
DesignPatternsPHP Documentation

Release 1.0

Dominik Liebler and contributors

May 29, 2015

1	Patterns	3
1.1	Creational	3
1.1.1	Abstract Factory	3
1.1.2	Builder	10
1.1.3	Factory Method	17
1.1.4	Multiton	22
1.1.5	Pool	24
1.1.6	Prototype	28
1.1.7	Simple Factory	31
1.1.8	Singleton	35
1.1.9	Static Factory	37
1.2	Structural	40
1.2.1	Adapter / Wrapper	40
1.2.2	Bridge	45
1.2.3	Composite	49
1.2.4	Data Mapper	53
1.2.5	Decorator	60
1.2.6	Dependency Injection	65
1.2.7	Facade	69
1.2.8	Fluent Interface	73
1.2.9	Proxy	76
1.2.10	Registry	79
1.3	Behavioral	81
1.3.1	Chain Of Responsibilities	81
1.3.2	Command	87
1.3.3	Iterator	91
1.3.4	Mediator	97
1.3.5	Memento	102
1.3.6	Null Object	106
1.3.7	Observer	110
1.3.8	Specification	114
1.3.9	State	122
1.3.10	Strategy	126
1.3.11	Template Method	130
1.3.12	Visitor	134
1.4	More	138
1.4.1	Delegation	138
1.4.2	Service Locator	141

1.4.3	Repository	148
2	Contribute	155
3	License	157

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.

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

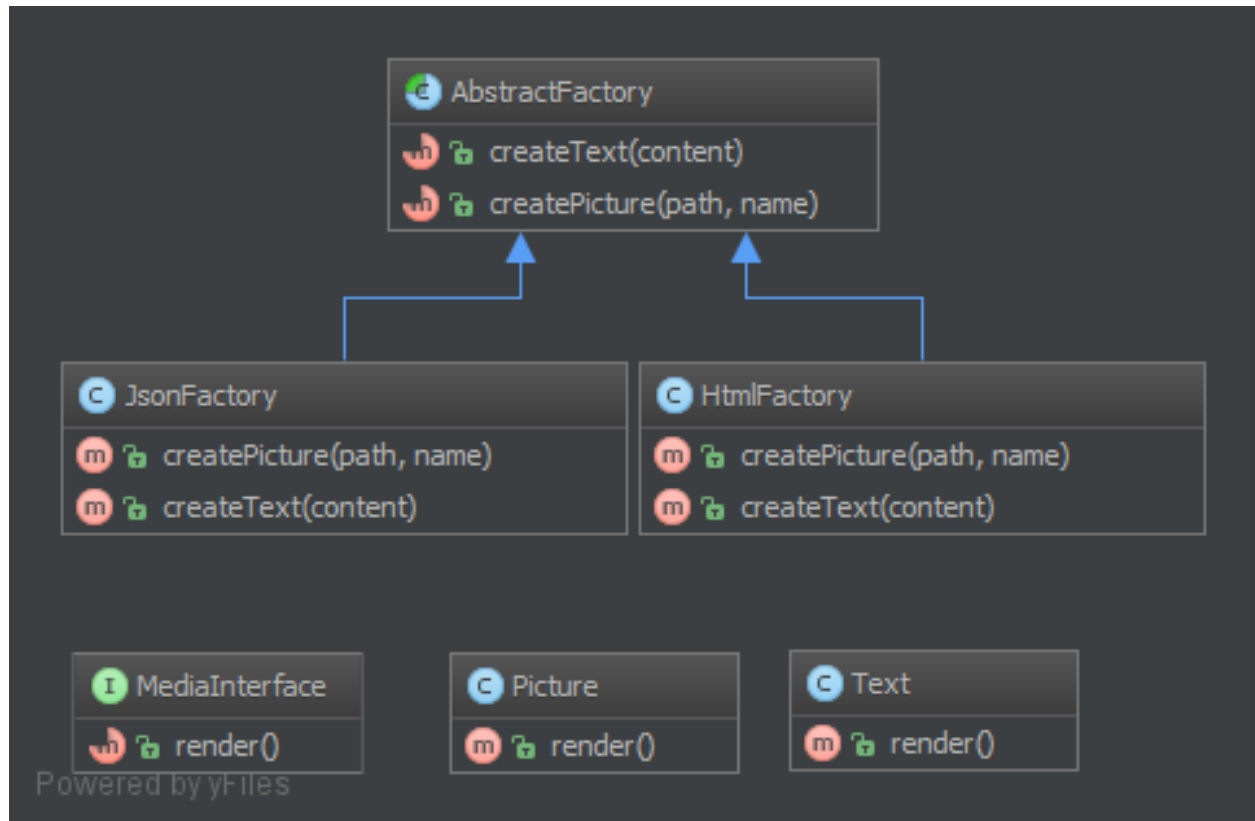
In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns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

```

1  <?php
2
3  namespace DesignPatterns\Creational\AbstractFactory;
4
5  /**
6   * class AbstractFactory
7   *
8   * Sometimes also known as "Kit" in a GUI libraries.
9   *
10  * This design pattern implements the Dependency Inversion Principle since
11  * it is the concrete subclass which creates concrete components.
12  *
13  * In this case, the abstract factory is a contract for creating some components
14  * for the web. There are two components : Text and Picture. There is two ways
15  * of rendering : HTML or JSON.
16  *
17  * Therefore 4 concretes classes, but the client just need to know this contract
18  * to build a correct http response (for a html page or for an ajax request)
19  */
20  abstract class AbstractFactory
21  {

```



```

22  /**
23   * Creates a text component
24   *
25   * @param string $content
26   *
27   * @return Text
28   */
29  abstract public function createText($content);
30
31  /**
32   * Creates a picture component
33   *
34   * @param string $path
35   * @param string $name
36   *
37   * @return Picture
38   */
39  abstract public function createPicture($path, $name = '');
40  }

```

JsonFactory.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\AbstractFactory;
4
5  /**
6   * Class JsonFactory
7   *
8   * JsonFactory is a factory for creating a family of JSON component
9   * (example for ajax)
10  */
11  class JsonFactory extends AbstractFactory
12  {
13
14      /**
15       * Creates a picture component
16       *
17       * @param string $path
18       * @param string $name
19       *
20       * @return Json\Picture|Picture
21       */
22      public function createPicture($path, $name = '')
23      {
24          return new Json\Picture($path, $name);
25      }
26
27      /**
28       * Creates a text component
29       *
30       * @param string $content
31       *
32       * @return Json\Text|Text
33       */
34      public function createText($content)
35      {
36          return new Json\Text($content);
37      }

```

38 }

HtmlFactory.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 /**
6  * Class HtmlFactory
7  *
8  * HtmlFactory is a concrete factory for HTML component
9  */
10 class HtmlFactory extends AbstractFactory
11 {
12     /**
13      * Creates a picture component
14      *
15      * @param string $path
16      * @param string $name
17      *
18      * @return Html\Picture\Picture
19      */
20     public function createPicture($path, $name = '')
21     {
22         return new Html\Picture($path, $name);
23     }
24
25     /**
26      * Creates a text component
27      *
28      * @param string $content
29      *
30      * @return Html\Text\Text
31      */
32     public function createText($content)
33     {
34         return new Html\Text($content);
35     }
36 }
```

MediaInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 /**
6  * Interface MediaInterface
7  *
8  * This contract is not part of the pattern, in general case, each component
9  * are not related
10 */
11 interface MediaInterface
12 {
13
14     /**
15      * some crude rendering from JSON or html output (depended on concrete class)
16      */
17 }
```

```
17     * @return string
18     */
19     public function render();
20 }
```

Picture.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 /**
6  * Class Picture
7  */
8 abstract class Picture implements MediaInterface
9 {
10
11     /**
12      * @var string
13      */
14     protected $path;
15
16     /**
17      * @var string
18      */
19     protected $name;
20
21     /**
22      * @param string $path
23      * @param string $name
24      */
25     public function __construct($path, $name = '')
26     {
27         $this->name = (string) $name;
28         $this->path = (string) $path;
29     }
30 }
```

Text.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory;
4
5 /**
6  * Class Text
7  */
8 abstract class Text implements MediaInterface
9 {
10
11     /**
12      * @var string
13      */
14     protected $text;
15
16     /**
17      * @param string $text
18      */
19     public function __construct($text)
20     {
```

```
20         $this->text = (string) $text;
21     }
22 }
```

Json/Picture.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\AbstractFactory\Json;
4
5  use DesignPatterns\Creational\AbstractFactory\Picture as BasePicture;
6
7  /**
8   * Class Picture
9   *
10  * Picture is a concrete image for JSON rendering
11  */
12  class Picture extends BasePicture
13  {
14      /**
15       * some crude rendering from JSON output
16       *
17       * @return string
18       */
19      public function render()
20      {
21          return json_encode(array('title' => $this->name, 'path' => $this->path));
22      }
23  }
```

Json/Text.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\AbstractFactory\Json;
4
5  use DesignPatterns\Creational\AbstractFactory\Text as BaseText;
6
7  /**
8   * Class Text
9   *
10  * Text is a text component with a JSON rendering
11  */
12  class Text extends BaseText
13  {
14      /**
15       * some crude rendering from JSON output
16       *
17       * @return string
18       */
19      public function render()
20      {
21          return json_encode(array('content' => $this->text));
22      }
23  }
```

Html/Picture.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory\Html;
4
5 use DesignPatterns\Creational\AbstractFactory\Picture as BasePicture;
6
7 /**
8  * Class Picture
9  *
10  * Picture is a concrete image for HTML rendering
11  */
12 class Picture extends BasePicture
13 {
14     /**
15      * some crude rendering from HTML output
16      *
17      * @return string
18      */
19     public function render()
20     {
21         return sprintf('', $this->path, $this->name);
22     }
23 }

```

Html/Text.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory\Html;
4
5 use DesignPatterns\Creational\AbstractFactory\Text as BaseText;
6
7 /**
8  * Class Text
9  *
10  * Text is a concrete text for HTML rendering
11  */
12 class Text extends BaseText
13 {
14     /**
15      * some crude rendering from HTML output
16      *
17      * @return string
18      */
19     public function render()
20     {
21         return '<div>' . htmlspecialchars($this->text) . '</div>';
22     }
23 }

```

Test

Tests/AbstractFactoryTest.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\AbstractFactory\Tests;

```

```
4
5 use DesignPatterns\Creational\AbstractFactory\AbstractFactory;
6 use DesignPatterns\Creational\AbstractFactory\HtmlFactory;
7 use DesignPatterns\Creational\AbstractFactory\JsonFactory;
8
9 /**
10  * AbstractFactoryTest tests concrete factories
11  */
12 class AbstractFactoryTest extends \PHPUnit_Framework_TestCase
13 {
14     public function getFactories()
15     {
16         return array(
17             array(new JsonFactory()),
18             array(new HtmlFactory())
19         );
20     }
21
22     /**
23      * This is the client of factories. Note that the client does not
24      * care which factory is given to him, he can create any component he
25      * wants and render how he wants.
26      *
27      * @dataProvider getFactories
28      */
29     public function testComponentCreation(AbstractFactory $factory)
30     {
31         $article = array(
32             $factory->createText('Lorem Ipsum'),
33             $factory->createPicture('/image.jpg', 'caption'),
34             $factory->createText('footnotes')
35         );
36
37         $this->assertContainsOnly('DesignPatterns\Creational\AbstractFactory\MediaInterface', $article);
38
39         /* this is the time to look at the Builder pattern. This pattern
40          * helps you to create complex object like that article above with
41          * a given Abstract Factory
42          */
43     }
44 }
```

1.1.2 Builder

Purpose

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

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

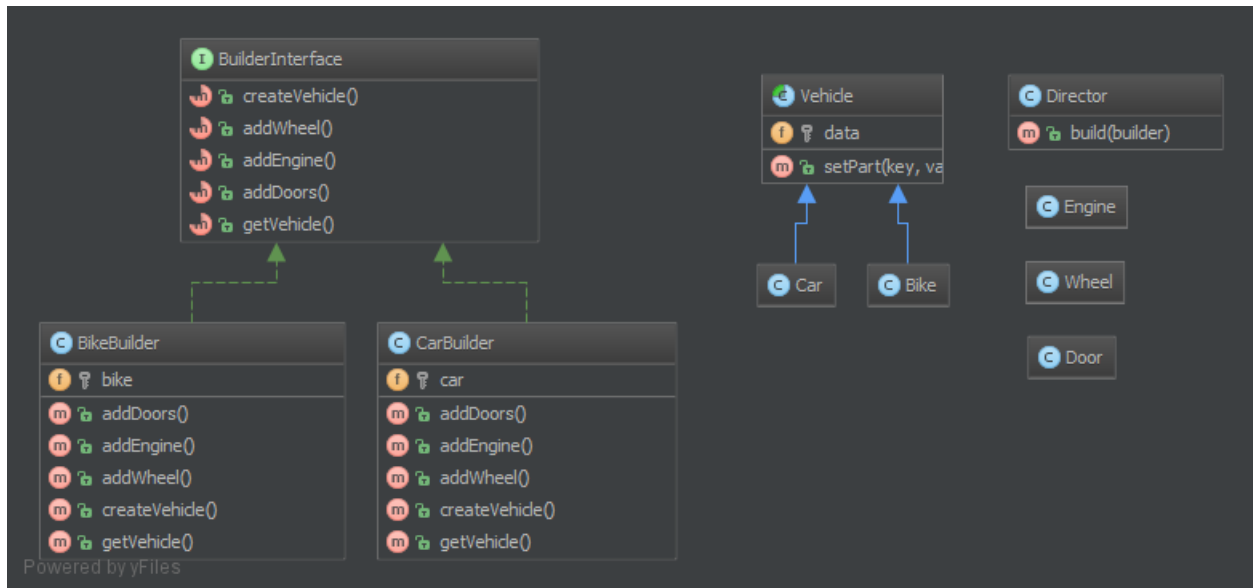
If you have a complex inheritance tree for objects, it is logical to have a complex inheritance tree for builders too.

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

Examples

- PHPUnit: Mock Builder

UML Diagram



Code

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

Director.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\Builder;
4
5  /**
6   * Director is part of the builder pattern. It knows the interface of the builder
7   * and builds a complex object with the help of the builder.
8   *
9   * You can also inject many builders instead of one to build more complex objects
10  */
11  class Director
12  {
13
14      /**
15       * The director don't know 'bout concrete part
16       *
17       * @param BuilderInterface $builder
18       *
19       * @return Parts\Vehicle
20       */
21      public function build(BuilderInterface $builder)
22      {
23          $builder->createVehicle();
24      }
25  }

```

```
24     $builder->addDoors ();
25     $builder->addEngine ();
26     $builder->addWheel ();
27
28     return $builder->getVehicle ();
29 }
30 }
```

BuilderInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder;
4
5 /**
6  *
7  */
8 interface BuilderInterface
9 {
10     /**
11      * @return mixed
12      */
13     public function createVehicle();
14
15     /**
16      * @return mixed
17      */
18     public function addWheel();
19
20     /**
21      * @return mixed
22      */
23     public function addEngine();
24
25     /**
26      * @return mixed
27      */
28     public function addDoors();
29
30     /**
31      * @return mixed
32      */
33     public function getVehicle();
34 }
```

BikeBuilder.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder;
4
5 /**
6  * BikeBuilder builds bike
7  */
8 class BikeBuilder implements BuilderInterface
9 {
10     /**
11      * @var Parts\Bike
12      */
```



```

13     protected $bike;
14
15     /**
16      * {@inheritdoc}
17      */
18     public function addDoors()
19     {
20     }
21
22     /**
23      * {@inheritdoc}
24      */
25     public function addEngine()
26     {
27         $this->bike->setPart('engine', new Parts\Engine());
28     }
29
30     /**
31      * {@inheritdoc}
32      */
33     public function addWheel()
34     {
35         $this->bike->setPart('forwardWheel', new Parts\Wheel());
36         $this->bike->setPart('rearWheel', new Parts\Wheel());
37     }
38
39     /**
40      * {@inheritdoc}
41      */
42     public function createVehicle()
43     {
44         $this->bike = new Parts\Bike();
45     }
46
47     /**
48      * {@inheritdoc}
49      */
50     public function getVehicle()
51     {
52         return $this->bike;
53     }
54 }

```

CarBuilder.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\Builder;
4
5  /**
6   * CarBuilder builds car
7   */
8  class CarBuilder implements BuilderInterface
9  {
10
11     /**
12      * @var Parts\Car
13      */
14     protected $car;

```

```
14
15     /**
16      * @return void
17      */
18     public function addDoors()
19     {
20         $this->car->setPart('rightdoor', new Parts\Door());
21         $this->car->setPart('leftDoor', new Parts\Door());
22     }
23
24     /**
25      * @return void
26      */
27     public function addEngine()
28     {
29         $this->car->setPart('engine', new Parts\Engine());
30     }
31
32     /**
33      * @return void
34      */
35     public function addWheel()
36     {
37         $this->car->setPart('wheelLF', new Parts\Wheel());
38         $this->car->setPart('wheelRF', new Parts\Wheel());
39         $this->car->setPart('wheelLR', new Parts\Wheel());
40         $this->car->setPart('wheelRR', new Parts\Wheel());
41     }
42
43     /**
44      * @return void
45      */
46     public function createVehicle()
47     {
48         $this->car = new Parts\Car();
49     }
50
51     /**
52      * @return Parts\Car
53      */
54     public function getVehicle()
55     {
56         return $this->car;
57     }
58 }
```

Parts/Vehicle.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder\Parts;
4
5 /**
6  * VehicleInterface is a contract for a vehicle
7  */
8 abstract class Vehicle
9 {
10     /**
11      * @var array
```

```
12     */
13     protected $data;
14
15     /**
16      * @param string $key
17      * @param mixed $value
18      */
19     public function setPart($key, $value)
20     {
21         $this->data[$key] = $value;
22     }
23 }
```

Parts/Bike.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder\Parts;
4
5 /**
6  * Bike is a bike
7  */
8 class Bike extends Vehicle
9 {
10 }
```

Parts/Car.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder\Parts;
4
5 /**
6  * Car is a car
7  */
8 class Car extends Vehicle
9 {
10 }
```

Parts/Engine.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder\Parts;
4
5 /**
6  * Class Engine
7  */
8 class Engine
9 {
10 }
```

Parts/Wheel.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Builder\Parts;
4
5 /**
6  * Class Wheel
```

```
7  */
8  class Wheel
9  {
10 }
```

Parts/Door.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Parts;
4
5  /**
6   * Class Door
7   */
8  class Door
9  {
10 }
```

Test

Tests/DirectorTest.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\Builder\Tests;
4
5  use DesignPatterns\Creational\Builder\Director;
6  use DesignPatterns\Creational\Builder\CarBuilder;
7  use DesignPatterns\Creational\Builder\BikeBuilder;
8  use DesignPatterns\Creational\Builder\BuilderInterface;
9
10 /**
11  * DirectorTest tests the builder pattern
12  */
13 class DirectorTest extends \PHPUnit_Framework_TestCase
14 {
15
16     protected $director;
17
18     protected function setUp()
19     {
20         $this->director = new Director();
21     }
22
23     public function getBuilder()
24     {
25         return array(
26             array(new CarBuilder()),
27             array(new BikeBuilder())
28         );
29     }
30
31     /**
32      * Here we test the build process. Notice that the client don't know
33      * anything about the concrete builder.
34      *
35      * @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

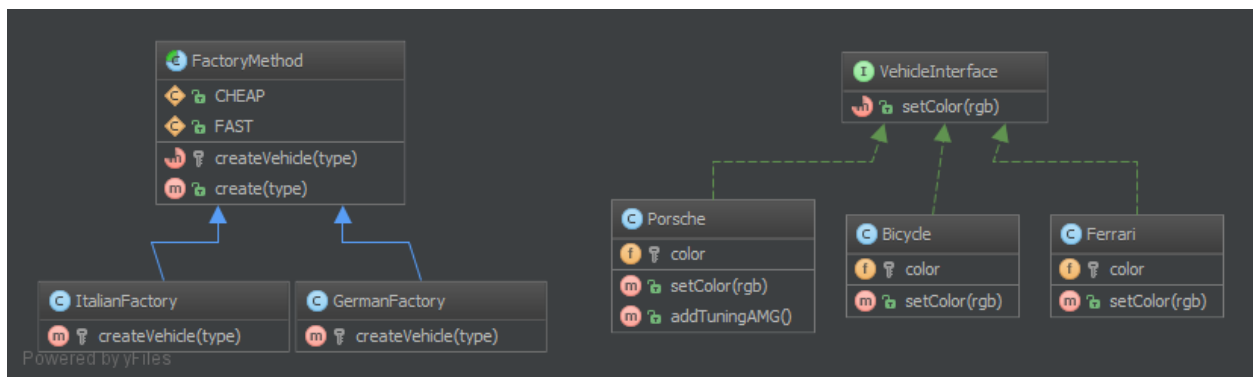
The good point over the SimpleFactory is you can subclass it to implement different ways to create objects

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;

```

```
12  const FAST = 2;
13
14  /**
15   * The children of the class must implement this method
16   *
17   * Sometimes this method can be public to get "raw" object
18   *
19   * @param string $type a generic type
20   *
21   * @return VehicleInterface a new vehicle
22   */
23  abstract protected function createVehicle($type);
24
25  /**
26   * Creates a new vehicle
27   *
28   * @param int $type
29   *
30   * @return VehicleInterface a new vehicle
31   */
32  public function create($type)
33  {
34      $obj = $this->createVehicle($type);
35      $obj->setColor("#f00");
36
37      return $obj;
38  }
39 }
```

ItalianFactory.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\FactoryMethod;
4
5  /**
6   * ItalianFactory is vehicle factory in Italy
7   */
8  class ItalianFactory extends FactoryMethod
9  {
10     /**
11      * {@inheritdoc}
12      */
13     protected function createVehicle($type)
14     {
15         switch ($type) {
16             case parent::CHEAP:
17                 return new Bicycle();
18                 break;
19             case parent::FAST:
20                 return new Ferrari();
21                 break;
22             default:
23                 throw new \InvalidArgumentException("$type is not a valid vehicle");
24         }
25     }
26 }
```

GermanFactory.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\FactoryMethod;
4
5 /**
6  * GermanFactory is a vehicle factory in Germany
7  */
8 class GermanFactory extends FactoryMethod
9 {
10     /**
11      * {@inheritdoc}
12      */
13     protected function createVehicle($type)
14     {
15         switch ($type) {
16             case parent::CHEAP:
17                 return new Bicycle();
18                 break;
19             case parent::FAST:
20                 $obj = new Porsche();
21                 // we can specialize the way we want some concrete Vehicle since
22                 // we know the class
23                 $obj->addTuningAMG();
24
25                 return $obj;
26                 break;
27             default:
28                 throw new \InvalidArgumentException("$type is not a valid vehicle");
29         }
30     }
31 }

```

VehicleInterface.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\FactoryMethod;
4
5 /**
6  * VehicleInterface is a contract for a vehicle
7  */
8 interface VehicleInterface
9 {
10     /**
11      * sets the color of the vehicle
12      *
13      * @param string $rgb
14      */
15     public function setColor($rgb);
16 }

```

Porsche.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\FactoryMethod;
4
5 /**
6  * Porsche is a german car

```

```
7  */
8  class Porsche implements VehicleInterface
9  {
10     /**
11      * @var string
12      */
13     protected $color;
14
15     /**
16      * @param string $rgb
17      */
18     public function setColor($rgb)
19     {
20         $this->color = $rgb;
21     }
22
23     /**
24      * although tuning by AMG is only offered for Mercedes Cars,
25      * this is a valid coding example ...
26      */
27     public function addTuningAMG()
28     {
29     }
30 }
```

Bicycle.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\FactoryMethod;
4
5  /**
6   * Bicycle is a bicycle
7   */
8  class Bicycle implements VehicleInterface
9  {
10     /**
11      * @var string
12      */
13     protected $color;
14
15     /**
16      * sets the color of the bicycle
17      *
18      * @param string $rgb
19      */
20     public function setColor($rgb)
21     {
22         $this->color = $rgb;
23     }
24 }
```

Ferrari.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\FactoryMethod;
4
5  /**
```



```

6  * Ferrari is a italian car
7  */
8  class Ferrari implements VehicleInterface
9  {
10     /**
11      * @var string
12      */
13     protected $color;
14
15     /**
16      * @param string $rgb
17      */
18     public function setColor($rgb)
19     {
20         $this->color = $rgb;
21     }
22 }

```

Test

Tests/FactoryMethodTest.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\FactoryMethod\Tests;
4
5  use DesignPatterns\Creational\FactoryMethod\FactoryMethod;
6  use DesignPatterns\Creational\FactoryMethod\GermanFactory;
7  use DesignPatterns\Creational\FactoryMethod\ItalianFactory;
8
9  /**
10   * FactoryMethodTest tests the factory method pattern
11   */
12  class FactoryMethodTest extends \PHPUnit_Framework_TestCase
13  {
14
15     protected $type = array(
16         FactoryMethod::CHEAP,
17         FactoryMethod::FAST
18     );
19
20     public function getShop()
21     {
22         return array(
23             array(new GermanFactory()),
24             array(new ItalianFactory())
25         );
26     }
27
28     /**
29      * @dataProvider getShop
30      */
31     public function testCreation(FactoryMethod $shop)
32     {
33         // this test method acts as a client for the factory. We don't care
34         // about the factory, all we know is it can produce vehicle
35         foreach ($this->type as $oneType) {

