DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

Aug 19, 2019

Contents

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

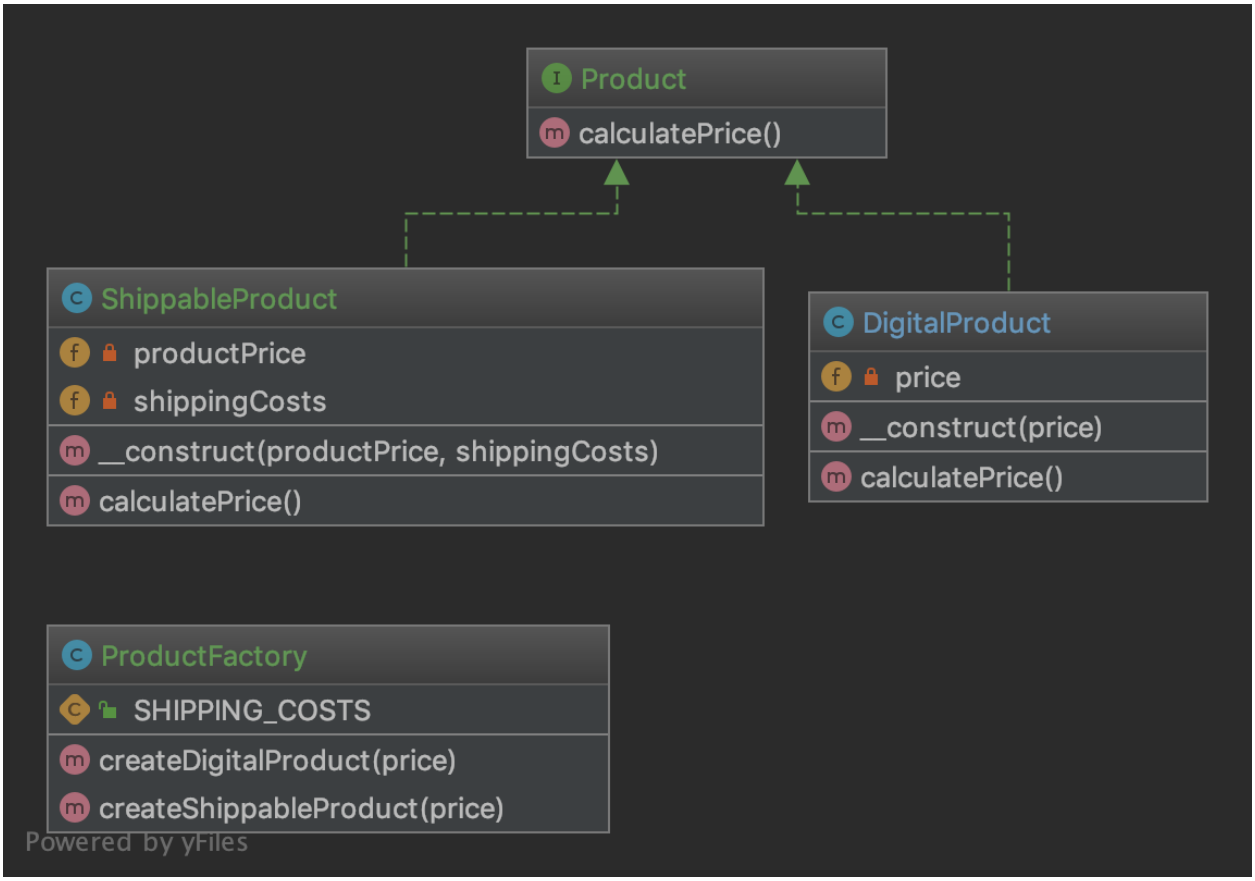
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns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



Code

You can also find this code on GitHub

Product.php

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

namespace DesignPatterns\Creational\AbstractFactory;

interface Product
{
    public function calculatePrice(): int;
}
```

ShippableProduct.php

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

namespace DesignPatterns\Creational\AbstractFactory;

class ShippableProduct implements Product
```

(continues on next page)

(continued from previous page)

```
{
    /**
     * @var float
     */
    private $productPrice;

    /**
     * @var float
     */
    private $shippingCosts;

    public function __construct(int $productPrice, int $shippingCosts)
    {
        $this->productPrice = $productPrice;
        $this->shippingCosts = $shippingCosts;
    }

    public function calculatePrice(): int
    {
        return $this->productPrice + $this->shippingCosts;
    }
}
```

DigitalProduct.php

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

namespace DesignPatterns\Creational\AbstractFactory;

class DigitalProduct implements Product
{
    /**
     * @var int
     */
    private $price;

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

    public function calculatePrice(): int
    {
        return $this->price;
    }
}
```

ProductFactory.php

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

namespace DesignPatterns\Creational\AbstractFactory;

class ProductFactory
{
```

(continues on next page)

(continued from previous page)

```
    const SHIPPING_COSTS = 50;

    public function createShippableProduct(int $price): Product
    {
        return new ShippableProduct($price, self::SHIPPING_COSTS);
    }

    public function createDigitalProduct(int $price): Product
    {
        return new DigitalProduct($price);
    }
}
```

Test

Tests/AbstractFactoryTest.php

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

namespace DesignPatterns\Creational\AbstractFactory\Tests;

use DesignPatterns\Creational\AbstractFactory\DigitalProduct;
use DesignPatterns\Creational\AbstractFactory\ProductFactory;
use DesignPatterns\Creational\AbstractFactory\ShippableProduct;
use PHPUnit\Framework\TestCase;

class AbstractFactoryTest extends TestCase
{
    public function testCanCreateDigitalProduct()
    {
        $factory = new ProductFactory();
        $product = $factory->createDigitalProduct(150);

        $this->assertInstanceOf(DigitalProduct::class, $product);
    }

    public function testCanCreateShippableProduct()
    {
        $factory = new ProductFactory();
        $product = $factory->createShippableProduct(150);

        $this->assertInstanceOf(ShippableProduct::class, $product);
    }

    public function testCanCalculatePriceForDigitalProduct()
    {
        $factory = new ProductFactory();
        $product = $factory->createDigitalProduct(150);

        $this->assertEquals(150, $product->calculatePrice());
    }

    public function testCanCalculatePriceForShippableProduct()
    {
```

(continues on next page)

(continued from previous page)

```
39          $factory = new ProductFactory();
40          $product = $factory->createShippableProduct(150);
41
42          $this->assertEquals(200, $product->calculatePrice());
43      }
44  }
```

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).

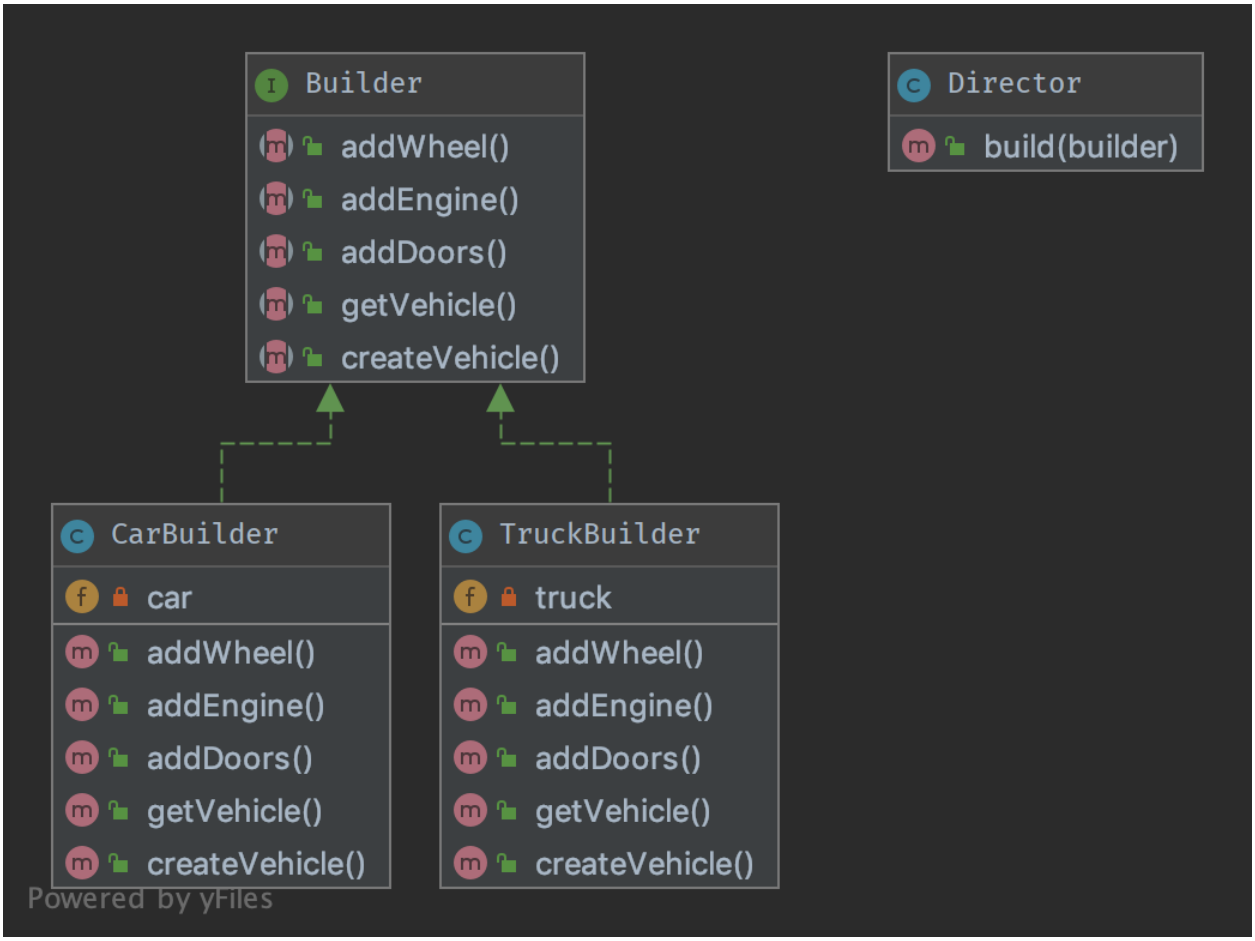
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

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\Vehicle;

class TruckBuilder implements Builder
{
    /**
     * @var Parts\Truck
     */
    private $truck;

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

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

(continues on next page)

(continued from previous page)

```
    }

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

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

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

CarBuilder.php

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

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

class CarBuilder implements Builder
{
    /**
     * @var Parts\Car
     */
    private $car;

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

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

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

(continues on next page)

(continued from previous page)

```
    }

    public function createVehicle()
    {
        $this->car = new Parts\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 $data = [];

    /**
     * @param string $key
     * @param object $value
     */
    public function setPart($key, $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
```

(continues on next page)

(continued from previous page)

```
{
}
```

Parts/Engine.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Engine
{
}
```

Parts/Wheel.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Wheel
{
}
```

Parts/Door.php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

Test

Tests/DirectorTest.php

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

namespace DesignPatterns\Creational\Builder\Tests;

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

class DirectorTest extends TestCase
{
    public function testCanBuildTruck()
```

(continues on next page)

(continued from previous page)

```
16      {
17          $truckBuilder = new TruckBuilder();
18          $newVehicle = (new Director())->build($truckBuilder);
19
20          $this->assertInstanceOf(Truck::class, $newVehicle);
21      }
22
23      public function testCanBuildCar()
24      {
25          $carBuilder = new CarBuilder();
26          $newVehicle = (new Director())->build($carBuilder);
27
28          $this->assertInstanceOf(Car::class, $newVehicle);
29      }
30  }
```

1.1.3 Factory Method

Purpose

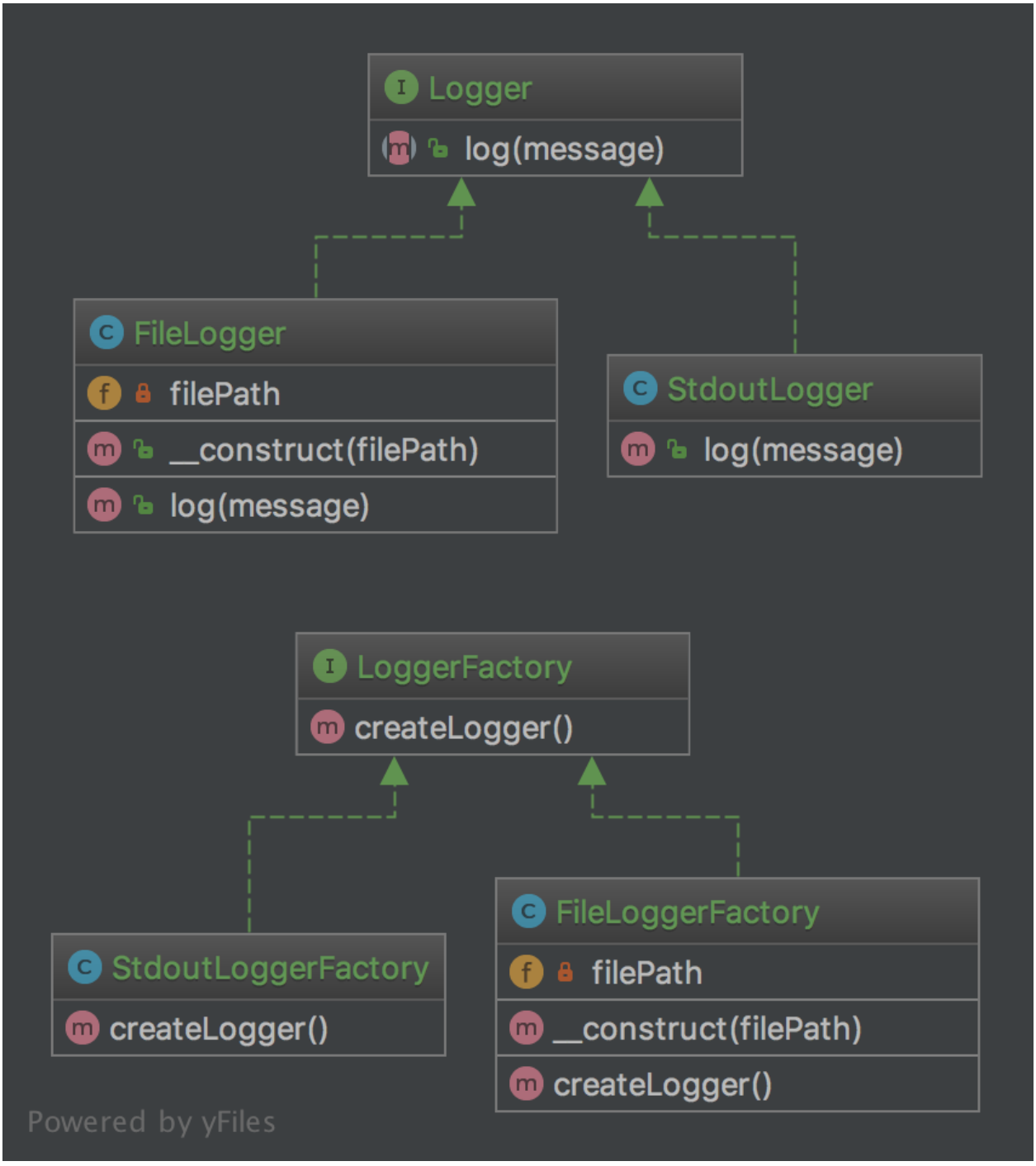
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects.

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

```

(continues on next page)

(continued from previous page)

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

StdoutLoggerFactory.php

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

namespace DesignPatterns\Creational\FactoryMethod;

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

FileLoggerFactory.php

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

namespace DesignPatterns\Creational\FactoryMethod;

class FileLoggerFactory implements LoggerFactory
{
    /**
     * @var string
     */
    private $filePath;

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

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

Test

Tests/FactoryMethodTest.php

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

namespace DesignPatterns\Creational\FactoryMethod\Tests;

use DesignPatterns\Creational\FactoryMethod\FileLogger;
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

1.1.4 Multiton

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

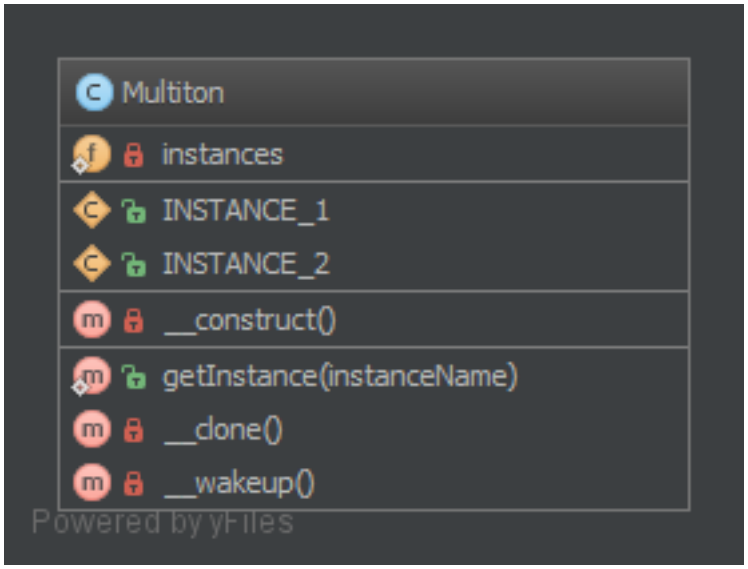
Purpose

To have only a list of named instances that are used, like a singleton but with n instances.

