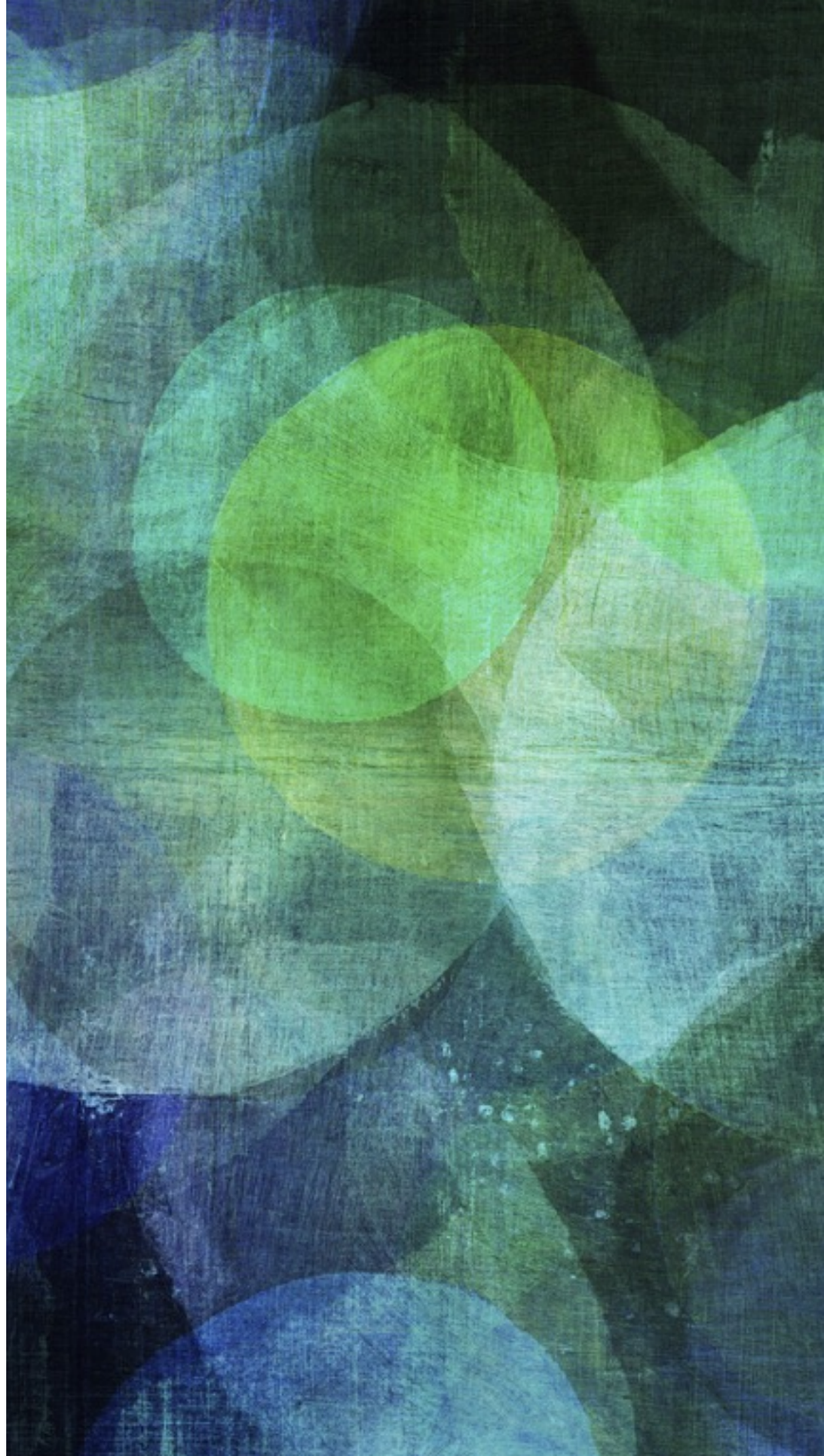
PHPUNIT

Tests pour PHP

1

Installation



INSTALLATION

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod +x phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
$ phpunit --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

- Necessite des extensions PHP normalement installées par défaut : dom, json, pire, reflection, sol
- PHP 5.6+

