
DesignPatternsPHP Documentation

Version 1.0

Dominik Liebler and contributors

sept. 30, 2023

Table des matières

1 Patterns	3
1.1 Créateurs	3
1.1.1 Abstract Factory	3
1.1.2 Builder	8
1.1.3 Factory Method	14
1.1.4 Pool	18
1.1.5 Prototype	22
1.1.6 Simple Factory	24
1.1.7 Singleton	26
1.1.8 Static Factory	29
1.2 Structuraux	32
1.2.1 Adapter / Wrapper	32
1.2.2 Bridge	37
1.2.3 Composite	41
1.2.4 Data Mapper	45
1.2.5 Decorator	49
1.2.6 Dependency Injection	53
1.2.7 Facade	57
1.2.8 Fluent Interface	60
1.2.9 Flyweight	62
1.2.10 Proxy	67
1.2.11 Registry	71
1.3 Comportementaux	74
1.3.1 Chain Of Responsibilities	74
1.3.2 Command	78
1.3.3 Interpreter	85
1.3.4 Iterator	89
1.3.5 Mediator	94
1.3.6 Memento	98
1.3.7 Null Object	103
1.3.8 Observer	107
1.3.9 Specification	110
1.3.10 State	115
1.3.11 Strategy	119
1.3.12 Template Method	124
1.3.13 Visitor	129

1.4	En savoir plus	134
1.4.1	Service Locator	134
1.4.2	Repository	138
1.4.3	Entity-Attribute-Value (EAV)	147

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.

CHAPITRE 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 Créateurs

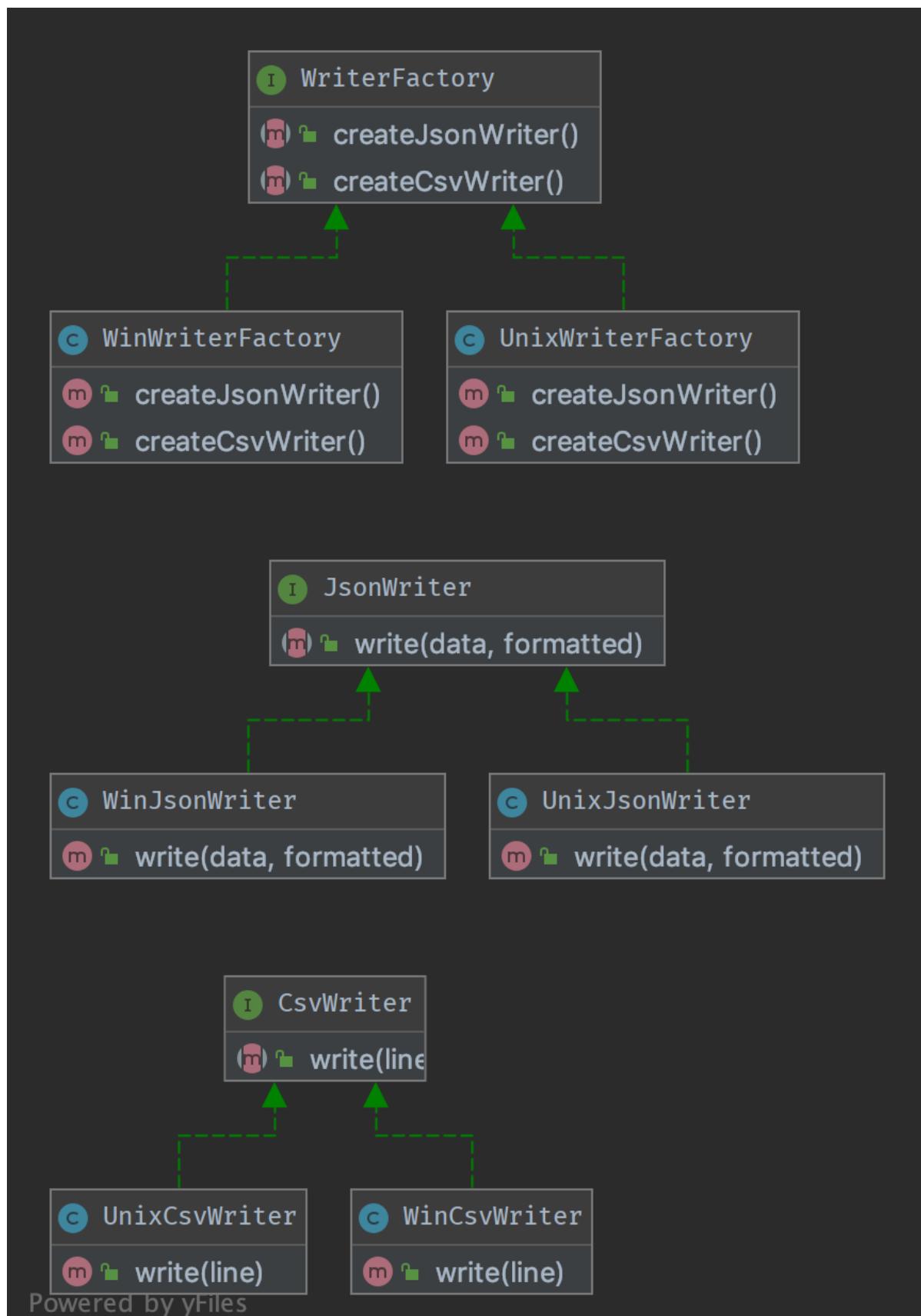
En ingénierie logicielle, les patrons de conception créateurs traitent des mécanismes de création d'objets d'une manière adaptée à la situation. La forme de base de la création d'objets pourrait entraîner des problèmes de conception ou ajouter de la complexité à la conception. Les patrons de conception créateurs résolvent cette problématique en contrôlant d'une manière ou d'une autre cette création d'objets.

1.1.1 Abstract Factory

Rôle

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

Diagramme UML



Powered by yFiles

Code

Vous pouvez également trouver ce code sur [GitHub](#)

WriterFactory.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 interface WriterFactory
6 {
7     public function createCsvWriter(): CsvWriter;
8     public function createJsonWriter(): JsonWriter;
9 }
```

CsvWriter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 interface CsvWriter
6 {
7     public function write(array $line): string;
8 }
```

JsonWriter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 interface JsonWriter
6 {
7     public function write(array $data, bool $formatted): string;
8 }
```

UnixCsvWriter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class UnixCsvWriter implements CsvWriter
6 {
7     public function write(array $line): string
8     {
9         return join(',', $line) . "\n";
10    }
11 }
```

UnixJsonWriter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class UnixJsonWriter implements JsonWriter
6 {
7     public function write(array $data, bool $formatted): string
8     {
9         $options = 0;
10
11         if ($formatted) {
12             $options = JSON_PRETTY_PRINT;
13         }
14
15         return json_encode($data, $options);
16     }
17 }
```

UnixWriterFactory.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class UnixWriterFactory implements WriterFactory
6 {
7     public function createCsvWriter(): CsvWriter
8     {
9         return new UnixCsvWriter();
10    }
11
12    public function createJsonWriter(): JsonWriter
13    {
14        return new UnixJsonWriter();
15    }
16 }
```

WinCsvWriter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class WinCsvWriter implements CsvWriter
6 {
7     public function write(array $line): string
8     {
9         return join(',', $line) . "\r\n";
10    }
11 }
```

WinJsonWriter.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class WinJsonWriter implements JsonWriter
6 {
7     public function write(array $data, bool $formatted): string
8     {
9         $options = 0;
10
11         if ($formatted) {
12             $options = JSON_PRETTY_PRINT;
13         }
14
15         return json_encode($data, $options);
16     }
17 }
```

WinWriterFactory.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 class WinWriterFactory implements WriterFactory
6 {
7     public function createCsvWriter(): CsvWriter
8     {
9         return new WinCsvWriter();
10    }
11
12    public function createJsonWriter(): JsonWriter
13    {
14        return new WinJsonWriter();
15    }
16 }
```

Test

Tests/AbstractFactoryTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\AbstractFactory\Tests;
6
7 use DesignPatterns\Creational\AbstractFactory\CsvWriter;
8 use DesignPatterns\Creational\AbstractFactory\JsonWriter;
9 use DesignPatterns\Creational\AbstractFactory\UnixWriterFactory;
10 use DesignPatterns\Creational\AbstractFactory\WinWriterFactory;
11 use DesignPatterns\Creational\AbstractFactory\WriterFactory;
```

(suite sur la page suivante)

(suite de la page précédente)

```
12 use PHPUnit\Framework\TestCase;
13
14 class AbstractFactoryTest extends TestCase
15 {
16     public function provideFactory()
17     {
18         return [
19             [new UnixWriterFactory()],
20             [new WinWriterFactory()]
21         ];
22     }
23
24 /**
25 * @dataProvider provideFactory
26 */
27 public function testCanCreateCsvWriterOnUnix(WriterFactory $writerFactory)
28 {
29     $this->assertInstanceOf(JsonWriter::class, $writerFactory->createJsonWriter());
30     $this->assertInstanceOf(CsvWriter::class, $writerFactory->createCsvWriter());
31 }
32 }
```

1.1.2 Builder

Rôle

Un builder est une interface permettant de construire certaines parties d'un objet complexe.

Parfois, lorsque le builder « sait » plus précisément ce qu'il construit, cette interface pourrait être une classe abstraite avec des méthodes par défaut (alias adapter).

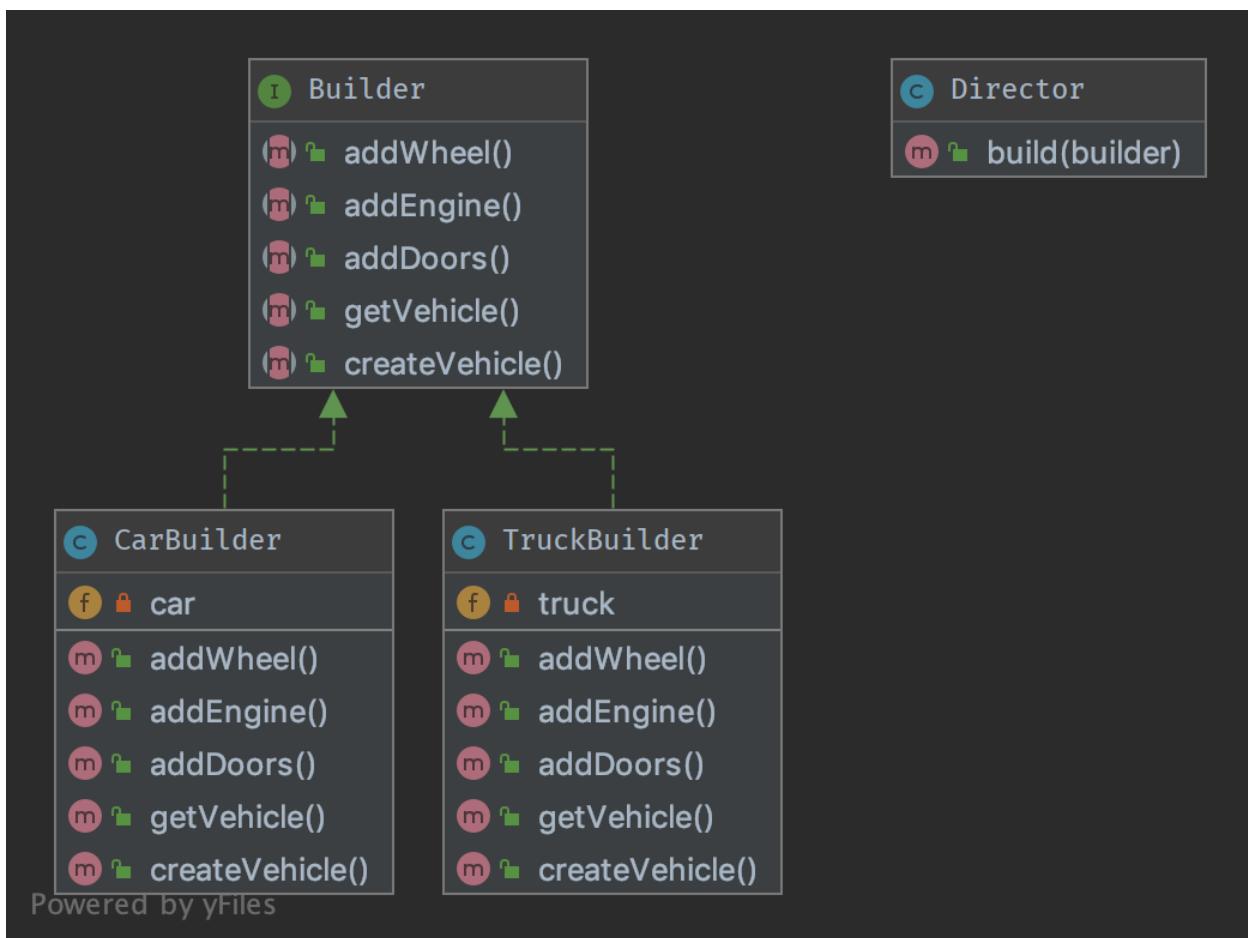
Si vous avez un arbre d'héritage complexe pour les objets, il est logique d'avoir un arbre d'héritage complexe pour les builders.

Note : Les builders ont souvent une désignation chaînée ou chaînage de méthodes (fluent interface), voir le Mock Builder de PHPUnit par exemple.

Exemples

— PHPUnit : Mock Builder

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Director.php

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace DesignPatterns\Creational\Builder;
6
7  use DesignPatterns\Creational\Builder\Parts\Vehicle;
8
9 /**
10 * Director is part of the builder pattern. It knows the interface of the builder
11 * and builds a complex object with the help of the builder
12 *
13 * You can also inject many builders instead of one to build more complex objects
14 */
15 class Director
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

16 {
17     public function build(Builder $builder): Vehicle
18     {
19         $builder->createVehicle();
20         $builder->addDoors();
21         $builder->addEngine();
22         $builder->addWheel();
23
24         return $builder->getVehicle();
25     }
26 }
```

Builder.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder;
6
7 use DesignPatterns\Creational\Builder\Parts\Vehicle;
8
9 interface Builder
10 {
11     public function createVehicle(): void;
12
13     public function addWheel(): void;
14
15     public function addEngine(): void;
16
17     public function addDoors(): void;
18
19     public function getVehicle(): Vehicle;
20 }
```

TruckBuilder.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder;
6
7 use DesignPatterns\Creational\Builder\Parts\Door;
8 use DesignPatterns\Creational\Builder\Parts\Engine;
9 use DesignPatterns\Creational\Builder\Parts\Wheel;
10 use DesignPatterns\Creational\Builder\Parts\Truck;
11 use DesignPatterns\Creational\Builder\Parts\Vehicle;
12
13 class TruckBuilder implements Builder
14 {
15     private Truck $truck;
```

(suite sur la page suivante)

(suite de la page précédente)

```

16     public function addDoors(): void
17     {
18         $this->truck->setPart('rightDoor', new Door());
19         $this->truck->setPart('leftDoor', new Door());
20     }
21
22
23     public function addEngine(): void
24     {
25         $this->truck->setPart('truckEngine', new Engine());
26     }
27
28     public function addWheel(): void
29     {
30         $this->truck->setPart('wheel1', new Wheel());
31         $this->truck->setPart('wheel2', new Wheel());
32         $this->truck->setPart('wheel3', new Wheel());
33         $this->truck->setPart('wheel4', new Wheel());
34         $this->truck->setPart('wheel5', new Wheel());
35         $this->truck->setPart('wheel6', new Wheel());
36     }
37
38     public function createVehicle(): void
39     {
40         $this->truck = new Truck();
41     }
42
43     public function getVehicle(): Vehicle
44     {
45         return $this->truck;
46     }
47 }
```

CarBuilder.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder;
6
7 use DesignPatterns\Creational\Builder\Parts\Door;
8 use DesignPatterns\Creational\Builder\Parts\Engine;
9 use DesignPatterns\Creational\Builder\Parts\Wheel;
10 use DesignPatterns\Creational\Builder\Parts\Car;
11 use DesignPatterns\Creational\Builder\Parts\Vehicle;
12
13 class CarBuilder implements Builder
14 {
15     private Car $car;
16
17     public function addDoors(): void
```

(suite sur la page suivante)

(suite de la page précédente)

```

18     {
19         $this->car->setPart('rightDoor', new Door());
20         $this->car->setPart('leftDoor', new Door());
21         $this->car->setPart('trunkLid', new Door());
22     }
23
24     public function addEngine(): void
25     {
26         $this->car->setPart('engine', new Engine());
27     }
28
29     public function addWheel(): void
30     {
31         $this->car->setPart('wheelLF', new Wheel());
32         $this->car->setPart('wheelRF', new Wheel());
33         $this->car->setPart('wheelLR', new Wheel());
34         $this->car->setPart('wheelRR', new Wheel());
35     }
36
37     public function createVehicle(): void
38     {
39         $this->car = new Car();
40     }
41
42     public function getVehicle(): Vehicle
43     {
44         return $this->car;
45     }
46 }
```

Parts/Vehicle.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder\Parts;
6
7 abstract class Vehicle
8 {
9     final public function setPart(string $key, object $value)
10    {
11        }
12 }
```

Parts/Truck.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder\Parts;
```

(suite sur la page suivante)

(suite de la page précédente)

```
6 class Truck extends Vehicle  
7 {  
8 }  
9
```

Parts/Car.php

```
1 <?php  
2  
3 declare(strict_types=1);  
4  
5 namespace DesignPatterns\Creational\Builder\Parts;  
6  
7 class Car extends Vehicle  
8 {  
9 }
```

Parts/Engine.php

```
1 <?php  
2  
3 declare(strict_types=1);  
4  
5 namespace DesignPatterns\Creational\Builder\Parts;  
6  
7 class Engine  
8 {  
9 }
```

Parts/Wheel.php

```
1 <?php  
2  
3 declare(strict_types=1);  
4  
5 namespace DesignPatterns\Creational\Builder\Parts;  
6  
7 class Wheel  
8 {  
9 }
```

Parts/Door.php

```
1 <?php  
2  
3 declare(strict_types=1);  
4  
5 namespace DesignPatterns\Creational\Builder\Parts;  
6  
7 class Door  
8 {  
9 }
```

Test

Tests/DirectorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Builder\Tests;
6
7 use DesignPatterns\Creational\Builder\Parts\Car;
8 use DesignPatterns\Creational\Builder\Parts\Truck;
9 use DesignPatterns\Creational\Builder\TruckBuilder;
10 use DesignPatterns\Creational\Builder\CarBuilder;
11 use DesignPatterns\Creational\Builder\Director;
12 use PHPUnit\Framework\TestCase;
13
14 class DirectorTest extends TestCase
15 {
16     public function testCanBuildTruck()
17     {
18         $truckBuilder = new TruckBuilder();
19         $newVehicle = (new Director())->build($truckBuilder);
20
21         $this->assertInstanceOf(Truck::class, $newVehicle);
22     }
23
24     public function testCanBuildCar()
25     {
26         $carBuilder = new CarBuilder();
27         $newVehicle = (new Director())->build($carBuilder);
28
29         $this->assertInstanceOf(Car::class, $newVehicle);
30     }
31 }
```

1.1.3 Factory Method

Rôle

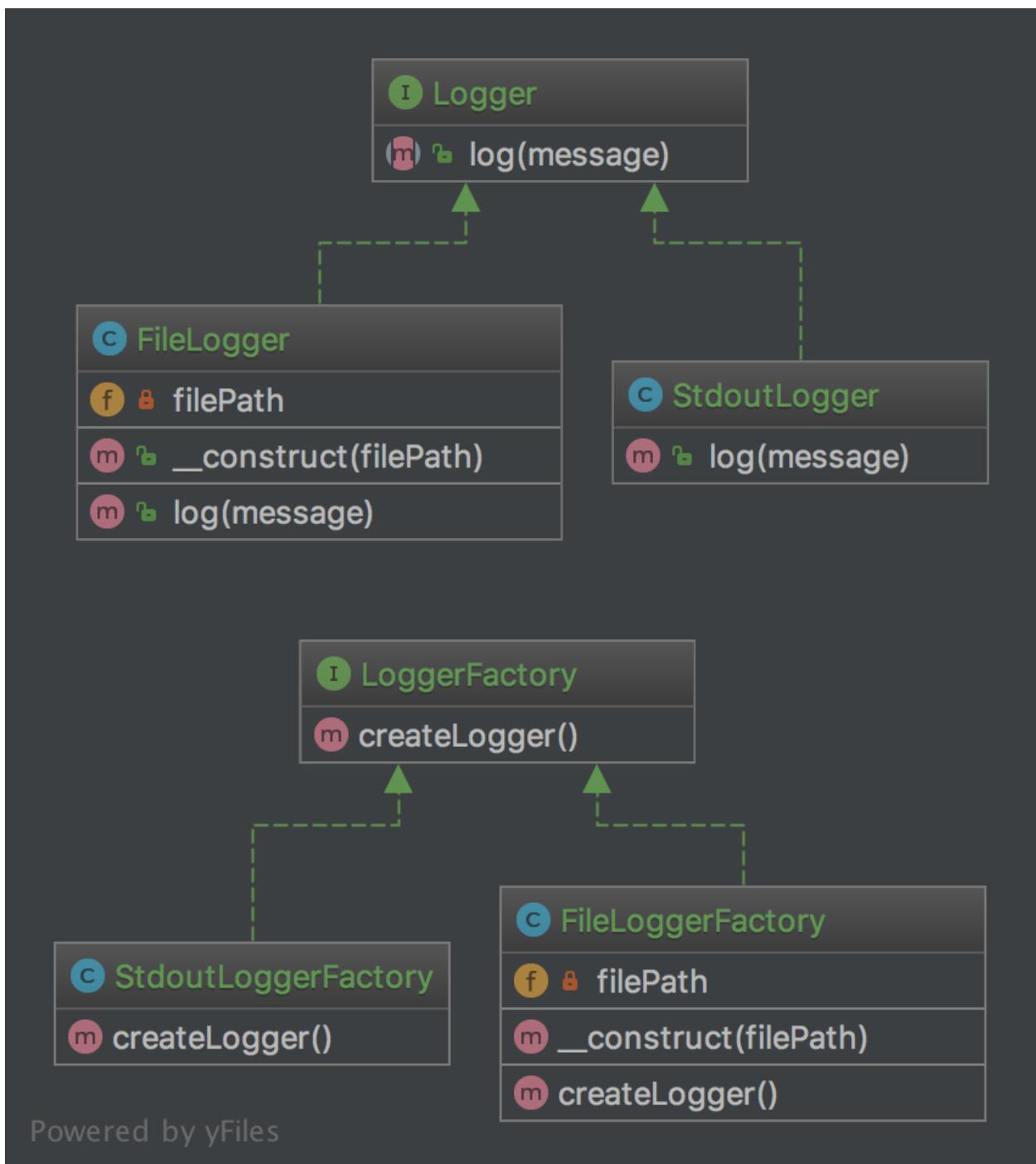
L'avantage de FactoryMethod sur SimpleFactory est que le factory peut être dérivée afin de mettre oeuvres différentes façons de créer un objet.

Pour les cas simples, cette classe abstraite peut être une simple interface.

Ce modèle est un « vrai » patron de conception car il applique le principe d'inversion des dépendances (Dependency Inversion principle) alias « D » dans SOLID (voir principes SOLID).

Cela signifie que FactoryMethod dépend d'abstractions et non de classes concrètes. C'est son principal avantage par rapport à SimpleFactory ou StaticFactory.

Diagramme UML



Powered by yFiles

Code

Vous pouvez également trouver ce code sur [GitHub](#)

Logger.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 interface Logger
8 {
9     public function log(string $message);
10}
```

StdoutLogger.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 class StdoutLogger implements Logger
8 {
9     public function log(string $message)
10    {
11        echo $message;
12    }
13}
```

FileLogger.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 class FileLogger implements Logger
8 {
9     public function __construct(private string $filePath)
10    {
11    }
12
13     public function log(string $message)
14    {
15         file_put_contents($this->filePath, $message . PHP_EOL, FILE_APPEND);
16    }
17}
```

LoggerFactory.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 interface LoggerFactory
8 {
9     public function createLogger(): Logger;
10 }
```

StdoutLoggerFactory.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 class StdoutLoggerFactory implements LoggerFactory
8 {
9     public function createLogger(): Logger
10     {
11         return new StdoutLogger();
12     }
13 }
```

FileLoggerFactory.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod;
6
7 class FileLoggerFactory implements LoggerFactory
8 {
9     public function __construct(private string $filePath)
10     {
11     }
12
13     public function createLogger(): Logger
14     {
15         return new FileLogger($this->filePath);
16     }
17 }
```

Test

Tests/FactoryMethodTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\FactoryMethod\Tests;
6
7 use DesignPatterns\Creational\FactoryMethod\FileLogger;
8 use DesignPatterns\Creational\FactoryMethod\FileLoggerFactory;
9 use DesignPatterns\Creational\FactoryMethod\StdoutLogger;
10 use DesignPatterns\Creational\FactoryMethod\StdoutLoggerFactory;
11 use PHPUnit\Framework\TestCase;
12
13 class FactoryMethodTest extends TestCase
14 {
15     public function testCanCreateStdoutLogging()
16     {
17         $loggerFactory = new StdoutLoggerFactory();
18         $logger = $loggerFactory->createLogger();
19
20         $this->assertInstanceOf(StdoutLogger::class, $logger);
21     }
22
23     public function testCanCreateFileLogging()
24     {
25         $loggerFactory = new FileLoggerFactory(sys_get_temp_dir());
26         $logger = $loggerFactory->createLogger();
27
28         $this->assertInstanceOf(FileLogger::class, $logger);
29     }
30 }
```

1.1.4 Pool

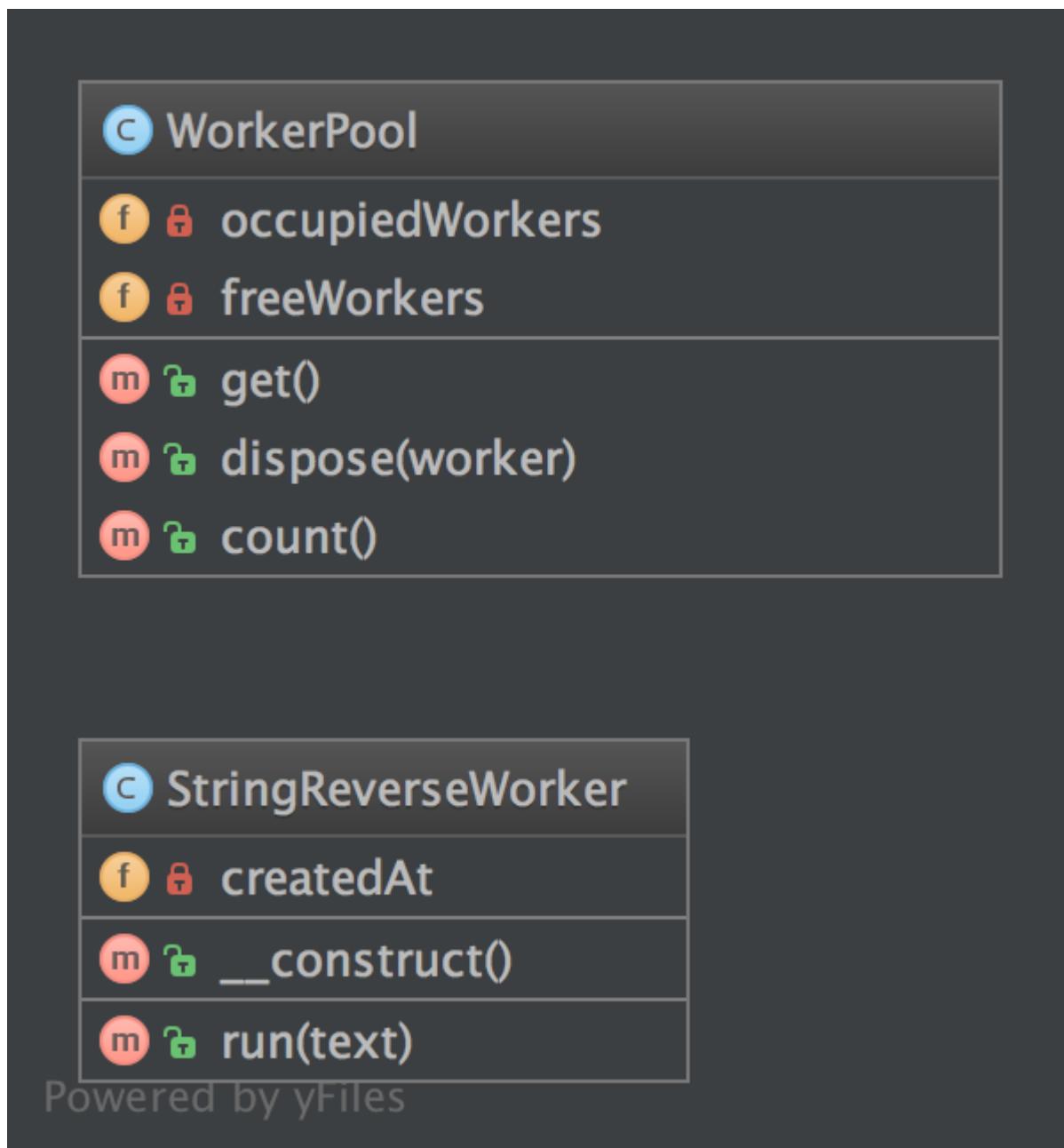
Purpose

Object pool pattern utilise un ensemble d'objets initialisés et prêts à être utilisés - un « pool » - plutôt que de les allouer et de les détruire à la demande. Un client du pool demandera un objet du pool et effectuera des opérations sur l'objet retourné. Lorsque le client a terminé, il renvoie l'objet, qui est au pool plutôt que de le détruire.