Examples

- 2 DB Connectors, e.g. one for MySQL, the other for SQLite
- multiple Loggers (one for debug messages, one for errors)

UML Diagram



Code

You can also find this code on GitHub

Multiton.php

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

namespace DesignPatterns\Creational\Multiton;

final class Multiton
{
    const INSTANCE_1 = '1';
    const INSTANCE_2 = '2';

    /**
     * @var Multiton[]
     */
    private static $instances = [];

    /**
     * this is private to prevent from creating arbitrary instances
     */
    private function __construct()
    {
    }

    public static function getInstance(string $instanceName): Multiton
    {
        if (!isset(self::$instances[$instanceName])) {
            self::$instances[$instanceName] = new self();
        }

        return self::$instances[$instanceName];
```

(continues on next page)

(continued from previous page)

```
30      }
31
32      /**
33       * prevent instance from being cloned
34       */
35      private function __clone()
36      {
37      }
38
39      /**
40       * prevent instance from being unserialized
41       */
42      private function __wakeup()
43      {
44      }
45  }
```

Test

1.1.5 Pool

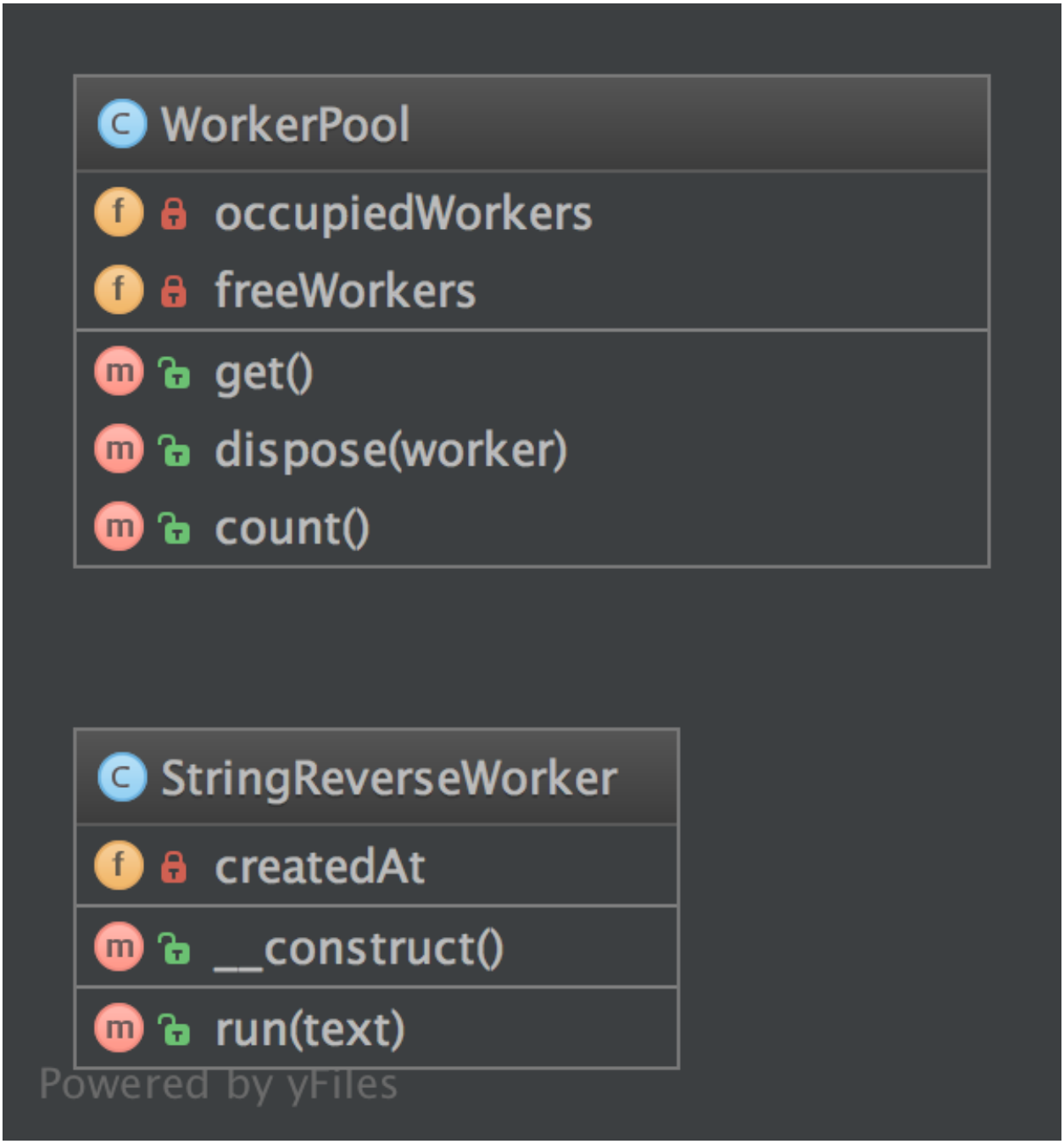
Purpose

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;

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

    /**
     * @var StringReverseWorker[]
     */
    private $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;

class StringReverseWorker
{
    /**
     * @var \DateTime
     */
    private $createdAt;
```

(continues on next page)

(continued from previous page)

```
12
13      public function __construct()
14      {
15          $this->createdAt = new \DateTime();
16      }
17
18      public function run(string $text)
19      {
20          return strrev($text);
21      }
22  }
```

Test

Tests/PoolTest.php

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

1.1.6 Prototype

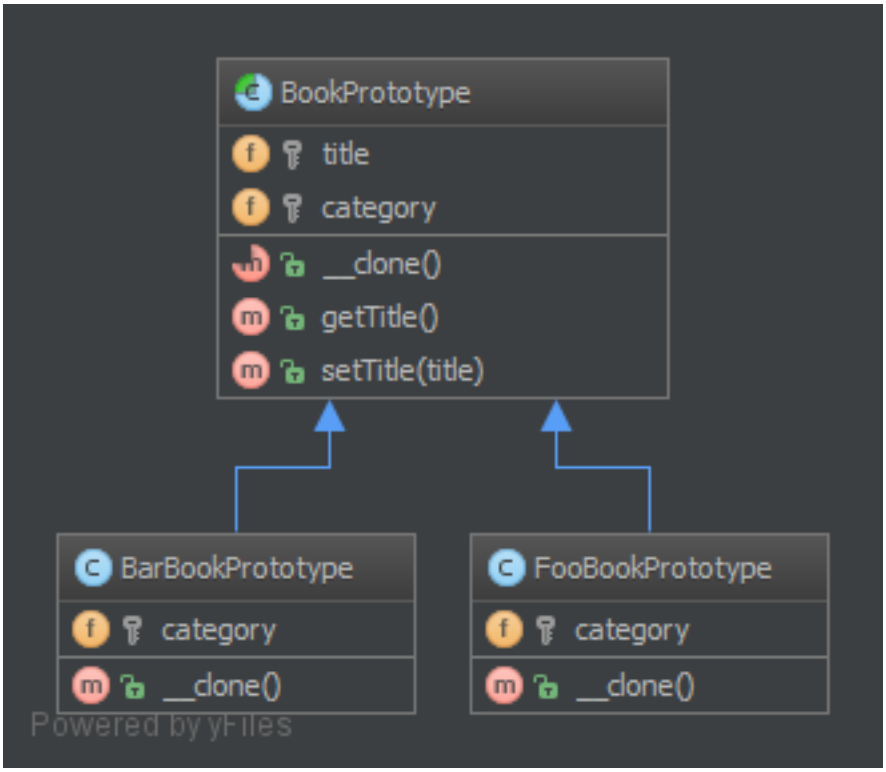
Purpose

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
{
    /**
     * @var string
     */
    protected $title;

    /**
     * @var string
     */
    protected $category;
```

(continues on next page)

(continued from previous page)

```php

    abstract public function __clone();

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

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

BarBookPrototype.php

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

namespace DesignPatterns\Creational\Prototype;

class BarBookPrototype extends BookPrototype
{
    /**
     * @var string
     */
    protected $category = 'Bar';

    public function __clone()
    {
    }
}
```

FooBookPrototype.php

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

namespace DesignPatterns\Creational\Prototype;

class FooBookPrototype extends BookPrototype
{
    /**
     * @var string
     */
    protected $category = 'Foo';

    public function __clone()
    {
    }
}
```

Test

Tests/PrototypeTest.php

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

namespace DesignPatterns\Creational\Prototype\Tests;

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

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

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

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

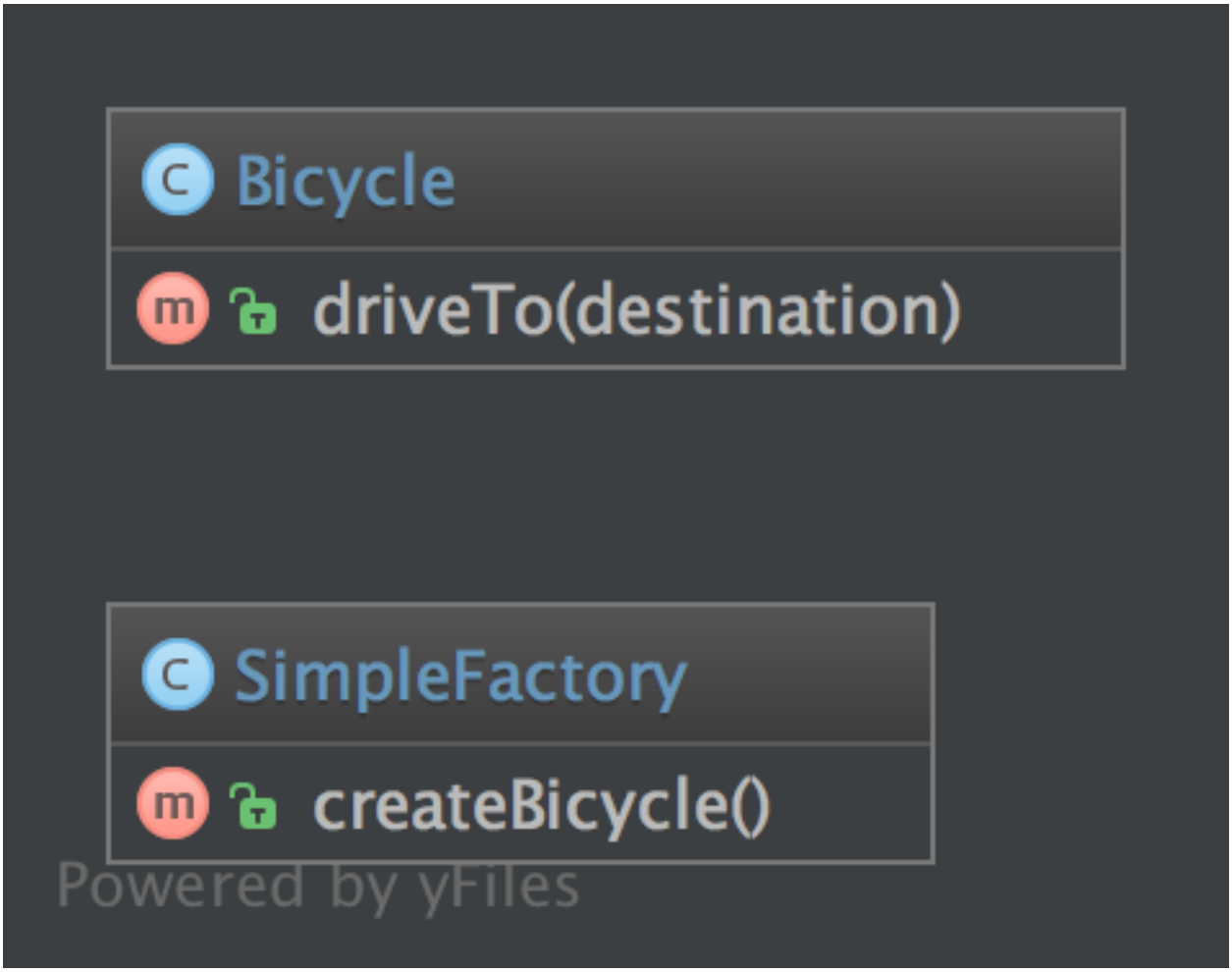
1.1.7 Simple Factory

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);
    }
}
```

1.1.8 Singleton

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

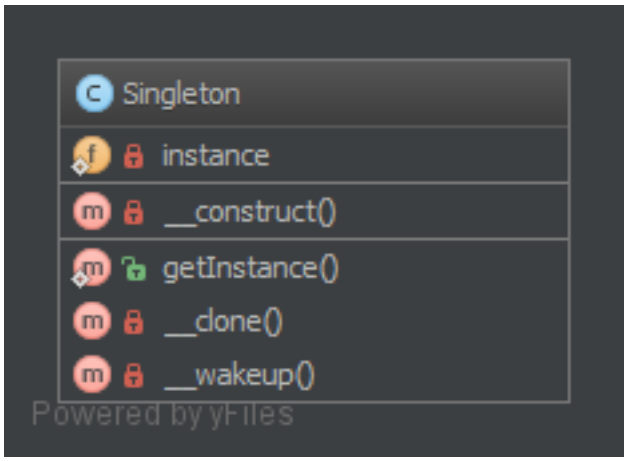
Purpose

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

Examples

- DB Connector
- Logger (may also be a Multiton if there are many log files for several purposes)
- 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
{
    /**
     * @var Singleton
     */
    private static $instance;

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

        return static::$instance;
    }

```

(continues on next page)

(continued from previous page)

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

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

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

Test

Tests/SingletonTest.php

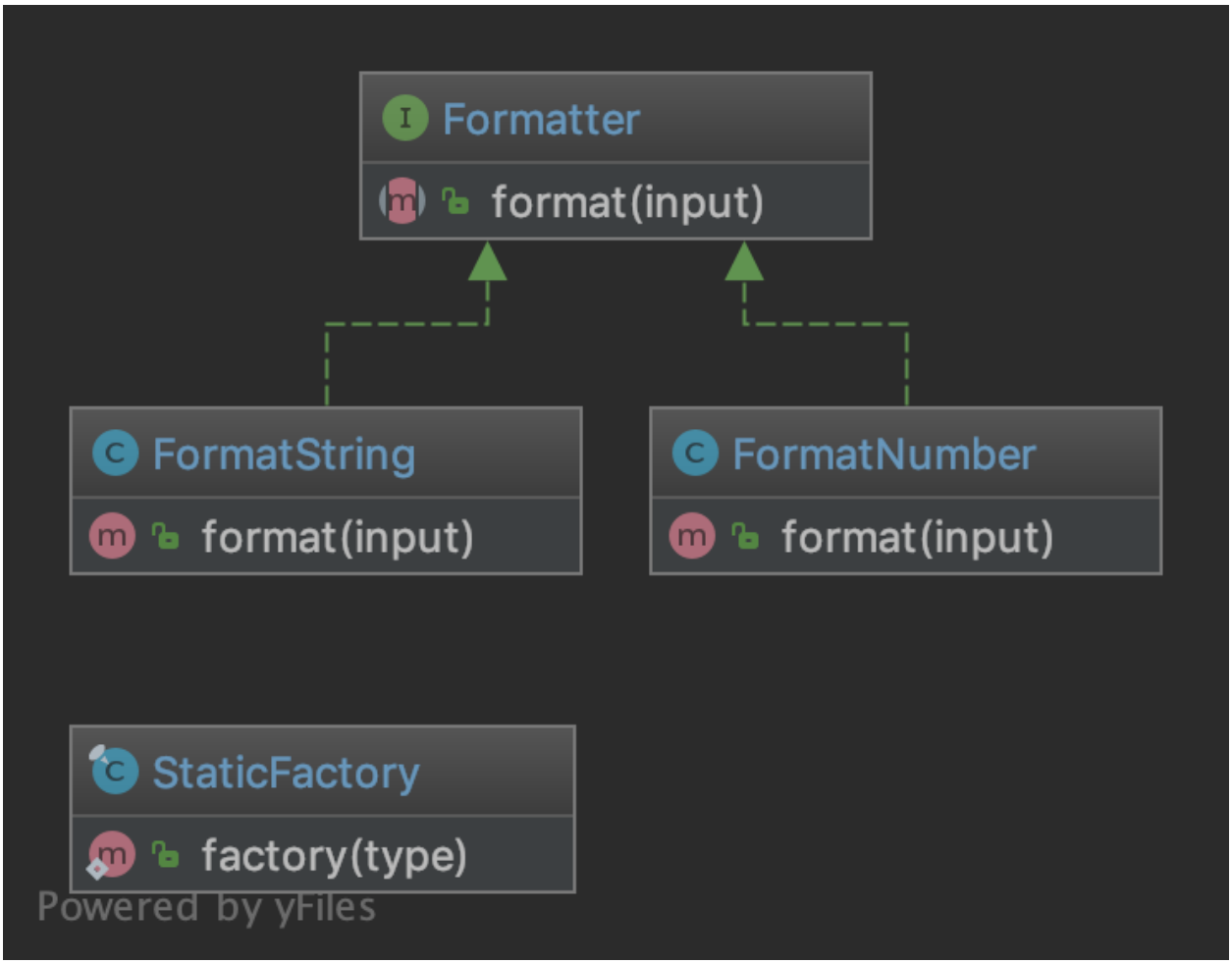
```
<?php
declare(strict_types=1);

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);
    }
}
```

1.1.9 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;

/**
 * Note1: Remember, static means global state which is evil because it can't be
 ↪mocked for tests
```

(continues on next page)

(continued from previous page)

```
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
final class StaticFactory
{
    /**
     * @param string $type
     *
     * @return Formatter
     */
    public static function factory(string $type): Formatter
    {
        if ($type == 'number') {
            return new FormatNumber();
        } elseif ($type == 'string') {
            return new FormatString();
        }

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

Formatter.php

```
<?php
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
{
```

(continues on next page)

(continued from previous page)

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

Test

Tests/StaticFactoryTest.php

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

namespace DesignPatterns\Creational\StaticFactory\Tests;

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

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

    public function testCanCreateStringFormatter()
    {
        $this->assertInstanceOf(
            'DesignPatterns\Creational\StaticFactory\FormatString',
            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

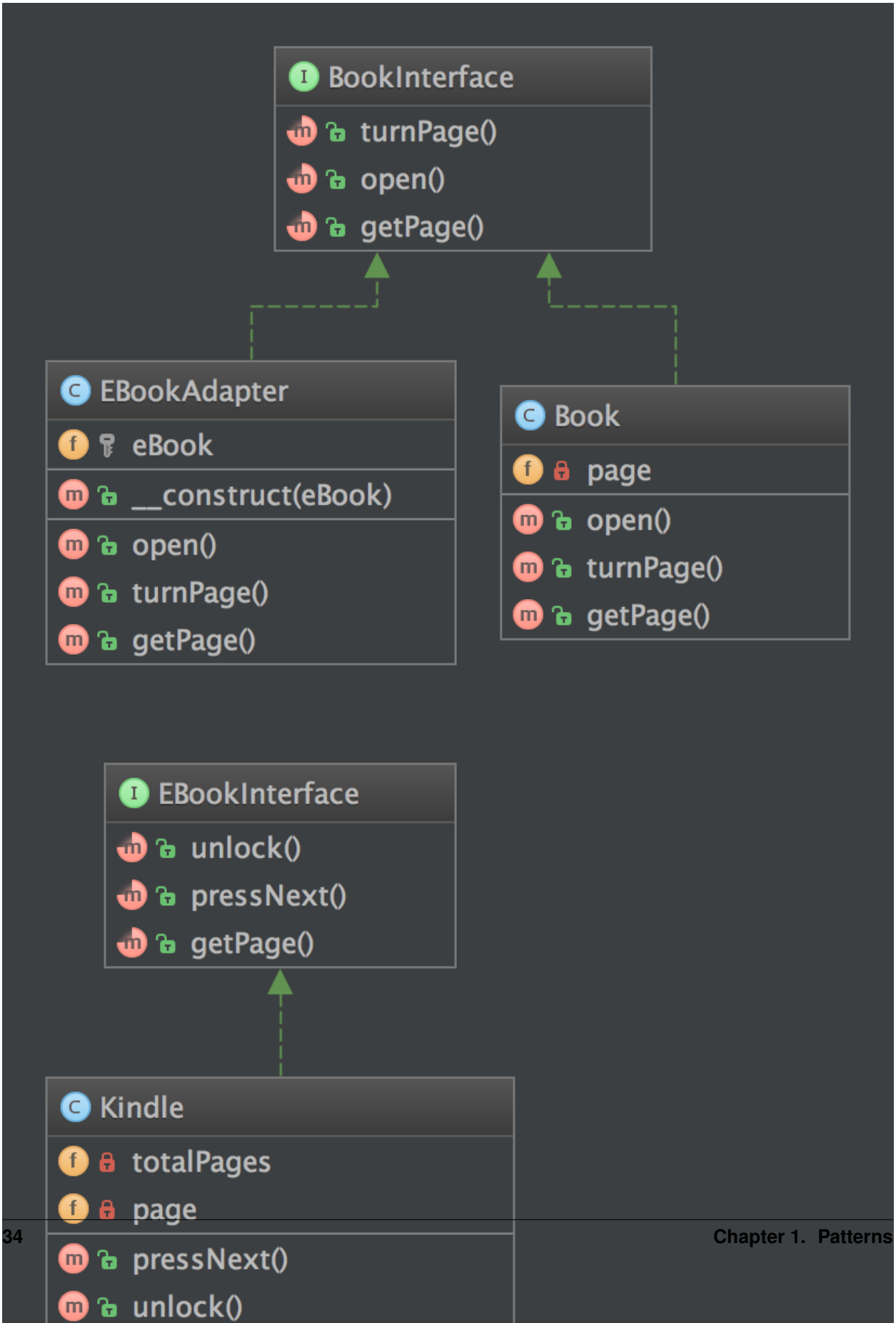
Purpose

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


BookInterface
turnPage()
open()
getPage()
EBookAdapter
eBook
__construct(eBook)
open()
turnPage()
getPage()
Book
page
open()
turnPage()
getPage()
EBookInterface
unlock()
pressNext()
getPage()
Kindle
totalPages
page
pressNext()
unlock()

