

Tests et Test Unitaire

Introduction

Liens avec un référentiel métier

- Tests d'intégration et d'acceptation d'un service
- Gestion d'environnements de développement et de test
- Réalisation des tests nécessaires à la validation d'éléments adaptés ou développés
- Réalisation des tests nécessaires à la mise en production d'éléments mis à jour

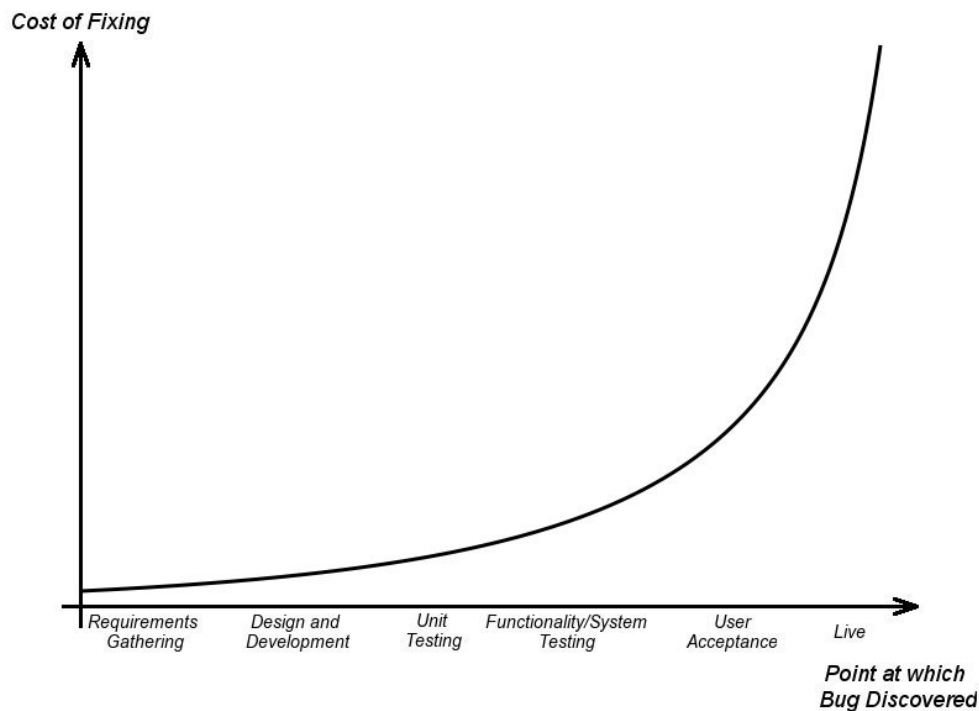
Savoirs associés:

- Test de performance, test de charge, test d'intrusion...

Le concept de tests unitaires n'est pas une nouveauté. Depuis le début de l'informatique, tester fait partie de l'activité quotidienne d'un développeur. Ce qui est nouveau aujourd'hui, c'est que l'on place cette activité, en particulier les tests unitaires, au coeur du processus de conception (ref. Méthodes Agiles, DevOps). Plusieurs raisons à cela :

- Concevoir le test unitaire d'un service avant même d'avoir codé ce dernier, **favorise la modularité** (petites unités à tester) et la **concision** (le développeur n'implémente que l'essentiel).
- Plus un bogue est détecté tôt plus facile sera sa correction, **moins il coûtera**.
- Disponibilité d'**outils (xUnit)** d'aide à la conception et exécution de tests unitaires (open source).
- Bonne **intégration** de ces outils dans les ateliers de génie logiciel.
- Augmente la **qualité** générale du code produit. Facile à maintenir (code modulaire) et à tester.
- Permet de **rejouer les tests** à volonté, afin de concourir aux tests de non-regression.

Mots clés : **test unitaire, xUnit, industrialisation du logiciel, intégration continue**



Un définition (simplifiée d'après wikipedia).

