**LIVRE BLANC** | MAI 2016

# Utilisation des tests basés sur des modèles pour stimuler le développement basé sur le comportement

Huw Price
CA Technologies

ca technologies

# Table des matières

**Section 1**

# Clarté et collaboration : deux principes moteurs

Avec le rapprochement qui s'opère entre les initiatives métier et IT, il devient indispensable de pouvoir communiquer les exigences métier selon des modalités directement transférables aux projets techniques. De plus en plus, les entreprises modernes s'appuient sur des logiciels pour apporter de la valeur à leurs clients. Les équipes IT se voient ainsi obligées de fournir plus rapidement et à moindre coût, des logiciels entièrement testés qui s'adaptent à l'évolution des besoins métier.

Toutefois, toute ambiguïté au niveau des exigences logicielles accroît le risque de confusion pour les équipes IT et de décalage par rapport à l'idée que se font les utilisateurs et les analystes métier du logiciel, ce qui réduit d'autant la probabilité pour ce dernier d'apporter de la valeur à l'entreprise.

Cette frustration ressentie par les entreprises qui développent des logiciels parfaitement conformes à une certaine interprétation des exigences pour finalement constater que l'intention initiale n'y est pas a conduit Dan North à vanter les mérites du « développement guidé par les comportements ». Les confusions et malentendus dont il a été maintes fois victime l'ont amené à comparer le développement à un « couloir sombre » : vous avancez à l'aveugle alors qu'il suffirait de quelques indications pour que tout aille bien.[1]

Au vu des nombreux défis qui jalonnent les projets informatiques classiques, cette frustration est de toute évidence très fréquente parmi les développeurs et les testeurs. En moyenne, seuls 4 % des projets démarrent avec de bonnes instructions quant aux exigences[2] ; 56 % des défauts sont dus à une mauvaise définition des besoins[3]. Ces défauts sont généralement repérés à un stade avancé du développement, en phase de production où la résolution est 50 fois plus longue que s'ils étaient repérés au moment de définir les exigences[4]. Les remaniements qu'implique une définition ambiguë des besoins sont responsables de 60 % des échecs ou des retards de projets logiciels ; 82 % des efforts déployés pour corriger les bogues sont liés à la mauvaise définition des exigences[5] et coûtent chaque année la somme exorbitante de 250 milliards de dollars[6]. Dan North avait donc raison en affirmant qu'il est tout à fait possible de présenter le développement axé sur les tests (*Test-Driven Development*, TDD) de telle manière à aboutir directement au bon résultat et à éviter les pièges liés à une mauvaise communication.

Le développement guidé par les comportements (*Behavior-Driven Development*, BDD) cherche à favoriser la collaboration entre les intervenants métier et IT, en s'inspirant de la notion de « langage universel » qui s'applique au développement axé sur les domaines. Étant donné que les équipes IT doivent répondre rapidement à l'évolution des exigences métier, ce concept est plutôt séduisant : un langage commun pouvant être compris par les parties prenantes principales et secondaires (celles qui se concentrent sur les objectifs métier et le comportement des applications, et celles qui mettent en œuvre les résultats et comportements escomptés), et qui de par sa quasi-formalité peut être facilement traduit en logique système à développer et tester.

La question qui se pose alors est la suivante : comment formuler précisément les « exigences axées sur les comportements » pour qu'elles soient correctement mises en œuvre au cours du développement ? Voilà précisément le thème de ce livre blanc. Nous allons examiner la manière dont ces principes clés de collaboration et de clarté entre les acteurs métier et IT peuvent être appliqués au mieux au cours du cycle de développement.

Au vu du processus de collecte des exigences couramment encouragé, nous allons expliquer pourquoi l'introduction d'une modélisation formelle peut être synergique pour le BDD, à condition de disposer des bons outils. Nous allons remettre en question l'idée selon laquelle une documentation exhaustive (par opposition à une « documentation minimaliste ») entraîne forcément une incapacité à réagir rapidement au changement. Une modélisation formelle permet en effet d'appréhender pleinement les exigences, ce qui est nécessaire pour optimiser la probabilité que le logiciel réponde en tout point au comportement souhaité des principales parties prenantes.