La mise en commun d'objets peut offrir un gain de performance significatif dans les situations où le coût d'initialisation d'une instance de classe est élevé, le taux d'instanciation d'une classe est élevé et le nombre d'instances utilisées à un moment donné est faible. L'objet mis en commun est obtenu en un temps prévisible alors que la création des nouveaux objets (notamment sur le réseau) peut prendre un temps variable.

Toutefois, ces avantages s'appliquent surtout aux objets qui sont coûteux en temps, tels que les connexions aux bases de données, les connexions aux sockets, les threads et les gros objets graphiques comme les polices ou les bitmaps. Dans certaines situations, la mise en commun d'objets simples (qui ne détiennent aucune ressource externe, mais occupent seulement la mémoire) peut ne pas être efficace et pourrait diminuer les performances.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

WorkerPool.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Pool;
6
7 use Countable;
8
9 class WorkerPool implements Countable
10 {
11     /**
12      * @var StringReverseWorker[]
13      */
14     private array $occupiedWorkers = [];
15
16     /**
17      * @var StringReverseWorker[]
18      */
19     private array $freeWorkers = [];
20
21     public function get(): StringReverseWorker
22     {
23         if (count($this->freeWorkers) === 0) {
24             $worker = new StringReverseWorker();
25         } else {
26             $worker = array_pop($this->freeWorkers);
27         }
28
29         $this->occupiedWorkers[spl_object_hash($worker)] = $worker;
30
31         return $worker;
32     }
33
34     public function dispose(StringReverseWorker $worker): void
35     {
36         $key = spl_object_hash($worker);
37         if (isset($this->occupiedWorkers[$key])) {
38             unset($this->occupiedWorkers[$key]);
39             $this->freeWorkers[$key] = $worker;
40         }
41     }
42
43     public function count(): int
44     {
45         return count($this->occupiedWorkers) + count($this->freeWorkers);
46     }
47 }
```

StringReverseWorker.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Pool;
6
7 class StringReverseWorker
8 {
9     public function run(string $text): string
10    {
11        return strrev($text);
12    }
13 }
```

Test

Tests/PoolTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Pool\Tests;
6
7 use DesignPatterns\Creational\Pool\WorkerPool;
8 use PHPUnit\Framework\TestCase;
9
10 class PoolTest extends TestCase
11 {
12     public function testCanGetNewInstancesWithGet()
13     {
14         $pool = new WorkerPool();
15         $worker1 = $pool->get();
16         $worker2 = $pool->get();
17
18         $this->assertCount(2, $pool);
19         $this->assertNotSame($worker1, $worker2);
20     }
21
22     public function testCanGetSameInstanceTwiceWhenDisposingItFirst()
23     {
24         $pool = new WorkerPool();
25         $worker1 = $pool->get();
26         $pool->dispose($worker1);
27         $worker2 = $pool->get();
28
29         $this->assertCount(1, $pool);
30         $this->assertSame($worker1, $worker2);
31     }
32 }
```

1.1.5 Prototype

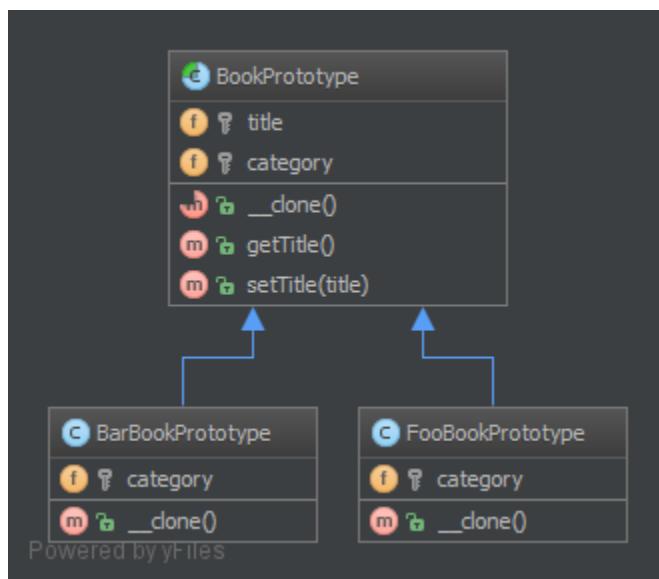
Rôle

Minimise le coût de création de nouveaux objets en clonant un objet prototype

Exemples

- Grandes quantités de données (par exemple, créer 1 000 000 de lignes dans une base de données en une seule fois via un ORM).

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

BookPrototype.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Prototype;
6
7 abstract class BookPrototype
8 {
9     protected string $title;
10    protected string $category;
11
12    abstract public function __clone();
13

```

(suite sur la page suivante)

(suite de la page précédente)

```
14     final public function getTitle(): string
15     {
16         return $this->title;
17     }
18
19     final public function setTitle(string $title): void
20     {
21         $this->title = $title;
22     }
23 }
```

BarBookPrototype.php

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

FooBookPrototype.php

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

Test

Tests/PrototypeTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Prototype\Tests;
6
7 use DesignPatterns\Creational\Prototype\BarBookPrototype;
8 use DesignPatterns\Creational\Prototype\FooBookPrototype;
9 use PHPUnit\Framework\TestCase;
10
11 class PrototypeTest extends TestCase
12 {
13     public function testCanGetFooBook()
14     {
15         $fooPrototype = new FooBookPrototype();
16         $barPrototype = new BarBookPrototype();
17
18         for ($i = 0; $i < 10; $i++) {
19             $book = clone $fooPrototype;
20             $book->setTitle('Foo Book No ' . $i);
21             $this->assertInstanceOf(FooBookPrototype::class, $book);
22         }
23
24         for ($i = 0; $i < 5; $i++) {
25             $book = clone $barPrototype;
26             $book->setTitle('Bar Book No ' . $i);
27             $this->assertInstanceOf(BarBookPrototype::class, $book);
28         }
29     }
30 }
```

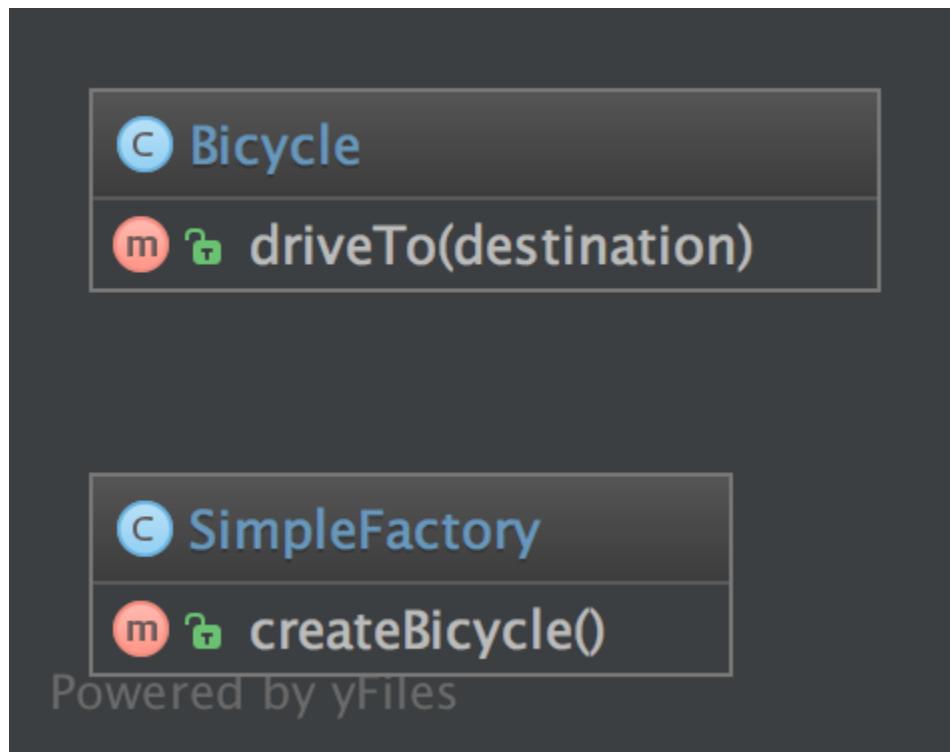
1.1.6 Simple Factory

Rôle

SimpleFactory est un pattern Factory simple.

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

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

SimpleFactory.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\SimpleFactory;
6
7 class SimpleFactory
8 {
9     public function createBicycle(): Bicycle
10    {
11        return new Bicycle();
12    }
13 }
  
```

Bicycle.php

```

1 <?php
2
3 declare(strict_types=1);
4
  
```

(suite sur la page suivante)

(suite de la page précédente)

```
5 namespace DesignPatterns\Creational\SimpleFactory;  
6  
7 class Bicycle  
8 {  
9     public function driveTo(string $destination)  
10    {  
11        }  
12    }
```

Usage

```
1 $factory = new SimpleFactory();  
2 $bicycle = $factory->createBicycle();  
3 $bicycle->driveTo('Paris');
```

Test

Tests/SimpleFactoryTest.php

```
1 <?php  
2  
3 declare(strict_types=1);  
4  
5 namespace DesignPatterns\Creational\SimpleFactory\Tests;  
6  
7 use DesignPatterns\Creational\SimpleFactory\Bicycle;  
8 use DesignPatterns\Creational\SimpleFactory\SimpleFactory;  
9 use PHPUnit\Framework\TestCase;  
10  
11 class SimpleFactoryTest extends TestCase  
12 {  
13     public function testCanCreateBicycle()  
14     {  
15         $bicycle = (new SimpleFactory())->createBicycle();  
16         $this->assertInstanceOf(Bicycle::class, $bicycle);  
17     }  
18 }
```

1.1.7 Singleton

SINGLETON EST CONSIDÉRÉ COMME UN ANTI-PATTERN! POUR UNE MEILLEURE TESTABILITÉ ET MAINTENABILITÉ, UTILISEZ L'INJECTION DE DÉPENDANCES!

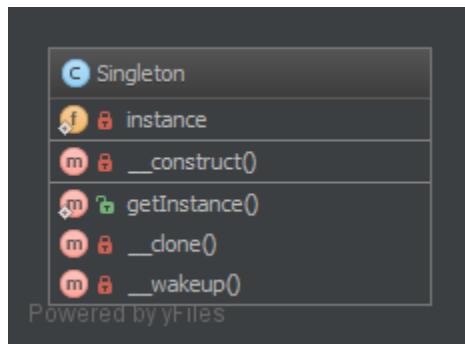
Rôle

Limite l'instanciation d'une classe à une seule instance.

Exemples

- Connexion à la base de données
- Logger
- Config Manager
- Threads Handling
- Fichier de verrouillage pour l'application (il n'y en a qu'un seul dans le système de fichiers...)

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Singleton.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Singleton;
6
7 use Exception;
8
9 final class Singleton
10 {
11     private static ?Singleton $instance = null;
12
13     /**
14      * gets the instance via lazy initialization (created on first usage)
15      */
16     public static function getInstance(): Singleton
17     {
18         if (self::$instance === null) {
19             self::$instance = new self();
20         }
21     }
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

21     return self::$instance;
22 }
23
24 /**
25 * is not allowed to call from outside to prevent from creating multiple instances,
26 * to use the singleton, you have to obtain the instance from
27 → Singleton::getInstance() instead
28 */
29 private function __construct()
30 {
31 }
32
33 /**
34 * prevent the instance from being cloned (which would create a second instance of
35 → it)
36 */
37 private function __clone()
38 {
39 }
40
41 /**
42 * prevent from being unserialized (which would create a second instance of it)
43 */
44 public function __wakeup()
45 {
46     throw new Exception("Cannot unserialize singleton");
47 }

```

Test

Tests/SingletonTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\Singleton\Tests;
6
7 use DesignPatterns\Creational\Singleton\Singleton;
8 use PHPUnit\Framework\TestCase;
9
10 class SingletonTest extends TestCase
11 {
12     public function testUniqueness()
13     {
14         $firstCall = Singleton::getInstance();
15         $secondCall = Singleton::getInstance();
16
17         $this->assertInstanceOf(Singleton::class, $firstCall);

```

(suite sur la page suivante)

(suite de la page précédente)

```

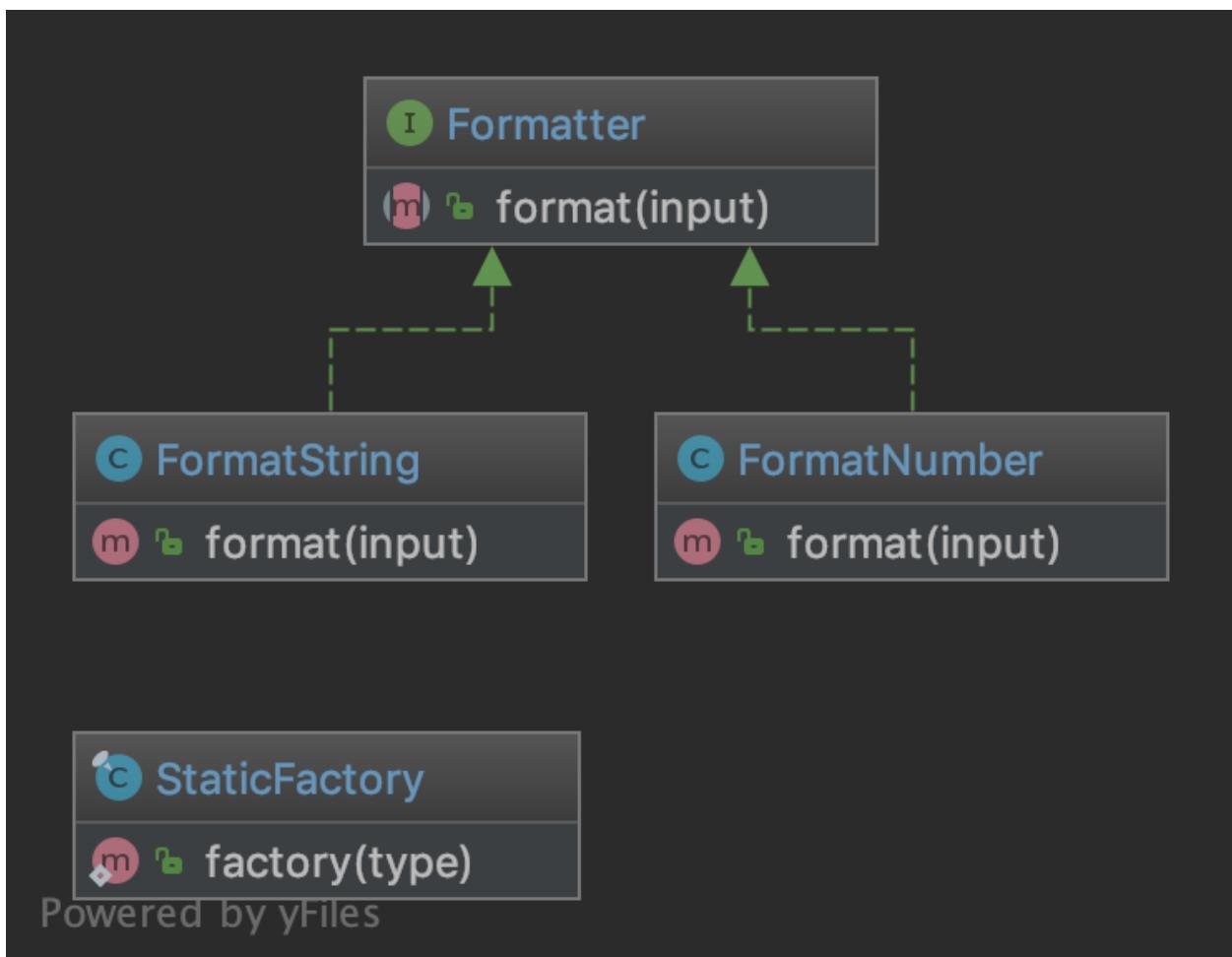
18     $this->assertSame($firstCall, $secondCall);
19 }
20 }
```

1.1.8 Static Factory

Rôle

Similaire à AbstractFactory, ce pattern est utilisé pour créer des séries d'objets liés ou dépendants. La différence entre ce modèle et le modèle AbstractFactory est que StaticFactory utilise une seule méthode statique pour créer tous les types d'objets qu'il peut créer. Cette méthode est généralement nommée `factory` ou `build`.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

StaticFactory.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\StaticFactory;
6
7 use InvalidArgumentException;
8
9 /**
10 * Note1: Remember, static means global state which is evil because it can't be mocked
11 * for tests
12 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
13 */
14 final class StaticFactory
15 {
16     public static function factory(string $type): Formatter
17     {
18         return match ($type) {
19             'number' => new FormatNumber(),
20             'string' => new FormatString(),
21             default => throw new InvalidArgumentException('Unknown format given'),
22         };
23     }
24 }
```

Formatter.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\StaticFactory;
6
7 interface Formatter
8 {
9     public function format(string $input): string;
10 }
```

FormatString.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\StaticFactory;
6
7 class FormatString implements Formatter
8 {
```

(suite sur la page suivante)

(suite de la page précédente)

```

9  public function format(string $input): string
10 {
11     return $input;
12 }
13 }
```

FormatNumber.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\StaticFactory;
6
7 class FormatNumber implements Formatter
8 {
9     public function format(string $input): string
10    {
11        return number_format((int) $input);
12    }
13 }
```

Test

Tests/StaticFactoryTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Creational\StaticFactory\Tests;
6
7 use InvalidArgumentException;
8 use DesignPatterns\Creational\StaticFactory\FormatNumber;
9 use DesignPatterns\Creational\StaticFactory\FormatString;
10 use DesignPatterns\Creational\StaticFactory\StaticFactory;
11 use PHPUnit\Framework\TestCase;
12
13 class StaticFactoryTest extends TestCase
14 {
15     public function testCanCreateNumberFormatter()
16     {
17         $this->assertInstanceOf(FormatNumber::class, StaticFactory::factory('number'));
18     }
19
20     public function testCanCreateStringFormatter()
21     {
22         $this->assertInstanceOf(FormatString::class, StaticFactory::factory('string'));
23     }
24
25     public function testException()
26     {
27         $this->expectException(InvalidArgumentException::class);
28         StaticFactory::factory('invalid');
29     }
30 }
```

(suite sur la page suivante)

(suite de la page précédente)

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

1.2 Structuraux

En ingénierie logicielle, les patrons de conception structuraux facilitent la conception en identifiant une manière simple de réaliser des relations entre les entités.

1.2.1 Adapter / Wrapper

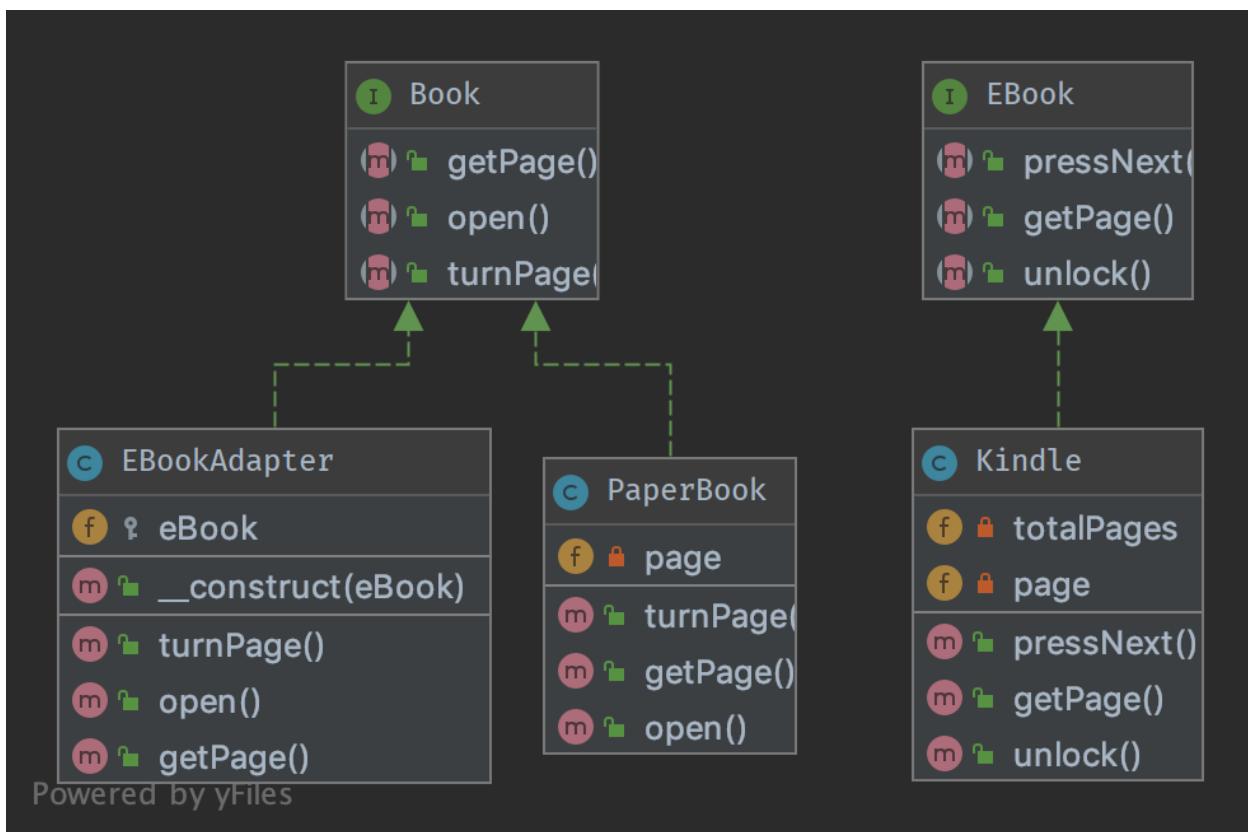
Rôle

Adapter permet de convertir l'interface d'une classe en une autre interface attendue. Il fait fonctionner ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces

Exemples

- DB Client libraries adapter
- Lors de l'utilisation de plusieurs services Web, les adaptateurs permettent de normaliser les données afin que le résultat soit le même pour tous.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Book.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Adapter;
6
7 interface Book
8 {
9     public function turnPage();
10
11    public function open();
12
13    public function getPage(): int;
14 }
  
```

PaperBook.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Adapter;
6
7 class PaperBook implements Book
8 {
9     private int $page;
10
11     public function open(): void
12     {
13         $this->page = 1;
14     }
15
16     public function turnPage(): void
17     {
18         $this->page++;
19     }
20
21     public function getPage(): int
22     {
23         return $this->page;
24     }
25 }
```

EBook.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Adapter;
6
7 interface EBook
8 {
9     public function unlock();
10
11     public function pressNext();
12
13     /**
14      * returns current page and total number of pages, like [10, 100] is page 10 of 100
15      *
16      * @return int[]
17      */
18     public function getPage(): array;
19 }
```

EBookAdapter.php

```
1 <?php
2
3 declare(strict_types=1);
```

(suite sur la page suivante)

(suite de la page précédente)

```

4
5 namespace DesignPatterns\Structural\Adapter;
6
7 /**
8 * This is the adapter here. Notice it implements Book,
9 * therefore you don't have to change the code of the client which is using a Book
10 */
11 class EBookAdapter implements Book
12 {
13     public function __construct(protected EBook $eBook)
14     {
15     }
16
17 /**
18 * This class makes the proper translation from one interface to another.
19 */
20 public function open()
21 {
22     $this->eBook->unlock();
23 }
24
25 public function turnPage()
26 {
27     $this->eBook->pressNext();
28 }
29
30 /**
31 * notice the adapted behavior here: EBook::getPage() will return two integers, but
32 ↵Book
33 * supports only a current page getter, so we adapt the behavior here
34 */
35 public function getPage(): int
36 {
37     return $this->eBook->getPage()[0];
38 }
```

Kindle.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Adapter;
6
7 /**
8 * this is the adapted class. In production code, this could be a class from another
9 ↵package, some vendor code.
10 * Notice that it uses another naming scheme and the implementation does something
11 ↵similar but in another way
12 */
13 class Kindle implements EBook
```

(suite sur la page suivante)

(suite de la page précédente)

```

12 {
13     private int $page = 1;
14     private int $totalPages = 100;
15
16     public function pressNext()
17     {
18         $this->page++;
19     }
20
21     public function unlock()
22     {
23     }
24
25     /**
26      * returns current page and total number of pages, like [10, 100] is page 10 of 100
27      *
28      * @return int[]
29      */
30     public function getPage(): array
31     {
32         return [$this->page, $this->totalPages];
33     }
34 }
```

Test

Tests/AdapterTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Adapter\Tests;
6
7 use DesignPatterns\Structural\Adapter\PaperBook;
8 use DesignPatterns\Structural\Adapter\EBookAdapter;
9 use DesignPatterns\Structural\Adapter\Kindle;
10 use PHPUnit\Framework\TestCase;
11
12 class AdapterTest extends TestCase
13 {
14     public function testCanTurnPageOnBook()
15     {
16         $book = new PaperBook();
17         $book->open();
18         $book->turnPage();
19
20         $this->assertSame(2, $book->getPage());
21     }
22
23     public function testCanTurnPageOnKindleLikeInANormalBook()
```

(suite sur la page suivante)

(suite de la page précédente)

