PHPUnit, fiabiliser vos développements avec les tests unitaires

Deuxième journée

Par Fabien Salles | Clever Age

Deuxième journée

- Comment lancer les tests et quand les écrires
- Allez plus loin avec PHPUnit
- D'autres outils pour tester vos applications
- Exercices
- Les bonnes pratiques dans l'écriture du code et des tests

Comment lancer nos tests ?

- Manuellement par la ligne de commande
- Serveur d'intégration continue
- Automatiquement suite à des modifications
- Avant de déployer son code

Serveur d'intégration continue

L'intégration continue est un concept qui consiste à intégrer continuellement et automatiquement tout développement sur un serveur d'intégration. Afin d'orchestrer cette automatisation, il existe des serveurs d'intégration continue permettant de configurer cette orchestration en fonction de vos besoin.

Les principaux serveur d'intégration continue sont :

- Jenkins,
- Travis CI,
- Banboo

Automatiser le lancement des tests en locale

Avec un task runner :

- Grunt
- Gulp

Avec watchr

Avec le File Watcher de PHPStorm

Forcer le lancement des tests en locale

Vous souhaitez obliger l'exécution des tests avant un commit ?

Utiliser PHP Git Hooks

Quand écrire les tests ?

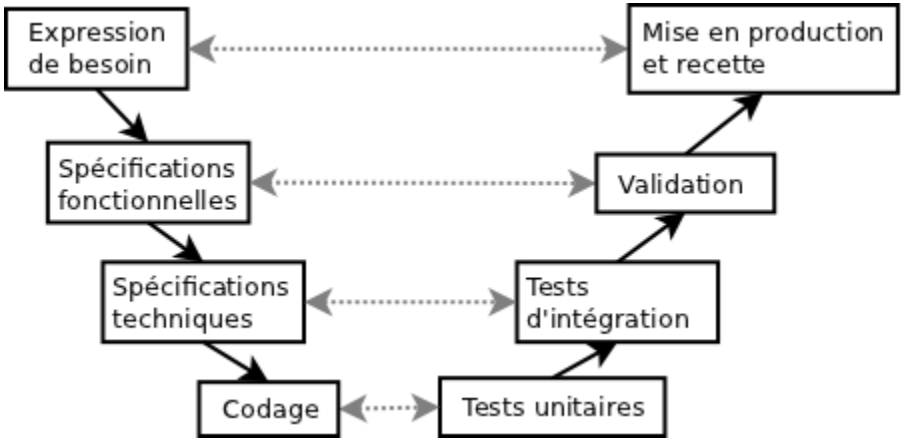
N'importe quand, il n'est jamais trop tard...

