DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

September 19, 2015

Contents

This is a collection of known design patterns and some sample code how to implement them in PHP. Every pattern has a small list of examples (most of them from Zend Framework, Symfony2 or Doctrine2 as I'm most familiar with this software).

I think the problem with patterns is that often people do know them but don't know when to apply which.

CHAPTER 1

Patterns

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

1.1 Creational

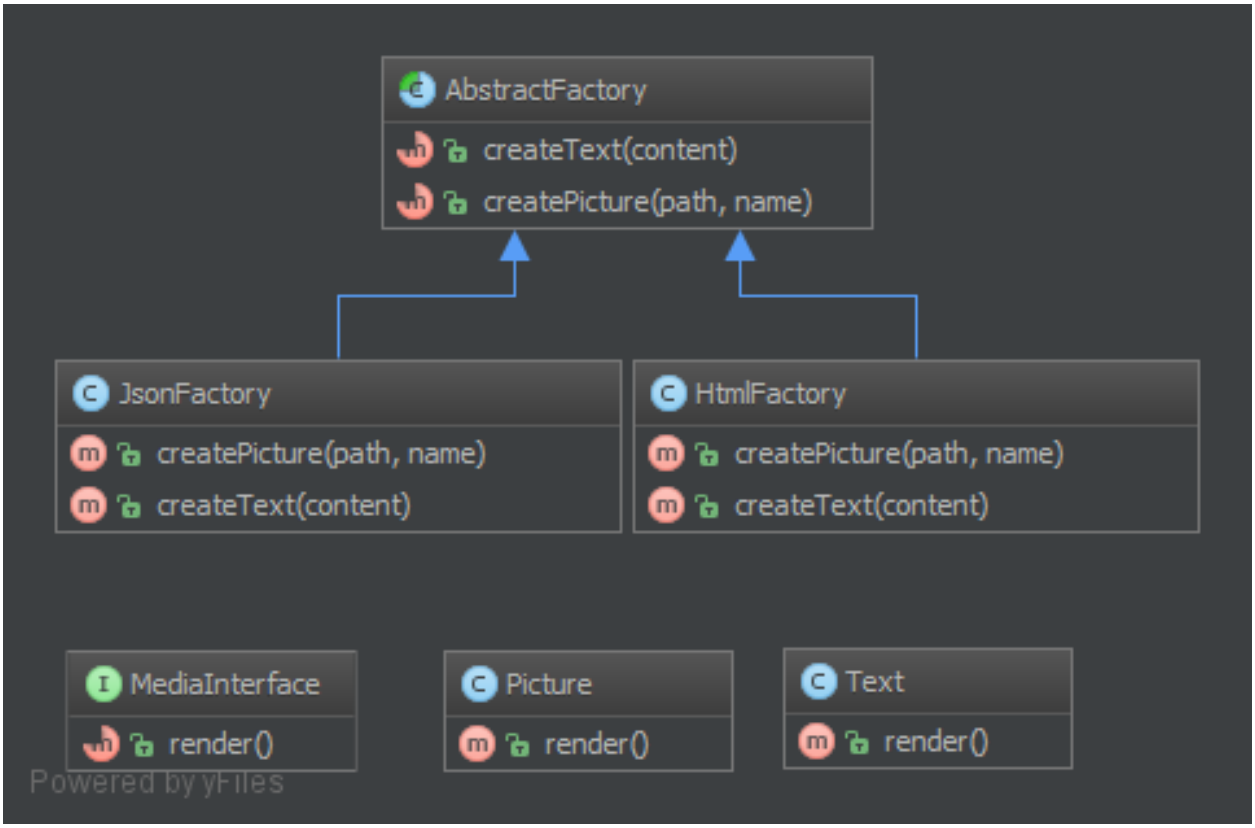
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns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

1.1.1 Abstract Factory

Purpose

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

UML Diagram



Code

You can also find these code on GitHub

AbstractFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * class AbstractFactory
 *
 * Sometimes also known as "Kit" in a GUI libraries.
 *
 * This design pattern implements the Dependency Inversion Principle since
 * it is the concrete subclass which creates concrete components.
 *
 * In this case, the abstract factory is a contract for creating some components
 * for the web. There are two components : Text and Picture. There is two ways
 * of rendering : HTML or JSON.
 *
 * Therefore 4 concretes classes, but the client just need to know this contract
 * to build a correct http response (for a html page or for an ajax request)
 */
abstract class AbstractFactory
{
```

```
    /**
     * Creates a text component
     *
     * @param string $content
     *
     * @return Text
     */
    abstract public function createText($content);

    /**
     * Creates a picture component
     *
     * @param string $path
     * @param string $name
     *
     * @return Picture
     */
    abstract public function createPicture($path, $name = '');
}
```

JsonFactory.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * Class JsonFactory
 *
 * JsonFactory is a factory for creating a family of JSON component
 * (example for ajax)
 */
class JsonFactory extends AbstractFactory
{

    /**
     * Creates a picture component
     *
     * @param string $path
     * @param string $name
     *
     * @return Json\Picture|Picture
     */
    public function createPicture($path, $name = '')
    {
        return new Json\Picture($path, $name);
    }

    /**
     * Creates a text component
     *
     * @param string $content
     *
     * @return Json\Text|Text
     */
    public function createText($content)
    {
        return new Json\Text($content);
    }
```

```
}
```

HtmlFactory.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * Class HtmlFactory
 *
 * HtmlFactory is a concrete factory for HTML component
 */
class HtmlFactory extends AbstractFactory
{
    /**
     * Creates a picture component
     *
     * @param string $path
     * @param string $name
     *
     * @return Html\Picture|Picture
     */
    public function createPicture($path, $name = '')
    {
        return new Html\Picture($path, $name);
    }

    /**
     * Creates a text component
     *
     * @param string $content
     *
     * @return Html\Text|Text
     */
    public function createText($content)
    {
        return new Html\Text($content);
    }
}
```

MediaInterface.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * Interface MediaInterface
 *
 * This contract is not part of the pattern, in general case, each component
 * are not related
 */
interface MediaInterface
{

    /**
     * some crude rendering from JSON or html output (depended on concrete class)
     *
```

```
     * @return string
     */
    public function render();
}
```

Picture.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * Class Picture
 */
abstract class Picture implements MediaInterface
{

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

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

    /**
     * @param string $path
     * @param string $name
     */
    public function __construct($path, $name = '')
    {
        $this->name = (string) $name;
        $this->path = (string) $path;
    }
}
```

Text.php

```
<?php

namespace DesignPatterns\Creational\AbstractFactory;

/**
 * Class Text
 */
abstract class Text implements MediaInterface
{
    /**
     * @var string
     */
    protected $text;

    /**
     * @param string $text
     */
    public function __construct($text)
    {
```

```
        $this->text = (string) $text;
    }
}
```

Json/Picture.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Json;

use DesignPatterns\Creational\AbstractFactory\Picture as BasePicture;

/**
 * Class Picture
 *
 * Picture is a concrete image for JSON rendering
 */
class Picture extends BasePicture
{
    /**
     * some crude rendering from JSON output
     *
     * @return string
     */
    public function render()
    {
        return json_encode(array('title' => $this->name, 'path' => $this->path));
    }
}
```

Json/Text.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Json;

use DesignPatterns\Creational\AbstractFactory\Text as BaseText;

/**
 * Class Text
 *
 * Text is a text component with a JSON rendering
 */
class Text extends BaseText
{
    /**
     * some crude rendering from JSON output
     *
     * @return string
     */
    public function render()
    {
        return json_encode(array('content' => $this->text));
    }
}
```

Html/Picture.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Html;

use DesignPatterns\Creational\AbstractFactory\Picture as BasePicture;

/**
 * Class Picture
 *
 * Picture is a concrete image for HTML rendering
 */
class Picture extends BasePicture
{
    /**
     * some crude rendering from HTML output
     *
     * @return string
     */
    public function render()
    {
        return sprintf('<img src="%s" title="%s"/>', $this->path, $this->name);
    }
}
```

Html/Text.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Html;

use DesignPatterns\Creational\AbstractFactory\Text as BaseText;

/**
 * Class Text
 *
 * Text is a concrete text for HTML rendering
 */
class Text extends BaseText
{
    /**
     * some crude rendering from HTML output
     *
     * @return string
     */
    public function render()
    {
        return '<div>' . htmlspecialchars($this->text) . '</div>';
    }
}
```

Test

Tests/AbstractFactoryTest.php

```php
<?php

namespace DesignPatterns\Creational\AbstractFactory\Tests;
```

```php

use DesignPatterns\Creational\AbstractFactory\AbstractFactory;
use DesignPatterns\Creational\AbstractFactory\HtmlFactory;
use DesignPatterns\Creational\AbstractFactory\JsonFactory;

/**
 * AbstractFactoryTest tests concrete factories
 */
class AbstractFactoryTest extends \PHPUnit_Framework_TestCase
{
    public function getFactories()
    {
        return array(
            array(new JsonFactory()),
            array(new HtmlFactory())
        );
    }

    /**
     * This is the client of factories. Note that the client does not
     * care which factory is given to him, he can create any component he
     * wants and render how he wants.
     *
     * @dataProvider getFactories
     */
    public function testComponentCreation(AbstractFactory $factory)
    {
        $article = array(
            $factory->createText('Lorem Ipsum'),
            $factory->createPicture('/image.jpg', 'caption'),
            $factory->createText('footnotes')
        );

        $this->assertContainsOnly('DesignPatterns\Creational\AbstractFactory\MediaInterface', $artic

        /* this is the time to look at the Builder pattern. This pattern
         * helps you to create complex object like that article above with
         * a given Abstract Factory
         */
    }
}
```

1.1.2 Builder

Purpose

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

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

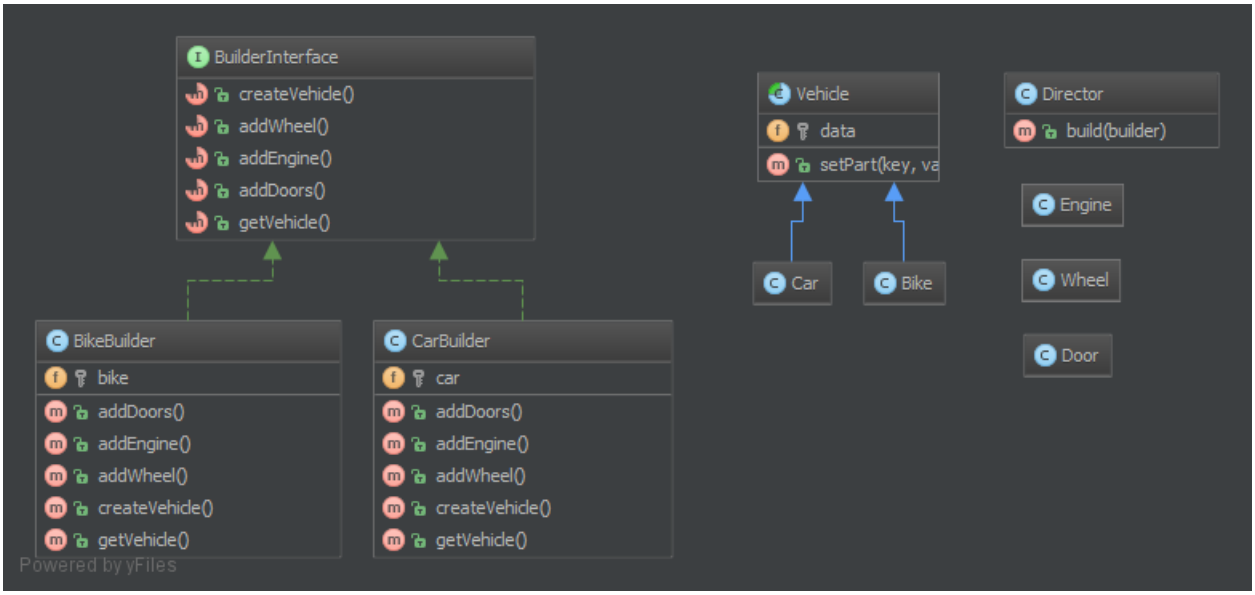
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

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

Examples

- PHPUnit: Mock Builder

UML Diagram



Code

You can also find these code on GitHub

Director.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

/**
 * 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
{

    /**
     * The director don't know about concrete part
     *
     * @param BuilderInterface $builder
     *
     * @return Parts\Vehicle
     */
    public function build(BuilderInterface $builder)
    {
        $builder->createVehicle();
```

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

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

BuilderInterface.php

```
<?php

namespace DesignPatterns\Creational\Builder;

/**
 *
 */
interface BuilderInterface
{
    /**
     * @return mixed
     */
    public function createVehicle();

    /**
     * @return mixed
     */
    public function addWheel();

    /**
     * @return mixed
     */
    public function addEngine();

    /**
     * @return mixed
     */
    public function addDoors();

    /**
     * @return mixed
     */
    public function getVehicle();
}
```

BikeBuilder.php

```
<?php

namespace DesignPatterns\Creational\Builder;

/**
 * BikeBuilder builds bike
 */
class BikeBuilder implements BuilderInterface
{
    /**
     * @var Parts\Bike
     */
```

```php
    protected $bike;

    /**
     * {@inheritdoc}
     */
    public function addDoors()
    {
    }

    /**
     * {@inheritdoc}
     */
    public function addEngine()
    {
        $this->bike->setPart('engine', new Parts\Engine());
    }

    /**
     * {@inheritdoc}
     */
    public function addWheel()
    {
        $this->bike->setPart('forwardWheel', new Parts\Wheel());
        $this->bike->setPart('rearWheel', new Parts\Wheel());
    }

    /**
     * {@inheritdoc}
     */
    public function createVehicle()
    {
        $this->bike = new Parts\Bike();
    }

    /**
     * {@inheritdoc}
     */
    public function getVehicle()
    {
        return $this->bike;
    }
}
```

CarBuilder.php

```php
<?php

namespace DesignPatterns\Creational\Builder;

/**
 * CarBuilder builds car
 */
class CarBuilder implements BuilderInterface
{
    /**
     * @var Parts\Car
     */
    protected $car;
```

```php

    /**
     * @return void
     */
    public function addDoors()
    {
        $this->car->setPart('rightdoor', new Parts\Door());
        $this->car->setPart('leftDoor', new Parts\Door());
    }

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

    /**
     * @return void
     */
    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());
    }

    /**
     * @return void
     */
    public function createVehicle()
    {
        $this->car = new Parts\Car();
    }

    /**
     * @return Parts\Car
     */
    public function getVehicle()
    {
        return $this->car;
    }
}
```

Parts/Vehicle.php

```php
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * VehicleInterface is a contract for a vehicle
 */
abstract class Vehicle
{
    /**
     * @var array
```

```
     */
    protected $data;

    /**
     * @param string $key
     * @param mixed  $value
     */
    public function setPart($key, $value)
    {
        $this->data[$key] = $value;
    }
}
```

Parts/Bike.php

```
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * Bike is a bike
 */
class Bike extends Vehicle
{
}
```

Parts/Car.php

```
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * Car is a car
 */
class Car extends Vehicle
{
}
```

Parts/Engine.php

```
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * Class Engine
 */
class Engine
{
}
```

Parts/Wheel.php

```
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * Class Wheel
```

```
 */
class Wheel
{
}
```

Parts/Door.php

```
<?php

namespace DesignPatterns\Creational\Builder\Parts;

/**
 * Class Door
 */
class Door
{
}
```

Test

Tests/DirectorTest.php

```
<?php

namespace DesignPatterns\Creational\Builder\Tests;

use DesignPatterns\Creational\Builder\Director;
use DesignPatterns\Creational\Builder\CarBuilder;
use DesignPatterns\Creational\Builder\BikeBuilder;
use DesignPatterns\Creational\Builder\BuilderInterface;

/**
 * DirectorTest tests the builder pattern
 */
class DirectorTest extends \PHPUnit_Framework_TestCase
{

    protected $director;

    protected function setUp()
    {
        $this->director = new Director();
    }

    public function getBuilder()
    {
        return array(
            array(new CarBuilder()),
            array(new BikeBuilder())
        );
    }

    /**
     * Here we test the build process. Notice that the client don't know
     * anything about the concrete builder.
     *
     * @dataProvider getBuilder
```

```
36          */
37      public function testBuild(BuilderInterface $builder)
38      {
39          $newVehicle = $this->director->build($builder);
40          $this->assertInstanceOf('DesignPatterns\Creational\Builder\Parts\Vehicle', $newVehicle);
41      }
42  }
```

1.1.3 Factory Method

Purpose

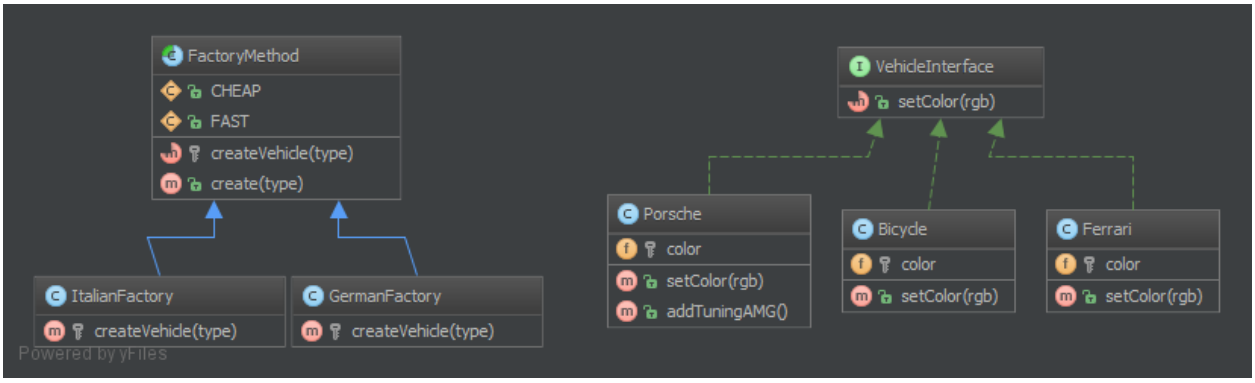
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects

For simple case, this abstract class could be just an interface

This pattern is a "real" Design Pattern because it achieves the "Dependency Inversion Principle" a.k.a the "D" in S.O.L.I.D principles.

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

UML Diagram



Code

You can also find these code on GitHub

FactoryMethod.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\FactoryMethod;
4
5   /**
6    * class FactoryMethod
7    */
8   abstract class FactoryMethod
9   {
10
11      const CHEAP = 1;
```

```
    const FAST = 2;

    /**
     * The children of the class must implement this method
     *
     * Sometimes this method can be public to get "raw" object
     *
     * @param string $type a generic type
     *
     * @return VehicleInterface a new vehicle
     */
    abstract protected function createVehicle($type);

    /**
     * Creates a new vehicle
     *
     * @param int $type
     *
     * @return VehicleInterface a new vehicle
     */
    public function create($type)
    {
        $obj = $this->createVehicle($type);
        $obj->setColor("#f00");

        return $obj;
    }
}
```

ItalianFactory.php

```
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
 * ItalianFactory is vehicle factory in Italy
 */
class ItalianFactory extends FactoryMethod
{
    /**
     * {@inheritdoc}
     */
    protected function createVehicle($type)
    {
        switch ($type) {
            case parent::CHEAP:
                return new Bicycle();
                break;
            case parent::FAST:
                return new Ferrari();
                break;
            default:
                throw new \InvalidArgumentException("$type is not a valid vehicle");
        }
    }
}
```

GermanFactory.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
 * GermanFactory is a vehicle factory in Germany
 */
class GermanFactory extends FactoryMethod
{
    /**
     * {@inheritdoc}
     */
    protected function createVehicle($type)
    {
        switch ($type) {
            case parent::CHEAP:
                return new Bicycle();
                break;
            case parent::FAST:
                $obj = new Porsche();
                // we can specialize the way we want some concrete Vehicle since
                // we know the class
                $obj->addTuningAMG();

                return $obj;
                break;
            default:
                throw new \InvalidArgumentException("$type is not a valid vehicle");
        }
    }
}
```

VehicleInterface.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
 * VehicleInterface is a contract for a vehicle
 */
interface VehicleInterface
{
    /**
     * sets the color of the vehicle
     *
     * @param string $rgb
     */
    public function setColor($rgb);
}
```

Porsche.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
 * Porsche is a german car
```

```php
 */
class Porsche implements VehicleInterface
{
    /**
     * @var string
     */
    protected $color;

    /**
     * @param string $rgb
     */
    public function setColor($rgb)
    {
        $this->color = $rgb;
    }

    /**
     * although tuning by AMG is only offered for Mercedes Cars,
     * this is a valid coding example ...
     */
    public function addTuningAMG()
    {
    }
}
```

Bicycle.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
 * Bicycle is a bicycle
 */
class Bicycle implements VehicleInterface
{
    /**
     * @var string
     */
    protected $color;

    /**
     * sets the color of the bicycle
     *
     * @param string $rgb
     */
    public function setColor($rgb)
    {
        $this->color = $rgb;
    }
}
```

Ferrari.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod;

/**
```

```php
 * Ferrari is a italian car
 */
class Ferrari implements VehicleInterface
{
    /**
     * @var string
     */
    protected $color;

    /**
     * @param string $rgb
     */
    public function setColor($rgb)
    {
        $this->color = $rgb;
    }
}
```

Test

Tests/FactoryMethodTest.php

```php
<?php

namespace DesignPatterns\Creational\FactoryMethod\Tests;

use DesignPatterns\Creational\FactoryMethod\FactoryMethod;
use DesignPatterns\Creational\FactoryMethod\GermanFactory;
use DesignPatterns\Creational\FactoryMethod\ItalianFactory;

/**
 * FactoryMethodTest tests the factory method pattern
 */
class FactoryMethodTest extends \PHPUnit_Framework_TestCase
{

    protected $type = array(
        FactoryMethod::CHEAP,
        FactoryMethod::FAST
    );

    public function getShop()
    {
        return array(
            array(new GermanFactory()),
            array(new ItalianFactory())
        );
    }

    /**
     * @dataProvider getShop
     */
    public function testCreation(FactoryMethod $shop)
    {
        // this test method acts as a client for the factory. We don't care
        // about the factory, all we know is it can produce vehicle
        foreach ($this->type as $oneType) {
```

```
            $vehicle = $shop->create($oneType);
            $this->assertInstanceOf('DesignPatterns\Creational\FactoryMethod\VehicleInterface', $vehi
        }
    }

    /**
     * @dataProvider getShop
     * @expectedException \InvalidArgumentException
     * @expectedExceptionMessage spaceship is not a valid vehicle
     */
    public function testUnknownType(FactoryMethod $shop)
    {
        $shop->create('spaceship');
    }
}
```

1.1.4 Multiton

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

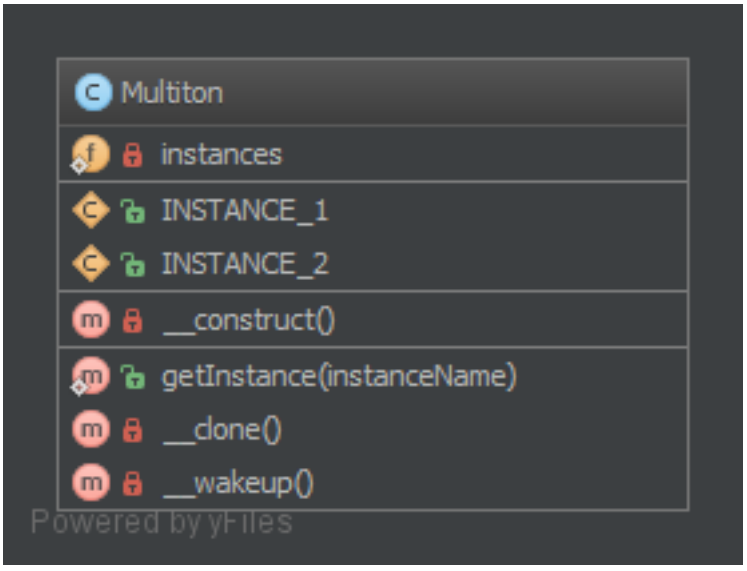
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

Multiton.php

```php
<?php

namespace DesignPatterns\Creational\Multiton;

/**
 * class Multiton
 */
class Multiton
{
    /**
     *
     * the first instance
     */
    const INSTANCE_1 = '1';

    /**
     *
     * the second instance
     */
    const INSTANCE_2 = '2';

    /**
     * holds the named instances
     *
     * @var array
     */
    private static $instances = array();

    /**
     * should not be called from outside: private!
     *
     */
    private function __construct()
    {
    }

    /**
     * gets the instance with the given name, e.g. Multiton::INSTANCE_1
     * uses lazy initialization
     *
     * @param string $instanceName
     *
     * @return Multiton
     */
    public static function getInstance($instanceName)
    {
        if (!array_key_exists($instanceName, self::$instances)) {
            self::$instances[$instanceName] = new self();
        }

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