Code

You can also find this code on GitHub

BookInterface.php

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

namespace DesignPatterns\Structural\Adapter;

interface BookInterface
{
    public function turnPage();

    public function open();

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

Book.php

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

namespace DesignPatterns\Structural\Adapter;

class Book implements BookInterface
{
    /**
     * @var int
     */
    private $page;

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

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

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

EBookAdapter.php

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

namespace DesignPatterns\Structural\Adapter;

/**
```

(continues on next page)

(continued from previous page)

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

    /**
     * @param EBookInterface $eBook
     */
    public function __construct(EBookInterface $eBook)
    {
        $this->eBook = $eBook;
    }

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

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

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

EBookInterface.php

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

namespace DesignPatterns\Structural\Adapter;

interface EBookInterface
{
    public function unlock();

    public function pressNext();

```

(continues on next page)

(continued from previous page)

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

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 EBookInterface
{
    /**
     * @var int
     */
    private $page = 1;

    /**
     * @var int
     */
    private $totalPages = 100;

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

    public function unlock()
    {
    }

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

Test

Tests/AdapterTest.php

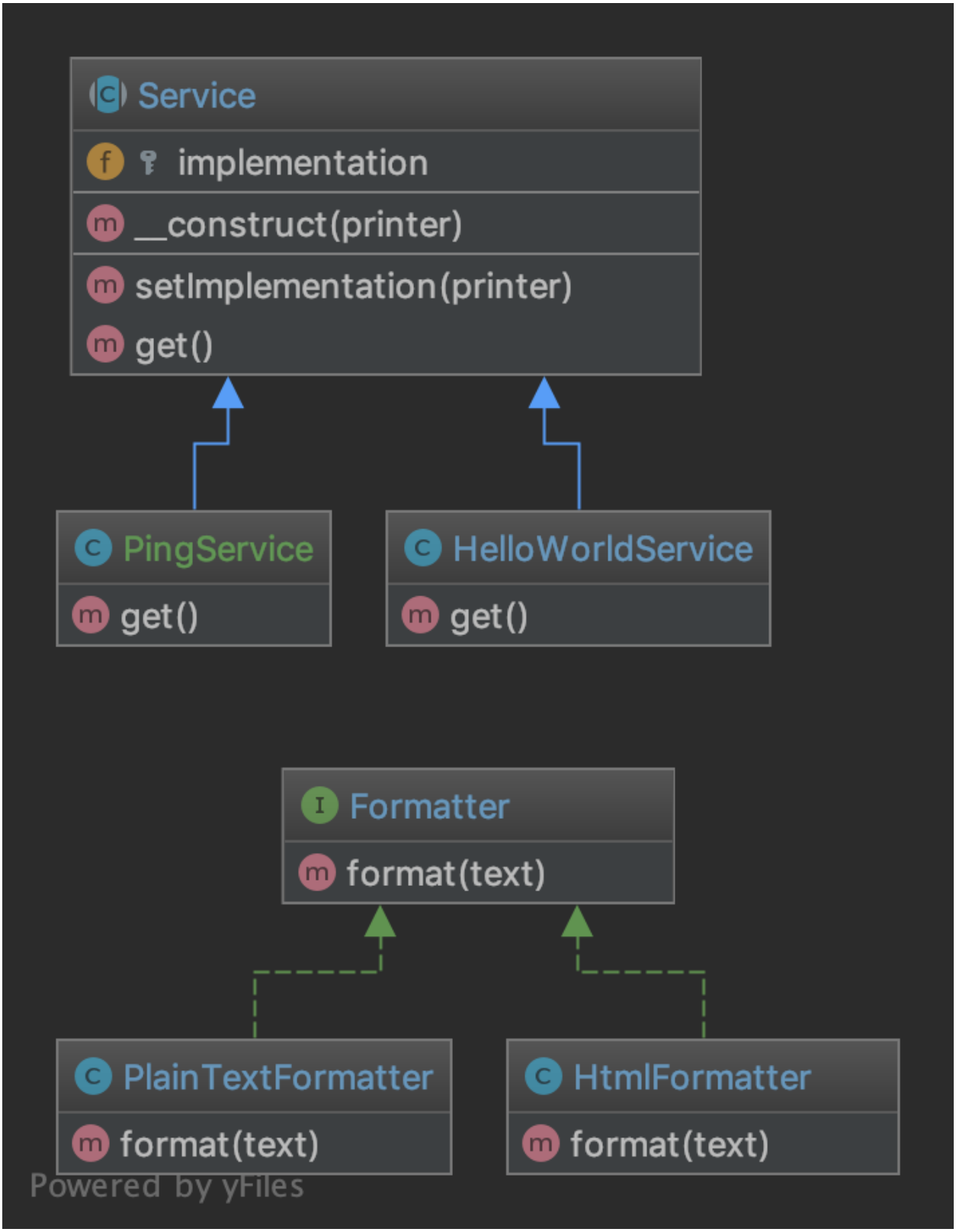
```php
<?php
declare(strict_types=1);

namespace DesignPatterns\Structural\Adapter\Tests;

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

class AdapterTest extends TestCase
{
    public function testCanTurnPageOnBook()
    {
        $book = new Book();
        $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
{
    /**
     * @var Formatter
     */
```

(continues on next page)

(continued from previous page)

```php
    protected $implementation;

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

    /**
     * @param Formatter $printer
     */
    public function setImplementation(Formatter $printer)
    {
        $this->implementation = $printer;
    }

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

HelloWorldService.php

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

namespace DesignPatterns\Structural\Bridge\Tests;

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

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

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

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

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

1.2.3 Composite

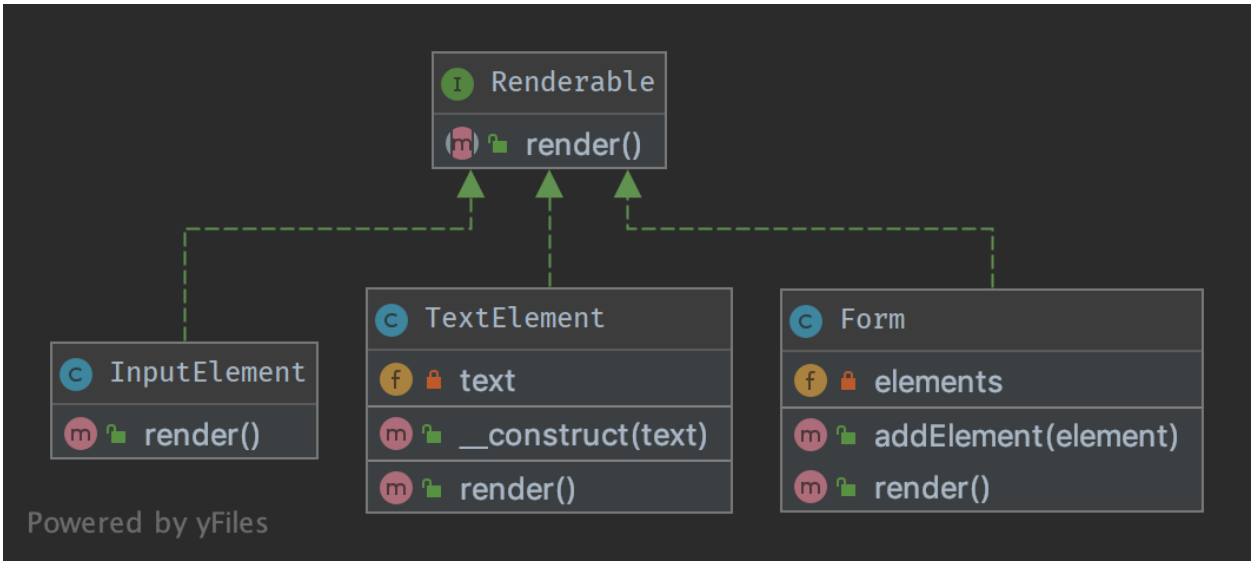
Purpose

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 $elements;

```

(continues on next page)

(continued from previous page)

```
    /**
     * 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
     *
     * @return string
     */
    public function render(): string
    {
        $formCode = '<form>';

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

        $formCode .= '</form>';

        return $formCode;
    }

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

InputElement.php

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

namespace DesignPatterns\Structural\Composite;

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

TextElement.php

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

namespace DesignPatterns\Structural\Composite;

class TextElement implements Renderable
{
    /**
```

(continues on next page)

(continued from previous page)

```
     * @var string
     */
    private $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;
use PHPUnit\Framework\TestCase;

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

        // This is just an example, in a real world scenario it is important to␣
↪remember that web browsers do not
        // currently support nested forms

        $this->assertSame(
            '<form>Email:<input type="text" /><form>Password:<input type="text" /></
↪form></form>',
            $form->render()
        );
    }
}
```

1.2.4 Data Mapper

Purpose

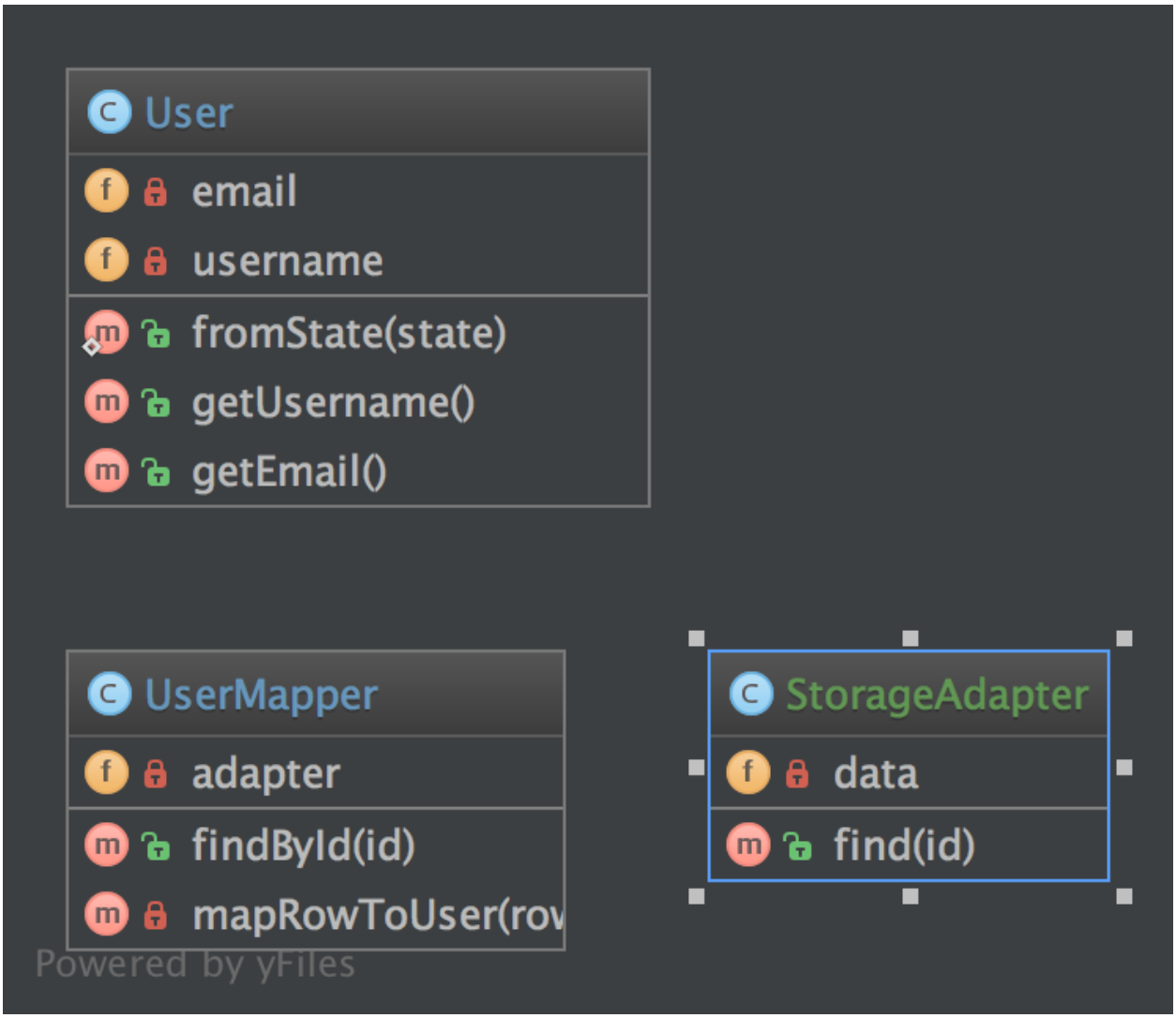
A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent 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
{
    /**
     * @var string
     */
    private $username;

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

    public static function fromState(array $state): User
    {
        // validate state before accessing keys!

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

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

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

    /**
     * @return string
     */
    public function getUsername()
    {
        return $this->username;
    }

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

UserMapper.php

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

namespace DesignPatterns\Structural\DataMapper;

class UserMapper
{
    /**
     * @var StorageAdapter
     */
    private $adapter;

    /**
     * @param StorageAdapter $storage
     */
    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
     *
     * @param int $id
     *
     * @return User
     */
    public function findById(int $id): User
    {
        $result = $this->adapter->find($id);

        if ($result === null) {
            throw new \InvalidArgumentException("User #$id not found");
        }

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

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

StorageAdapter.php

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

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
```

(continues on next page)

(continued from previous page)

```
{
    /**
     * @var array
     */
    private $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 DesignPatterns\Structural\DataMapper\StorageAdapter;
use DesignPatterns\Structural\DataMapper\User;
use DesignPatterns\Structural\DataMapper\UserMapper;
use PHPUnit\Framework\TestCase;

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

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

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

    public function testWillNotMapInvalidData()
    {
```

(continues on next page)

(continued from previous page)

```
25          $this->expectException(\InvalidArgumentException::class);
26
27          $storage = new StorageAdapter([]);
28          $mapper = new UserMapper($storage);
29
30          $mapper->findById(1);
31      }
32  }
```

1.2.5 Decorator

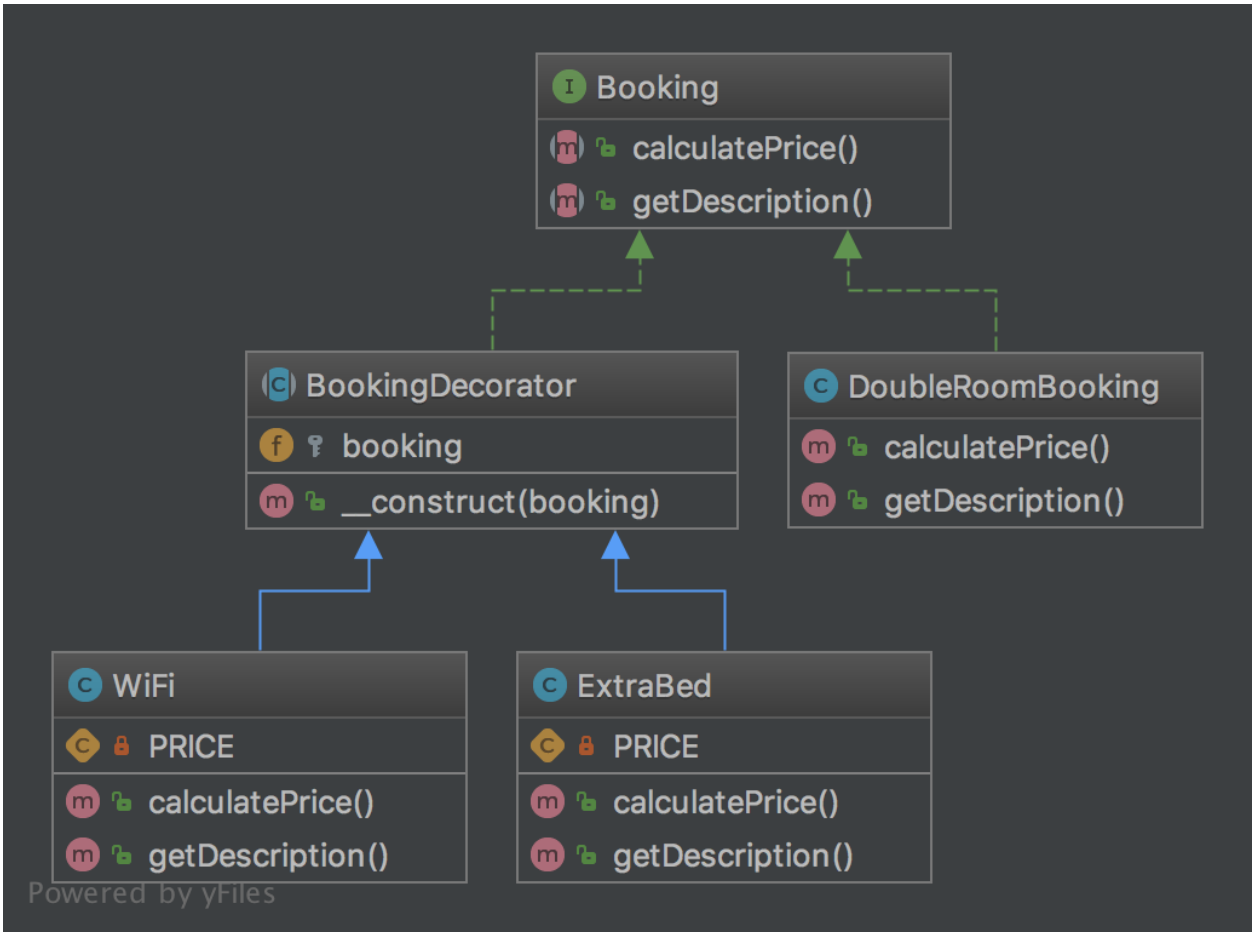
Purpose

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);
```

(continues on next page)

(continued from previous page)

```php

namespace DesignPatterns\Structural\Decorator;

abstract class BookingDecorator implements Booking
{
    /**
     * @var Booking
     */
    protected $booking;

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

DoubleRoomBooking.php

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

namespace DesignPatterns\Structural\Decorator;

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

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

ExtraBed.php

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

namespace DesignPatterns\Structural\Decorator;

class ExtraBed extends BookingDecorator
{
    private const PRICE = 30;

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

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

WiFi.php

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

namespace DesignPatterns\Structural\Decorator;

class WiFi extends BookingDecorator
{
    private const PRICE = 2;

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

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

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()
    {
```

(continues on next page)

(continued from previous page)

```
32          $booking = new DoubleRoomBooking();
33          $booking = new WiFi($booking);
34          $booking = new ExtraBed($booking);
35
36          $this->assertSame(72, $booking->calculatePrice());
37          $this->assertSame('double room with wifi with extra bed', $booking->
    ↪getDescription());
38      }
39  }
```