**Section 2**

## Incomplétude

Si nous nous en référons aux principes du BDD précédemment évoqués, il ne fait aucun doute que les exigences doivent être compréhensibles à la fois pour le personnel métier et IT, et doivent pouvoir être mises en œuvre en tant qu'exigences techniques, qui serviront aux testeurs et aux développeurs. La capture des exigences à travers l'interaction de ces parties prenantes stratégiques qui, au final, bénéficieront directement des logiciels, en soumettant des cas de figure réels ou des exemples concrets du comportement souhaité, apparaît en théorie comme un moyen sûr de garantir que ces exigences reflètent la manière dont le métier envisage le système.

Toutefois, malgré la bonne intention de cette méthode de développement « inspirée de l'extérieur » qui cherche à lever toute ambiguïté, la nature des exigences et la manière dont elles sont obtenues soulèvent des questions sur leur mise en œuvre par les systèmes de test et de développement.

Comme évoqué par Llyr Wyn Jones dans « *A Critique of Testing* », l'implémentation des exigences peut être assimilée à un processus de transformations d'informations. Les informations entrent sous une forme et ressortent sous une autre. C'est ce qui arrive, par exemple, au moment du codage, lorsque le développeur transforme les exigences en élément logiciel. L'ensemble du cycle de développement d'un logiciel (*Software Development Lifecycle*, SDLC) peut être comparé à un flux d'informations : dans un premier temps, l'utilisateur travaille avec un analyste métier pour définir des exigences et des cas d'utilisation ; ensuite, le développeur rédige le code, puis les testeurs écrivent manuellement les plans et les scripts de test. À chaque étape, les informations sont transmises et converties dans un format différent.

Cette analogie permet à Llyr Wyn Jones de définir l'ambiguïté par référence au concept d'« incertitude ». Selon lui, cela donne lieu à des résultats différents, car l'incertitude concernant l'interprétation des instructions permet de mettre en œuvre l'exigence de plusieurs façons. De la même manière, une incomplétude apparaît lorsque des informations nécessaires concernant un système restent à l'état d'hypothèses.[7] Dans les deux cas, cela aboutit à une « incertitude » et introduit un risque de malentendu entre les parties prenantes principales et secondaires.

En résumé, l'incomplétude, tout comme l'ambiguïté, réduit les probabilités que le logiciel se comporte comme l'espèrent les principales parties prenantes. L'incomplétude augmente la probabilité de voir apparaître en phase de développement le type de malentendus que Dan North a tenté de résoudre, et s'oppose de façon évidente aux principes qui ont conduit à la formulation du BDD.

---

**Section 3**

## Incomplétude dans les exigences axées sur le comportement

Dans le cadre du BDD, le flux d'informations décrit par Llyr Wyn Jones se produit dans la « boucle de retour », là où les exigences sont formulées, les logiciels sont développés et testés et où les rapports sont retransmis aux parties prenantes principales afin d'être évalués. Dans ce contexte, l'incertitude réside dans le fait que la « boucle » sera répétée indéfiniment, car le logiciel livré ne répond jamais exactement aux attentes et les exigences évoluent constamment.

En calquant les exigences et les scénarios de test sur les objectifs métier et le comportement souhaité, la phase de test met l'accent sur ce qui doit se passer. En d'autres termes, elle se concentre sur les scénarios optimistes. Cette vision se retrouve dans les langages naturels utilisés pour écrire les scénarios.

Si nous envisageons une logique système en termes booléens, celle-ci est réduite à trois fonctions : NON, OU et ET, XOR (soit le OU exclusif) étant une combinaison des trois. Ces fonctions peuvent ensuite être présentées à l'aide des arguments SI... ALORS.

