Faire un test à la fois vous met sur la bonne voie (Libres conseils 15/42)

Chaque jeudi à 21h, rendez-vous sur le framapad de traduction, le travail collaboratif sera ensuite publié ici même.

Traduction Framalang : lamessen, Sky, Kalupa, ga3lig, Wxcafe, goofy, Astalaseven, Slystone, okram, KoS, Lycoris, peupleLa, Julius22

La Voie des tests conduit à la Lumière

Jonathan "Duke" Leto

Jonathan Leto, dit « Le Duc » est un développeur de logiciel, un mathématicien dont les travaux sont publiés, un ninja de Git et un passionné de cyclisme qui vit à Portland, en Oregon. C'est l'un des développeurs principaux de la machine virtuelle Parrot et le fondateur de Leto Labs LLC.

Lorsque j'ai commencé à m'impliquer dans le logiciel libre et open source, je n'avais aucune idée de ce que pouvaient être les tests ni de leur importance. J'avais travaillé sur des projets personnels de programmation auparavant, mais la première fois que j'ai réellement travaillé sur un projet collaboratif (c'est-à-dire en faisant un peu de *commit*) c'était pour Yacas, acronyme de Yet Another Computer Algebra System, (NdT : encore un autre logiciel de calcul algébrique similaire à Mathematica).

À ce stade de mon parcours, les tests ne venaient qu'après coup. Mon méta-algorithme général était : bidouiller du code > voir si ça fonctionne> écrire (éventuellement) un test simple pour démontrer que ça fonctionne. Un test difficile à écrire n'était généralement jamais écrit.

C'est le premier pas sur la voie qui mène à la Lumière grâce aux tests. Vous savez que les tests sont probablement une bonne idée, mais vous n'en avez pas vu clairement les bénéfices, alors vous vous contentez de les écrire de temps en temps.

Si je pouvais ouvrir un trou de souris dans l'espace-temps et donner à mon moi plus jeune un conseil plein de sagesse sur les tests, ce serait :

> *« **Certains tests sont plus importants, sur le long terme, que le code qu'ils testent.** »*

Il y a sans doute quelques personnes qui pensent en ce moment même que je mets mon casque de protection psychique (NdT : il s'agit d'un chapeau pour se protéger contre la manipulation à distance du cerveau) quand je m'assois pour écrire du code. Comment les tests pourraient-ils être plus importants que le code qu'ils testent ? Les tests sont la preuve que votre code marche réellement ; ils vous montrent le chemin vers l'écriture d'un code propre et vous apportent aussi la souplesse qui vous permettra de modifier le code tout en sachant que les fonctionnalités seront toujours là. En effet, plus votre code source grossit, plus vos tests sont importants, car ils vous permettent de changer une partie dudit code en sachant que le reste fonctionnera.

Une autre raison essentielle qui justifie l'écriture de tests est la possibilité de spécifier que quelque chose est un souhait explicite et non une conséquence imprévue ou un oubli. Si vous avez un cahier des charges, vous pouvez utiliser des tests pour vérifier qu'il est respecté, ce qui est très important, voire indispensable dans certaines industries. Un test, c'est comme quand on raconte une histoire : l'histoire de votre conception du code et de la façon dont il devrait fonctionner. Soit le code change et évolue, soit il mute en code infectieux (1).

Très souvent, vous écrirez des tests une première fois pour ensuite remettre totalement en cause votre réalisation voire la réécrire à partir de zéro. Les tests survivent souvent au code pour lesquels ils ont été conçus à l'origine. Par exemple, un jeu de tests peut être utilisé quel que soit le nombre de fois où votre code est transformé. Les tests sont en fait l'examen de passage qui vous permettra de jeter une ancienne réalisation et de dire « cette nouvelle version a une bien meilleure structure et passe notre jeu de tests ». J'ai vu cela se produire bien des fois dans les communautés Perl et Parrot, où vous pouvez souvent me voir traîner. Les tests vous permettent de changer les choses rapidement et de savoir si quelque chose est cassé. Ils sont comme des propulseurs pour les développeurs.