1.2.6 Dependency Injection

Purpose

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

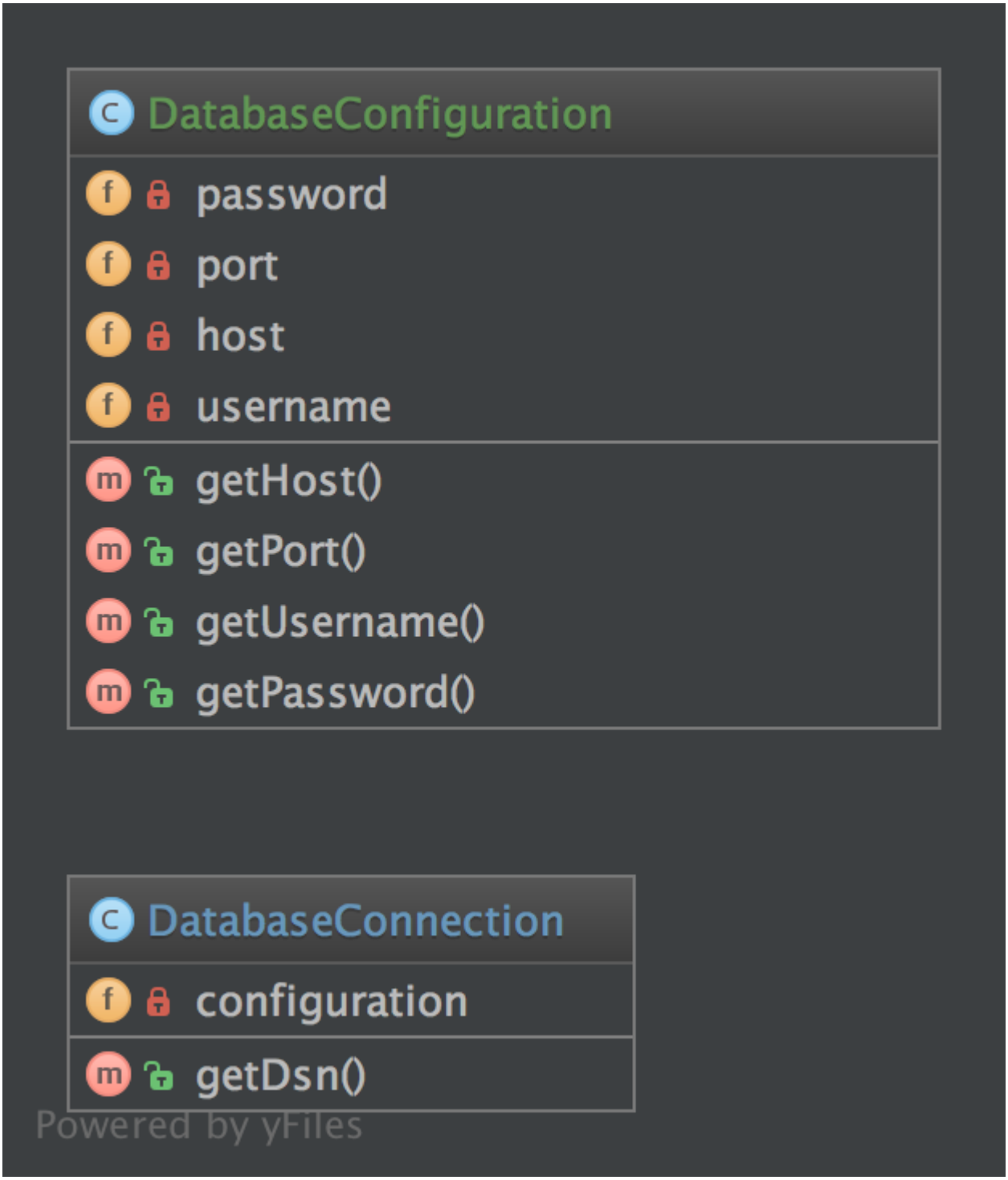
Usage

`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
{
    /**
     * @var string
     */
    private $host;

    /**
     * @var int
     */
    private $port;

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

    /**
     * @var string
     */
    private $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
{
    /**
     * @var DatabaseConfiguration
     */
    private $configuration;

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

    public function getDsn(): string
    {
        // this is just for the sake of demonstration, not a real DSN
        // notice that only the injected config is used here, so there is
        // a real separation of concerns here

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

Test

Tests/DependencyInjectionTest.php

```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);

```

(continues on next page)

(continued from previous page)

```
17          $this->assertSame('domnikl:1234@localhost:3306', $connection->getDsn());
18      }
19  }
```

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.

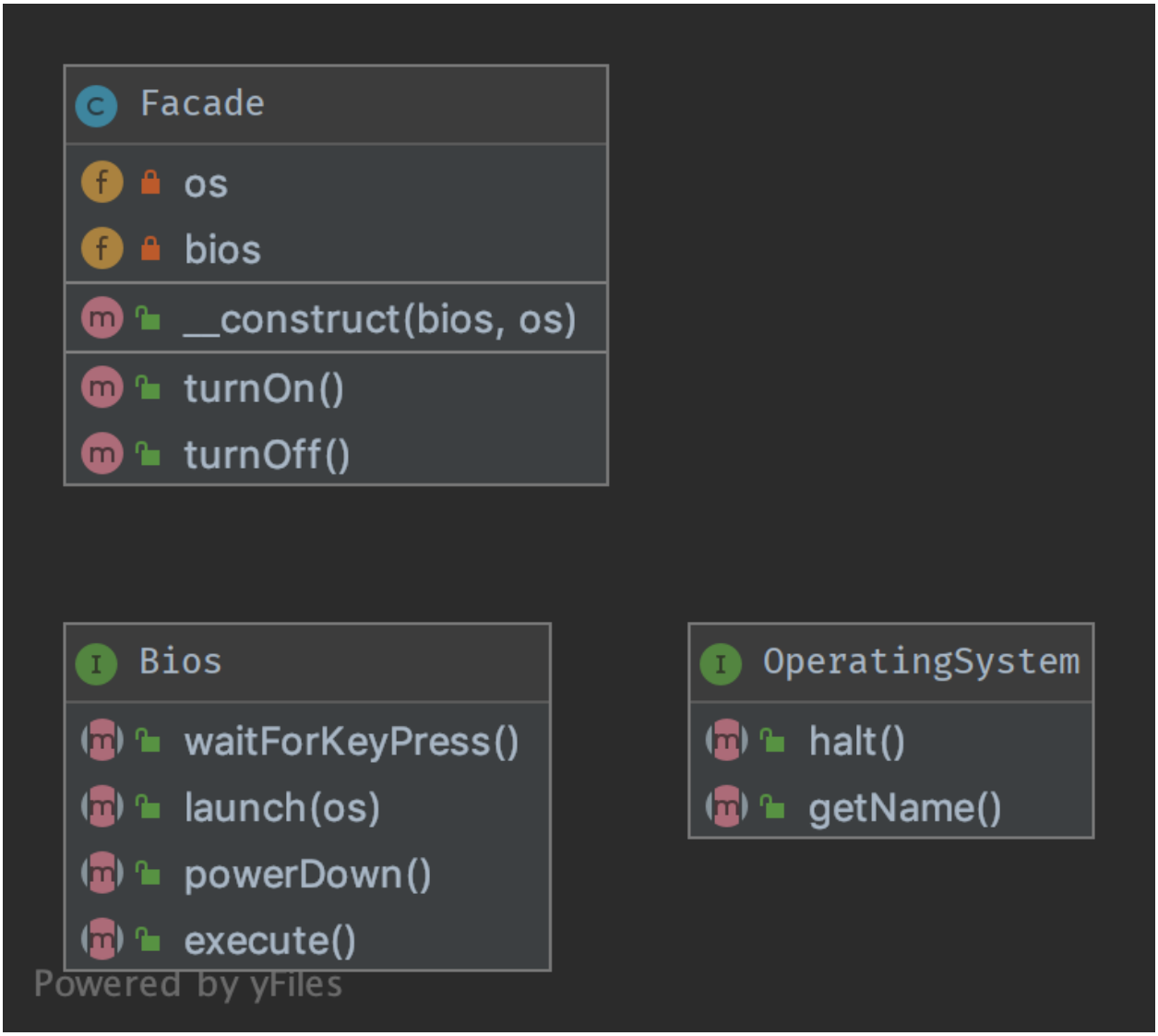
A Facade is meant to decouple a client and a sub-system by embedding many (but sometimes just one) interface, and of course to reduce complexity.

- 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
declare(strict_types=1);

namespace DesignPatterns\Structural\Facade;

class Facade
{
    /**
     * @var OperatingSystem
     */
```

(continues on next page)

(continued from previous page)

```
    private $os;

    /**
     * @var Bios
     */
    private $bios;

    /**
     * @param Bios $bios
     * @param OperatingSystem   $os
     */
    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();
```

(continues on next page)

(continued from previous page)

```
11
12      public function launch(OperatingSystem $os);
13
14      public function powerDown();
15  }
```

Test

Tests/FacadeTest.php

```
1   <?php
2   declare(strict_types=1);
3
4   namespace DesignPatterns\Structural\Facade\Tests;
5
6   use DesignPatterns\Structural\Facade\Facade;
7   use DesignPatterns\Structural\Facade\OperatingSystem;
8   use PHPUnit\Framework\TestCase;
9
10  class FacadeTest extends TestCase
11  {
12      public function testComputerOn()
13      {
14          /** @var OperatingSystem|\PHPUnit_Framework_MockObject_MockObject $os */
15          $os = $this->createMock('DesignPatterns\Structural\Facade\OperatingSystem');
16
17          $os->method('getName')
18              ->will($this->returnValue('Linux'));
19
20          $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\Bios')
21              ->setMethods(['launch', 'execute', 'waitForKeyPress'])
22              ->disableAutoload()
23              ->getMock();
24
25          $bios->expects($this->once())
26              ->method('launch')
27              ->with($os);
28
29          $facade = new Facade($bios, $os);
30
31          // the facade interface is simple
32          $facade->turnOn();
33
34          // but you can also access the underlying components
35          $this->assertSame('Linux', $os->getName());
36      }
37  }
```

1.2.8 Fluent Interface

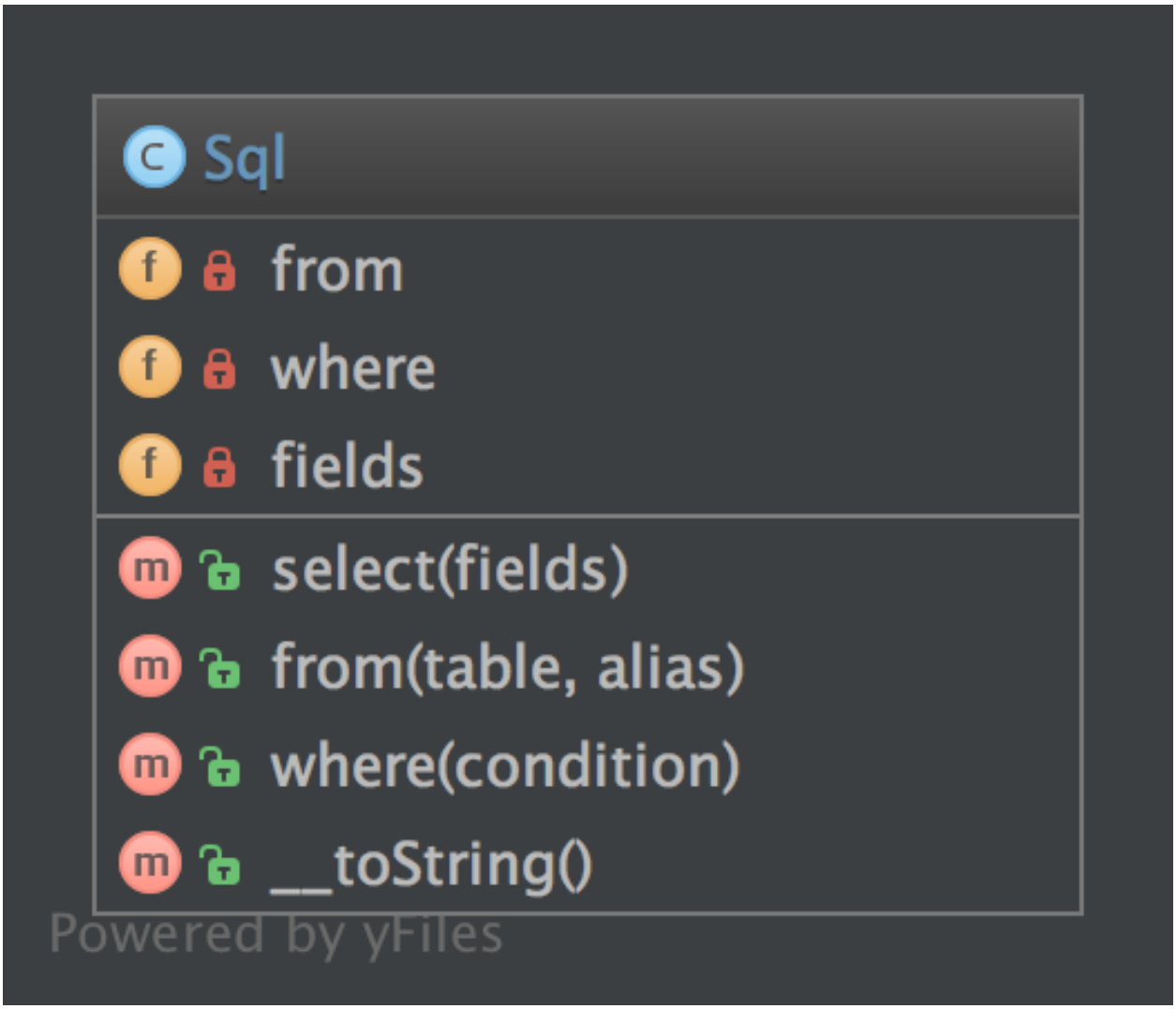
Purpose

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
declare(strict_types=1);

namespace DesignPatterns\Structural\FluentInterface;
```

(continues on next page)

(continued from previous page)

```php

class Sql
{
    /**
     * @var array
     */
    private $fields = [];

    /**
     * @var array
     */
    private $from = [];

    /**
     * @var array
     */
    private $where = [];

    public function select(array $fields): Sql
    {
        $this->fields = $fields;

        return $this;
    }

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

        return $this;
    }

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

        return $this;
    }

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

Test

Tests/FluentInterfaceTest.php

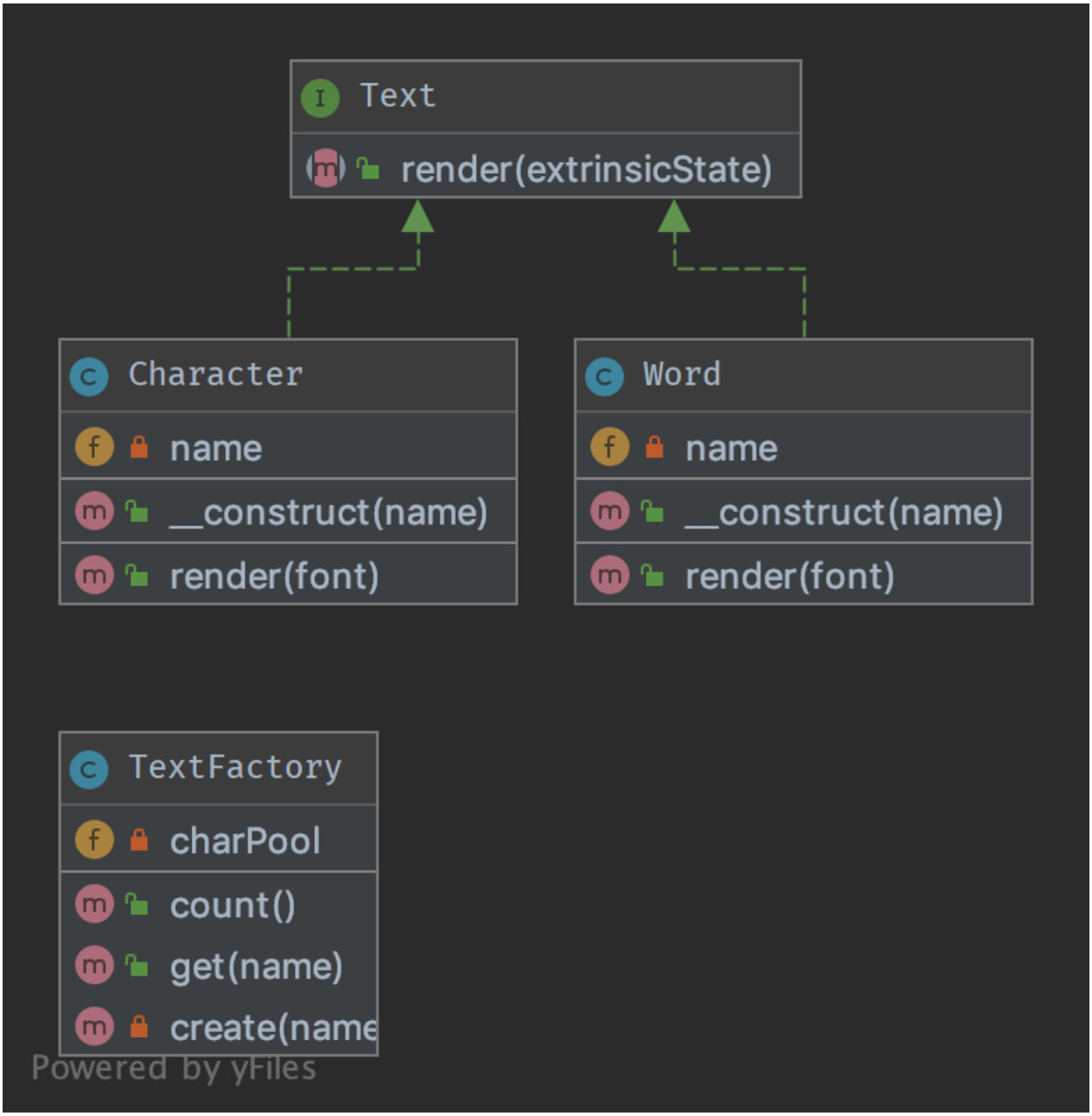
```php
<?php
declare(strict_types=1);

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
declare(strict_types=1);

namespace DesignPatterns\Structural\Flyweight;
```

(continues on next page)

(continued from previous page)

```

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

Word.php

```
<?php

namespace DesignPatterns\Structural\Flyweight;

class Word implements Text
{
    /**
     * @var string
     */
    private $name;

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

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

Character.php

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

namespace DesignPatterns\Structural\Flyweight;

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

    public function __construct(string $name)
    {
```

(continues on next page)

(continued from previous page)

```
        $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;

/**
 * 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 $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);
    }
}
```

Test

Tests/FlyweightTest.php

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

namespace DesignPatterns\Structural\Flyweight\Tests;

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

class FlyweightTest extends TestCase
{
    private $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 $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