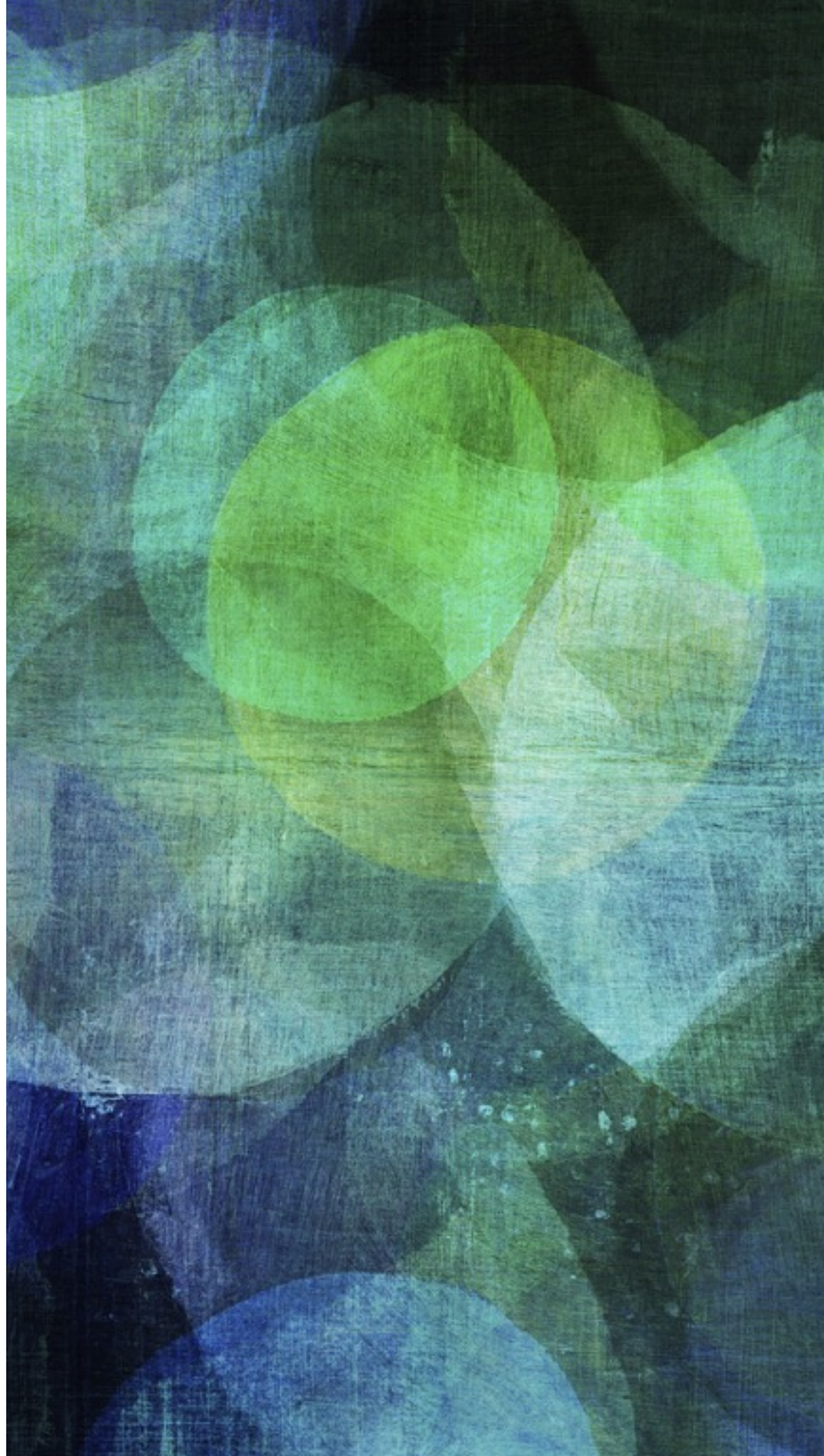
INSTALLATION XDEBUG

- Xdebug permet une visualisation plus riche des erreurs dans PHP
- A n'utiliser que sur des serveurs de développement
- Nécessite de redémarrer Apache

```
$ apt-get -y install php5-xdebug
```

2

Tests unitaires



DÉFINITION

- Un **test unitaire** est le test d'une classe ou des méthodes d'une classe indépendamment d'autres fonctionnalités

EXEMPLE

```php
// src/AppBundle/Util/Calculator.php
namespace AppBundle\Util;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

```php
// tests/AppBundle/Util/CalculatorTest.php
namespace Tests\AppBundle\Util;

use AppBundle\Util\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // assert that your calculator added the numbers correctly!
        $this->assertEquals(42, $result);
    }
}
```

EXECUTER LE TEST

```
# run all tests of the application
$ phpunit

# run all tests in the Util directory
$ phpunit tests/AppBundle/Util

# run tests for the Calculator class
$ phpunit tests/AppBundle/Util/CalculatorTest.php

# run all tests for the entire Bundle
$ phpunit tests/AppBundle/
```

CRÉER LA CLASSE DE TEST

- A toute classe d'un *bundle* Symfony, on peut faire correspondre une classe de test :

```
// src/AppBundle/Util/Calculator.php
namespace AppBundle\Util;

class Calculator
{
}
```

Les classes de tests ont leur propre répertoire à la racine de l'application

```
// tests/AppBundle/Util/CalculatorTest.php
namespace Tests\AppBundle\Util;

use AppBundle\Util\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
}
```

UNE MÉTHODE DE TEST UNITAIRE

1) Le test est déclaré par une annotation DocBlock ou...

2) Les méthodes de tests sont préfixées par mot-clef 'test'

