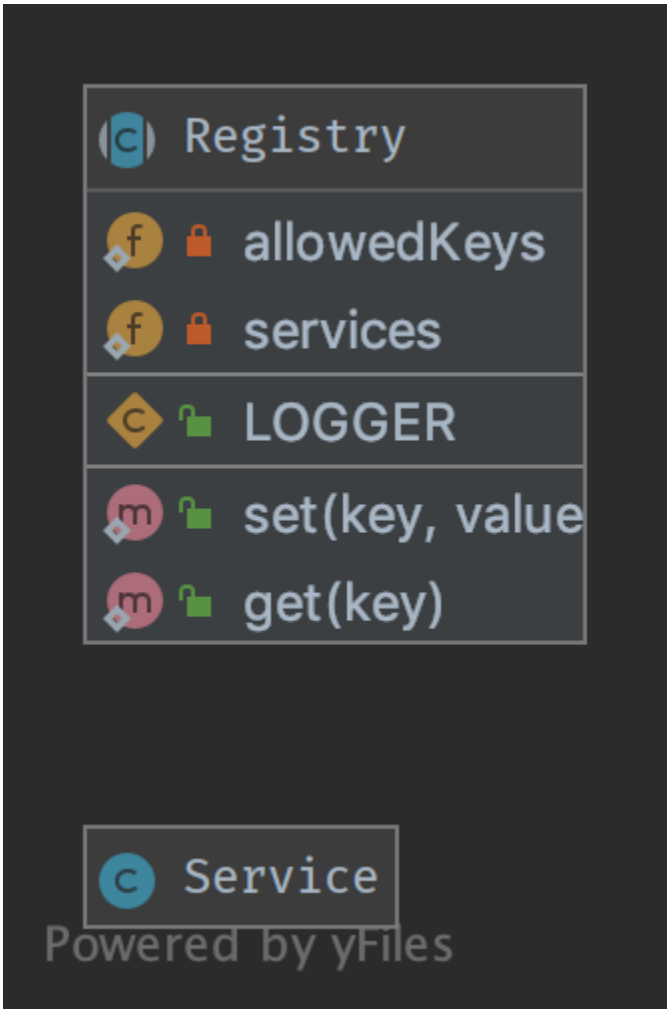
```
25      }
26  }
```

### 1.2.11 Rigistry

#### Предназначение

За да се приложи централно хранилище за обекти, често използвани в цялото приложение, обикновено се реализира с помощта на абстрактен клас само със статични методи (или с помощта на Singleton модел). Не забравяйте, че това въвежда глобално състояние, което трябва да се избягва по всяко време! Вместо това го внедрете с помощта на инжекция на зависимост!

#### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Registry.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Structural\Registry;

use InvalidArgumentException;

abstract class Registry
{
    public const LOGGER = 'logger';

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

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

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

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

    final 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
<?php

```

(continues on next page)

(continued from previous page)

```
namespace DesignPatterns\Structural\Registry;

class Service
{
}
```

### Тест

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\TestCase;

class RegistryTest extends TestCase
{
    private Service $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
     *
```

(continues on next page)

(continued from previous page)

```
40          * @runInSeparateProcess
41          */
42         public function testThrowsExceptionWhenTryingToGetNotSetKey()
43         {
44             $this->expectException(InvalidArgumentException::class);
45
46             Registry::get(Registry::LOGGER);
47         }
48     }
```

## 1.3 Поведенчески

В софтуерното инженерство, поведенчески шаблони за дизайн са шаблони, които идентифицират общите комуникационни модели между обектите и реализират тези модели. По този начин тези модели увеличават гъвкавостта при осъществяване на тази комуникация.

### 1.3.1 Верига отговорности

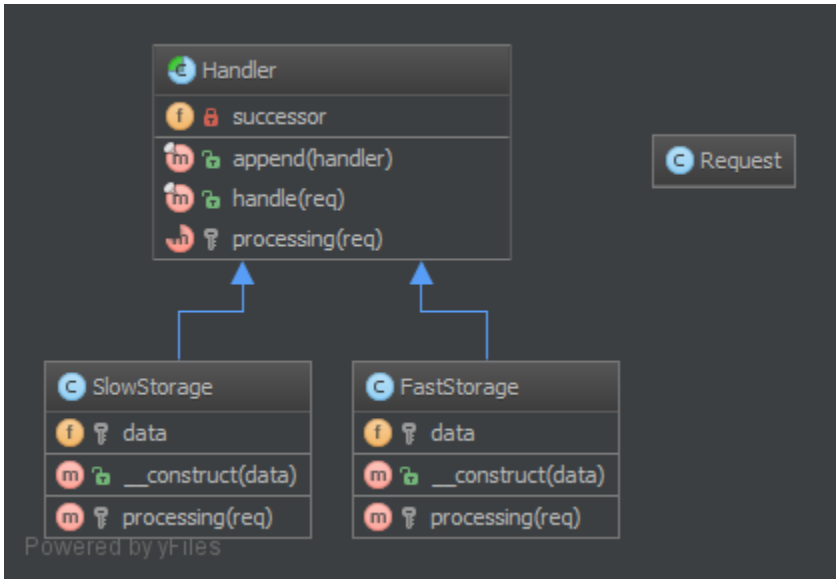
#### Purpose

Изграждане на верига от обекти за обработка на повикване в последователен ред. Ако един обект не може да обработи повикване, той делегира повикването на следващия по веригата и т.н.

#### Examples

- logging framework, където всеки елемент на веригата решава автономно какво да прави със съобщение
- Спам филтър
- Кеширане: първият обект е екземпляр от напр. Memcached интерфейс, ако този „пропусне“, той делегира повикването на интерфейса на базата данни

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Handler.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

use Psr\Http\Message\RequestInterface;

abstract class Handler
{
    public function __construct(private ?Handler $successor = null)
    {
    }

    /**
     * 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);
        }
```

(continues on next page)

(continued from previous page)

```
27
28          return $processed;
29      }
30
31      abstract protected function processing(RequestInterface $request): ?string;
32  }
```

Responsible/FastStorage.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
6
7   use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
8   use Psr\Http\Message\RequestInterface;
9
10  class HttpInMemoryCacheHandler extends Handler
11  {
12      public function __construct(private array $data, ?Handler $successor = null)
13      {
14          parent::__construct($successor);
15      }
16
17      protected function processing(RequestInterface $request): ?string
18      {
19          $key = sprintf(
20              '%s?%s',
21              $request->getUri()->getPath(),
22              $request->getUri()->getQuery()
23          );
24
25          if ($request->getMethod() == 'GET' && isset($this->data[$key])) {
26              return $this->data[$key];
27          }
28
29          return null;
30      }
31  }
```

Responsible/SlowStorage.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
6
7   use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
8   use Psr\Http\Message\RequestInterface;
9
```

(continues on next page)

(continued from previous page)

```
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-memory) DB for the results

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

### Тест

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);
```

(continues on next page)

(continued from previous page)

```
36
37          $this->assertSame('Hello In Memory!', $this->chain->handle($request));
38      }
39
40      public function testCanRequestKeyInSlowStorage()
41      {
42          $uri = $this->createMock(UriInterface::class);
43          $uri->method('getPath')->willReturn('/foo/baz');
44          $uri->method('getQuery')->willReturn('');
45
46          $request = $this->createMock(RequestInterface::class);
47          $request->method('getMethod')
48              ->willReturn('GET');
49          $request->method('getUri')->willReturn($uri);
50
51          $this->assertSame('Hello World!', $this->chain->handle($request));
52      }
53  }
```

### 1.3.2 Команда

#### Предназначение

За капсулиране на извикване и отделяне.

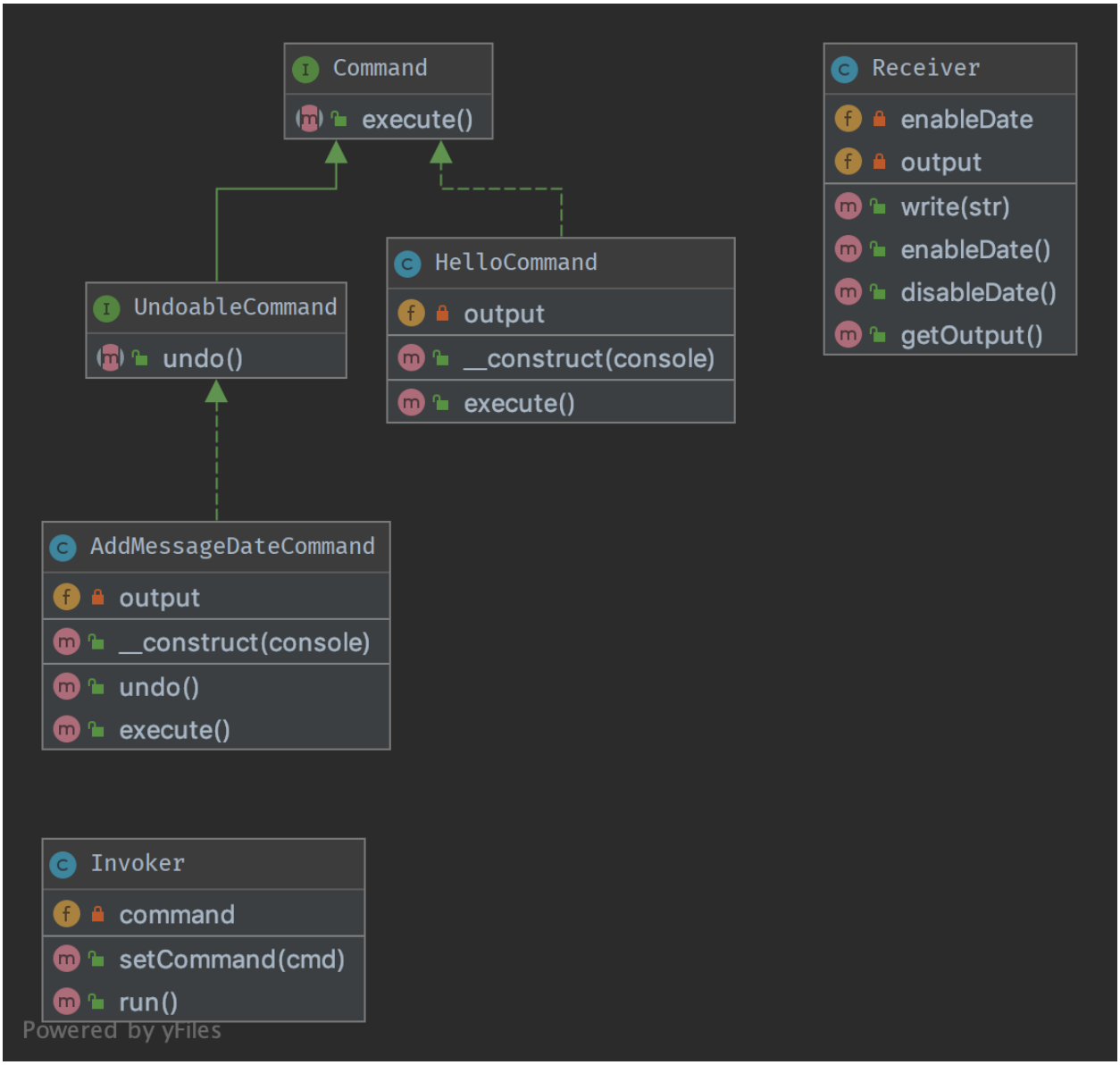
Ние имаме Повикващ и Получател. Този модел използва „Command“, за да делегира извикването на метода срещу Получателя и представя същия метод „execute“. Следователно Повикващ просто знае да извика „execute“, за да обработи командата на клиента. Получателят е отделен от Повикващия.

Вторият аспект на този модел е undo(), който отменя метода execute(). Command може да се агрегира и комбинира по-сложни команди с минимално copy&paste и разчитане на композиция над наследяване.

#### Примери

- A text editor : all events are commands which can be undone, stacked and saved.
- големите CLI инструменти използват подкоманди, за да разпределят различни задачи и да ги опаковат в „модули“, като всеки от тях може да бъде изпълнен с командния модел (напр. vagrant)

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Command.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Command;

interface Command
```

