DesignPatternsPHP Documentation

Выпуск 1.0

Dominik Liebler and contributors

апр. 16, 2019

Оглавление

Это набор известных шаблонов проектирования (паттернов) и некоторые примеры их реализации в PHP. Каждый паттерн содержит небольшой перечень примеров (большинство из них для ZendFramework, Symfony2 или Doctrine2, так как я лучше всего знаком с этим программным обеспечением).

Я считаю, проблема паттернов в том, что люди часто знакомы с ними, но не представляют как их применять.

Глава 1

Паттерны

Паттерны могут быть условно сгруппированы в три различные категории. Нажмите на **заголовок каждой страницы с паттерном** для детального объяснения паттерна в Википедии.

1.1 Порождающие шаблоны проектирования (Creational)

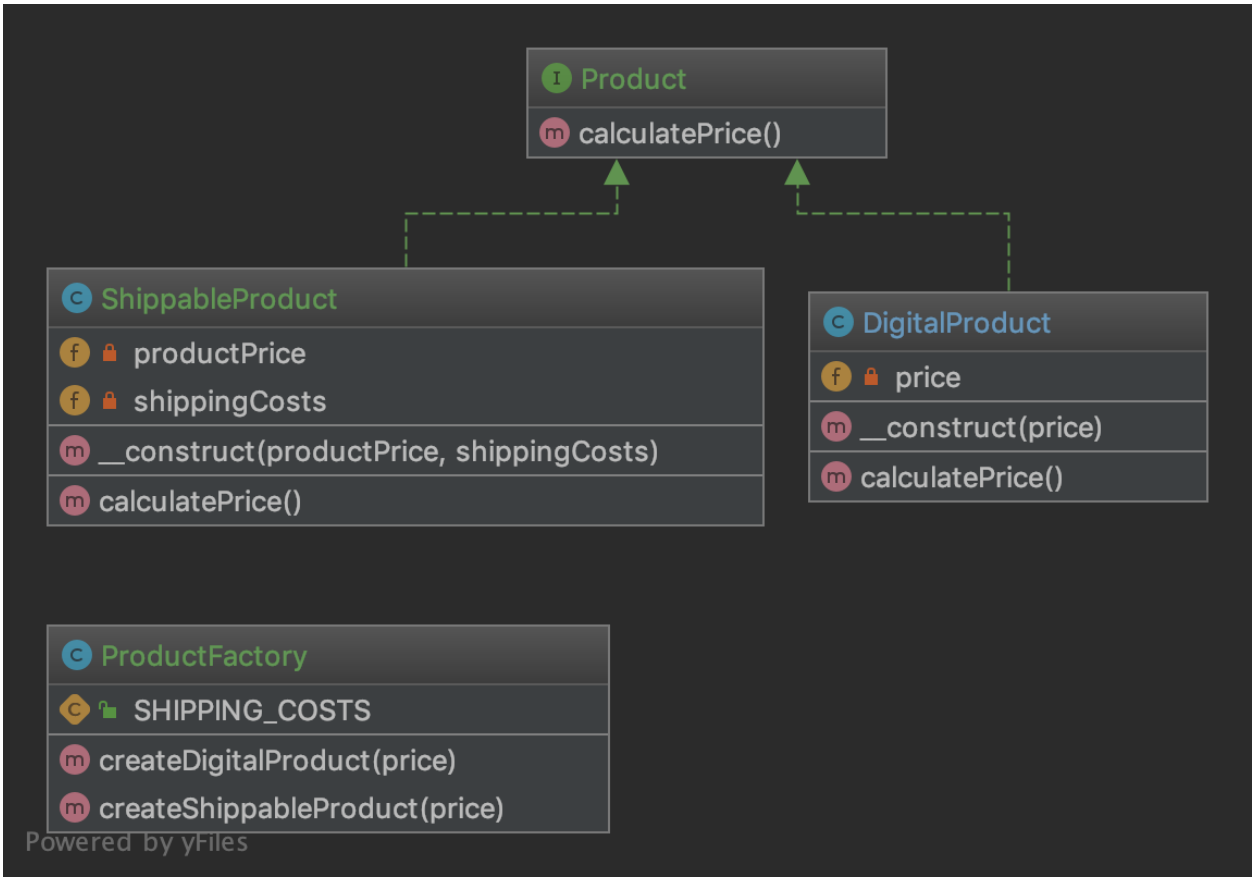
В разработке программного обеспечения, Порождающие шаблоны проектирования – это паттерны, которые имеют дело с механизмами создания объекта и пытаются создать объекты в порядке, подходящем к ситуации. Обычная форма создания объекта может привести к проблемам проектирования или увеличивать сложность конструкции. Порождающие шаблоны проектирования решают эту проблему, определённым образом контролируя процесс создания объекта.

1.1.1 Абстрактная фабрика (Abstract Factory)

Назначение

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.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Product.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

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

ShippableProduct.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

class ShippableProduct implements Product
{
    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     * @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

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

namespace DesignPatterns\Creational\AbstractFactory;

class ProductFactory
{
    const SHIPPING_COSTS = 50;

    public function createShippableProduct(int $price): Product
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    {
        return new ShippableProduct($price, self::SHIPPING_COSTS);
    }

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

Тест

Tests/AbstractFactoryTest.php

```
<?php

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

        $this->assertEquals(200, $product->calculatePrice());
```

(continues on next page)

(продолжение с предыдущей страницы)

```
42      }
43  }
```

1.1.2 Строитель (Builder)

Назначение

Строитель — это интерфейс для производства частей сложного объекта.

Иногда, если Строитель лучше знает о том, что он строит, этот интерфейс может быть абстрактным классом с методами по-умолчанию (адаптер).

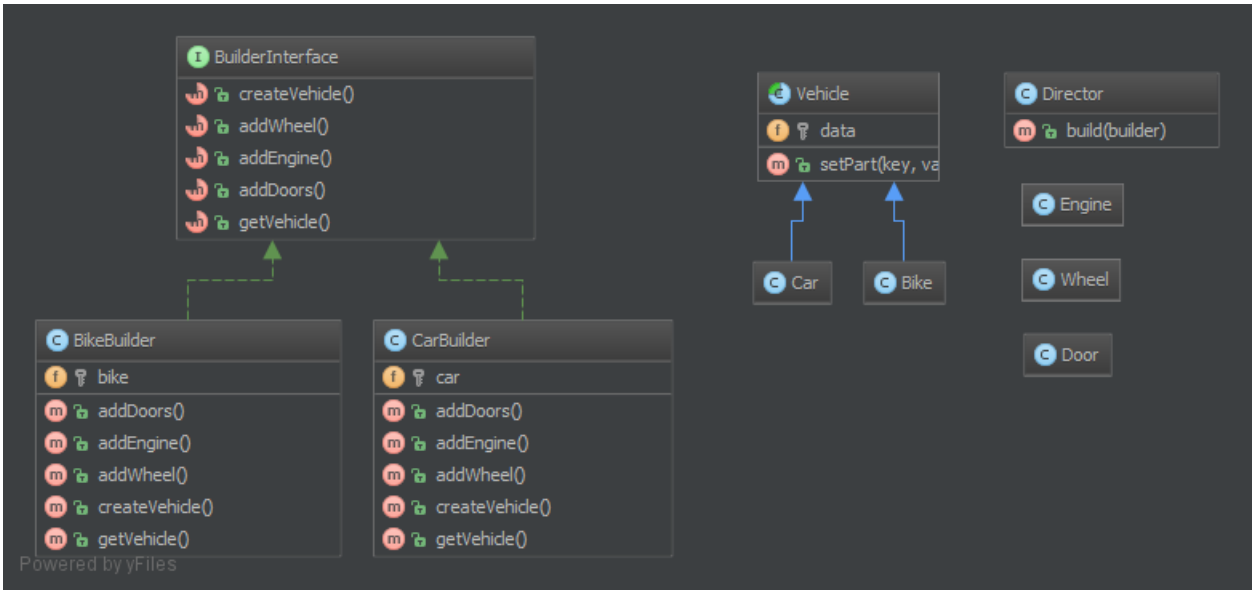
Если у вас есть сложное дерево наследования для объектов, логично иметь сложное дерево наследования и для их строителей.

Примечание: Строители могут иметь текучий интерфейс, например, строитель макетов в PHPUnit.

Примеры

- PHPUnit: Mock Builder

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Director.php

```php
<?php

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(BuilderInterface $builder): Vehicle
    {
        $builder->createVehicle();
        $builder->addDoors();
        $builder->addEngine();
        $builder->addWheel();

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

BuilderInterface.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

interface BuilderInterface
{
    public function createVehicle();

    public function addWheel();

    public function addEngine();

    public function addDoors();

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

TruckBuilder.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

class TruckBuilder implements BuilderInterface
{
    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     * @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());
    }

    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

namespace DesignPatterns\Creational\Builder;

use DesignPatterns\Creational\Builder\Parts\Vehicle;

class CarBuilder implements BuilderInterface
{
    /**
     * @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());
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    }

    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());
    }

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

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

Parts/Vehicle.php

```
<?php

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

namespace DesignPatterns\Creational\Builder\Parts;

class Truck extends Vehicle
```

(continues on next page)

(продолжение с предыдущей страницы)

```
{
}
```

Parts/Car.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Car extends Vehicle
{
}
```

Parts/Engine.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Engine
{
}
```

Parts/Wheel.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Wheel
{
}
```

Parts/Door.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

class Door
{
}
```

Тест

Tests/DirectorTest.php

```php
<?php

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;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
use DesignPatterns\Creational\Builder\Director;
use PHPUnit\Framework\TestCase;

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

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

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

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

1.1.3 Фабричный Метод (Factory Method)

Назначение

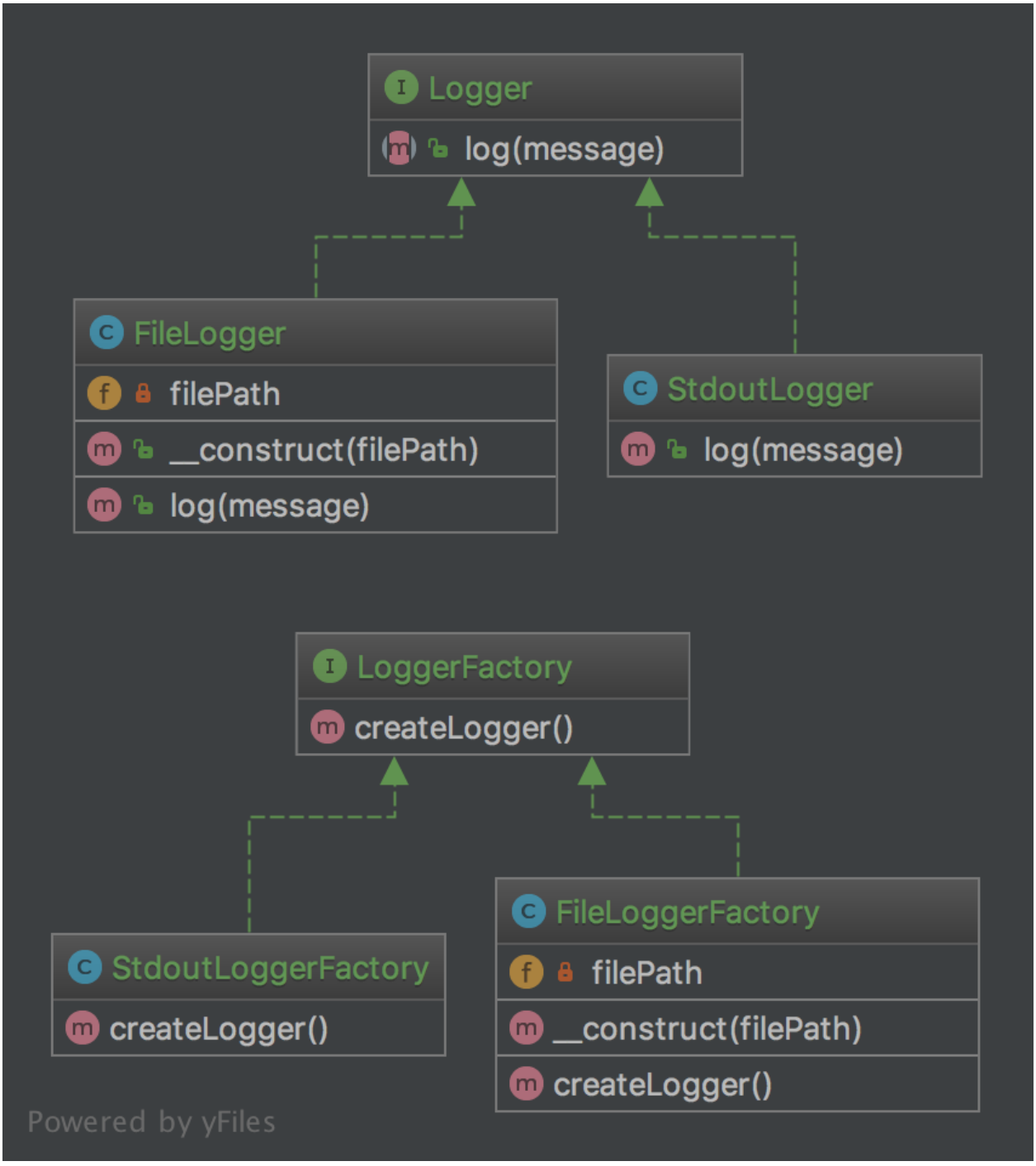
Выгодное отличие от SimpleFactory в том, что вы можете вынести реализацию создания объектов в подклассы.

В простых случаях, этот абстрактный класс может быть только интерфейсом.

Этот паттерн является «настоящим» Шаблоном Проектирования, потому что он следует «Принципу инверсии зависимостей» также известному как «D» в S.O.L.I.D.

Это означает, что класс FactoryMethod зависит от абстракций, а не от конкретных классов. Это существенный плюс в сравнении с SimpleFactory или StaticFactory.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Logger.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

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

StdoutLogger.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

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

FileLogger.php

```php
<?php

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

namespace DesignPatterns\Creational\FactoryMethod;

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

StdoutLoggerFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

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

FileLoggerFactory.php

```php
<?php

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);
    }
}
```

Тест

Tests/FactoryMethodTest.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod\Tests;

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

class FactoryMethodTest extends TestCase
{
    public function testCanCreateStdoutLogging()
    {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
15          $loggerFactory = new StdoutLoggerFactory();
16          $logger = $loggerFactory->createLogger();
17
18          $this->assertInstanceOf(StdoutLogger::class, $logger);
19      }
20
21      public function testCanCreateFileLogging()
22      {
23          $loggerFactory = new FileLoggerFactory(sys_get_temp_dir());
24          $logger = $loggerFactory->createLogger();
25
26          $this->assertInstanceOf(FileLogger::class, $logger);
27      }
28  }
```

1.1.4 Пул одиночек (Multiton)

Этот шаблон считается анти-паттерном! Для лучшей тестируемости и сопровождения кода используйте внедрение зависимости (Dependency Injection)!

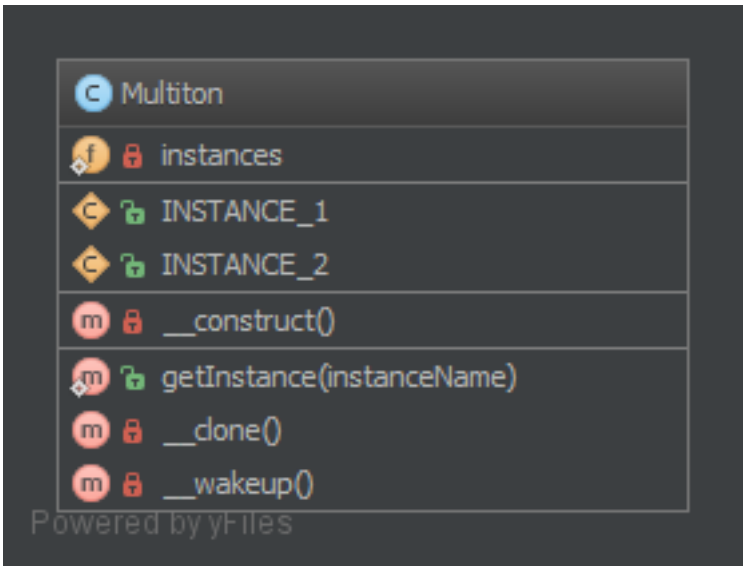
Назначение

Содержит список именованных созданных экземпляров классов, которые в итоге используются как Singleton-ы, но в заданном заранее N-ном количестве.

Примеры

- Два объекта для доступа к базам данных, к примеру, один для MySQL, а второй для SQLite
- Несколько логгирующих объектов (один для отладочных сообщений, другой для ошибок и т.п.)

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Multiton.php

```php
<?php

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];
    }

    /**
     * prevent instance from being cloned
     */
    private function __clone()
    {
    }

    /**
     * prevent instance from being unserialized
     */
    private function __wakeup()
    {
    }
}
```

Тест

1.1.5 Объектный пул (Pool)

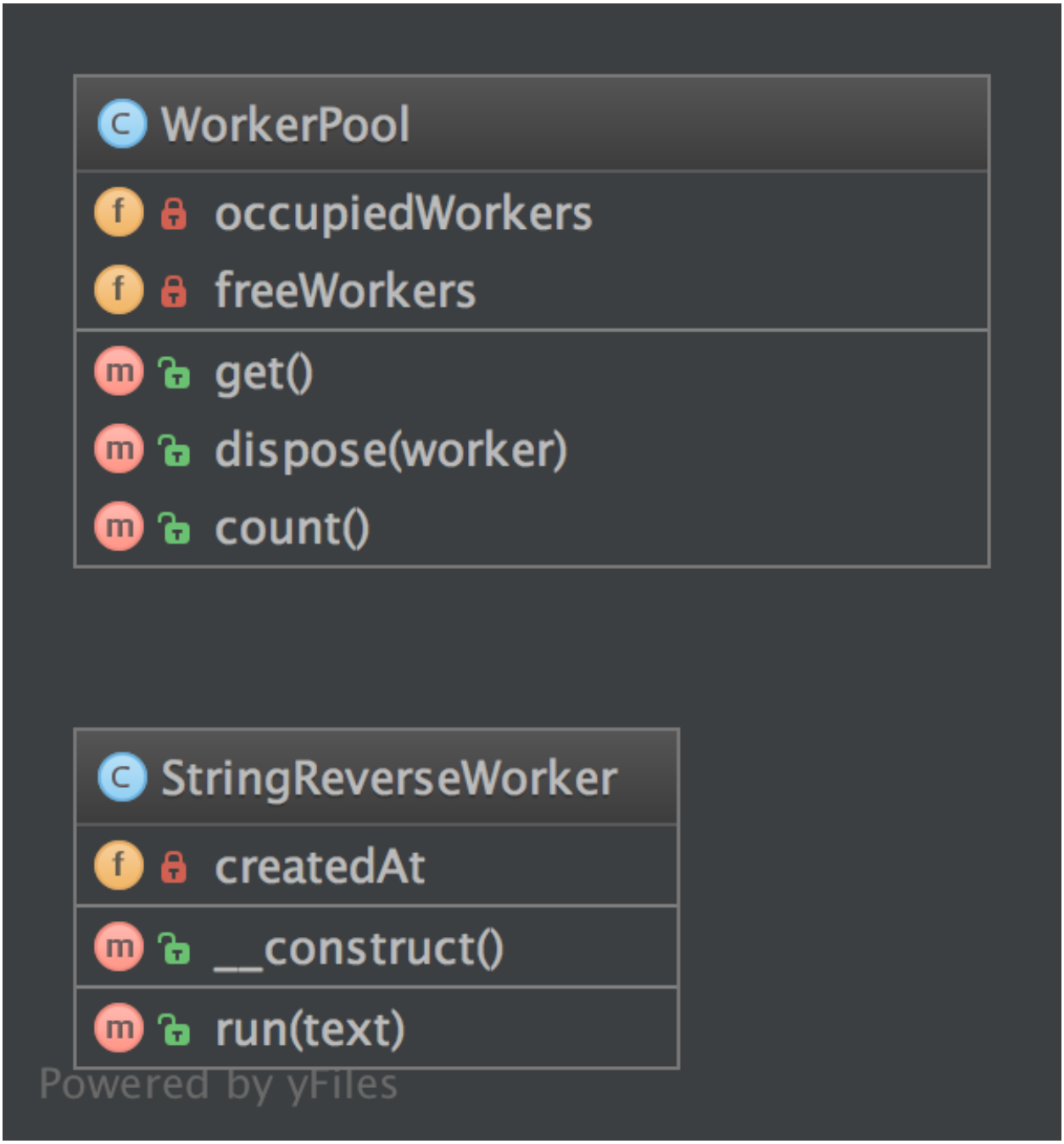
Назначение

Порождающий паттерн, который предоставляет набор заранее инициализированных объектов, готовых к использованию («пул»), что не требует каждый раз создавать и уничтожать их.