```
/**
 * @test
 */
public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // assert that your calculator added the numbers correctly!
        $this->assertEquals(42, $result);
    }
```

Les résultats des tests sont déterminé par des assertions

DÉPENDANCES

- Une méthode de test peut dépendre d'une autre méthode et récupérer le résultat de cette dernière

```php
public function testEmpty()
{
    $stack = array();
    $this->assertEmpty($stack);

    return $stack;
}

/**
 * @depends testEmpty
 */
public function testPush(array $stack)
{
    array_push($stack, 'foo');
    $this->assertEquals('foo', $stack[count($stack)-1]);
    $this->assertNotEmpty($stack);

    return $stack;
}
```

1) testEmpty s'assure que l'on crée un tableau vide

2) La dépendance est déclarée entre les deux méthodes de test

3) la valeur de retour de testEmpty est injectée dans testPush

FOURNISSEUR DE DONNÉES

- Une méthode peut être déclarée comme fournissant des données à une méthode de test

1) On crée une source de données

```
public function additionProvider()
{
   $schema = [‘ajouter 1’ => [1,1,2], ‘ajouter 4’ => [2,4,6]];
   return $schema;
}

/**
 * @dataProvider additionProvider
 */
public function testAddition($x, $y, $resultat)
{
  $this->assertEquals($resultat, $x + $y);
}
```

2) On lie la source à la méthode de test

3) Les données sont utilisées comme paramètres du test

UTILISER UN ITÉRATEUR

- Une sous-classe d'Iterator peut tout à fait être une source de données (exploitation d'un fichier de données de tests)

```
class DataProducer extends Iterator
{
  /* ... code ... */
}
```

Implémentation des méthodes : current, next, rewind, key, etc.

```
public function additionProvider()
{
   return new DataProducer;
}

/**
 * @dataProvider additionProvider
 */
public function testAddition($x, $y, $resultat)
{
  $this->assertEquals($resultat, $x + $y);
}
```

PRIORITÉS

- Si l'on utilise à la fois @depends et @dataProvider, les sources de données sont passées dans la signature avant les dépendances

```
/**
   * @depends testProducerFirst
   * @depends testProducerSecond
   * @dataProvider provider
   */
public function testConsumer()
{
   $this->assertEquals(
      array('provider1', 'first', 'second'),
      func_get_args()
   );
}
```

La source de données est le premier paramètre indépendamment de l'ordre de déclaration

LES EXCEPTIONS

- Il est possible de tester les exceptions qui sont levées, en utilisant soit des méthodes soit des annotations

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

ou

```
/**
 * @expectedException InvalidArgumentException
 */
public function testException()
{
}
```

LES ERREURS PHP

- Il est possible de détecter les erreurs renvoyées par PHP par la « suppression d'erreur »

TESTER LA SORTIE

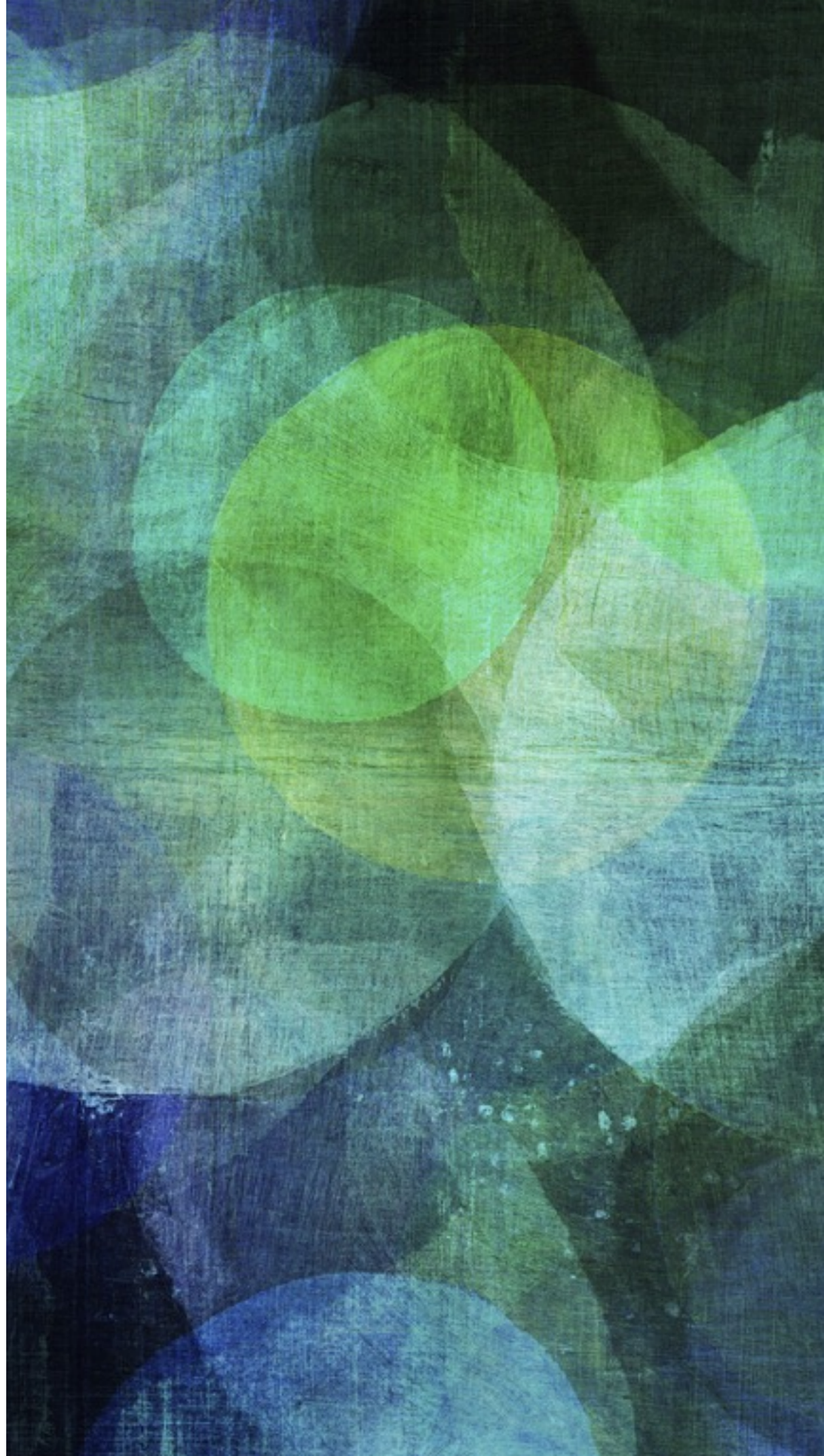
- Il est possible de tester la sortie avec la méthode expectOutputString

