
DesignPatternsPHP Documentation

Versió 1.0

Dominik Liebler and contributors

30 de set., 2023

1	Patterns	3
1.1	Creational	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	Structural	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	63
1.2.10	Proxy	68
1.2.11	Registry	72
1.3	Behavioral	75
1.3.1	Chain Of Responsibilities	75
1.3.2	Command	79
1.3.3	Interpreter	86
1.3.4	Iterator	90
1.3.5	Mediator	95
1.3.6	Memento	99
1.3.7	Null Object	104
1.3.8	Observer	108
1.3.9	Specification	111
1.3.10	State	116
1.3.11	Strategy	120
1.3.12	Template Method	125
1.3.13	Visitor	129

1.4	More	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.

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 Creational

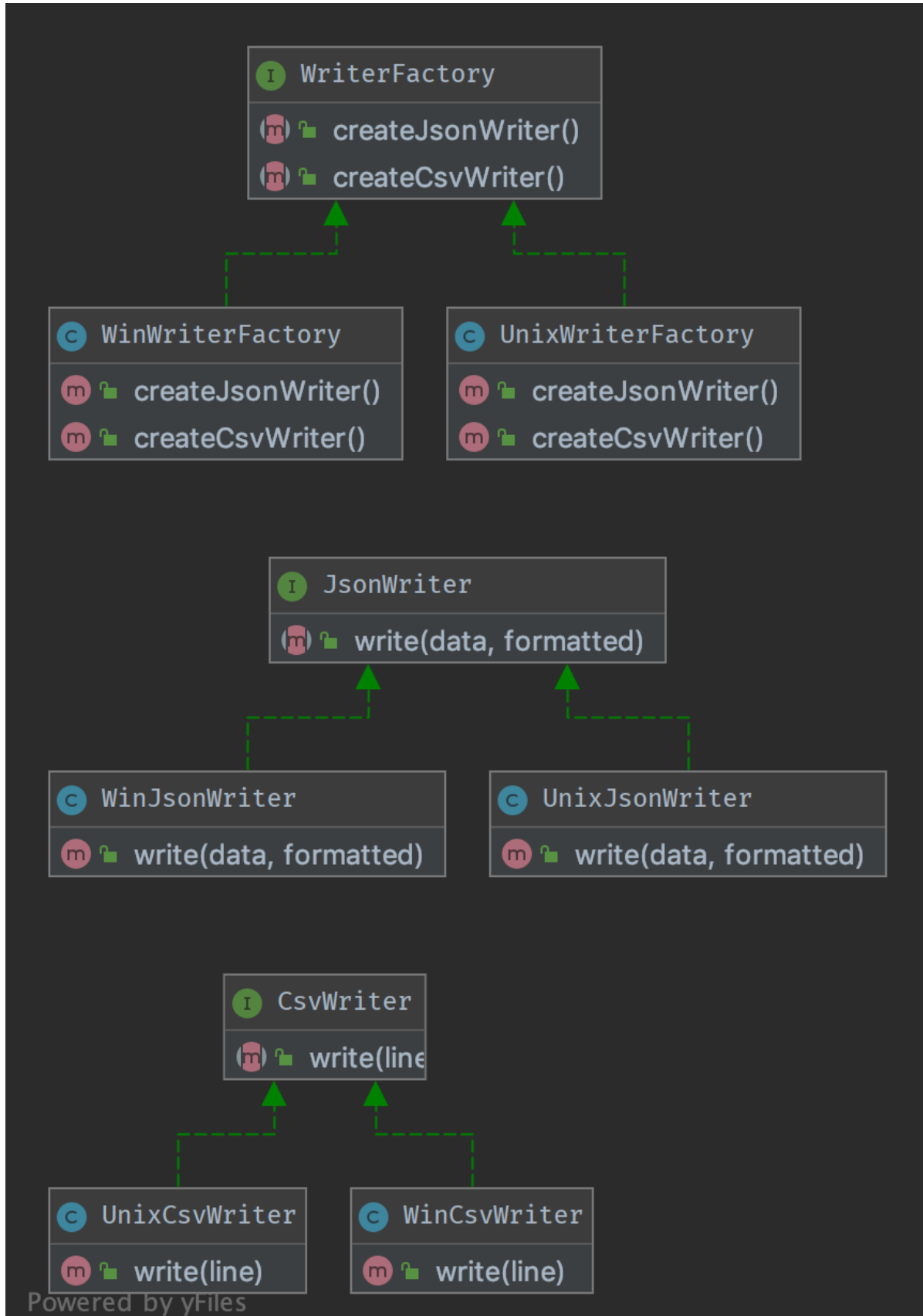
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns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

1.1.1 Abstract Factory

Purpose

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

UML Diagram



Code

You can also find this code on [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/AbstractFactory/Test.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;

```

(continua a la pàgina següent)

```
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

Purpose

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

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

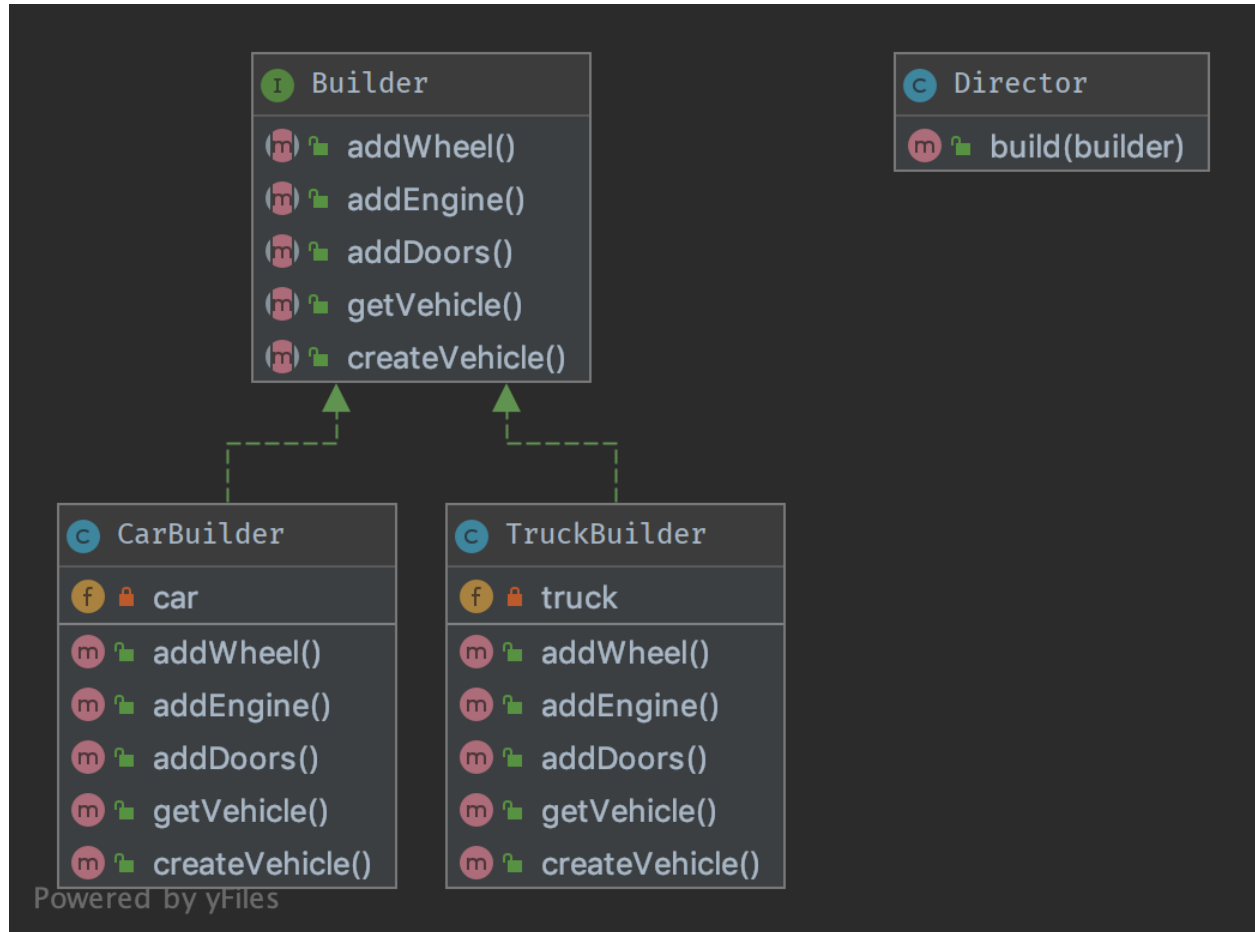
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

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

Examples

- PHPUnit: Mock Builder

UML Diagram



Code

You can also find this code on [GitHub](#)

Director.php

```

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
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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;

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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;

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```
6
7 class Truck extends Vehicle
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

Purpose

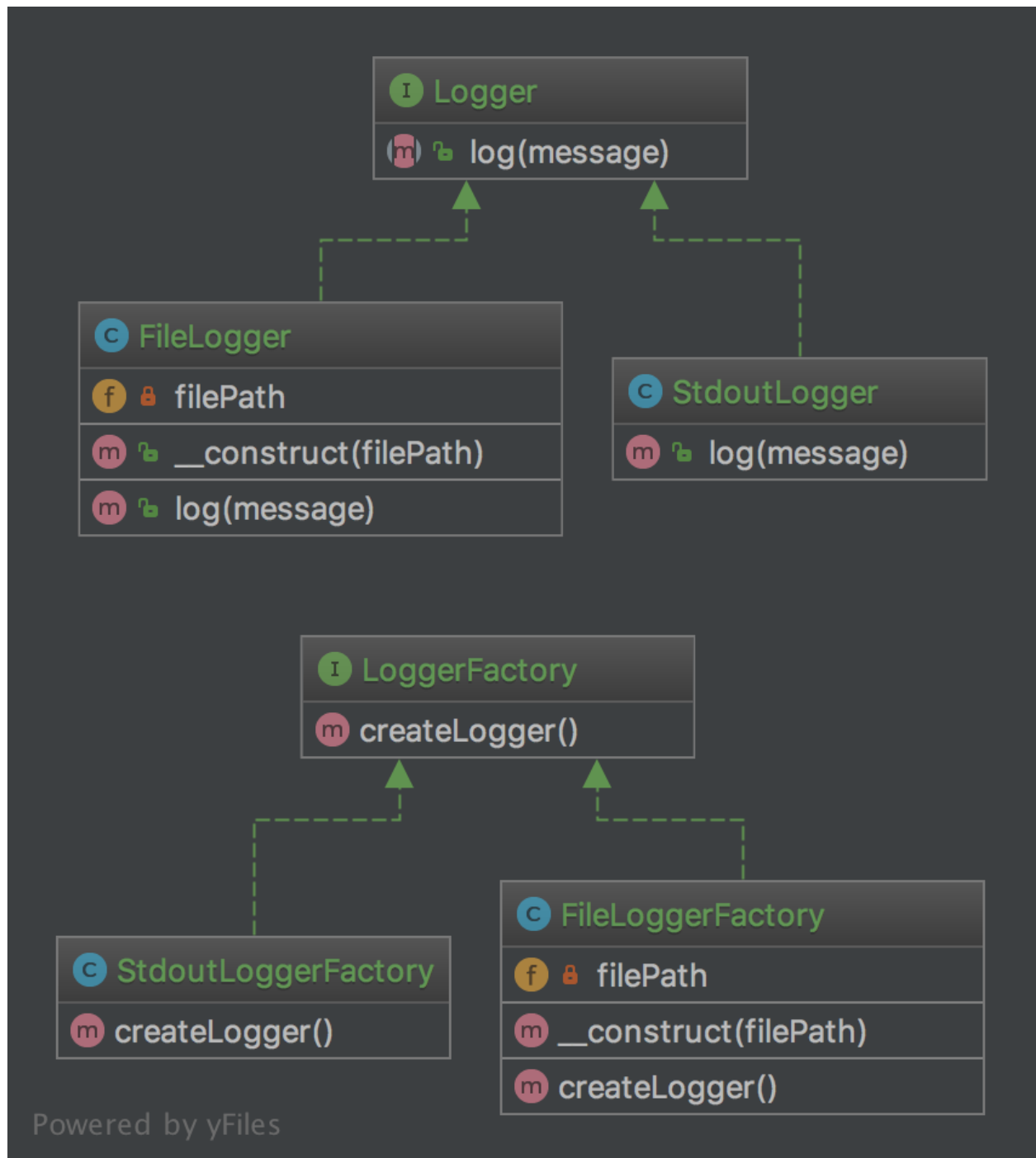
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects.

For simple cases, this abstract class could be just an interface.

This pattern is a «real» Design Pattern because it achieves the Dependency Inversion principle a.k.a the «D» in SOLID principles.

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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

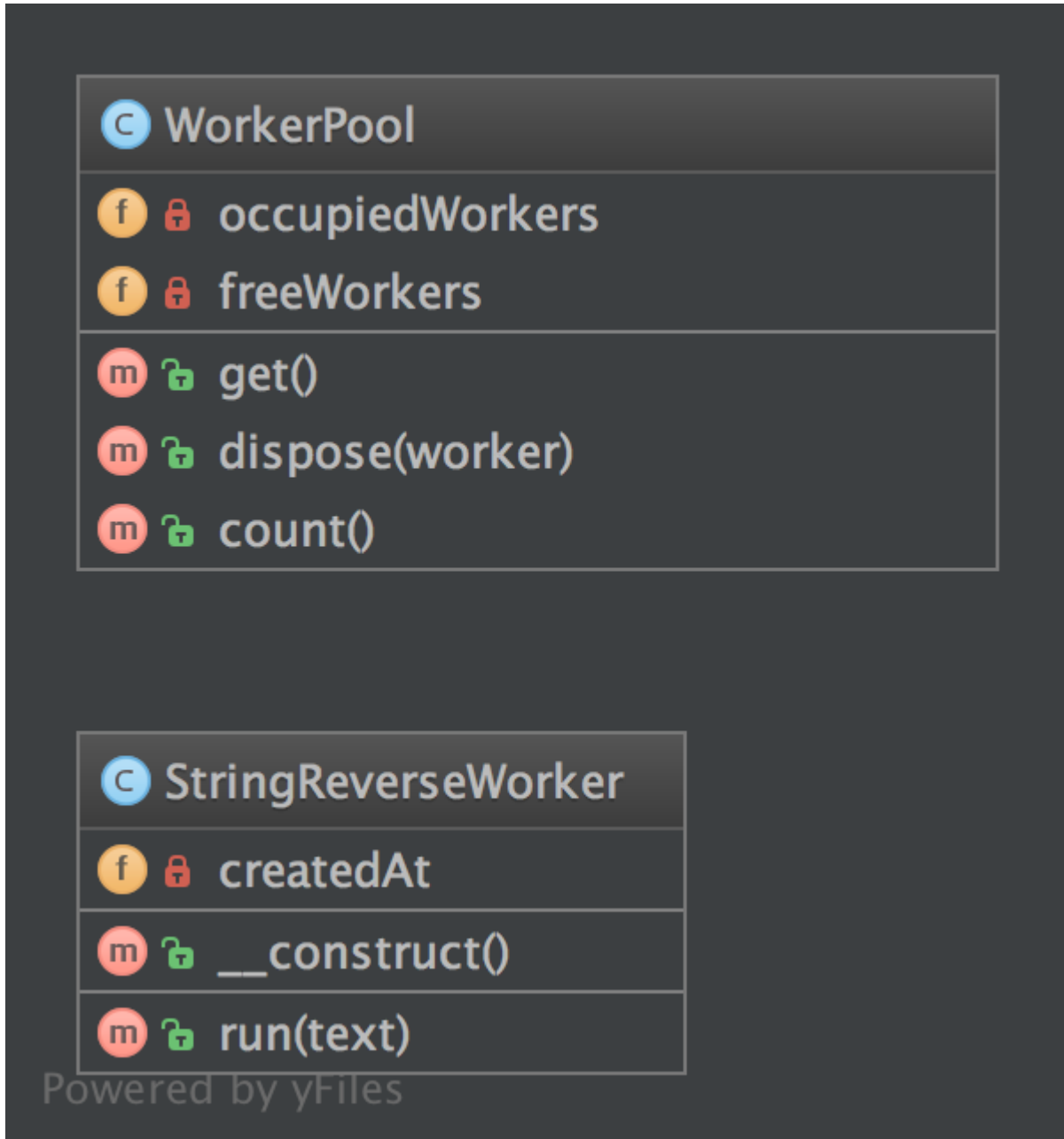
Purpose

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

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

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

UML Diagram



Code

You can also find this code on [GitHub](#)

WorkerPool.php

