Introduction au développement piloté par les tests

Une autre façon de développer

Par Paul Underwoord - Sébastien Germez (traducteur)

Date de publication : 9 avril 2013

Dernière mise à jour : 9 avril 2013

Dans cet article nous allons expliquer ce qu'est le développement piloté par les tests et voir quelques cas d'utilisation.

Commentez

I - Traduction

Ce tutoriel est la traduction la plus fidèle possible du tutoriel original de **Paul Underwood**, **Test Driven Development**.

II - Introduction

À travers cet article, nous allons expliquer à quoi correspond un développement piloté par les tests. Nous verrons l'utilisation de cette méthode de travail par le biais d'exemples et comment celle-ci va nous permettre d'améliorer notre processus de développement.

III - Qu'est-ce que le développement piloté par les tests ?

Le développement piloté par les tests (couramment appelé BDD pour Behavior driven development sur la toile) est un processus de développement qui correspond exactement à son nom, c'est-à-dire que le développement de notre application sera fait en fonction d'un certain nombre de tests. Ce qui implique d'écrire tous les tests avant de commencer n'importe quel développement. À première vue, cela peut paraître étrange comme technique dans la mesure où tous les tests échoueront comme rien n'a encore été développé. Cependant, cette technique permet justement de développer l'application pour faire en sorte que tous les tests écrits réussissent.

Le développement piloté par les tests peut vous permettre de rendre votre code plus simple, plus solide et d'éviter de développer n'importe quoi. C'est parce que vous n'avez plus besoin d'écrire que le code nécessaire pour passer vos tests et rien d'autre. Vous pouvez aussi vous assurer de ne pas développer des fonctionnalités qui ne sont pas requises en écrivant vos tests en premier et en les lançant ; si les tests réussissent la fonctionnalité existe déjà, nul besoin de l'écrire.

Ces deux principes de développement s'appellent :

- KISS : keep it simple, stupid ;
- YAGNI : you ain't gonna need it.

Comme le développement va être piloté par des tests vous aurez plus de code à écrire et l'application sera un peu plus longue à développer, mais le code finalement produit sera plus sûr, car il aura été correctement testé et que ces tests seront documentés. Avec une bonne structure de tests, vous vous assurez de livrer une application sans bogues, qui prennent en général du temps à corriger après coup.

Cela signifie que si vous devez ajouter une nouvelle fonctionnalité à un programme existant, vous pouvez simplement la développer et lancer ensuite tous les tests pour vous assurer qu'elle fonctionne et qu'aucun effet de bord n'est apparu.

La façon dont le développement piloté par les tests est effectué va conduire le développeur à écrire du code plus extensible, flexible et modulaire. Cette méthode de travail nécessite que le développeur crée de petites portions de code et se concentre sur les classes pour ne les relier entre elles qu'à la fin.

IV - Créer une nouvelle fonctionnalité en utilisant le développement piloté par les tests

- Écrivez vos tests.
- Lancez tous les tests et assurez-vous qu'ils sont en échec.
- Écrivez le code nécessaire pour réussir un test.
- Relancez tous les tests pour vous assurer que seul celui-ci passe.
- Prenez un autre test et écrivez le code nécessaire pour qu'il réussisse.
- Continuez ainsi jusqu'à ce que tous les tests réussissent.
- Factorisez votre code.
- Assurez-vous de nouveau que tous les tests passent.

- La nouvelle fonctionnalité est implémentée.

Il y a un processus qui doit être suivi lorsque vous développez votre application avec un pilotage par les tests : pour implémenter une nouvelle fonctionnalité, vous devez en premier lieu écrire le test qui va vérifier le fonctionnement de cette dernière. Le test va tout d'abord échouer puisque vous n'aurez pas encore écrit le code fonctionnel. L'avantage de commencer par écrire le test est que vous vous assurez de comprendre les prérequis pour que la fonctionnalité marche avant d'écrire le code. Cela vous permettra de voir toutes les situations dans lesquelles la nouvelle fonctionnalité pourrait ne pas fonctionner et de pallier ces situations.

La deuxième étape est de lancer tous les tests et de s'assurer que seuls les nouveaux échouent. Si ce n'est pas le cas, c'est que le code nécessaire à l'implémentation de votre fonctionnalité existe déjà et qu'il n'y a rien à faire.

Ensuite, nous pouvons écrire le code fonctionnel nécessaire à faire réussir un test. Relancez tous les tests, assurez-vous que celui pour lequel vous venez d'écrire du code passe et que les autres échouent (on parle ici des tests relatifs à la nouvelle fonctionnalité). Vous pouvez ensuite choisir un autre test, écrire le code et relancez les tests pour voir que celui que vous venez de choisir passe. Attention vous ne devez en aucun cas supprimer du code pour faire fonctionner un nouveau test ! À cette étape, la façon dont vous écrivez votre code importe peu.