```
public function testExpectBarActualBaz()
{
    $this->expectOutputString('bar');
    print 'baz';
}
```

Détermine la chaîne de caractère attendue à l'affichage

3

Fixtures



PRÉSENTATION

- L'un des travaux les plus pénibles dans les jeux de tests est de mettre en place l'environnement de données adéquat.
- Cet environnement doit être un état temporaire de l'application, qui doit donc retourner à l'état initial à l'issue de l'exécution des tests
- C'est ce que l'on appelle l'appareillage (*fixture*) des tests

SETUP ET TEARDOWN

- Une classe de tests peut spécifier deux méthodes :
  - `setUp()` permet d'initialiser un je de données pour tous les tests de la classe
  - `tearDown()` permet de détruire le jeu de données après les tests. En pratique, tearDown() n'est utile que si l'on crée des ressources persistantes (fichiers disque, etc.)

AUTRES MÉTHODES

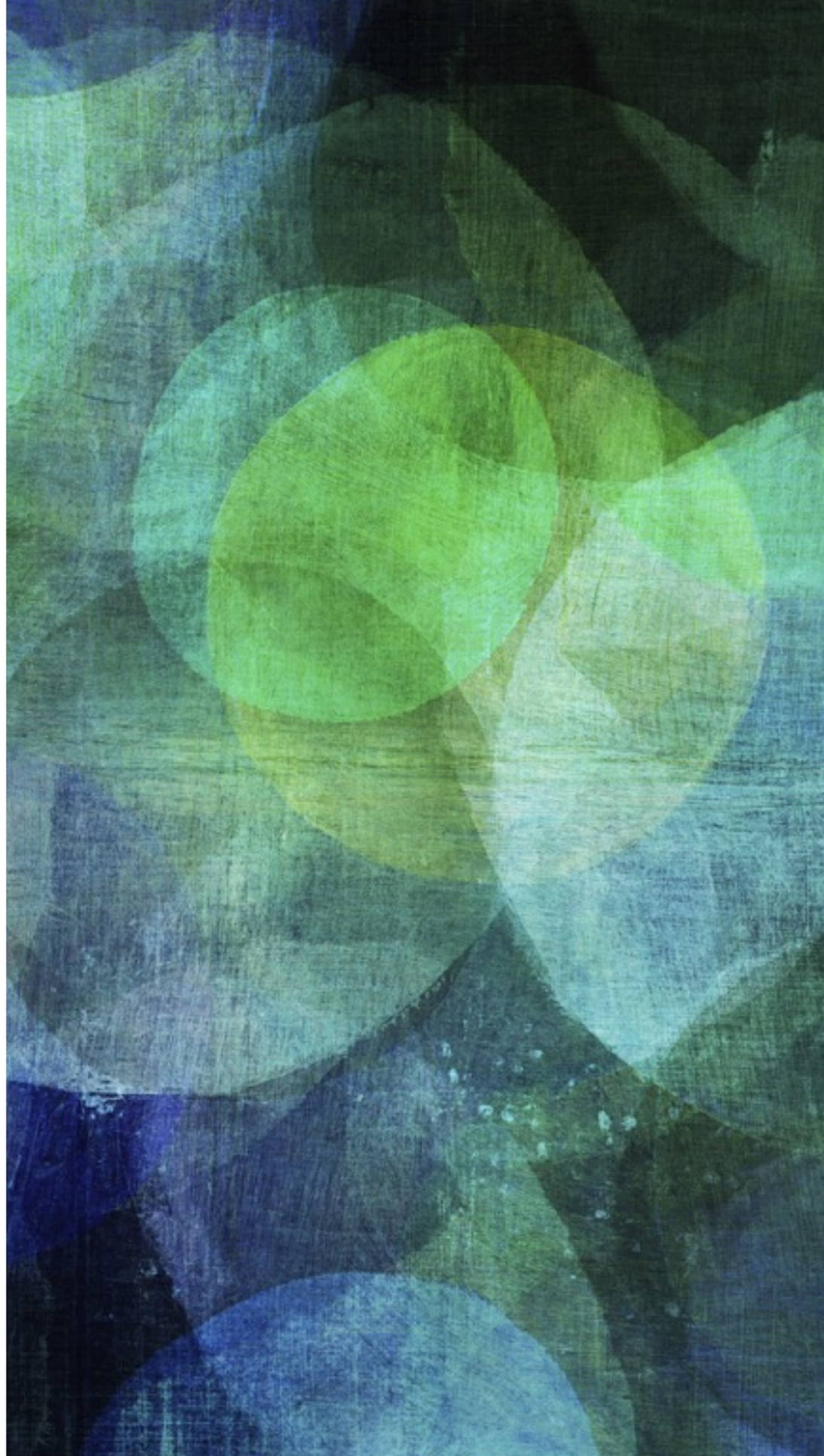
- Il est possible de spécifier des méthodes à exécuter au début et à la fin du jeu de tests sur la classe.
- il est également possible de vérifier l'état initial et état final des tests

