DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

Jul 12, 2020

Contents

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 . . . 21
- 1.1.6 Simple Factory . . . 24
- 1.1.7 Singleton . . . 26
- 1.1.8 Static Factory . . . 28

1.2 Structural . . . 31
- 1.2.1 Adapter / Wrapper . . . 31
- 1.2.2 Bridge . . . 35
- 1.2.3 Composite . . . 39
- 1.2.4 Data Mapper . . . 42
- 1.2.5 Decorator . . . 46
- 1.2.6 Dependency Injection . . . 50
- 1.2.7 Facade . . . 53
- 1.2.8 Fluent Interface . . . 56
- 1.2.9 Flyweight . . . 59
- 1.2.10 Proxy . . . 62
- 1.2.11 Registry . . . 66

1.3 Behavioral . . . 70
- 1.3.1 Chain Of Responsibilities . . . 70
- 1.3.2 Command . . . 73
- 1.3.3 Iterator . . . 79
- 1.3.4 Mediator . . . 83
- 1.3.5 Memento . . . 87
- 1.3.6 Null Object . . . 91
- 1.3.7 Observer . . . 95
- 1.3.8 Specification . . . 98
- 1.3.9 State . . . 103
- 1.3.10 Strategy . . . 106
- 1.3.11 Template Method . . . 110
- 1.3.12 Visitor . . . 114

1.4 More . . . 118

1.4.1 Service Locator 118
1.4.2 Repository 123
1.4.3 Entity-Attribute-Value (EAV) 131

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.

CHAPTER 1

Patterns

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

1.1 Creational

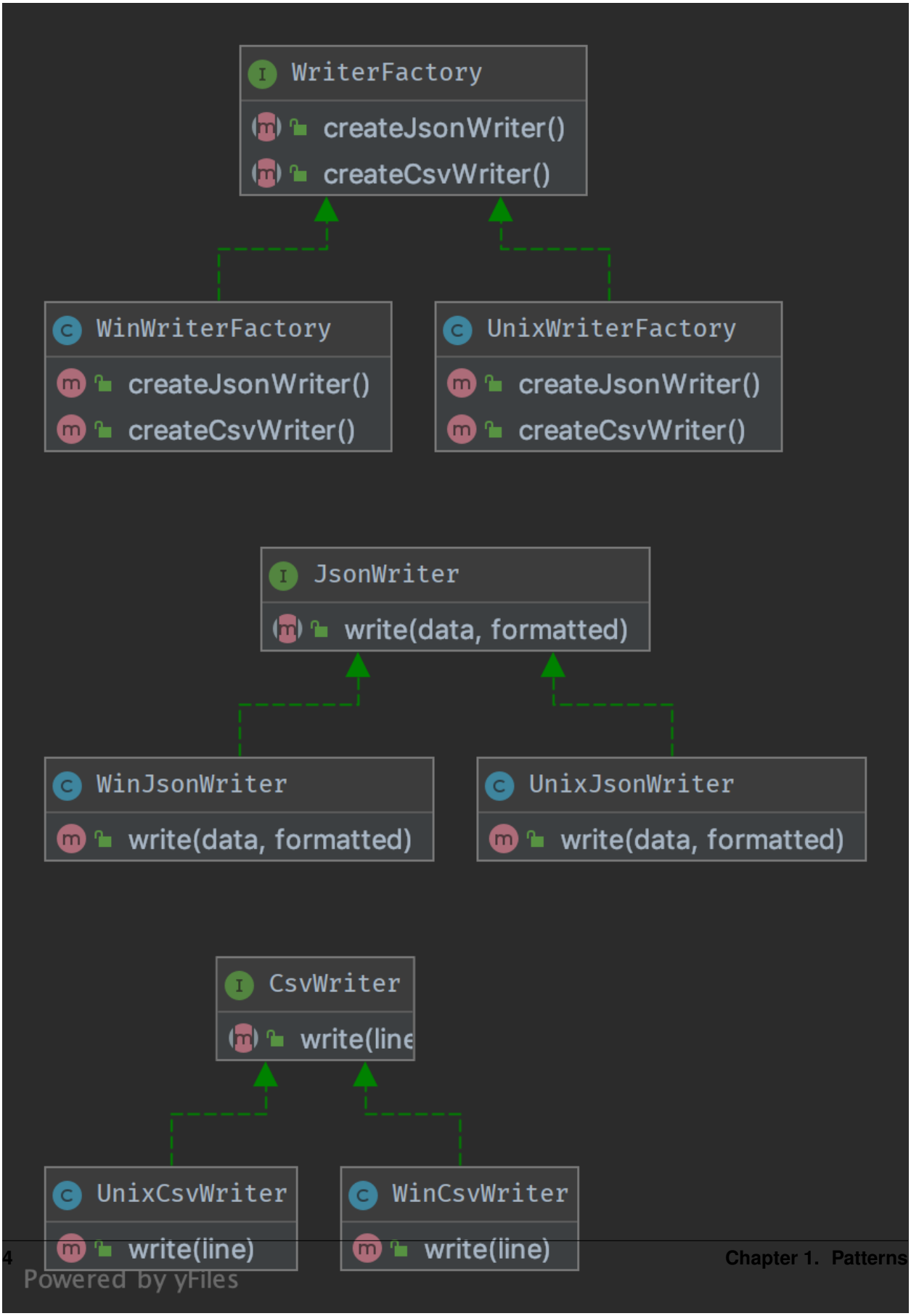
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns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


WriterFactory
createJsonWriter()
createCsvWriter()
WinWriterFactory
createJsonWriter()
createCsvWriter()
UnixWriterFactory
createJsonWriter()
createCsvWriter()
JsonWriter
write(data, formatted)
WinJsonWriter
write(data, formatted)
UnixJsonWriter
write(data, formatted)
CsvWriter
write(line
UnixCsvWriter
write(line)
WinCsvWriter
write(line)
Powered by yFiles

Code

You can also find this code on GitHub

WriterFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

CsvWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

JsonWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

UnixCsvWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

UnixJsonWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

```

(continues on next page)

(continued from previous page)

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

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

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

UnixWriterFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

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

WinCsvWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

WinJsonWriter.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

(continues on next page)

(continued from previous page)

```
9
10
11          return json_encode($data, JSON_PRETTY_PRINT);
12      }
13  }
```

WinWriterFactory.php

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

Test

Tests/AbstractFactoryTest.php

```
1   <?php declare(strict_types=1);
2
3   namespace DesignPatterns\Creational\AbstractFactory\Tests;
4
5   use DesignPatterns\Creational\AbstractFactory\CsvWriter;
6   use DesignPatterns\Creational\AbstractFactory\JsonWriter;
7   use DesignPatterns\Creational\AbstractFactory\UnixWriterFactory;
8   use DesignPatterns\Creational\AbstractFactory\WinWriterFactory;
9   use DesignPatterns\Creational\AbstractFactory\WriterFactory;
10  use PHPUnit\Framework\TestCase;
11
12  class AbstractFactoryTest extends TestCase
13  {
14      public function provideFactory()
15      {
16          return [
17              [new UnixWriterFactory()],
18              [new WinWriterFactory()]
19          ];
20      }
21
22      /**
23       * @dataProvider provideFactory
24       *
25       * @param WriterFactory $writerFactory
26       */
```

(continues on next page)

(continued from previous page)

```
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).

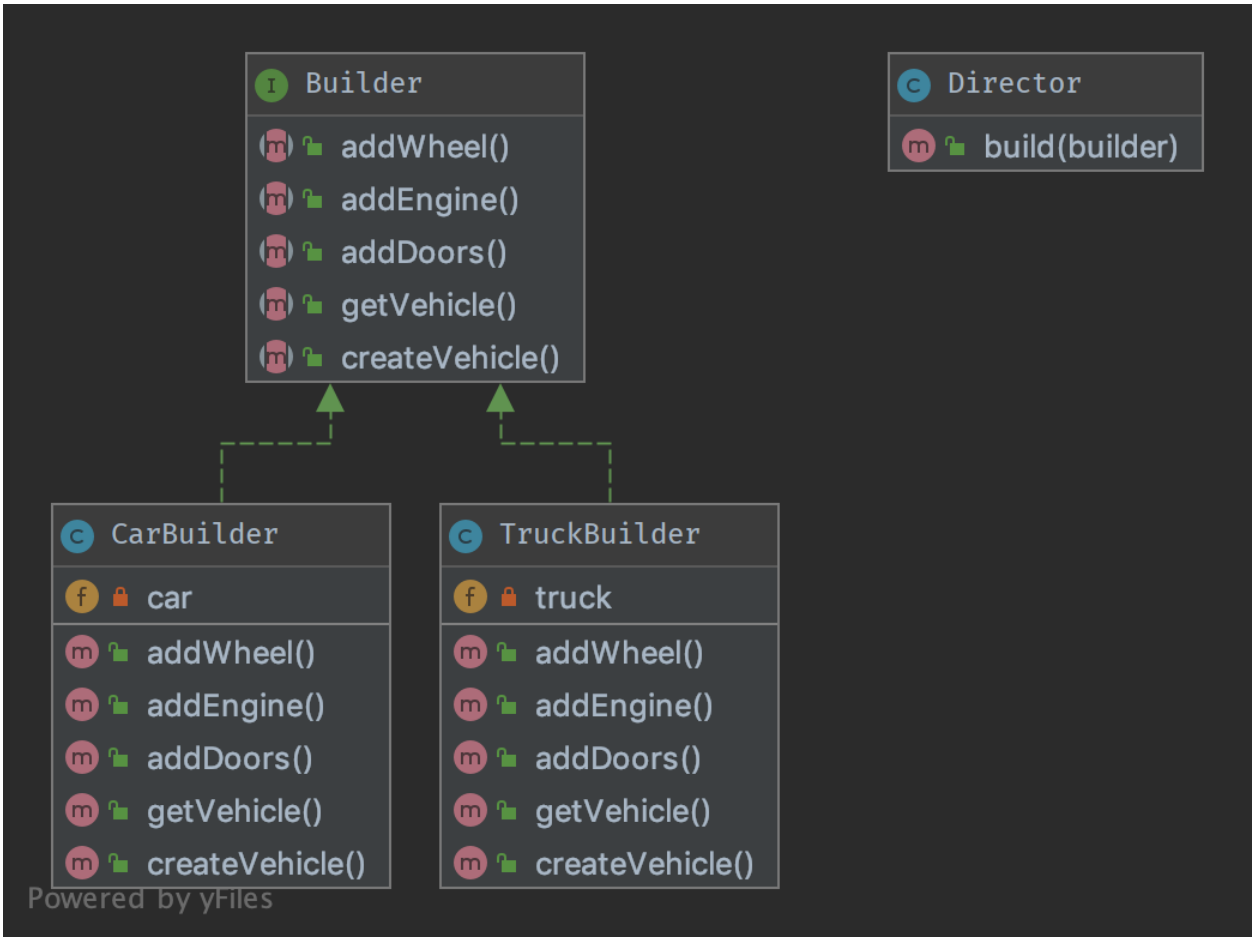
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

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

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

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

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

(continues on next page)

(continued from previous page)

```
        $builder->createVehicle();
        $builder->addDoors();
        $builder->addEngine();
        $builder->addWheel();

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

Builder.php

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

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

interface Builder
{
    public function createVehicle();

    public function addWheel();

    public function addEngine();

    public function addDoors();

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

TruckBuilder.php

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

namespace DesignPatterns\Creational\Builder;

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

class TruckBuilder implements Builder
{
    private Truck $truck;

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

CarBuilder.php

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

namespace DesignPatterns\Creational\Builder;

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

class CarBuilder implements Builder
{
    private Car $car;

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

Parts/Vehicle.php

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

namespace DesignPatterns\Creational\Builder\Parts;

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

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

Parts/Truck.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Truck extends Vehicle
{
}
```

Parts/Car.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Car extends Vehicle
{
}
```

Parts/Engine.php

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

namespace DesignPatterns\Creational\Builder\Parts;

```

(continues on next page)

(continued from previous page)

```
class Engine
{
}
```

Parts/Wheel.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Wheel
{
}
```

Parts/Door.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

Test

Tests/DirectorTest.php

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

namespace DesignPatterns\Creational\Builder\Tests;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
28         }
29 }
```

1.1.3 Factory Method

Purpose

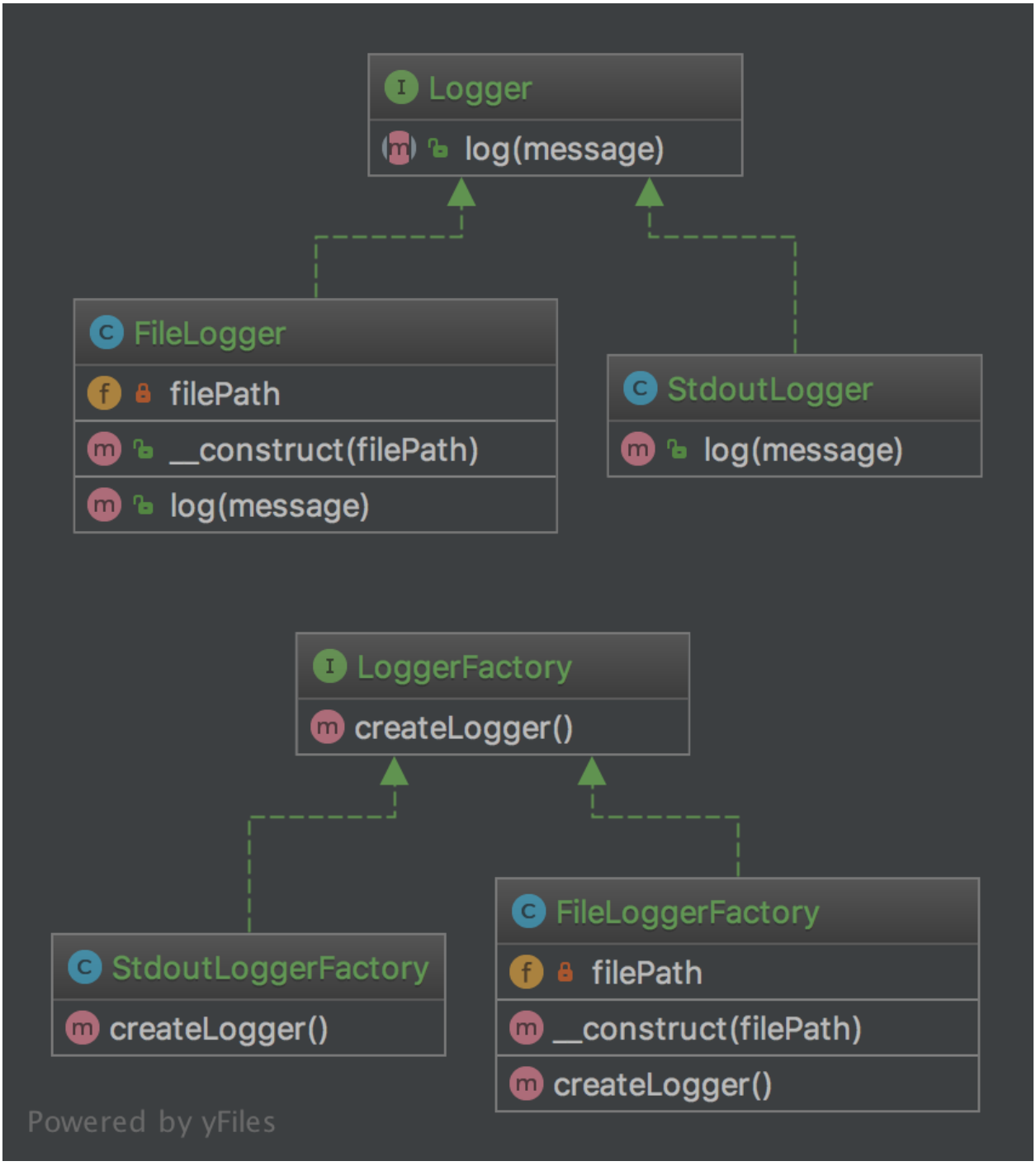
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects.

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

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

namespace DesignPatterns\Creational\FactoryMethod;

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

StdoutLogger.php

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

namespace DesignPatterns\Creational\FactoryMethod;

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

FileLogger.php

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

namespace DesignPatterns\Creational\FactoryMethod;

class FileLogger implements Logger
{
    private string $filePath;

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

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

LoggerFactory.php

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

namespace DesignPatterns\Creational\FactoryMethod;

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

StdoutLoggerFactory.php

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

namespace DesignPatterns\Creational\FactoryMethod;

class StdoutLoggerFactory implements LoggerFactory
{
    public function createLogger(): Logger
    {
        return new StdoutLogger();
    }
}
```

FileLoggerFactory.php

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

namespace DesignPatterns\Creational\FactoryMethod;

class FileLoggerFactory implements LoggerFactory
{
    private string $filePath;

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

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

Test

Tests/FactoryMethodTest.php

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

namespace DesignPatterns\Creational\FactoryMethod\Tests;

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

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

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

(continues on next page)

(continued from previous page)

```
20
21      public function testCanCreateFileLogging()
22      {
23          $loggerFactory = new FileLoggerFactory(sys_get_temp_dir());
24          $logger = $loggerFactory->createLogger();
25
26          $this->assertInstanceOf(FileLogger::class, $logger);
27      }
28  }
```

1.1.4 Pool

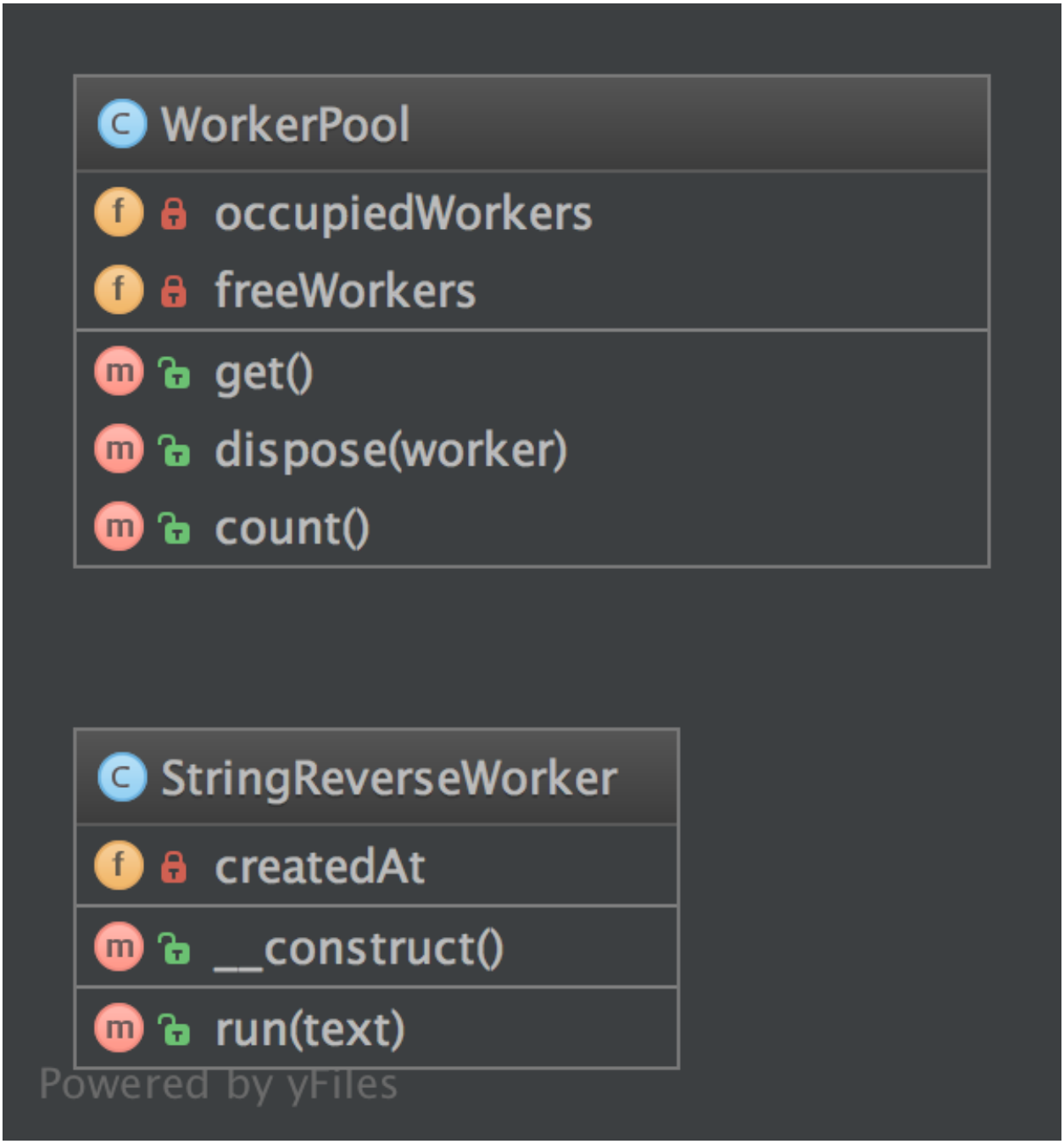
Purpose

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

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