Un test est un ensemble de cas à tester, éventuellement accompagné d'une procédure d'exécution (séquence d'actions à exécuter). Il est lié à un objectif.

Un *cas à tester* ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de **trois éléments clés** :

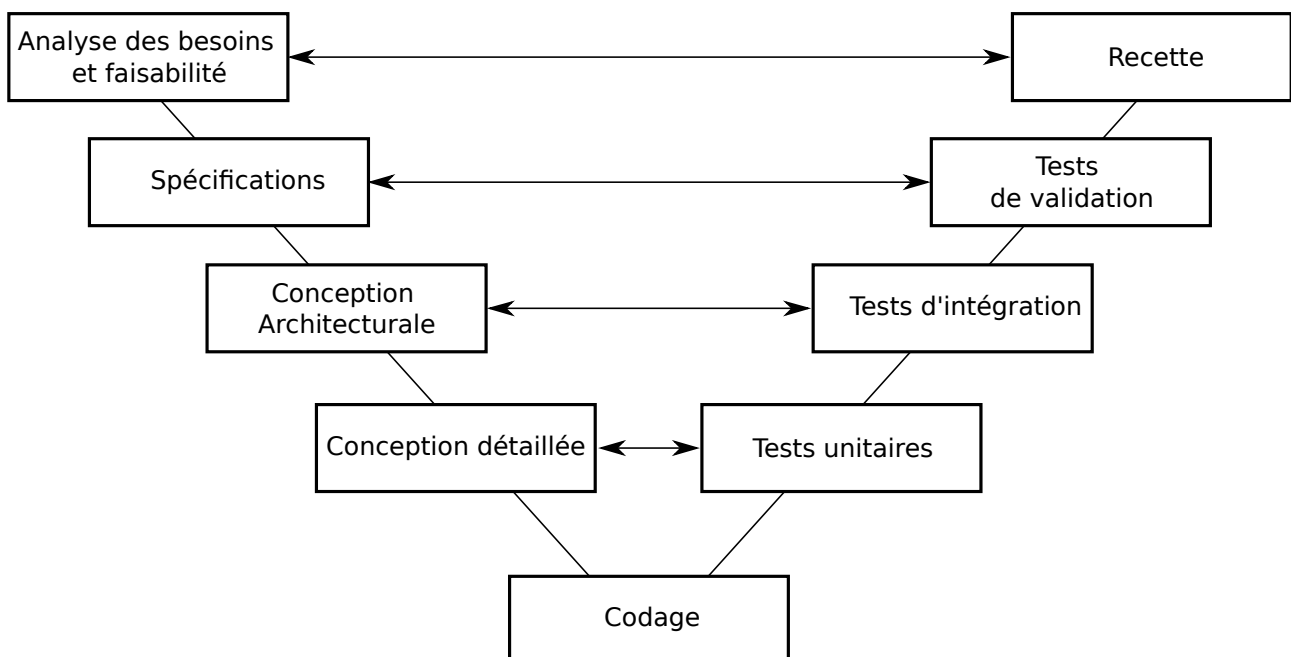
1. les données en entrée,
2. l'objet à tester
3. les observations attendues.

Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions. Un bon test respecte également l'**exigence de répétabilité**.

L'*activité de test* d'un logiciel est un des **processus du développement de logiciels** . Elle utilise différents types et techniques de tests pour vérifier que le logiciel est conforme à son cahier des charges (vérification du produit) et aux attentes du client (validation du produit).

Les tests de vérification ou de validation visent à s'assurer que ce système réagit de la façon prévue par ses concepteurs (spécifications) ou est conforme aux attentes du client l'ayant commandé (besoins), respectivement.

Structure historique du cycle de vie en V, aujourd'hui intégré dans un cycle itératif et incrémental, voire repensé par une approche *test first*.



https://fr.wikipedia.org/wiki/Cycle_en_V

Plus le nombre d'erreurs trouvées est important, plus il y a de chances qu'il y ait davantage d'erreurs dans le composant logiciel visé. Inversement, *l'absence d'erreurs ne signifie pas que le composant en est exempt*.