```

```php
    /**
     * prevent instance from being cloned
     *
     * @return void
     */
    private function __clone()
    {
    }

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

Test

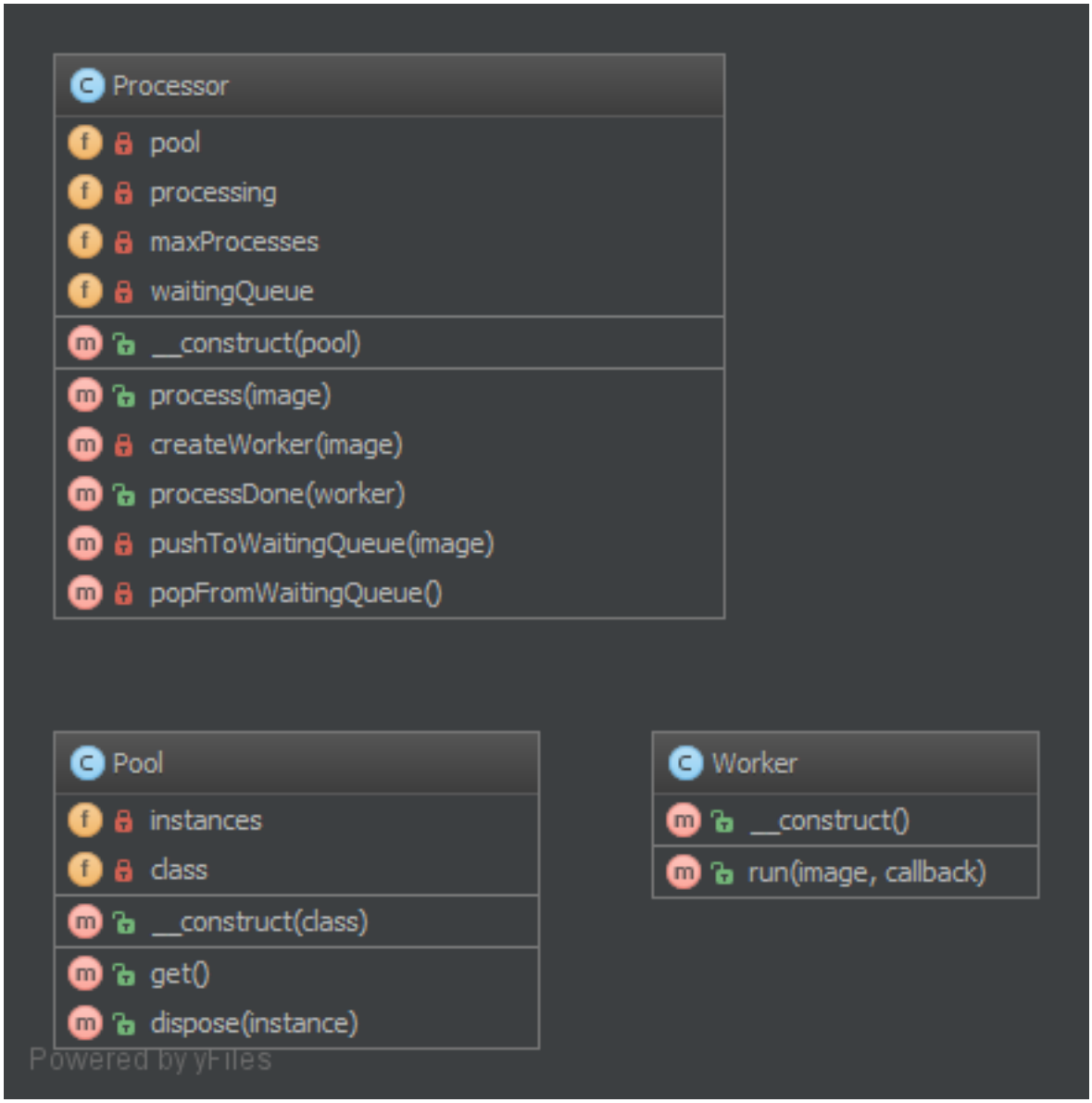
1.1.5 Pool

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

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

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

UML Diagram



Code

You can also find these code on GitHub

Pool.php

```php
<?php

namespace DesignPatterns\Creational\Pool;

class Pool
{

    private $instances = array();
```

```
    private $class;

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

    public function get()
    {
        if (count($this->instances) > 0) {
            return array_pop($this->instances);
        }

        return new $this->class();
    }

    public function dispose($instance)
    {
        $this->instances[] = $instance;
    }
}
```

Processor.php

```
<?php

namespace DesignPatterns\Creational\Pool;

class Processor
{

    private $pool;
    private $processing = 0;
    private $maxProcesses = 3;
    private $waitingQueue = [];

    public function __construct(Pool $pool)
    {
        $this->pool = $pool;
    }

    public function process($image)
    {
        if ($this->processing++ < $this->maxProcesses) {
            $this->createWorker($image);
        } else {
            $this->pushToWaitingQueue($image);
        }
    }

    private function createWorker($image)
    {
        $worker = $this->pool->get();
        $worker->run($image, array($this, 'processDone'));
    }

    public function processDone($worker)
    {
```

```
        $this->processing--;
        $this->pool->dispose($worker);

        if (count($this->waitingQueue) > 0) {
            $this->createWorker($this->popFromWaitingQueue());
        }
    }

    private function pushToWaitingQueue($image)
    {
        $this->waitingQueue[] = $image;
    }

    private function popFromWaitingQueue()
    {
        return array_pop($this->waitingQueue);
    }
}
```

Worker.php

```
<?php

namespace DesignPatterns\Creational\Pool;

class Worker
{

    public function __construct()
    {
        // let's say that constuctor does really expensive work...
        // for example creates "thread"
    }

    public function run($image, array $callback)
    {
        // do something with $image...
        // and when it's done, execute callback
        call_user_func($callback, $this);
    }
}
```

Test

Tests/PoolTest.php

```
<?php

namespace DesignPatterns\Creational\Pool\Tests;

use DesignPatterns\Creational\Pool\Pool;

class PoolTest extends \PHPUnit_Framework_TestCase
{
    public function testPool()
    {
        $pool = new Pool('DesignPatterns\Creational\Pool\Tests\TestWorker');
```

```
        $worker = $pool->get();

        $this->assertEquals(1, $worker->id);

        $worker->id = 5;
        $pool->dispose($worker);

        $this->assertEquals(5, $pool->get()->id);
        $this->assertEquals(1, $pool->get()->id);
    }
}
```

Tests/TestWorker.php

```
<?php

namespace DesignPatterns\Creational\Pool\Tests;

class TestWorker
{
    public $id = 1;
}
```

1.1.6 Prototype

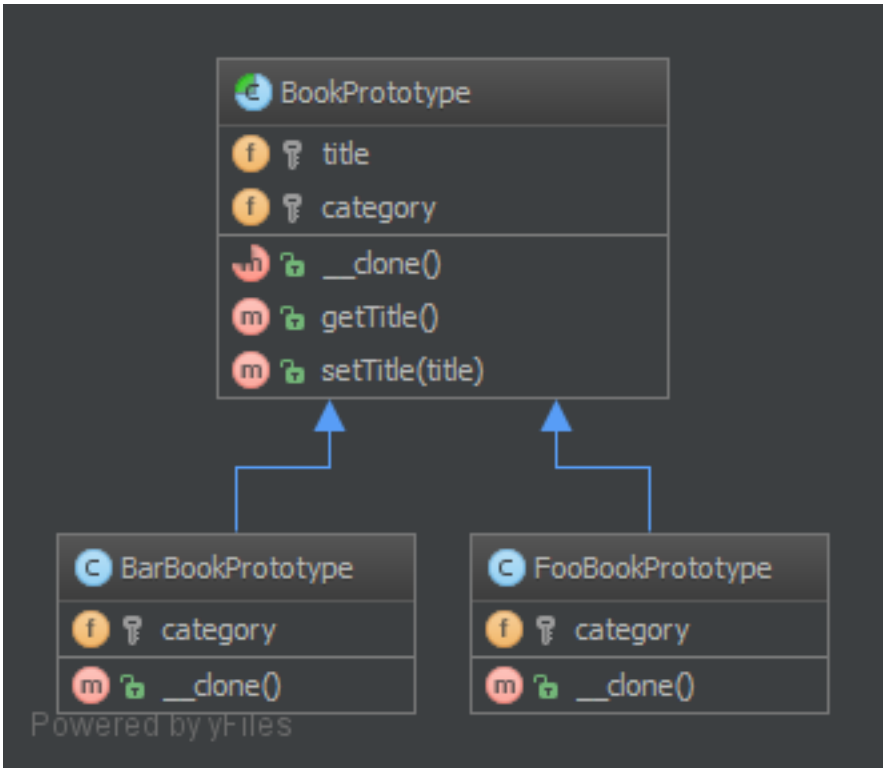
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

index.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

$fooPrototype = new FooBookPrototype();
$barPrototype = new BarBookPrototype();

// now lets say we need 10,000 books of foo and 5,000 of bar ...
for ($i = 0; $i < 10000; $i++) {
    $book = clone $fooPrototype;
    $book->setTitle('Foo Book No ' . $i);
}

for ($i = 0; $i < 5000; $i++) {
    $book = clone $barPrototype;
    $book->setTitle('Bar Book No ' . $i);
}
```

BookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;
```

```php

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

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

    /**
     * @abstract
     * @return void
     */
    abstract public function __clone();

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

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

BarBookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

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

    /**
     * empty clone
     */
    public function __clone()
```

```
    {
    }
}
```

FooBookPrototype.php

```php
<?php

namespace DesignPatterns\Creational\Prototype;

/**
 * Class FooBookPrototype
 */
class FooBookPrototype extends BookPrototype
{
    protected $category = 'Foo';

    /**
     * empty clone
     */
    public function __clone()
    {
    }
}
```

Test

1.1.7 Simple Factory

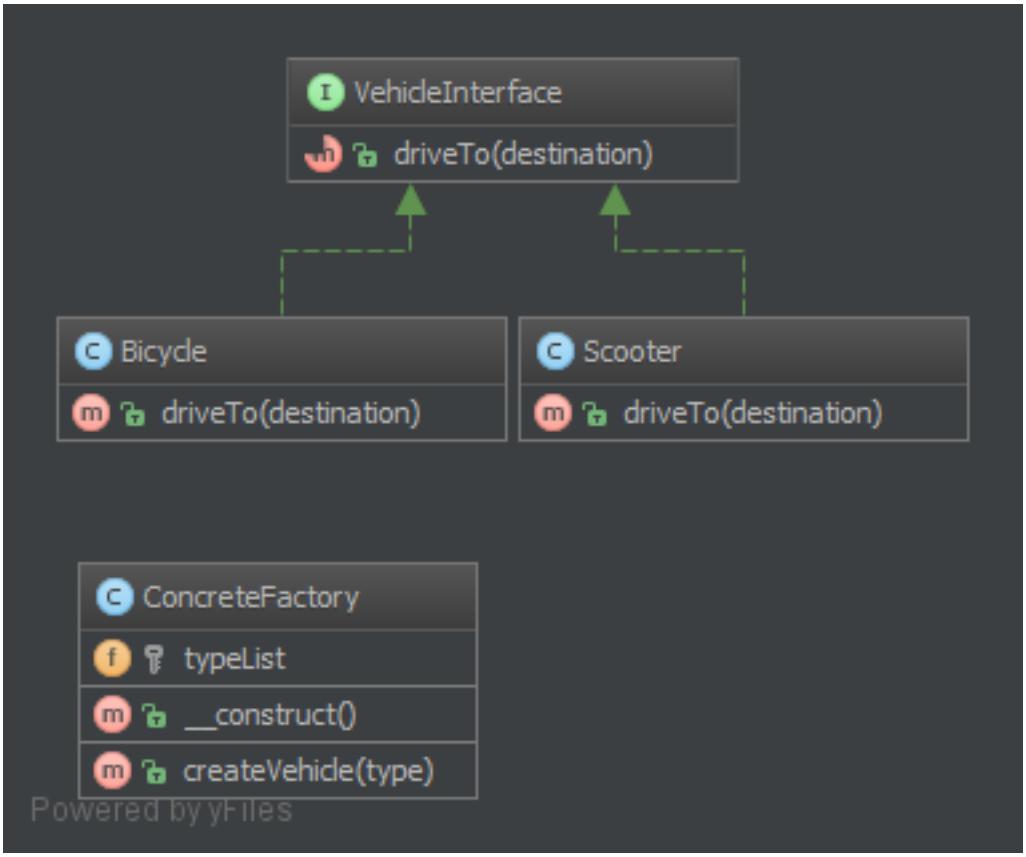
Purpose

ConcreteFactory is a simple factory pattern.

It differs from the static factory because it is NOT static and as you know: static => global => evil!

Therefore, you can have multiple factories, differently parametrized, you can subclass it and you can mock-up it.

UML Diagram



Code

You can also find these code on GitHub

ConcreteFactory.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

/**
 * class ConcreteFactory
 */
class ConcreteFactory
{
    /**
     * @var array
     */
    protected $typeList;

    /**
     * You can imagine to inject your own type list or merge with
     * the default ones...
     */
    public function __construct()
    {
```

```php
        $this->typeList = array(
            'bicycle' => __NAMESPACE__ . '\Bicycle',
            'other' => __NAMESPACE__ . '\Scooter'
        );
    }

    /**
     * Creates a vehicle
     *
     * @param string $type a known type key
     *
     * @return VehicleInterface a new instance of VehicleInterface
     * @throws \InvalidArgumentException
     */
    public function createVehicle($type)
    {
        if (!array_key_exists($type, $this->typeList)) {
            throw new \InvalidArgumentException("$type is not valid vehicle");
        }
        $className = $this->typeList[$type];

        return new $className();
    }
}
```

VehicleInterface.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

/**
 * VehicleInterface is a contract for a vehicle
 */
interface VehicleInterface
{
    /**
     * @param mixed $destination
     *
     * @return mixed
     */
    public function driveTo($destination);
}
```

Bicycle.php

```php
<?php

namespace DesignPatterns\Creational\SimpleFactory;

/**
 * Bicycle is a bicycle
 */
class Bicycle implements VehicleInterface
{
    /**
     * @param mixed $destination
     *
     * @return mixed|void
```

```
     */
    public function driveTo($destination)
    {
    }
}
```

Scooter.php

```
<?php

namespace DesignPatterns\Creational\SimpleFactory;

/**
 * Scooter is a Scooter
 */
class Scooter implements VehicleInterface
{
    /**
     * @param mixed $destination
     */
    public function driveTo($destination)
    {
    }
}
```

Test

Tests/SimpleFactoryTest.php

```
<?php

namespace DesignPatterns\Creational\SimpleFactory\Tests;

use DesignPatterns\Creational\SimpleFactory\ConcreteFactory;

/**
 * SimpleFactoryTest tests the Simple Factory pattern
 */
class SimpleFactoryTest extends \PHPUnit_Framework_TestCase
{

    protected $factory;

    protected function setUp()
    {
        $this->factory = new ConcreteFactory();
    }

    public function getType()
    {
        return array(
            array('bicycle'),
            array('other')
        );
    }

    /**
```

```
29          * @dataProvider getType
30          */
31         public function testCreation($type)
32         {
33             $obj = $this->factory->createVehicle($type);
34             $this->assertInstanceOf('DesignPatterns\Creational\SimpleFactory\VehicleInterface', $obj);
35         }
36
37         /**
38          * @expectedException \InvalidArgumentException
39          */
40         public function testBadType()
41         {
42             $this->factory->createVehicle('car');
43         }
44 }
```

1.1.8 Singleton

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

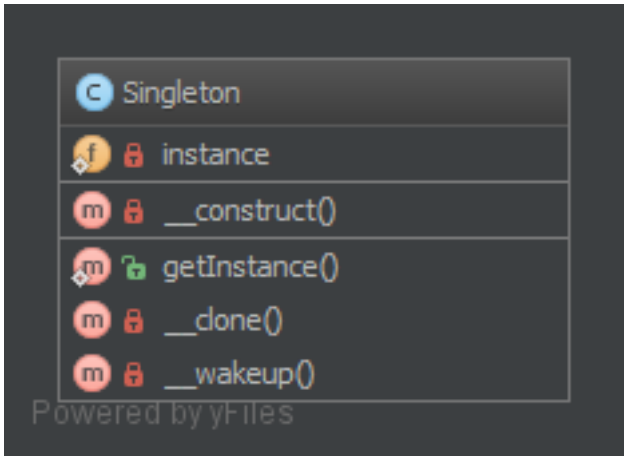
Purpose

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

Examples

- DB Connector
- Logger (may also be a Multiton if there are many log files for several purposes)
- Lock file for the application (there is only one in the filesystem ...)

UML Diagram



Code

You can also find these code on GitHub

Singleton.php

```php
<?php

namespace DesignPatterns\Creational\Singleton;

/**
 * class Singleton
 */
class Singleton
{
    /**
     * @var Singleton reference to singleton instance
     */
    private static $instance;

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

        return static::$instance;
    }

    /**
     * is not allowed to call from outside: private!
     *
     */
    private function __construct()
    {
    }

    /**
     * prevent the instance from being cloned
     *
     * @return void
     */
    private function __clone()
    {
    }

    /**
     * prevent from being unserialized
     *
     * @return void
     */
    private function __wakeup()
    {
    }
```

```
}
```

Test

Tests/SingletonTest.php

```php
<?php

namespace DesignPatterns\Creational\Singleton\Tests;

use DesignPatterns\Creational\Singleton\Singleton;

/**
 * SingletonTest tests the singleton pattern
 */
class SingletonTest extends \PHPUnit_Framework_TestCase
{

    public function testUniqueness()
    {
        $firstCall = Singleton::getInstance();
        $this->assertInstanceOf('DesignPatterns\Creational\Singleton\Singleton', $firstCall);
        $secondCall = Singleton::getInstance();
        $this->assertSame($firstCall, $secondCall);
    }

    public function testNoConstructor()
    {
        $obj = Singleton::getInstance();

        $refl = new \ReflectionObject($obj);
        $meth = $refl->getMethod('__construct');
        $this->assertTrue($meth->isPrivate());
    }
}
```

1.1.9 Static Factory

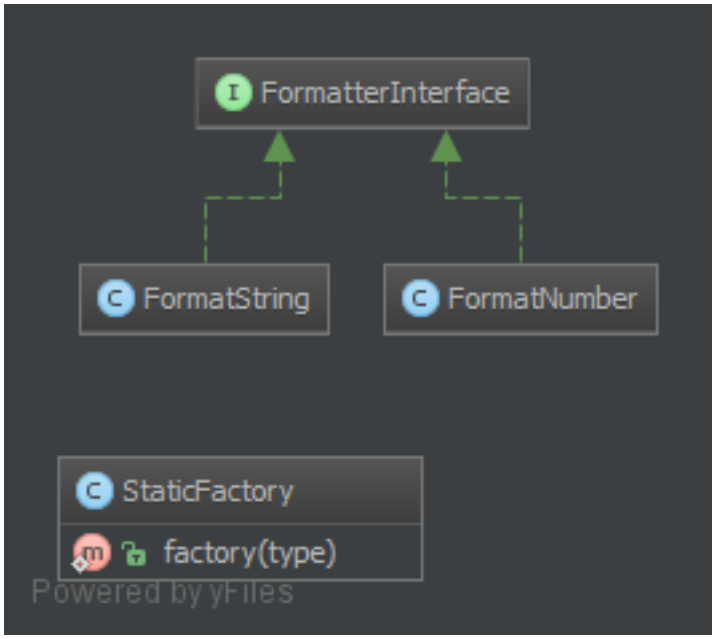
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

StaticFactory.php

```php
<?php

namespace DesignPatterns\Creational\StaticFactory;

/**
 * Note1: Remember, static => global => evil
 * Note2: Cannot be subclassed or mock-upped or have multiple different instances
 */
class StaticFactory
{
    /**
     * the parametrized function to get create an instance
     *
     * @param string $type
     *
     * @static
     *
     * @throws \InvalidArgumentException
     * @return FormatterInterface
     */
    public static function factory($type)
    {
        $className = __NAMESPACE__ . '\Format' . ucfirst($type);

        if (!class_exists($className)) {
            throw new \InvalidArgumentException('Missing format class.');
        }
```

```
28
29          return new $className();
30      }
31  }
```

FormatterInterface.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   /**
6    * Class FormatterInterface
7    */
8   interface FormatterInterface
9   {
10  }
```

FormatString.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   /**
6    * Class FormatString
7    */
8   class FormatString implements FormatterInterface
9   {
10  }
```

FormatNumber.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory;
4
5   /**
6    * Class FormatNumber
7    */
8   class FormatNumber implements FormatterInterface
9   {
10  }
```

Test

Tests/StaticFactoryTest.php

```
1   <?php
2
3   namespace DesignPatterns\Creational\StaticFactory\Tests;
4
5   use DesignPatterns\Creational\StaticFactory\StaticFactory;
6
7   /**
8    * Tests for Static Factory pattern
9    *
10   */
```

```php
class StaticFactoryTest extends \PHPUnit_Framework_TestCase
{

    public function getTypeList()
    {
        return array(
            array('string'),
            array('number')
        );
    }

    /**
     * @dataProvider getTypeList
     */
    public function testCreation($type)
    {
        $obj = StaticFactory::factory($type);
        $this->assertInstanceOf('DesignPatterns\Creational\StaticFactory\FormatterInterface', $obj);
    }

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

1.2 Structural

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

1.2.1 Adapter / Wrapper

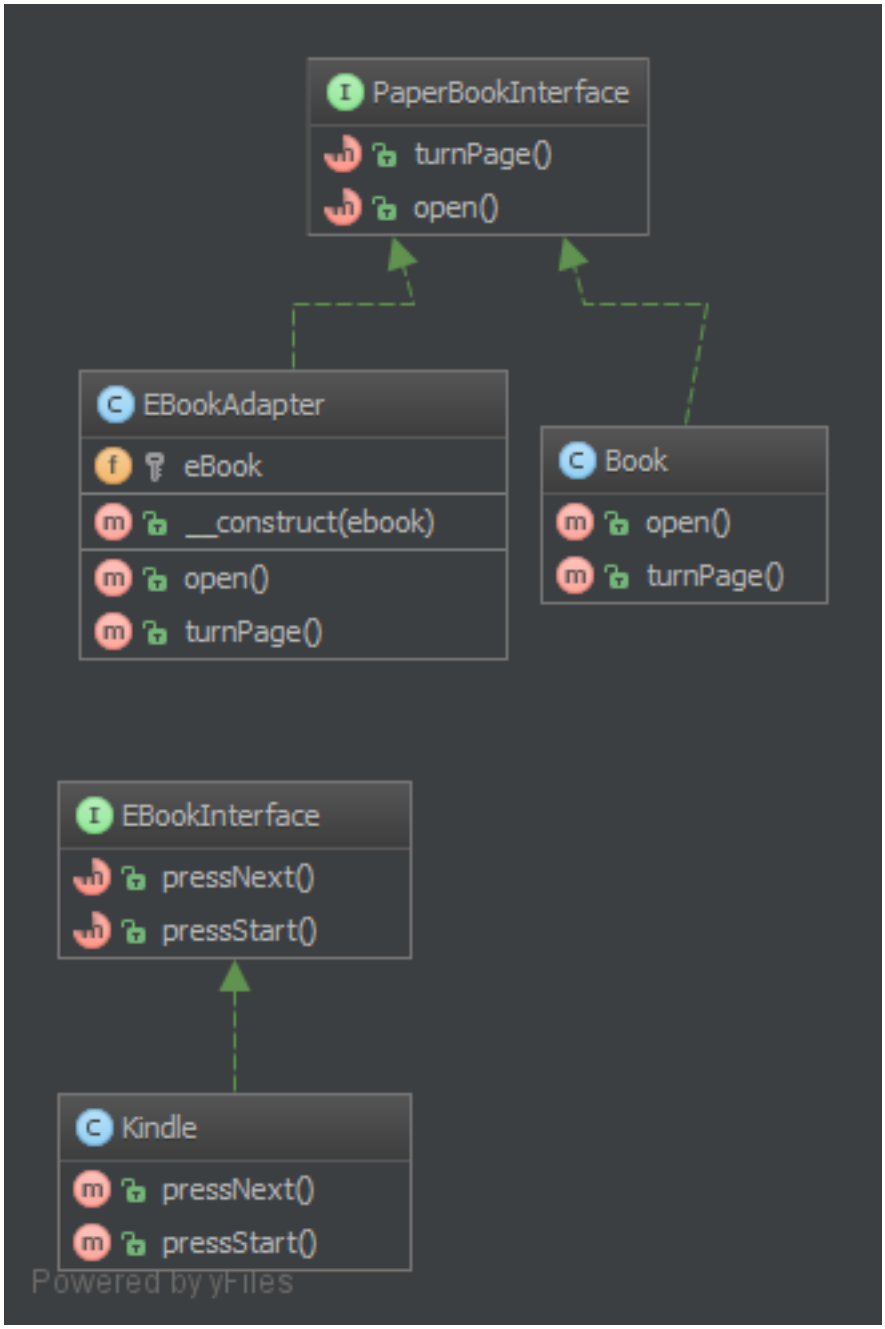
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

PaperBookInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Adapter;
```

```php

/**
 * PaperBookInterface is a contract for a book
 */
interface PaperBookInterface
{
    /**
     * method to turn pages
     *
     * @return mixed
     */
    public function turnPage();

    /**
     * method to open the book
     *
     * @return mixed
     */
    public function open();
}
```

Book.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * Book is a concrete and standard paper book
 */
class Book implements PaperBookInterface
{
    /**
     * {@inheritdoc}
     */
    public function open()
    {
    }

    /**
     * {@inheritdoc}
     */
    public function turnPage()
    {
    }
}
```

EBookAdapter.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * EBookAdapter is an adapter to fit an e-book like a paper book
 *
 * This is the adapter here. Notice it implements PaperBookInterface,
 * therefore you don't have to change the code of the client which using paper book.
 */
```

```php
class EBookAdapter implements PaperBookInterface
{
    /**
     * @var EBookInterface
     */
    protected $eBook;

    /**
     * Notice the constructor, it "wraps" an electronic book
     *
     * @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->pressStart();
    }

    /**
     * turns pages
     */
    public function turnPage()
    {
        $this->eBook->pressNext();
    }
}
```