```

(continues on next page)

(continued from previous page)

```php
namespace DesignPatterns\Creational\Pool;

use Countable;

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

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

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

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

        return $worker;
    }

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

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

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

StringReverseWorker.php

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

namespace DesignPatterns\Creational\Pool;

use DateTime;

class StringReverseWorker
{
    private DateTime $createdAt;

```

(continues on next page)

(continued from previous page)

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

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

Test

Tests/PoolTest.php

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

namespace DesignPatterns\Creational\Pool\Tests;

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

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

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

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

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

1.1.5 Prototype

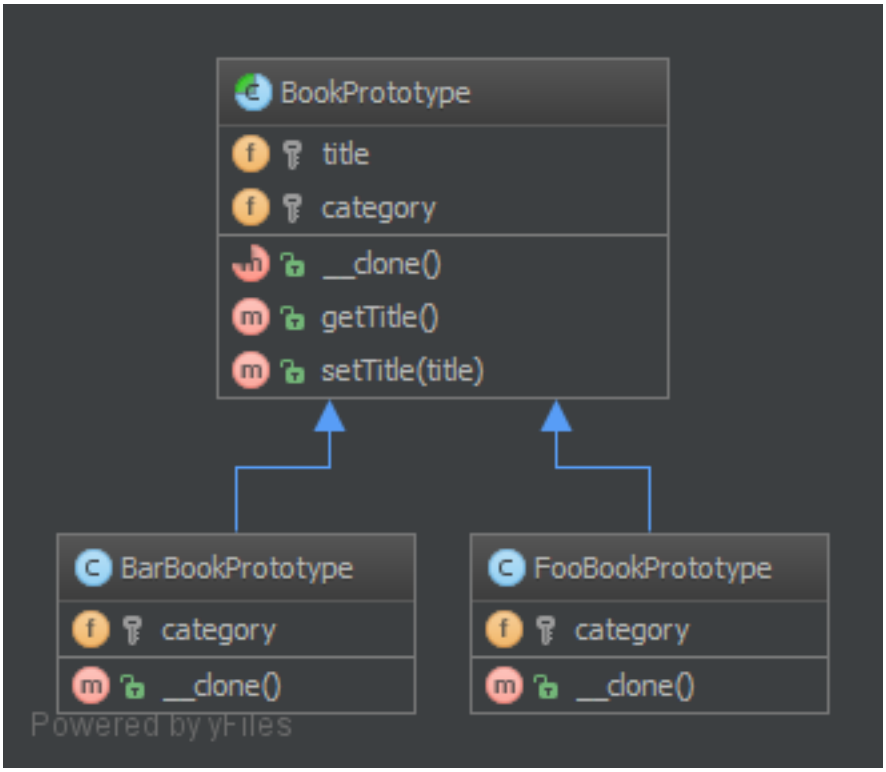
Purpose

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

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

namespace DesignPatterns\Creational\Prototype;

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

    abstract public function __clone();

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

```

(continues on next page)

(continued from previous page)

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

BarBookPrototype.php

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

namespace DesignPatterns\Creational\Prototype;

class BarBookPrototype extends BookPrototype
{
    protected string $category = 'Bar';

    public function __clone()
    {
    }
}
```

FooBookPrototype.php

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

namespace DesignPatterns\Creational\Prototype;

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

    public function __clone()
    {
    }
}
```

Test

Tests/PrototypeTest.php

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

namespace DesignPatterns\Creational\Prototype\Tests;

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

class PrototypeTest extends TestCase
{
    public function testCanGetFooBook()
    {
        $fooPrototype = new FooBookPrototype();
        $barPrototype = new BarBookPrototype();

```

(continues on next page)

(continued from previous page)

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

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

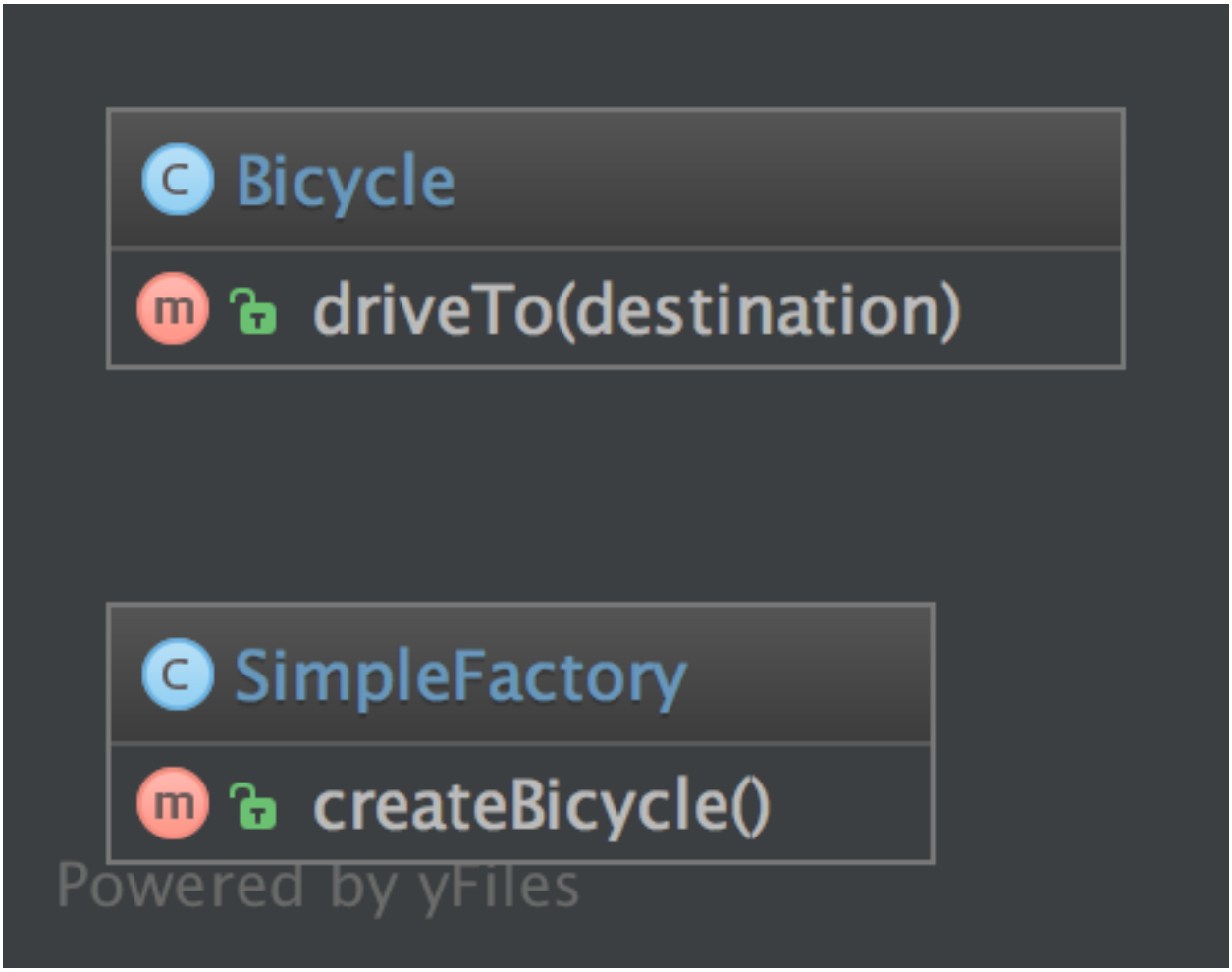
1.1.6 Simple Factory

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

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

namespace DesignPatterns\Creational\SimpleFactory;

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

Bicycle.php

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

namespace DesignPatterns\Creational\SimpleFactory;

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

Usage

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

Test

Tests/SimpleFactoryTest.php

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

namespace DesignPatterns\Creational\SimpleFactory\Tests;

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

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

1.1.7 Singleton

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

Purpose

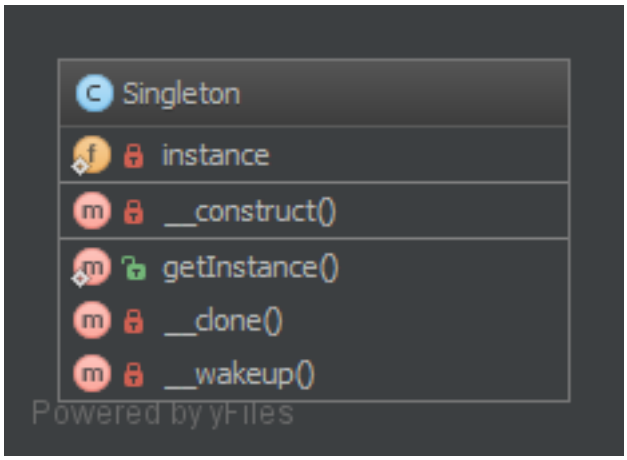
To have only one instance of this object in the application that will handle all calls.

Examples

- DB Connector

- Logger
- Lock file for the application (there is only one in the filesystem . . .)

UML Diagram



Code

You can also find this code on GitHub

Singleton.php

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

namespace DesignPatterns\Creational\Singleton;

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

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

        return static::$instance;
    }

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

(continues on next page)

(continued from previous page)

```
    }

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

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

Test

Tests/SingletonTest.php

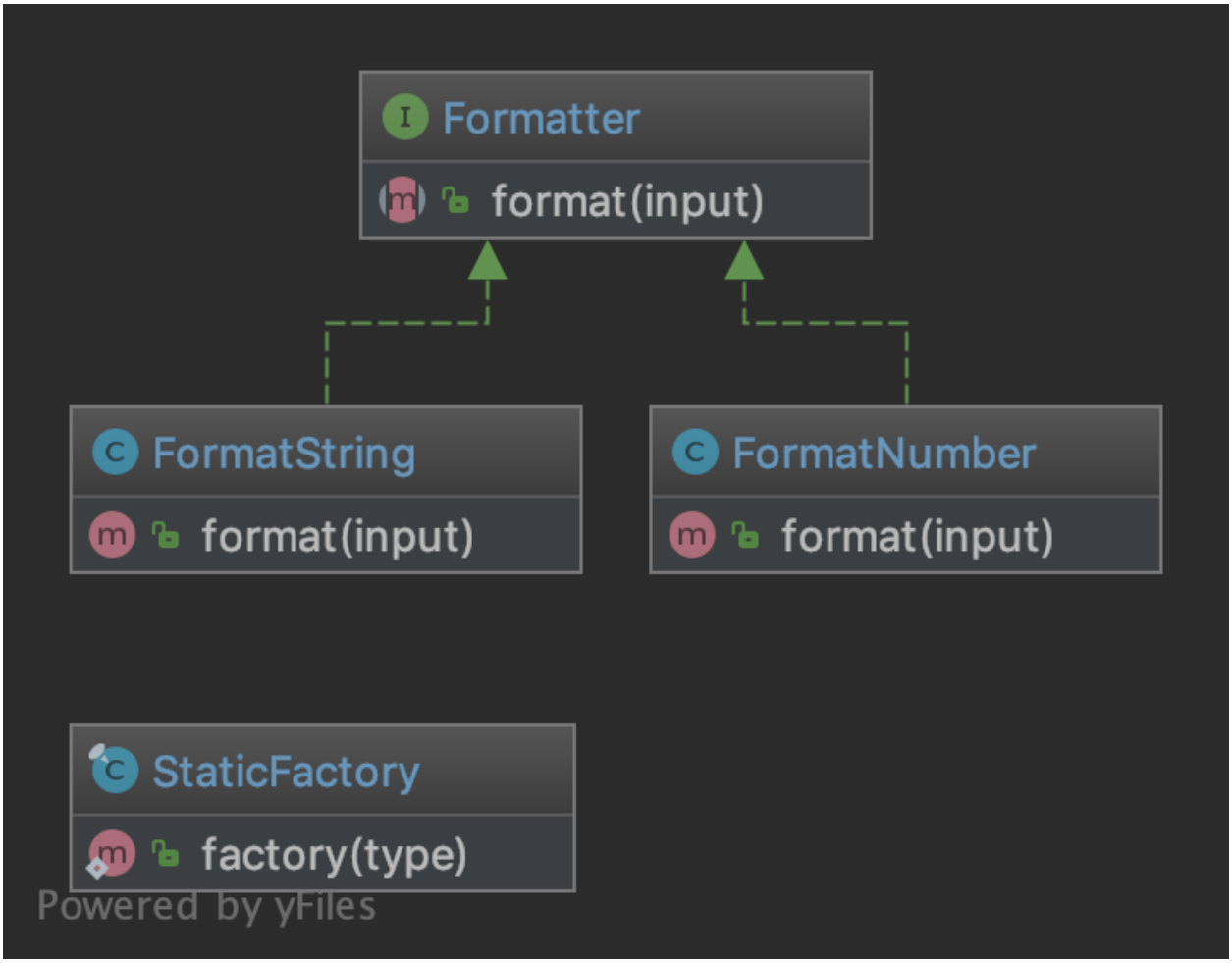
```
<?php declare(strict_types=1);

namespace DesignPatterns\Creational\Singleton\Tests;

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

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

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

1.1.8 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

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

namespace DesignPatterns\Creational\StaticFactory;

use InvalidArgumentException;

/**
 * Note1: Remember, static means global state which is evil because it can't be mocked for tests
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
final class StaticFactory
{
    public static function factory(string $type): Formatter
```

(continues on next page)

(continued from previous page)

```
    {
        if ($type == 'number') {
            return new FormatNumber();
        } elseif ($type == 'string') {
            return new FormatString();
        }

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

Formatter.php

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

namespace DesignPatterns\Creational\StaticFactory;

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

FormatString.php

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

namespace DesignPatterns\Creational\StaticFactory;

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

FormatNumber.php

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

namespace DesignPatterns\Creational\StaticFactory;

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

Test

Tests/StaticFactoryTest.php

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

namespace DesignPatterns\Creational\StaticFactory\Tests;

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

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

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

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

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

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

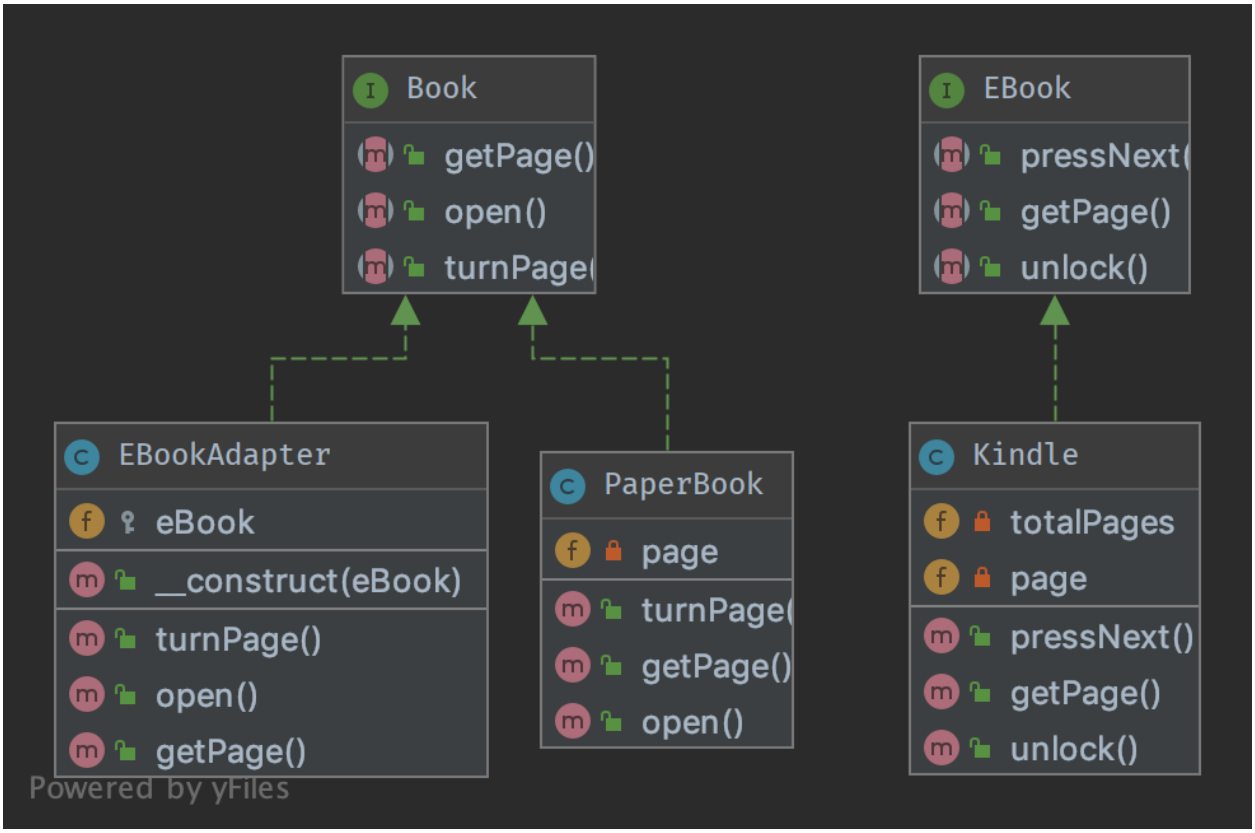
Purpose

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

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

namespace DesignPatterns\Structural\Adapter;

interface Book
{
    public function turnPage();

    public function open();