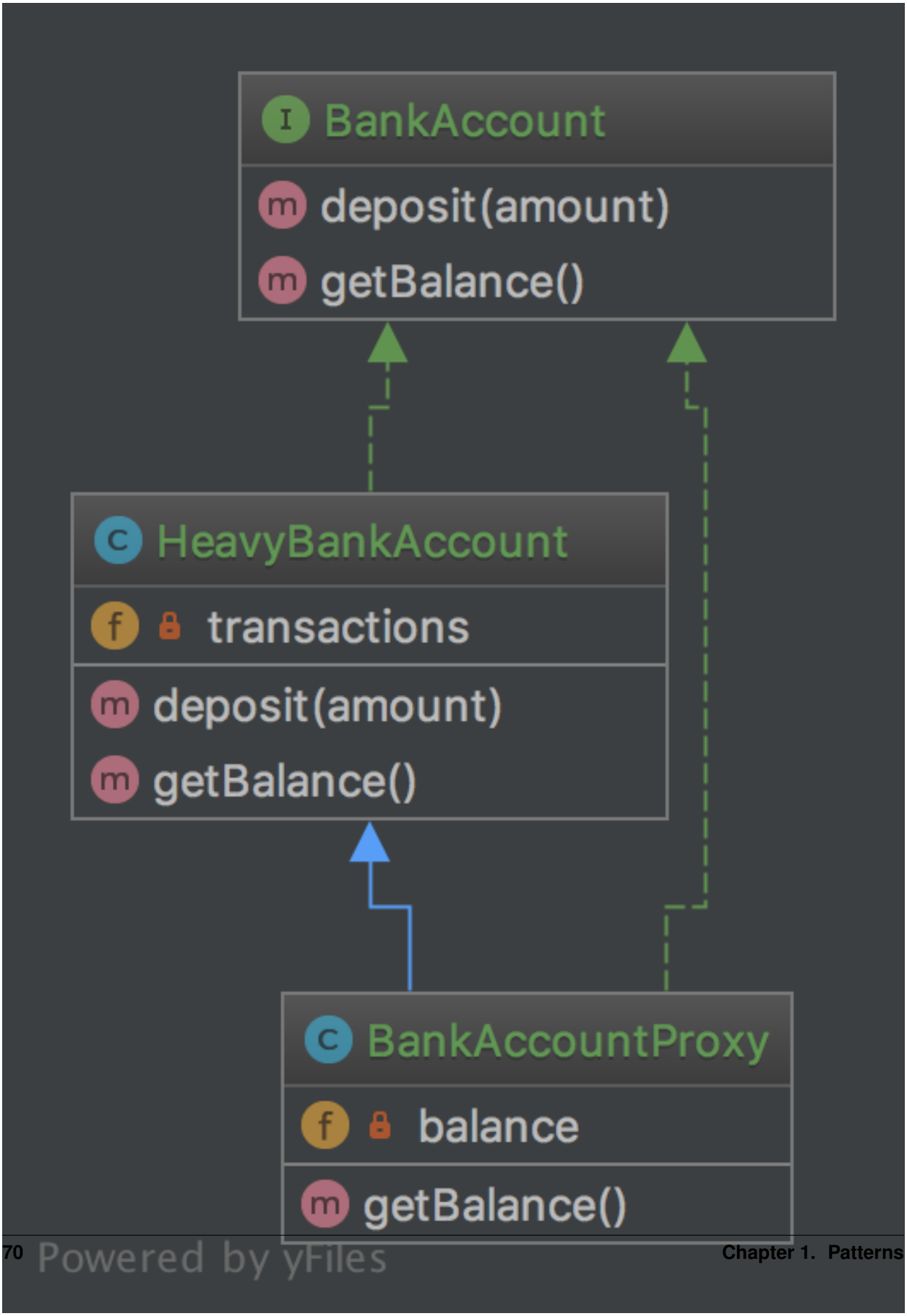
Purpose

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 $transactions = [];

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

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

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

BankAccountProxy.php

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

namespace DesignPatterns\Structural\Proxy;

class BankAccountProxy extends HeavyBankAccount implements BankAccount
{
    /**
     * @var int
```

(continues on next page)

(continued from previous page)

```
     */
    private $balance;

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

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

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

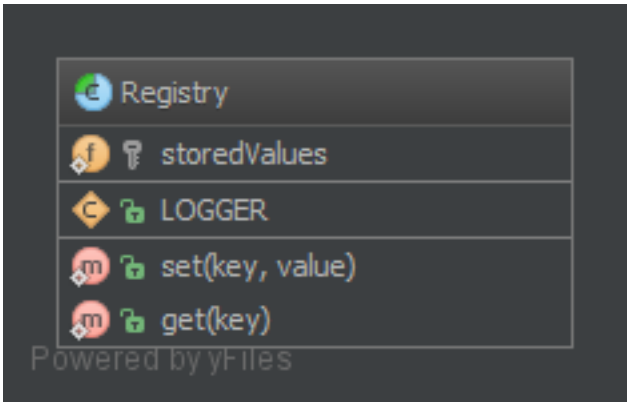
Test

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;

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 array
     */
    private static $storedValues = [];

    /**
     * @var array
     */
    private static $allowedKeys = [
        self::LOGGER,
    ];

    /**
     * @param string $key
     * @param mixed  $value
     *
     * @return void
     */
    public static function set(string $key, $value)
    {
        if (!in_array($key, self::$allowedKeys)) {
            throw new \InvalidArgumentException('Invalid key given');
        }

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

    /**
     * @param string $key
     *
     * @return mixed
     */
    public static function get(string $key)
    {
        if (!in_array($key, self::$allowedKeys) || !isset(self::$storedValues[$key])) {
            throw new \InvalidArgumentException('Invalid key given');
        }

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

Test

Tests/RegistryTest.php

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

namespace DesignPatterns\Structural\Registry\Tests;

use DesignPatterns\Structural\Registry\Registry;
use stdClass;
use PHPUnit\Framework\TestCase;

class RegistryTest extends TestCase
{
    public function testSetAndGetLogger()
    {
        $key = Registry::LOGGER;
        $logger = new stdClass();

        Registry::set($key, $logger);
        $storedLogger = Registry::get($key);

        $this->assertSame($logger, $storedLogger);
        $this->assertInstanceOf(stdClass::class, $storedLogger);
    }

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

        Registry::set('foobar', new stdClass());
    }

    /**
     * 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

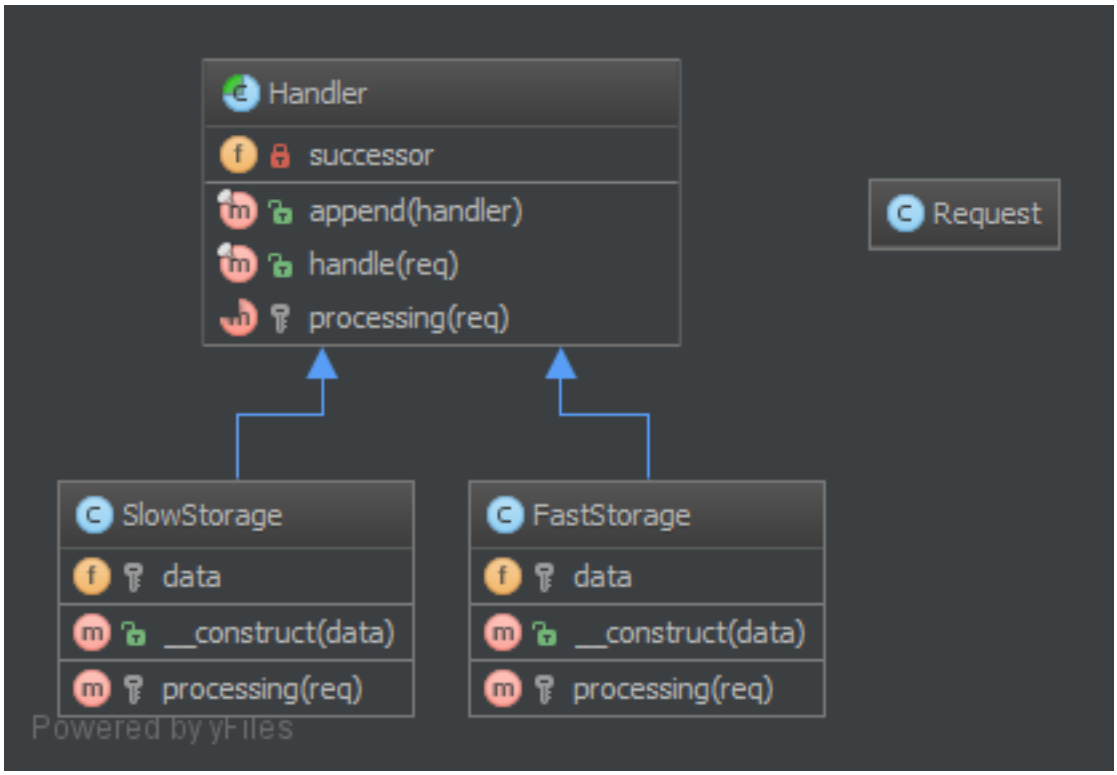
Purpose

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
declare(strict_types=1);

```

(continues on next page)

(continued from previous page)

```php
namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

abstract class Handler
{
    /**
     * @var Handler|null
     */
    private $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
     *
     * @param RequestInterface $request
     *
     * @return string|null
     */
    final public function handle(RequestInterface $request)
    {
        $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);
}
```

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
{
    /**
     * @var array
     */
    private $data;

```

(continues on next page)

(continued from previous page)

```php
    /**
     * @param array $data
     * @param Handler|null $successor
     */
    public function __construct(array $data, Handler $successor = null)
    {
        parent::__construct($successor);

        $this->data = $data;
    }

    /**
     * @param RequestInterface $request
     *
     * @return string|null
     */
    protected function processing(RequestInterface $request)
    {
        $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
<?php
declare(strict_types=1);

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

class SlowDatabaseHandler extends Handler
{
    /**
     * @param RequestInterface $request
     *
     * @return string|null
     */
    protected function processing(RequestInterface $request)
    {
        // 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
<?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;

class ChainTest extends TestCase
{
    /**
     * @var Handler
     */
    private $chain;

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

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

        $request = $this->createMock('Psr\Http\Message\RequestInterface');
        $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('Psr\Http\Message\UriInterface');
        $uri->method('getPath')->willReturn('/foo/baz');
        $uri->method('getQuery')->willReturn('');

        $request = $this->createMock('Psr\Http\Message\RequestInterface');
        $request->method('getMethod')
            ->willReturn('GET');
        $request->method('getUri')->willReturn($uri);

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

(continues on next page)

(continued from previous page)

```
53 }
```

1.3.2 Command

Purpose

To encapsulate invocation and decoupling.

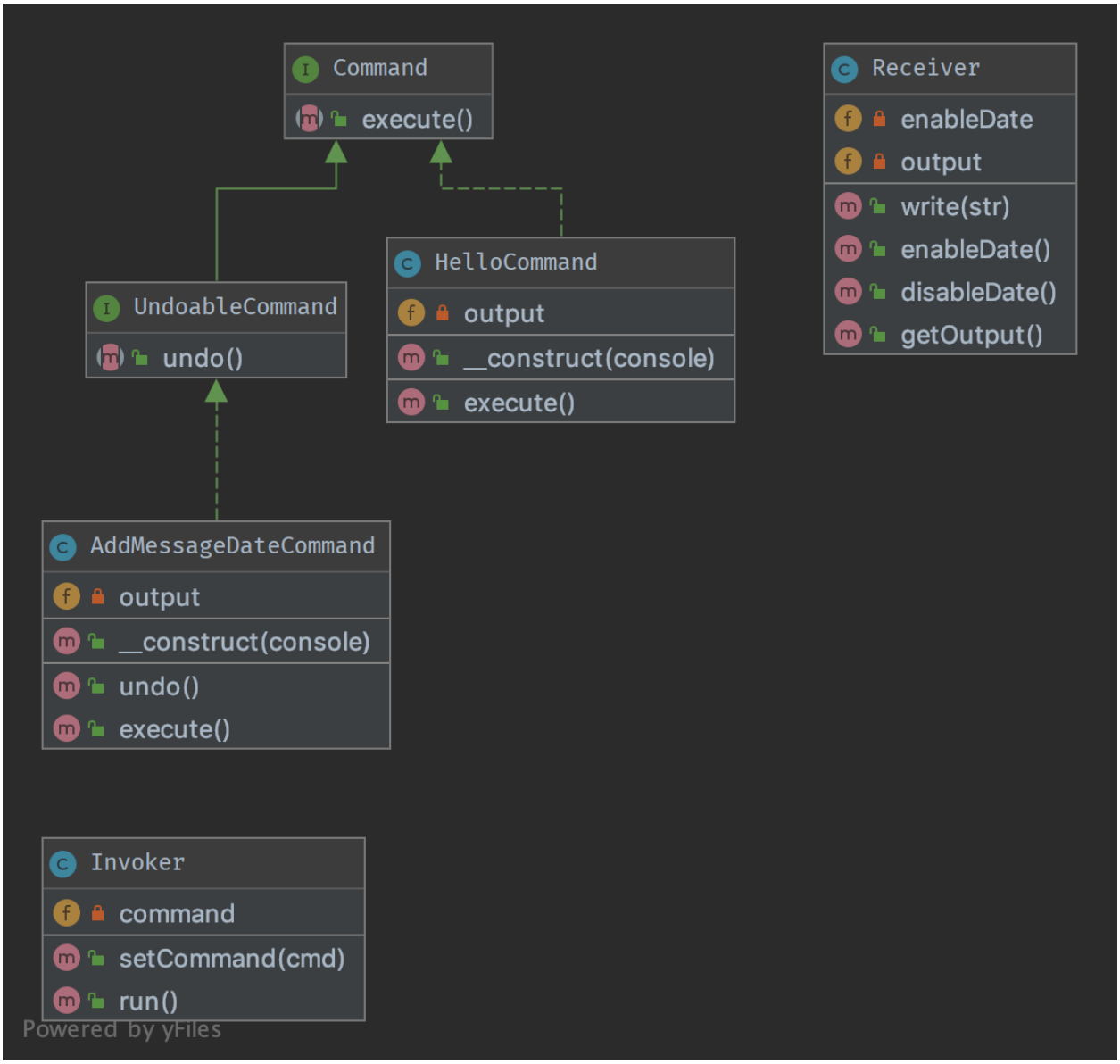
We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

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

Examples

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

UML Diagram



Code

You can also find this code on GitHub

Command.php

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

namespace DesignPatterns\Behavioral\Command;

interface Command
{
```

(continues on next page)

(continued from previous page)

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

HelloCommand.php

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

namespace DesignPatterns\Behavioral\Command;

/**
 * This concrete command calls "print" on the Receiver, but an external
 * invoker just knows that it can call "execute"
 */
class HelloCommand implements Command
{
    /**
     * @var Receiver
     */
    private $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
     *
     * @param Receiver $console
     */
    public function __construct(Receiver $console)
    {
        $this->output = $console;
    }

    /**
     * execute and output "Hello World".
     */
    public function execute()
    {
        // sometimes, there is no receiver and this is the command which does all the␣
↪work
        $this->output->write('Hello World');
    }
}
```

Receiver.php

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

namespace DesignPatterns\Behavioral\Command;

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

(continues on next page)

(continued from previous page)

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

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

    /**
     * @param string $str
     */
    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.
```

(continues on next page)

(continued from previous page)

```php
 */
class Invoker
{
    /**
     * @var Command
     */
    private $command;

    /**
     * in the invoker we find this kind of method for subscribing the command
     * There can be also a stack, a list, a fixed set ...
     *
     * @param Command $cmd
     */
    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
<?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());
    }
}
```

1.3.3 Iterator

Purpose

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

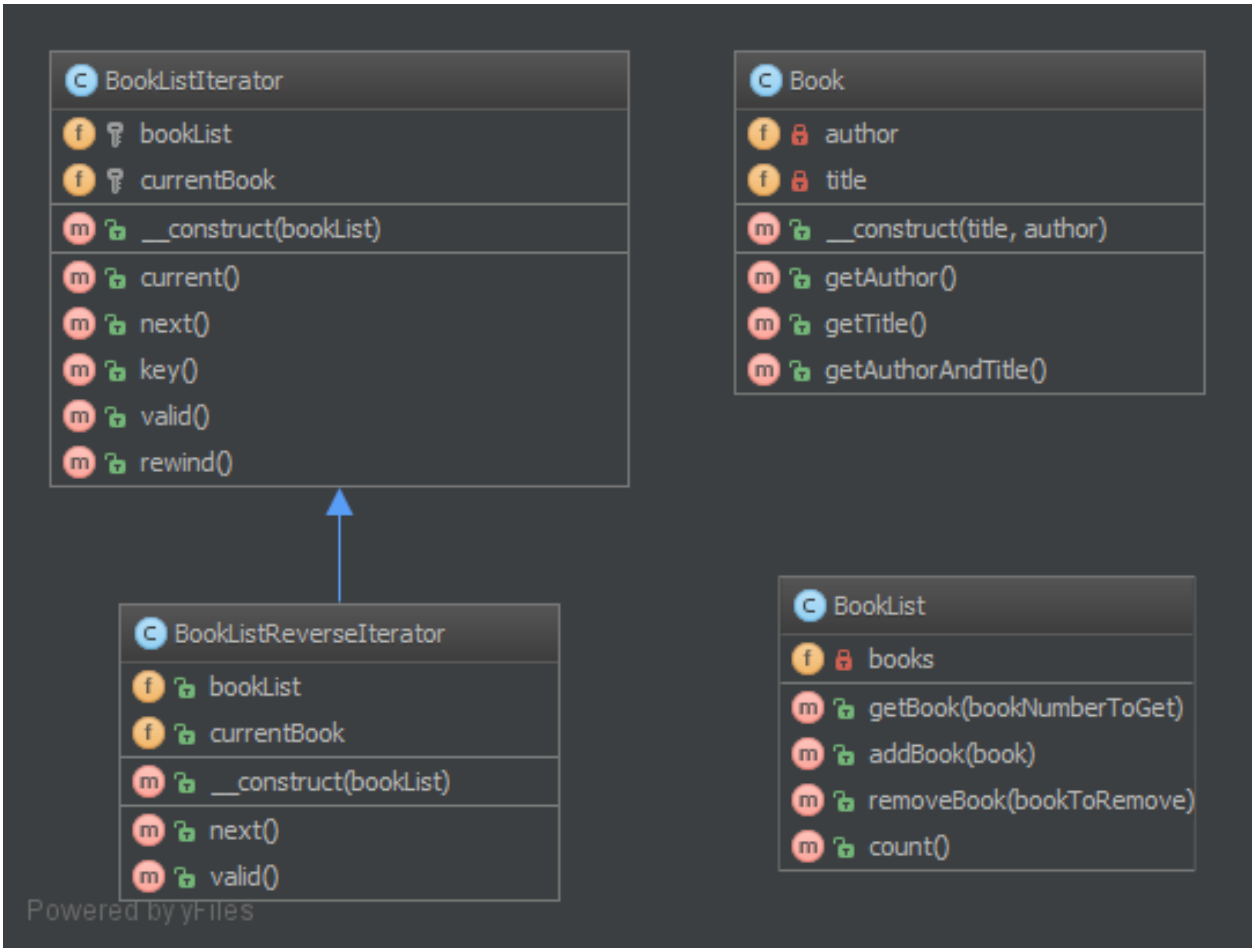
Examples

- 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
{
    /**
     * @var string
     */
    private $author;

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

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

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

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

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

BookList.php

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

namespace DesignPatterns\Behavioral\Iterator;

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

(continues on next page)