(continues on next page)

(continued from previous page)

```
{
    /**
     * 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
{
    /**
     * Each concrete command is built with different receivers.
     * There can be one, many or completely no receivers, but there can be other␣
↪commands in the parameters
     */
    public function __construct(private Receiver $output)
    {
    }

    /**
     * execute and output "Hello World".
     */
    public function execute()
    {
```

(continues on next page)

(continued from previous page)

```
        // 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
{
    /**
     * Each concrete command is built with different receivers.
     * There can be one, many or completely no receivers, but there can be other commands in the parameters.
     */
    public function __construct(private Receiver $output)
    {
    }

    /**
     * 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
<?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
<?php

```

(continues on next page)

(continued from previous page)

```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();
    }
}
```

### Тест

Tests/CommandTest.php

```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();
```

(continues on next page)

(continued from previous page)

```
18
19          $invoker->setCommand(new HelloCommand($receiver));
20          $invoker->run();
21          $this->assertSame('Hello World', $receiver->getOutput());
22      }
23  }
```

Tests/UndoableCommandTest.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\Command\Tests;
6
7   use DesignPatterns\Behavioral\Command\AddMessageDateCommand;
8   use DesignPatterns\Behavioral\Command\HelloCommand;
9   use DesignPatterns\Behavioral\Command\Invoker;
10  use DesignPatterns\Behavioral\Command\Receiver;
11  use PHPUnit\Framework\TestCase;
12
13  class UndoableCommandTest extends TestCase
14  {
15      public function testInvocation()
16      {
17          $invoker = new Invoker();
18          $receiver = new Receiver();
19
20          $invoker->setCommand(new HelloCommand($receiver));
21          $invoker->run();
22          $this->assertSame('Hello World', $receiver->getOutput());
23
24          $messageDateCommand = new AddMessageDateCommand($receiver);
25          $messageDateCommand->execute();
26
27          $invoker->run();
28          $this->assertSame("Hello World\nHello World [" . date('Y-m-d') . ']', $receiver->
    ↪getOutput());
29
30          $messageDateCommand->undo();
31
32          $invoker->run();
33          $this->assertSame("Hello World\nHello World [" . date('Y-m-d') . "]\nHello World
    ↪", $receiver->getOutput());
34      }
35  }
```

### 1.3.3 Interpreter

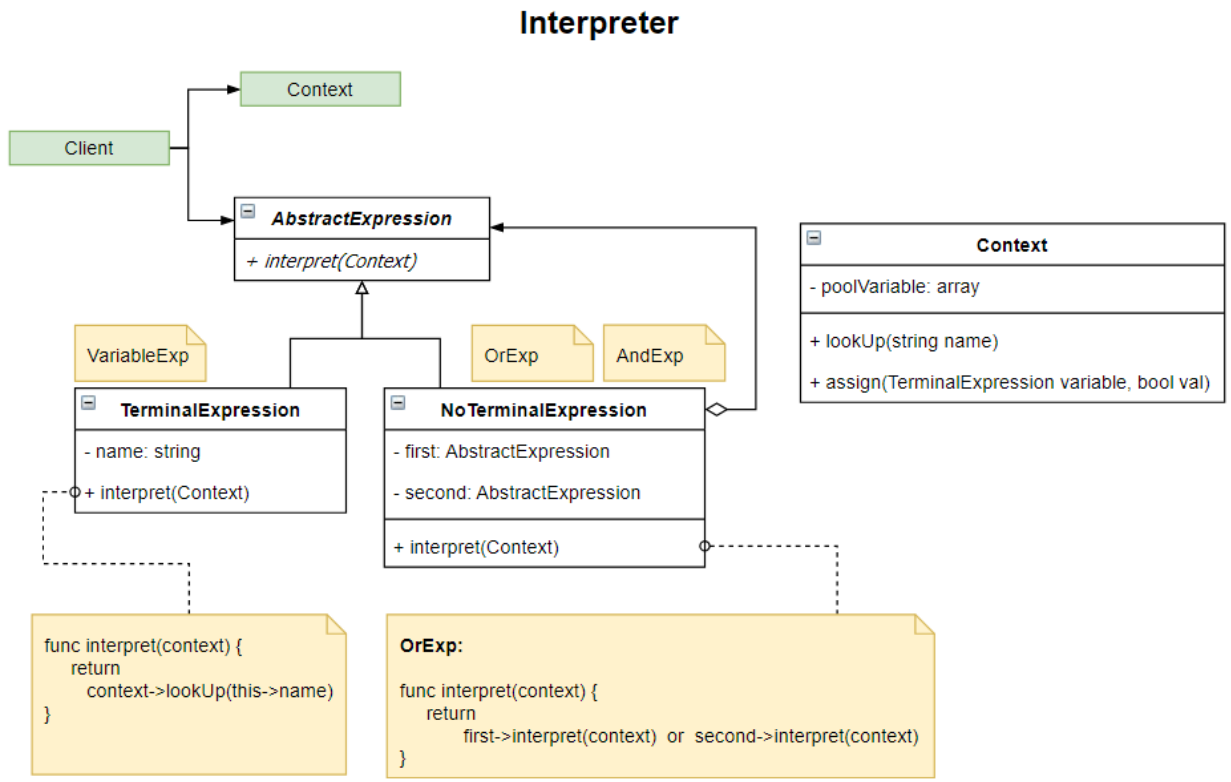
#### Purpose

For a given language, it defines the representation of its grammar as „No Terminal Expression“ and „Terminal Expression“, as well as an interpreter for the sentences of that language.

#### Examples

- An example of a binary logic interpreter, each definition is defined by its own class

#### UML Diagram




#### Code

You can also find this code on GitHub

AbstractExp.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter;

```

(continues on next page)

(continued from previous page)

```
abstract class AbstractExp
{
    abstract public function interpret(Context $context): bool;
}
```

Context.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter;

use Exception;

class Context
{
    private array $poolVariable;

    public function lookUp(string $name): bool
    {
        if (!key_exists($name, $this->poolVariable)) {
            throw new Exception("no exist variable: $name");
        }

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

    public function assign(VariableExp $variable, bool $val)
    {
        $this->poolVariable[$variable->getName()] = $val;
    }
}
```

VariableExp.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter;

/**
 * This TerminalExpression
 */
class VariableExp extends AbstractExp
{
    public function __construct(private string $name)
    {
    }

    public function interpret(Context $context): bool
```

(continues on next page)

(continued from previous page)

```
    {
        return $context->lookUp($this->name);
    }

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

AndExp.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter;

/**
 * This NoTerminalExpression
 */
class AndExp extends AbstractExp
{
    public function __construct(private AbstractExp $first, private AbstractExp $second)
    {
    }

    public function interpret(Context $context): bool
    {
        return $this->first->interpret($context) && $this->second->interpret($context);
    }
}
```

OrExp.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter;

/**
 * This NoTerminalExpression
 */
class OrExp extends AbstractExp
{
    public function __construct(private AbstractExp $first, private AbstractExp $second)
    {
    }

    public function interpret(Context $context): bool
    {
```

(continues on next page)

(continued from previous page)

```
        return $this->first->interpret($context) || $this->second->interpret($context);
    }
}
```

### Test

Tests/InterpreterTest.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Interpreter\Tests;

use DesignPatterns\Behavioral\Interpreter\AndExp;
use DesignPatterns\Behavioral\Interpreter\Context;
use DesignPatterns\Behavioral\Interpreter\OrExp;
use DesignPatterns\Behavioral\Interpreter\VariableExp;
use PHPUnit\Framework\TestCase;

class InterpreterTest extends TestCase
{
    private Context $context;
    private VariableExp $a;
    private VariableExp $b;
    private VariableExp $c;

    public function setUp(): void
    {
        $this->context = new Context();
        $this->a = new VariableExp('A');
        $this->b = new VariableExp('B');
        $this->c = new VariableExp('C');
    }

    public function testOr()
    {
        $this->context->assign($this->a, false);
        $this->context->assign($this->b, false);
        $this->context->assign($this->c, true);

        // A  B
        $exp1 = new OrExp($this->a, $this->b);
        $result1 = $exp1->interpret($this->context);

        $this->assertFalse($result1, 'A  B must false');

        // $exp1  C
        $exp2 = new OrExp($exp1, $this->c);
        $result2 = $exp2->interpret($this->context);

```

(continues on next page)

(continued from previous page)

```
44          $this->assertTrue($result2, '(A  B)  C must true');
45      }
46
47      public function testAnd()
48      {
49          $this->context->assign($this->a, true);
50          $this->context->assign($this->b, true);
51          $this->context->assign($this->c, false);
52
53          // A  B
54          $exp1 = new AndExp($this->a, $this->b);
55          $result1 = $exp1->interpret($this->context);
56
57          $this->assertTrue($result1, 'A  B must true');
58
59          // $exp1  C
60          $exp2 = new AndExp($exp1, $this->c);
61          $result2 = $exp2->interpret($this->context);
62
63          $this->assertFalse($result2, '(A  B)  C must false');
64      }
65  }
```

### 1.3.4 Итератор

#### Предназначение

За да направите обекта итеративен и да го направите да изглежда като колекция от обекти.