Хранение объектов в пуле может заметно повысить производительность в ситуациях, когда стоимость инициализации экземпляра класса высока, скорость экземпляра класса высока, а количество одновременно используемых экземпляров в любой момент времени является низкой. Время на извлечение объекта из пула легко прогнозируется, в отличие от создания новых объектов (особенно с сетевым оверхедом), что занимает неопределённое время.

Однако эти преимущества в основном относится к объектам, которые изначально являются дорогостоящими по времени создания. Например, соединения с базой данных, соединения сокетов, потоков и инициализация больших графических объектов, таких как шрифты или растровые изображения. В некоторых ситуациях, использование простого пула объектов (которые не зависят от внешних ресурсов, а только занимают память) может оказаться неэффективным и приведёт к снижению производительности.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

WorkerPool.php

```
1 <?php
2
```

(continues on next page)

(продолжение с предыдущей страницы)

```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

namespace DesignPatterns\Creational\Pool;

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

```

(continues on next page)

(продолжение с предыдущей страницы)

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

Тест

Tests/PoolTest.php

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

1.1.6 Прототип (Prototype)

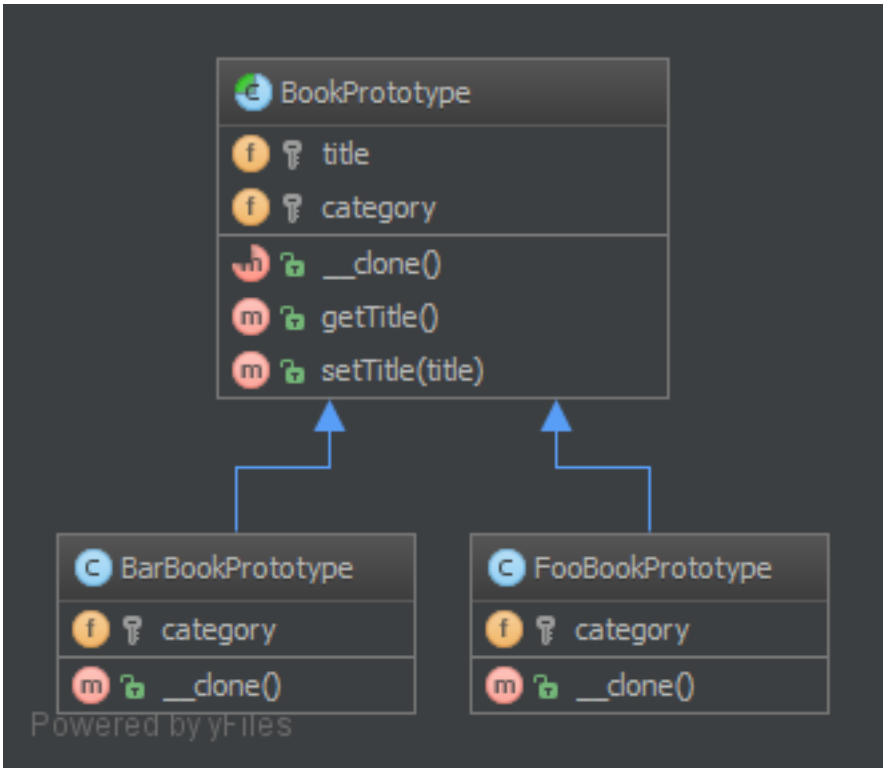
Назначение

Помогает избежать затрат на создание объектов стандартным способом (new Foo()), а вместо этого создаёт прототип и затем клонирует его.

Примеры

- Большие объемы данных (например, создать 1000000 строк в базе данных сразу через ORM).

Диаграмма UML



Код

Вы можете найти этот код на GitHub

BookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

abstract class BookPrototype
{
    /**
     * @var string
     */
    protected $title;

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

```

(continues on next page)

(продолжение с предыдущей страницы)

```
    abstract public function __clone();

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

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

BarBookPrototype.php

```
<?php

namespace DesignPatterns\Creational\Prototype;

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

    public function __clone()
    {
    }
}
```

FooBookPrototype.php

```
<?php

namespace DesignPatterns\Creational\Prototype;

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

    public function __clone()
    {
    }
}
```

Тест

Tests/PrototypeTest.php

```
<?php

```

(continues on next page)

(продолжение с предыдущей страницы)

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

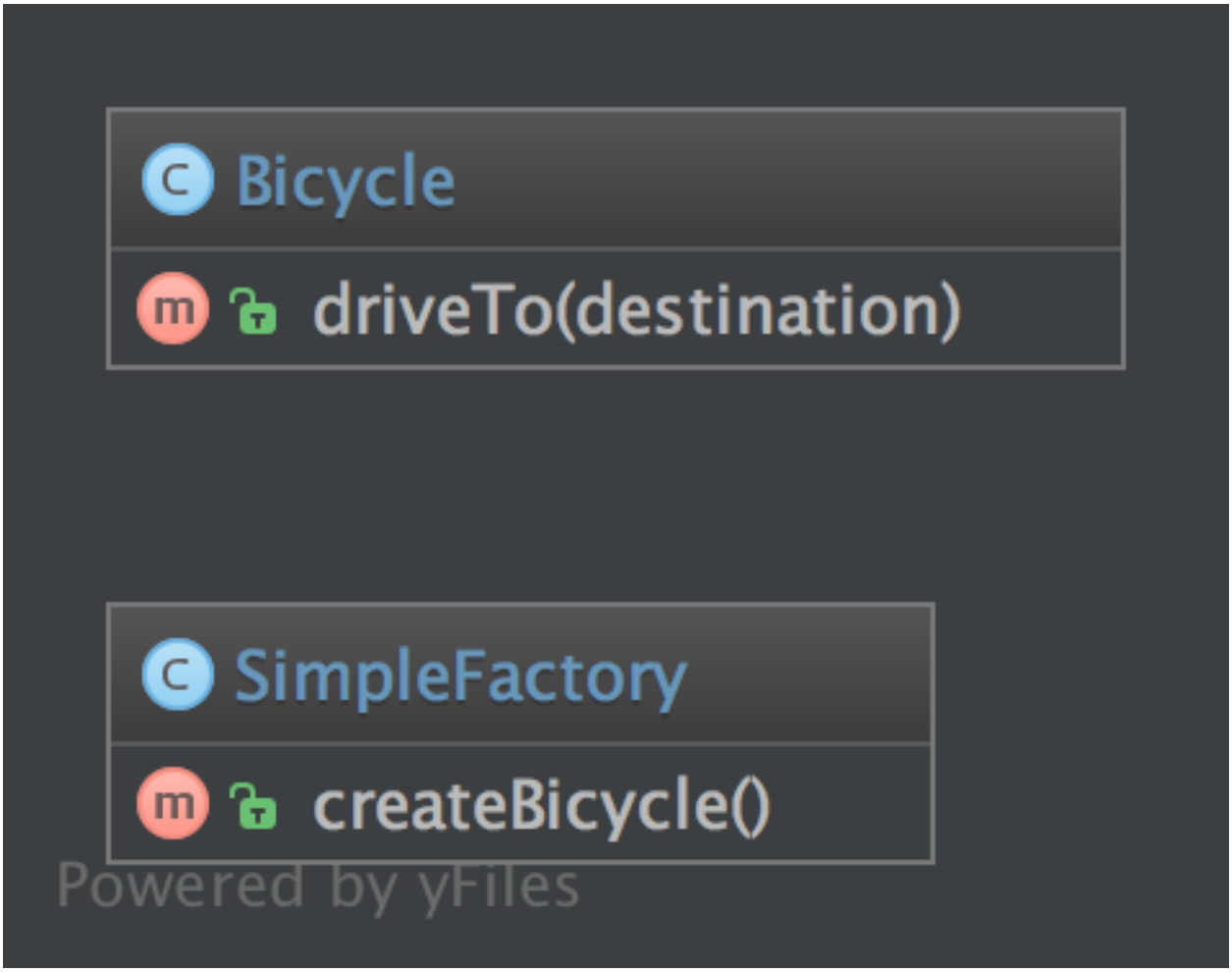
1.1.7 Простая Фабрика (Simple Factory)

Назначение

SimpleFactory в примере ниже, это паттерн «Простая Фабрика».

Она отличается от Статической Фабрики тем, что собственно *не является статической*. Таким образом, вы можете иметь множество фабрик с разными параметрами. Простая фабрика всегда должна быть предпочтительнее Статической фабрики!

Диаграмма UML



Код

Вы можете найти этот код на GitHub

SimpleFactory.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

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

Bicycle.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

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

Usage

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

Тест

Tests/SimpleFactoryTest.php

```php
<?php

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 Одиночка (Singleton)

Это считается анти-паттерном! Для лучшей тестируемости и сопровождения кода используйте Инъекцию Зависимости (Dependency Injection)!

Назначение

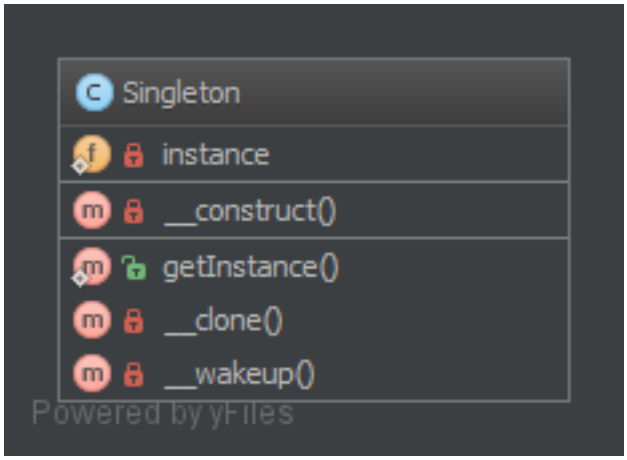
Позволяет содержать только один экземпляр объекта в приложении, которое будет обрабатывать все обращения, запрещая создавать новый экземпляр.

Примеры

- DB Connector для подключения к базе данных

- Logger (также может быть Multiton если есть много журналов для нескольких целей)
- Блокировка файла в приложении (есть только один в файловой системе с одновременным доступом к нему)

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Singleton.php

```php
<?php

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;
    }

    /**
     * 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
     */
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    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()
    {
    }
}
```

Тест

Tests/SingletonTest.php

```
<?php

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 Статическая Фабрика (Static Factory)

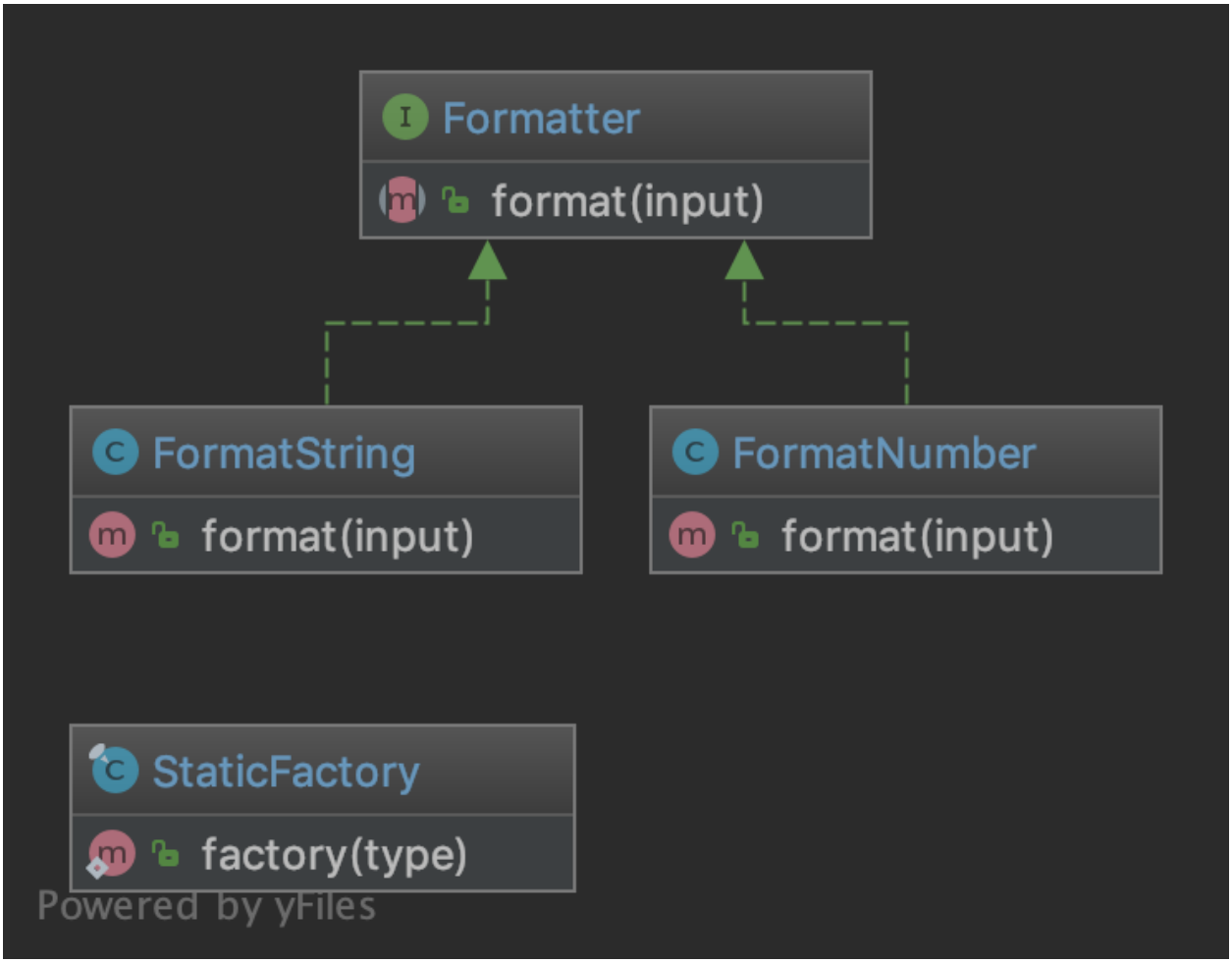
Назначение

Подобно AbstractFactory, этот паттерн используется для создания ряда связанных или зависимых объектов. Разница между этим шаблоном и Абстрактной Фабрикой заключается в том, что Статическая Фабрика использует только один статический метод, чтобы создать все допустимые типы объектов. Этот метод, обычно, называется `factory` или `build`.

Примеры

- Zend Framework: `Zend_Cache_Backend` or `_Frontend` use a factory method to create cache backends and frontends

Диаграмма UML



Код

Вы можете найти этот код на GitHub

StaticFactory.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

/**
 * Note1: Remember, static means global state which is evil because it can't be mocked for tests
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances.
 */
```

(continues on next page)

(продолжение с предыдущей страницы)

```php
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
<?php

namespace DesignPatterns\Creational\StaticFactory;

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

FormatString.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

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

FormatNumber.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

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

Тест

Tests/StaticFactoryTest.php

```php
<?php

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')
        );
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testException()
    {
        StaticFactory::factory('object');
    }
}
```

1.2 Структурные шаблоны проектирования (Structural)

При разработке программного обеспечения, Структурные шаблоны проектирования упрощают проектирование путем выявления простого способа реализовать отношения между субъектами.

1.2.1 Адаптер (Adapter / Wrapper)

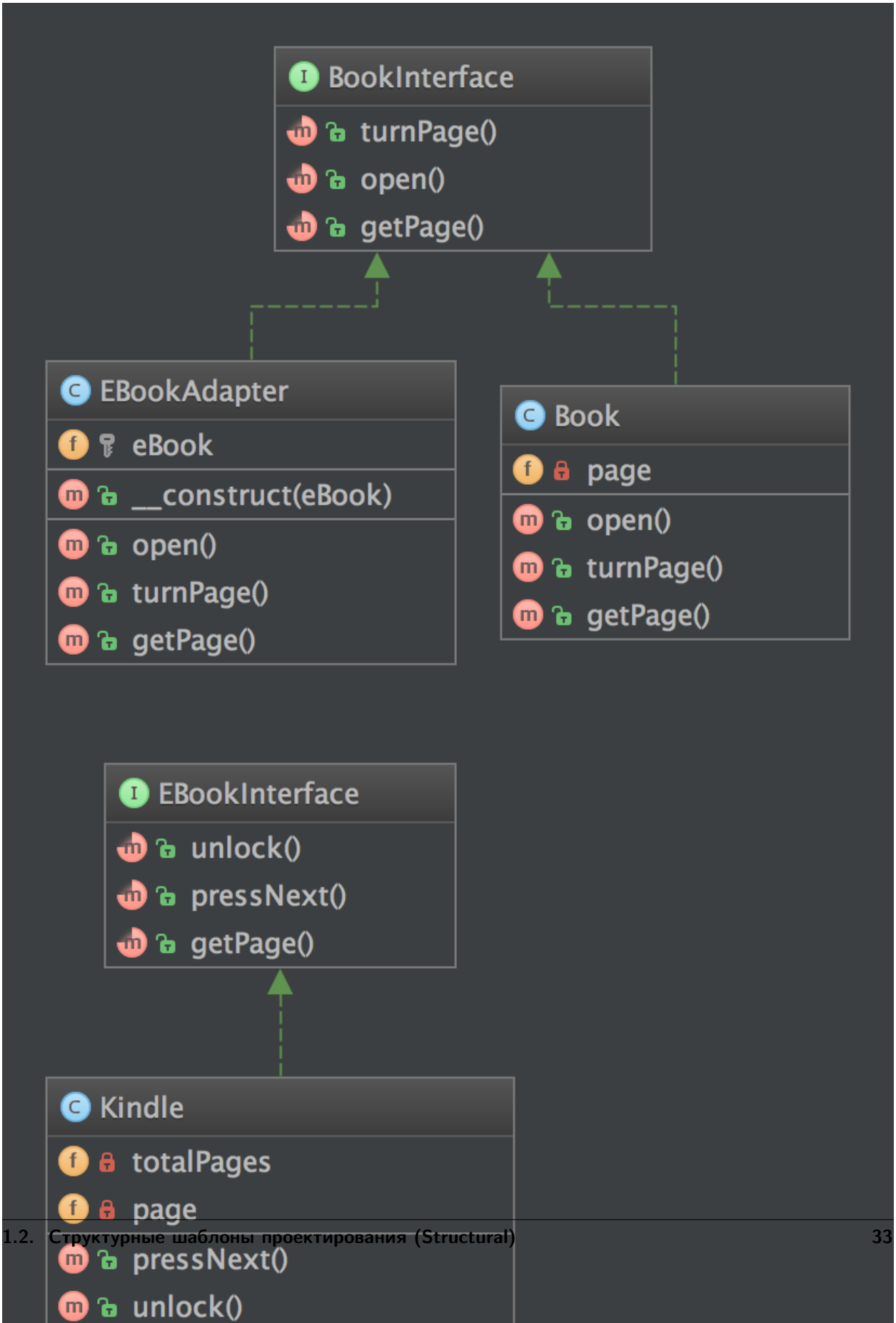
Назначение

Привести нестандартный или неудобный интерфейс какого-то класса в интерфейс, совместимый с вашим кодом. Адаптер позволяет классам работать вместе стандартным образом, что обычно не получается из-за несовместимых интерфейсов, предоставляя для этого прослойку с интерфейсом, удобным для клиентов, самостоятельно используя оригинальный интерфейс.