Lorsque l'on souhaite tester l'application

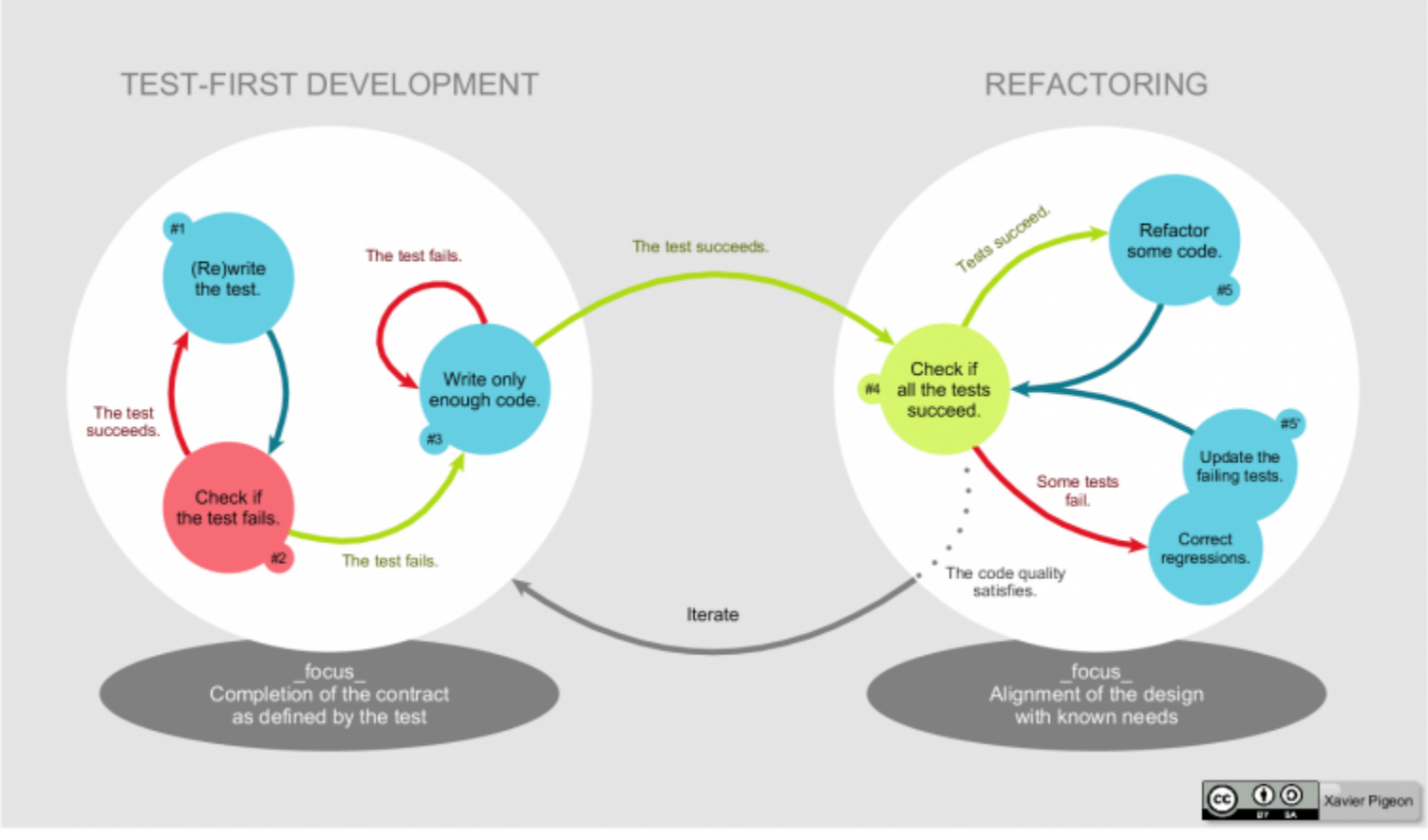
> *Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. -- Martin Fowler*

Chaque fois que vous êtes tentez de tester quelque chose manuellement, écrivez un test automatisé à la place

Dans la phase d'un projet



Avant le développement



Faut-il faire du TDD ?

- **Cela dépend de vous** : de vos connaissances dans l'écriture de tests unitaires, de votre façon de les écrires, de votre façon de penser
- **De votre projet** : votre équipe en est-elle capable ? votre projet permet-il de facilement coder des tests ?
- **De vos objectif** : visez-vous 100% de couverture de code ou recherchez-vous de la valeur ajoutée (business values) ?

Avant un refactoring

> *Changes in a system can be made in two pimary ways. I like to call them Edit and Pray and Cover and Modify. -- Michael Feathers, Working effectively with legacy code.*

Edit and Pray (ou Working with care) :

- Comprendre le code que nous nous apprêtons à modifier
- Tester l'application manuellement

Cover and Modify :

- Créer un jeu de test pour couvrir le maximum de cas de figures et effectuer les modifications par la suite.

Pourquoi refactorer ?

- Les besoins du client évoluent, le code doit évoluer avec
- Eviter la dette technique
- Eviter la duplication : DRY (Don’t Repeat Yourself)
- Eviter le code Spaghetti, Legacy

> *Leave the code cleaner than we found it. (Boy Scout Rule)* -- *Robert C. Martin, Clean Code.*

Quels tests devons nous faire avant un refactoring ?

Cela dépend du refactoring, de l’application, de la couverture de code et des tests existants.

Notre refactoring est-il important ?

- Qu'est-ce que je souhaite changer ?
- Quels sont les modules/classes que ma modification impacte ?
- Quels sont mes options de modifications ?

Prenons le pire scénario

Nous nous rendons compte que le refactoring est important et nous n'avons aucuns tests.