Un objet ne peut être testé que si on peut déterminer *précisément* le comportement attendu en fonction des conditions auxquelles il est soumis. Si la spécification ne permet pas cette détermination, la propriété du logiciel qu'elle définit ne peut être testée.

Un test visé à mettre en évidence des défauts de l'objet testé. Cependant il n'a pas pour objectif :

- de diagnostiquer la cause des erreurs,
- de les corriger,
- de prouver la correction de l'objet testé.

Une classification par niveau
couramment acceptée :

Niveau 4	Test d'acceptance (la recette)
Niveau 3	Test Système (validation)
Niveau 2	Test Intégration
Niveau 1	Test Unitaire

Quelques courtes définitions

Test de recette : L'application doit fonctionner dans son environnement de production (très souvent chargé d'autres services). On s'assure que l'application répond acceptablement à la demande initiale.

Test de validation : Les scénarios de cas d'utilisation des services sont vérifiés, à base de spécifications fonctionnelles (métier).

Test d'intégration : L'application est déployée dans un environnement le plus proche possible de l'environnement de production. On s'assure que le système n'en n'est pas perturbé et que les tests unitaires fonctionnent correctement dans cet environnement (test de non-regression).

Test unitaire : Chaque service (fonction, méthodes) est testé le plus indépendant des autres, au regard de ses spécifications techniques.

Test unitaire

Un **test unitaire** est un programme qui vérifie le bon fonctionnement d'un module (unité fonctionnelle) au travers de **situations déduites des spécifications** du module testé ; à partir de données d'entrée prédéterminées (l'état du système en entrée est connu), le test sollicite le module et confronte les données réellement obtenues avec celles théoriquement attendues, puis en déduit un état de succès ou d'échec. Mise à part des cas exceptionnels d'extrême simplicité, LE bon fonctionnement d'un module ne peut jamais être vérifié car le nombre de cas à traiter croît exponentiellement avec le nombre de situations différentes que le module sera susceptible de rencontrer. La couverture des tests n'est donc jamais atteinte.

Réserve d'usage

Attention ! **La réussite des tests ne permet pas de conclure au bon fonctionnement du logiciel.**

*On essaye cependant, heuristiquement, de faire en sorte que si un bug est présent, le test le mette en évidence, notamment en exigeant une bonne **couverture des tests** :*

- *couverture en points de programme : chaque point de programme doit avoir été testé au moins une fois*
- *couverture en chemins de programme : chaque séquence de points de programme possible dans une exécution doit avoir été testée au moins une fois (impossible en général).*
- *couverture fonctionnelle : chaque fonctionnalité métier de l'application doit être vérifiée par au moins un cas de test (tests IHM par exemple)*
- *tests de bout en bout (End-to-End) : test le logiciel dans une situation très proche de la production (pré-production) : différents scénarios utilisateur, différentes bases de données.*

Initiation par la pratique une méthode de conception par les tests unitaires

Objectif : Présenter à l'étudiant une logique de raisonnement propre à la programmation de tests unitaires.

L'approche proposée est guidée par les différentes catégories de cas (**cas généraux** et **aux limites**) : "vide", "presque vide", général, aléatoire, "presque plein", "plein". Prenons un exemple.

SPECS : On souhaite disposer d'une méthode utilitaire qui prend en paramètre un tableau d'objets, et qui retourne une représentation textuelle du tableau en HTML. Si le tableau est vide, la méthode retourne la chaîne « (vide) », sinon, si le nombre d'éléments dans le tableau le permet, il y a autant de <tr> que d'éléments dans le tableau à concurrence de maxRows lignes. S'il y a plus d'éléments dans le tableau que de lignes à afficher, les <td> de la dernière ligne seront défini par « ... ».