Le langage Gherkin, par exemple, met davantage l'accent sur les arguments SI, ET et ALORS. Il décrit les conditions initiales et clauses de départ d'un scénario, ainsi que les « déclencheurs » présents (l'argument SI) et le résultat attendu du scénario (la partie ALORS). La partie ET vient du fait que chaque étape peut avoir plusieurs éléments énoncés ou déclencheurs. La partie OU est apparemment absente lors de la collecte de ces scénarios de test dans ce format, ce qui ressort davantage dans la version exprimée en langage naturel utilisée par Rspec, qui peut formater des scénarios sous forme d'une série d'arguments « Quand ... Alors »[8]. En d'autres termes, le langage naturel ne tient pas compte de ce qui se passe en l'absence de déclencheurs.

Pourtant, les décisions qui découlent de cette proposition « OU » sont indispensables à une logique système, et chacune représente un chemin distinct possible, que le système devra prendre en considération lors de la phase de test. Cela est d'autant plus vrai pour les chemins négatifs, qui doivent représenter près de 80 % de l'effort de test total. Les résultats inattendus se produisent précisément lorsque les déclencheurs sont absents. La partie « SI » n'est donc pas remplie, alors que ce sont justement ces résultats inattendus et aberrations qui sont les plus susceptibles de provoquer l'effondrement d'un système.

Certaines personnes avancent que le BDD est un bon moyen d'identifier les chemins oubliés lors de la conception du programme, qui entraînent des résultats inattendus si et quand ils se produisent. Après tout, le BDD est un processus itératif. En d'autres termes, le raccourcissement de la « boucle de retour » signifie que ces chemins négatifs peuvent être résolus plus tôt, ce qui permettrait d'avancer la phase de test dans le cycle. Certains suggèrent même que les entreprises doivent « accepter cette incertitude »[9].

Toutefois, comme nous l'avons indiqué, si les équipes IT attendent la phase de test pour détecter les défauts, elles devront alors faire face à un coût et une perte de temps supplémentaires, et en peu de temps, les itérations se transformeront en mini procédures en cascade et les équipes n'auront d'autre choix que de repousser la phase de test ou de tirer un trait dessus. C'est la principale difficulté de l'observabilité des défauts. Le résultat attendu (Alors) d'un scénario devrait être une condition vérifiable, et l'objectif supposé de la phase de test est de vérifier d'une part, que le résultat attendu s'est produit et d'autre part, qu'il résulte des bons déclencheurs et de la clause de départ appropriée. Lorsque les tests sont effectués au coup par coup, il n'y a aucun moyen de savoir avec certitude si les résultats obtenus se sont produits pour les bonnes raisons et non parce que deux défauts ou plus s'annulent mutuellement.

---

**Section 4**

## Tests basés sur des modèles et BDD

Pour avancer vers l'exhaustivité et renforcer la probabilité que les logiciels soient conformes aux attentes, le BDD doit introduire la notion du « OU » en plus de celle du « ET ». Les divers scénarios de BDD présentés et les langages utilisés tels que Gherkin doivent ensuite être connectés afin de refléter la logique d'un système. Dans ce cas, les tests de BDD examinent également ce qui se produit si les déclencheurs n'interviennent pas au moment prévu et orientent le système vers un chemin négatif.

Lors de l'écriture et de l'exécution des tests, les testeurs constituent généralement des modèles, même s'ils sont implicites. Pour un test négatif par exemple, ils peuvent examiner ce qui se produit si un déclencheur n'arrive pas, mais que les clauses de départ sont vraies. De même, en BDD, si deux scénarios de test ont une étape en commun, ils sont implicitement connectés par « OU ». Ils peuvent, par exemple, partager un déclencheur mais pas l'autre, ce qui implique une décision dans la logique système.

Toutefois, lorsque cette modélisation est réalisée de manière ponctuelle, les tests sont en mesure de couvrir uniquement les scénarios auxquels les testeurs et les développeurs ont pensé. C'est le cas en BDD lorsque les récits d'utilisateurs sont présentés sous forme d'unités linéaires distinctes, et que leur relation dans la logique système n'apparaît pas. Dans ce cas, la couverture fonctionnelle des tests est généralement faible, de l'ordre de 10 à 20 %, car même un système basique est susceptible d'avoir des milliers de combinaisons d'entrées et sorties, autrement dit, bien trop pour permettre à une personne de les retenir et de les relier avec précision.