Solution :

Golden Master Testing : méthode de test en boite noir permettant de vérifier que pour un ensemble de paramètres données en entrée, les données en sortie ne varie pas.

Les étapes du Golden Master Testing

1. Comprendre comment le système retourne les données
2. Trouver une façon de capturer ces données
3. Trouver un format de sortie adapté
4. Générer des données aléatoires et sauvegarder les paramatères en entrées et les résultats en sorties
5. Ecrire des tests en utilisant ces données
6. Commiter les tests
7. Vérifier que les tests sont suffisants
8. S'il le sont, attaquer le refactoring. S'il le sont pas, revenir au point 3.

De manière générale, quels tests doit-on écrire ?

Pensez au **ROI** !

Le retour sur investissement (RSI), aussi appelé aussi ROI (Return on Investment), est un indicateur financier, il permet de mesurer et de comparer le rendement d'un investissement.

Par exemple, il est prouvé que passer 1 heure en TDD permet par la suite de diminuer de 2h30 le temps de correction de bugs et rework.

Tester avant tout les fonctionnalités importante !

Behaviour-driven development (BDD)

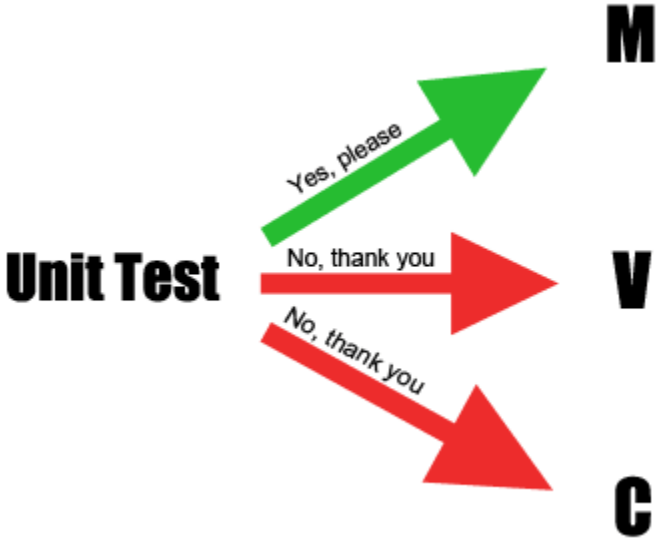
C'est une méthode agile qui a vu le jour par le constat que le TDD amenait des confusions et des malentendus chez beaucoup de personnes :

- Par où commencer ?
- Que devons nous tester ?
- Que ne devons nous pas tester ?
- Comment appeler nos tests ?
- Quels cas de figures devons nous prendre en compte ?
- Comment interpréter les erreurs ?

Le BDD permet de penser aux comportements à tester plutôt qu'aux détails techniques

- Le nom de la méthode doit être une phrase
- Le mot **test** disparaît au profit de **should**
- Le test représente un comportement attendu permettant de comprendre l'intérêt du code
- Cela aide aussi a écrire les tests sur le(s) comportement(s) que le code ne gère pas

Que peux-on tester unitairement ?



Aller plus loin avec PHPUnit

- Organiser vos tests
- La couverture de code
- Documenter grâce à vos tests
- Tester une base de données
- Utiliser Prophecy
- Etendre PHPUnit

Organiser vos tests

- Par les groupes via les annotations
- Par les suites de tests via :
  - le système de fichier
  - la configuration XML