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

PaperBook.php

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

namespace DesignPatterns\Structural\Adapter;

```

(continues on next page)

(continued from previous page)

```php
class PaperBook implements Book
{
    private int $page;

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

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

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

EBook.php

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

namespace DesignPatterns\Structural\Adapter;

interface EBook
{
    public function unlock();

    public function pressNext();

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

EBookAdapter.php

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

namespace DesignPatterns\Structural\Adapter;

/**
 * This is the adapter here. Notice it implements Book,
 * therefore you don't have to change the code of the client which is using a Book
 */
class EBookAdapter implements Book
{
    protected EBook $eBook;

    public function __construct(EBook $eBook)
    {
```

(continues on next page)

(continued from previous page)

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

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

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

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

Kindle.php

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

namespace DesignPatterns\Structural\Adapter;

/**
 * this is the adapted class. In production code, this could be a class from another package, some vendor code.
 * Notice that it uses another naming scheme and the implementation does something similar but in another way
 */
class Kindle implements EBook
{
    private int $page = 1;
    private int $totalPages = 100;

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

    public function unlock()
    {
    }

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

(continues on next page)

(continued from previous page)

```
     * @return int[]
     */
    public function getPage(): array
    {
        return [$this->page, $this->totalPages];
    }
}
```

Test

Tests/AdapterTest.php

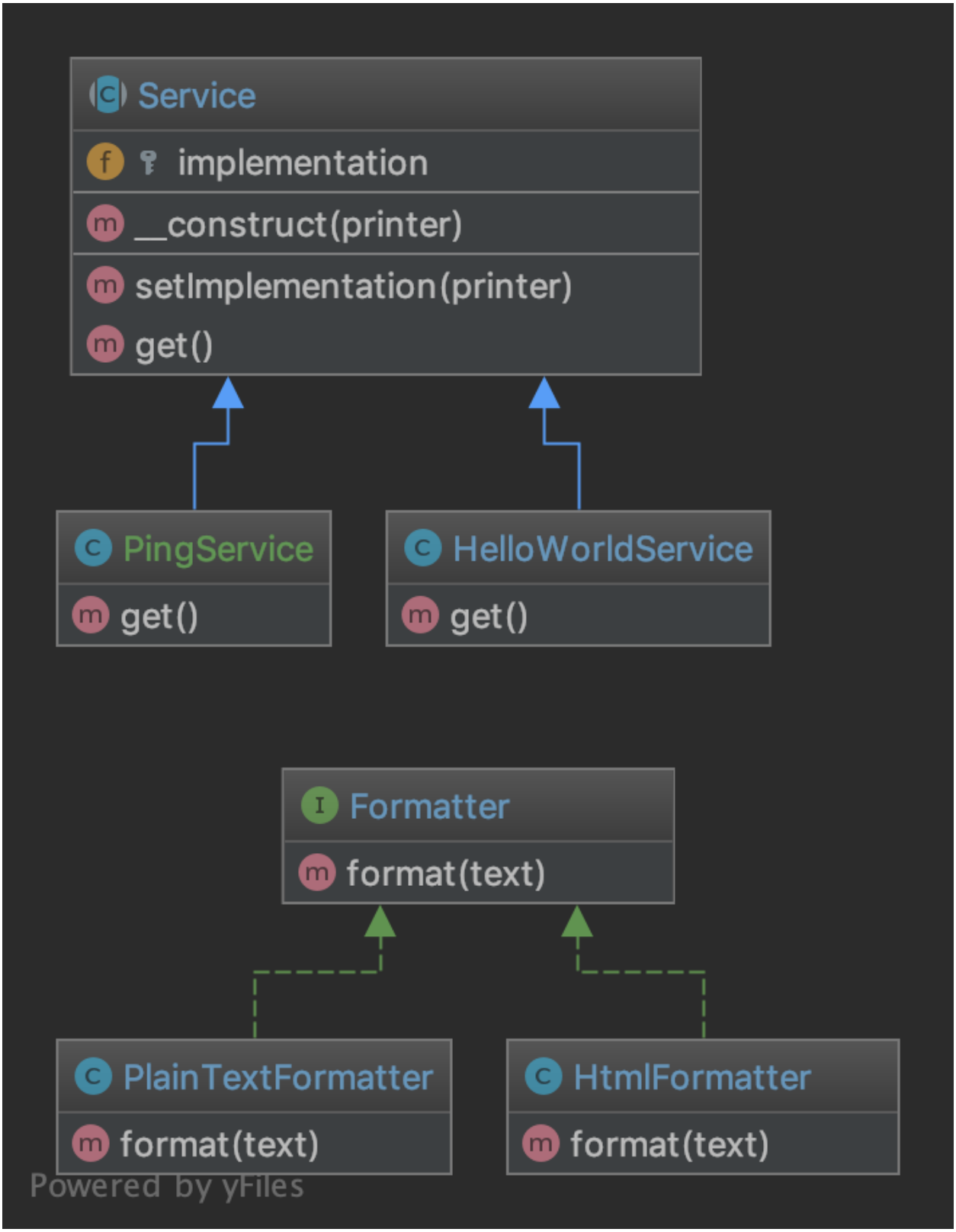
```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter\Tests;

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

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

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

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

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

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

1.2.2 Bridge

Purpose

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

UML Diagram


Service
implementation
__construct(printer)
setImplementation(printer)
get()
PingService
get()
HelloWorldService
get()
Formatter
format(text)
PlainTextFormatter
format(text)
HtmlFormatter
format(text)
Powered by yFiles

Code

You can also find this code on GitHub

Formatter.php

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

namespace DesignPatterns\Structural\Bridge;

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

PlainTextFormatter.php

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

namespace DesignPatterns\Structural\Bridge;

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

HtmlFormatter.php

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

namespace DesignPatterns\Structural\Bridge;

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

Service.php

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

namespace DesignPatterns\Structural\Bridge;

abstract class Service
{
    protected Formatter $implementation;

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

```

(continues on next page)

(continued from previous page)

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

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

HelloWorldService.php

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

namespace DesignPatterns\Structural\Bridge;

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

PingService.php

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

namespace DesignPatterns\Structural\Bridge;

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

Test

Tests/BridgeTest.php

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

namespace DesignPatterns\Structural\Bridge\Tests;

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

class BridgeTest extends TestCase
{
    public function testCanPrintUsingThePlainTextFormatter()
    {
        $service = new HelloWorldService(new PlainTextFormatter());

```

(continues on next page)

(continued from previous page)

```
16          $this->assertSame('Hello World', $service->get());
17      }
18
19      public function testCanPrintUsingTheHtmlFormatter()
20      {
21          $service = new HelloWorldService(new HtmlFormatter());
22
23          $this->assertSame('<p>Hello World</p>', $service->get());
24      }
25  }
```

1.2.3 Composite

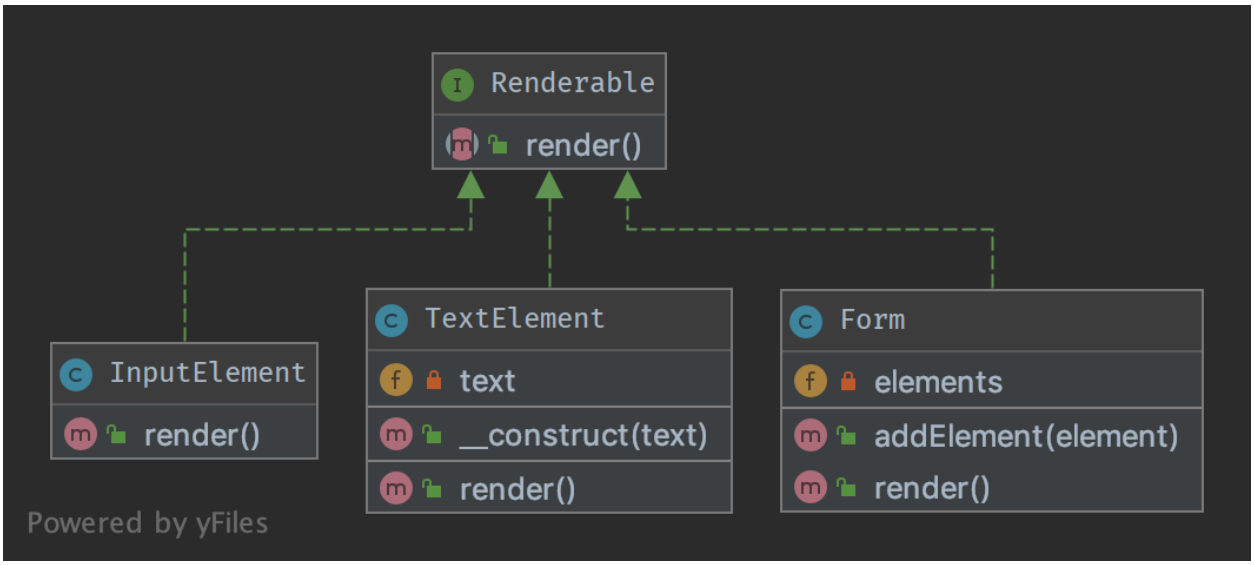
Purpose

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

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

namespace DesignPatterns\Structural\Composite;

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

Form.php

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

namespace DesignPatterns\Structural\Composite;

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

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

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

        $formCode .= '</form>';

        return $formCode;
    }

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

InputElement.php

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

(continues on next page)

(continued from previous page)

```

namespace DesignPatterns\Structural\Composite;

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

TextElement.php

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

namespace DesignPatterns\Structural\Composite;

class TextElement implements Renderable
{
    private string $text;

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

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

Test

Tests/CompositeTest.php

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

namespace DesignPatterns\Structural\Composite\Tests;

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

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

(continues on next page)

(continued from previous page)

```
20          $form->addElement($embed);
21
22          // This is just an example, in a real world scenario it is important to␣
    ↪remember that web browsers do not
23          // currently support nested forms
24
25          $this->assertSame(
26              '<form>Email:<input type="text" /><form>Password:<input type="text" /></
    ↪form></form>',
27              $form->render()
28          );
29      }
30  }
```

1.2.4 Data Mapper

Purpose

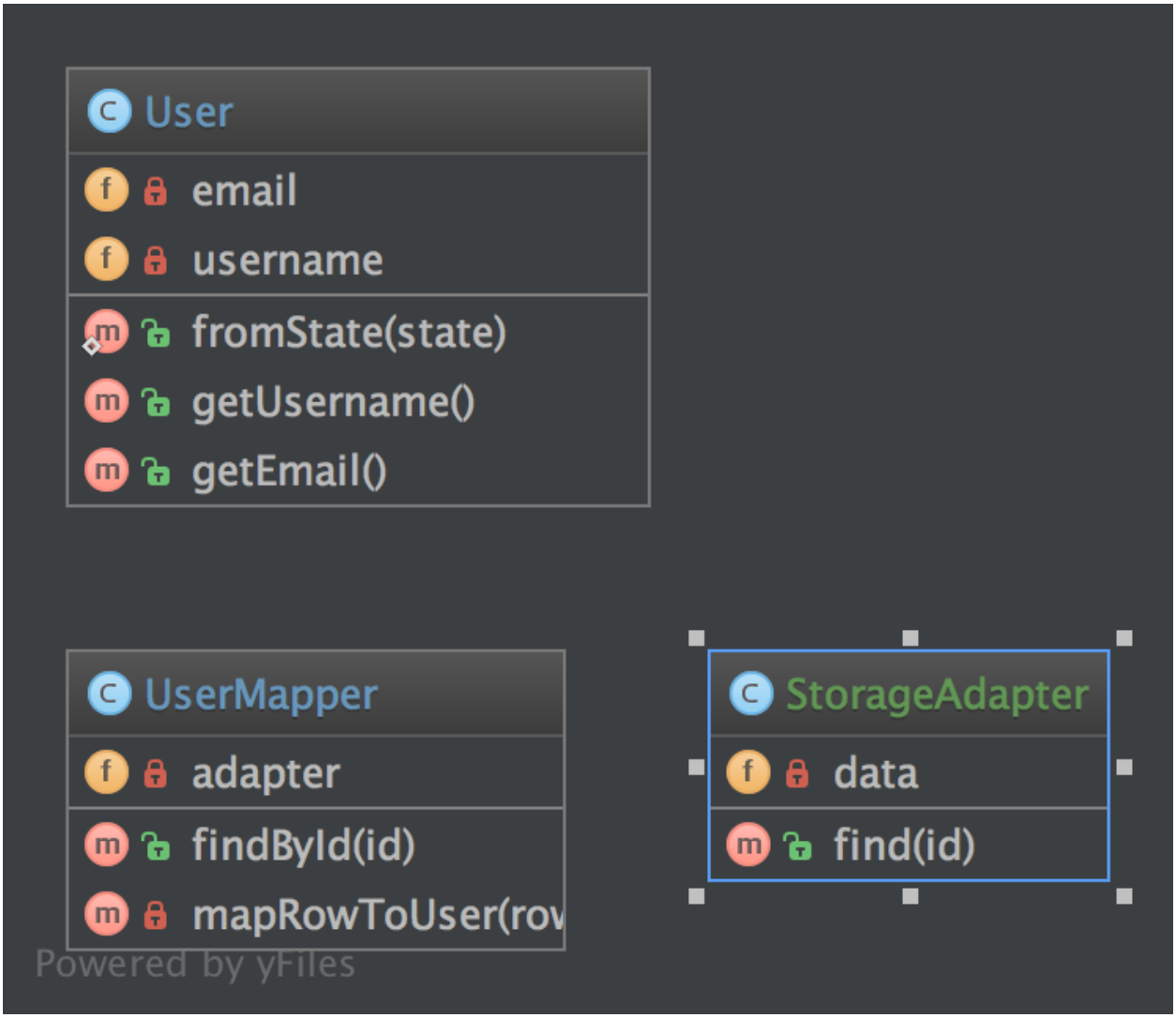
A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent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

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

namespace DesignPatterns\Structural\DataMapper;

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

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

(continues on next page)

(continued from previous page)

```
        // validate state before accessing keys!

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

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

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

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

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

UserMapper.php

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

namespace DesignPatterns\Structural\DataMapper;

use InvalidArgumentException;

class UserMapper
{
    private StorageAdapter $adapter;

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

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

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

(continues on next page)

(continued from previous page)

```
            throw new InvalidArgumentException("User #$id not found");
        }

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

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

StorageAdapter.php

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

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
{
    private array $data = [];

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

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

        return null;
    }
}
```

Test

Tests/DataMapperTest.php

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

namespace DesignPatterns\Structural\DataMapper\Tests;

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

(continues on next page)

(continued from previous page)

```php

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

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

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

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

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

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

1.2.5 Decorator

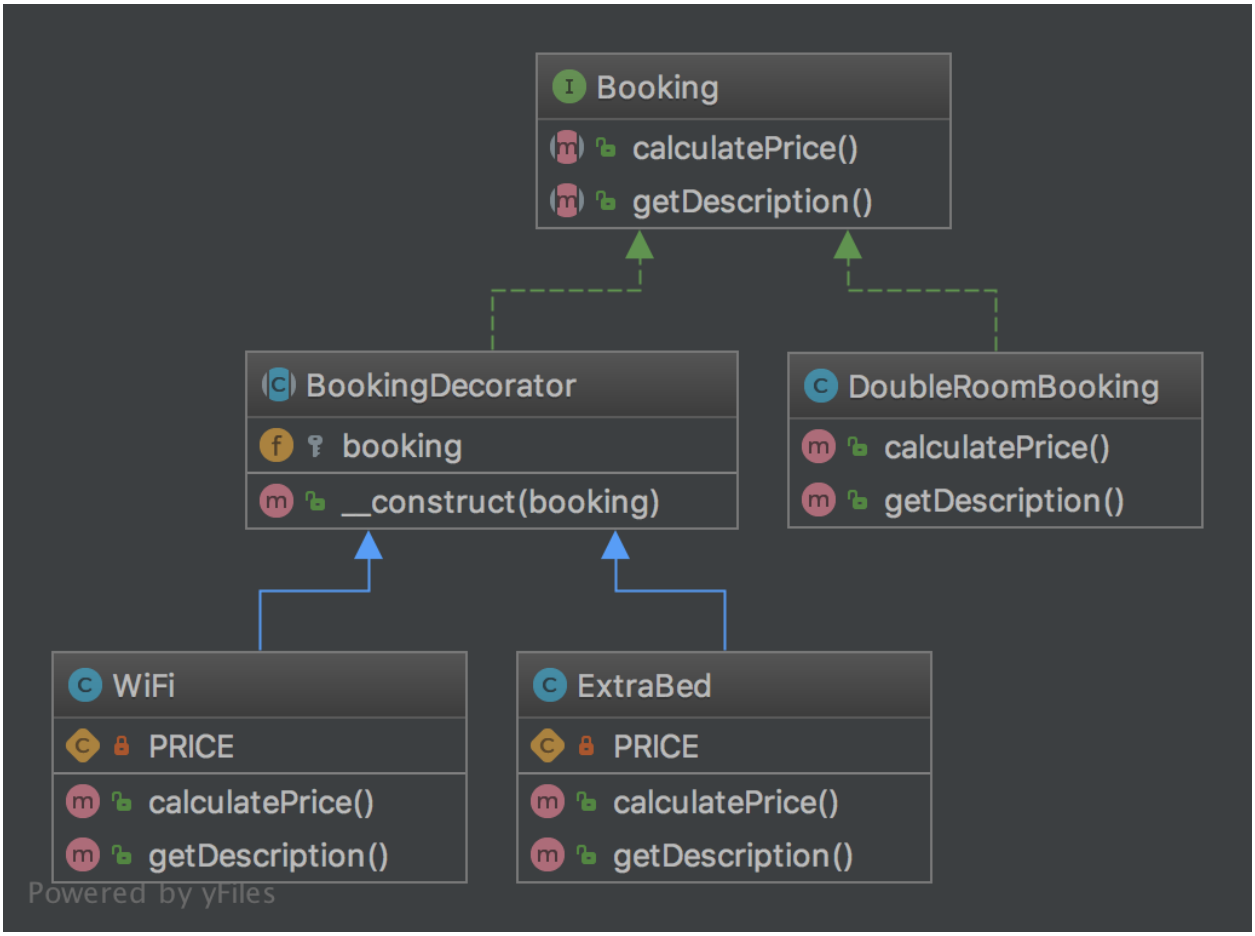
Purpose

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

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

namespace DesignPatterns\Structural\Decorator;

interface Booking
{
    public function calculatePrice(): int;

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

BookingDecorator.php

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

namespace DesignPatterns\Structural\Decorator;
```

(continues on next page)

(continued from previous page)

```

abstract class BookingDecorator implements Booking
{
    protected Booking $booking;

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

DoubleRoomBooking.php

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

namespace DesignPatterns\Structural\Decorator;

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

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

ExtraBed.php

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

namespace DesignPatterns\Structural\Decorator;

class ExtraBed extends BookingDecorator
{
    private const PRICE = 30;

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

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

WiFi.php

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

namespace DesignPatterns\Structural\Decorator;

```

(continues on next page)

