Pratiquer le Test Driven Development - TDD

Master info S2

Université Lille1 - FST | TD SVL – 2017-2018 | Département informatique

L'approche classique du test (qui n'est pas le TDD) consiste à coder puis tester à la fin. Vous l'expérimentez en TP, ça ne marche pas :

- il ne reste plus de temps pour tester ;
- plus le bug est détecté tard, plus il remet de choix en cause, et plus sa correction est coûteuse ;
- comme le code est déjà écrit, on risque d'écrire — le nez sur le code — un test faux qui valide un code faux ;
- or un code non ou mal testé n'a aucune valeur.

Le cycle de développement en V / cascade est source de l'échec de nombreux projets :

Effet tunnel Dans le meilleur des cas le cahier des charges est rédigé avec le client (c'est pire s'il vient avec). Ensuite l'équipe attaque la réalisation sans plus interagir avec le client. Quand elle ressort du *tunnel* (si elle ressort) au moment de montrer le produit au client lors du test de recette, il s'avère fréquemment que les *besoins du client ont changé*, ou que ses besoins trop imprécis ont été mal interprétés par l'équipe.

Conception puis codage L'approche « je conçois un design et rédige une spécification détaillée type UML de toute l'application puis je code » peut très bien ne pas marcher. On peut très bien découvrir au codage un problème qui remet en cause le design et était invisible au stade des abstractions. On a perdu son temps et mis le projet en péril.

Les méthodes agiles (Manifesto for Agile Software Development : 2001, test first, eXtreme programming, TDD...) font le constat de l'échec du V et préconisent un cycle de développement :

- basé fondamentalement sur les tests ;
- *incrémental* : fonctionnalité par fonctionnalité avec validation du client à chaque fonctionnalité ;
- *itératif* : on enchaîne les étapes conception-code-test sur des cycles courts (ex : sprint de 2 semaines).

On avance à petits pas testés, et on valide chaque pas fréquemment avec le client.

En TDD (TDD by example - Kent Beck : 2002)

- on écrit le test **avant** d'écrire le code, ce qui oblige à réfléchir à ce que fait le code avant de coder ;
- une ligne de code n'est écrite que si un test la rend nécessaire, ce qui rend impossible la livraison de code non testé ;
- dans la mesure du possible on laisse l'architecture émerger au fil du développement (ce qui n'interdit pas une phase légère de conception à base d'UML), ce qui garantit une *architecture testable* et *faiblement couplée*. En ce sens, le TDD est plus une approche de conception objet de qualité que de recherche de bugs ;
- on considère les tests comme une *documentation exécutable* de l'API du système, une spécification par l'exemple, et on les bichonne autant que son code.

Le mantra du TDD est *Keep the bar green to keep the code clean* ou (K Benck) *Write a test, Make it run, Make it right* ou *Red - Green - Refactor* (cf Fig « simple TDD cycle », la flèche horizontale est rouge, les autres sont vertes). Plus précisément :

- écrire un test ;
- l'exécuter et constater qu'il échoue (barre rouge) ; si le verdict est en fait une *erreur* due au fait que le code ne compile pas (en Java) ou que le code n'existe pas (en Python) car le code applicatif n'a pas encore été écrit, alors écrire le code applicatif minimal du point de vue du langage, et vérifier cette fois que le test échoue à cause de l'oracle ;
- écrire le code applicatif *le plus simple* qui permet de faire passer le test, et seulement ce code ;
- lancer le test et vérifie qu'il passe (barre verte) ;
- réusiner les tests et le code.

Il est important de bien lire le message d'erreur lié à la barre rouge (Fig « Feedback on test diagnostics »).

Le code applicatif qui fait passer le test est « le plus simple » pour éviter les grands pas. K Beck propose 3 approches :

- *Fake it till still you make it* : on ne sait pas trop quel code écrire, on utilise une constante en dur qui est l'exemple spécifié par le test ;
- *triangulation* : le code contient déjà un ou plusieurs exemles, on le généralise à tous les cas possibles ;
- *use an obvious implementation* pour le cas où on voit tout de suite quel code écrire (en gardant bien à l'esprit que toute évidente qu'elle soit, l'implémentation a peut-être un problème de borne, de boucle, etc) ;

Quoi tester ? On reprend l'approche fonctionnelle vue en début d'UE, mais on l'applique *avant d'écrire le code* :

- on liste les fonctionnalités du système / on les priorise
- on choisit la fonctionnalité qui a le + de sens / la prio la + forte
- on identifie ses comportements
- on choisit un comportement
- on écrit un test qui illustre son fonctionnement
- et c'est parti !