```

```
36         $vehicle = $shop->create($oneType);
37         $this->assertInstanceOf('DesignPatterns\Creational\FactoryMethod\VehicleInterface', $vehicle);
38     }
39 }
40
41 /**
42  * @dataProvider getShop
43  * @expectedException \InvalidArgumentException
44  * @expectedExceptionMessage spaceship is not a valid vehicle
45  */
46 public function testUnknownType(FactoryMethod $shop)
47 {
48     $shop->create('spaceship');
49 }
50 }
```

1.1.4 Multiton

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

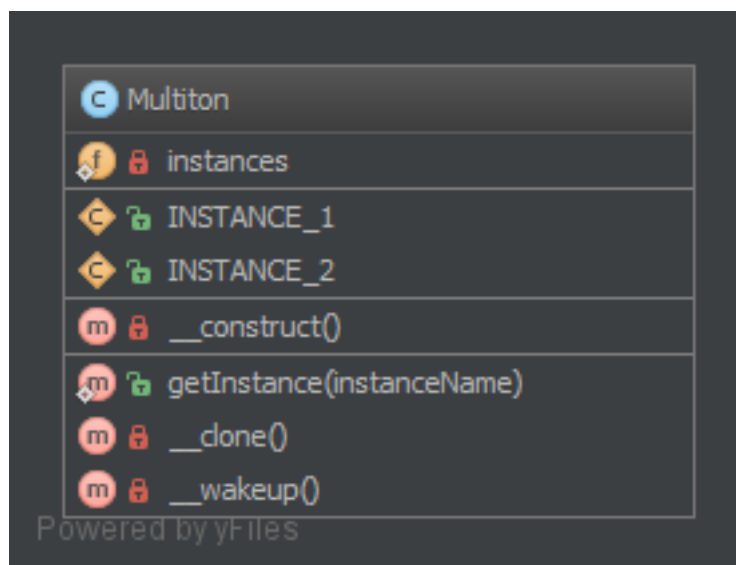
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Creational\Multiton;
4
5  /**
6   * class Multiton
7   */
8  class Multiton
9  {
10     /**
11      *
12      * the first instance
13      */
14     const INSTANCE_1 = '1';
15
16     /**
17      *
18      * the second instance
19      */
20     const INSTANCE_2 = '2';
21
22     /**
23      * holds the named instances
24      *
25      * @var array
26      */
27     private static $instances = array();
28
29     /**
30      * should not be called from outside: private!
31      *
32      */
33     private function __construct()
34     {
35     }
36
37     /**
38      * gets the instance with the given name, e.g. Multiton::INSTANCE_1
39      * uses lazy initialization
40      *
41      * @param string $instanceName
42      *
43      * @return Multiton
44      */
45     public static function getInstance($instanceName)
46     {
47         if (!array_key_exists($instanceName, self::$instances)) {
48             self::$instances[$instanceName] = new self();
49         }
50
51         return self::$instances[$instanceName];
52     }
53

```

```
54  /**
55   * prevent instance from being cloned
56   *
57   * @return void
58   */
59  private function __clone()
60  {
61  }
62
63  /**
64   * prevent instance from being unserialized
65   *
66   * @return void
67   */
68  private function __wakeup()
69  {
70  }
71 }
```

Test

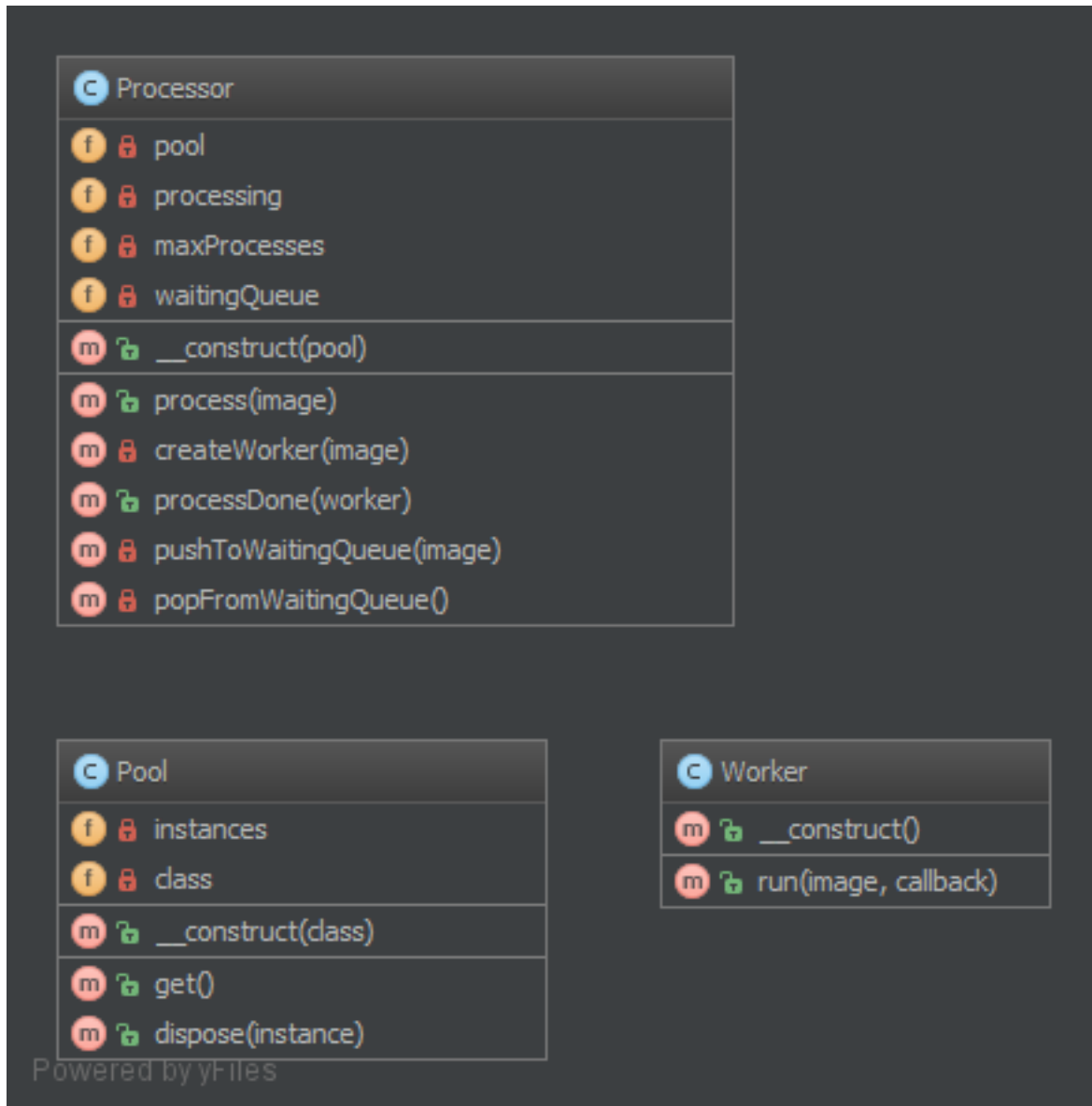
1.1.5 Pool

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

```

1  <?php
2
3  namespace DesignPatterns\Creational\Pool;
4
5  class Pool
6  {
7
8      private $instances = array();
  
```

```
9     private $class;
10
11     public function __construct($class)
12     {
13         $this->class = $class;
14     }
15
16     public function get()
17     {
18         if (count($this->instances) > 0) {
19             return array_pop($this->instances);
20         }
21
22         return new $this->class();
23     }
24
25     public function dispose($instance)
26     {
27         $this->instances[] = $instance;
28     }
29 }
```

Processor.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Pool;
4
5 class Processor
6 {
7
8     private $pool;
9     private $processing = 0;
10    private $maxProcesses = 3;
11    private $waitingQueue = [];
12
13    public function __construct(Pool $pool)
14    {
15        $this->pool = $pool;
16    }
17
18    public function process($image)
19    {
20        if ($this->processing++ < $this->maxProcesses) {
21            $this->createWorker($image);
22        } else {
23            $this->pushToWaitingQueue($image);
24        }
25    }
26
27    private function createWorker($image)
28    {
29        $worker = $this->pool->get();
30        $worker->run($image, array($this, 'processDone'));
31    }
32
33    public function processDone($worker)
34    {

```

```

35     $this->processing--;
36     $this->pool->dispose($worker);
37
38     if (count($this->waitingQueue) > 0) {
39         $this->createWorker($this->popFromWaitingQueue());
40     }
41 }
42
43 private function pushToWaitingQueue($image)
44 {
45     $this->waitingQueue[] = $image;
46 }
47
48 private function popFromWaitingQueue()
49 {
50     return array_pop($this->waitingQueue);
51 }
52 }

```

Worker.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\Pool;
4
5  class Worker
6  {
7
8      public function __construct()
9      {
10         // let's say that constructor does really expensive work...
11         // for example creates "thread"
12     }
13
14     public function run($image, array $callback)
15     {
16         // do something with $image...
17         // and when it's done, execute callback
18         call_user_func($callback, $this);
19     }
20 }

```

Test

Tests/PoolTest.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\Pool\Tests;
4
5  use DesignPatterns\Creational\Pool\Pool;
6
7  class PoolTest extends \PHPUnit_Framework_TestCase
8  {
9      public function testPool()
10     {
11         $pool = new Pool('DesignPatterns\Creational\Pool\Tests\TestWorker');

```

```
12     $worker = $pool->get();
13
14     $this->assertEquals(1, $worker->id);
15
16     $worker->id = 5;
17     $pool->dispose($worker);
18
19     $this->assertEquals(5, $pool->get()->id);
20     $this->assertEquals(1, $pool->get()->id);
21 }
22 }
```

Tests/TestWorker.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Pool\Tests;
4
5 class TestWorker
6 {
7     public $id = 1;
8 }
```

1.1.6 Prototype

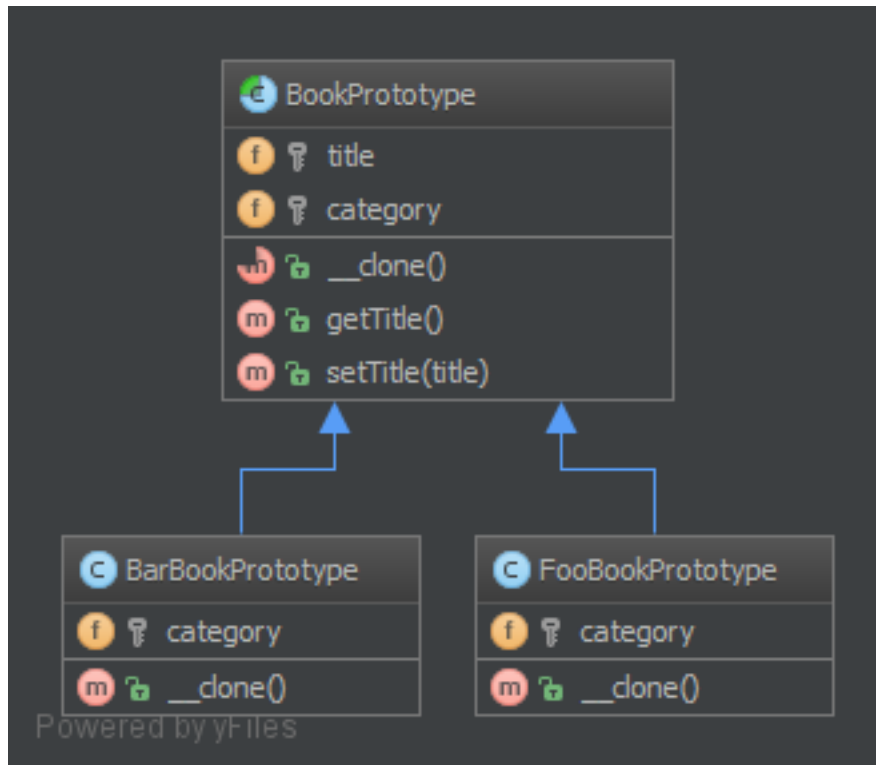
Purpose

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

```

1 <?php
2
3 namespace DesignPatterns\Creational\Prototype;
4
5 $fooPrototype = new FooBookPrototype();
6 $barPrototype = new BarBookPrototype();
7
8 // now lets say we need 10,000 books of foo and 5,000 of bar ...
9 for ($i = 0; $i < 10000; $i++) {
10     $book = clone $fooPrototype;
11     $book->setTitle('Foo Book No ' . $i);
12 }
13
14 for ($i = 0; $i < 5000; $i++) {
15     $book = clone $barPrototype;
16     $book->setTitle('Bar Book No ' . $i);
17 }
  
```

BookPrototype.php

```

1 <?php
2
3 namespace DesignPatterns\Creational\Prototype;
  
```

```
4
5 /**
6  * class BookPrototype
7  */
8 abstract class BookPrototype
9 {
10     /**
11      * @var string
12      */
13     protected $title;
14
15     /**
16      * @var string
17      */
18     protected $category;
19
20     /**
21      * @abstract
22      * @return void
23      */
24     abstract public function __clone();
25
26     /**
27      * @return string
28      */
29     public function getTitle()
30     {
31         return $this->title;
32     }
33
34     /**
35      * @param string $title
36      */
37     public function setTitle($title)
38     {
39         $this->title = $title;
40     }
41 }
```

BarBookPrototype.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\Prototype;
4
5 /**
6  * Class BarBookPrototype
7  */
8 class BarBookPrototype extends BookPrototype
9 {
10     /**
11      * @var string
12      */
13     protected $category = 'Bar';
14
15     /**
16      * empty clone
17      */
18     public function __clone()
```

```
19     {
20     }
21 }
```

FooBookPrototype.php

```
1  <?php
2
3  namespace DesignPatterns\Creational\Prototype;
4
5  /**
6   * Class FooBookPrototype
7   */
8  class FooBookPrototype extends BookPrototype
9  {
10     protected $category = 'Foo';
11
12     /**
13      * empty clone
14      */
15     public function __clone()
16     {
17     }
18 }
```

Test

1.1.7 Simple Factory

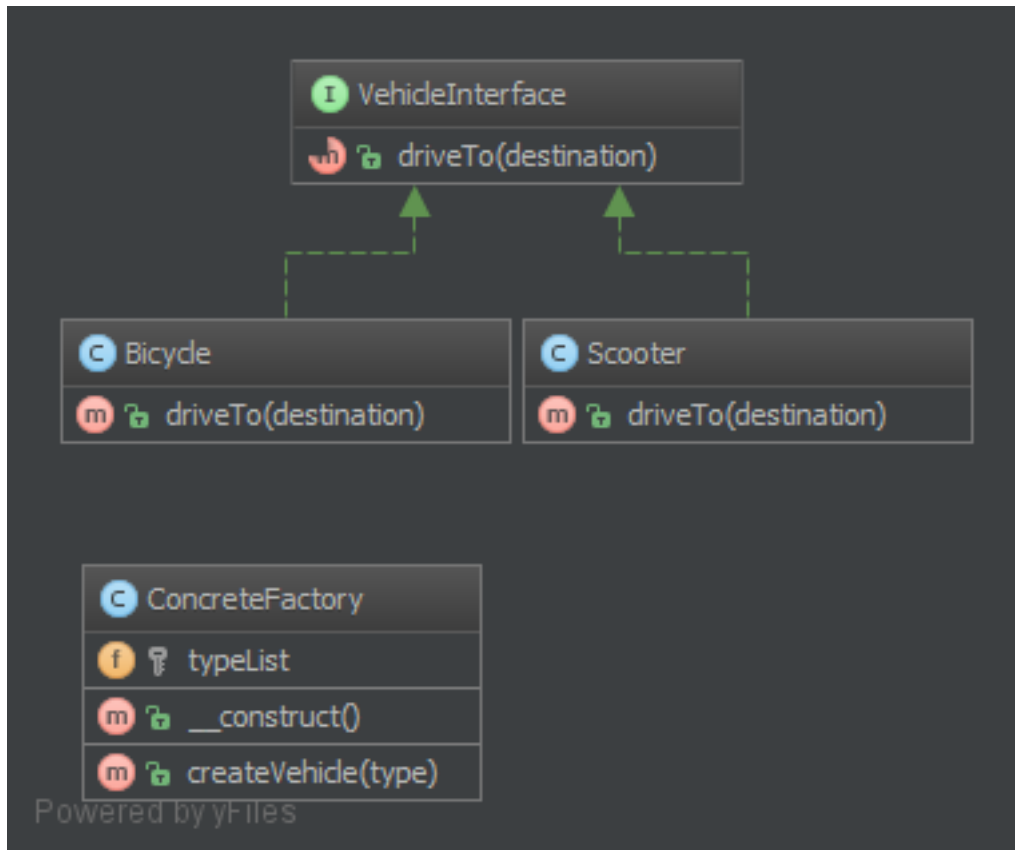
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Creational\SimpleFactory;
4
5  /**
6   * class ConcreteFactory
7   */
8  class ConcreteFactory
9  {
10     /**
11      * @var array
12      */
13     protected $typeList;
14
15     /**
16      * You can imagine to inject your own type list or merge with
17      * the default ones...
18      */
19     public function __construct()
20     {
  
```

```

21         $this->typeList = array(
22             'bicycle' => __NAMESPACE__ . '\Bicycle',
23             'other' => __NAMESPACE__ . '\Scooter'
24         );
25     }
26
27     /**
28      * Creates a vehicle
29      *
30      * @param string $type a known type key
31      *
32      * @return VehicleInterface a new instance of VehicleInterface
33      * @throws \InvalidArgumentException
34      */
35     public function createVehicle($type)
36     {
37         if (!array_key_exists($type, $this->typeList)) {
38             throw new \InvalidArgumentException("$type is not valid vehicle");
39         }
40         $className = $this->typeList[$type];
41
42         return new $className();
43     }
44 }

```

VehicleInterface.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\SimpleFactory;
4
5  /**
6   * VehicleInterface is a contract for a vehicle
7   */
8  interface VehicleInterface
9  {
10     /**
11      * @param mixed $destination
12      *
13      * @return mixed
14      */
15     public function driveTo($destination);
16 }

```

Bicycle.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\SimpleFactory;
4
5  /**
6   * Bicycle is a bicycle
7   */
8  class Bicycle implements VehicleInterface
9  {
10     /**
11      * @param mixed $destination
12      *
13      * @return mixed/void

```

```
14     */
15     public function driveTo($destination)
16     {
17     }
18 }
```

Scooter.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\SimpleFactory;
4
5 /**
6  * Scooter is a Scooter
7  */
8 class Scooter implements VehicleInterface
9 {
10     /**
11      * @param mixed $destination
12      */
13     public function driveTo($destination)
14     {
15     }
16 }
```

Test

Tests/SimpleFactoryTest.php

```
1 <?php
2
3 namespace DesignPatterns\Creational\SimpleFactory\Tests;
4
5 use DesignPatterns\Creational\SimpleFactory\ConcreteFactory;
6
7 /**
8  * SimpleFactoryTest tests the Simple Factory pattern
9  */
10 class SimpleFactoryTest extends \PHPUnit_Framework_TestCase
11 {
12
13     protected $factory;
14
15     protected function setUp()
16     {
17         $this->factory = new ConcreteFactory();
18     }
19
20     public function getType()
21     {
22         return array(
23             array('bicycle'),
24             array('other')
25         );
26     }
27
28     /**
```

```

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!

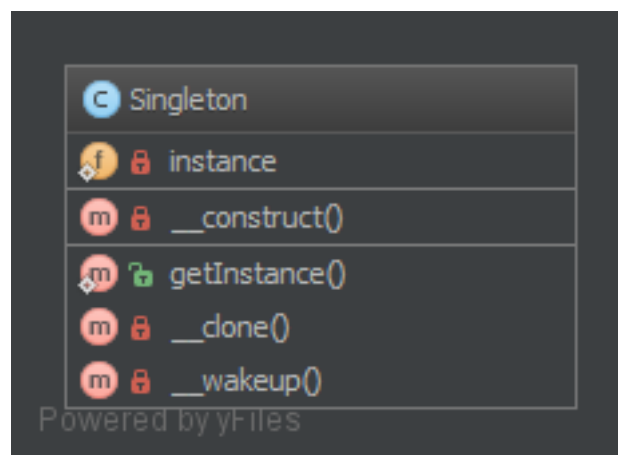
Purpose

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

```
1 <?php
2
3 namespace DesignPatterns\Creational\Singleton;
4
5 /**
6  * class Singleton
7  */
8 class Singleton
9 {
10     /**
11      * @var Singleton reference to singleton instance
12      */
13     private static $instance;
14
15     /**
16      * gets the instance via lazy initialization (created on first usage)
17      *
18      * @return self
19      */
20     public static function getInstance()
21     {
22         if (null === static::$instance) {
23             static::$instance = new static;
24         }
25
26         return static::$instance;
27     }
28
29     /**
30      * is not allowed to call from outside: private!
31      *
32      */
33     private function __construct()
34     {
35     }
36
37     /**
38      * prevent the instance from being cloned
39      *
40      * @return void
41      */
42     private function __clone()
43     {
44     }
45
46     /**
47      * prevent from being unserialized
48      *
49      * @return void
50      */
51     private function __wakeup()
52     {
53     }
```


54 }

Test

Tests/SingletonTest.php

```

1  <?php
2
3  namespace DesignPatterns\Creational\Singleton\Tests;
4
5  use DesignPatterns\Creational\Singleton\Singleton;
6
7  /**
8   * SingletonTest tests the singleton pattern
9   */
10 class SingletonTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testUniqueness()
14     {
15         $firstCall = Singleton::getInstance();
16         $this->assertInstanceOf('DesignPatterns\Creational\Singleton\Singleton', $firstCall);
17         $secondCall = Singleton::getInstance();
18         $this->assertSame($firstCall, $secondCall);
19     }
20
21     public function testNoConstructor()
22     {
23         $obj = Singleton::getInstance();
24
25         $refl = new \ReflectionObject($obj);
26         $meth = $refl->getMethod('__construct');
27         $this->assertTrue($meth->isPrivate());
28     }
29 }

```

1.1.9 Static Factory

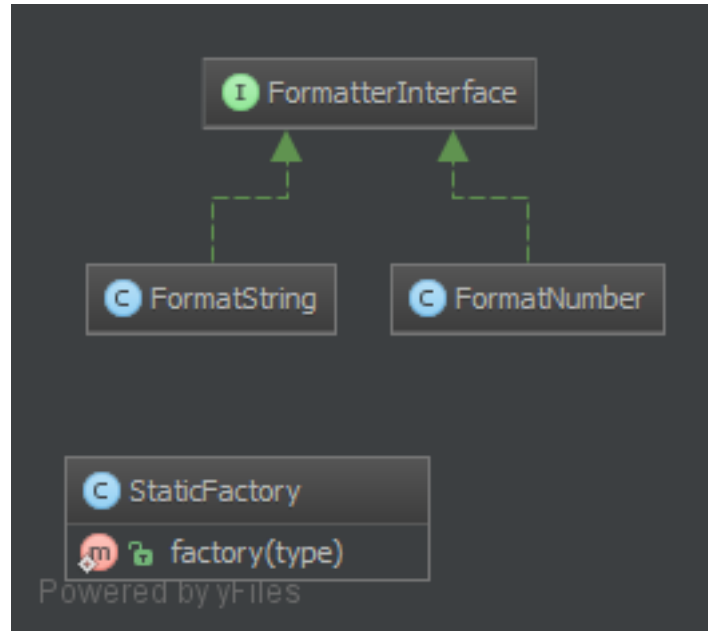
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Creational\StaticFactory;
4
5  /**
6   * Note1: Remember, static => global => evil
7   * Note2: Cannot be subclassed or mock-upped or have multiple different instances
8   */
9  class StaticFactory
10 {
11     /**
12      * the parametrized function to get create an instance
13      *
14      * @param string $type
15      *
16      * @static
17      *
18      * @throws \InvalidArgumentException
19      * @return FormatterInterface
20      */
21     public static function factory($type)
22     {
23         $className = __NAMESPACE__ . '\Format' . ucfirst($type);
24
25         if (!class_exists($className)) {
26             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
```

```
11 class StaticFactoryTest extends \PHPUnit_Framework_TestCase
12 {
13
14     public function getTypeList()
15     {
16         return array(
17             array('string'),
18             array('number')
19         );
20     }
21
22     /**
23      * @dataProvider getTypeList
24      */
25     public function testCreation($type)
26     {
27         $obj = StaticFactory::factory($type);
28         $this->assertInstanceOf('DesignPatterns\Creational\StaticFactory\FormatterInterface', $obj);
29     }
30
31     /**
32      * @expectedException InvalidArgumentException
33      */
34     public function testException()
35     {
36         StaticFactory::factory("");
37     }
38 }
```