(continued from previous page)

```php
class WiFi extends BookingDecorator
{
    private const PRICE = 2;

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

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

Test

Tests/DecoratorTest.php

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

namespace DesignPatterns\Structural\Decorator\Tests;

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

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

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

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

        $this->assertSame(42, $booking->calculatePrice());
        $this->assertSame('double room with wifi', $booking->getDescription());
    }

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

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

(continues on next page)

(continued from previous page)

```
37         }
38 }
```

1.2.6 Dependency Injection

Purpose

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

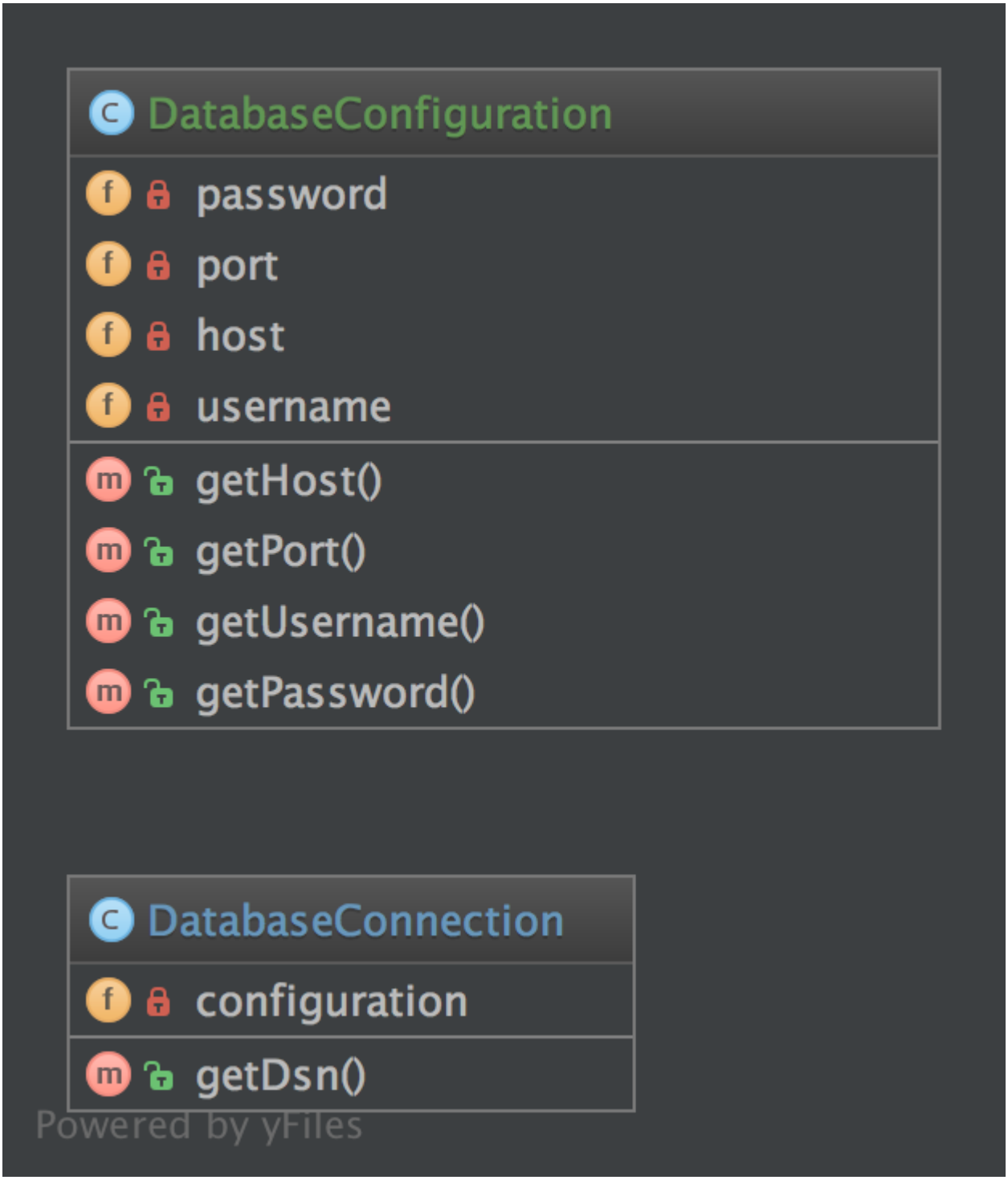
Usage

`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

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

namespace DesignPatterns\Structural\DependencyInjection;

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

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

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

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

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

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

DatabaseConnection.php

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

namespace DesignPatterns\Structural\DependencyInjection;

class DatabaseConnection
{
    private DatabaseConfiguration $configuration;

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

    public function getDsn(): string
```

(continues on next page)

(continued from previous page)

```
    {
        // this is just for the sake of demonstration, not a real DSN
        // notice that only the injected config is used here, so there is
        // a real separation of concerns here

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

Test

Tests/DependencyInjectionTest.php

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

namespace DesignPatterns\Structural\DependencyInjection\Tests;

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

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

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

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.

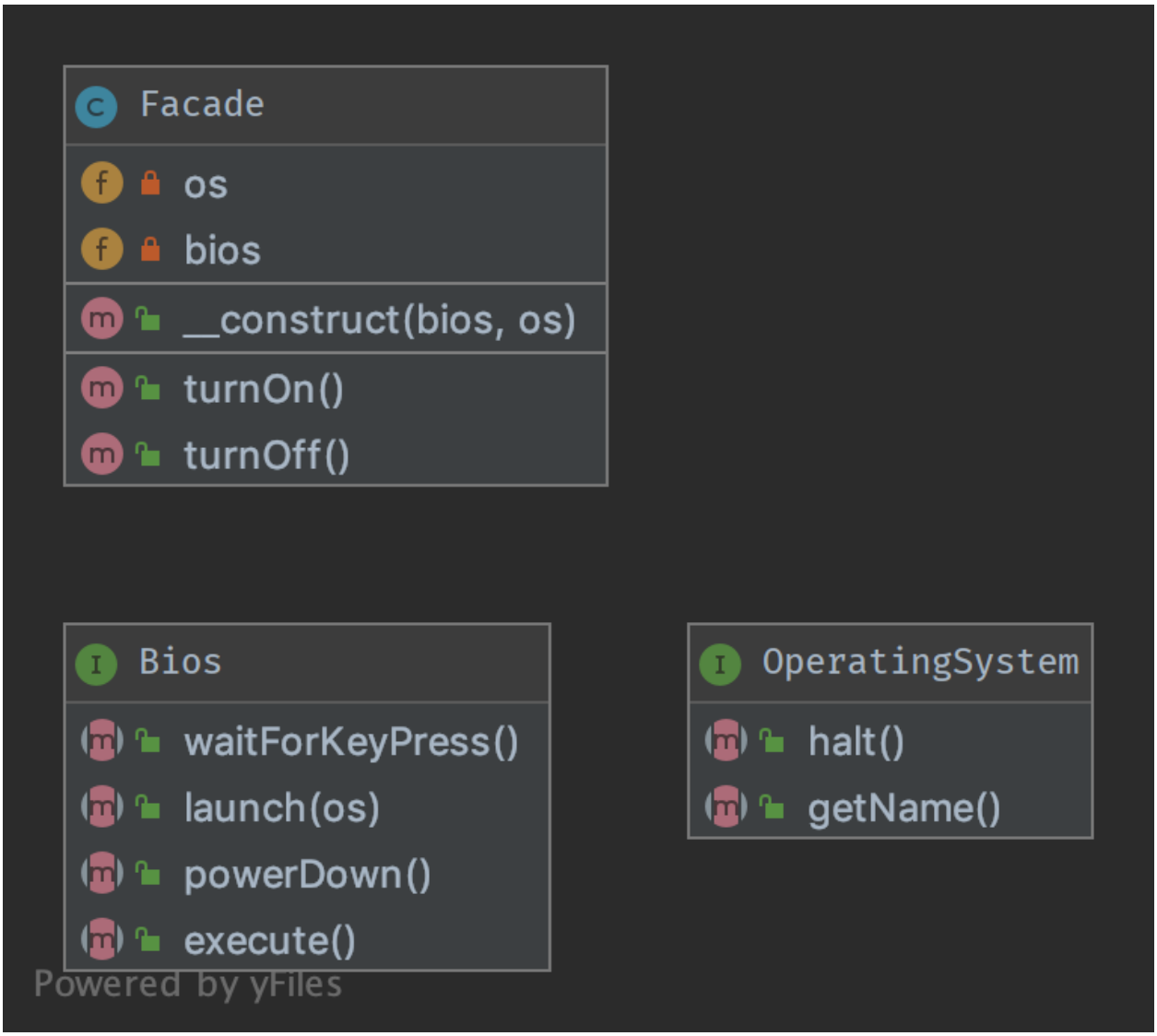
A Facade is meant to decouple a client and a sub-system by embedding many (but sometimes just one) interface, and of course to reduce complexity.

- 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

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

namespace DesignPatterns\Structural\Facade;

class Facade
{
```

(continues on next page)

(continued from previous page)

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

    public function __construct(Bios $bios, OperatingSystem $os)
    {
        $this->bios = $bios;
        $this->os = $os;
    }

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

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

OperatingSystem.php

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

namespace DesignPatterns\Structural\Facade;

interface OperatingSystem
{
    public function halt();

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

Bios.php

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

namespace DesignPatterns\Structural\Facade;

interface Bios
{
    public function execute();

    public function waitForKeyPress();

    public function launch(OperatingSystem $os);

    public function powerDown();
}
```

Test

Tests/FacadeTest.php

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

namespace DesignPatterns\Structural\Facade\Tests;

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

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

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

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

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

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

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

1.2.8 Fluent Interface

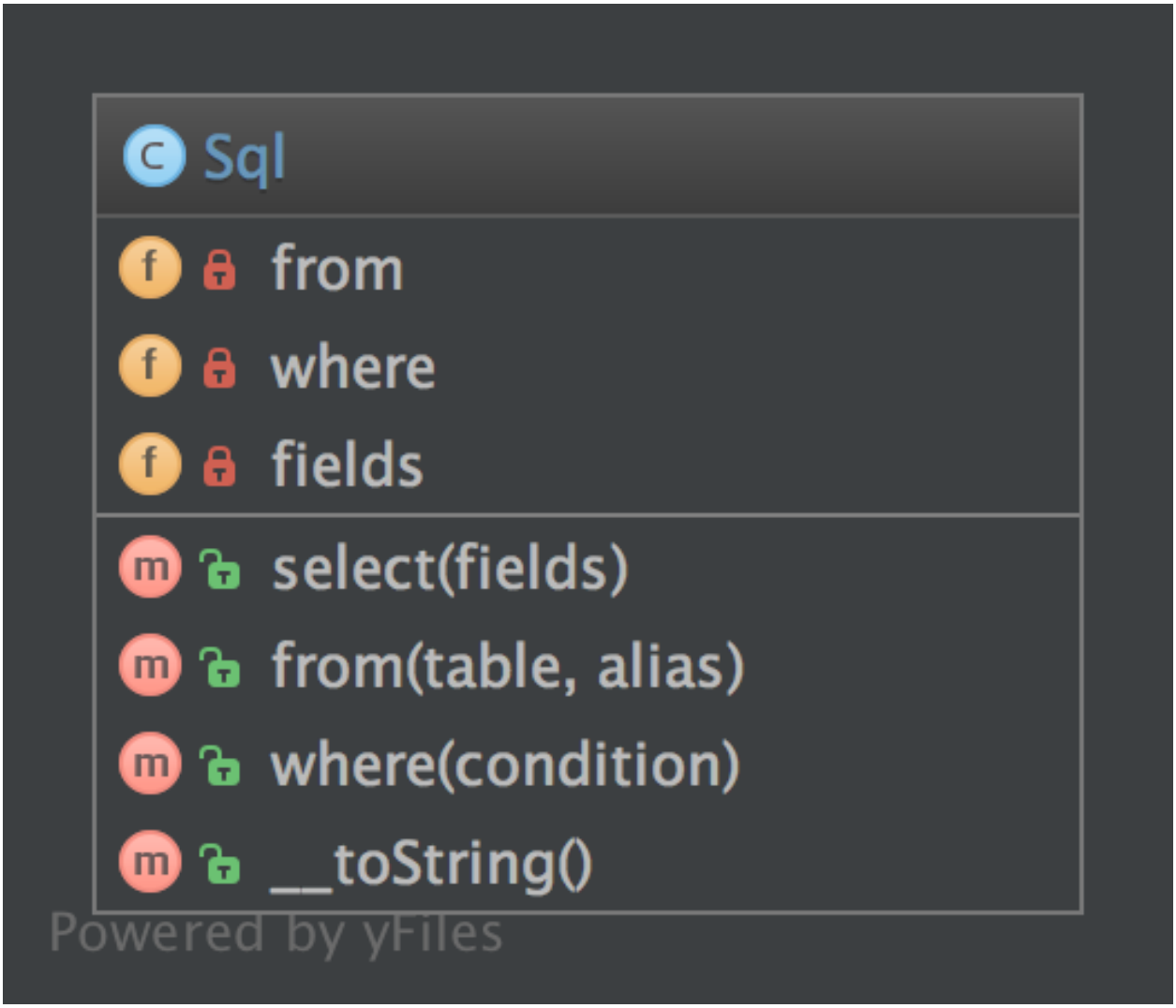
Purpose

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

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

namespace DesignPatterns\Structural\FluentInterface;

class Sql
{
    private array $fields = [];
    private array $from = [];
    private array $where = [];

    public function select(array $fields): Sql
```

(continues on next page)

(continued from previous page)

```
    {
        $this->fields = $fields;

        return $this;
    }

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

        return $this;
    }

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

        return $this;
    }

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

Test

Tests/FluentInterfaceTest.php

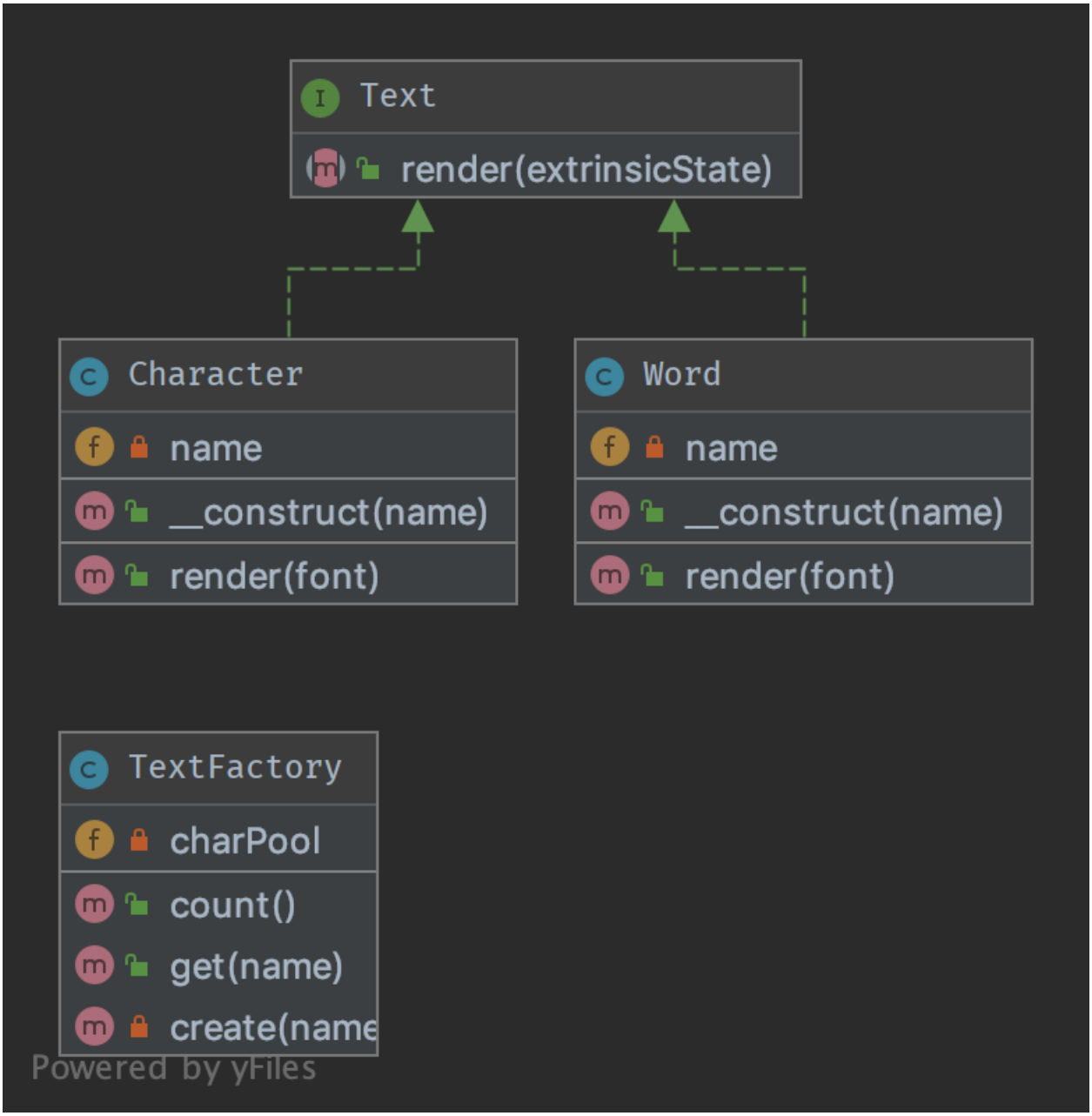
```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\FluentInterface\Tests;

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

class FluentInterfaceTest extends TestCase
{
    public function testBuildSQL()
    {
        $query = (new Sql())
                ->select(['foo', 'bar'])
                ->from('foobar', 'f')
                ->where('f.bar = ?');

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

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

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

namespace DesignPatterns\Structural\Flyweight;

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

Word.php

```php
<?php

namespace DesignPatterns\Structural\Flyweight;

class Word implements Text
{
    private string $name;

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

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

Character.php

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

namespace DesignPatterns\Structural\Flyweight;

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

```

(continues on next page)

(continued from previous page)

