Avis d'expert, 2010

Test-Driven Development: un pacte diabolique ?

Introduction

Les Développements dirigés par les tests (TDD Test-Driven Development) sont récemment apparus comme le modèle type de développement de logiciels du modèle Agile. Dans sa forme la plus pure, le développement de logiciels selon les préceptes TDD consiste uniquement en:

- Un ensemble de « user stories» (histoire utilisateur),
- Un ensemble de tâches associées à chaque user story,
- Un ensemble de cas de tests pour chaque tâche, et du code source conçu, avec des variables et des unités bien nommées, qui ont été refactorisées.

Un Pacte Diabolique se réfère à la légende de Faust datant de l'époque médiévale et de la renaissance, dans laquelle un savant, (Faust) négocie un accord avec le diable (Méphistophélès) dans lequel il échange son âme pour une extension de ses pouvoirs terrestres. Dans la version de Christopher Marlowe, le Dr. Faust a des pouvoirs suprêmes pendant 24 années, et fini coupé en morceaux sur un tas de fumier. La légende a parfois une fin heureuse où le diable est battu, comme dans le classique de la musique country (américaine) appelé —The Devil in Georgia‖ (ou le duel au violon) où le diable parie son violon d'or qu'il joue mieux au violon que le héros, Johnny (Johnny gagne). Dans un usage contemporain, un pacte diabolique réfère habituellement sur un choix opportuniste plutôt qu'excellent, ou une préférence de récompenses immédiates par opposition à des bénéfices plus durables.

Le processus TDD idéal

Dans sa forme la plus pure, TDD commence avec un squelette de programme, soit une procédure en langage traditionnel, ou une méthode en langage objet. Le développeur en TDD conçoit un ensemble de cas de test pour la première tâche de la première «user story», et exécute ces cas de test sur ce code (vide). Bien sûr, tous les tests échouent. Ensuite, le développeur écrit juste assez de code pour que les cas de test passent tous. Ceci requiert de la discipline, car il y a une tendance naturelle à écrire plus de code que strictement nécessaire. L'ensemble des cas de test sont exécutés à nouveau sur le nouveau code. Si tous les tests passent, le code est considéré comme solide, et un nouvel ensemble de cas de tests est généré (pour la tâche suivante). Le code existant est exécuté avec les cas de test étendus, et, comme on s'y attend, échoue pour les nouveaux cas de tests. On ajoute ensuite juste assez de code de façon à passer avec succès les nouveaux cas de tests, et les cas de tests sont répétés. Si un cas de test échoue, la cause doit se trouver dans le code ajouté récemment. Le développeur TDD vérifie le code, en utilisant le(s) test(s) défaillant(s), corrige le défaut et réexécute la suite de cas de tests. Ce cycle continue jusqu'à ce que toutes les tâches de toutes les «user stories» soient testées et implémentées.

A divers points dans le processus TDD, le développeur peut s'arrêter afin de réécrire (refactoriser est le terme agile) le code existant dans une forme plus élégante. Ceci peut impliquer l'identification de nouveaux composants (procédures, méthodes), ou cela peut être une simple restructuration du code. Une longue série d'instructions Si-Alors (If-Then), par exemple, peut être refactorisée en une instruction Case/Switch, ou une logique de If-Then-Else imbriqués en profondeur peut être refactorisée en une logique composée de conditions complexes. Ce processus TDD complet est décrit dans la Figure 1.

Les Avantages du Processus TDD Idéal

L'avantage principal du processus TDD idéal est l'isolation des defaults qui est quasi garanti par les cycles courts Test-Code-Test. Quand les tâches correspondant à une user story sont petites, le nouveau code est alors créé en de petits incréments. Le test isole les defaults dans le nouveau code de façon quasi immédiate.

Un second avantage, tout aussi important, est que « il y a toujours quelque-chose qui fonctionne » (le mantra agile). Comme tout le processus TDD est dirigé par des «user stories» qui reflètent les priorités des utilisateurs, à la fin d'un cycle complet pour une —user story‖ (voir Figure 1), le développeur TDD peut livrer une fonctionnalité testée au client, dans l'ordre qui a été demandé. Pris ensemble, une isolation efficace des défauts et un logiciel qui fonctionne continuellement, donnent un niveau de confiance important à la fois au client et au développeur. Ceci doit être mis en opposition avec la relation parfois conflictuelle entre clients et développeurs dans un cycle de développement séquentiel (en cascade ou en V).

Le troisième avantage majeur revendiqué par ceux qui préconisent TDD est l'accroissement de productivité qui résulte de la non exécution des autres étapes d'un mode de développement non-agile : inspections techniques, conception générale et détaillée, niveaux de plans de tests détaillés, et documentation, contribuant ainsi à une productivité apparemment accrue. ("apparemment" annonce ici le pacte diabolique.)

