Gilles MANTEL

Valtech Technology

Test-Driven Requirements

octobre 07

Résumé

La gestion des exigences dirigée par les tests, ou Test-Driven Requirements (TDR), représente l'étape ultime dans l'adoption d'un processus de développement dit « lean ». L'article se propose de dresser un état de l'art du Test-Driven Requirements après avoir parcouru ses principes fondateurs que sont le Lean Software Development et le Test-Driven Development. Nous détaillerons différentes pratiques de TDR, telles que l'écriture de spécifications fonctionnelles testables avec des outils comme FIT, ou la génération de test basée sur l'interprétation de modèles comportementaux. La mise en œuvre de ces pratiques sera illustrée par des retours d'expérience. Les impacts sur l'organisation et la redistribution des rôles seront également mis en lumière.

Table des matières

Liste des Figures

1. La genèse

Qu'est-ce que le Test-Driven Requirements ? Cette appellation rassemble plusieurs pratiques permettant d'étendre les méthodes agiles au-delà de l'équipe de développement. Nous rencontrons souvent des équipes de maîtrise d'œuvre qui ont déployé les méthodes agiles pour concevoir et développer des applications mais se trouvent confrontées à des silos organisationnels dans les phases amont, recueil de besoin et spécifications fonctionnelles, et aval, homologation et recette, d'un projet. Ces silos sont souvent un frein atténuant tout le bénéfice d'un déploiement des méthodes agiles au sein d'un projet de développement logiciel.

Les pratiques du TDR trouvent leur source dans la transposition du « lean thinking » au monde du développement logiciel.

1.1 Lean thinking

L'utilisation du mot lean fait référence à l'organisation des chaînes de production d'industriels japonais qui ont révolutionné les méthodes de production dans les années 80. Le plus fameux d'entre eux est le constructeur automobile Toyota.

Le système de production de Toyota est souvent représenté sous la forme d'une maison, le toit représentant les valeurs vers lesquelles tend Toyota.

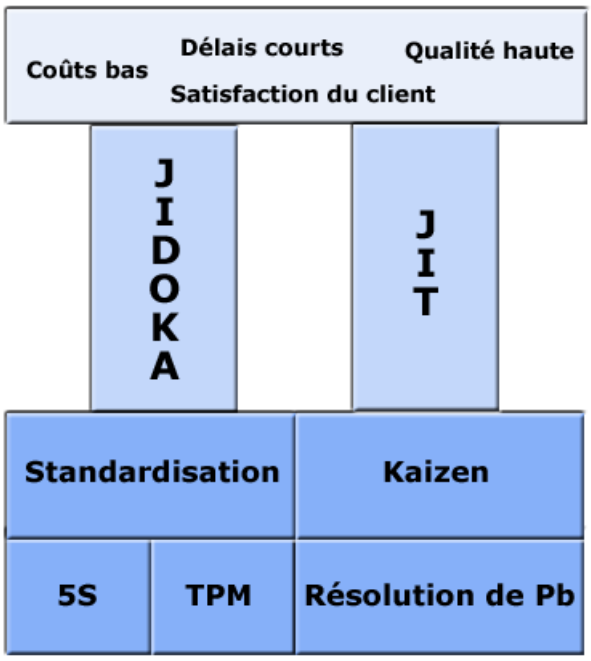


Figure 1: Système de production de Toyota

Ces valeurs sont supportées par les deux piliers suivants :

- Just-in-time : notion de produire la bonne pièce à la bonne quantité et au bon moment. Sous cette appellation, on retrouve un ensemble de méthodes tournées vers l'organisation de sa production selon un mode « pull », c'est-à-dire organiser sa chaîne de production de manière à ce qu'une étape ne produise pas un élément tant qu'il n'est pas demandé par l'étape suivante. Cela revient à piloter sa chaîne de production en fonction de son carnet de commande et plus en fonction d'une estimation de la demande et de la capacité de

production de sa chaîne. Cette organisation permet également de diminuer ses stocks au maximum, voire de les réduire à zéro.

- Jidoka : notion d'empêcher un maillon de la chaîne de passer une pièce défectueuse au maillon suivant. Si un défaut est identifié sur un élément dans la chaine de production, on arrête immédiatement la chaîne, on répare le défaut, et on s'assure qu'il ne se reproduira plus, on redémarre ensuite la chaîne de production. Les défauts peuvent provenir des machines, des matières premières, de la formation du personnel, etc.