```
public static function setUpBeforeClass() {}
protected function setUp() {}
protected function assertPreConditions() {}
protected function assertPostConditions() {}
protected function tearDown() {}
public static function tearDownAfterClass() {}
protected function onNotSuccessfulTest(Exception $e) {}
```

e.g. : ouverture d'une connexion à une base de données

VARIABLES GLOBALES

- Certaines données, comme celles contenues dans les variables super-globales peuvent être personnalisées par l'annotation @backupGlobals
- Cela permet de prendre en charge l'initialisait et la restauration de ces variables



3

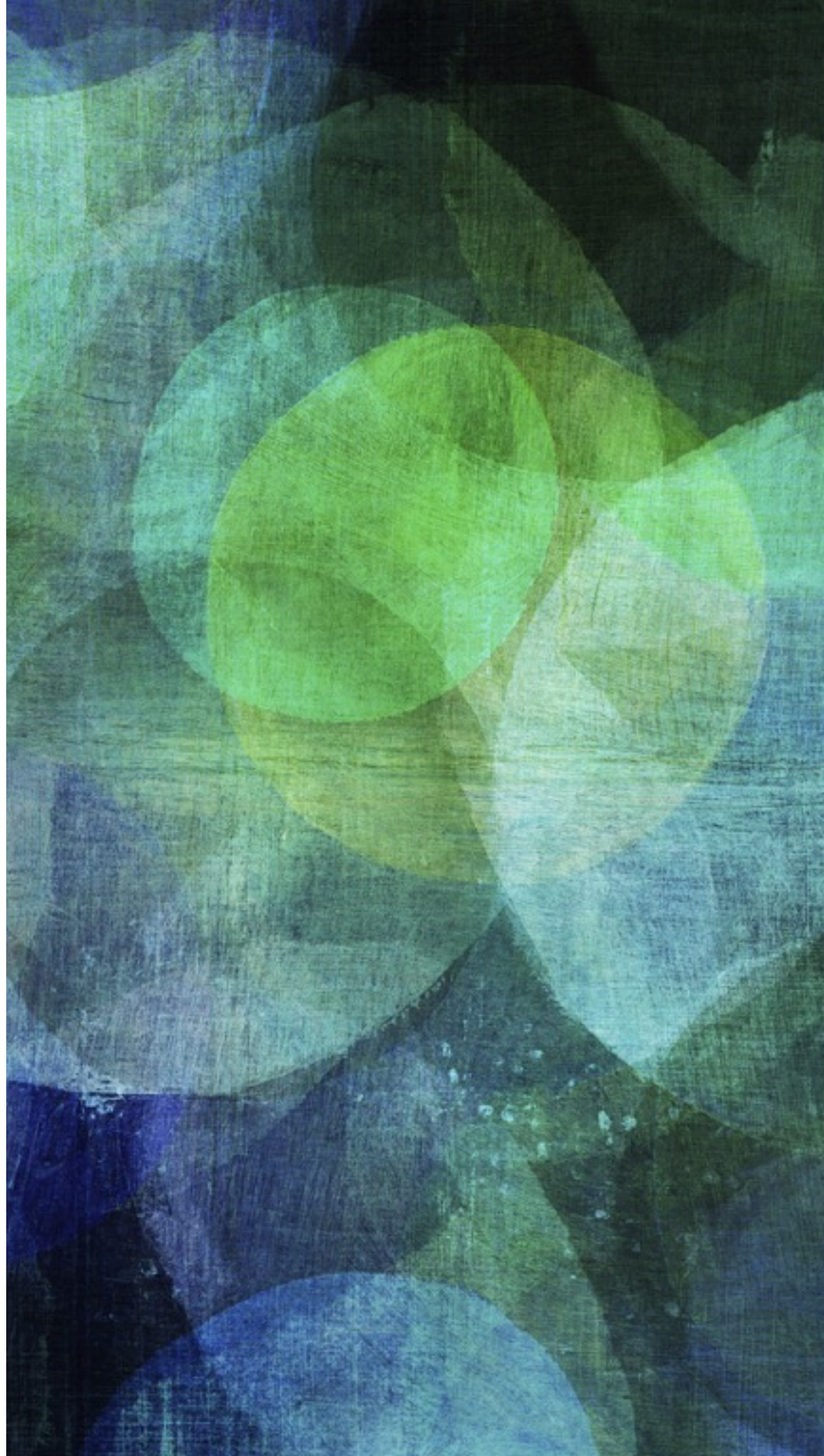
Tests fonctionnels

DÉFINITION

- Un test fonctionnel est destiné à vérifier un scénario d'utilisation de l'application
  - Création d'un utilisateur
  - Envoi d'une requête HTTP
  - Accès au modèle
  - Analyse de la réponse

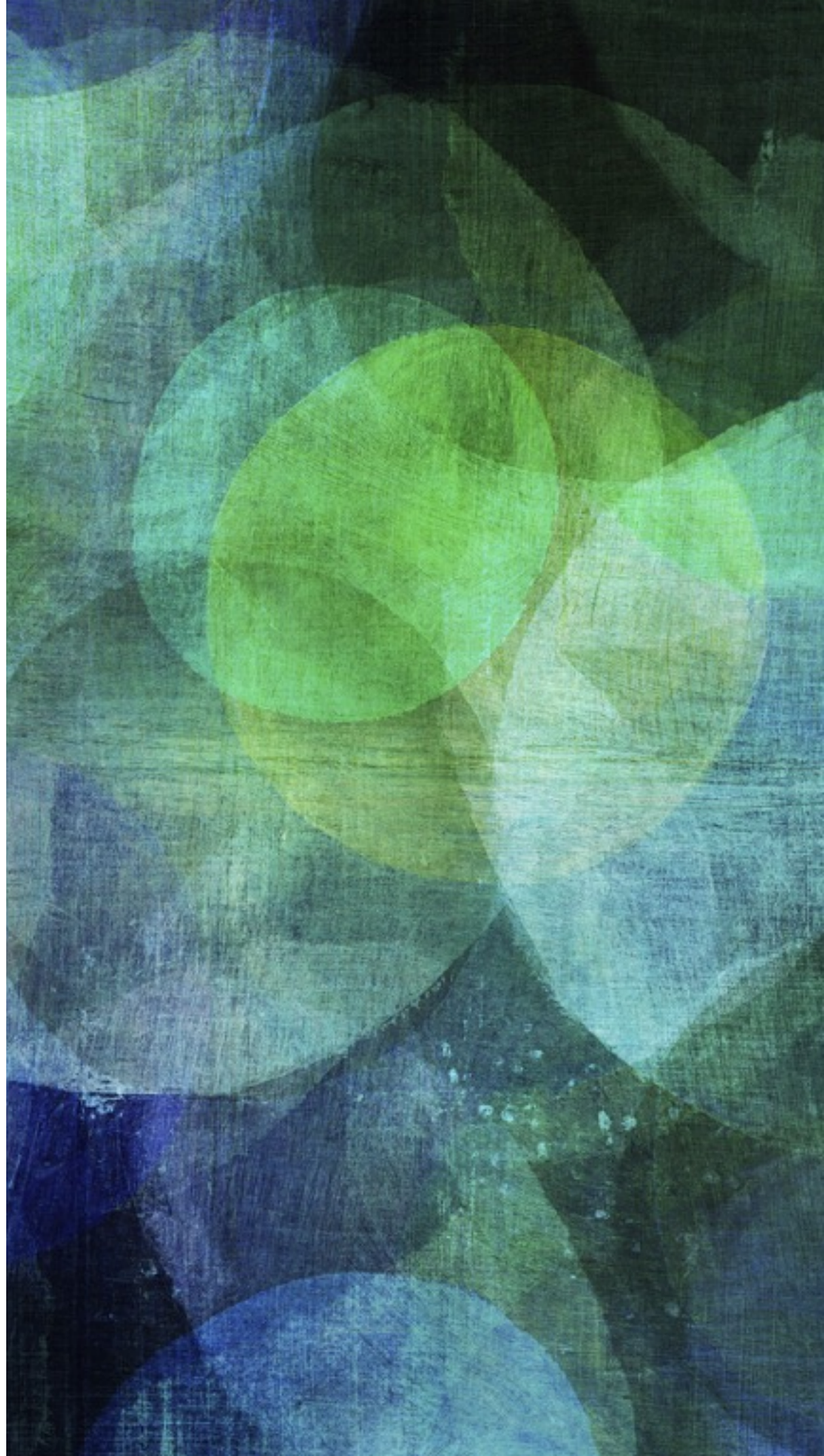
4

Le Crawler



5

Le modèle et Doctrine



METTRE EN ŒUVRE UNE BASE DE TESTS

- Pour éviter les conflits et les effets de bord, configurer l'accès à une base spécifique dans config_test.yml