EBookInterface.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * EBookInterface is a contract for an electronic book
 */
interface EBookInterface
{
    /**
     * go to next page
     *
     * @return mixed
     */
    public function pressNext();

    /**
     * start the book
     *
     * @return mixed
     */
    public function pressStart();
```

```
}
```

Kindle.php

```php
<?php

namespace DesignPatterns\Structural\Adapter;

/**
 * Kindle is a concrete electronic book
 */
class Kindle implements EBookInterface
{
    /**
     * {@inheritdoc}
     */
    public function pressNext()
    {
    }

    /**
     * {@inheritdoc}
     */
    public function pressStart()
    {
    }
}
```

Test

Tests/AdapterTest.php

```php
<?php

namespace DesignPatterns\Structural\Adapter\Tests;

use DesignPatterns\Structural\Adapter\EBookAdapter;
use DesignPatterns\Structural\Adapter\Kindle;
use DesignPatterns\Structural\Adapter\PaperBookInterface;
use DesignPatterns\Structural\Adapter\Book;

/**
 * AdapterTest shows the use of an adapted e-book that behave like a book
 * You don't have to change the code of your client
 */
class AdapterTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @return array
     */
    public function getBook()
    {
        return array(
            array(new Book()),
            // we build a "wrapped" electronic book in the adapter
            array(new EBookAdapter(new Kindle()))
        );
```

```
26      }
27
28      /**
29       * This client only knows paper book but surprise, surprise, the second book
30       * is in fact an electronic book, but both work the same way
31       *
32       * @param PaperBookInterface $book
33       *
34       * @dataProvider getBook
35       */
36      public function testIAmAnOldClient(PaperBookInterface $book)
37      {
38          $this->assertTrue(method_exists($book, 'open'));
39          $this->assertTrue(method_exists($book, 'turnPage'));
40      }
41  }
```

1.2.2 Bridge

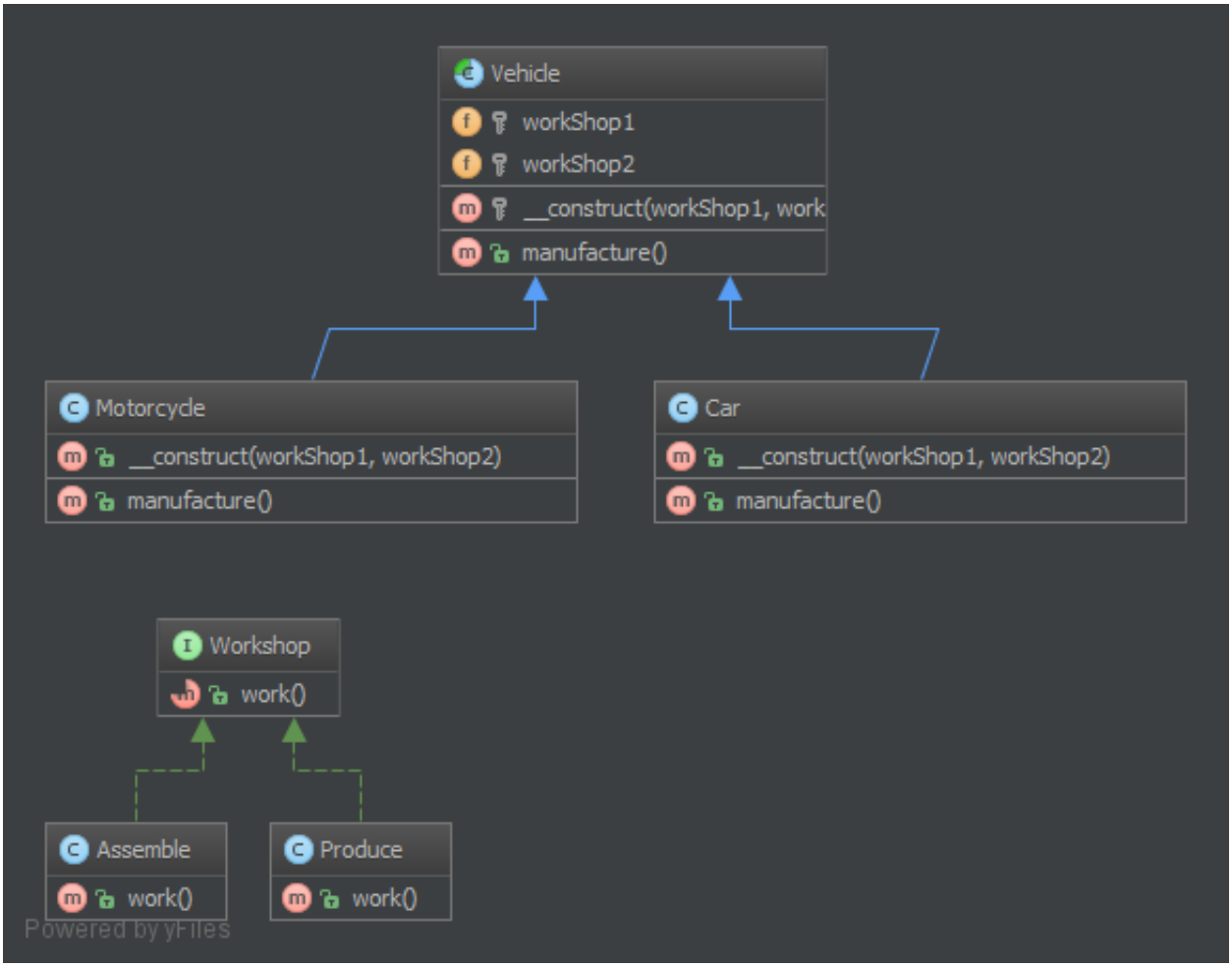
Purpose

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

Sample:

- Symfony DoctrineBridge

UML Diagram



Code

You can also find these code on GitHub

Workshop.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

/**
 * Implementer
 */
interface Workshop
{

    public function work();
}
```

Assemble.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

class Assemble implements Workshop
{

    public function work()
    {
        print 'Assembled';
    }
}
```

Produce.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

/**
 * Concrete Implementation
 */
class Produce implements Workshop
{

    public function work()
    {
        print 'Produced ';
    }
}
```

Vehicle.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

/**
 * Abstraction
 */
abstract class Vehicle
{

    protected $workShop1;
    protected $workShop2;

    protected function __construct(Workshop $workShop1, Workshop $workShop2)
    {
        $this->workShop1 = $workShop1;
        $this->workShop2 = $workShop2;
    }

    public function manufacture()
    {
    }
}
```

Motorcycle.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

/**
 * Refined Abstraction
 */
class Motorcycle extends Vehicle
{

    public function __construct(Workshop $workShop1, Workshop $workShop2)
    {
        parent::__construct($workShop1, $workShop2);
    }

    public function manufacture()
    {
        print 'Motorcycle ';
        $this->workShop1->work();
        $this->workShop2->work();
    }
}
```

Car.php

```php
<?php

namespace DesignPatterns\Structural\Bridge;

/**
 * Refined Abstraction
 */
class Car extends Vehicle
{

    public function __construct(Workshop $workShop1, Workshop $workShop2)
    {
        parent::__construct($workShop1, $workShop2);
    }

    public function manufacture()
    {
        print 'Car ';
        $this->workShop1->work();
        $this->workShop2->work();
    }
}
```

Test

Tests/BridgeTest.php

```php
<?php

namespace DesignPatterns\Structural\Bridge\Tests;

use DesignPatterns\Structural\Bridge\Assemble;
```

```php
use DesignPatterns\Structural\Bridge\Car;
use DesignPatterns\Structural\Bridge\Motorcycle;
use DesignPatterns\Structural\Bridge\Produce;

class BridgeTest extends \PHPUnit_Framework_TestCase
{

    public function testCar()
    {
        $vehicle = new Car(new Produce(), new Assemble());
        $this->expectOutputString('Car Produced Assembled');
        $vehicle->manufacture();
    }

    public function testMotorcycle()
    {
        $vehicle = new Motorcycle(new Produce(), new Assemble());
        $this->expectOutputString('Motorcycle Produced Assembled');
        $vehicle->manufacture();
    }
}
```

1.2.3 Composite

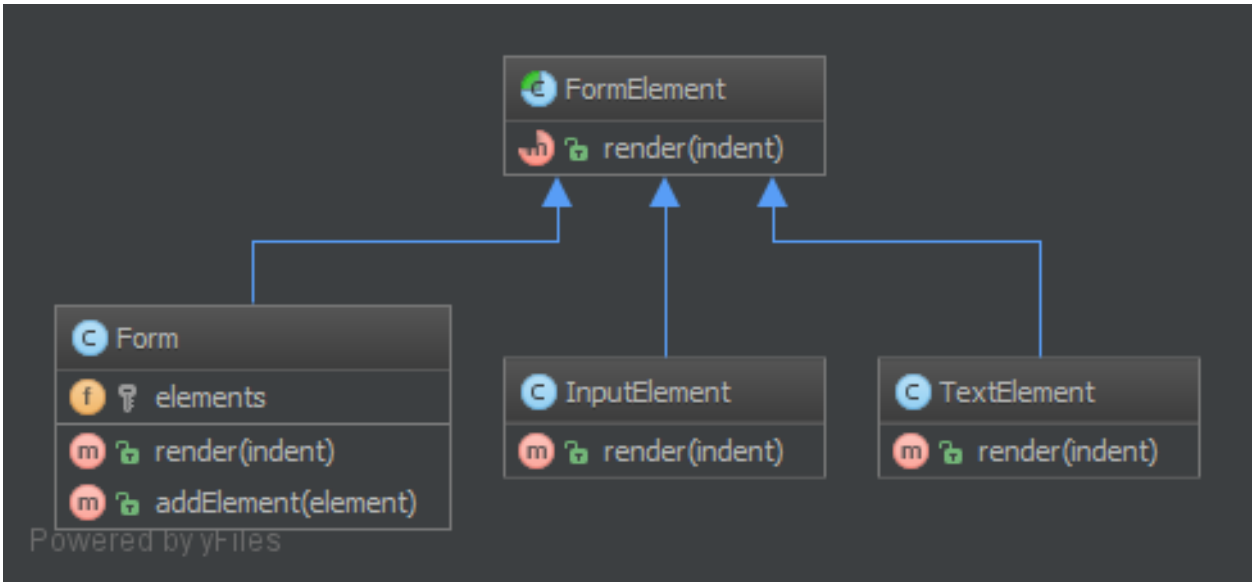
Purpose

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

Examples

- a form class instance handles all its form elements like a single instance of the form, when `render()` is called, it subsequently runs through all its child elements and calls `render()` on them
- `Zend_Config`: a tree of configuration options, each one is a `Zend_Config` object itself

UML Diagram



Code

You can also find these code on GitHub

FormElement.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

/**
 * Class FormElement
 */
abstract class FormElement
{
    /**
     * renders the elements' code
     *
     * @param int $indent
     *
     * @return mixed
     */
    abstract public function render($indent = 0);
}
```

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.
 */
```

```php
class Form extends FormElement
{
    /**
     * @var array|FormElement[]
     */
    protected $elements;

    /**
     * runs through all elements and calls render() on them, then returns the complete representation
     * of the form
     *
     * from the outside, one will not see this and the form will act like a single object instance
     *
     * @param int $indent
     *
     * @return string
     */
    public function render($indent = 0)
    {
        $formCode = '';

        foreach ($this->elements as $element) {
            $formCode .= $element->render($indent + 1) . PHP_EOL;
        }

        return $formCode;
    }

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

InputElement.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

/**
 * Class InputElement
 */
class InputElement extends FormElement
{
    /**
     * renders the input element HTML
     *
     * @param int $indent
     *
     * @return mixed|string
     */
    public function render($indent = 0)
    {
        return str_repeat('  ', $indent) . '<input type="text" />';
    }
```

```
}
```

TextElement.php

```php
<?php

namespace DesignPatterns\Structural\Composite;

/**
 * Class TextElement
 */
class TextElement extends FormElement
{
    /**
     * renders the text element
     *
     * @param int $indent
     *
     * @return mixed|string
     */
    public function render($indent = 0)
    {
        return str_repeat('  ', $indent) . 'this is a text element';
    }
}
```

Test

Tests/CompositeTest.php

```php
<?php

namespace DesignPatterns\Structural\Composite\Tests;

use DesignPatterns\Structural\Composite;

/**
 * FormTest tests the composite pattern on Form
 */
class CompositeTest extends \PHPUnit_Framework_TestCase
{

    public function testRender()
    {
        $form = new Composite\Form();
        $form->addElement(new Composite\TextElement());
        $form->addElement(new Composite\InputElement());
        $embed = new Composite\Form();
        $embed->addElement(new Composite\TextElement());
        $embed->addElement(new Composite\InputElement());
        $form->addElement($embed);  // here we have a embedded form (like SF2 does)

        $this->assertRegExp('#^\s{4}#m', $form->render());
    }

    /**
     * The all point of this pattern, a Composite must inherit from the node
```

```
28          * if you want to builld trees
29          */
30         public function testFormImplementsFormEelement()
31         {
32             $className = 'DesignPatterns\Structural\Composite\Form';
33             $abstractName = 'DesignPatterns\Structural\Composite\FormElement';
34             $this->assertTrue(is_subclass_of($className, $abstractName));
35         }
36 }
```

1.2.4 Data Mapper

Purpose

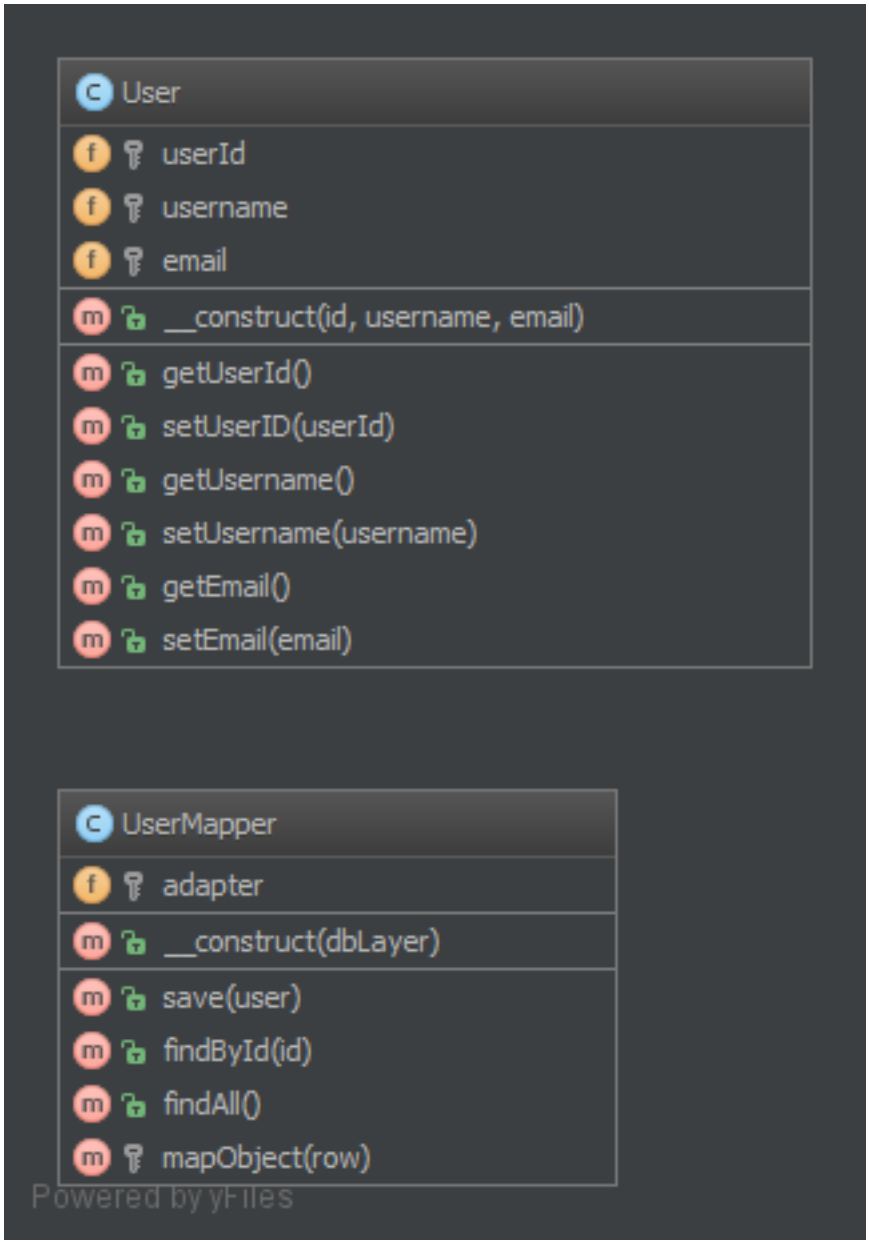
A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent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

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

Examples

- DB Object Relational Mapper (ORM) : Doctrine2 uses DAO named as “EntityRepository”

UML Diagram



Code

You can also find these code on GitHub

User.php

```
<?php

namespace DesignPatterns\Structural\DataMapper;

/**
 * DataMapper pattern
```

```php
 *
 * This is our representation of a DataBase record in the memory (Entity)
 *
 * Validation would also go in this object
 *
 */
class User
{
    /**
     * @var int
     */
    protected $userId;

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

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

    /**
     * @param null $id
     * @param null $username
     * @param null $email
     */
    public function __construct($id = null, $username = null, $email = null)
    {
        $this->userId = $id;
        $this->username = $username;
        $this->email = $email;
    }

    /**
     * @return int
     */
    public function getUserId()
    {
        return $this->userId;
    }

    /**
     * @param int $userId
     */
    public function setUserID($userId)
    {
        $this->userId = $userId;
    }

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

```
65
66      /**
67       * @param string $username
68       */
69      public function setUsername($username)
70      {
71          $this->username = $username;
72      }
73
74      /**
75       * @return string
76       */
77      public function getEmail()
78      {
79          return $this->email;
80      }
81
82      /**
83       * @param string $email
84       */
85      public function setEmail($email)
86      {
87          $this->email = $email;
88      }
89  }
```

UserMapper.php

```
1   <?php
2
3   namespace DesignPatterns\Structural\DataMapper;
4
5   /**
6    * class UserMapper
7    */
8   class UserMapper
9   {
10      /**
11       * @var DBAL
12       */
13      protected $adapter;
14
15      /**
16       * @param DBAL $dbLayer
17       */
18      public function __construct(DBAL $dbLayer)
19      {
20          $this->adapter = $dbLayer;
21      }
22
23      /**
24       * saves a user object from memory to Database
25       *
26       * @param User $user
27       *
28       * @return boolean
29       */
30      public function save(User $user)
31      {
```

```php
        /* $data keys should correspond to valid Table columns on the Database */
        $data = array(
            'userid'   => $user->getUserId(),
            'username' => $user->getUsername(),
            'email'    => $user->getEmail(),
        );

        /* if no ID specified create new user else update the one in the Database */
        if (null === ($id = $user->getUserId())) {
            unset($data['userid']);
            $this->adapter->insert($data);

            return true;
        } else {
            $this->adapter->update($data, array('userid = ?' => $id));

            return true;
        }
    }

    /**
     * finds a user from Database based on ID and returns a User object located
     * in memory
     *
     * @param int $id
     *
     * @throws \InvalidArgumentException
     * @return User
     */
    public function findById($id)
    {
        $result = $this->adapter->find($id);

        if (0 == count($result)) {
            throw new \InvalidArgumentException("User #$id not found");
        }
        $row = $result->current();

        return $this->mapObject($row);
    }

    /**
     * fetches an array from Database and returns an array of User objects
     * located in memory
     *
     * @return array
     */
    public function findAll()
    {
        $resultSet = $this->adapter->findAll();
        $entries   = array();

        foreach ($resultSet as $row) {
            $entries[] = $this->mapObject($row);
        }

        return $entries;
    }
```

```
90
91      /**
92       * Maps a table row to an object
93       *
94       * @param array $row
95       *
96       * @return User
97       */
98      protected function mapObject(array $row)
99      {
100         $entry = new User();
101         $entry->setUserID($row['userid']);
102         $entry->setUsername($row['username']);
103         $entry->setEmail($row['email']);
104
105         return $entry;
106     }
107 }
```

Test

Tests/DataMapperTest.php

```
1  <?php
2
3  namespace DesignPatterns\Structural\DataMapper\Tests;
4
5  use DesignPatterns\Structural\DataMapper\UserMapper;
6  use DesignPatterns\Structural\DataMapper\User;
7
8  /**
9   * UserMapperTest tests the datamapper pattern
10  */
11 class DataMapperTest extends \PHPUnit_Framework_TestCase
12 {
13     /**
14      * @var UserMapper
15      */
16     protected $mapper;
17
18     /**
19      * @var DBAL
20      */
21     protected $dbal;
22
23     protected function setUp()
24     {
25         $this->dbal = $this->getMockBuilder('DesignPatterns\Structural\DataMapper\DBAL')
26                 ->disableAutoload()
27                 ->setMethods(array('insert', 'update', 'find', 'findAll'))
28                 ->getMock();
29
30         $this->mapper = new UserMapper($this->dbal);
31     }
32
33     public function getNewUser()
34     {
```

```
35          return array(array(new User(null, 'Odysseus', 'Odysseus@ithaca.gr')));
36      }
37
38      public function getExistingUser()
39      {
40          return array(array(new User(1, 'Odysseus', 'Odysseus@ithaca.gr')));
41      }
42
43      /**
44       * @dataProvider getNewUser
45       */
46      public function testPersistNew(User $user)
47      {
48          $this->dbal->expects($this->once())
49                  ->method('insert');
50          $this->mapper->save($user);
51      }
52
53      /**
54       * @dataProvider getExistingUser
55       */
56      public function testPersistExisting(User $user)
57      {
58          $this->dbal->expects($this->once())
59                  ->method('update');
60          $this->mapper->save($user);
61      }
62
63      /**
64       * @dataProvider getExistingUser
65       */
66      public function testRestoreOne(User $existing)
67      {
68          $row = array(
69              'userid'   => 1,
70              'username' => 'Odysseus',
71              'email'    => 'Odysseus@ithaca.gr'
72          );
73          $rows = new \ArrayIterator(array($row));
74          $this->dbal->expects($this->once())
75                  ->method('find')
76                  ->with(1)
77                  ->will($this->returnValue($rows));
78
79          $user = $this->mapper->findById(1);
80          $this->assertEquals($existing, $user);
81      }
82
83      /**
84       * @dataProvider getExistingUser
85       */
86      public function testRestoreMulti(User $existing)
87      {
88          $rows = array(array('userid' => 1, 'username' => 'Odysseus', 'email' => 'Odysseus@ithaca.gr'
89          $this->dbal->expects($this->once())
90                  ->method('findAll')
91                  ->will($this->returnValue($rows));
92
```

```
93          $user = $this->mapper->findAll();
94          $this->assertEquals(array($existing), $user);
95      }
96
97      /**
98       * @expectedException \InvalidArgumentException
99       * @expectedExceptionMessage User #404 not found
100      */
101     public function testNotFound()
102     {
103         $this->dbal->expects($this->once())
104                 ->method('find')
105                 ->with(404)
106                 ->will($this->returnValue(array()));
107
108         $user = $this->mapper->findById(404);
109     }
110 }
```

1.2.5 Decorator

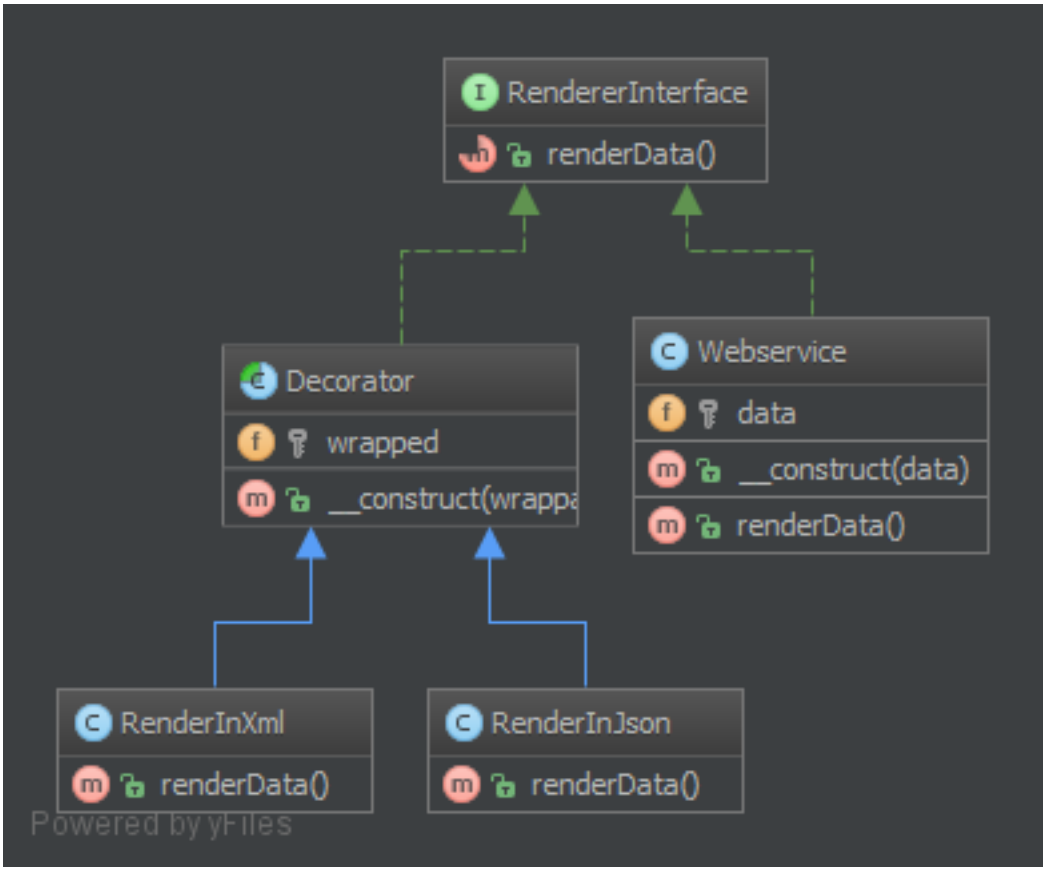
Purpose

To dynamically add new functionality to class instances.

Examples

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

UML Diagram



Code

You can also find these code on GitHub

RendererInterface.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * Class RendererInterface
 */
interface RendererInterface
{
    /**
     * render data
     *
     * @return mixed
     */
    public function renderData();
}
```