Parmi les fondations, on trouve les concepts suivants :

- Kaizen : amélioration continue des processus, c'est-à-dire tendre vers la perfection en impliquant tous les acteurs depuis le top-management jusqu'à l'opérateur.
- 5S : la gestion de l'espace de travail, c'est-à-dire garder un espace de travail propre, rangé, efficace.

Je ne décrirai pas plus le système de production de Toyota dans cet article et j'invite les lecteurs désireux de creuser le sujet à lire l'ouvrage référence en la matière *The Machine That Changed the World : The Story of Lean Production*, de Womack, Jones, et Roos.

1.2 Lean sofware development

Les principes du lean thinking ont été transposés au monde du développement logiciel par les travaux de nombreux experts en développement informatique, les contributeurs les plus importants et les plus reconnus étant les époux Poppendieck (http://www.poppendieck.com/). Je retiendrai plus précisément les éléments suivants dans le cadre de mon article :

- Eliminer les gâchis : tout le lean thinking est orienté vers l'élimination des gâchis. Dans le monde du développement logiciel les gâchis sont de plusieurs sortes : fonctionnalités non nécessaires, stocks d'exigences en attente de développement, anomalies non détectées, tests de non-régression manuels, temps d'attente, multiplication des intermédiaires et des échanges formels entre ces intermédiaires, perte d'information, etc.
- Qualité intrinsèque : l'objectif est d'éliminer les défauts le plus tôt possible et de faire en sorte que les défauts ne puissent pas se produire.
- Livrer rapidement : réduire ses délais de développement pour rapprocher le plus possible le moment d'une demande d'évolution de sa livraison.
- Respecter les personnes : d'une part se concentrer sur les personnes qui apportent directement de la valeur pour le client, d'autre part accueillir les idées d'amélioration de chaque personne impliquée dans le processus et pas simplement du top-management ou d'un groupe d'experts.

1.3 Les étapes d'une évolution lean

L'organisation des projets informatiques il y a quelques années ressemblait à la figure ci-dessous : des activités d'ingénierie logiciel successivement réalisées, que ce soit au sein d'un cycle en cascade, ou au sein d'une itération.

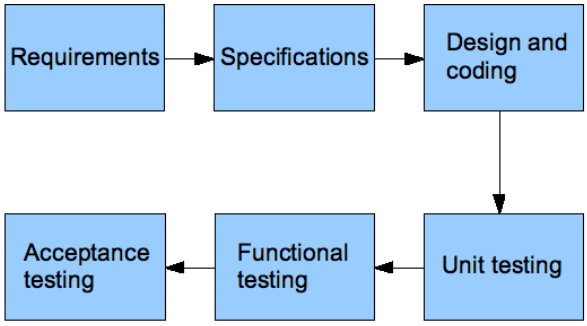


Figure 2: L'organisation classique des activités d'ingénierie logiciel

Cette organisation est caractéristique d'une approche prédictive du développement logiciel et donc à l'opposé d'une organisation « pull » préconisée par le lean-thinking.

L'émergence des pratiques agiles au sein des maîtrises d'œuvre ont permis aux développeurs d'organiser plus efficacement leur travail avec le Test-Driven Development, dit TDD, et d'améliorer fortement la qualité du code. Le principe du TDD est qu'aucun code n'est produit tant qu'un test unitaire n'a pas échoué. Cette approche illustre parfaitement la mise en application du lean thinking au monde du développement logiciel : l'étape de codage ne produit rien tant que l'étape de test unitaire ne le demande pas. Nous avons ici un fonctionnement parfait en mode « pull ». C'est l'écriture d'un nouveau test unitaire qui va déclencher l'écriture de nouveau code. Je reviendrai sur le TDD dans la suite de cet article. Ce qu'il faut retenir ici, c'est que la mise en œuvre du lean thinking revient à fusionner les étapes de codage et tests unitaires pour donner naissance à une étape de TDD. La figure ci-dessous illustre ainsi l'organisation des nombreux projets d'aujourd'hui ayant entrepris le déploiement de méthodes agiles.