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

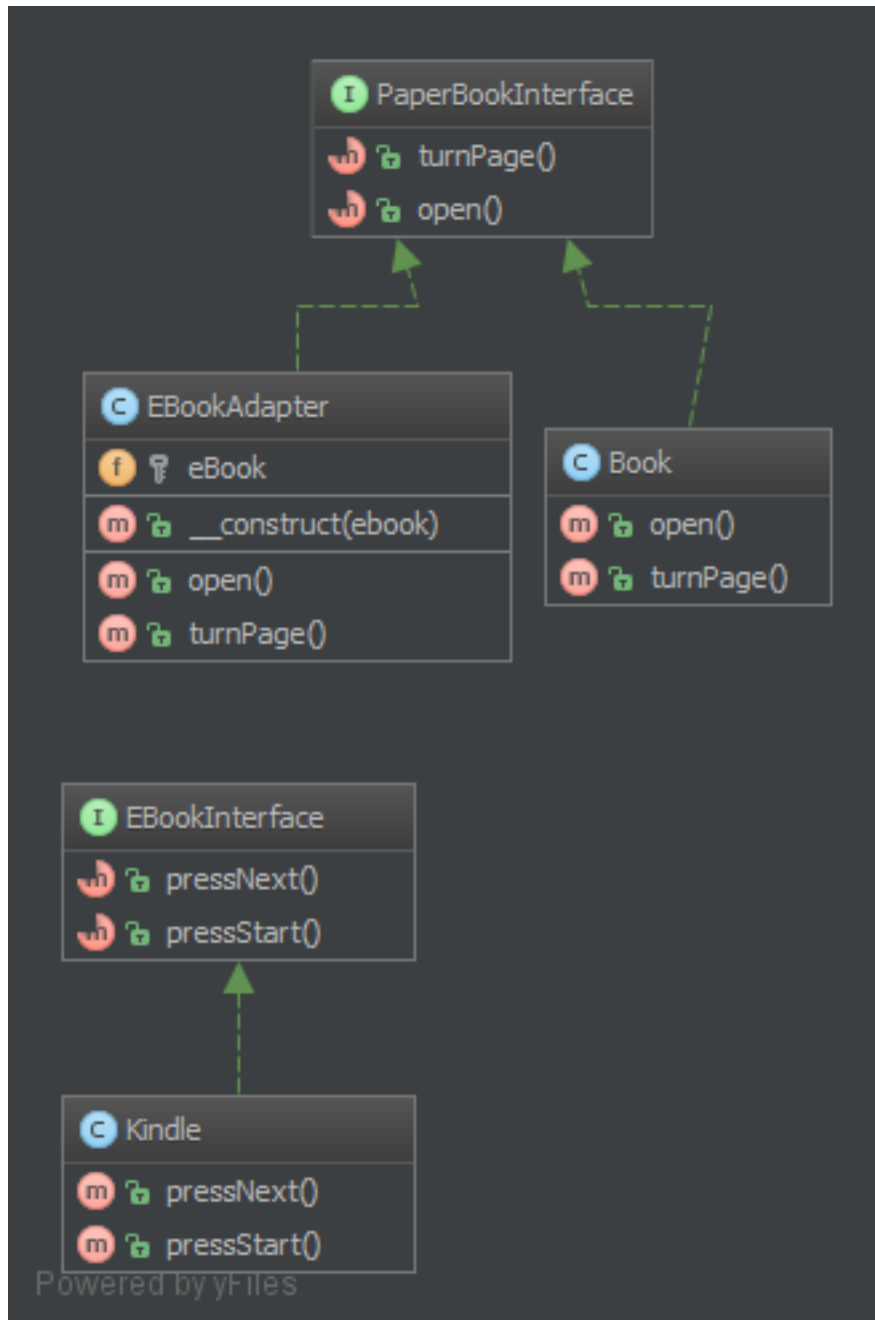
Purpose

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 webservises 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;
```

```
4
5  /**
6   * PaperBookInterface is a contract for a book
7   */
8  interface PaperBookInterface
9  {
10     /**
11      * method to turn pages
12      *
13      * @return mixed
14      */
15     public function turnPage();
16
17     /**
18      * method to open the book
19      *
20      * @return mixed
21      */
22     public function open();
23 }
```

Book.php

```
1  <?php
2
3  namespace DesignPatterns\Structural\Adapter;
4
5  /**
6   * Book is a concrete and standard paper book
7   */
8  class Book implements PaperBookInterface
9  {
10     /**
11      * {@inheritdoc}
12      */
13     public function open()
14     {
15     }
16
17     /**
18      * {@inheritdoc}
19      */
20     public function turnPage()
21     {
22     }
23 }
```

EBookAdapter.php

```
1  <?php
2
3  namespace DesignPatterns\Structural\Adapter;
4
5  /**
6   * EBookAdapter is an adapter to fit an e-book like a paper book
7   *
8   * This is the adapter here. Notice it implements PaperBookInterface,
9   * therefore you don't have to change the code of the client which using paper book.
10  */
```

```

11 class EBookAdapter implements PaperBookInterface
12 {
13     /**
14      * @var EBookInterface
15      */
16     protected $eBook;
17
18     /**
19      * Notice the constructor, it "wraps" an electronic book
20      *
21      * @param EBookInterface $ebook
22      */
23     public function __construct(EBookInterface $ebook)
24     {
25         $this->eBook = $ebook;
26     }
27
28     /**
29      * This class makes the proper translation from one interface to another
30      */
31     public function open()
32     {
33         $this->eBook->pressStart();
34     }
35
36     /**
37      * turns pages
38      */
39     public function turnPage()
40     {
41         $this->eBook->pressNext();
42     }
43 }

```

EBookInterface.php

```

1 <?php
2
3 namespace DesignPatterns\Structural\Adapter;
4
5 /**
6  * EBookInterface is a contract for an electronic book
7  */
8 interface EBookInterface
9 {
10     /**
11      * go to next page
12      *
13      * @return mixed
14      */
15     public function pressNext();
16
17     /**
18      * start the book
19      *
20      * @return mixed
21      */
22     public function pressStart();

```

23

}

Kindle.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Adapter;
4
5 /**
6  * Kindle is a concrete electronic book
7  */
8 class Kindle implements EBookInterface
9 {
10     /**
11      * {@inheritdoc}
12      */
13     public function pressNext()
14     {
15     }
16
17     /**
18      * {@inheritdoc}
19      */
20     public function pressStart()
21     {
22     }
23 }
```

Test

Tests/AdapterTest.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Adapter\Tests;
4
5 use DesignPatterns\Structural\Adapter\EBookAdapter;
6 use DesignPatterns\Structural\Adapter\Kindle;
7 use DesignPatterns\Structural\Adapter\PaperBookInterface;
8 use DesignPatterns\Structural\Adapter\Book;
9
10 /**
11  * AdapterTest shows the use of an adapted e-book that behave like a book
12  * You don't have to change the code of your client
13  */
14 class AdapterTest extends \PHPUnit_Framework_TestCase
15 {
16     /**
17      * @return array
18      */
19     public function getBook()
20     {
21         return array(
22             array(new Book()),
23             // we build a "wrapped" electronic book in the adapter
24             array(new EBookAdapter(new Kindle()))
25         );
26     }
27 }
```



```
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         $book->open();
39         $book->turnPage();
40     }
41 }
```

1.2.2 Bridge

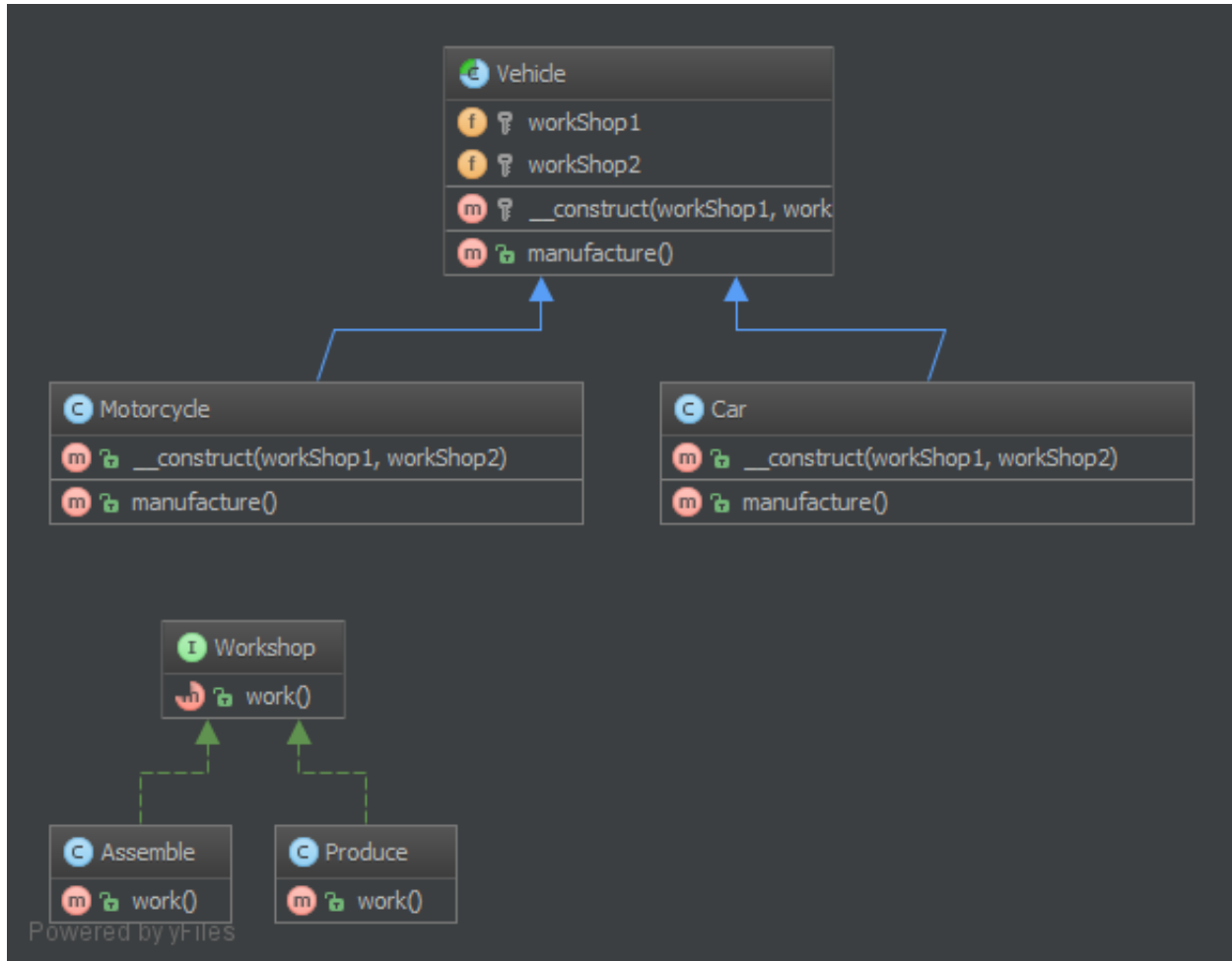
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Structural\Bridge;
4
5  /**
6   * Implementer
7   */
8  interface Workshop
9  {
10
11     public function work();
12 }
  
```

Assemble.php

```

1 <?php
2
3 namespace DesignPatterns\Structural\Bridge;
4
5 class Assemble implements Workshop
6 {
7
8     public function work()
9     {
10         print 'Assembled';
11     }
12 }

```

Produce.php

```

1 <?php
2
3 namespace DesignPatterns\Structural\Bridge;
4
5 /**
6  * Concrete Implementation
7  */
8 class Produce implements Workshop
9 {
10
11     public function work()
12     {
13         print 'Produced ';
14     }
15 }

```

Vehicle.php

```

1 <?php
2
3 namespace DesignPatterns\Structural\Bridge;
4
5 /**
6  * Abstraction
7  */
8 abstract class Vehicle
9 {
10
11     protected $workShop1;
12     protected $workShop2;
13
14     protected function __construct(Workshop $workShop1, Workshop $workShop2)
15     {
16         $this->workShop1 = $workShop1;
17         $this->workShop2 = $workShop2;
18     }
19
20     public function manufacture()
21     {
22     }
23 }

```

Motorcycle.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Bridge;
4
5 /**
6  * Refined Abstraction
7  */
8 class Motorcycle extends Vehicle
9 {
10
11     public function __construct(Workshop $workShop1, Workshop $workShop2)
12     {
13         parent::__construct($workShop1, $workShop2);
14     }
15
16     public function manufacture()
17     {
18         print 'Motorcycle ';
19         $this->workShop1->work();
20         $this->workShop2->work();
21     }
22 }
```

Car.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Bridge;
4
5 /**
6  * Refined Abstraction
7  */
8 class Car extends Vehicle
9 {
10
11     public function __construct(Workshop $workShop1, Workshop $workShop2)
12     {
13         parent::__construct($workShop1, $workShop2);
14     }
15
16     public function manufacture()
17     {
18         print 'Car ';
19         $this->workShop1->work();
20         $this->workShop2->work();
21     }
22 }
```

Test

Tests/BridgeTest.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Bridge\Tests;
4
5 use DesignPatterns\Structural\Bridge\Assemble;
```

```
6 use DesignPatterns\Structural\Bridge\Car;
7 use DesignPatterns\Structural\Bridge\Motorcycle;
8 use DesignPatterns\Structural\Bridge\Produce;
9
10 class BridgeTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testCar()
14     {
15         $vehicle = new Car(new Produce(), new Assemble());
16         $this->expectOutputString('Car Produced Assembled');
17         $vehicle->manufacture();
18     }
19
20     public function testMotorcycle()
21     {
22         $vehicle = new Motorcycle(new Produce(), new Assemble());
23         $this->expectOutputString('Motorcycle Produced Assembled');
24         $vehicle->manufacture();
25     }
26 }
```

1.2.3 Composite

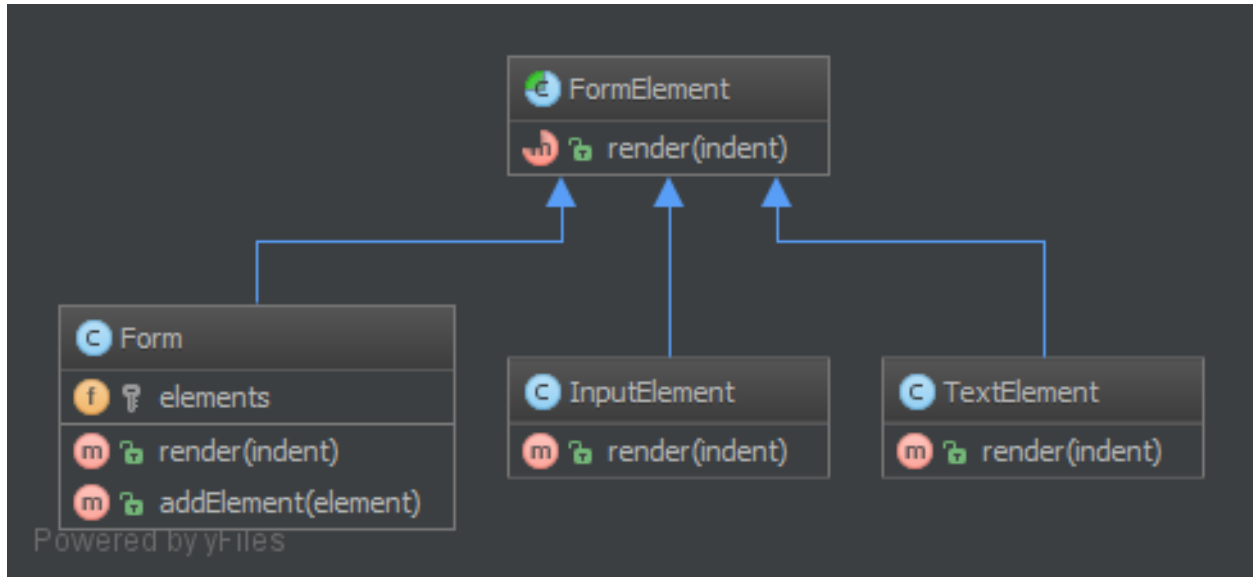
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Structural\Composite;
4
5  /**
6   * Class FormElement
7   */
8  abstract class FormElement
9  {
10     /**
11      * renders the elements' code
12      *
13      * @param int $indent
14      *
15      * @return mixed
16      */
17     abstract public function render($indent = 0);
18 }
  
```

Form.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Composite;
4
5  /**
6   * The composite node MUST extend the component contract. This is mandatory for building
7   * a tree of components.
8   */
  
```

```

9  class Form extends FormElement
10 {
11     /**
12      * @var array/FormElement[]
13      */
14     protected $elements;
15
16     /**
17      * runs through all elements and calls render() on them, then returns the complete representation
18      * of the form
19      *
20      * from the outside, one will not see this and the form will act like a single object instance
21      *
22      * @param int $indent
23      *
24      * @return string
25      */
26     public function render($indent = 0)
27     {
28         $formCode = '';
29
30         foreach ($this->elements as $element) {
31             $formCode .= $element->render($indent + 1) . PHP_EOL;
32         }
33
34         return $formCode;
35     }
36
37     /**
38      * @param FormElement $element
39      */
40     public function addElement(FormElement $element)
41     {
42         $this->elements[] = $element;
43     }
44 }

```

InputElement.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Composite;
4
5  /**
6   * Class InputElement
7   */
8  class InputElement extends FormElement
9  {
10     /**
11      * renders the input element HTML
12      *
13      * @param int $indent
14      *
15      * @return mixed/string
16      */
17     public function render($indent = 0)
18     {
19         return str_repeat(' ', $indent) . '<input type="text" />';
20     }

```

21

}

TextElement.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Composite;
4
5 /**
6  * Class TextElement
7  */
8 class TextElement extends FormElement
9 {
10     /**
11      * renders the text element
12      *
13      * @param int $indent
14      *
15      * @return mixed/string
16      */
17     public function render($indent = 0)
18     {
19         return str_repeat(' ', $indent) . 'this is a text element';
20     }
21 }
```

Test

Tests/CompositeTest.php

```
1 <?php
2
3 namespace DesignPatterns\Test\Composite\Tests;
4
5 use DesignPatterns\Structural\Composite;
6
7 /**
8  * FormTest tests the composite pattern on Form
9  */
10 class CompositeTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testRender()
14     {
15         $form = new Composite\Form();
16         $form->addElement(new Composite\TextElement());
17         $form->addElement(new Composite\InputElement());
18         $embed = new Composite\Form();
19         $embed->addElement(new Composite\TextElement());
20         $embed->addElement(new Composite\InputElement());
21         $form->addElement($embed); // here we have a embedded form (like SF2 does)
22
23         $this->assertRegExp('#^\s{4}#m', $form->render());
24     }
25
26     /**
27      * 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

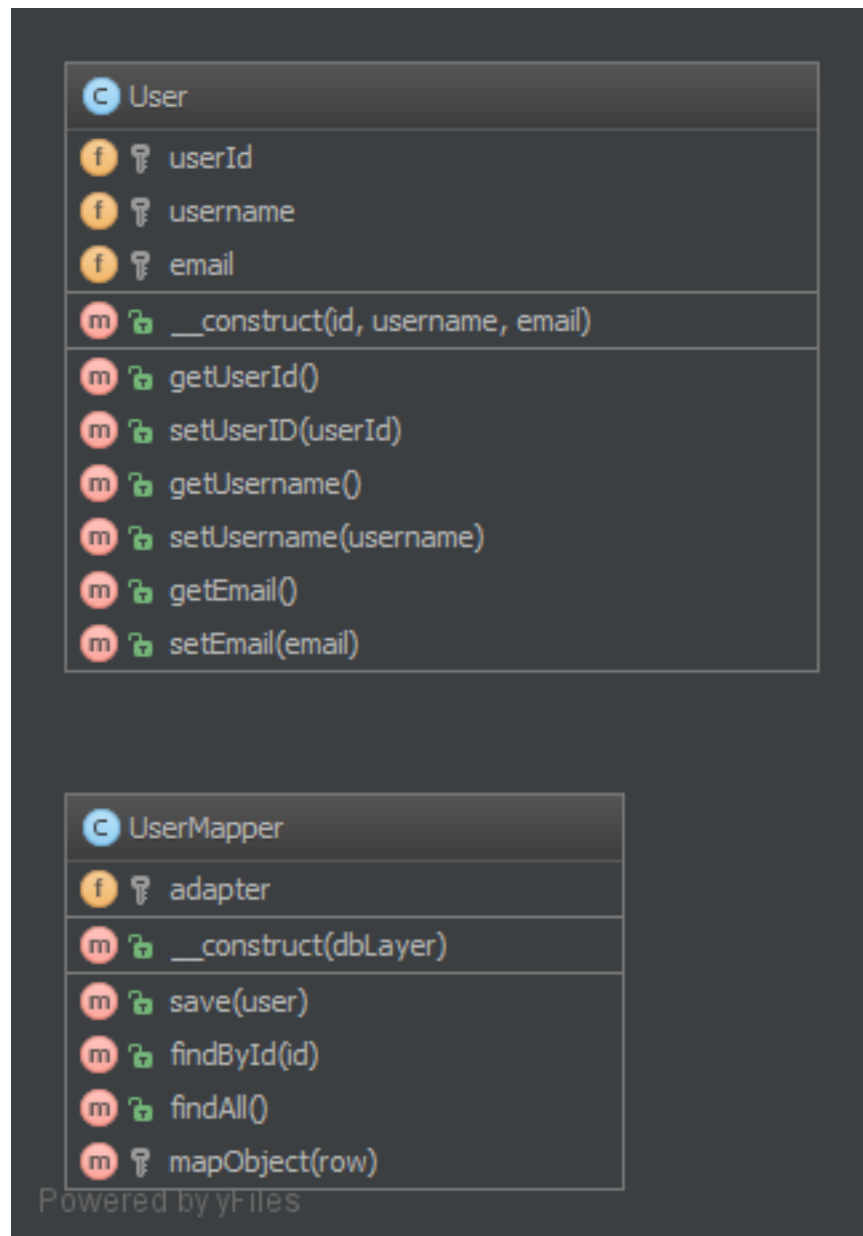
A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). The goal of the pattern is to keep the in memory representation and the persistent data store independent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

```
1 <?php
2
3 namespace DesignPatterns\Structural\DataMapper;
4
5 /**
6  * DataMapper pattern
```

```
7  *
8  * This is our representation of a DataBase record in the memory (Entity)
9  *
10 * Validation would also go in this object
11 *
12 */
13 class User
14 {
15     /**
16      * @var int
17      */
18     protected $userId;
19
20     /**
21      * @var string
22      */
23     protected $username;
24
25     /**
26      * @var string
27      */
28     protected $email;
29
30     /**
31      * @param null $id
32      * @param null $username
33      * @param null $email
34      */
35     public function __construct($id = null, $username = null, $email = null)
36     {
37         $this->userId = $id;
38         $this->username = $username;
39         $this->email = $email;
40     }
41
42     /**
43      * @return int
44      */
45     public function getUserId()
46     {
47         return $this->userId;
48     }
49
50     /**
51      * @param int $userId
52      */
53     public function setUserID($userId)
54     {
55         $this->userId = $userId;
56     }
57
58     /**
59      * @return string
60      */
61     public function getUsername()
62     {
63         return $this->username;
64     }
65 }
```

```
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     {
```

```

32      /* $data keys should correspond to valid Table columns on the Database */
33      $data = array(
34          'userid' => $user->getUserId(),
35          'username' => $user->getUsername(),
36          'email' => $user->getEmail(),
37      );
38
39      /* if no ID specified create new user else update the one in the Database */
40      if (null === ($id = $user->getUserId())) {
41          unset($data['userid']);
42          $this->adapter->insert($data);
43
44          return true;
45      } else {
46          $this->adapter->update($data, array('userid = ?' => $id));
47
48          return true;
49      }
50  }
51
52  /**
53   * finds a user from Database based on ID and returns a User object located
54   * in memory
55   *
56   * @param int $id
57   *
58   * @throws \InvalidArgumentException
59   * @return User
60   */
61  public function findById($id)
62  {
63      $result = $this->adapter->find($id);
64
65      if (0 == count($result)) {
66          throw new \InvalidArgumentException("User #{$id} not found");
67      }
68      $row = $result->current();
69
70      return $this->mapObject($row);
71  }
72
73  /**
74   * fetches an array from Database and returns an array of User objects
75   * located in memory
76   *
77   * @return array
78   */
79  public function findAll()
80  {
81      $resultSet = $this->adapter->findAll();
82      $entries = array();
83
84      foreach ($resultSet as $row) {
85          $entries[] = $this->mapObject($row);
86      }
87
88      return $entries;
89  }