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

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

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

TextFactory.php

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

namespace DesignPatterns\Structural\Flyweight;

use Countable;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
}
```

Test

Tests/FlyweightTest.php

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

namespace DesignPatterns\Structural\Flyweight\Tests;

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

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

    private array $fonts = ['Arial', 'Times New Roman', 'Verdana', 'Helvetica'];

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

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

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

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

            $this->assertSame(sprintf('Word %s with font foobar', $word), $rendered);
        }

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

1.2.10 Proxy

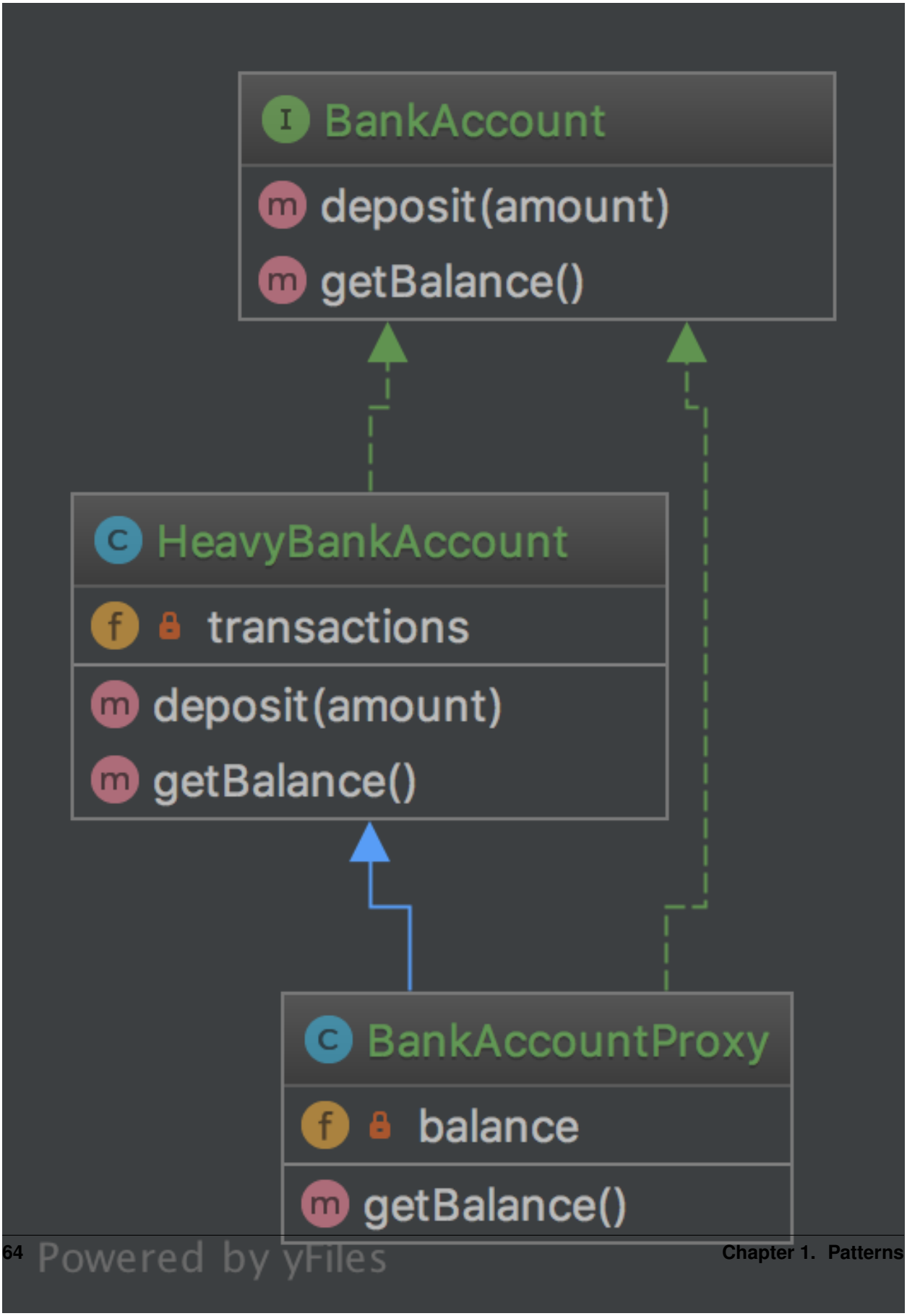
Purpose

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


BankAccount
deposit(amount)
getBalance()
HeavyBankAccount
transactions
deposit(amount)
getBalance()
BankAccountProxy
balance
getBalance()
Powered by yFiles

Code

You can also find this code on GitHub

BankAccount.php

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

namespace DesignPatterns\Structural\Proxy;

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

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

HeavyBankAccount.php

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

namespace DesignPatterns\Structural\Proxy;

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

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

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

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

BankAccountProxy.php

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

namespace DesignPatterns\Structural\Proxy;

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

    public function getBalance(): int
    {
        // because calculating balance is so expensive,
```

(continues on next page)

(continued from previous page)

```
        // the usage of BankAccount::getBalance() is delayed until it really is needed
        // and will not be calculated again for this instance

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

        return $this->balance;
    }
}
```

Test

ProxyTest.php

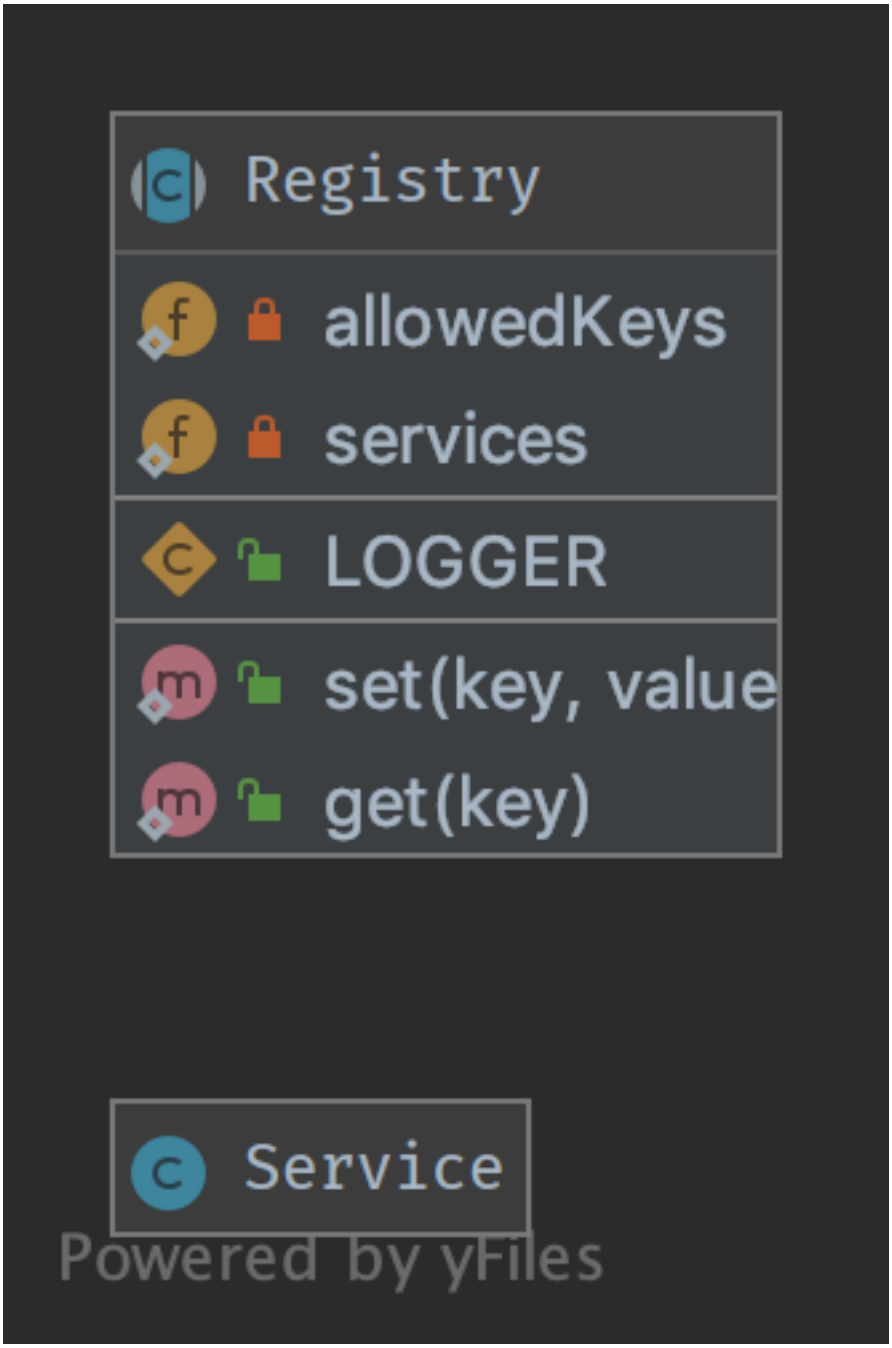
```
<?php declare(strict_types=1);

namespace DesignPatterns\Structural\Proxy\Tests;

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

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

```php
namespace DesignPatterns\Structural\Registry;

use InvalidArgumentException;

abstract class Registry
{
    const LOGGER = 'logger';

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

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

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

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

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

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

Service.php

```php
<?php

namespace DesignPatterns\Structural\Registry;

class Service
{

}
```

Test

Tests/RegistryTest.php

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

namespace DesignPatterns\Structural\Registry\Tests;

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

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

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

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

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

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

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

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

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

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

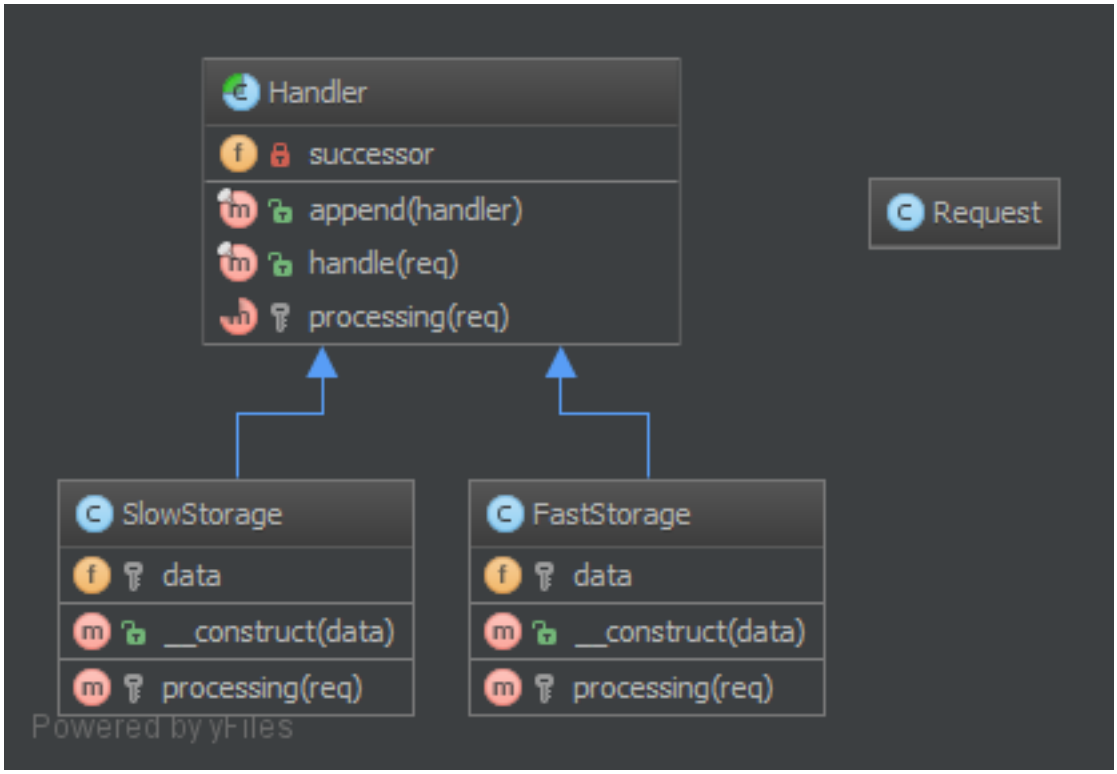
Purpose

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

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

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

use Psr\Http\Message\RequestInterface;

abstract class Handler
{
    private ?Handler $successor = null;

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

    /**
     * This approach by using a template method pattern ensures you that
     * each subclass will not forget to call the successor
     */
    final public function handle(RequestInterface $request): ?string
    {
        $processed = $this->processing($request);

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

        return $processed;
    }

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

Responsible/FastStorage.php

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

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

class HttpInMemoryCacheHandler extends Handler
{
    private array $data;

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

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

(continues on next page)

(continued from previous page)

```
    }

    protected function processing(RequestInterface $request): ?string
    {
        $key = sprintf(
            '%s?%s',
            $request->getUri()->getPath(),
            $request->getUri()->getQuery()
        );

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

        return null;
    }
}
```

Responsible/SlowStorage.php

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

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

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

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

Test

Tests/ChainTest.php

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

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
use␣
↪DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\HttpInMemoryCacheHandler;
↪
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowDatabaseHandler;
use PHPUnit\Framework\TestCase;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\UriInterface;

```

(continues on next page)

(continued from previous page)

```php
class ChainTest extends TestCase
{
    private Handler $chain;

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

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

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

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

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

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

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

1.3.2 Command

Purpose

To encapsulate invocation and decoupling.

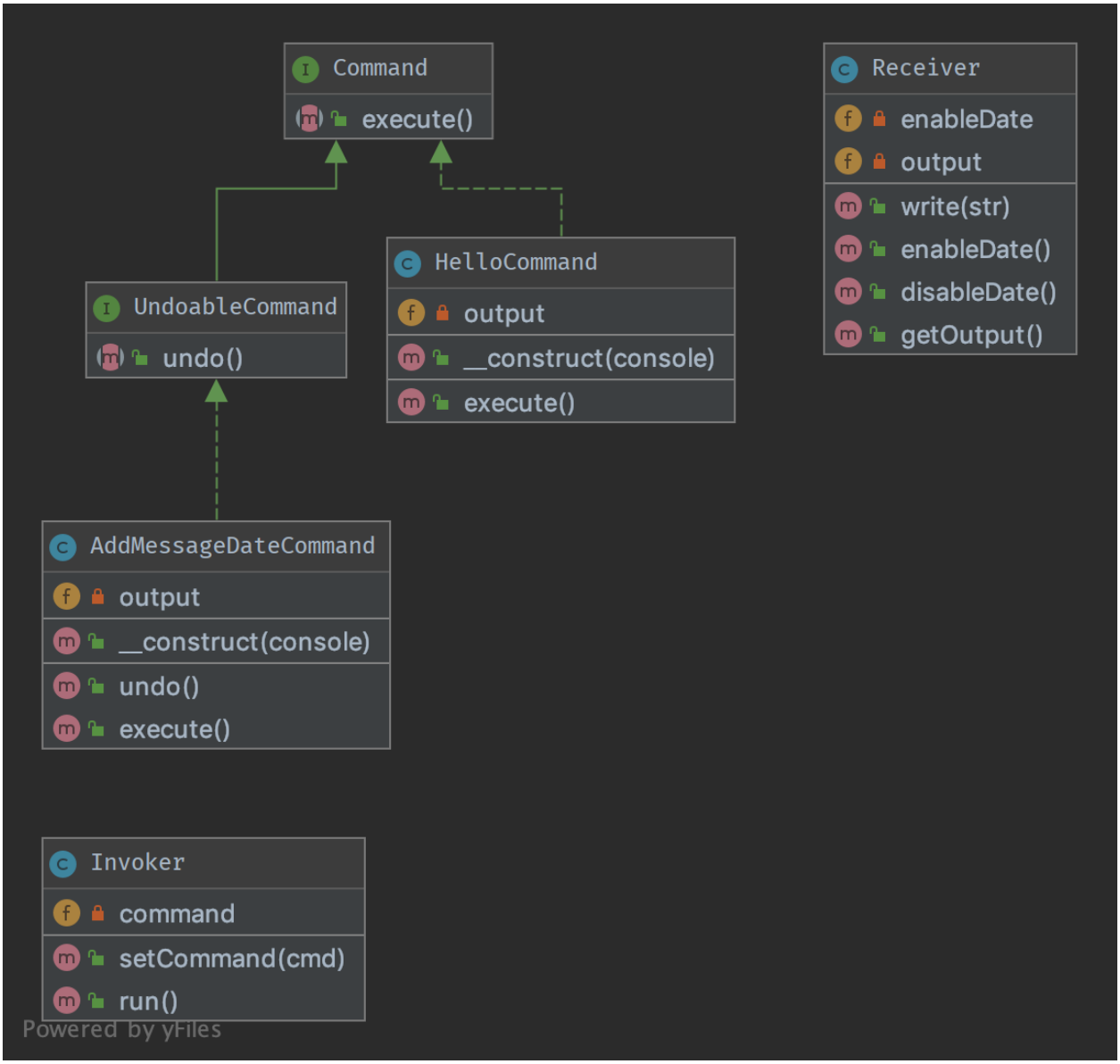
We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

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

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

namespace DesignPatterns\Behavioral\Command;

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

UndoableCommand.php

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

namespace DesignPatterns\Behavioral\Command;

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

HelloCommand.php

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

namespace DesignPatterns\Behavioral\Command;

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

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

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

(continues on next page)

(continued from previous page)

```
        $this->output->write('Hello World');
    }
}
```

AddMessageDateCommand.php

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

namespace DesignPatterns\Behavioral\Command;

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

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

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

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

Receiver.php

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

namespace DesignPatterns\Behavioral\Command;

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

(continues on next page)

(continued from previous page)

```
 */
class Receiver
{
    private bool $enableDate = false;

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

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

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

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

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

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

Invoker.php

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

namespace DesignPatterns\Behavioral\Command;

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

    /**
     * in the invoker we find this kind of method for subscribing the command
```

(continues on next page)

(continued from previous page)

```
     * There can be also a stack, a list, a fixed set ...
     */
    public function setCommand(Command $cmd)
    {
        $this->command = $cmd;
    }

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

Test

Tests/CommandTest.php

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

namespace DesignPatterns\Behavioral\Command\Tests;

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

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

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

Tests/UndoableCommandTest.php

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

namespace DesignPatterns\Behavioral\Command\Tests;

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

class UndoableCommandTest extends TestCase
```

(continues on next page)

(continued from previous page)

```
{
    public function testInvocation()
    {
        $invoker = new Invoker();
        $receiver = new Receiver();

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

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

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

        $messageDateCommand->undo();

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

1.3.3 Iterator

Purpose

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

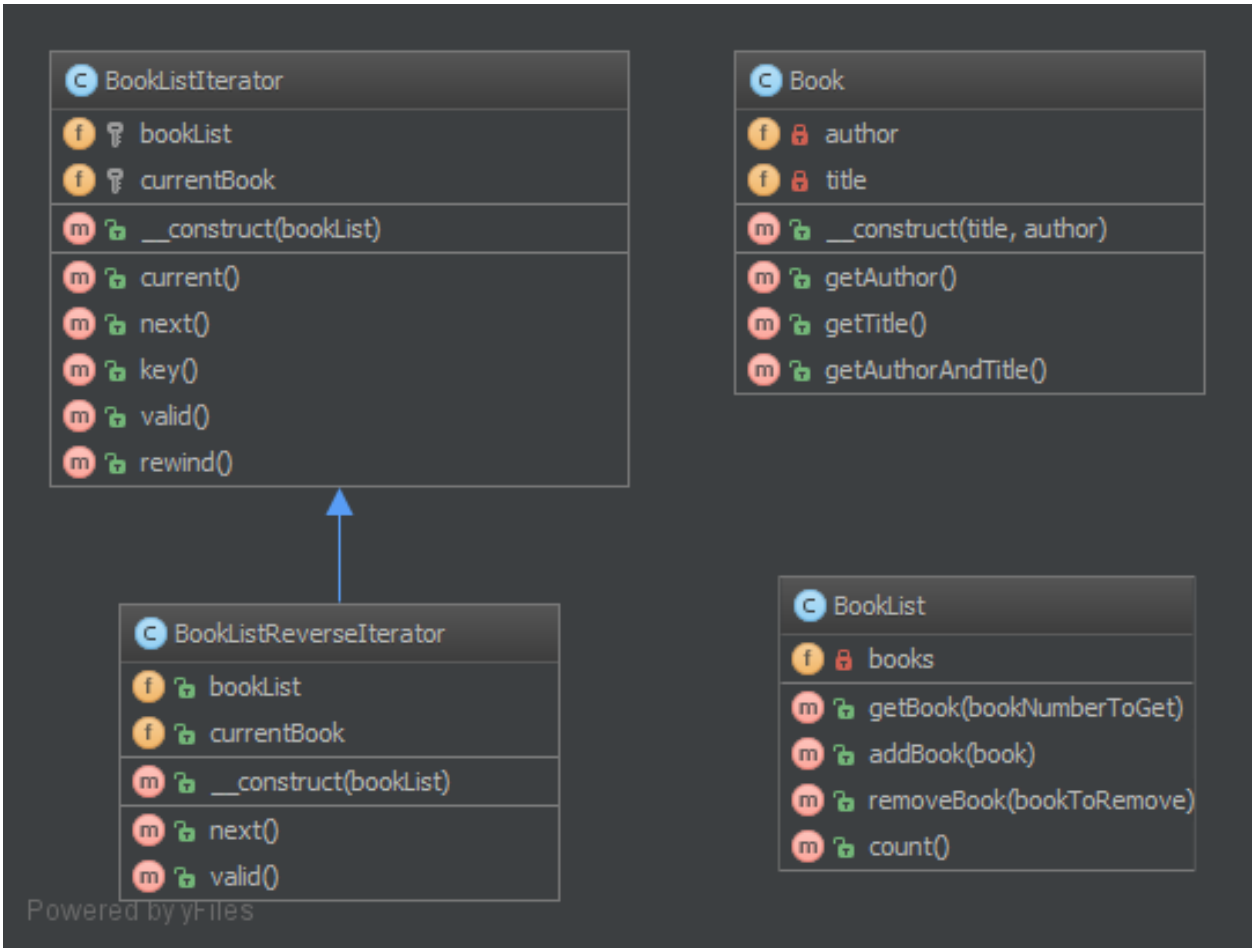
Examples

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

Note

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

UML Diagram



Code

You can also find this code on GitHub

Book.php

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

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    private string $author;
    private string $title;

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