Exemple :

```
<phpunit bootstrap="src/autoload.php">
  <testsuites>
    <testsuite name="money">
      <file>tests/IntlFormatterTest.php</file>
      <file>tests/MoneyTest.php</file>
      <file>tests/CurrencyTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

La couverture de code (1/3)

PHPUnit utilise le composant PHP_CodeCoverage se basant sur les fonctionnalités de converture de code de xdebug

Les métriques utilisés sont :

- **Line coverage** : vérifie qu'une ligne est couverte
- **Function and Method coverage** : vérifie que toutes les lignes d'une fonction ou méthode ont bien été exécutés
- **Class and Traits coverage** : vérifie que toutes les méthodes d'une classe ou d'un trait ont été exécutés
- **Change Risk Anti-Patterns (CRAP) Index** : prédit l'effort pour maintenir le code

La génération du rapport (2/3)

On peut générer le raport de la couverture de code dans différents formats via l'option **--coverage-*** ou grâce au fichier de configuration :

- **--coverage-html** : format le plus utilisé
- **--coverage-text** : très peu utilisé mais utile dans certains cas de figure
- **--coverage-clover** : format XML interprétable par des outils d'intégration continue
- **--coverage-php** : génération plus rapide et utile lorsque l'on sépare l'éxecution des tests et que l'on souhaite rassembler la couverture de code avec PHPCOV

Configurer les fichiers à utiliser (3/3)

- Par défaut, PHPUnit permet de gérer une **blacklist**.
- Une meilleur pratique est de configurer une **whitelist**

Documenter grâce aux tests

Si vous êtes assez rigoureux sur le nommage de vos tests, vous pouvez utiliser l'option **testbox** pour générer une documentation :

```
phpunit --testdox BankAccountTest
PHPUnit 6.1.0 by Sebastian Bergmann and contributors.

BankAccount
 [x] Balance is initially zero
 [x] Balance cannot become negative
```

On peut également utiliser le format text, html, et xml (--testdox-text, --testdox-html et --testdox-xml)

Tester une base de données

> *Database testing*

Prophecy

C'est un framework de mock issue d'un autre framework de test unitaire (PHPSpec)

Il est flexible, puissant, s'appuie sur la philosophie BDD et s'intègre parfaitement à PHPUnit au point qu'il a même une rubrique accordée au sein de sa documentation.

> *Documentation du Framework*

Dummy avec Prophecy

Ici on ne passe pas par une méthode générique faisant référence à un type de doublure de test et amenant de l'ambiguité :

```
// initalisation du prophet
$prophet = new Prophecy\Prophet();
// initialisation du dummy object
$prophecy = $prophet->prophesize(Logger::class);
// On révèle le dummy pour pouvoir l'utiliser
$dummy = $prophecy->reveal();
```

Stub avec Prophecy

```php
// initalisation du prophet
$prophet = new Prophecy\Prophet();
// initialisation du stub object
$prophecy = $prophet->prophesize(Logger::class);
// pré-configuration du retour de la fonction
$prophecy->getLevel()->willReturn(Logger::INFO)
// On révèle le stub pour pouvoir l'utiliser
$stub = $prophecy->reveal();
```

Fake avec Prophecy

```
$prophet = new Prophecy\Prophet();
$urlsByRoute = array(
    'index' => '/',
    'about_me' => '/about-me'
);
$urlGenerator = $prophet->prophesize(UrlGenerator::class);
$urlGenerator
    ->generate(\Prophecy\Argument::any())
    ->will(
        function ($routeName) use ($urlsByRoute) {
            if (isset($urlsByRoute[$routeName])) {
                return $urlsByRoute[$routeName];
            }

            throw new UnknownRouteException();
        }
    ));

// On révèle le fake pour pouvoir l'utiliser
$fake = $prophecy->reveal();
```

L'object Argument

Avec Prophecy on doit fournir un paramètre lorsque la signature de la méthode en demande un.

On n'est cependant pas obliger de donner la vrai valeur attendu, on peut utiliser l'objet Argument :

- Argument::any() : utilise n'importe quel argument
- Argument::cetera() : utilise n'import quel argument pour le reste de la signature
- Argument::type($typeOrClass) : vérifie que l'argument match un type particulier

Mock avec Prophecy

```
// initalisation du prophet
$prophet = new Prophecy\Prophet();
// initialisation du mock object
$prophecy = $prophet->prophesize(Logger::class);
// predilection
$prophecy->getLevel()->shouldBeCalled()
```

Spy avec Prophecy