```

```
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

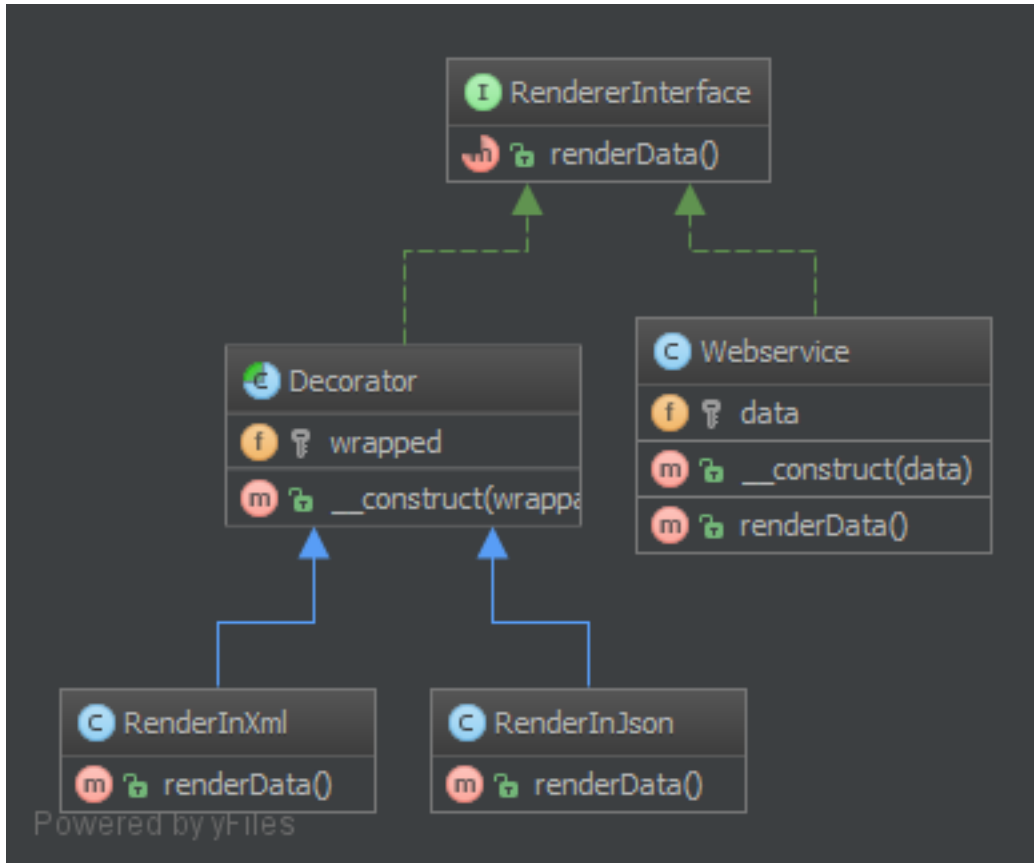
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Structural\Decorator;
4
5  /**
6   * Class RendererInterface
7   */
8  interface RendererInterface
9  {
10     /**
11      * render data
12      *
13      * @return mixed
14      */
15     public function renderData();
16 }

```

WebService.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Decorator;
4
5 /**
6  * Class Webservice
7  */
8 class Webservice implements RendererInterface
9 {
10     /**
11      * @var mixed
12      */
13     protected $data;
14
15     /**
16      * @param mixed $data
17      */
18     public function __construct($data)
19     {
20         $this->data = $data;
21     }
22
23     /**
24      * @return string
25      */
26     public function renderData()
27     {
28         return $this->data;
29     }
30 }
```

Decorator.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Decorator;
4
5 /**
6  * the Decorator MUST implement the RendererInterface contract, this is the key-feature
7  * of this design pattern. If not, this is no longer a Decorator but just a dumb
8  * wrapper.
9  */
10
11 /**
12  * class Decorator
13  */
14 abstract class Decorator implements RendererInterface
15 {
16     /**
17      * @var RendererInterface
18      */
19     protected $wrapped;
20
21     /**
22      * You must type-hint the wrapped component :
23      * It ensures you can call renderData() in the subclasses !
24      *
25      * @param RendererInterface $wrappable
26      */
27 }
```

```

27     public function __construct(RendererInterface $wrappable)
28     {
29         $this->wrapped = $wrappable;
30     }
31 }

```

RenderInXml.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Decorator;
4
5  /**
6   * Class RenderInXml
7   */
8  class RenderInXml extends Decorator
9  {
10     /**
11      * render data as XML
12      *
13      * @return mixed|string
14      */
15     public function renderData()
16     {
17         $output = $this->wrapped->renderData();
18
19         // do some fancy conversion to xml from array ...
20
21         $doc = new \DOMDocument();
22
23         foreach ($output as $key => $val) {
24             $doc->appendChild($doc->createElement($key, $val));
25         }
26
27         return $doc->saveXML();
28     }
29 }

```

RenderInJson.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Decorator;
4
5  /**
6   * Class RenderInJson
7   */
8  class RenderInJson extends Decorator
9  {
10     /**
11      * render data as JSON
12      *
13      * @return mixed|string
14      */
15     public function renderData()
16     {
17         $output = $this->wrapped->renderData();
18
19         return json_encode($output);

```

```
20     }
21 }
```

Test

Tests/DecoratorTest.php

```
1  <?php
2
3  namespace DesignPatterns\Structural\Decorator\Tests;
4
5  use DesignPatterns\Structural\Decorator;
6
7  /**
8   * DecoratorTest tests the decorator pattern
9   */
10 class DecoratorTest extends \PHPUnit_Framework_TestCase
11 {
12
13     protected $service;
14
15     protected function setUp()
16     {
17         $this->service = new Decorator\Webbservice(array('foo' => 'bar'));
18     }
19
20     public function testJsonDecorator()
21     {
22         // Wrap service with a JSON decorator for renderers
23         $service = new Decorator\RenderInJson($this->service);
24         // Our Renderer will now output JSON instead of an array
25         $this->assertEquals('{"foo":"bar"}', $service->renderData());
26     }
27
28     public function testXmlDecorator()
29     {
30         // Wrap service with a JSON decorator for renderers
31         $service = new Decorator\RenderInXml($this->service);
32         // Our Renderer will now output XML instead of an array
33         $xml = '<?xml version="1.0"?><foo>bar</foo>';
34         $this->assertXmlStringEqualsXmlString($xml, $service->renderData());
35     }
36
37     /**
38      * The first key-point of this pattern :
39      */
40     public function testDecoratorMustImplementsRenderer()
41     {
42         $className = 'DesignPatterns\Structural\Decorator\Decorator';
43         $interfaceName = 'DesignPatterns\Structural\Decorator\RendererInterface';
44         $this->assertTrue(is_subclass_of($className, $interfaceName));
45     }
46
47     /**
48      * Second key-point of this pattern : the decorator is type-hinted
49      *
50      * @expectedException \PHPUnit_Framework_Error
```

```

51     */
52     public function testDecoratorTypeHinted()
53     {
54         $this->getMockForAbstractClass('DesignPatterns\Structural\Decorator\Decorator', array(new \s
55     }
56
57     /**
58      * The decorator implements and wraps the same interface
59     */
60     public function testDecoratorOnlyAcceptRenderer()
61     {
62         $mock = $this->getMock('DesignPatterns\Structural\Decorator\RendererInterface');
63         $dec = $this->getMockForAbstractClass('DesignPatterns\Structural\Decorator\Decorator', array
64         $this->assertNotNull($dec);
65     }
66 }

```

1.2.6 Dependency Injection

Purpose

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

Usage

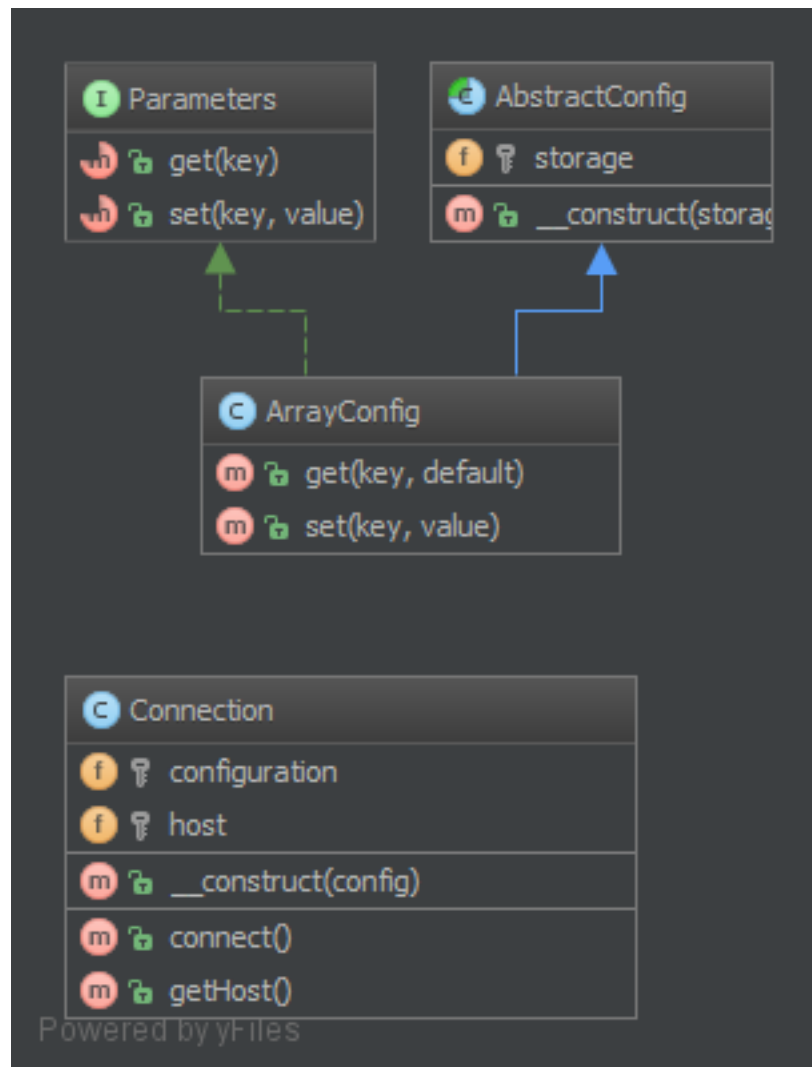
Configuration gets injected and `Connection` will get all that it needs from `$config`. Without DI, the configuration would be created directly in `Connection`, which is not very good for testing and extending `Connection`.

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
9  {
10     /**
11      * @var Storage of data
12      */

```

```

13     protected $storage;
14
15     public function __construct($storage)
16     {
17         $this->storage = $storage;
18     }
19 }

```

Parameters.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\DependencyInjection;
4
5  /**
6   * Parameters interface
7   */
8  interface Parameters
9  {
10     /**
11      * Get parameter
12      *
13      * @param string|int $key
14      *
15      * @return mixed
16      */
17     public function get($key);
18
19     /**
20      * Set parameter
21      *
22      * @param string|int $key
23      * @param mixed      $value
24      */
25     public function set($key, $value);
26 }

```

ArrayConfig.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\DependencyInjection;
4
5  /**
6   * class ArrayConfig
7   *
8   * uses array as data source
9   */
10 class ArrayConfig extends AbstractConfig implements Parameters
11 {
12     /**
13      * Get parameter
14      *
15      * @param string|int $key
16      * @param null $default
17      * @return mixed
18      */
19     public function get($key, $default = null)
20     {

```

```
21         if (isset($this->storage[$key])) {
22             return $this->storage[$key];
23         }
24         return $default;
25     }
26
27     /**
28      * Set parameter
29      *
30      * @param string|int $key
31      * @param mixed $value
32      */
33     public function set($key, $value)
34     {
35         $this->storage[$key] = $value;
36     }
37 }
```

Connection.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\DependencyInjection;
4
5 /**
6  * Class Connection
7  */
8 class Connection
9 {
10     /**
11      * @var Configuration
12      */
13     protected $configuration;
14
15     /**
16      * @var Currently connected host
17      */
18     protected $host;
19
20     /**
21      * @param Parameters $config
22      */
23     public function __construct(Parameters $config)
24     {
25         $this->configuration = $config;
26     }
27
28     /**
29      * connection using the injected config
30      */
31     public function connect()
32     {
33         $host = $this->configuration->get('host');
34         // connection to host, authentication etc...
35
36         //if connected
37         $this->host = $host;
38     }
```



```

39
40     /*
41      * Get currently connected host
42      *
43      * @return string
44      */
45     public function getHost()
46     {
47         return $this->host;
48     }
49 }

```

Test

Tests/DependencyInjectionTest.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\DependencyInjection\Tests;
4
5  use DesignPatterns\Structural\DependencyInjection\ArrayConfig;
6  use DesignPatterns\Structural\DependencyInjection\Connection;
7
8  class DependencyInjectionTest extends \PHPUnit_Framework_TestCase
9  {
10     protected $config;
11     protected $source;
12
13     public function setUp()
14     {
15         $this->source = include 'config.php';
16         $this->config = new ArrayConfig($this->source);
17     }
18
19     public function testDependencyInjection()
20     {
21         $connection = new Connection($this->config);
22         $connection->connect();
23         $this->assertEquals($this->source['host'], $connection->getHost());
24     }
25 }

```

Tests/config.php

```

1  <?php
2
3  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

```
1 <?php
2
3 namespace DesignPatterns\Structural\Facade;
4
5 /**
6  *
```

```

7  *
8  */
9  class Facade
10 {
11     /**
12      * @var OsInterface
13      */
14     protected $os;
15
16     /**
17      * @var BiosInterface
18      */
19     protected $bios;
20
21     /**
22      * This is the perfect time to use a dependency injection container
23      * to create an instance of this class
24      *
25      * @param BiosInterface $bios
26      * @param OsInterface $os
27      */
28     public function __construct(BiosInterface $bios, OsInterface $os)
29     {
30         $this->bios = $bios;
31         $this->os = $os;
32     }
33
34     /**
35      * turn on the system
36      */
37     public function turnOn()
38     {
39         $this->bios->execute();
40         $this->bios->waitForKeyPress();
41         $this->bios->launch($this->os);
42     }
43
44     /**
45      * turn off the system
46      */
47     public function turnOff()
48     {
49         $this->os->halt();
50         $this->bios->powerDown();
51     }
52 }

```

OsInterface.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Facade;
4
5  /**
6   * Class OsInterface
7   */
8  interface OsInterface
9  {
10     /**

```

```
11     * halt the OS
12     */
13     public function halt();
14 }
```

BiosInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Facade;
4
5 /**
6  * Class BiosInterface
7  */
8 interface BiosInterface
9 {
10     /**
11      * execute the BIOS
12      */
13     public function execute();
14
15     /**
16      * wait for halt
17      */
18     public function waitForKeyPress();
19
20     /**
21      * launches the OS
22      *
23      * @param OsInterface $os
24      */
25     public function launch(OsInterface $os);
26
27     /**
28      * power down BIOS
29      */
30     public function powerDown();
31 }
```

Test

Tests/FacadeTest.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Facade\Tests;
4
5 use DesignPatterns\Structural\Facade\Facade as Computer;
6
7 /**
8  * FacadeTest shows example of facades
9  */
10 class FacadeTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function getComputer()
14     {
```

```

15     $bios = $this->getMockBuilder('DesignPatterns\Structural\Facade\BiosInterface')
16         ->setMethods(array('launch', 'execute', 'waitForKeyPress'))
17         ->disableAutoload()
18         ->getMock();
19     $operatingSys = $this->getMockBuilder('DesignPatterns\Structural\Facade\OsInterface')
20         ->setMethods(array('getName'))
21         ->disableAutoload()
22         ->getMock();
23     $bios->expects($this->once())
24         ->method('launch')
25         ->with($operatingSys);
26     $operatingSys
27         ->expects($this->once())
28         ->method('getName')
29         ->will($this->returnValue('Linux'));
30
31     $facade = new Computer($bios, $operatingSys);
32     return array(array($facade, $operatingSys));
33 }
34
35 /**
36  * @dataProvider getComputer
37  */
38 public function testComputerOn(Computer $facade, $os)
39 {
40     // interface is simpler :
41     $facade->turnOn();
42     // but I can access to lower component
43     $this->assertEquals('Linux', $os->getName());
44 }
45 }

```

1.2.8 Fluent Interface

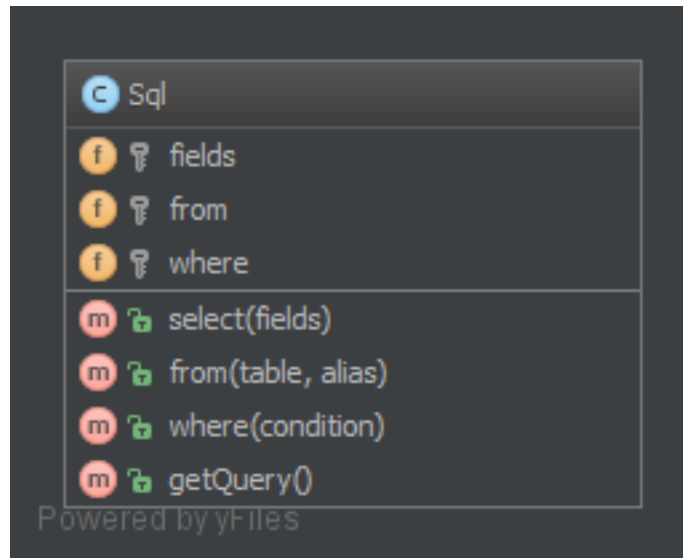
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Structural\FluentInterface;
4
5  /**
6   * class SQL
7   */
8  class Sql
9  {
10     /**
11      * @var array
12      */
13     protected $fields = array();
14
15     /**
16      * @var array
17      */
18     protected $from = array();
19
20     /**
21      * @var array
22      */
23     protected $where = array();
24
25     /**
26      * adds select fields
27      *
28      * @param array $fields
29      *
30      * @return SQL
  
```

```

31     */
32     public function select(array $fields = array())
33     {
34         $this->fields = $fields;
35
36         return $this;
37     }
38
39     /**
40      * adds a FROM clause
41      *
42      * @param string $table
43      * @param string $alias
44      *
45      * @return SQL
46      */
47     public function from($table, $alias)
48     {
49         $this->from[] = $table . ' AS ' . $alias;
50
51         return $this;
52     }
53
54     /**
55      * adds a WHERE condition
56      *
57      * @param string $condition
58      *
59      * @return SQL
60      */
61     public function where($condition)
62     {
63         $this->where[] = $condition;
64
65         return $this;
66     }
67
68     /**
69      * Gets the query, just an example of building a query,
70      * no check on consistency
71      *
72      * @return string
73      */
74     public function getQuery()
75     {
76         return 'SELECT ' . implode(' ', $this->fields)
77             . ' FROM ' . implode(' ', $this->from)
78             . ' WHERE ' . implode(' AND ', $this->where);
79     }
80 }

```

Test

Tests/FluentInterfaceTest.php

```

1 <?php
2

```

```
3 namespace DesignPatterns\Structural\FluentInterface\Tests;
4
5 use DesignPatterns\Structural\FluentInterface\Sql;
6
7 /**
8  * FluentInterfaceTest tests the fluent interface SQL
9  */
10 class FluentInterfaceTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testBuildSQL()
14     {
15         $instance = new Sql();
16         $query = $instance->select(array('foo', 'bar'))
17             ->from('foobar', 'f')
18             ->where('f.bar = ?')
19             ->getQuery();
20
21         $this->assertEquals('SELECT foo,bar FROM foobar AS f WHERE f.bar = ?', $query);
22     }
23 }
```

1.2.9 Proxy

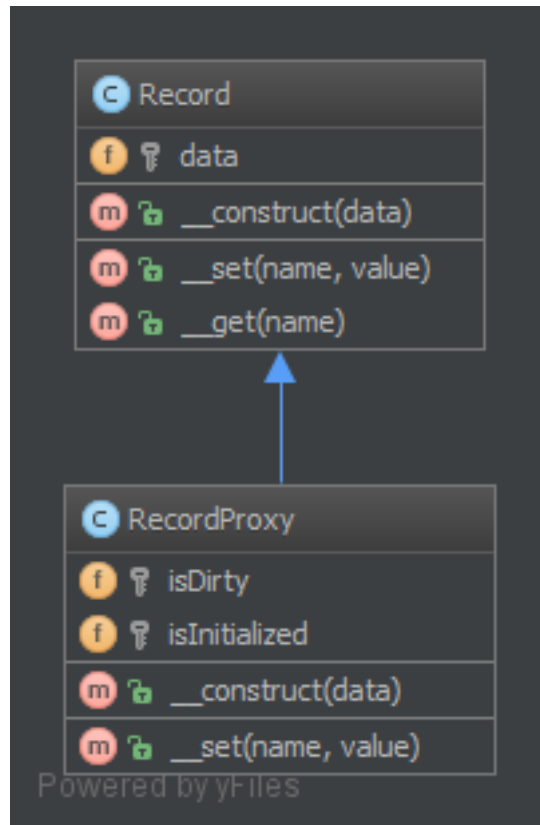
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Structural\Proxy;
4
5  /**
6   * class Record
7   */
8  class Record
9  {
10     /**
11      * @var array|null
12      */
13     protected $data;
14
15     /**
16      * @param null $data
17      */
18     public function __construct($data = null)
19     {
20         $this->data = (array) $data;
  
```

```
21     }
22
23     /**
24      * magic setter
25      *
26      * @param string $name
27      * @param mixed $value
28      *
29      * @return void
30      */
31     public function __set($name, $value)
32     {
33         $this->data[(string) $name] = $value;
34     }
35
36     /**
37      * magic getter
38      *
39      * @param string $name
40      *
41      * @return mixed|null
42      */
43     public function __get($name)
44     {
45         if (array_key_exists($name, $this->data)) {
46             return $this->data[(string) $name];
47         } else {
48             return null;
49         }
50     }
51 }
```

RecordProxy.php

```
1 <?php
2
3 namespace DesignPatterns\Structural\Proxy;
4
5 /**
6  * Class RecordProxy
7  */
8 class RecordProxy extends Record
9 {
10     /**
11      * @var bool
12      */
13     protected $isDirty = false;
14
15     /**
16      * @var bool
17      */
18     protected $isInitialized = false;
19
20     /**
21      * @param array $data
22      */
23     public function __construct($data)
24     {
25         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

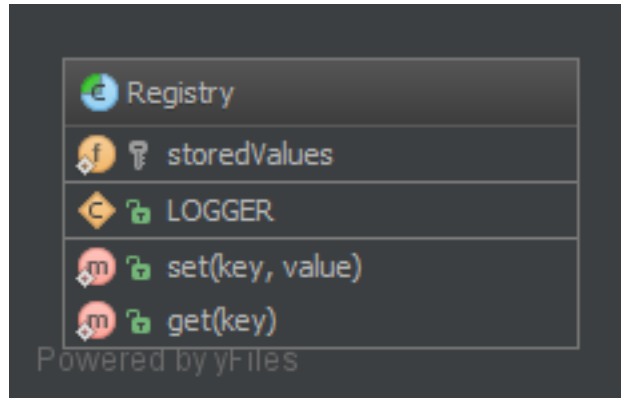
Purpose

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: `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

```
1 <?php
2
3 namespace DesignPatterns\Structural\Registry;
4
5 /**
6  * class Registry
7  */
8 abstract class Registry
9 {
10     const LOGGER = 'logger';
11
12     /**
13      * @var array
14      */
15     protected static $storedValues = array();
16
17     /**
18      * sets a value
19      *
20      * @param string $key
21      * @param mixed $value
22      *
23      * @static
24      * @return void
25      */
26     public static function set($key, $value)
27     {
28         self::$storedValues[$key] = $value;
29     }
30
31     /**
32      * gets a value from the registry
33      *
34      * @param string $key
35      */
```

```

36     * @static
37     * @return mixed
38     */
39     public static function get($key)
40     {
41         return self::$storedValues[$key];
42     }
43
44     // typically there would be methods to check if a key has already been registered and so on ...
45 }

```

Test

Tests/RegistryTest.php

```

1  <?php
2
3  namespace DesignPatterns\Structural\Registry\Tests;
4
5  use DesignPatterns\Structural\Registry\Registry;
6
7  class RegistryTest extends \PHPUnit_Framework_TestCase
8  {
9
10     public function testSetAndGetLogger()
11     {
12         Registry::set(Registry::LOGGER, new \StdClass());
13
14         $logger = Registry::get(Registry::LOGGER);
15         $this->assertInstanceOf('StdClass', $logger);
16     }
17 }

```

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:

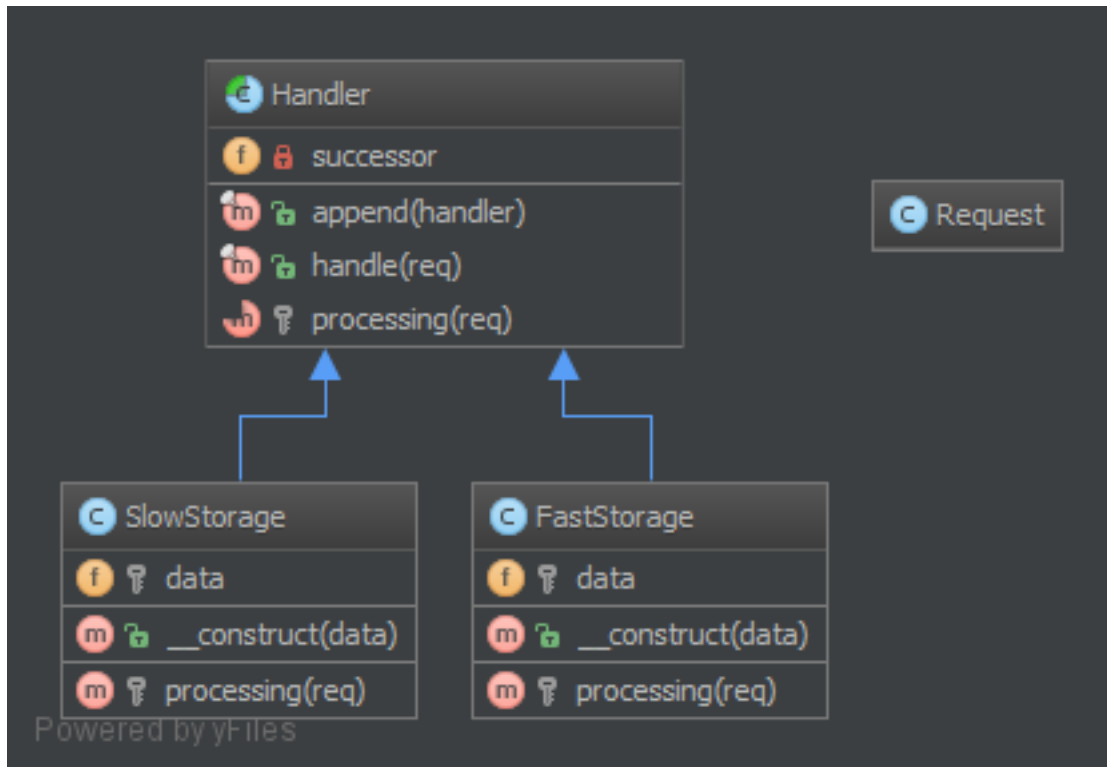
To build a chain of objects to handle a call in sequential order. If one object cannot handle a call, it delegates the call to the next in the chain and so forth.