Webservice.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * Class Webservice
 */
class Webservice implements RendererInterface
{
    /**
     * @var mixed
     */
    protected $data;

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

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

Decorator.php

```php
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * the Decorator MUST implement the RendererInterface contract, this is the key-feature
 * of this design pattern. If not, this is no longer a Decorator but just a dumb
 * wrapper.
 */

/**
 * class Decorator
 */
abstract class Decorator implements RendererInterface
{
    /**
     * @var RendererInterface
     */
    protected $wrapped;

    /**
     * You must type-hint the wrapped component :
     * It ensures you can call renderData() in the subclasses !
     *
     * @param RendererInterface $wrappable
     */
```

```
    public function __construct(RendererInterface $wrappable)
    {
        $this->wrapped = $wrappable;
    }
}
```

RenderInXml.php

```
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * Class RenderInXml
 */
class RenderInXml extends Decorator
{
    /**
     * render data as XML
     *
     * @return mixed|string
     */
    public function renderData()
    {
        $output = $this->wrapped->renderData();

        // do some fancy conversion to xml from array ...

        $doc = new \DOMDocument();

        foreach ($output as $key => $val) {
            $doc->appendChild($doc->createElement($key, $val));
        }

        return $doc->saveXML();
    }
}
```

RenderInJson.php

```
<?php

namespace DesignPatterns\Structural\Decorator;

/**
 * Class RenderInJson
 */
class RenderInJson extends Decorator
{
    /**
     * render data as JSON
     *
     * @return mixed|string
     */
    public function renderData()
    {
        $output = $this->wrapped->renderData();

        return json_encode($output);
```

```
    }
}
```

Test

Tests/DecoratorTest.php

```php
<?php

namespace DesignPatterns\Structural\Decorator\Tests;

use DesignPatterns\Structural\Decorator;

/**
 * DecoratorTest tests the decorator pattern
 */
class DecoratorTest extends \PHPUnit_Framework_TestCase
{

    protected $service;

    protected function setUp()
    {
        $this->service = new Decorator\Webservice(array('foo' => 'bar'));
    }

    public function testJsonDecorator()
    {
        // Wrap service with a JSON decorator for renderers
        $service = new Decorator\RenderInJson($this->service);
        // Our Renderer will now output JSON instead of an array
        $this->assertEquals('{"foo":"bar"}', $service->renderData());
    }

    public function testXmlDecorator()
    {
        // Wrap service with a JSON decorator for renderers
        $service = new Decorator\RenderInXml($this->service);
        // Our Renderer will now output XML instead of an array
        $xml = '<?xml version="1.0"?><foo>bar</foo>';
        $this->assertXmlStringEqualsXmlString($xml, $service->renderData());
    }

    /**
     * The first key-point of this pattern :
     */
    public function testDecoratorMustImplementsRenderer()
    {
        $className = 'DesignPatterns\Structural\Decorator\Decorator';
        $interfaceName = 'DesignPatterns\Structural\Decorator\RendererInterface';
        $this->assertTrue(is_subclass_of($className, $interfaceName));
    }

    /**
     * Second key-point of this pattern : the decorator is type-hinted
     *
     * @expectedException \PHPUnit_Framework_Error
```

```
51          */
52      public function testDecoratorTypeHinted()
53      {
54          if (version_compare(PHP_VERSION, '7', '>=')) {
55              throw new \PHPUnit_Framework_Error('Skip test for PHP 7', 0, __FILE__, __LINE__);
56          }
57
58          $this->getMockForAbstractClass('DesignPatterns\Structural\Decorator\Decorator', array(new \st
59      }
60
61      /**
62       * Second key-point of this pattern : the decorator is type-hinted
63       *
64       * @requires PHP 7
65       * @expectedException TypeError
66       */
67      public function testDecoratorTypeHintedForPhp7()
68      {
69          $this->getMockForAbstractClass('DesignPatterns\Structural\Decorator\Decorator', array(new \st
70      }
71
72      /**
73       * The decorator implements and wraps the same interface
74       */
75      public function testDecoratorOnlyAcceptRenderer()
76      {
77          $mock = $this->getMock('DesignPatterns\Structural\Decorator\RendererInterface');
78          $dec = $this->getMockForAbstractClass('DesignPatterns\Structural\Decorator\Decorator', array
79          $this->assertNotNull($dec);
80      }
81  }
```

1.2.6 Dependency Injection

Purpose

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

Usage

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

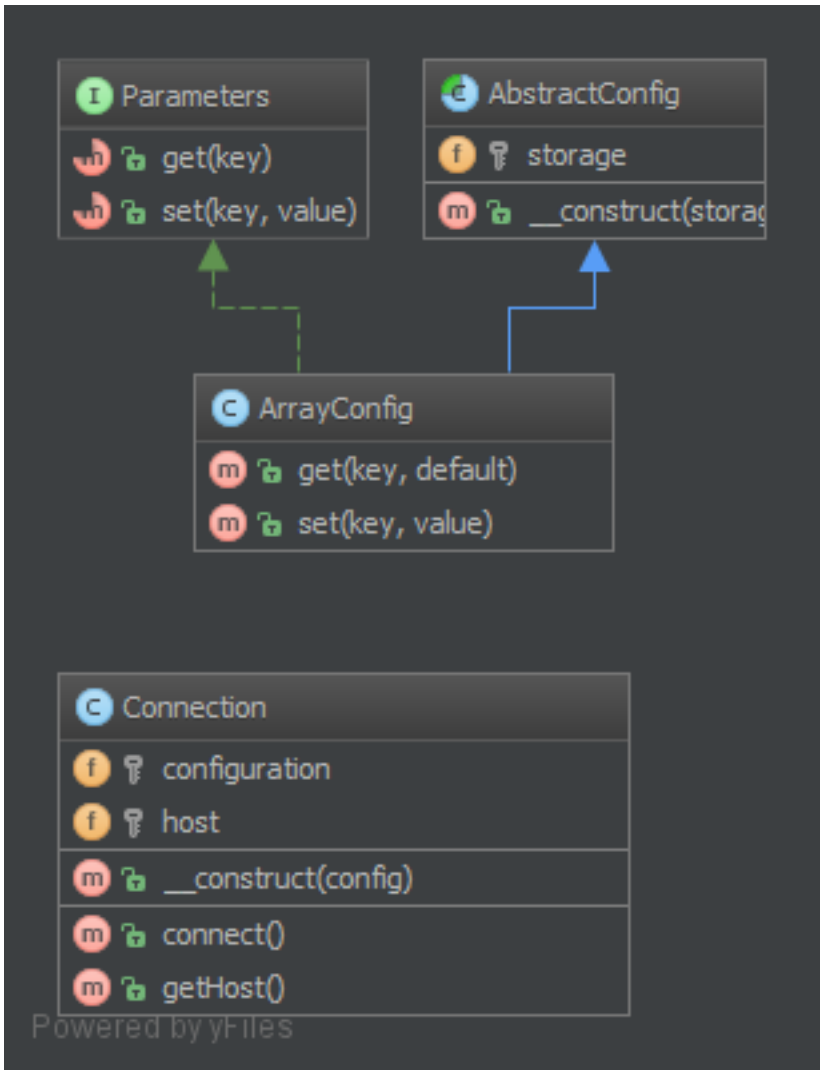
Notice we are following Inversion of control principle in `Connection` by asking `$config` to implement `Parameters` interface. This decouples our components. We don't care where the source of information comes from, we only care that `$config` has certain methods to retrieve that information. Read more about Inversion of control here.

Examples

- The Doctrine2 ORM uses dependency injection e.g. for configuration that is injected into a `Connection` object. For testing purposes, one can easily create a mock object of the configuration and inject that into the `Connection` object

- Symfony and Zend Framework 2 already have containers for DI that create objects via a configuration array and inject them where needed (i.e. in Controllers)

UML Diagram



Code

You can also find these code on GitHub

AbstractConfig.php

```
1  <?php
2
3  namespace DesignPatterns\Structural\DependencyInjection;
4
5  /**
6   * class AbstractConfig
7   */
8  abstract class AbstractConfig
```

```
{
    /**
     * @var Storage of data
     */
    protected $storage;

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

Parameters.php

```
<?php

namespace DesignPatterns\Structural\DependencyInjection;

/**
 * Parameters interface
 */
interface Parameters
{
    /**
     * Get parameter
     *
     * @param string|int $key
     *
     * @return mixed
     */
    public function get($key);

    /**
     * Set parameter
     *
     * @param string|int $key
     * @param mixed      $value
     */
    public function set($key, $value);
}
```

ArrayConfig.php

```
<?php

namespace DesignPatterns\Structural\DependencyInjection;

/**
 * class ArrayConfig
 *
 * uses array as data source
 */
class ArrayConfig extends AbstractConfig implements Parameters
{
    /**
     * Get parameter
     *
     * @param string|int $key
     * @param null $default
```

```
     * @return mixed
     */
    public function get($key, $default = null)
    {
        if (isset($this->storage[$key])) {
            return $this->storage[$key];
        }
        return $default;
    }

    /**
     * Set parameter
     *
     * @param string|int $key
     * @param mixed $value
     */
    public function set($key, $value)
    {
        $this->storage[$key] = $value;
    }
}
```

Connection.php

```
<?php

namespace DesignPatterns\Structural\DependencyInjection;

/**
 * Class Connection
 */
class Connection
{
    /**
     * @var Configuration
     */
    protected $configuration;

    /**
     * @var Currently connected host
     */
    protected $host;

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

    /**
     * connection using the injected config
     */
    public function connect()
    {
        $host = $this->configuration->get('host');
        // connection to host, authentication etc...

```

```
        //if connected
        $this->host = $host;
    }

    /*
     * Get currently connected host
     *
     * @return string
     */
    public function getHost()
    {
        return $this->host;
    }
}
```

Test

Tests/DependencyInjectionTest.php

```
<?php

namespace DesignPatterns\Structural\DependencyInjection\Tests;

use DesignPatterns\Structural\DependencyInjection\ArrayConfig;
use DesignPatterns\Structural\DependencyInjection\Connection;

class DependencyInjectionTest extends \PHPUnit_Framework_TestCase
{
    protected $config;
    protected $source;

    public function setUp()
    {
        $this->source = include 'config.php';
        $this->config = new ArrayConfig($this->source);
    }

    public function testDependencyInjection()
    {
        $connection = new Connection($this->config);
        $connection->connect();
        $this->assertEquals($this->source['host'], $connection->getHost());
    }
}
```

Tests/config.php

```
<?php

return array('host' => 'github.com');
```

1.2.7 Facade

Purpose

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

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

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

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

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

UML Diagram

Code

You can also find these code on GitHub

Facade.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

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

    /**
     * @var BiosInterface
     */
    protected $bios;

    /**
     * This is the perfect time to use a dependency injection container
     * to create an instance of this class
     *
     * @param BiosInterface $bios
     * @param OsInterface   $os
     */
    public function __construct(BiosInterface $bios, OsInterface $os)
    {
        $this->bios = $bios;
        $this->os = $os;
    }

    /**
     * turn on the system
     */
    public function turnOn()
    {
        $this->bios->execute();
        $this->bios->waitForKeyPress();
        $this->bios->launch($this->os);
    }

    /**
     * turn off the system
     */
    public function turnOff()
    {
        $this->os->halt();
        $this->bios->powerDown();
    }
}
```

OsInterface.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

```

```php
/**
 * Class OsInterface
 */
interface OsInterface
{
    /**
     * halt the OS
     */
    public function halt();
}
```

BiosInterface.php

```php
<?php

namespace DesignPatterns\Structural\Facade;

/**
 * Class BiosInterface
 */
interface BiosInterface
{
    /**
     * execute the BIOS
     */
    public function execute();

    /**
     * wait for halt
     */
    public function waitForKeyPress();

    /**
     * launches the OS
     *
     * @param OsInterface $os
     */
    public function launch(OsInterface $os);

    /**
     * power down BIOS
     */
    public function powerDown();
}
```

Test

Tests/FacadeTest.php

```php
<?php

namespace DesignPatterns\Structural\Facade\Tests;

use DesignPatterns\Structural\Facade\Facade as Computer;
use DesignPatterns\Structural\Facade\OsInterface;

/**
```

```php
 * FacadeTest shows example of facades
 */
class FacadeTest extends \PHPUnit_Framework_TestCase
{

    public function getComputer()
    {
        $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\BiosInterface')
                ->setMethods(array('launch', 'execute', 'waitForKeyPress'))
                ->disableAutoload()
                ->getMock();
        $operatingSys = $this->getMockBuilder('DesignPatterns\Structural\Facade\OsInterface')
                ->setMethods(array('getName'))
                ->disableAutoload()
                ->getMock();
        $bios->expects($this->once())
                ->method('launch')
                ->with($operatingSys);
        $operatingSys
                ->expects($this->once())
                ->method('getName')
                ->will($this->returnValue('Linux'));

        $facade = new Computer($bios, $operatingSys);
        return array(array($facade, $operatingSys));
    }

    /**
     * @dataProvider getComputer
     */
    public function testComputerOn(Computer $facade, OsInterface $os)
    {
        // interface is simpler :
        $facade->turnOn();
        // but I can access to lower component
        $this->assertEquals('Linux', $os->getName());
    }
}
```

1.2.8 Fluent Interface

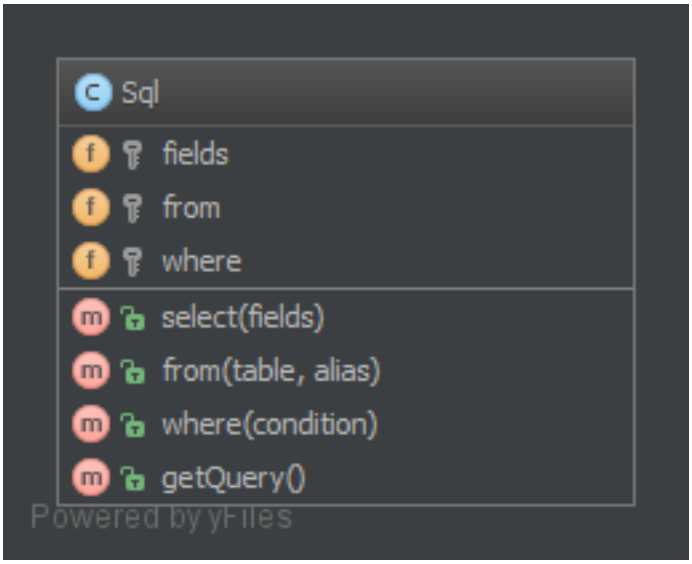
Purpose

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

Examples

- Doctrine2's QueryBuilder works something like that example class below
- PHPUnit uses fluent interfaces to build mock objects
- Yii Framework: CDbCommand and CActiveRecord use this pattern, too

UML Diagram



Code

You can also find these code on GitHub

Sql.php

```php
<?php

namespace DesignPatterns\Structural\FluentInterface;

/**
 * class SQL
 */
class Sql
{
    /**
     * @var array
     */
    protected $fields = array();

    /**
     * @var array
     */
    protected $from = array();

    /**
     * @var array
     */
    protected $where = array();

    /**
     * adds select fields
     *
     * @param array $fields
     *
     * @return SQL
```

```php
     */
    public function select(array $fields = array())
    {
        $this->fields = $fields;

        return $this;
    }

    /**
     * adds a FROM clause
     *
     * @param string $table
     * @param string $alias
     *
     * @return SQL
     */
    public function from($table, $alias)
    {
        $this->from[] = $table . ' AS ' . $alias;

        return $this;
    }

    /**
     * adds a WHERE condition
     *
     * @param string $condition
     *
     * @return SQL
     */
    public function where($condition)
    {
        $this->where[] = $condition;

        return $this;
    }

    /**
     * Gets the query, just an example of building a query,
     * no check on consistency
     *
     * @return string
     */
    public function getQuery()
    {
        return 'SELECT ' . implode(',', $this->fields)
                . ' FROM ' . implode(',', $this->from)
                . ' WHERE ' . implode(' AND ', $this->where);
    }
}
```

Test

Tests/FluentInterfaceTest.php

```php
<?php

```

```php
namespace DesignPatterns\Structural\FluentInterface\Tests;

use DesignPatterns\Structural\FluentInterface\Sql;

/**
 * FluentInterfaceTest tests the fluent interface SQL
 */
class FluentInterfaceTest extends \PHPUnit_Framework_TestCase
{

    public function testBuildSQL()
    {
        $instance = new Sql();
        $query = $instance->select(array('foo', 'bar'))
                ->from('foobar', 'f')
                ->where('f.bar = ?')
                ->getQuery();

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

1.2.9 Proxy

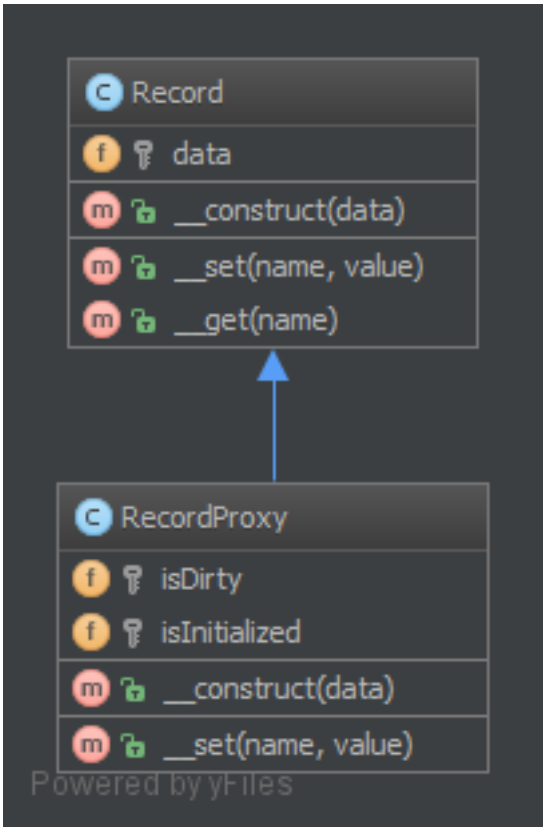
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

Record.php

```php
<?php

namespace DesignPatterns\Structural\Proxy;

/**
 * class Record
 */
class Record
{
    /**
     * @var array|null
     */
    protected $data;

    /**
     * @param null $data
     */
    public function __construct($data = null)
    {
        $this->data = (array) $data;
```

```
    }

    /**
     * magic setter
     *
     * @param string $name
     * @param mixed  $value
     *
     * @return void
     */
    public function __set($name, $value)
    {
        $this->data[(string) $name] = $value;
    }

    /**
     * magic getter
     *
     * @param string $name
     *
     * @return mixed|null
     */
    public function __get($name)
    {
        if (array_key_exists($name, $this->data)) {
            return $this->data[(string) $name];
        } else {
            return null;
        }
    }
}
```

RecordProxy.php

```
<?php

namespace DesignPatterns\Structural\Proxy;

/**
 * Class RecordProxy
 */
class RecordProxy extends Record
{
    /**
     * @var bool
     */
    protected $isDirty = false;

    /**
     * @var bool
     */
    protected $isInitialized = false;

    /**
     * @param array $data
     */
    public function __construct($data)
    {
        parent::__construct($data);
```

```
26
27          // when the record has data, mark it as initialized
28          // since Record will hold our business logic, we don't want to
29          // implement this behaviour there, but instead in a new proxy class
30          // that extends the Record class
31          if (null !== $data) {
32              $this->isInitialized = true;
33              $this->isDirty = true;
34          }
35      }
36
37      /**
38       * magic setter
39       *
40       * @param string $name
41       * @param mixed  $value
42       *
43       * @return void
44       */
45      public function __set($name, $value)
46      {
47          $this->isDirty = true;
48          parent::__set($name, $value);
49      }
50  }
```

Test

1.2.10 Registry

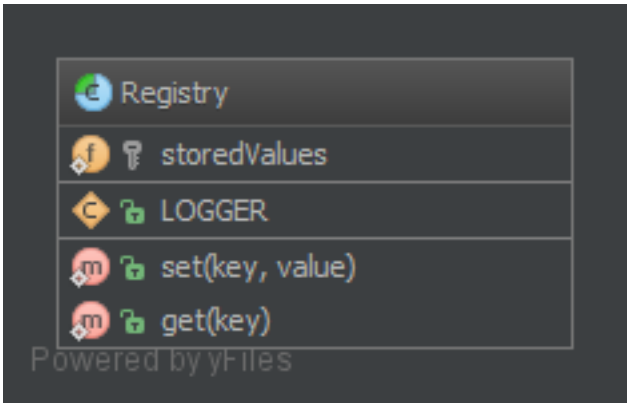
Purpose

To implement a central storage for objects often used throughout the application, is typically implemented using an abstract class with only static methods (or using the Singleton pattern)

Examples

- Zend Framework 1: `Zend_Registry` holds the application's logger object, front controller etc.
- Yii Framework: `CWebApplication` holds all the application components, such as `CWebUser`, `CUrlManager`, etc.

UML Diagram



Code

You can also find these code on GitHub

Registry.php

```php
<?php

namespace DesignPatterns\Structural\Registry;

/**
 * class Registry
 */
abstract class Registry
{
    const LOGGER = 'logger';

    /**
     * @var array
     */
    protected static $storedValues = array();

    /**
     * sets a value
     *
     * @param string $key
     * @param mixed  $value
     *
     * @static
     * @return void
     */
    public static function set($key, $value)
    {
        self::$storedValues[$key] = $value;
    }

    /**
     * gets a value from the registry
     *
     * @param string $key
     *
```

```
     * @static
     * @return mixed
     */
    public static function get($key)
    {
        return self::$storedValues[$key];
    }

    // typically there would be methods to check if a key has already been registered and so on ...
}
```

Test

Tests/RegistryTest.php

```
<?php

namespace DesignPatterns\Structural\Registry\Tests;

use DesignPatterns\Structural\Registry\Registry;

class RegistryTest extends \PHPUnit_Framework_TestCase
{

    public function testSetAndGetLogger()
    {
        Registry::set(Registry::LOGGER, new \StdClass());

        $logger = Registry::get(Registry::LOGGER);
        $this->assertInstanceOf('StdClass', $logger);
    }
}
```

1.3 Behavioral

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

1.3.1 Chain Of Responsibilities

Purpose:

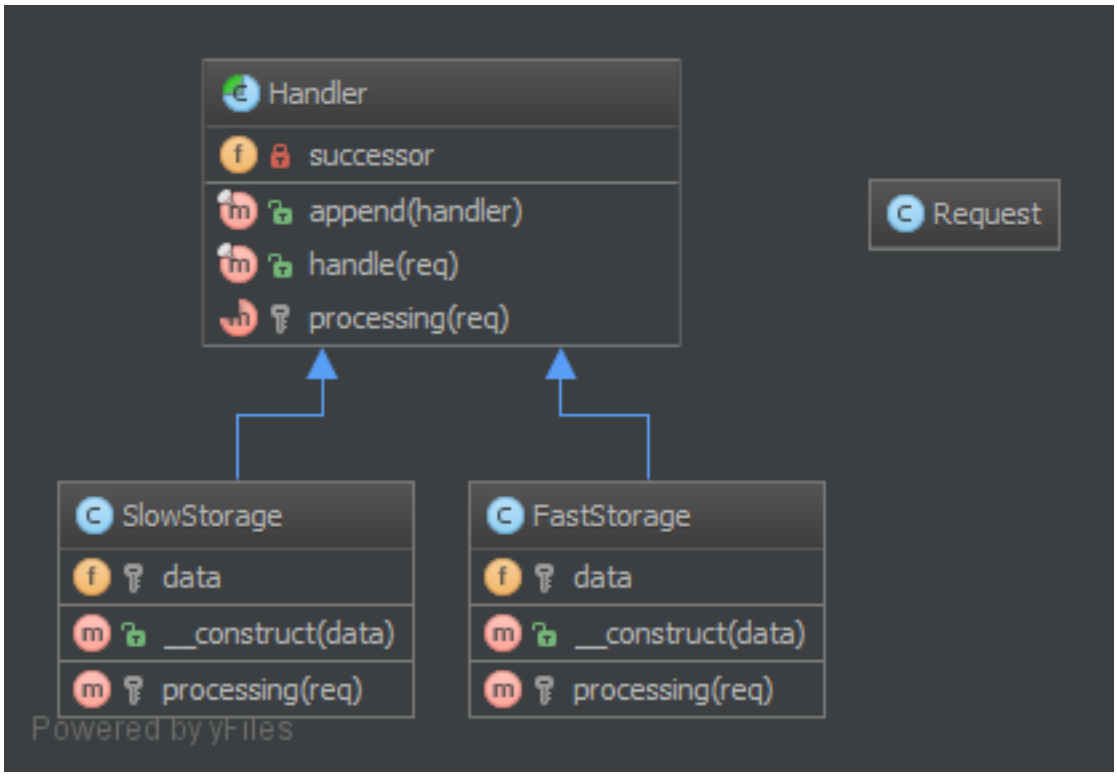
To build a chain of objects to handle a call in sequential order. If one object cannot handle a call, it delegates the call to the next in the chain and so forth.