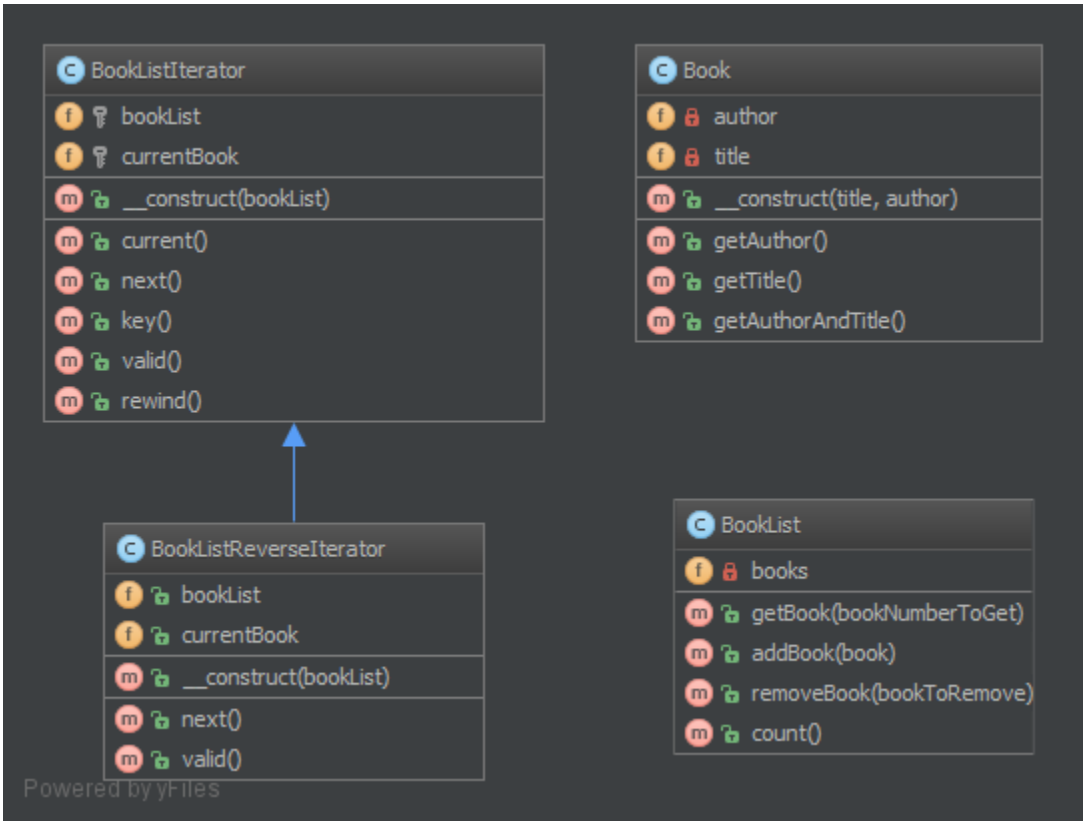
#### Примери

- за обработка на файл ред по ред, като просто прекарате всички редове (които имат представяне на обект) за файл (което, разбира се, също е обект)

### Забележка

Стандартната PHP библиотека (SPL) определя интерфейсен итератор, който е най-подходящ за това! Често бихте искали да внедрите и Countable интерфейса, за да позволите``count ($ object)`` на вашия итеративен обект

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Book.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    public function __construct(private string $title, private string $author)
    {
    }
```

(continues on next page)

(continued from previous page)

```
12
13      public function getAuthor(): string
14      {
15          return $this->author;
16      }
17
18      public function getTitle(): string
19      {
20          return $this->title;
21      }
22
23      public function getAuthorAndTitle(): string
24      {
25          return $this->getTitle() . ' by ' . $this->getAuthor();
26      }
27  }
```

BookList.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\Iterator;
6
7   use Countable;
8   use Iterator;
9
10  class BookList implements Countable, Iterator
11  {
12      /**
13       * @var Book[]
14       */
15      private array $books = [];
16      private int $currentIndex = 0;
17
18      public function addBook(Book $book)
19      {
20          $this->books[] = $book;
21      }
22
23      public function removeBook(Book $bookToRemove)
24      {
25          foreach ($this->books as $key => $book) {
26              if ($book->getAuthorAndTitle() === $bookToRemove->getAuthorAndTitle()) {
27                  unset($this->books[$key]);
28              }
29          }
30
31          $this->books = array_values($this->books);
32      }
33
```

(continues on next page)

(continued from previous page)

```
    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(): void
    {
        $this->currentIndex++;
    }

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

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

### Тест

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'));
```

(continues on next page)

(continued from previous page)

```
        $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()
    {
```

(continues on next page)

(continued from previous page)

```
69          $book = new Book('Clean Code', 'Robert C. Martin');
70
71          $bookList = new BookList();
72          $bookList->addBook($book);
73          $bookList->removeBook($book);
74
75          $this->assertCount(0, $bookList);
76      }
77  }
```

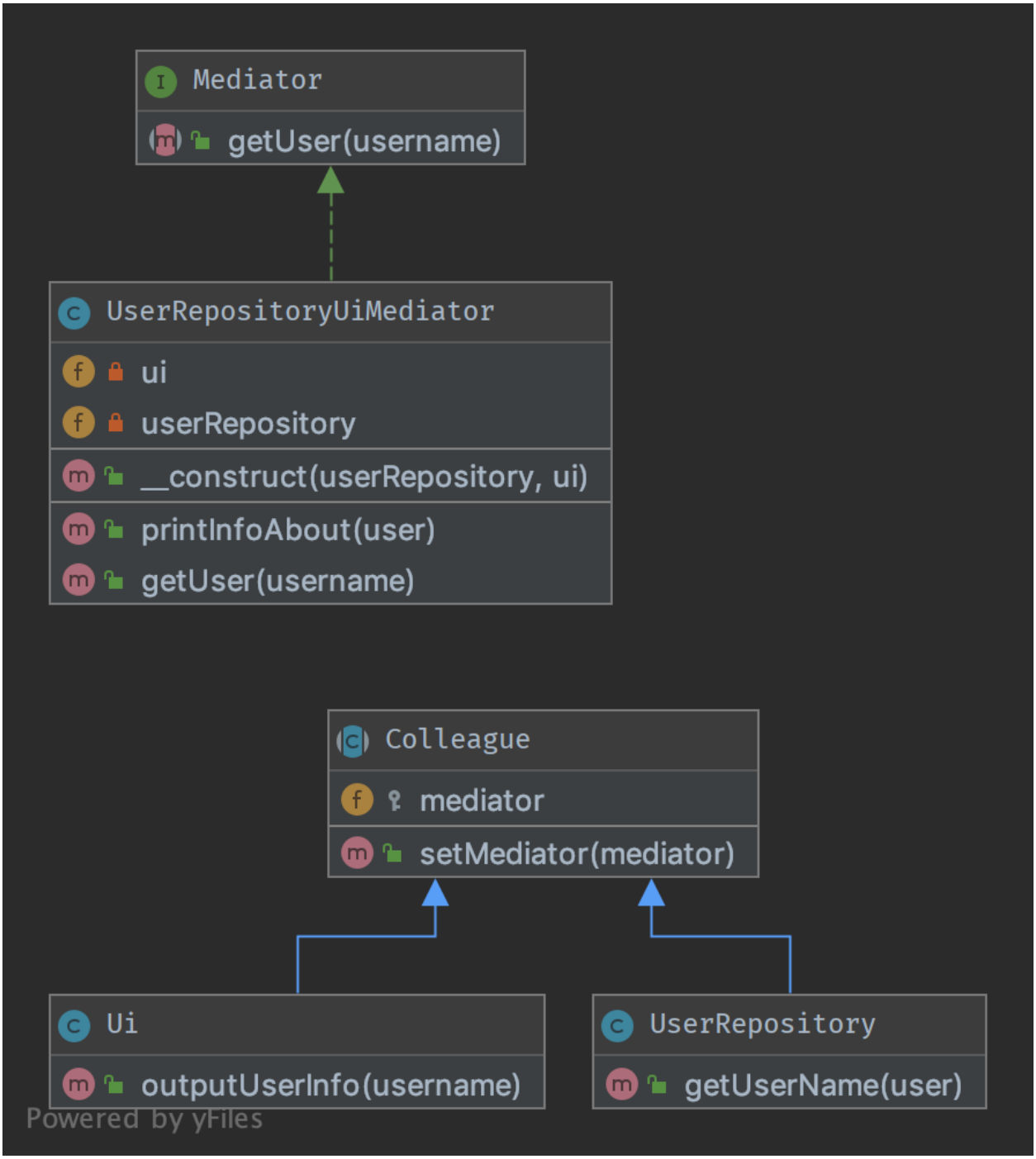
### 1.3.5 Посредник

#### Предназначение

Този модел осигурява лесен начин за отделяне на много компоненти, работещи заедно. Това е добра алтернатива на Observer, АКО имате централен обект, като контролер (но не в смисъла на MVC).

