Les tests dans le développement logiciel, du cycle en V aux méthodes agiles

Isabelle Blasquez, Hervé Leblanc, Christian Percebois

▶ To cite this version:

Isabelle Blasquez, Hervé Leblanc, Christian Percebois. Les tests dans le développement logiciel, du cycle en V aux méthodes agiles. Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques, 2017, 36 (1-2), pp.7-50. 10.3166/tsi.2017.00003 . hal-02864393

HAL Id: hal-02864393
https://hal.science/hal-02864393

Submitted on 11 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive Toulouse Archive Ouverte

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in:
http://oatao.univ-toulouse.fr/22306

Official URL

https://tsi.revuesonline.com/article.jsp?articleId=39250

To cite this version: Blasquez, Isabelle and Leblanc, Hervé and Percebois, Christian *Les tests dans le développement logiciel, du cycle en V aux méthodes agiles.* (2017) Technique et Science Informatiques, 36 (1-2). 7-50. ISSN 0752-4072

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

Les tests dans le développement logiciel, du cycle en V aux méthodes agiles

Isabelle Blasquez[1], **Hervé Leblanc**[2], **Christian Percebois**[2]

1. *Université de Limoges*
   isabelle.blasquez@unilim.fr
2. *Université de Toulouse, laboratoire IRIT*
   herve.leblanc,christian.percebois@irit.fr

RÉSUMÉ. *Le test logiciel est une méthode empirique utilisée pour la vérification et la validation de systèmes complexes. Il est notamment déployé lors de la phase ascendante du cycle en V au travers des tests unitaires, d'intégration et d'acceptation. Ces différents tests, dits classiques, s'appliquent a posteriori à un code déjà développé. Le développement agile, promouvant à l'extrême certaines bonnes pratiques du génie logiciel, fait jouer un rôle de première importance aux tests. En particulier, les cycles de développement dirigés par les tests utilisent les tests pour spécifier en sus de vérifier et forcent à leur automatisation. Dans cet article, nous montrons que les tests classiques et les tests agiles ne sont pas antinomiques ; bien au contraire ces deux approches peuvent s'enrichir l'une de l'autre.*

ABSTRACT. *Software testing is an empirical approach increasingly used for verification and validation of complex systems. It is especially deployed on the upward-sloping branch of the V-model through unit testing, integration testing and acceptance testing. Usually, these tests are performed after the development phase on an already written production code. Agile software development pushes some best traditional software engineering practices at extreme levels. In this context, testing is considered as a first and major element of a developement process. Test driven development cycles not only use test cases to check errors but also to specify requirements and lead to test automation. In this paper, we show that usual and agile testing are not opposite, but rather can mutually enhance one another.*

MOTS-CLÉS : *test logiciel, cycle en V, méthodes agiles*

KEYWORDS: *software testing, V-model, agile software development*

DOI:10.3166/TSI.x.1-44 © 2017 Lavoisier

1. Introduction

Les rapports CHAOS du Standish Group fournissent chaque année une vision du taux de réussite des projets informatiques en menant une grande enquête auprès des

services informatiques de sociétés. L'édition 2013 de ce rapport montre qu'environ 40% des projets informatiques sont considérés comme des succès (Standish-Group, 2013). Il souligne une augmentation du taux de réussite des projets et constate dans le même temps une augmentation du nombre de projets agiles et de petits projets. L'étude attribue d'ailleurs pour 10 points sur 100 l'augmentation des projets ayant du succès à l'adoption des méthodes agiles. Ces rapports sont souvent repris et cités dans des présentations sur les méthodes agiles pour pointer une crise perpétuelle dans l'industrie logicielle, pour appeler à changer de paradigme de processus de développement et pour conforter la pertinence de ces « nouvelles » approches.

L'émergence spontanée de ces méthodes portées par des praticiens sur des bases empiriques n'a pas favorisé leur adoption par la communauté du génie logiciel et plus particulièrement la communauté s'intéressant aux tests. Dans un article publié sur le site de la Communauté Française des Tests Logiciels (Jorgensen, 2010), l'auteur se pose des questions sur la pertinence de l'approche TDD (*Test Driven Development*, développement dirigé par les tests) sur des cas réels, en particulier : le passage à l'échelle sur des systèmes complexes, le traitement des besoins non fonctionnels, la détection de l'inconsistance entre plusieurs besoins exprimés par des histoires utilisateur (*user stories*) et la naïveté d'une approche ascendante pour des architectures émergentes. Même si le TDD n'est qu'une pratique et ne correspond pas à un processus de développement logiciel, il existe des points communs entre la communauté agile et la communauté des tests logiciels.

La communauté scientifique traitant des tests étant antérieure à la communauté agile, nous présentons en premier lieu la place des tests dans le développement logiciel dit classique et poursuivons par la place des tests dans un cycle en V, premier cycle où les tests apparaissent comme des artefacts indispensables au développement logiciel. Dans un deuxième temps, nous nous intéressons à la mise en œuvre des cycles de développement agiles dirigés par les tests. Nous montrons ensuite que la différence d'utilisation des tests dans ces processus de développement induisent des changements sur les propriétés intrinsèques des tests et sur les métiers de testeur et développeur. Les tests classiques et les tests agiles ne sont pas antinomiques ; bien au contraire ces deux approches peuvent s'enrichir l'une de l'autre. Nous proposons en conclusion une réécriture des valeurs agiles dédiée aux tests.

La littérature concernant les tests agiles est composée en grande partie de livres et de blogs. Faute d'un nombre suffisant de publications significatives dans des conférences du domaine, nous n'avons pu produire un document s'approchant d'une revue de littérature systématique, comme par exemple (Yusifoğlu *et al.*, 2015). Cet article se veut un panorama des tests dans le développement logiciel, du cycle en V aux méthodes agiles. De plus, il décrit en quoi les méthodes agiles offrent un nouvel éclairage sur l'activité de test.

2. Les tests dans le développement logiciel

Tout au long d'un processus de développement logiciel d'un système complexe, les méthodes de vérification et de validation cohabitent afin d'assurer une certaine qualité du produit final. La qualité interne est liée aux artefacts de développement et la qualité externe est mesurée par la satisfaction des exigences client (IEEE, 1993).

Les tests sont un des moyens pouvant être mis en œuvre pour vérifier et valider les différentes productions d'un développement logiciel. Après avoir explicité la place des tests dans un tel contexte, nous donnons sous la forme d'une typologie, le glossaire commun des principaux tests qu'ils soient effectués dans un cycle en V dit traditionnel ou dans un cycle dit agile. Un point clé relatif à l'adoption des tests dans un processus de développement logiciel concerne leur automatisation.

2.1. Les tests comme outil de vérification et de validation

La vérification permet de s'assurer *a priori* que le produit fonctionnera correctement. Le processus de vérification évalue un système ou un composant afin de déterminer si les conditions imposées au début d'une phase de développement sont satisfaites (IEEE, 1990). La vérification a donc pour but de démontrer que la phase de développement est conforme à son plan de réalisation. Vérifier c'est répondre à la question suivante :

Am I building the product right ? [1] (Boehm, 1984)

La validation permet de s'assurer *a posteriori* que le produit respecte les exigences du client. Le processus de validation évalue un système ou un composant afin de déterminer s'il satisfait les exigences spécifiées auparavant. La validation peut s'effectuer à tout moment du développement du cycle de développement (IEEE, 1990). Valider c'est répondre à la question suivante :

Am I building the right product ? [2] (Boehm, 1984)

Parmi les méthodes de vérification et validation citons les revues, les analyses, les tests, les preuves de programmes, ... Le tableau 1 récapitule les différentes approches. Ces méthodes peuvent être dynamiques si elles nécessitent l'exécution d'un composant du logiciel ou du système, ou statiques si elles s'appuient sur une analyse des artefacts existants. En cas de résultat positif, les méthodes statiques formelles ont l'avantage d'avoir la certitude qu'une propriété est satisfaite pour toute donnée, mais elles ont pour inconvénient de ne donner que peu d'indications en cas d'échec. Les méthodes dynamiques ont l'avantage en cas de résultat négatif de donner explicitement un contre-exemple à une situation défaillante, mais elles ont pour inconvénient de ne pas être exhaustives. Néanmoins, les approches statiques et dynamiques peuvent être utilisées conjointement dans un cycle de développement (Gaudel, 2011). Les tests ne peuvent pas garantir l'exactitude du système en raison de leur non-exhaustivité :

---

1. Suis-je en train de construire le produit correctement ?
2. Suis-je en train de construire le produit correct ?

> *Program testing can be used to show the presence of bugs, but never to show their absence !* [3] (Dijkstra, 1972)

Tableau 1. Méthodes de vérification et de validation

	Vérification	**Validation**
Méthodes statiques	revue de code revue de conception inspection analyse statique preuve de programme vérification de modèle	
Méthodes dynamiques	**test** analyse dynamique	**test**

Le test est une des méthodes de vérification et de validation les plus utilisées. En moyenne les tests représentent 30 à 70% du coût de développement d'un logiciel selon la complexité du système (Printz, Pradat-Peyre, 2009).