Examples:

- logging framework, where each chain element decides autonomously what to do with a log message
- a Spam filter

- Caching: first object is an instance of e.g. a Memcached Interface, if that "misses" it delegates the call to the database interface
- Yii Framework: CFilterChain is a chain of controller action filters. the executing point is passed from one filter to the next along the chain, and only if all filters say "yes", the action can be invoked at last.

UML Diagram



Code

You can also find these code on GitHub

Request.php

```php
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

/**
 * Request is a request which goes through the chain of responsibilities.
 *
 * About the request: Sometimes, you don't need an object, just an integer or
 * an array. But in this case of a full example, I've made a class to illustrate
 * this important idea in the CoR (Chain of Responsibilities). In the real world,
 * I recommend to always use a class, even a \stdClass if you want, it proves
 * to be more adaptive because a single handler doesn't know much about the
 * outside world and it is more difficult if, one day, you want to add some
 * criterion in a decision process.
 */
class Request
```

```
{
    // getter and setter but I don't want to generate too much noise in handlers
}
```

Handler.php

```php
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities;

/**
 * Handler is a generic handler in the chain of responsibilities
 *
 * Yes you could have a lighter CoR with a simpler handler but if you want your CoR
 * to be extendable and decoupled, it's a better idea to do things like that in real
 * situations. Usually, a CoR is meant to be changed everytime and evolves, that's
 * why we slice the workflow in little bits of code.
 */
abstract class Handler
{
    /**
     * @var Handler
     */
    private $successor = null;

    /**
     * Append a responsibility to the end of chain
     *
     * A prepend method could be done with the same spirit
     *
     * You could also send the successor in the constructor but in PHP that is a
     * bad idea because you have to remove the type-hint of the parameter because
     * the last handler has a null successor.
     *
     * And if you override the constructor, that Handler can no longer have a
     * successor. One solution is to provide a NullObject (see pattern).
     * It is more preferable to keep the constructor "free" to inject services
     * you need with the DiC of symfony2 for example.
     *
     * @param Handler $handler
     */
    final public function append(Handler $handler)
    {
        if (is_null($this->successor)) {
            $this->successor = $handler;
        } else {
            $this->successor->append($handler);
        }
    }

    /**
     * Handle the request.
     *
     * This approach by using a template method pattern ensures you that
     * each subclass will not forget to call the successor. Besides, the returned
     * boolean value indicates you if the request have been processed or not.
     *
     * @param Request $req
     *
```

```
     * @return bool
     */
    final public function handle(Request $req)
    {
        $req->forDebugOnly = get_called_class();
        $processed = $this->processing($req);
        if (!$processed) {
            // the request has not been processed by this handler => see the next
            if (!is_null($this->successor)) {
                $processed = $this->successor->handle($req);
            }
        }

        return $processed;
    }

    /**
     * Each concrete handler has to implement the processing of the request
     *
     * @param Request $req
     *
     * @return bool true if the request has been processed
     */
    abstract protected function processing(Request $req);
}
```

Responsible/SlowStorage.php

```
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Request;

/**
 * This is mostly the same code as FastStorage but in fact, it may greatly differs
 *
 * One important fact about CoR: each item in the chain MUST NOT assume its position
 * in the chain. A CoR is not responsible if the request is not handled UNLESS
 * you make an "ExceptionHandler" which throws execption if the request goes there.
 *
 * To be really extendable, each handler doesn't know if there is something after it.
 *
 */
class SlowStorage extends Handler
{
    /**
     * @var array
     */
    protected $data = array();

    /**
     * @param array $data
     */
    public function __construct($data = array())
    {
        $this->data = $data;
    }
```

```
32
33      protected function processing(Request $req)
34      {
35          if ('get' === $req->verb) {
36              if (array_key_exists($req->key, $this->data)) {
37                  $req->response = $this->data[$req->key];
38
39                  return true;
40              }
41          }
42
43          return false;
44      }
45  }
```

Responsible/FastStorage.php

```
1   <?php
2
3   namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
4
5   use DesignPatterns\Behavioral\ChainOfResponsibilities\Handler;
6   use DesignPatterns\Behavioral\ChainOfResponsibilities\Request;
7
8   /**
9    * Class FastStorage
10   */
11  class FastStorage extends Handler
12  {
13      /**
14       * @var array
15       */
16      protected $data = array();
17
18      /**
19       * @param array $data
20       */
21      public function __construct($data = array())
22      {
23          $this->data = $data;
24      }
25
26      protected function processing(Request $req)
27      {
28          if ('get' === $req->verb) {
29              if (array_key_exists($req->key, $this->data)) {
30                  // the handler IS responsible and then processes the request
31                  $req->response = $this->data[$req->key];
32                  // instead of returning true, I could return the value but it proves
33                  // to be a bad idea. What if the value IS "false" ?
34                  return true;
35              }
36          }
37
38          return false;
39      }
40  }
```

Test

Tests/ChainTest.php

```php
<?php

namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;

use DesignPatterns\Behavioral\ChainOfResponsibilities\Request;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\FastStorage;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowStorage;
use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;

/**
 * ChainTest tests the CoR
 */
class ChainTest extends \PHPUnit_Framework_TestCase
{

    /**
     * @var FastStorage
     */
    protected $chain;

    protected function setUp()
    {
        $this->chain = new FastStorage(array('bar' => 'baz'));
        $this->chain->append(new SlowStorage(array('bar' => 'baz', 'foo' => 'bar')));
    }

    public function makeRequest()
    {
        $request = new Request();
        $request->verb = 'get';

        return array(
            array($request)
        );
    }

    /**
     * @dataProvider makeRequest
     */
    public function testFastStorage($request)
    {
        $request->key = 'bar';
        $ret = $this->chain->handle($request);

        $this->assertTrue($ret);
        $this->assertObjectHasAttribute('response', $request);
        $this->assertEquals('baz', $request->response);
        // despite both handle owns the 'bar' key, the FastStorage is responding first
        $className = 'DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\FastStorage';
        $this->assertEquals($className, $request->forDebugOnly);
    }

    /**
     * @dataProvider makeRequest
     */
```

```
56      public function testSlowStorage($request)
57      {
58          $request->key = 'foo';
59          $ret = $this->chain->handle($request);
60
61          $this->assertTrue($ret);
62          $this->assertObjectHasAttribute('response', $request);
63          $this->assertEquals('bar', $request->response);
64          // FastStorage has no 'foo' key, the SlowStorage is responding
65          $className = 'DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowStorage';
66          $this->assertEquals($className, $request->forDebugOnly);
67      }
68
69      /**
70       * @dataProvider makeRequest
71       */
72      public function testFailure($request)
73      {
74          $request->key = 'kurukuku';
75          $ret = $this->chain->handle($request);
76
77          $this->assertFalse($ret);
78          // the last responsible :
79          $className = 'DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowStorage';
80          $this->assertEquals($className, $request->forDebugOnly);
81      }
82  }
```

1.3.2 Command

Purpose

To encapsulate invocation and decoupling.

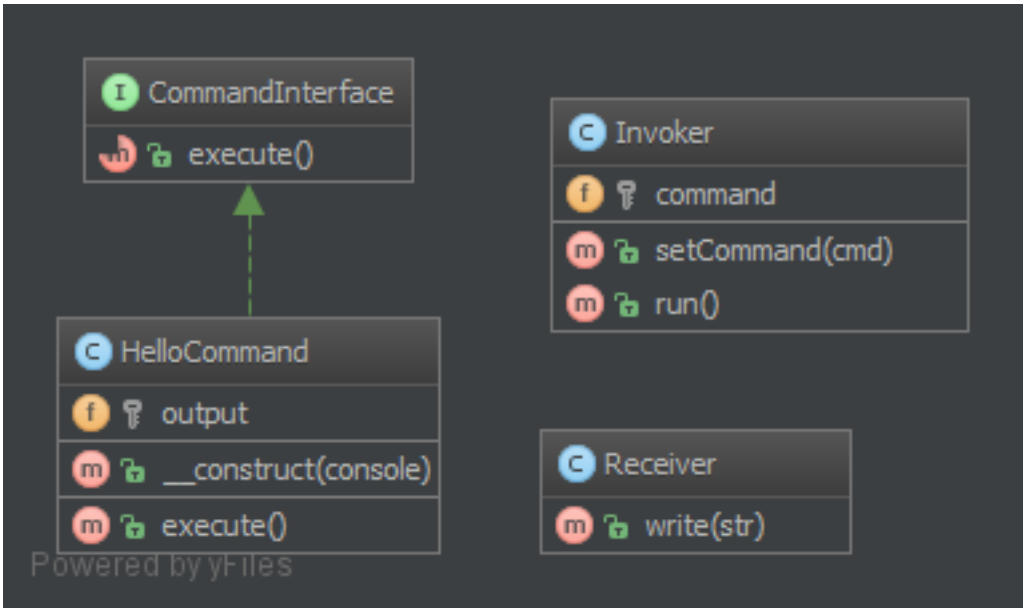
We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

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

Examples

- A text editor : all events are Command which can be undone, stacked and saved.
- Symfony2: SF2 Commands that can be run from the CLI are built with just the Command pattern in mind
- big CLI tools use subcommands to distribute various tasks and pack them in "modules", each of these can be implemented with the Command pattern (e.g. vagrant)

UML Diagram



Code

You can also find these code on GitHub

CommandInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * class CommandInterface
 */
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 know he can call "execute"
 */
class HelloCommand implements CommandInterface
{
```

```php
    /**
     * @var Receiver
     */
    protected $output;

    /**
     * Each concrete command is builded with different receivers.
     * Can be one, many, none or even other Command in 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
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * Receiver is specific service with its own contract and can be only concrete
 */
class Receiver
{
    /**
     * @param string $str
     */
    public function write($str)
    {
        echo $str;
    }
}
```

Invoker.php

```php
<?php

namespace DesignPatterns\Behavioral\Command;

/**
 * Invoker is using the command given to it.
 * Example : an Application in SF2
 */
class Invoker
{
```

```php
    /**
     * @var CommandInterface
     */
    protected $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
     */
    public function run()
    {
        // here is a key feature of the invoker
        // the invoker is the same whatever is the command
        $this->command->execute();
    }
}
```

Test

Tests/CommandTest.php

```php
<?php

namespace DesignPatterns\Behavioral\Command\Tests;

use DesignPatterns\Behavioral\Command\Invoker;
use DesignPatterns\Behavioral\Command\Receiver;
use DesignPatterns\Behavioral\Command\HelloCommand;

/**
 * CommandTest has the role of the Client in the Command Pattern
 */
class CommandTest extends \PHPUnit_Framework_TestCase
{

    /**
     * @var Invoker
     */
    protected $invoker;

    /**
     * @var Receiver
     */
    protected $receiver;

    protected function setUp()
    {
```

```
27          $this->invoker = new Invoker();
28          $this->receiver = new Receiver();
29      }
30
31      public function testInvocation()
32      {
33          $this->invoker->setCommand(new HelloCommand($this->receiver));
34          $this->expectOutputString('Hello World');
35          $this->invoker->run();
36      }
37  }
```

1.3.3 Iterator

Purpose

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

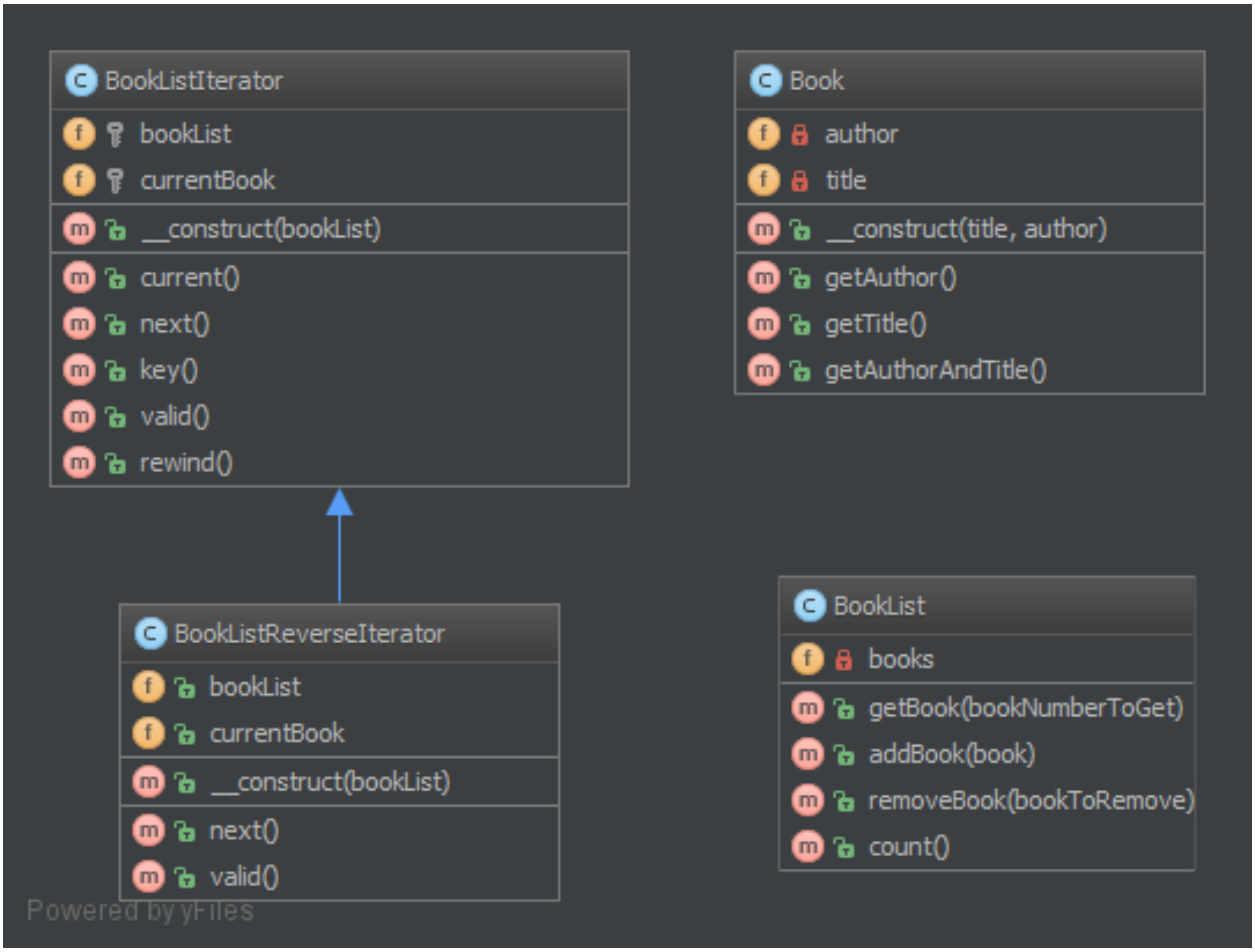
Examples

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

Note

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

UML Diagram



Code

You can also find these code on GitHub

Book.php

```php
<?php

namespace DesignPatterns\Behavioral\Iterator;

class Book
{

    private $author;

    private $title;

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

```
17
18      public function getAuthor()
19      {
20          return $this->author;
21      }
22
23      public function getTitle()
24      {
25          return $this->title;
26      }
27
28      public function getAuthorAndTitle()
29      {
30          return $this->getTitle() . ' by ' . $this->getAuthor();
31      }
32  }
```

BookList.php

```
1   <?php
2
3   namespace DesignPatterns\Behavioral\Iterator;
4
5   class BookList implements \Countable
6   {
7
8       private $books;
9
10      public function getBook($bookNumberToGet)
11      {
12          if (isset($this->books[$bookNumberToGet])) {
13              return $this->books[$bookNumberToGet];
14          }
15
16          return null;
17      }
18
19      public function addBook(Book $book)
20      {
21          $this->books[] = $book;
22
23          return $this->count();
24      }
25
26      public function removeBook(Book $bookToRemove)
27      {
28          foreach ($this->books as $key => $book) {
29              /** @var Book $book */
30              if ($book->getAuthorAndTitle() === $bookToRemove->getAuthorAndTitle()) {
31                  unset($this->books[$key]);
32              }
33          }
34
35          return $this->count();
36      }
37
38      public function count()
39      {
40          return count($this->books);
```

```
    }
}
```

BookListIterator.php

```php
<?php

namespace DesignPatterns\Behavioral\Iterator;

class BookListIterator implements \Iterator
{

    /**
     * @var BookList
     */
    private $bookList;

    /**
     * @var int
     */
    protected $currentBook = 0;

    public function __construct(BookList $bookList)
    {
        $this->bookList = $bookList;
    }

    /**
     * Return the current book
     * @link http://php.net/manual/en/iterator.current.php
     * @return Book Can return any type.
     */
    public function current()
    {
        return $this->bookList->getBook($this->currentBook);
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Move forward to next element
     * @link http://php.net/manual/en/iterator.next.php
     * @return void Any returned value is ignored.
     */
    public function next()
    {
        $this->currentBook++;
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Return the key of the current element
     * @link http://php.net/manual/en/iterator.key.php
     * @return mixed scalar on success, or null on failure.
     */
    public function key()
    {
        return $this->currentBook;
    }

```

```
    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Checks if current position is valid
     * @link http://php.net/manual/en/iterator.valid.php
     * @return boolean The return value will be casted to boolean and then evaluated.
     *       Returns true on success or false on failure.
     */
    public function valid()
    {
        return null !== $this->bookList->getBook($this->currentBook);
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Rewind the Iterator to the first element
     * @link http://php.net/manual/en/iterator.rewind.php
     * @return void Any returned value is ignored.
     */
    public function rewind()
    {
        $this->currentBook = 0;
    }
}
```

BookListReverseIterator.php

```
<?php

namespace DesignPatterns\Behavioral\Iterator;

class BookListReverseIterator implements \Iterator
{

    /**
     * @var BookList
     */
    private $bookList;

    /**
     * @var int
     */
    protected $currentBook = 0;

    public function __construct(BookList $bookList)
    {
        $this->bookList = $bookList;
        $this->currentBook = $this->bookList->count() - 1;
    }

    /**
     * Return the current book
     * @link http://php.net/manual/en/iterator.current.php
     * @return Book Can return any type.
     */
    public function current()
    {
        return $this->bookList->getBook($this->currentBook);
    }

```

```php
    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Move forward to next element
     * @link http://php.net/manual/en/iterator.next.php
     * @return void Any returned value is ignored.
     */
    public function next()
    {
        $this->currentBook--;
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Return the key of the current element
     * @link http://php.net/manual/en/iterator.key.php
     * @return mixed scalar on success, or null on failure.
     */
    public function key()
    {
        return $this->currentBook;
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Checks if current position is valid
     * @link http://php.net/manual/en/iterator.valid.php
     * @return boolean The return value will be casted to boolean and then evaluated.
     *       Returns true on success or false on failure.
     */
    public function valid()
    {
        return null !== $this->bookList->getBook($this->currentBook);
    }

    /**
     * (PHP 5 &gt;= 5.0.0)<br/>
     * Rewind the Iterator to the first element
     * @link http://php.net/manual/en/iterator.rewind.php
     * @return void Any returned value is ignored.
     */
    public function rewind()
    {
        $this->currentBook = $this->bookList->count() - 1;
    }
}
```

Test

Tests/IteratorTest.php

```php
<?php

namespace DesignPatterns\Behavioral\Iterator\Tests;

use DesignPatterns\Behavioral\Iterator\Book;
use DesignPatterns\Behavioral\Iterator\BookList;
use DesignPatterns\Behavioral\Iterator\BookListIterator;
```

```php
use DesignPatterns\Behavioral\Iterator\BookListReverseIterator;

class IteratorTest extends \PHPUnit_Framework_TestCase
{

    /**
     * @var BookList
     */
    protected $bookList;

    protected function setUp()
    {
        $this->bookList = new BookList();
        $this->bookList->addBook(new Book('Learning PHP Design Patterns', 'William Sanders'));
        $this->bookList->addBook(new Book('Professional Php Design Patterns', 'Aaron Saray'));
        $this->bookList->addBook(new Book('Clean Code', 'Robert C. Martin'));
    }

    public function expectedAuthors()
    {
        return array(
            array(
                array(
                    'Learning PHP Design Patterns by William Sanders',
                    'Professional Php Design Patterns by Aaron Saray',
                    'Clean Code by Robert C. Martin'
                )
            ),
        );
    }

    /**
     * @dataProvider expectedAuthors
     */
    public function testUseAIteratorAndValidateAuthors($expected)
    {
        $iterator = new BookListIterator($this->bookList);

        while ($iterator->valid()) {
            $expectedBook = array_shift($expected);
            $this->assertEquals($expectedBook, $iterator->current()->getAuthorAndTitle());
            $iterator->next();
        }
    }

    /**
     * @dataProvider expectedAuthors
     */
    public function testUseAReverseIteratorAndValidateAuthors($expected)
    {
        $iterator = new BookListReverseIterator($this->bookList);

        while ($iterator->valid()) {
            $expectedBook = array_pop($expected);
            $this->assertEquals($expectedBook, $iterator->current()->getAuthorAndTitle());
            $iterator->next();
        }
    }
```

```
66
67      /**
68       * Test BookList Remove
69       */
70      public function testBookRemove()
71      {
72          $this->bookList->removeBook($this->bookList->getBook(0));
73          $this->assertEquals($this->bookList->count(), 2);
74      }
75  }
```

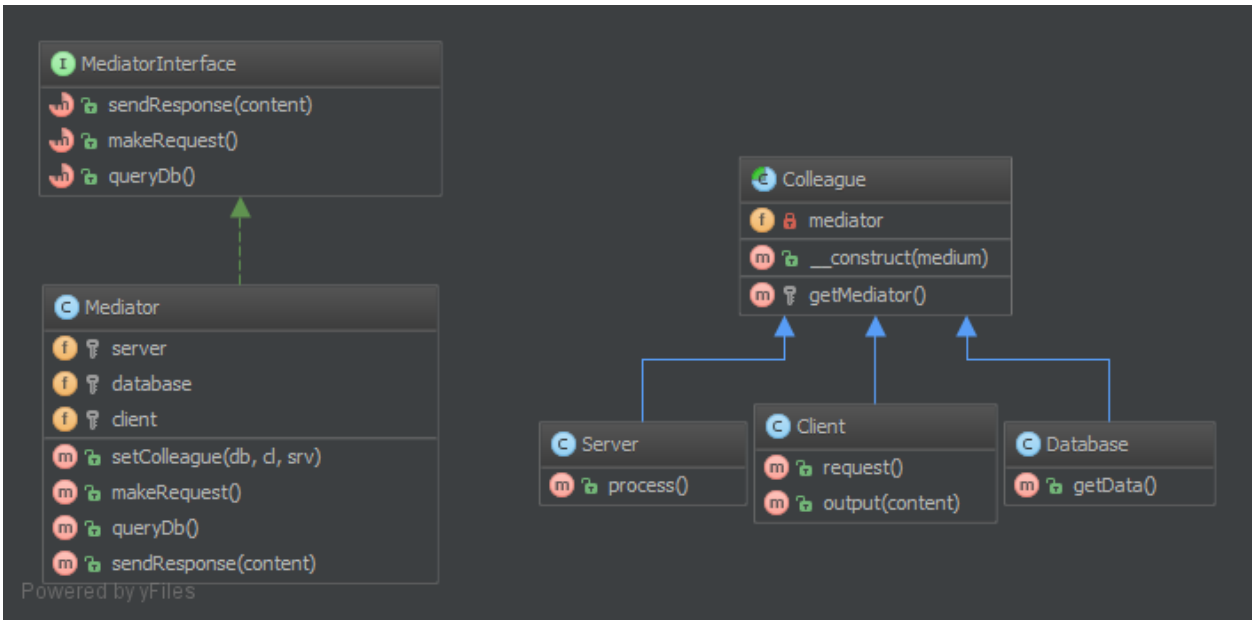
1.3.4 Mediator

Purpose

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

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

UML Diagram



Code

You can also find these code on GitHub

MediatorInterface.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator;
4
```

```php
/**
 * MediatorInterface is a contract for the Mediator
 * This interface is not mandatory but it is better for LSP concerns
 */
interface MediatorInterface
{
    /**
     * sends the response
     *
     * @param string $content
     */
    public function sendResponse($content);

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

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

Mediator.php

```php
<?php

namespace DesignPatterns\Behavioral\Mediator;

use DesignPatterns\Behavioral\Mediator\Subsystem;

/**
 * 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
     */
    protected $server;

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

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

    /**
     * @param Subsystem\Database $db
     * @param Subsystem\Client   $cl
     * @param Subsystem\Server   $srv
     */
```

```
    public function setColleague(Subsystem\Database $db, Subsystem\Client $cl, Subsystem\Server $srv)
    {
        $this->database = $db;
        $this->server = $srv;
        $this->client = $cl;
    }

    /**
     * make request
     */
    public function makeRequest()
    {
        $this->server->process();
    }

    /**
     * query db
     * @return mixed
     */
    public function queryDb()
    {
        return $this->database->getData();
    }

    /**
     * send response
     *
     * @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 colleague.
 */
abstract class Colleague
{
    /**
     * this ensures no change in subclasses
     *
     * @var MediatorInterface
     */
    private $mediator;

    /**
     * @param MediatorInterface $medium
     */
    public function __construct(MediatorInterface $medium)
```

```
    {
        // in this way, we are sure the concrete colleague knows the mediator
        $this->mediator = $medium;
    }

    // for subclasses
    protected function getMediator()
    {
        return $this->mediator;
    }
}
```