Avantages psychologiques Grâce aux petits pas on a moins peur d'attaquer une tâche complexe, ce qui réduit le risque de procrastination. L'alternance rouge /vert donne le sentiment d'avancer. Les tests agissent comme un filet de sécurité : on est plus serein et on réduit le risque de panique de dernière minute.

Avantages techniques On passe beaucoup moins de temps à déboguer, on a toujours qqchose à montrer au client (dont des tests), le test de non-régression est facile, l'architecture est testable et de bonne qualité. Quand on écrit le test, on se réfère uniquement à la spécification et on réduit le risque de valider un code faux par un test faux. Les tests comme documentation facilitent la maintenance et la reprise du code applicatif par un autre collègue.

FAQ

Pourquoi écrire *un seul* test à la fois ? C'est l'optique des petits pas : on valide un comportement par un test avant de passer au suivant. Plus le pas est grand, plus c'est long de rectifier le tir en cas de problème.

Pourquoi ne pas écrire tout le code fonctionnel d'un coup ? On risque d'introduire du code applicatif non testé ou inutile (voir aussi YAGNI). En TDD, on n'introduit un accesseur que pour les besoins du test, pas de manière systématique.

Pourquoi *exécuter* le test avant d'écrire le code ? Permet de détecter des étourderies type construire le test par copier-coller d'un autre test, et oublier de le modifier (alors barre verte) ; oublier dans la méthode de test le `@Test` / nom commençant par `test` (alors test pas exécuté et barre verte) ; oublier l'oracle (barre très verte).

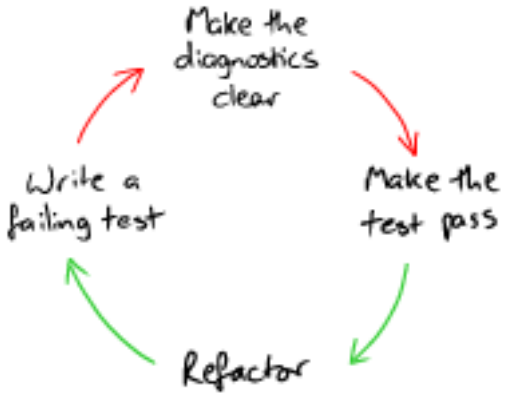
Et si j'écris le test juste après mon petit bout de code ? j'ai 0 en SVL ! « Juste après » est mieux que « tout à la fin » mais en SVL on pratique le test first. Pratiquer le pair-programming peut grandement aider à se discipliner au début.

Pourquoi réusiner ? Pour améliorer la qualité du code, le code affreux mais le plus simple possible n'étant autorisé que le temps du rouge / vert. On réusine aussi les tests. Si ça devient difficile d'écrire un nouveau test, il est temps de réusiner (Fig « listening to the tests », lire « hard to write a test ? » sur la flèche illisible).

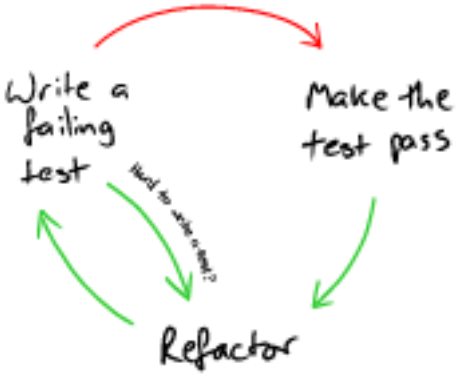
Le TDD, c'est du test ou pas ? Oui, mais c'est plus que ça : sa philosophie est autant « tester c'est construire » que « tester c'est douter ». On ne doit pas s'interdire d'ajouter d'autres tests qui passeront peut-être directement au vert, comme les tests aux bornes, les tests documentant un bug réparé, etc. La détection de bugs se fait aussi lors des tests de plus haut niveau.

C'est utilisé en entreprise ? Oui, ça devient tout à fait banal.

Simple TDD cycle



Feedback on test diagnostics



Listening to the tests