Всички компоненти (наречени колега) са свързани само с интерфейса на Посредника и това е хубаво нещо, защото в ООП един добър приятел е по-добър от много. Това е ключовата характеристика на този шаблон.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Mediator.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

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

Colleague.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

abstract class Colleague
{
    protected Mediator $mediator;

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

Ui.php

```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
<?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
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Mediator;

class UserRepositoryUiMediator implements Mediator
{
    public function __construct(private UserRepository $userRepository, private 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);
    }
}
```

### Тест

Tests/MediatorTest.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Ui;
```

(continues on next page)

(continued from previous page)

```
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');
    }
}
```

### 1.3.6 Спомен

#### Предназначение

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

Шаблон е реализиран с три обекта: Originator (Оригинатор), а Caretaker (Пазител) и Memento (Спомен).

Memento - обект, който * съдържа конкретна уникална снимка на състоянието * на всеки обект или ресурс: низ, число, масив, екземпляр на клас и така нататък. Уникалността в този случай не означава забрана за съществуването на подобни състояния в различни моментни снимки. Това означава, че състояние може да бъде извлечено като независим клон. Всеки обект, съхраняван в Memento, трябва да бъде * пълно копие на оригиналния обект, а не препратка * към оригиналния обект. Обектът Memento е „непрозрачен обект“ (обектът, който никой не може или не трябва да променя).

Originator - това е обект, който съдържа * действителното състояние на външен обект е строго определен тип *. Originator е в състояние да създаде уникално копие на това състояние и да го върне увито в спомен. Originator не знае историята на промените. Можете да зададете конкретно състояние на Originator отвън, което ще се счита за действително. Originator трябва да се увери, че дадено състояние съответства на разрешения тип обект. Originator може (но не трябва) да има някакви методи, но те * не могат да правят промени в състоянието на запазения обект *.

Caretaker * контролира историята на състояния *. Той може да прави промени в обект; взима решение за запазване на състоянието на външен обект в Originator; поиска моментна снимка на текущото състояние от Originator; или зададе състоянието на Originator еквивалентно на някаква снимка от историята.

### Примери

- To seed генератор на псевдослучайни числа
- Състоянието в краен автомат
- Контрол за междинни състояния на ORM Model преди запазване

### UML Диаграма

### Код

Можете също да намерите този код в GitHub

Memento.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

class Memento
{
    public function __construct(private State $state)
    {
    }

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

State.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

use InvalidArgumentException;

class State implements \Stringable
{
    public const STATE_CREATED = 'created';
    public const STATE_OPENED = 'opened';
    public const STATE_ASSIGNED = 'assigned';
    public 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)
```

(continues on next page)

(continued from previous page)

```
    {
        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);
```

(continues on next page)

(continued from previous page)

```
    }

    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;
    }
}
```

### Тест

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
```

(continues on next page)

(continued from previous page)

```
29          $ticket->restoreFromMemento($memento);
30
31          $this->assertSame(State::STATE_OPENED, (string) $ticket->getState());
32          $this->assertNotSame($openedState, $ticket->getState());
33      }
34  }
```

### 1.3.7 Празен обект

#### Предназначение

Null Object не е шаблон на GoF дизайн, а схема, която се появява достатъчно често, за да се счита за шаблон. Той има следните предимства:

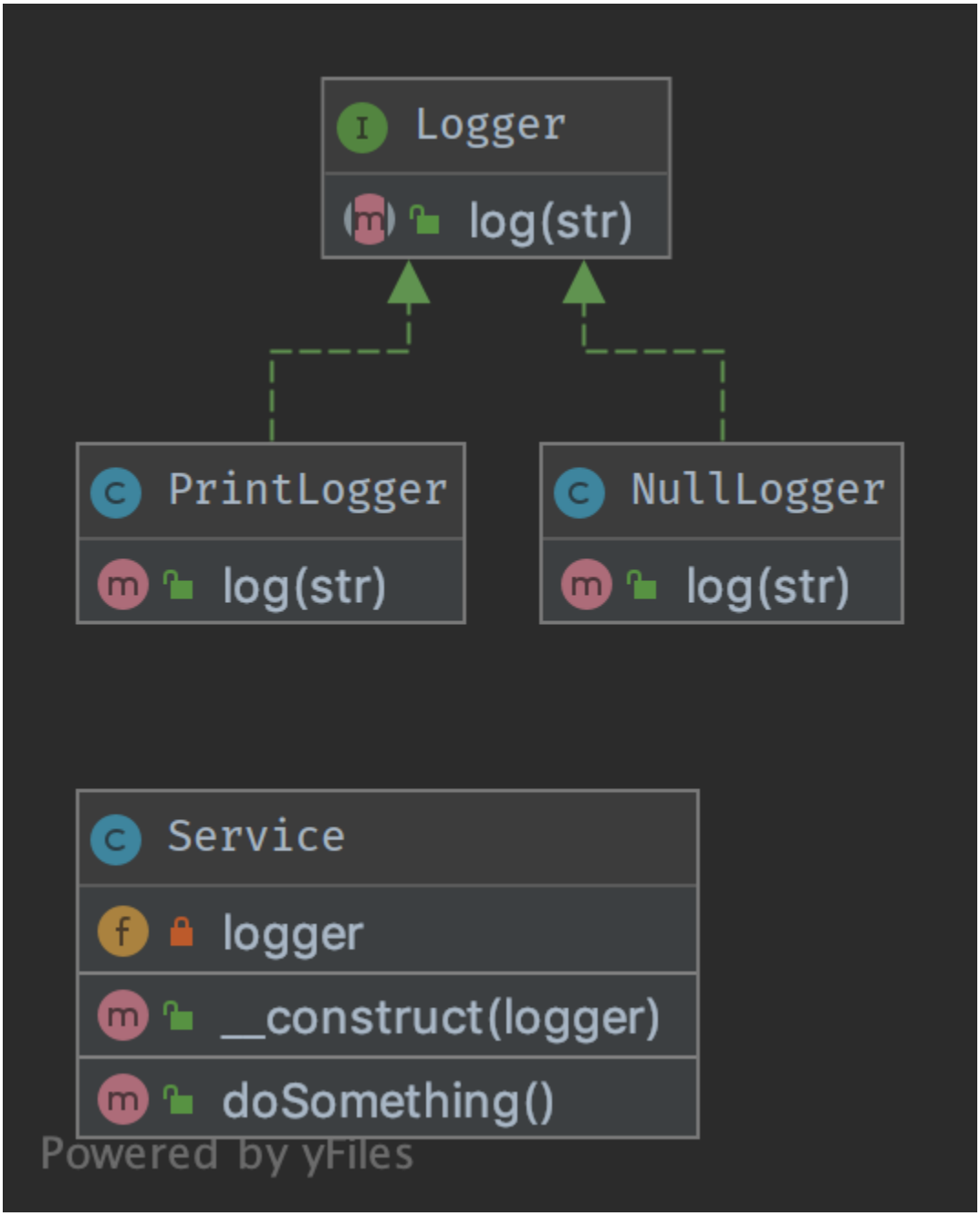
- Клиентският код е опростен
- Намалява шанса за изключения от нулев указател
- По-малко условия изискват по-малко тестови случаи

Методите, които връщат обект или null, вместо това трябва да върнат обект или `NullObject`. `NullObject`s опростяване на кодов шаблон, като `if (!is_null($obj)) { $obj->callSomething(); }` to just `$obj->callSomething();` чрез премахване на условната проверка в кода.

#### Примери

- Null logger или null output за запазване на стандартен начин на взаимодействие между обектите, дори ако не трябва да прави нищо
- нулев манипулатор в шаблон на верига отговорности
- нулева команда в Command шаблон

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Service.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    public function __construct(private Logger $logger)
    {
    }

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

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
```

(continues on next page)

(continued from previous page)

```
{
    public function log(string $str)
    {
        echo $str;
    }
}
```

NullLogger.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\NullObject;

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

### Тест

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\
```

(continues on next page)

(continued from previous page)

```
↪Service::doSomething');
25          $service->doSomething();
26      }
27  }
```

### 1.3.8 Наблюдател

#### Предназначение

За да се приложи поведение за публикуване/абониране на обект, всеки път, когато обект „Subject“ промени състоянието си, прикачените „Observers“ ще бъдат уведомени. Използва се за съкращаване на количеството на свързани обекти и вместо това използва разхлабено свързване.

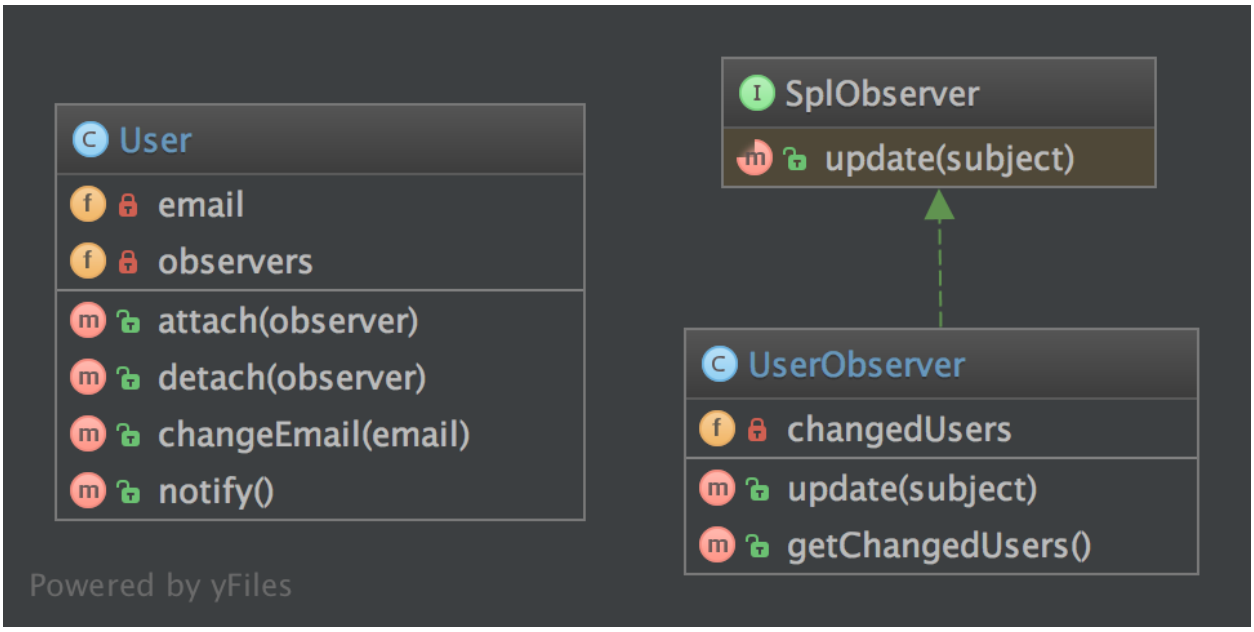
#### Примери

- се наблюдава система за опашки от съобщения (message queue system), която показва напредъка на заданието в GUI

#### Забележка

PHP вече дефинира два интерфейса, които могат да помогнат за реализирането на този шаблон: SplObserver и SplSubject.

#### UML Диаграма



### Код

Можете също да намерите този код в GitHub

User.php