Présentation de la classe à tester

```
package utils
class UtilHtml
    public String toHtmlTab(Object[] tab, int maxRows)
        param tab : le tableau d'objet à représenter en HTML
        param maxRows : le nombre max de lignes à construire
        returns : représentation HTML du tableau,
                à concurrence de maxRows lignes, ou "(vide)",
                avec gestion du surplus de lignes ("..." comme valeurs de la
                dernière ligne)
```

Étude de différents cas de test

Cas général : n éléments dans le tableau et demande une version HTML à n éléments

```
UtilHtml uh = new UtilHtml();
String arg[] = {"A", "B", "C"};
String expectedHtmlTab =
    "<table><tr><td>A</td></tr><tr><td>B</td></tr><tr><td>C</td></tr></table>";
assertEquals(expectedHtmlTab, uh.toHtmlTab(arg, arg.length));
```

Quelques cas « limites »

- Tableau vide

```
UtilHtml uh = new UtilHtml();
String arg[] = {};
String expectedHtmlTab = "(vide)";
assertEquals(expectedHtmlTab, uh.toHtmlTab(arg, 10));
```

- Tableau avec 1 élément

```
UtilHtml uh = new UtilHtml();
String arg[] = {"A"};
String expectedHtmlTab = "<table><tr><td>A</td></tr></table>";
assertEquals(expectedHtmlTab, uh.toHtmlTab(arg, arg.length));
```

- Tableau de n éléments, demande n-1 éléments

```
UtilHtml uh = new UtilHtml();
String arg[] = {"A", "B", "C"};
String expectedHtmlTab =
    "<table><tr><td>A</td></tr><tr><td>...</td></tr></table>";
assertEquals(expectedHtmlTab, uh.toHtmlTab(arg, arg.length-1));
```

- Tableau de n éléments, demande n/2 éléments

```
UtilHtml uh = new UtilHtml();
String arg[] = {"A", "B", "C"};
String expectedHtmlTab =
    "<table><tr><td>...</td></tr></table>";
assertEquals(expectedHtmlTab, uh.toHtmlTab(arg, arg.length/2));
```

Les « cas limites » des cas « limites »

Doit-on tester le fonctionnement en cas de références **null** ?

La réponse est non ! La programmation par contrat a montré qu'il faut responsabiliser l'appelant.

Doit-on tester le fonctionnement en cas de valeur nulle ou négative de maxRows ?

La réponse est plus nuancée. La sémantique de ce paramètre (côté appelé) n'est pas assez précise.

Cette question nous amène à renforcer les spécifications de ce paramètre. Exemple :

```
public class UtilHtml {  
  
    /** retourne une représentation HTML d'un tableau...  
    * @param tab  
    *         le tableau  
    * @param maxRows  
    *         le nombre maximum de lignes souhaité (doit être >= 0)  
    * @return Si le tableau est vide, la méthode retourne la chaîne "(vide)"  
    *         si le tableau contient moins de maxRows lignes, le deuxième  
    *         paramètre). Il y a autant de <tr>  
    *         que d'éléments dans le tableau à concurrence de maxRows lignes,  
    *         s'il le nombre d'éléments dans le tableau est supérieur à maxRows,  
    *         la dernière ligne sera définie par "..."  
    */  
    public String toHtmlTab(Object[] tab, int maxRows) {  
        ...  
    }  
}
```

- Tableau de n éléments, demande 0 élément

```
UtilHtml uh = new UtilHtml();  
String arg[] = {"A", "B", "C"};  
String expectedHtmlTab = ""; // EST-CE UNE BONNE SOLUTION ?  
assertEquals("toHtmlTab : cas maxRows = 0",  
             expectedHtmlTab, uh.toHtmlTab(arg, 0));
```

À voir également la relation entre l'attendu fonctionnel et les spécifications : [BDD \(Behaviour-Driven Development\)](#)

Exemples d'outils

- PHP : [PHPUnit](#), [Atoum](#), [SimpleTest](#)
- JAVA : [JUnit](#)
- etc.