Примеры

- Адаптер клиентских библиотек для работы с базами данных
- нормализовать данные нескольких различных веб-сервисов, в одинаковую структуру, как будто вы работаете со стандартным сервисом (например при работе с API соцсетей)

Диаграмма UML



Код

Вы можете найти этот код на GitHub

BookInterface.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

interface BookInterface
{
    public function turnPage();

    public function open();

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

Book.php

```php
<?php

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

namespace DesignPatterns\Structural\Adapter;

/**
 * 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
 */
```

(continues on next page)

(продолжение с предыдущей страницы)

```php
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

namespace DesignPatterns\Structural\Adapter;

interface EBookInterface
{
    public function unlock();

    public function pressNext();

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

(continues on next page)

(продолжение с предыдущей страницы)

```
     * @return int[]
     */
    public function getPage(): array;
}
```

Kindle.php

```
<?php

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];
    }
}
```

Тест

Tests/AdapterTest.php

```
<?php

```

(continues on next page)

(продолжение с предыдущей страницы)

```php
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 Мост (Bridge)

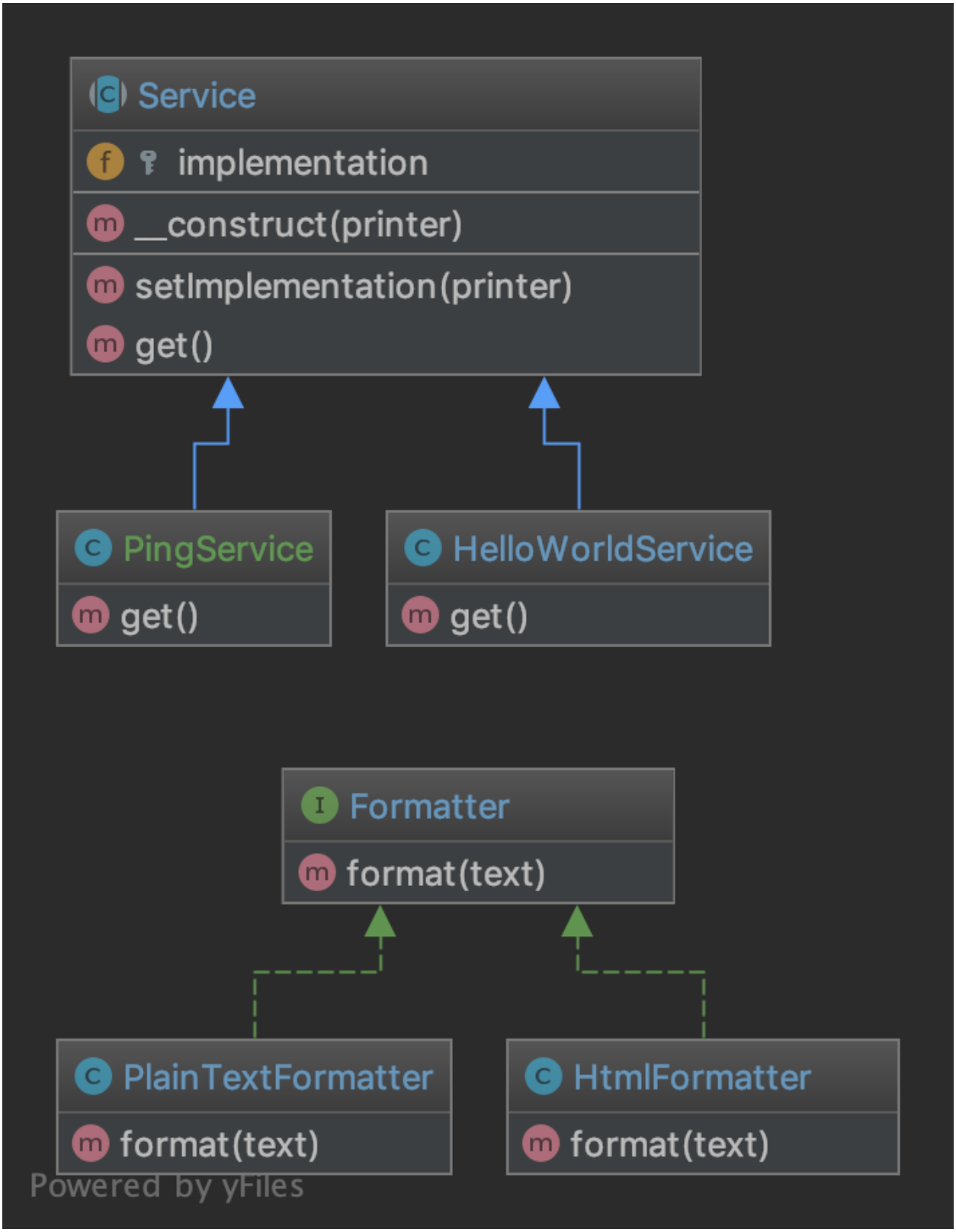
Назначение

Отделить абстракцию от её реализации так, что они могут изменяться независимо друг от друга.

Examples

- Symfony DoctrineBridge

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Formatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

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

PlainTextFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

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

HtmlFormatter.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

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

Service.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

abstract class Service
{
    /**
     * @var Formatter
     */
    protected $implementation;

    /**
     * @param Formatter $printer
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    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

namespace DesignPatterns\Structural\Bridge;

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

PingService.php

```
<?php

namespace DesignPatterns\Structural\Bridge;

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

Тест

Tests/BridgeTest.php

```
<?php

namespace DesignPatterns\Structural\Bridge\Tests;

use DesignPatterns\Structural\Bridge\HelloWorldService;
use DesignPatterns\Structural\Bridge\HtmlFormatter;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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 Компоновщик (Composite)

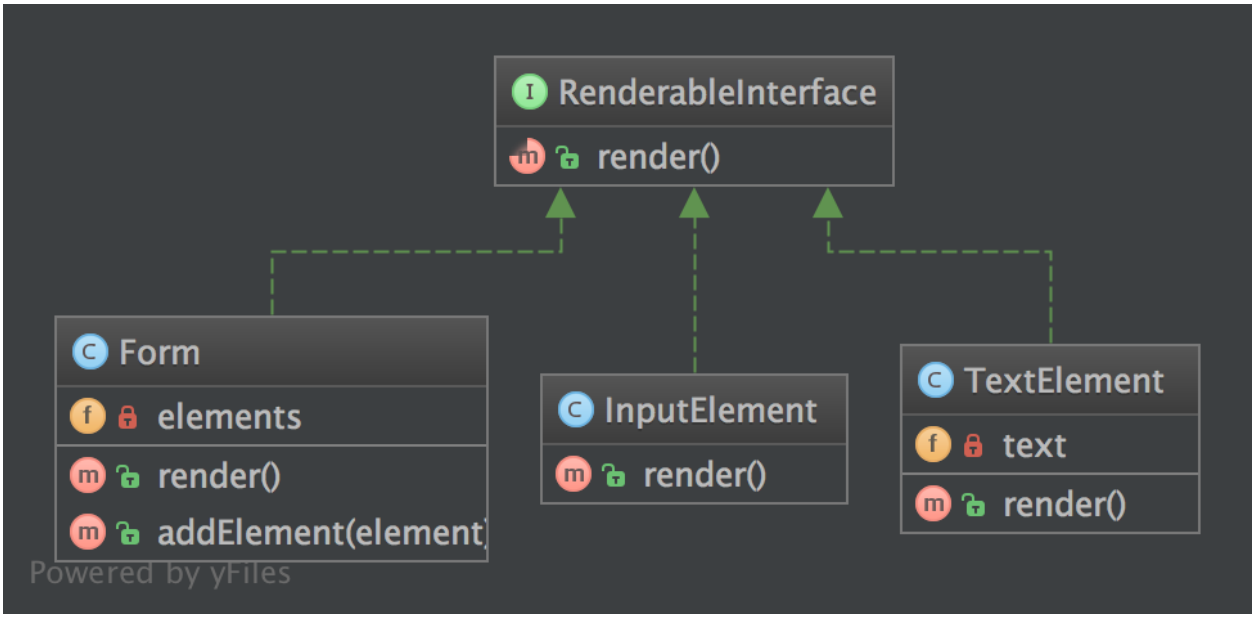
Назначение

Взаимодействие с иерархической группой объектов также, как и с отдельно взятым экземпляром.

Примеры

- Экземпляр класса Form обрабатывает все свои элементы формы, как будто это один экземпляр. И когда вызывается метод `render()`, он перебирает все дочерние элементы и вызывает их собственный `render()`.
- `Zend_Config`: дерево вариантов конфигурации, где каждая из конфигураций тоже представляет собой объект `Zend_Config`

Диаграмма UML



Код

Вы можете найти этот код на GitHub

RenderableInterface.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

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

Form.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

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

    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

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

InputElement.php

```
<?php

namespace DesignPatterns\Structural\Composite;

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

TextElement.php

```
<?php

namespace DesignPatterns\Structural\Composite;

class TextElement implements RenderableInterface
{
    /**
     * @var string
     */
    private $text;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
11
12      public function __construct(string $text)
13      {
14          $this->text = $text;
15      }
16
17      public function render(): string
18      {
19          return $this->text;
20      }
21  }
```

Тест

Tests/CompositeTest.php

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

1.2.4 Преобразователь Данных (Data Mapper)

Назначение

Преобразователь Данных — это паттерн, который выступает в роли посредника для двунаправленной передачи данных между постоянным хранилищем данных (часто, реляционной базы данных) и представления данных в памяти (слой домена, то что уже загружено и используется для логической

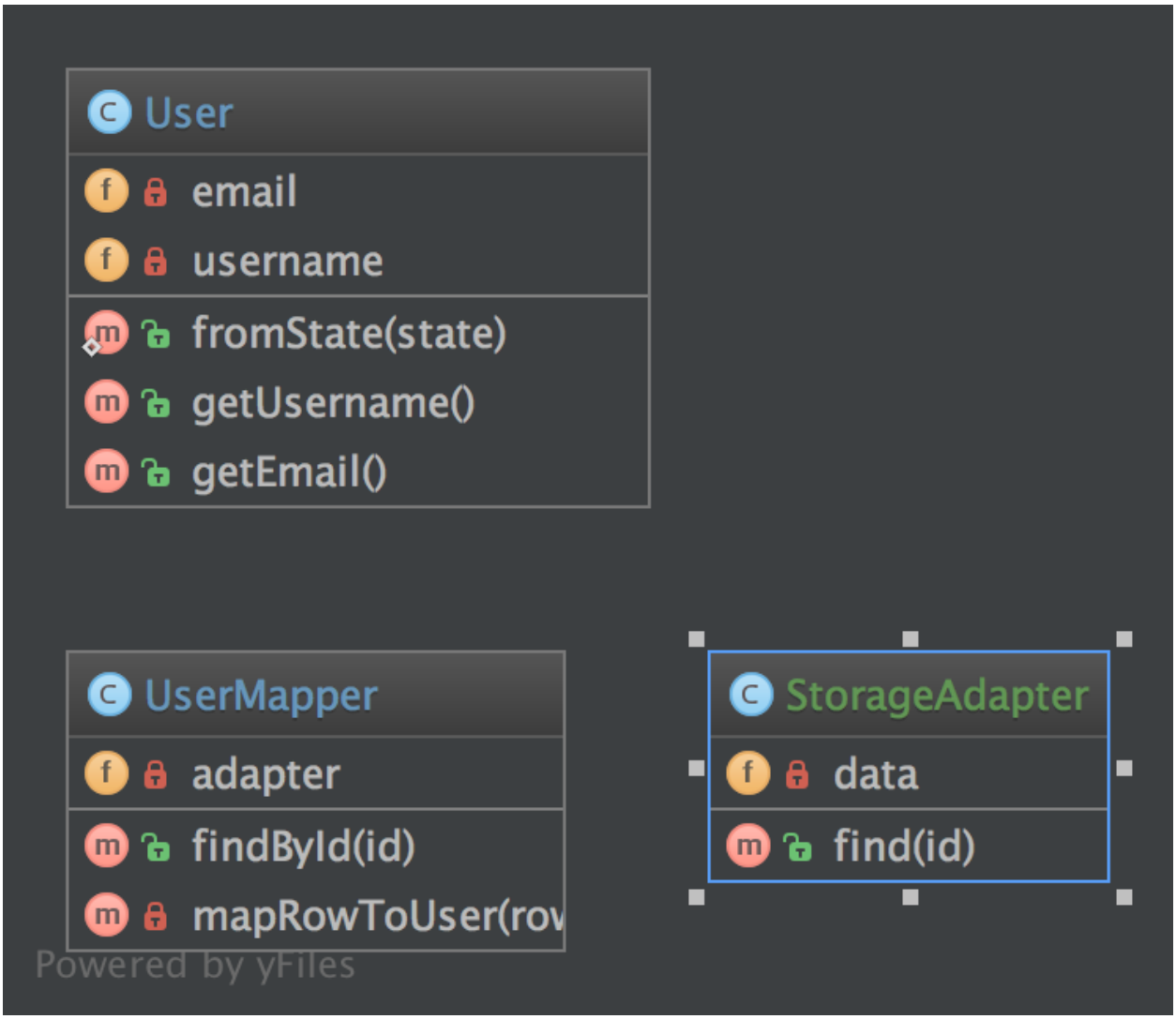
обработки). Цель паттерна в том, чтобы держать представление данных в памяти и постоянное хранилище данных независимыми друг от друга и от самого преобразователя данных. Слой состоит из одного или более mapper-а (или объектов доступа к данным), отвечающих за передачу данных. Реализации mapper-ов различаются по назначению. Общие mapper-ы могут обрабатывать всевозможные типы сущностей доменов, а выделенные mapper-ы будет обрабатывать один или несколько конкретных типов.

Ключевым моментом этого паттерна, в отличие от Активной Записи (Active Records) является то, что модель данных следует Принципу Единой Обязанности SOLID.

Примеры

- DB Object Relational Mapper (ORM) : Doctrine2 использует DAO под названием «EntityRepository»

Диаграмма UML



Код

Вы можете найти этот код на GitHub

User.php

```php
<?php

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

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

namespace DesignPatterns\Structural\DataMapper;

class StorageAdapter
{
    /**
     * @var array
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    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;
    }
}
```

Тест

Tests/DataMapperTest.php

```
<?php

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);
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testWillNotMapInvalidData()
    {
        $storage = new StorageAdapter([]);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
28          $mapper = new UserMapper($storage);
29
30          $mapper->findById(1);
31      }
32  }
```

1.2.5 Декоратор (Decorator)

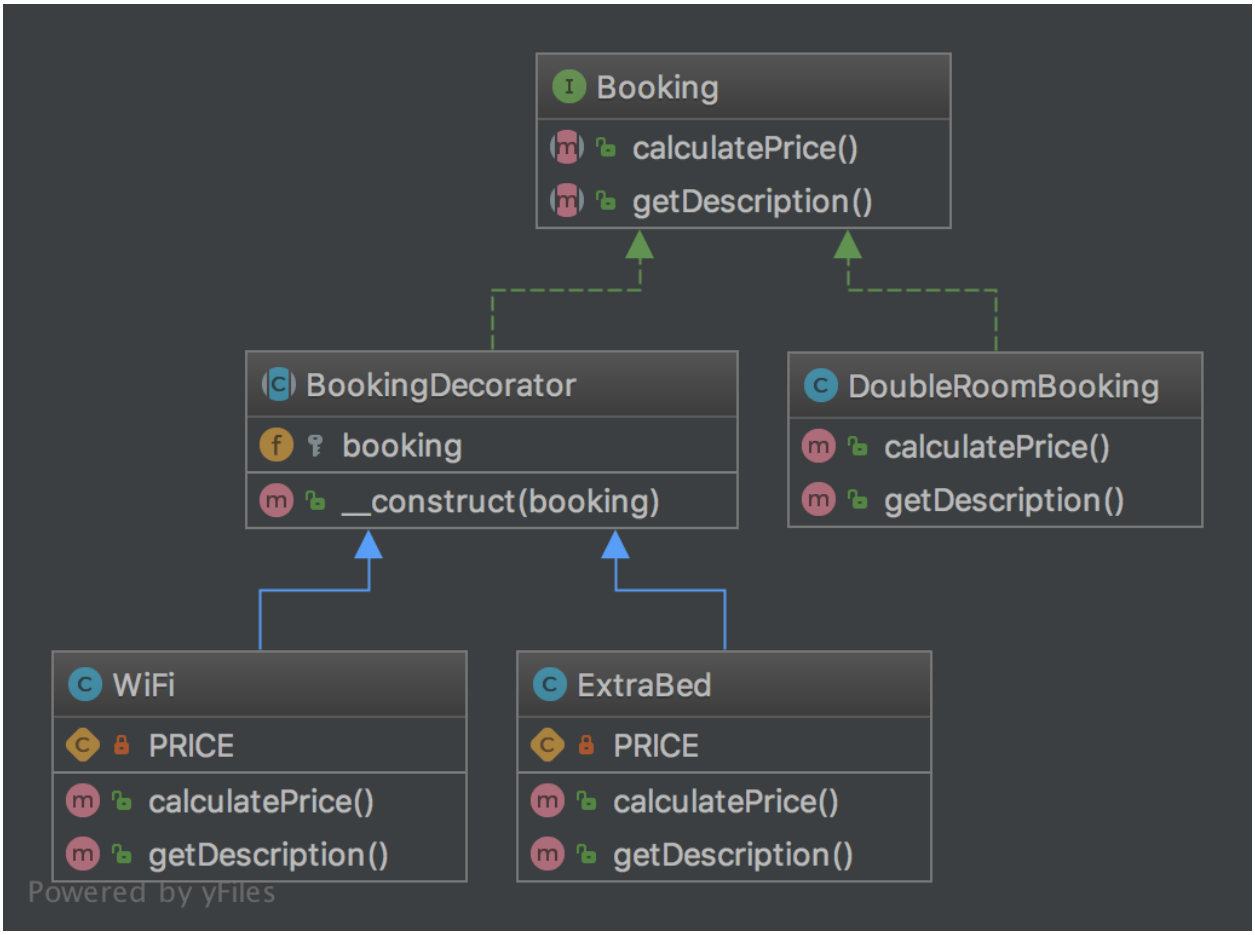
Назначение

Динамически добавляет новую функциональность в экземпляры классов.

Примеры

- Zend Framework: декораторы для экземпляров `Zend_Form_Element`
- Web Service Layer: Декораторы JSON и XML для REST сервисов (в этом случае, конечно, только один из них может быть разрешен).

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Booking.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

interface Booking
{
    public function calculatePrice(): int;

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

BookingDecorator.php

```php
<?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

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

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

namespace DesignPatterns\Structural\Decorator;

class WiFi extends BookingDecorator
{
    private const PRICE = 2;

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

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

Тест

Tests/DecoratorTest.php

```php
<?php

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()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
13      {
14          $booking = new DoubleRoomBooking();
15
16          $this->assertSame(40, $booking->calculatePrice());
17          $this->assertSame('double room', $booking->getDescription());
18      }
19
20      public function testCanCalculatePriceForDoubleRoomBookingWithWiFi()
21      {
22          $booking = new DoubleRoomBooking();
23          $booking = new WiFi($booking);
24
25          $this->assertSame(42, $booking->calculatePrice());
26          $this->assertSame('double room with wifi', $booking->getDescription());
27      }
28
29      public function testCanCalculatePriceForDoubleRoomBookingWithWiFiAndExtraBed()
30      {
31          $booking = new DoubleRoomBooking();
32          $booking = new WiFi($booking);
33          $booking = new ExtraBed($booking);
34
35          $this->assertSame(72, $booking->calculatePrice());
36          $this->assertSame('double room with wifi with extra bed', $booking->getDescription());
37      }
38  }
```

1.2.6 Внедрение Зависимости (Dependency Injection)

Назначение

Для реализации слабосвязанной архитектуры. Чтобы получить более тестируемый, сопровождаемый и расширяемый код.