Si le BDD nécessite de toute façon une modélisation de la part des testeurs, il n'y aucune raison pour que cette modélisation ne puisse pas être assurée de façon systématique, et fournir une documentation claire et surtout exhaustive sur les exigences. Ce systématisme est indispensable si le test doit couvrir tous les chemins possibles d'un système, y compris les chemins négatifs et les résultats inattendus.

**Section 5**

## Modélisation d'organigramme dans le cadre du BDD

Vous trouverez ci-dessous une proposition pour incorporer la modélisation au développement de manière synergétique par rapport aux principes évoqués du BDD.

1. Spécification par l'exemple. Il est intéressant de noter que les exigences peuvent rester régies par le comportement et dériver de l'interaction des parties prenantes quant à l'apparence qu'elles souhaiteraient donner au système. Soit les exigences formulées par les acteurs métier peuvent être retranscrites dans un organigramme, soit les principales parties prenantes élaborent elles-mêmes un flux. Cette dernière solution est possible car un organigramme fait office de langage universel. Philip Howard de l'institut Bloor Research a décrit un modèle d'organigramme comme un élément sur lequel peut reposer toute la logique fonctionnelle d'un système requise par les testeurs et les développeurs, sans laisser les équipes métier dans l'incompréhension[10].

   Si testeurs et développeurs utilisent des exigences BDD existantes, il convient alors de connecter les étapes des scénarios qui se recoupent et qui aboutissent à des décisions (OU) dans la logique système. Cela oblige les modélisateurs à raisonner en termes de logique système et à appliquer le principe de complétude, tandis que CA Agile Requirements Designer (anciennement Grid Tools' Agile Designer) identifie tout chemin interrompu et fournit des suggestions de chemin pour tout scénario susceptible d'avoir été oublié. Les chemins oubliés ou négatifs sont alors identifiés avant que la boucle de retour soit terminée, ce qui réduit le nombre d'itérations requises.

2. Modélisation de scénarios. Dans les faits, les récits d'utilisateurs sont trop élaborés pour les phases de test et développement. Les scénarios à développer et tester sont plutôt modélisés dans un organigramme, dans lequel chaque étape de scénario forme un bloc de processus et de décision et où les scénarios deviennent les chemins d'une logique système. Si tous les scénarios sont modélisés, chaque récit d'utilisateur est potentiellement couvert, car les scénarios découlent de ces récits.

   Tous les aspects du récit sont transposables dans un organigramme. Par exemple, un bloc de processus peut spécifier la partie prenante à l'origine de l'action, en décrivant la manière dont « le client tente de retirer de l'argent d'un distributeur automatique de billets ». Les chemins d'un organigramme peuvent ensuite refléter un test unitaire dans le BDD, en spécifiant un rôle, une fonctionnalité réclamée et un avantage ou résultat attendu.

   Les méthodes combinatoires font qu'il n'existe pas un chemin unique du début à la fin pour chaque scénario. À la place, les composants de la logique fonctionnelle de scénarios ou de fonctionnalités multiples sont combinés dans certaines parties de l'organigramme. Cela crée une différence majeure entre des tests dérivés d'exigences BDD recueillies en langage naturel comme le langage Gherkin d'une part, et la modélisation de la logique fonctionnelle clé d'un système

d'autre part : dans une modélisation, un seul plan de test peut couvrir plusieurs scénarios, ces derniers étant consolidés au sein d'un système, plutôt que d'être présentés comme des unités distinctes. Cette consolidation des récits d'utilisateur en tant que système à part entière est une étape clé dans la réduction de l'incertitude.

Par exemple, plusieurs scénarios peuvent être consolidés jusqu'à un certain point énoncé dès lors qu'un système a été modélisé sous forme d'organigramme. Le « point » en question peut être un organigramme secondaire, ou une partie de chemins multiples, qui sont ensuite définis par un bloc décisionnel.