Relancez à nouveau tous les tests. Si chacun des tests réussit, vous pouvez factoriser votre code pour qu'il soit plus propre, lisible et maintenable. Une fois que vous avez fini la refactorisation, relancez les tests pour vous assurer qu'ils réussissent toujours.

Une fois que tous vos tests passent et que votre code est propre, vous avez terminé. Vous pouvez répéter le processus pour la prochaine fonctionnalité.

V - Exemple d'un développement piloté par les tests

Dans notre exemple, nous allons développer une calculatrice basique, nous savons quelles fonctionnalités sont nécessaires : l'addition, la soustraction, la multiplication et la division de nombres. Nous pouvons donc écrire les tests correspondants, ce qui va nous aider à comprendre le comportement attendu de ces quatre fonctionnalités. Puis nous pourrons les développer.

Pour le premier test, nous allons additionner deux nombres. Nous savons que nous allons avoir besoin d'écrire une fonction qui prend deux nombres en paramètres d'entrée et qui retourne un nombre valant l'addition des deux nombres en entrée.

```
function add( $number1, $number2 )
{
    return $number1 + $number2;
}
```

Si nous lançons ce test, il va réussir et la valeur de retour sera celle attendue.

Cependant nous avons aussi besoin de tester des cas où le test échouera : que se passera-t-il si un des deux nombres est nul ? Quel est le comportement attendu ? Même chose si un des deux paramètres est une chaîne de caractères au lieu d'un nombre.

On va donc créer des tests pour ces cas de figure. On peut par exemple créer un test qui prend une valeur nulle en entrée ; le test échouera. Nous pouvons donc maintenant modifier notre fonction de départ pour prendre en compte ce cas de figure.

```
function add( $number1, $number2 )
{
    if(is_null( $number1 ) || is_null( $number2 ))
    {
        return false;
    }
```

```
    return $number1 + $number2;
}
```

On peut maintenant relancer ce test avec une valeur nulle en premier ou second paramètre et il retournera false. Le test est donc réussi.

On va ensuite modifier la fonction pour prendre en compte le cas où une des deux valeurs (voire les deux) n'est pas un nombre (entier ou décimal).

```
function add( $number1, $number2 )
{
    // If the value is null or not a number then return false
    if(is_null( $number1 ) ||
       is_null( $number2 ) ||
       !is_numeric( $number1 ) ||
       !is_numeric( $number2 ))
    {
        return false;
    }
    return $number1 + $number2;
}
```

Si on lance le test en passant une valeur nulle ou une valeur non numérique, le test retournera false et sera donc réussi.

Comme on peut le voir, cette fonction est maintenant plus complète que ce qu'elle était au début de l'exercice, car on lui a ajouté un certain nombre de vérifications pour s'assurer que les valeurs d'entrée sont bien celles attendues.

Mais nous pouvons encore améliorer ce code en le factorisant : nous allons retirer les fonctions is_null qui ne sont plus nécessaires. La fonction is_numeric seule suffit pour tester à la fois si les valeurs d'entrée sont numériques et non nulles.

```
function add( $number1, $number2 )
{
    // If the value is null or not a number then return false
    if(!is_numeric( $number1 ) || !is_numeric( $number2 ))
    {
        return false;
    }
    return $number1 + $number2;
}
```

C'est une très bonne façon de programmer dans la mesure où vous êtes le seul à utiliser cette fonction et que vous savez donc à l'avance quel est le comportement attendu. Mais que se passe-t-il si vous faites partie d'une équipe de développeurs et que l'un d'entre eux veut implémenter votre fonction ? Il pourra essayer de passer des valeurs nulles ou encore des chaînes de caractères et ne comprendra pas pourquoi la fonction ne fonctionne pas. C'est pourquoi vous devez le diriger vers vos tests pour qu'il en comprenne le fonctionnement exact.

C'est un gros bénéfice dans une équipe, car le développeur n'a plus besoin d'expliquer comment se comporte sa fonction, les tests le font à sa place. C'est ce qu'on appelle du code documenté seul, ce sont les tests qui font office de documentation pour les fonctions.

VI - Conclusion

Bien évidemment notre exemple est très simple et le code qui en découle également. Mais il permet quand même de voir les bénéfices de cette méthode. Essayez de l'utiliser dans vos projets ! Vous vous assurerez de n'écrire que le code nécessaire, que l'implémentation d'une nouvelle fonctionnalité ne crée pas d'effets de bord et de trouver tous les cas de figure dans lesquels votre fonctionnalité ne marchera pas.

VII - Remerciements

Je tiens à remercier **Paul Underwood** de m'avoir autorisé à traduire son tutoriel.
Je remercie également **mumen** pour sa relecture orthographique.