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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\ChainOfResponsibilities;
4
5  /**
6   * Request is a request which goes through the chain of responsibilities.
7   *
8   * About the request: Sometimes, you don't need an object, just an integer or
9   * an array. But in this case of a full example, I've made a class to illustrate
10  * this important idea in the CoR (Chain of Responsibilities). In the real world,
11  * I recommend to always use a class, even a \stdClass if you want, it proves
12  * to be more adaptive because a single handler doesn't know much about the
13  * outside world and it is more difficult if, one day, you want to add some
14  * criterion in a decision process.
15  */
16  class Request

```

```

17 {
18     // getter and setter but I don't want to generate too much noise in handlers
19 }

```

Handler.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\ChainOfResponsibilities;
4
5  /**
6   * Handler is a generic handler in the chain of responsibilities
7   *
8   * Yes you could have a lighter CoR with a simpler handler but if you want your CoR
9   * to be extendable and decoupled, it's a better idea to do things like that in real
10  * situations. Usually, a CoR is meant to be changed everytime and evolves, that's
11  * why we slice the workflow in little bits of code.
12  */
13  abstract class Handler
14  {
15      /**
16       * @var Handler
17       */
18      private $successor = null;
19
20      /**
21       * Append a responsibility to the end of chain
22       *
23       * A prepend method could be done with the same spirit
24       *
25       * You could also send the successor in the constructor but in PHP that is a
26       * bad idea because you have to remove the type-hint of the parameter because
27       * the last handler has a null successor.
28       *
29       * And if you override the constructor, that Handler can no longer have a
30       * successor. One solution is to provide a NullObject (see pattern).
31       * It is more preferable to keep the constructor "free" to inject services
32       * you need with the DiC of symfony2 for example.
33       *
34       * @param Handler $handler
35       */
36      final public function append(Handler $handler)
37      {
38          if (is_null($this->successor)) {
39              $this->successor = $handler;
40          } else {
41              $this->successor->append($handler);
42          }
43      }
44
45      /**
46       * Handle the request.
47       *
48       * This approach by using a template method pattern ensures you that
49       * each subclass will not forget to call the successor. Besides, the returned
50       * boolean value indicates you if the request have been processed or not.
51       *
52       * @param Request $req
53       */

```

```
54     * @return bool
55     */
56     final public function handle(Request $req)
57     {
58         $req->forDebugOnly = get_called_class();
59         $processed = $this->processing($req);
60         if (!$processed) {
61             // the request has not been processed by this handler => see the next
62             if (!is_null($this->successor)) {
63                 $processed = $this->successor->handle($req);
64             }
65         }
66
67         return $processed;
68     }
69
70     /**
71      * Each concrete handler has to implement the processing of the request
72      *
73      * @param Request $req
74      *
75      * @return bool true if the request has been processed
76      */
77     abstract protected function processing(Request $req);
78 }
```

Responsible/SlowStorage.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   * This is mostly the same code as FastStorage but in fact, it may greatly differs
10   *
11   * One important fact about CoR: each item in the chain MUST NOT assume its position
12   * in the chain. A CoR is not responsible if the request is not handled UNLESS
13   * you make an "ExceptionHandler" which throws exception if the request goes there.
14   *
15   * To be really extendable, each handler doesn't know if there is something after it.
16   *
17   */
18  class SlowStorage extends Handler
19  {
20      /**
21       * @var array
22       */
23      protected $data = array();
24
25      /**
26       * @param array $data
27       */
28      public function __construct($data = array())
29      {
30          $this->data = $data;
31      }
32  }
```



```

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

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\ChainOfResponsibilities\Tests;
4
5  use DesignPatterns\Behavioral\ChainOfResponsibilities\Request;
6  use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\FastStorage;
7  use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\SlowStorage;
8  use DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible;
9
10 /**
11  * ChainTest tests the CoR
12  */
13 class ChainTest extends \PHPUnit_Framework_TestCase
14 {
15
16     /**
17      * @var FastStorage
18      */
19     protected $chain;
20
21     protected function setUp()
22     {
23         $this->chain = new FastStorage(array('bar' => 'baz'));
24         $this->chain->append(new SlowStorage(array('bar' => 'baz', 'foo' => 'bar')));
25     }
26
27     public function makeRequest()
28     {
29         $request = new Request();
30         $request->verb = 'get';
31
32         return array(
33             array($request)
34         );
35     }
36
37     /**
38      * @dataProvider makeRequest
39      */
40     public function testFastStorage($request)
41     {
42         $request->key = 'bar';
43         $ret = $this->chain->handle($request);
44
45         $this->assertTrue($ret);
46         $this->assertObjectHasAttribute('response', $request);
47         $this->assertEquals('baz', $request->response);
48         // despite both handle owns the 'bar' key, the FastStorage is responding first
49         $className = 'DesignPatterns\Behavioral\ChainOfResponsibilities\Responsible\FastStorage';
50         $this->assertEquals($className, $request->forDebugOnly);
51     }
52
53     /**
54      * @dataProvider makeRequest
55      */
```

```

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.

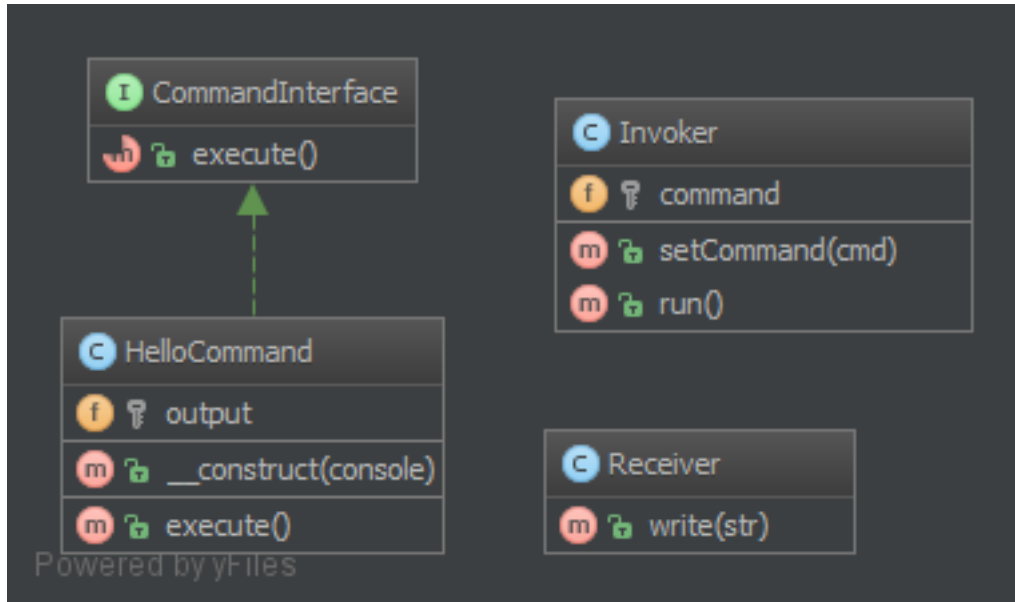
We have an Invoker and a Receiver. This pattern uses a “Command” to delegate the method call against the Receiver and presents the same method “execute”. Therefore, the Invoker just knows to call “execute” to process the Command of the client. The Receiver is decoupled from the Invoker.

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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Command;
4
5  /**
6   * class CommandInterface
7   */
8  interface CommandInterface
9  {
10     /**
11      * this is the most important method in the Command pattern,
12      * The Receiver goes in the constructor.
13      */
14     public function execute();
15 }
  
```

HelloCommand.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Command;
4
5  /**
6   * This concrete command calls "print" on the Receiver, but an external
7   * invoker just know he can call "execute"
8   */
9  class HelloCommand implements CommandInterface
10 {
  
```

```

11  /**
12   * @var Receiver
13   */
14  protected $output;
15
16  /**
17   * Each concrete command is builded with different receivers.
18   * Can be one, many, none or even other Command in parameters
19   *
20   * @param Receiver $console
21   */
22  public function __construct(Receiver $console)
23  {
24      $this->output = $console;
25  }
26
27  /**
28   * execute and output "Hello World"
29   */
30  public function execute()
31  {
32      // sometimes, there is no receiver and this is the command which
33      // does all the work
34      $this->output->write('Hello World');
35  }
36  }

```

Receiver.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Command;
4
5  /**
6   * Receiver is specific service with its own contract and can be only concrete
7   */
8  class Receiver
9  {
10     /**
11      * @param string $str
12      */
13     public function write($str)
14     {
15         echo $str;
16     }
17 }

```

Invoker.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Command;
4
5  /**
6   * Invoker is using the command given to it.
7   * Example : an Application in SF2
8   */
9  class Invoker
10 {

```

```
11  /**
12   * @var CommandInterface
13   */
14  protected $command;
15
16  /**
17   * In the invoker we find this kind of method for subscribing the command.
18   * There can be also a stack, a list, a fixed set...
19   *
20   * @param CommandInterface $cmd
21   */
22  public function setCommand(CommandInterface $cmd)
23  {
24      $this->command = $cmd;
25  }
26
27  /**
28   * executes the command
29   */
30  public function run()
31  {
32      // here is a key feature of the invoker
33      // the invoker is the same whatever is the command
34      $this->command->execute();
35  }
36 }
```

Test

Tests/CommandTest.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Command\Tests;
4
5  use DesignPatterns\Behavioral\Command\Invoker;
6  use DesignPatterns\Behavioral\Command\Receiver;
7  use DesignPatterns\Behavioral\Command\HelloCommand;
8
9  /**
10   * CommandTest has the role of the Client in the Command Pattern
11   */
12  class CommandTest extends \PHPUnit_Framework_TestCase
13  {
14
15      /**
16       * @var Invoker
17       */
18      protected $invoker;
19
20      /**
21       * @var Receiver
22       */
23      protected $receiver;
24
25      protected function setUp()
26      {
```

```
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.

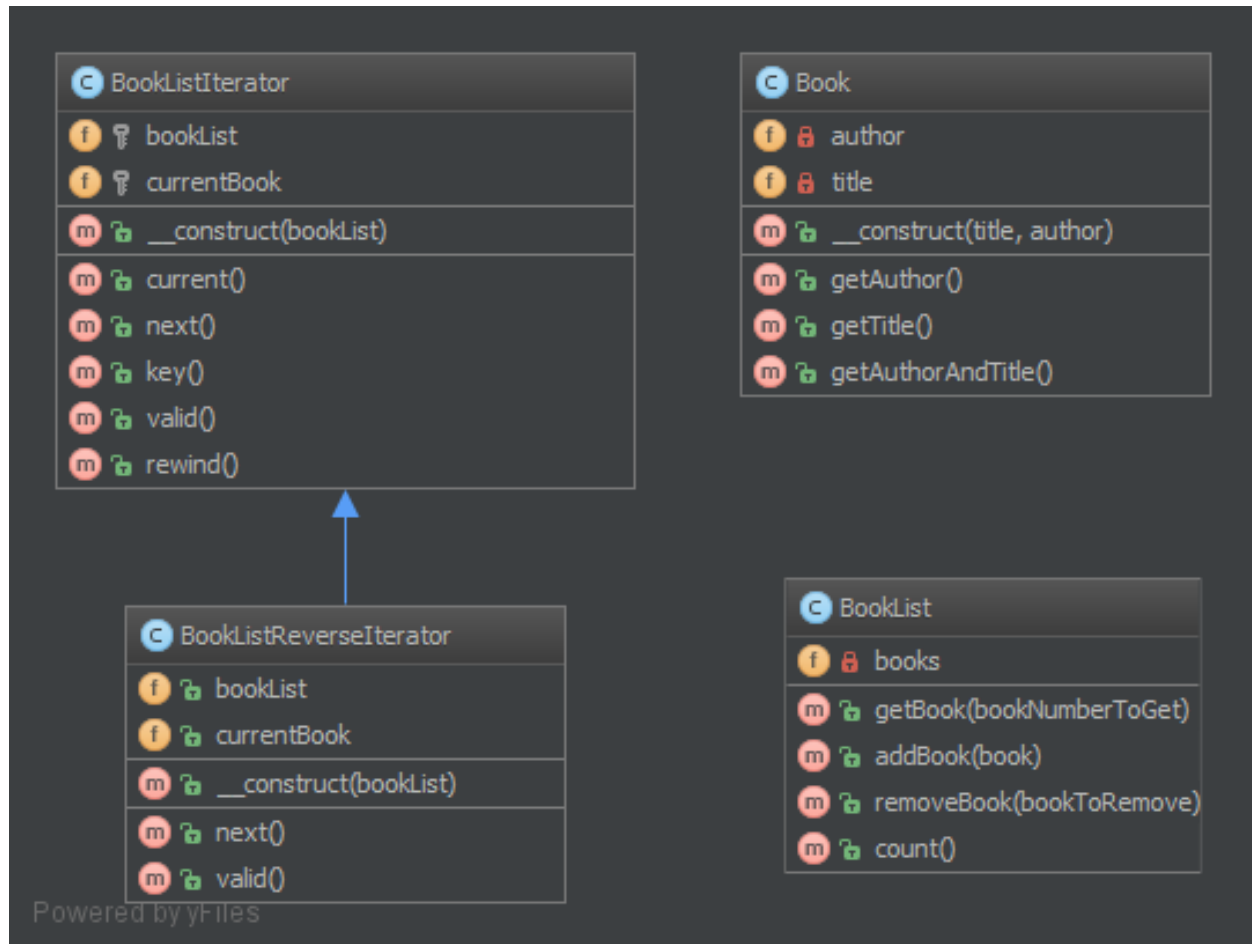
Examples

- 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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Iterator;
4
5  class Book
6  {
7
8      private $author;
9
10     private $title;
11
12     public function __construct($title, $author)
13     {
14         $this->author = $author;
15         $this->title = $title;
16     }
  
```



```

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 ((int)$bookNumberToGet <= $this->count()) {
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);

```

```
41     }
42 }
```

BookListIterator.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Iterator;
4
5  class BookListIterator implements \Iterator
6  {
7
8      /**
9       * @var BookList
10      */
11     protected $bookList;
12
13     /**
14      * @var int
15      */
16     protected $currentBook = 0;
17
18     public function __construct(BookList $bookList)
19     {
20         $this->bookList = $bookList;
21     }
22
23     /**
24      * Return the current book
25      * @link http://php.net/manual/en/iterator.current.php
26      * @return Book Can return any type.
27      */
28     public function current()
29     {
30         return $this->bookList->getBook($this->currentBook);
31     }
32
33     /**
34      * (PHP 5 >= 5.0.0)<br/>
35      * Move forward to next element
36      * @link http://php.net/manual/en/iterator.next.php
37      * @return void Any returned value is ignored.
38      */
39     public function next()
40     {
41         $this->currentBook++;
42     }
43
44     /**
45      * (PHP 5 >= 5.0.0)<br/>
46      * Return the key of the current element
47      * @link http://php.net/manual/en/iterator.key.php
48      * @return mixed scalar on success, or null on failure.
49      */
50     public function key()
51     {
52         return $this->currentBook;
53     }
54 }
```

```

55  /**
56   * (PHP 5 >= 5.0.0)<br/>
57   * Checks if current position is valid
58   * @link http://php.net/manual/en/iterator.valid.php
59   * @return boolean The return value will be casted to boolean and then evaluated.
60   * Returns true on success or false on failure.
61   */
62  public function valid()
63  {
64      return $this->currentBook < $this->bookList->count();
65  }
66
67  /**
68   * (PHP 5 >= 5.0.0)<br/>
69   * Rewind the Iterator to the first element
70   * @link http://php.net/manual/en/iterator.rewind.php
71   * @return void Any returned value is ignored.
72   */
73  public function rewind()
74  {
75      $this->currentBook = 0;
76  }
77  }

```

BookListReverseIterator.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Iterator;
4
5  class BookListReverseIterator extends BookListIterator
6  {
7
8      public function __construct(BookList $bookList)
9      {
10         $this->bookList = $bookList;
11         $this->currentBook = $this->bookList->count() - 1;
12     }
13
14     public function next()
15     {
16         $this->currentBook--;
17     }
18
19     public function valid()
20     {
21         return 0 <= $this->currentBook;
22     }
23 }

```

Test

Tests/IteratorTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Iterator\Tests;

```

```
4
5 use DesignPatterns\Behavioral\Iterator\Book;
6 use DesignPatterns\Behavioral\Iterator\BookList;
7 use DesignPatterns\Behavioral\Iterator\BookListIterator;
8 use DesignPatterns\Behavioral\Iterator\BookListReverseIterator;
9
10 class IteratorTest extends \PHPUnit_Framework_TestCase
11 {
12
13     /**
14      * @var BookList
15      */
16     protected $bookList;
17
18     protected function setUp()
19     {
20         $this->bookList = new BookList();
21         $this->bookList->addBook(new Book('Learning PHP Design Patterns', 'William Sanders'));
22         $this->bookList->addBook(new Book('Professional Php Design Patterns', 'Aaron Saray'));
23         $this->bookList->addBook(new Book('Clean Code', 'Robert C. Martin'));
24     }
25
26     public function expectedAuthors()
27     {
28         return array(
29             array(
30                 array(
31                     'Learning PHP Design Patterns by William Sanders',
32                     'Professional Php Design Patterns by Aaron Saray',
33                     'Clean Code by Robert C. Martin'
34                 )
35             ),
36         );
37     }
38
39     /**
40      * @dataProvider expectedAuthors
41      */
42     public function testUseAIteratorAndValidateAuthors($expected)
43     {
44         $iterator = new BookListIterator($this->bookList);
45
46         while ($iterator->valid()) {
47             $expectedBook = array_shift($expected);
48             $this->assertEquals($expectedBook, $iterator->current()->getAuthorAndTitle());
49             $iterator->next();
50         }
51     }
52
53     /**
54      * @dataProvider expectedAuthors
55      */
56     public function testUseAReverseIteratorAndValidateAuthors($expected)
57     {
58         $iterator = new BookListReverseIterator($this->bookList);
59
60         while ($iterator->valid()) {
61             $expectedBook = array_pop($expected);
```

```

62         $this->assertEquals($expectedBook, $iterator->current()->getAuthorAndTitle());
63         $iterator->next();
64     }
65 }
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 }

```

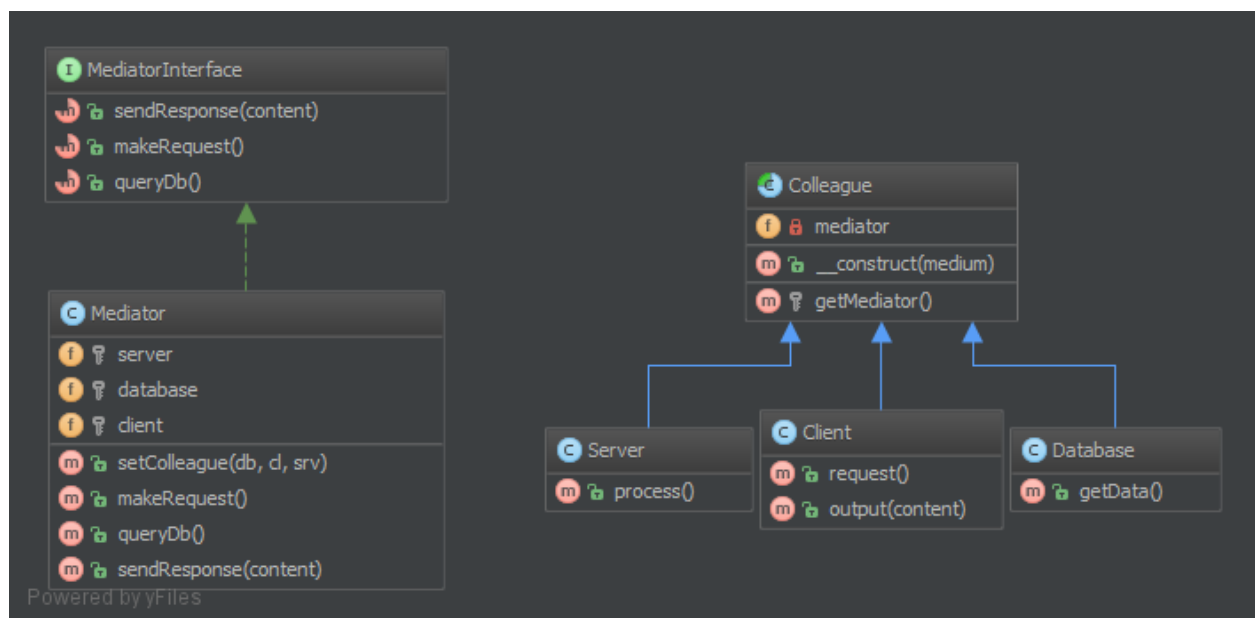
1.3.4 Mediator

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
5 /**
6  * MediatorInterface is a contract for the Mediator
7  * This interface is not mandatory but it is better for LSP concerns
8  */
9 interface MediatorInterface
10 {
11     /**
12      * sends the response
13      *
14      * @param string $content
15      */
16     public function sendResponse($content);
17
18     /**
19      * makes a request
20      */
21     public function makeRequest();
22
23     /**
24      * queries the DB
25      */
26     public function queryDb();
27 }
```

Mediator.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\Mediator;
4
5 use DesignPatterns\Behavioral\Mediator\Subsystem;
6
7 /**
8  * Mediator is the concrete Mediator for this design pattern.
9  * In this example, I have made a "Hello World" with the Mediator Pattern.
10 */
11 class Mediator implements MediatorInterface
12 {
13
14     /**
15      * @var Subsystem\Server
16      */
17     protected $server;
18
19     /**
20      * @var Subsystem\Database
21      */
22     protected $database;
23
24     /**
25      * @var Subsystem\Client
26      */
27     protected $client;
28
29     /**
```

```

30     * @param Subsystem\Database $db
31     * @param Subsystem\Client $cl
32     * @param Subsystem\Server $srv
33     */
34     public function setColleague(Subsystem\Database $db, Subsystem\Client $cl, Subsystem\Server $srv)
35     {
36         $this->database = $db;
37         $this->server = $srv;
38         $this->client = $cl;
39     }
40
41     /**
42     * make request
43     */
44     public function makeRequest()
45     {
46         $this->server->process();
47     }
48
49     /**
50     * query db
51     * @return mixed
52     */
53     public function queryDb()
54     {
55         return $this->database->getData();
56     }
57
58     /**
59     * send response
60     *
61     * @param string $content
62     */
63     public function sendResponse($content)
64     {
65         $this->client->output($content);
66     }
67 }

```

Colleague.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator;
4
5  /**
6   * Colleague is an abstract colleague who works together but he only knows
7   * the Mediator, not other colleague.
8   */
9  abstract class Colleague
10 {
11     /**
12     * this ensures no change in subclasses
13     *
14     * @var MediatorInterface
15     */
16     private $mediator;
17

```

```
18  /**
19   * @param MediatorInterface $mediator
20   */
21  public function __construct(MediatorInterface $mediator)
22  {
23      // in this way, we are sure the concrete colleague knows the mediator
24      $this->mediator = $mediator;
25  }
26
27  // for subclasses
28  protected function getMediator()
29  {
30      return $this->mediator;
31  }
32 }
```

Subsystem/Client.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5  use DesignPatterns\Behavioral\Mediator\Colleague;
6
7  /**
8   * Client is a client that make request et get response
9   */
10 class Client extends Colleague
11 {
12     /**
13      * request
14      */
15     public function request()
16     {
17         $this->getMediator()->makeRequest();
18     }
19
20     /**
21      * output content
22      *
23      * @param string $content
24      */
25     public function output($content)
26     {
27         echo $content;
28     }
29 }
```