```
// initalisation du prophet
$prophet = new Prophecy\Prophet();
// initialisation du spy
$prophecy = $prophet->prophesize(Logger::class);
// on révèle le spy
$spy = $prophecy->reveal();
// vérification./df
$prophecy->getLevel()->shouldHaveBeenCalled()
```

Etendre PHPUnit

PHPUnit propose la possibilité d'étendre ses fonctionnalités grace à l'héritage

On peut ainsi :

- Créer des assertions custom
- Modifier les phrases que retourne PHUnit lors de l'éxécution des tests
- Faire des actions avant et après l'exécution des tests avec les Test Decorator
- Ecrire d'autres types de tests : Data-driven testing

Autres Framework de tests unitaires

- Atoum : Chaque test est exécuté dans un processus distinct
- PHPSpec : Met en avant les mocks et utilise une syntaxe BDD
- Kahlan : Framework BDD utilisant une syntaxe RSpec/JSpec

Framework de test fonctionnels

- Selenium
- Behat
- Codeception
- Et bien d'autres !

Avez-vous des questions ?

Exercices

- Golden Master Testing
- Client API :
  - Starting a New PHP Package The Right Way
  - Basic TDD in Your New PHP Package
  - API Client TDD with Mocked Responses
- PHPSpec
- Behat
- Atoum

Les bonnes pratiques au niveau du code

SOLID

- Single Responsability Principe
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsability Principe

> *A class should have one reason to change*

Si une classe a plusieurs responsabilités, ces dernières se retrouveront liées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la rigidité et la fragilité du code.

Open Closed Principe

> *Classes, methods should be open for extension, but closed for modifications.*

Ce principe découle du premier. Il vaut mieux utiliser la classe ou la méthode pour l'étendre, l'encapsuler plutôt que de la modifier pour répondre à de nouveaux besoins.

Liskov Substitution Principle

> *Subtypes must be substituable for their base types.*

1. Lorsqu'on utilise une classe parente, son remplacement par une classe enfant ne doit pas altérer le comportement
2. Lorsqu'on utilise une interface, son remplacement par une classe qui l'implémente ne doit pas altérer le comportement

Interface Segregation Principle

> *Clients should not be forced to depend on methods that they do not use.*

Le but de ce principe est d’utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel. L'utilisation de ces interfaces réduit le couplage entre les classes

Dependency Inversion Principle (1/3)

Ce principe est dans la continuité du précédent.

> *High level modules should not depend on low level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

DIP : Autrement dit (2/3)

> *Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.*

Ici "High Level modules" représentent les classes possédant des dépendances externes. "Low level modules" sont des classes qui implémentent des opérations bas niveaux (ex : écriture dans une base MySQL)

DIP : Application (3/3)

Le principe est de passer par des interfaces faisant office de couche d'abstraction afin que les modules de haut niveau ne soit pas directement couplés au modules de bas niveau. Il n'y a donc pas de dépendances entre les modules de haut niveau et bas niveau mais tous les 2 dépendent de la couche d'abstraction.

High Level Classes --> Abstraction Layer --> Low Level Classes

DDD : Domain-Driven Design

- On se focalise sur le métier
- Le code doit mettre en avant les contraintes métiers
- On facilite la compréhension et la communication entre le métier et les équipes techniques
- On s'abstrait des contraintes techniques et du framework

> *DDD vite fait*

CQRS : Command Query Responsibility Segregation

On sépare l'écriture et la lecture avec d'un côté des **Commandes** et de l'autres des **Queries**

Si cela vous intèresse :

- livre CQRS Journey
- librairie de PHPLeague Tactician

KISS : Keep It simple, stupid

Lorsque l'on essaye de garder un code le plus simple possible :

- les solutions apparaissent plus clairement
- nous sommes capable de nous concentrer sur les choses les plus importantes
- les problèmes peuvent être résolus rapidement

> *Simplicity is the ultimate sophistication. — Leonardo da Vinci*

Prenons l'exemple d'un test