```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 and sends notifications to
 * them in case changes are made on the User object
 */
class User implements SplSubject
{
    private SplObjectStorage $observers;
    private $email;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
}
```

UserObserver.php

```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): void
    {
        $this->changedUsers[] = clone $subject;
    }

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

### Тест

Tests/ObserverTest.php

```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;

```

(continues on next page)

(continued from previous page)

```php
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());
    }
}
```

### 1.3.9 Спецификация

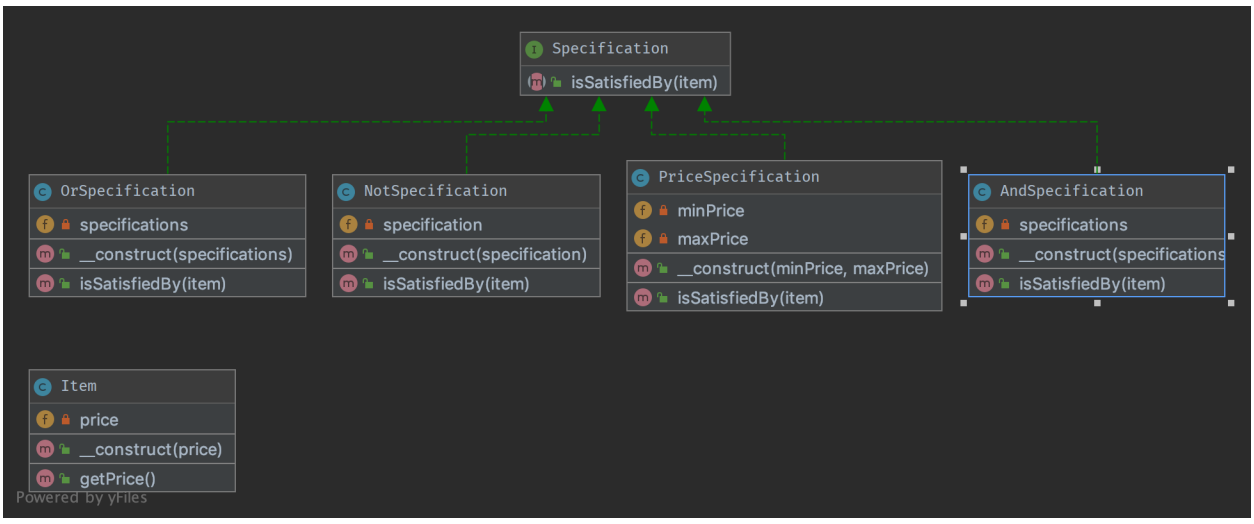
#### Предназначение

Изгражда ясна спецификация на бизнес правилата, където обектите могат да бъдат проверени. Композитният клас на спецификация (The composite specification class) има един метод, наречен `isSatisfiedBy`, който връща или true или false в зависимост от това дали даден обект удовлетворява спецификацията.

#### Примери

- RulerZ

#### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Item.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

class Item
{
    public function __construct(private float $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
```

(continues on next page)

(continued from previous page)

```
     */
    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
{
    public function __construct(private ?float $minPrice, private ?float $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
```

(continues on next page)

(continued from previous page)

```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
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification;

class NotSpecification implements Specification
{
    public function __construct(private Specification $specification)
    {
    }

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

(continues on next page)

(continued from previous page)

```
    }
}
```

### Тест

Tests/SpecificationTest.php

```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);
```

(continues on next page)

(continued from previous page)

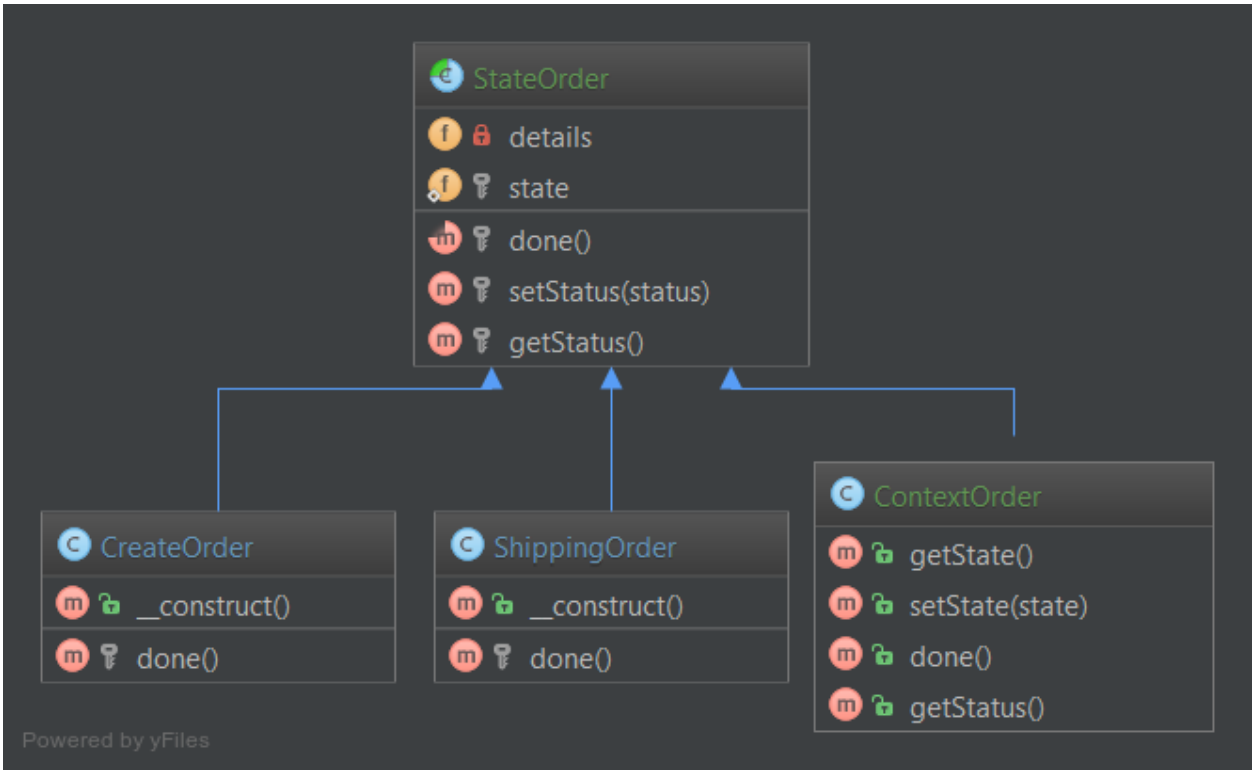
```
45
46          $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));
47          $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));
48      }
49  }
```

### 1.3.10 Състояние

#### Предназначение

Инкапсулира различно поведение за една и съща процедура (for the same routine) въз основа на състоянието на обекта. Това може да бъде по-чист начин за обект да промени поведението си по време на изпълнение, без да прибягва до големи монолитни условни изявления.

#### UML Диаграма



### Код

Можете също да намерите този код в GitHub

OrderContext.php

```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
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

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

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

StateCreated.php

```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
<?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
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class StateDone implements State
{
    public function proceedToNext(OrderContext $context)
    {
```

(continues on next page)

(continued from previous page)

```
        // there is nothing more to do
    }

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

### Тест

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();
```

(continues on next page)

(continued from previous page)