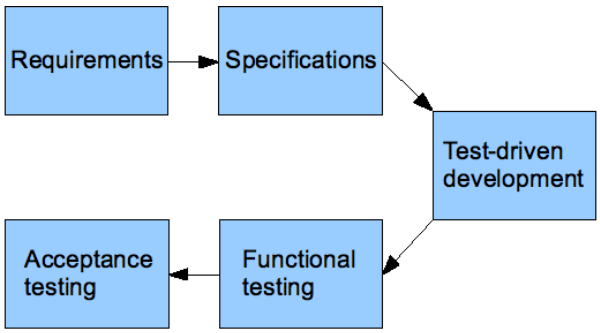
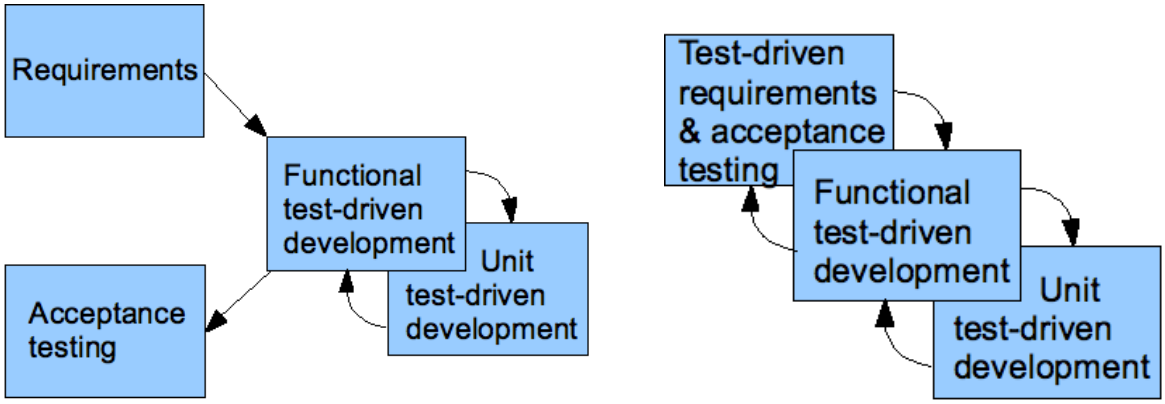


Figure 3: TDD ou la première étape vers un processus lean

Dans ce schéma, les étapes amont et aval restent encore cloisonnées et conduisent à gérer de nombreux stocks: des spécifications qui attendent d'être développées, des versions qui attendent d'être testées, des recetteurs qui attendent des livraisons, etc.

Une organisation qui recherche l'élimination des gâchis doit donc continuer le travail de lissage de son processus en intégrant les activités de la même manière que le TDD a permis aux tests unitaires et codage une approche lean.

Il est ainsi envisageable de fusionner les étapes de spécifications et tests fonctionnels avec ce que l'on appellera le functional test-driven development. De même, on peut fusionner les deux dernières activités avec ce que l'on appelle le test-driven requirements, comme l'illustre le schéma ci-dessous :



Source : Changing Roles, Dave Nicolette, March 2007

Figure 4: Les deux dernières étapes vers un processus lean

Le processus schématisé à droite illustre une organisation lean, dans laquelle le client « tire » les fonctionnalités de la première cellule, plutôt que de les pousser dans la chaîne de production et d'attendre qu'elles soient réalisées. Le TDR est donc l'aboutissement d'une démarche de changement de ses pratiques vers un processus lean.

Comment mettre en œuvre ces différentes étapes ?

2. Du TDD au TDR

2.1 Rappel sur le TDD

Le test-driven development est une pratique agile issue de l'Extreme Programming, dit XP.

Les trois étapes du TDD sont les suivantes :

1. Ecrire un test et s'assurer qu'il échoue, cela permet de garantir l'intérêt du test unitaire
2. Ecrire le minimum de code qui va faire réussir le test
3. Améliorer le code en s'assurant que le test continue de réussir

Revenir à l'étape 1 avec un nouveau test unitaire

Les tests unitaires, si importants soient-ils, présentent plusieurs limitations. Tout d'abord le test unitaire est… unitaire : son but est de vérifier la bonne adéquation du code à la conception. Ensuite, les jeux de données sont souvent peu représentatifs du métier ou du fonctionnel.

Il est cependant possible d'utiliser un outil de tests unitaires pour faire des tests plus proche du fonctionnel. Avec Junit par exemple, si l'on souhaite variabiliser un jeu de données, on peut envisager d'utiliser des données stockées dans un fichier XML. Cela demande généralement un investissement non négligeable et les tests restent peu accessibles à des personnes de la MOA ou à des analystes fonctionnels.