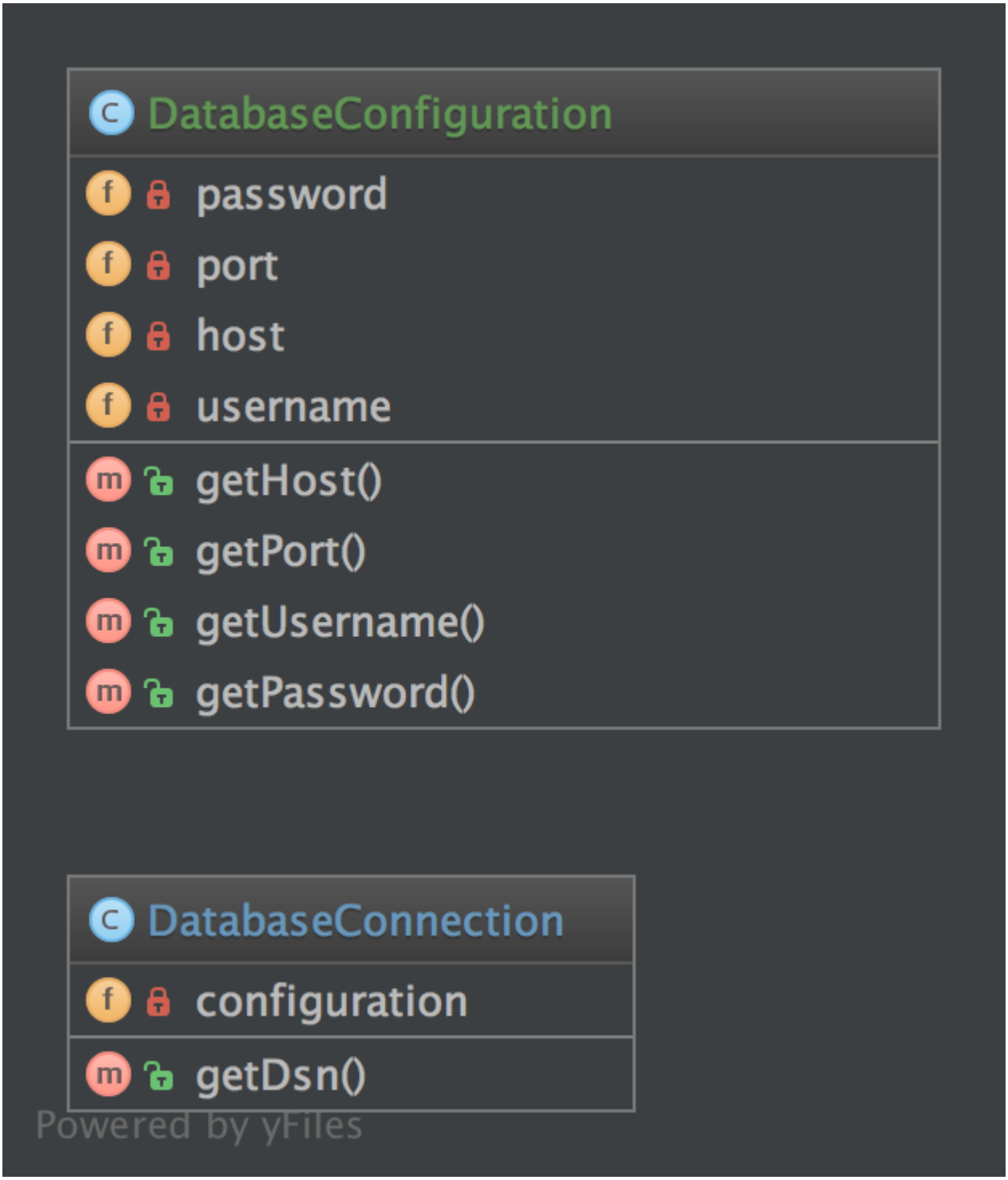
Использование

Объект `DatabaseConfiguration` внедряется в `DatabaseConnection` и последний получает всё, что ему необходимо из переменной `$ config`. Без DI, конфигурация будет создана непосредственно в `Connection`, что не очень хорошо для тестирования и расширения `Connection`, так как связывает эти классы напрямую.

Примеры

- The Doctrine2 ORM использует Внедрение Зависимости например для конфигурации, которая внедряется в объект `Connection`. Для целей тестирования, можно легко создать макет объекта конфигурации и внедрить его в объект `Connection`, подменив оригинальный.
- Symfony и Zend Framework 2 уже содержат контейнеры для DI, которые создают объекты с помощью массива из конфигурации, и внедряют их в случае необходимости (т.е. в Контроллерах).

Диаграмма UML



Код

Вы можете найти этот код на GitHub

DatabaseConfiguration.php

```php
<?php

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

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()
        );
    }
}
```

Тест

Tests/DependencyInjectionTest.php

```php
<?php

namespace DesignPatterns\Structural\DependencyInjection\Tests;

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
18 }
```

1.2.7 Фасад (Facade)

Назначение

Основная цель паттерна Фасад заключается не в том, чтобы помешать вам прочитать инструкцию комплексной API. Это только побочный эффект. Главная цель всё же состоит в уменьшении связности кода и соблюдении Закона Деметры.

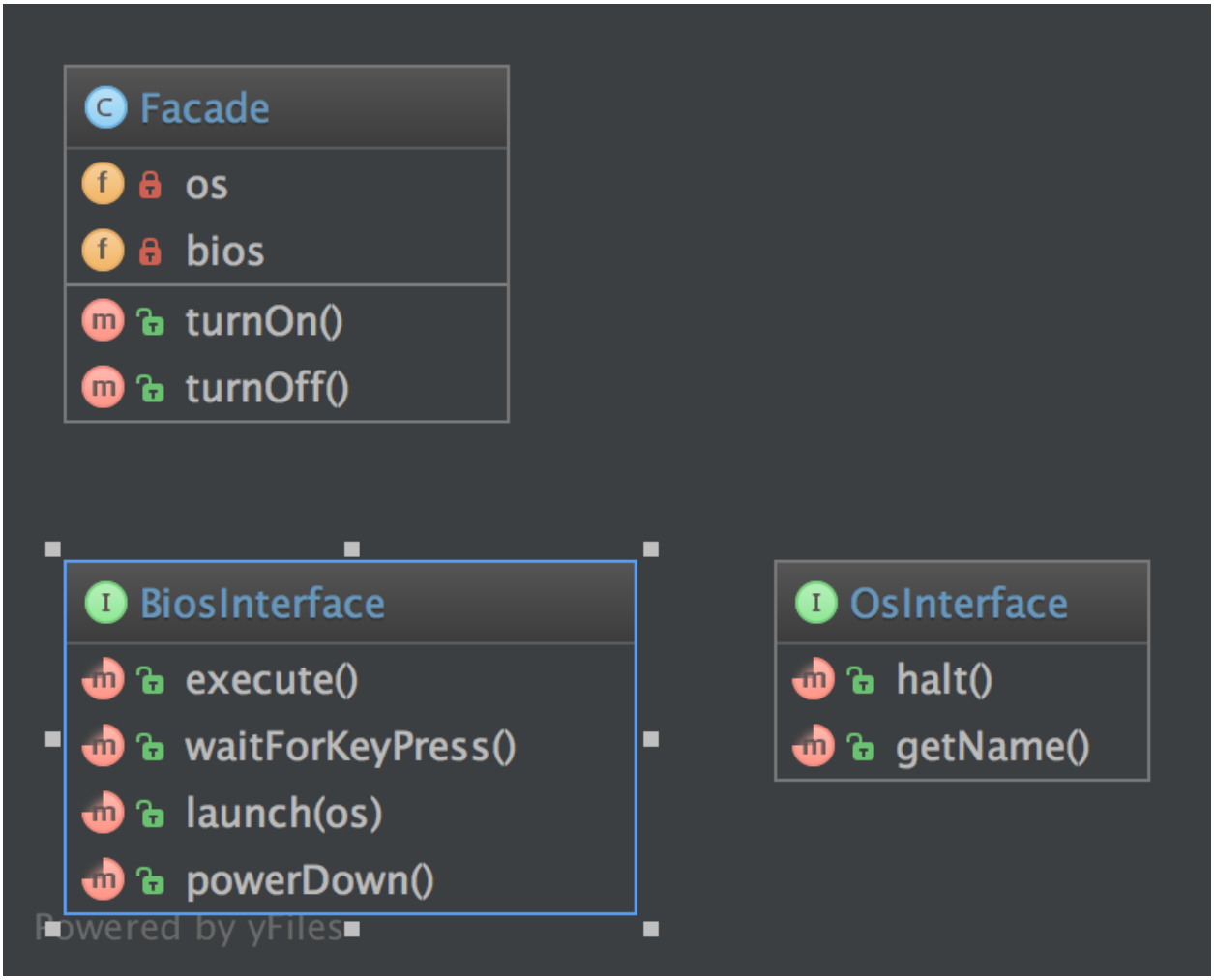
Фасад предназначен для разделения клиента и подсистемы путем внедрения многих (но иногда только одного) интерфейсов, и, конечно, уменьшения общей сложности.

- Фасад не запрещает прямой доступ к подсистеме. Просто он делает его проще и понятнее.
- Вы можете (и вам стоило бы) иметь несколько фасадов для одной подсистемы.

Вот почему хороший фасад не содержит созданий экземпляров классов (`new`) внутри. Если внутри фасада создаются объекты для реализации каждого метода, это не Фасад, это Строитель или [Абстрактная|Статическая|Простая] Фабрика [или Фабричный Метод].

Лучший фасад не содержит `new` или конструктора с type-hinted параметрами. Если вам необходимо создавать новые экземпляры классов, в таком случае лучше использовать Фабрику в качестве аргумента.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Facade.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

class Facade
{
    /**
     * @var OsInterface
     */
    private $os;

    /**
     * @var BiosInterface
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    private $bios;

    /**
     * @param BiosInterface $bios
     * @param OsInterface   $os
     */
    public function __construct(BiosInterface $bios, OsInterface $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();
    }
}
```

OsInterface.php

```
<?php

namespace DesignPatterns\Structural\Facade;

interface OsInterface
{
    public function halt();

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

BiosInterface.php

```
<?php

namespace DesignPatterns\Structural\Facade;

interface BiosInterface
{
    public function execute();

    public function waitForKeyPress();

    public function launch(OsInterface $os);

    public function powerDown();
}
```

Тест

Tests/FacadeTest.php

```php
<?php

namespace DesignPatterns\Structural\Facade\Tests;

use DesignPatterns\Structural\Facade\Facade;
use DesignPatterns\Structural\Facade\OsInterface;
use PHPUnit\Framework\TestCase;

class FacadeTest extends TestCase
{
    public function testComputerOn()
    {
        /** @var OsInterface|\PHPUnit_Framework_MockObject_MockObject $os */
        $os = $this->createMock('DesignPatterns\Structural\Facade\OsInterface');

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

        $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\BiosInterface')
            ->setMethods(['launch', 'execute', 'waitForKeyPress'])
            ->disableAutoload()
            ->getMock();

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

        $facade = new Facade($bios, $os);

        // the facade interface is simple
        $facade->turnOn();

        // but you can also access the underlying components
        $this->assertSame('Linux', $os->getName());
    }
}
```

1.2.8 Текучий Интерфейс (Fluent Interface)

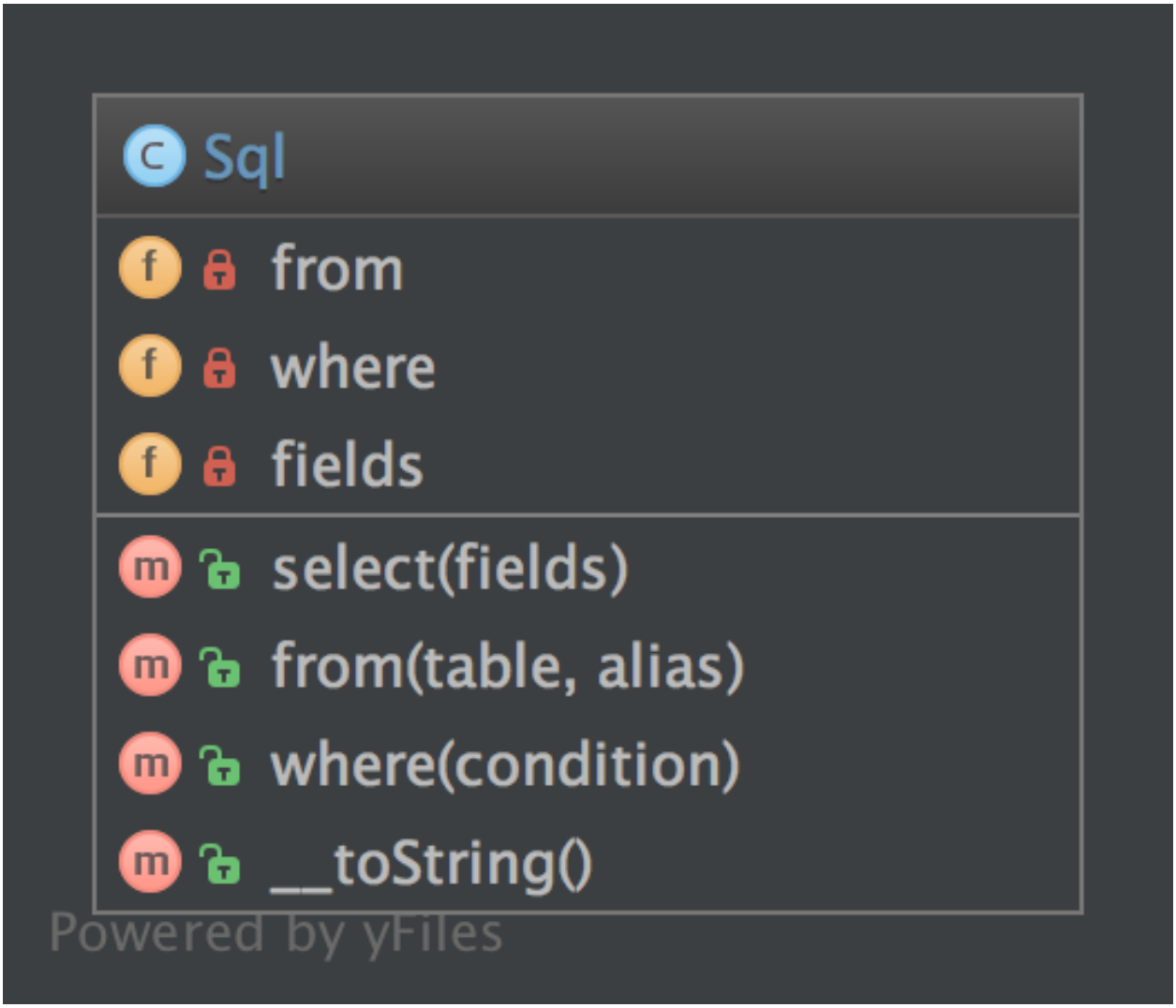
Назначение

Писать код, который легко читается, как предложения в естественном языке (вроде русского или английского).