Subsystem/Client.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

/**
 * Client is a client that make request et get response
 */
class Client extends Colleague
{
    /**
     * request
     */
    public function request()
    {
        $this->getMediator()->makeRequest();
    }

    /**
     * output content
     *
     * @param string $content
     */
    public function output($content)
    {
        echo $content;
    }
}
```

Subsystem/Database.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

/**
 * Database is a database service
 */
class Database extends Colleague
{
    /**
     * @return string
```

```
     */
    public function getData()
    {
        return "World";
    }
}
```

Subsystem/Server.php

```
<?php

namespace DesignPatterns\Behavioral\Mediator\Subsystem;

use DesignPatterns\Behavioral\Mediator\Colleague;

/**
 * Server serves responses
 */
class Server extends Colleague
{
    /**
     * process on server
     */
    public function process()
    {
        $data = $this->getMediator()->queryDb();
        $this->getMediator()->sendResponse("Hello $data");
    }
}
```

Test

Tests/MediatorTest.php

```
<?php

namespace DesignPatterns\Tests\Mediator\Tests;

use DesignPatterns\Behavioral\Mediator\Mediator;
use DesignPatterns\Behavioral\Mediator\Subsystem\Database;
use DesignPatterns\Behavioral\Mediator\Subsystem\Client;
use DesignPatterns\Behavioral\Mediator\Subsystem\Server;

/**
 * MediatorTest tests hello world
 */
class MediatorTest extends \PHPUnit_Framework_TestCase
{

    protected $client;

    protected function setUp()
    {
        $media = new Mediator();
        $this->client = new Client($media);
        $media->setColleague(new Database($media), $this->client, new Server($media));
    }
```

```
24
25      public function testOutputHelloWorld()
26      {
27          // testing if Hello World is output :
28          $this->expectOutputString('Hello World');
29          // as you see, the 3 components Client, Server and Database are totally decoupled
30          $this->client->request();
31          // Anyway, it remains complexity in the Mediator that's why the pattern
32          // Observer is preferable in mnay situations.
33      }
34  }
```

1.3.5 Memento

Purpose

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

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

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

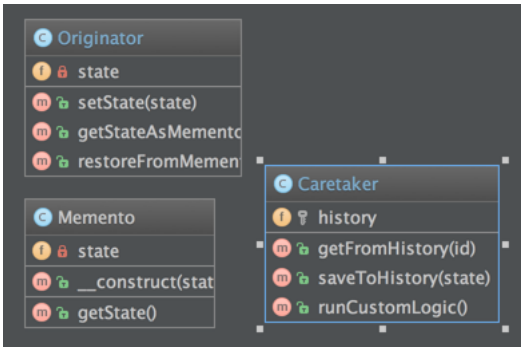
Originator – it is an object that contains the *actual state of an external object is strictly specified type*. Originator is able to create a unique copy of this state and return it wrapped in a Memento. The Originator does not know the history of changes. You can set a concrete state to Originator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

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

Memento.php

```php
<?php

namespace DesignPatterns\Behavioral\Memento;

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

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

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

Originator.php

```php
<?php

namespace DesignPatterns\Behavioral\Memento;

class Originator
{
    /* @var mixed */
    private $state;

```

```
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    /**
     * @param mixed $state
     */
    public function setState($state)
    {
        // you must check type of state inside child of this class
        // or use type-hinting for full pattern implementation
        $this->state = $state;
    }

    /**
     * @return Memento
     */
    public function getStateAsMemento()
    {
        // you must save a separate copy in Memento
        $state = is_object($this->state) ? clone $this->state : $this->state;

        return new Memento($state);
    }

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

Caretaker.php

```
<?php

namespace DesignPatterns\Behavioral\Memento;

class Caretaker
{
    protected $history = array();

    /**
     * @return Memento
     */
    public function getFromHistory($id)
    {
        return $this->history[$id];
    }

    /**
     * @param Memento $state
     */
    public function saveToHistory(Memento $state)
    {
        $this->history[] = $state;
    }

    public function runCustomLogic()
    {
        $originator = new Originator();
```

```
28
29          //Setting state to State1
30          $originator->setState("State1");
31          //Setting state to State2
32          $originator->setState("State2");
33          //Saving State2 to Memento
34          $this->saveToHistory($originator->getStateAsMemento());
35          //Setting state to State3
36          $originator->setState("State3");
37
38          // We can request multiple mementos, and choose which one to roll back to.
39          // Saving State3 to Memento
40          $this->saveToHistory($originator->getStateAsMemento());
41          //Setting state to State4
42          $originator->setState("State4");
43
44          $originator->restoreFromMemento($this->getFromHistory(1));
45          //State after restoring from Memento: State3
46
47          return $originator->getStateAsMemento()->getState();
48      }
49  }
```

Test

Tests/MementoTest.php

```
1   <?php
2
3   namespace DesignPatterns\Behavioral\Memento\Tests;
4
5   use DesignPatterns\Behavioral\Memento\Caretaker;
6   use DesignPatterns\Behavioral\Memento\Memento;
7   use DesignPatterns\Behavioral\Memento\Originator;
8
9   /**
10   * MementoTest tests the memento pattern
11   */
12  class MementoTest extends \PHPUnit_Framework_TestCase
13  {
14
15      public function testUsageExample()
16      {
17          $originator = new Originator();
18          $caretaker = new Caretaker();
19
20          $character = new \stdClass();
21          // new object
22          $character->name = "Gandalf";
23          // connect Originator to character object
24          $originator->setState($character);
25
26          // work on the object
27          $character->name = "Gandalf the Grey";
28          // still change something
29          $character->race = "Maia";
30          // time to save state
```

```
31          $snapshot = $originator->getStateAsMemento();
32          // put state to log
33          $caretaker->saveToHistory($snapshot);
34
35          // change something
36          $character->name = "Sauron";
37          // and again
38          $character->race = "Ainur";
39          // state inside the Originator was equally changed
40          $this->assertAttributeEquals($character, "state", $originator);
41
42          // time to save another state
43          $snapshot = $originator->getStateAsMemento();
44          // put state to log
45          $caretaker->saveToHistory($snapshot);
46
47          $rollback = $caretaker->getFromHistory(0);
48          // return to first state
49          $originator->restoreFromMemento($rollback);
50          // use character from old state
51          $character = $rollback->getState();
52
53          // yes, that what we need
54          $this->assertEquals("Gandalf the Grey", $character->name);
55          // make new changes
56          $character->name = "Gandalf the White";
57
58          // and Originator linked to actual object again
59          $this->assertAttributeEquals($character, "state", $originator);
60      }
61
62      public function testStringState()
63      {
64          $originator = new Originator();
65          $originator->setState("State1");
66
67          $this->assertAttributeEquals("State1", "state", $originator);
68
69          $originator->setState("State2");
70          $this->assertAttributeEquals("State2", "state", $originator);
71
72          $snapshot = $originator->getStateAsMemento();
73          $this->assertAttributeEquals("State2", "state", $snapshot);
74
75          $originator->setState("State3");
76          $this->assertAttributeEquals("State3", "state", $originator);
77
78          $originator->restoreFromMemento($snapshot);
79          $this->assertAttributeEquals("State2", "state", $originator);
80      }
81
82      public function testSnapshotIsClone()
83      {
84          $originator = new Originator();
85          $object = new \stdClass();
86
87          $originator->setState($object);
88          $snapshot = $originator->getStateAsMemento();
```

```
89          $object->new_property = 1;
90
91          $this->assertAttributeEquals($object, "state", $originator);
92          $this->assertAttributeNotEquals($object, "state", $snapshot);
93
94          $originator->restoreFromMemento($snapshot);
95          $this->assertAttributeNotEquals($object, "state", $originator);
96      }
97
98      public function testCanChangeActualState()
99      {
100         $originator = new Originator();
101         $first_state = new \stdClass();
102
103         $originator->setState($first_state);
104         $snapshot = $originator->getStateAsMemento();
105         $second_state = $snapshot->getState();
106
107         // still actual
108         $first_state->first_property = 1;
109         // just history
110         $second_state->second_property = 2;
111         $this->assertAttributeEquals($first_state, "state", $originator);
112         $this->assertAttributeNotEquals($second_state, "state", $originator);
113
114         $originator->restoreFromMemento($snapshot);
115         // now it lost state
116         $first_state->first_property = 11;
117         // must be actual
118         $second_state->second_property = 22;
119         $this->assertAttributeEquals($second_state, "state", $originator);
120         $this->assertAttributeNotEquals($first_state, "state", $originator);
121     }
122
123     public function testStateWithDifferentObjects()
124     {
125         $originator = new Originator();
126
127         $first = new \stdClass();
128         $first->data = "foo";
129
130         $originator->setState($first);
131         $this->assertAttributeEquals($first, "state", $originator);
132
133         $first_snapshot = $originator->getStateAsMemento();
134         $this->assertAttributeEquals($first, "state", $first_snapshot);
135
136         $second       = new \stdClass();
137         $second->data = "bar";
138         $originator->setState($second);
139         $this->assertAttributeEquals($second, "state", $originator);
140
141         $originator->restoreFromMemento($first_snapshot);
142         $this->assertAttributeEquals($first, "state", $originator);
143     }
144
145     public function testCaretaker()
146     {
```

```
147         $caretaker = new Caretaker();
148         $memento1 = new Memento("foo");
149         $memento2 = new Memento("bar");
150         $caretaker->saveToHistory($memento1);
151         $caretaker->saveToHistory($memento2);
152         $this->assertAttributeEquals(array($memento1, $memento2), "history", $caretaker);
153         $this->assertEquals($memento1, $caretaker->getFromHistory(0));
154         $this->assertEquals($memento2, $caretaker->getFromHistory(1));
155
156     }
157
158     public function testCaretakerCustomLogic()
159     {
160         $caretaker = new Caretaker();
161         $result = $caretaker->runCustomLogic();
162         $this->assertEquals("State3", $result);
163     }
164 }
```

1.3.6 Null Object

Purpose

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

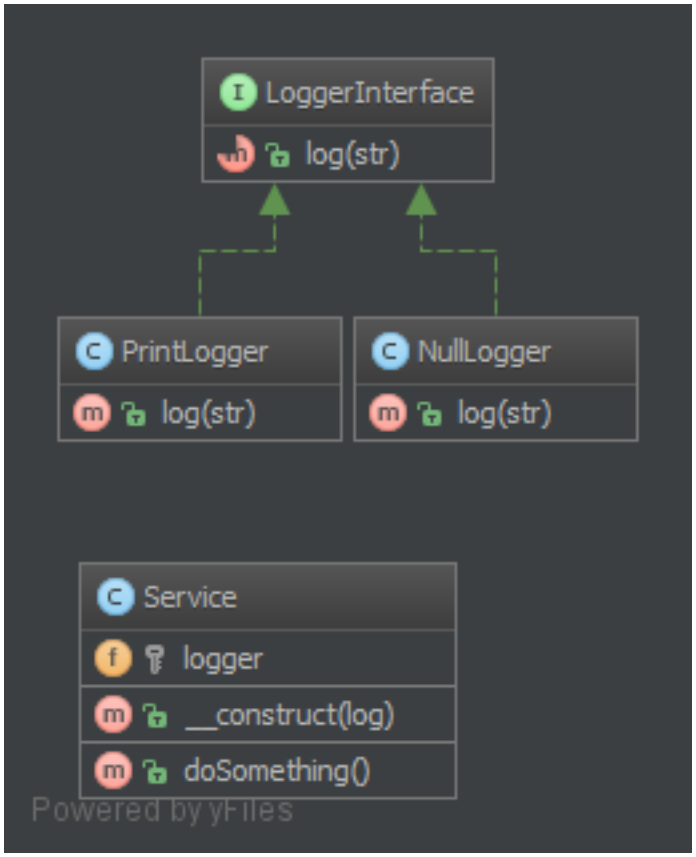
- Client code is simplified
- Reduces the chance of null pointer exceptions
- Fewer conditionals require less test cases

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

Examples

- 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

UML Diagram



Code

You can also find these code on GitHub

Service.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * Service is dummy service that uses a logger
 */
class Service
{
    /**
     * @var LoggerInterface
     */
    protected $logger;

    /**
     * we inject the logger in ctor and it is mandatory
     *
     * @param LoggerInterface $log
     */
    public function __construct(LoggerInterface $log)
```

```
    {
        $this->logger = $log;
    }

    /**
     * do something ...
     */
    public function doSomething()
    {
        // no more check "if (!is_null($this->logger))..." with the NullObject pattern
        $this->logger->log('We are in ' . __METHOD__);
        // something to do...
    }
}
```

LoggerInterface.php

```
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * LoggerInterface is a contract for logging something
 *
 * Key feature: NullLogger MUST inherit from this interface like any other Loggers
 */
interface LoggerInterface
{
    /**
     * @param string $str
     *
     * @return mixed
     */
    public function log($str);
}
```

PrintLogger.php

```
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * PrintLogger is a logger that prints the log entry to standard output
 */
class PrintLogger implements LoggerInterface
{
    /**
     * @param string $str
     */
    public function log($str)
    {
        echo $str;
    }
}
```

NullLogger.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject;

/**
 * Performance concerns : ok there is a call for nothing but we spare an "if is_null"
 * I didn't run a benchmark but I think it's equivalent.
 *
 * Key feature : of course this logger MUST implement the same interface (or abstract)
 * like the other loggers.
 */
class NullLogger implements LoggerInterface
{
    /**
     * {@inheritdoc}
     */
    public function log($str)
    {
        // do nothing
    }
}
```

Test

Tests/LoggerTest.php

```php
<?php

namespace DesignPatterns\Behavioral\NullObject\Tests;

use DesignPatterns\Behavioral\NullObject\NullLogger;
use DesignPatterns\Behavioral\NullObject\Service;
use DesignPatterns\Behavioral\NullObject\PrintLogger;

/**
 * LoggerTest tests for different loggers
 */
class LoggerTest extends \PHPUnit_Framework_TestCase
{

    public function testNullObject()
    {
        // one can use a singleton for NullObjet : I don't think it's a good idea
        // because the purpose behind null object is to "avoid special case".
        $service = new Service(new NullLogger());
        $this->expectOutputString(null);  // no output
        $service->doSomething();
    }

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

1.3.7 Observer

Purpose

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

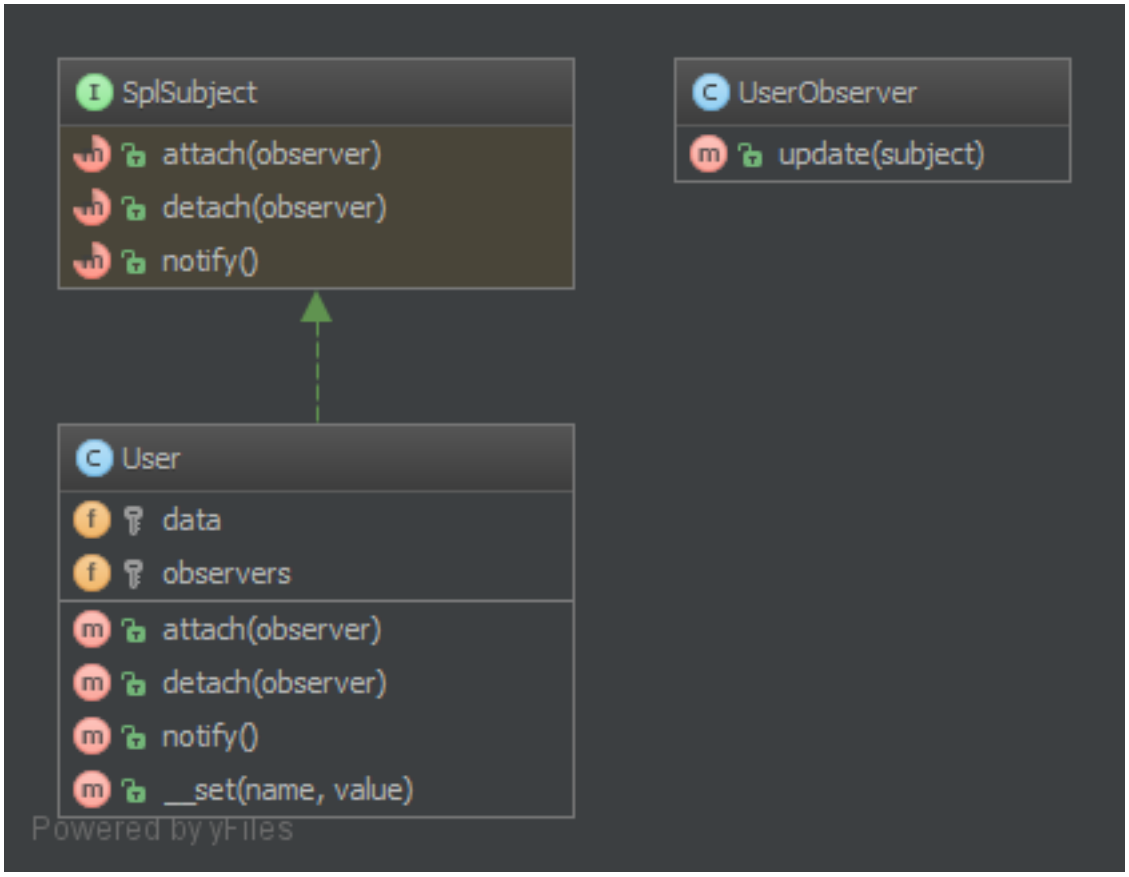
Examples

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

Note

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

UML Diagram



Code

You can also find these code on GitHub

User.php

```php
<?php

namespace DesignPatterns\Behavioral\Observer;

/**
 * Observer pattern : The observed object (the subject)
 *
 * The subject maintains a list of Observers and sends notifications.
 *
 */
class User implements \SplSubject
{
    /**
     * user data
     *
     * @var array
     */
    protected $data = array();

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

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

    /**
     * attach a new observer
     *
     * @param \SplObserver $observer
     *
     * @return void
     */
    public function attach(\SplObserver $observer)
    {
        $this->observers->attach($observer);
    }

    /**
     * detach an observer
     *
     * @param \SplObserver $observer
     *
     * @return void
     */
    public function detach(\SplObserver $observer)
    {
        $this->observers->detach($observer);
    }

    /**
     * notify observers
     *
```

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

    /**
     * Ideally one would better write setter/getter for all valid attributes and only call notify()
     * on attributes that matter when changed
     *
     * @param string $name
     * @param mixed  $value
     *
     * @return void
     */
    public function __set($name, $value)
    {
        $this->data[$name] = $value;

        // notify the observers, that user has been updated
        $this->notify();
    }
}
```

UserObserver.php

```
<?php

namespace DesignPatterns\Behavioral\Observer;

/**
 * class UserObserver
 */
class UserObserver implements \SplObserver
{
    /**
     * This is the only method to implement as an observer.
     * It is called by the Subject (usually by SplSubject::notify() )
     *
     * @param \SplSubject $subject
     */
    public function update(\SplSubject $subject)
    {
        echo get_class($subject) . ' has been updated';
    }
}
```

Test

Tests/ObserverTest.php

```
<?php

```

```php
namespace DesignPatterns\Behavioral\Observer\Tests;

use DesignPatterns\Behavioral\Observer\UserObserver;
use DesignPatterns\Behavioral\Observer\User;

/**
 * ObserverTest tests the Observer pattern
 */
class ObserverTest extends \PHPUnit_Framework_TestCase
{

    protected $observer;

    protected function setUp()
    {
        $this->observer = new UserObserver();
    }

    /**
     * Tests the notification
     */
    public function testNotify()
    {
        $this->expectOutputString('DesignPatterns\Behavioral\Observer\User has been updated');
        $subject = new User();

        $subject->attach($this->observer);
        $subject->property = 123;
    }

    /**
     * Tests the subscribing
     */
    public function testAttachDetach()
    {
        $subject = new User();
        $reflection = new \ReflectionProperty($subject, 'observers');

        $reflection->setAccessible(true);
        /** @var \SplObjectStorage $observers */
        $observers = $reflection->getValue($subject);

        $this->assertInstanceOf('SplObjectStorage', $observers);
        $this->assertFalse($observers->contains($this->observer));

        $subject->attach($this->observer);
        $this->assertTrue($observers->contains($this->observer));

        $subject->detach($this->observer);
        $this->assertFalse($observers->contains($this->observer));
    }

    /**
     * Tests the update() invocation on a mockup
     */
    public function testUpdateCalling()
    {
        $subject = new User();
```

```
61          $observer = $this->getMock('SplObserver');
62          $subject->attach($observer);
63
64          $observer->expects($this->once())
65              ->method('update')
66              ->with($subject);
67
68          $subject->notify();
69      }
70  }
```

1.3.8 Specification

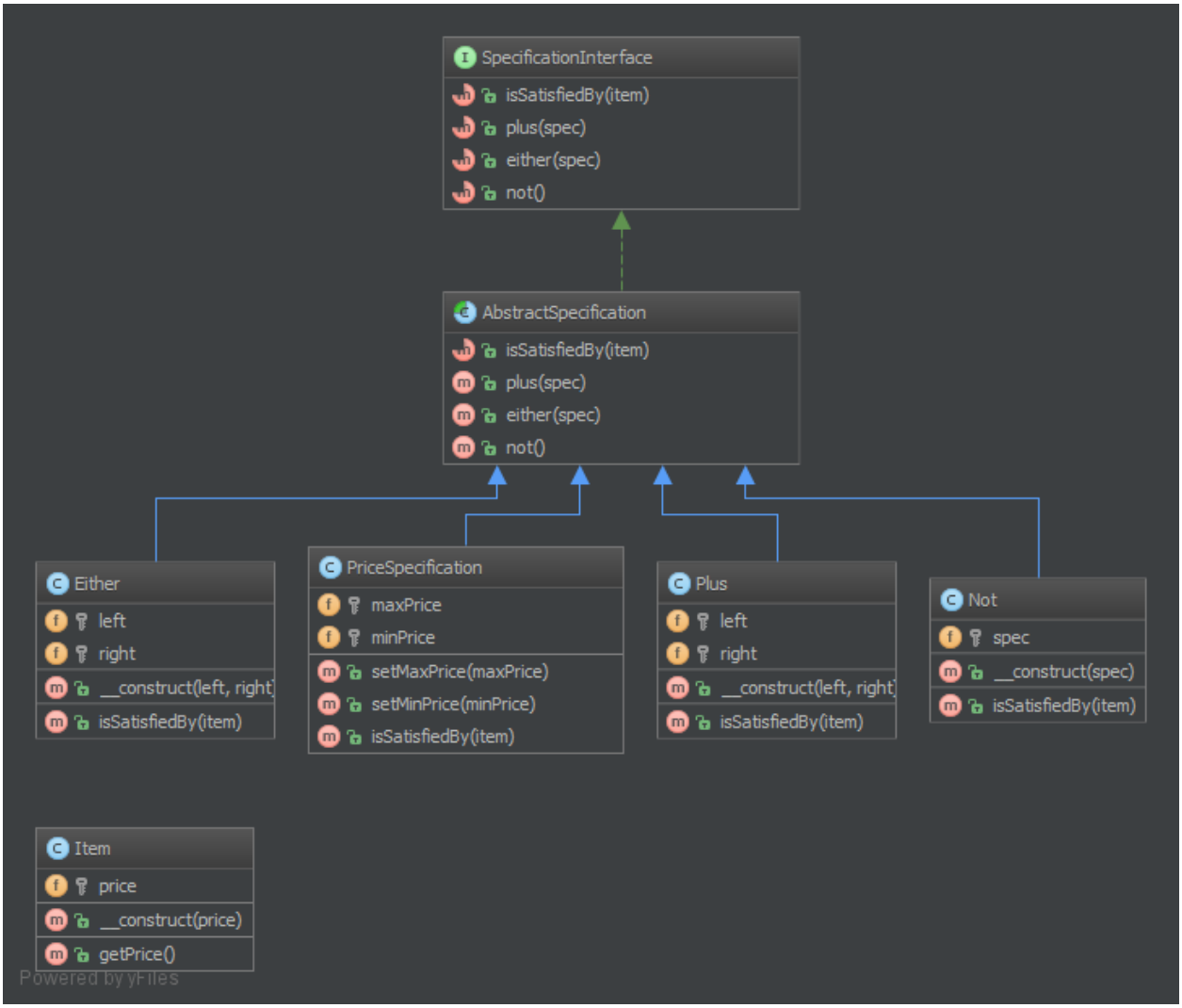
Purpose

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

Examples

- RulerZ

UML Diagram



Code

You can also find these code on GitHub

Item.php

```php
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * An trivial item
 */
class Item
{
    protected $price;

    /**
     * An item must have a price
```

```
     *
     * @param int $price
     */
    public function __construct($price)
    {
        $this->price = $price;
    }

    /**
     * Get the items price
     *
     * @return int
     */
    public function getPrice()
    {
        return $this->price;
    }
}
```

SpecificationInterface.php

```
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * An interface for a specification
 */
interface SpecificationInterface
{
    /**
     * A boolean evaluation indicating if the object meets the specification
     *
     * @param Item $item
     *
     * @return bool
     */
    public function isSatisfiedBy(Item $item);

    /**
     * Creates a logical AND specification
     *
     * @param SpecificationInterface $spec
     */
    public function plus(SpecificationInterface $spec);

    /**
     * Creates a logical OR specification
     *
     * @param SpecificationInterface $spec
     */
    public function either(SpecificationInterface $spec);

    /**
     * Creates a logical not specification
     */
    public function not();
}
```