DÉFINITION 1. — *Le test consiste à exécuter et évaluer un système ou un composant sous des conditions spécifiques, pour vérifier qu'il répond à ses spécifications ou pour identifier des différences entre les résultats spécifiés et attendus et les résultats effectivement obtenus (IEEE, 1990).*

La définition du standard IEEE restreint le rôle des tests à du contrôle uniquement. Cependant, les tests pourraient être utilisés pour des aspects qualitatifs du développement logiciel. Myers propose d'utiliser les tests pour détecter des erreurs et ainsi accroître la fiabilité, la qualité et la confiance dans les programmes. Ces erreurs permettent de corriger et par là même d'améliorer le code.

> *Testing is the process of executing a program with the intent of finding errors.* [4] (Myers, Sandler, 2004)

Ces erreurs peuvent intervenir à différentes étapes du cycle de développement. Chaque étape d'un cycle de développement logiciel se prêtera donc à un type de test spécifique. C'est un des axes de classification qu'a développé Tretmans (Tretmans, 1999) et que nous explicitons à la section suivante.

2.2. Typologie des tests

La classification des tests proposée par Tretmans est en général représentée dans un repère à trois dimensions comme le montre la figure 1. La première dimension

3. Tester des programmes peut être utilisé pour montrer la présence d'erreurs, mais jamais leur absence !
4. Tester consiste à excéuter des programmes avec l'intention de trouver des erreurs.

représente le niveau d'abstraction du système à tester, aussi nommé SUT (*System Under Test*, système sous test) (ISTQB, 2015). Nous avons choisi de renommer cet axe « granularité » qui ne connote pas un cycle de développement en V. La deuxième dimension représente les critères de qualité que l'on est en droit d'attendre d'un système logiciel. Ces critères peuvent-être fonctionnels ou non. Nous avons nommé cet axe « qualité » et non « types » ou « caractéristiques » comme énoncé habituellement. La dernière dimension représente le niveau de visibilité des artefacts à tester. D'après Tretmans, ces trois axes sont indépendants et complémentaires. Chaque point posé dans cet espace définit une catégorie ou un type de tests. Une approche test « boîte blanche » ou test « boîte noire » peut être choisie pour chacun des niveaux du système à tester.

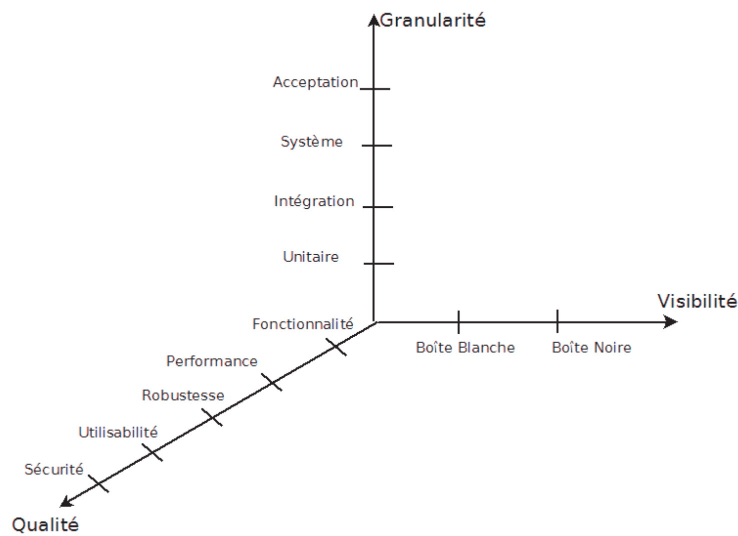

Figure 1. Typologie des tests

2.2.1. *Axe de la granularité*

Ce premier axe permet de positionner un test selon la granularité de la partie du système à tester : de la plus petite unité testable jusqu'au système complet. La classification comporte quatre niveaux : unitaire, intégration, système, acceptation. Le test unitaire s'intéresse aux éléments logiciels minimaux appelés aussi composants, le test d'intégration aux interactions entre les composants, le test système au système complet et le test d'acceptation à l'application réellement installée sur son site d'utilisation.

DÉFINITION 2. — *Un test unitaire est un test qui permet de tester de manière isolée une unité logicielle ou un groupe d'unités (IEEE, 1990).*

Le test unitaire est également appelé test de composant. Dans la programmation par objets, l'objet sous test peut être alors une méthode spécifique, une classe ou un groupe de classes formant un composant. Il permet de s'assurer que la logique du pro-

gramme est respectée et que le composant a un comportement conforme à sa conception (IEEE, 1993). Les tests unitaires sont des tests dits en isolation car les composants sont testés individuellement les uns des autres, et ce dans n'importe quel ordre.

DÉFINITION 3. — *Un test d'intégration est un test dans lequel les composants logiciels, les composants matériels ou les deux sont combinés pour tester et évaluer leurs interactions (IEEE, 1990).*

Un test d'intégration considère des composants testés unitairement. Alors que le test unitaire se focalise sur le comportement d'un composant isolé, le test d'integration s'attache à vérifier le bon comportement de l'assemblage de plusieurs composants.

Il existe plusieurs stratégies quant à l'intégration des composants et de leurs tests associés. Les trois principales sont la stratégie Big-Bang et les stratégies incrémentales descendante (*top-down*) et ascendante (*bottom-up*) (ISTQB, 2015). L'approche Big-Bang consiste à combiner tous les composants directement en un système complet. Avec une telle stratégie, il est difficile de localiser une erreur. Les stratégies incrémentales se basent sur le graphe de la relation de dépendance entre composants pour déterminer un ordre d'intégration.

DÉFINITION 4. — *Un test système est un test mené sur un système entièrement intégré afin de vérifier si celui-ci respecte les exigences spécifiées (IEEE, 1990).*

Le test système est le premier test qui s'intéresse au comportement global du système, le test d'intégration se focalisant sur les connexions d'interfaces entre composants. Le test système se doit d'être effectué dans un environnement le plus proche possible de l'environnement de production au regard du système déployé.

DÉFINITION 5. — *Un test d'acceptation (ou anciennement test de recette) permet de déterminer si un système satisfait ou non à ses critères d'acceptation et permet au client d'accepter ou non le système (IEEE, 1990).*

Si le test système porte sur l'adéquation des fonctionnalités du système aux spécifications, le test d'acceptation confirme en conditions réelles la validité du système d'un point de vue du client. Le test d'acceptation est donc la dernière étape avant la mise en œuvre opérationnelle d'un système. Il est d'abord effectué par l'équipe de développement (*alpha testing*), puis par un panel d'utilisateurs finaux à l'aide de scénarios réels (*beta testing*).

Dans un cycle de développement, la granularité correspond à une phase ou à un niveau. Le but d'un test n'est pas encore explicité en termes de critères de qualité. C'est l'objectif de la classification suivante.

2.2.2. Axe de la qualité

Les méthodes de vérification et de validation permettent d'assurer une certaine qualité logicielle. La norme ISO 25010 (ISO/IEC, 2010) définit des critères de qualité pouvant être pris en compte dans une telle démarche. Ces critères sont l'aptitude fonctionnelle, l'efficacité en termes de performance, la compatibilité, la facilité d'uti-

lisation, la fiabilité, la sécurité, la maintenabilité et la portabilité. Les tests mentionnés sur cet axe sont relatifs à une partie des critères énoncés ci-dessus. Cet axe couvre à la fois les aspects fonctionnels (le « quoi ») et les aspects techniques (le « comment ») du système.

DÉFINITION 6. — *Un test de fonctionnalité est un test qui permet de s'assurer que le produit logiciel fournit un ensemble de services répondant aux besoins sous des conditions d'utilisation déterminées (ISTQB, 2015).*

L'objectif de ce type de test est de garantir le comportement du système par rapport à un ensemble de spécifications dédiées à une fonctionnalité du système. Ce type de test est souvent appelé à tort test fonctionnel, d'après une interprétation erronée de l'anglicisme *functionality testing*.

DÉFINITION 7. — *Un test de performance est un test qui permet d'évaluer la capacité d'un système ou d'un composant à fonctionner correctement sous contraintes de temps et de ressources (ISTQB, 2015).*

Le test de charge en est un exemple : il consiste à évaluer le comportement du système lorsque la charge augmente en nombre d'utilisateurs et/ou en nombre de transactions. Ce test détermine quelle charge limite le système est capable de supporter (ISTQB, 2015).

DÉFINITION 8. — *Un test de robustesse est un test qui permet de tester la capacité d'un système ou d'un composant à fonctionner correctement en présence d'entrées invalides ou de conditions d'utilisation stressantes (ISTQB, 2015).*

L'objectif de ce type de test est de s'assurer que le système supporte des utilisations imprévues, comme un dysfonctionnement matériel ou logiciel. Par exemple, l'arrêt inattendu de certains composants de l'infrastructure pourra être simulé afin de vérifier que l'architecture ou l'application est tolérante à ce type d'incident.