Примеры

- Doctrine2's QueryBuilder работает примерно также, как пример ниже.
- PHPUnit использует текучий интерфейс, чтобы создавать макеты объектов.
- Yii Framework: CDbCommand и CActiveRecord тоже используют этот паттерн.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Sql.php

```php
<?php

namespace DesignPatterns\Structural\FluentInterface;

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

```

(continues on next page)

(продолжение с предыдущей страницы)

```
    /**
     * @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)
        );
    }
}
```

Тест

Tests/FluentInterfaceTest.php

```
<?php

namespace DesignPatterns\Structural\FluentInterface\Tests;

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

class FluentInterfaceTest extends TestCase
{
```

(continues on next page)

(продолжение с предыдущей страницы)

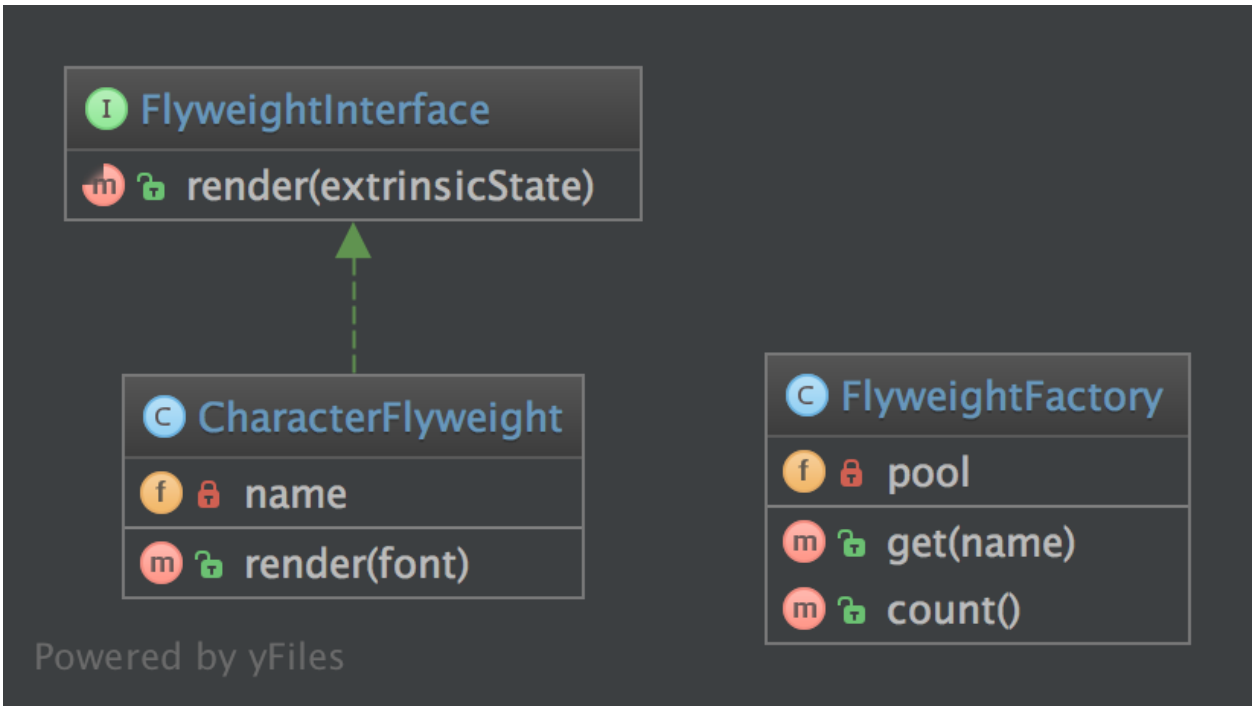
```
10      public function testBuildSQL()
11      {
12          $query = (new Sql())
13                  ->select(['foo', 'bar'])
14                  ->from('foobar', 'f')
15                  ->where('f.bar = ?');
16
17          $this->assertSame('SELECT foo, bar FROM foobar AS f WHERE f.bar = ?', (string) $query);
18      }
19  }
```

1.2.9 Приспособленец (Flyweight)

Назначение

Для уменьшения использования памяти Приспособленец разделяет как можно больше памяти между аналогичными объектами. Это необходимо, когда используется большое количество объектов, состояние которых не сильно отличается. Обычной практикой является хранение состояния во внешних структурах и передавать их в объект-приспособленец, когда необходимо.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

FlyweightInterface.php

```php
<?php

namespace DesignPatterns\Structural\Flyweight;

interface FlyweightInterface
{
    public function render(string $extrinsicState): string;
}
```

CharacterFlyweight.php

```php
<?php

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 CharacterFlyweight implements FlyweightInterface
{
    /**
     * 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)
    {
        $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);
    }
}
```

FlyweightFactory.php

```php
<?php

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 FlyweightFactory implements \Countable
{
    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     * @var CharacterFlyweight[]
     */
    private $pool = [];

    public function get(string $name): CharacterFlyweight
    {
        if (!isset($this->pool[$name])) {
            $this->pool[$name] = new CharacterFlyweight($name);
        }

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

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

Тест

Tests/FlyweightTest.php

```
<?php

namespace DesignPatterns\Structural\Flyweight\Tests;

use DesignPatterns\Structural\Flyweight\FlyweightFactory;
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 FlyweightFactory();

        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);
            }
        }

        // 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), $factory);
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
32 }
```

1.2.10 Прокси (Proxy)

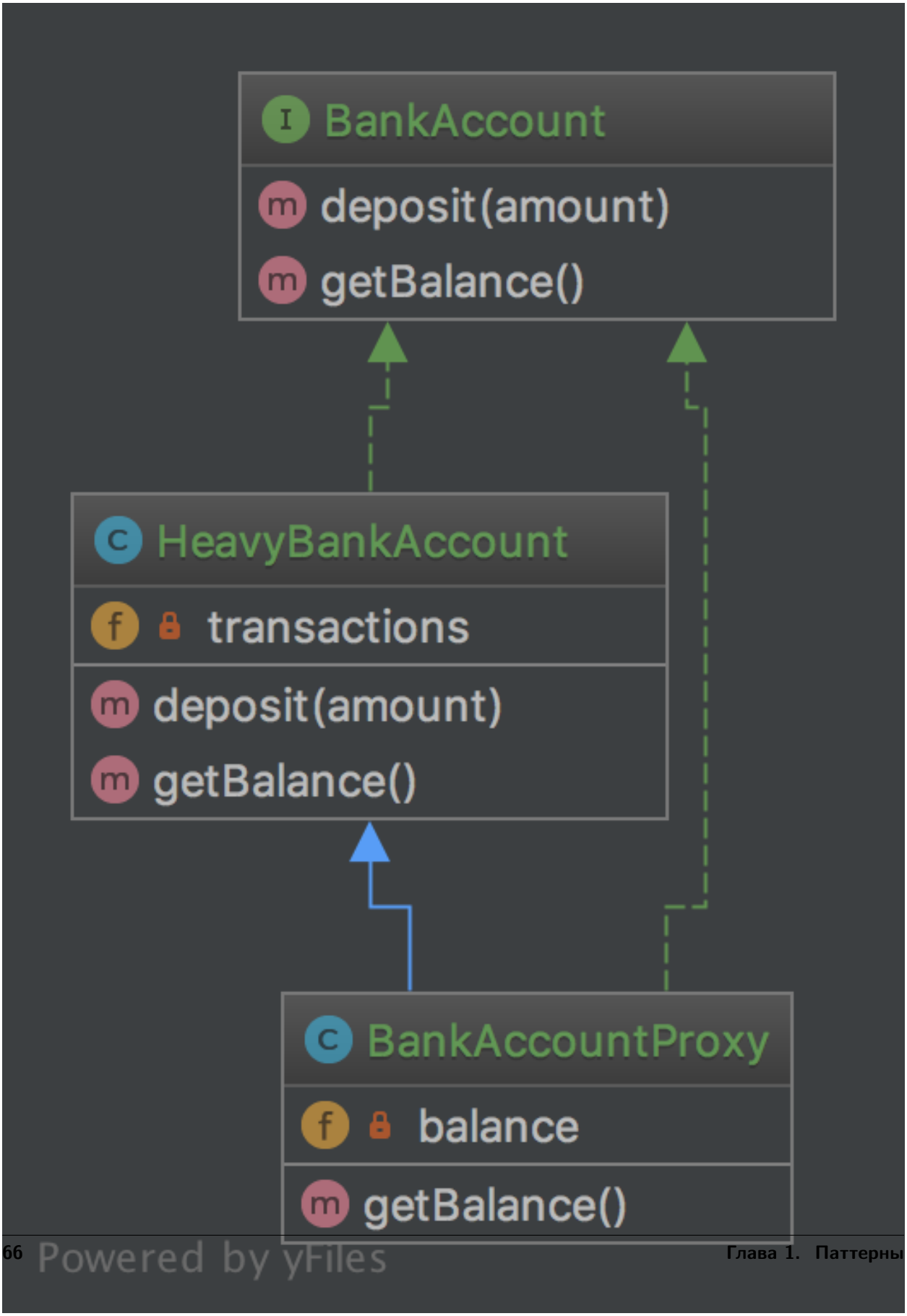
Назначение

Создать интерфейс взаимодействия с любым классом, который трудно или невозможно использовать в оригинальном виде.

Примеры

- Doctrine2 использует прокси для реализации магии фреймворка (например, для ленивой инициализации), в то время как пользователь работает со своими собственными классами сущностей и никогда не будет использовать прокси.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

BankAccount.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

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

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

HeavyBankAccount.php

```php
<?php

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

namespace DesignPatterns\Structural\Proxy;

class BankAccountProxy extends HeavyBankAccount implements BankAccount
{
    /**
     * @var int
     */
    private $balance;

```

(continues on next page)

(продолжение с предыдущей страницы)

```
12      public function getBalance(): int
13      {
14          // because calculating balance is so expensive,
15          // the usage of BankAccount::getBalance() is delayed until it really is needed
16          // and will not be calculated again for this instance
17
18          if ($this->balance === null) {
19              $this->balance = parent::getBalance();
20          }
21
22          return $this->balance;
23      }
24  }
```

Тест

1.2.11 Реестр (Registry)

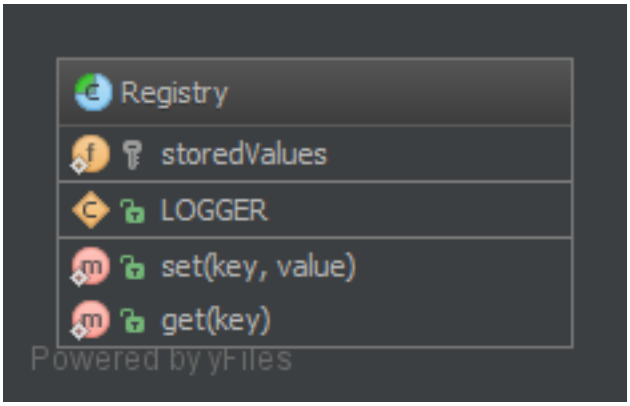
Назначение

Для реализации централизованного хранения объектов, часто используемых во всем приложении, как правило, реализуется с помощью абстрактного класса только с статическими методами (или с помощью шаблона Singleton). Помните что это вводит глобальное состояние, которого следует избегать. Используйте Dependency Injection вместо Registry.

Примеры

- Zend Framework 1: `Zend_Registry` содержит объект журналирования приложения (логгер), фронт-контроллер и т.д.
- Yii Framework: `CWebApplication` содержит все компоненты приложения, такие как `CWebUser`, `CUrlManager`, и т.д.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Registry.php

```php
<?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];
    }
}
```

Тест

Tests/RegistryTest.php

```php
<?php

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);
    }

    /**
     * @expectedException \InvalidArgumentException
     */
    public function testThrowsExceptionWhenTryingToSetInvalidKey()
    {
        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
     * @expectedException \InvalidArgumentException
     */
    public function testThrowsExceptionWhenTryingToGetNotSetKey()
    {
        Registry::get(Registry::LOGGER);
    }
}
```

1.3 Поведенческие шаблоны проектирования (Behavioral)

Поведенческие шаблоны проектирования определяют общие закономерности связей между объектами, реализующими данные паттерны. Следование этим шаблонам уменьшает связность системы и облегчает коммуникацию между объектами, что улучшает гибкость программного продукта.

1.3.1 Цепочка Обязанностей (Chain Of Responsibilities)

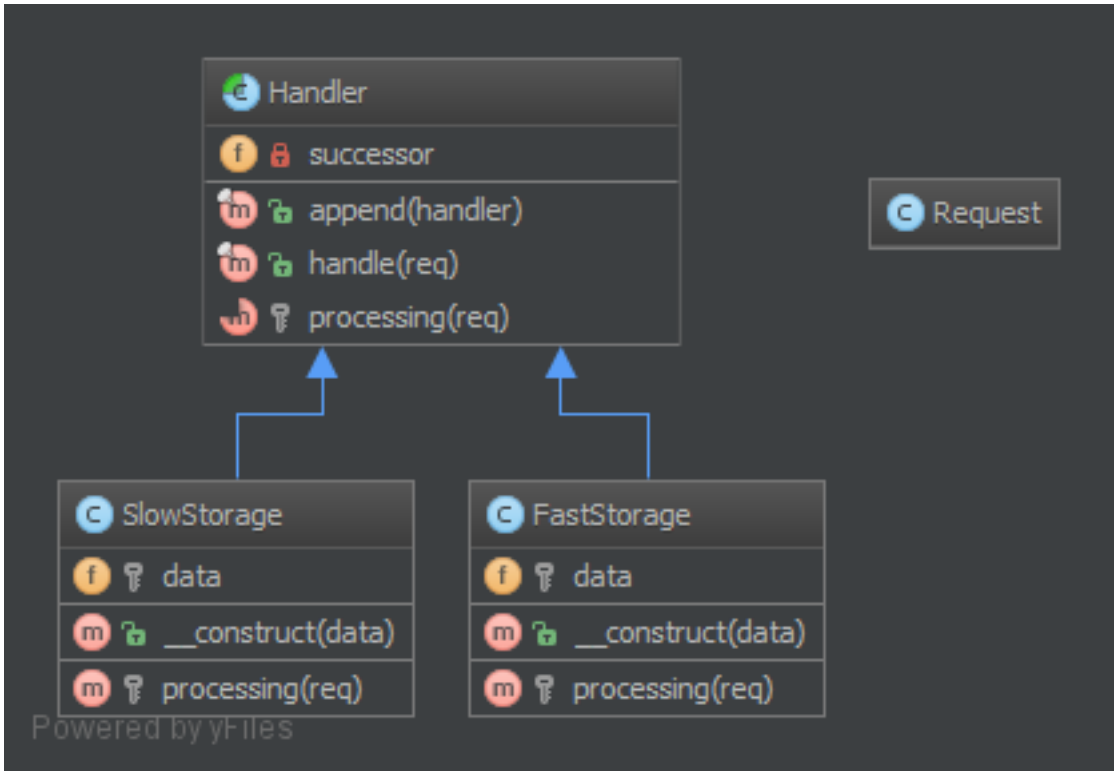
Purpose

Построить цепочку объектов для обработки вызова в последовательном порядке. Если один объект не может справиться с вызовом, он делегирует вызов для следующего в цепи и так далее.

Examples

- фреймворк для записи журналов, где каждый элемент цепи самостоятельно принимает решение, что делать с сообщением для логгирования.
- фильтр спама
- кеширование: первый объект является экземпляром, к примеру, интерфейса Memcached. Если запись в кеше отсутствует, вызов делегируется интерфейсу базы данных.
- Yii Framework: CFilterChain — это цепочка фильтров действий контроллера. Точка вызова передаётся от фильтра к фильтру по цепочке и только если все фильтры скажут “да”, действие в итоге может быть вызвано.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Handler.php

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

        return $processed;
    }

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

Responsible/FastStorage.php

```php
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

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

class HttpInMemoryCacheHandler extends Handler
{
    /**
     * @var array
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    private $data;

    /**
     * @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

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

```

(continues on next page)

(продолжение с предыдущей страницы)

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

Тест

Tests/ChainTest.php

```
<?php

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()
    {
        $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');
```

(continues on next page)

(продолжение с предыдущей страницы)

```
48          $request->method('getUri')->willReturn($uri);
49
50          $this->assertSame('Hello World!', $this->chain->handle($request));
51      }
52  }
```

1.3.2 Команда (Command)

Назначение

Инкапсулировать действие и его параметры

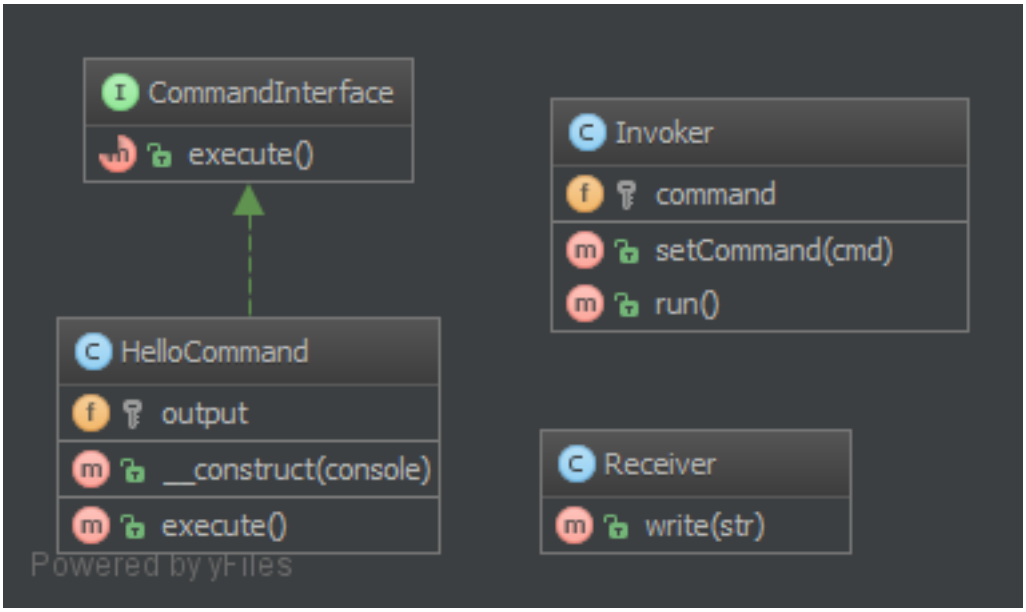
Допустим, у нас есть объекты Invoker (Командир) и Receiver (Исполнитель). Этот паттерн использует реализацию интерфейса «Команда», чтобы вызвать некий метод Исполнителя используя для этого известный Командиру метод «execute()». Командир просто знает, что нужно вызвать метод “execute()”, для обработки команды клиента, не разбираясь в деталях реализации Исполнителя. Исполнитель отделен от Командира.

Вторым аспектом этого паттерна является метод undo(), который отменяет действие, выполняемое методом execute(). Команды также могут быть объединены в более общие команды с минимальным копированием-вставкой и полагаясь на композицию поверх наследования.

Примеры

- текстовый редактор: все события являются Командами, которые могут быть отменены, выстроены в определённую последовательность и сохранены.
- Symfony2: SF2 Commands, это команды, которые построены согласно данному паттерну и могут выполняться из командной строки.
- большие утилиты для командной строки (например, Vagrant) используют вложенные команды для разделения различных задач и упаковки их в «модули», каждый из которых может быть реализован с помощью паттерна «Команда».

Диаграмма UML



Код

Вы также можете найти этот код на GitHub

CommandInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

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

HelloCommand.php

```php
<?php

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 CommandInterface
{
    /**
     * @var Receiver
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    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

namespace DesignPatterns\Behavioral\Command;

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    }

    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

namespace DesignPatterns\Behavioral\Command;

/**
 * Invoker is using the command given to it.
 * Example : an Application in SF2.
 */
class Invoker
{
    /**
     * @var CommandInterface
     */
    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 CommandInterface $cmd
     */
    public function setCommand(CommandInterface $cmd)
    {
        $this->command = $cmd;
    }

    /**
     * executes the command; the invoker is the same whatever is the command
     */
    public function run()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    {
        $this->command->execute();
    }
}
```

Тест

Tests/CommandTest.php

```php
<?php

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 Итератор (Iterator)

Назначение

Добавить коллекции объектов функционал последовательного доступа к содержащимся в ней экземплярам объектов без реализации этого функционала в самой коллекции.