Il était donc important de franchir le palier permettant de faire des tests fonctionnels de la même manière que des tests unitaires dans TDD, mais de rendre ces tests abordables par la population concernée.

2.2 Appliquer les principes du TDD au niveau fonctionnel

FIT (http://fit.c2.com/) est un outil créé par Ward Cunningham, l'un des contributeurs du processus XP. L'idée initiale de l'auteur était de créer un moyen simple de tester les story-cards, équivalent des spécifications fonctionnelles dans le processus XP, et de pouvoir associer le client, concept fondamental dans XP, à la création de ces story-tests.

FIT permet ainsi de spécifier des tests à un niveau fonctionnel et de les exécuter automatiquement sans avoir besoin de déployer et d'exécuter l'application.

Le principe de fonctionnement est de décrire des tests sous une forme tabulaire, dans une page web standard, et d'accompagner ce tableau d'un bout de code, appelé fixture, qui fait le lien entre les tests et les API de l'application. Le moteur FIT s'occupe du reste. Il utilise la fixture pour lire le tableau et appeler les API de l'application avec les paramètres adéquats. Il retourne ensuite le résultat du test dans le même tableau en mettant en couleur les cases du tableau selon le résultat du test. FIT met à disposition de ses utilisateurs plusieurs formats de tableaux pour permettre la description d'un test fonctionnel :

- ColumnFixture : tableau permettant de spécifier des données en entrée et des résultats attendus.
- ActionFixture : tableau permettant de spécifier un enchainement d'actions. Ce format de tableau permet de décrire des scénarios de tests
- RowFixture : tableau permettant de spécifier des lignes de résultats. Ce format de tableau est généralement utilisé pour faire des initialisations ou pour vérifier des résultats de recherches.

FIT permet également de définir ses propres formats de tableaux pour les adapter à ses besoins. Ces formats personnalisés peuvent se baser sur l'un des trois autres formats standards.

L'utilisation de FIT nous permet donc d'effectuer des tests fonctionnels sans avoir à attendre une version du logiciel exécutable, ce qui représente un confort immense pour les équipes de développement. Le développeur est ainsi capable de lancer de tels tests autant de fois que nécessaire pendant qu'il développe et s'assurer que ses évolutions ou corrections n'introduisent pas de régression. Il peut aussi utiliser ces tests à la manière du TDD, c'est-à-dire les créer, ou demander à un analyste de les créer, avant d'écrire le code et les utiliser ensuite comme une mesure d'avancement du travail. On parle ainsi de FTDD, Functional Test-Driven Development.

A titre d'exemple, j'ai expérimenté FIT sur un projet au forfait dont le développement était effectué en Inde. Lors d'un voyage à Bangalore avec un collègue, nous avons décidé de mettre en œuvre FIT pour tester le codage d'une évolution sur une règle de gestion. Etant analyste sur ce projet, j'avais animé les réunions de recueil de besoin avec la MOA. J'ai écrit dans un tableau les nouveaux résultats attendus par rapport à différents jeux de données. Mon tableau était presque identique à la façon dont nous avions illustré nos discussions avec la MOA au travers de différents exemples. J'y ai également ajouté des jeux de données de non-régression, afin de m'assurer que l'évolution n'entrainait pas d'impact sur les parties inchangées. Mon collègue, architecte technique, a passé un peu de temps avec le développeur pour identifier les services à appeler pour récupérer les résultats décrits dans le tableau. Il a ensuite écrit la fixture qui allait permettre d'appeler ces services avec les jeux de données indiqués dans mon tableau. Nous avons ensuite lancé le test sur le code existant afin de nous assurer que les résultats retournés n'étaient pas conformes aux résultats attendus par le test sur les données concernant l'évolution. Nous sommes ensuite allés nous asseoir avec le développeur qui venait de terminer le codage de l'évolution. Nous avons lancé notre test, et moins de 2 minutes plus tard nous pouvions affirmer que l'évolution était correctement

développée. Cela sans avoir à exécuter le logiciel, sans avoir à exécuter des tests avec l'interface graphique, sans avoir à rechercher le bon contexte dans la base de données, etc. Nous avons consacré environ deux heures sur la mise en place de ce test en comptant l'apprentissage de FIT. L'investissement est donc dérisoire par rapport au résultat.