AbstractSpecification.php

```php
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * An abstract specification allows the creation of wrapped specifications
 */
abstract class AbstractSpecification implements SpecificationInterface
{
    /**
     * Checks if given item meets all criteria
     *
     * @param Item $item
     *
     * @return bool
     */
    abstract public function isSatisfiedBy(Item $item);

    /**
     * Creates a new logical AND specification
     *
     * @param SpecificationInterface $spec
     *
     * @return SpecificationInterface
     */
    public function plus(SpecificationInterface $spec)
    {
        return new Plus($this, $spec);
    }

    /**
     * Creates a new logical OR composite specification
     *
     * @param SpecificationInterface $spec
     *
     * @return SpecificationInterface
     */
    public function either(SpecificationInterface $spec)
    {
        return new Either($this, $spec);
    }

    /**
     * Creates a new logical NOT specification
     *
     * @return SpecificationInterface
     */
    public function not()
    {
        return new Not($this);
    }
}
```

Either.php

```php
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * A logical OR specification
```

```php
 */
class Either extends AbstractSpecification
{

    protected $left;
    protected $right;

    /**
     * A composite wrapper of two specifications
     *
     * @param SpecificationInterface $left
     * @param SpecificationInterface $right
     */
    public function __construct(SpecificationInterface $left, SpecificationInterface $right)
    {
        $this->left = $left;
        $this->right = $right;
    }

    /**
     * Returns the evaluation of both wrapped specifications as a logical OR
     *
     * @param Item $item
     *
     * @return bool
     */
    public function isSatisfiedBy(Item $item)
    {
        return $this->left->isSatisfiedBy($item) || $this->right->isSatisfiedBy($item);
    }
}
```

PriceSpecification.php

```php
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * A specification to check an Item is priced between min and max
 */
class PriceSpecification extends AbstractSpecification
{
    protected $maxPrice;
    protected $minPrice;

    /**
     * Sets the optional maximum price
     *
     * @param int $maxPrice
     */
    public function setMaxPrice($maxPrice)
    {
        $this->maxPrice = $maxPrice;
    }

    /**
     * Sets the optional minimum price
     *
     * @param int $minPrice
```

```
     */
    public function setMinPrice($minPrice)
    {
        $this->minPrice = $minPrice;
    }

    /**
     * Checks if Item price falls between bounds
     *
     * @param Item $item
     *
     * @return bool
     */
    public function isSatisfiedBy(Item $item)
    {
        if (!empty($this->maxPrice) && $item->getPrice() > $this->maxPrice) {
            return false;
        }
        if (!empty($this->minPrice) && $item->getPrice() < $this->minPrice) {
            return false;
        }

        return true;
    }
}
```

Plus.php

```
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * A logical AND specification
 */
class Plus extends AbstractSpecification
{

    protected $left;
    protected $right;

    /**
     * Creation of a logical AND of two specifications
     *
     * @param SpecificationInterface $left
     * @param SpecificationInterface $right
     */
    public function __construct(SpecificationInterface $left, SpecificationInterface $right)
    {
        $this->left = $left;
        $this->right = $right;
    }

    /**
     * Checks if the composite AND of specifications passes
     *
     * @param Item $item
     *
     * @return bool
     */
```

```
    public function isSatisfiedBy(Item $item)
    {
        return $this->left->isSatisfiedBy($item) && $this->right->isSatisfiedBy($item);
    }
}
```

Not.php

```
<?php
namespace DesignPatterns\Behavioral\Specification;

/**
 * A logical Not specification
 */
class Not extends AbstractSpecification
{

    protected $spec;

    /**
     * Creates a new specification wrapping another
     *
     * @param SpecificationInterface $spec
     */
    public function __construct(SpecificationInterface $spec)
    {
        $this->spec = $spec;
    }

    /**
     * Returns the negated result of the wrapped specification
     *
     * @param Item $item
     *
     * @return bool
     */
    public function isSatisfiedBy(Item $item)
    {
        return !$this->spec->isSatisfiedBy($item);
    }
}
```

Test

Tests/SpecificationTest.php

```
<?php

namespace DesignPatterns\Behavioral\Specification\Tests;

use DesignPatterns\Behavioral\Specification\PriceSpecification;
use DesignPatterns\Behavioral\Specification\Item;

/**
 * SpecificationTest tests the specification pattern
 */
class SpecificationTest extends \PHPUnit_Framework_TestCase
```

```
{
    public function testSimpleSpecification()
    {
        $item = new Item(100);
        $spec = new PriceSpecification();

        $this->assertTrue($spec->isSatisfiedBy($item));

        $spec->setMaxPrice(50);
        $this->assertFalse($spec->isSatisfiedBy($item));

        $spec->setMaxPrice(150);
        $this->assertTrue($spec->isSatisfiedBy($item));

        $spec->setMinPrice(101);
        $this->assertFalse($spec->isSatisfiedBy($item));

        $spec->setMinPrice(100);
        $this->assertTrue($spec->isSatisfiedBy($item));
    }

    public function testNotSpecification()
    {
        $item = new Item(100);
        $spec = new PriceSpecification();
        $not = $spec->not();

        $this->assertFalse($not->isSatisfiedBy($item));

        $spec->setMaxPrice(50);
        $this->assertTrue($not->isSatisfiedBy($item));

        $spec->setMaxPrice(150);
        $this->assertFalse($not->isSatisfiedBy($item));

        $spec->setMinPrice(101);
        $this->assertTrue($not->isSatisfiedBy($item));

        $spec->setMinPrice(100);
        $this->assertFalse($not->isSatisfiedBy($item));
    }

    public function testPlusSpecification()
    {
        $spec1 = new PriceSpecification();
        $spec2 = new PriceSpecification();
        $plus = $spec1->plus($spec2);

        $item = new Item(100);

        $this->assertTrue($plus->isSatisfiedBy($item));

        $spec1->setMaxPrice(150);
        $spec2->setMinPrice(50);
        $this->assertTrue($plus->isSatisfiedBy($item));

        $spec1->setMaxPrice(150);
        $spec2->setMinPrice(101);
```

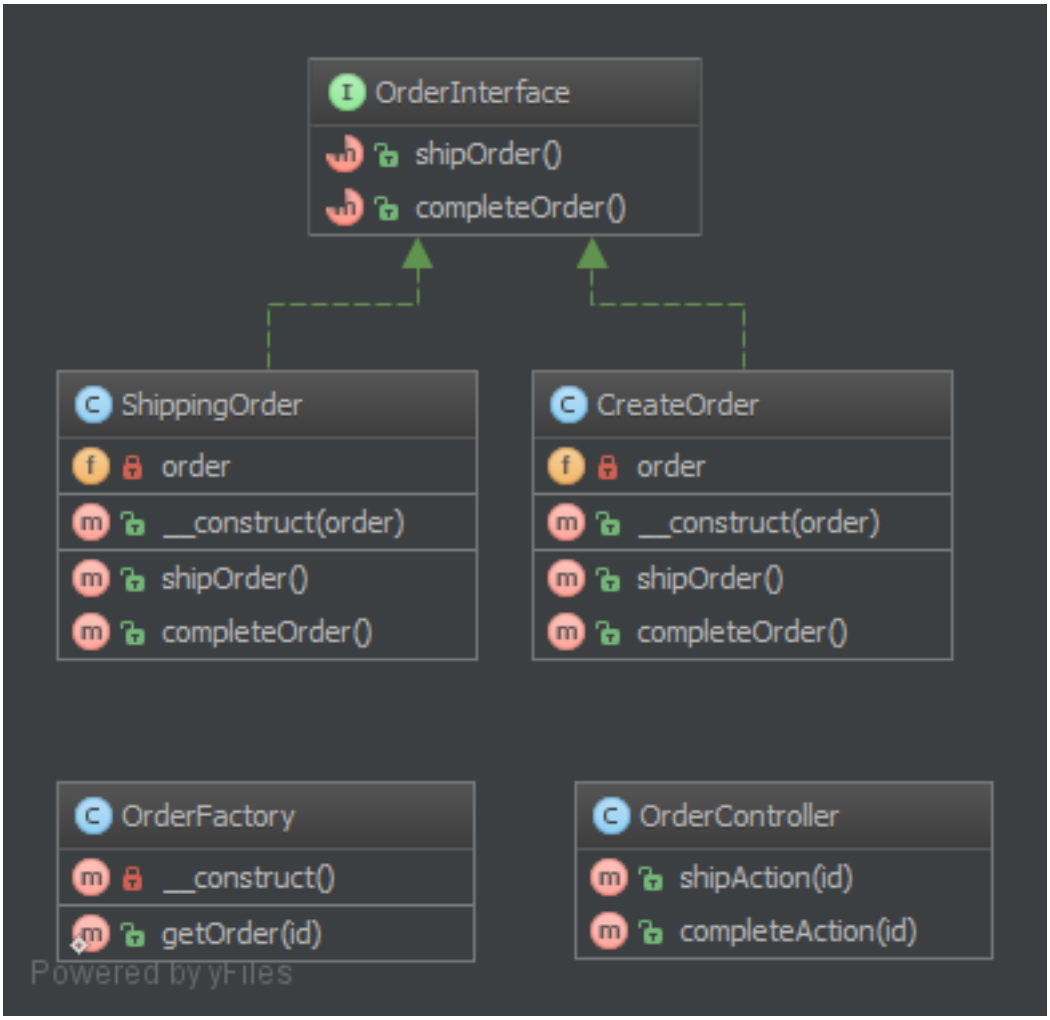
```
70          $this->assertFalse($plus->isSatisfiedBy($item));
71
72          $spec1->setMaxPrice(99);
73          $spec2->setMinPrice(50);
74          $this->assertFalse($plus->isSatisfiedBy($item));
75      }
76
77      public function testEitherSpecification()
78      {
79          $spec1 = new PriceSpecification();
80          $spec2 = new PriceSpecification();
81          $either = $spec1->either($spec2);
82
83          $item = new Item(100);
84
85          $this->assertTrue($either->isSatisfiedBy($item));
86
87          $spec1->setMaxPrice(150);
88          $spec2->setMaxPrice(150);
89          $this->assertTrue($either->isSatisfiedBy($item));
90
91          $spec1->setMaxPrice(150);
92          $spec2->setMaxPrice(0);
93          $this->assertTrue($either->isSatisfiedBy($item));
94
95          $spec1->setMaxPrice(0);
96          $spec2->setMaxPrice(150);
97          $this->assertTrue($either->isSatisfiedBy($item));
98
99          $spec1->setMaxPrice(99);
100         $spec2->setMaxPrice(99);
101         $this->assertFalse($either->isSatisfiedBy($item));
102     }
103 }
```

1.3.9 State

Purpose

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

UML Diagram



Code

You can also find these code on GitHub

OrderController.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

/**
 * Class OrderController
 */
class OrderController
{
    /**
     * @param int $id
     */
    public function shipAction($id)
    {
```

```
        $order = OrderFactory::getOrder($id);
        try {
            $order->shipOrder();
        } catch (Exception $e) {
            //handle error!
        }
        // response to browser
    }

    /**
     * @param int $id
     */
    public function completeAction($id)
    {
        $order = OrderFactory::getOrder($id);
        try {
            $order->completeOrder();
        } catch (Exception $e) {
            //handle error!
        }
        // response to browser
    }
}
```

OrderFactory.php

```
<?php

namespace DesignPatterns\Behavioral\State;

/**
 * Class OrderFactory
 */
class OrderFactory
{
    private function __construct()
    {
        throw new \Exception('Can not instance the OrderFactory class!');
    }

    /**
     * @param int $id
     *
     * @return CreateOrder|ShippingOrder
     * @throws \Exception
     */
    public static function getOrder($id)
    {
        $order = 'Get Order From Database';

        switch ($order['status']) {
            case 'created':
                return new CreateOrder($order);
            case 'shipping':
                return new ShippingOrder($order);
            default:
                throw new \Exception('Order status error!');
                break;
```

```
        }
    }
}
```

OrderInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

/**
 * Class OrderInterface
 */
interface OrderInterface
{
    /**
     * @return mixed
     */
    public function shipOrder();

    /**
     * @return mixed
     */
    public function completeOrder();
}
```

ShippingOrder.php

```php
<?php

namespace DesignPatterns\Behavioral\State;

/**
 * Class ShippingOrder
 */
class ShippingOrder implements OrderInterface
{
    /**
     * @var array
     */
    private $order;

    /**
     * @param array $order
     *
     * @throws \Exception
     */
    public function __construct(array $order)
    {
        if (empty($order)) {
            throw new \Exception('Order can not be empty!');
        }
        $this->order = $order;
    }

    /**
     * @return mixed|void
     * @throws \Exception
     */
```

```
    public function shipOrder()
    {
        //Can not ship the order which status is shipping, throw exception;
        throw new \Exception('Can not ship the order which status is shipping!');
    }

    /**
     * @return mixed
     */
    public function completeOrder()
    {
        $this->order['status'] = 'completed';
        $this->order['updatedTime'] = time();

        // Setting the new order status into database;
        return $this->updateOrder($this->order);
    }
}
```

CreateOrder.php

```
<?php

namespace DesignPatterns\Behavioral\State;

/**
 * Class CreateOrder
 */
class CreateOrder implements OrderInterface
{
    /**
     * @var array
     */
    private $order;

    /**
     * @param array $order
     *
     * @throws \Exception
     */
    public function __construct(array $order)
    {
        if (empty($order)) {
            throw new \Exception('Order can not be empty!');
        }
        $this->order = $order;
    }

    /**
     * @return mixed
     */
    public function shipOrder()
    {
        $this->order['status'] = 'shipping';
        $this->order['updatedTime'] = time();

        // Setting the new order status into database;
        return $this->updateOrder($this->order);
```

```
38      }
39
40      /**
41       * @return mixed|void
42       * @throws \Exception
43       */
44      public function completeOrder()
45      {
46          //Can not complete the order which status is created, throw exception;
47          throw new \Exception('Can not complete the order which status is created!');
48      }
49  }
```

Test

1.3.10 Strategy

Terminology:

- Context
- Strategy
- Concrete Strategy

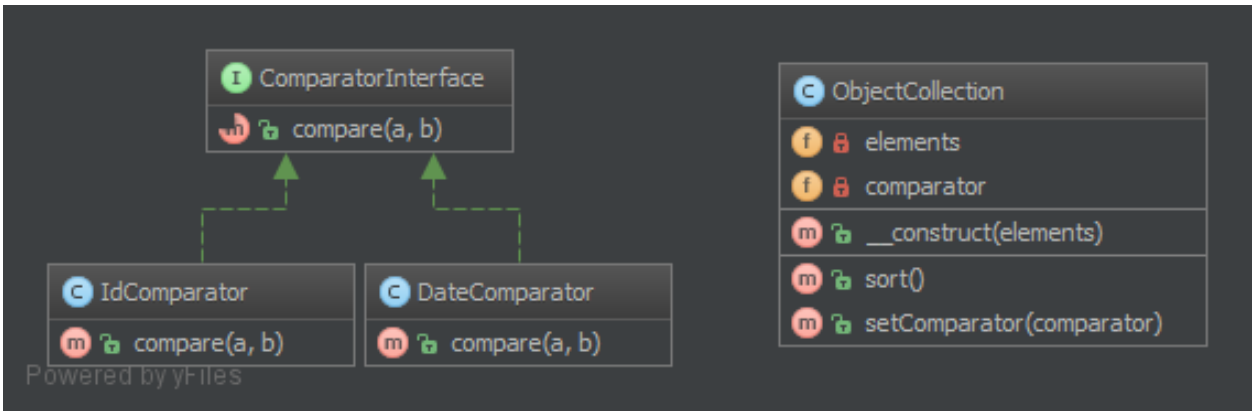
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

ObjectCollection.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

/**
 * Class ObjectCollection
 */
class ObjectCollection
{
    /**
     * @var array
     */
    private $elements;

    /**
     * @var ComparatorInterface
     */
    private $comparator;

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

    /**
     * @return array
     */
    public function sort()
    {
        if (!$this->comparator) {
            throw new \LogicException("Comparator is not set");
        }

        $callback = array($this->comparator, 'compare');
        uasort($this->elements, $callback);

        return $this->elements;
    }

    /**
     * @param ComparatorInterface $comparator
     *
     * @return void
     */
    public function setComparator(ComparatorInterface $comparator)
    {
        $this->comparator = $comparator;
    }
}
```

ComparatorInterface.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

/**
 * Class ComparatorInterface
 */
interface ComparatorInterface
{
    /**
     * @param mixed $a
     * @param mixed $b
     *
     * @return bool
     */
    public function compare($a, $b);
}
```

DateComparator.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

/**
 * Class DateComparator
 */
class DateComparator implements ComparatorInterface
{
    /**
     * {@inheritdoc}
     */
    public function compare($a, $b)
    {
        $aDate = new \DateTime($a['date']);
        $bDate = new \DateTime($b['date']);

        if ($aDate == $bDate) {
            return 0;
        } else {
            return $aDate < $bDate ? -1 : 1;
        }
    }
}
```

IdComparator.php

```php
<?php

namespace DesignPatterns\Behavioral\Strategy;

/**
 * Class IdComparator
 */
class IdComparator implements ComparatorInterface
{
    /**
     * {@inheritdoc}
```

```
     */
    public function compare($a, $b)
    {
        if ($a['id'] == $b['id']) {
            return 0;
        } else {
            return $a['id'] < $b['id'] ? -1 : 1;
        }
    }
}
```

Test

Tests/StrategyTest.php

```
<?php

namespace DesignPatterns\Behavioral\Strategy\Tests;

use DesignPatterns\Behavioral\Strategy\DateComparator;
use DesignPatterns\Behavioral\Strategy\IdComparator;
use DesignPatterns\Behavioral\Strategy\ObjectCollection;
use DesignPatterns\Behavioral\Strategy\Strategy;

/**
 * Tests for Strategy pattern
 */
class StrategyTest extends \PHPUnit_Framework_TestCase
{

    public function getIdCollection()
    {
        return array(
            array(
                array(array('id' => 2), array('id' => 1), array('id' => 3)),
                array('id' => 1)
            ),
            array(
                array(array('id' => 3), array('id' => 2), array('id' => 1)),
                array('id' => 1)
            ),
        );
    }

    public function getDateCollection()
    {
        return array(
            array(
                array(array('date' => '2014-03-03'), array('date' => '2015-03-02'), array('date' =>
                array('date' => '2013-03-01')
            ),
            array(
                array(array('date' => '2014-02-03'), array('date' => '2013-02-01'), array('date' =>
                array('date' => '2013-02-01')
            ),
        );
    }
```

```
43
44      /**
45       * @dataProvider getIdCollection
46       */
47      public function testIdComparator($collection, $expected)
48      {
49          $obj = new ObjectCollection($collection);
50          $obj->setComparator(new IdComparator());
51          $elements = $obj->sort();
52
53          $firstElement = array_shift($elements);
54          $this->assertEquals($expected, $firstElement);
55      }
56
57      /**
58       * @dataProvider getDateCollection
59       */
60      public function testDateComparator($collection, $expected)
61      {
62          $obj = new ObjectCollection($collection);
63          $obj->setComparator(new DateComparator());
64          $elements = $obj->sort();
65
66          $firstElement = array_shift($elements);
67          $this->assertEquals($expected, $firstElement);
68      }
69  }
```

1.3.11 Template Method

Purpose

Template Method is a behavioral design pattern.

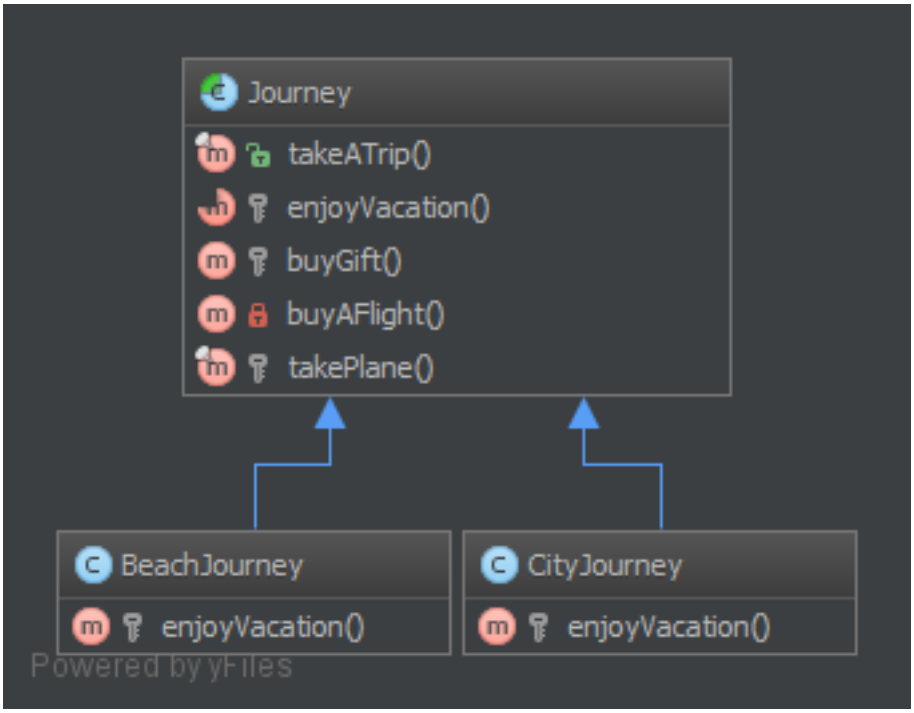
Perhaps you have encountered it many times already. The idea is to let subclasses of this abstract template "finish" the behavior of an algorithm.

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

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

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

UML Diagram



Code

You can also find these code on GitHub

Journey.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

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

```

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

    /**
     * This method is also part of the algorithm but it is optional.
     * This is an "adapter" (do not confuse with the Adapter pattern, not related)
     * You can override it only if you need to.
     */
    protected function buyGift()
    {
    }

    /**
     * This method will be unknown by subclasses (better)
     */
    private function buyAFlight()
    {
        echo "Buying a flight\n";
    }

    /**
     * Subclasses will get access to this method but cannot override it and
     * compromise this algorithm (warning : cause of cyclic dependencies)
     */
    final protected function takePlane()
    {
        echo "Taking the plane\n";
    }

    // A note regarding the keyword "final" : don't use it when you start coding :
    // add it after you narrow and know exactly what change and what remain unchanged
    // in this algorithm.
    // [abstract] x [3 access] x [final] = 12 combinations, it can be hard !
}
```

BeachJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

/**
 * BeachJourney is vacation at the beach
 */
class BeachJourney extends Journey
{
    /**
     * prints what to do to enjoy your vacation
     */
    protected function enjoyVacation()
    {
        echo "Swimming and sun-bathing\n";
    }
}
```

CityJourney.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod;

/**
 * CityJourney is a journey in a city
 */
class CityJourney extends Journey
{
    /**
     * prints what to do in your journey to enjoy vacation
     */
    protected function enjoyVacation()
    {
        echo "Eat, drink, take photos and sleep\n";
    }
}
```

Test

Tests/JourneyTest.php

```php
<?php

namespace DesignPatterns\Behavioral\TemplateMethod\Tests;

use DesignPatterns\Behavioral\TemplateMethod;

/**
 * JourneyTest tests all journeys
 */
class JourneyTest extends \PHPUnit_Framework_TestCase
{

    public function testBeach()
    {
        $journey = new TemplateMethod\BeachJourney();
        $this->expectOutputRegex('#sun-bathing#');
        $journey->takeATrip();
    }

    public function testCity()
    {
        $journey = new TemplateMethod\CityJourney();
        $this->expectOutputRegex('#drink#');
        $journey->takeATrip();
    }

    /**
     * How to test an abstract template method with PHPUnit
     */
    public function testLasVegas()
    {
        $journey = $this->getMockForAbstractClass('DesignPatterns\Behavioral\TemplateMethod\Journey')
        $journey->expects($this->once())
            ->method('enjoyVacation')
            ->will($this->returnCallback(array($this, 'mockUpVacation')));
```

```
36          $this->expectOutputRegex('#Las Vegas#');
37          $journey->takeATrip();
38      }
39
40      public function mockUpVacation()
41      {
42          echo "Fear and loathing in Las Vegas\n";
43      }
44  }
```

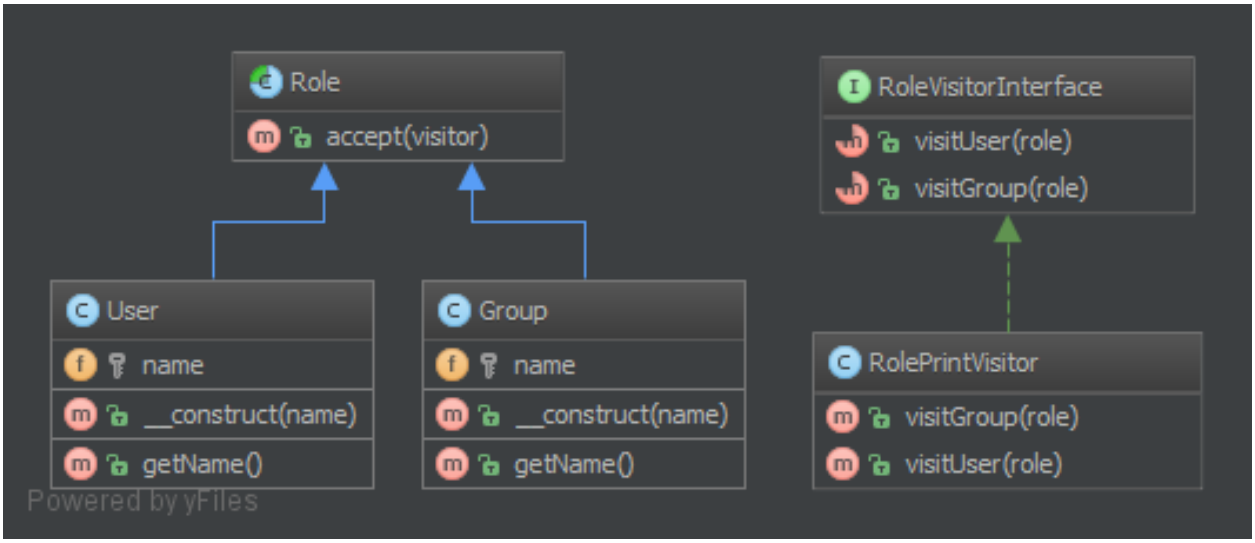
1.3.12 Visitor

Purpose

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

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

UML Diagram



Code

You can also find these code on GitHub

RoleVisitorInterface.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Visitor;
4
5  /**
6   * Visitor Pattern
```

```php
 *
 * The contract for the visitor.
 *
 * Note 1 : in C++ or java, with method polymorphism based on type-hint, there are many
 * methods visit() with different type for the 'role' parameter.
 *
 * Note 2 : the visitor must not choose itself which method to
 * invoke, it is the Visitee that make this decision.
 */
interface RoleVisitorInterface
{
    /**
     * Visit a User object
     *
     * @param \DesignPatterns\Behavioral\Visitor\User $role
     */
    public function visitUser(User $role);

    /**
     * Visit a Group object
     *
     * @param \DesignPatterns\Behavioral\Visitor\Group $role
     */
    public function visitGroup(Group $role);
}
```