...

Références

[Test Driven Development: By Example - Kent Beck](#)

[Scott W. Ambler : une introduction au Développement Guidé par les Tests \(TDD\)](#)

[Institut Agile : Développement par les tests et présentation des principaux concepts liés](#)

...

Travaux Pratiques avec PHP + Symfony

Approche *Test First*

A/ Installation de PHPUnit & Co

PHPUnit est un des nombreux dérivés du projet open source JUnit des auteurs [Kent Beck](#) et [Erich Gamma](#). Voici une procédure d'utilisation (à partir de la racine d'un projet symfony)

1/ Installer PHPUnit da façon locale à votre projet actuel, en mode dev

```
composer require --dev phpunit
```

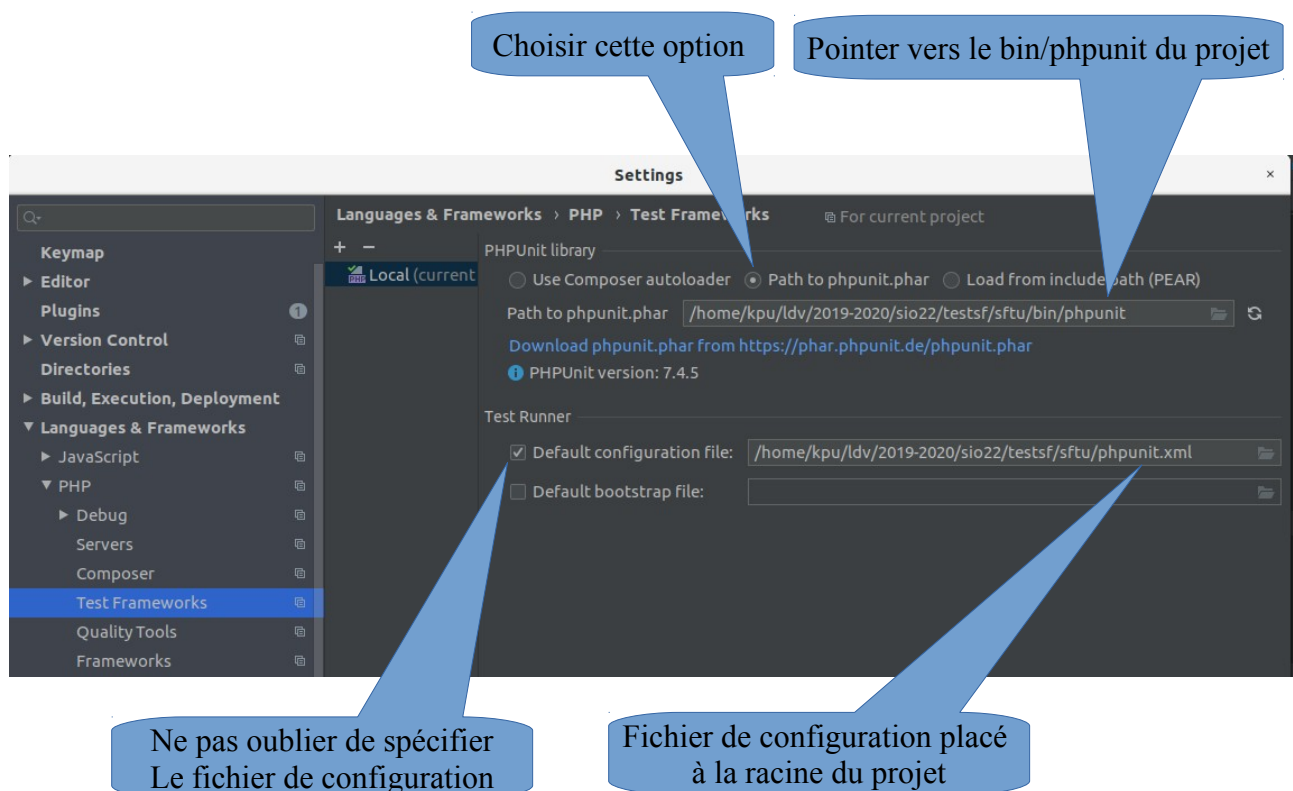
Ensuite, suivant les recommandations de Symfony, installer le bridge :

```
composer require --dev symfony/phpunit-bridge
```