Subsystem/Database.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5  use DesignPatterns\Behavioral\Mediator\Colleague;
6
7  /**
8   * Database is a database service
9   */
```



```

10 class Database extends Colleague
11 {
12     /**
13      * @return string
14      */
15     public function getData()
16     {
17         return "World";
18     }
19 }

```

Subsystem/Server.php

```

1 <?php
2
3 namespace DesignPatterns\Behavioral\Mediator\Subsystem;
4
5 use DesignPatterns\Behavioral\Mediator\Colleague;
6
7 /**
8  * Server serves responses
9  */
10 class Server extends Colleague
11 {
12     /**
13      * process on server
14      */
15     public function process()
16     {
17         $data = $this->getMediator()->queryDb();
18         $this->getMediator()->sendResponse("Hello $data");
19     }
20 }

```

Test

Tests/MediatorTest.php

```

1 <?php
2
3 namespace DesignPatterns\Tests\Mediator\Tests;
4
5 use DesignPatterns\Behavioral\Mediator\Mediator;
6 use DesignPatterns\Behavioral\Mediator\Subsystem\Database;
7 use DesignPatterns\Behavioral\Mediator\Subsystem\Client;
8 use DesignPatterns\Behavioral\Mediator\Subsystem\Server;
9
10 /**
11  * MediatorTest tests hello world
12  */
13 class MediatorTest extends \PHPUnit_Framework_TestCase
14 {
15
16     protected $client;
17
18     protected function setUp()
19     {

```

```
20     $media = new Mediator();
21     $this->client = new Client($media);
22     $media->setColleague(new Database($media), $this->client, new Server($media));
23 }
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 many situations.
33 }
34 }
```

1.3.5 Memento

Purpose

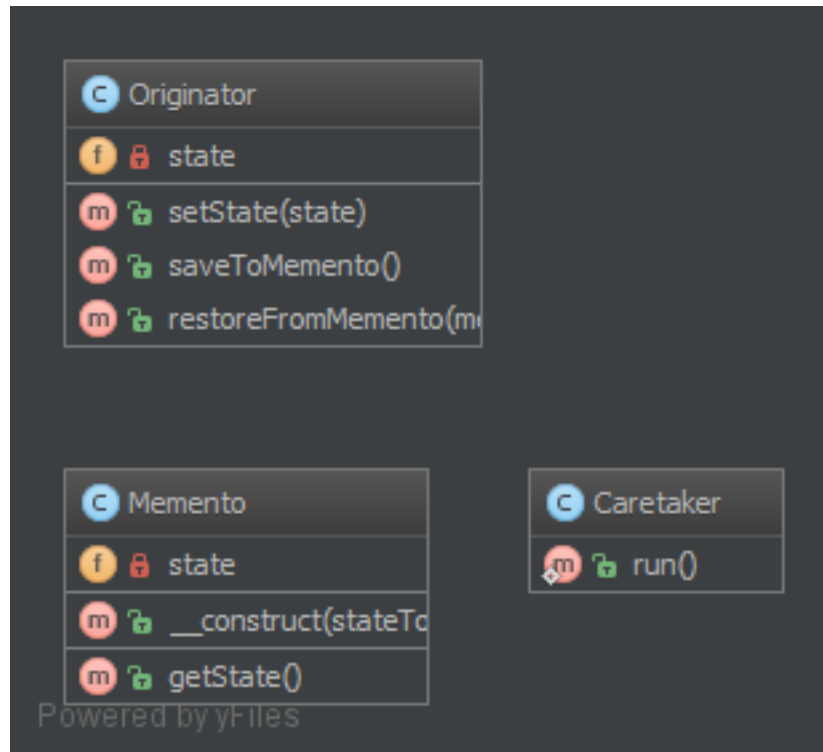
Provide the ability to restore an object to its previous state (undo via rollback).

The memento pattern is implemented with three objects: the originator, a caretaker and a memento. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

Examples

- The seed of a pseudorandom number generator
- The state in a finite state machine

UML Diagram



Code

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

Memento.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento;
4
5  class Memento
6  {
7      /** @var mixed */
8      private $state;
9
10     /**
11      * @param mixed $stateToSave
12      */
13     public function __construct($stateToSave)
14     {
15         $this->state = $stateToSave;
16     }
17
18     /**
19      * @return mixed
20      */
21     public function getState()
22     {
23         return $this->state;
24     }
25 }
```

```
24     }
25 }
```

Originator.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento;
4
5  class Originator
6  {
7      /* @var mixed */
8      private $state;
9
10     // The class could also contain additional data that is not part of the
11     // state saved in the memento..
12
13     /**
14      * @param mixed $state
15      */
16     public function setState($state)
17     {
18         $this->state = $state;
19     }
20
21     /**
22      * @return Memento
23      */
24     public function saveToMemento()
25     {
26         $state = is_object($this->state) ? clone $this->state : $this->state;
27
28         return new Memento($state);
29     }
30
31     public function restoreFromMemento(Memento $memento)
32     {
33         $this->state = $memento->getState();
34     }
35 }
```

Caretaker.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento;
4
5  class Caretaker
6  {
7      public static function run()
8      {
9          /* @var $savedStates Memento[] */
10
11         $savedStates = array();
12
13         $originator = new Originator();
14
15         //Setting state to State1
16         $originator->setState("State1");
```

```

17         //Setting state to State2
18         $originator->setState("State2");
19         //Saving State2 to Memento
20         $savedStates[] = $originator->saveToMemento();
21         //Setting state to State3
22         $originator->setState("State3");
23
24         // We can request multiple mementos, and choose which one to roll back to.
25         // Saving State3 to Memento
26         $savedStates[] = $originator->saveToMemento();
27         //Setting state to State4
28         $originator->setState("State4");
29
30         $originator->restoreFromMemento($savedStates[1]);
31         //State after restoring from Memento: State3
32     }
33 }

```

Test

Tests/MementoTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Memento\Tests;
4
5  use DesignPatterns\Behavioral\Memento\Originator;
6
7  /**
8   * MementoTest tests the memento pattern
9   */
10 class MementoTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testStringState()
14     {
15         $originator = new Originator();
16         $originator->setState("State1");
17
18         $this->assertAttributeEquals("State1", "state", $originator);
19
20         $originator->setState("State2");
21
22         $this->assertAttributeEquals("State2", "state", $originator);
23
24         $savedState = $originator->saveToMemento();
25
26         $this->assertAttributeEquals("State2", "state", $savedState);
27
28         $originator->setState("State3");
29
30         $this->assertAttributeEquals("State3", "state", $originator);
31
32         $originator->restoreFromMemento($savedState);
33
34         $this->assertAttributeEquals("State2", "state", $originator);
35     }

```

```
36
37 public function testObjectState()
38 {
39     $originator = new Originator();
40
41     $foo        = new \stdClass();
42     $foo->data = "foo";
43
44     $originator->setState($foo);
45
46     $this->assertAttributeEquals($foo, "state", $originator);
47
48     $savedState = $originator->saveToMemento();
49
50     $this->assertAttributeEquals($foo, "state", $savedState);
51
52     $bar        = new \stdClass();
53     $bar->data = "bar";
54
55     $originator->setState($bar);
56
57     $this->assertAttributeEquals($bar, "state", $originator);
58
59     $originator->restoreFromMemento($savedState);
60
61     $this->assertAttributeEquals($foo, "state", $originator);
62
63     $foo->data = null;
64
65     $this->assertAttributeNotEquals($foo, "state", $savedState);
66
67     $this->assertAttributeNotEquals($foo, "state", $originator);
68 }
69 }
```

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

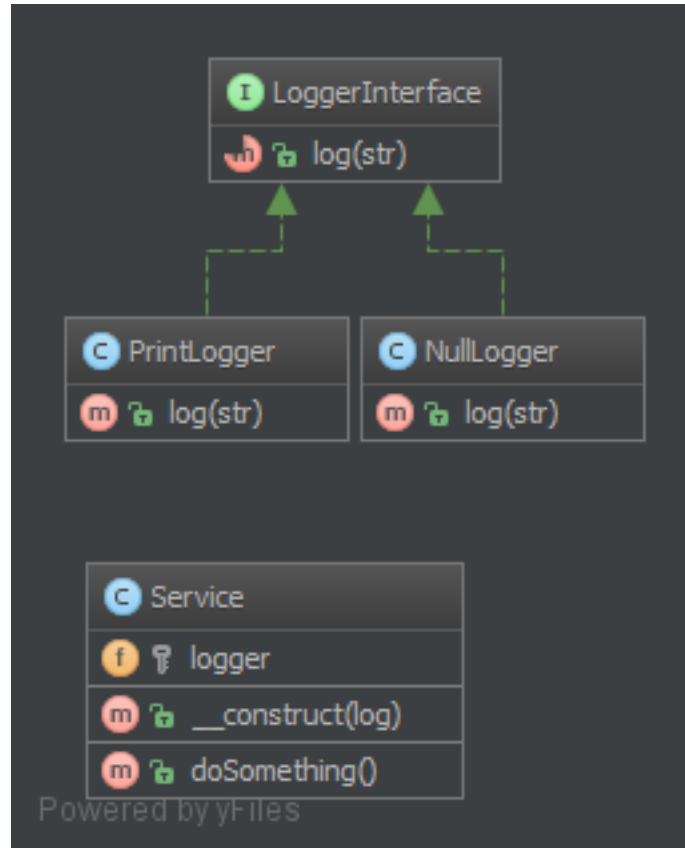
Methods that return an object or null should instead return an object or NullObject. NullObjects simplify boilerplate code such as `if (!is_null($obj)) { $obj->callSomething(); }` to just `$obj->callSomething();` by eliminating the conditional check in client code.

Examples

- Symfony2: null logger or 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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\NullObject;
4
5  /**
6   * Service is dummy service that uses a logger
7   */
8  class Service
9  {
10     /**
11      * @var LoggerInterface
12      */
13     protected $logger;
14
15     /**

```

```
16     * we inject the logger in ctor and it is mandatory
17     *
18     * @param LoggerInterface $log
19     */
20     public function __construct(LoggerInterface $log)
21     {
22         $this->logger = $log;
23     }
24
25     /**
26     * do something ...
27     */
28     public function doSomething()
29     {
30         // no more check "if (!is_null($this->logger))..." with the NullObject pattern
31         $this->logger->log('We are in ' . __METHOD__);
32         // something to do...
33     }
34 }
```

LoggerInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\NullObject;
4
5 /**
6  * LoggerInterface is a contract for logging something
7  *
8  * Key feature: NullLogger MUST inherit from this interface like any other Loggers
9  */
10 interface LoggerInterface
11 {
12     /**
13     * @param string $str
14     *
15     * @return mixed
16     */
17     public function log($str);
18 }
```

PrintLogger.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\NullObject;
4
5 /**
6  * PrintLogger is a logger that prints the log entry to standard output
7  */
8 class PrintLogger implements LoggerInterface
9 {
10     /**
11     * @param string $str
12     */
13     public function log($str)
14     {
15         echo $str;
16     }
17 }
```


17

}

NullLogger.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\NullObject;
4
5  /**
6   * Performance concerns : ok there is a call for nothing but we spare an "if is_null"
7   * I didn't run a benchmark but I think it's equivalent.
8   *
9   * Key feature : of course this logger MUST implement the same interface (or abstract)
10  * like the other loggers.
11  */
12  class NullLogger implements LoggerInterface
13  {
14      /**
15       * {@inheritdoc}
16       */
17      public function log($str)
18      {
19          // do nothing
20      }
21  }
```

Test

Tests/LoggerTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\NullObject\Tests;
4
5  use DesignPatterns\Behavioral\NullObject\NullLogger;
6  use DesignPatterns\Behavioral\NullObject\Service;
7  use DesignPatterns\Behavioral\NullObject\PrintLogger;
8
9  /**
10   * LoggerTest tests for different loggers
11   */
12  class LoggerTest extends \PHPUnit_Framework_TestCase
13  {
14
15      public function testNullObject()
16      {
17          // one can use a singleton for NullObjet : I don't think it's a good idea
18          // because the purpose behind null object is to "avoid special case".
19          $service = new Service(new NullLogger());
20          $this->expectOutputString(null); // no output
21          $service->doSomething();
22      }
23
24      public function testStandardLogger()
25      {
26          $service = new Service(new PrintLogger());
27          $this->expectOutputString('We are in DesignPatterns\Behavioral\NullObject\Service::doSomething');
```

```
28     $service->doSomething();  
29 }  
30 }
```

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.

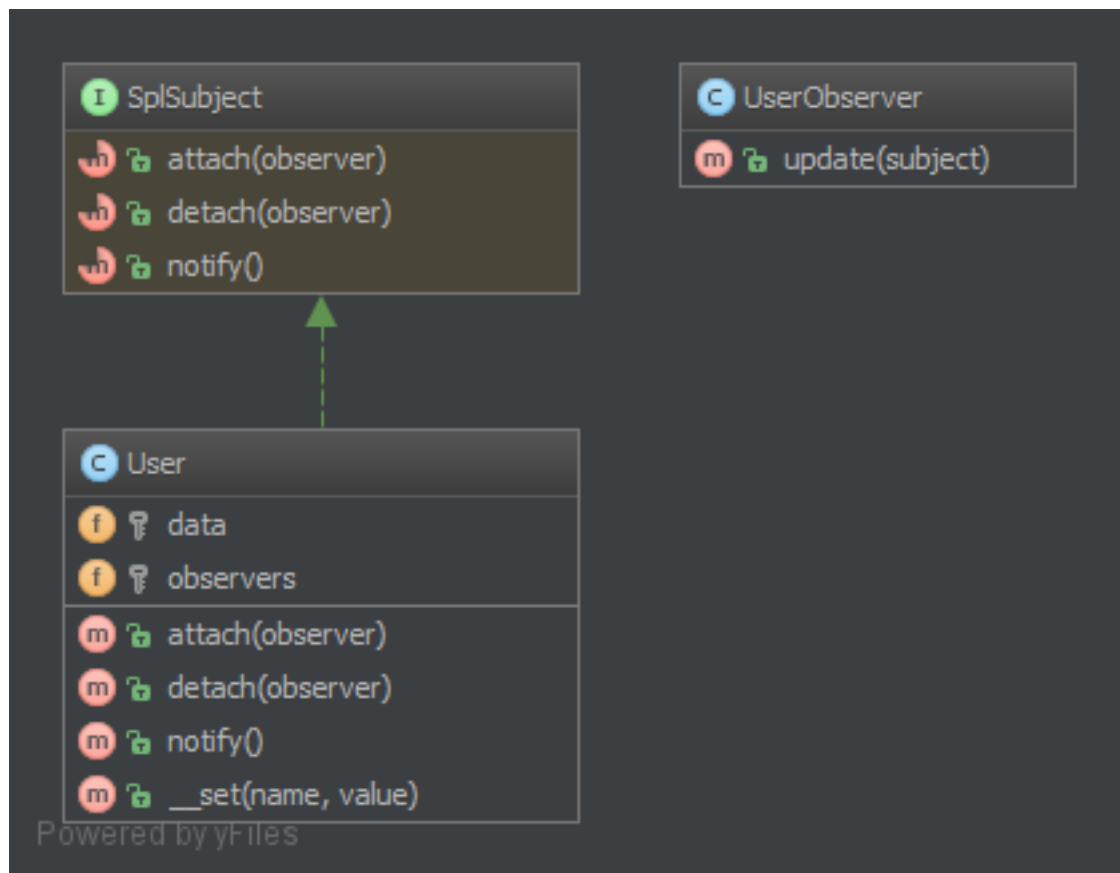
Examples

- 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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Observer;
4
5  /**
6   * Observer pattern : The observed object (the subject)
7   *
8   * The subject maintains a list of Observers and sends notifications.
9   *
10  */
11  class User implements \SplSubject
12  {
13      /**
14       * user data
15       *
16       * @var array
17       */
18      protected $data = array();
19
20      /**
21       * observers
22       *
23       * @var array
24       */
25      protected $observers = array();
26
27      /**
28       * attach a new observer
29       *
30       * @param \SplObserver $observer
31       *
32       * @return void
33       */
34      public function attach(\SplObserver $observer)
35      {
36          $this->observers[] = $observer;
37      }
38
39      /**
40       * detach an observer
41       *
42       * @param \SplObserver $observer
43       *
44       * @return void
45       */
46      public function detach(\SplObserver $observer)
47      {
48          $index = array_search($observer, $this->observers);
49
50          if (false !== $index) {
51              unset($this->observers[$index]);
52          }
53      }

```

```
54
55     /**
56      * notify observers
57      *
58      * @return void
59      */
60     public function notify()
61     {
62         /** @var \SplObserver $observer */
63         foreach ($this->observers as $observer) {
64             $observer->update($this);
65         }
66     }
67
68     /**
69      * Ideally one would better write setter/getter for all valid attributes and only call notify()
70      * on attributes that matter when changed
71      *
72      * @param string $name
73      * @param mixed $value
74      *
75      * @return void
76      */
77     public function __set($name, $value)
78     {
79         $this->data[$name] = $value;
80
81         // notify the observers, that user has been updated
82         $this->notify();
83     }
84 }
```

UserObserver.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Observer;
4
5  /**
6   * class UserObserver
7   */
8  class UserObserver implements \SplObserver
9  {
10     /**
11      * This is the only method to implement as an observer.
12      * It is called by the Subject (usually by SplSubject::notify() )
13      *
14      * @param \SplSubject $subject
15      */
16     public function update(\SplSubject $subject)
17     {
18         echo get_class($subject) . ' has been updated';
19     }
20 }
```

Test

Tests/ObserverTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Observer\Tests;
4
5  use DesignPatterns\Behavioral\Observer\UserObserver;
6  use DesignPatterns\Behavioral\Observer\User;
7
8  /**
9   * ObserverTest tests the Observer pattern
10  */
11  class ObserverTest extends \PHPUnit_Framework_TestCase
12  {
13
14      protected $observer;
15
16      protected function setUp()
17      {
18          $this->observer = new UserObserver();
19      }
20
21      /**
22       * Tests the notification
23      */
24      public function testNotify()
25      {
26          $this->expectOutputString('DesignPatterns\Behavioral\Observer\User has been updated');
27          $subject = new User();
28
29          $subject->attach($this->observer);
30          $subject->property = 123;
31      }
32
33      /**
34       * Tests the subscribing
35      */
36      public function testAttachDetach()
37      {
38          $subject = new User();
39          $this->assertAttributeEmpty('observers', $subject);
40          $subject->attach($this->observer);
41          $this->assertAttributeNotEmpty('observers', $subject);
42          $subject->detach($this->observer);
43          $this->assertAttributeEmpty('observers', $subject);
44      }
45
46      /**
47       * Tests the update() invocation on a mockup
48      */
49      public function testUpdateCalling()
50      {
51          $subject = new User();
52          $observer = $this->getMock('SplObserver');
53          $subject->attach($observer);
54
55          $observer->expects($this->once())
56              ->method('update')
57              ->with($subject);
58

```

```

59     $subject->notify();
60 }
61 }

```

1.3.8 Specification

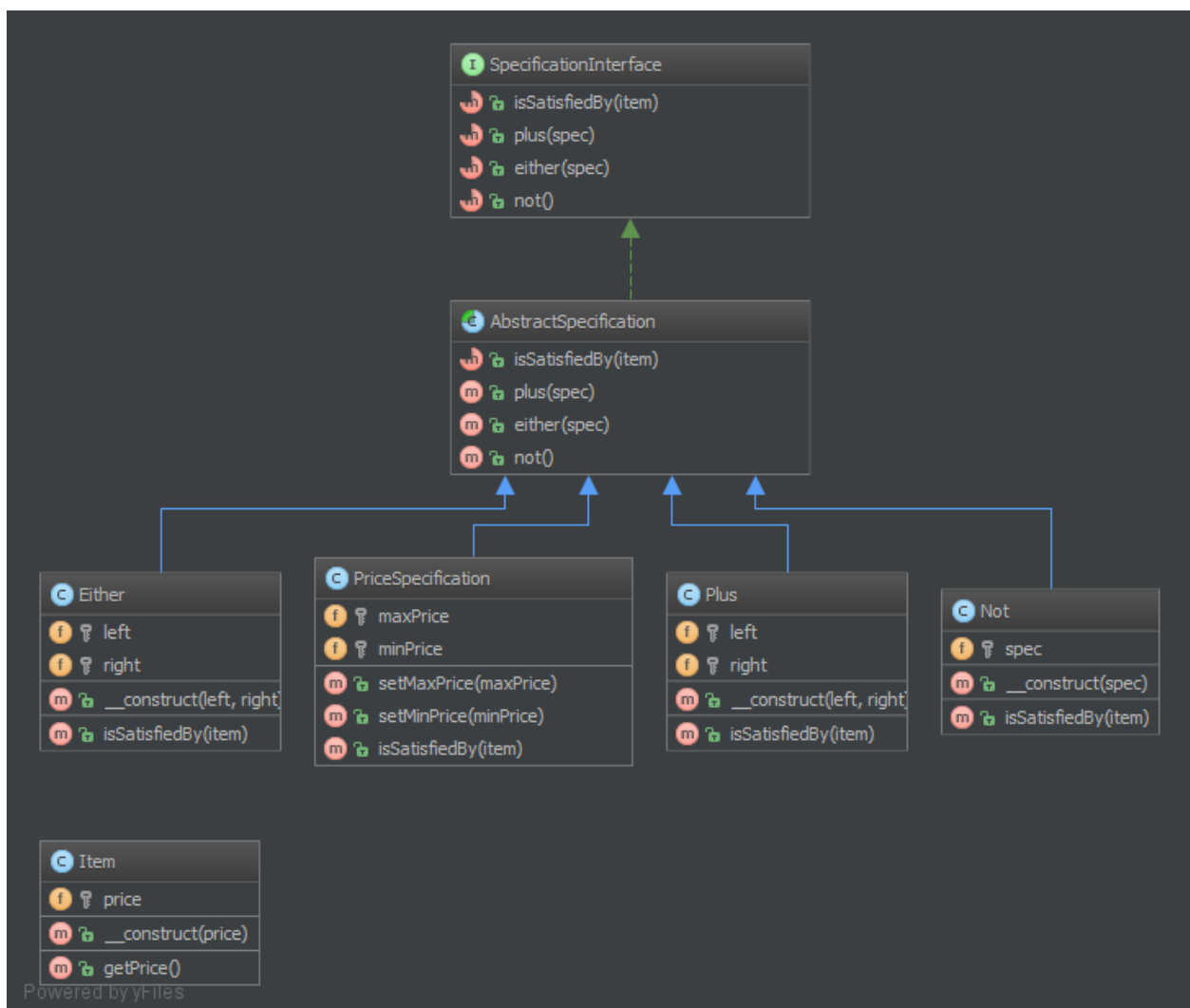
Purpose

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

```

1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * An trivial item
6   */
7  class Item
8  {
9      protected $price;
10
11      /**
12       * An item must have a price
13       *
14       * @param int $price
15       */
16      public function __construct($price)
17      {
18          $this->price = $price;
19      }
20
21      /**
22       * Get the items price
23       *
24       * @return int
25       */
26      public function getPrice()
27      {
28          return $this->price;
29      }
30  }
```

SpecificationInterface.php

```

1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * An interface for a specification
6   */
7  interface SpecificationInterface
8  {
9      /**
10       * A boolean evaluation indicating if the object meets the specification
11       *
12       * @param Item $item
13       *
14       * @return bool
15       */
16      public function isSatisfiedBy(Item $item);
17
18      /**
19       * Creates a logical AND specification
20       *
21       */
22  }
```

```
21     * @param SpecificationInterface $spec
22     */
23     public function plus(SpecificationInterface $spec);
24
25     /**
26     * Creates a logical OR specification
27     *
28     * @param SpecificationInterface $spec
29     */
30     public function either(SpecificationInterface $spec);
31
32     /**
33     * Creates a logical not specification
34     */
35     public function not();
36 }
```

AbstractSpecification.php

```
1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * An abstract specification allows the creation of wrapped specifications
6   */
7  abstract class AbstractSpecification implements SpecificationInterface
8  {
9      /**
10       * Checks if given item meets all criteria
11       *
12       * @param Item $item
13       *
14       * @return bool
15       */
16       abstract public function isSatisfiedBy(Item $item);
17
18       /**
19       * Creates a new logical AND specification
20       *
21       * @param SpecificationInterface $spec
22       *
23       * @return SpecificationInterface
24       */
25       public function plus(SpecificationInterface $spec)
26       {
27           return new Plus($this, $spec);
28       }
29
30       /**
31       * Creates a new logical OR composite specification
32       *
33       * @param SpecificationInterface $spec
34       *
35       * @return SpecificationInterface
36       */
37       public function either(SpecificationInterface $spec)
38       {
39           return new Either($this, $spec);
```



```

40     }
41
42     /**
43      * Creates a new logical NOT specification
44      *
45      * @return SpecificationInterface
46      */
47     public function not()
48     {
49         return new Not($this);
50     }
51 }

```

Either.php