DÉFINITION 9. — *Un test d'utilisabilité est un test qui permet de déterminer dans quelle mesure le logiciel est appréhendé, c'est-à-dire facile à apprendre, facile à utiliser et attrayant pour les utilisateurs (ISTQB, 2015).*

Ce type de test consiste à observer un utilisateur et à relever ses efforts pour comprendre, apprendre et exploiter le logiciel. Ces tests peuvent être soit scénarisés, soit exploratoires lorsque l'utilisateur est invité à utiliser librement le logiciel. Ils permettent de mettre en évidence des problèmes d'ergonomie.

DÉFINITION 10. — *Un test de sécurité est un test qui permet de prévenir les accès non autorisés, qu'ils soient accidentels ou délibérés, sur les programmes ou les données (ISTQB, 2015).*

L'objectif de ce type de test consiste à s'assurer que le système n'est pas vulnérable lors d'une attaque de l'extérieur. Des simulations d'attaques peuvent être mises en œuvre pour découvrir des faiblesses du système qui porteraient atteinte à son intégrité.

2.2.3. *Axe de la visibilité*

Le troisième axe de la classification s'intéresse à la visibilité de la structure interne du système à tester.

DÉFINITION 11. — *Un test boîte blanche est un test qui est exécuté sur un système ou un composant dont la structure interne est connue (IEEE, 1990).*

Ce type de test est aussi appelé test structurel.

DÉFINITION 12. — *Un test boîte noire est un test qui ignore les mécanismes internes d'un système et qui vérifie si les sorties obtenues sont bien celles prévues pour des entrées et des conditions d'exécution données (IEEE, 1990).*

Le test boîte noire est centré sur les exigences tandis que le test boîte blanche est centré sur les potentielles erreurs de programmation.

Par delà cette typologie, certains tests peuvent être rejoués tout au long du cycle de développement, leur conférant un autre rôle que la vérification et la validation.

2.3. *Automatisation des tests*

Une dimension fondamentale oubliée dans la typologie des tests de la figure 1 est l'automatisation qui peut être totale, partielle ou inexistante. En effet, les activités de test sont coûteuses et nécessitent des ressources. Elles peuvent atteindre jusqu'à 50% des coûts du développement, et plus pour les applications critiques. Le test induit de nombreuses tâches indirectes comme le maintien des scripts de test, l'exécution des tests, la comparaison des résultats attendus avec les résultats réels. L'objectif de l'automatisation des tests est de réduire autant que possible ces coûts, de minimiser l'erreur humaine, et de mettre en place plus facilement des tests de régression (Ammann, Offutt, 2008).

DÉFINITION 13. — *L'automatisation des tests consiste à utiliser des logiciels pour exécuter ou supporter les activités liées aux tests (ISTQB, 2015).*

L'une des activités liée au support des tests concerne la génération automatique de cas de tests à partir de plusieurs sources : des modèles, du code et des spécifications formelles (Mathur, 2008). Dans notre comparatif sur la place des tests, nous restreignons l'automatisation aux activités d'exécution et de génération de rapports d'erreurs.

Avec l'automatisation, réeffectuer un test déjà effectué sur un programme devient pertinent pour vérifier la non-régression d'un code. Les tests de régression peuvent être exécutés à tout moment et s'appliquent à pratiquement tout type de test de la typologie. Ils préviennent de la régression du logiciel et sont aussi appelés tests de non-régression. Etant exécutés idéalement à chaque restructuration interne du logiciel, les tests de régression doivent être automatisés et rendre rapidement un verdict.

DÉFINITION 14. — *Un test de régression est un test effectué sur un programme préalablement testé, après une modification, pour s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées du logiciel (ISTQB, 2015).*

Dans la typologie présentée, nous pouvons remarquer que les trois axes de la classification des tests sont diversement couverts par les normes. Les axes relatifs à la granularité et à la visibilité sont bien couverts par les normes IEEE. Par contre, les tests de l'axe de la qualité (ou des « critères en ité ») sont simplement énoncés dans la norme ISTQB, en faisant uniquement référence au « critère en ité », spécifié par la norme ISO (ISO/IEC, 2010). Ces tests ne sont pas caractérisés par une mise en œuvre explicite.

Ces différents types de tests sont agencés de manière différente dans un cycle de développement traditionnel et un cycle de développement agile. Nous commençons par positionner les tests dans le cycle en V, la première approche les ayant mis en exergue.

3. Les tests dans le cycle en V

Le cycle en V (Overmyer, 1990) est un standard de l'industrie logicielle depuis les années 80, une extension du modèle en cascade (*waterfall model*) proposé par (Royce, 1970). Contrairement aux idées reçues, le modèle en cascade n'est pas une suite de phases s'exécutant séquentiellement de manière naïve, afin d'obtenir à partir de la définition des exigences, un logiciel sûr et conforme aux attentes du donneur d'ordre. Le modèle en cascade a servi à Royce à introduire des bonnes pratiques de développement dédiées à la production de systèmes complexes.

Dans le modèle en cascade, les différentes phases se chevauchent, les itérations ne sont jamais confinées entre deux phases successives du processus et parmi les bonnes pratiques énumérées, l'on peut citer :

– Faire le logiciel deux fois, surtout s'il est complètement original, ce qui annonce le cycle en W : un cycle en V exécuté deux fois, une fois pour le prototype et une fois pour le produit final.

– Planifier, contrôler et monitorer les tests, ce qui introduit un processus dédié aux tests.

– Faire participer le client au projet, ce qui est l'un des buts principaux des méthodes agiles !

À partir du modèle en cascade, le cycle en V étend la portée des tests et exhibe les différentes itérations possibles. Après avoir présenté les différentes phases du cycle, nous nous attachons à décrire précisément ce qu'est un processus de test. Nous concluons en plaçant le processus de test dans les phases d'un développement traditionnel.

3.1. *Le cycle en* V

Ce modèle de développement, dominant pour la production de systèmes complexes, positionne visuellement dans un V ses différentes phases faisant référence à la Vérification et à la Validation. La figure 2 présente les deux branches du V ainsi que les boucles de rétroaction (*feedback*) consécutives à l'exécution des différents types de tests.

La branche descendante du V enchaîne les différentes phases d'un modèle en cascade : de l'analyse des besoins jusqu'à la programmation des composants logiciels. Les spécifications doivent répondre aux exigences, les modèles de conception doivent réaliser les fonctionnalités spécifiées et le code doit fournir une implantation correcte des modèles de conception. Les différents plans de tests se doivent d'être élaborés parallèlement. La branche ascendante du V enchaîne une succession d'étapes de vérification où les composants une fois implémentés sont testés et intégrés graduellement dans les différents sous-systèmes jusqu'au logiciel complet. Les tests de recette ou d'acceptation valident le système obtenu. A chaque phase de la branche descendante un plan de test est généré, donnant lieu à un retour lors de la phase duale ascendante.

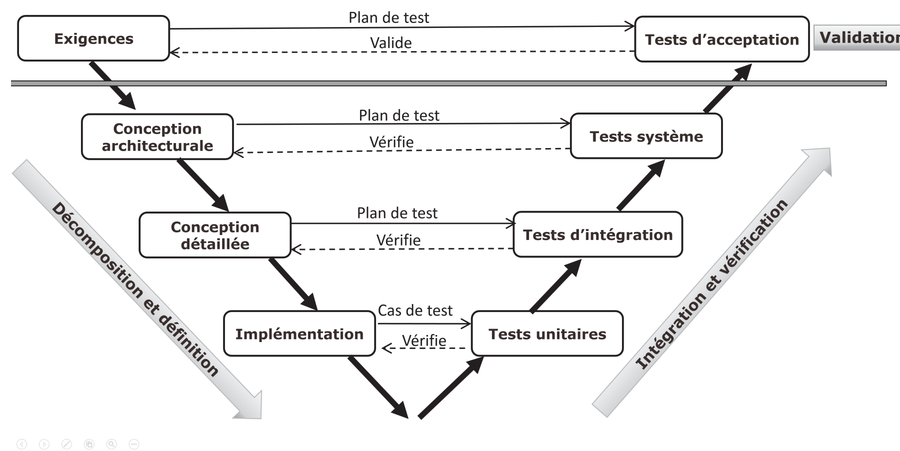

Figure 2. Cycle de développement en V

Le *Guide for Software Verification and Validation Plans* (IEEE, 1993) définit les différentes phases de ce cycle de la manière suivante.

La phase d'analyse des exigences est la période du cycle de vie pendant laquelle les exigences, fonctionnelles et non fonctionnelles du produit logiciel, sont définies et documentées. Les exigences fournissent des contraintes à la fois qualitatives et quantitatives sur la conception ultérieure du système et sur sa mise en œuvre. Cette phase donne lieu à l'écriture d'un document de spécifications (*Software Requirements Spe-*

cifications) qui précise les missions du logiciel. Ce document est une trace des besoins utilisateurs et sera utilisé dans toutes les autres phases du cycle de développement.

La phase de conception est la période du cycle de vie pendant laquelle l'architecture logicielle, les composants logiciels, les données et les interfaces sont conçus et documentés afin de satisfaire aux exigences. Bien que les logiciels simples puissent être conçus en une seule étape, la conception est souvent décomposée en deux étapes. La première étape, appelée « conception architecturale », spécifie les caractéristiques architecturales en termes de sous-systèmes et d'interfaces. La seconde étape, appelée « conception détaillée », est une succession d'étapes qui explicite les sous-systèmes de manière suffisament précise pour en dériver le code.