Les charpentiers ont un adage qui dit quelque chose comme ça :

> *« Mesurer deux fois, couper une fois. »*

Le code serait la coupe, le test serait la mesure.

La méthode de développement basée sur les tests fait économiser beaucoup de temps, parce qu'au lieu de vous prendre la tête à bricoler le code sans but défini, les tests précisent votre objectif.

Les tests sont aussi un très bon retour d'expérience. Chaque fois que vous faites une nouvelle passe de test, vous savez que votre code s'améliore et qu'il a une fonctionnalité de plus ou un bogue de moins.

Il est facile de se dire « je veux ajouter 50 fonctionnalités » et de passer toute la journée à bricoler le code tout en jonglant en permanence entre différents travaux. La plupart du temps, peu de choses aboutiront. La méthode de développement basée sur les tests aide à se concentrer sur la réussite d'un seul test à la fois.

Si votre code échoue devant ne serait-ce qu'un seul test, vous avez pour mission de le faire réussir. Votre cerveau se concentre alors sur quelque chose de très spécifique, et dans la plupart des cas cela produit de meilleurs résultats que passer constamment d'une tâche à une autre.

La plupart des informations relatives au développement basé sur les tests sont très spécifiques à un langage ou à une situation, mais ce n'est pas une obligation. Voilà comment aborder l'ajout d'une nouvelle fonctionnalité ou la correction d'un bogue dans n'importe quel langage :

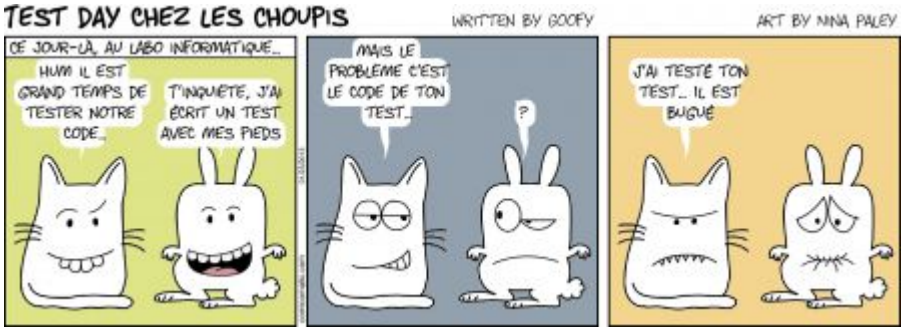
1. Écrivez un test qui fait échouer votre code, mais qui, selon vous, sera passé quand la fonctionnalité sera implémentée ou que le bogue sera corrigé. Mode expert : pendant l'écriture du test, pensez à l'exécuter de temps en temps, même s'il n'est pas encore fini, et tentez de deviner le message d'erreur effectif que le test renverra. À force d'écrire des tests et de les faire tourner, cela devient plus facile ;
2. Bidouillez le code ;
3. Exécutez le test. S'il marche, passez au point 4, sinon retournez au point 2 ;
4. C'est fini ! Dansez le sirtaki.

Cette méthode fonctionne pour n'importe quel type de test et n'importe quel langage. Si vous ne deviez retenir qu'une seule chose de ce texte, souvenez-vous des étapes ci-dessus.

Voici maintenant quelques directives plus générales de conduite de tests qui vous serviront bien et fonctionneront dans n'importe quelle situation :

1. Comprendre la différence entre ce qu'on teste et ce qu'on utilise comme un outil pour tester autre chose ;
2. Les tests sont fragiles. Vous pouvez toujours écrire un test qui contrôle la validité d'un message d'erreur. Mais que se passera-t-il si le message d'erreur change ? Que se passera-t-il quand quelqu'un traduira votre code en catalan ? Que se passera-t-il lorsque quelqu'un exécutera votre code sur un système d'exploitation dont vous n'avez jamais entendu parler ? Plus votre test est résistant, plus il aura de valeur.