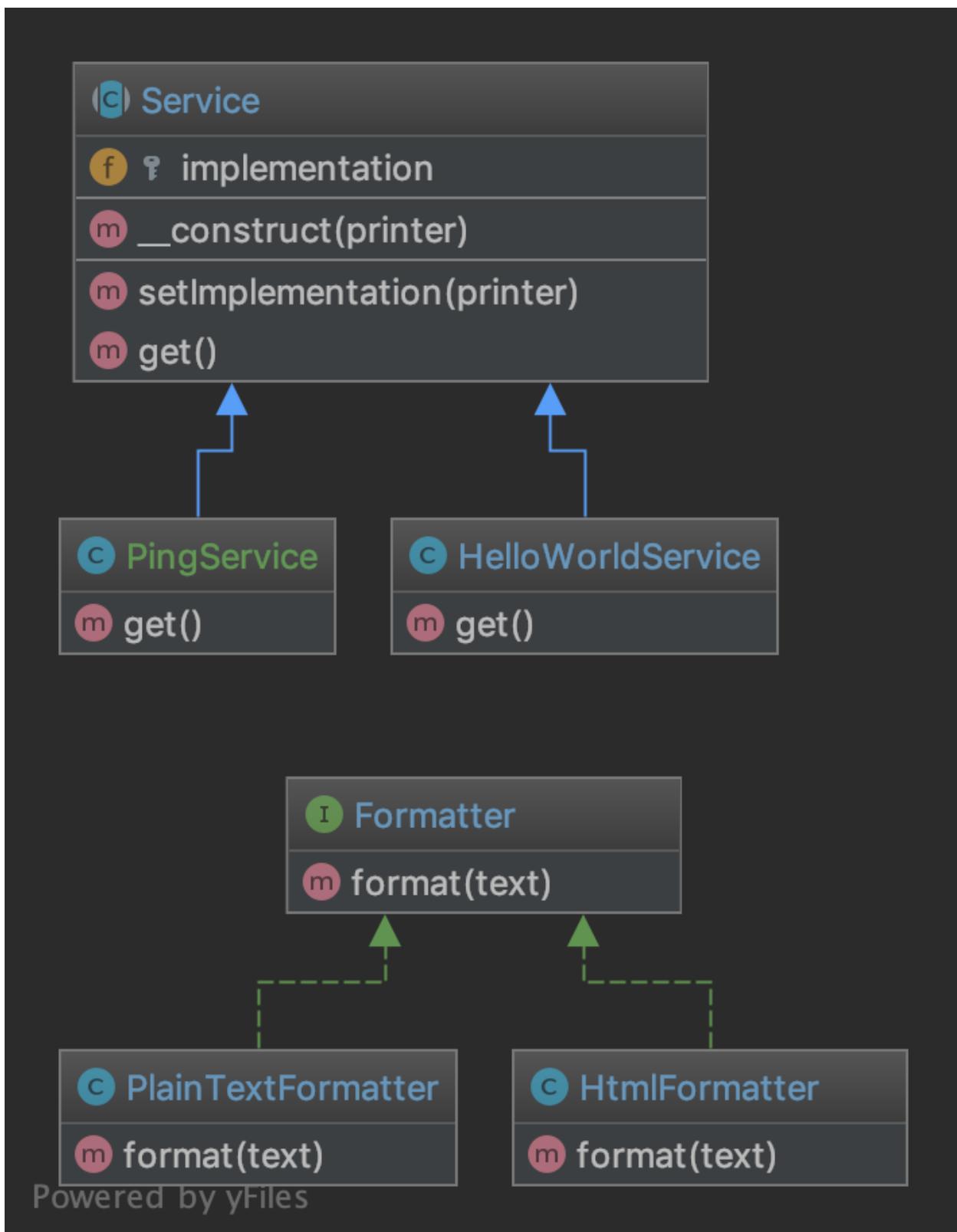
```
24 {  
25     $kindle = new Kindle();  
26     $book = new EBookAdapter($kindle);  
27  
28     $book->open();  
29     $book->turnPage();  
30  
31     $this->assertSame(2, $book->getPage());  
32 }  
33 }
```

1.2.2 Bridge

Rôle

Découpler une abstraction de son implémentation afin que les deux puissent varier indépendamment.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Formatter.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Bridge;
6
7 interface Formatter
8 {
9     public function format(string $text): string;
10}
```

PlainTextFormatter.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Bridge;
6
7 class PlainTextFormatter implements Formatter
8 {
9     public function format(string $text): string
10    {
11        return $text;
12    }
13}
```

HtmlFormatter.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Bridge;
6
7 class HtmlFormatter implements Formatter
8 {
9     public function format(string $text): string
10    {
11        return sprintf('<p>%s</p>', $text);
12    }
13}
```

Service.php

```

1 <?php
2
3 declare(strict_types=1);
```

(suite sur la page suivante)

(suite de la page précédente)

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

Test

Tests/BridgeTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Bridge\Tests;
6
7 use DesignPatterns\Structural\Bridge\HelloWorldService;
8 use DesignPatterns\Structural\Bridge\HtmlFormatter;
9 use DesignPatterns\Structural\Bridge\PlainTextFormatter;
10 use PHPUnit\Framework\TestCase;
11
12 class BridgeTest extends TestCase
13 {
14     public function testCanPrintUsingThePlainTextFormatter()
15     {
16         $service = new HelloWorldService(new PlainTextFormatter());
17
18         $this->assertSame('Hello World', $service->get());
19     }
20
21     public function testCanPrintUsingTheHtmlFormatter()
22     {
23         $service = new HelloWorldService(new HtmlFormatter());
24
25         $this->assertSame('<p>Hello World</p>', $service->get());
26     }
27 }
```

1.2.3 Composite

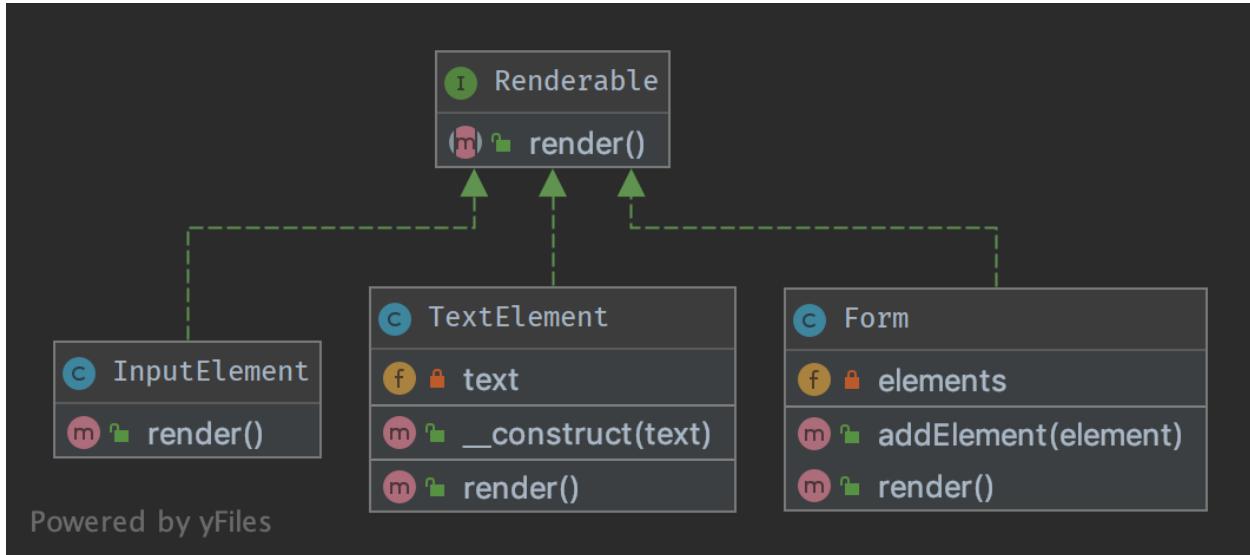
Rôle

Traiter un groupe d'objets de la même manière qu'une seule instance.

Exemples

- Une instance de classe de formulaire traitant tous ses éléments de formulaire comme une seule instance du formulaire. Lorsque `render()` est appelé, il passe ensuite par tous ses éléments enfants et appelle `render()` sur eux.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Renderable.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Composite;
6
7 interface Renderable
8 {
9     public function render(): string;
10 }
  
```

Form.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Composite;
6
7 /**
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

8 * The composite node MUST extend the component contract. This is mandatory for building
9 * a tree of components.
10 */
11 class Form implements Renderable
12 {
13     /**
14      * @var Renderable[]
15      */
16     private array $elements;
17
18     /**
19      * runs through all elements and calls render() on them, then returns the complete
20      * representation
21      * of the form.
22      *
23      * from the outside, one will not see this and the form will act like a single
24      * object instance
25      */
26     public function render(): string
27     {
28         $formCode = '<form>';
29
30         foreach ($this->elements as $element) {
31             $formCode .= $element->render();
32         }
33
34         return $formCode . '</form>';
35     }
36
37     public function addElement(Renderable $element)
38     {
39         $this->elements[] = $element;
40     }
41 }
```

InputElement.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Composite;
6
7 class InputElement implements Renderable
8 {
9     public function render(): string
10    {
11        return '<input type="text" />';
12    }
13 }
```

TextElement.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Composite;
6
7 class TextElement implements Renderable
8 {
9     public function __construct(private string $text)
10    {
11    }
12
13     public function render(): string
14    {
15         return $this->text;
16    }
17 }
```

Test

Tests/CompositeTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Composite\Tests;
6
7 use DesignPatterns\Structural\Composite\Form;
8 use DesignPatterns\Structural\Composite\TextElement;
9 use DesignPatterns\Structural\Composite\InputElement;
10 use PHPUnit\Framework\TestCase;
11
12 class CompositeTest extends TestCase
13 {
14     public function testRender()
15     {
16         $form = new Form();
17         $form->addElement(new TextElement('Email:'));
18         $form->addElement(new InputElement());
19         $embed = new Form();
20         $embed->addElement(new TextElement('Password:'));
21         $embed->addElement(new InputElement());
22         $form->addElement($embed);
23
24         // This is just an example, in a real world scenario it is important to remember
25         // that web browsers do not
26         // currently support nested forms
27
28         $this->assertSame(
29             '<form>Email:<input type="text" /><form>Password:<input type="text" /></form>
30             </form>',
31         );
32     }
33 }
```

(suite sur la page suivante)

(suite de la page précédente)

```
29         $form->render()
30     );
31 }
32 }
```

1.2.4 Data Mapper

Rôle

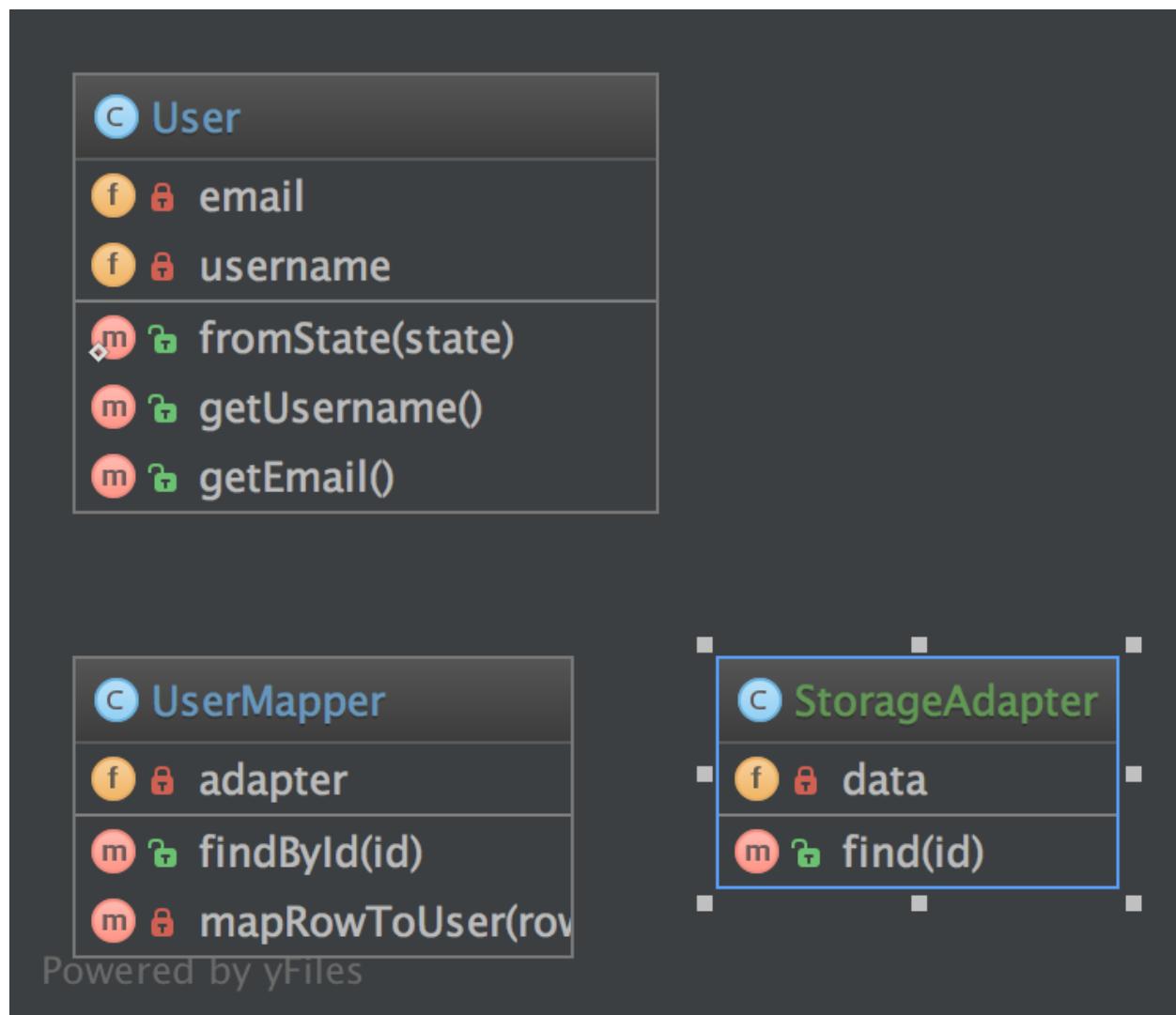
Un Data Mapper est une couche d'accès aux données qui effectue un transfert bidirectionnel de données entre un stockage de données permanent (souvent une base de données relationnelle) et une représentation de données en mémoire (la couche de domaine). L'objectif de ce modèle est de faire en sorte que la représentation en mémoire et le stock de données permanent soient indépendants l'un de l'autre et du Data Mapper lui-même. La couche est composée d'un ou plusieurs mappeurs (ou objets d'accès aux données), qui effectuent le transfert de données. Les mappeurs génériques traiteront de nombreux types d'entités de domaine différents, les mappeurs spécialisés n'en traiteront qu'un ou quelques-uns.

Le point clé de ce modèle est que, contrairement au pattern [Active Record](https://en.wikipedia.org/wiki/Active_record_pattern), le modèle de données suit le principe de responsabilité unique.

Examples

- DB Object Relational Mapper (ORM) : Doctrine2 utilise un DAO nommé « EntityRepository ».

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

User.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DataMapper;
6
7 class User
8 {
9     public static function fromState(array $state): User
10    {

```

(suite sur la page suivante)

(suite de la page précédente)

```

11 // validate state before accessing keys!
12
13     return new self(
14         $state['username'],
15         $state['email']
16     );
17 }
18
19 public function __construct(private string $username, private string $email)
20 {
21 }
22
23 public function getUsername(): string
24 {
25     return $this->username;
26 }
27
28 public function getEmail(): string
29 {
30     return $this->email;
31 }
32 }
```

UserMapper.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DataMapper;
6
7 use InvalidArgumentException;
8
9 class UserMapper
10 {
11     public function __construct(private StorageAdapter $adapter)
12     {
13     }
14
15     /**
16      * finds a user from storage based on ID and returns a User object located
17      * in memory. Normally this kind of logic will be implemented using the Repository_
18      ↵pattern.
19      * However the important part is in mapRowToUser() below, that will create a_
20      ↵business object from the
21      * data fetched from storage
22      */
23     public function findById(int $id): User
24     {
25         $result = $this->adapter->find($id);
26
27         if ($result === null) {
```

(suite sur la page suivante)

(suite de la page précédente)

```

26     throw new InvalidArgumentException("User #\$id not found");
27 }
28
29     return $this->mapRowToUser($result);
30 }
31
32 private function mapRowToUser(array $row): User
33 {
34     return User::fromState($row);
35 }
36 }
```

StorageAdapter.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DataMapper;
6
7 class StorageAdapter
8 {
9     public function __construct(private array $data)
10    {
11    }
12
13 /**
14 * @return array|null
15 */
16 public function find(int $id)
17 {
18     if (isset($this->data[$id])) {
19         return $this->data[$id];
20     }
21
22     return null;
23 }
24 }
```

Test

Tests/DataMapperTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DataMapper\Tests;
6
7 use InvalidArgumentException;
8 use DesignPatterns\Structural\DataMapper\StorageAdapter;
```

(suite sur la page suivante)

(suite de la page précédente)

```
9  use DesignPatterns\Structural\DataMapper\User;
10 use DesignPatterns\Structural\DataMapper\UserMapper;
11 use PHPUnit\Framework\TestCase;
12
13 class DataMapperTest extends TestCase
14 {
15     public function testCanMapUserFromStorage()
16     {
17         $storage = new StorageAdapter([1 => ['username' => 'someone', 'email' =>
18             'someone@example.com']]);
19         $mapper = new UserMapper($storage);
20
21         $user = $mapper->findById(1);
22
23         $this->assertInstanceOf(User::class, $user);
24     }
25
26     public function testWillNotMapInvalidData()
27     {
28         $this->expectException(InvalidArgumentException::class);
29
30         $storage = new StorageAdapter([]);
31         $mapper = new UserMapper($storage);
32
33         $mapper->findById(1);
34     }
}
```

1.2.5 Decorator

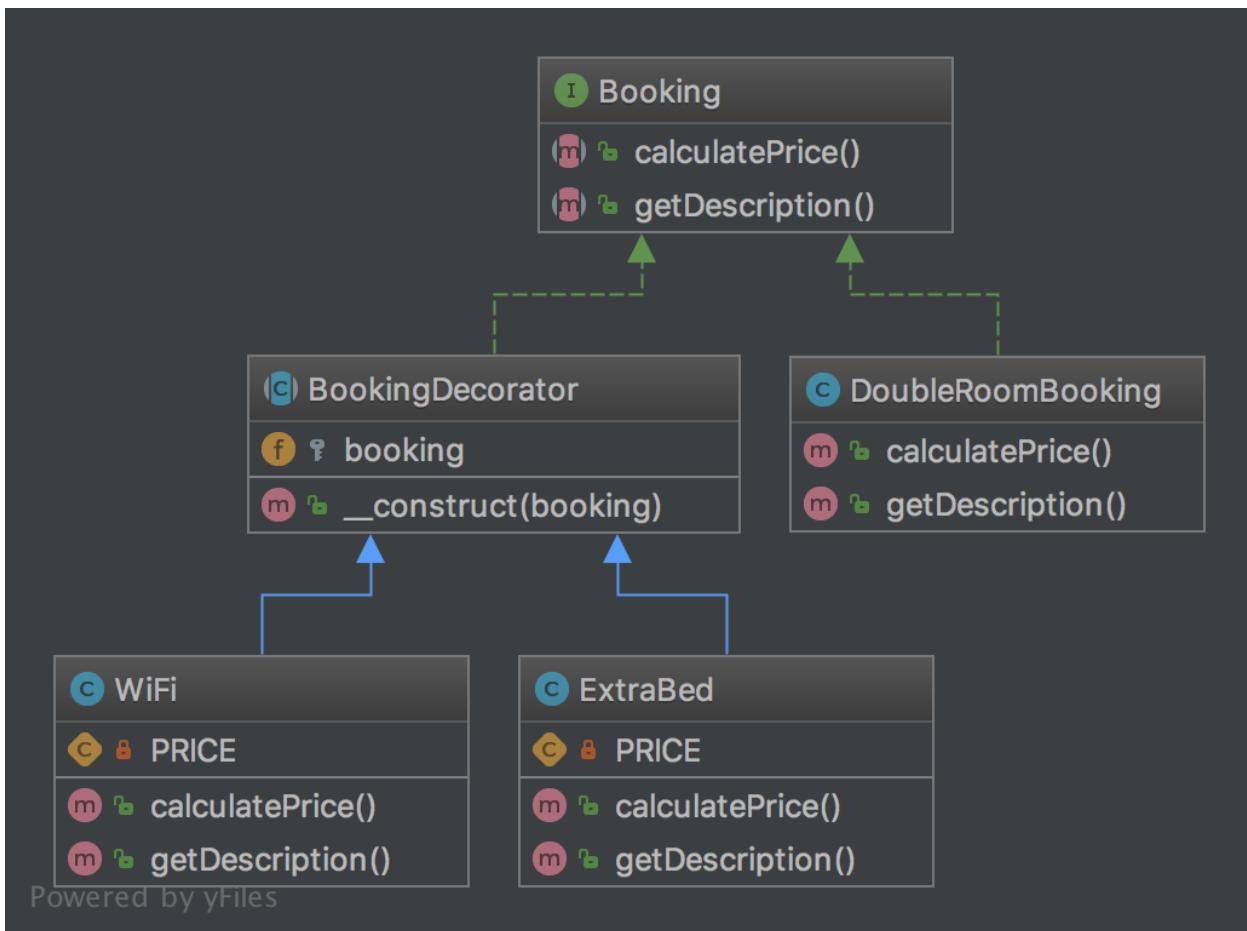
Rôle

Ajoute dynamiquement de nouvelles fonctionnalités aux instances de classe.

Exemples

- Couche de service web : Décorateurs JSON et XML pour un service REST (dans ce cas, un seul d'entre eux doit être autorisé bien sûr)

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Booking.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator;
6
7 interface Booking
8 {
9     public function calculatePrice(): int;
10
11    public function getDescription(): string;
12 }
  
```

BookingDecorator.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator;
6
7 abstract class BookingDecorator implements Booking
8 {
9     public function __construct(protected Booking $booking)
10    {
11    }
12 }

```

DoubleRoomBooking.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator;
6
7 class DoubleRoomBooking implements Booking
8 {
9     public function calculatePrice(): int
10    {
11        return 40;
12    }
13
14     public function getDescription(): string
15    {
16        return 'double room';
17    }
18 }

```

ExtraBed.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator;
6
7 class ExtraBed extends BookingDecorator
8 {
9     private const PRICE = 30;
10
11     public function calculatePrice(): int
12    {
13        return $this->booking->calculatePrice() + self::PRICE;
14    }
15
16     public function getDescription(): string
17    {

```

(suite sur la page suivante)

(suite de la page précédente)

```

18     return $this->booking->getDescription() . ' with extra bed';
19 }
20 }
```

WiFi.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator;
6
7 class WiFi extends BookingDecorator
8 {
9     private const PRICE = 2;
10
11    public function calculatePrice(): int
12    {
13        return $this->booking->calculatePrice() + self::PRICE;
14    }
15
16    public function getDescription(): string
17    {
18        return $this->booking->getDescription() . ' with wifi';
19    }
20 }
```

Test

Tests/DecoratorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Decorator\Tests;
6
7 use DesignPatterns\Structural\Decorator\DoubleRoomBooking;
8 use DesignPatterns\Structural\Decorator\ExtraBed;
9 use DesignPatterns\Structural\Decorator\WiFi;
10 use PHPUnit\Framework\TestCase;
11
12 class DecoratorTest extends TestCase
13 {
14     public function testCanCalculatePriceForBasicDoubleRoomBooking()
15     {
16         $booking = new DoubleRoomBooking();
17
18         $this->assertSame(40, $booking->calculatePrice());
19         $this->assertSame('double room', $booking->getDescription());
20     }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

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->
39             getDescription());
40     }

```

1.2.6 Dependency Injection

Rôle

Implémenter une architecture faiblement couplée afin d'obtenir une meilleur testabilité, maintenabilité et extensibilité du code.

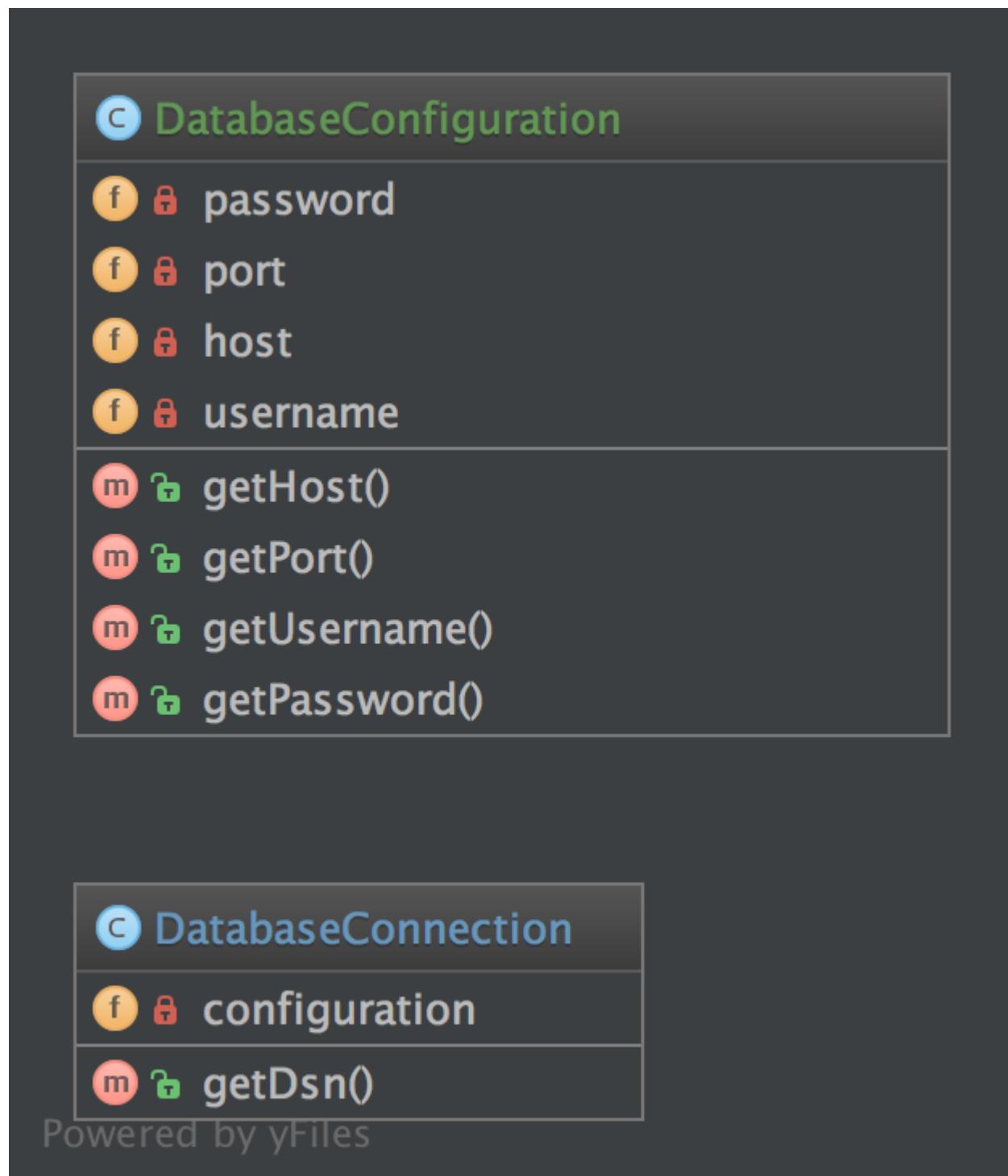
Usage

`DatabaseConfiguration` est injectée et `DatabaseConnexion` obtiendra tout ce dont elle a besoin à partir de `$config`. Sans DI, la configuration serait créée directement dans `DatabaseConnexion`, ce qui n'est pas très bon pour les tests et les extensions.

Exemples

- L'ORM Doctrine2 utilise l'injection de dépendances, par exemple pour la configuration qui est injectée dans un objet `Connexion`. À des fins de test, on peut facilement créer un objet fantaisie de la configuration et l'injecter dans l'objet `Connexion`.
- De nombreux frameworks disposent déjà de conteneurs pour DI qui créent des objets via un tableau de configuration et les injectent là où c'est nécessaire (c'est-à-dire dans les contrôleur).

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

DatabaseConfiguration.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DependencyInjection;
6
7 class DatabaseConfiguration
8 {
9     public function __construct(
10         private string $host,
11         private int $port,
12         private string $username,
13         private string $password
14     ) {
15     }
16
17     public function getHost(): string
18     {
19         return $this->host;
20     }
21
22     public function getPort(): int
23     {
24         return $this->port;
25     }
26
27     public function getUsername(): string
28     {
29         return $this->username;
30     }
31
32     public function getPassword(): string
33     {
34         return $this->password;
35     }
36 }
```

DatabaseConnection.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DependencyInjection;
6
7 class DatabaseConnection
8 {
9     public function __construct(private DatabaseConfiguration $configuration)
10 }
```

(suite sur la page suivante)

(suite de la page précédente)

```
10  {
11  }
12
13  public function getDsn(): string
14  {
15      // this is just for the sake of demonstration, not a real DSN
16      // notice that only the injected config is used here, so there is
17      // a real separation of concerns here
18
19      return sprintf(
20          '%s:%s@%s:%d',
21          $this->configuration->getUsername(),
22          $this->configuration->getPassword(),
23          $this->configuration->getHost(),
24          $this->configuration->getPort()
25      );
26  }
27 }
```

Test

Tests/DependencyInjectionTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\DependencyInjection\Tests;
6
7 use DesignPatterns\Structural\DependencyInjection\DatabaseConfiguration;
8 use DesignPatterns\Structural\DependencyInjection\DatabaseConnection;
9 use PHPUnit\Framework\TestCase;
10
11 class DependencyInjectionTest extends TestCase
12 {
13     public function testDependencyInjection()
14     {
15         $config = new DatabaseConfiguration('localhost', 3306, 'user', '1234');
16         $connection = new DatabaseConnection($config);
17
18         $this->assertSame('user:1234@localhost:3306', $connection->getDsn());
19     }
20 }
```

1.2.7 Facade

Rôle

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

Une façade est destinée à découpler un client et un système secondaire en intégrant plusieurs (ou parfois une seule) interfaces, et bien sûr à réduire la complexité.

