TDD, de kossé !?

Introduction au *Test-Driven Development* avec Laravel

Programmation sans TDD (tests manuels)

- Le programmeur reçoit sa tâche.
- Il dessine des schémas dans son calepin pour séparer les différentes étapes de la fonctionnalité.
- Il code ses fonctionnalités.
- Il teste chaque étape manuellement avec des *dumps* réguliers (s'il est consciencieux!)
- Il vérifie si la fonctionnalité entière fonctionne comme convenue.
- Il réécrit le code de manière plus claire.
- Il *reteste* chaque étape manuellement avec des *dumps* réguliers (s'il est consciencieux!)

Programmation sans TDD (suite)

- Le programmeur reçoit sa deuxième tâche.
- Il dessine des schémas dans son calepin pour séparer les différentes étapes de la fonctionnalité.
- Il code ses fonctionnalités.
- Il teste chaque étape manuellement avec des *dumps* réguliers (s'il est consciencieux!)
- Il vérifie si la fonctionnalité entière fonctionne comme convenue.
- Il réécrit le code de manière plus claire.
- Il *reteste* chaque étape manuellement avec des *dumps* réguliers (s'il est consciencieux!)
- Il réessaye la première fonctionnalité... mais s'aperçoit qu'elle s'est cassée avec le codage de la deuxième (régression).
- Il *reteste* chaque étape manuellement la première fonctionnalité avec des *dumps* réguliers (s'il est consciencieux!)
- ... Et chaque fonctionnalité augmente la charge de travail et le stress du programmeur !

Tests automatiques

- Le programmeur reçoit sa tâche.
- Il fait un test qui teste le tout début de sa fonctionnalité (ex.: une exception levée de validation).
- Il roule le test afin de vérifier qu'il échoue, pour vérifier la véracité du test.
- Il fait le programme minimum pour faire passer le test.
- Il roule le test afin de vérifier qu'il passe.
- Il réécrit le code pour plus de clarté.
- Il roule le test afin de vérifier qu'il passe toujours.
- Il fait un autre test pour la condition suivante de sa fonctionnalité...

Tests automatiques (suite)

- Le programmeur reçoit sa deuxième tâche.
- Il fait un test qui teste le tout début de sa tâche (souvent une exception levée).
- Il roule les tests afin de vérifier que ce test échoue, mais que les autres passent toujours.
- Il fait le programme minimum pour faire passer le test qui échoue.
- Il roule les tests afin de vérifier que tous les tests passent.
- Il réécrit le code pour plus de clarté.
- Il roule les tests afin de vérifier qu'ils passent toujours.

Développement *dirigé* par tests

- Le test est écrit au tout début.
- Il *dirige* le développement.
- Le développement sert avant tout à faire réussir le test.
- Écriture du test = conception de la fonctionnalité.

TDD = démarche scientifique

- Karl Popper : pour qu'elle soit scientifique, une théorie doit avoir des expériences cruciales qui fournissent des possibilités de réfutation.
- On pose une hypothèse (test).
- On vérifie qu'elle est fausse avant notre code (réfutation).
- On fait le code (expérience).
- On vérifie que l'hypothèse devient vrai avec le code (corroboration).

Types de test

- **Tests unitaires *(unit tests)* :** testent seulement la classe, les dépendances sont entièrement «moquées» (*mocked*).

- **Tests fonctionnels (*feature/functional tests)* :** testent une fonctionnalité ou une classe avec ses dépendances réelles.

- **Tests d'intégration *(integration/acceptance tests)* :** testent un scénario utilisateur complet.

Tests unitaires

- Testent une seule unité (souvent la classe).
- Tout le reste du programme interagissant avec l'unité est simulé.
  - Objets *stub*
  - Objets *mock*
- Utilisés pour les calculateurs, les transformateurs (sans bases de données).

Objet *Stub*

- Un objet fait manuellement par le programmeur.
- Imitation moins fine de la classe (souvent par interface).
- Réduit à sa plus simple expression pour retourner seulement les valeurs nécessaires aux tests.

Objet *Mock*

- Un objet fait par une librairie spécialisée.
  - Imitation plus fine de la classe (passe pour la classe elle-même).
  - A toute l'ossature des méthodes de la classe.
  - Les valeurs de retour des méthodes sont déclarées par des méthodes spéciales de la librairie.

Tests fonctionnels

- Testent une unité avec toutes ses sous-unités (dépendances et arguments).

- Toutes les entités sont réelles (aucun *stub* ou *mock*).

- Utilisés pour les tests nécessitant les bases de données et les appels d'API.

  - Nécessite un traitement pour rafraichir la base de données à chaque test.

Tests d'intégration

- Imite l'usager qui utilise l'application dans un scénario donné.
    - «En tant qu'usager, je veux créer une facture afin d'en garder une copie pour des usages ultérieurs».
- Demande un navigateur (ou une simulation) et souvent une librairie de tests différents.

Couverture de code

- Idéalement, dans du TDD, nous voulons que les tests unitaires/fonctionnels couvrent 100% du code d'application.

- Les librairies de test ont des outils pour savoir ce qui est couvert ou non par les tests.

Dans Laravel

- Tests unitaires : PHPUnit + Mockery
- Tests fonctionnels : PHPUnit + extensions Laravel
- Tests d'intégration : Laravel Dusk
- Autres possibilités :
  - Atoum : tests unitaires et fonctionnels
  - Codeception : tous les tests

Références

- PHPUnit (https://phpunit.de)
- Mockery (http://docs.mockery.io/en/latest/)
- *Karl Popper et les critères de la scientificité* (https://philosciences.com/philosophie-et-science/methode-scientifique-paradigme-scientifique/112-karl-popper-et-les-criteres-de-la-scientificite)
- *Setting up XDebug with PHPStorm and Laravel Homestead* (https://edcs.me/blog/setting-up-xdebug-with-phpstorm-and-laravel-homestead/)
- *Using Laravel Dusk with Vagrant Homestead* (https://www.jesusamieiro.com/using-laravel-dusk-with-vagrant-homestead/)