La phase d'implémentation est la période du cycle de vie pendant laquelle le logiciel est créé à partir des spécifications de conception. Les tâches de cette phase se concentrent autour du code où les composants sont implémentés et testés individuellement.

La phase de test est la période du cycle de vie consacrée à l'intégration et à l'évaluation des composants et du logiciel afin de vérifier les exigences aussi bien au niveau système qu'utilisateur. Cette phase est constituée d'une succession d'activités de test. Lorsque des défauts sont détectés, un cycle de maintenance est enclenché sur la phase descendante en vis-à-vis.

3.2. Le processus de test

Les tests suivent eux-même un processus de développement représenté au centre de la figure 3. Ce processus est générique : il est valide pour tout type de test quelle que soit sa granularité. Le processus de test regroupe les activités de conception, d'implémentation, d'exécution et d'évaluation des critères de sortie qui sont représentées selon le modèle de processus de test proposé par (Sommerville, 2010). Si le processus de test s'exécute dans un processus plus global de développement, ici le cycle en V, trois activités supplémentaires liant les deux processus sont nécessaires : une activité de planification des tests en amont (branche descendante), une activité de clôture des tests en aval (branche ascendante) et en continu une activité transverse de suivi et de contrôle régulant le processus de test lui-même. Après la présentation de ces activités, nous explicitons les artefacts de développement propres aux tests et l'enchaînement des différentes activités du processus de test.

3.2.1. Contexte

Les processus de test sont alignés sur les phases correspondantes de la branche descendante du cycle en V par l'envoi d'un plan de test issu de l'activité planification et par la gestion du retour du rapport de test issu de l'activité clôture de test. Ces activités contextuelles permettent de lier les phases de développement et les phases de vérification et validation comme mis en évidence à la figure 2.

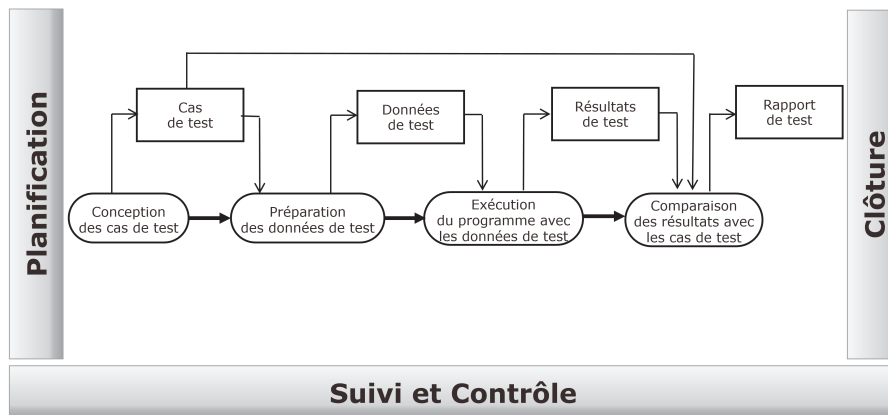

Figure 3. Processus générique de test dans un cycle en V

La planification consiste à définir les objectifs généraux et à spécifier les activités de test et les ressources nécessaires à mettre en œuvre pour atteindre ces objectifs. Elle donne lieu à un plan de test qui décrit la portée, l'approche, les ressources et le planning de l'activité de test.

DÉFINITION 15. — *Un plan de test indique ce qui doit être testé, ce qui ne doit pas l'être, les tâches de test à effectuer, les responsables pour chaque tâche et les risques (IEEE, 1993).*

Le plan de test est le document d'entrée de l'activité de conception de test qui permettra de concevoir les différents cas de test.

La clôture des tests est menée en fin de processus. Elle permet de s'assurer que chaque activité de test a été menée à terme, de transmettre les artefacts de test, d'archiver les résultats et les documents produits au cours du processus, d'organiser des retours d'expérience. Les leçons apprises, à la fois sur le processus et sur le cycle de développement du logiciel, peuvent être documentées afin d'améliorer les livraisons futures. Ces tâches devraient être explicitement inclues dans le plan de test.

Le suivi et le contrôle permettent de s'assurer que ce qui a été planifié est exécuté correctement. C'est une activité récurrente dans le processus de test qui implique de mesurer l'avancement des activités, à la fois en termes de temps passé, mais aussi en termes d'objectifs prévus. Elle compare l'avancement réel par rapport à l'avancement prévu et révise la planification si nécessaire en proposant des actions correctives pour atteindre les objectifs. Les métriques liées à cette activité peuvent inclure la couverture des risques par les tests, la découverte d'anomalies, le temps passé à développer et exécuter les tests.

La conception des cas de test consiste à transformer les objectifs de test généraux en cas de test en identifiant les comportements à tester pour satisfaire ces objectifs. Le cœur du processus de test s'organise autour d'un test.

DÉFINITION 16. — *Un test est défini comme un ensemble de cas de test, accompagné éventuellement de procédures de test (IEEE, 1990).*

DÉFINITION 17. — *Un cas de test se compose d'un ensemble de valeurs d'entrée, de conditions d'exécution, de résultats attendus et qui est développé pour un objectif donné ou une condition de test particulière, tel qu'exécuter un chemin particulier d'un programme ou vérifier le respect d'une exigence spécifique (IEEE, 1990).*

DÉFINITION 18. — *Une procédure de test contient le détail des instructions pour la mise en place, l'exécution et l'évaluation des résultats des tests pour un cas de test donné (IEEE, 1990).*

Une procédure de test permet d'expliciter la mise en œuvre du cas de test.

Le format des documents afférents aux différentes activités d'un processus de test est spécifié par le standard (IEEE, 2008). Le niveau de formalisme dépend du contexte des tests incluant la maturité des tests et des processus de développement, les contraintes de temps, les exigences de sûreté de fonctionnement et les personnes impliquées.

La préparation des données de test permet de concrétiser les cas de test en déterminant des données de tests spécifiques, des résultats attendus précis et éventuellement des scripts de test. Cette activité comprend entre autres l'organisation des tests (à la fois manuels et automatisés) dans un certain ordre d'exécution, la finalisation des données et des environnements de test et l'éventuelle création d'une suite de procédures de test. Les cas de test et les procédures de test sont donc finalisés, implémentés et priorisés durant cette activité ; c'est pourquoi on emploie également les termes de génération de cas de test et de génération de procédures de test pour désigner les activités liées à la préparation des données.

L'exécution du test avec les données de test permet d'exécuter les cas de test selon les procédures de test et de consigner les résultats obtenus.

La comparaison des résultats obtenus avec les résultats attendus permet via un oracle de test de fournir un verdict consigné dans un rapport de test.

DÉFINITION 19. — *Un rapport de test est un document qui décrit le déroulement et les résultats des tests effectués sur un système ou un composant (IEEE, 1990).*

Bien que logiquement séquentielles, les activités du processus de test peuvent se chevaucher partiellement ou être mises en œuvre de manière concurrente. Une adaptation de ces activités se fera donc en fonction du système ou du projet.

3.3. Alignement du cycle en V et du processus de test

Selon la granularité du test, les ressources, les méthodes et les environnements diffèrent. Chaque granularité de test dispose donc de son propre plan de test.

Le *Guide for Software Verification and Validation Plans* (IEEE, 1993) propose, pour chaque granularité de test, une répartition des différentes activités de test dans les quatre phases majeures d'un développement logiciel : exigences, conception, implémentation et test. La figure 4 illustre l'alignement des activités de test dans le cadre du cycle en V.

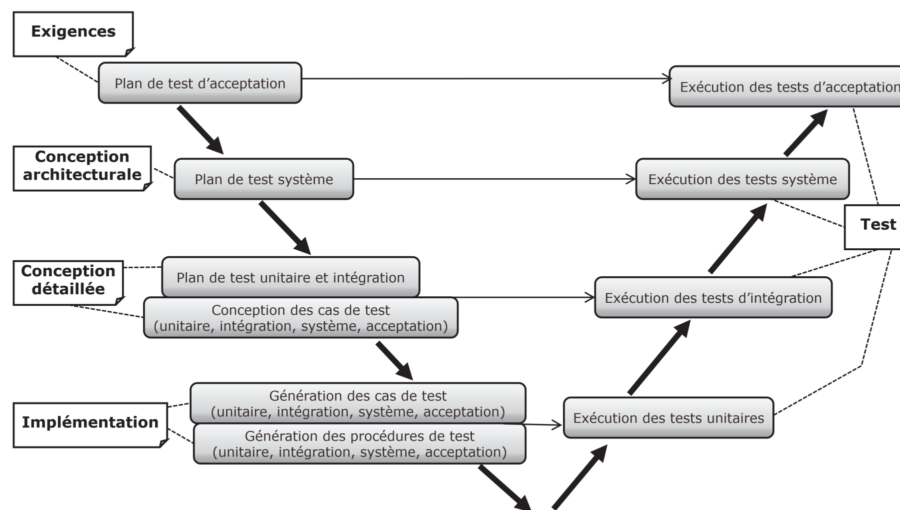

Figure 4. Alignement du processus de test dans le cycle en V

Durant la phase d'analyse des exigences, les activités de test se concentrent sur la génération des plans de test d'acceptation dont les bonnes pratiques sont détaillées dans (IEEE, 1993) et (IEEE, 2008). Outre vérifier la testabilité des exigences retenues, la rédaction des plans de test permet d'identifier une ambiguïté, un manque de clarté ou un oubli. La phase de conception enrichit et affine la planification des tests. La phase d'implémentation se focalise sur la génération des cas de test et des procédures de test.