Non seulement FIT permet d'effectuer des tests quasi-fonctionnels à la façon de tests unitaires automatisés, rapidement et automatiquement, mais il permet aussi dans une certaine mesure de corriger plus rapidement le code. On peut en effet l'utiliser comme outil de diagnostic du code. Dans une activité de correction classique, lorsqu'un développeur recherche la cause d'une anomalie, son travail consiste souvent à lancer une instance locale de l'application sur sa machine, en mode « debug », après avoir placé des points d'arrêt aux endroits du code qu'il pense fautifs. Il déroule ensuite la manipulation du testeur, travail qui nécessite souvent des interactions avec ce dernier afin de clarifier son scenario ou de retrouver un jeu de données similaire dans l'environnement de développement. Après avoir atteint un point d'arrêt, le développeur va exécuter l'application pas-à-pas, tout en observant les valeurs des variables afin de détecter le point fautif. FIT peut être utilisé pour sonder les différentes parties du code par l'écriture de tests qui reflètent les manipulations d'un testeur, et zoomer progressivement, en écrivant des tests plus fins, sur les niveaux plus détaillés pour essayer d'identifier les fonctions fautives. Au-delà de l'économie de temps par rapport à un débogage classique, cela permet également de se doter d'une batterie de tests de non-régression à l'issue de la correction, et donc de s'assurer que l'anomalie ne réapparaîtra pas.

2.3 Des spécifications exécutables

Nous pensons chez Valtech que la pratique du FTDD n'est pas suffisante si l'on veut obtenir un processus lean, et que l'on peut pousser encore plus loin l'intégration des activités de spécifications et des tests.

FIT permet en effet d'ignorer le texte d'une page web et de n'utiliser que les tableaux marqués comme étant des tests. Il est donc possible de créer des pages web qui contiennent du texte libre, par exemple des règles de gestion, et contiennent des tableaux de tests, par exemple les tableaux qui vérifient ces règles. Il est donc possible de rassembler au sein d'un même « livrable » les exigences et les tests qui les vérifient. Nous atteignons ici la dernière étape de mise en place d'un processus lean. Dans cette organisation les différentes étapes, dont la production du code, sont conduites par la demande de la dernière étape de la façon suivante :

- Test-Driven Requirements : client et fournisseur travaillent sur un même livrable, c'est-à-dire l'ensemble des exigences et les tests qui les valident. L'écriture des tests sous format tabulaire permet de valider la compréhension des uns et des autres. C'est pour cela qu'à cette étape nous disons que les exigences sont dirigées par les tests. On commence par écrire les tests s'inspirant des exemples que fournit la MOA, et on écrit ensuite les règles de gestion ou cas d'utilisation correspondants. L'écriture de tests est utilisée comme un outil de recueil du besoin.

- FTDD : comme vu précédemment, cette étape consiste à écrire les fixtures qui permettent de mettre en œuvre les tests écrits à l'étape précédente. Pour ce faire, les développeurs se basent sur des API existantes ou non. Si les services n'existent pas encore, un travail de conception permettra de les définir et d'écrire les fixtures qui ont besoin de ces API. Cette étape permet aussi aux développeurs d'enrichir les tests existants lorsqu'ils ne sont pas assez détaillés. A cette étape, les spécifications deviennent exécutables. L'intégration de ces tests dans l'usine logicielle permettra de faire apparaître de nouveaux tests fonctionnels en échec et donc de déclencher le travail de l'étape suivante.

- TDD : à cette étape, les développeurs conçoivent les différentes classes et méthodes implémentant le service en définissant d'abord les tests unitaires permettant de vérifier leur fonctionnement. La validation progressive des tests unitaires permet de mesurer l'avancement du travail. Au fur et à mesure de l'implémentation des classes bas-niveau, les tests fonctionnels vont progressivement réussir. Lorsque l'ensemble des tests réussit, le produit est prêt à être livré !

Dans cette organisation, le codage devient en pratique la dernière activité d'ingénierie logiciel avant la livraison. Il faut quand même compter quelques ajustements au cours de ces différentes étapes, car le codage peut mettre en lumière des faiblesses dans les exigences ou dans la conception. Ces faiblesses sont adressées en réajustant les tests fonctionnels et unitaires.