A ce niveau, vous devrez avoir le programme phpunit dans le dossier bin de l'application, ainsi qu'un fichier de configuration à la racine de votre projet nommé : phpunit.xml.dist. Si ce n'est pas le cas, il faut le faire générer par la commande :

```
bin/phpunit --generate-configuration  
=> accepter les valeurs proposées par défaut
```

2/ Configurer PHPStorm suite à ces modifications (à faire aussi si vous télécharger le projet exemple).



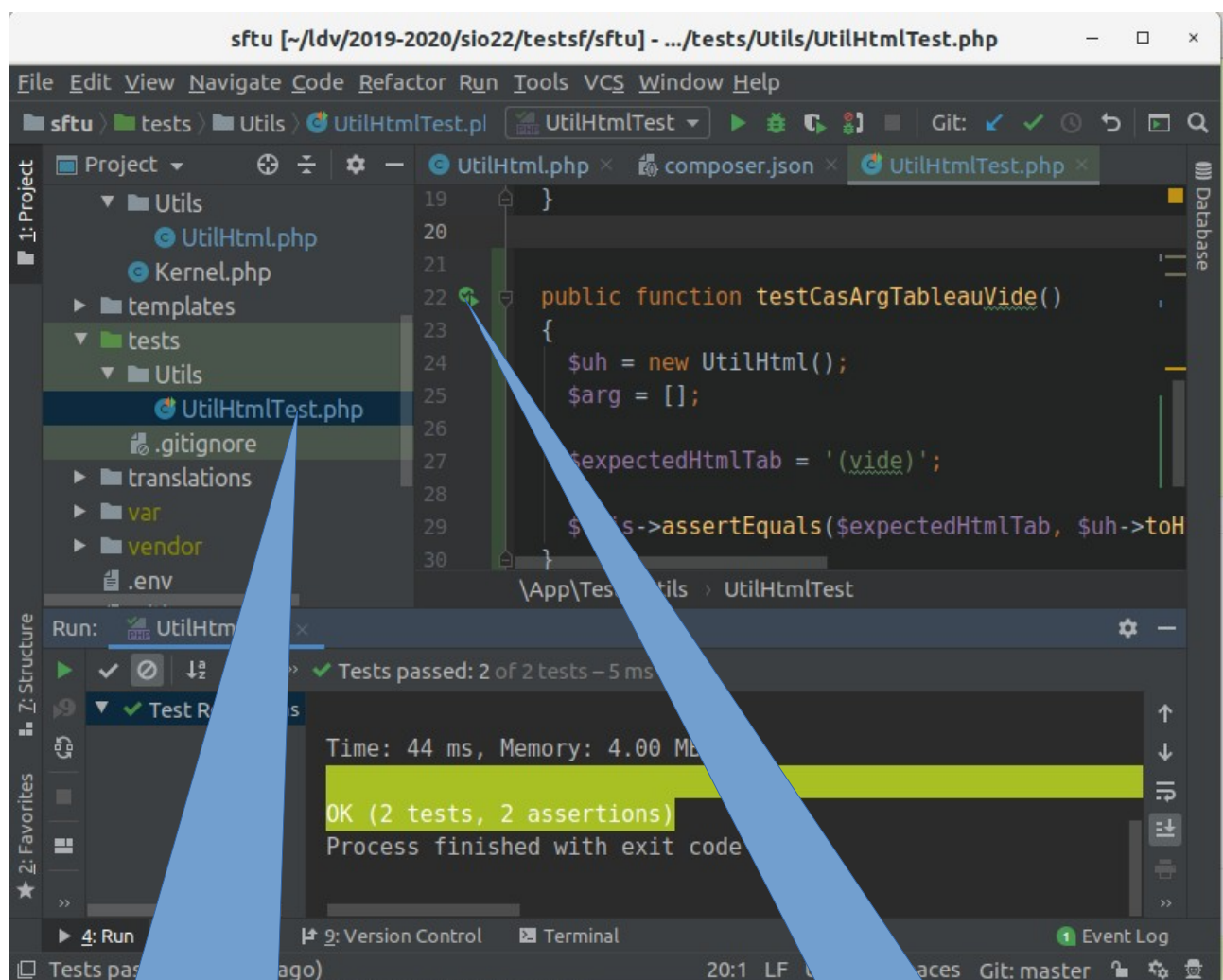
3/ Application setup

Nous vous proposons une application `sftu` intégrant un test unitaire opérationnel.
Avec git, installer l'application `sftu` **à partir de votre racine des projets**:

```
git clone https://github.com/ocapuozzo/sftu
cd sftu
```

À partir de la racine du projet, faire un `composer update` afin de télécharger ses dépendances.
En effet, il est inutile (même nuisible) d'intégrer des bibliothèques tierces sur un dépôt distant, c'est le rôle du fichier `.gitignore` à la racine du projet (renseigné par défaut par symfony). N'oubliez pas de configurer PhpStorm (voir étape 2)

Lancement des tests



Un clic droit sur le nom du Fichier,
puis sélectionner : Run ... (PHPUnit)
pour lancer **tous** les tests de cette classe

Un clic ici pour lancer
la méthode de test
de cette ligne

Pour pouvez aussi passer directement par le terminal et lancer la commande (et obtenir le même résultat) : **php bin/phpunit tests/Utils/UtilHtmlTest.php**