```
# app/config/config_test.yml
doctrine:
    # ...
    dbal:
        host:    localhost
        dbname:  testdb
        user:    testdb
        password: testdb
```

SIMULER UN OBJET

➤ Pour créer un *doppelgänger* des entités :

```
use ExempleBundle\UneClasse;

class StubTest extends PHPUnit_Framework_TestCase
{
   public function testStub()
   {
      $stub = $this->getMockBuilder(‘UneClasse::class’)
                   ->getMock();
   }
}
```

SIMULER UNE MÉTHODE

- Pour créer un *doppelgänger* des entités :

```
use ExempleBundle\UneClasse;

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testStub()
    {
        $stub = $this->method(‘uneMethode’)
                     ->will($this->returnValue(0));
    }
}
```

- Si la classe originelle implémenter une méthode nommée method, il faut alors encapsuler l’appel avec **expects**.

```
$stub = $this->expects($this->once())—>method(‘uneMethode’)->will($this->returnValue(0));
```

SIMULER REPOSITORIES ET ENTITYMANAGER

- Pour tester le modèle, simuler l'entité ne suffit pas, il faut aussi simuler la classe de requêtes associée et l'Entity Manager lui-même.

```
$employeeRepository = $this
    ->getMockBuilder(EntityRepository::class)
    ->disableOriginalConstructor()
    ->getMock();
$employeeRepository->expects($this->once())
    ->method('find')
    ->will($this->returnValue($employee));
```

ENTITY MANAGER

```
$entityManager = $this
   ->getMockBuilder(ObjectManager::class)
   ->disableOriginalConstructor()
   ->getMock();
$entityManager->expects($this->once())
   ->method('getRepository')
   ->will($this->returnValue($employeeRepository));
```

UNE MÉTHODE PLUS RÉALISTE

- Etendre la classe KernelTestCase permet de faire des tests plus efficacement.

```
// tests/AppBundle/Entity/ProductRepositoryTest.php
namespace Tests\AppBundle\Entity;

use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class ProductRepositoryTest extends KernelTestCase
{