```

1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * A logical OR specification
6   */
7  class Either extends AbstractSpecification
8  {
9
10     protected $left;
11     protected $right;
12
13     /**
14      * A composite wrapper of two specifications
15      *
16      * @param SpecificationInterface $left
17      * @param SpecificationInterface $right
18      */
19     public function __construct(SpecificationInterface $left, SpecificationInterface $right)
20     {
21         $this->left = $left;
22         $this->right = $right;
23     }
24
25     /**
26      * Returns the evaluation of both wrapped specifications as a logical OR
27      *
28      * @param Item $item
29      *
30      * @return bool
31      */
32     public function isSatisfiedBy(Item $item)
33     {
34         return $this->left->isSatisfiedBy($item) || $this->right->isSatisfiedBy($item);
35     }
36 }

```

PriceSpecification.php

```

1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * A specification to check an Item is priced between min and max

```

```
6  */
7  class PriceSpecification extends AbstractSpecification
8  {
9      protected $maxPrice;
10     protected $minPrice;
11
12     /**
13      * Sets the optional maximum price
14      *
15      * @param int $maxPrice
16      */
17     public function setMaxPrice($maxPrice)
18     {
19         $this->maxPrice = $maxPrice;
20     }
21
22     /**
23      * Sets the optional minimum price
24      *
25      * @param int $minPrice
26      */
27     public function setMinPrice($minPrice)
28     {
29         $this->minPrice = $minPrice;
30     }
31
32     /**
33      * Checks if Item price falls between bounds
34      *
35      * @param Item $item
36      *
37      * @return bool
38      */
39     public function isSatisfiedBy(Item $item)
40     {
41         if (!empty($this->maxPrice) && $item->getPrice() > $this->maxPrice) {
42             return false;
43         }
44         if (!empty($this->minPrice) && $item->getPrice() < $this->minPrice) {
45             return false;
46         }
47
48         return true;
49     }
50 }
```

Plus.php

```
1  <?php
2  namespace DesignPatterns\Behavioral\Specification;
3
4  /**
5   * A logical AND specification
6   */
7  class Plus extends AbstractSpecification
8  {
9
10     protected $left;
11     protected $right;
```

```

12
13     /**
14      * Creation of a locical AND of two specifications
15      *
16      * @param SpecificationInterface $left
17      * @param SpecificationInterface $right
18      */
19     public function __construct(SpecificationInterface $left, SpecificationInterface $right)
20     {
21         $this->left = $left;
22         $this->right = $right;
23     }
24
25     /**
26      * Checks if the composite AND of specifications passes
27      *
28      * @param Item $item
29      *
30      * @return bool
31      */
32     public function isSatisfiedBy(Item $item)
33     {
34         return $this->left->isSatisfiedBy($item) && $this->right->isSatisfiedBy($item);
35     }
36 }

```

Not.php

```

1 <?php
2 namespace DesignPatterns\Behavioral\Specification;
3
4 /**
5  * A logical Not specification
6  */
7 class Not extends AbstractSpecification
8 {
9
10     protected $spec;
11
12     /**
13      * Creates a new specification wrapping another
14      *
15      * @param SpecificationInterface $spec
16      */
17     public function __construct(SpecificationInterface $spec)
18     {
19         $this->spec = $spec;
20     }
21
22     /**
23      * Returns the negated result of the wrapped specification
24      *
25      * @param Item $item
26      *
27      * @return bool
28      */
29     public function isSatisfiedBy(Item $item)
30     {
31         return !$this->spec->isSatisfiedBy($item);
32     }
33 }

```

```
32     }
33 }
```

Test

Tests/SpecificationTest.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\Specification\Tests;
4
5  use DesignPatterns\Behavioral\Specification\PriceSpecification;
6  use DesignPatterns\Behavioral\Specification\Item;
7
8  /**
9   * SpecificationTest tests the specification pattern
10  */
11  class SpecificationTest extends \PHPUnit_Framework_TestCase
12  {
13      public function testSimpleSpecification()
14      {
15          $item = new Item(100);
16          $spec = new PriceSpecification();
17
18          $this->assertTrue($spec->isSatisfiedBy($item));
19
20          $spec->setMaxPrice(50);
21          $this->assertFalse($spec->isSatisfiedBy($item));
22
23          $spec->setMaxPrice(150);
24          $this->assertTrue($spec->isSatisfiedBy($item));
25
26          $spec->setMinPrice(101);
27          $this->assertFalse($spec->isSatisfiedBy($item));
28
29          $spec->setMinPrice(100);
30          $this->assertTrue($spec->isSatisfiedBy($item));
31      }
32
33      public function testNotSpecification()
34      {
35          $item = new Item(100);
36          $spec = new PriceSpecification();
37          $not = $spec->not();
38
39          $this->assertFalse($not->isSatisfiedBy($item));
40
41          $spec->setMaxPrice(50);
42          $this->assertTrue($not->isSatisfiedBy($item));
43
44          $spec->setMaxPrice(150);
45          $this->assertFalse($not->isSatisfiedBy($item));
46
47          $spec->setMinPrice(101);
48          $this->assertTrue($not->isSatisfiedBy($item));
49
50          $spec->setMinPrice(100);
```

```

51     $this->assertFalse($not->isSatisfiedBy($item));
52 }
53
54 public function testPlusSpecification()
55 {
56     $spec1 = new PriceSpecification();
57     $spec2 = new PriceSpecification();
58     $plus = $spec1->plus($spec2);
59
60     $item = new Item(100);
61
62     $this->assertTrue($plus->isSatisfiedBy($item));
63
64     $spec1->setMaxPrice(150);
65     $spec2->setMinPrice(50);
66     $this->assertTrue($plus->isSatisfiedBy($item));
67
68     $spec1->setMaxPrice(150);
69     $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 }

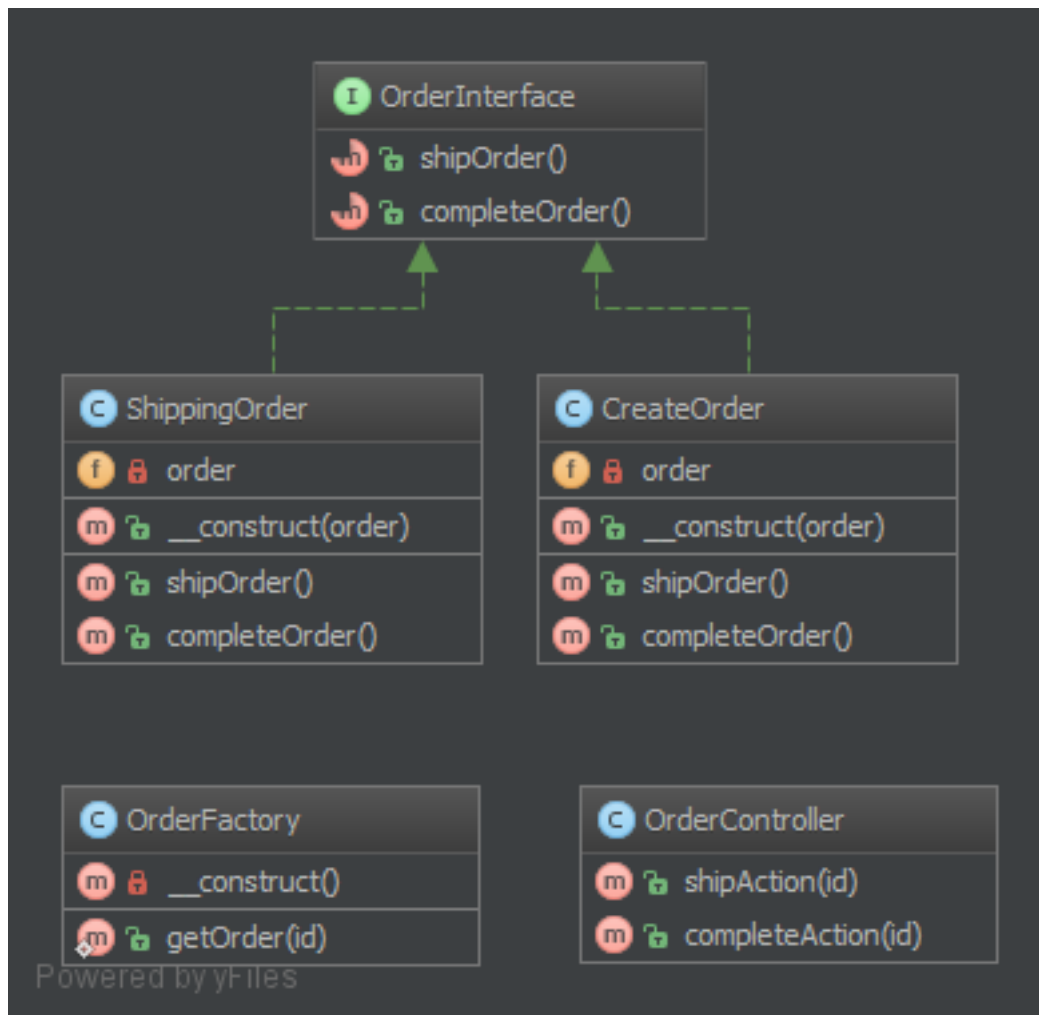
```

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

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

```

6  * Class OrderController
7  */
8  class OrderController
9  {
10     /**
11      * @param int $id
12      */
13     public function shipAction($id)
14     {
15         $order = OrderFactory::getOrder($id);
16         try {
17             $order->shipOrder();
18         } catch (Exception $e) {
19             //handle error!
20         }
21         // response to browser
22     }
23
24     /**
25      * @param int $id
26      */
27     public function completeAction($id)
28     {
29         $order = OrderFactory::getOrder($id);
30         try {
31             $order->completeOrder();
32         } catch (Exception $e) {
33             //handle error!
34         }
35         // response to browser
36     }
37 }

```

OrderFactory.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\State;
4
5  /**
6   * Class OrderFactory
7   */
8  class OrderFactory
9  {
10     private function __construct()
11     {
12         throw new \Exception('Can not instance the OrderFactory class!');
13     }
14
15     /**
16      * @param int $id
17      *
18      * @return CreateOrder|ShippingOrder
19      * @throws \Exception
20      */
21     public static function getOrder($id)
22     {
23         $order = 'Get Order From Database';
24     }

```

```
25         switch ($order['status']) {
26             case 'created':
27                 return new CreateOrder($order);
28             case 'shipping':
29                 return new ShippingOrder($order);
30             default:
31                 throw new \Exception('Order status error!');
32                 break;
33         }
34     }
35 }
```

OrderInterface.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\State;
4
5  /**
6   * Class OrderInterface
7   */
8  interface OrderInterface
9  {
10     /**
11      * @return mixed
12      */
13     public function shipOrder();
14
15     /**
16      * @return mixed
17      */
18     public function completeOrder();
19 }
```

ShippingOrder.php

```
1  <?php
2
3  namespace DesignPatterns\Behavioral\State;
4
5  /**
6   * Class ShippingOrder
7   */
8  class ShippingOrder implements OrderInterface
9  {
10     /**
11      * @var array
12      */
13     private $order;
14
15     /**
16      * @param array $order
17      *
18      * @throws \Exception
19      */
20     public function __construct(array $order)
21     {
22         if (empty($order)) {
23             throw new \Exception('Order can not be empty!');
```



```

24     }
25     $this->order = $order;
26 }
27
28 /**
29  * @return mixed|void
30  * @throws \Exception
31  */
32 public function shipOrder()
33 {
34     //Can not ship the order which status is shipping, throw exception;
35     throw new \Exception('Can not ship the order which status is shipping!');
36 }
37
38 /**
39  * @return mixed
40  */
41 public function completeOrder()
42 {
43     $this->order['status'] = 'completed';
44     $this->order['updatedAt'] = time();
45
46     // Setting the new order status into database;
47     return $this->updateOrder($this->order);
48 }
49 }

```

CreateOrder.php

```

1 <?php
2
3 namespace DesignPatterns\Behavioral\State;
4
5 /**
6  * Class CreateOrder
7  */
8 class CreateOrder implements OrderInterface
9 {
10     /**
11      * @var array
12      */
13     private $order;
14
15     /**
16      * @param array $order
17      *
18      * @throws \Exception
19      */
20     public function __construct(array $order)
21     {
22         if (empty($order)) {
23             throw new \Exception('Order can not be empty!');
24         }
25         $this->order = $order;
26     }
27
28     /**
29      * @return mixed
30      */

```

```
31 public function shipOrder()  
32 {  
33     $this->order['status'] = 'shipping';  
34     $this->order['updatedAt'] = time();  
35  
36     // Setting the new order status into database;  
37     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

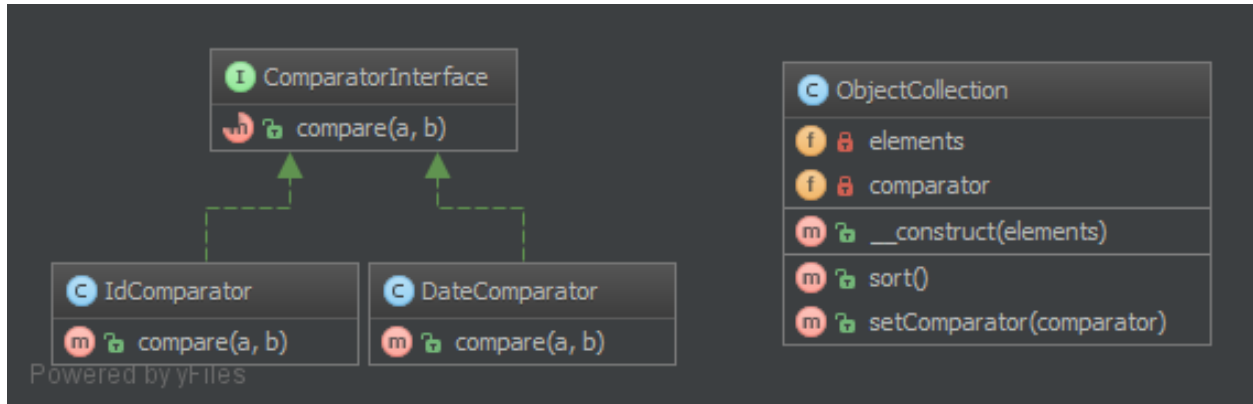
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Strategy;
4
5  /**
6   * Class ObjectCollection
7   */
8  class ObjectCollection
9  {
10     /**
11      * @var array
12      */
13     private $elements;
14
15     /**
16      * @var ComparatorInterface
17      */
18     private $comparator;
19
20     /**
21      * @param array $elements
22      */
23     public function __construct(array $elements = array())
24     {
25         $this->elements = $elements;
26     }
27
28     /**
29      * @return array
30      */
31     public function sort()
32     {
33         if (!$this->comparator) {
34             throw new \LogicException("Comparator is not set");
35         }
36     }
37 }

```

```
36         $callback = array($this->comparator, 'compare');
37         uasort($this->elements, $callback);
38
39         return $this->elements;
40     }
41
42
43     /**
44      * @param ComparatorInterface $comparator
45      *
46      * @return void
47      */
48     public function setComparator(ComparatorInterface $comparator)
49     {
50         $this->comparator = $comparator;
51     }
52 }
```

ComparatorInterface.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\Strategy;
4
5 /**
6  * Class ComparatorInterface
7  */
8 interface ComparatorInterface
9 {
10     /**
11      * @param mixed $a
12      * @param mixed $b
13      *
14      * @return bool
15      */
16     public function compare($a, $b);
17 }
```

DateComparator.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\Strategy;
4
5 /**
6  * Class DateComparator
7  */
8 class DateComparator implements ComparatorInterface
9 {
10     /**
11      * {@inheritdoc}
12      */
13     public function compare($a, $b)
14     {
15         $aDate = new \DateTime($a['date']);
16         $bDate = new \DateTime($b['date']);
17
18         if ($aDate == $bDate) {
19             return 0;
20         }
21     }
22 }
```

```

20         } else {
21             return $aDate < $bDate ? -1 : 1;
22         }
23     }
24 }

```

IdComparator.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Strategy;
4
5  /**
6   * Class IdComparator
7   */
8  class IdComparator implements ComparatorInterface
9  {
10     /**
11      * {@inheritdoc}
12      */
13     public function compare($a, $b)
14     {
15         if ($a['id'] == $b['id']) {
16             return 0;
17         } else {
18             return $a['id'] < $b['id'] ? -1 : 1;
19         }
20     }
21 }

```

Test

Tests/StrategyTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Strategy\Tests;
4
5  use DesignPatterns\Behavioral\Strategy\DateComparator;
6  use DesignPatterns\Behavioral\Strategy\IdComparator;
7  use DesignPatterns\Behavioral\Strategy\ObjectCollection;
8  use DesignPatterns\Behavioral\Strategy\Strategy;
9
10 /**
11  * Tests for Strategy pattern
12  */
13 class StrategyTest extends \PHPUnit_Framework_TestCase
14 {
15
16     public function getIdCollection()
17     {
18         return array(
19             array(
20                 array('id' => 2), array('id' => 1), array('id' => 3)),
21                 array('id' => 1)
22             ),
23             array(

```

```
24         array(array('id' => 3), array('id' => 2), array('id' => 1)),
25         array('id' => 1)
26     ),
27 );
28 }
29
30 public function getDateCollection()
31 {
32     return array(
33         array(
34             array(array('date' => '2014-03-03'), array('date' => '2015-03-02'), array('date' =>
35             array('date' => '2013-03-01')
36         ),
37         array(
38             array(array('date' => '2014-02-03'), array('date' => '2013-02-01'), array('date' =>
39             array('date' => '2013-02-01')
40         ),
41     );
42 }
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.

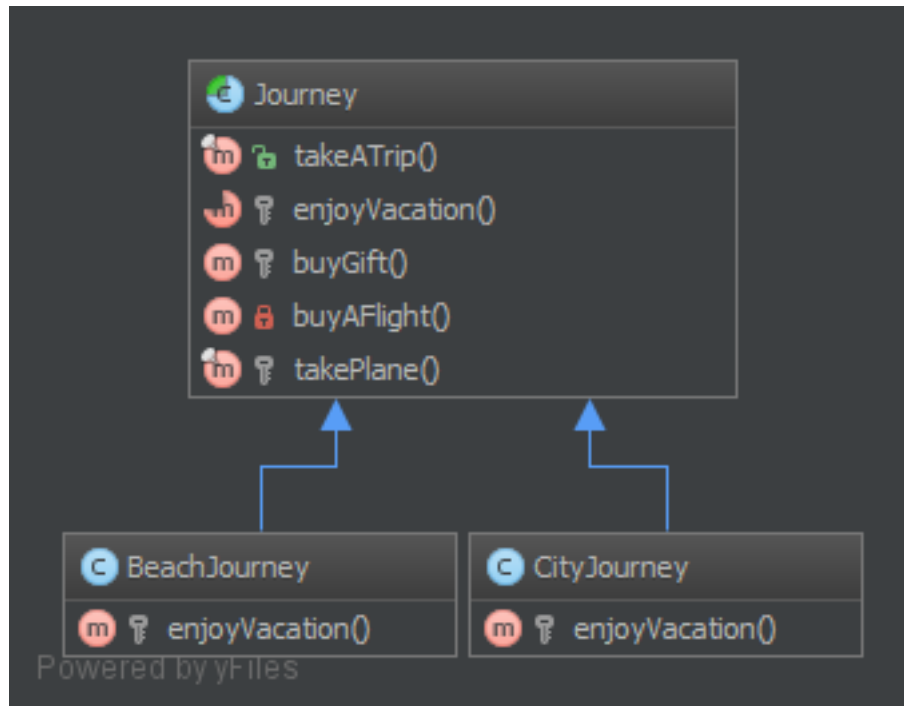
A.k.a the “Hollywood principle”: “Don’t call us, we call you.” This class is not called by subclasses but the inverse.

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

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\TemplateMethod;
4
5  /**
6   *
7   */
8  abstract class Journey
9  {
10     /**
11      * This is the public service provided by this class and its subclasses.
12      * Notice it is final to "freeze" the global behavior of algorithm.
13      * If you want to override this contract, make an interface with only takeATrip()
14      * and subclass it.
15      */
16     final public function takeATrip()
17     {

```

```
18         $this->buyAFlight();
19         $this->takePlane();
20         $this->enjoyVacation();
21         $this->buyGift();
22         $this->takePlane();
23     }
24
25     /**
26      * This method must be implemented, this is the key-feature of this pattern
27      */
28     abstract protected function enjoyVacation();
29
30     /**
31      * This method is also part of the algorithm but it is optional.
32      * This is an "adapter" (do not confuse with the Adapter pattern, not related)
33      * You can override it only if you need to.
34      */
35     protected function buyGift()
36     {
37     }
38
39     /**
40      * This method will be unknown by subclasses (better)
41      */
42     private function buyAFlight()
43     {
44         echo "Buying a flight\n";
45     }
46
47     /**
48      * Subclasses will get access to this method but cannot override it and
49      * compromise this algorithm (warning : cause of cyclic dependencies)
50      */
51     final protected function takePlane()
52     {
53         echo "Taking the plane\n";
54     }
55
56     // A note regarding the keyword "final" : don't use it when you start coding :
57     // add it after you narrow and know exactly what change and what remain unchanged
58     // in this algorithm.
59     // [abstract] x [3 access] x [final] = 12 combinations, it can be hard !
60 }
```

BeachJourney.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\TemplateMethod;
4
5 /**
6  * BeachJourney is vacation at the beach
7  */
8 class BeachJourney extends Journey
9 {
10     /**
11      * prints what to do to enjoy your vacation
12      */
```



```

13     protected function enjoyVacation()
14     {
15         echo "Swimming and sun-bathing\n";
16     }
17 }

```

CityJourney.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\TemplateMethod;
4
5  /**
6   * CityJourney is a journey in a city
7   */
8  class CityJourney extends Journey
9  {
10     /**
11      * prints what to do in your journey to enjoy vacation
12      */
13     protected function enjoyVacation()
14     {
15         echo "Eat, drink, take photos and sleep\n";
16     }
17 }

```

Test

Tests/JourneyTest.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\TemplateMethod\Tests;
4
5  use DesignPatterns\Behavioral\TemplateMethod;
6
7  /**
8   * JourneyTest tests all journeys
9   */
10 class JourneyTest extends \PHPUnit_Framework_TestCase
11 {
12
13     public function testBeach()
14     {
15         $journey = new TemplateMethod\BeachJourney();
16         $this->expectOutputRegex('#sun-bathing#');
17         $journey->takeATrip();
18     }
19
20     public function testCity()
21     {
22         $journey = new TemplateMethod\CityJourney();
23         $this->expectOutputRegex('#drink#');
24         $journey->takeATrip();
25     }
26
27     /**

```

```

28      * How to test an abstract template method with PHPUnit
29      */
30      public function testLasVegas()
31      {
32          $journey = $this->getMockForAbstractClass('DesignPatterns\Behavioral\TemplateMethod\Journey');
33          $journey->expects($this->once())
34              ->method('enjoyVacation')
35              ->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  }

```

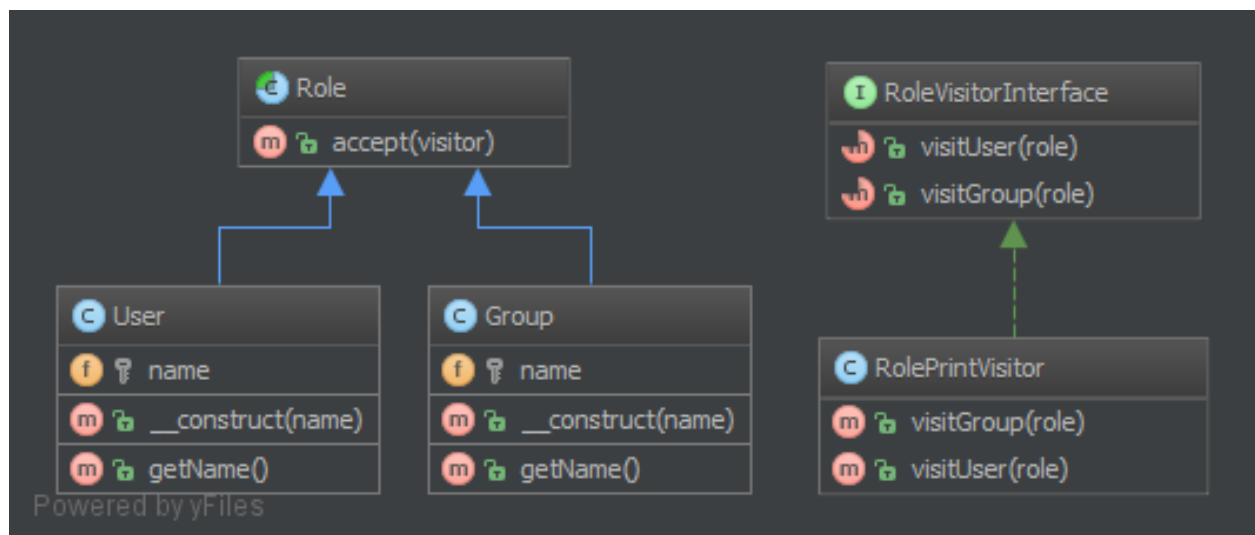
1.3.12 Visitor

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
7   *
8   * The contract for the visitor.
9   *
10  * Note 1 : in C++ or java, with method polymorphism based on type-hint, there are many
11  * methods visit() with different type for the 'role' parameter.
12  *
13  * Note 2 : the visitor must not choose itself which method to
14  * invoke, it is the Visitee that make this decision.
15  */
16  interface RoleVisitorInterface
17  {
18      /**
19       * Visit a User object
20       *
21       * @param \DesignPatterns\Behavioral\Visitor\User $role
22       */
23      public function visitUser(User $role);
24
25      /**
26       * Visit a Group object
27       *
28       * @param \DesignPatterns\Behavioral\Visitor\Group $role
29       */
30      public function visitGroup(Group $role);
31  }

```

RolePrintVisitor.php

```

1  <?php
2
3  namespace DesignPatterns\Behavioral\Visitor;
4
5  /**
6   * Visitor Pattern
7   *
8   * An implementation of a concrete Visitor
9   */
10  class RolePrintVisitor implements RoleVisitorInterface
11  {
12      /**
13       * {@inheritdoc}
14       */
15      public function visitGroup(Group $role)
16      {
17          echo "Role: " . $role->getName();
18      }
19
20      /**
21       * {@inheritdoc}
22       */
23      public function visitUser(User $role)

```

```
24 {
25     echo "Role: " . $role->getName();
26 }
27 }
```

Role.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\Visitor;
4
5 /**
6  * class Role
7  */
8 abstract class Role
9 {
10     /**
11      * This method handles a double dispatch based on the short name of the Visitor
12      *
13      * Feel free to override it if your object must call another visiting behavior
14      *
15      * @param \DesignPatterns\Behavioral\Visitor\RoleVisitorInterface $visitor
16      *
17      * @throws \InvalidArgumentException
18      */
19     public function accept(RoleVisitorInterface $visitor)
20     {
21         // this trick to simulate double-dispatch based on type-hinting
22         $klass = get_called_class();
23         preg_match('#([^\s\|]+)$#', $klass, $extract);
24         $visitingMethod = 'visit' . $extract[1];
25
26         // this ensures strong typing with visitor interface, not some visitor objects
27         if (!method_exists(__NAMESPACE__ . '\RoleVisitorInterface', $visitingMethod)) {
28             throw new \InvalidArgumentException("The visitor you provide cannot visit a $klass instance");
29         }
30
31         call_user_func(array($visitor, $visitingMethod), $this);
32     }
33 }
```

User.php

```
1 <?php
2
3 namespace DesignPatterns\Behavioral\Visitor;
4
5 /**
6  * Visitor Pattern
7  *
8  * One example for a visatee. Each visatee has to extends Role
9  */
10 class User extends Role
11 {
12     /**
13      * @var string
14      */
15     protected $name;
16 }
```

```

17  /**
18   * @param string $name
19   */
20  public function __construct($name)
21  {
22      $this->name = (string) $name;
23  }
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

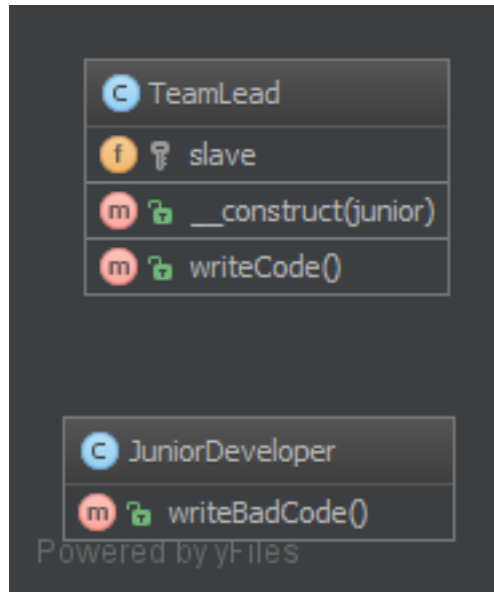
Purpose

...