Le long de la branche ascendante du cycle en V, les activités de test se concentrent sur l'exécution des tests et se basent sur les plans de test élaborés lors des phases correspondantes de la branche descendante. Un constat d'échec à l'exécution d'un test implique donc un retour sur la phase de la branche descendante en vis-à-vis et des corrections sur le code, la conception ou les spécifications du système suivant le niveau de test où le problème a été détecté. La nécessité d'émettre des rapports de test informatifs est donc indispensable afin de pouvoir détecter et corriger le plus précisément possible l'origine de la divergence constatée.

Pour un cycle de développement classique en V, le test est considéré comme un processus d'assurance qualité (Ruparelia, 2010). L'idée est de contrôler la qualité d'un logiciel, le long de la branche ascendante du V, généralement en boîte noire et par des équipes dédiées. Le test est alors vu comme un moyen d'assurer au client le bon fonctionnement du logiciel. Les cycles de développement et de test sont alignés, toutefois l'exécution des tests ne sera mise en œuvre qu'une fois l'implémentation réalisée. Dans les méthodes agiles, nous verrons que ces deux cycles sont fusionnés.

4. Les tests dans une démarche agile

Les méthodes agiles respectent le manifeste agile (Beck *et al.*, 2001) signé en 2001 par 17 experts ou consultants en développement logiciel [5]. Ce manifeste énonce 4 valeurs fondamentales :

- les individus et leurs interactions plutôt que les processus et les outils,
- des logiciels opérationnels plutôt qu'une documentation exhaustive,
- la collaboration avec les clients plutôt que la négociation contractuelle,
- l'adaptation au changement plutôt que le suivi d'un plan.

Ces valeurs sont illustrées par 12 principes qui définissent une autre vision du développement logiciel. Ces principes favorisent en particulier l'implication de toutes les parties prenantes d'un projet et l'apport continu de la valeur métier par des livraisons fréquentes de versions operationnelles. Une méthode est dite « agile » si elle satisfait les valeurs et les principes de ce manifeste.

Après une présentation des méthodes agiles les plus utilisées aujourd'hui, nous explicitons les différents cycles de développement agiles à partir des tests, puis détaillons leur mise en œuvre et caractérisons les tests agiles.

4.1. Les méthodes agiles

Les méthodes agiles sont basées sur des cycles courts de développement. En favorisant des retours réguliers et fréquents (le *feedback*), les cycles sont itératifs. En se focalisant sur la valeur métier, les cycles deviennent aussi incrémentaux. Les méthodes agiles se nourissent d'un ensemble de bonnes pratiques qui interagissent les unes avec les autres. Les pratiques agiles peuvent être divisées en deux groupes complémentaires (Mancuso, 2015) : les pratiques orientées processus et les pratiques orientées techniques.

Les méthodes Scrum (Schwaber, Beedle, 2001), eXtreme Programming (Beck, 2000) et Kanban (Anderson, 2010) sont les trois méthodes agiles les plus utilisées en

5. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland et Dave Thomas

Europe et aux Etats-Unis en 2013 (Version One, 2013). Scrum et Kanban mettent en avant des pratiques orientées processus. Elles permettent d'organiser un développement aussi bien en termes de gestion d'équipe que de gestion de priorités vis-à-vis des besoins du client. Elles sont très souvent complétées par des pratiques orientées techniques issues d'eXtreme Programming.

La méthode Scrum a été adaptée au génie logiciel en 1995 par Ken Schwaber et Jeff Sutherland de manière empirique (Schwaber, 1995), à partir des travaux originaux d'Hirotaka Takeuchi et Ikujiro Nonaka (Takeuchi, Nonaka, 1986). Elle se déroule en sprints, des itérations de durée fixe pendant lesquelles aucun changement de spécification n'est autorisé et à la fin desquelles un incrément opérationel est livré. Cet incrément opérationnel a été choisi de manière à obtenir le meilleur rapport entre la valeur métier et l'effort de développement. La méthode Scrum est définie par trois rôles (l'équipe de développement, le responsable de produit nommé *product owner* et le garant de l'application de la méthode nommé *Scrum master*), trois cérémonies (la planification de sprint, la mêlée quotidienne et la revue de sprint) et trois artefacts (le *backlog* [6] du produit, le *backlog* de sprint et des indicateurs de progression d'un sprint). Les dernières évolutions de la méthode peuvent être consultées dans le guide Scrum (Schwaber, Sutherland, 2013).

La méthode Kanban (Anderson, 2010) est une méthode d'amélioration de processus de développement qui peut complémenter efficacement l'utilisation de la méthode Scrum (Kniberg, 2010). Kanban ne prescrit aucun rôle, ni événement, mais préconise cinq pratiques pour optimiser la valeur métier et le flux des travaux : visualiser le flux des travaux, limiter le nombre de travaux en cours, rendre le processus explicite, gérer le flux des travaux et identifier des opportunités d'amélioration.

L'eXtreme Programming, plus communément appelée XP, est une méthode agile destinée à des équipes de petite ou moyenne taille qui développent des logiciels dans un contexte où les besoins sont vagues et changent rapidement (Beck, 2000). XP propose une douzaine de pratiques relatives à la programmation, au fonctionnement interne de l'équipe, à la planification et aux relations avec le client. Chacune de ces pratiques pousse à l'*extrême* une bonne pratique de développement logiciel, à l'image de la programmation par binôme (*pair programming*) qui n'est autre qu'une revue de code pratiquée en continu.

4.2. Cycles de développement agiles à partir des tests

La méthode eXtreme Programming peut être considérée comme étant à l'origine de l'utilisation des tests comme instrument de conception (*test first*). Cependant, elle se focalise uniquement sur les tests unitaires et évoque simplement les tests d'acceptation. D'autres bonnes pratiques *test first* ont émergé de la communauté agile, telles que les tests d'acceptation *first*, la spécification par des scénarios comportementaux, la

---

6. liste des choses à faire

spécification par l'exemple. Toutes ces propositions confèrent aux tests un rôle central dans le développement logiciel. Nous appelons « tests agiles » les tests utilisés dans les approches *test first*. Les tests des méthodes agiles ne respectant pas ce dernier principe ne seront pas considérés par la suite car ils n'apportent aucun éclairage nouveau sur les tests logiciels.

Les approches *test first* s'appliquent à l'équipe de développement ou au client. Ces différents cycles partagent la même cinématique. Nous assemblons ces derniers dans un cycle de développement agile complet.

4.2.1. Le développement dirigé par les tests (TDD)

Le développement dirigé par les tests (*Test Driven Development* ou *TDD*) est une approche itérative et incrémentale de codage piloté par les tests unitaires. C'est un changement de paradigme en rupture avec les méthodes de programmation traditionnelles. Habituellement les tests sont écrits après la conception et le codage dans un but de vérification. Dans le TDD, les tests sont écrits juste avant le code ; le code écrit après les tests se doit de faire passer les tests. Accompagnée par des techniques de *refactoring* (Fowler, 1999) ou de *clean code* (Martin, 2008), la conception évolue au fil de l'eau pour satisfaire des critères de qualité. La conception induite par ce type de développement est dite « émergente ».

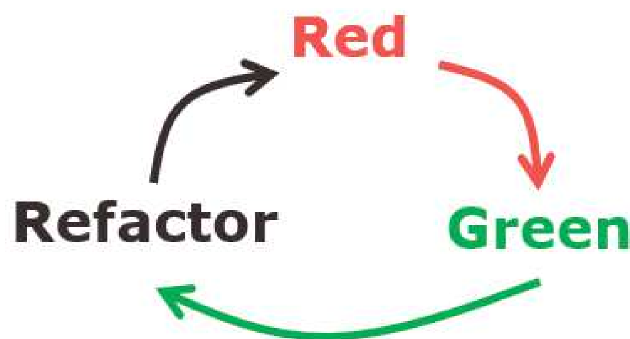

Figure 5. Le mantra du TDD

Plus qu'un cycle de développement, c'est une discipline de programmation que (Beck, 2002) a associé à un mantra présenté à la figure 5. Ce cycle se décompose en trois étapes : *Red-Green-Refactor* et ne doit pas excéder quelques minutes. La pratique est indissociable de la famille d'outils de test *xUnit*, à qui elle doit son vocabulaire : « barre verte » signifie que l'ensemble des tests unitaires accumulés passent avec succès et « barre rouge » signifie qu'au moins un test est en échec [7].