De la même manière, plusieurs « Quand » peuvent jalonner un chemin donné, sous forme de blocs décisionnels ou de blocs de processus. Plusieurs scénarios sont alors consolidés, de sorte que si deux scénarios ou plus ont des déclencheurs en commun, ils peuvent transiter par les mêmes blocs. Chaque décision rencontrée peut conduire vers un autre chemin, ou ensemble de chemins, jusqu'à ce qu'un système soit entièrement modélisé, chemins négatifs compris. Par exemple, si deux scénarios ont un seul déclencheur de différence, ils peuvent suivre le même chemin jusqu'au tout dernier bloc décisionnel, où leurs chemins se séparent.

3. La boucle de retour : vérification du modèle. Si un système a été modélisé par des utilisateurs et des analystes métier, la vérification du modèle est alors facultative et le BDD peut passer à l'étape suivante. Dans ce cas, le délai de retour sera très bref.

   Si un système a été modélisé par des testeurs et des développeurs, la vérification du modèle est tout aussi simple et rapide. Dans CA Agile Requirements Designer, cette vérification peut s'effectuer au moyen de scénarios d'utilisation. Chaque chemin de l'organigramme représente un cas d'utilisation distinct, l'équivalent d'un plan de test ou scénario dans un système modélisé. La vérification consiste à passer en revue un ensemble de scénarios, formulés dans un vocabulaire d'usage courant et sous forme d'organigramme, pour s'assurer que le système a été conçu comme prévu.

4. La boucle de retour : validation du modèle. L'organigramme étant soutenu par la logique fonctionnelle d'un système, les plans de test peuvent en être déduits automatiquement. Comme évoqué précédemment, chaque chemin possible d'un système fait l'objet d'un plan de test. CA Agile Requirements Designer est capable d'identifier automatiquement les chemins possibles et de calculer précisément la couverture fonctionnelle.

   Par ailleurs, les tests redondants ou non pertinents peuvent être supprimés et d'autres peuvent être divisés afin de réduire au minimum le nombre de plans de test requis pour offrir une couverture fonctionnelle la plus large possible. Comme indiqué plus haut, la logique de plusieurs scénarios pouvant être consolidée, il est possible de tester des scénarios avec un nombre réduit de plans de test, par exemple 3 plans de test peuvent suffire à tester 15 scénarios. CA Test Case Optimizer contient plusieurs algorithmes qui permettent d'identifier les chemins offrant la couverture fonctionnelle la plus étendue. Dans un cas, par exemple, CA Agile Requirements Designer a ramené le nombre de tests à 17 contre 326 précédemment, pour une couverture de 100 %.

   Une fois ces chemins stockés, ils peuvent être exportés et exécutés. Cette opération peut être effectuée manuellement, en envoyant les plans de test reliés à des données de test et à des résultats attendus, vers un outil de gestion des tests tel que HP ALM/QC. Les chemins peuvent également être exportés en tant que scripts de test exécutables par de nombreux moteurs d'automatisation. Dans les deux cas, la « documentation dynamique » et les rapports sont générés par l'outil de gestion des tests utilisé.

Ce processus automatisé peut rappeler celui de Cucumber, à la différence que le délai de retour est bien plus court et le nombre de boucles de retour requises bien moindre. Tout d'abord, la génération automatique de plans de test supprime la nécessité de les définir manuellement ; dans le cas de Cucumber, ce processus élimine le besoin de convertir manuellement le langage Gherkin en définitions d'étape dans Ruby. Par exemple, en 90 minutes CA Agile Requirements Designer a créé pour un logiciel donné 108 plans de test offrant une couverture fonctionnelle totale, y compris l'organigramme. En pratique, le gain de temps ne s'arrête pas là, puisque l'organigramme peut être facilement adapté et réutilisé.