```
1 <?php
2
3 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

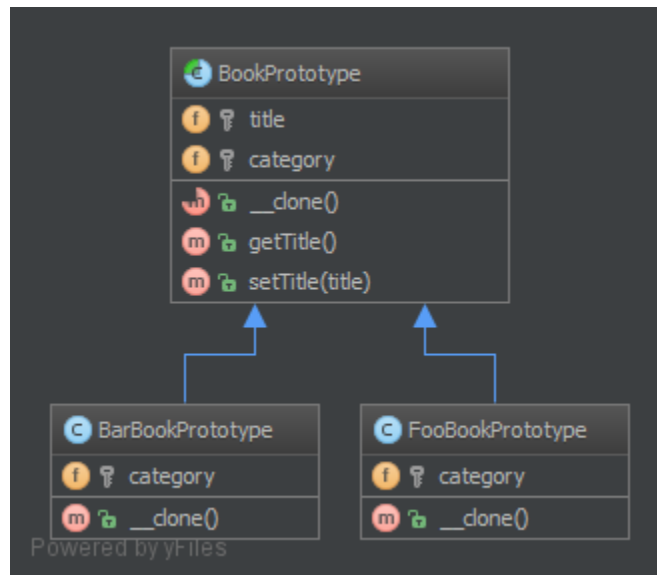
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

BookPrototype.php

```

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
14     final public function getTitle(): string
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```
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 }
```

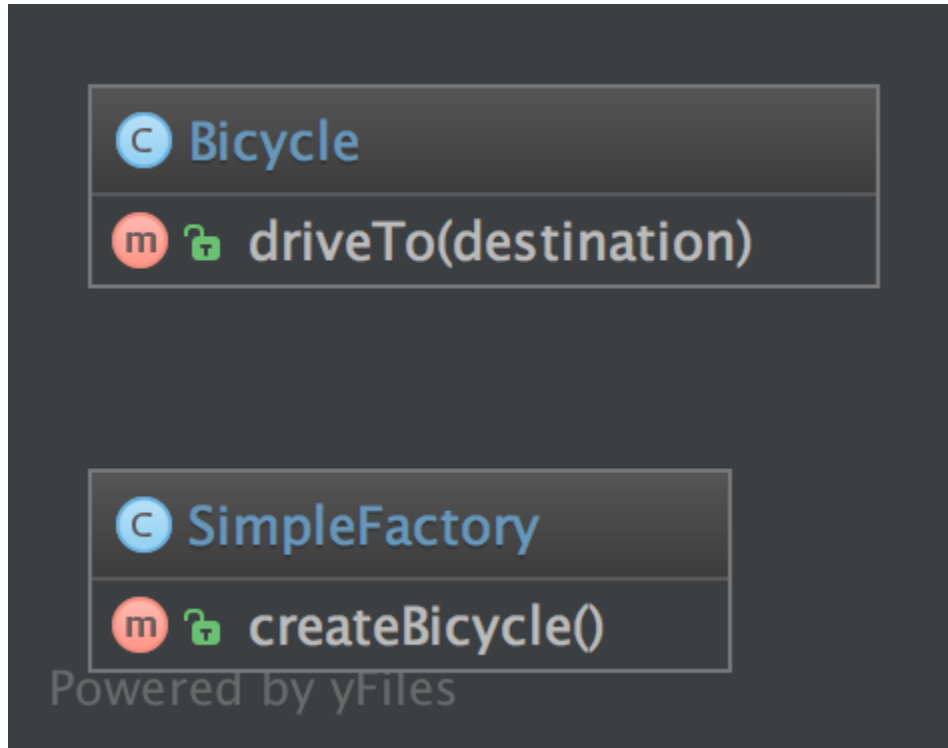
1.1.6 Simple Factory

Purpose

SimpleFactory is a simple factory pattern.

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

UML Diagram



Code

You can also find this code on [GitHub](#)

SimpleFactory.php

```

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
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```
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

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

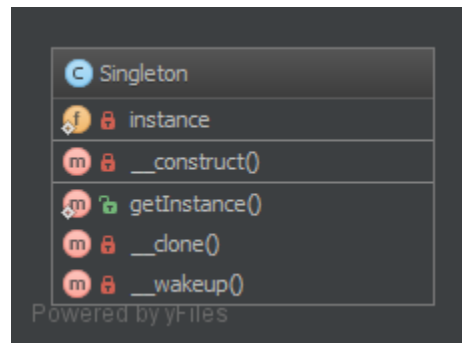
Purpose

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

Examples

- DB Connector
- Logger
- Config Manager
- Threads Handling
- Lock file for the application (there is only one in the filesystem ...)

UML Diagram



Code

You can also find this code on [GitHub](#)

Singleton.php

```

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) {
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

19         self::$instance = new self();
20     }
21
22     return self::$instance;
23 }
24
25 /**
26  * is not allowed to call from outside to prevent from creating multiple instances,
27  * to use the singleton, you have to obtain the instance from
↳ 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
↳ it)
35  */
36 private function __clone()
37 {
38 }
39
40 /**
41  * prevent from being unserialized (which would create a second instance of it)
42  */
43 public function __wakeup()
44 {
45     throw new Exception("Cannot unserialize singleton");
46 }
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();

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

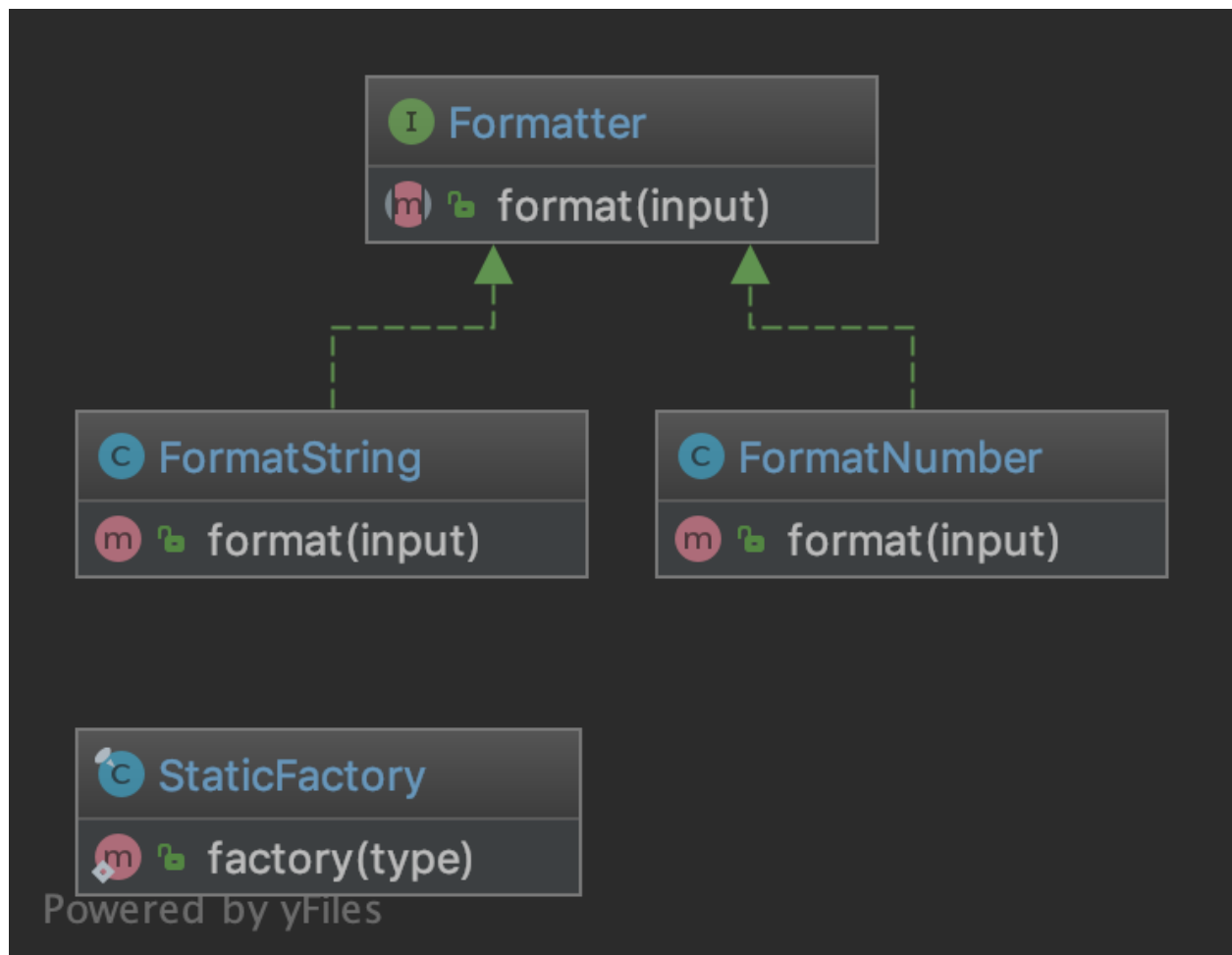
```
16     $this->assertInstanceOf(Singleton::class, $firstCall);
17     $this->assertSame($firstCall, $secondCall);
18 }
19 }
20 }
```

1.1.8 Static Factory

Purpose

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

UML Diagram



Code

You can also find this code on [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 {
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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()

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

1.2 Structural

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

1.2.1 Adapter / Wrapper

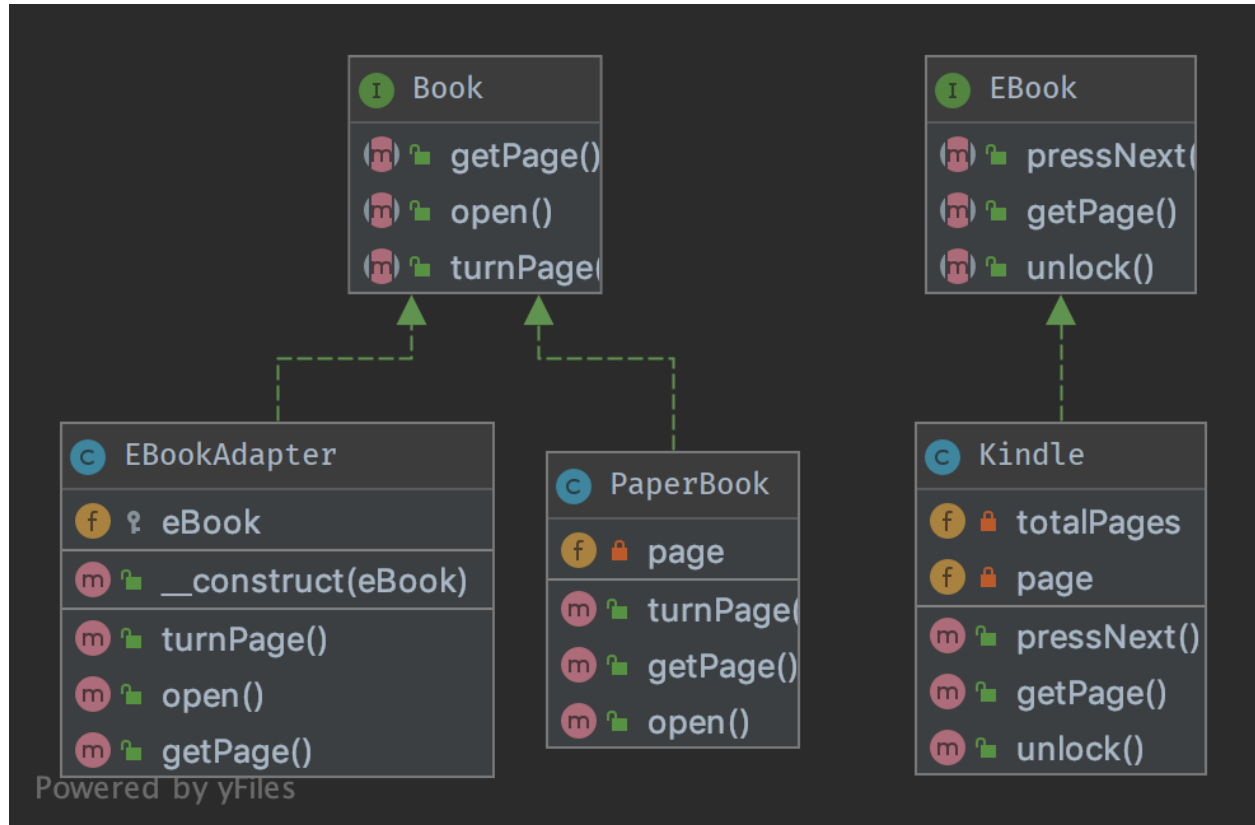
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [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);
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     }
39 }

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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()

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

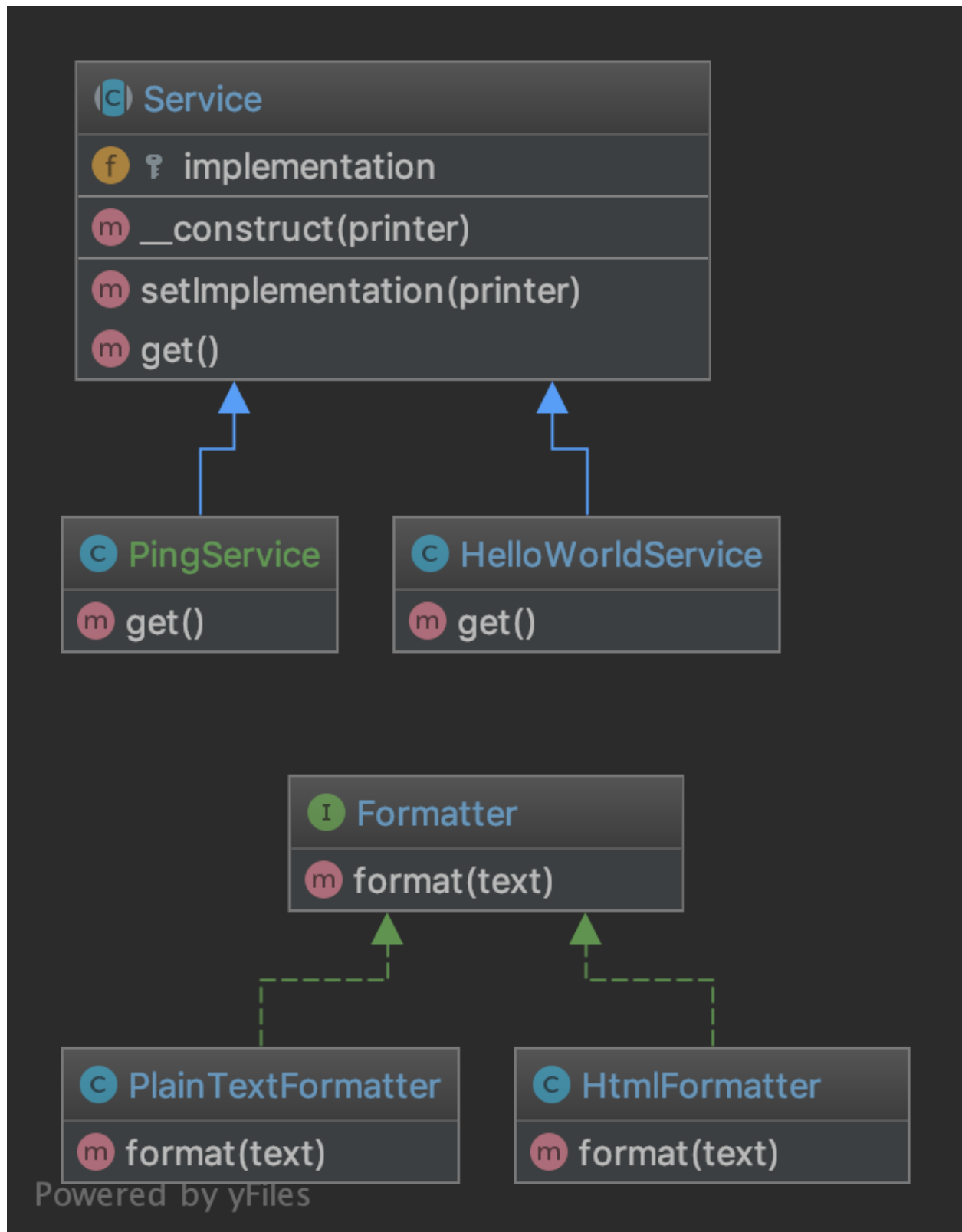
```
24 {  
25     $kindle = new Kindle();  
26     $book = new EBookAdapter($kindle);  
27  
28     $book->open();  
29     $book->turnPage();  
30  
31     $this->assertSame(2, $book->getPage());  
32 }  
33 }
```