```

(continues on next page)

(continued from previous page)

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

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

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

BookList.php

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

namespace DesignPatterns\Behavioral\Iterator;

use Countable;
use Iterator;

class BookList implements Countable, Iterator
{
    /**
     * @var Book[]
     */
    private array $books = [];
    private int $currentIndex = 0;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    }

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

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

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

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

Test

Tests/IteratorTest.php

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

namespace DesignPatterns\Behavioral\Iterator\Tests;

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

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

        $books = [];

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

        $this->assertSame(
            [
                'Learning PHP Design Patterns by William Sanders',
```

(continues on next page)

(continued from previous page)

```
                'Professional Php Design Patterns by Aaron Saray',
                'Clean Code by Robert C. Martin',
            ],
            $books
        );
    }

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

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

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

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

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

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

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

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

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

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

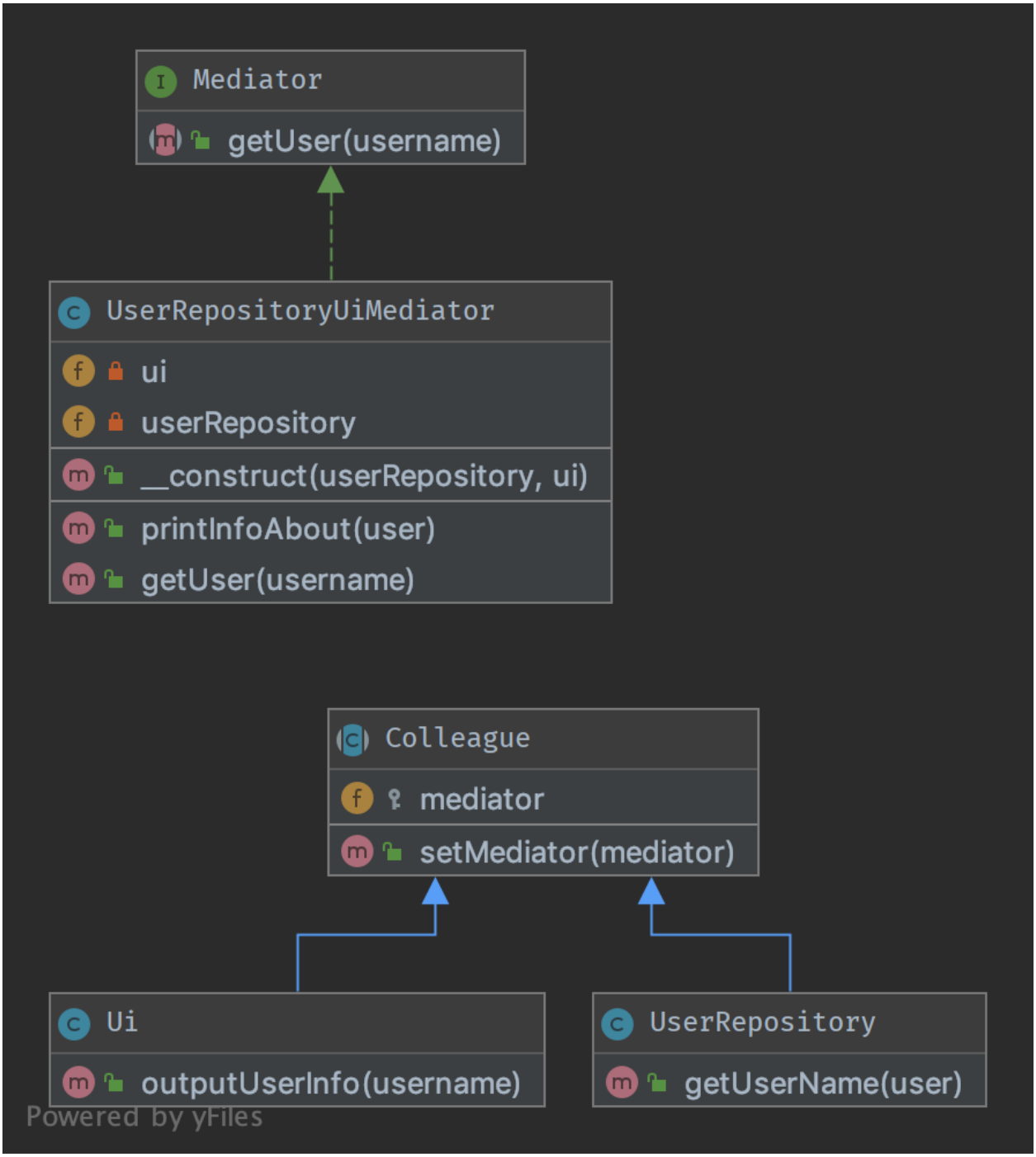
1.3.4 Mediator

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

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

namespace DesignPatterns\Behavioral\Mediator;

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

Colleague.php

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

namespace DesignPatterns\Behavioral\Mediator;

abstract class Colleague
{
    protected Mediator $mediator;

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

Ui.php

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

namespace DesignPatterns\Behavioral\Mediator;

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

UserRepository.php

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

namespace DesignPatterns\Behavioral\Mediator;

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

UserRepositoryUiMediator.php

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

namespace DesignPatterns\Behavioral\Mediator;

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

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

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

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

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

Test

Tests/MediatorTest.php

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

namespace DesignPatterns\Tests\Mediator\Tests;

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

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

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

1.3.5 Memento

Purpose

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

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

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

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

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

Examples

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

UML Diagram

Code

You can also find this code on GitHub

Memento.php

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

namespace DesignPatterns\Behavioral\Memento;

class Memento
```

(continues on next page)

(continued from previous page)

```
{
    private State $state;

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

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

State.php

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

namespace DesignPatterns\Behavioral\Memento;

use InvalidArgumentException;

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

    private string $state;

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

    public function __construct(string $state)
    {
        self::ensureIsValidState($state);

        $this->state = $state;
    }

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

    public function __toString(): string
    {
```

(continues on next page)

(continued from previous page)

```
        return $this->state;
    }
}
```

Ticket.php

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

namespace DesignPatterns\Behavioral\Memento;

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

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

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

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

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

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

    public function restoreFromMemento(Memento $memento)
    {
        $this->currentState = $memento->getState();
    }

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

Test

Tests/MementoTest.php

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

namespace DesignPatterns\Behavioral\Memento\Tests;

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

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

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

        $memento = $ticket->saveToMemento();

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

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

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

1.3.6 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

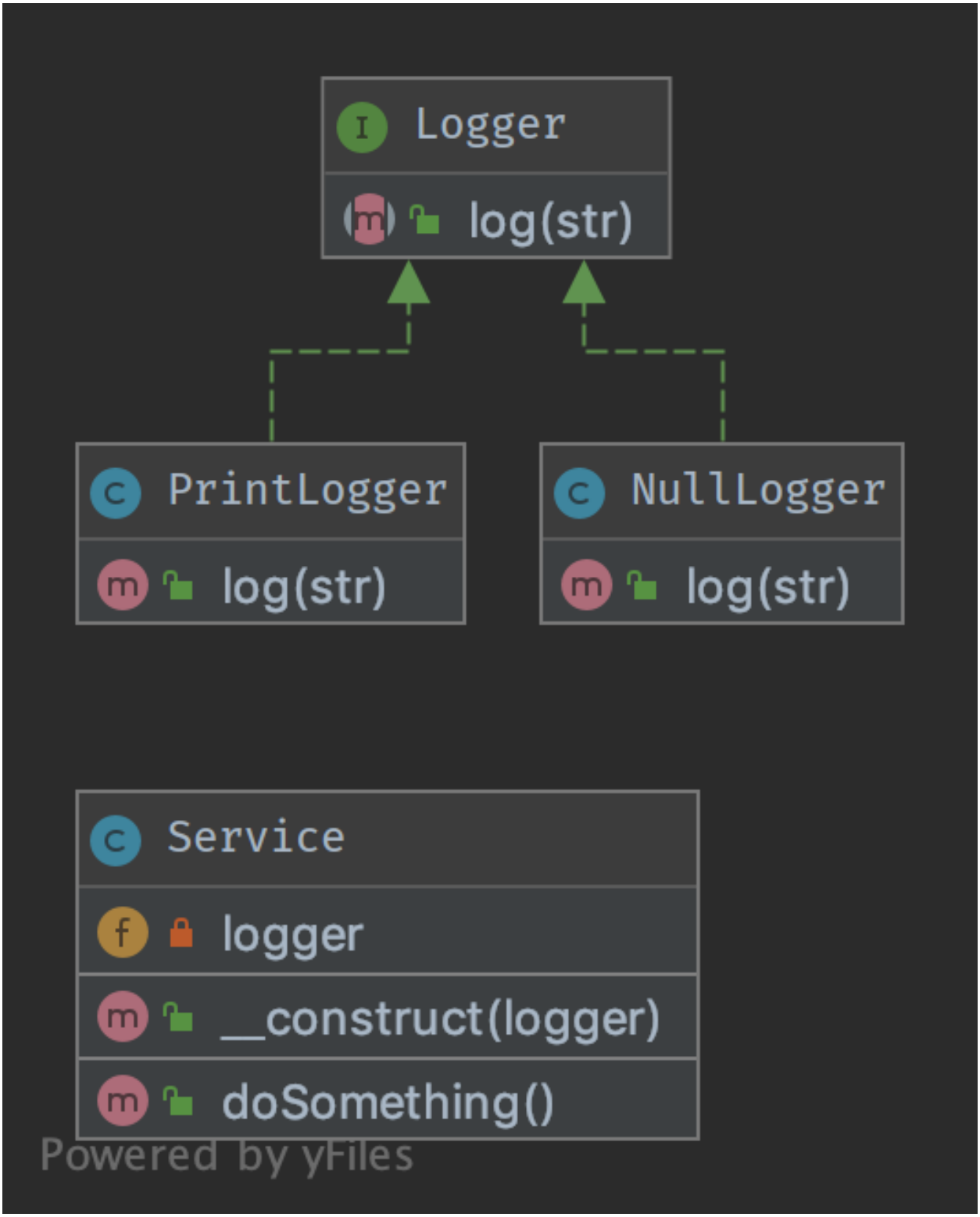
Methods that return an object or null should instead return an object or `NullObject`. `NullObjects` simplify boilerplate code such as `if (!is_null($obj)) { $obj->callSomething(); }` to just `$obj->callSomething();` by eliminating the conditional check in client code.

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


Logger
log(str)
PrintLogger
log(str)
NullLogger
log(str)
Service
logger
__construct(logger)
doSomething()
Powered by yFiles

Code

You can also find this code on GitHub

Service.php

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

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    private Logger $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }

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

Logger.php

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

namespace DesignPatterns\Behavioral\NullObject;

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

PrintLogger.php

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

namespace DesignPatterns\Behavioral\NullObject;

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

NullLogger.php

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

namespace DesignPatterns\Behavioral\NullObject;

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

Test

Tests/LoggerTest.php

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

namespace DesignPatterns\Behavioral\NullObject\Tests;

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

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

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

1.3.7 Observer

Purpose

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

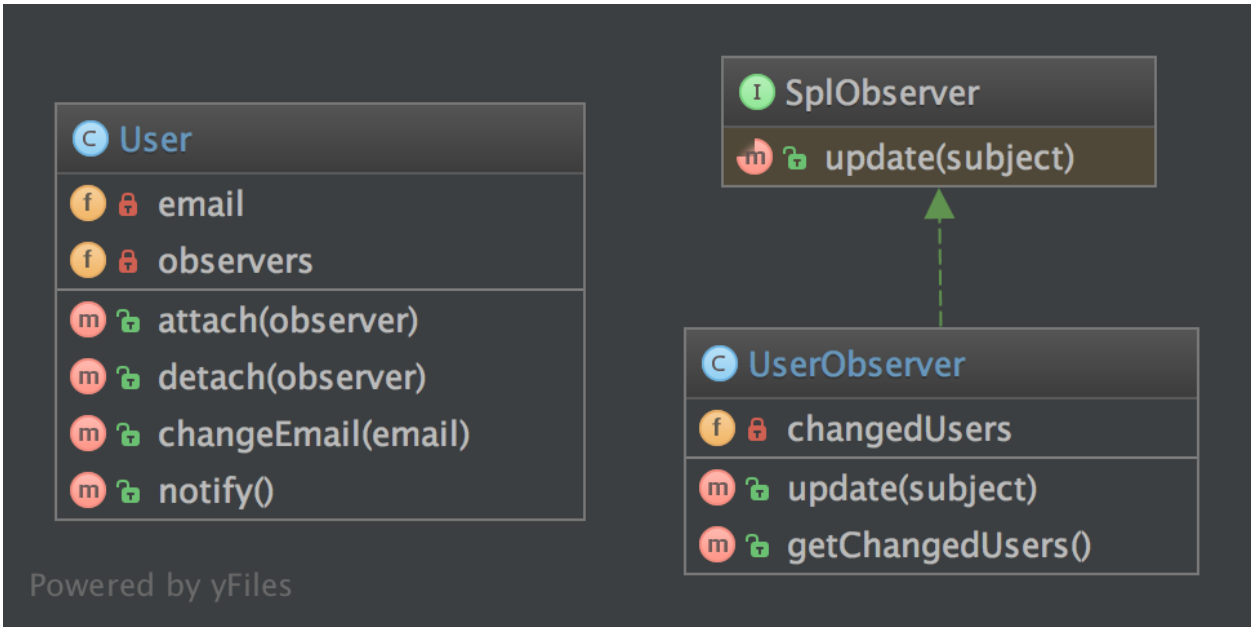
Examples

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

Note

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

UML Diagram



Code

You can also find this code on GitHub

User.php

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

namespace DesignPatterns\Behavioral\Observer;

use SplSubject;
use SplObjectStorage;
use SplObserver;

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

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

(continues on next page)

(continued from previous page)

```
    }

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

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

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

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

UserObserver.php

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

namespace DesignPatterns\Behavioral\Observer;

use SplObserver;
use SplSubject;

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

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

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

(continues on next page)

(continued from previous page)

```
    }
}
```

Test

Tests/ObserverTest.php

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

namespace DesignPatterns\Behavioral\Observer\Tests;

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

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

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

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

1.3.8 Specification

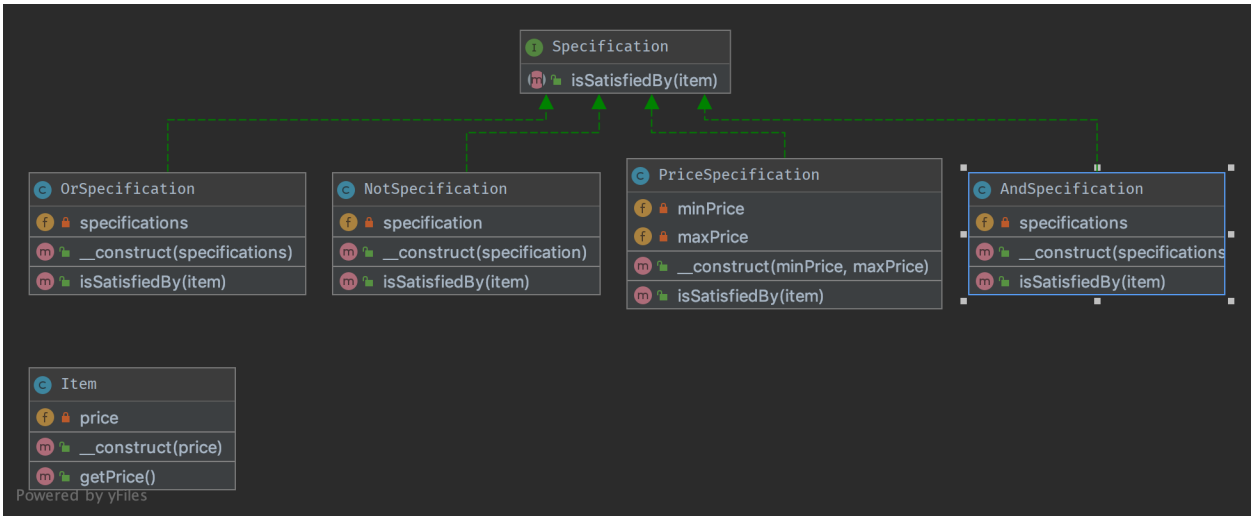
Purpose

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

Examples

- RulerZ

UML Diagram



Code

You can also find this code on GitHub

Item.php

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

namespace DesignPatterns\Behavioral\Specification;

class Item
{
    private float $price;

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

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

Specification.php

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

namespace DesignPatterns\Behavioral\Specification;

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

OrSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

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

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

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

        return false;
    }
}
```

PriceSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

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

    public function __construct(?float $minPrice, ?float $maxPrice)
    {
        $this->minPrice = $minPrice;
        $this->maxPrice = $maxPrice;
    }

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

```

(continues on next page)

(continued from previous page)

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

        return true;
    }
}
```

AndSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

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

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

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

        return true;
    }
}
```

NotSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

class NotSpecification implements Specification
{
    private Specification $specification;

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

(continues on next page)