RolePrintVisitor.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

/**
 * Visitor Pattern
 *
 * An implementation of a concrete Visitor
 */
class RolePrintVisitor implements RoleVisitorInterface
{
    /**
     * {@inheritdoc}
     */
    public function visitGroup(Group $role)
    {
        echo "Role: " . $role->getName();
    }

    /**
     * {@inheritdoc}
     */
    public function visitUser(User $role)
    {
        echo "Role: " . $role->getName();
    }
}
```

Role.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

/**
 * class Role
 */
abstract class Role
{
    /**
     * This method handles a double dispatch based on the short name of the Visitor
     *
     * Feel free to override it if your object must call another visiting behavior
     *
     * @param \DesignPatterns\Behavioral\Visitor\RoleVisitorInterface $visitor
     *
     * @throws \InvalidArgumentException
     */
    public function accept(RoleVisitorInterface $visitor)
    {
        // this trick to simulate double-dispatch based on type-hinting
        $klass = get_called_class();
        preg_match('#([^\\\\]+)$#', $klass, $extract);
        $visitingMethod = 'visit' . $extract[1];

        // this ensures strong typing with visitor interface, not some visitor objects
        if (!method_exists(__NAMESPACE__ . '\RoleVisitorInterface', $visitingMethod)) {
            throw new \InvalidArgumentException("The visitor you provide cannot visit a $klass instar
        }

        call_user_func(array($visitor, $visitingMethod), $this);
    }
}
```

User.php

```php
<?php

namespace DesignPatterns\Behavioral\Visitor;

/**
 * Visitor Pattern
 *
 * One example for a visitee. Each visitee has to extends Role
 */
class User extends Role
{
    /**
     * @var string
     */
    protected $name;

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

```
24
25      /**
26       * @return string
27       */
28      public function getName()
29      {
30          return "User " . $this->name;
31      }
32  }
```

Group.php

```
1   <?php
2
3   namespace DesignPatterns\Behavioral\Visitor;
4
5   /**
6    * An example of a Visitor: Group
7    */
8   class Group extends Role
9   {
10      /**
11       * @var string
12       */
13      protected $name;
14
15      /**
16       * @param string $name
17       */
18      public function __construct($name)
19      {
20          $this->name = (string) $name;
21      }
22
23      /**
24       * @return string
25       */
26      public function getName()
27      {
28          return "Group: " . $this->name;
29      }
30  }
```

Test

Tests/VisitorTest.php

```
1   <?php
2
3   namespace DesignPatterns\Tests\Visitor\Tests;
4
5   use DesignPatterns\Behavioral\Visitor;
6
7   /**
8    * VisitorTest tests the visitor pattern
9    */
10  class VisitorTest extends \PHPUnit_Framework_TestCase
11  {
```

```
12
13      protected $visitor;
14
15      protected function setUp()
16      {
17          $this->visitor = new Visitor\RolePrintVisitor();
18      }
19
20      public function getRole()
21      {
22          return array(
23              array(new Visitor\User("Dominik"), 'Role: User Dominik'),
24              array(new Visitor\Group("Administrators"), 'Role: Group: Administrators')
25          );
26      }
27
28      /**
29       * @dataProvider getRole
30       */
31      public function testVisitSomeRole(Visitor\Role $role, $expect)
32      {
33          $this->expectOutputString($expect);
34          $role->accept($this->visitor);
35      }
36
37      /**
38       * @expectedException \InvalidArgumentException
39       * @expectedExceptionMessage Mock
40       */
41      public function testUnknownObject()
42      {
43          $mock = $this->getMockForAbstractClass('DesignPatterns\Behavioral\Visitor\Role');
44          $mock->accept($this->visitor);
45      }
46  }
```

1.4 More

1.4.1 Delegation

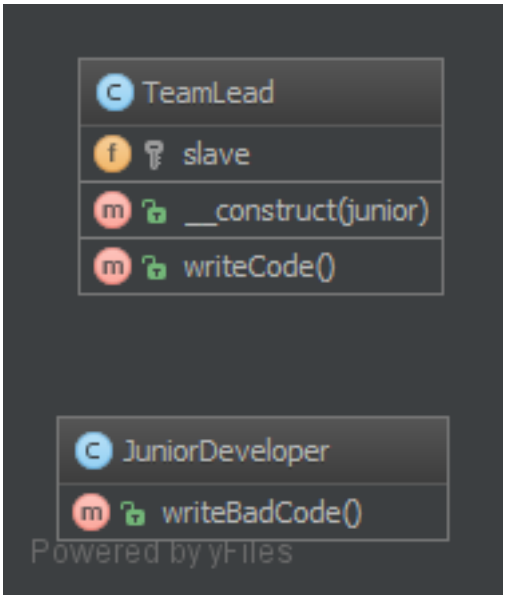
Purpose

Demonstrate the Delegator pattern, where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. In this case TeamLead professes to writeCode and Usage uses this, while TeamLead delegates writeCode to JuniorDeveloper's writeBadCode function. This inverts the responsibility so that Usage is unknowingly executing writeBadCode.

Examples

Please review JuniorDeveloper.php, TeamLead.php, and then Usage.php to see it all tied together.

UML Diagram



Code

You can also find these code on GitHub

Usage.php

```php
<?php

namespace DesignPatterns\More\Delegation;

// instantiate TeamLead and appoint to assistants JuniorDeveloper
$teamLead = new TeamLead(new JuniorDeveloper());

// team lead delegate write code to junior developer
echo $teamLead->writeCode();
```

TeamLead.php

```php
<?php

namespace DesignPatterns\More\Delegation;

/**
 * Class TeamLead
 * @package DesignPatterns\Delegation
 * The `TeamLead` class, he delegate work to `JuniorDeveloper`
 */
class TeamLead
{
    /** @var JuniorDeveloper */
    protected $slave;

    /**
     * Give junior developer into teamlead submission
     * @param JuniorDeveloper $junior
```

```
     */
    public function __construct(JuniorDeveloper $junior)
    {
        $this->slave = $junior;
    }

    /**
     * TeamLead drink coffee, junior work
     * @return mixed
     */
    public function writeCode()
    {
        return $this->slave->writeBadCode();
    }
}
```

JuniorDeveloper.php

```
<?php

namespace DesignPatterns\More\Delegation;

/**
 * Class JuniorDeveloper
 * @package DesignPatterns\Delegation
 */
class JuniorDeveloper
{
    public function writeBadCode()
    {
        return "Some junior developer generated code...";
    }
}
```

Test

Tests/DelegationTest.php

```
<?php

namespace DesignPatterns\More\Delegation\Tests;

use DesignPatterns\More\Delegation;

/**
 * DelegationTest tests the delegation pattern
 */
class DelegationTest extends \PHPUnit_Framework_TestCase
{
    public function testHowTeamLeadWriteCode()
    {
        $junior = new Delegation\JuniorDeveloper();
        $teamLead = new Delegation\TeamLead($junior);
        $this->assertEquals($junior->writeBadCode(), $teamLead->writeCode());
    }
}
```

1.4.2 Service Locator

Purpose

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

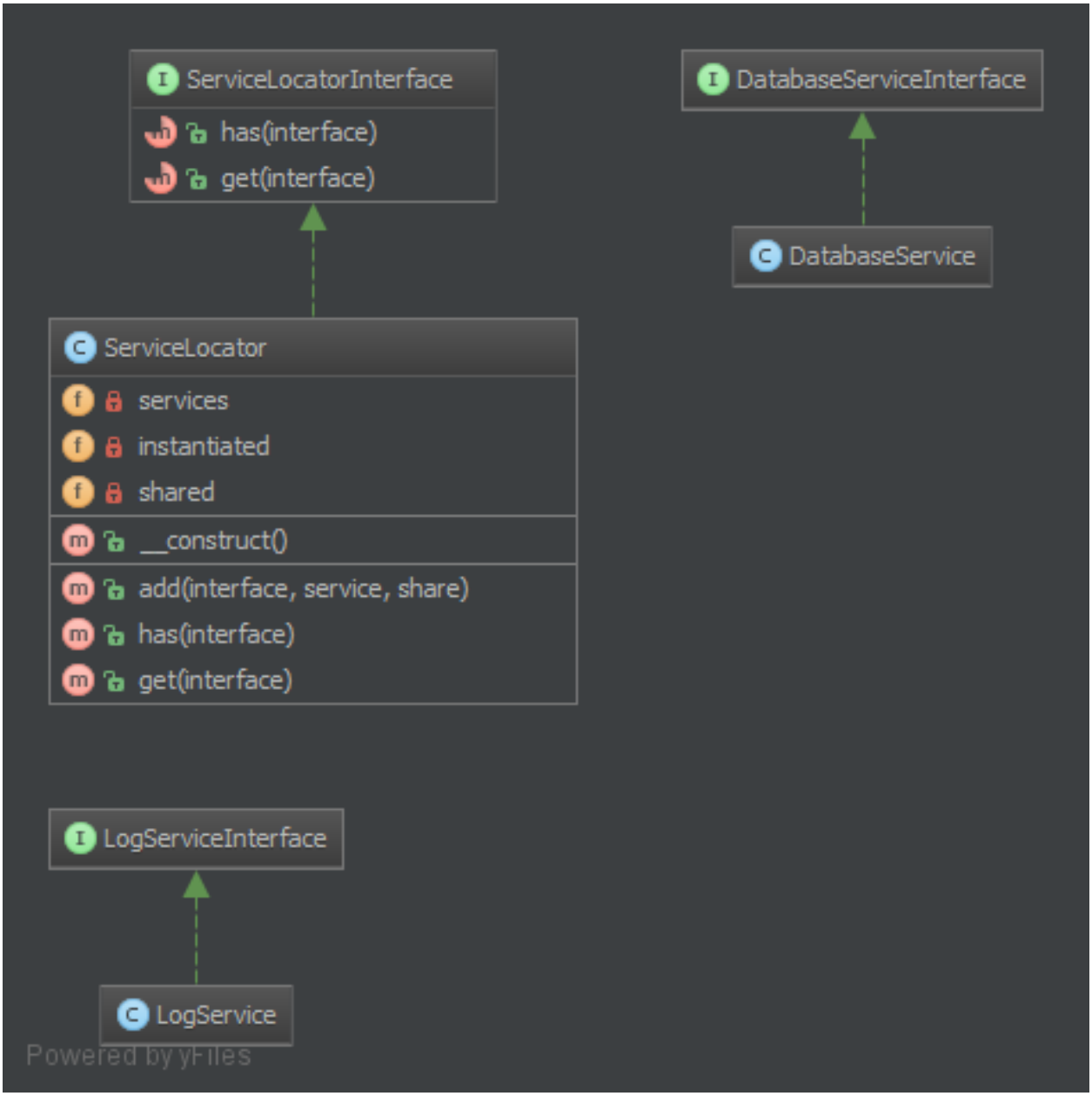
Usage

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

Examples

- Zend Framework 2 uses Service Locator to create and share services used in the framework(i.e. EventManager, ModuleManager, all custom user services provided by modules, etc...)

UML Diagram



Code

You can also find these code on GitHub

ServiceLocatorInterface.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

interface ServiceLocatorInterface
{
```

```
    /**
     * Checks if a service is registered.
     *
     * @param string $interface
     *
     * @return bool
     */
    public function has($interface);

    /**
     * Gets the service registered for the interface.
     *
     * @param string $interface
     *
     * @return mixed
     */
    public function get($interface);
}
```

ServiceLocator.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

class ServiceLocator implements ServiceLocatorInterface
{
    /**
     * All services.
     *
     * @var array
     */
    private $services;

    /**
     * The services which have an instance.
     *
     * @var array
     */
    private $instantiated;

    /**
     * True if a service can be shared.
     *
     * @var array
     */
    private $shared;

    public function __construct()
    {
        $this->services     = array();
        $this->instantiated = array();
        $this->shared       = array();
    }

    /**
     * Registers a service with specific interface.
     *
     * @param string $interface
```

```
     * @param string|object $service
     * @param bool $share
     */
    public function add($interface, $service, $share = true)
    {
        /**
         * When you add a service, you should register it
         * with its interface or with a string that you can use
         * in the future even if you will change the service implementation.
         */

        if (is_object($service) && $share) {
            $this->instantiated[$interface] = $service;
        }
        $this->services[$interface] = (is_object($service) ? get_class($service) : $service);
        $this->shared[$interface]   = $share;
    }

    /**
     * Checks if a service is registered.
     *
     * @param string $interface
     *
     * @return bool
     */
    public function has($interface)
    {
        return (isset($this->services[$interface]) || isset($this->instantiated[$interface]));
    }

    /**
     * Gets the service registered for the interface.
     *
     * @param string $interface
     *
     * @return mixed
     */
    public function get($interface)
    {
        // Retrieves the instance if it exists and it is shared
        if (isset($this->instantiated[$interface]) && $this->shared[$interface]) {
            return $this->instantiated[$interface];
        }

        // otherwise gets the service registered.
        $service = $this->services[$interface];

        // You should check if the service class exists and
        // the class is instantiable.

        // This example is a simple implementation, but
        // when you create a service, you can decide
        // if $service is a factory or a class.
        // By registering a factory you can create your services
        // using the DependencyInjection pattern.

        // ...

```

```
        // Creates the service object
        $object = new $service();

        // and saves it if the service must be shared.
        if ($this->shared[$interface]) {
            $this->instantiated[$interface] = $object;
        }
        return $object;
    }
}
```

LogServiceInterface.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

interface LogServiceInterface
{
}
```

LogService.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

class LogService implements LogServiceInterface
{
}
```

DatabaseServiceInterface.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

interface DatabaseServiceInterface
{
}
```

DatabaseService.php

```
<?php

namespace DesignPatterns\More\ServiceLocator;

class DatabaseService implements DatabaseServiceInterface
{
}
```

Test

Tests/ServiceLocatorTest.php

```
<?php

namespace DesignPatterns\More\ServiceLocator\Tests;
```

```php

use DesignPatterns\More\ServiceLocator\DatabaseService;
use DesignPatterns\More\ServiceLocator\LogService;
use DesignPatterns\More\ServiceLocator\ServiceLocator;
use \PHPUnit_Framework_TestCase as TestCase;

class ServiceLocatorTest extends TestCase
{
    /**
     * @var LogService
     */
    private $logService;

    /**
     * @var DatabaseService
     */
    private $databaseService;

    /**
     * @var ServiceLocator
     */
    private $serviceLocator;

    public function setUp()
    {
        $this->serviceLocator  = new ServiceLocator();
        $this->logService      = new LogService();
        $this->databaseService = new DatabaseService();
    }

    public function testHasServices()
    {
        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
            $this->logService
        );

        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
            $this->databaseService
        );

        $this->assertTrue($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\LogServiceI
        $this->assertTrue($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\DatabaseServ

        $this->assertFalse($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\FakeService
    }

    public function testServicesWithObject()
    {
        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
            $this->logService
        );

        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
            $this->databaseService
```

```
        );

        $this->assertSame(
            $this->logService,
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
        );

        $this->assertSame(
            $this->databaseService,
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
        );
    }

    public function testServicesWithClass()
    {
        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
            get_class($this->logService)
        );

        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
            get_class($this->databaseService)
        );

        $this->assertNotSame(
            $this->logService,
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
        );

        $this->assertInstanceOf(
            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
        );

        $this->assertNotSame(
            $this->databaseService,
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
        );

        $this->assertInstanceOf(
            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
        );
    }

    public function testServicesNotShared()
    {
        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
            $this->logService,
            false
        );

        $this->serviceLocator->add(
            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
            $this->databaseService,
            false
```

```
120         );
121
122         $this->assertNotSame(
123             $this->logService,
124             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
125         );
126
127         $this->assertInstanceOf(
128             'DesignPatterns\More\ServiceLocator\LogServiceInterface',
129             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
130         );
131
132         $this->assertNotSame(
133             $this->databaseService,
134             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
135         );
136
137         $this->assertInstanceOf(
138             'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
139             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
140         );
141     }
142 }
```

1.4.3 Repository

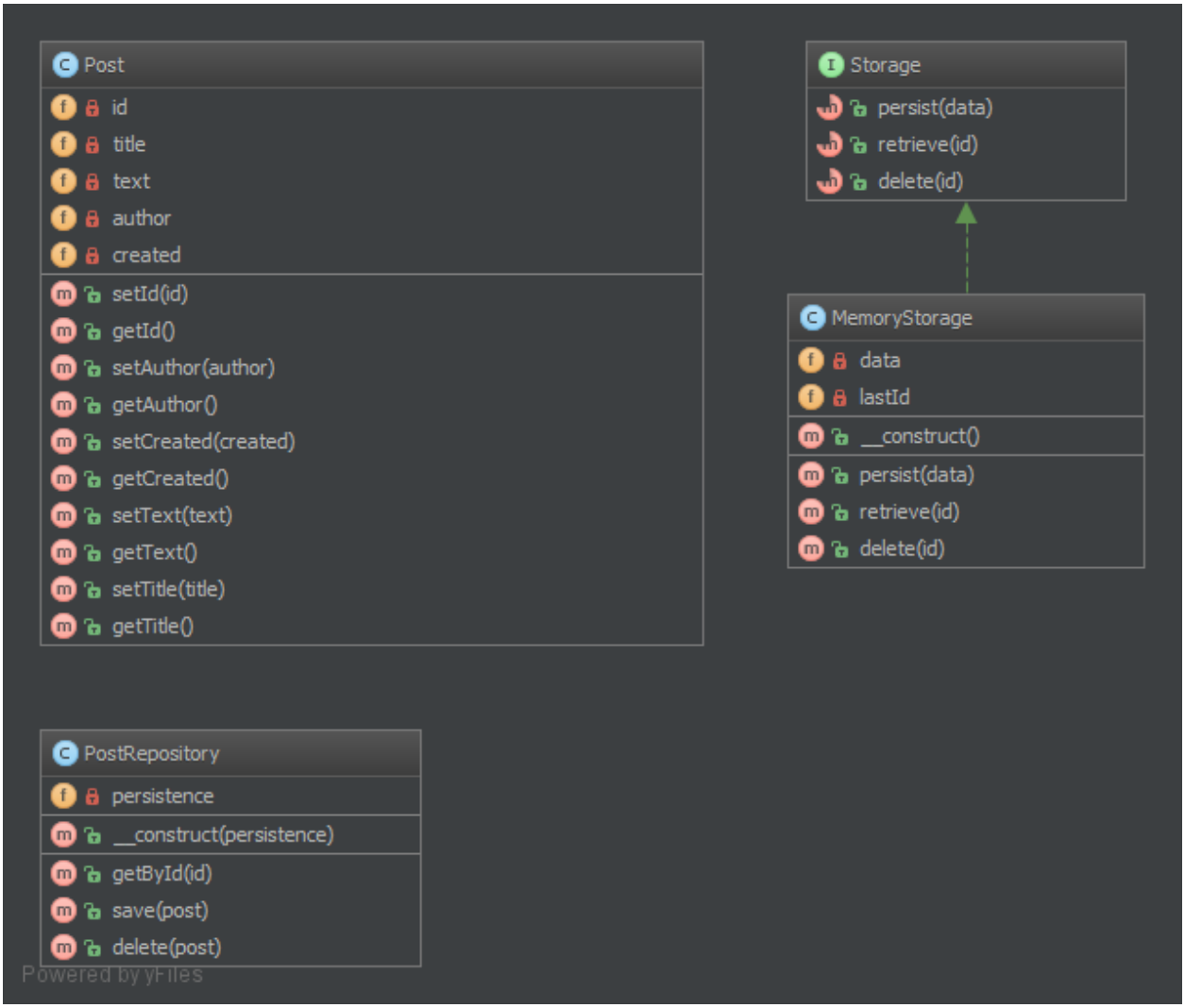
Purpose

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

Examples

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

UML Diagram



Code

You can also find these code on GitHub

Post.php

```php
<?php

namespace DesignPatterns\Repository;

/**
 * Post represents entity for some post that user left on the site
 *
 * Class Post
 * @package DesignPatterns\Repository
 */
class Post
{
```

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

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

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

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

    /**
     * @var \DateTime
     */
    private $created;

    /**
     * @param int $id
     */
    public function setId($id)
    {
        $this->id = $id;
    }

    /**
     * @return int
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @param string $author
     */
    public function setAuthor($author)
    {
        $this->author = $author;
    }

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

    /**
```

```
     * @param \DateTime $created
     */
    public function setCreated($created)
    {
        $this->created = $created;
    }

    /**
     * @return \DateTime
     */
    public function getCreated()
    {
        return $this->created;
    }

    /**
     * @param string $text
     */
    public function setText($text)
    {
        $this->text = $text;
    }

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

    /**
     * @param string $title
     */
    public function setTitle($title)
    {
        $this->title = $title;
    }

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

PostRepository.php

```
<?php

namespace DesignPatterns\Repository;

/**
 * Repository for class Post
 * This class is between Entity layer(class Post) and access object layer(interface Storage)
 *
```

```php
 * Repository encapsulates the set of objects persisted in a data store and the operations performed
 * 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
 * @package DesignPatterns\Repository
 */
class PostRepository
{
    private $persistence;

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

    /**
     * Returns Post object by specified id
     *
     * @param int $id
     * @return Post|null
     */
    public function getById($id)
    {
        $arrayData = $this->persistence->retrieve($id);
        if (is_null($arrayData)) {
            return null;
        }

        $post = new Post();
        $post->setId($arrayData['id']);
        $post->setAuthor($arrayData['author']);
        $post->setCreated($arrayData['created']);
        $post->setText($arrayData['text']);
        $post->setTitle($arrayData['title']);

        return $post;
    }

    /**
     * Save post object and populate it with id
     *
     * @param Post $post
     * @return Post
     */
    public function save(Post $post)
    {
        $id = $this->persistence->persist(array(
            'author' => $post->getAuthor(),
            'created' => $post->getCreated(),
            'text' => $post->getText(),
            'title' => $post->getTitle()
        ));

        $post->setId($id);
        return $post;
```

```
    }

    /**
     * Deletes specified Post object
     *
     * @param Post $post
     * @return bool
     */
    public function delete(Post $post)
    {
        return $this->persistence->delete($post->getId());
    }
}
```

Storage.php

```
<?php

namespace DesignPatterns\Repository;

/**
 * Interface Storage
 *
 * This interface describes methods for accessing storage.
 * Concrete realization could be whatever we want - in memory, relational database, NoSQL database ar
 *
 * @package DesignPatterns\Repository
 */
interface Storage
{
    /**
     * Method to persist data
     * Returns new id for just persisted data
     *
     * @param array() $data
     * @return int
     */
    public function persist($data);

    /**
     * Returns data by specified id.
     * If there is no such data null is returned
     *
     * @param int $id
     * @return array|null
     */
    public function retrieve($id);

    /**
     * Delete data specified by id
     * If there is no such data - false returns, if data has been successfully deleted - true returns
     *
     * @param int $id
     * @return bool
     */
    public function delete($id);
}
```

MemoryStorage.php

```php
<?php

namespace DesignPatterns\Repository;

/**
 * Class MemoryStorage
 * @package DesignPatterns\Repository
 */
class MemoryStorage implements Storage
{

    private $data;
    private $lastId;

    public function __construct()
    {
        $this->data = array();
        $this->lastId = 0;
    }

    /**
     * {@inheritdoc}
     */
    public function persist($data)
    {
        $this->data[++$this->lastId] = $data;
        return $this->lastId;
    }

    /**
     * {@inheritdoc}
     */
    public function retrieve($id)
    {
        return isset($this->data[$id]) ? $this->data[$id] : null;
    }

    /**
     * {@inheritdoc}
     */
    public function delete($id)
    {
        if (!isset($this->data[$id])) {
            return false;
        }

        $this->data[$id] = null;
        unset($this->data[$id]);

        return true;
    }
}
```

Test

CHAPTER 2

Contribute

Please feel free to fork and extend existing or add your own examples and send a pull request with your changes! To establish a consistent code quality, please check your code using PHP CodeSniffer against PSR2 standard using `./vendor/bin/phpcs -p --standard=PSR2 --ignore=vendor ..`

CHAPTER 3

License

(The MIT License)

Copyright (c) 2014 Dominik Liebler and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.