1.2.2 Bridge

Purpose

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

UML Diagram



Code

You can also find this code on [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);
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

HelloWorldService.php

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

PingService.php

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

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

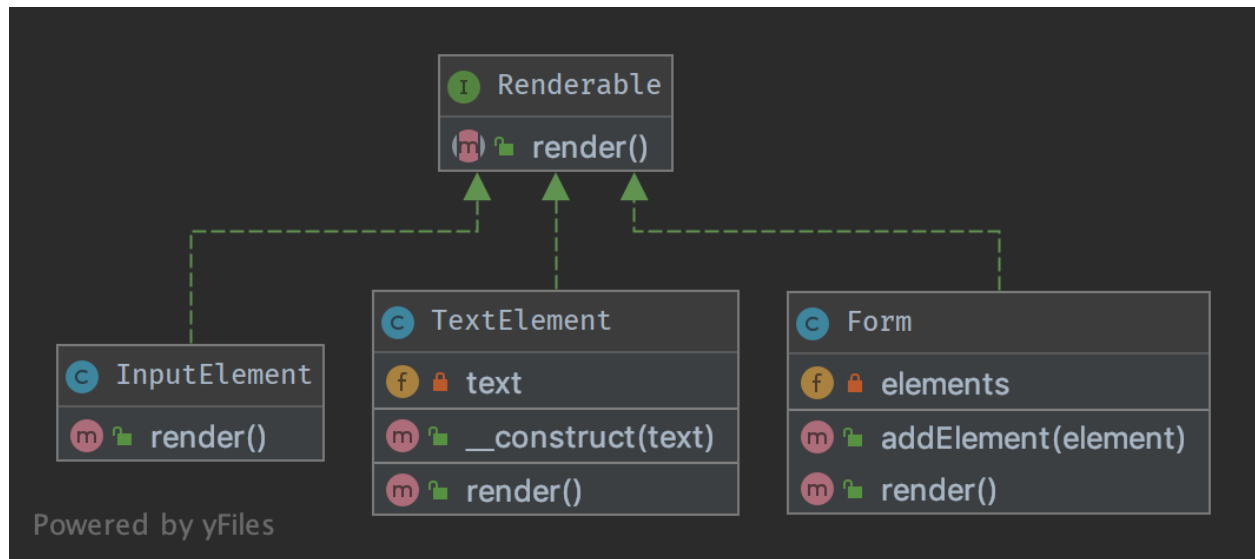
Purpose

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

Examples

- a form class instance handles all its form elements like a single instance of the form, when `render()` is called, it subsequently runs through all its child elements and calls `render()` on them

UML Diagram



Code

You can also find this code on [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 /**
8  * The composite node MUST extend the component contract. This is mandatory for building
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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      * from the outside, one will not see this and the form will act like a single
23      ↪ object instance
24      */
25     public function render(): string
26     {
27         $formCode = '<form>';
28
29         foreach ($this->elements as $element) {
30             $formCode .= $element->render();
31         }
32
33         return $formCode . '</form>';
34     }
35
36     public function addElement(Renderable $element)
37     {
38         $this->elements[] = $element;
39     }
40 }

```

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>',

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

1.2.4 Data Mapper

Purpose

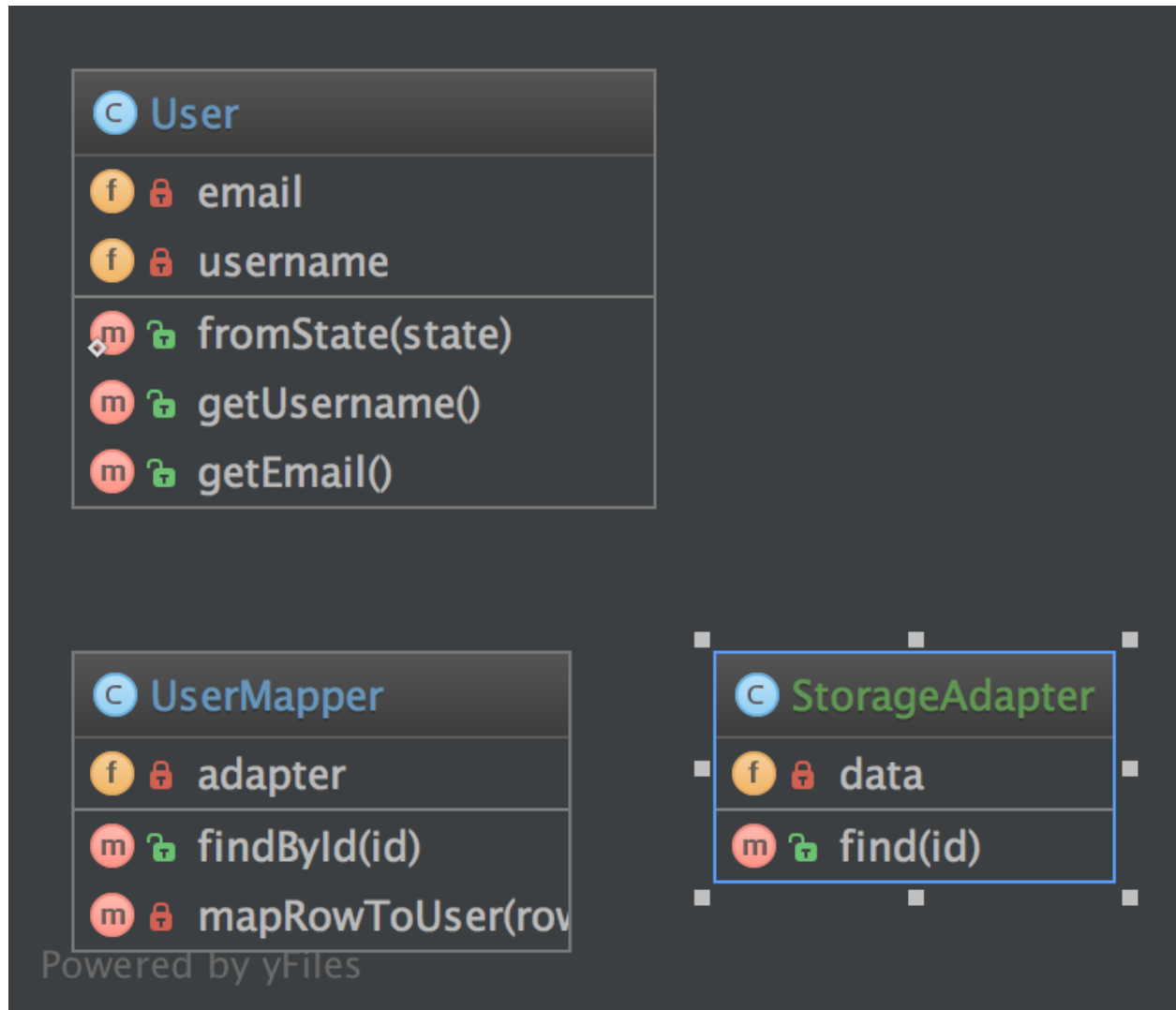
A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent of each other and the data mapper itself. The layer is composed of one or more mappers (or Data Access Objects), performing the data transfer. Mapper implementations vary in scope. Generic mappers will handle many different domain entity types, dedicated mappers will handle one or a few.

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

User.php

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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) {
28
29         }
30     }
31 }

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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;

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     }
35 }

```

1.2.5 Decorator

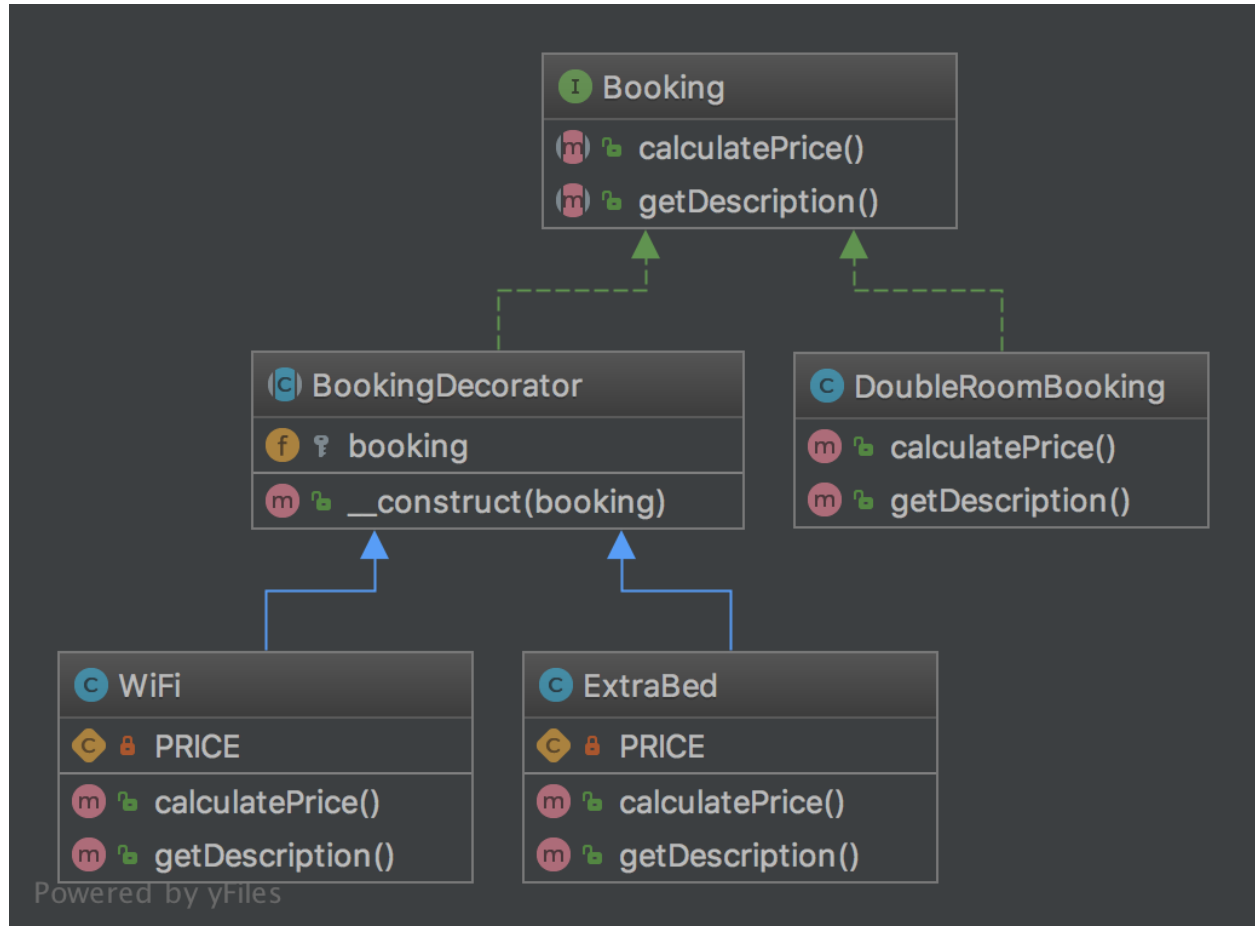
Purpose

To dynamically add new functionality to class instances.