Le quatrième avantage, indéniable, est que TDD est amusant. Les participants ont des succès rapides et fréquents, et sont rarement englués dans des aspects de taille et de complexité.

Questions sur le Processus TDD idéal

La communauté agile se réfère souvent aux rapports déprimants du GAO américain (US Government Account Office) et du groupe Gartner sur les faibles taux de succès des logiciels développés avec la méthode en Cascade. Ces rapports ont récemment été discrédités, donc les arguments de la communauté agile se résument à faire un épouvan-tail des projets basés sur la méthode de développement en cascade. La majorité des projets non-agiles actuels utilisent une sorte de cycle itératif.

1. Comment les projets TDD montent-ils en puissance pour des applications de grande taille? Pouvons-nous imaginer quelque-chose comme un système avionique embarqué, ou un commutateur téléphonique développé avec la méthode TDD ? La réponse de TDD à ceci est que la seule limite en taille est la séquence des « user stories » qui dirigent le processus. Bien sûr, il y a les inévitables limitations en terme de délais et de budget, mais, tant que le client ajoute des « user-stories », le processus continue. Un problème avec ceci est que la séquence des « user stories » est plate, il n'y a pas de notion de hiérarchie. Les approches non-agiles traitent la taille par l'utilisation de modèles, de niveaux d'abstraction, et une certaine notion de hiérarchie. Rien de tout cela n'est présent dans TDD.

Un autre problème est qu'il y a des limites à ce qu'un développeur agile peut « garder à l'esprit » à un moment donné.

2. Comment TDD traite-t-il la complexité applicative ?
Une manière de décrire la complexité qui s'applique à la fois aux projets TDD et aux projets non-agiles est de créer une matrice d'incidence qui relie les éléments comportementaux avec les éléments structurels : ceci pourrait être les caractéristiques par rapport aux procédures, ou les cas d'utilisation par rapport aux classes. Ceci peut clairement être effectué à la fois dans les projets non-agiles et les projets TDD. Les données introduites dans la matrice montrent quels éléments de structure sont requis pour supporter l'exécution de l'élément comportemental, càd, quelle classe supporte un cas d'utilisation donné. Dans ce cadre, que la matrice soit dense ou éparse est un indicateur utile de la complexité. Une matrice d'incidence dense veut dire que les composants structurels sont liés de multiples façons – indiquant ainsi que les interfaces seront un souci. Dans les projets TDD, cette matrice est généralement éparse, alors que dans des projets de grande taille et des projets complexes, elle est dense. Il n'y a pas grand-chose dans le processus TDD qui aide à traiter la complexité des interfaces. L'espoir principal est la refactorisation, et ceci devient à la fois de plus en plus fréquent et de plus en plus difficile au fur et à mesure de l'accroissement de la complexité.

3. Comment TDD traite-t-il la compréhension du système ?
Les seuls éléments d'un projet TDD qui supportent la compréhension du système sont des éléments de code bien nommés, des « user stories », et peut-être, une liaison entre les « user-stories » et le code. (Cette liaison n'est pas requise dans les projets purement TDD.) D'un autre côté, les projets non-agiles ont des modèles avec différents niveaux d'abstraction. Les éléments de conception [(documentation, architecture, etc.)], en plus d'un code correctement commenté, sont une aide à la compréhension d'un programme, et ni l'un ni l'autre ne sont présents dans des projets TDD. De plus, la compréhension du système devient de plus en plus difficile au fur et à mesure de l'accroissement de la taille du système.

4. Y a-t-il une quelconque garantie que les cas de tests développés dans le processus TDD constituent réellement un bon test ?
La raison principale pour laquelle nous testons est que tout le monde peut se tromper. Dans des projets non-agiles, il y a plusieurs niveaux d'inspections techniques et plusieurs niveaux de tests. Dans un projet TDD, les « user stories », suivies des cas de tests, conduisent l'effort de codage. Certes, le code passe les tests avec succès, mais il n'y a aucune vérification que les cas de tests TDD sont suffisants. On pourrait argumenter que les projets TDD utilisent des tests basés sur le code, mais des praticiens sérieux de TDD acceptant que pour être efficaces, ils ont besoin d'une compétence bien établie en théorie et en techniques de test. L'absence de toute modélisation suggère que TDD sera efficace au niveau unitaire [(tests de composants)], et ne testera que des cas d'utilisation (et des « user stories ») au niveau système. Il y a peu d'opportunités de tests d'interfaces (tests d'intégration), et aucune opportunité de tester en se basant sur les modèles [MBT Model Based Testing]. Des niveaux de tests supplémentaires, tels du test de stress et de sécurité, ne seront exécutés que s'ils sont ―requis‖ par les séquences de « user stories ».