(continued from previous page)

```
    }

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

Test

Tests/SpecificationTest.php

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

namespace DesignPatterns\Behavioral\Specification\Tests;

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

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

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

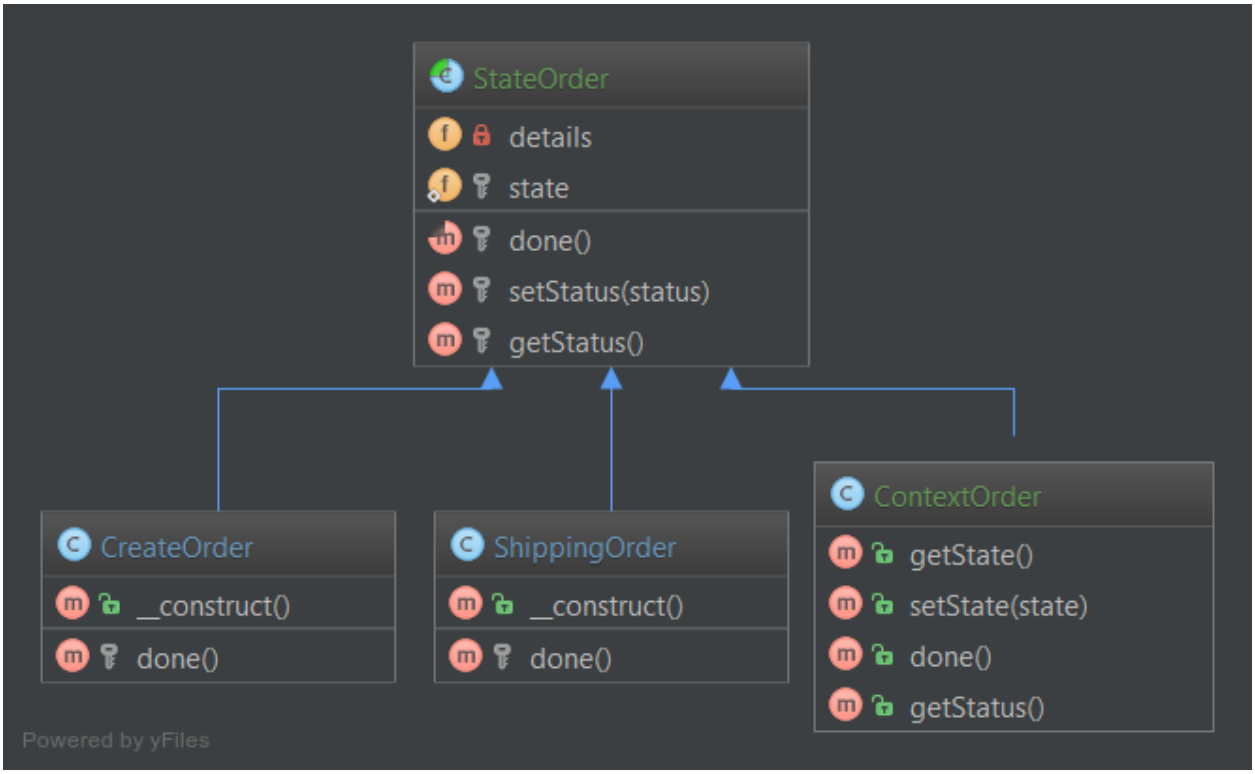
```
        $this->assertTrue($notSpec->isSatisfiedBy(new Item(150)));
        $this->assertFalse($notSpec->isSatisfiedBy(new Item(50)));
    }
}
```

1.3.9 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

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

namespace DesignPatterns\Behavioral\State;

class OrderContext
```

(continues on next page)

(continued from previous page)

```
{
    private State $state;

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

        return $order;
    }

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

    public function proceedToNext()
    {
        $this->state->proceedToNext($this);
    }

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

State.php

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

namespace DesignPatterns\Behavioral\State;

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

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

StateCreated.php

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

namespace DesignPatterns\Behavioral\State;

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

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

(continues on next page)

(continued from previous page)

```
}
```

StateShipped.php

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

namespace DesignPatterns\Behavioral\State;

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

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

StateDone.php

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

namespace DesignPatterns\Behavioral\State;

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

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

Test

Tests/StateTest.php

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

namespace DesignPatterns\Behavioral\State\Tests;

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

class StateTest extends TestCase
{
    public function testIsCreatedWithStateCreated()
    {
```

(continues on next page)

(continued from previous page)

```
12          $orderContext = OrderContext::create();
13
14          $this->assertSame('created', $orderContext->toString());
15      }
16
17      public function testCanProceedToStateShipped()
18      {
19          $contextOrder = OrderContext::create();
20          $contextOrder->proceedToNext();
21
22          $this->assertSame('shipped', $contextOrder->toString());
23      }
24
25      public function testCanProceedToStateDone()
26      {
27          $contextOrder = OrderContext::create();
28          $contextOrder->proceedToNext();
29          $contextOrder->proceedToNext();
30
31          $this->assertSame('done', $contextOrder->toString());
32      }
33
34      public function testStateDoneIsTheLastPossibleState()
35      {
36          $contextOrder = OrderContext::create();
37          $contextOrder->proceedToNext();
38          $contextOrder->proceedToNext();
39          $contextOrder->proceedToNext();
40
41          $this->assertSame('done', $contextOrder->toString());
42      }
43  }
```

1.3.10 Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy

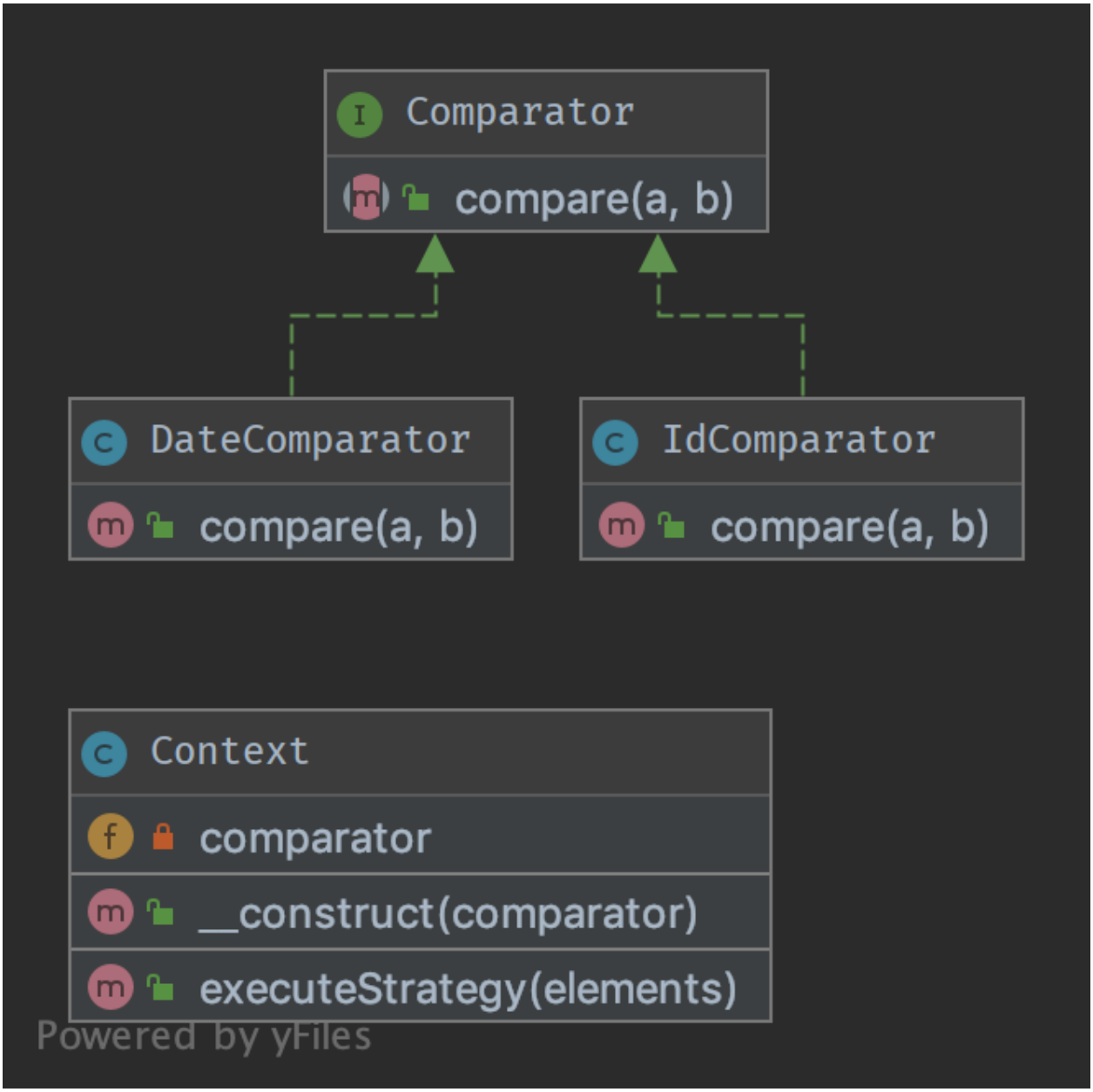
Purpose

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

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

namespace DesignPatterns\Behavioral\Strategy;

class Context
```

(continues on next page)

(continued from previous page)

```
{
    private Comparator $comparator;

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

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

        return $elements;
    }
}
```

Comparator.php

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

namespace DesignPatterns\Behavioral\Strategy;

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

DateComparator.php

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

namespace DesignPatterns\Behavioral\Strategy;

use DateTime;

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

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

IdComparator.php

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

namespace DesignPatterns\Behavioral\Strategy;
```

(continues on next page)

(continued from previous page)

```
class IdComparator implements Comparator
{
    public function compare($a, $b): int
    {
        return $a['id'] <=> $b['id'];
    }
}
```

Test

Tests/StrategyTest.php

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

namespace DesignPatterns\Behavioral\Strategy\Tests;

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

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

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

    /**
```

(continues on next page)

(continued from previous page)

```
41          * @dataProvider provideIntegers
42          *
43          * @param array $collection
44          * @param array $expected
45          */
46         public function testIdComparator($collection, $expected)
47         {
48             $obj = new Context(new IdComparator());
49             $elements = $obj->executeStrategy($collection);
50
51             $firstElement = array_shift($elements);
52             $this->assertSame($expected, $firstElement);
53         }
54
55         /**
56          * @dataProvider provideDates
57          *
58          * @param array $collection
59          * @param array $expected
60          */
61         public function testDateComparator($collection, $expected)
62         {
63             $obj = new Context(new DateComparator());
64             $elements = $obj->executeStrategy($collection);
65
66             $firstElement = array_shift($elements);
67             $this->assertSame($expected, $firstElement);
68         }
69     }
```

1.3.11 Template Method

Purpose

Template Method is a behavioral design pattern.

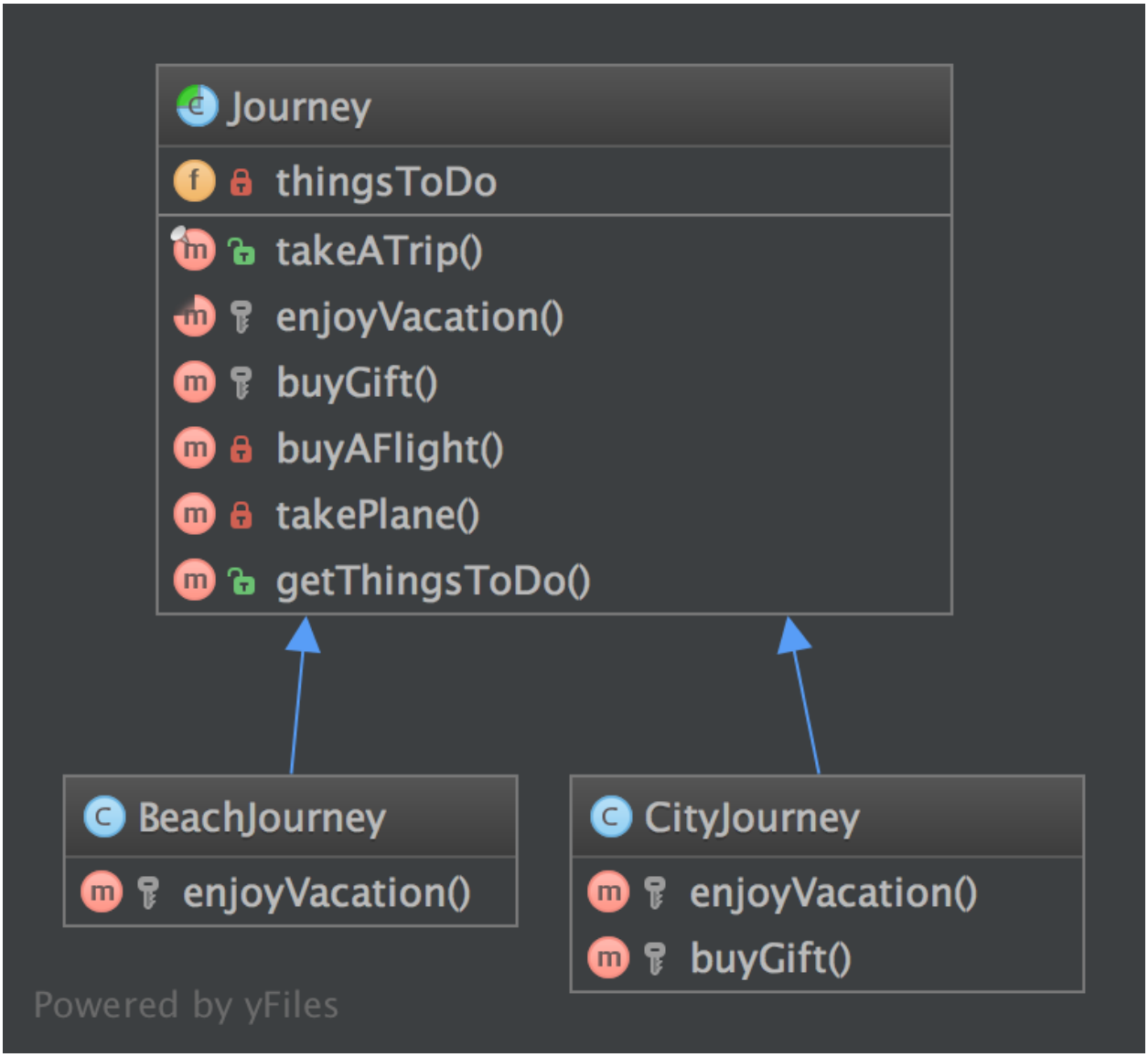
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template "finish" the behavior of an algorithm.

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

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

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

UML Diagram



Code

You can also find this code on GitHub

Journey.php

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

namespace DesignPatterns\Behavioral\TemplateMethod;

abstract class Journey
{
    /**
     * @var string[]
```

(continues on next page)

(continued from previous page)

```
     */
    private array $thingsToDo = [];

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

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

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

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

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

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

    private function takePlane(): string
    {
        return 'Taking the plane';
    }

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

BeachJourney.php

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

namespace DesignPatterns\Behavioral\TemplateMethod;

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

CityJourney.php

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

namespace DesignPatterns\Behavioral\TemplateMethod;

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

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

Test

Tests/JourneyTest.php

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

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

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

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

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

(continues on next page)

(continued from previous page)

```
21
22      public function testCanGetOnAJourneyToACity()
23      {
24          $cityJourney = new CityJourney();
25          $cityJourney->takeATrip();
26
27          $this->assertSame(
28              [
29                  'Buy a flight ticket',
30                  'Taking the plane',
31                  'Eat, drink, take photos and sleep',
32                  'Buy a gift',
33                  'Taking the plane'
34              ],
35              $cityJourney->getThingsToDo()
36          );
37      }
38  }
```

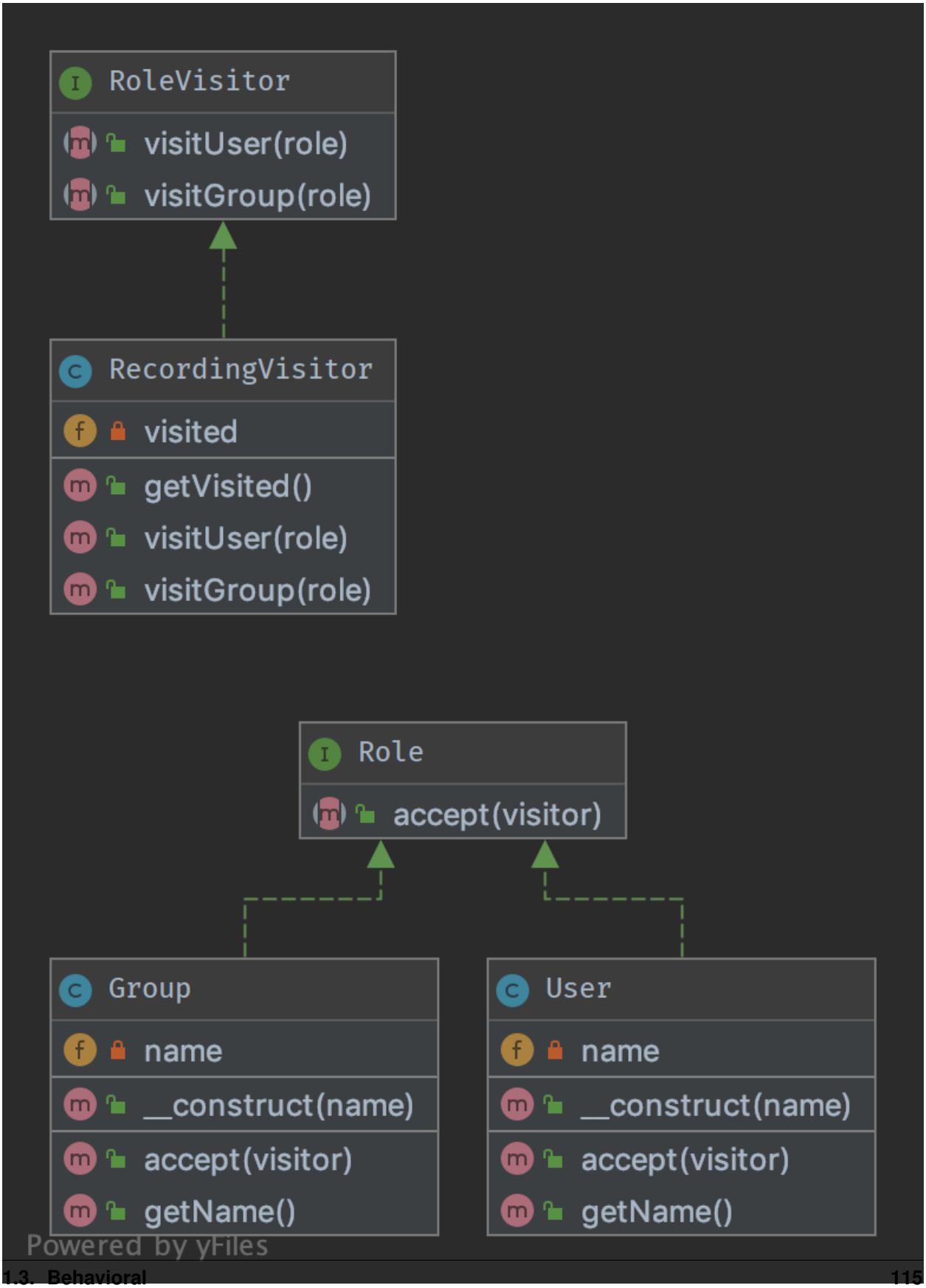
1.3.12 Visitor

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


RoleVisitor
visitUser(role)
visitGroup(role)
RecordingVisitor
visited
getVisited()
visitUser(role)
visitGroup(role)
Role
accept(visitor)
Group
name
__construct(name)
accept(visitor)
getName()
User
name
__construct(name)
accept(visitor)
getName()
Powered by yFiles

Code

You can also find this code on GitHub

RoleVisitor.php

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

namespace DesignPatterns\Behavioral\Visitor;

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

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

RecordingVisitor.php

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

namespace DesignPatterns\Behavioral\Visitor;

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

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

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

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

Role.php

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

namespace DesignPatterns\Behavioral\Visitor;
```

(continues on next page)

(continued from previous page)

```
4
5  interface Role
6  {
7      public function accept(RoleVisitor $visitor);
8  }
```

User.php

```
1  <?php declare(strict_types=1);
2
3  namespace DesignPatterns\Behavioral\Visitor;
4
5  class User implements Role
6  {
7      private string $name;
8
9      public function __construct(string $name)
10     {
11         $this->name = $name;
12     }
13
14     public function getName(): string
15     {
16         return sprintf('User %s', $this->name);
17     }
18
19     public function accept(RoleVisitor $visitor)
20     {
21         $visitor->visitUser($this);
22     }
23 }
```

Group.php

```
1  <?php declare(strict_types=1);
2
3  namespace DesignPatterns\Behavioral\Visitor;
4
5  class Group implements Role
6  {
7      private string $name;
8
9      public function __construct(string $name)
10     {
11         $this->name = $name;
12     }
13
14     public function getName(): string
15     {
16         return sprintf('Group: %s', $this->name);
17     }
18
19     public function accept(RoleVisitor $visitor)
20     {
21         $visitor->visitGroup($this);
22     }
23 }
```

Test

Tests/VisitorTest.php

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

namespace DesignPatterns\Tests\Visitor\Tests;

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

class VisitorTest extends TestCase
{
    private RecordingVisitor $visitor;

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

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

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

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.