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     }

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

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

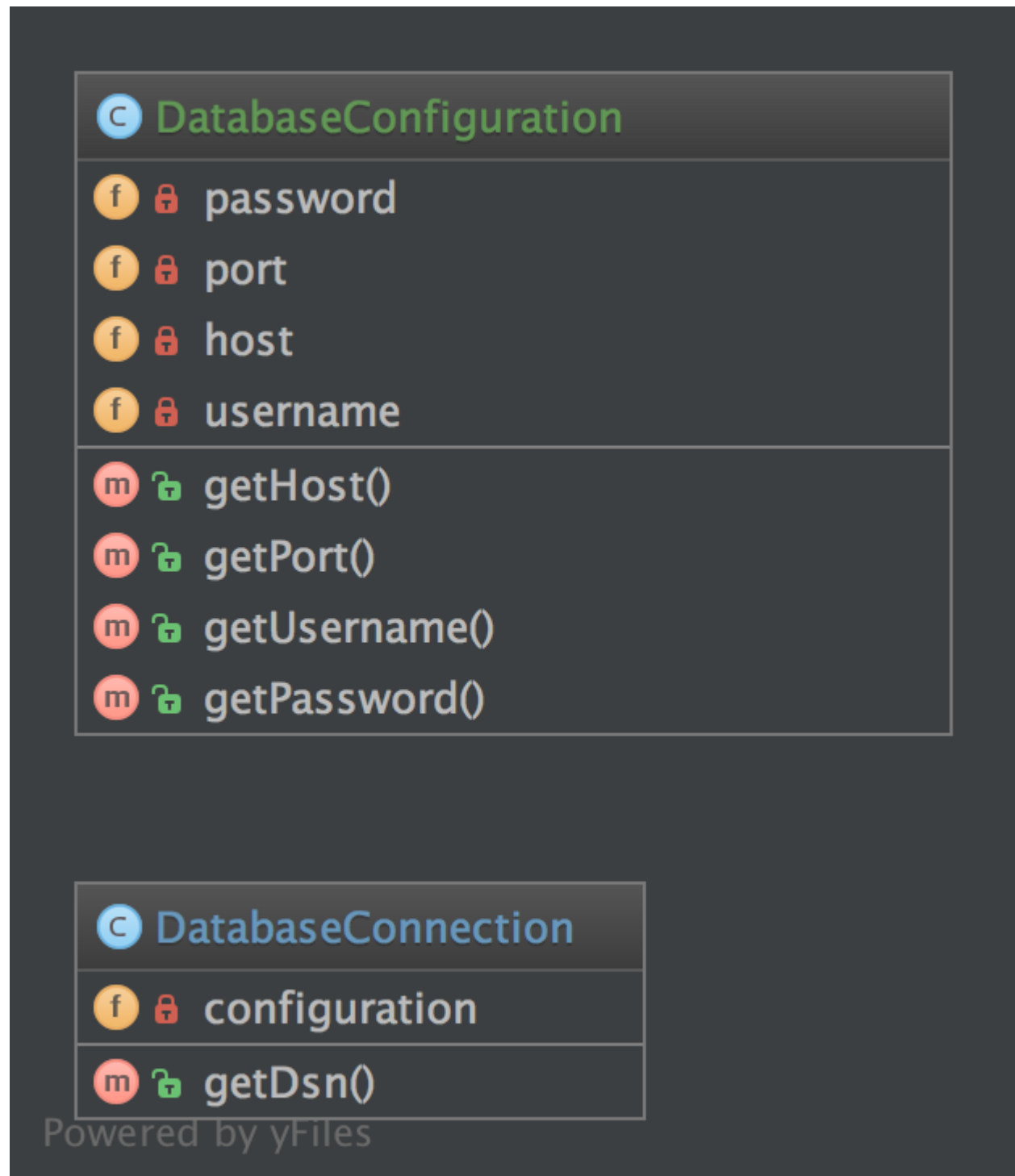
Usage

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

DatabaseConfiguration.php

```

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)
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

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.

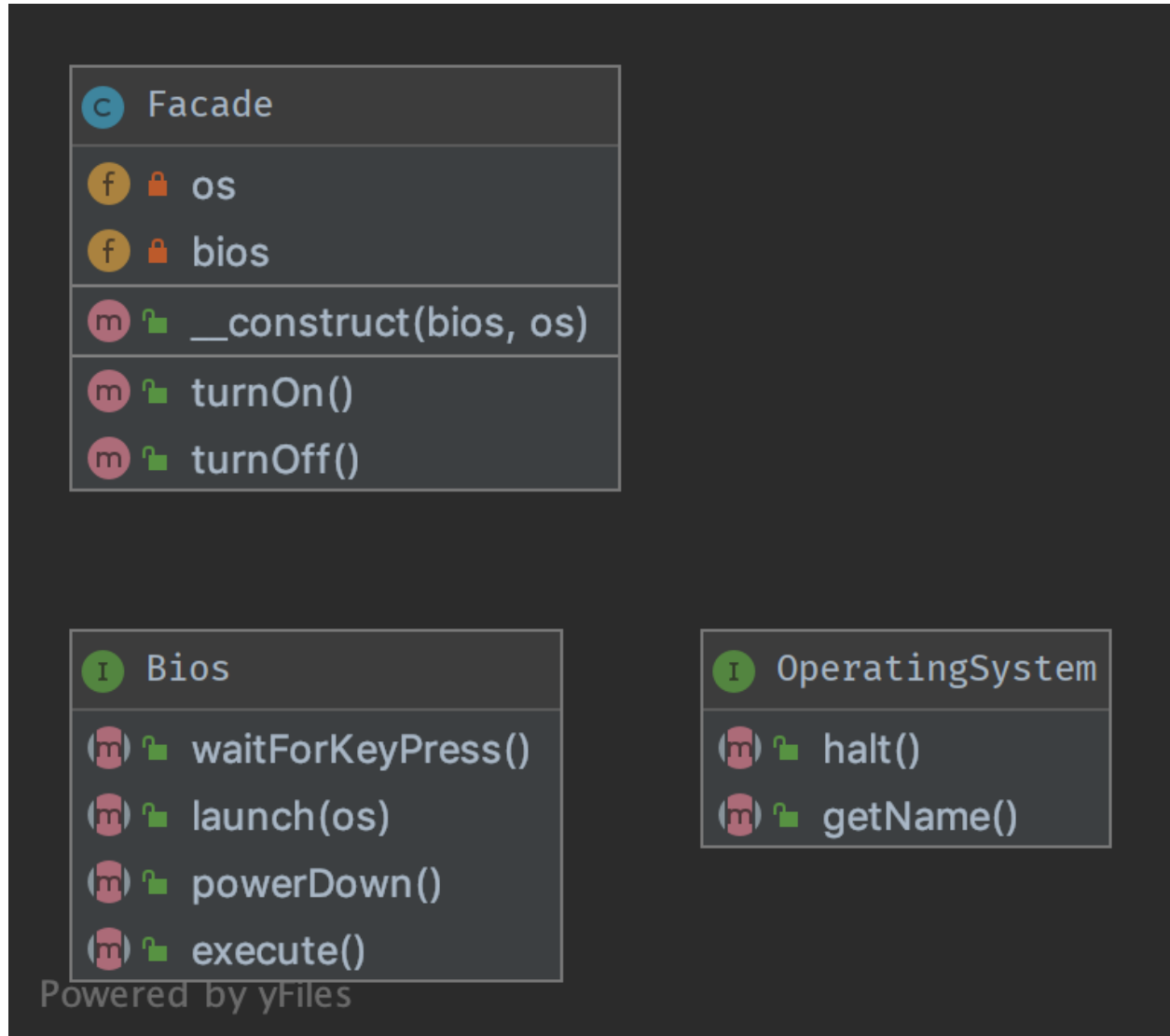
A Facade is meant to decouple a client and a sub-system by embedding many (but sometimes just one) interface, and of course to reduce complexity.

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

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

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Facade.php

```

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)
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

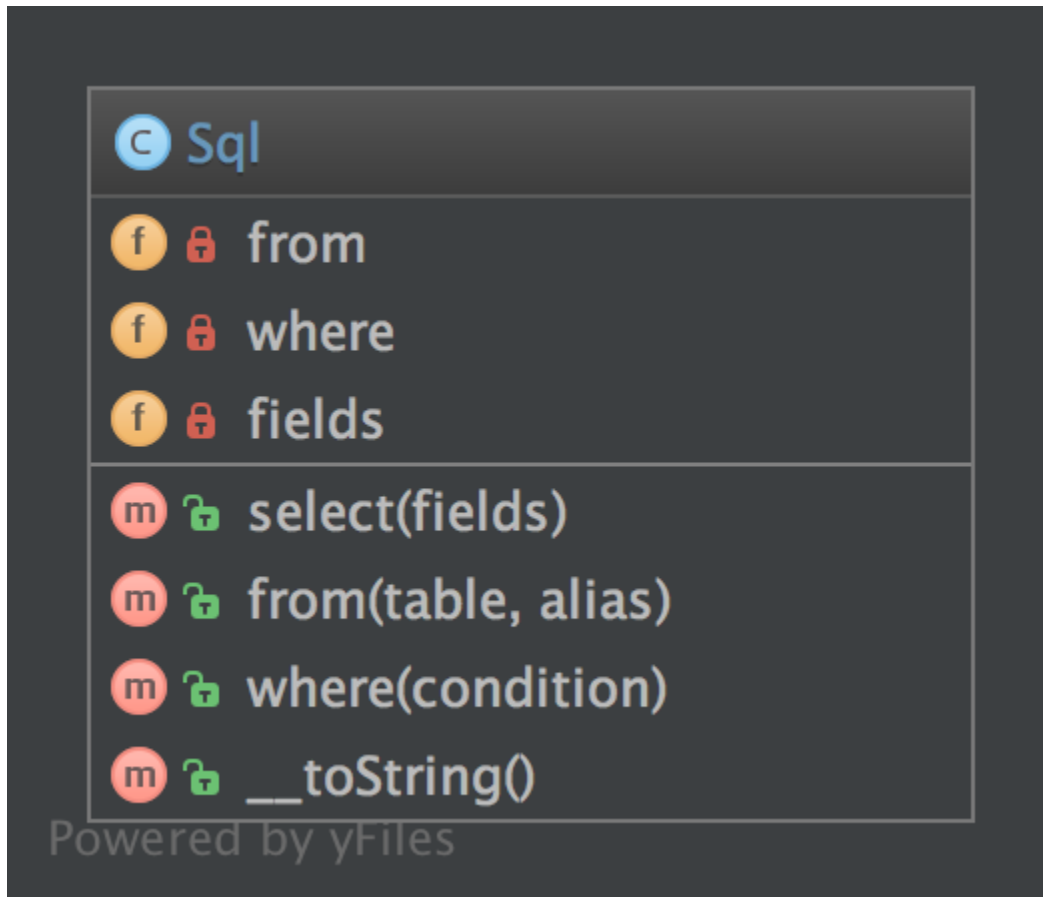
Purpose

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

Examples

- Doctrine2's QueryBuilder works something like that example class below
- PHPUnit uses fluent interfaces to build mock objects

UML Diagram



Code

You can also find this code on [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 = [];
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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()

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

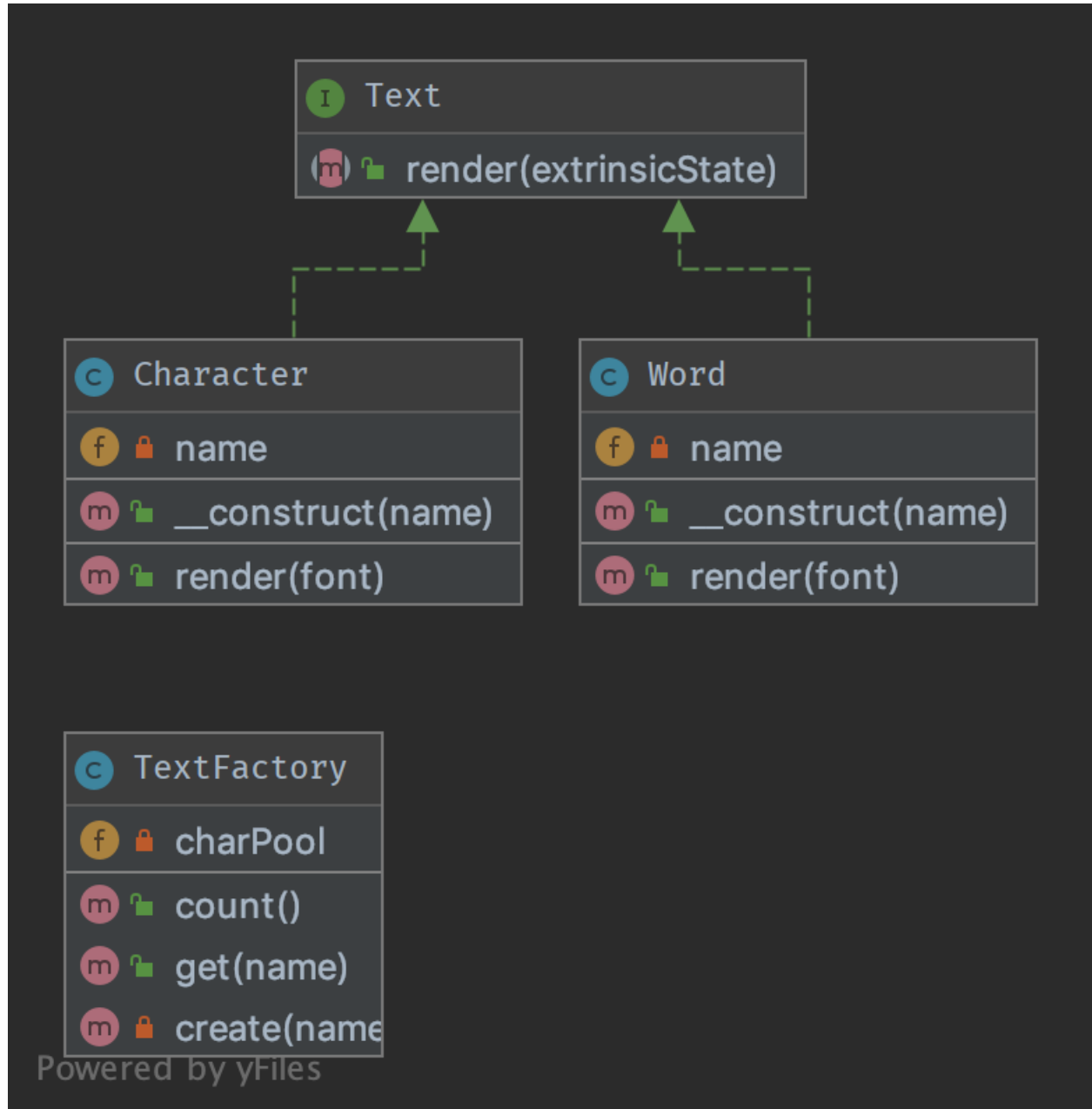
```
13 {
14     $query = (new Sql())
15         ->select(['foo', 'bar'])
16         ->from('foobar', 'f')
17         ->where('f.bar = ?');
18
19     $this->assertSame('SELECT foo, bar FROM foobar AS f WHERE f.bar = ?', (string)
14 ↪ $query);
20 }
21 }
```

1.2.9 Flyweight

Purpose

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Text.php

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

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

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      }

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

}

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 → $rendered);
29                 }
30             }
31
32             foreach ($this->fonts as $word) {
33                 $flyweight = $factory->get($word);
34                 $rendered = $flyweight->render('foobar');
35
36                 $this->assertSame(sprintf('Word %s with font foobar', $word), $rendered);
37             }
38
39             // Flyweight pattern ensures that instances are shared
40             // instead of having hundreds of thousands of individual objects
41             // there must be one instance for every char that has been reused for displaying
42 → in different fonts
43             $this->assertCount(count($this->characters) + count($this->fonts), $factory);
44         }
45     }
46 }

```

(continua a la pàgina següent)

44

```
}
```

1.2.10 Proxy

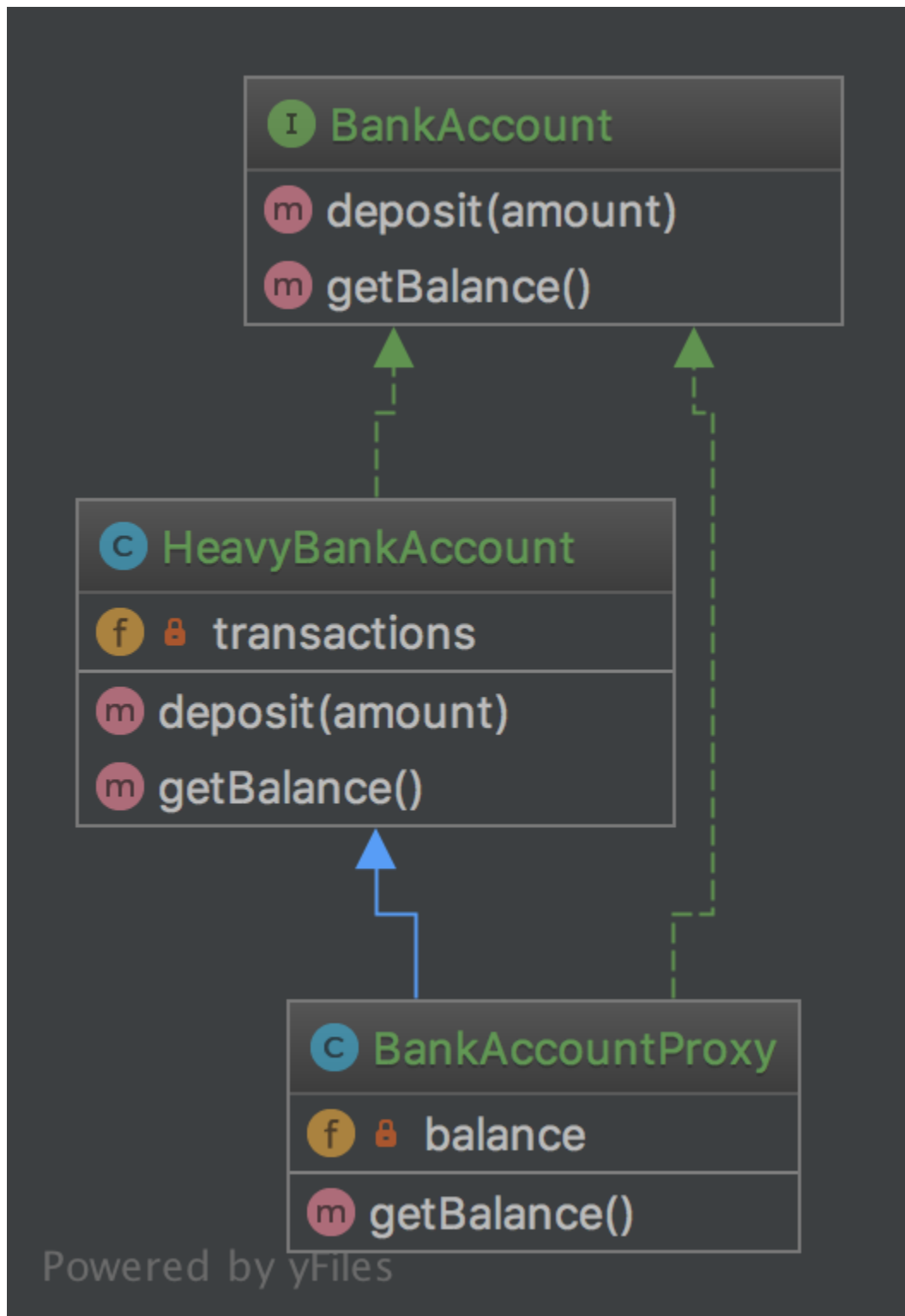
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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);
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

4 namespace DesignPatterns\Structural\Proxy;
5
6 class BankAccountProxy extends HeavyBankAccount implements BankAccount
7 {
8     private ?int $balance = null;
9
10    public function getBalance(): int
11    {
12        // because calculating balance is so expensive,
13        // the usage of BankAccount::getBalance() is delayed until it really is needed
14        // and will not be calculated again for this instance
15
16        if ($this->balance === null) {
17            $this->balance = parent::getBalance();
18        }
19
20        return $this->balance;
21    }
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());
27     }
28 }