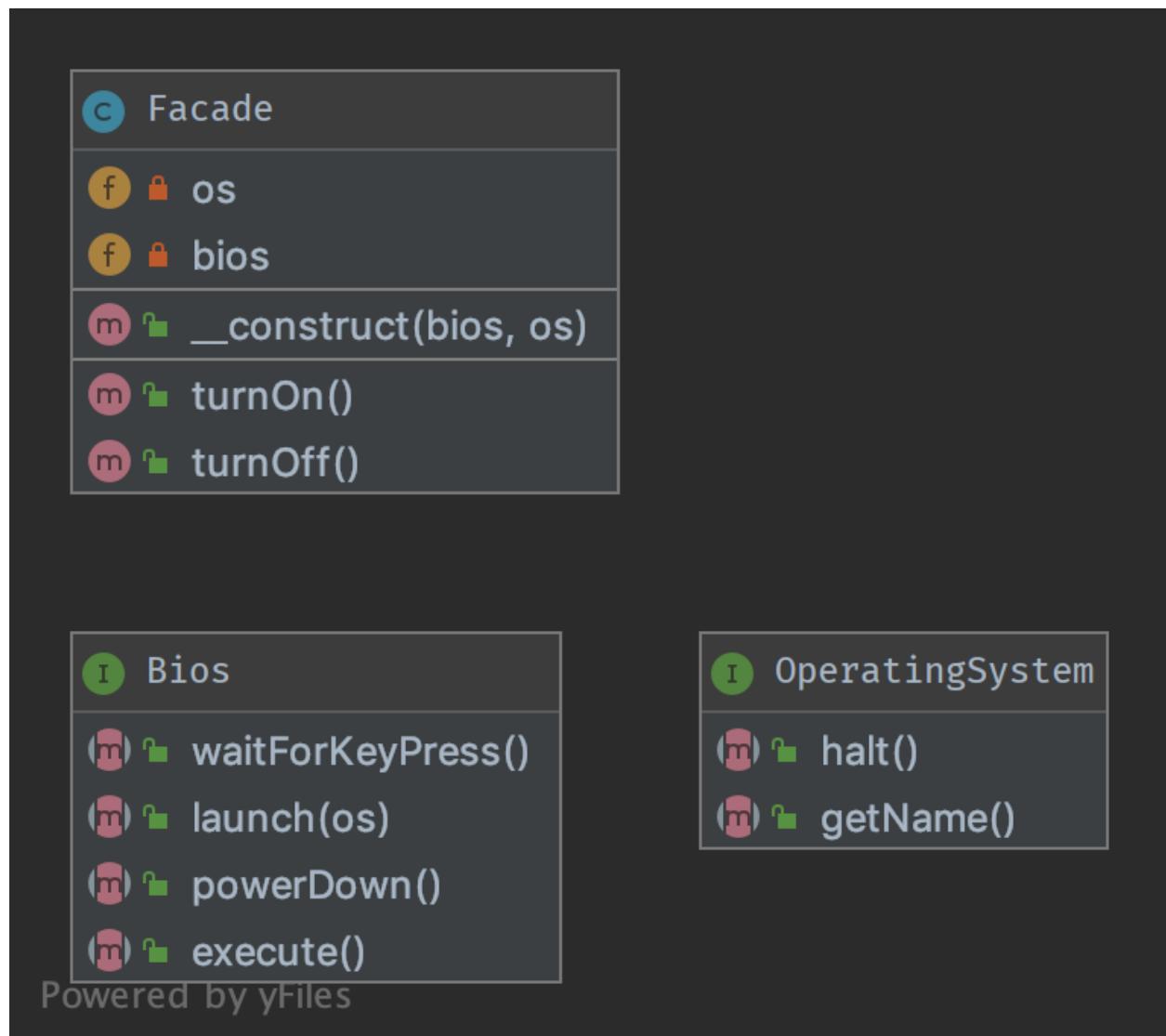
- Une façade ne vous interdit pas l'accès au sous-système.

- Vous pouvez (vous devriez) avoir plusieurs façades pour un seul sous-système.

C'est pourquoi une bonne façade n'a pas de `new`. S'il y a plusieurs créations pour chaque méthode, ce n'est pas une façade, c'est un Builder ou une [méthode] d'usine [abstraite|statique|simple].

La meilleure façade n'a pas de `new` et un constructeur avec des paramètres indiquant le type d'interface. Si vous avez besoin de créer de nouvelles instances, utilisez un Factory comme argument.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Facade.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Facade;
6
7 class Facade
8 {
9     public function __construct(private Bios $bios, private OperatingSystem $os)
10    {
11    }
12
13    public function turnOn()
14    {
15        $this->bios->execute();
16        $this->bios->waitForKeyPress();
17        $this->bios->launch($this->os);
18    }
19
20    public function turnOff()
21    {
22        $this->os->halt();
23        $this->bios->powerDown();
24    }
25 }
```

OperatingSystem.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Facade;
6
7 interface OperatingSystem
8 {
9     public function halt();
10
11     public function getName(): string;
12 }
```

Bios.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Facade;
```

(suite sur la page suivante)

(suite de la page précédente)

```

6 interface Bios
7 {
8     public function execute();
9
10    public function waitForKeyPress();
11
12    public function launch(OperatingSystem $os);
13
14    public function powerDown();
15
16 }

```

Test

Tests/FacadeTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Facade\Tests;
6
7 use DesignPatterns\Structural\Facade\Bios;
8 use DesignPatterns\Structural\Facade\Facade;
9 use DesignPatterns\Structural\Facade\OperatingSystem;
10 use PHPUnit\Framework\TestCase;
11
12 class FacadeTest extends TestCase
13 {
14     public function testComputerOn()
15     {
16         $os = $this->createMock(OperatingSystem::class);
17
18         $os->method('getName')
19             ->will($this->returnValue('Linux'));
20
21         $bios = $this->createMock(Bios::class);
22
23         $bios->method('launch')
24             ->with($os);
25
26         /** @noinspection PhpParamsInspection */
27         $facade = new Facade($bios, $os);
28         $facade->turnOn();
29
30         $this->assertSame('Linux', $os->getName());
31     }
32 }

```

1.2.8 Fluent Interface

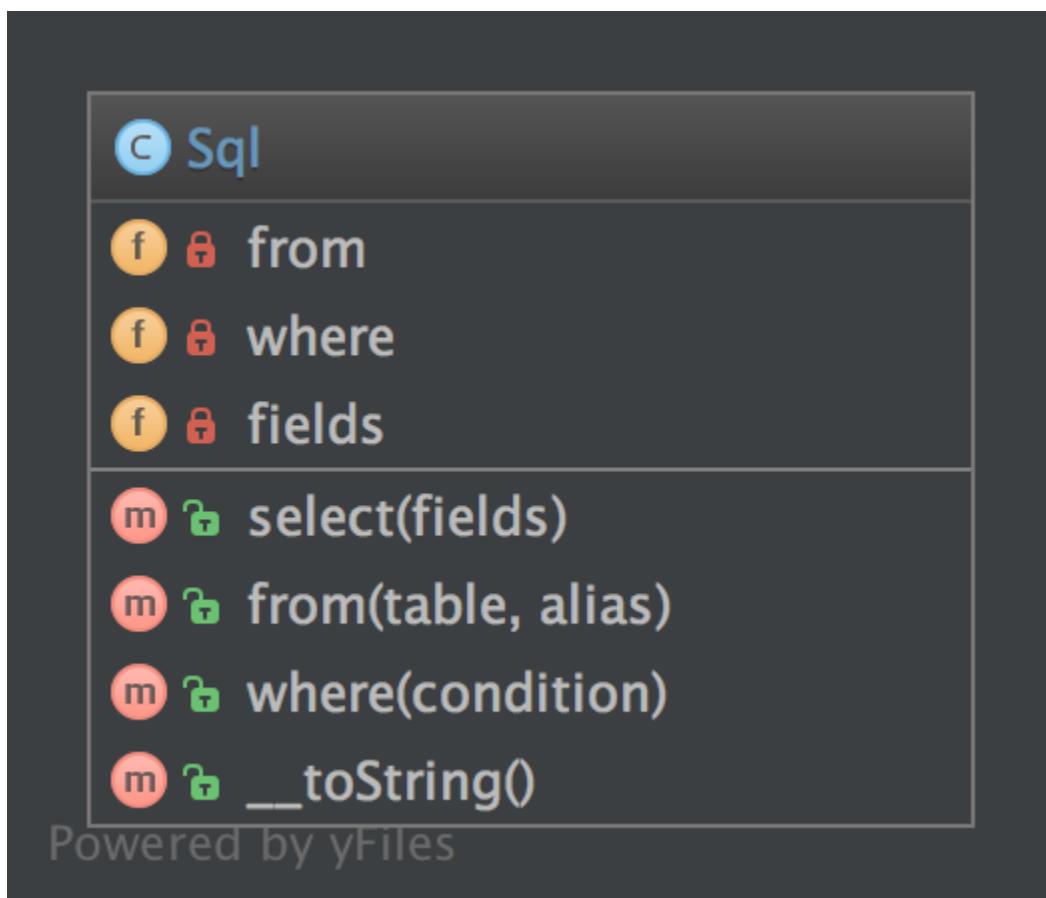
Rôle

Écrire du code qui soit facilement lisible, tout comme les phrases d'une langue naturelle (comme l'anglais).

Exemples

- Le QueryBuilder de Doctrine2 fonctionne comme la classe d'exemple ci-dessous.
- PHPUnit utilise Fluent Interface pour construire des mocks.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Sql.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\FluentInterface;
6
7 class Sql implements \Stringable
8 {
9     private array $fields = [];
10    private array $from = [];
11    private array $where = [];
12
13    public function select(array $fields): Sql
14    {
15        $this->fields = $fields;
16
17        return $this;
18    }
19
20    public function from(string $table, string $alias): Sql
21    {
22        $this->from[] = $table . ' AS ' . $alias;
23
24        return $this;
25    }
26
27    public function where(string $condition): Sql
28    {
29        $this->where[] = $condition;
30
31        return $this;
32    }
33
34    public function __toString(): string
35    {
36        return sprintf(
37            'SELECT %s FROM %s WHERE %s',
38            join(', ', $this->fields),
39            join(', ', $this->from),
40            join(' AND ', $this->where)
41        );
42    }
43}
```

Test

Tests/FluentInterfaceTest.php

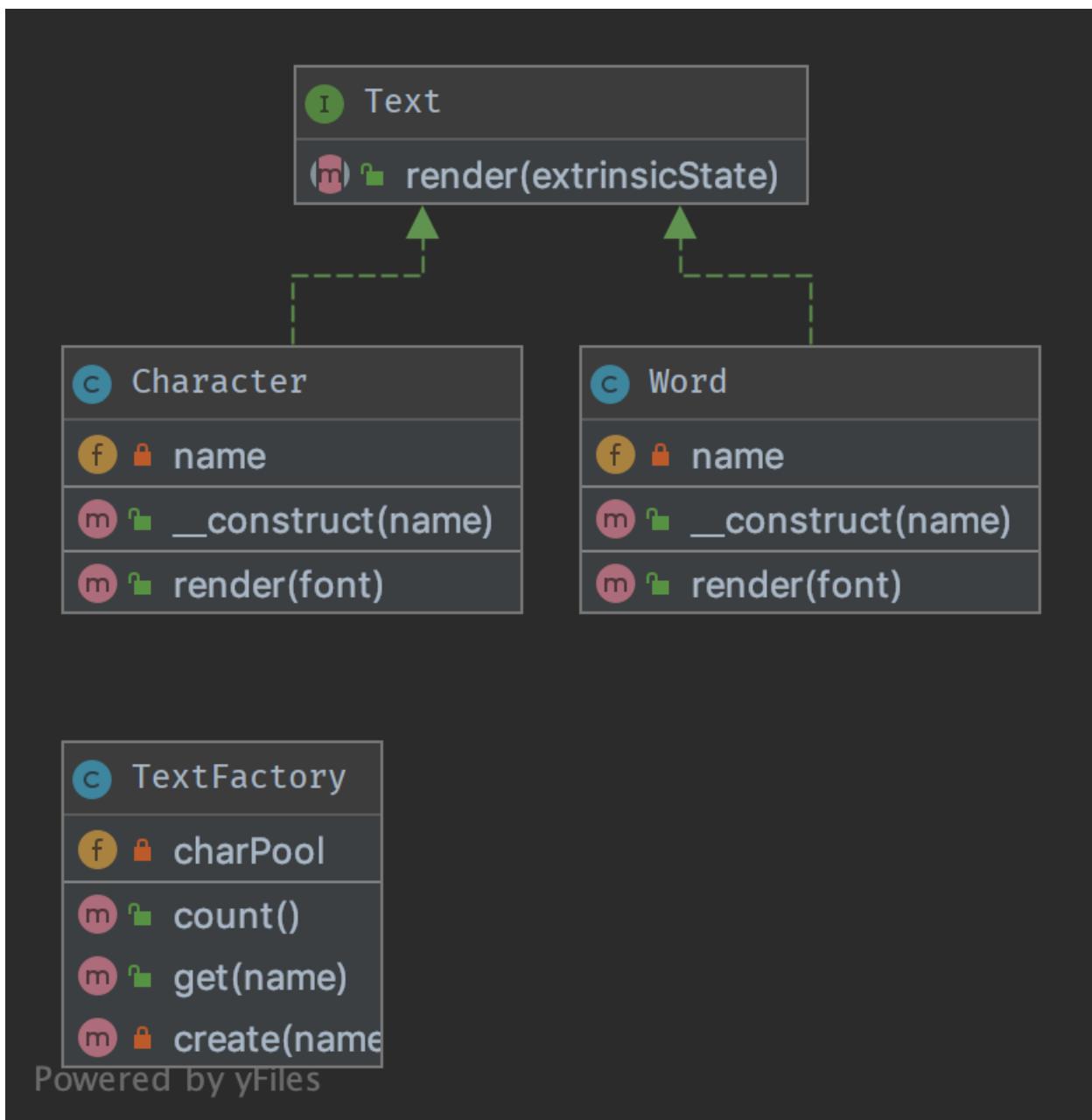
```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\FluentInterface\Tests;
6
7 use DesignPatterns\Structural\FluentInterface\Sql;
8 use PHPUnit\Framework\TestCase;
9
10 class FluentInterfaceTest extends TestCase
11 {
12     public function testBuildSQL()
13     {
14         $query = (new Sql())
15             ->select(['foo', 'bar'])
16             ->from('foobar', 'f')
17             ->where('f.bar = ?');
18
19         $this->assertSame('SELECT foo, bar FROM foobar AS f WHERE f.bar = ?', (string)
20             $query);
21     }
21 }
```

1.2.9 Flyweight

Rôle

Pour minimiser l'utilisation de la mémoire, un Flyweight partage autant que possible la mémoire avec des objets similaires. Cela est nécessaire lorsqu'on utilise un grand nombre d'objets dont l'état ne diffère pas beaucoup. Une pratique courante consiste à conserver l'état dans des structures de données externes et à les transmettre à l'objet Flyweight en cas de besoin.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Text.php

```

1 <?php
2
3 declare(strict_types=1);
4
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

5 namespace DesignPatterns\Structural\Flyweight;
6
7 /**
8 * This is the interface that all flyweights need to implement
9 */
10 interface Text
11 {
12     public function render(string $extrinsicState): string;
13 }
```

Word.php

```

1 <?php
2
3 namespace DesignPatterns\Structural\Flyweight;
4
5 class Word implements Text
6 {
7     public function __construct(private string $name)
8     {
9     }
10
11    public function render(string $extrinsicState): string
12    {
13        return sprintf('Word %s with font %s', $this->name, $extrinsicState);
14    }
15 }
```

Character.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Flyweight;
6
7 /**
8 * Implements the flyweight interface and adds storage for intrinsic state, if any.
9 * Instances of concrete flyweights are shared by means of a factory.
10 */
11 class Character implements Text
12 {
13     /**
14      * Any state stored by the concrete flyweight must be independent of its context.
15      * For flyweights representing characters, this is usually the corresponding
16      ↵character code.
17      */
18     public function __construct(private string $name)
19     {
20     }
21
22     public function render(string $extrinsicState): string
```

(suite sur la page suivante)

(suite de la page précédente)

```

22     {
23         // Clients supply the context-dependent information that the flyweight needs to
24         // draw itself
25         // For flyweights representing characters, extrinsic state usually contains e.g.
26         // the font.
27
28         return sprintf('Character %s with font %s', $this->name, $extrinsicState);
    }
}

```

TextFactory.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Flyweight;
6
7 use Countable;
8
9 /**
10 * A factory manages shared flyweights. Clients should not instantiate them directly,
11 * but let the factory take care of returning existing objects or creating new ones.
12 */
13 class TextFactory implements Countable
14 {
15     /**
16      * @var Text[]
17      */
18     private array $charPool = [];
19
20     public function get(string $name): Text
21     {
22         if (!isset($this->charPool[$name])) {
23             $this->charPool[$name] = $this->create($name);
24         }
25
26         return $this->charPool[$name];
27     }
28
29     private function create(string $name): Text
30     {
31         if (strlen($name) == 1) {
32             return new Character($name);
33         }
34         return new Word($name);
35     }
36
37     public function count(): int
38     {
39         return count($this->charPool);
40     }
}

```

(suite sur la page suivante)

(suite de la page précédente)

41 }

Test

Tests/FlyweightTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Flyweight\Tests;
6
7 use DesignPatterns\Structural\Flyweight\TextFactory;
8 use PHPUnit\Framework\TestCase;
9
10 class FlyweightTest extends TestCase
11 {
12     private array $characters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
13         'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'];
14
15     private array $fonts = ['Arial', 'Times New Roman', 'Verdana', 'Helvetica'];
16
17     public function testFlyweight()
18     {
19         $factory = new TextFactory();
20
21         for ($i = 0; $i <= 10; $i++) {
22             foreach ($this->characters as $char) {
23                 foreach ($this->fonts as $font) {
24                     $flyweight = $factory->get($char);
25                     $rendered = $flyweight->render($font);
26
27                     $this->assertSame(sprintf('Character %s with font %s', $char, $font),
28                         $rendered);
29                 }
30             }
31         }
32
33         foreach ($this->fonts as $word) {
34             $flyweight = $factory->get($word);
35             $rendered = $flyweight->render('foobar');
36
37             $this->assertSame(sprintf('Word %s with font foobar', $word), $rendered);
38         }
39
40         // Flyweight pattern ensures that instances are shared
41         // instead of having hundreds of thousands of individual objects
42         // there must be one instance for every char that has been reused for displaying
43         // in different fonts
44         $this->assertCount(count($this->characters) + count($this->fonts), $factory);
45     }

```

(suite sur la page suivante)

(suite de la page précédente)

44

}

1.2.10 Proxy

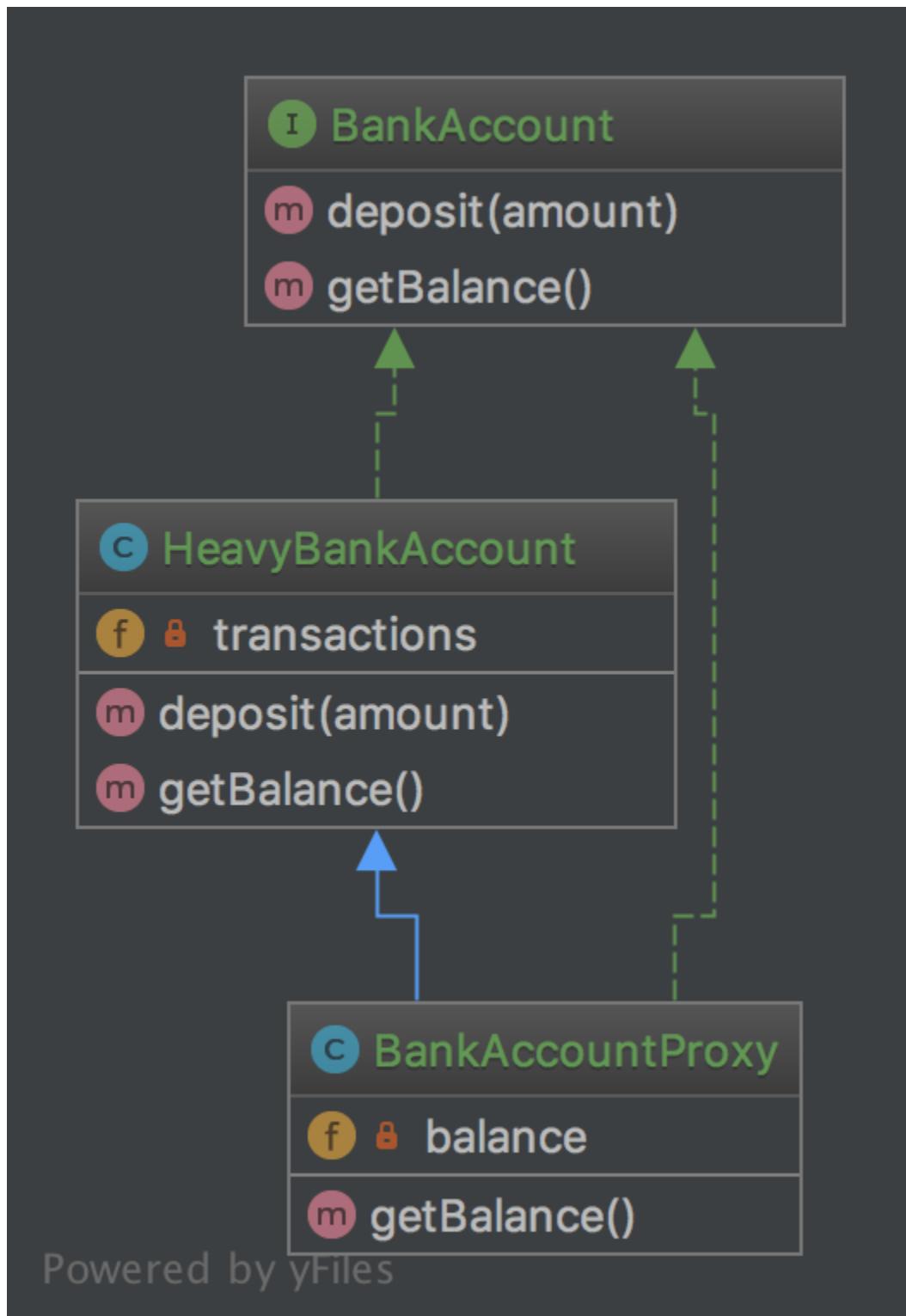
Rôle

Créer une interface pour interagir avec toute classe qu'il est difficile ou impossible d'utiliser dans sa forme originale.

Exemples

- Doctrine2 utilise des proxies pour y implémenter la magie du framework (e.g. lazy initialization), tandis que l'utilisateur travaille toujours avec ses propres entités et n'utilisera ni ne touchera jamais les proxies

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

BankAccount.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Proxy;
6
7 interface BankAccount
8 {
9     public function deposit(int $amount);
10
11    public function getBalance(): int;
12 }
```

HeavyBankAccount.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Proxy;
6
7 class HeavyBankAccount implements BankAccount
8 {
9     /**
10      * @var int[]
11      */
12     private array $transactions = [];
13
14     public function deposit(int $amount)
15     {
16         $this->transactions[] = $amount;
17     }
18
19     public function getBalance(): int
20     {
21         // this is the heavy part, imagine all the transactions even from
22         // years and decades ago must be fetched from a database or web service
23         // and the balance must be calculated from it
24
25         return array_sum($this->transactions);
26     }
27 }
```

BankAccountProxy.php

```

1 <?php
2
3 declare(strict_types=1);
```

(suite sur la page suivante)

(suite de la page précédente)

```

4
5 namespace DesignPatterns\Structural\Proxy;
6
7 class BankAccountProxy extends HeavyBankAccount implements BankAccount
8 {
9     private ?int $balance = null;
10
11    public function getBalance(): int
12    {
13        // because calculating balance is so expensive,
14        // the usage of BankAccount::getBalance() is delayed until it really is needed
15        // and will not be calculated again for this instance
16
17        if ($this->balance === null) {
18            $this->balance = parent::getBalance();
19        }
20
21        return $this->balance;
22    }
23}

```

Test

ProxyTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Proxy\Tests;
6
7 use DesignPatterns\Structural\Proxy\BankAccountProxy;
8 use PHPUnit\Framework\TestCase;
9
10 class ProxyTest extends TestCase
11 {
12     public function testProxyWillOnlyExecuteExpensiveGetBalanceOnce()
13     {
14         $bankAccount = new BankAccountProxy();
15         $bankAccount->deposit(30);
16
17         // this time balance is being calculated
18         $this->assertSame(30, $bankAccount->getBalance());
19
20         // inheritance allows for BankAccountProxy to behave to an outsider exactly like
21         →ServerBankAccount
22         $bankAccount->deposit(50);
23
24         // this time the previously calculated balance is returned again without re-
25         →calculating it
26         $this->assertSame(30, $bankAccount->getBalance());

```

(suite sur la page suivante)

(suite de la page précédente)

```

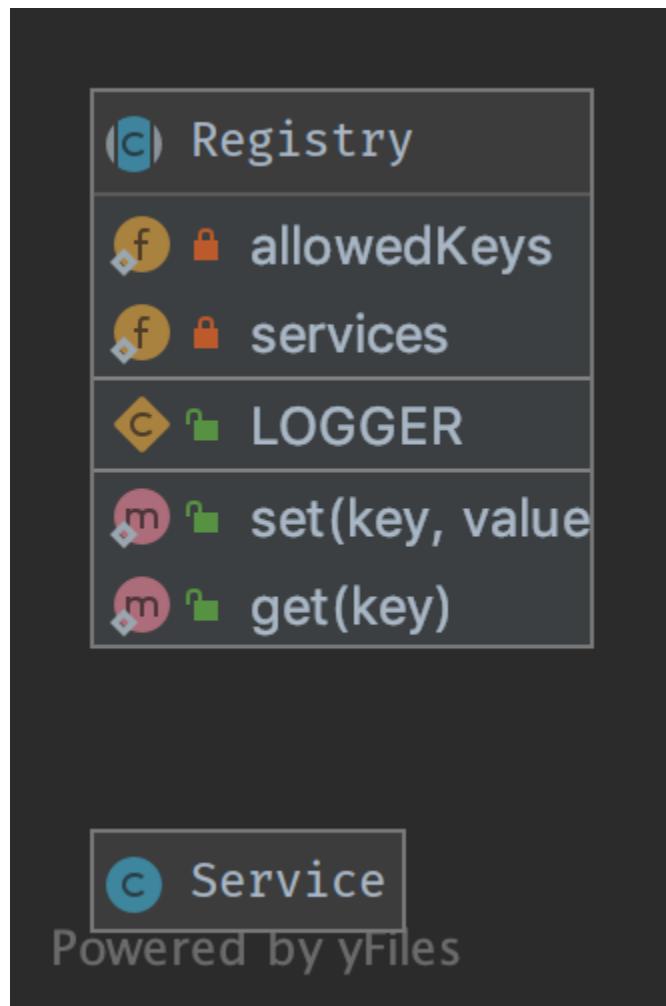
25     }
26 }
```

1.2.11 Registry

Rôle

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

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Registry.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Registry;
6
7 use InvalidArgumentException;
8
9 abstract class Registry
10 {
11     public const LOGGER = 'logger';
12
13     /**
14      * this introduces global state in your application which can not be mocked up for
15      * testing
16      * and is therefore considered an anti-pattern! Use dependency injection instead!
17      *
18      * @var Service[]
19      */
20     private static array $services = [];
21
22     private static array $allowedKeys = [
23         self::LOGGER,
24     ];
25
26     final public static function set(string $key, Service $value)
27     {
28         if (!in_array($key, self::$allowedKeys)) {
29             throw new InvalidArgumentException('Invalid key given');
30         }
31
32         self::$services[$key] = $value;
33     }
34
35     final public static function get(string $key): Service
36     {
37         if (!in_array($key, self::$allowedKeys) || !isset(self::$services[$key])) {
38             throw new InvalidArgumentException('Invalid key given');
39         }
40
41         return self::$services[$key];
42     }
43 }
```