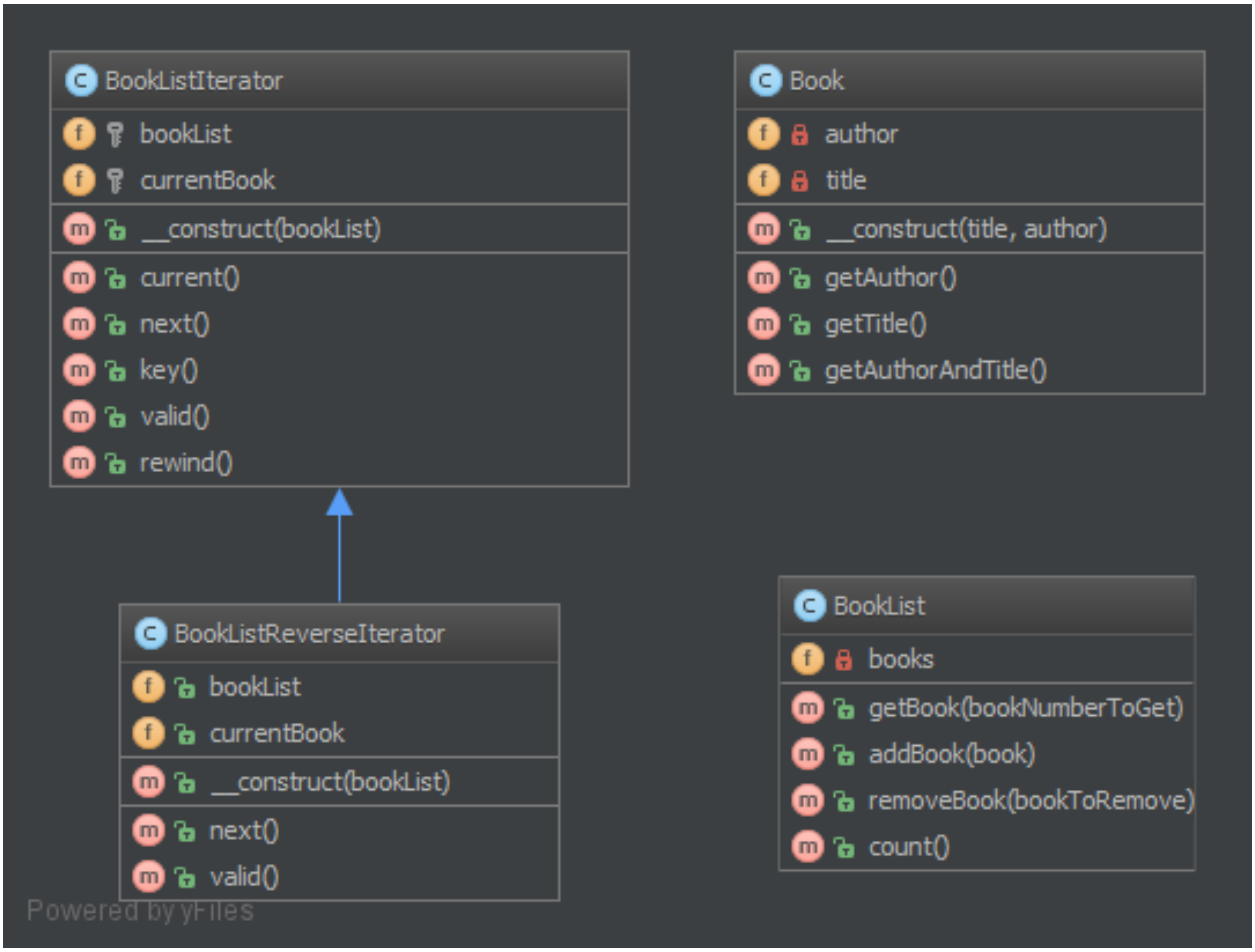
Примеры

- построчный перебор файла, который представлен в виде объекта, содержащего строки, тоже являющиеся объектами. Обработчик будет запущен поверх всех объектов.

Примечание

Стандартная библиотека PHP SPL определяет интерфейс Iterator, который хорошо подходит для данных целей. Также вам может понадобиться реализовать интерфейс Countable, чтобы разрешить вызывать `count($object)` в вашем листаемом объекте.

Диаграмма UML



Код

Также вы можете найти этот код на GitHub

Book.php

```
<?php

namespace DesignPatterns\Behavioral\Iterator;

class Book
{
    /**
     * @var string
     */
    private $author;

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

(continues on next page)

(продолжение с предыдущей страницы)

```
16
17      public function __construct(string $title, string $author)
18      {
19          $this->author = $author;
20          $this->title = $title;
21      }
22
23      public function getAuthor(): string
24      {
25          return $this->author;
26      }
27
28      public function getTitle(): string
29      {
30          return $this->title;
31      }
32
33      public function getAuthorAndTitle(): string
34      {
35          return $this->getTitle().' by '.$this->getAuthor();
36      }
37  }
```

BookList.php

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

(continues on next page)

(продолжение с предыдущей страницы)

```
32
33      public function count(): int
34      {
35          return count($this->books);
36      }
37
38      public function current(): Book
39      {
40          return $this->books[$this->currentIndex];
41      }
42
43      public function key(): int
44      {
45          return $this->currentIndex;
46      }
47
48      public function next()
49      {
50          $this->currentIndex++;
51      }
52
53      public function rewind()
54      {
55          $this->currentIndex = 0;
56      }
57
58      public function valid(): bool
59      {
60          return isset($this->books[$this->currentIndex]);
61      }
62  }
```

Тест

Tests/IteratorTest.php

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

(continues on next page)

(продолжение с предыдущей страницы)

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

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

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

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

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

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

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

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

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

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

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

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

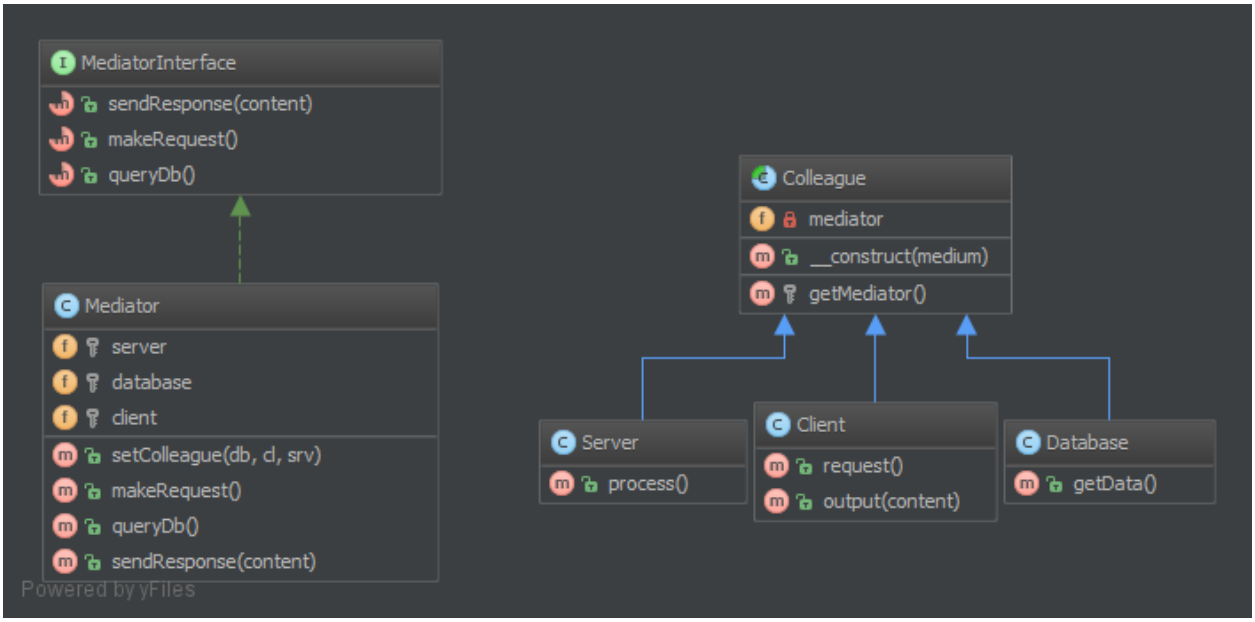
1.3.4 Посредник (Mediator)

Назначение

Этот паттерн позволяет снизить связность множества компонентов, работающих совместно. Объектам больше нет нужды вызывать друг друга напрямую. Это хорошая альтернатива Наблюдателю, если у вас есть “центр интеллекта” вроде контроллера (но не в смысле MVC)

Все компоненты (называемые «Коллеги») объединяются в интерфейс MediatorInterface и это хорошо, потому что в рамках ООП, «старый друг лучше новых двух».

Диаграмма UML



Код

Вы можете найти этот код на GitHub

MediatorInterface.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator;

/**
 * MediatorInterface is a contract for the Mediator
 * This interface is not mandatory but it is better for Liskov substitution principle concerns.
 */
interface MediatorInterface
{
    /**
     * sends the response.
     *
     * @param string $content
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    public function sendResponse($content);

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

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

Mediator.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator;

/**
 * Mediator is the concrete Mediator for this design pattern
 *
 * In this example, I have made a "Hello World" with the Mediator Pattern
 */
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
     * @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);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        $this->client->setMediator($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

namespace DesignPatterns\Behavioral\Mediator;

/**
 * Colleague is an abstract colleague who works together but he only knows
 * the Mediator, not other colleagues
 */
abstract class Colleague
{
    /**
     * this ensures no change in subclasses.
     *
     * @var MediatorInterface
     */
    protected $mediator;

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

Subsystem/Client.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

```

(continues on next page)

(продолжение с предыдущей страницы)

```php
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

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

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));
    }
}
```

Тест

Tests/MediatorTest.php

```php
<?php

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 Хранитель (Memento)

Назначение

Шаблон предоставляет возможность восстановить объект в его предыдущем состоянии (отменить действие посредством отката к предыдущему состоянию) или получить доступ к состоянию объекта, не раскрывая его реализацию (т.е. сам объект не обязан иметь функциональность для возврата текущего состояния).

Шаблон Хранитель реализуется тремя объектами: «Создателем» (originator), «Опекуном» (caretaker) и «Хранитель» (memento).

Хранитель - это объект, который *хранит конкретный снимок состояния* некоторого объекта или ресурса: строки, числа, массива, экземпляра класса и так далее. Уникальность в данном случае подразумевает не запрет на существование одинаковых состояний в разных снимках, а то, что состояние можно извлечь в виде независимой копии. Любой объект, сохраняемый в Хранителе, должен быть *полной копией исходного объекта, а не ссылкой* на исходный объект. Сам объект Хранитель является «непрозрачным объектом» (тот, который никто не может и не должен изменять).

Создатель — это объект, который *содержит в себе актуальное состояние внешнего объекта строго заданного типа* и умеет создавать уникальную копию этого состояния, возвращая её, обёрнутую в объект Хранителя. Создатель не знает истории изменений. Создателю можно принудительно установить конкретное состояние извне, которое будет считаться актуальным. Создатель должен позаботиться о том, чтобы это состояние соответствовало типу объекта, с которым ему разрешено работать. Создатель может (но не обязан) иметь любые методы, но они *не могут менять сохранённое состояние объекта.*

Опекун *управляет историей снимков состояний.* Он может вносить изменения в объект, принимать решение о сохранении состояния внешнего объекта в Создателе, запрашивать от Создателя снимок текущего состояния, или привести состояние Создателя в соответствие с состоянием какого-то снимка из истории.

Примеры

- Зерно генератора псевдослучайных чисел.
- Состояние конечного автомата
- Контроль промежуточных состояний модели в ORM перед сохранением

Диаграмма UML

Код

Вы можете найти этот код на GitHub

Memento.php

```php
<?php

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

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)

(продолжение с предыдущей страницы)

```
    /**
     * @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

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)

(продолжение с предыдущей страницы)

```
32          $this->currentState = new State(State::STATE_CLOSED);
33      }
34
35      public function saveToMemento(): Memento
36      {
37          return new Memento(clone $this->currentState);
38      }
39
40      public function restoreFromMemento(Memento $memento)
41      {
42          $this->currentState = $memento->getState();
43      }
44
45      public function getState(): State
46      {
47          return $this->currentState;
48      }
49  }
```

Тест

Tests/MementoTest.php

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

(continues on next page)

(продолжение с предыдущей страницы)

```
32 }
```

1.3.6 Объект Null (Null Object)

Назначение

NullObject не шаблон из книги Банды Четырёх, но схема, которая появляется достаточно часто, чтобы считаться паттерном. Она имеет следующие преимущества:

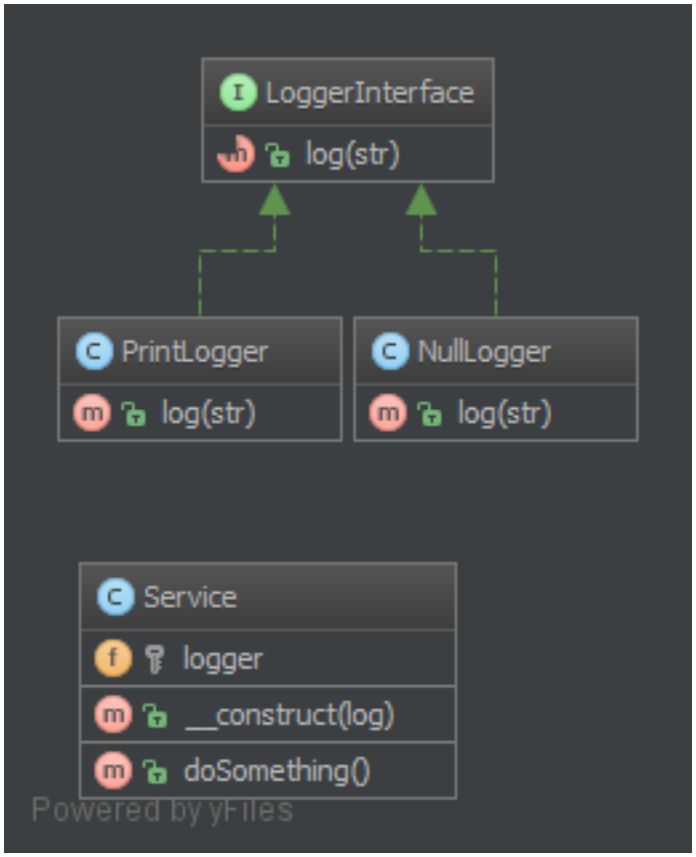
- Клиентский код упрощается
- Уменьшает шанс исключений из-за нулевых указателей (и ошибок PHP различного уровня)
- Меньше дополнительных условий — значит меньше тесткейсов

Методы, которые возвращают объект или Null, вместо этого должны вернуть объект `NullObject`. Это упрощённый формальный код, устраняющий необходимость проверки `if (!is_null($obj)) { $obj->callSomething(); }`, заменяя её на обычный вызов `$obj->callSomething();`.

Примеры

- Symfony2: null logger of profiler
- Symfony2: null output in Symfony/Console
- null handler in a Chain of Responsibilities pattern
- null command in a Command pattern

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Service.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

class Service
{
    /**
     * @var LoggerInterface
     */
    private $logger;

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

```

(continues on next page)

(продолжение с предыдущей страницы)

```
    /**
     * 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__);
    }
}
```

LoggerInterface.php

```
<?php

namespace DesignPatterns\Behavioral\NullObject;

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

PrintLogger.php

```
<?php

namespace DesignPatterns\Behavioral\NullObject;

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

NullLogger.php

```
<?php

namespace DesignPatterns\Behavioral\NullObject;

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

Тест

Tests/LoggerTest.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject\Tests;

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

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

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

1.3.7 Наблюдатель (Observer)

Назначение

Для реализации публикации/подписки на поведение объекта, всякий раз, когда объект «Subject» меняет свое состояние, прикрепленные объекты «Observers» будут уведомлены. Паттерн используется, чтобы сократить количество связанных напрямую объектов и вместо этого использует слабую связь (loose coupling).

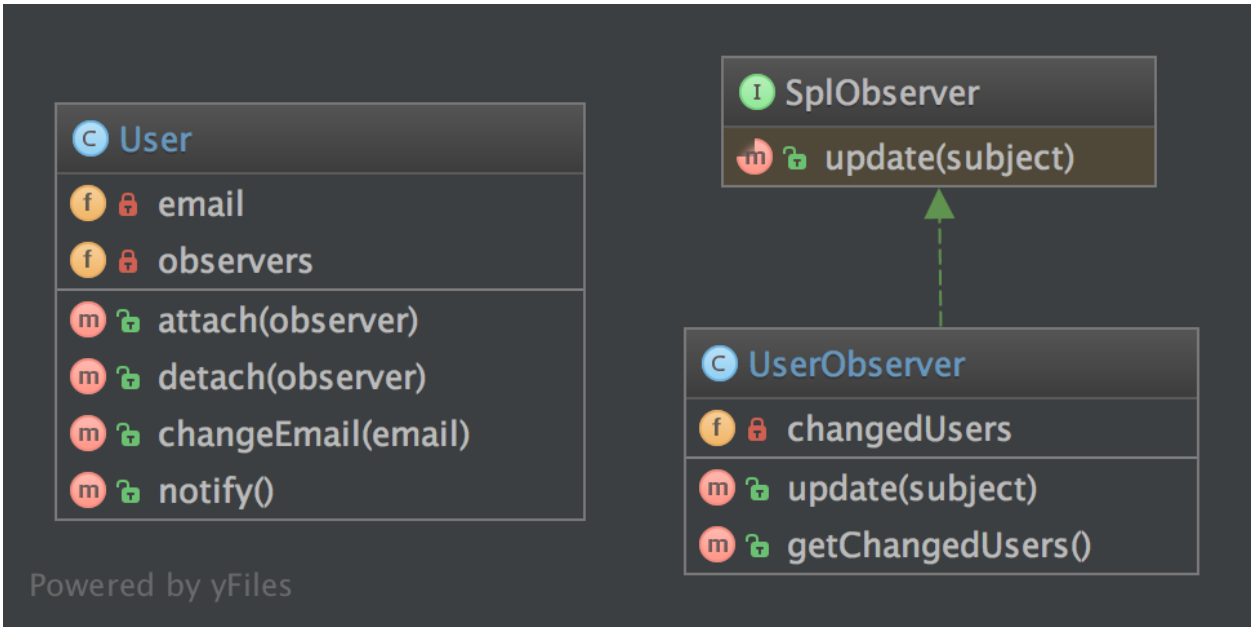
Примеры

- Система очереди сообщений наблюдает за очередями, чтобы отображать прогресс в GUI

Примечание

PHP предоставляет два стандартных интерфейса, которые могут помочь реализовать этот шаблон: SplObserver и SplSubject.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

User.php

```php
<?php

namespace DesignPatterns\Behavioral\Observer;

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

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

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

```

(continues on next page)

(продолжение с предыдущей страницы)

```
    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

namespace DesignPatterns\Behavioral\Observer;

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

    /**
     * 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;
    }
}
```

Тест

Tests/ObserverTest.php

```php
<?php

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 Спецификация (Specification)

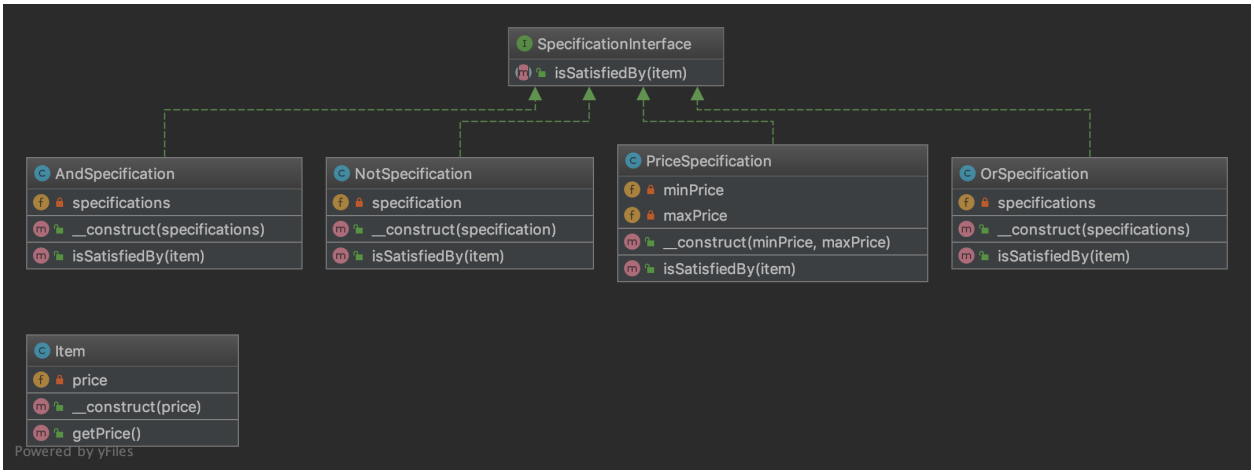
Назначение

Строит ясное описание бизнес-правил, на соответствие которым могут быть проверены объекты. Композитный класс спецификация имеет один метод, называемый `isSatisfiedBy`, который возвращает истину или ложь в зависимости от того, удовлетворяет ли данный объект спецификации.