7. http://referentiel.institut-agile.fr/tdd.html

La première étape consiste à écrire un nouveau test unitaire et vérifier qu'il échoue en obtenant une barre rouge. Il est ainsi établi que le test apporte un nouveau comportement à implémenter au code de production.

La deuxième étape consiste à écrire au plus vite un code de production pour faire passer le test précédent ainsi que les tests antérieurs. Tous les moyens sont bons pour obtenir une barre verte, quitte à s'autoriser de mauvaises pratiques (duplication, patch, complexification du code, . . .).

L'objectif du TDD étant de produire rapidement un code opérationnel puis *propre*, un temps spécifique consacré au *refactoring* s'avère nécessaire dans la troisième étape. Un *refactoring* consiste à changer la structure interne d'un logiciel sans en changer son comportement observable (Fowler, 1999) ; l'ensemble des tests unitaires produits jusqu'à maintenant joue le rôle de tests de non-régression. Le guidage d'un *refactoring* n'est pas chose aisée : comment faire au plus simple en respectant des bonnes pratiques de conception ? Une solution consiste à exécuter cette phase en *pair-programming*, une revue de code permanente à deux. Une autre solution consiste à vérifier continuellement les principes SOLID (Martin, 2002) et à intégrer des patrons de conception (Gamma *et al.*, 1995).

Pour obtenir des boucles de *feedback* fréquentes, ces trois étapes doivent être rapides et répétées aussi souvent que nécessaire. Ces itérations par petits pas (*baby steps*) permettent d'accroître la confiance des développeurs en leur code. Le *refactoring* contribue à travers tous les cycles du TDD à l'obtention d'une conception simple et évolutive appelée conception émergente par opposition à une approche de conception classique planifiée (Fowler, 2001). Il est à noter que Kent Beck avait initialement employé l'expression *Test First Design* pour l'acronyme TDD.

4.2.2. Le développement dirigé par les tests d'acceptation (ATDD)

Le développement dirigé par les tests d'acceptation (*Acceptance Test Driven Development* ou *ATDD*) est une approche itérative et incrémentale centrée sur les exigences du client. Pour ce faire, une spécification collaborative avec le client, à base d'exemples transformés en tests automatisés dans des environnements dédiés (Fitnesse [8], Robot Framework [9], . . .), doit être orchestrée. Le cycle de développement ATDD présenté à la figure 6 se décompose en quatre étapes (Hendrickson, 2008) : Discussion, Distillation, Développement, Démonstration (*Discuss, Distill, Develop, Demo*).

Une fonctionnalité client, sous la forme d'une histoire utilisateur, est en entrée de la première étape. Une histoire utilisateur est un artefact du développement agile correspondant à un élément fonctionnel élémentaire qui a de la valeur pour le métier (Aubry, 2011). Il s'agit ici de raffiner cette histoire utilisateur entre le client et l'équipe de développement en identifiant de manière collégiale des exemples perti-

---

8. http://fitnesse.org/
9. http://robotframework.org/

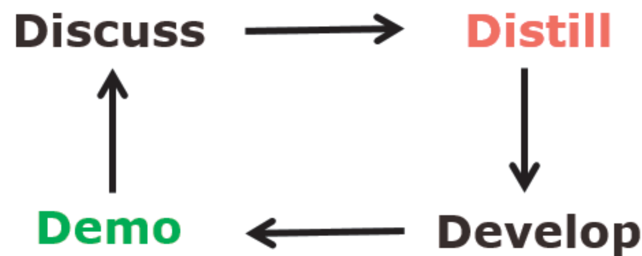

Figure 6. Cycle de l'ATDD

nents pouvant amener à des cas de test. Pour favoriser la communication, un langage simplifié du domaine est envisageable. Cette étape est également appelée atelier de spécifications (Adzic, 2011).

La deuxième étape consiste à distiller ou à transformer les exemples pertinents de l'étape précédente en tests d'acceptation automatisés pour fournir un cadre de travail aux développeurs, en particulier pour donner la condition nécessaire à la fin d'une histoire utilisateur. Chaque exemple est rédigé dans un format propre à l'équipe, adapté à la compréhension de tous et compatible avec l'environnement de test choisi.

La troisième étape consiste à développer la fonctionnalité spécifiée précédemment en étant guidé par les tests d'acceptation. Un cycle de développement agile en TDD peut être utilisé pour mener à bien cette phase de codage.

La quatrième étape consiste à organiser une démonstration des besoins élaborés à la première étape, une fois le développement terminé. L'équipe valide le comportement attendu en vérifiant que tous les tests d'acceptation passent au vert. De plus, elle doit procéder à des tests exploratoires sur le produit livré. Cette étape peut entrainer des ajustements sur les besoins et des changements dans les exigences.

A l'issue d'un cycle, l'ensemble des tests d'acceptation produits sont rejoués dans des environnements d'intégration continue et à ce titre garantissent une non-regression du comportement du logiciel. Si la documentation utilisateur est générée à partir des tests d'acceptation, on parle de documentation vivante (Adzic, 2011).

4.2.3. Vers un cycle de développement agile complet à partir des tests

Piloter un développement uniquement par les tests unitaires amène inévitablement à s'interroger sur l'adéquation aux besoins du client. Piloter un développement uniquement par les tests d'acceptation amène inévitablement à s'interroger sur la qualité du code produit. L'ATDD et le TDD sont donc complémentaires. Le TDD permet de construire correctement un produit ; l'ATDD permet de construire un produit correct.

L'ATDD a pour but de développer incrémentalement un produit logiciel, fonctionnalité par fonctionnalité. L'ajout d'un nouveau test d'acceptation représente l'ensemble des tests nécessaires pour se persuader que le développement de la fonction-

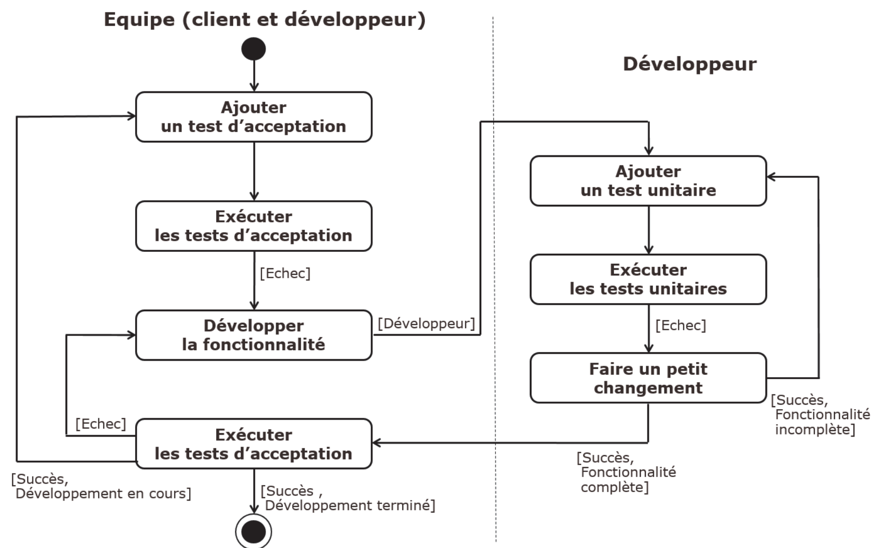

Figure 7. Complémentarité des cycles ATDD-TDD inspirée par Scott W. Ambler

nalité est terminé. Le TDD a pour but de développer incrémentalement une fonctionnalité, test unitaire par test unitaire pour que le développeur ait confiance en son code et produise un code de qualité. La figure 7 montre la complémentarité de ces deux cycles. L'ATDD donne alors un périmètre et une vision globale au développeur codant pas à pas une fonctionnalité (« Développer la fonctionnalité » vers « Ajouter un test unitaire ») et permet de mettre le terme « fini » sur une histoire utilisateur. Le terme « fini » dénote un code de qualité qui vérifie tous les besoins exprimés par le test d'acceptation (« Faire un petit changement » vers « Exécuter les tests d'acceptation »). Les deux cycles sont donc incrémentaux et complémentaires.

L'écriture d'exemples lors du cycle de l'ATDD conduit à l'émergence des exigences utilisateur. Si un test est avant tout un outil de confirmation au sens vérification et validation, un exemple est avant tout un outil de spécification. Un exemple favorise des échanges autour du comportement de l'application en confrontant la vision de chaque membre de l'équipe et vise à cerner et à documenter le périmètre des fonctionnalités à développer. L'utilisation de bons exemples spécifiés collaborativement, l'utilisation de langages dédiés pour une lisibilité « irréprochable » des tests et l'automatisation sont les grands principes de l'ATDD (Gartner, 2012).

4.3. Mise en œuvre des cycles de développement test first

Le *feedback* est une des cinq valeurs de l'eXtreme Programming. Il est au coeur de tout développement agile. Il intervient aussi bien sur le produit que sur la manière de travailler de l'équipe. Obtenir une boucle de *feedback* rapide est indispensable pour d'une part mettre en place au plus tôt une amélioration continue du processus et du

produit, et pour d'autre part s'adapter rapidement aux changements de spécification et de priorisation des développements.