Service.php

```

1 <?php
2
```

(suite sur la page suivante)

(suite de la page précédente)

```

3 namespace DesignPatterns\Structural\Registry;
4
5 class Service
6 {
7 }
```

Test

Tests/RegistryTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Structural\Registry\Tests;
6
7 use InvalidArgumentException;
8 use DesignPatterns\Structural\Registry\Registry;
9 use DesignPatterns\Structural\Registry\Service;
10 use PHPUnit\Framework\TestCase;
11
12 class RegistryTest extends TestCase
13 {
14     private Service $service;
15
16     protected function setUp(): void
17     {
18         $this->service = $this->getMockBuilder(Service::class)->getMock();
19     }
20
21     public function testSetAndGetLogger()
22     {
23         Registry::set(Registry::LOGGER, $this->service);
24
25         $this->assertSame($this->service, Registry::get(Registry::LOGGER));
26     }
27
28     public function testThrowsExceptionWhenTryingToSetInvalidKey()
29     {
30         $this->expectException(InvalidArgumentException::class);
31
32         Registry::set('foobar', $this->service);
33     }
34
35 /**
36 * notice @runInSeparateProcess here: without it, a previous test might have set it
37 * already and
38 * testing would not be possible. That's why you should implement Dependency
39 * Injection where an
40 * injected class may easily be replaced by a mockup
41 *
```

(suite sur la page suivante)

(suite de la page précédente)

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

1.3 Comportementaux

En ingénierie logicielle, les patrons de conception comportementaux identifient les modèles de communication communs entre les objets et réalisent ces modèles. Ce faisant, ces patrons augmentent la flexibilité dans la réalisation de cette communication.

1.3.1 Chain Of Responsibilities

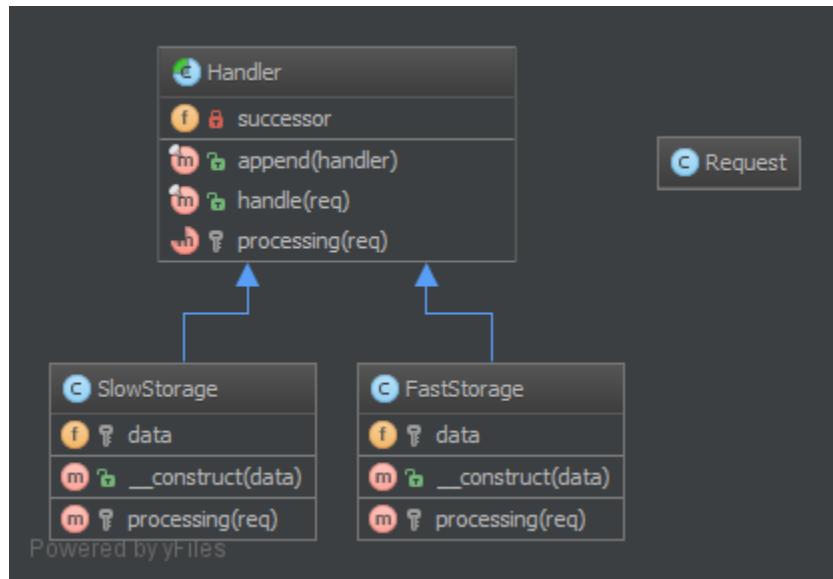
Purpose

Construire une chaîne d'objets pour traiter un appel dans un ordre séquentiel. Si un objet ne peut pas traiter un appel, il délègue l'appel au suivant dans la chaîne et ainsi de suite.

Examples

- Logging Framework, dans lequel chaque élément de la chaîne décide lui-même de ce qu'il faut faire avec un message de journalisation.
- Filtre anti-spam
- Mise en cache : le premier objet est, par exemple, une instance d'une interface Memcached, s'il « manque », il délègue l'appel à l'interface de base de données.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Handler.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\ChainOfResponsibilities;
6
7 use Psr\Http\Message\RequestInterface;
8
9 abstract class Handler
10 {
11     public function __construct(private ?Handler $successor = null)
12     {
13     }
14
15     /**
16      * This approach by using a template method pattern ensures you that
17      * each subclass will not forget to call the successor
18      */
19     final public function handle(RequestInterface $request): ?string
20     {
21         $processed = $this->processing($request);
22
23         if ($processed === null && $this->successor !== null) {
24             // the request has not been processed by this handler => see the next
25             $processed = $this->successor->handle($request);
26         }
27     }
28 }
29 
```

(suite sur la page suivante)

(suite de la page précédente)

```

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

```

(suite sur la page suivante)

(suite de la page précédente)

```

10 class SlowDatabaseHandler extends Handler
11 {
12     protected function processing(RequestInterface $request): ?string
13     {
14         // this is a mockup, in production code you would ask a slow (compared to in-
15         // memory) DB for the results
16
17         return 'Hello World!';
18     }

```

Test

Tests/ChainTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;
6
7 use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
8 use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\
9     \HttpInMemoryCacheHandler;
10 use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowDatabaseHandler;
11 use PHPUnit\Framework\TestCase;
12 use Psr\Http\Message\RequestInterface;
13 use Psr\Http\Message\UriInterface;
14
15 class ChainTest extends TestCase
16 {
17     private Handler $chain;
18
19     protected function setUp(): void
20     {
21         $this->chain = new HttpInMemoryCacheHandler(
22             ['/foo/bar?index=1' => 'Hello In Memory!'],
23             new SlowDatabaseHandler()
24         );
25     }
26
27     public function testCanRequestKeyInFastStorage()
28     {
29         $uri = $this->createMock(UriInterface::class);
30         $uri->method('getPath')->willReturn('/foo/bar');
31         $uri->method('getQuery')->willReturn('index=1');
32
33         $request = $this->createMock(RequestInterface::class);
34         $request->method('getMethod')
35             ->willReturn('GET');
36         $request->method('getUri')->willReturn($uri);

```

(suite sur la page suivante)

(suite de la page précédente)

```
36     $this->assertSame('Hello In Memory!', $this->chain->handle($request));
37 }
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 Command

Rôle

Encapsuler l'invocation et le découplage.

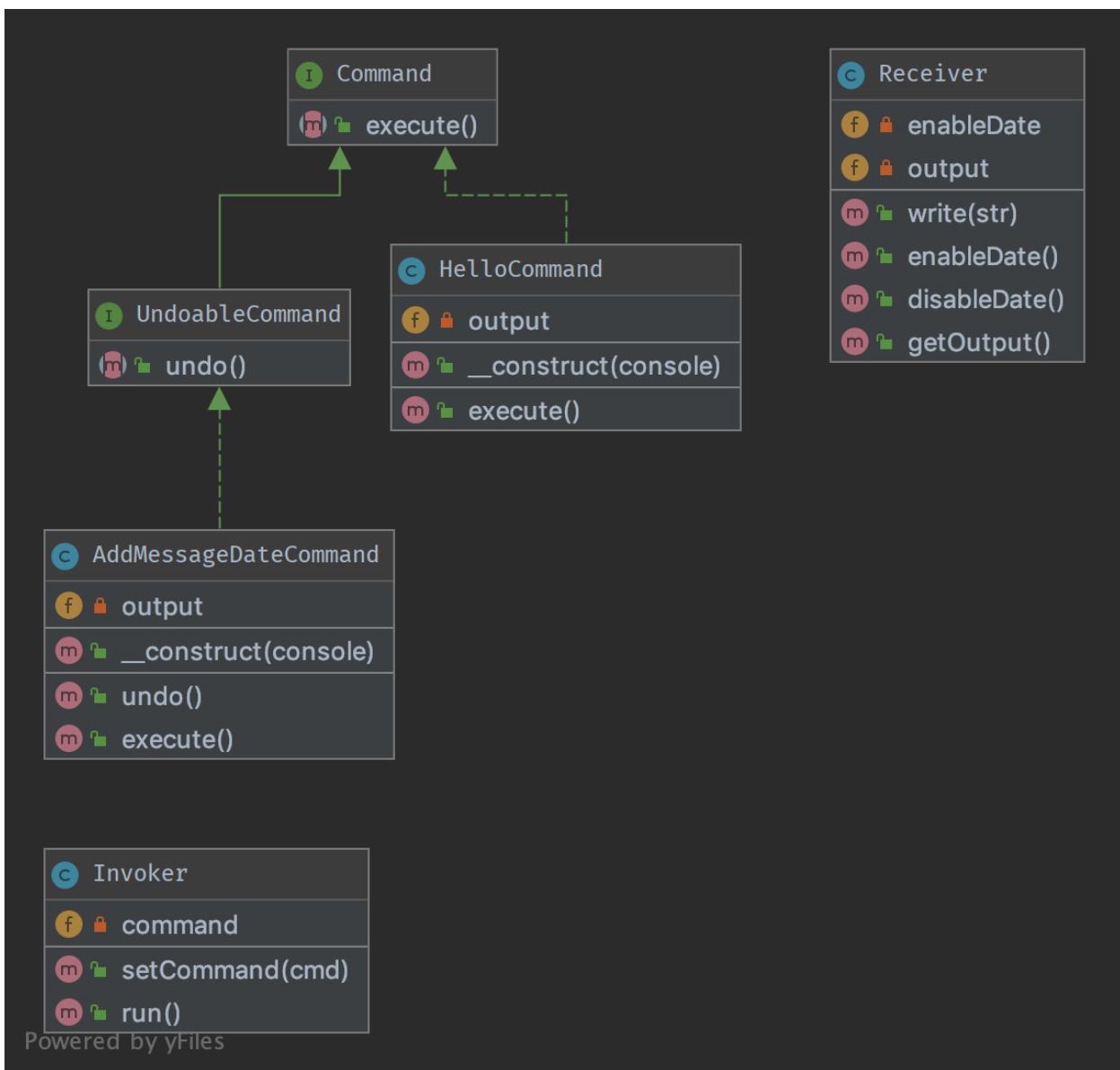
Nous avons un Invoker (exécuteur) et un Receiver (récepteur). Ce modèle utilise une « commande » pour déléguer l'exécution d'une méthode à un récepteur et fournir une méthode « execute » unifiée. De cette façon, l'exécuteur ne connaît que la méthode « execute » pour exécuter la commande. Le récepteur est donc découpé de l'appelant.

Le deuxième aspect de ce modèle est la méthode `undo()`, qui permet d'annuler l'exécution de la méthode `execute()`. Les commandes peuvent également être agrégées pour créer des commandes complexes avec une surcharge minimale et par composition (plutôt que par héritage).

Exemples

- A text editor : all events are commands which can be undone, stacked and saved.
- Les outils complexes de la console sont composés de sous-commandes afin de mieux modulariser les tâches. Chacune de ces commandes peut être implémentée comme un modèle de commande (par exemple vagrant).

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Command.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 interface Command

```

(suite sur la page suivante)

(suite de la page précédente)

```

8  {
9      /**
10      * this is the most important method in the Command pattern,
11      * The Receiver goes in the constructor.
12      */
13     public function execute();
14 }
```

UndoableCommand.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 interface UndoableCommand extends Command
8 {
9     /**
10     * This method is used to undo change made by command execution
11     */
12    public function undo();
13 }
```

HelloCommand.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 /**
8 * This concrete command calls "print" on the Receiver, but an external
9 * invoker just knows that it can call "execute"
10 */
11 class HelloCommand implements Command
12 {
13     /**
14     * Each concrete command is built with different receivers.
15     * There can be one, many or completely no receivers, but there can be other
16     * commands in the parameters
17     */
18     public function __construct(private Receiver $output)
19     {
20     }
21
22     /**
23     * execute and output "Hello World".
24     */
25     public function execute()
26     {
```

(suite sur la page suivante)

(suite de la page précédente)

```

26     // sometimes, there is no receiver and this is the command which does all the
27     ↵work
28     $this->output->write('Hello World');
29 }
```

AddMessageDateCommand.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 /**
8 * This concrete command tweaks receiver to add current date to messages
9 * invoker just knows that it can call "execute"
10 */
11 class AddMessageDateCommand implements UndoableCommand
12 {
13     /**
14      * Each concrete command is built with different receivers.
15      * There can be one, many or completely no receivers, but there can be other
16      ↵commands in the parameters.
17      */
18     public function __construct(private Receiver $output)
19     {
20     }
21
22     /**
23      * Execute and make receiver to enable displaying messages date.
24      */
25     public function execute()
26     {
27         // sometimes, there is no receiver and this is the command which
28         // does all the work
29         $this->output->enableDate();
30     }
31
32     /**
33      * Undo the command and make receiver to disable displaying messages date.
34      */
35     public function undo()
36     {
37         // sometimes, there is no receiver and this is the command which
38         // does all the work
39         $this->output->disableDate();
40     }
}
```

Receiver.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 /**
8 * Receiver is a specific service with its own contract and can be only concrete.
9 */
10 class Receiver
11 {
12     private bool $enableDate = false;
13
14     /**
15      * @var string[]
16      */
17     private array $output = [];
18
19     public function write(string $str)
20     {
21         if ($this->enableDate) {
22             $str .= ' [' . date('Y-m-d') . ']';
23         }
24
25         $this->output[] = $str;
26     }
27
28     public function getOutput(): string
29     {
30         return join("\n", $this->output);
31     }
32
33 /**
34 * Enable receiver to display message date
35 */
36 public function enableDate()
37 {
38     $this->enableDate = true;
39 }
40
41 /**
42 * Disable receiver to display message date
43 */
44 public function disableDate()
45 {
46     $this->enableDate = false;
47 }

```

Invoker.php

```

1 <?php
2

```

(suite sur la page suivante)

(suite de la page précédente)

```

3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command;
6
7 /**
8 * Invoker is using the command given to it.
9 * Example : an Application in SF2.
10 */
11 class Invoker
12 {
13     private Command $command;
14
15     /**
16      * in the invoker we find this kind of method for subscribing the command
17      * There can be also a stack, a list, a fixed set ...
18     */
19     public function setCommand(Command $cmd)
20     {
21         $this->command = $cmd;
22     }
23
24     /**
25      * executes the command; the invoker is the same whatever is the command
26     */
27     public function run()
28     {
29         $this->command->execute();
30     }
31 }
```

Test

Tests/CommandTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Command\Tests;
6
7 use DesignPatterns\Behavioral\Command\HelloCommand;
8 use DesignPatterns\Behavioral\Command\Invoker;
9 use DesignPatterns\Behavioral\Command\Receiver;
10 use PHPUnit\Framework\TestCase;
11
12 class CommandTest extends TestCase
13 {
14     public function testInvocation()
15     {
16         $invoker = new Invoker();
17         $receiver = new Receiver();
```

(suite sur la page suivante)

(suite de la page précédente)

```

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-
29             ->getOutput());
30
31         $messageDateCommand->undo();
32
33         $invoker->run();
34         $this->assertSame("Hello World\nHello World [" . date('Y-m-d') . "]\nHello World
35             -", $receiver->getOutput());
36     }
37 }
```

1.3.3 Interpreter

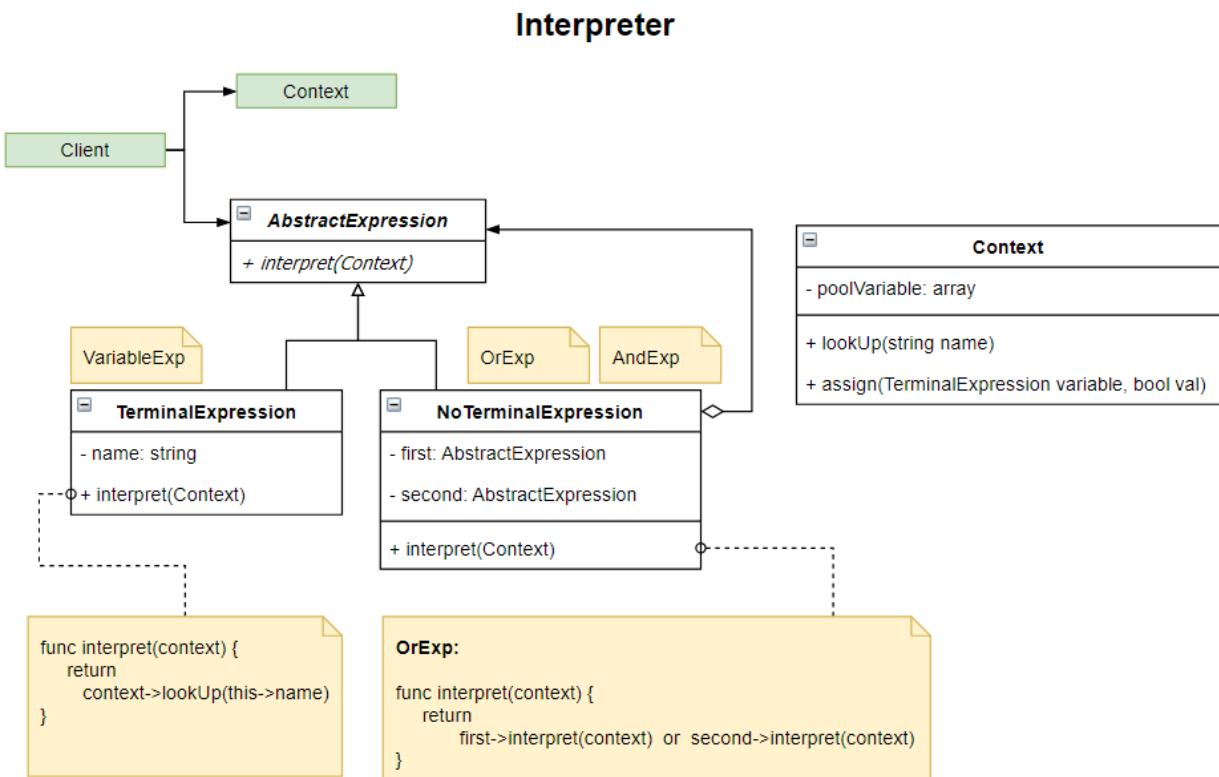
Purpose

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

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter;
6
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

7 abstract class AbstractExp
8 {
9     abstract public function interpret(Context $context): bool;
10 }
```

Context.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter;
6
7 use Exception;
8
9 class Context
10 {
11     private array $poolVariable;
12
13     public function lookUp(string $name): bool
14     {
15         if (!key_exists($name, $this->poolVariable)) {
16             throw new Exception("no exist variable: $name");
17         }
18
19         return $this->poolVariable[$name];
20     }
21
22     public function assign(VariableExp $variable, bool $val)
23     {
24         $this->poolVariable[$variable->getName()] = $val;
25     }
26 }
```

VariableExp.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter;
6
7 /**
8 * This TerminalExpression
9 */
10 class VariableExp extends AbstractExp
11 {
12     public function __construct(private string $name)
13     {
14     }
15
16     public function interpret(Context $context): bool
```

(suite sur la page suivante)

(suite de la page précédente)

```

17     {
18         return $context->lookUp($this->name);
19     }
20
21     public function getName(): string
22     {
23         return $this->name;
24     }
25 }
```

AndExp.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter;
6
7 /**
8 * This NoTerminalExpression
9 */
10 class AndExp extends AbstractExp
11 {
12     public function __construct(private AbstractExp $first, private AbstractExp $second)
13     {
14     }
15
16     public function interpret(Context $context): bool
17     {
18         return $this->first->interpret($context) && $this->second->interpret($context);
19     }
20 }
```

OrExp.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter;
6
7 /**
8 * This NoTerminalExpression
9 */
10 class OrExp extends AbstractExp
11 {
12     public function __construct(private AbstractExp $first, private AbstractExp $second)
13     {
14     }
15
16     public function interpret(Context $context): bool
17     {
```

(suite sur la page suivante)

(suite de la page précédente)

```

18     return $this->first->interpret($context) || $this->second->interpret($context);
19 }
20 }
```

Test

Tests/InterpreterTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Interpreter\Tests;
6
7 use DesignPatterns\Behavioral\Interpreter\AndExp;
8 use DesignPatterns\Behavioral\Interpreter\Context;
9 use DesignPatterns\Behavioral\Interpreter\OrExp;
10 use DesignPatterns\Behavioral\Interpreter\VariableExp;
11 use PHPUnit\Framework\TestCase;
12
13 class InterpreterTest extends TestCase
14 {
15     private Context $context;
16     private VariableExp $a;
17     private VariableExp $b;
18     private VariableExp $c;
19
20     public function setUp(): void
21     {
22         $this->context = new Context();
23         $this->a = new VariableExp('A');
24         $this->b = new VariableExp('B');
25         $this->c = new VariableExp('C');
26     }
27
28     public function testOr()
29     {
30         $this->context->assign($this->a, false);
31         $this->context->assign($this->b, false);
32         $this->context->assign($this->c, true);
33
34         // A B
35         $exp1 = new OrExp($this->a, $this->b);
36         $result1 = $exp1->interpret($this->context);
37
38         $this->assertFalse($result1, 'A B must false');
39
40         // $exp1 C
41         $exp2 = new OrExp($exp1, $this->c);
42         $result2 = $exp2->interpret($this->context);
43 }
```

(suite sur la page suivante)

(suite de la page précédente)

```

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 Iterator

Rôle

Rendre un objet itérable et le faire apparaître comme une collection d'objets.

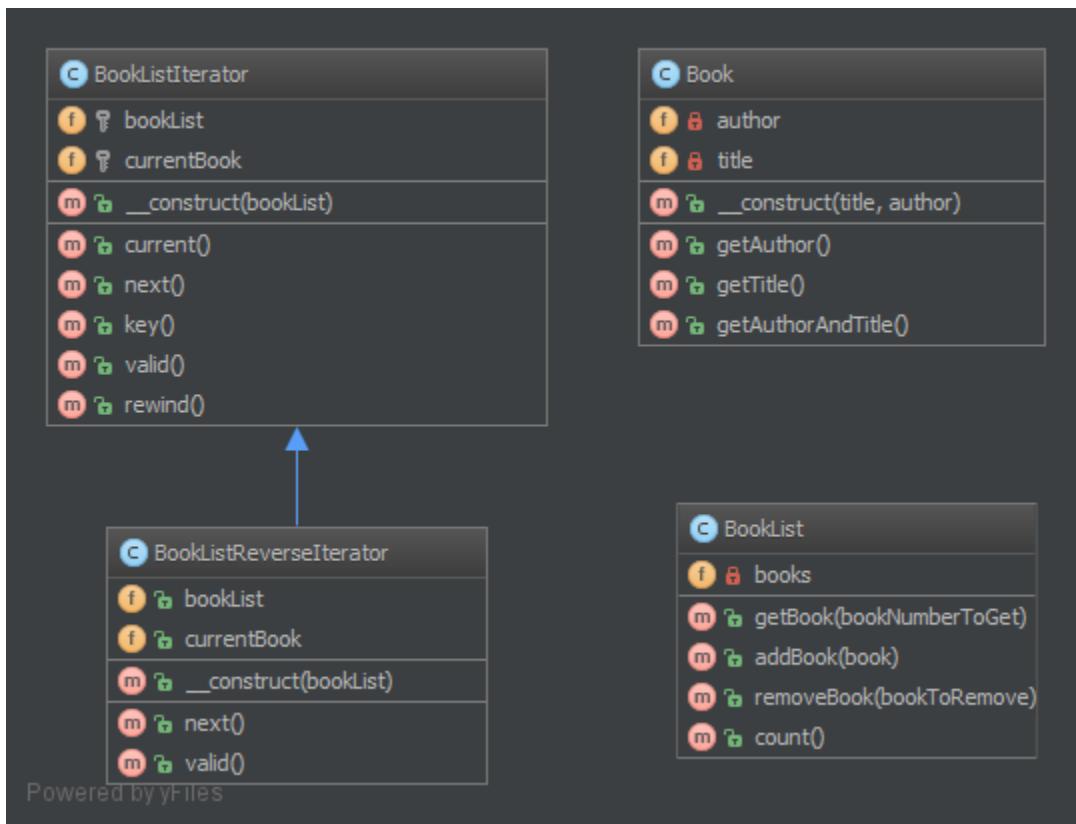
Exemples

- Si un fichier et chacune de ses lignes sont représentées sous form d'objet, Iterator permet de parcourir le fichier.

Note

La bibliothèque standard de PHP (SPL) définit une interface Iterator qui est la mieux adaptée pour cela ! Souvent vous voudrez implémenter l'interface Countable aussi, pour permettre count(\$object) sur votre objet itérable.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Book.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Iterator;
6
7 class Book
8 {
9     public function __construct(private string $title, private string $author)
10    {
11    }
12
13     public function getAuthor(): string
14    {
15         return $this->author;
16    }
17
18     public function getTitle(): string

```

(suite sur la page suivante)

(suite de la page précédente)

```

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
34     public function count(): int
35     {
36         return count($this->books);
37     }
38
39     public function current(): Book
40     {
```

(suite sur la page suivante)

(suite de la page précédente)

```

41     return $this->books[$this->currentIndex];
42 }
43
44 public function key(): int
45 {
46     return $this->currentIndex;
47 }
48
49 public function next(): void
50 {
51     $this->currentIndex++;
52 }
53
54 public function rewind(): void
55 {
56     $this->currentIndex = 0;
57 }
58
59 public function valid(): bool
60 {
61     return isset($this->books[$this->currentIndex]);
62 }
63 }
```

Test

Tests/IteratorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Iterator\Tests;
6
7 use DesignPatterns\Behavioral\Iterator\Book;
8 use DesignPatterns\Behavioral\Iterator\BookList;
9 use PHPUnit\Framework\TestCase;
10
11 class IteratorTest extends TestCase
12 {
13     public function testCanIterateOverBookList()
14     {
15         $bookList = new BookList();
16         $bookList->addBook(new Book('Learning PHP Design Patterns', 'William Sanders'));
17         $bookList->addBook(new Book('Professional Php Design Patterns', 'Aaron Saray'));
18         $bookList->addBook(new Book('Clean Code', 'Robert C. Martin'));
19
20         $books = [];
21
22         foreach ($bookList as $book) {
23             $books[] = $book->getAuthorAndTitle();
24         }
25
26         $this->assertEquals(['William Sanders', 'Aaron Saray', 'Robert C. Martin'], $books);
27     }
28 }
```

(suite sur la page suivante)

(suite de la page précédente)