Примеры

- RulerZ

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Item.php

```php
<?php

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;
    }
}
```

SpecificationInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

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

OrSpecification.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

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

    /**
     * @param SpecificationInterface[] ...$specifications
     */
    public function __construct(SpecificationInterface ...$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

namespace DesignPatterns\Behavioral\Specification;

class PriceSpecification implements SpecificationInterface
{
    /**
     * @var float|null
     */
    private $maxPrice;

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

    /**
     * @param float $minPrice
     * @param float $maxPrice
     */
    public function __construct($minPrice, $maxPrice)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    {
        $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

namespace DesignPatterns\Behavioral\Specification;

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

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

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

        return true;
    }
}
```

NotSpecification.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification;

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

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

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

Тест

Tests/SpecificationTest.php

```php
<?php

namespace DesignPatterns\Behavioral\Specification\Tests;

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

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

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

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

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

```

(continues on next page)

(продолжение с предыдущей страницы)

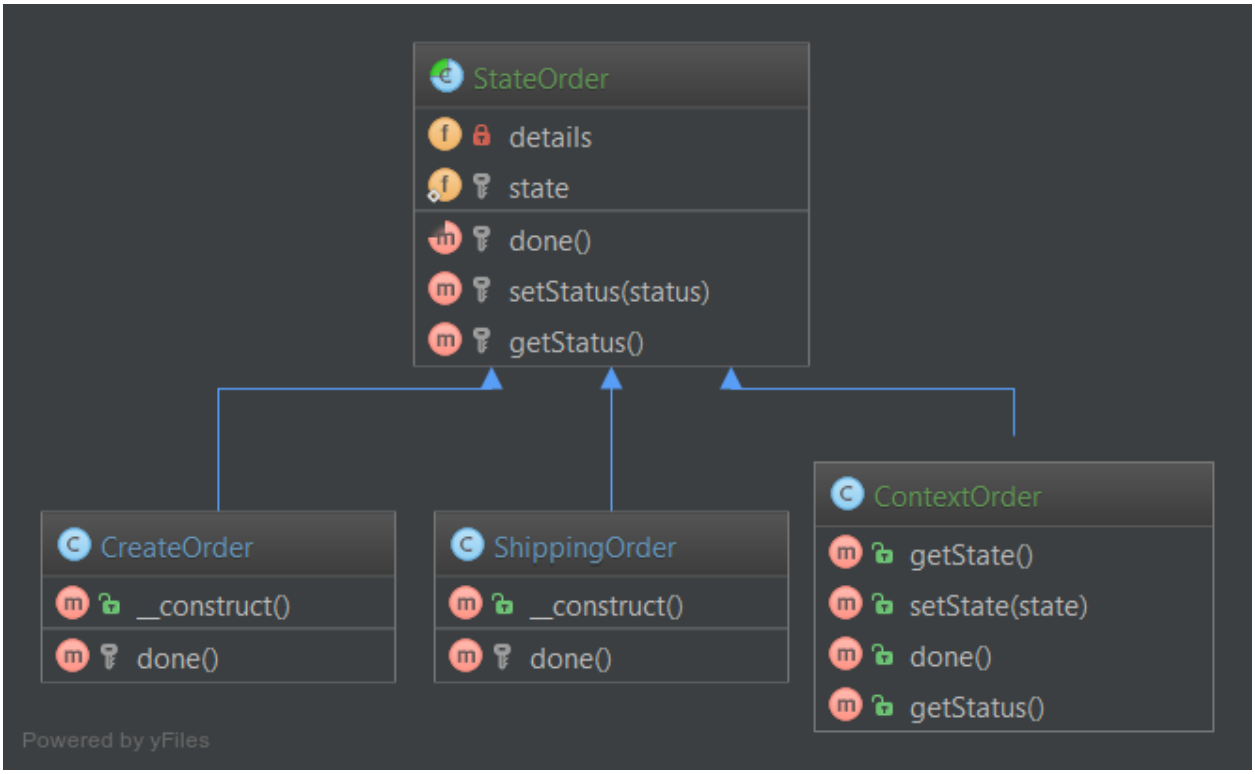
```
        $andSpec = new AndSpecification($spec1, $spec2);

        $this->assertFalse($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 Состояние (State)

Назначение

Инкапсулирует изменение поведения одних и тех же методов в зависимости от состояния объекта. Этот паттерн поможет изящным способом изменить поведение объекта во время выполнения не прибегая к большим монолитным условным операторам.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

OrderContext.php

```php
<?php

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)

(продолжение с предыдущей страницы)

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

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

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

State.php

```
<?php

namespace DesignPatterns\Behavioral\State;

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

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

StateCreated.php

```
<?php

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

namespace DesignPatterns\Behavioral\State;

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

(continues on next page)

(продолжение с предыдущей страницы)

```
        $context->setState(new StateDone());
    }

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

StateDone.php

```
<?php

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';
    }
}
```

Тест

Tests/StateTest.php

```
<?php

namespace DesignPatterns\Behavioral\State\Tests;

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

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

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
24
25      public function testCanProceedToStateDone()
26      {
27          $contextOrder = OrderContext::create();
28          $contextOrder->proceedToNext();
29          $contextOrder->proceedToNext();
30
31          $this->assertSame('done', $contextOrder->toString());
32      }
33
34      public function testStateDoneIsTheLastPossibleState()
35      {
36          $contextOrder = OrderContext::create();
37          $contextOrder->proceedToNext();
38          $contextOrder->proceedToNext();
39          $contextOrder->proceedToNext();
40
41          $this->assertSame('done', $contextOrder->toString());
42      }
43  }
```

1.3.10 Стратегия (Strategy)

Терминология:

- Context
- Strategy
- Concrete Strategy

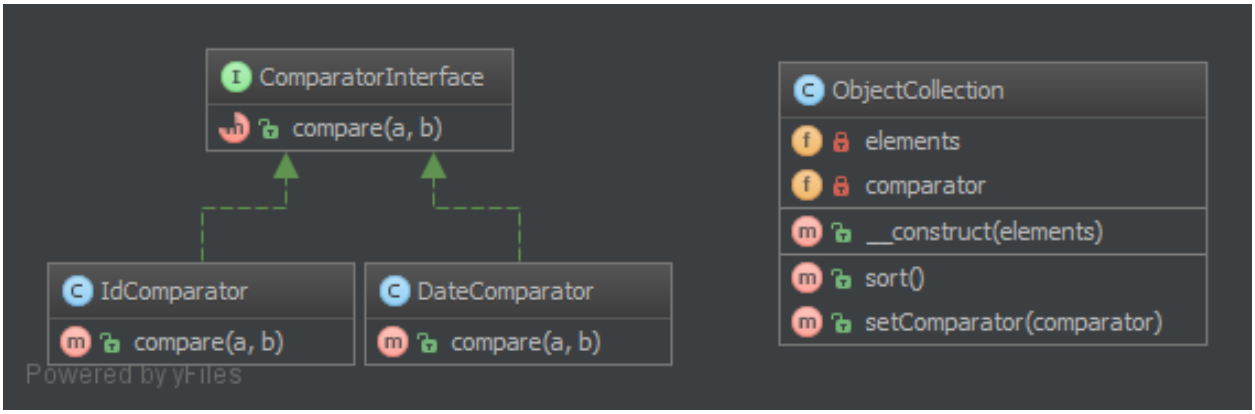
Назначение

Чтобы разделить стратегии и получить возможность быстрого переключения между ними. Также этот паттерн является хорошей альтернативой наследованию (вместо расширения абстрактного класса).

Примеры

- сортировка списка объектов, одна стратегия сортирует по дате, другая по id
- упростить юнит тестирование: например переключение между файловым хранилищем и хранилищем в оперативной памяти

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Context.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

class Context
{
    /**
     * @var ComparatorInterface
     */
    private $comparator;

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

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

        return $elements;
    }
}
```

ComparatorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

interface ComparatorInterface
{
    /**
     * @param mixed $a
```

(continues on next page)

(продолжение с предыдущей страницы)

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

DateComparator.php

```
<?php

namespace DesignPatterns\Behavioral\Strategy;

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

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

IdComparator.php

```
<?php

namespace DesignPatterns\Behavioral\Strategy;

class IdComparator implements ComparatorInterface
{
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return int
     */
    public function compare($a, $b): int
    {
        return $a['id'] <=> $b['id'];
    }
}
```

Тест

Tests/StrategyTest.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy\Tests;

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

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

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

    /**
     * @dataProvider provideIntegers
     *
     * @param array $collection
     * @param array $expected
     */
    public function testIdComparator($collection, $expected)
    {
        $obj = new Context(new IdComparator());
        $elements = $obj->executeStrategy($collection);

        $firstElement = array_shift($elements);
        $this->assertSame($expected, $firstElement);
    }

    /**
     * @dataProvider provideDates
     *
```

(continues on next page)

(продолжение с предыдущей страницы)

```
58          * @param array $collection
59          * @param array $expected
60          */
61         public function testDateComparator($collection, $expected)
62         {
63             $obj = new Context(new DateComparator());
64             $elements = $obj->executeStrategy($collection);
65
66             $firstElement = array_shift($elements);
67             $this->assertSame($expected, $firstElement);
68         }
69     }
```

1.3.11 Шаблонный Метод (Template Method)

Назначение

Шаблонный метод, это поведенческий паттерн проектирования.

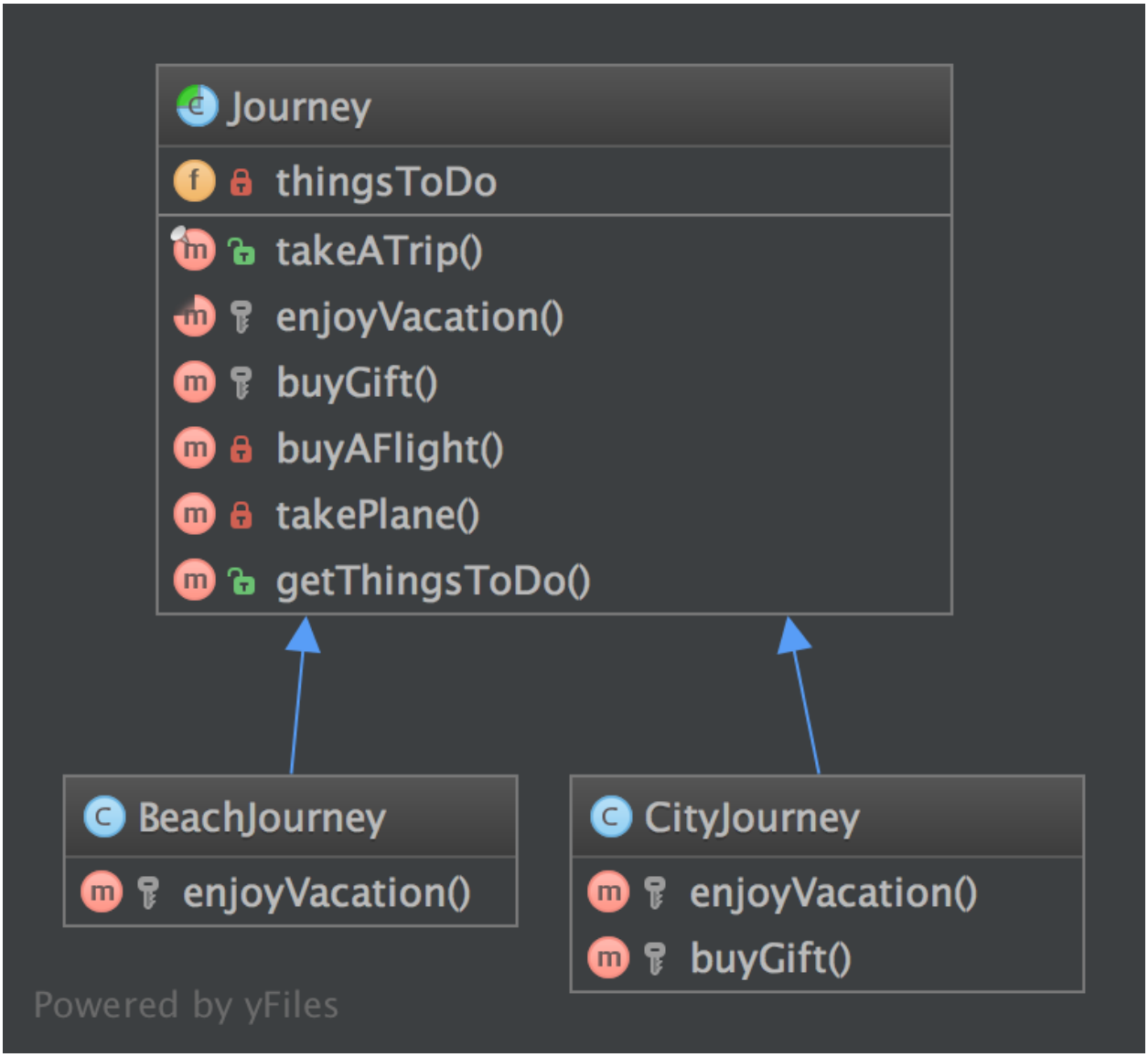
Возможно, вы сталкивались с этим уже много раз. Идея состоит в том, чтобы позволить наследникам абстрактного шаблона переопределить поведение алгоритмов родителя.

Как в «Голливудском принципе»: «Не звоните нам, мы сами вам позвоним». Этот класс не вызывается подклассами, но наоборот: подклассы вызываются родителем. Как? С помощью метода в родительской абстракции, конечно.

Другими словами, это каркас алгоритма, который хорошо подходит для библиотек (в фреймворках, например). Пользователь просто реализует уточняющие методы, а суперкласс делает всю основную работу.

Это простой способ изолировать логику в конкретные классы и уменьшить копипаст, поэтому вы повсеместно встретите его в том или ином виде.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Journey.php

```
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

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

(continues on next page)

(продолжение с предыдущей страницы)

```
9         */
10        private $thingsToDo = [];
11
12        /**
13         * This is the public service provided by this class and its subclasses.
14         * Notice it is final to "freeze" the global behavior of algorithm.
15         * If you want to override this contract, make an interface with only takeATrip()
16         * and subclass it.
17         */
18        final public function takeATrip()
19        {
20            $this->thingsToDo[] = $this->buyAFlight();
21            $this->thingsToDo[] = $this->takePlane();
22            $this->thingsToDo[] = $this->enjoyVacation();
23            $buyGift = $this->buyGift();
24
25            if ($buyGift !== null) {
26                $this->thingsToDo[] = $buyGift;
27            }
28
29            $this->thingsToDo[] = $this->takePlane();
30        }
31
32        /**
33         * This method must be implemented, this is the key-feature of this pattern.
34         */
35        abstract protected function enjoyVacation(): string;
36
37        /**
38         * This method is also part of the algorithm but it is optional.
39         * You can override it only if you need to
40         *
41         * @return null|string
42         */
43        protected function buyGift()
44        {
45            return null;
46        }
47
48        private function buyAFlight(): string
49        {
50            return 'Buy a flight ticket';
51        }
52
53        private function takePlane(): string
54        {
55            return 'Taking the plane';
56        }
57
58        /**
59         * @return string[]
60         */
61        public function getThingsToDo(): array
62        {
63            return $this->thingsToDo;
64        }
65    }
```

BeachJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

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

CityJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

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

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

Тест

Tests/JourneyTest.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

use DesignPatterns\Behavioral\TemplateMethod;
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()
        );
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    }

    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()
        );
    }
}
```

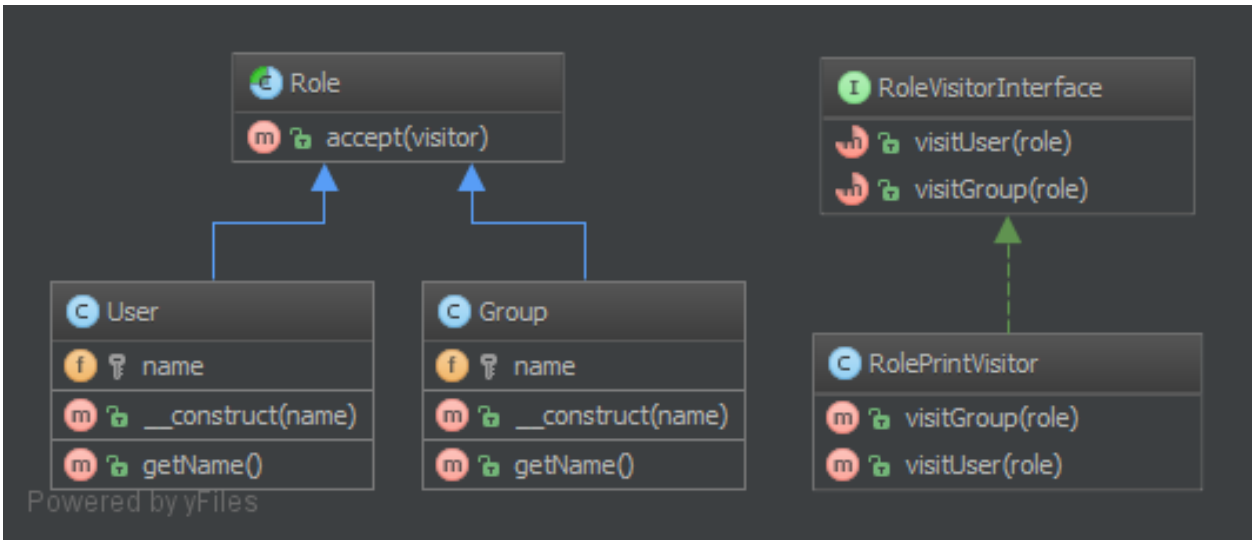
1.3.12 Посетитель (Visitor)

Назначение

Шаблон «Посетитель» выполняет операции над объектами других классов. Главной целью является сохранение разделения направленности задач отдельных классов. При этом классы обязаны определить специальный контракт, чтобы позволить использовать их Посетителям (метод «принять роль» `Role::accept` в примере).