```

(continua a la pàgina següent)

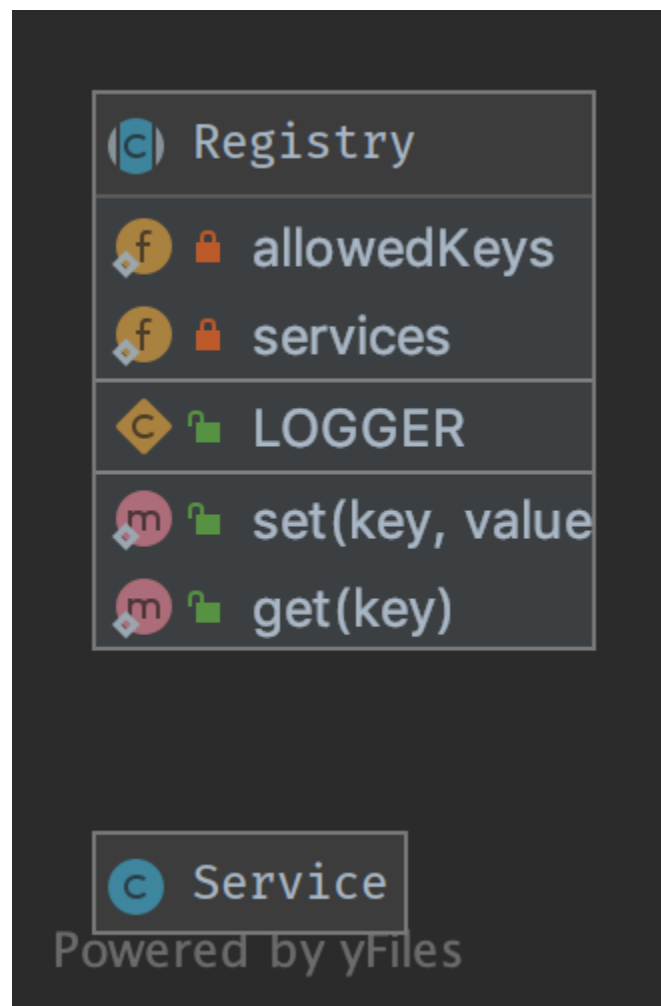
```
25     }  
26 }
```

1.2.11 Registry

Purpose

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

UML Diagram



Code

You can also find this code on [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 therefor 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
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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      */

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

1.3 Behavioral

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

1.3.1 Chain Of Responsibilities

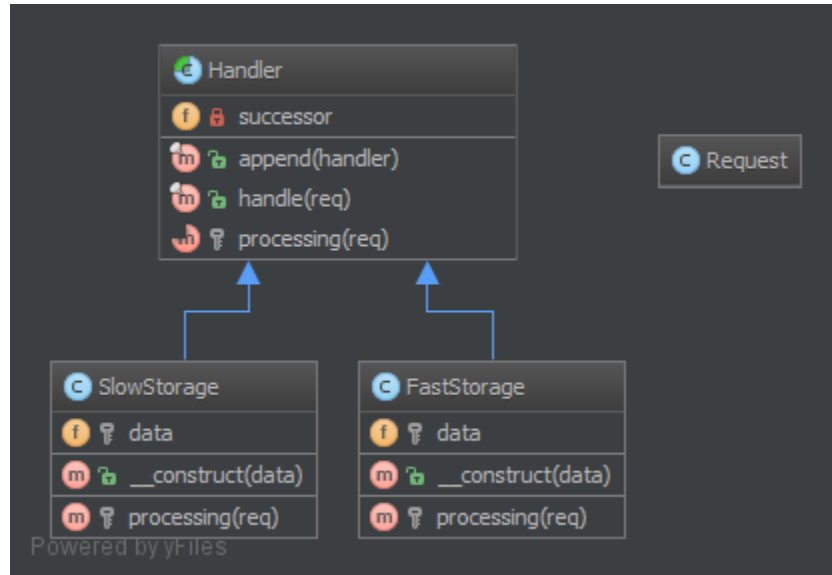
Purpose

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

Examples

- logging framework, where each chain element decides autonomously what to do with a log message
- a Spam filter
- Caching: first object is an instance of e.g. a Memcached Interface, if that «misses» it delegates the call to the database interface

UML Diagram



Code

You can also find this code on [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 }

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     }
19 }

```

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);

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

To encapsulate invocation and decoupling.

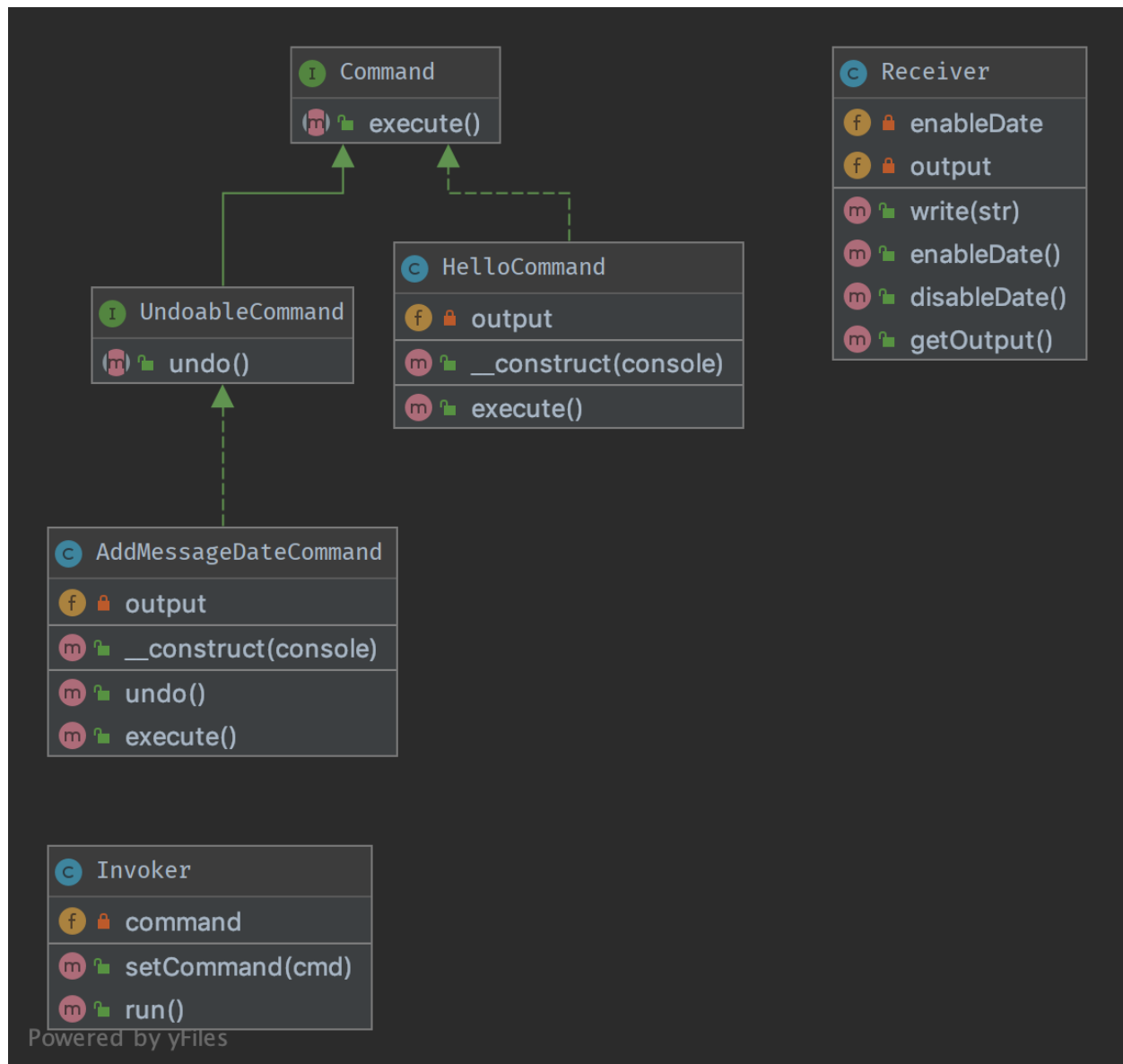
We have an Invoker and a Receiver. This pattern uses a «Command» to delegate the method call against the Receiver and presents the same method «execute». Therefore, the Invoker just knows to call «execute» to process the Command of the client. The Receiver is decoupled from the Invoker.

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

Examples

- A text editor : all events are commands which can be undone, stacked and saved.
- big CLI tools use subcommands to distribute various tasks and pack them in «modules», each of these can be implemented with the Command pattern (e.g. vagrant)

UML Diagram



Code

You can also find this code on [GitHub](#)

Command.php

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

26         // sometimes, there is no receiver and this is the command which does all the
↪work
27         $this->output->write('Hello World');
28     }
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
↪commands in the parameters.
16       */
17      public function __construct(private Receiver $output)
18      {
19      }
20
21      /**
22       * Execute and make receiver to enable displaying messages date.
23       */
24      public function execute()
25      {
26          // sometimes, there is no receiver and this is the command which
27          // does all the work
28          $this->output->enableDate();
29      }
30
31      /**
32       * Undo the command and make receiver to disable displaying messages date.
33       */
34      public function undo()
35      {
36          // sometimes, there is no receiver and this is the command which
37          // does all the work
38          $this->output->disableDate();
39      }
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     }
48 }

```

Invoker.php

```

1 <?php
2

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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();

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

Tests/UndoableCommandTest.php

```

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

```

1.3.3 Interpreter

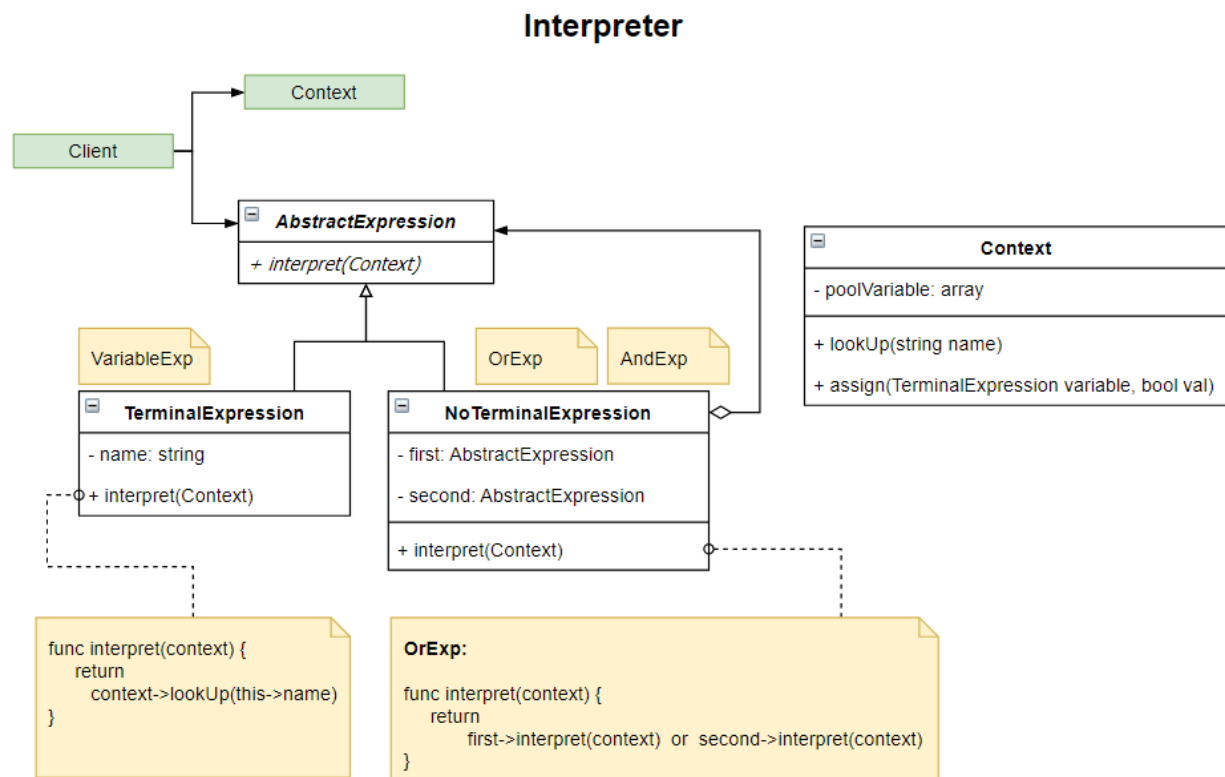
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

AbstractExp.php

```

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

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

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

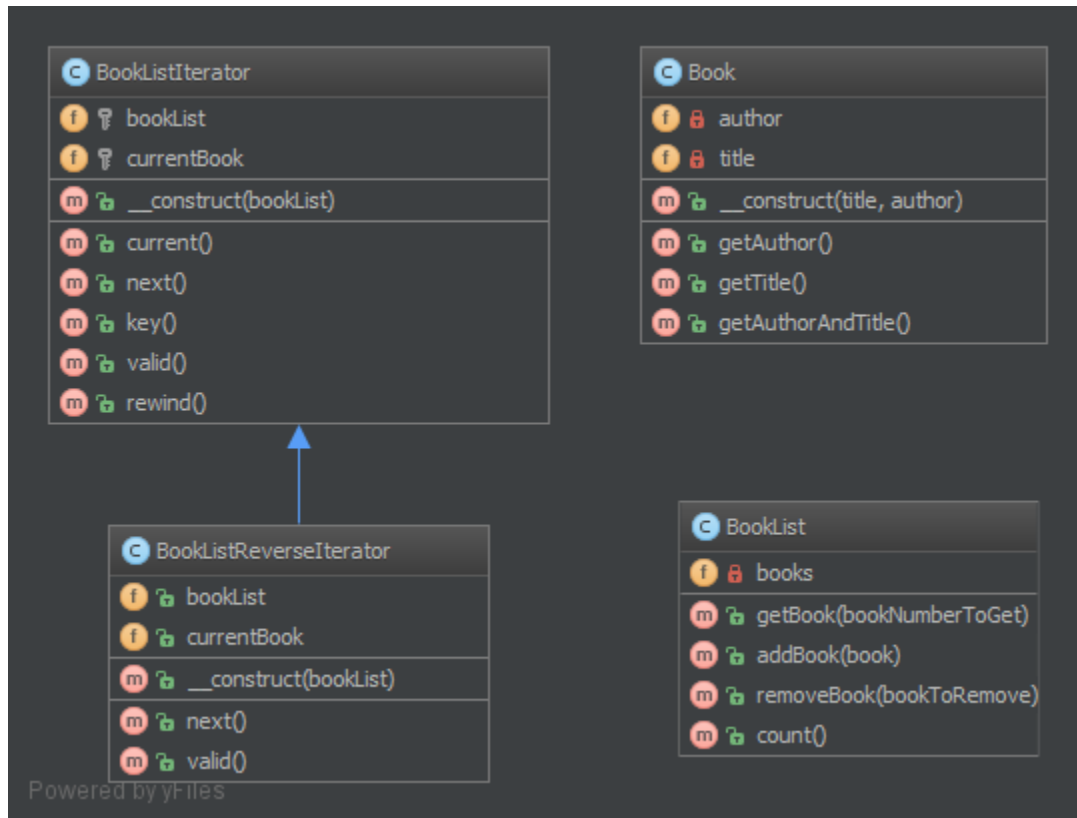
Examples

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

Note

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Book.php

```

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
  
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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();

```

(continua a la pàgina següent)

```
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);
```

(continua a la pàgina següent)