```

24 }
25
26     $this->assertSame(
27         [
28             'Learning PHP Design Patterns by William Sanders',
29             'Professional Php Design Patterns by Aaron Saray',
30             'Clean Code by Robert C. Martin',
31         ],
32         $books
33     );
34 }
35
36 public function testCanIterateOverBookListAfterRemovingBook()
37 {
38     $book = new Book('Clean Code', 'Robert C. Martin');
39     $book2 = new Book('Professional Php Design Patterns', 'Aaron Saray');
40
41     $bookList = new BookList();
42     $bookList->addBook($book);
43     $bookList->addBook($book2);
44     $bookList->removeBook($book);
45
46     $books = [];
47     foreach ($bookList as $book) {
48         $books[] = $book->getAuthorAndTitle();
49     }
50
51     $this->assertSame(
52         ['Professional Php Design Patterns by Aaron Saray'],
53         $books
54     );
55 }
56
57 public function testCanAddBookToList()
58 {
59     $book = new Book('Clean Code', 'Robert C. Martin');
60
61     $bookList = new BookList();
62     $bookList->addBook($book);
63
64     $this->assertCount(1, $bookList);
65 }
66
67 public function testCanRemoveBookFromList()
68 {
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);

```

(suite sur la page suivante)

(suite de la page précédente)

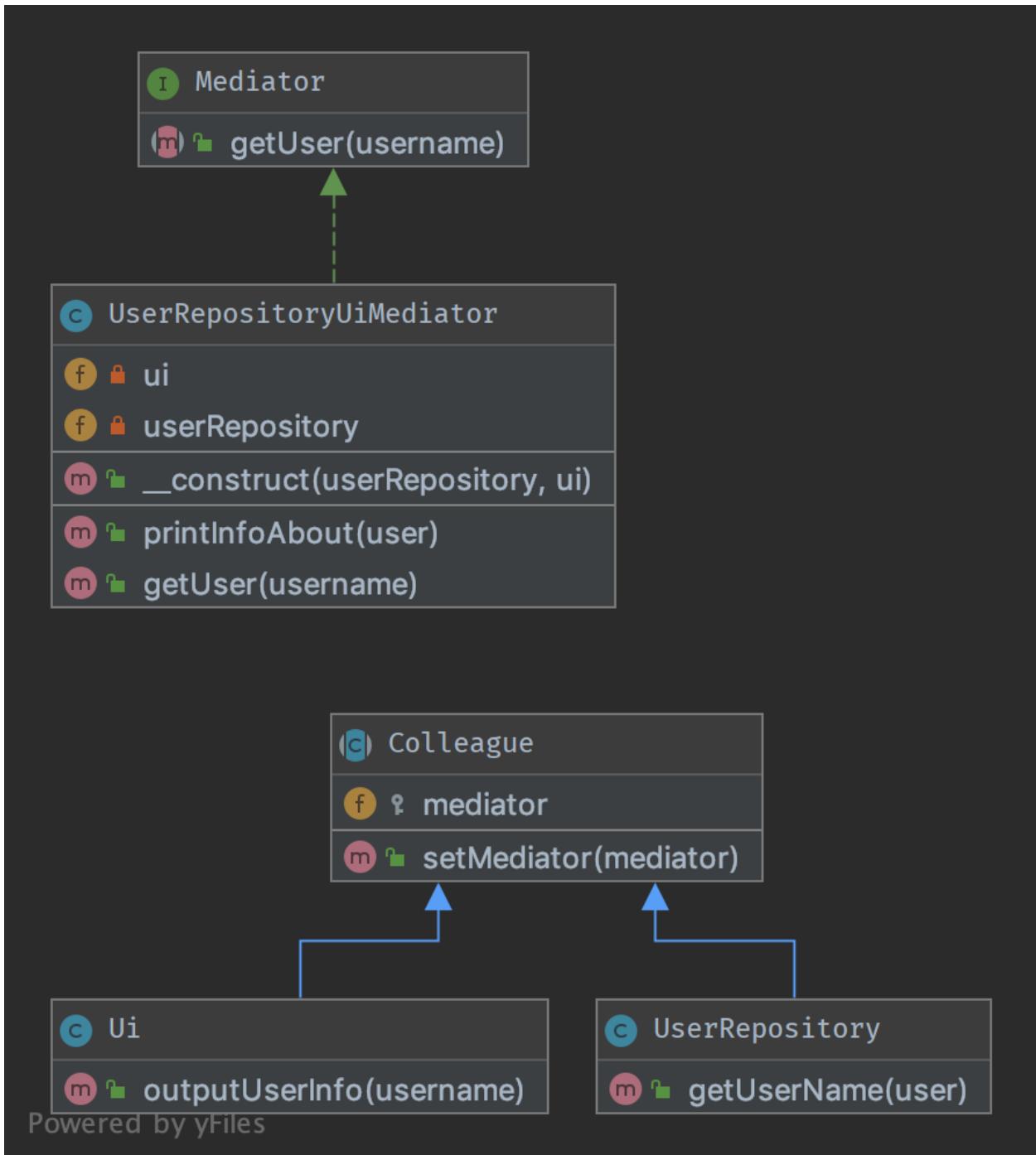
76 }
77 }

1.3.5 Mediator

Rôle

Ce pattern fournit un moyen facile de découpler de nombreux composants travaillant ensemble. C'est une bonne alternative à Observer SI vous avez une « intelligence centrale », comme un contrôleur (mais pas au sens du MVC).

Tous les composants (appelés Colleague) sont uniquement couplés à l'interface du Médiateur et c'est une bonne chose car en POO, un bon ami vaut mieux que plusieurs. C'est la caractéristique clé de ce modèle.

Diagramme UML

Code

Vous pouvez également trouver ce code sur [GitHub](#)

Mediator.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Mediator;
6
7 interface Mediator
8 {
9     public function getUser(string $username): string;
10 }
```

Colleague.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Mediator;
6
7 abstract class Colleague
8 {
9     protected Mediator $mediator;
10
11     final public function setMediator(Mediator $mediator)
12     {
13         $this->mediator = $mediator;
14     }
15 }
```

Ui.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Mediator;
6
7 class Ui extends Colleague
8 {
9     public function outputUserInfo(string $username)
10     {
11         echo $this->mediator->getUser($username);
12     }
13 }
```

UserRepository.php

```
1 <?php
```

(suite sur la page suivante)

(suite de la page précédente)

```

3 declare(strict_types=1);

4

5 namespace DesignPatterns\Behavioral\Mediator;

6

7 class UserRepository extends Colleague
8 {
9     public function getUserName(string $user): string
10    {
11        return 'User: ' . $user;
12    }
13}

```

UserRepositoryUiMediator.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Mediator;
6
7 class UserRepositoryUiMediator implements Mediator
8 {
9     public function __construct(private UserRepository $userRepository, private Ui $ui)
10    {
11        $this->userRepository->setMediator($this);
12        $this->ui->setMediator($this);
13    }
14
15    public function printInfoAbout(string $user)
16    {
17        $this->ui->outputUserInfo($user);
18    }
19
20    public function getUser(string $username): string
21    {
22        return $this->userRepository->getUserName($username);
23    }
24}

```

Test

Tests/MediatorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Tests\Mediator\Tests;
6
7 use DesignPatterns\Behavioral\Mediator\Ui;
8 use DesignPatterns\Behavioral\Mediator\UserRepository;

```

(suite sur la page suivante)

(suite de la page précédente)

```

9  use DesignPatterns\Behavioral\Mediator\UserRepositoryUiMediator;
10 use PHPUnit\Framework\TestCase;
11
12 class MediatorTest extends TestCase
13 {
14     public function testOutputHelloWorld()
15     {
16         $mediator = new UserRepositoryUiMediator(new UserRepository(), new Ui());
17
18         $this->expectOutputString('User: Dominik');
19         $mediator->printInfoAbout('Dominik');
20     }
21 }
```

1.3.6 Memento

Rôle

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

Le pattern Memento est mis en œuvre avec trois objets : Originator, Caretaker et un Memento.

Memento - Objet contenant un instantané concret et unique de l'état de tout objet ou ressource : chaîne de caractères, nombre, tableau, instance de classe, etc. L'unicité dans ce cas n'implique pas l'interdiction de l'existence d'états similaires dans différents instantanés. Cela signifie que l'état peut être extrait en tant que clone indépendant. Tout objet stocké dans le Memento doit être une copie complète de l'objet original plutôt qu'une référence à l'objet original. L'objet Memento est un « objet opaque » (l'objet que personne ne peut ou ne doit modifier).

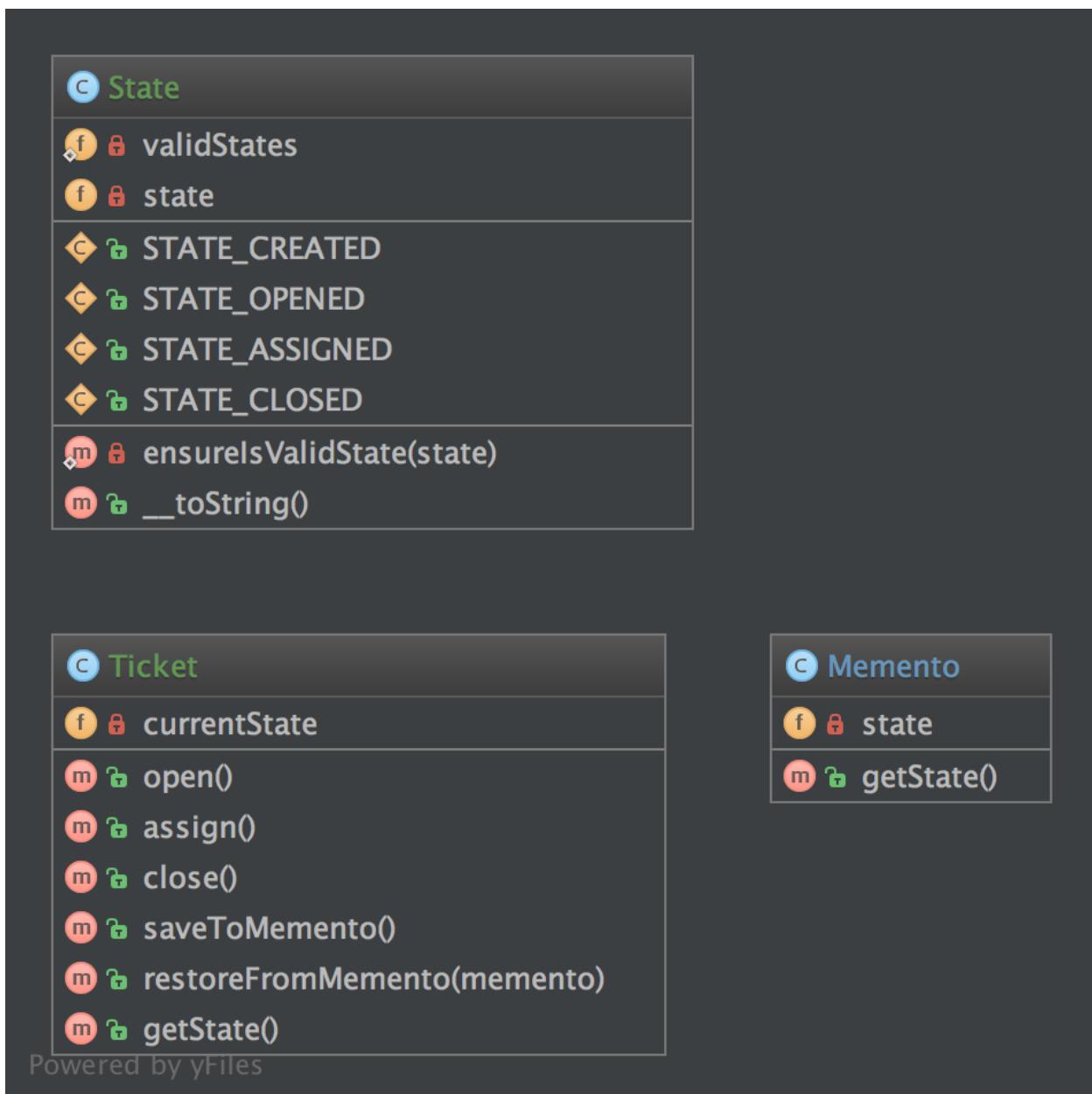
Originator - Objet contenant l'état réel d'un objet externe de type strictement spécifié. L'Originator est capable de créer une copie unique de cet état et de la renvoyer enveloppée dans un Memento. L'Originator ne connaît pas l'historique des changements. Vous pouvez définir un état concret pour l'Originator depuis l'extérieur, qui sera considéré comme réel. L'Originator doit s'assurer que l'état donné correspond au type d'objet autorisé. L'Originator peut (mais ne doit pas) avoir des méthodes, mais elles ne peuvent pas modifier l'état de l'objet sauvegardé.

Caretaker - Objet contrôlant l'historique des états. Il peut apporter des modifications à un objet, prendre la décision de sauvegarder l'état d'un objet externe dans l'Originator, demander à l'Originator un instantané de l'état actuel, ou mettre l'état de l'Originator en équivalence avec un instantané de l'histoire.

Exemples

- La source d'un générateur de nombres pseudo-aléatoires
- L'état dans une machine à états finis
- Contrôle des états intermédiaires du modèle ORM <http://en.wikipedia.org/wiki/Object-relational_mapping> avant la sauvegarde

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Memento.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Memento;

```

(suite sur la page suivante)

(suite de la page précédente)

```

6   class Memento
7   {
8       public function __construct(private State $state)
9       {
10      }
11
12
13      public function getState(): State
14      {
15          return $this->state;
16      }
17  }

```

State.php

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace DesignPatterns\Behavioral\Memento;
6
7  use InvalidArgumentException;
8
9  class State implements \Stringable
10 {
11     public const STATE_CREATED = 'created';
12     public const STATE_OPENED = 'opened';
13     public const STATE_ASSIGNED = 'assigned';
14     public const STATE_CLOSED = 'closed';
15
16     private string $state;
17
18     /**
19      * @var string[]
20     */
21     private static array $validStates = [
22         self::STATE_CREATED,
23         self::STATE_OPENED,
24         self::STATE_ASSIGNED,
25         self::STATE_CLOSED,
26     ];
27
28     public function __construct(string $state)
29     {
30         self::ensureIsValidState($state);
31
32         $this->state = $state;
33     }
34
35     private static function ensureIsValidState(string $state)
36     {
37         if (!in_array($state, self::$validStates)) {

```

(suite sur la page suivante)

(suite de la page précédente)

```

38         throw new InvalidArgumentException('Invalid state given');
39     }
40 }
41
42 public function __toString(): string
43 {
44     return $this->state;
45 }
46 }
```

Ticket.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Memento;
6
7 /**
8 * Ticket is the "Originator" in this implementation
9 */
10 class Ticket
11 {
12     private State $currentState;
13
14     public function __construct()
15     {
16         $this->currentState = new State(State::STATE_CREATED);
17     }
18
19     public function open()
20     {
21         $this->currentState = new State(State::STATE_OPENED);
22     }
23
24     public function assign()
25     {
26         $this->currentState = new State(State::STATE_ASSIGNED);
27     }
28
29     public function close()
30     {
31         $this->currentState = new State(State::STATE_CLOSED);
32     }
33
34     public function saveToMemento(): Memento
35     {
36         return new Memento(clone $this->currentState);
37     }
38
39     public function restoreFromMemento(Memento $memento)
40     {
```

(suite sur la page suivante)

(suite de la page précédente)

```

41     $this->currentState = $memento->getState();
42 }
43
44 public function getState(): State
45 {
46     return $this->currentState;
47 }
48 }
```

Test

Tests/MementoTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Memento\Tests;
6
7 use DesignPatterns\Behavioral\Memento\State;
8 use DesignPatterns\Behavioral\Memento\Ticket;
9 use PHPUnit\Framework\TestCase;
10
11 class MementoTest extends TestCase
12 {
13     public function testOpenTicketAssignAndSetBackToOpen()
14     {
15         $ticket = new Ticket();
16
17         // open the ticket
18         $ticket->open();
19         $openedState = $ticket->getState();
20         $this->assertSame(State::STATE_OPENED, (string) $ticket->getState());
21
22         $memento = $ticket->saveToMemento();
23
24         // assign the ticket
25         $ticket->assign();
26         $this->assertSame(State::STATE_ASSIGNED, (string) $ticket->getState());
27
28         // now restore to the opened state, but verify that the state object has been
29         // cloned for the memento
30         $ticket->restoreFromMemento($memento);
31
32         $this->assertSame(State::STATE_OPENED, (string) $ticket->getState());
33         $this->assertNotSame($openedState, $ticket->getState());
34     }
35 }
```

1.3.7 Null Object

Rôle

NullObject n'est pas un patron de conception du GoF mais un schéma qui apparaît suffisamment fréquemment pour être considéré comme un patron. Il présente les avantages suivants :

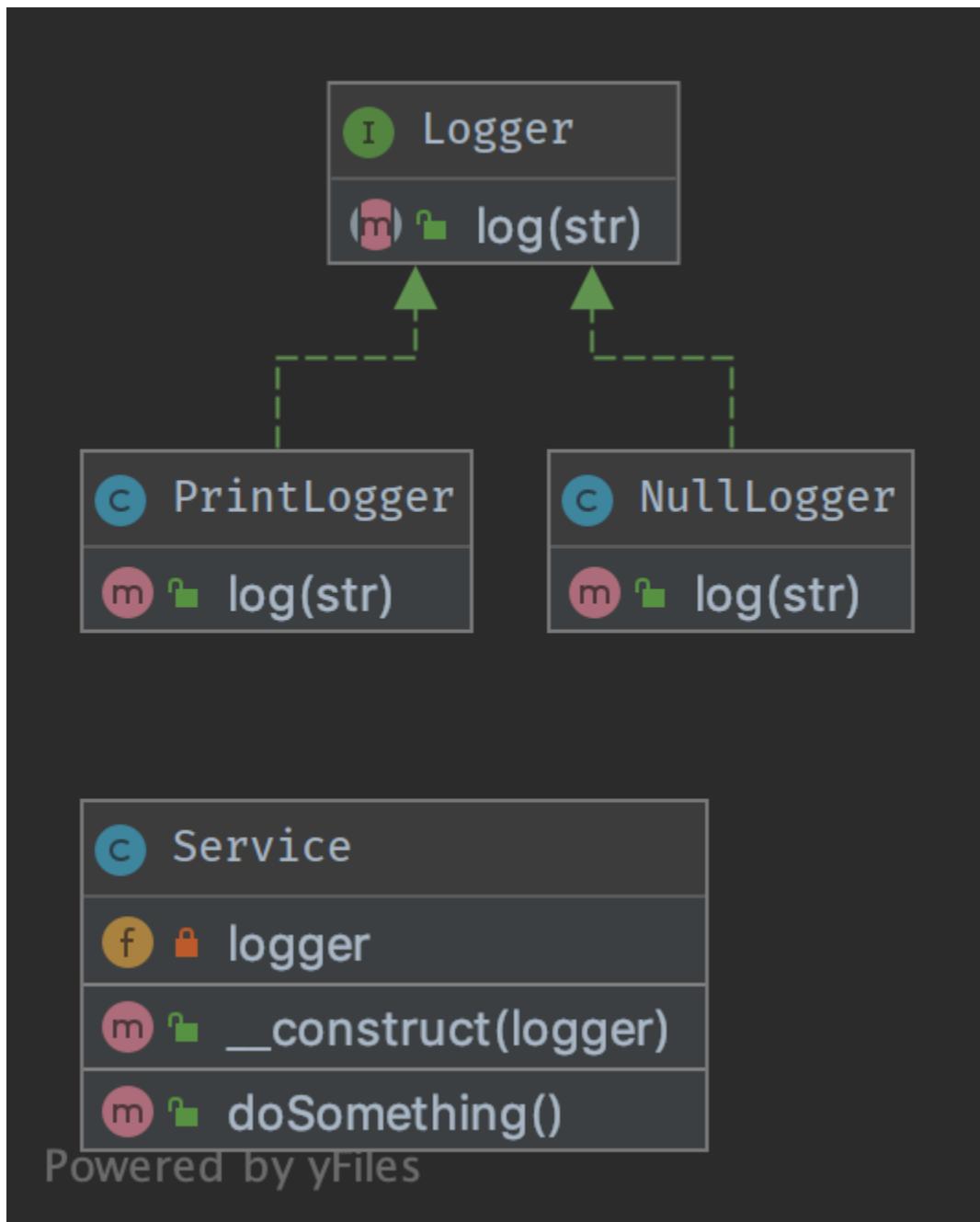
- Le code client est simplifié
- Réduit le risque d'exceptions de pointeur null
- Moins de conditions exigent moins de cas de test

Les méthodes renvoyant un objet ou null devraient plutôt renvoyer un objet ou NullObject. Les NullObjects simplifient le code passe-partout tel que `if (!is_null($obj)) { $obj->callSomething() ; }` en le remplaçant par `$obj->callSomething();` en éliminant la vérification conditionnelle dans le code client.

Exemples

- Null logger or null output to preserve a standard way of interaction between objects, even if the shouldn't do anything
- Null handler dans un modèle de chaîne de responsabilités
- Null command dans un patron Command

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Service.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\NullObject;
6
7 class Service
8 {
9     public function __construct(private Logger $logger)
10    {
11    }
12
13    /**
14     * do something ...
15     */
16    public function doSomething()
17    {
18        // notice here that you don't have to check if the logger is set with eg. is_
19        // null(), instead just use it
20        $this->logger->log('We are in ' . __METHOD__);
21    }
}

```

Logger.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\NullObject;
6
7 /**
8  * Key feature: NullLogger must inherit from this interface like any other loggers
9  */
10 interface Logger
11 {
12     public function log(string $str);
13 }

```

PrintLogger.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\NullObject;
6
7 class PrintLogger implements Logger

```

(suite sur la page suivante)

(suite de la page précédente)

```

8  {
9      public function log(string $str)
10     {
11         echo $str;
12     }
13 }
```

NullLogger.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\NullObject;
6
7 class NullLogger implements Logger
8 {
9     public function log(string $str)
10    {
11        // do nothing
12    }
13 }
```

Test

Tests/LoggerTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\NullObject\Tests;
6
7 use DesignPatterns\Behavioral\NullObject\NullLogger;
8 use DesignPatterns\Behavioral\NullObject\PrintLogger;
9 use DesignPatterns\Behavioral\NullObject\Service;
10 use PHPUnit\Framework\TestCase;
11
12 class LoggerTest extends TestCase
13 {
14     public function testNullObject()
15     {
16         $service = new Service(new NullLogger());
17         $this->expectOutputString('');
18         $service->doSomething();
19     }
20
21     public function testStandardLogger()
22     {
23         $service = new Service(new PrintLogger());
24         $this->expectOutputString('We are in DesignPatterns\Behavioral\NullObject\
```

(suite sur la page suivante)

(suite de la page précédente)

```

25     ↵Service::doSomething());
26         $service->doSomething();
27     }
}

```

1.3.8 Observer

Rôle

Implémenter un comportement de publication/abonnement à un objet, chaque fois qu'un objet « Sujet » change d'état, les « Observateurs » attachés seront notifiés. Il est utilisé pour réduire le nombre d'objets couplés et utilise plutôt un couplage souple.

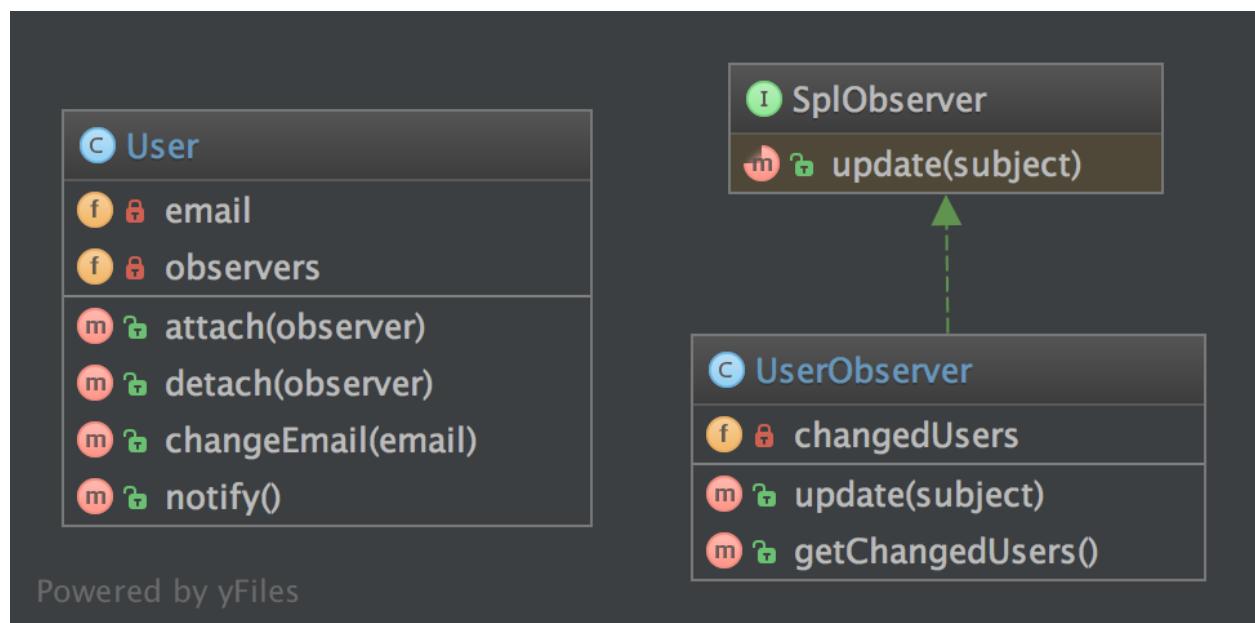
Exemples

- Un système de file d'attente de messages est observé pour montrer la progression d'un travail dans une interface graphique.

Note

PHP définit déjà deux interfaces qui peuvent aider à implémenter ce modèle : `SplObserver` et `SplSubject`.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

User.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Observer;
6
7 use SplSubject;
8 use SplObjectStorage;
9 use SplObserver;
10
11 /**
12 * 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
13 */
14
15 class User implements SplSubject
16 {
17     private SplObjectStorage $observers;
18     private $email;
19
20     public function __construct()
21     {
22         $this->observers = new SplObjectStorage();
23     }
24
25     public function attach(SplObserver $observer): void
26     {
27         $this->observers->attach($observer);
28     }
29
30     public function detach(SplObserver $observer): void
31     {
32         $this->observers->detach($observer);
33     }
34
35     public function changeEmail(string $email): void
36     {
37         $this->email = $email;
38         $this->notify();
39     }
40
41     public function notify(): void
42     {
43         /** @var SplObserver $observer */
44         foreach ($this->observers as $observer) {
45             $observer->update($this);
46         }
47     }

```

(suite sur la page suivante)

(suite de la page précédente)

48

}

UserObserver.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Observer;
6
7 use SplObserver;
8 use SplSubject;
9
10 class UserObserver implements SplObserver
11 {
12     /**
13      * @var SplSubject[]
14      */
15     private array $changedUsers = [];
16
17     /**
18      * It is called by the Subject, usually by SplSubject::notify()
19      */
20     public function update(SplSubject $subject): void
21     {
22         $this->changedUsers[] = clone $subject;
23     }
24
25     /**
26      * @return SplSubject[]
27      */
28     public function getChangedUsers(): array
29     {
30         return $this->changedUsers;
31     }
32 }
```

Test

Tests/ObserverTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Observer\Tests;
6
7 use DesignPatterns\Behavioral\Observer\User;
8 use DesignPatterns\Behavioral\Observer\UserObserver;
9 use PHPUnit\Framework\TestCase;
```

(suite sur la page suivante)

(suite de la page précédente)

```

11 class ObserverTest extends TestCase
12 {
13     public function testChangeInUserLeadsToUserObserverBeingNotified()
14     {
15         $observer = new UserObserver();
16
17         $user = new User();
18         $user->attach($observer);
19
20         $user->changeEmail('foo@bar.com');
21         $this->assertCount(1, $observer->getChangedUsers());
22     }
23 }

```

1.3.9 Specification

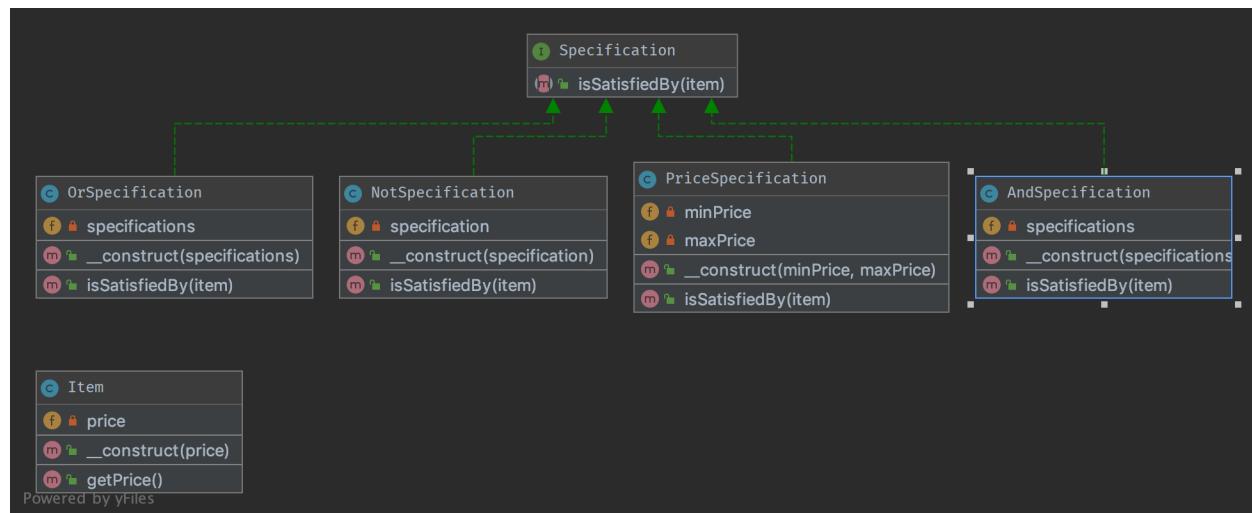
Rôle

Construire une spécification claire des règles de gestion, à laquelle les objets peuvent être comparés. La classe de spécification composite possède une méthode appelée `isSatisfiedBy` qui renvoie soit vrai soit faux selon que l'objet soit conforme ou non aux règles de gestion.