```
# ...quelques lignes configurant des doublures de tests..
$foo = $this->createMock("a most excellent Foo")
$foo->method('bar')->willReturn("hello!")
# ...quelques lignes supplémentaires configurant d'autres doublures de tests...
# ...quelques lignes supplémentaires utilisant des méthodes afin de créer la logi
# ...trop d'assertions parce qu'on ne sait pas quoi tester
$this->assertEquals("hello!", $foo->bar())
# ...quelques autres assertions... L'écriture de test est difficile !!!
```

Si vous sortez de votre zone de confort et que vous appliquez les principes que l'on a évoqués vous devriez facilement trouver ce cas de figure un jour !

En cas d'erreur, qu'est ce qui est incorrect ? La valeur attendue ou la valeur actuelle ?

> *When we write tests this way, most of the time if the implementation changes, we end up changing the expectations of the test as well and yeah, the tests pass automatically. But without knowing much about its behaviour. These tests are a mirror of the implementation, therefore tautological.—Fabio Pereira*

Pourquoi notre test ne respecte pas le principe KISS ?

- Notre code n'est pas bien conçu et il y a de forte chance qu'il ne respecte pas les principes SOLID
- Notre test n'est pas focalisé sur la chose la plus importante
- Notre test manque peut être d'utilité

Si on ne trouve aucunes améliorations à apporter à notre modélisation pour simplifier notre test, c'est peut être parce qu'on se trouve sur un **test d'intégration** et qu'on ne devrait pas le tester unitairement !

Comment améliorer nos tests ?

Un grand nombre de sujets abordés ici sont controversés.

Les avis peuvent diverger et les techniques mentionner peuvent correspondre à certaines personnes mais pas à d'autres.

Encore une fois, il faut que vous soyez en accord avec la façon dont vous écrivez vos tests !

DAMP : Descriptive And Meaningful Phrases

Pour maintenir le code il faut le comprendre et pour le comprendre il faut le lire.

DAMP accroit la maintenance en réduisant le temps nécessaire pour le lire et le comprendre

DAMP not DRY (Don't Repeat Yourself)

- L'écriture de code pour les tests est différente de l'écriture de code applicatif.
- Lorsque l'on enlève de la duplication dans nos tests, on a tendance à complexifier la lecture
- L'inter-dépendance entre les tests et vu comme un anti-pattern

> *Tests are procedural by nature. Allowing developers to read tests in a procedural manner would be the natural choice. - Jay Fields, Working effectively with unit tests.*

Comment respecter le principe DAMP

- Ne pas tester différentes choses en même temps
- Chaque méthode doit encapsuler entièrement le cycle de vie et la vérification d'un jeu de test
- Un test ne doit pas dépendre d'un autre
- Une méthode de test doit remplir un but facilement identifiable

Le nom d'un test

- La première motivation pour écrire le nom d'une méthode doit être la documentation
- Le nom de la méthode doit décrire ce que vous tester

Eviter les commentaires

> *It's surprising how often you look at thickly commented code and notice that the comments are there because the code is bad. - Fowler & Beck, Refactoring.*

- Si vous avez besoin d'un commentaire pour expliquer ce qu'un bloque de code fait, utiliser le principe **Extract Method**
- Si la méthode est déjà extraite et que vous devez tout de même commenter pour expliquer ce qu'elle fait, utiliser le principe **Rename Method**

> *Code never lies, comments sometimes do. - Ron Jeffries.*

Four-Phase Test

Les Frameworks xUnit implémentent le pattern **Four-Phase** représenté par les étapes **SetUp**, **Exercice**, **Verify** et **TearDown**.

On a vu que les étapes **Exercice** et **Verify** se situaient dans la méthode d'un test et que les parties **Setup** et **TearDown** étaient souvent détachées.

Une bonne pratique et d'utiliser les méthodes **setUp** et **tearDown** uniquement pour initialiser l'environnement mais pas les fixtures.

Pourquoi ne pas utiliser setUp pour les fixtures ?

- **setUp** évite la duplication mais complique la lecture du code parce qu'elle est souvent utilisé comme une méthode **"fourre-tout"**.
- Elle oblige à comprendre tout ce qu'elle fait et devient contre-productif lorsque les fixtures qu'elle initialise ne sont pas utilisées par tous les tests
- Si vous utilisez les principes décris ici, vous ne devriez pas avoir besoin de l'utiliser

AAA Principles (1/2)

1. Arrange all necessary preconditions and inputs.
2. Act on the object or method under test.
3. Assert that the expected results have occurred.

En BDD équivaux à GivenWhenThen

AAA Principles (2/2)

```
public function testSizeOfTheCollectionReflectsNewElements()
{
    // Arrange
    $collection = new Collection();

    // Act
    $collection->add('element');