```
39          $contextOrder->proceedToNext();
40          $contextOrder->proceedToNext();
41          $contextOrder->proceedToNext();
42
43          $this->assertSame('done', $contextOrder->toString());
44      }
45  }
```

### 1.3.11 Стратегия

#### Терминология:

- Контекст
- Стратегия
- Конкретна стратегия

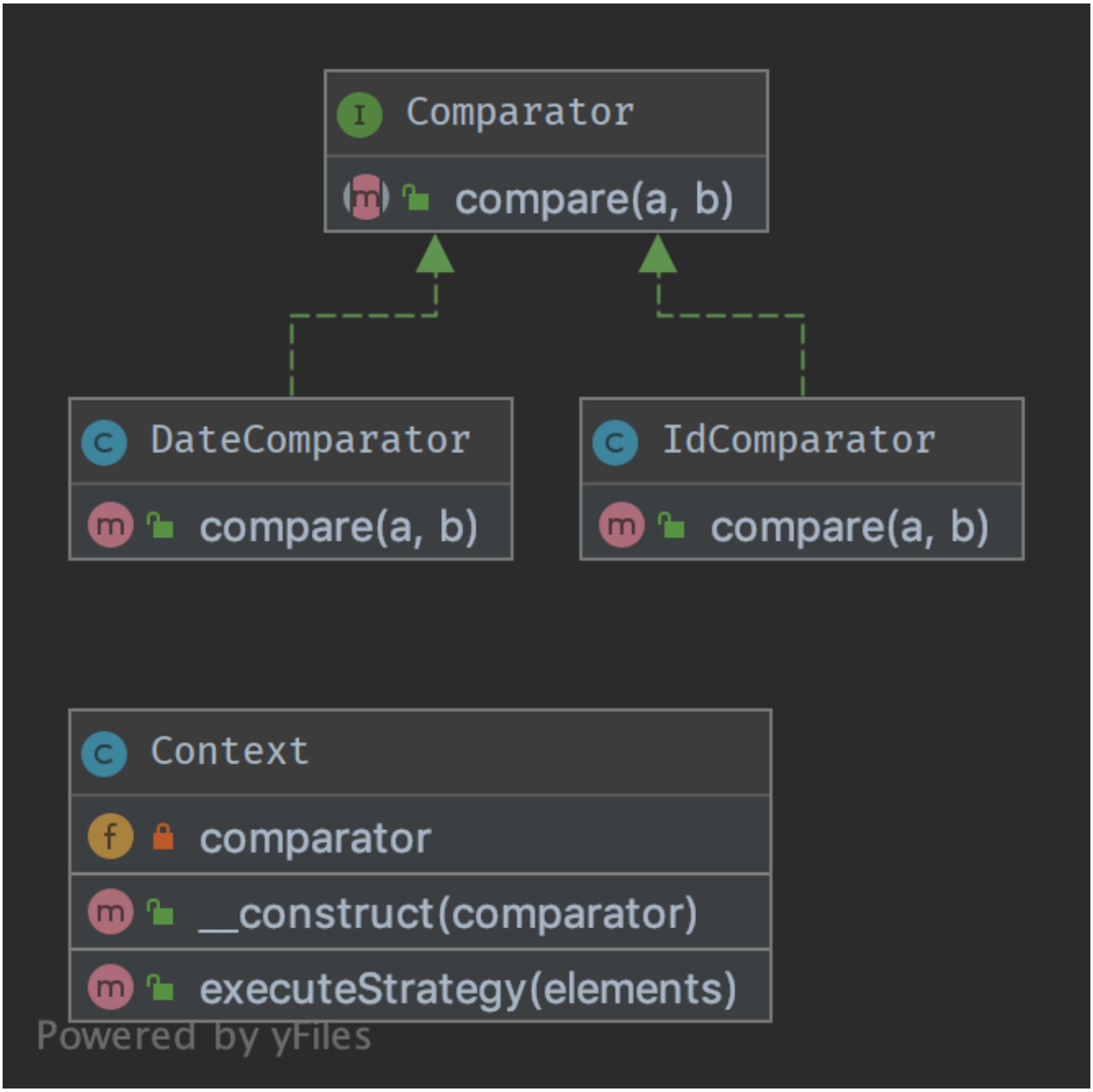
#### Предназначение

Да се отделят стратегии и да се даде възможност за бързо превключване между тях. Също така този модел е добра алтернатива на наследяването (вместо да има разширен абстрактен клас).

#### Примери

- сортиране на списък с обекти, едната стратегия по дата, другата по id
- опростяване на модулното тестване: напр. превключване между файл и съхранение в паметта (in-memory storage)

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Context.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;
```

(continues on next page)

(continued from previous page)

```
6
7   class Context
8   {
9       public function __construct(private Comparator $comparator)
10      {
11      }
12
13      public function executeStrategy(array $elements): array
14      {
15          uasort($elements, [$this->comparator, 'compare']);
16
17          return $elements;
18      }
19  }
```

Comparator.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\Strategy;
6
7   interface Comparator
8   {
9       /**
10       * @param mixed $a
11       * @param mixed $b
12       */
13      public function compare($a, $b): int;
14  }
```

DateComparator.php

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DesignPatterns\Behavioral\Strategy;
6
7   use DateTime;
8
9   class DateComparator implements Comparator
10  {
11      public function compare($a, $b): int
12      {
13          $aDate = new DateTime($a['date']);
14          $bDate = new DateTime($b['date']);
15
16          return $aDate <=> $bDate;
17      }
18  }
```

IdComparator.php

```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'];
    }
}
```

### Тест

Tests/StrategyTest.php

```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-
```

(continues on next page)

(continued from previous page)

```
↪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);
    }
}
```

### 1.3.12 Шаблонен метод

#### Предназначение

Шаблонен метод е поведенчески шаблон за дизайн.

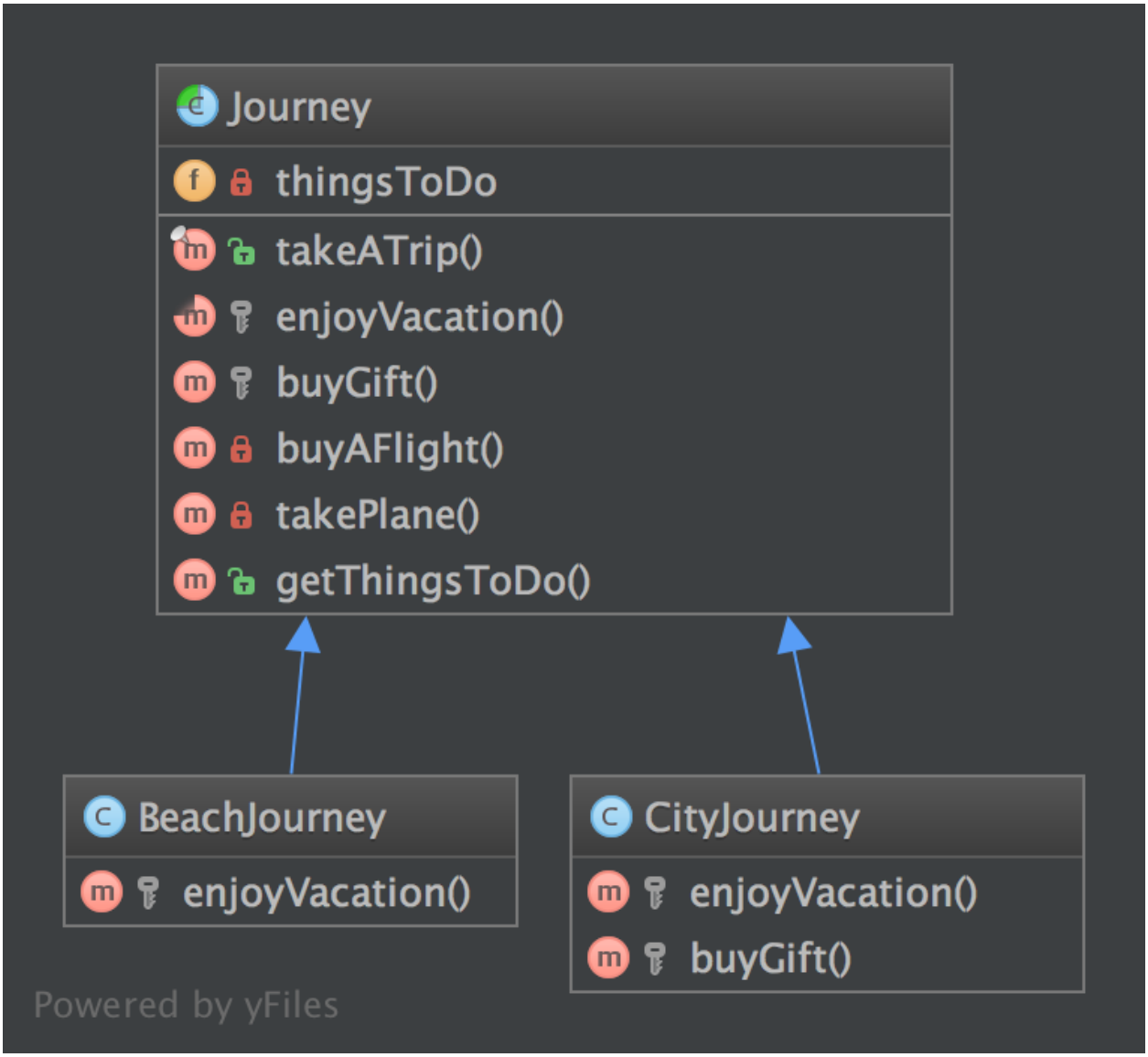
Може би вече сте го срещали много пъти. Идеята е да се остави подкласовете на този абстрактен шаблон да „довършат“ поведението на алгоритъм.

А.к.а „холивудският принцип“ („Hollywood principle“): „Не ни се обаждайте, ние ще ви се обадим“. Този клас не се извиква от подкласове, а обратно. Как? С абстракция разбира се.

С други думи, това е скелет на алгоритъм, подходящ за фреймуърк библиотеки. Потребителят трябва само да напиша един метод и суперкласът свърши работата.

Това е лесен начин да отделите конкретни класове и да намалите copy-paste, затова ще го намерите навсякъде.

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Journey.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\TemplateMethod;

abstract class Journey
{
```

(continues on next page)

(continued from previous page)

```
    /**
     * @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[]
     */
```

(continues on next page)

(continued from previous page)

```
    final public function getThingsToDo(): array
    {
        return $this->thingsToDo;
    }
}
```

BeachJourney.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

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";
    }
}
```