[\(ve de la pàgina anterior\)](#)

```
76     }  
77 }
```

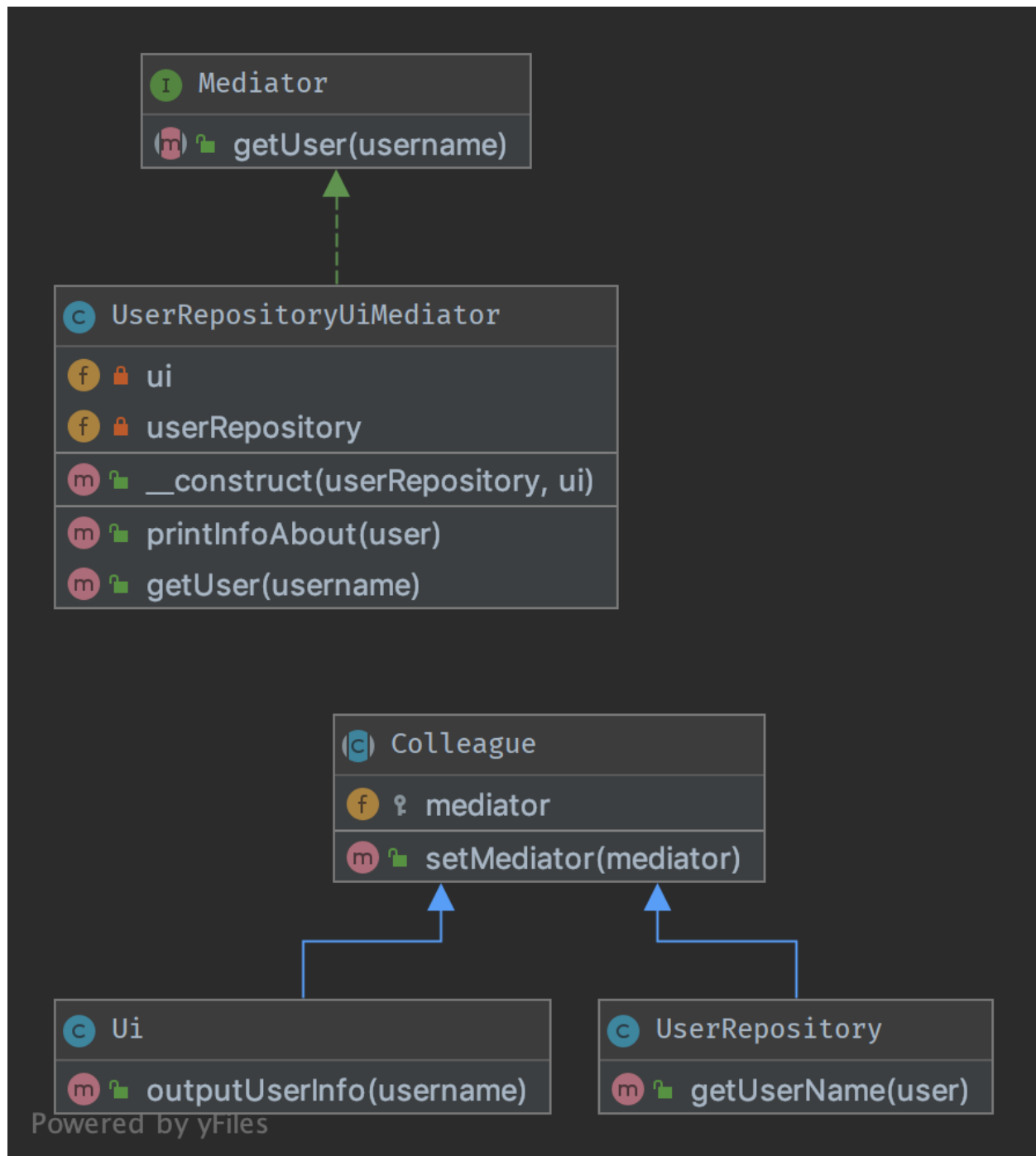
1.3.5 Mediator

Purpose

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

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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
2

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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;

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

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).

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

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

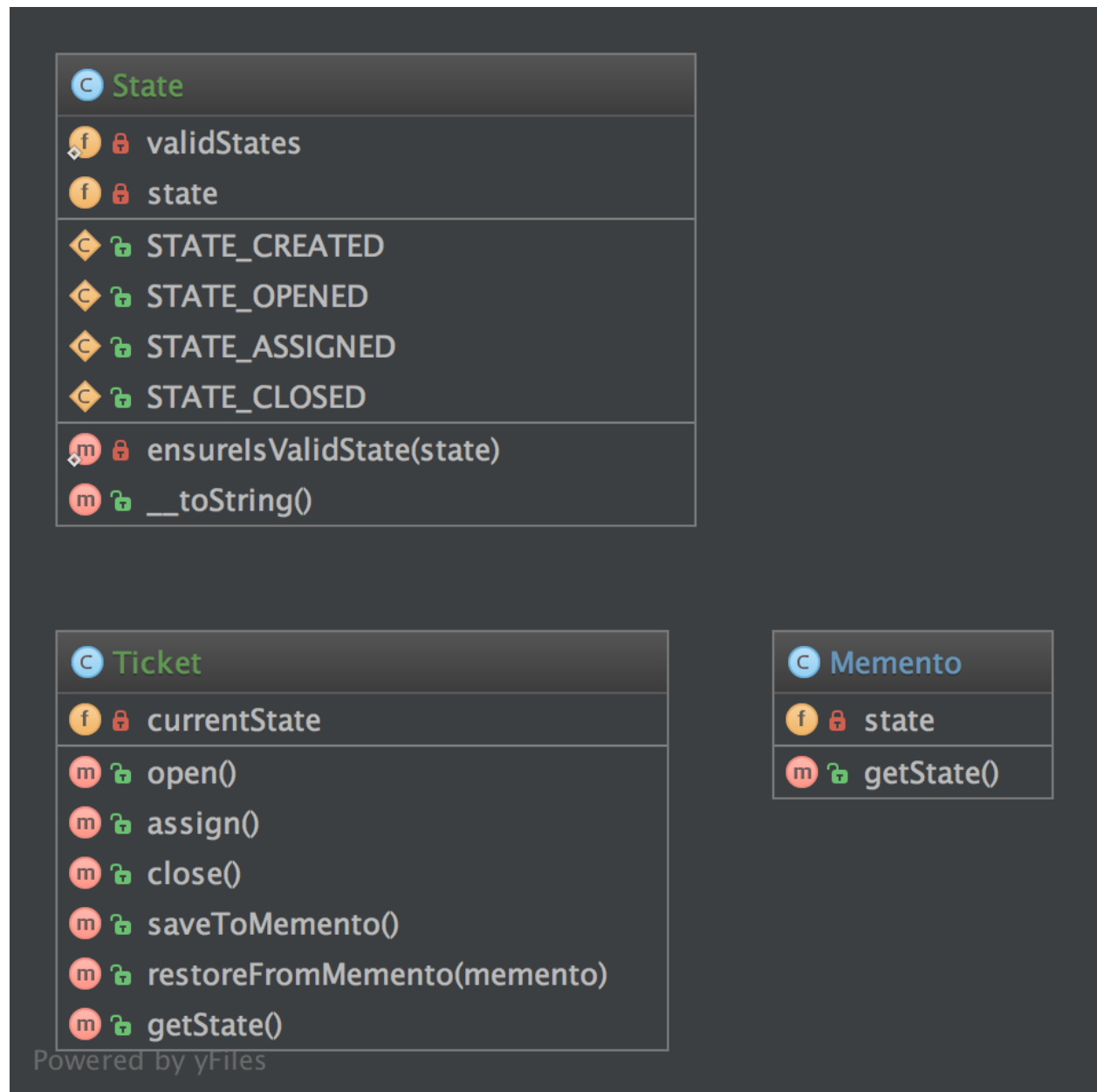
Originator – it is an object that contains the *actual state of an external object is strictly specified type*. Originator is able to create a unique copy of this state and return it wrapped in a Memento. The Originator does not know the history of changes. You can set a concrete state to Originator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

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Memento.php

```

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

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

6
7 class Memento
8 {
9     public function __construct(private State $state)
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)) {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

Purpose

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

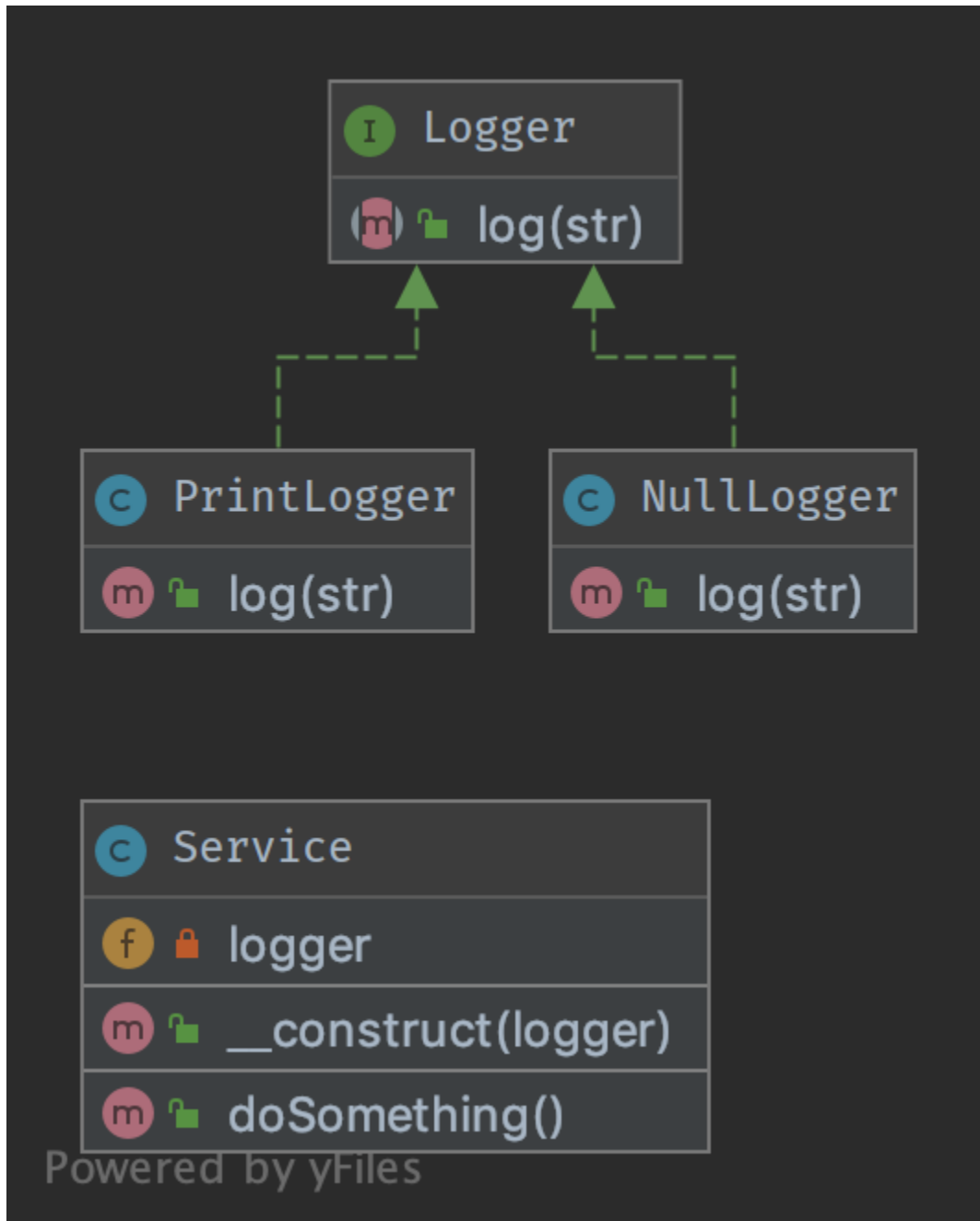
- Client code is simplified
- Reduces the chance of null pointer exceptions
- Fewer conditionals require less test cases

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

Examples

- Null logger or null output to preserve a standard way of interaction between objects, even if the shouldn't do anything
- null handler in a Chain of Responsibilities pattern
- null command in a Command pattern

UML Diagram



Code

You can also find this code on [GitHub](#)

Service.php

```
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     }
22 }
```

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
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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\

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

1.3.8 Observer

Purpose

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

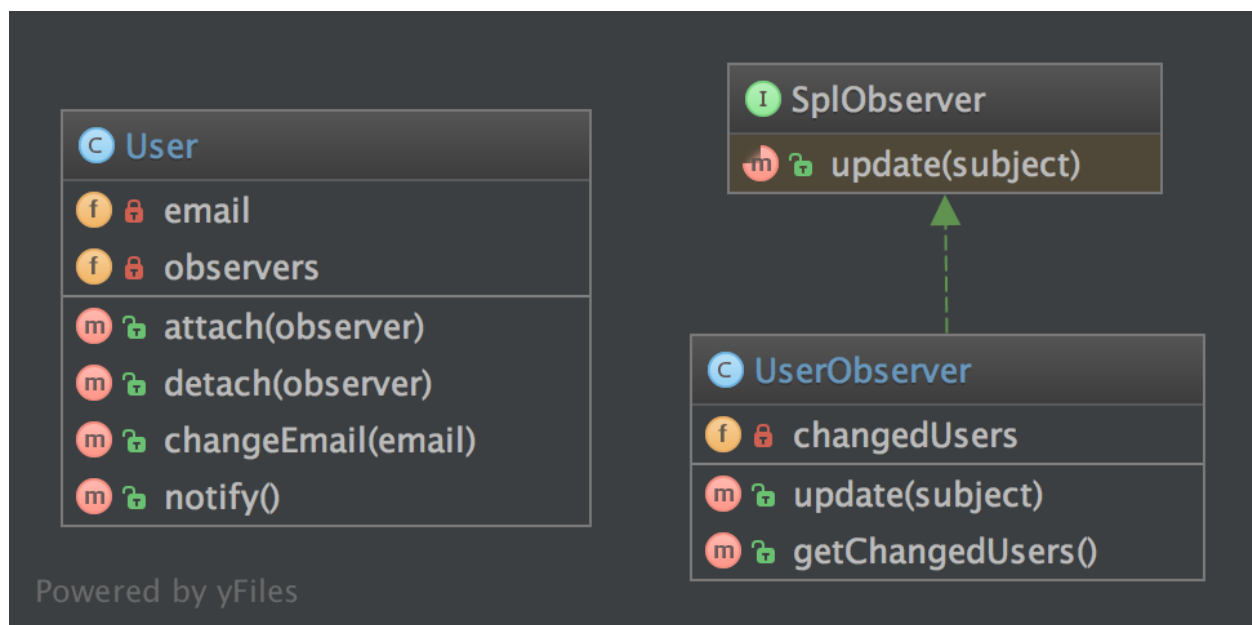
Examples

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

Note

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

UML Diagram



Code

You can also find this code on [GitHub](#)

User.php