    // Assert
    $this->assertSame(1, count($collection);
}
```

Utiliser des méthodes factory

Lorsque vous devez initialisez plusieurs objets dans vos tests vous pouvez déporter la création dans une autre méthode.

Cela :

- améliorera la lisibilité du test,
- réduira l'importance de ces initialisations pour augmenter celle des assertions,
- pourra potentiellement permettre de réutiliser la/les méthode(s) dans d'autres tests.

Assertion Last Principle

L'assertion devrait être la dernière partie du code dans un test.

Ne pas tester de méthode privé

Une méthode privé contient des détails d'implémentation cachés pour l'utilisateur. Si les méthodes publiques qui l'appel sont suffisament testées il n'y a aucune raison de casser l'encapsulation pour tester une méthode privé.

Si celle-ci contient trop de logique qui se retrouve être importante et complexe, le mieux et de la transformer en classe publique et de la tester séparemment.

Exécuter vos tests unitaires avant vos tests d'intégration

Une annomalie est plus facilement compréhensible sur un test unitaire totalement isolé de ses dépendances.

L'erreur sera plus explicite que lorsque les tests d'intégration tomberont en échec

Eviter les Fake et les Spy

- https://stackoverflow.com/questions/12827580/mocking-vs-spying-in-mocking-frameworks
- https://php-and-symfony.matthiasnoback.nl/2014/07/test-doubles/
- https://8thlight.com/blog/uncle-bob/2014/05/14/TheLittleMocker.html

Une assertion par test

Une assertion par test permet de se focaliser sur un seul comportement. C'est plus simple à lire et maintenir.

De plus lorsqu'il y a un échec, le message d'erreur est plus facilement compréhensible et l'assertion n'en bloque pas d'autres !

Un Mock par test

Dans la même logique on peut essayer de limiter au maximum le nombre de mock que l'on crée.

Et on peut même aller encore plus loin :

- Si votre test a une assertion, ne pas ajouter de mock
- Si votre test a un mock, ne pas ajouter d'assertion

Implementation Over Specification

Lorsque l'on a un test avec beaucoup de Mock et que l'on vérifie pleins de choses (le retour, les arguments mis en paramètres de la méthode mocké...) on se retrouve avec un couplage fort entre notre test et le code que l'on test (et mock). Plus on a de spécification, plus nos tests seront fragiles et pourront tomber en erreur.

Do not mock any class you do not own

- https://www.reddit.com/r/PHP/comments/3iqjp6/do_not_mock_any_class_
- http://spring.io/blog/2007/01/15/unit-testing-with-stubs-and-mocks/
- https://inviqa.com/blog/php-test-doubles-patterns-prophecy

Mocks for Commands, Stubs for Queries

http://blog.ploeh.dk/2013/10/23/mocks-for-commands-stubs-for-queries/

Ne tester qu'une fois les arguments de la fonction qu'on mock

Si vous ressentez le besoin de tester des arguments, il est inutile de le faire plusieurs fois si la fonction est appelée plusieurs fois. Il vaut mieux se concentrer sur les assertions et limiter les cas d'échecs lorsque la manière dont le mock retourne le résultat change.

Si vous êtes trop rigide sur le nombre de fois que la méthode a été appelé, l'ordre de ses appels, les arguments utilisés, votre test sera en erreur au moindre changement structurel du SUT.

Negative testing

En générale on test le fait que l'application fonctionne correctement. Si une erreur est trouvée, le test est en échec. On appel cela le **Positive Testing**.

Le **Negative testing** est le fait de tester des comportements invalides.

Classical Testing

> *Préférer faire des tests d'intégration plutôt que de remplacer toutes les dépendances*

Mockist Testing

> *Préferer remplacer toutes les dépendances et faire de vrais tests unitaires*

Devrais-je être un classicist ou un mockist ?

https://www.martinfowler.com/articles/mocksArentStubs.html

Conclusion

- Il est simple de coder mais difficile de bien coder
- Il est simple d'écrire des tests mais diffcile d'en écrire des biens

> *The beauty of testing is found not in the effort but in the efficiency. Knowing what should be tested is beautiful, and knowing what is being tested is beautiful. -- Murali Nandigama.*