### Тест

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()
        );
    }
}
```

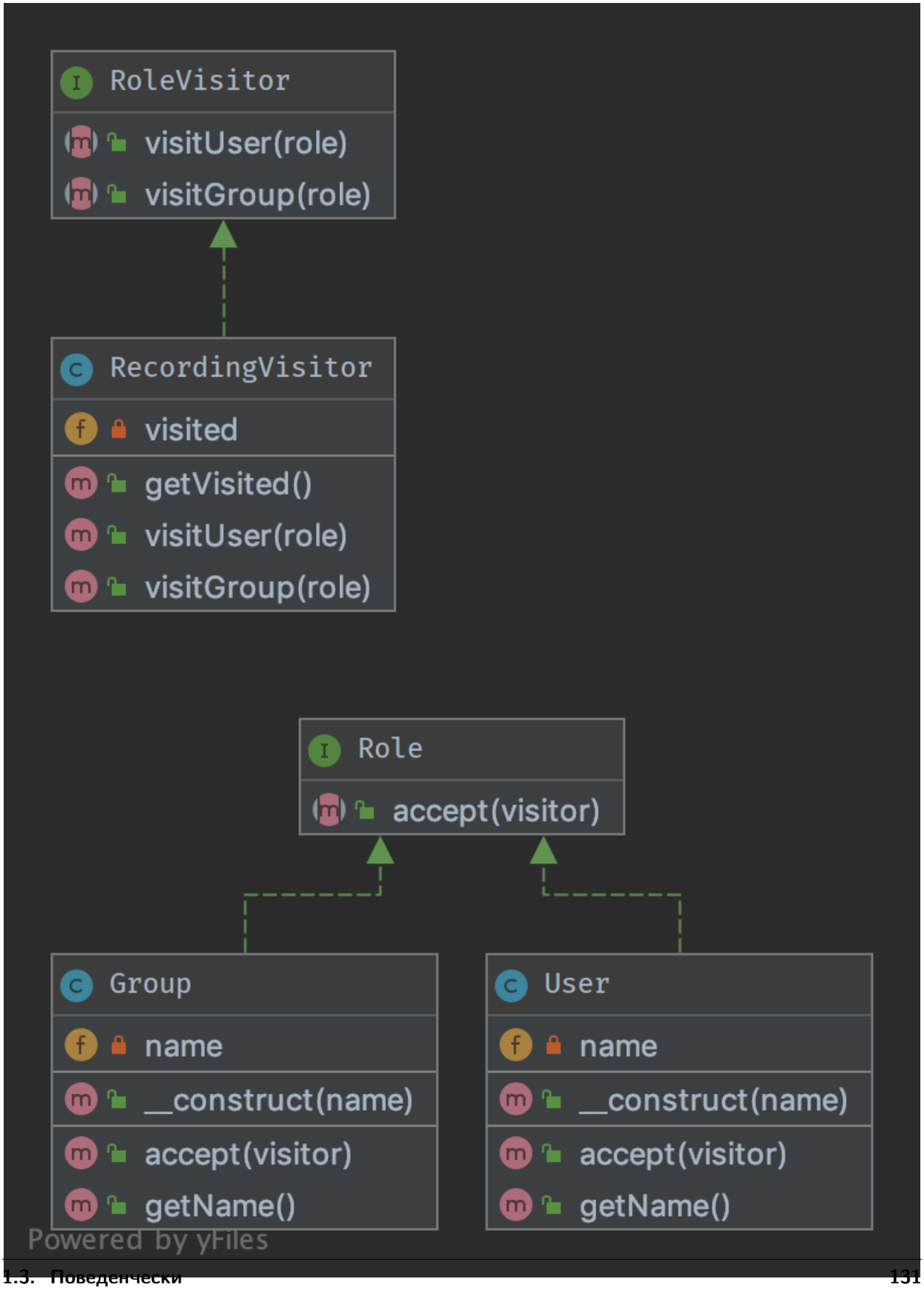
### 1.3.13 Посетител

#### Предназначение

Посетител (шаблон) ви позволява да възлагате операции върху обекти на други обекти. Основната причина да направите това е да поддържате разделение на проблемите. Но класовете трябва да дефинират договор, който да позволи на посетителите (методът `Role::accept` в примера).

Договорът е абстрактен клас, но можете да имате и изчистен интерфейс. В този случай всеки посетител трябва сам да избере кой метод на посетителя да извика.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

RoleVisitor.php

```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
<?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;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Role.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

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

User.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

class User implements Role
{
    public function __construct(private string $name)
    {
    }

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

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

Group.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Visitor;

class Group implements Role
{
    public function __construct(private string $name)
    {
```

(continues on next page)

(continued from previous page)

```
    }

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

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

### Тест

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)
```

(continues on next page)

(continued from previous page)

```
35      {
36          $role->accept($this->visitor);
37          $this->assertSame($role, $this->visitor->getVisited()[0]);
38      }
39  }
```

## 1.4 Допълнително

### 1.4.1 Локатор на служби

** ТОВА СЕ СЧИТА ЗА АНТИ-ШАБЛОН (anti-pattern)! **

Някои хора считат локатор на услуги за анти-модел (anti-pattern). Той нарушава принципа на инверсия на зависимостите. Локатор на услуги скрива зависимостите на класа, вместо да ги излага, както бихте направили, като използвате инжектирането на зависимост. В случай на промени в тези зависимости рискувате да нарушите функционалността на класовете, които ги използват, което прави вашата система трудна за поддръжка.

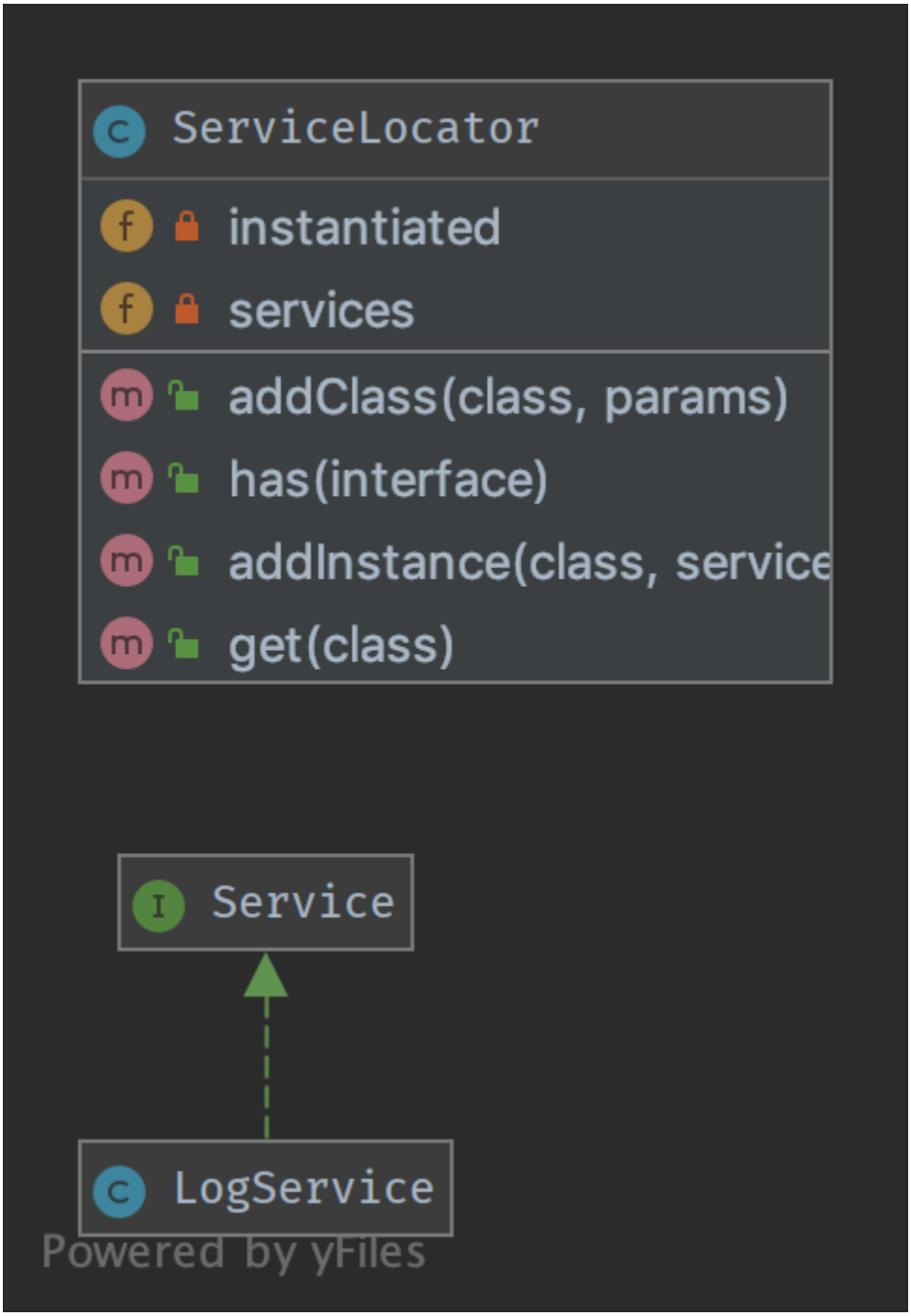
#### Предназначение

Да приложите свободно свързана архитектура (loosely coupled architecture), за да получите по-добре тестваем, поддържаем и разширяем код. Моделът DI и шаблонът Локатор на служби са изпълнение на Inverse of Control модел.

#### Използване

С `ServiceLocator` можете да регистрирате служба за даден интерфейс. С помощта на интерфейса можете да я извлечете и да я използвате в класовете на приложението, без да знаете изпълнението й. Можете да конфигурирате и инжектирате Service Locator обект в bootstrap.

### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Service.php

```php
<?php

namespace DesignPatterns\More\ServiceLocator;

interface Service
{
}
```

ServiceLocator.php

```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
```

(continues on next page)

(continued from previous page)

```
    {
        if (isset($this->instantiated[$class])) {
            return $this->instantiated[$class];
        }

        $object = new $class(...$this->services[$class]);

        if (!$object instanceof Service) {
            throw new InvalidArgumentException('Could not register service: is no␣
↪instance of Service');
        }

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

        return $object;
    }
}
```

LogService.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\More\ServiceLocator;

class LogService implements Service
{
}
```