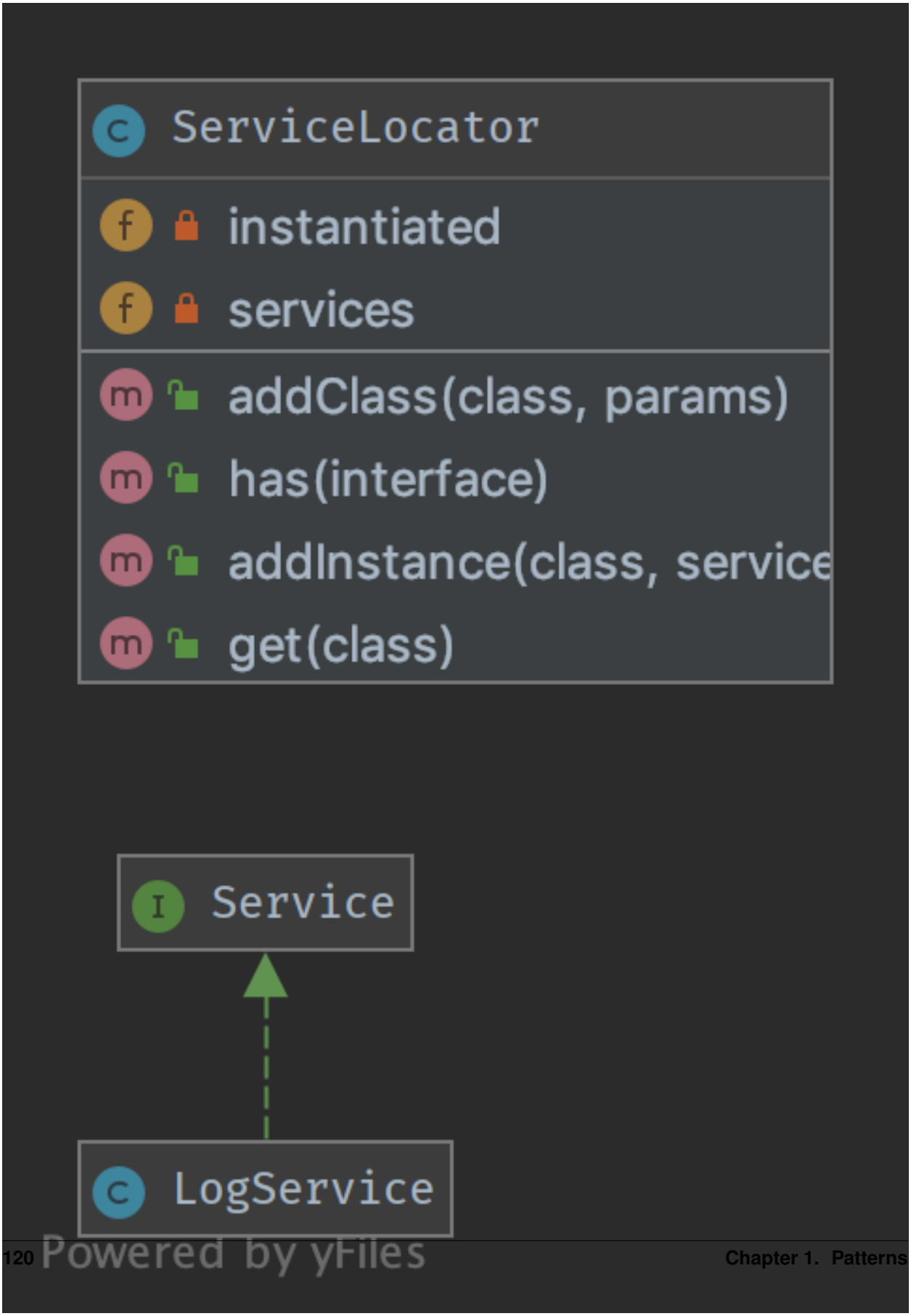
Purpose

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


ServiceLocator
instantiated
services
addClass(class, params)
has(interface)
addInstance(class, service
get(class)
Service
LogService
Powered by yFiles

Code

You can also find this code on GitHub

Service.php

```php
<?php

namespace DesignPatterns\More\ServiceLocator;

interface Service
{

}
```

ServiceLocator.php

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

namespace DesignPatterns\More\ServiceLocator;

use OutOfRangeException;
use InvalidArgumentException;

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

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

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

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

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

    public function get(string $class): Service
    {
        if (isset($this->instantiated[$class])) {
            return $this->instantiated[$class];
        }

```

(continues on next page)

(continued from previous page)

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

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

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

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

        return $object;
    }
}
```

LogService.php

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

namespace DesignPatterns\More\ServiceLocator;

class LogService implements Service
{

}
```

Test

Tests/ServiceLocatorTest.php

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

namespace DesignPatterns\More\ServiceLocator\Tests;

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

class ServiceLocatorTest extends TestCase
{
```

(continues on next page)

(continued from previous page)

```
    private ServiceLocator $serviceLocator;

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

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

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

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

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

1.4.2 Repository

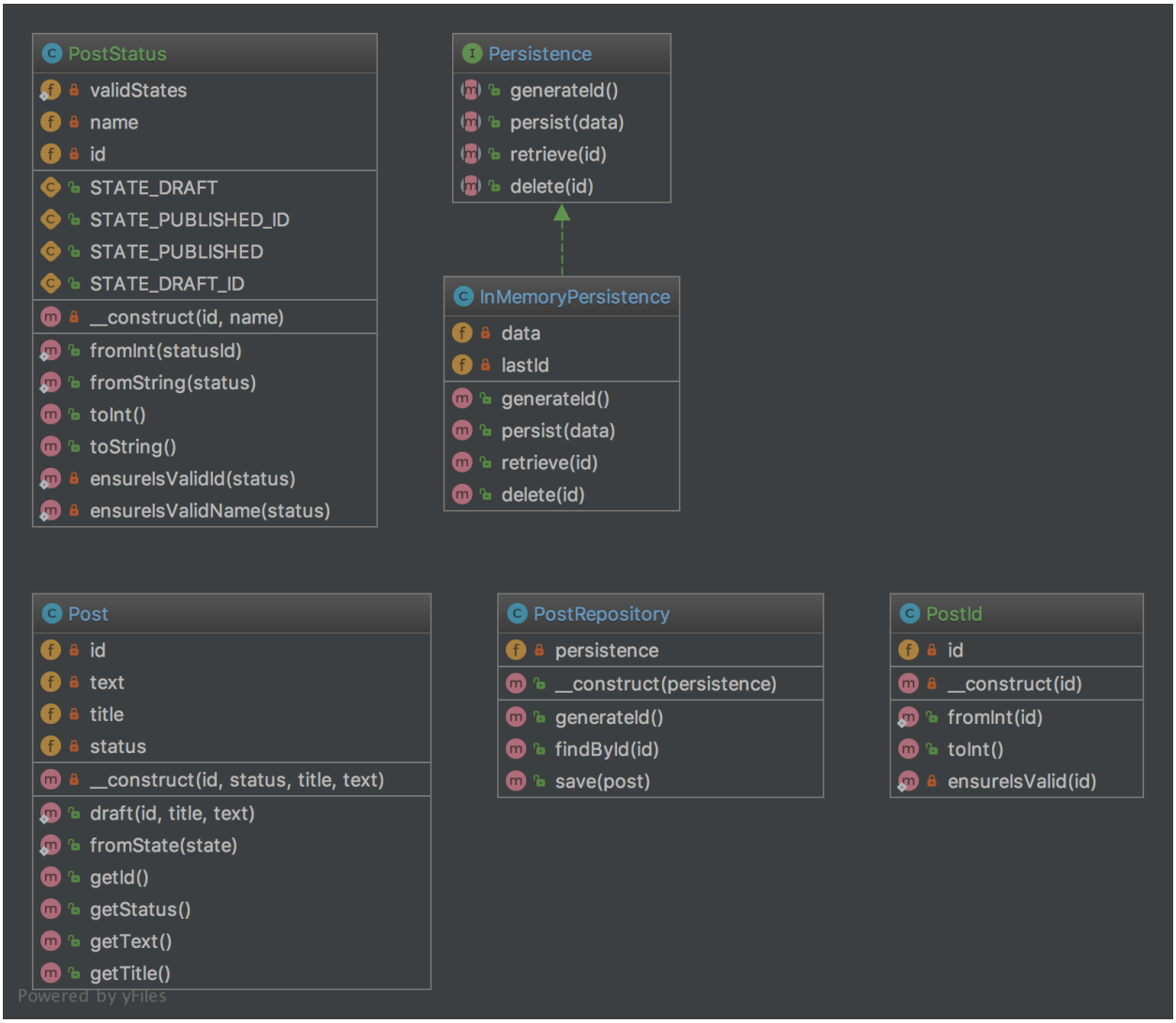
Purpose

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

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

namespace DesignPatterns\More\Repository\Domain;

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

PostId.php

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

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;
```

(continues on next page)

(continued from previous page)

```php

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

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

        return new self($id);
    }

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

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

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

PostStatus.php

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

namespace DesignPatterns\More\Repository\Domain;

use InvalidArgumentException;

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

(continues on next page)

(continued from previous page)

```
14      const STATE_PUBLISHED_ID = 2;
15
16      const STATE_DRAFT = 'draft';
17      const STATE_PUBLISHED = 'published';
18
19      private static array $validStates = [
20          self::STATE_DRAFT_ID => self::STATE_DRAFT,
21          self::STATE_PUBLISHED_ID => self::STATE_PUBLISHED,
22      ];
23
24      private int $id;
25      private string $name;
26
27      public static function fromInt(int $statusId)
28      {
29          self::ensureIsValidId($statusId);
30
31          return new self($statusId, self::$validStates[$statusId]);
32      }
33
34      public static function fromString(string $status)
35      {
36          self::ensureIsValidName($status);
37          $state = array_search($status, self::$validStates);
38
39          if ($state === false) {
40              throw new InvalidArgumentException('Invalid state given!');
41          }
42
43          return new self($state, $status);
44      }
45
46      private function __construct(int $id, string $name)
47      {
48          $this->id = $id;
49          $this->name = $name;
50      }
51
52      public function toInt(): int
53      {
54          return $this->id;
55      }
56
57      /**
58       * there is a reason that I avoid using __toString() as it operates outside of
    ↪the stack in PHP
59       * and is therefor not able to operate well with exceptions
60       */
61      public function toString(): string
62      {
63          return $this->name;
64      }
65
66      private static function ensureIsValidId(int $status)
67      {
68          if (!in_array($status, array_keys(self::$validStates), true)) {
69              throw new InvalidArgumentException('Invalid status id given');
```

(continues on next page)

(continued from previous page)

```
        }
    }


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

PostRepository.php

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

namespace DesignPatterns\More\Repository;

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

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

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

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

    public function findById(PostId $id): Post
    {
        try {
            $arrayData = $this->persistence->retrieve($id->toInt());
        } catch (OutOfBoundsException $e) {
            throw new OutOfBoundsException(sprintf('Post with id %d does not exist',
↪$id->toInt()), 0, $e);
        }

```

(continues on next page)

(continued from previous page)

```
        return Post::fromState($arrayData);
    }

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

Persistence.php

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

namespace DesignPatterns\More\Repository;

interface Persistence
{
    public function generateId(): int;

    public function persist(array $data);

    public function retrieve(int $id): array;

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

InMemoryPersistence.php

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

namespace DesignPatterns\More\Repository;

use OutOfBoundsException;

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

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

        return $this->lastId;
    }

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

    public function retrieve(int $id): array
```

(continues on next page)

(continued from previous page)

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

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

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

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

Test

Tests/PostRepositoryTest.php

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

namespace DesignPatterns\More\Repository\Tests;

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

class PostRepositoryTest extends TestCase
{
    private PostRepository $repository;

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

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

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

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

(continues on next page)

(continued from previous page)

```
34
35      public function testCanPersistPostDraft()
36      {
37          $postId = $this->repository->generateId();
38          $post = Post::draft($postId, 'Repository Pattern', 'Design Patterns PHP');
39          $this->repository->save($post);
40
41          $this->repository->findById($postId);
42
43          $this->assertEquals($postId, $this->repository->findById($postId)->getId());
44          $this->assertEquals(PostStatus::STATE_DRAFT, $post->getStatus()->toString());
45      }
46  }
```

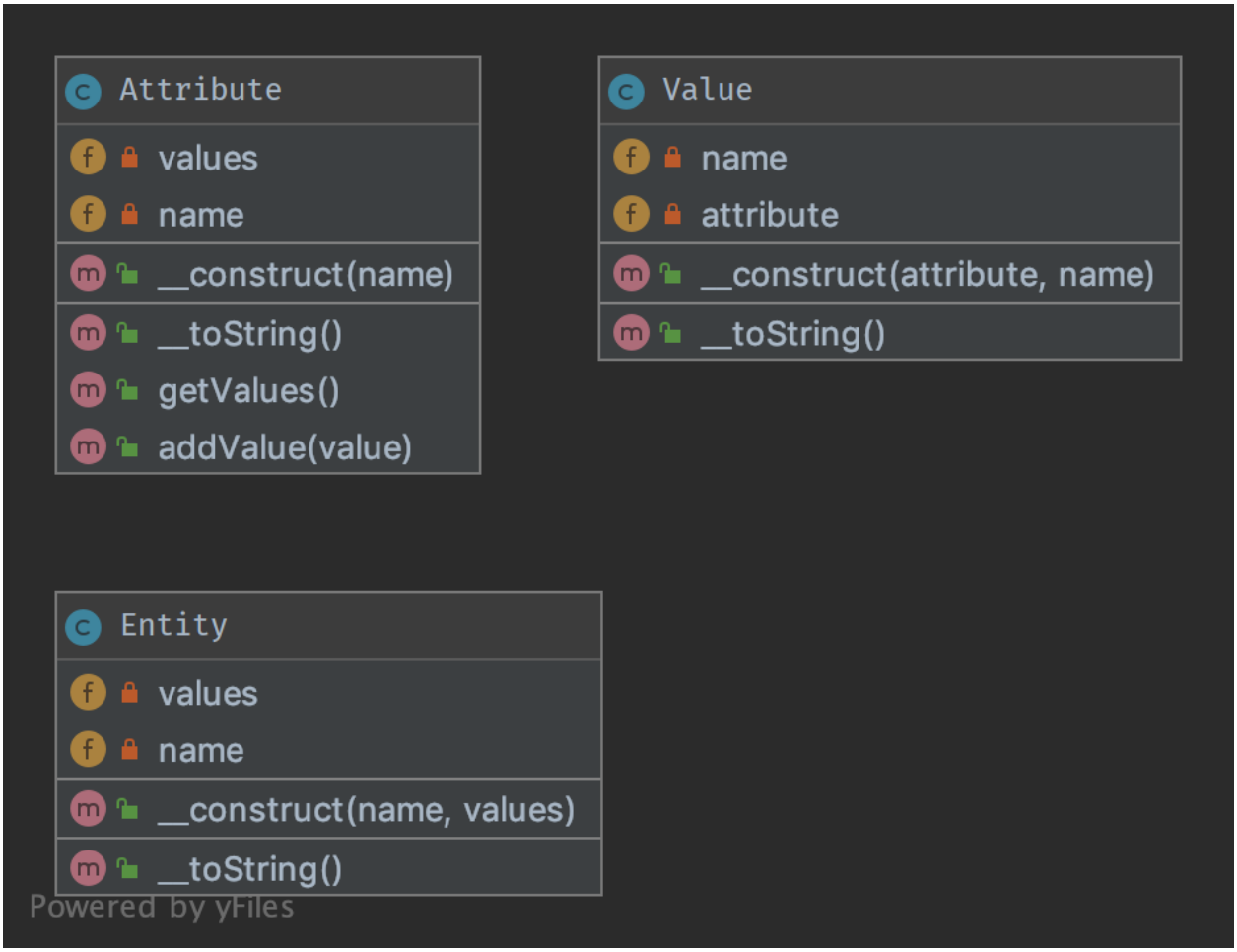
1.4.3 Entity-Attribute-Value (EAV)

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

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

namespace DesignPatterns\More\EAV;

use SplObjectStorage;

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

    /**
     * @var string
```

(continues on next page)

(continued from previous page)

```
     */
    private string $name;

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

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

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

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

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

Attribute.php

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

namespace DesignPatterns\More\EAV;

use SplObjectStorage;

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

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

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

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

(continues on next page)

(continued from previous page)

```
    }

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

Value.php

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

namespace DesignPatterns\More\EAV;

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

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

        $attribute->addValue($this);
    }

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

Test

Tests/EAVTest.php

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

namespace DesignPatterns\More\EAV\Tests;

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

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

        $memoryAttribute = new Attribute('memory');
```

(continues on next page)

(continued from previous page)

```
19          $memory8Gb = new Value($memoryAttribute, '8GB');
20
21          $entity = new Entity('MacBook Pro', [$colorSilver, $colorBlack, $memory8Gb]);
22
23          $this->assertEquals('MacBook Pro, color: silver, color: black, memory: 8GB',␣
    ↪(string) $entity);
24      }
25  }
```