Pensez à cela quand vous écrivez des tests. Vous voulez qu'ils soient résistants, c'est-à-dire que les tests, dans la plupart des cas, ne devraient avoir à changer que quand les fonctionnalités changent. Si vous devez modifier vos tests régulièrement, sans que les fonctionnalités aient changé, c'est que vous faites une erreur quelque part.



Types de tests

Bien des personnes sont perdues quand on leur parle de tests d'intégration, tests unitaires, tests d'acceptation et autres tests à toutes les sauces. Il ne faut pas trop se soucier de ces termes. Plus vous écrirez de tests, mieux vous en distinguerez les nuances et les différences entre les tests deviendront plus apparentes. Tout le monde n'a pas la même définition de ce que sont les tests, mais c'est utile d'avoir des termes pour décrire les types de tests.

Tests unitaires contre tests d'intégration

Les tests unitaires et les tests d'intégration couvrent un large spectre. Les tests unitaires testent de petits segments de code et les tests d'intégration vérifient comment ces segments se combinent. Il revient à l'auteur du test de décider ce que comprend une unité, mais c'est le plus souvent au niveau d'une fonction ou d'une méthode, même si certains langages appellent ces choses différemment.

Pour rendre cela un peu plus concret, nous établirons une analogie sommaire en utilisant des fonctions. Imaginez que f(x) et g(x) soient deux fonctions qui représentent deux unités de code. Pour l'aspect concret, supposons qu'elles représentent deux fonctions spécifiques du code de base de votre projet libre et open source.

Un test d'intégration affirme quelque chose comme la composition de la fonction, par exemple $f(g(a)) = b$. Un test d'intégration consiste à tester la façon dont plusieurs choses s'intègrent ou travaillent ensemble, plutôt que la façon dont chaque partie fonctionne individuellement. Si l'algèbre n'est pas votre truc, une autre façon de comprendre est de considérer que les tests unitaires ne testent qu'une partie de la machine à la fois, tandis que les tests d'intégration s'assurent que la plupart des parties fonctionnent à l'unisson. Un bel exemple de test d'intégration est le test de conduite d'une voiture. Vous ne vérifiez pas la pression atmosphérique, ni ne mesurez le voltage des bougies d'allumage. Vous vous assurez que le véhicule fonctionne globalement.

La plupart du temps, il est préférable d'avoir les deux. Je commence souvent avec les tests unitaires puis j'ajoute les tests d'intégration au besoin puisqu'on a besoin d'éliminer d'abord les bogues les plus basiques, puis de trouver les bogues plus subtils issus d'un emboitement imparfait des morceaux, à l'opposé de pièces qui ne fonctionnent pas individuellement. Beaucoup de gens écrivent d'abord des tests d'intégration puis se plongent dans les tests unitaires. Le plus important n'est pas de savoir lequel vous écrirez en premier, mais d'écrire les deux types de tests.

Vers la Lumière

La méthode de développement basée sur les tests est un chemin, pas un aboutissement. Sachez apprécier le voyage et assurez-vous de faire une pause

pour respirer les fleurs si vous êtes égaré.

(1) Équivalent approché du terme *bitrot* qui en argot de codeur désigne ce fait quasi-universel : si un bout de code ne change pas mais que tout repose sur lui, il « pourrit ». Il y a alors habituellement très peu de chances pour qu'il fonctionne tant qu'aucune modification ne sera apportée pour l'adapter à de nouveaux logiciels ou nouveaux matériels.

Comic strip réalisé avec le Face-O-Matic de Nina Paley et Margot Burns

Copyheart ? 2011 by Margo Burns and Nina Paley. Copying is an act of love. Please copy!