Exemples

— RulerZ

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Item.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 class Item
8 {
9     public function __construct(private float $price)
10    {
11    }
12
13     public function getPrice(): float
14    {
15         return $this->price;
16    }
17 }
```

Specification.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 interface Specification
8 {
9     public function isSatisfiedBy(Item $item): bool;
10 }
```

OrSpecification.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 class OrSpecification implements Specification
8 {
9     /**
10      * @var Specification[]
11      */
12     private array $specifications;
13
14     /**
15      * @param Specification[] $specifications
16     */
```

(suite sur la page suivante)

(suite de la page précédente)

```

16   */
17   public function __construct(Specification ...$specifications)
18   {
19       $this->specifications = $specifications;
20   }
21
22 /**
23 * if at least one specification is true, return true, else return false
24 */
25 public function isSatisfiedBy(Item $item): bool
26 {
27     foreach ($this->specifications as $specification) {
28         if ($specification->isSatisfiedBy($item)) {
29             return true;
30         }
31     }
32
33     return false;
34 }
35 }
```

PriceSpecification.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 class PriceSpecification implements Specification
8 {
9     public function __construct(private ?float $minPrice, private ?float $maxPrice)
10    {
11    }
12
13    public function isSatisfiedBy(Item $item): bool
14    {
15        if ($this->maxPrice !== null && $item->getPrice() > $this->maxPrice) {
16            return false;
17        }
18
19        if ($this->minPrice !== null && $item->getPrice() < $this->minPrice) {
20            return false;
21        }
22
23        return true;
24    }
25 }
```

AndSpecification.php

```
1 <?php
```

(suite sur la page suivante)

(suite de la page précédente)

```

2 declare(strict_types=1);
3
4 namespace DesignPatterns\Behavioral\Specification;
5
6 class AndSpecification implements Specification
7 {
8
9     /**
10      * @var Specification[]
11      */
12     private array $specifications;
13
14     /**
15      * @param Specification[] $specifications
16      */
17     public function __construct(Specification ...$specifications)
18     {
19         $this->specifications = $specifications;
20     }
21
22     /**
23      * if at least one specification is false, return false, else return true.
24      */
25     public function isSatisfiedBy(Item $item): bool
26     {
27         foreach ($this->specifications as $specification) {
28             if (!$specification->isSatisfiedBy($item)) {
29                 return false;
30             }
31         }
32
33         return true;
34     }
35 }
```

NotSpecification.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 class NotSpecification implements Specification
8 {
9     public function __construct(private Specification $specification)
10    {
11    }
12
13     public function isSatisfiedBy(Item $item): bool
14    {
15        return !$this->specification->isSatisfiedBy($item);
16    }
17 }
```

(suite sur la page suivante)

(suite de la page précédente)

```
16     }
17 }
```

Test

Tests/SpecificationTest.php

```
<?php

declare(strict_types=1);

namespace DesignPatterns\Behavioral\Specification\Tests;

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

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

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

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

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

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

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

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

(suite sur la page suivante)

(suite de la page précédente)

```

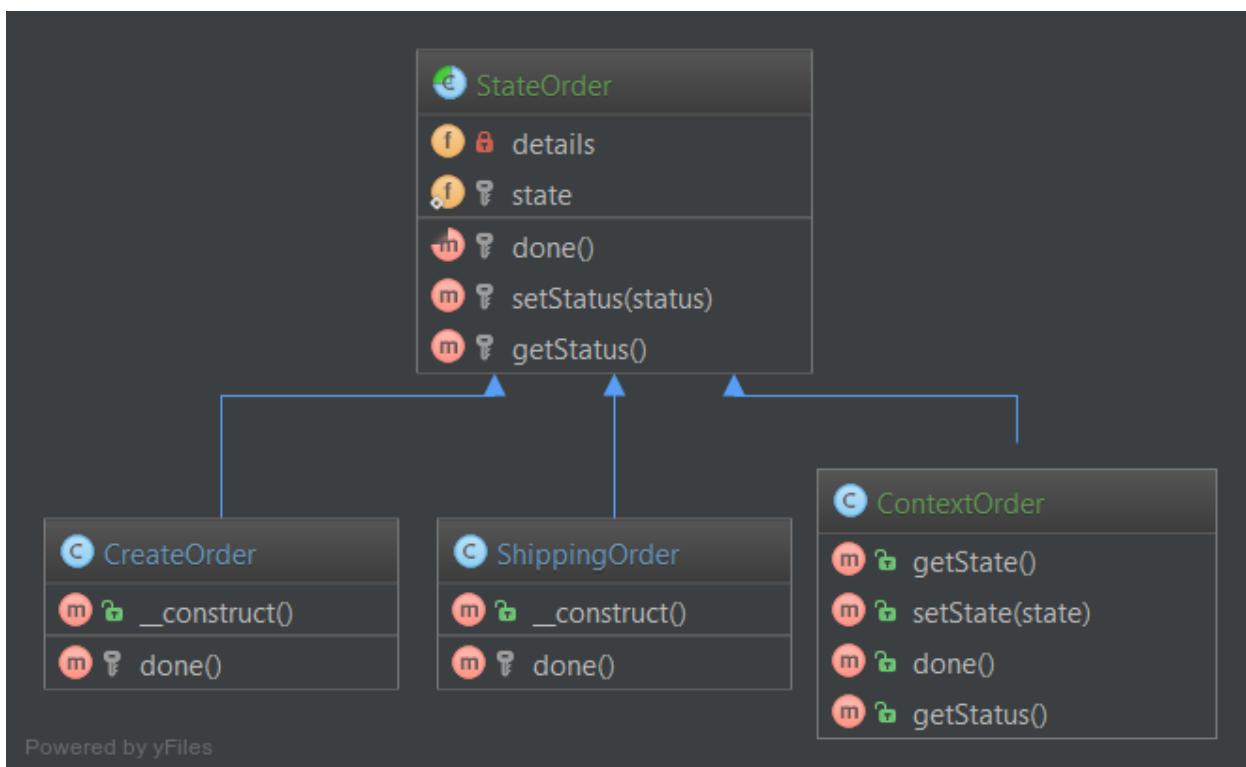
45
46     $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));
47     $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));
48 }
49 }
```

1.3.10 State

Rôle

Encapsule des comportements différents pour la même routine en fonction de l'état de l'objet. Cela peut être un moyen plus propre pour un objet de changer son comportement au moment de l'exécution sans avoir recours à de grandes déclarations conditionnelles monolithiques.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

OrderContext.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State;
6
7 class OrderContext
8 {
9     private State $state;
10
11    public static function create(): OrderContext
12    {
13        $order = new self();
14        $order->state = new StateCreated();
15
16        return $order;
17    }
18
19    public function setState(State $state)
20    {
21        $this->state = $state;
22    }
23
24    public function proceedToNext()
25    {
26        $this->state->proceedToNext($this);
27    }
28
29    public function toString()
30    {
31        return $this->state->toString();
32    }
33 }
```

State.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State;
6
7 interface State
8 {
9     public function proceedToNext(OrderContext $context);
10
11     public function toString(): string;
12 }
```

StateCreated.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State;
6
7 class StateCreated implements State
8 {
9     public function proceedToNext(OrderContext $context)
10    {
11        $context->setState(new StateShipped());
12    }
13
14    public function toString(): string
15    {
16        return 'created';
17    }
18 }
```

StateShipped.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State;
6
7 class StateShipped implements State
8 {
9     public function proceedToNext(OrderContext $context)
10    {
11        $context->setState(new StateDone());
12    }
13
14    public function toString(): string
15    {
16        return 'shipped';
17    }
18 }
```

StateDone.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State;
6
7 class StateDone implements State
8 {
9     public function proceedToNext(OrderContext $context)
10    {
```

(suite sur la page suivante)

(suite de la page précédente)

```

11     // there is nothing more to do
12 }
13
14 public function toString(): string
15 {
16     return 'done';
17 }
18 }
```

Test

Tests/StateTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\State\Tests;
6
7 use DesignPatterns\Behavioral\State\OrderContext;
8 use PHPUnit\Framework\TestCase;
9
10 class StateTest extends TestCase
11 {
12     public function testIsCreatedWithStateCreated()
13     {
14         $orderContext = OrderContext::create();
15
16         $this->assertSame('created', $orderContext->toString());
17     }
18
19     public function testCanProceedToStateShipped()
20     {
21         $contextOrder = OrderContext::create();
22         $contextOrder->proceedToNext();
23
24         $this->assertSame('shipped', $contextOrder->toString());
25     }
26
27     public function testCanProceedToStateDone()
28     {
29         $contextOrder = OrderContext::create();
30         $contextOrder->proceedToNext();
31         $contextOrder->proceedToNext();
32
33         $this->assertSame('done', $contextOrder->toString());
34     }
35
36     public function testStateDoneIsTheLastPossibleState()
37     {
38         $contextOrder = OrderContext::create();
```

(suite sur la page suivante)

(suite de la page précédente)

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

1.3.11 Strategy

Terminologie

- Context
- Strategy
- Concrete Strategy

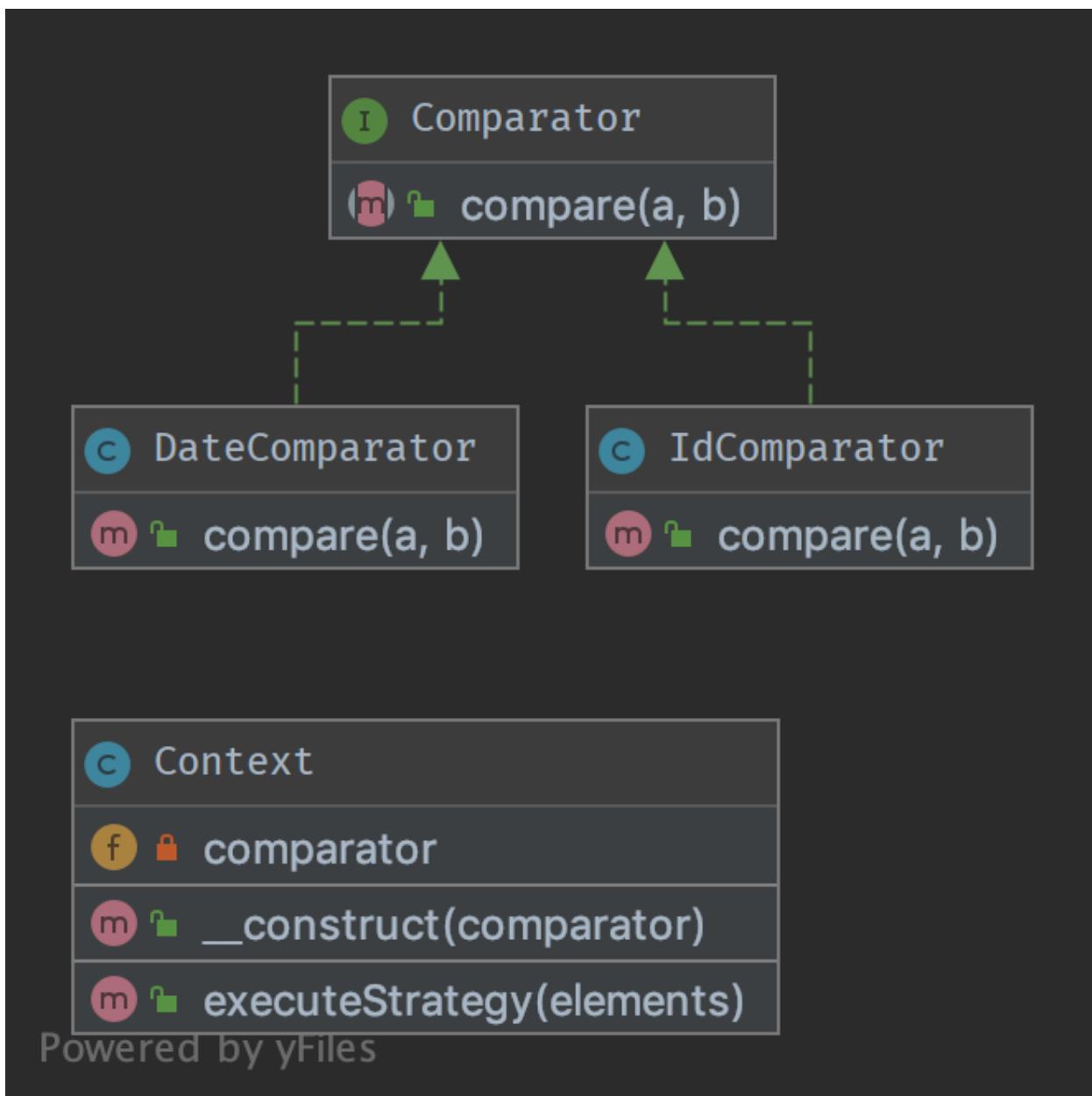
Rôle

Pour séparer les stratégies et permettre un passage rapide de l'une à l'autre. Ce modèle est également une bonne alternative à l'héritage (au lieu d'avoir une classe abstraite qui est dérivée).

Exemples

- Trier une liste d'objets, une stratégie par date, l'autre par identifiant
- Simplifier les tests unitaires : par exemple, passer du stockage en fichier au stockage en RAM

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Context.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Strategy;
  
```

(suite sur la page suivante)

(suite de la page précédente)

```

6 class Context
7 {
8     public function __construct(private Comparator $comparator)
9     {
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

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Strategy;
6
7 class IdComparator implements Comparator
8 {
9     public function compare($a, $b): int
10    {
11        return $a['id'] <=> $b['id'];
12    }
13 }
```

Test

Tests/StrategyTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Strategy\Tests;
6
7 use DesignPatterns\Behavioral\Strategy\Context;
8 use DesignPatterns\Behavioral\Strategy\DateComparator;
9 use DesignPatterns\Behavioral\Strategy\IdComparator;
10 use PHPUnit\Framework\TestCase;
11
12 class StrategyTest extends TestCase
13 {
14     public function provideIntegers()
15     {
16         return [
17             [
18                 [['id' => 2], ['id' => 1], ['id' => 3]],
19                 ['id' => 1],
20             ],
21             [
22                 [['id' => 3], ['id' => 2], ['id' => 1]],
23                 ['id' => 1],
24             ],
25         ];
26     }
27
28     public function provideDates()
29     {
30         return [
31             [
32                 ['date' => '2014-03-03'], ['date' => '2015-03-02'], ['date' => '2013-03-
```

(suite sur la page suivante)

(suite de la page précédente)

```

33     ↵01']] ,
34         ['date' => '2013-03-01'],
35     ],
36     [
37         [['date' => '2014-02-03'], ['date' => '2013-02-01'], ['date' => '2015-02-
38         ↵02']] ,
39         ['date' => '2013-02-01'],
40     ],
41 }
42 /**
43 * @dataProvider provideIntegers
44 *
45 * @param array $collection
46 * @param array $expected
47 */
48 public function testIdComparator($collection, $expected)
49 {
50     $obj = new Context(new IdComparator());
51     $elements = $obj->executeStrategy($collection);
52
53     $firstElement = array_shift($elements);
54     $this->assertSame($expected, $firstElement);
55 }
56 /**
57 * @dataProvider provideDates
58 *
59 * @param array $collection
60 * @param array $expected
61 */
62 public function testDateComparator($collection, $expected)
63 {
64     $obj = new Context(new DateComparator());
65     $elements = $obj->executeStrategy($collection);
66
67     $firstElement = array_shift($elements);
68     $this->assertSame($expected, $firstElement);
69 }
70 }
71

```

1.3.12 Template Method

Rôle

Template Method est un patron de conception comportemental.

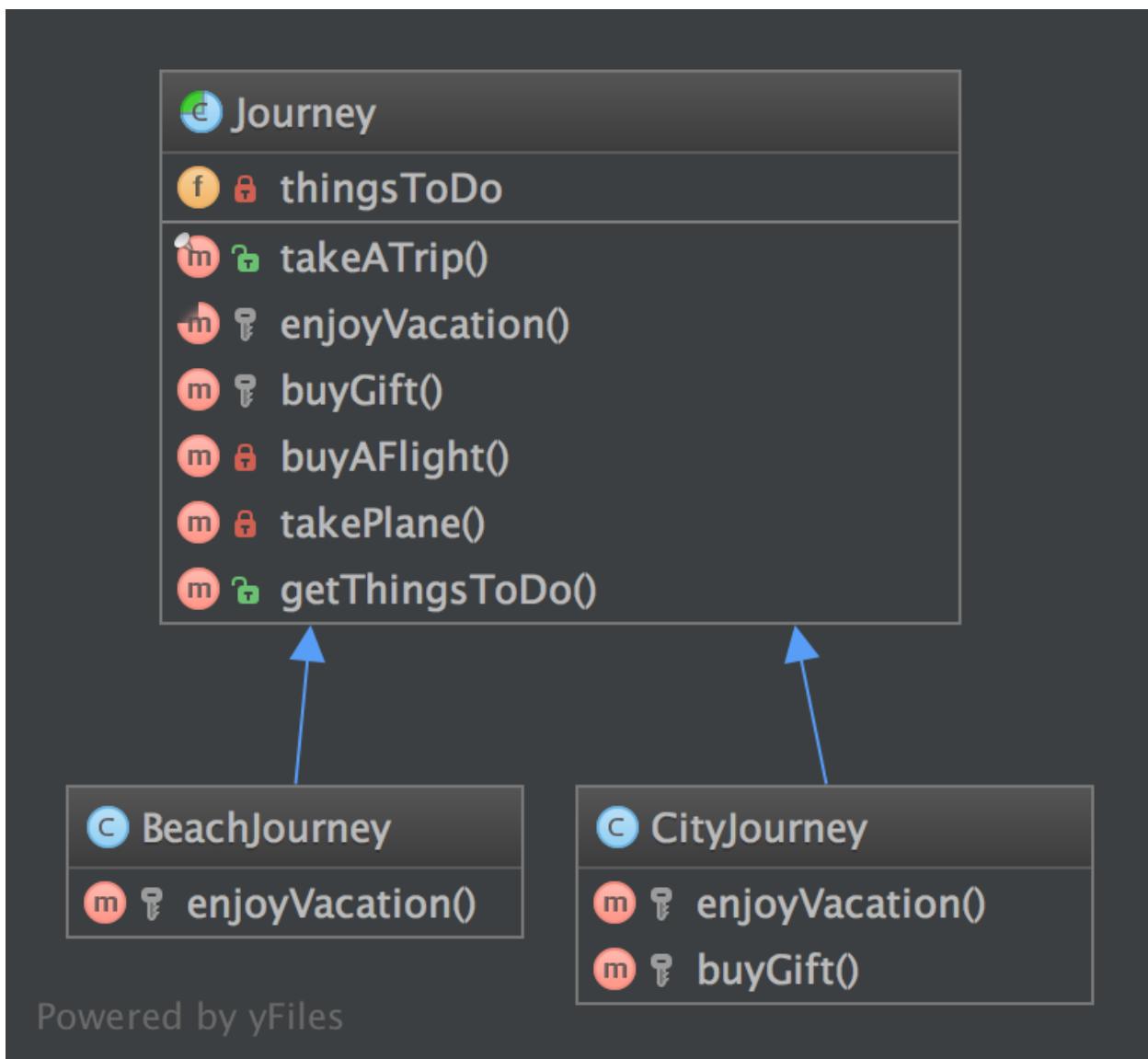
Vous l'avez peut-être déjà rencontré à plusieurs reprises. L'idée est de laisser les classes dérivées de ce template abstrait « finaliser » le comportement d'un algorithme.

Par exemple, le « Hollywood principle » : « Ne nous appelez pas, nous vous appelons. » Cette classe n'est pas appelée par la classe dérivée, mais au contraire. Cela se fait par le biais d'une abstraction.

En d'autres termes, il s'agit d'un squelette d'algorithme, bien adapté aux librairies de frameworks. L'utilisateur n'a qu'à implémenter une méthode et la classe parente fait le travail.

C'est un moyen facile de découpler les classes concrètes et de réduire le copier-coller, c'est pourquoi vous le trouverez partout.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Journey.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\TemplateMethod;
6
7 abstract class Journey
8 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
9  /**
10   * @var string[]
11   */
12  private array $thingsToDo = [];
13
14 /**
15  * This is the public service provided by this class and its subclasses.
16  * Notice it is final to "freeze" the global behavior of algorithm.
17  * If you want to override this contract, make an interface with only takeATrip()
18  * and subclass it.
19  */
20 final public function takeATrip()
21 {
22     $this->thingsToDo[] = $this->buyAFlight();
23     $this->thingsToDo[] = $this->takePlane();
24     $this->thingsToDo[] = $this->enjoyVacation();
25     $buyGift = $this->buyGift();
26
27     if ($buyGift !== null) {
28         $this->thingsToDo[] = $buyGift;
29     }
30
31     $this->thingsToDo[] = $this->takePlane();
32 }
33
34 /**
35  * This method must be implemented, this is the key-feature of this pattern.
36  */
37 abstract protected function enjoyVacation(): string;
38
39 /**
40  * This method is also part of the algorithm but it is optional.
41  * You can override it only if you need to
42  */
43 protected function buyGift(): ?string
44 {
45     return null;
46 }
47
48 private function buyAFlight(): string
49 {
50     return 'Buy a flight ticket';
51 }
52
53 private function takePlane(): string
54 {
55     return 'Taking the plane';
56 }
57
58 /**
59  * @return string[]
60  */
```

(suite sur la page suivante)

(suite de la page précédente)

```
61     final public function getThingsToDo(): array
62     {
63         return $this->thingsToDo;
64     }
65 }
```

BeachJourney.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\TemplateMethod;
6
7 class BeachJourney extends Journey
8 {
9     protected function enjoyVacation(): string
10    {
11        return "Swimming and sun-bathing";
12    }
13 }
```

CityJourney.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\TemplateMethod;
6
7 class CityJourney extends Journey
8 {
9     protected function enjoyVacation(): string
10    {
11        return "Eat, drink, take photos and sleep";
12    }
13
14     protected function buyGift(): ?string
15    {
16        return "Buy a gift";
17    }
18 }
```

Test

Tests/JourneyTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\TemplateMethod\Tests;
6
7 use DesignPatterns\Behavioral\TemplateMethod\BeachJourney;
8 use DesignPatterns\Behavioral\TemplateMethod\CityJourney;
9 use PHPUnit\Framework\TestCase;
10
11 class JourneyTest extends TestCase
12 {
13     public function testCanGetOnVacationOnTheBeach()
14     {
15         $beachJourney = new BeachJourney();
16         $beachJourney->takeATrip();
17
18         $this->assertSame(
19             ['Buy a flight ticket', 'Taking the plane', 'Swimming and sun-bathing',
20             ↪'Taking the plane'],
21             $beachJourney->getThingsToDo()
22         );
23     }
24
25     public function testCanGetOnAJourneyToACity()
26     {
27         $cityJourney = new CityJourney();
28         $cityJourney->takeATrip();
29
30         $this->assertSame(
31             [
32                 'Buy a flight ticket',
33                 'Taking the plane',
34                 'Eat, drink, take photos and sleep',
35                 'Buy a gift',
36                 'Taking the plane'
37             ],
38             $cityJourney->getThingsToDo()
39         );
40     }
}
```

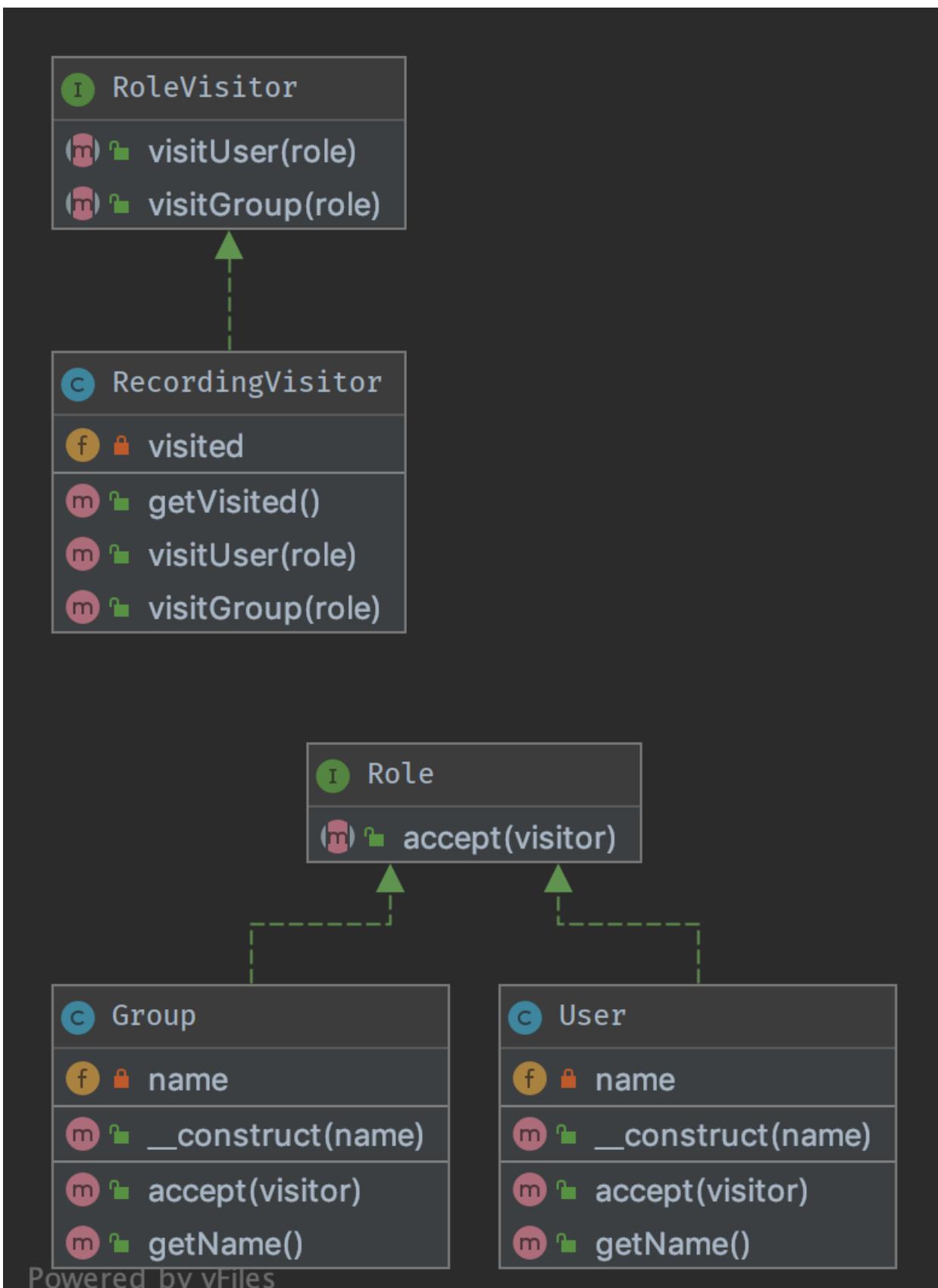
1.3.13 Visitor

Rôle

Le pattern Visitor vous permet de sous-traiter des opérations sur des objets à d'autres objets. La principale raison de faire cela est de maintenir une séparation des préoccupations. Mais les classes doivent définir un contrat pour autoriser les visiteurs (la méthode `Role::accept` dans l'exemple).

Le contrat est une classe abstraite mais vous pouvez aussi avoir une interface propre. Dans ce cas, chaque visiteur doit choisir lui-même la méthode à invoquer sur le visiteur.