### Тест

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();
```

(continues on next page)

(continued from previous page)

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

### 1.4.2 Repository

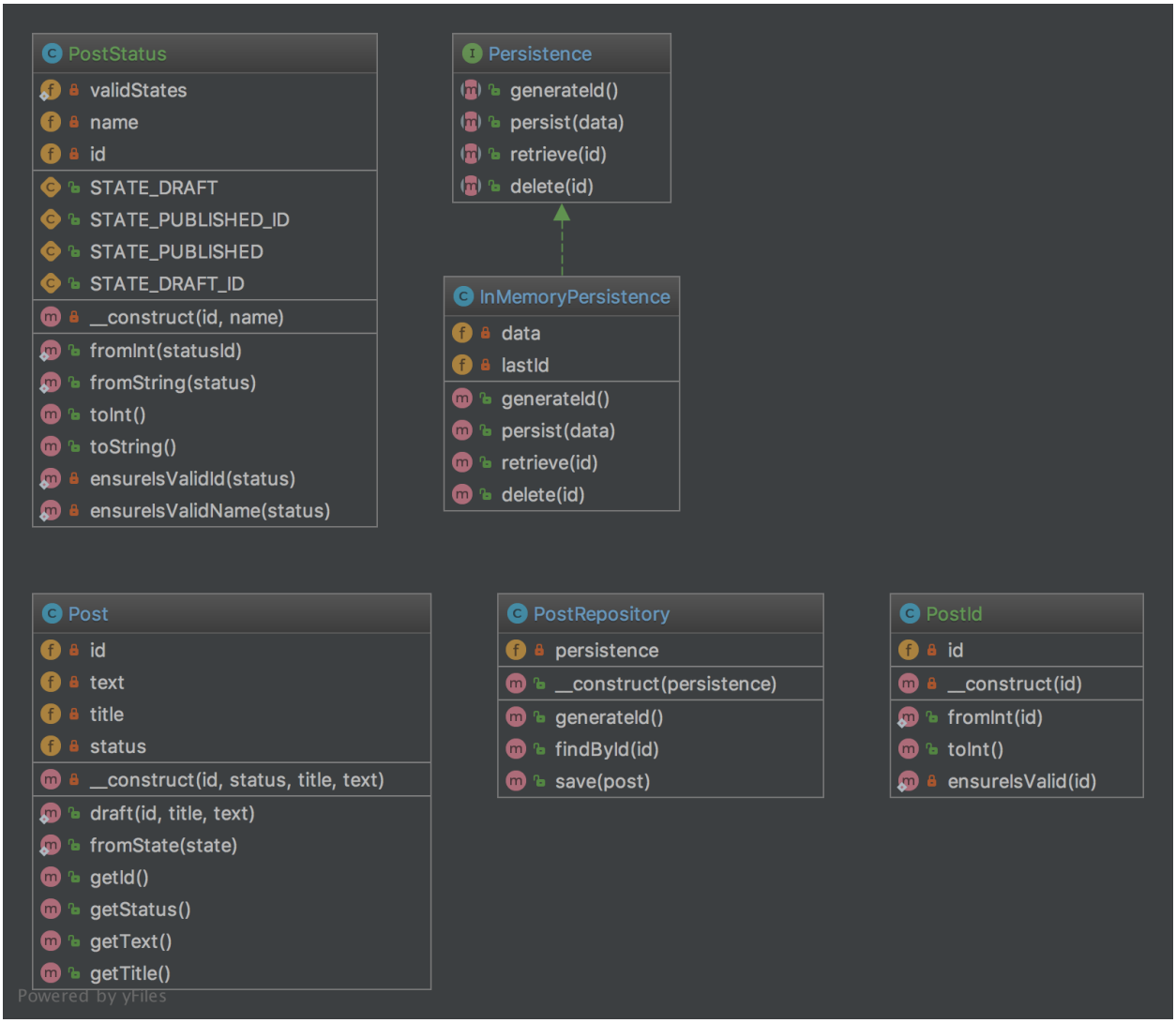
#### Предназначение

Посредничи между домейн и слоевете за картографиране на данни (data mapping layers), използвайки интерфейс, подобен на колекция, за достъп до обекти на домейн. Хранилището капсулира набора от обекти, запазени в хранилището за данни, и операциите, извършени над тях, осигурявайки по-обектно ориентиран изглед на слоя за устойчивост. Хранилището също така поддържа целта за постигане на чисто разделяне и еднопосочна зависимост между домейна и слоевете за картографиране на данни (data mapping layers).

#### Примери

- Доктрина 2 ORM: има хранилище (reposiory), което посредничи между Entity и DBAL и съдържа методи за извличане на обекти
- Laravel фреймуърк

### UML Диаграма




### Код

Можете също да намерите този код в GitHub

Post.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\More\Repository\Domain;

class Post
{
    public static function draft(PostId $id, string $title, string $text): Post
    {
```

(continues on next page)

(continued from previous page)

```
        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(
        private PostId $id,
        private PostStatus $status,
        private string $title,
        private string $text
    ) {
    }

    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);
```

(continues on next page)

(continued from previous page)

```php

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;

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

        return new self($id);
    }

    private function __construct(private int $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
<?php

declare(strict_types=1);

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;

/**
```

(continues on next page)

(continued from previous page)

```
 * Like PostId, this is a value object which holds the value of the current status of a Post. It can be constructed
 * either from a string or int and is able to validate itself. An instance can then be converted back to int or string.
 */
class PostStatus
{
    public const STATE_DRAFT_ID = 1;
    public const STATE_PUBLISHED_ID = 2;

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

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

    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(private int $id, private string $name)
    {
    }

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

    /**
     * there is a reason that I avoid using __toString() as it operates outside of the stack in PHP
     * and is therefore not able to operate well with exceptions
     */
    public function toString(): string
```

(continues on next page)

(continued from previous page)

```
    {
        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␣
↪(Persistence).
 *
 * Repository encapsulates the set of objects persisted in a data store and the␣
↪operations performed over them
 * providing a more object-oriented view of the persistence layer
 *
 * Repository also supports the objective of achieving a clean separation and one-way␣
↪dependency
 * between the domain and data mapping layers
 */
class PostRepository
{
    public function __construct(private Persistence $persistence)
    {
    }

    public function generateId(): PostId
    {
```

(continues on next page)

(continued from previous page)

```
        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->toInt()), 0, $e);
        }

        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;
```

(continues on next page)

(continued from previous page)

```php

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]);
    }
}
```

### Тест

Tests/PostRepositoryTest.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\More\Repository\Tests;

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

(continues on next page)

(continued from previous page)

```php
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());
    }
}
```

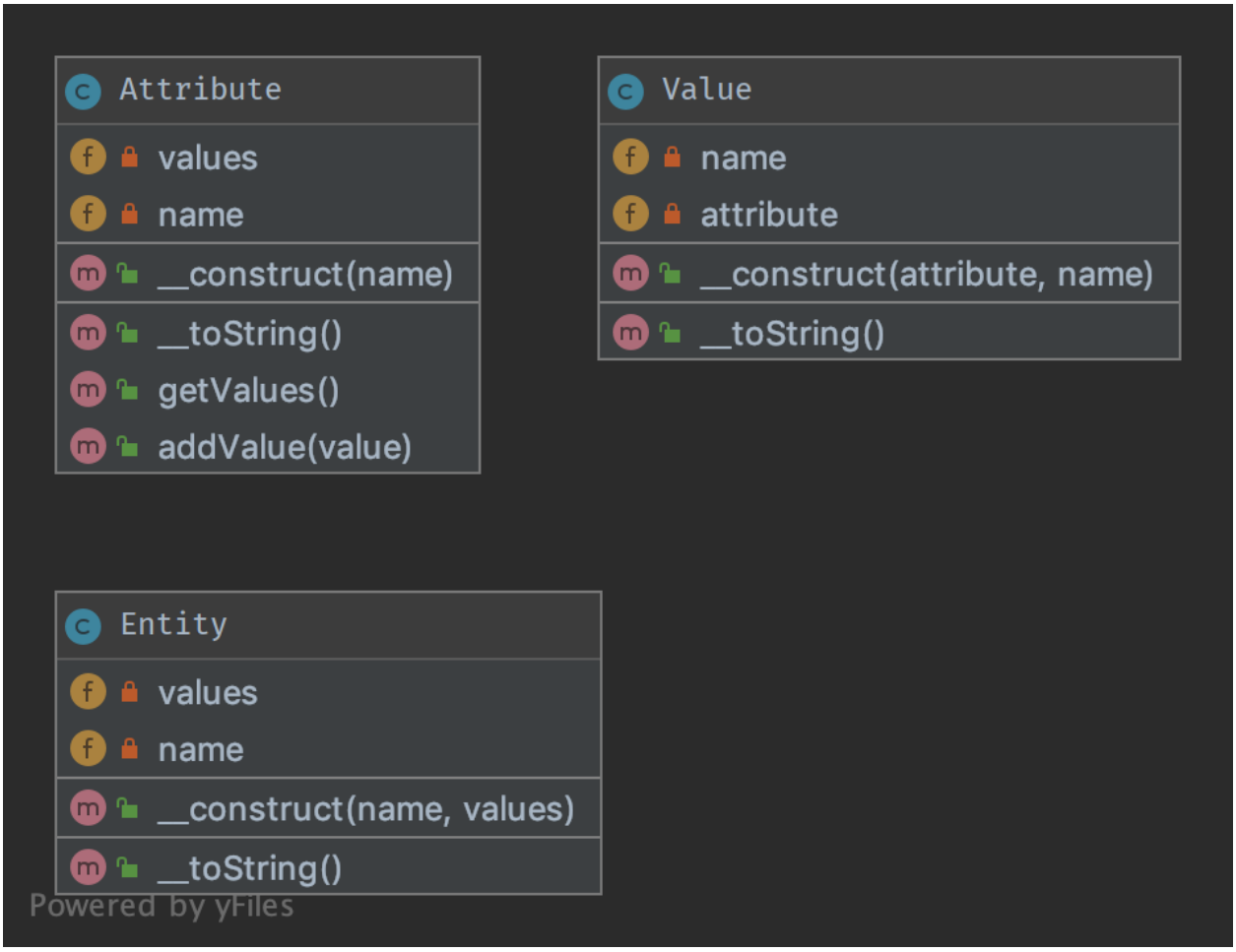
### 1.4.3 Entity-Attribute-Value (EAV)

Моделът Entity–attribute–value (EAV), за да се приложи EAV модел с PHP.

#### Предназначение

Моделът Entity–attribute–value (EAV) е модел на данни за описване на обекти, при които броят на атрибутите (свойства, параметри), които могат да бъдат използвани за тяхното описване, е потенциално голям, но броят, който действително ще се прилага за даден обект, е относително скромен.

#### UML Диаграма



### Код

Можете също да намерите този код в GitHub

Entity.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\More\EAV;

use SplObjectStorage;

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

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

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

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

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

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

Attribute.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\More\EAV;

use SplObjectStorage;
```

(continues on next page)

(continued from previous page)

```php

class Attribute implements \Stringable
{
    private SplObjectStorage $values;

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

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

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

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

Value.php

```php
<?php

declare(strict_types=1);

namespace DesignPatterns\More\EAV;

class Value implements \Stringable
{
    public function __construct(private Attribute $attribute, private string $name)
    {
        $attribute->addValue($this);
    }

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

### Тест

Tests/EAVTest.php

```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(): void
    {
        $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);
    }
}
```