(continued from previous page)

```
12
13      /**
14       * @var int
15       */
16      private $currentIndex = 0;
17
18      public function addBook(Book $book)
19      {
20          $this->books[] = $book;
21      }
22
23      public function removeBook(Book $bookToRemove)
24      {
25          foreach ($this->books as $key => $book) {
26              if ($book->getAuthorAndTitle() === $bookToRemove->getAuthorAndTitle()) {
27                  unset($this->books[$key]);
28              }
29          }
30
31          $this->books = array_values($this->books);
32      }
33
34      public function count(): int
35      {
36          return count($this->books);
37      }
38
39      public function current(): Book
40      {
41          return $this->books[$this->currentIndex];
42      }
43
44      public function key(): int
45      {
46          return $this->currentIndex;
47      }
48
49      public function next()
50      {
51          $this->currentIndex++;
52      }
53
54      public function rewind()
55      {
56          $this->currentIndex = 0;
57      }
58
59      public function valid(): bool
60      {
61          return isset($this->books[$this->currentIndex]);
62      }
63  }
```

Test

Tests/IteratorTest.php

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

namespace DesignPatterns\Behavioral\Iterator\Tests;

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

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

        $books = [];

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

        $this->assertSame(
            [
                'Learning PHP Design Patterns by William Sanders',
                'Professional Php Design Patterns by Aaron Saray',
                'Clean Code by Robert C. Martin',
            ],
            $books
        );
    }

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

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

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

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

```

(continues on next page)

(continued from previous page)

```
56      public function testCanAddBookToList()
57      {
58          $book = new Book('Clean Code', 'Robert C. Martin');
59
60          $bookList = new BookList();
61          $bookList->addBook($book);
62
63          $this->assertCount(1, $bookList);
64      }
65
66      public function testCanRemoveBookFromList()
67      {
68          $book = new Book('Clean Code', 'Robert C. Martin');
69
70          $bookList = new BookList();
71          $bookList->addBook($book);
72          $bookList->removeBook($book);
73
74          $this->assertCount(0, $bookList);
75      }
76  }
```

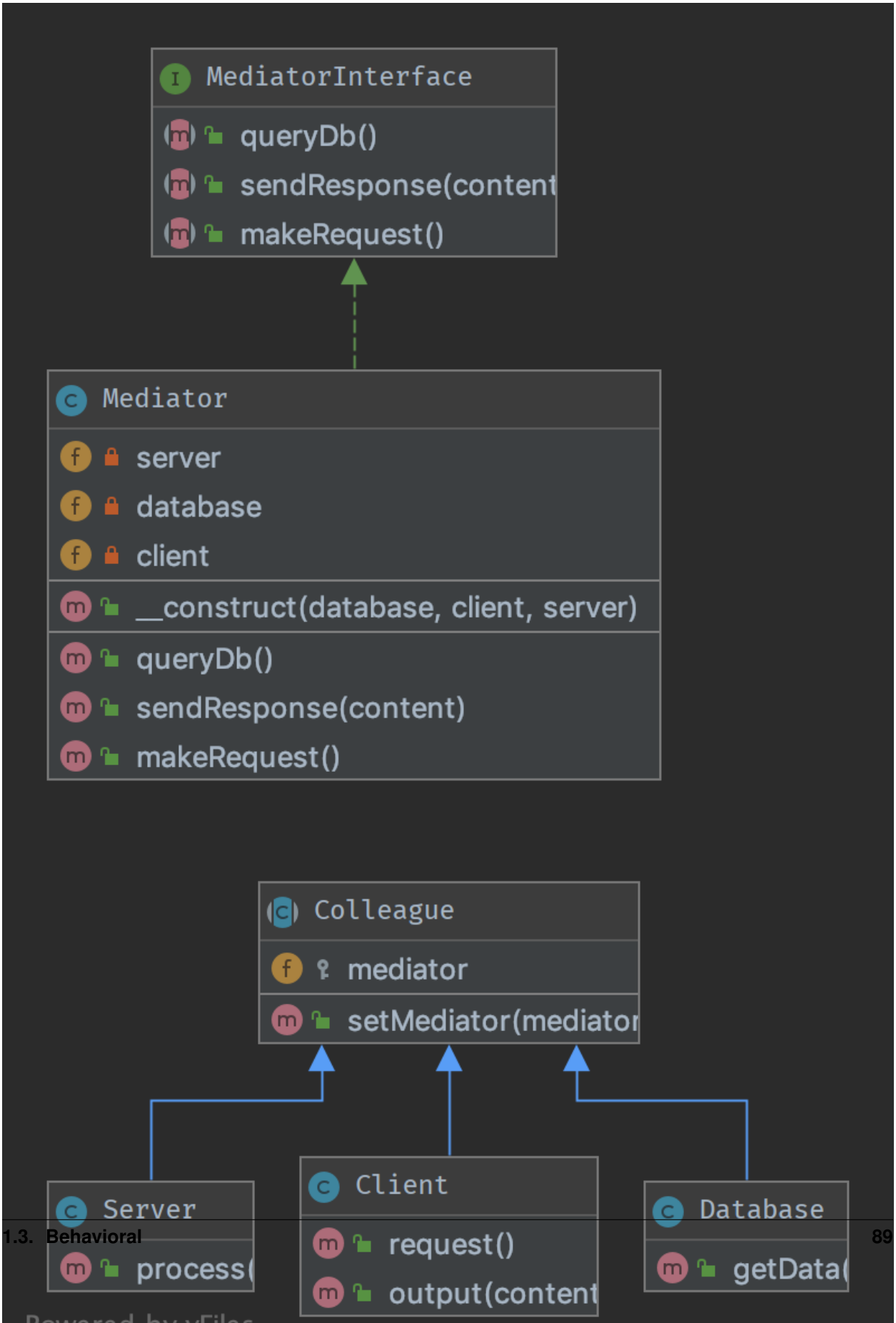
1.3.4 Mediator

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 MediatorInterface 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


MediatorInterface
queryDb()
sendResponse(content
makeRequest()
Mediator
server
database
client
__construct(database, client, server)
queryDb()
sendResponse(content)
makeRequest()
Colleague
mediator
setMediator(mediator
Server
process(
Client
request()
output(content
Database
getData(

Code

You can also find this code on GitHub

MediatorInterface.php

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

namespace DesignPatterns\Behavioral\Mediator;

interface MediatorInterface
{
    /**
     * sends the response.
     *
     * @param string $content
     */
    public function sendResponse($content);

    /**
     * makes a request
     */
    public function makeRequest();

    /**
     * queries the DB
     */
    public function queryDb();
}
```

Mediator.php

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

namespace DesignPatterns\Behavioral\Mediator;

class Mediator implements MediatorInterface
{
    /**
     * @var Subsystem\Server
     */
    private $server;

    /**
     * @var Subsystem\Database
     */
    private $database;

    /**
     * @var Subsystem\Client
     */
    private $client;

    /**
     * @param Subsystem\Database $database
     * @param Subsystem\Client $client
```

(continues on next page)

(continued from previous page)

```
     * @param Subsystem\Server $server
     */
    public function __construct(Subsystem\Database $database, Subsystem\Client
↪$client, Subsystem\Server $server)
    {
        $this->database = $database;
        $this->server = $server;
        $this->client = $client;

        $this->database->setMediator($this);
        $this->server->setMediator($this);
        $this->client->setMediator($this);
    }

    public function makeRequest()
    {
        $this->server->process();
    }

    public function queryDb(): string
    {
        return $this->database->getData();
    }

    /**
     * @param string $content
     */
    public function sendResponse($content)
    {
        $this->client->output($content);
    }
}
```

Colleague.php

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

namespace DesignPatterns\Behavioral\Mediator;

abstract class Colleague
{
    /**
     * @var MediatorInterface
     */
    protected $mediator;

    /**
     * @param MediatorInterface $mediator
     */
    public function setMediator(MediatorInterface $mediator)
    {
        $this->mediator = $mediator;
    }
}
```

Subsystem/Client.php

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

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

/**
 * Client is a client that makes requests and gets the response.
 */
class Client extends Colleague
{
    public function request()
    {
        $this->mediator->makeRequest();
    }

    public function output(string $content)
    {
        echo $content;
    }
}
```

Subsystem/Database.php

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

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

class Database extends Colleague
{
    public function getData(): string
    {
        return 'World';
    }
}
```

Subsystem/Server.php

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

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

class Server extends Colleague
{
    public function process()
    {
        $data = $this->mediator->queryDb();
        $this->mediator->sendResponse(sprintf("Hello %s", $data));
    }
}
```

Test

Tests/MediatorTest.php

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

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Mediator;
use DesignPatterns\Behavioral\Mediator\Subsystem\Client;
use DesignPatterns\Behavioral\Mediator\Subsystem\Database;
use DesignPatterns\Behavioral\Mediator\Subsystem\Server;
use PHPUnit\Framework\TestCase;

class MediatorTest extends TestCase
{
    public function testOutputHelloWorld()
    {
        $client = new Client();
        new Mediator(new Database(), $client, new Server());

        $this->expectOutputString('Hello World');
        $client->request();
    }
}
```

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
<?php
declare(strict_types=1);

namespace DesignPatterns\Behavioral\Memento;

class Memento
{
    /**
     * @var State
     */
    private $state;

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

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

State.php

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

namespace DesignPatterns\Behavioral\Memento;

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

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

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

(continues on next page)

(continued from previous page)

```
    /**
     * @param string $state
     */
    public function __construct(string $state)
    {
        self::ensureIsValidState($state);

        $this->state = $state;
    }

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

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

Ticket.php

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

namespace DesignPatterns\Behavioral\Memento;

/**
 * Ticket is the "Originator" in this implementation
 */
class Ticket
{
    /**
     * @var State
     */
    private $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()
```

(continues on next page)

(continued from previous page)

```
    {
        $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
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());
```

(continues on next page)

(continued from previous page)

```
31          $this->assertNotSame($openedState, $ticket->getState());
32      }
33  }
```

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:

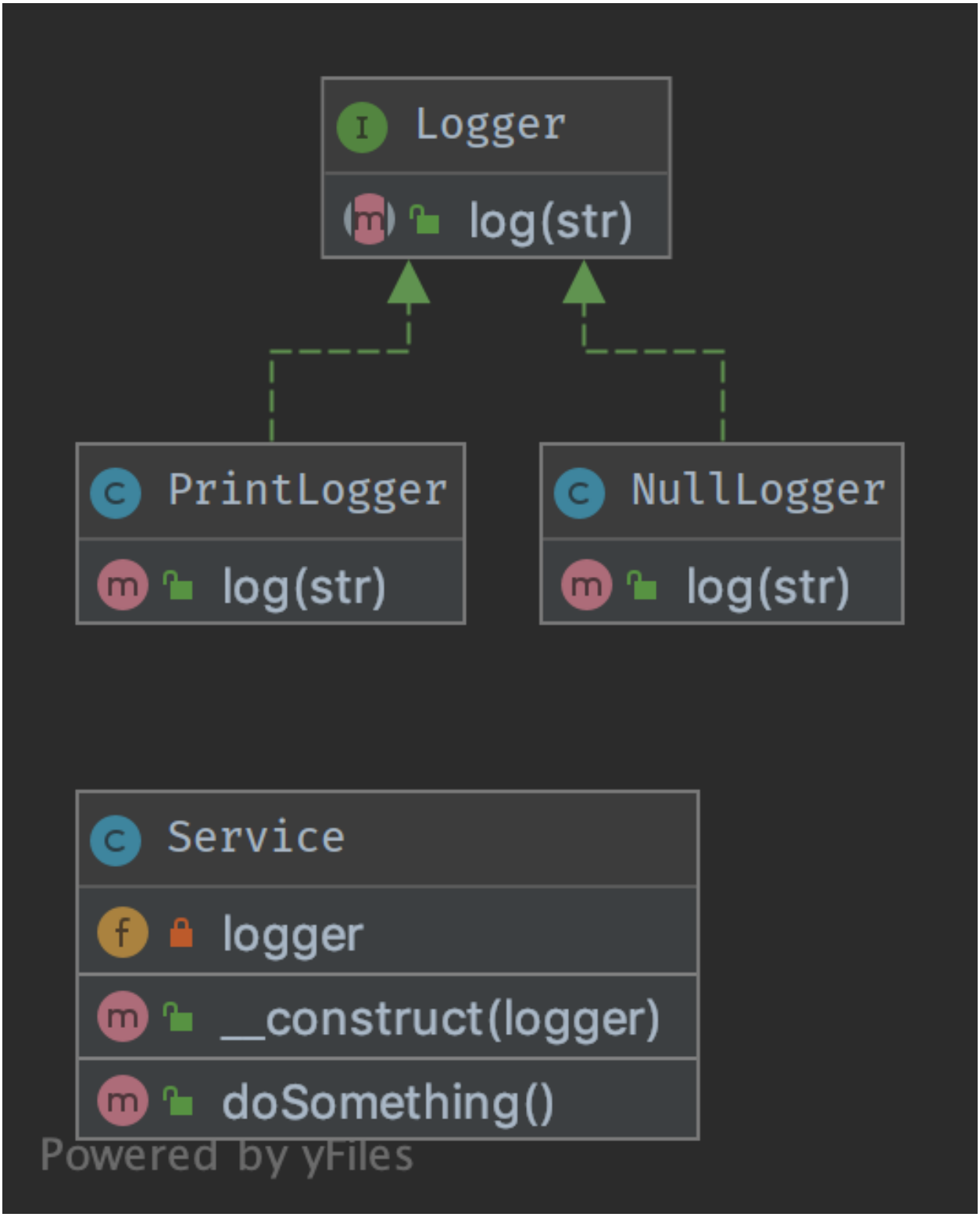
- Client code is simplified
- Reduces the chance of null pointer exceptions
- Fewer conditionals require less test cases

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
{
    /**
     * @var Logger
     */
    private $logger;

    /**
     * @param 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;
```

(continues on next page)

(continued from previous page)

```
5
6   class PrintLogger implements Logger
7   {
8       public function log(string $str)
9       {
10          echo $str;
11      }
12  }
```

NullLogger.php

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

Test

Tests/LoggerTest.php

```
1   <?php
2   declare(strict_types=1);
3
4   namespace DesignPatterns\Behavioral\NullObject\Tests;
5
6   use DesignPatterns\Behavioral\NullObject\NullLogger;
7   use DesignPatterns\Behavioral\NullObject\PrintLogger;
8   use DesignPatterns\Behavioral\NullObject\Service;
9   use PHPUnit\Framework\TestCase;
10
11  class LoggerTest extends TestCase
12  {
13      public function testNullObject()
14      {
15          $service = new Service(new NullLogger());
16          $this->expectOutputString('');
17          $service->doSomething();
18      }
19
20      public function testStandardLogger()
21      {
22          $service = new Service(new PrintLogger());
23          $this->expectOutputString('We are in␣
    ↪DesignPatterns\Behavioral\NullObject\Service::doSomething');
24          $service->doSomething();
25      }
26  }
```

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.

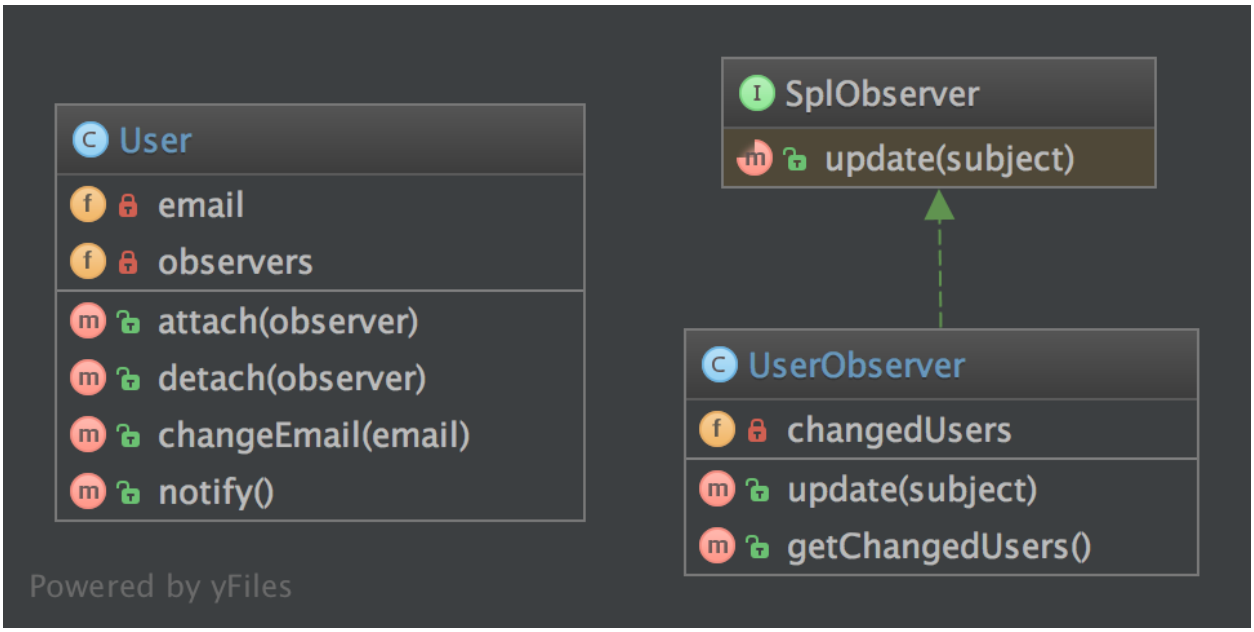
Examples

- 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
declare(strict_types=1);

namespace DesignPatterns\Behavioral\Observer;

/**
 * User implements the observed object (called Subject), it maintains a list of␣
↪observers and sends notifications to
```

(continues on next page)

(continued from previous page)

```
 * them in case changes are made on the User object
 */
class User implements \SplSubject
{
    /**
     * @var string
     */
    private $email;

    /**
     * @var \SplObjectStorage
     */
    private $observers;

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

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

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

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

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

UserObserver.php

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

namespace DesignPatterns\Behavioral\Observer;

class UserObserver implements \SplObserver
{
    /**
     * @var User[]
     */
    private $changedUsers = [];
```

(continues on next page)

(continued from previous page)

```php

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

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

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

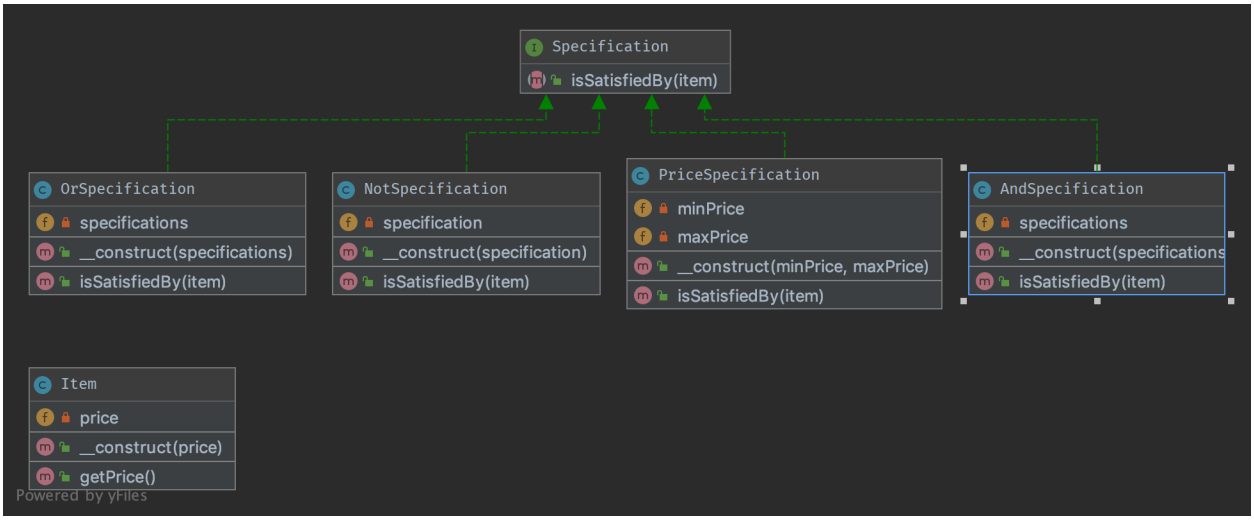
Purpose

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
{
    /**
     * @var float
     */
    private $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 $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
{
    /**
     * @var float|null
```