Dans ces conditions, un *feedback* rapide nécessite une stratégie d'automatisation des tests. L'emploi des techniques de doublures de test permet de considérer les tests d'intégration et les tests système comme des tests unitaires « agiles ». Poussées à l'extrême, ces techniques induisent un nouveau style de TDD qui considère une conception a priori. Du côté client, les spécifications sont formalisées en tests d'acceptation automatisés.

4.3.1. Stratégie d'automatisation des tests

L'exécution des tests au plus tôt et le plus souvent possible permet aux développeurs d'apprendre sur leur code, de le modifier en toute confiance et de découvrir au plus tôt leurs erreurs. Le TDD met en place la plus petite boucle de *feedback* du développement agile : le delta entre la conception détaillée et l'implémentation est de l'ordre de quelques minutes. L'automatisation des tests est donc un pré-requis au développement agile. Mike Cohn illustre par une pyramide, représentée à la figure 8, la stratégie d'automatisation optimale dans un développement agile en l'état actuel des outils d'automatisation de test (Cohn, 2009).

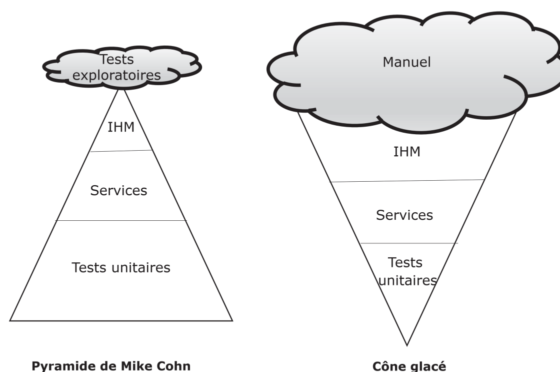

Figure 8. Stratégie d'automatisation des tests

Dans un développement où les processus de test sont peu ou prou intégrés, la pyramide est souvent inversée, ce qui représente un anti-pattern nommé cône glacé. Bien que la proportion de tests puisse être importante, leur répartition s'organise bien souvent comme suit : peu de tests unitaires, quelques tests intermédiaires à un niveau composant ou service, des tests fonctionnels automatisés via l'interface utilisateur et, surtout de nombreux tests manuels représentés par la glace sur le cornet. Le risque d'une telle stratégie est que les scénarii de tests manuels soient trop longs à être joués et que les tests de non-régression ne soient plus exécutés.

Le modèle de la pyramide, quant à lui, montre clairement la place prépondérante des tests unitaires dans l'automatisation. Avec le *refactoring*, ils permettent de se pré-

munir au plus tôt de simples problèmes de qualité de code qui peuvent causer à long terme de la dette technique, métaphore inspirée du contexte financier et employée initialement par (Cunningham, 1992). Des outils de test en continu comme Infinitest [10] permettent d'exécuter automatiquement les tests unitaires dès qu'un changement est détecté dans le code, réduisant au minimum la boucle de *feedback*.

Les tests de service, dans la partie intermédiaire de la pyramide, ne sont autres que des tests d'acceptation. Ils permettent de tester les services de l'application indépendamment de leur interface utilisateur afin de s'assurer le plus rapidement possible de l'adéquation du produit à ses spécifications.

Les tests d'interface utilisateur, au sommet de la pyramide, sont restreints en raison de leur mise en place et de leur maintenance très coûteuses. A ces tests automatisés sont ajoutés quelques tests manuels exploratoires chargés d'anticiper des situations imprévues. Ils prennent de plus en plus d'importance au sein de la communauté agile (Hendrickson, 2013).

4.3.2. Tests d'intégration et tests système vus comme des tests unitaires

Tout objet qui remplace un objet réel lors d'un test a été qualifié par le terme générique de doublure de tests (*test double*) (Meszaros, 2007). Jusque là, le terme de bouchon (*stub*) était le seul terme utilisé dans la littérature pour identifier une implémentation squelettique ou spéciale d'un composant logiciel déployée pour développer ou tester un composant qui l'utilise ou qui en est dépendant (IEEE, 1990).

Une doublure permet de reproduire l'état et/ou le comportement du composant dont dépend le code à tester. Meszaros propose cinq types de doublure de test comme présentés à la figure 9 : le fantôme (*dummy*), le bouchon (*stub*), l'espion (*spy*), l'objet factice (*mock*) et l'imposteur (*fake*). Le terme d'objet factice est apparu dans la communauté eXtreme Programming dans les années 2000 (Mackinnon *et al.*, 2001 ; Freeman *et al.*, 2004).

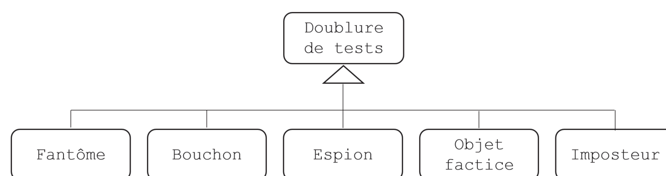

Figure 9. Classification des doublures de test

Le fantôme est la doublure de test la plus simple. Il ne contient aucune implémentation. Dans le code d'un test, il est utilisé comme paramètre d'appel à une méthode.

Le bouchon fait l'objet d'une implémentation minimale afin de fournir des réponses prédéfinies lorsque l'objet sous test a besoin de ses services.

10. https://infinitest.github.io/

L'espion est similaire au bouchon, mais il enregistre en plus les paramètres de ses appels pour un traitement ultérieur.

L'objet factice est un objet simulé dont le comportement est décrit spécifiquement pour un test unitaire dans un test unitaire. De plus, il a la capacité à vérifier la validité et l'enchaînement d'appels de méthode sur l'objet simulé par un langage d'expectations. Des outils dédiés, tels que Mockito [11] et EasyMock [12], permettent de les mettre en œuvre en fournissant une implémentation de substitution propre à chaque cas de test.

L'imposteur contient une implémentation alternative opérationnelle. Il est utilisé pour simplifier une dépendance, par exemple une base de données en mémoire au lieu d'une base de données réelle.

Les doublures permettent donc de vérifier le comportement d'un système sous test sans disposer de tous ses objets réels. Les *mocks* assurent de plus la vérification d'un protocole d'interactions de l'objet simulé avec les autres objets du système.

A l'aide de ces techniques, les tests d'intégration deviennent indépendants tout comme les tests unitaires et permettent de se focaliser sur un comportement attendu ou une interaction attendue. Les tests unitaires et d'intégration s'écrivent de la même façon avec les mêmes outils bien qu'ils abordent des points de vue différents sur le code en production.

4.3.3. TDD mockiste : doublures de test poussées à l'extrême

Dans une phase de conception classique, un grand nombre d'abstractions composant le logiciel sont imaginées et documentées en amont de l'écriture du code. Interrompre une conception avant la fin du codage ne permet plus d'intégrer ce que nous apprenons du code dans la conception et donc de l'améliorer pendant le développement restant. Dans une approche *test first*, les abstractions recherchées apparaissent au fur et à mesure dans le code. On parle alors de conception continue, évolutive ou émergente (Shore, 2004). La figure 10 présente les deux styles de TDD envisagés pour guider la conception, soit au fil de l'eau (TDD classique), soit par anticipation (TDD mockiste).

Le style de TDD dit classique est celui décrit initialement dans (Beck, 2002). Le développement s'organise depuis les modèles métiers jusqu'aux couches externes (interfaces, bases de données . . .) dans une approche appelée *middle-out*. Les objets sont découverts au fil de l'eau, au fur et à mesure des besoins. Les doublures peuvent être utilisées pour faciliter l'isolation d'un test lors de la découverte d'un nouvel objet non encore implémenté.

Le style de TDD dit mockiste est celui décrit dans (Freeman, Pryce, 2009). Il est centré sur une vision globale de l'architecture du système a priori. Le développement commence par la mise en place d'un squelette d'implémentation minimal (*walking*

11. http://mockito.org/
12. http://easymock.org/

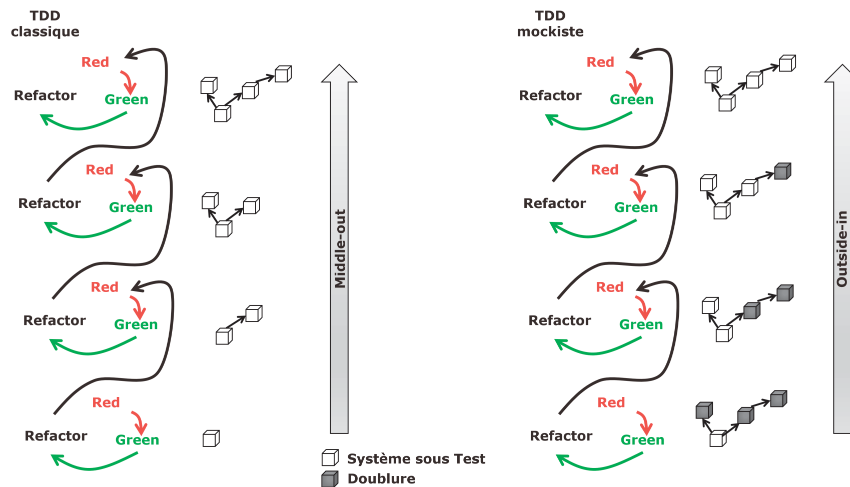

Figure 10. Les deux styles de TDD

skeleton) qui permet d'exécuter une fonction de bout en bout en reliant entre eux les principaux composants (Cockburn, 2004). Pour ce faire, des doublures sont systématiquement utilisées et les objets nécessaires au développement sont simulés avant d'être implémentés dans une approche appelée *outside-in*. Les *mocks* favorisent la découverte des interfaces au plus tôt et permettent un *feedback* plus rapide sur une architecture globale prédéfinie. Par rapport au style de TDD dit classique, la conception est anticipée.