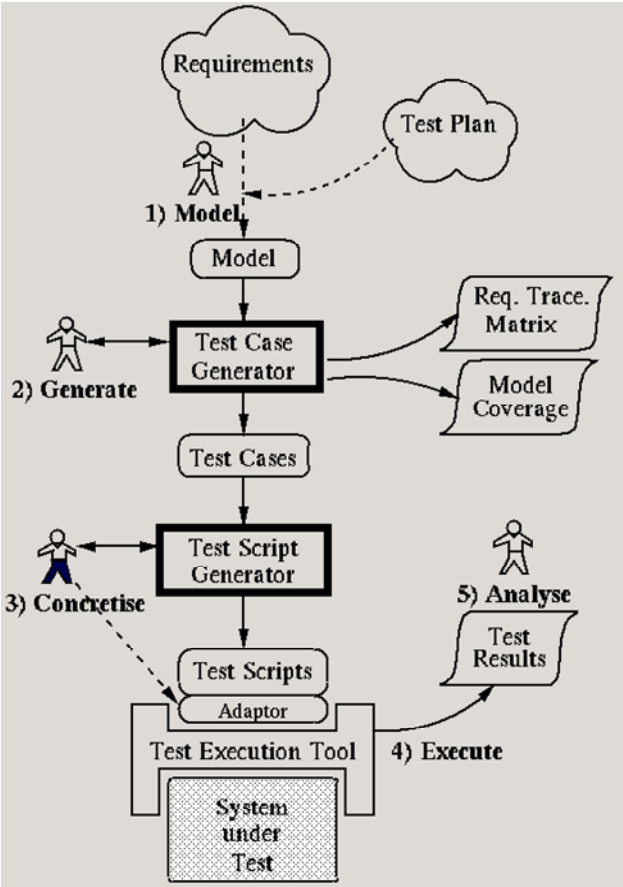
Une telle organisation ne peut être supportée qu'avec les outils adéquats. FIT se révèle trop limité pour répondre totalement à ce besoin, notamment sur l'aspect TDR. Il faut disposer d'outils permettant une meilleure collaboration des différents acteurs du processus, et une structuration plus contextuelle de l'information. La forme du wiki paraît idéale pour répondre à cet objectif en permettant une structuration et une contextualisation efficace de l'information, ainsi qu'une facilité de mise à jour. Chez Valtech nous utilisons déjà le wiki sur certains projets pour écrire les spécifications fonctionnelles basées sur les cas d'utilisation. Ce support est infiniment plus confortable pour réaliser les activités de recueil et spécifications que les classiques documents Word. Il permet également une grande souplesse dans la gestion de configuration car le format wiki est simple à manipuler et l'on peut fusionner facilement deux versions d'une même page.

Deux outils permettent aujourd'hui de combiner wiki et moteur de tests : Fitnesse et Greenpepper. Fitnesse (http://fitnesse.org/) est un outil libre qui utilise une version modifiée de FIT comme moteur de tests et permet de décrire ses tests dans un wiki. Il nécessite par contre de déployer le code source sur un serveur pour pouvoir exécuter ses tests. C'est un bon outil pour l'étape TDR et tests de recette, mais n'est pas suffisant pour supporter une approche compréhensive du TDR jusqu'au code.

Greenpepper (http://www.greenpeppersoftware.com/) est un outil plus puissant combinant un moteur de tests open-source et une série d'extensions payantes. Les extensions permettent l'utilisation du wiki Confluence et l'intégration à une usine logiciel par des scripts Maven. Une extension permet également d'intégrer JIRA, un moteur de workflow. Les tests peuvent être exécutés indépendamment depuis un serveur ou depuis le poste du développeur. Il est également possible de choisir différents contextes d'exécution. On peut ainsi gérer plusieurs environnements de tests selon les phases de tests à effectuer (développement, homologation, recette, pré-production) ou selon la version des spécifications. Chez l'un des clients pour qui je travaille, j'ai rencontré une équipe ayant mis en œuvre Greenpepper. L'équipe est répartie sur deux sites avec des développements effectués à Paris, qui compte 6 développeurs, et à Bangalore, qui compte une dizaine de développeurs. Leur délai moyen entre la fin de l'écriture du code pour une livraison donnée et la mise en production cette nouvelle version (j'écris bien la mise en production) est de **trois heures**. Ce délai inclut les phases de tests d'homologation et de pré-production (tests d'intégration avec les systèmes environnants). La seule activité manuelle effectuée après l'écriture de la dernière ligne de code consiste à vérifier le bon fonctionnement de l'interface graphique, car cette couche n'est pas encore testable avec des outils comme Greenpepper ou FIT.