5. Comment TDD traite-t-il les exigences non-fonctionnelles, telles que la performance, la fiabilité, la sécurité, la capacité de débit ou la bande passante, et la maintenabilité ?
Rien dans TDD ne permet de traiter ces questions. Les projets non-agiles ont traité avec succès de ces questions, principalement avec une modélisation rigoureuse et de l'analyse. Des niveaux de tests plus profonds tels du test de stress ou de sécurité sont uniquement exécutés s'ils sont « requis » par les séquences de « user stories ».

6. Vu que les cas de tests au niveau des tâches dirigent le processus TDD, que se passe-t-il si ces tests sont inadéquats, p.ex. incomplets, insuffisants ou incorrects ?
Les cas de tests TDD génèrent une sorte de vision tunnel pour le développeur TDD – Si les tests passent avec succès, le code est correct, et c'est tout. Il n'y a pas de vérifications croisées similaires à celles générées par l'utilisation responsable de tests basés sur les spécifications et sur le code.

7. Considérons les types de défauts qui ne peuvent être détectés que par le test du flux de données (technique basées sur les points dans un programme où les variables sont définies et les points où ces valeurs sont utilisées). L'hypothèse TDD est que le passage avec succès de tous les tests de toutes les tâches signifie que le code est correct. Dans une longue séquence de ―user stories‖ et de tâches, comment TDD garantit-il que les cas de test au niveau tâche identifieront les défauts du flux de données dans un code qui correspond à des « user stories » qui ne sont pas en séquence dans la demande de l'utilisateur?
Il n'y a rien dans le processus TDD qui traite explicitement les défauts des flux de données. Cette déficience s'accroit inévitablement en proportion avec la taille du projet.

8. Par rapport aux défauts des flux de données, considérons un projet TDD dans lequel plusieurs paires de programmeurs (un autre précepte agile) travaillent en parallèle, probablement sur des « user stories » séparées. Quel est le processus TDD qui garantit que les composants créés et testés isolément avec succès seront sujet à une forme de test d'intégration ?
Le seul espoir dans ce cas est un très bon test de niveau système. TDD n'a pas le concept traditionnel de test d'intégration autre que de réexécuter complètement tout le code existant. A nouveau, ce problème est proportionnel à la taille.

9. Comment la documentation délibérément parcellaire, supporte-t-elle le besoin éventuel de maintenance du programme ?
Ceci est un problème sévère. La seule réponse de la communauté TDD est que le code devrait être écrit clairement avec des conventions de nommage utiles.

10. Comment la refactorisation répétée, un processus exclusivement de bas en haut, produit-il une conception élégante ?
Ceci est une affirmation très controversée, supportée par bien peu de preuves. Une refactorisation répétée, si elle est faite efficacement, résultera en du code élégant, mais une conception élégante est un niveau d'abstraction au-dessus de celui du code source, et TDD évite de façon explicite une phase de conception séparée. Dans des expériences faites avec des étudiants à Grand Valley State University, une conception « élégante » n'a que très rarement été observée sur des projets de type TDD.

11. Dans des développements de logiciels traditionnels (non agiles), la phase de conception est le point où les composants du logiciels sont identifiés / créés, et ensuite améliorés par rapport aux considérations de performance (voir point 5 ci-dessus). Comment une « vision tunnel » générée par TDD peut-elle amener une conception qui traite efficacement des exigences de performances ?
Une fois de plus, rien dans TDD ne traite directement des aspects de performances. Une possibilité serait que ce soit pris en compte lors de la refactorisation.

12. Dans une longue séquence de « user stories », comment TDD peut-il détecter des inconsistances même légères entre les tâches d'une « user story », ou entre les « user stories » elles même ? Il n'y a rien dans les processus TDD qui ne traite la détection d'inconsistances subtiles.

Un pacte diabolique ?

Ces douze questions peuvent toutes être adressées par les approches de développement logiciel non-agiles. Certes, un processus de développement mature, discipliné est requis, mais cela peut se faire et cela se fait. Dans la mesure où TDD ne peut pas répondre à ces questions, et qu'il y a probablement d'autres questions aussi sévères qui ne sont pas traitées, TDD est un pacte diabolique. Il échange un temps de réponse court et élimine les efforts traditionnels pour l'assurance de meilleures réponses à ces douze questions.

Le professeur Paul C. Jorgensen, Ph.D. fait partie du comité d'écriture du syllabus ISTQB niveau Avancé et a été récemment élu vice-chair en charge du glossaire de l'ISTQB.

Paul C. Jorgensen, Ph.D., Professor School of Computing and Information Systems, Grand Valley State University, Allendale, MI 49341, USA—

Traduction: Bernard Homès