Par ailleurs, étant donné que les développeurs disposent désormais d'exigences vérifiées, complètes et claires, il est plus probable qu'ils produisent le logiciel prévu du premier coup. Comme nous l'avons indiqué, l'ambiguïté est responsable de 56 % des défauts, et à cet égard CA Agile Requirements Designer réduit l'introduction de défauts de 95 %. Tout défaut qui échapperait aux développeurs serait repéré en phase de test dès la première fois, grâce à la couverture fonctionnelle de 100 % fournie par les plans de test. Outre le gain de temps grâce aux remaniements évités, les développeurs peuvent également gagner du temps sur l'exécution des tests. La reproduction de tests permet généralement de réduire la phase de test de 30 %, tandis que l'optimisation de tests peut réduire nettement le nombre total de tests requis pour couvrir chaque scénario.

**Section 6**

## Adaptation rapide et facile aux changements

Une objection pourrait être formulée selon laquelle la modélisation d'organigramme est fastidieuse dans un environnement BDD, et que ce temps pourrait être mis à profit plus utilement en répétant les tests.

Tout d'abord, il convient de souligner que cette critique n'est valable que si les testeurs et développeurs sont eux-mêmes chargés de l'élaboration des organigrammes : si les analystes métier et les utilisateurs privilégiaient l'utilisation d'organigrammes à la place de langages comme Gherkin, cette critique serait sans fondement. Par ailleurs, même si les équipes techniques doivent inverser l'ingénierie des exigences BDD, le temps passé à réaliser un organigramme reste généralement minime. Une fois l'organigramme terminé, il peut facilement être réutilisé, même si les exigences changent. Cela va à l'encontre de l'idée répandue qu'une documentation complète implique forcément une incapacité à réagir rapidement aux changements.

Dans CA Agile Requirements Designer, rien de plus facile et rapide que de modifier un organigramme ; il suffit à l'utilisateur d'ajouter un nouvel élément de logique fonctionnelle. Toute cassure au niveau d'un plan de test est automatiquement identifiée et réparée, et les plans de test en doublon ou redondants sont supprimés. CA Agile Requirements Designer génère ensuite tout nouveau plan de test nécessaire pour assurer une couverture fonctionnelle la plus complète possible.

Cela permet d'optimiser la valeur du travail initial, à savoir la réalisation de l'organigramme, et supprime le temps perdu à vérifier manuellement chaque plan de test en cas de changement. À titre d'exemple, une entreprise n'a mis que deux minutes à modifier un organigramme existant à la suite d'une demande de changement. CA Agile Requirements Designer a identifié et réparé automatiquement les trois plans de test concernés et laissé les 64 autres intacts. Comme le montre cet exemple, la modélisation d'un organigramme implique par ailleurs que seuls les aspects d'un système ayant subi des modifications feront l'objet de nouveaux tests, ce qui améliore l'efficacité de la boucle de retour.

En somme, le temps passé à créer un organigramme est, dans la plupart des cas, compensé par le temps gagné sur certaines tâches : remaniement et débogage dus à des exigences incomplètes et ambiguës, création et exécution des plans de test, et pour finir implémentation des demandes de changement. Par conséquent, le principe d'une documentation complète ne s'oppose pas à celui d'un développement guidé par les comportements, mais peut bien au contraire améliorer sa mise en œuvre.

**Section 7**

## Résumé

La modélisation d'organigramme est une technique qui permet d'introduire une documentation complète dans un BDD, sans porter atteinte aux principes fondamentaux. L'organigramme peut découler directement des interactions avec les parties prenantes et doit être compréhensible à la fois par les acteurs métier et IT. Il utilise un langage universel qui peut être remplacé par le langage fonctionnel d'un système sur lequel s'appuient les équipes chargées des tests et du développement. Un organigramme permet également de réduire le « délai de retour », car les cycles de test sont raccourcis et les efforts manuels correspondants plus limités.

Une modélisation formelle augmente les chances d'obtenir du premier coup un logiciel conforme aux exigences, car elle regroupe un ensemble clair et complet d'exigences axées sur les comportements. Cela réduit le nombre total de boucles de retour nécessaires avant qu'un logiciel soit conforme aux exigences métier, afin que les entreprises puissent reporter leurs efforts sur plus d'innovation au lieu d'être accaparées par des tests à répétition et des remaniements fastidieux. Cette capacité à développer en permanence des solutions dans le but d'optimiser leur valeur auprès des clients est renforcée par la capacité à s'adapter rapidement et facilement aux changements.