```

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
13   * ↪ observers and sends notifications to
14   * them in case changes are made on the User object
15   */
16  class User implements SplSubject
17  {
18      private SplObjectStorage $observers;
19      private $email;
20
21      public function __construct()
22      {
23          $this->observers = new SplObjectStorage();
24      }
25
26      public function attach(SplObserver $observer): void
27      {
28          $this->observers->attach($observer);
29      }
30
31      public function detach(SplObserver $observer): void
32      {
33          $this->observers->detach($observer);
34      }
35
36      public function changeEmail(string $email): void
37      {
38          $this->email = $email;
39          $this->notify();
40      }
41
42      public function notify(): void
43      {
44          /** @var SplObserver $observer */
45          foreach ($this->observers as $observer) {
46              $observer->update($this);
47          }
48      }
49  }

```

(continua a la pàgina següent)

}

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;
10
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

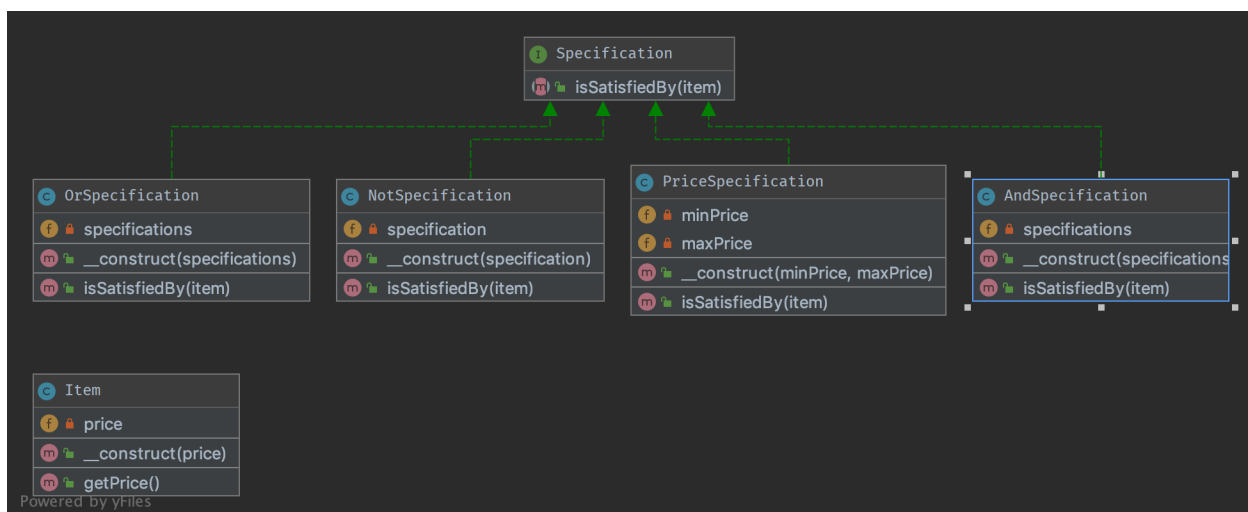
Purpose

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

Examples

- [RulerZ](#)

UML Diagram



Code

You can also find this code on [GitHub](#)

Item.php

```
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
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

2
3 declare(strict_types=1);
4
5 namespace DesignPatterns\Behavioral\Specification;
6
7 class AndSpecification implements Specification
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 }

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

16     }
17 }

```

Test

Tests/SpecificationTest.php

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace DesignPatterns\Behavioral\Specification\Tests;
6
7  use DesignPatterns\Behavioral\Specification\Item;
8  use DesignPatterns\Behavioral\Specification\NotSpecification;
9  use DesignPatterns\Behavioral\Specification\OrSpecification;
10 use DesignPatterns\Behavioral\Specification\AndSpecification;
11 use DesignPatterns\Behavioral\Specification\PriceSpecification;
12 use PHPUnit\Framework\TestCase;
13
14 class SpecificationTest extends TestCase
15 {
16     public function testCanOr()
17     {
18         $spec1 = new PriceSpecification(50, 99);
19         $spec2 = new PriceSpecification(101, 200);
20
21         $orSpec = new OrSpecification($spec1, $spec2);
22
23         $this->assertFalse($orSpec->isSatisfiedBy(new Item(100)));
24         $this->assertTrue($orSpec->isSatisfiedBy(new Item(51)));
25         $this->assertTrue($orSpec->isSatisfiedBy(new Item(150)));
26     }
27
28     public function testCanAnd()
29     {
30         $spec1 = new PriceSpecification(50, 100);
31         $spec2 = new PriceSpecification(80, 200);
32
33         $andSpec = new AndSpecification($spec1, $spec2);
34
35         $this->assertFalse($andSpec->isSatisfiedBy(new Item(150)));
36         $this->assertFalse($andSpec->isSatisfiedBy(new Item(1)));
37         $this->assertFalse($andSpec->isSatisfiedBy(new Item(51)));
38         $this->assertTrue($andSpec->isSatisfiedBy(new Item(100)));
39     }
40
41     public function testCanNot()
42     {
43         $spec1 = new PriceSpecification(50, 100);
44         $notSpec = new NotSpecification($spec1);

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

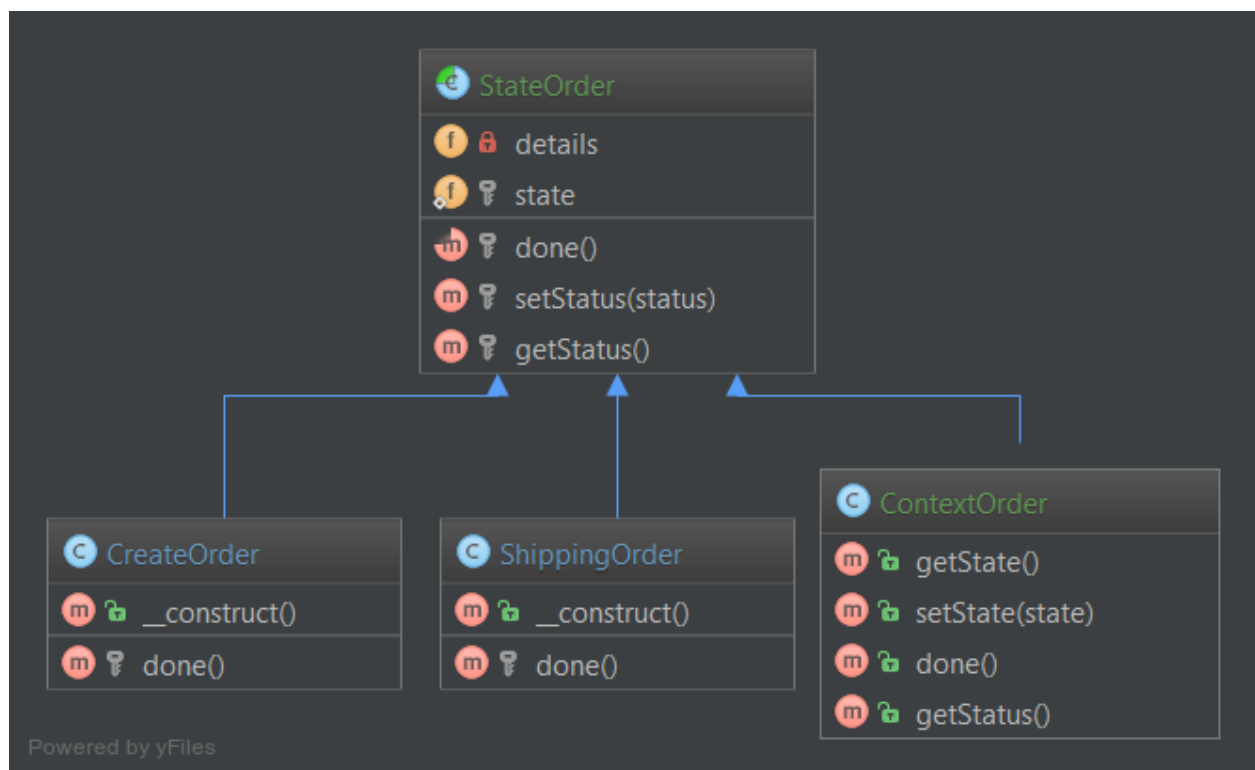
```
45  
46     $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));  
47     $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));  
48 }  
49 }
```

1.3.10 State

Purpose

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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    {
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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();

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

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

1.3.11 Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy

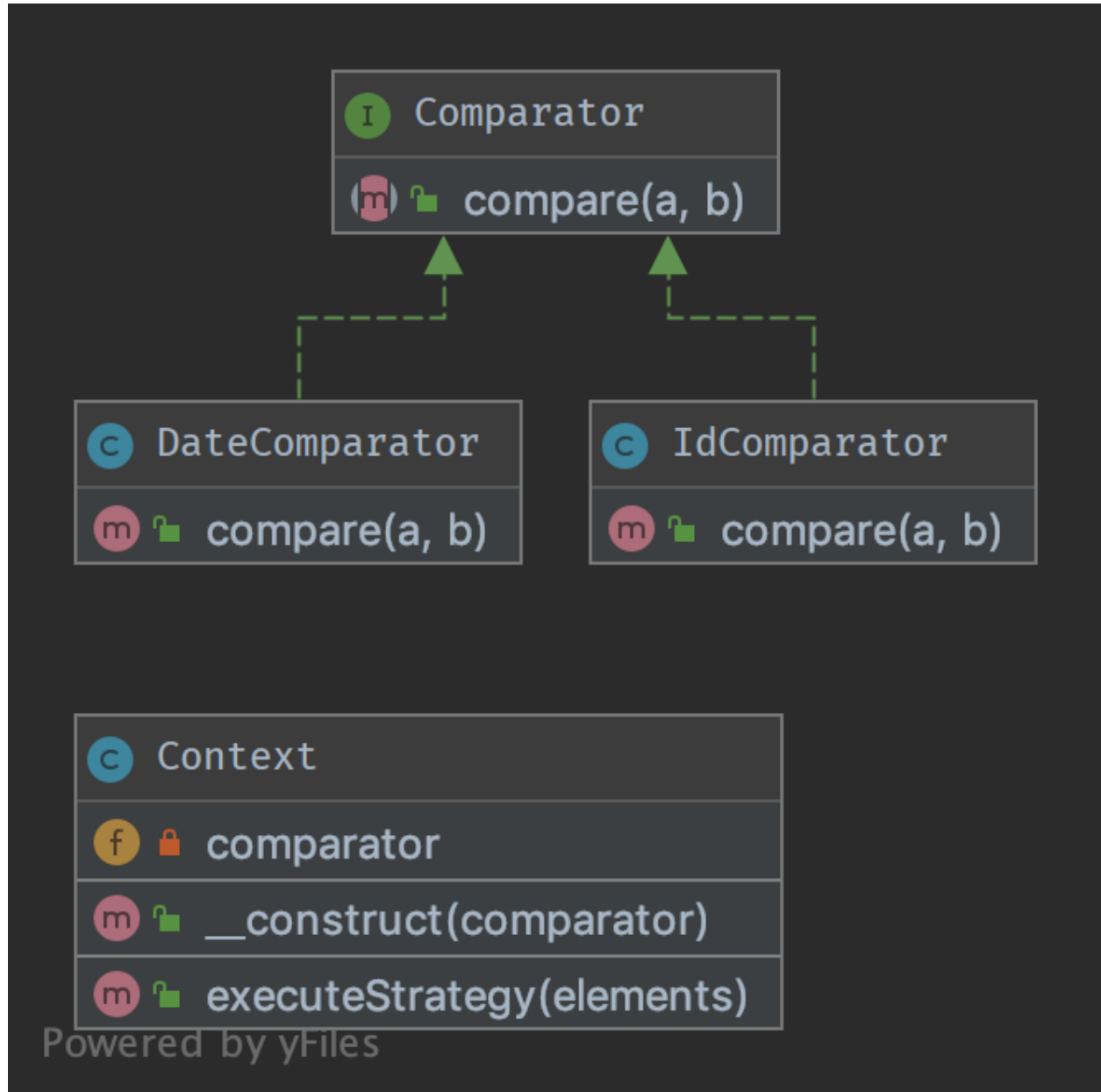
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Context.php

```

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

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

Comparator.php

```

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

```

DateComparator.php

```

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

```

IdComparator.php

```

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\DataComparator;
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-

```

(continua a la pàgina següent)

```

33     ↪01']],
34         ['date' => '2013-03-01'],
35     ],
36     [
37         ↪02']],
38         ['date' => '2013-02-01'],
39     ],
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 /**
58  * @dataProvider provideDates
59  *
60  * @param array $collection
61  * @param array $expected
62  */
63 public function testDateComparator($collection, $expected)
64 {
65     $obj = new Context(new DateComparator());
66     $elements = $obj->executeStrategy($collection);
67
68     $firstElement = array_shift($elements);
69     $this->assertSame($expected, $firstElement);
70 }
71 }

```


1.3.12 Template Method

Purpose

Template Method is a behavioral design pattern.

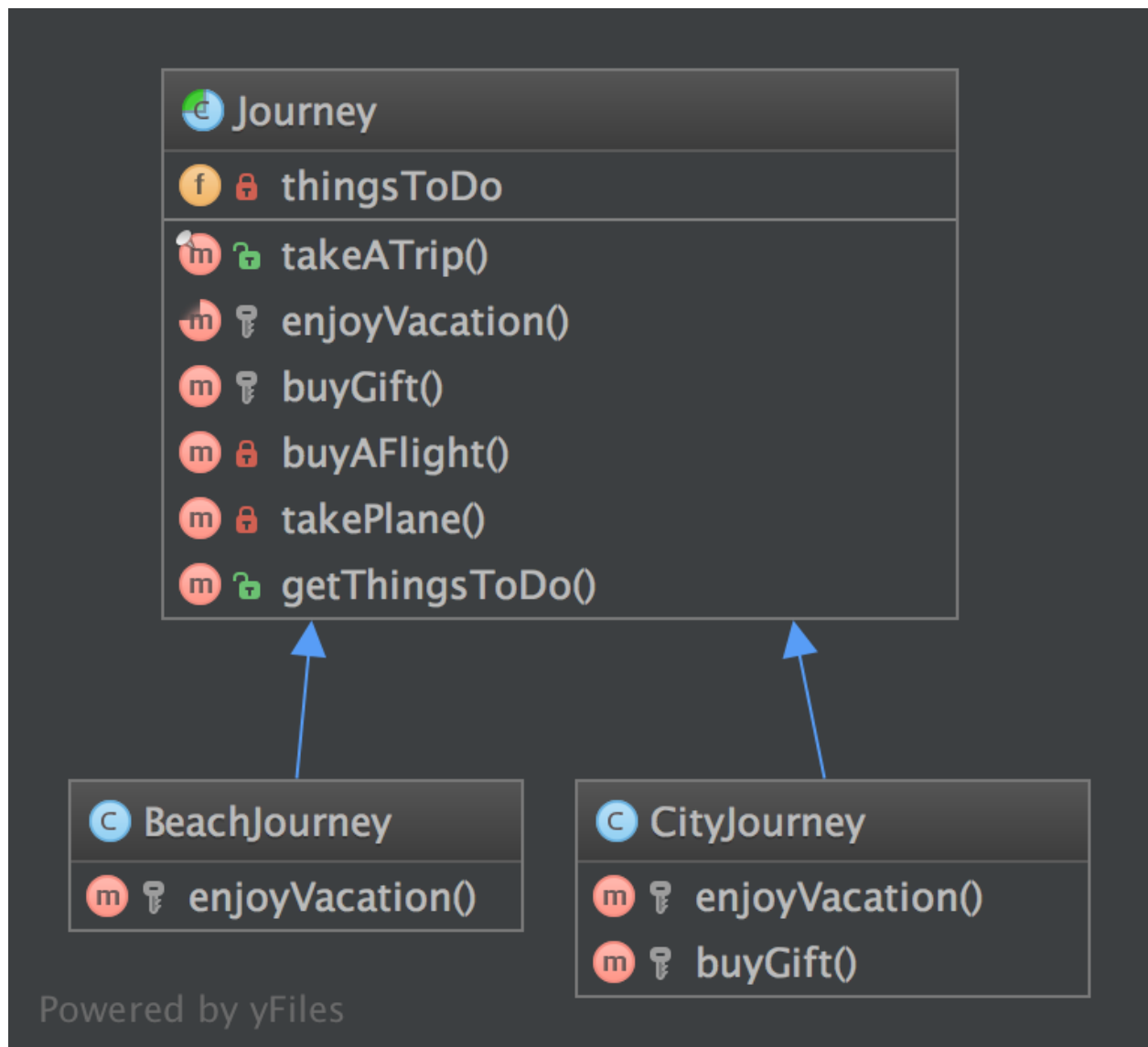
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template «finish» the behavior of an algorithm.