Examples

...

UML Diagram



Code

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

Usage.php

```

1 <?php
2
3 namespace DesignPatterns\More\Delegation;
4
5 // instantiate TeamLead and appoint to assistants JuniorDeveloper
6 $teamLead = new TeamLead(new JuniorDeveloper());
7
8 // team lead delegate write code to junior developer
9 echo $teamLead->writeCode();
  
```

TeamLead.php

```

1 <?php
2
3 namespace DesignPatterns\More\Delegation;
4
5 /**
6  * Class TeamLead
7  * @package DesignPatterns\Delegation
8  * The `TeamLead` class, he delegate work to `JuniorDeveloper`
9  */
10 class TeamLead
11 {
12     /** @var JuniorDeveloper */
  
```

```
13     protected $slave;
14
15     /**
16      * Give junior developer into teamlead submission
17      * @param JuniorDeveloper $junior
18      */
19     public function __construct(JuniorDeveloper $junior)
20     {
21         $this->slave = $junior;
22     }
23
24     /**
25      * TeamLead drink coffee, junior work
26      * @return mixed
27      */
28     public function writeCode()
29     {
30         return $this->slave->writeBadCode();
31     }
32 }
```

JuniorDeveloper.php

```
1 <?php
2
3 namespace DesignPatterns\More\Delegation;
4
5 /**
6  * Class JuniorDeveloper
7  * @package DesignPatterns\Delegation
8  */
9 class JuniorDeveloper
10 {
11     public function writeBadCode()
12     {
13         return "Some junior developer generated code...";
14     }
15 }
```

Test

Tests/DelegationTest.php

```
1 <?php
2
3 namespace DesignPatterns\More\Delegation\Tests;
4
5 use DesignPatterns\More\Delegation;
6
7 /**
8  * DelegationTest tests the delegation pattern
9  */
10 class DelegationTest extends \PHPUnit_Framework_TestCase
11 {
12     public function testHowTeamLeadWriteCode()
13     {
14         $junior = new Delegation\JuniorDeveloper();
```



```
15     $teamLead = new Delegation\TeamLead($junior);
16     $this->assertEquals($junior->writeBadCode(), $teamLead->writeCode());
17 }
18 }
```

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.

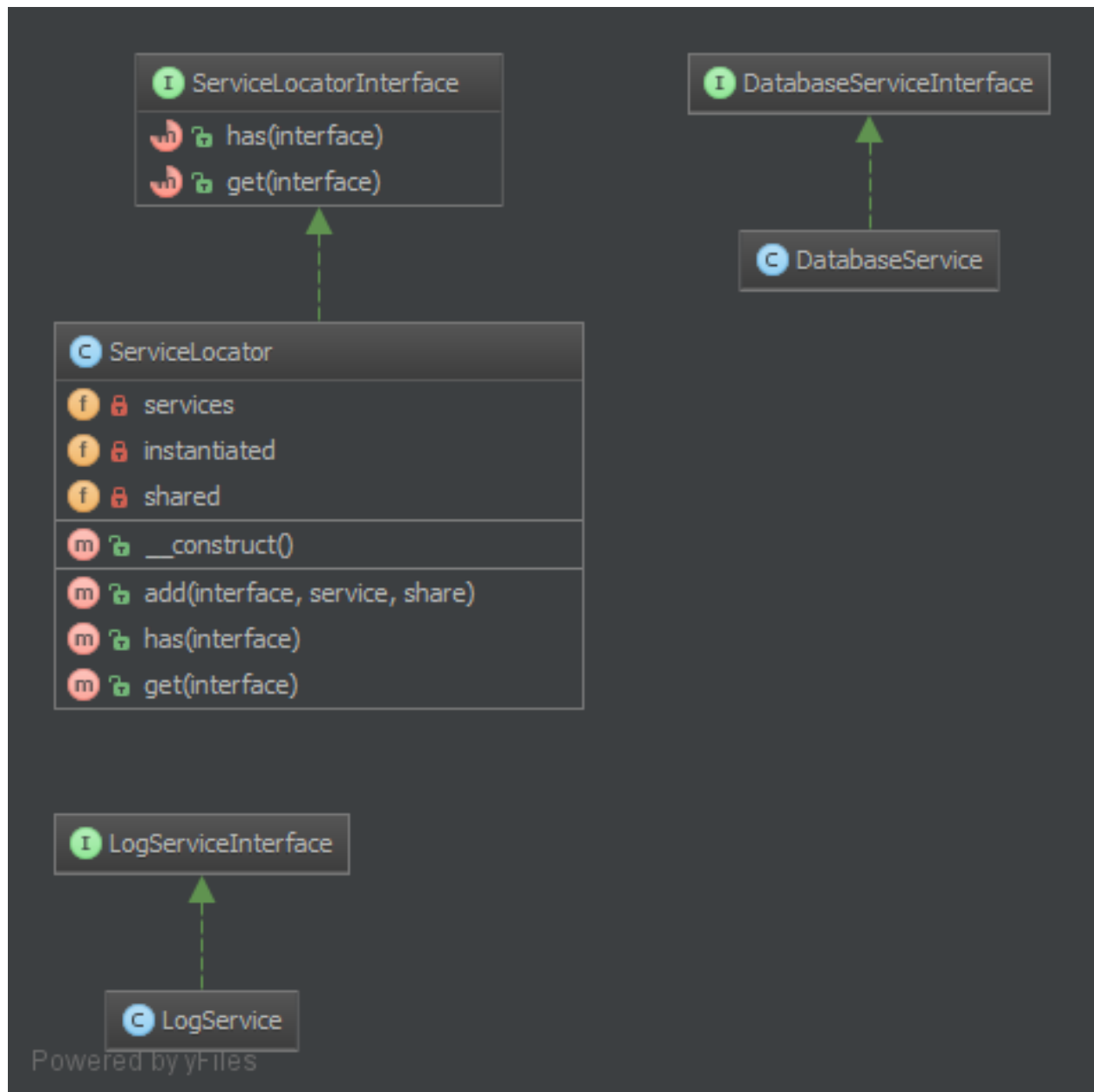
Usage

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

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  interface ServiceLocatorInterface
6  {

```

```

7  /**
8   * Checks if a service is registered.
9   *
10  * @param string $interface
11  *
12  * @return bool
13  */
14  public function has($interface);
15
16  /**
17   * Gets the service registered for the interface.
18   *
19   * @param string $interface
20   *
21   * @return mixed
22   */
23  public function get($interface);
24  }

```

ServiceLocator.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  class ServiceLocator implements ServiceLocatorInterface
6  {
7      /**
8       * All services.
9       *
10      * @var array
11      */
12      private $services;
13
14      /**
15       * The services which have an instance.
16       *
17       * @var array
18       */
19      private $instantiated;
20
21      /**
22       * True if a service can be shared.
23       *
24       * @var array
25       */
26      private $shared;
27
28      public function __construct()
29      {
30          $this->services      = array();
31          $this->instantiated = array();
32          $this->shared       = array();
33      }
34
35      /**
36       * Registers a service with specific interface.
37       *
38       * @param string $interface

```

```

39  * @param string/object $service
40  * @param bool $share
41  */
42  public function add($interface, $service, $share = true)
43  {
44      /**
45       * When you add a service, you should register it
46       * with its interface or with a string that you can use
47       * in the future even if you will change the service implementation.
48       */
49
50      if (is_object($service) && $share) {
51          $this->instantiated[$interface] = $service;
52      }
53      $this->services[$interface] = (is_object($service) ? get_class($service) : $service);
54      $this->shared[$interface] = $share;
55  }
56
57  /**
58   * Checks if a service is registered.
59   *
60   * @param string $interface
61   *
62   * @return bool
63   */
64  public function has($interface)
65  {
66      return (isset($this->services[$interface]) || isset($this->instantiated[$interface]));
67  }
68
69  /**
70   * Gets the service registered for the interface.
71   *
72   * @param string $interface
73   *
74   * @return mixed
75   */
76  public function get($interface)
77  {
78      // Retrieves the instance if it exists and it is shared
79      if (isset($this->instantiated[$interface]) && $this->shared[$interface]) {
80          return $this->instantiated[$interface];
81      }
82
83      // otherwise gets the service registered.
84      $service = $this->services[$interface];
85
86      // You should check if the service class exists and
87      // the class is instantiable.
88
89      // This example is a simple implementation, but
90      // when you create a service, you can decide
91      // if $service is a factory or a class.
92      // By registering a factory you can create your services
93      // using the DependencyInjection pattern.
94
95      // ...
96

```

```

97     // Creates the service object
98     $object = new $service();
99
100    // and saves it if the service must be shared.
101    if ($this->shared[$interface]) {
102        $this->instantiated[$interface] = $object;
103    }
104    return $object;
105 }
106 }

```

LogServiceInterface.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  interface LogServiceInterface
6  {
7  }

```

LogService.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  class LogService implements LogServiceInterface
6  {
7  }

```

DatabaseServiceInterface.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  interface DatabaseServiceInterface
6  {
7  }

```

DatabaseService.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator;
4
5  class DatabaseService implements DatabaseServiceInterface
6  {
7  }

```

Test

Tests/ServiceLocatorTest.php

```

1  <?php
2
3  namespace DesignPatterns\More\ServiceLocator\Tests;

```

```
4
5 use DesignPatterns\More\ServiceLocator\DatabaseService;
6 use DesignPatterns\More\ServiceLocator\LogService;
7 use DesignPatterns\More\ServiceLocator\ServiceLocator;
8 use \PHPUnit\Framework\TestCase as TestCase;
9
10 class ServiceLocatorTest extends TestCase
11 {
12     /**
13      * @var LogService
14      */
15     private $logService;
16
17     /**
18      * @var DatabaseService
19      */
20     private $databaseService;
21
22     /**
23      * @var ServiceLocator
24      */
25     private $serviceLocator;
26
27     public function setUp()
28     {
29         $this->serviceLocator = new ServiceLocator();
30         $this->logService     = new LogService();
31         $this->databaseService = new DatabaseService();
32     }
33
34     public function testHasServices()
35     {
36         $this->serviceLocator->add(
37             'DesignPatterns\More\ServiceLocator\LogServiceInterface',
38             $this->logService
39         );
40
41         $this->serviceLocator->add(
42             'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
43             $this->databaseService
44         );
45
46         $this->assertTrue($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\LogServiceInterface'));
47         $this->assertTrue($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface'));
48
49         $this->assertFalse($this->serviceLocator->has('DesignPatterns\More\ServiceLocator\FakeServiceInterface'));
50     }
51
52     public function testServicesWithObject()
53     {
54         $this->serviceLocator->add(
55             'DesignPatterns\More\ServiceLocator\LogServiceInterface',
56             $this->logService
57         );
58
59         $this->serviceLocator->add(
60             'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
61             $this->databaseService
62         );
63     }
64 }
```

```

62         );
63
64         $this->assertSame(
65             $this->logService,
66             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
67         );
68
69         $this->assertSame(
70             $this->databaseService,
71             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
72         );
73     }
74
75     public function testServicesWithClass()
76     {
77         $this->serviceLocator->add(
78             'DesignPatterns\More\ServiceLocator\LogServiceInterface',
79             get_class($this->logService)
80         );
81
82         $this->serviceLocator->add(
83             'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
84             get_class($this->databaseService)
85         );
86
87         $this->assertNotSame(
88             $this->logService,
89             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
90         );
91
92         $this->assertInstanceOf(
93             'DesignPatterns\More\ServiceLocator\LogServiceInterface',
94             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\LogServiceInterface')
95         );
96
97         $this->assertNotSame(
98             $this->databaseService,
99             $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
100        );
101
102        $this->assertInstanceOf(
103            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
104            $this->serviceLocator->get('DesignPatterns\More\ServiceLocator\DatabaseServiceInterface')
105        );
106    }
107
108    public function testServicesNotShared()
109    {
110        $this->serviceLocator->add(
111            'DesignPatterns\More\ServiceLocator\LogServiceInterface',
112            $this->logService,
113            false
114        );
115
116        $this->serviceLocator->add(
117            'DesignPatterns\More\ServiceLocator\DatabaseServiceInterface',
118            $this->databaseService,
119            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

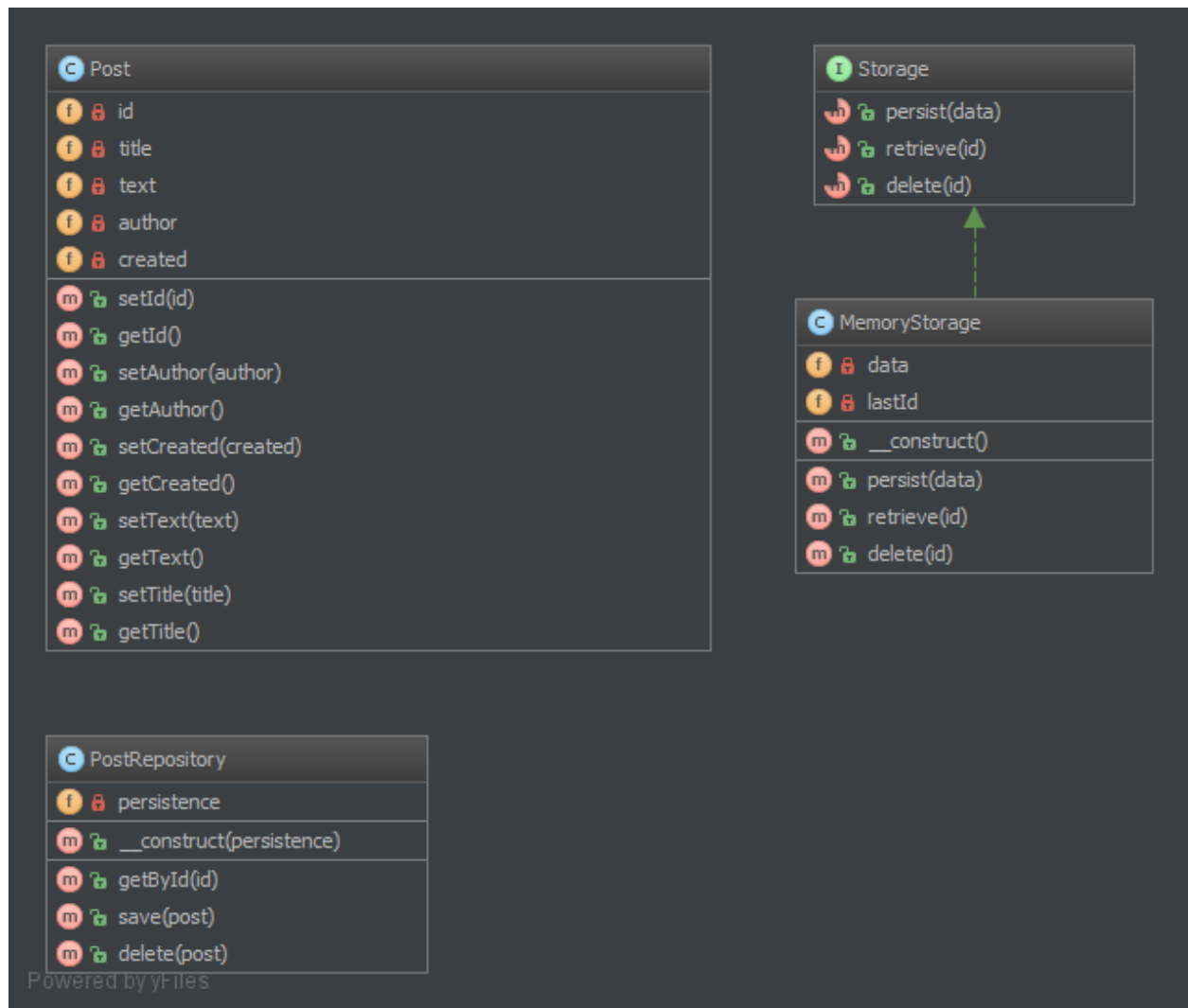
Purpose

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

```

1  <?php
2
3  namespace DesignPatterns\Repository;
4
5  /**
6   * Post represents entity for some post that user left on the site
7   *
8   * Class Post
9   * @package DesignPatterns\Repository
10  */
11  class Post
12  {

```

```
13  /**
14   * @var int
15   */
16  private $id;
17
18  /**
19   * @var string
20   */
21  private $title;
22
23  /**
24   * @var string
25   */
26  private $text;
27
28  /**
29   * @var string
30   */
31  private $author;
32
33  /**
34   * @var \DateTime
35   */
36  private $created;
37
38  /**
39   * @param int $id
40   */
41  public function setId($id)
42  {
43      $this->id = $id;
44  }
45
46  /**
47   * @return int
48   */
49  public function getId()
50  {
51      return $this->id;
52  }
53
54  /**
55   * @param string $author
56   */
57  public function setAuthor($author)
58  {
59      $this->author = $author;
60  }
61
62  /**
63   * @return string
64   */
65  public function getAuthor()
66  {
67      return $this->author;
68  }
69
70  /**
```

```

71     * @param \DateTime $created
72     */
73     public function setCreated($created)
74     {
75         $this->created = $created;
76     }
77
78     /**
79     * @return \DateTime
80     */
81     public function getCreated()
82     {
83         return $this->created;
84     }
85
86     /**
87     * @param string $text
88     */
89     public function setText($text)
90     {
91         $this->text = $text;
92     }
93
94     /**
95     * @return string
96     */
97     public function getText()
98     {
99         return $this->text;
100    }
101
102    /**
103    * @param string $title
104    */
105    public function setTitle($title)
106    {
107        $this->title = $title;
108    }
109
110    /**
111    * @return string
112    */
113    public function getTitle()
114    {
115        return $this->title;
116    }
117 }

```

PostRepository.php

```

1  <?php
2
3  namespace DesignPatterns\Repository;
4
5  /**
6   * Repository for class Post
7   * This class is between Entity layer(class Post) and access object layer(interface Storage)
8   */

```

```
9  * Repository encapsulates the set of objects persisted in a data store and the operations performed
10 * providing a more object-oriented view of the persistence layer
11 *
12 * Repository also supports the objective of achieving a clean separation and one-way dependency
13 * between the domain and data mapping layers
14 *
15 * Class PostRepository
16 * @package DesignPatterns\Repository
17 */
18 class PostRepository
19 {
20     private $persistence;
21
22     public function __construct(Storage $persistence)
23     {
24         $this->persistence = $persistence;
25     }
26
27     /**
28      * Returns Post object by specified id
29      *
30      * @param int $id
31      * @return Post|null
32      */
33     public function getById($id)
34     {
35         $arrayData = $this->persistence->retrieve($id);
36         if (is_null($arrayData)) {
37             return null;
38         }
39
40         $post = new Post();
41         $post->setId($arrayData['id']);
42         $post->setAuthor($arrayData['author']);
43         $post->setCreated($arrayData['created']);
44         $post->setText($arrayData['text']);
45         $post->setTitle($arrayData['title']);
46
47         return $post;
48     }
49
50     /**
51      * Save post object and populate it with id
52      *
53      * @param Post $post
54      * @return Post
55      */
56     public function save(Post $post)
57     {
58         $id = $this->persistence->persist(array(
59             'author' => $post->getAuthor(),
60             'created' => $post->getCreated(),
61             'text' => $post->getText(),
62             'title' => $post->getTitle()
63         ));
64
65         $post->setId($id);
66         return $post;
```

```

67     }
68
69     /**
70      * Deletes specified Post object
71      *
72      * @param Post $post
73      * @return bool
74      */
75     public function delete(Post $post)
76     {
77         return $this->persistence->delete($post->getId());
78     }
79 }

```

Storage.php

```

1  <?php
2
3  namespace DesignPatterns\Repository;
4
5  /**
6   * Interface Storage
7   *
8   * This interface describes methods for accessing storage.
9   * Concrete realization could be whatever we want - in memory, relational database, NoSQL database and so on.
10  *
11  * @package DesignPatterns\Repository
12  */
13  interface Storage
14  {
15      /**
16       * Method to persist data
17       * Returns new id for just persisted data
18       *
19       * @param array() $data
20       * @return int
21       */
22      public function persist($data);
23
24      /**
25       * Returns data by specified id.
26       * If there is no such data null is returned
27       *
28       * @param int $id
29       * @return array|null
30       */
31      public function retrieve($id);
32
33      /**
34       * Delete data specified by id
35       * If there is no such data - false returns, if data has been successfully deleted - true returns
36       *
37       * @param int $id
38       * @return bool
39       */
40      public function delete($id);
41  }

```

MemoryStorage.php

```
1 <?php
2
3 namespace DesignPatterns\Repository;
4
5 /**
6  * Class MemoryStorage
7  * @package DesignPatterns\Repository
8  */
9 class MemoryStorage implements Storage
10 {
11
12     private $data;
13     private $lastId;
14
15     public function __construct()
16     {
17         $this->data = array();
18         $this->lastId = 0;
19     }
20
21     /**
22      * {@inheritdoc}
23      */
24     public function persist($data)
25     {
26         $this->data[++$this->lastId] = $data;
27         return $this->lastId;
28     }
29
30     /**
31      * {@inheritdoc}
32      */
33     public function retrieve($id)
34     {
35         return isset($this->data[$id]) ? $this->data[$id] : null;
36     }
37
38     /**
39      * {@inheritdoc}
40      */
41     public function delete($id)
42     {
43         if (!isset($this->data[$id])) {
44             return false;
45         }
46
47         $this->data[$id] = null;
48         unset($this->data[$id]);
49
50         return true;
51     }
52 }
```

Test

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 ..`

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.