**Section 8**

## À propos de l'auteur

Avec une carrière qui a débuté il y a plus de 30 ans, Huw Price a travaillé comme architecte technique principal pour plusieurs éditeurs de logiciels européens et américains. Il a apporté son aide à des banques internationales, de grandes agences de service public et des fournisseurs de soins de santé, pour la conception d'une architecture sophistiquée. Élu « Directeur informatique de l'année 2010 » par QA Guild, Huw s'est spécialisé dans la conception d'outils d'automatisation de tests et a lancé plusieurs produits innovants qui ont redéfini le modèle de test utilisé par l'industrie du logiciel. Il intervient désormais dans le cadre d'événements de renommée mondiale et son travail a été publié dans de nombreuses revues spécialisées, parmi lesquelles Professional Tester et CIO Magazine.

Sa dernière entreprise, Grid-Tools, a été acquise par CA Technologies en juin 2015. Depuis près de 10 ans déjà, Grid-Tools a redéfini la manière dont les grandes entreprises abordent leur stratégie de test. Grâce au leadership et à l'approche visionnaire de Huw, l'entreprise a adopté une solide approche de la phase de test, axée sur les données, en lançant de nouveaux concepts formulés par Huw comme les « objets-données », l'« héritage des données » et « un entrepôt centralisé de données de test ».

**Restez connecté à CA Technologies sur ca.com/fr**

CA Technologies (NASDAQ : CA) fournit les logiciels qui aident les entreprises à opérer leur transformation numérique. Dans tous les secteurs, les modèles économiques des entreprises sont redéfinis par les applications. Partout, une application sert d'interface entre une entreprise et un utilisateur. CA Technologies aide ces entreprises à saisir les opportunités créées par cette révolution numérique et à naviguer dans « l'Économie des applications ». Grâce à ses logiciels pour planifier, développer, gérer la performance et la sécurité des applications, CA Technologies aide ainsi ces entreprises à devenir plus productives, à offrir une meilleure qualité d'expérience à leurs utilisateurs, et leur ouvre de nouveaux relais de croissance et de compétitivité sur tous les environnements : mobile, Cloud, distribué ou mainframe. Pour en savoir plus, rendez-vous sur **ca.com/fr**.

1 Dan North, « Introducing BDD (2006) », extrait le 06/03/2015 à partir du site http://dannorth.net/introducing-bdd/.

2 « Chaos Manifesto 2013 » (Standish Group: 2013), extrait le 07/03/2015 à partir du site http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf.

3 « Requirements Based Testing Process Overview » (Bender RBT: 2009), extrait le 05/03/2015 à partir du site http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf.

4 « Why testing should start early in software development life cycle? » (Software Testing Class: 2012), extrait le 06/03/2015 à partir du site http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/.

5 Bender RBT, « Requirements Based Testing Process Overview ».

6 Kathleen Barret, « Business Analysis: The Evolution of a Profession » (IIBA: 2013), extrait le 05/03/2015 à partir du site http://www.iiba.org/Careers/Careers/Business-Analysis-The-Evolution-of-a-Profession.aspx.

7 Voir Erik Kamsties, « Understanding Ambiguity in Requirements », cité dans « Engineering and Managing Software Requirements » (Springer: 2005), P. 250.

8 Pour obtenir un exemple, se reporter à http://en.wikipedia.org/wiki/Behavior-driven_development#Story_versus_specification

9 Dan North, « Embracing Uncertainty » (Goto Con: 2013), extrait le 06/03/2015 à partir du site http://gotocon.com/dl/goto-chicago-2013/slides/DanNorth_EmbracingUncertainty.pdf

10 « Test Case Generation », extrait le 06/03/2015 à partir du site https://www.ca.com/fr/collateral/industry-analyst-report/bloor-research-market-report-test-case-generation.register.html



CS200-196736_0516