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

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

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Journey.php

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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  */
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 }

```

(continua a la pàgina següent)

```
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     }  
41 }
```

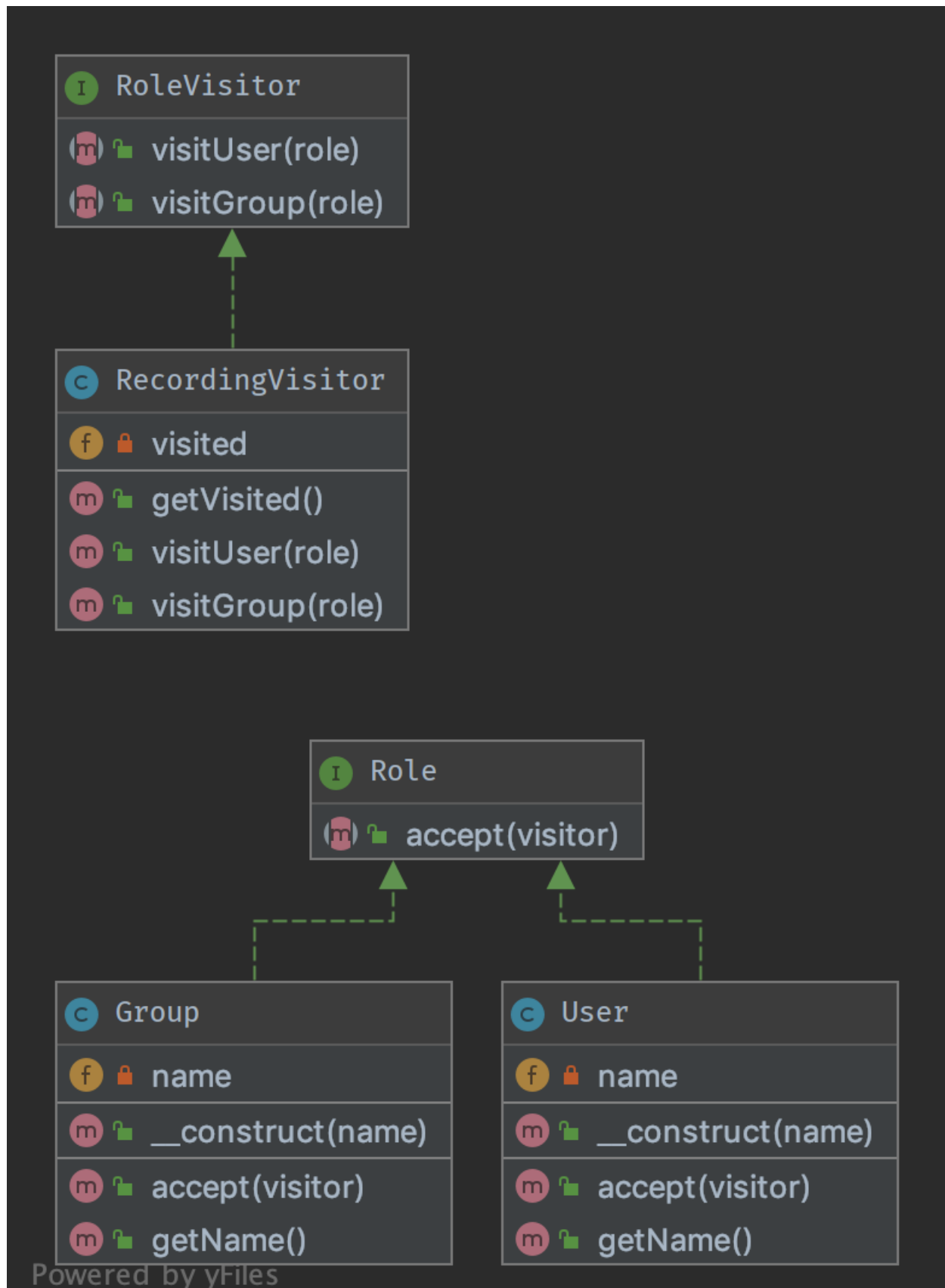
1.3.13 Visitor

Purpose

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

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

UML Diagram



Code

You can also find this code on [GitHub](#)

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;
30      }
31  }
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```
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     {
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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)

```

(continua a la pàgina següent)

[\(ve de la pàgina anterior\)](#)

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

1.4 More

1.4.1 Service Locator

THIS IS CONSIDERED TO BE AN ANTI-PATTERN!

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

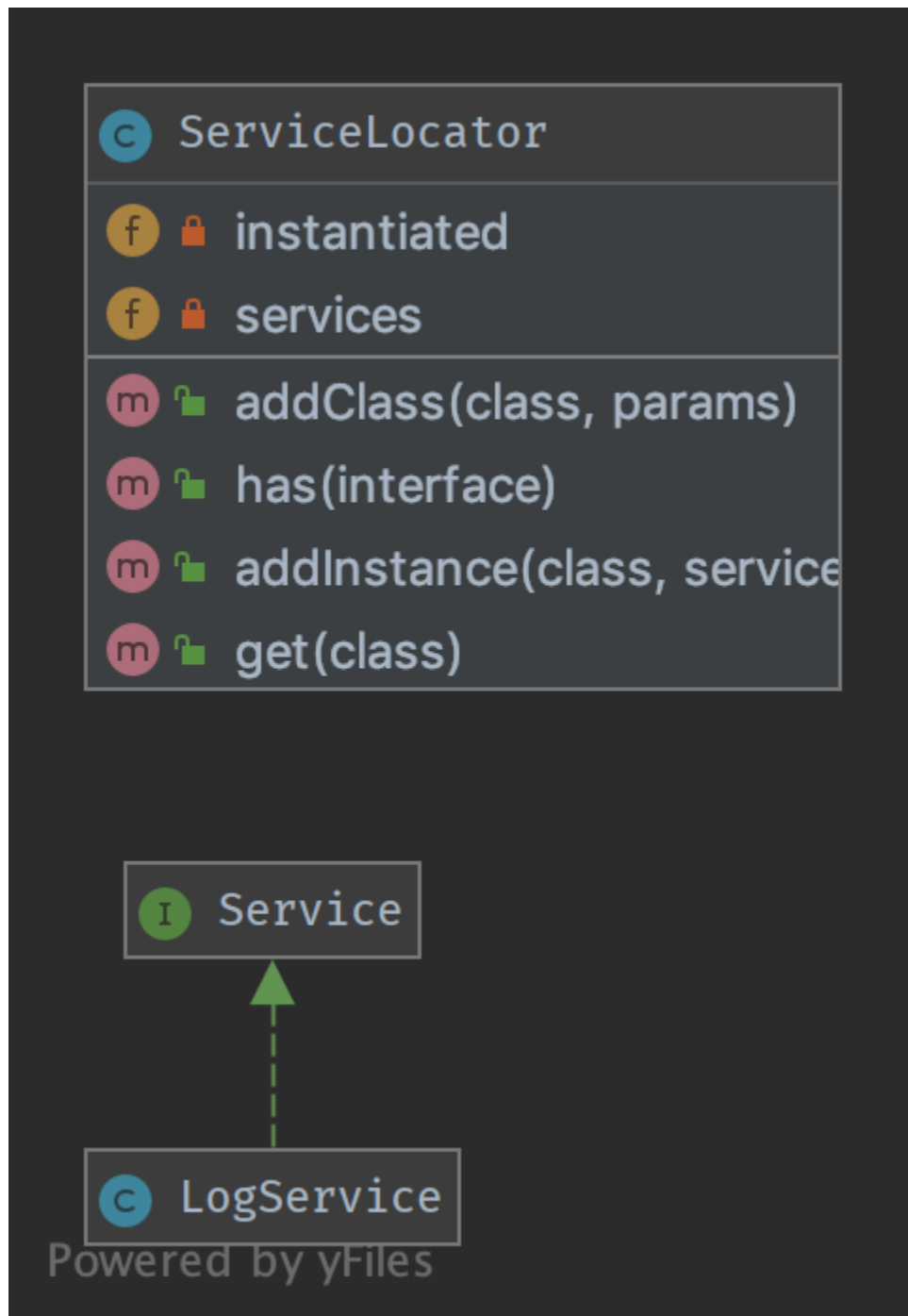
Purpose

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

Usage

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

UML Diagram



Code

You can also find this code on [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
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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();

```

(continua a la pàgina següent)

```
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

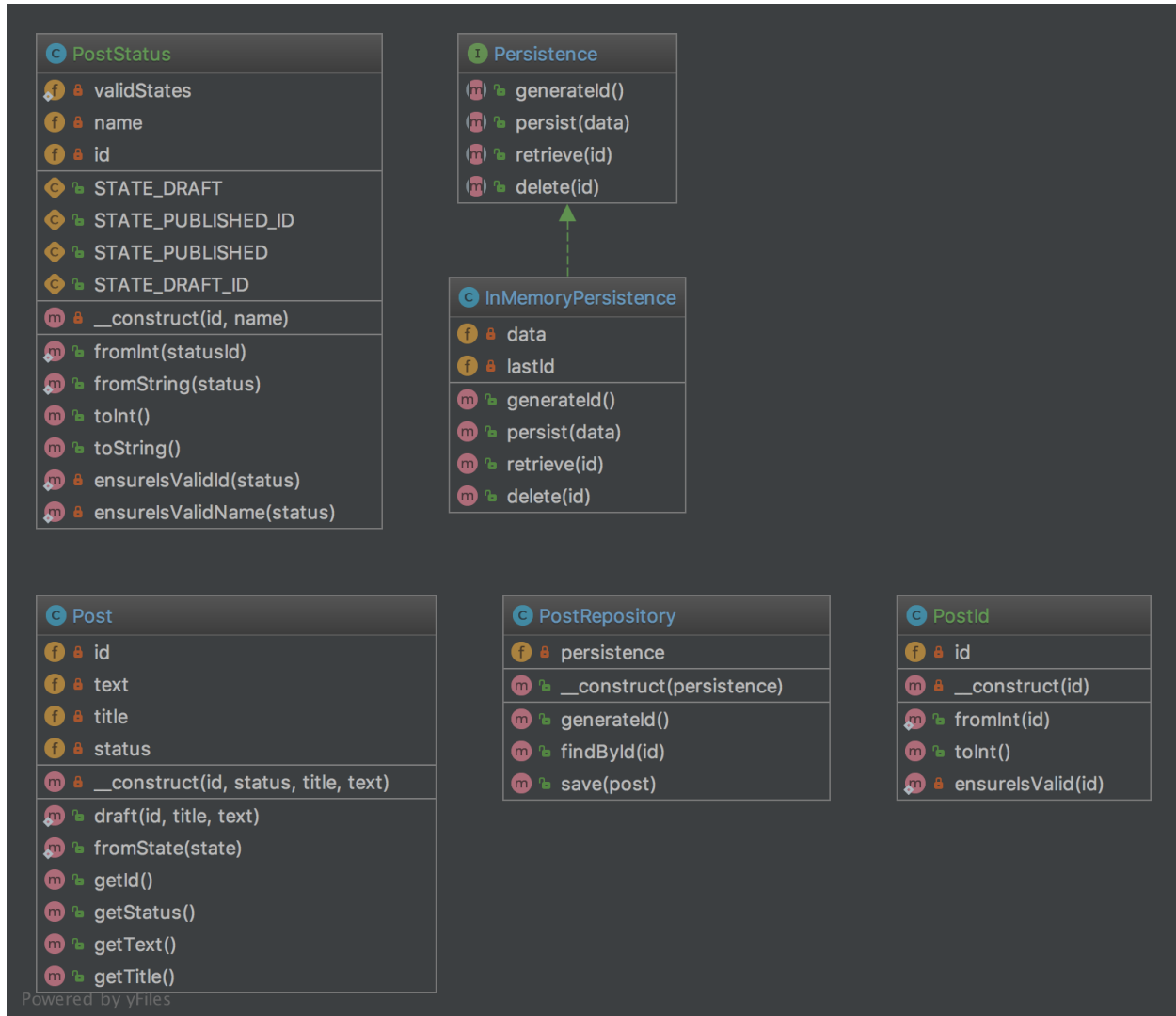
Purpose

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

Examples

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

UML Diagram



Code

You can also find this code on [GitHub](#)

Post.php

```

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     {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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);

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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 it's 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     private function __construct(private int $id)
29     {
30     }
31
32     public function toInt(): int
33     {
34         return $this->id;
35     }
36
37     private static function ensureIsValid(int $id)
38     {
39         if ($id <= 0) {
40             throw new InvalidArgumentException('Invalid PostId given');
41         }
42     }
43 }

```

PostStatus.php

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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  class PostRepository
24  {
25      public function __construct(private Persistence $persistence)
26      {
27
28
29      public function generateId(): PostId
30      {

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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;

```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

6
7 use OutOfBoundsException;
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;

```

(continua a la pàgina següent)

```
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 }
```

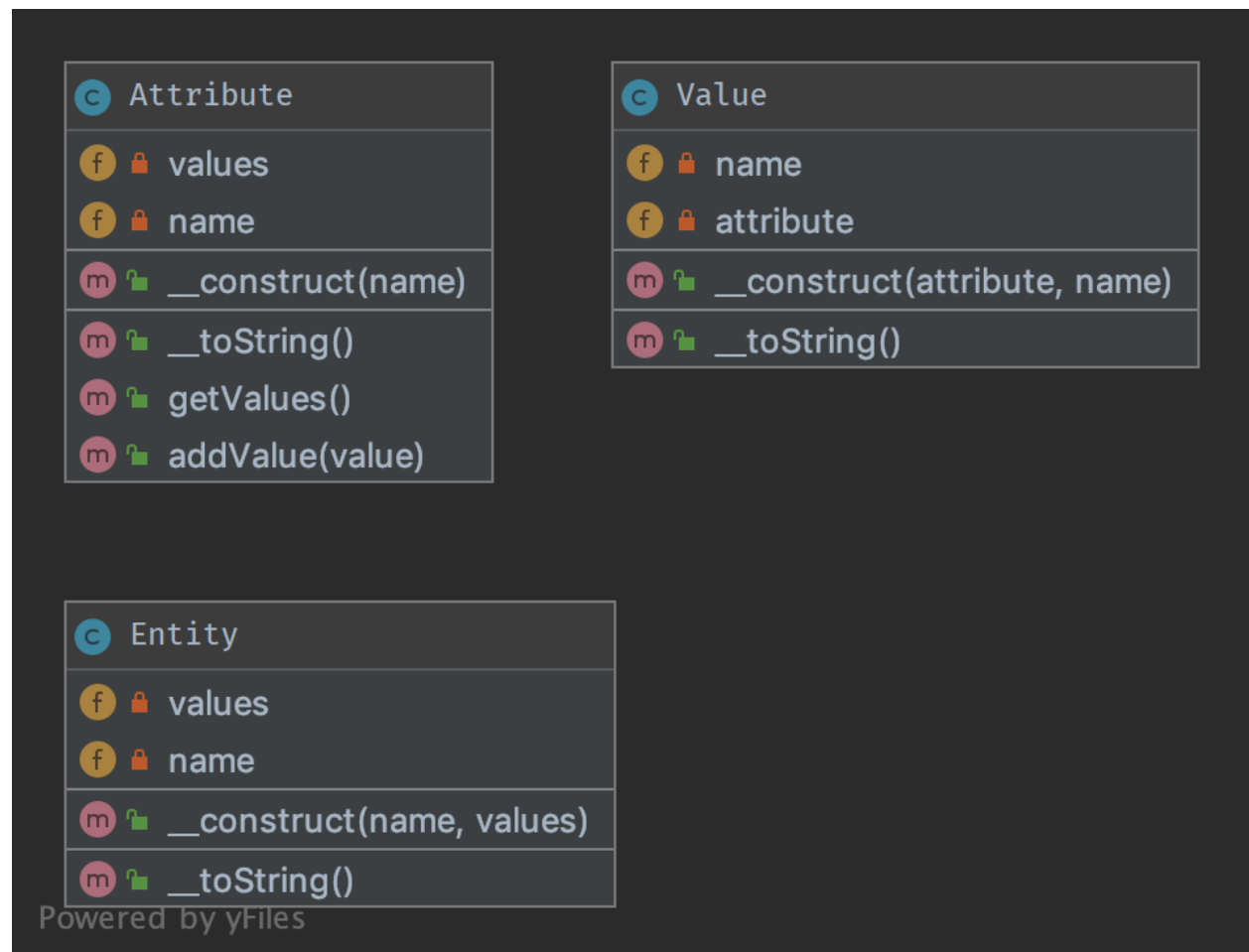
1.4.3 Entity-Attribute-Value (EAV)

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

Purpose

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

UML Diagram



Code

You can also find this code on [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;
```

(continua a la pàgina següent)

(ve de la pàgina anterior)

```

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