}
```

SETUP

```
protected function setUp()
{
    self::bootKernel();
    $this->em = static::$kernel->getContainer()
        ->get('doctrine')
        ->getManager();
}
```

1) Initialisation du noyau de Symfony

2) permet l'utilisation des services

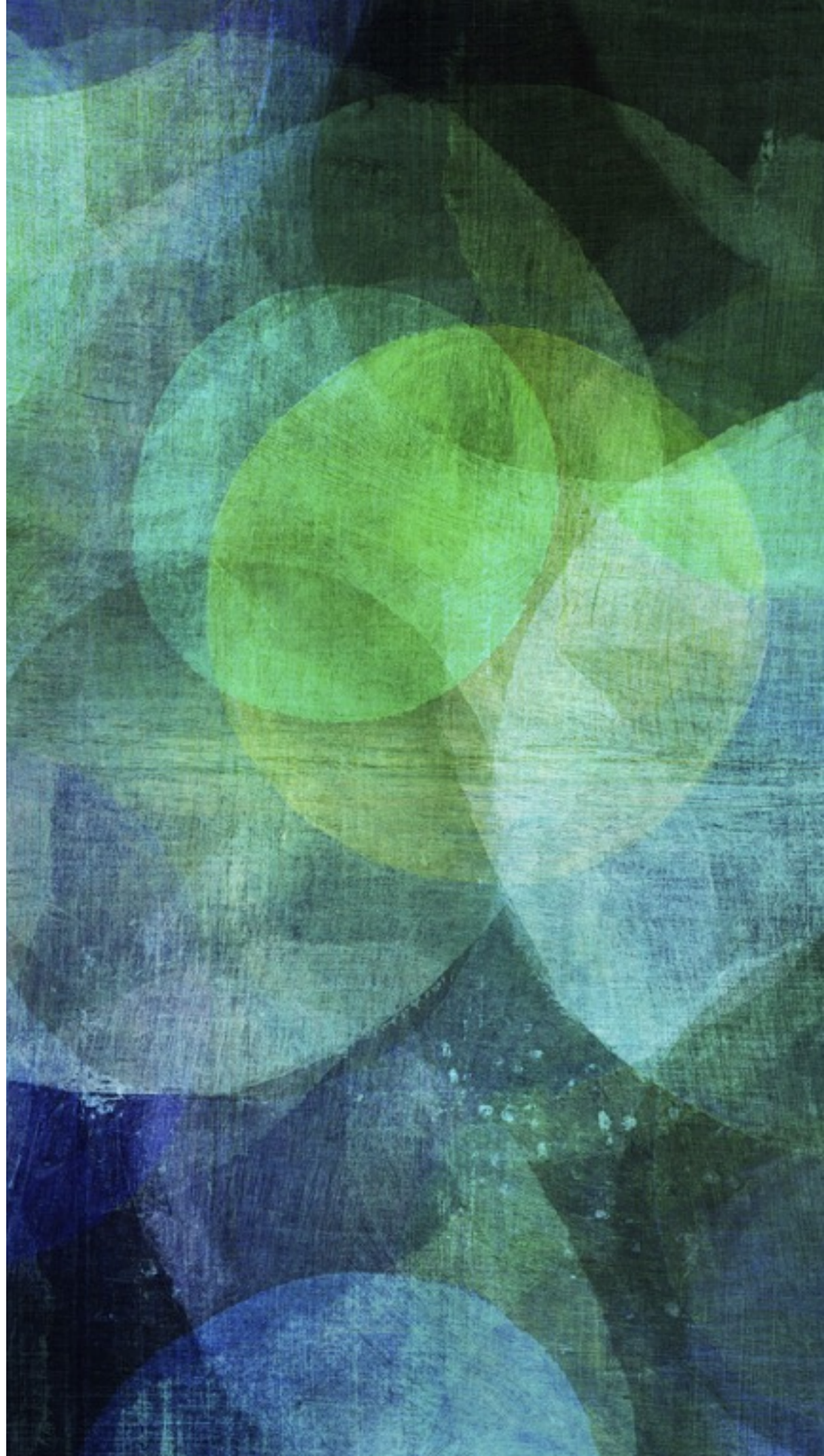
EXEMPLE

```
public function testSearchByCategoryName()
{
    $products = $this->em
        ->getRepository('AppBundle:Product')
        ->searchByCategoryName('foo')
    ;

    $this->assertCount(1, $products);
}
```

4

Les bases de données avec PHPUnit



INSTALLATION DE PHPUNIT/DBUNIT

- Pour pouvoir tester les modèles, il faut installer le package DbUnit

```
composer require 'phpunit/dbunit>=1.2.*'
```

CONFIGURATION DU TEST

```php
<?php
class MyGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
     * @return PHPUnit_Extensions_Database_DB_IDatabaseConnection
     */
    public function getConnection()
    {
        $pdo = new PDO('sqlite::memory:');
        return $this->createDefaultDBConnection($pdo, ':memory:');
    }

    /**
     * @return PHPUnit_Extensions_Database_DataSet_IDataSet
     */
    public function getDataSet()
    {
        return $this->createFlatXMLDataSet(dirname(__FILE__).'/_files/guestbook-seed.xml');
    }
}
?>
```

CONNEXION A UNE BASE EXISTANTE

```
  final public function getConnection()
  {
     if ($this->conn === null) {
        if (self::$pdo == null) {
           self::$pdo = new PDO( $GLOBALS['DB_DSN'], $GLOBALS['DB_USER'],
$GLOBALS['DB_PASSWD'] );
        }
        $this->conn = $this->createDefaultDBConnection(self::$pdo, $GLOBALS['DB_DBNAME']);
     }

  return $this->conn;
}
```

FICHIER DE CONFIGURATION

- Un fichier de configuration de l'accès à la base de données peut être créé dans le répertoire 'tests' : phpunit.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit>
  <php>
    <var name="DB_DSN" value="mysql:dbname=myguestbook;host=localhost" />
    <var name="DB_USER" value="user" />
    <var name="DB_PASSWD" value="passwd" />
    <var name="DB_DBNAME" value="myguestbook" />
  </php>
</phpunit>
```

5

Exécuter les tests