Diagramme UML



Powered by yFiles

Code

Vous pouvez également trouver ce code sur [GitHub](#)

RoleVisitor.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Visitor;
6
7 /**
8 * Note: the visitor must not choose itself which method to
9 * invoke, it is the visited object that makes this decision
10 */
11 interface RoleVisitor
12 {
13     public function visitUser(User $role);
14
15     public function visitGroup(Group $role);
16 }
```

RecordingVisitor.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Visitor;
6
7 class RecordingVisitor implements RoleVisitor
8 {
9     /**
10      * @var Role[]
11     */
12     private array $visited = [];
13
14     public function visitGroup(Group $role)
15     {
16         $this->visited[] = $role;
17     }
18
19     public function visitUser(User $role)
20     {
21         $this->visited[] = $role;
22     }
23
24     /**
25      * @return Role[]
26     */
27     public function getVisited(): array
28     {
29         return $this->visited;
```

(suite sur la page suivante)

(suite de la page précédente)

```
30     }
31 }
```

Role.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Visitor;
6
7 interface Role
8 {
9     public function accept(RoleVisitor $visitor);
10}
```

User.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Visitor;
6
7 class User implements Role
8 {
9     public function __construct(private string $name)
10    {
11    }
12
13     public function getName(): string
14    {
15         return sprintf('User %s', $this->name);
16    }
17
18     public function accept(RoleVisitor $visitor)
19    {
20         $visitor->visitUser($this);
21    }
22 }
```

Group.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Visitor;
6
7 class Group implements Role
8 {
9     public function __construct(private string $name)
10    {
```

(suite sur la page suivante)

(suite de la page précédente)

```

11 }
12
13     public function getName(): string
14     {
15         return sprintf('Group: %s', $this->name);
16     }
17
18     public function accept(RoleVisitor $visitor)
19     {
20         $visitor->visitGroup($this);
21     }
22 }
```

Test

Tests/VisitorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Tests\Visitor\Tests;
6
7 use DesignPatterns\Behavioral\Visitor\RecordingVisitor;
8 use DesignPatterns\Behavioral\Visitor\User;
9 use DesignPatterns\Behavioral\Visitor\Group;
10 use DesignPatterns\Behavioral\Visitor\Role;
11 use DesignPatterns\Behavioral\Visitor;
12 use PHPUnit\Framework\TestCase;
13
14 class VisitorTest extends TestCase
15 {
16     private RecordingVisitor $visitor;
17
18     protected function setUp(): void
19     {
20         $this->visitor = new RecordingVisitor();
21     }
22
23     public function provideRoles()
24     {
25         return [
26             [new User('Dominik')],
27             [new Group('Administrators')],
28         ];
29     }
30
31 /**
32 * @dataProvider provideRoles
33 */
34     public function testVisitSomeRole(Role $role)
```

(suite sur la page suivante)

(suite de la page précédente)

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

1.4 En savoir plus

1.4.1 Service Locator

CECI EST CONSIDÉRÉ COMME UN ANTI-PATTERN!

Le Service Locator est considéré par certains comme un anti-pattern. Il viole le principe d'inversion des dépendances. Le Service Locator cache les dépendances des classes au lieu de les exposer comme vous le feriez en utilisant l'injection de dépendances. En cas de modification de ces dépendances, vous risquez de casser la fonctionnalité des classes qui les utilisent, ce qui rend votre système difficile à maintenir.

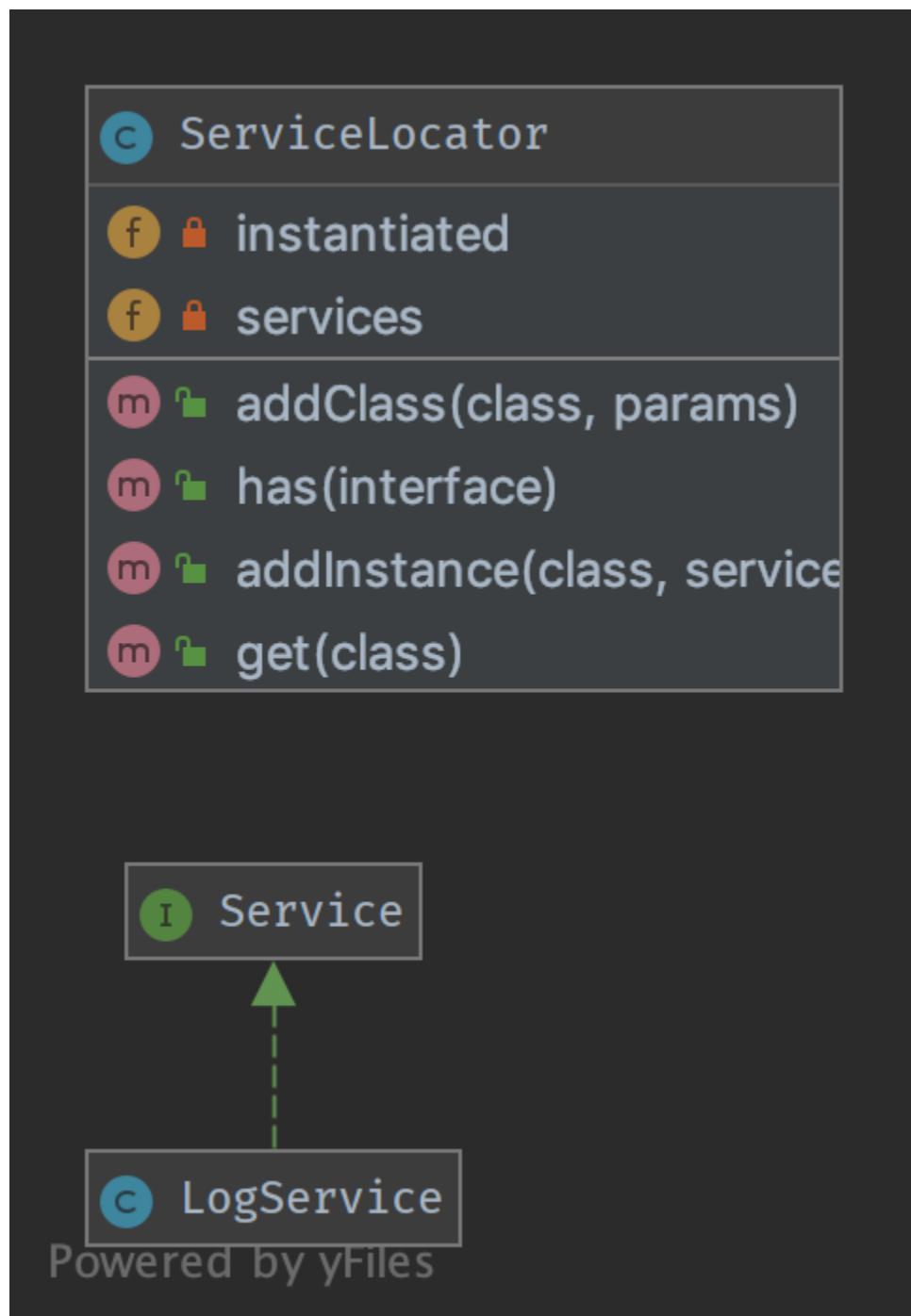
Rôle

Mettre en œuvre une architecture faiblement couplée afin d'obtenir un code plus facile à tester, à maintenir et à étendre. Le modèle DI et le modèle Service Locator sont une mise en œuvre du modèle Inverse of Control.

Rôle

Avec ServiceLocator, vous pouvez enregistrer un service pour une interface donnée. En utilisant l'interface, vous pouvez récupérer le service et l'utiliser dans les classes de l'application sans connaître son implémentation. Vous pouvez configurer et injecter l'objet Service Locator au démarrage.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Service.php

```

1 <?php
2
3 namespace DesignPatterns\More\ServiceLocator;
4
5 interface Service
6 {
7 }
```

ServiceLocator.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\ServiceLocator;
6
7 use OutOfRangeException;
8 use InvalidArgumentException;
9
10 class ServiceLocator
11 {
12     /**
13      * @var string[][]
14      */
15     private array $services = [];
16
17     /**
18      * @var Service[]
19      */
20     private array $instantiated = [];
21
22     public function addInstance(string $class, Service $service)
23     {
24         $this->instantiated[$class] = $service;
25     }
26
27     public function addClass(string $class, array $params)
28     {
29         $this->services[$class] = $params;
30     }
31
32     public function has(string $interface): bool
33     {
34         return isset($this->services[$interface]) || isset($this->instantiated[
35             $interface]);
36     }
37
38     public function get(string $class): Service
```

(suite sur la page suivante)

(suite de la page précédente)

```

38     {
39         if (isset($this->instantiated[$class])) {
40             return $this->instantiated[$class];
41         }
42
43         $object = new $class(...$this->services[$class]);
44
45         if (!$object instanceof Service) {
46             throw new InvalidArgumentException('Could not register service: is no ↴  
instance of Service');
47         }
48
49         $this->instantiated[$class] = $object;
50
51         return $object;
52     }
53 }
```

LogService.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\ServiceLocator;
6
7 class LogService implements Service
8 {
9 }
```

Test

Tests/ServiceLocatorTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\ServiceLocator\Tests;
6
7 use DesignPatterns\More\ServiceLocator\LogService;
8 use DesignPatterns\More\ServiceLocator\ServiceLocator;
9 use PHPUnit\Framework\TestCase;
10
11 class ServiceLocatorTest extends TestCase
12 {
13     private ServiceLocator $serviceLocator;
14
15     public function setUp(): void
16     {
17         $this->serviceLocator = new ServiceLocator();
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

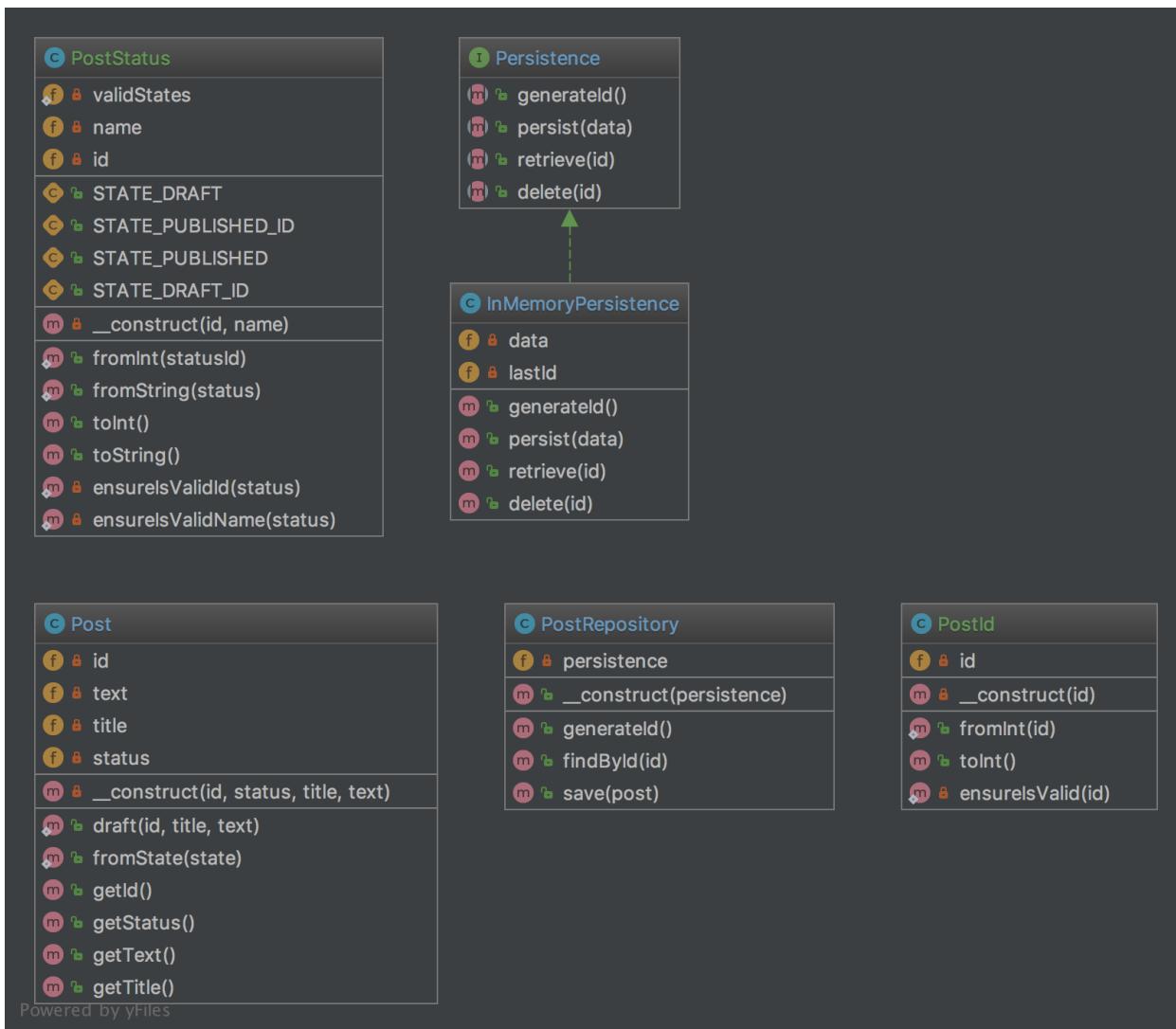
Rôle

Assure la médiation entre les couches de domaine et de mappage de données en utilisant une interface de type collection pour accéder aux objets de domaine. Repository encapsule l'ensemble des objets persistés dans un stockage de données et les opérations effectuées sur eux, fournissant une vue orientée objet de la couche de persistance. Repository permet également de réaliser une séparation nette et une dépendance unidirectionnelle entre les couches de domaine et de mappage de données.

Exemples

- Doctrine 2 ORM : Le Repository sert de médiateur entre Entity et DBAL et contient des méthodes pour récupérer les objets.
- Laravel Framework

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Post.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository\Domain;
6
7 class Post
8 {
9     public static function draft(PostId $id, string $title, string $text): Post
10    {
  
```

(suite sur la page suivante)

(suite de la page précédente)

```
11     return new self(
12         $id,
13         PostStatus::fromString(PostStatus::STATE_DRAFT),
14         $title,
15         $text
16     );
17 }
18
19 public static function fromState(array $state): Post
20 {
21     return new self(
22         PostId::fromInt($state['id']),
23         PostStatus::fromInt($state['statusId']),
24         $state['title'],
25         $state['text']
26     );
27 }
28
29 private function __construct(
30     private PostId $id,
31     private PostStatus $status,
32     private string $title,
33     private string $text
34 ) {
35 }
36
37 public function getId(): PostId
38 {
39     return $this->id;
40 }
41
42 public function getStatus(): PostStatus
43 {
44     return $this->status;
45 }
46
47 public function getText(): string
48 {
49     return $this->text;
50 }
51
52 public function getTitle(): string
53 {
54     return $this->title;
55 }
56 }
```

PostId.php

```
1 <?php
2
3 declare(strict_types=1);
```

(suite sur la page suivante)

(suite de la page précédente)

```

4
5 namespace DesignPatterns\More\Repository\Domain;
6
7 use InvalidArgumentException;
8
9 /**
10 * This is a perfect example of a value object that is identifiable by its value alone.
11 * and
12 * is guaranteed to be valid each time an instance is created. Another important
13 * property of value objects
14 * is immutability.
15 *
16 * Notice also the use of a named constructor (fromInt) which adds a little context when
17 * creating an instance.
18 */
19 class PostId
20 {
21     public static function fromInt(int $id): PostId
22     {
23         self::ensureIsValid($id);
24
25         return new self($id);
26     }
27
28
29     public function toInt(): int
30     {
31         return $this->id;
32     }
33
34     private static function ensureIsValid(int $id)
35     {
36         if ($id <= 0) {
37             throw new InvalidArgumentException('Invalid PostId given');
38         }
39     }
40 }
```

PostStatus.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository\Domain;
6
7 use InvalidArgumentException;
8
9 /**
```

(suite sur la page suivante)

(suite de la page précédente)

```

10  * Like PostId, this is a value object which holds the value of the current status of a
11  * ↵Post. It can be constructed
12  * either from a string or int and is able to validate itself. An instance can then be
13  * ↵converted back to int or string.
14  */
15 class PostStatus
16 {
17     public const STATE_DRAFT_ID = 1;
18     public const STATE_PUBLISHED_ID = 2;
19
20     public const STATE_DRAFT = 'draft';
21     public const STATE_PUBLISHED = 'published';
22
23     private static array $validStates = [
24         self::STATE_DRAFT_ID => self::STATE_DRAFT,
25         self::STATE_PUBLISHED_ID => self::STATE_PUBLISHED,
26     ];
27
28     public static function fromInt(int $statusId)
29     {
30         self::ensureIsValidId($statusId);
31
32         return new self($statusId, self::$validStates[$statusId]);
33     }
34
35     public static function fromString(string $status)
36     {
37         self::ensureIsValidName($status);
38         $state = array_search($status, self::$validStates);
39
40         if ($state === false) {
41             throw new InvalidArgumentException('Invalid state given!');
42         }
43
44         return new self($state, $status);
45     }
46
47     private function __construct(private int $id, private string $name)
48     {
49     }
50
51     public function toInt(): int
52     {
53         return $this->id;
54     }
55
56     /**
57      * there is a reason that I avoid using __toString() as it operates outside of the
58      * ↵stack in PHP
59      * and is therefore not able to operate well with exceptions
60      */
61     public function toString(): string

```

(suite sur la page suivante)

(suite de la page précédente)

```

59     {
60         return $this->name;
61     }
62
63     private static function ensureIsValidId(int $status)
64     {
65         if (!in_array($status, array_keys(self::$validStates), true)) {
66             throw new InvalidArgumentException('Invalid status id given');
67         }
68     }
69
70
71     private static function ensureIsValidName(string $status)
72     {
73         if (!in_array($status, self::$validStates, true)) {
74             throw new InvalidArgumentException('Invalid status name given');
75         }
76     }
77 }
```

PostRepository.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository;
6
7 use OutOfBoundsException;
8 use DesignPatterns\More\Repository\Domain\Post;
9 use DesignPatterns\More\Repository\Domain\PostId;
10
11 /**
12 * This class is situated between Entity layer (class Post) and access object layer
13 * (Persistence).
14 *
15 * Repository encapsulates the set of objects persisted in a data store and the
16 * operations performed over them
17 * providing a more object-oriented view of the persistence layer
18 *
19 * Repository also supports the objective of achieving a clean separation and one-way
20 * dependency
21 * between the domain and data mapping layers
22 */
23
24 class PostRepository
25 {
26     public function __construct(private Persistence $persistance)
27     {
28     }
29
30     public function generateId(): PostId
31     {
32 }
```

(suite sur la page suivante)

(suite de la page précédente)

```

28     return PostId::fromInt($this->persistence->generateId());
29 }
30
31     public function findById(PostId $id): Post
32 {
33     try {
34         $arrayData = $this->persistence->retrieve($id->toInt());
35     } catch (OutOfBoundsException $e) {
36         throw new OutOfBoundsException(sprintf('Post with id %d does not exist', $id-
37             >toInt(), 0, $e));
38     }
39
40     return Post::fromState($arrayData);
41 }
42
43     public function save(Post $post)
44 {
45         $this->persistence->persist([
46             'id' => $post->getId()->toInt(),
47             'statusId' => $post->getStatus()->toInt(),
48             'text' => $post->getText(),
49             'title' => $post->getTitle(),
50         ]);
51 }

```

Persistence.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository;
6
7 interface Persistence
8 {
9     public function generateId(): int;
10
11    public function persist(array $data);
12
13    public function retrieve(int $id): array;
14
15    public function delete(int $id);
16 }

```

InMemoryPersistence.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository;

```

(suite sur la page suivante)

(suite de la page précédente)

```

6   use OutOfBoundsException;
7
8
9 class InMemoryPersistence implements Persistence
10 {
11     private array $data = [];
12     private int $lastId = 0;
13
14     public function generateId(): int
15     {
16         $this->lastId++;
17
18         return $this->lastId;
19     }
20
21     public function persist(array $data)
22     {
23         $this->data[$this->lastId] = $data;
24     }
25
26     public function retrieve(int $id): array
27     {
28         if (!isset($this->data[$id])) {
29             throw new OutOfBoundsException(sprintf('No data found for ID %d', $id));
30         }
31
32         return $this->data[$id];
33     }
34
35     public function delete(int $id)
36     {
37         if (!isset($this->data[$id])) {
38             throw new OutOfBoundsException(sprintf('No data found for ID %d', $id));
39         }
40
41         unset($this->data[$id]);
42     }
43 }
```

Test

Tests/PostRepositoryTest.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\Repository\Tests;
6
7 use OutOfBoundsException;
8 use DesignPatterns\More\Repository\Domain\PostId;
```

(suite sur la page suivante)

(suite de la page précédente)

```
9  use DesignPatterns\More\Repository\Domain\PostStatus;
10 use DesignPatterns\More\Repository\InMemoryPersistence;
11 use DesignPatterns\More\Repository\Domain\Post;
12 use DesignPatterns\More\Repository\PostRepository;
13 use PHPUnit\Framework\TestCase;
14
15 class PostRepositoryTest extends TestCase
16 {
17     private PostRepository $repository;
18
19     protected function setUp(): void
20     {
21         $this->repository = new PostRepository(new InMemoryPersistence());
22     }
23
24     public function testCanGenerateId()
25     {
26         $this->assertEquals(1, $this->repository->generateId()->toInt());
27     }
28
29     public function testThrowsExceptionWhenTryingToFindPostWhichDoesNotExist()
30     {
31         $this->expectException(OutOfBoundsException::class);
32         $this->expectExceptionMessage('Post with id 42 does not exist');
33
34         $this->repository->findById(PostId::fromInt(42));
35     }
36
37     public function testCanPersistPostDraft()
38     {
39         $postId = $this->repository->generateId();
40         $post = Post::draft($postId, 'Repository Pattern', 'Design Patterns PHP');
41         $this->repository->save($post);
42
43         $this->repository->findById($postId);
44
45         $this->assertEquals($postId, $this->repository->findById($postId)->getId());
46         $this->assertEquals(PostStatus::STATE_DRAFT, $post->getStatus()->toString());
47     }
48 }
```

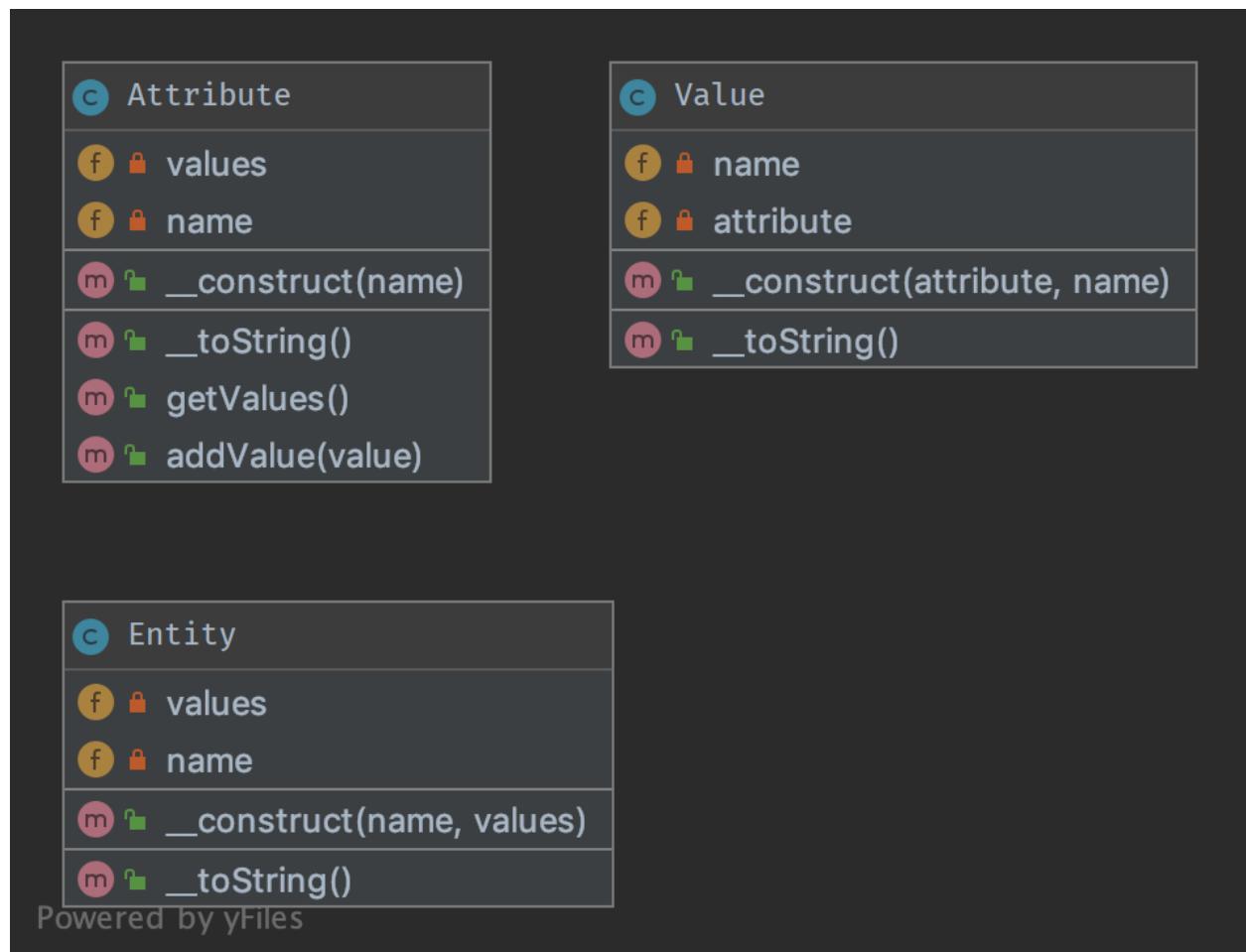
1.4.3 Entity-Attribute-Value (EAV)

Le pattern Entité-attribut-valeur (EAV) afin d'implémenter le modèle EAV avec PHP.

Rôle

Le modèle Entité-attribut-valeur (EAV) est un modèle de données permettant de décrire des entités pour lesquelles le nombre d'attributs (propriétés, paramètres) pouvant être utilisés pour les décrire est potentiellement vaste, mais le nombre qui s'appliquera réellement à une entité donnée est relativement modeste.

Diagramme UML



Code

Vous pouvez également trouver ce code sur [GitHub](#)

Entity.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\EAV;
6
7 use SplObjectStorage;
8
9 class Entity implements \Stringable
10 {
11     /**
12      * @var SplObjectStorage<Value, Value>
13      */
14     private $values;
15
16     /**
17      * @param Value[] $values
18      */
19     public function __construct(private string $name, array $values)
20     {
21         $this->values = new SplObjectStorage();
22
23         foreach ($values as $value) {
24             $this->values->attach($value);
25         }
26     }
27
28     public function __toString(): string
29     {
30         $text = [$this->name];
31
32         foreach ($this->values as $value) {
33             $text[] = (string) $value;
34         }
35
36         return join(' ', $text);
37     }
38 }
```

Attribute.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\EAV;
6
7 use SplObjectStorage;
```

(suite sur la page suivante)

(suite de la page précédente)

```

8 class Attribute implements \Stringable
9 {
10     private SplObjectStorage $values;
11
12     public function __construct(private string $name)
13     {
14         $this->values = new SplObjectStorage();
15     }
16
17     public function addValue(Value $value): void
18     {
19         $this->values->attach($value);
20     }
21
22     public function getValues(): SplObjectStorage
23     {
24         return $this->values;
25     }
26
27     public function __toString(): string
28     {
29         return $this->name;
30     }
31 }
32

```

Value.php

```

1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\EAV;
6
7 class Value implements \Stringable
8 {
9     public function __construct(private Attribute $attribute, private string $name)
10    {
11        $attribute->addValue($this);
12    }
13
14    public function __toString(): string
15    {
16        return sprintf('%s: %s', (string) $this->attribute, $this->name);
17    }
18 }

```

Test

Tests/EAVTest.php

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\More\EAV\Tests;
6
7 use DesignPatterns\More\EAV\Attribute;
8 use DesignPatterns\More\EAV\Entity;
9 use DesignPatterns\More\EAV\Value;
10 use PHPUnit\Framework\TestCase;
11
12 class EAVTest extends TestCase
13 {
14     public function testCanAddAttributeToEntity(): void
15     {
16         $colorAttribute = new Attribute('color');
17         $colorSilver = new Value($colorAttribute, 'silver');
18         $colorBlack = new Value($colorAttribute, 'black');
19
20         $memoryAttribute = new Attribute('memory');
21         $memory8Gb = new Value($memoryAttribute, '8GB');
22
23         $entity = new Entity('MacBook Pro', [$colorSilver, $colorBlack, $memory8Gb]);
24
25         $this->assertEquals('MacBook Pro, color: silver, color: black, memory: 8GB', ↵
26             (string) $entity);
27     }
}
```