Контракт, как правило, это абстрактный класс, но вы можете использовать чистый интерфейс. В этом случае, каждый посетитель должен сам выбирать, какой метод ссылается на посетителя.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

RoleVisitorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

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

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

RoleVisitor.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

class RoleVisitor implements RoleVisitorInterface
{
    /**
     * @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

namespace DesignPatterns\Behavioral\Visitor;
```

(continues on next page)

(продолжение с предыдущей страницы)

```

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

User.php

```
<?php

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(RoleVisitorInterface $visitor)
    {
        $visitor->visitUser($this);
    }
}
```

Group.php

```
<?php

namespace DesignPatterns\Behavioral\Visitor;

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    }

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

Тест

Tests/VisitorTest.php

```
<?php

namespace DesignPatterns\Tests\Visitor\Tests;

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

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

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

    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]);
    }
}
```

1.4 Дополнительно

1.4.1 Локатор Служб (Service Locator)

Этот шаблон считается анти-паттерном!

Некоторые считают Локатор Служб анти-паттерном. Он нарушает принцип инверсии зависимостей (Dependency Inversion principle) из набора принципов SOLID. Локатор Служб скрывает зависимости данного класса вместо их совместного использования, как в случае шаблона Внедрение Зависимости (Dependency Injection). В случае изменения данных зависимостей мы рискуем сломать функционал классов, которые их используют, вследствие чего затрудняется поддержка системы.

Назначение

Для реализации слабосвязанной архитектуры, чтобы получить хорошо тестируемый, сопровождаемый и расширяемый код. Паттерн Инъекция зависимостей (DI) и паттерн Локатор Служб — это реализация паттерна Инверсия управления (Inversion of Control, IoC).

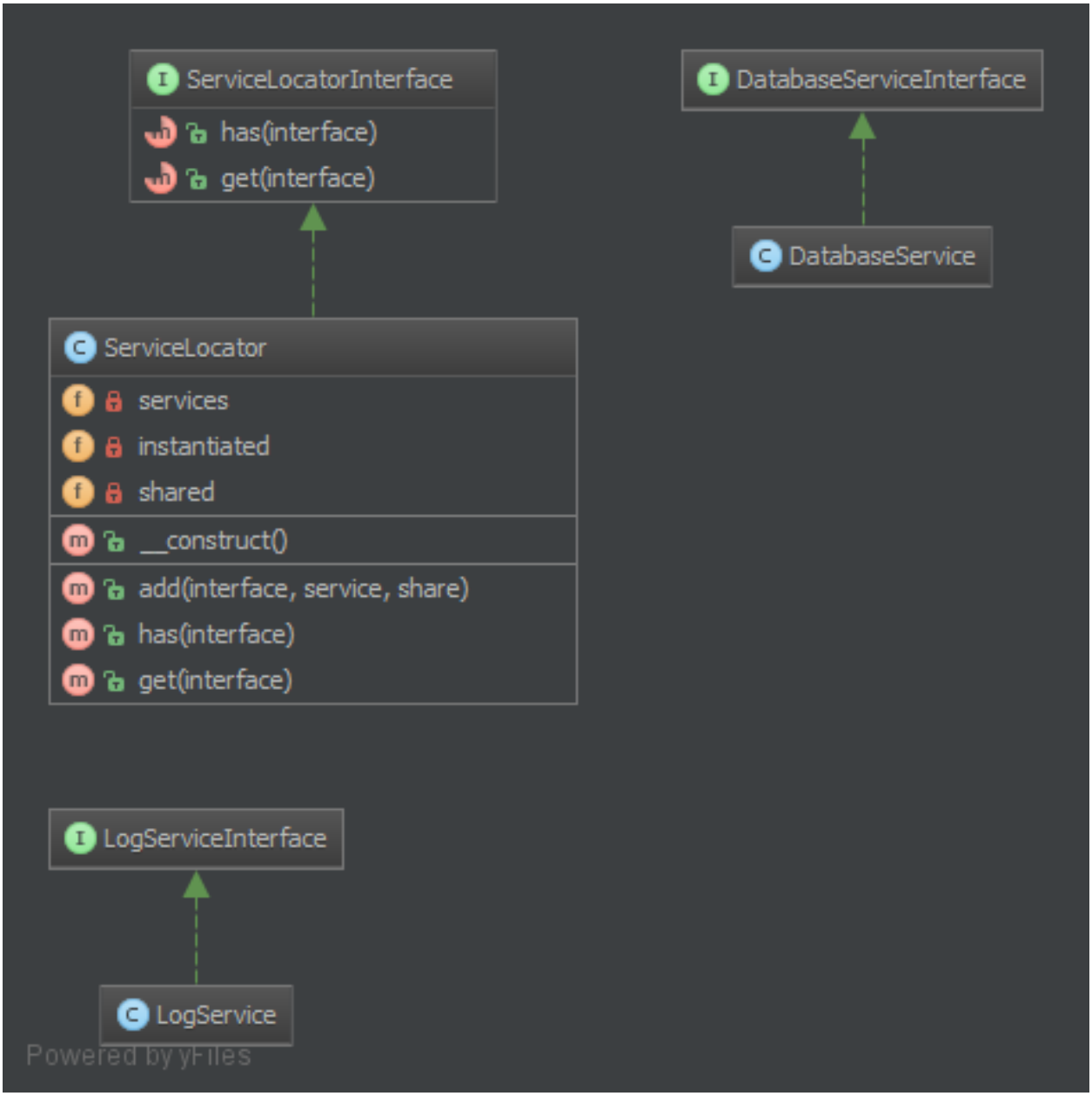
Использование

С `Локатором Служб` вы можете зарегистрировать сервис для определенного интерфейса. С помощью интерфейса вы можете получить зарегистрированный сервис и использовать его в классах приложения, не зная его реализацию. Вы можете настроить и внедрить объект Service Locator на начальном этапе сборки приложения.

Примеры

- Zend Framework 2 использует Service Locator для создания и совместного использования сервисов, задействованных в фреймворке (т.е. EventManager, ModuleManager, все пользовательские сервисы, предоставляемые модулями, и т.д ...)

Диаграмма UML



Код

Вы можете найти этот код на GitHub

ServiceLocator.php

```
1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  class ServiceLocator
```

(continues on next page)

(продолжение с предыдущей страницы)

```
{
    /**
     * @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
     */
    public function get(string $class)
    {
        if (isset($this->instantiated[$class]) && $this->shared[$class]) {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            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

namespace DesignPatterns\More\ServiceLocator;

class LogService
{
}
```

Тест

Tests/ServiceLocatorTest.php

```
<?php

namespace DesignPatterns\More\ServiceLocator\Tests;

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

class ServiceLocatorTest extends TestCase
{
    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
12          * @var ServiceLocator
13          */
14         private $serviceLocator;
15
16         public function setUp()
17         {
18             $this->serviceLocator = new ServiceLocator();
19         }
20
21         public function testHasServices()
22         {
23             $this->serviceLocator->addInstance(LogService::class, new LogService());
24
25             $this->assertTrue($this->serviceLocator->has(LogService::class));
26             $this->assertFalse($this->serviceLocator->has(self::class));
27         }
28
29         public function testGetWillInstantiateLogServiceIfNoInstanceHasBeenCreatedYet()
30         {
31             $this->serviceLocator->addClass(LogService::class, []);
32             $logger = $this->serviceLocator->get(LogService::class);
33
34             $this->assertInstanceOf(LogService::class, $logger);
35         }
36     }
```

1.4.2 Хранилище (Repository)

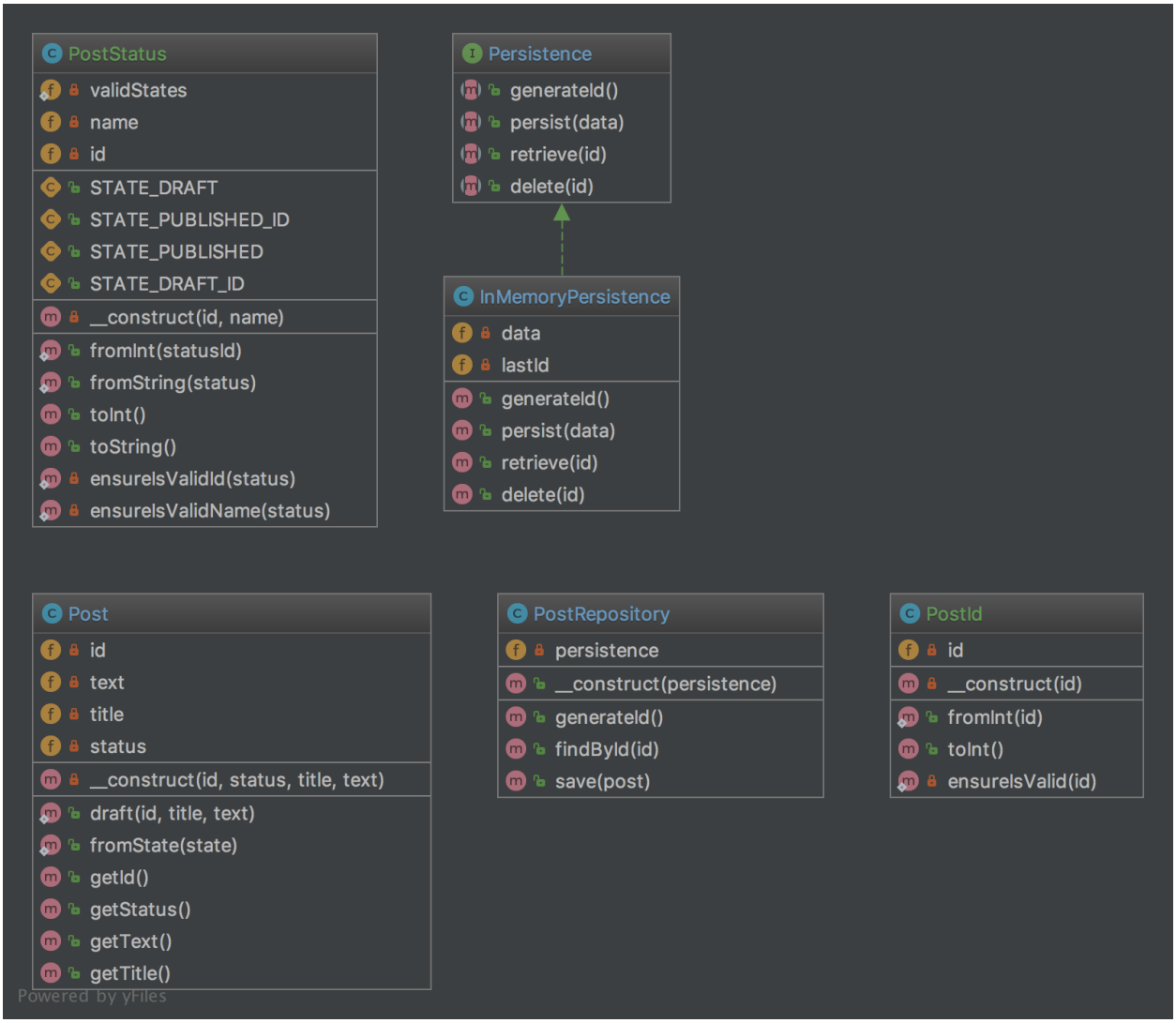
Назначение

Посредник между уровнями области определения (хранилище) и распределения данных. Использует интерфейс, похожий на коллекции, для доступа к объектам области определения. Репозиторий инкапсулирует набор объектов, сохраняемых в хранилище данных, и операции выполняемые над ними, обеспечивая более объектно-ориентированное представление реальных данных. Репозиторий также преследует цель достижения полного разделения и односторонней зависимости между уровнями области определения и распределения данных.

Примеры

- Doctrine 2 ORM: в ней есть Repository, который является связующим звеном между Entity и DBAL и содержит методы для получения объектов.
- Laravel Framework

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Post.php

```
<?php

namespace DesignPatterns\More\Repository\Domain;

class Post
{
    /**
     * @var PostId
     */
    private $id;

```

(continues on next page)

(продолжение с предыдущей страницы)

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

(continues on next page)

(продолжение с предыдущей страницы)

```
68          return $this->status;
69      }
70
71      public function getText(): string
72      {
73          return $this->text;
74      }
75
76      public function getTitle(): string
77      {
78          return $this->title;
79      }
80  }
```

PostId.php

```
1   <?php
2
3   namespace DesignPatterns\More\Repository\Domain;
4
5   /**
6    * This is a perfect example of a value object that is identifiable by it's value alone and
7    * is guaranteed to be valid each time an instance is created. Another important property of value␣
    ↪objects
8    * is immutability.
9    *
10   * Notice also the use of a named constructor (fromInt) which adds a little context when creating␣
    ↪an instance.
11   */
12  class PostId
13  {
14      /**
15       * @var int
16       */
17      private $id;
18
19      public static function fromInt(int $id)
20      {
21          self::ensureIsValid($id);
22
23          return new self($id);
24      }
25
26      private function __construct(int $id)
27      {
28          $this->id = $id;
29      }
30
31      public function toInt(): int
32      {
33          return $this->id;
34      }
35
36      private static function ensureIsValid(int $id)
37      {
38          if ($id <= 0) {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            throw new \InvalidArgumentException('Invalid PostId given');
        }
    }
}
```

PostStatus.php

```
<?php

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);
    }

    private function __construct(int $id, string $name)
    {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        $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

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
 * between the domain and data mapping layers
 */
class PostRepository
{
    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     * @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

namespace DesignPatterns\More\Repository;

interface Persistence
{
    public function generateId(): int;

    public function persist(array $data);

    public function retrieve(int $id): array;

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

InMemoryPersistence.php

```php
<?php

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]);
    }
}
```

Тест

Tests/PostRepositoryTest.php

```php
<?php

namespace DesignPatterns\More\Repository\Tests;

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

(continues on next page)

(продолжение с предыдущей страницы)

```php
use DesignPatterns\More\Repository\Domain\PostStatus;
use DesignPatterns\More\Repository\InMemoryPersistence;
use DesignPatterns\More\Repository\Domain\Post;
use DesignPatterns\More\Repository\PostRepository;
use PHPUnit\Framework\TestCase;

class PostRepositoryTest extends TestCase
{
    /**
     * @var PostRepository
     */
    private $repository;

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

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

    /**
     * @expectedException \OutOfBoundsException
     * @expectedExceptionMessage Post with id 42 does not exist
     */
    public function testThrowsExceptionWhenTryingToFindPostWhichDoesNotExist()
    {
        $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 Сущность-Атрибут-Значение

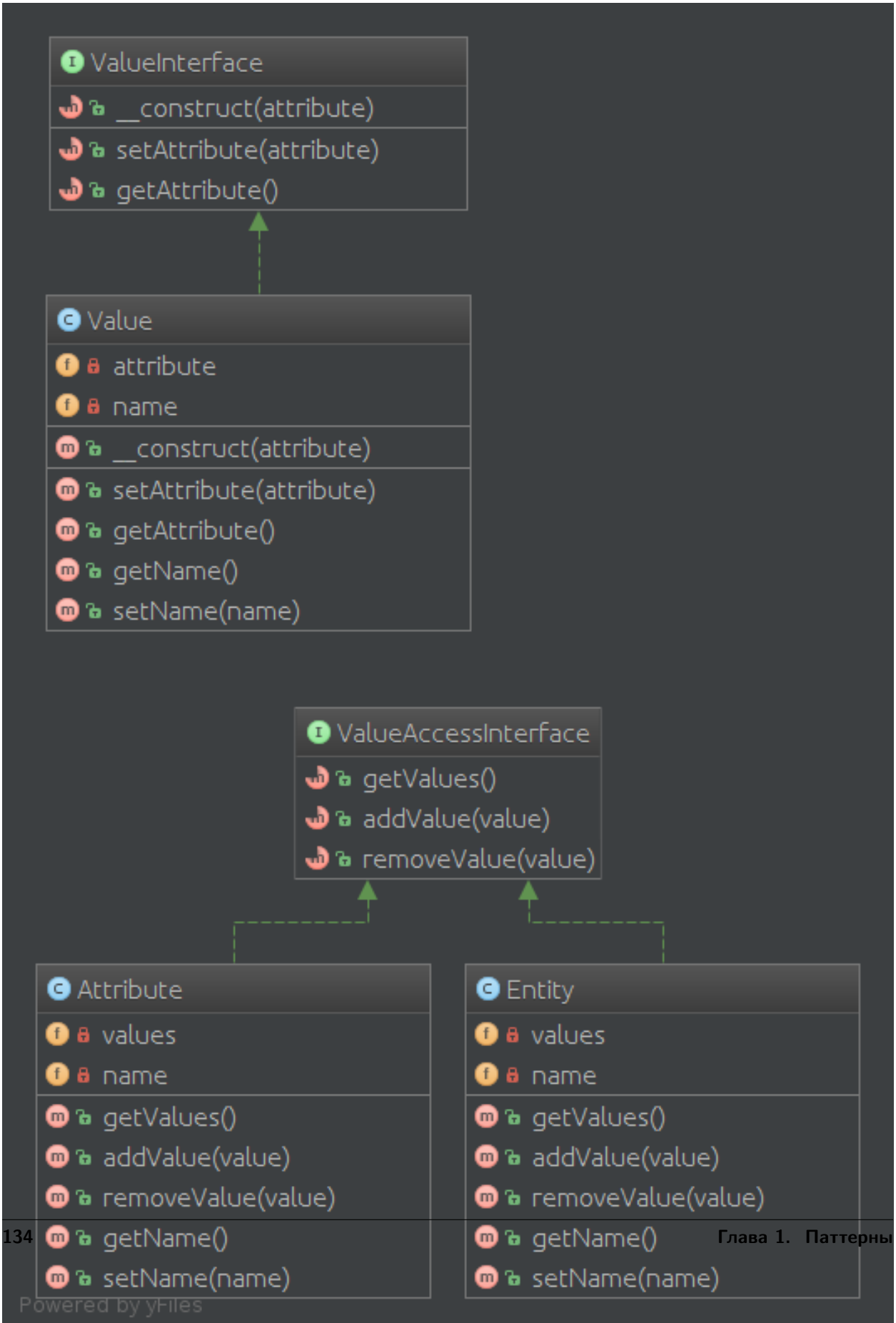
Шаблон Сущность-Атрибут-Значение используется для реализации модели EAV на PHP

Назначение

Модель Сущность-Атрибут-Значение (EAV) - это модель данных, предназначенная для описания сущностей, в которых количество атрибутов (свойств, параметров), характеризующих их, потенциально

огромно, но то количество, которое реально будет использоваться в конкретной сущности, относительно мало.

Диаграмма UML



Код

Вы можете найти этот код на GitHub

Entity.php

```php
<?php

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

namespace DesignPatterns\More\EAV;

class Attribute
{
    /**
     * @var \SplObjectStorage
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     */
    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

namespace DesignPatterns\More\EAV;

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

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

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

```

(continues on next page)

(продолжение с предыдущей страницы)

```
        $attribute->addValue($this);
    }

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

Тест

Tests/EAVTest.php

```
<?php

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);
    }
}
```

Глава 2

Участие в разработке

Если вы обнаружили ошибки или отсутствие перевода, вы можете прислать пулл реквест с вашими изменениями. Чтобы сохранять высокое качество кода, пожалуйста, проверяйте ваш код с помощью PHP CodeSniffer на соответствие стандарту PSR2, используя команду `./vendor/bin/phpcs -p --standard=PSR2 --ignore=vendor .`.