(continues on next page)

(continued from previous page)

```
     */
    private $maxPrice;

    /**
     * @var float|null
     */
    private $minPrice;

    /**
     * @param float $minPrice
     * @param float $maxPrice
     */
    public function __construct($minPrice, $maxPrice)
    {
        $this->minPrice = $minPrice;
        $this->maxPrice = $maxPrice;
    }

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

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

        return true;
    }
}
```

AndSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

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

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

    /**
     * if at least one specification is false, return false, else return true.
     */
```

(continues on next page)

(continued from previous page)

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

        return true;
    }
}
```

NotSpecification.php

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

namespace DesignPatterns\Behavioral\Specification;

class NotSpecification implements Specification
{
    /**
     * @var Specification
     */
    private $specification;

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

    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
{
```

(continues on next page)

(continued from previous page)

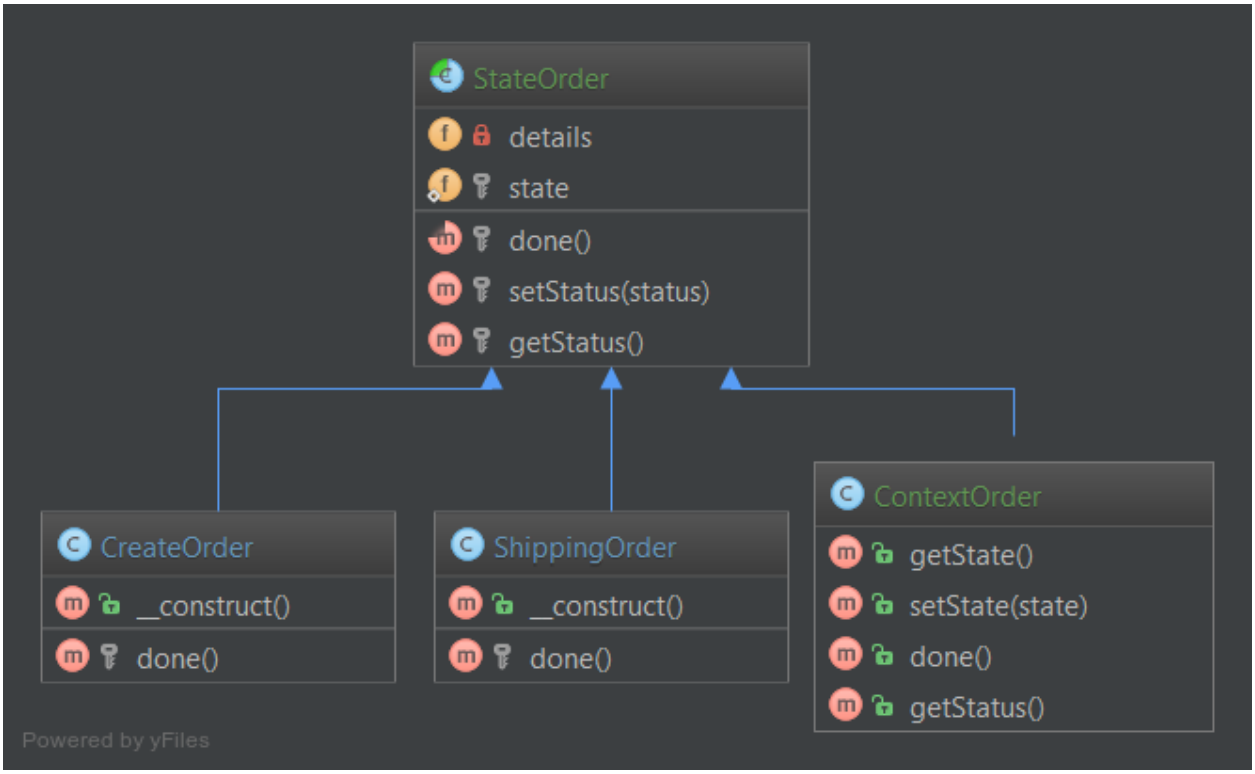
```
    public function testCanOr()
    {
        $spec1 = new PriceSpecification(50, 99);
        $spec2 = new PriceSpecification(101, 200);

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

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

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

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

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

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

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

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
<?php
declare(strict_types=1);

namespace DesignPatterns\Behavioral\State;

class OrderContext
{
    /**
     * @var State
     */
    private $state;

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

        return $order;
    }

    public function setState(State $state)
    {
```

(continues on next page)

(continued from previous page)

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

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

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

State.php

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

namespace DesignPatterns\Behavioral\State;

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

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

StateCreated.php

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

namespace DesignPatterns\Behavioral\State;

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

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

StateShipped.php

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

namespace DesignPatterns\Behavioral\State;

class StateShipped implements State
{
```

(continues on next page)

(continued from previous page)

```
    public function proceedToNext(OrderContext $context)
    {
        $context->setState(new StateDone());
    }

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

StateDone.php

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

namespace DesignPatterns\Behavioral\State;

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

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

Test

Tests/StateTest.php

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

namespace DesignPatterns\Behavioral\State\Tests;

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

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

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

    public function testCanProceedToStateShipped()
    {
        $contextOrder = OrderContext::create();
```

(continues on next page)

(continued from previous page)

```
        $contextOrder->proceedToNext();

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

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

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

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

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

1.3.10 Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy

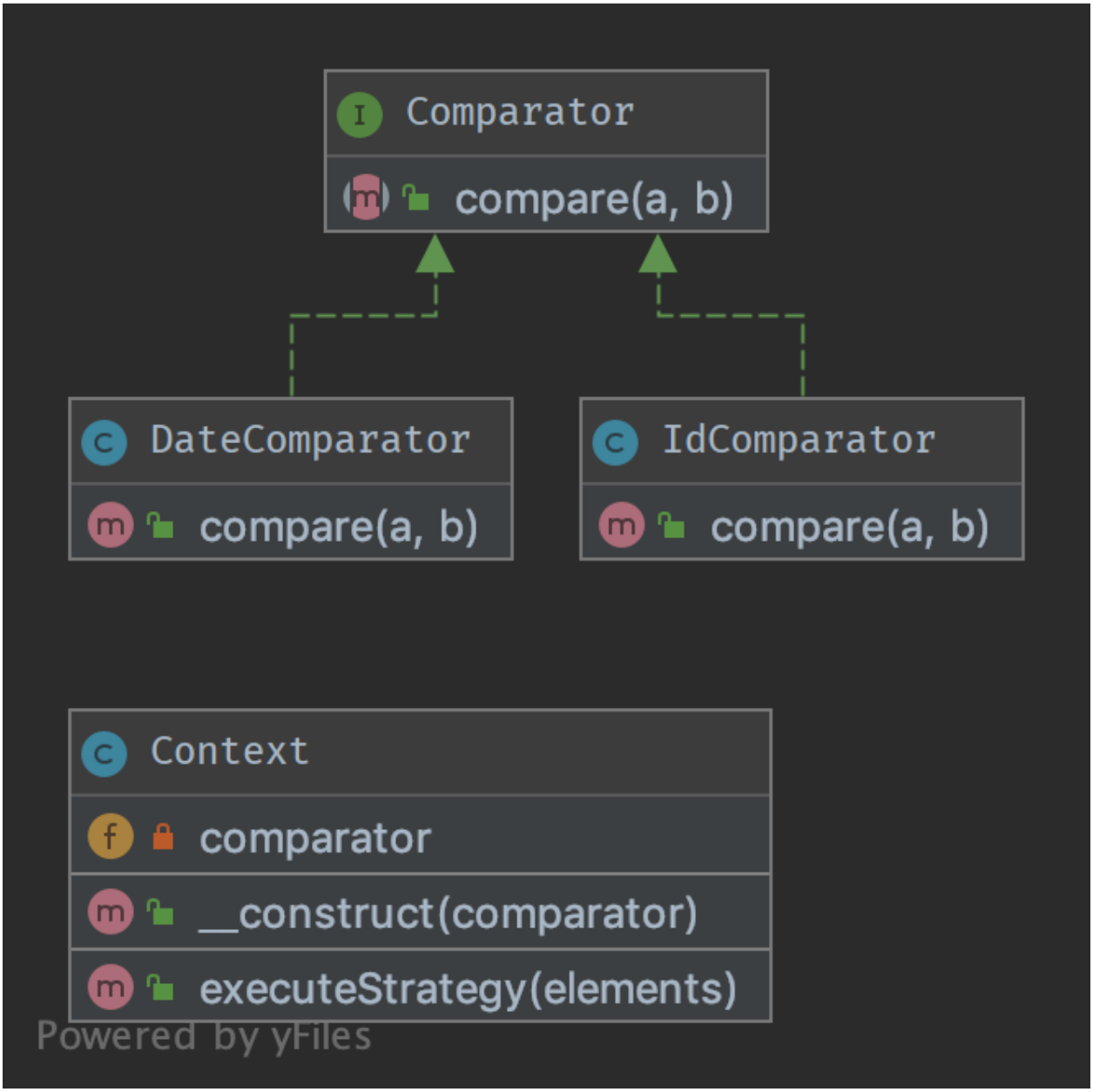
Purpose

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;

```

(continues on next page)

(continued from previous page)

```php
class Context
{
    /**
     * @var Comparator
     */
    private $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
<?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
<?php
declare(strict_types=1);

namespace DesignPatterns\Behavioral\Strategy;

class DateComparator implements Comparator
{
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return int
     */
    public function compare($a, $b): int
    {
        $aDate = new \DateTime($a['date']);
        $bDate = new \DateTime($b['date']);
```

(continues on next page)

(continued from previous page)

```
18
19          return $aDate <=> $bDate;
20      }
21  }
```

IdComparator.php

```
1   <?php
2   declare(strict_types=1);
3
4   namespace DesignPatterns\Behavioral\Strategy;
5
6   class IdComparator implements Comparator
7   {
8       /**
9        * @param mixed $a
10       * @param mixed $b
11       *
12       * @return int
13       */
14      public function compare($a, $b): int
15      {
16          return $a['id'] <=> $b['id'];
17      }
18  }
```

Test

Tests/StrategyTest.php

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

(continues on next page)

(continued from previous page)

```
26
27      public function provideDates()
28      {
29          return [
30              [
31                  [['date' => '2014-03-03'], ['date' => '2015-03-02'], ['date' => '2013-
    ↪03-01']],
32                  ['date' => '2013-03-01'],
33              ],
34              [
35                  [['date' => '2014-02-03'], ['date' => '2013-02-01'], ['date' => '2015-
    ↪02-02']],
36                  ['date' => '2013-02-01'],
37              ],
38          ];
39      }
40
41      /**
42       * @dataProvider provideIntegers
43       *
44       * @param array $collection
45       * @param array $expected
46       */
47      public function testIdComparator($collection, $expected)
48      {
49          $obj = new Context(new IdComparator());
50          $elements = $obj->executeStrategy($collection);
51
52          $firstElement = array_shift($elements);
53          $this->assertSame($expected, $firstElement);
54      }
55
56      /**
57       * @dataProvider provideDates
58       *
59       * @param array $collection
60       * @param array $expected
61       */
62      public function testDateComparator($collection, $expected)
63      {
64          $obj = new Context(new DateComparator());
65          $elements = $obj->executeStrategy($collection);
66
67          $firstElement = array_shift($elements);
68          $this->assertSame($expected, $firstElement);
69      }
70  }
```

1.3.11 Template Method

Purpose

Template Method is a behavioral design pattern.

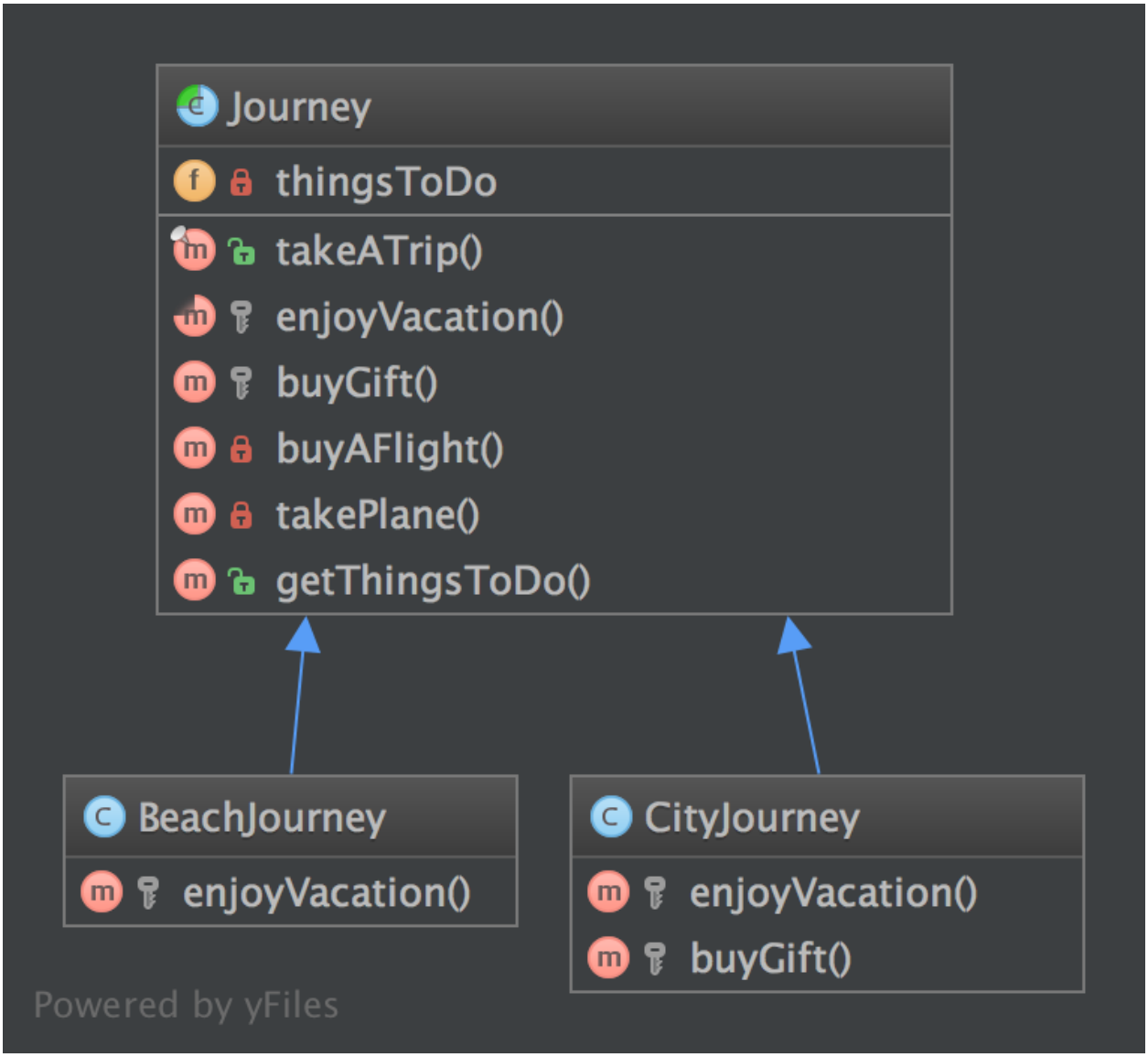
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template "finish" the behavior of an algorithm.

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
<?php
declare(strict_types=1);

namespace DesignPatterns\Behavioral\TemplateMethod;

abstract class Journey
{
    /**
     * @var string[]
     */
    private $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
     *
     * @return null|string
     */
    protected function buyGift()
    {
        return null;
    }

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

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

(continues on next page)

(continued from previous page)

```
58
59      /**
60       * @return string[]
61       */
62      public function getThingsToDo(): array
63      {
64          return $this->thingsToDo;
65      }
66  }
```

BeachJourney.php

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

CityJourney.php

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

Test

Tests/JourneyTest.php

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

(continues on next page)

(continued from previous page)

```php
use PHPUnit\Framework\TestCase;

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

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

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

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

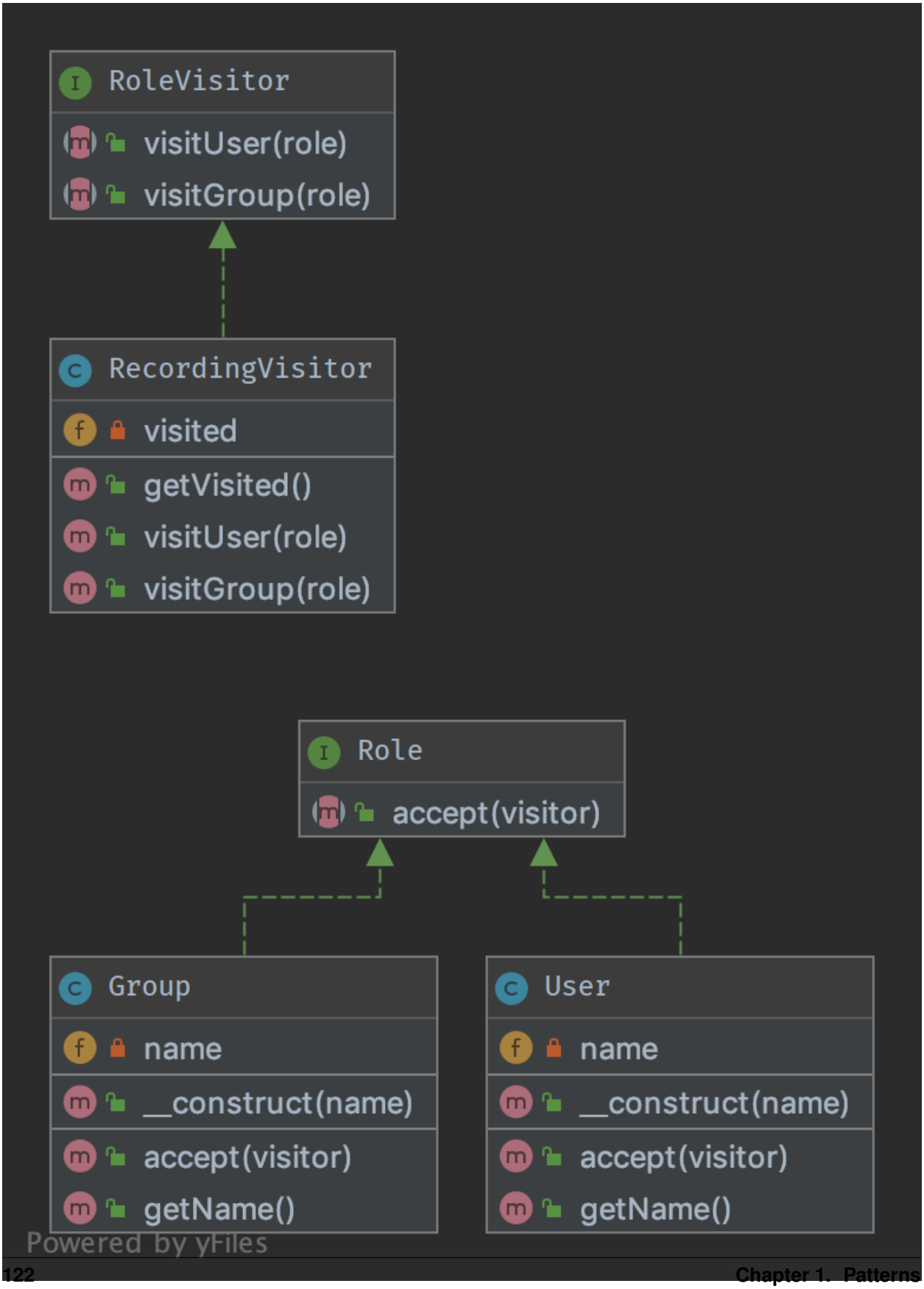
1.3.12 Visitor

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 Visitee that make 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 $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;

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

User.php

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

namespace DesignPatterns\Behavioral\Visitor;

class User implements Role
{
    /**
     * @var string
     */
    private $name;

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

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

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

Group.php

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

namespace DesignPatterns\Behavioral\Visitor;

class Group implements Role
{
    /**
     * @var string
     */
    private $name;

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

(continues on next page)

(continued from previous page)

```
    }

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

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

Test

Tests/VisitorTest.php

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

namespace DesignPatterns\Tests\Visitor\Tests;

use DesignPatterns\Behavioral\Visitor;
use PHPUnit\Framework\TestCase;

class VisitorTest extends TestCase
{
    /**
     * @var Visitor\RecordingVisitor
     */
    private $visitor;

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

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

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

(continues on next page)

(continued from previous page)

```
39 }
```

1.4 More

1.4.1 Service Locator

THIS IS CONSIDERED TO BE AN ANTI-PATTERN!