Sinon, vous pouvez partir d'un projet from scratch, mais attention de bien respecter les arborescences et règle de nommage (la classe de test reprend le nom de la classe testée et y ajoute Test (par exemple `UtilHtmlTest`) et les méthodes sont préfixées par `test` (par exemple `test3elements()`). Voir les instructions ici : <https://symfony.com/doc/current/testing.html#the-phpunit-testing-framework> (en vous limitant aux test unitaires – cette ressource traitant également les tests fonctionnels)

TP

B/ Programmation des tests et conception progressive de l'objet métier

Implémenter l'exemple du support de cours.

1. Concevoir la classe de test `UtilHtmlTest` et y implémenter, **une par une**, les méthodes de test présentées **en ne codant une nouvelle méthode de test que si les tous les tests passent**. La méthode `toHtmlTab` de la classe utilitaire `UtilHtml` devra, in fine, passer tous les tests (c'est l'objet « métier »)
2. Traiter le cas où `maxRows=0`

C/ Développement en mode *Test First*

Un collègue vous demande d'écrire une classe de test (à ajouter au projet actuel) pour tester la méthode suivante, d'une classe nommée `C` (en java à transposer en PHP Objet) :

```
// class C
// ...
// Params : tab
//   (non null), une référence à un tableau de double (flottant sur 64 bits)
// Returns :
//   nombre d'éléments de tab de valeur maximale.
// Exception : EmptyArgumentException
//   si tab est vide
// Exemple : si on passe à nMax le tableau double[] t = {1,3,2,3},
//           alors nMax retournera 2
// public int nMax(double[] tab)

// fin classe C
```

4. Concevoir, dans une approche itérative et incrémentale, la classe de test demandée, comportant au moins **4 méthodes** de test traitant **un cas général** et **des cas aux limites**.