2.4 Eliminer les gâchis avec le MBT

Le Model-Based Testing propose une alternative aux outils comme FIT et ses dérivés. Le principe est d'élaborer un modèle à partir duquel on dérive automatiquement les actifs nécessaires aux activités de tests. Par exemple, on peut générer des cahiers de recette, ou des scripts de tests.



Source: Generations of test automation, Mark Utting

Figure 5: L'approche MBT

De la précision du modèle dépend la profondeur des tests que l'on peut générer. L'approche impose donc un fort investissement dans les activités de modélisation. Elle a du sens pour les organisations déjà matures sur cet aspect pour qui l'apprentissage sera modéré.

Un outil comme Leirios Test Designer (http://www.leirios.com/) permet d'utiliser un modèle comportemental formalisé avec un sous-ensemble d'UML, appelé modèle de test, pour générer les actifs de tests. Ce modèle est parcouru par un moteur calculant le nombre de tests et les vérifications à effectuer pour en déduire les cas de tests. Ces cas de test peuvent ensuite être traduits en scripts de tests avec un adaptateur adéquat.

3. Quel avenir pour le TDR ?

3.1 Usine logiciel

A court terme, il est indispensable de se doter d'une usine logiciel permettant l'exécution en permanence des tests unitaires et fonctionnels si l'on souhaite réussir sa mise en œuvre du TDR. Ce point participe à la mise en pratique de l'aspect « jidoka » du lean-thinking. En effet, l'exécution en permanence des tests fonctionnels, ainsi promu au rang de tests de non-régression, pourra indiquer si une anomalie s'est glissée lors de la production de nouveau code, ou la modification du code existant.

Au-delà de l'objectif de non-régression, l'usine logiciel agira comme un séquenceur de toutes les activités d'ingénierie logiciel pour les évolutions d'une application. Ainsi l'intégration de nouvelles spécifications associées à leurs tests dans le référentiel projet mettra en lumière le besoin de développer les fixtures associées. Puis, l'intégration des fixtures montrera l'échec des tests associés et déclenchera ainsi le travail de TDD pour concevoir et développer le code nécessaire à la réussite de ces tests.

3.2 Mutation organisationnelle

La mise en place du TDR implique également d'entreprendre une mutation organisationnelle afin de faire sauter les séparations entre les activités amont et aval des équipes de développement. Si une organisation n'entreprend pas ce travail, les spécifications et tests « à la mode TDR » seront une charge supplémentaire pour toute l'équipe. Non seulement cette organisation n'aura pas éliminé les gâchis générés par l'exécution classique de ses activités d'ingénierie, mais elle en ajoutera si elle investit sur le TDR.

J'ai ainsi été confronté à un échec de la mise en place de FIT sur un projet. Mes interlocuteurs étaient pourtant enthousiastes lors de notre intervention avec un collègue. Nous avions mis en place des tests FIT sur un aspect du système assez épineux et souvent sujet à des régressions. Ces tests ont permis la compréhension et la vérification d'évolutions portant sur des règles de gestion. Après notre départ, aucun nouveau test n'a été écrit et l'exécution des tests existants n'a pas dépassée la fin de la semaine. Lorsque nous avons cherché les raisons pour lesquelles cette initiative ne s'était pas propagée, on nous a répondu quelque chose comme « c'était une très bonne idée, mais nous n'avons vraiment pas le temps de continuer ». Cette réponse est symptomatique d'une organisation qui est restée inchangée. Le but n'est pas de consacrer du temps supplémentaire pour faire du TDR, le but est de modifier sa façon de faire les choses. Notons que dans cet exemple, nous n'avions pas pu intégrer les tests FIT à l'usine logiciel avant de sortir du projet, ce qui vient également renforcer le point précédent.

3.3 Mutation professionnelle

A long terme, la pratique du TDR entrainera la disparition de certains rôles caractéristiques des silos organisationnels pour faire place à de nouveaux profils. Par exemple, les rôles d'analyste et de testeur sont amenés à fusionner car il n'existe plus de besoin d'avoir des ressources spécialisées sur les exigences et sur les tests, ces activités étant elles-mêmes fusionnées. Ce point est adressé en détail dans un article de Dave Nicolette sur l'évolution des rôles dans une organisation lean (http://www.davenicolette.net/articles/changing-roles.html).