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

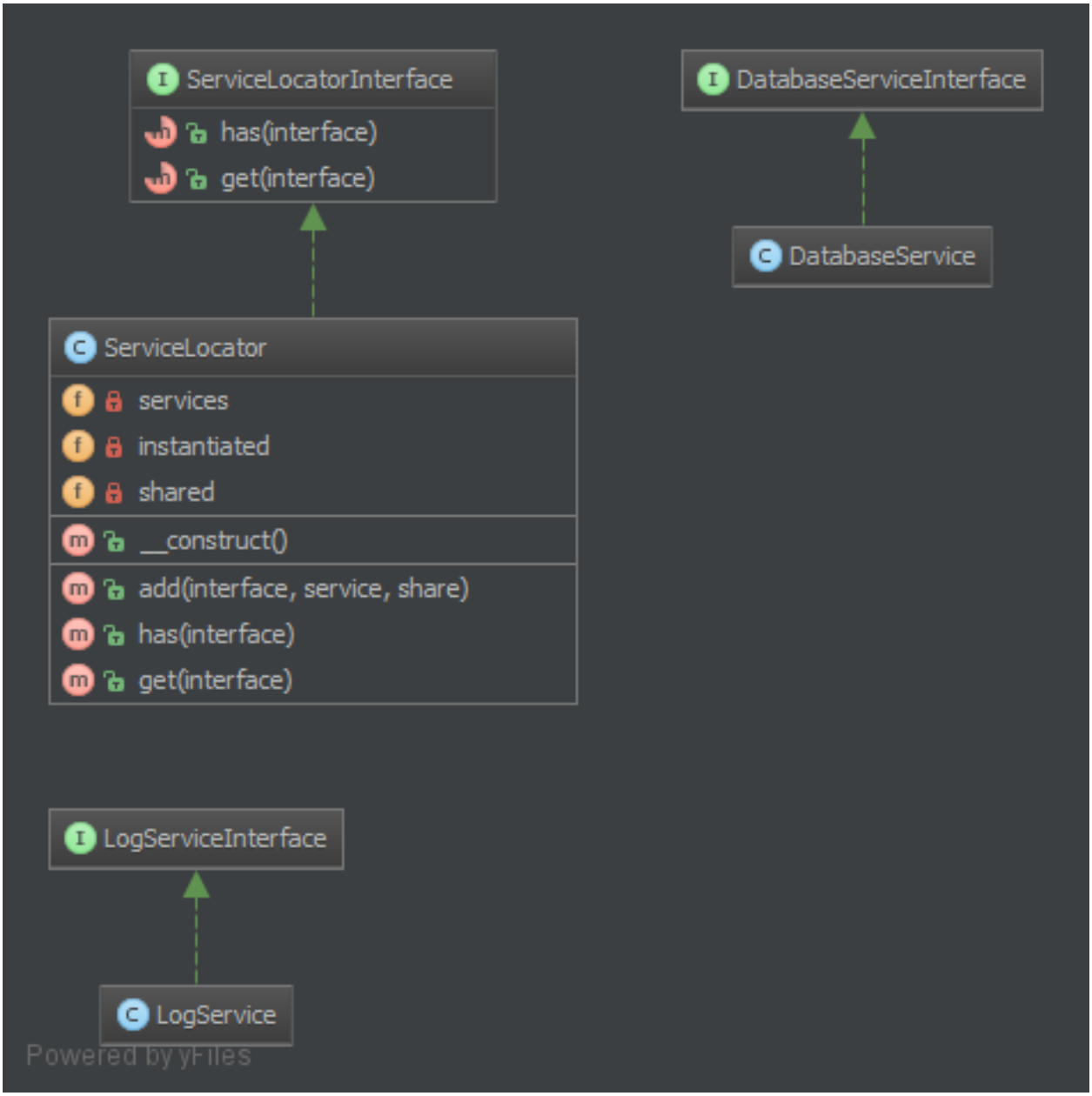
Purpose

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

Usage

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

UML Diagram



Code

You can also find this code on GitHub

ServiceLocator.php

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

namespace DesignPatterns\More\ServiceLocator;

```

(continues on next page)

(continued from previous page)

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

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

    /**
     * @var array
     */
    private $shared = [];

    /**
     * instead of supplying a class here, you could also store a service for an interface
     *
     * @param string $class
     * @param object $service
     * @param bool $share
     */
    public function addInstance(string $class, $service, bool $share = true)
    {
        $this->services[$class] = $service;
        $this->instantiated[$class] = $service;
        $this->shared[$class] = $share;
    }

    /**
     * instead of supplying a class here, you could also store a service for an interface
     *
     * @param string $class
     * @param array $params
     * @param bool $share
     */
    public function addClass(string $class, array $params, bool $share = true)
    {
        $this->services[$class] = $params;
        $this->shared[$class] = $share;
    }

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

    /**
     * @param string $class
     *
     * @return object
     */
```

(continues on next page)

(continued from previous page)

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

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

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

        if ($this->shared[$class]) {
            $this->instantiated[$class] = $object;
        }

        return $object;
    }
}
```

LogService.php

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

namespace DesignPatterns\More\ServiceLocator;

class LogService
{
}
```

Test

Tests/ServiceLocatorTest.php

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

namespace DesignPatterns\More\ServiceLocator\Tests;

use DesignPatterns\More\ServiceLocator\LogService;
use DesignPatterns\More\ServiceLocator\ServiceLocator;
```

(continues on next page)

(continued from previous page)

```php
use PHPUnit\Framework\TestCase;

class ServiceLocatorTest extends TestCase
{
    /**
     * @var ServiceLocator
     */
    private $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

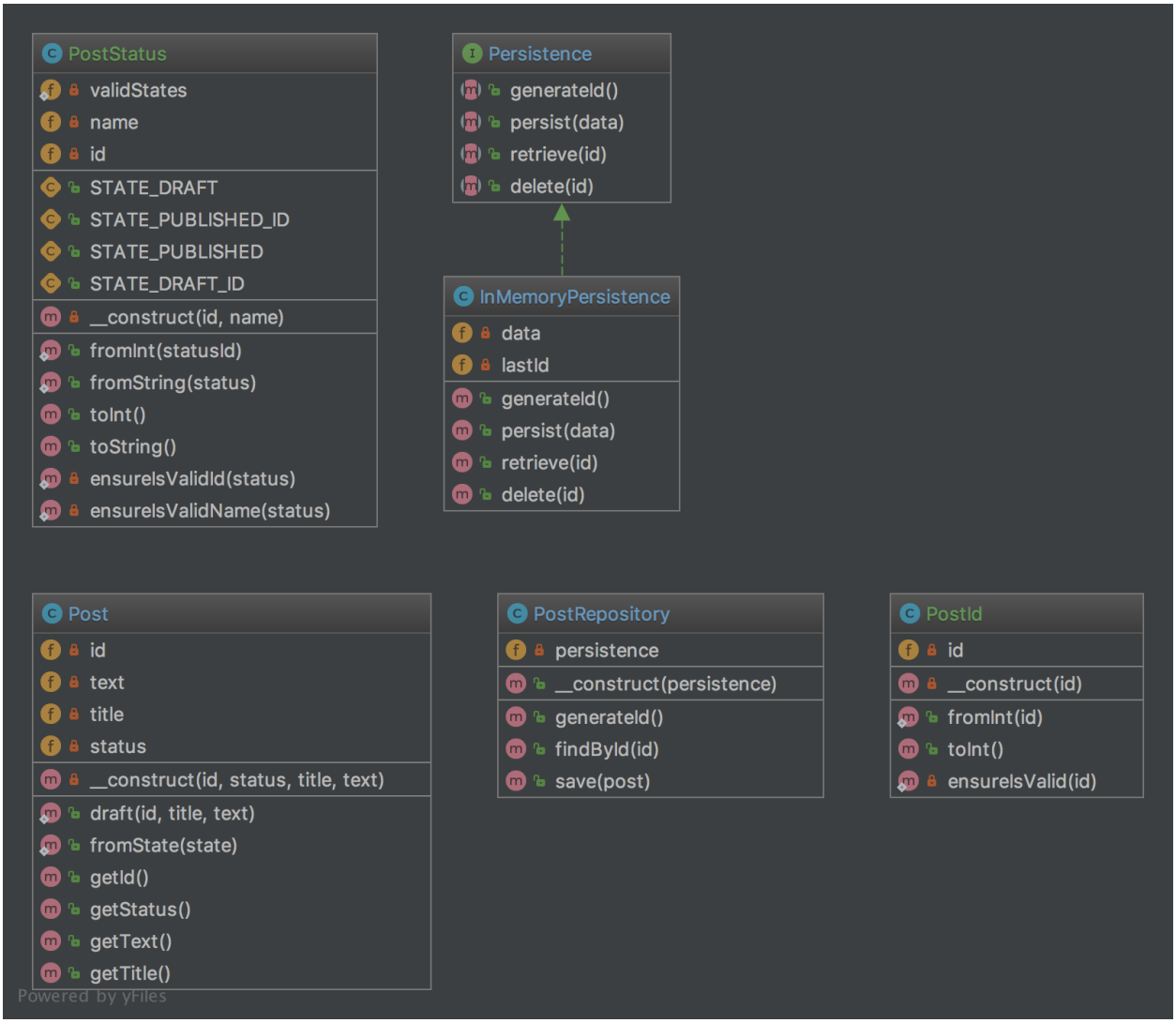
Purpose

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
{
    /**
     * @var PostId
     */
    private $id;
```

(continues on next page)

(continued from previous page)

```
12
13      /**
14       * @var PostStatus
15       */
16      private $status;
17
18      /**
19       * @var string
20       */
21      private $title;
22
23      /**
24       * @var string
25       */
26      private $text;
27
28      public static function draft(PostId $id, string $title, string $text): Post
29      {
30          return new self(
31              $id,
32              PostStatus::fromString(PostStatus::STATE_DRAFT),
33              $title,
34              $text
35          );
36      }
37
38      public static function fromState(array $state): Post
39      {
40          return new self(
41              PostId::fromInt($state['id']),
42              PostStatus::fromInt($state['statusId']),
43              $state['title'],
44              $state['text']
45          );
46      }
47
48      /**
49       * @param PostId $id
50       * @param PostStatus $status
51       * @param string $title
52       * @param string $text
53       */
54      private function __construct(PostId $id, PostStatus $status, string $title,␣
↪string $text)
55      {
56          $this->id = $id;
57          $this->status = $status;
58          $this->text = $text;
59          $this->title = $title;
60      }
61
62      public function getId(): PostId
63      {
64          return $this->id;
65      }
66
67      public function getStatus(): PostStatus
```

(continues on next page)

(continued from previous page)

```
    {
        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;

/**
 * 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
{
    /**
     * @var int
     */
    private $id;

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

(continues on next page)

(continued from previous page)

```
    {
        if ($id <= 0) {
            throw new \InvalidArgumentException('Invalid PostId given');
        }
    }
}
```

PostStatus.php

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

namespace DesignPatterns\More\Repository\Domain;

/**
 * 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;
    const STATE_PUBLISHED_ID = 2;

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

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

    /**
     * @var int
     */
    private $id;

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

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

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

    public static function fromString(string $status)
    {
        self::ensureIsValidName($status);

        return new self(array_search($status, self::$validStates), $status);
    }

```

(continues on next page)

(continued from previous page)

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

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

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

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


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

PostRepository.php

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

namespace DesignPatterns\More\Repository;

use 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
```

(continues on next page)

(continued from previous page)

```
 * between the domain and data mapping layers
 */
class PostRepository
{
    /**
     * @var Persistence
     */
    private $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);
        }

        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;

```

(continues on next page)

(continued from previous page)

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

InMemoryPersistence.php

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

namespace DesignPatterns\More\Repository;

class InMemoryPersistence implements Persistence
{
    /**
     * @var array
     */
    private $data = [];

    /**
     * @var int
     */
    private $lastId = 0;

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

        return $this->lastId;
    }

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

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

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

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

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

Test

Tests/PostRepositoryTest.php

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

namespace DesignPatterns\More\Repository\Tests;

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
{
    /**
     * @var PostRepository
     */
    private $repository;

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

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

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

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

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

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

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

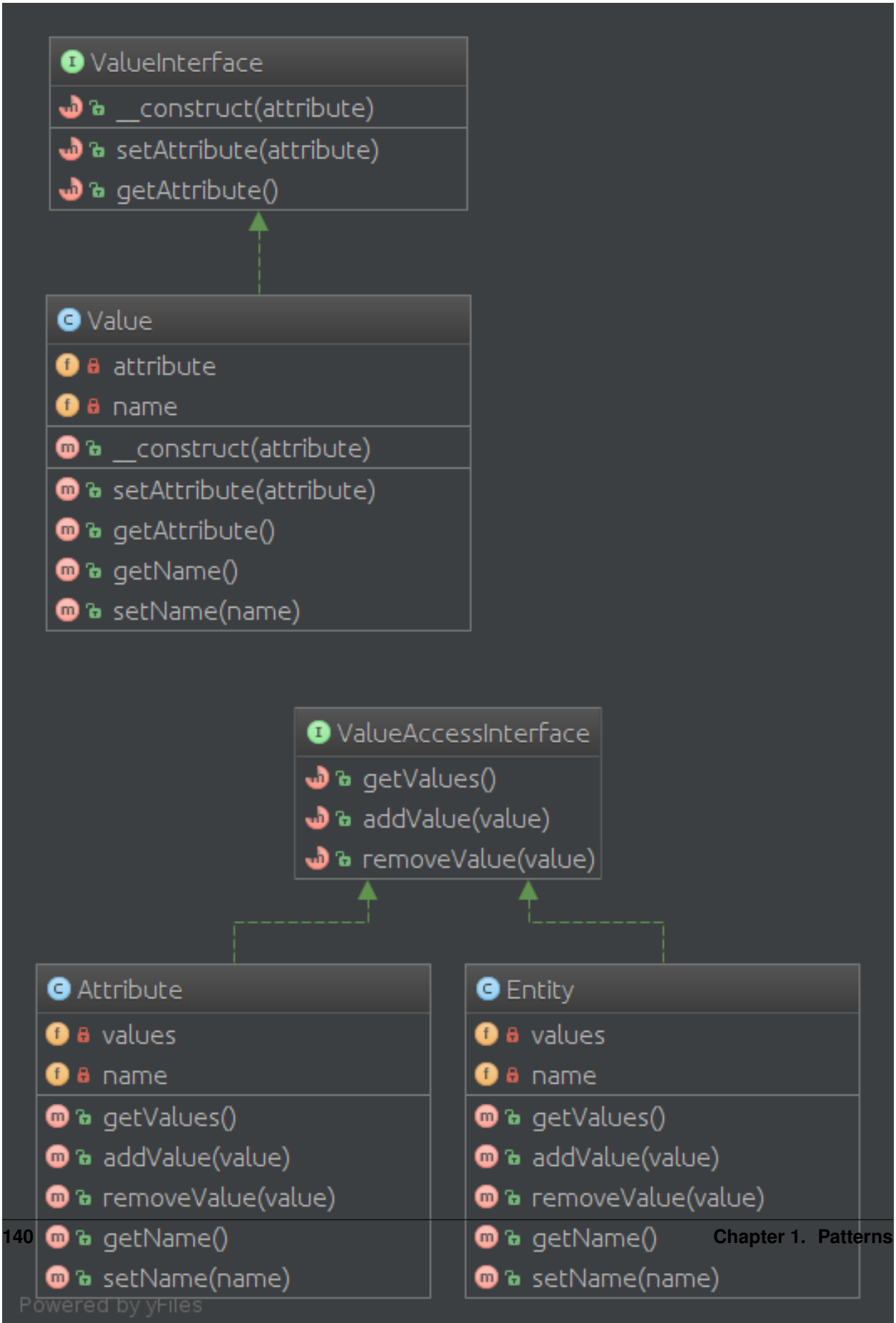
1.4.3 Entity-Attribute-Value (EAV)

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


ValueInterface
__construct(attribute)
setAttribute(attribute)
getAttribute()
Value
attribute
name
__construct(attribute)
setAttribute(attribute)
getAttribute()
getName()
setName(name)
ValueAccessInterface
getValues()
addValue(value)
removeValue(value)
Attribute
values
name
getValues()
addValue(value)
removeValue(value)
getName()
setName(name)
Entity
values
name
getValues()
addValue(value)
removeValue(value)
getName()
setName(name)
Powered by yFiles

Code

You can also find this code on GitHub

Entity.php

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

namespace DesignPatterns\More\EAV;

class Entity
{
    /**
     * @var \SplObjectStorage
     */
    private $values;

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

    /**
     * @param string $name
     * @param Value[] $values
     */
    public function __construct(string $name, $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
<?php
declare(strict_types=1);

namespace DesignPatterns\More\EAV;

class Attribute
{
```

(continues on next page)

(continued from previous page)

```
    /**
     * @var \SplObjectStorage
     */
    private $values;

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

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

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

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

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

Value.php

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

namespace DesignPatterns\More\EAV;

class Value
{
    /**
     * @var Attribute
     */
    private $attribute;

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

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

(continues on next page)

(continued from previous page)

```
        $this->attribute = $attribute;

        $attribute->addValue($this);
    }

    public function __toString(): string
    {
        return sprintf('%s: %s', $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');
        $memory8Gb = new Value($memoryAttribute, '8GB');

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

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

CHAPTER 2

Contribute

If you encounter any bugs or missing translations, please feel free to fork and send a pull request with your changes. To establish a consistent code quality, please check your code using PHP CodeSniffer against PSR2 standard using `./vendor/bin/phpcs -p --standard=PSR2 --ignore=vendor .`.