Quel que soit le style de TDD choisi, des réunions de conception au tableau blanc (*quick design session*) sont programmées à la demande pour s'accorder sur des décisions de conception (Jeffries *et al.*, 2000). Ces choix de conception, cooptés à un instant donné, ne préjugent pas des architectures à venir.

4.3.4. Des exemples utilisateur aux tests automatisés

Trois formats d'exemples utilisateur ont émergé ces dernières années en fonction des entrées attendues par les robots de test : les tables, les mots-clés et les scénarios. Ces formats facilitent la formalisation des exemples utilisateur en tests automatisés, l'ajout de nouveaux cas de test et la communication entre le client et l'équipe de développement. Quel que ce soit le format choisi, les développeurs doivent ensuite écrire un code glue spécifique pour lier l'exemple en langage naturel et son implémentation en tant que test automatisé.

Un test d'acceptation peut être décrit dans une représentation tabulaire. FITnesse est une plateforme exécutable basée sur le framework FIT (Cunningham, Mugridge, 2005). C'est un serveur web qui propose un environnement d'édition wiki pour facili-

ter la communication. Le client peut lancer les tests lui-même et voir l'état d'avancement des fonctionnalités.

Un test d'acceptation peut être décrit à l'aide de données structurées par des mots-clés : on parle aussi de *data-driven* et *keyword-driven* (Laukkanen, 2006). Robot Framework est un *framework* de tests automatisés piloté par mots-clés.

Un test d'acceptation peut être décrit par des scénarios textuels. Les mots-clés *Given, When, Then* sont alors utilisés pour structurer ces exemples. De nombreux *frameworks*, adaptés à divers langages de programmation, permettent d'écrire des scénarios. Les plus connus sont JBehave [13], Cucumber [14] et SpecFlow [15].

L'approche par scénarios est actuellement la plus « médiatisée » et est connue sous le nom de *Behavior Driven Development* (BDD) ou développement dirigé par le comportement (North, 2006). Proposé par Dan North en 2003, le BDD fut initialement une proposition didactique d'amélioration du TDD pour faciliter son apprentissage. Il consistait à retrouver l'intention d'un test dans le nom de la méthode associée au test. En 2004, Dan North et Chris Matts étendent ce principe en reliant les tests formulés en termes de comportements attendus aux critères d'acceptation d'une histoire utilisateur. A cet effet, le pattern *Given, When, Then* a été proposé. Aujourd'hui, les protagonistes promeuvent le BDD au rang d'une méthodologie complète depuis les histoires utilisateur jusqu'au code en utilisant une approche TDD mockiste (Chelimsky *et al.*, 2010). En ce qui nous concerne, nous considérons que le BDD est une mise en œuvre complète du cycle de l'ATDD.

4.4. Nature des tests agiles

De par la manière *eXtrême* dont il est intégré au processus de développement, un test agile possède des propriétés spécifiques. Nous donnons une caractérisation de ces propriétés, puis énonçons trois définitions portant sur les tests agiles, les tests unitaires agiles et les tests d'acceptation agiles.

4.4.1. Caractéristiques d'un test agile

Selon Kent Beck, un test agile doit être isolé et automatisé pour que le développeur puisse travailler en toute confiance (Beck, 2000). L'isolation garantit qu'aucune interaction n'existe entre les tests et évite tout scénario d'échec de tests en cascade. L'automatisation fournit un verdict du test sans équivoque quant au fonctionnement du système.

Dans une approche *test first*, le test est écrit avant le code de production. Les tests permettent de se substituer en partie à une spécification. Ils permettent également de modifier et de faire évoluer le code de production en toute confiance. Le code de test

13. http://jbehave.org/
14. https://cukes.info/
15. http://www.specflow.org/

devient alors aussi important que le code de production. En devenant un citoyen de première classe, un test agile se doit donc de respecter un niveau de qualité équivalent à celui du code de production : l'expression test propre (*clean test*) est d'ailleurs employée en écho à l'expression code propre (*clean code*) (Martin, 2008).

Pour formater et structurer au mieux l'écriture d'un test agile, le patron *Arrange-Act-Assert* (patron *AAA*) est préconisé (Beck, 2002). Ce patron décompose le test en trois étapes distinctes : l'initialisation de l'acteur sous test, l'exécution de l'action à tester et l'assertion sur la réaction de l'acteur. Les étapes d'exécution et d'assertion sont propres à chaque test. L'étape d'initialisation peut être commune à plusieurs tests. Cette bonne pratique utilisée par la communauté des testeurs a été remise au goût du jour par la communauté Agile.

L'acronyme FIRST introduit des propriétés relatives aux tests et aux suites de tests agiles (Martin, 2008). Le test doit être rapide (***F****ast*) pour être fréquemment activé. Le test doit être indépendant (***I****ndependent*) pour être exécuté de manière isolée des autres tests. Le test doit être reproductible (***R****epeatable*) pour être exécuté dans n'importe quel environnement. Le test doit être auto-validant (***S****elf-Validating*) pour obtenir rapidement un verdict présenté en termes d'assertions. Le test doit être écrit au moment opportun (***T****imely*) pour respecter l'approche *test first*.

Les propriétés FIRST et le patron AAA caractérisent au mieux les tests agiles. L'acronyme FIRST met en exergue des principes énoncés par le manifeste du test automatisé (Meszaros *et al.*, 2003).

4.4.2. Tests unitaires et tests d'accceptation agiles

Selon l'eXtreme Programming, les tests émanent de deux acteurs : le développeur et le client.

Les tests relatifs au développeur sont qualifiés de tests unitaires. Ce sont des tests techniques (Crispin, Gregory, 2009). La notion de test unitaire en agilité, bien que légèrement différente de la définition classique, n'est pas clairement énoncée dans la littérature. Nous retiendrons la définition proposée par Meszaros (Meszaros, 2007) qui indique qu'un test unitaire est un test qui vérifie le comportement d'une partie restreinte du système. Cette définition rejoint celle de Kent Beck qui considère que le test unitaire est un test à petite échelle (Beck, 2002).

Cependant lorsque l'architecture logicielle se complexifie, les tests unitaires ne sont plus suffisants, des tests d'intégration sont nécessaires pour vérifier les interactions entre différents composants. Pour respecter l'approche *test first*, et notamment la rapidité (***F****ast*), l'isolation (***I****ndependant*), et la répétition des tests (***R****epeatable*), la communauté agile a proposé diverses techniques, dont les doublures, qui permettent aux tests d'intégration d'être joués comme des tests unitaires (Freeman *et al.*, 2004).

Avant de donner la définition d'un test unitaire agile, précisons tout d'abord ce que nous entendons par test agile.

DÉFINITION 20. — *Un test agile est un test utilisé dans une approche test first.*

DÉFINITION 21. — *Un test unitaire agile est un test spécifique à une micro ou macro-fonctionnalité du système sous test respectant les propriétés de l'acronyme* FIRST.

Les tests relatifs au client sont qualifiés de tests d'acceptation. Ce sont des tests métiers (Crispin, Gregory, 2009). Les tests d'acceptation sont définis comme un moyen de valider le comportement du système conformément aux histoires utilisateur introduites par le client (Cohn, 2004).

DÉFINITION 22. — *Un test d'acceptation agile est un ensemble de tests couvrant une fonctionnalité du système conformément à une histoire utilisateur.*

Les tests unitaires et les tests d'acceptation sont donc les deux niveaux de test agile.

5. Vers une nouvelle vision des tests

Dans la suite de cet article, nous mettons l'accent sur la principale différence entre les tests du cycle en V et les tests des méthodes agiles : la place des tests dans les processus de développement. Nous appelons *test last*, les tests des méthodes classiques ou agiles où le test est uniquement utilisé comme méthode de vérification. Nous appelons *test first*, les tests des méthodes agiles où le test est principalement utilisé pour guider le développement. Cette différence de point de vue sur l'utilisation des tests implique des différences sur la nature des tests que nous illustrons sur un exemple. Poussée à l'extrême dans un développement agile, l'automatisation des tests couplée à l'intégration continue permet de corriger les défauts au plus tôt. Enfin, les valeurs agiles ont un impact sur les rôles joués par les développeurs et les testeurs dans les équipes de développement.

5.1. Sur la place des tests et la qualité

5.1.1. Rôle et place des tests dans le processus

Le développement classique et le développement agile offrent une vision différente du rôle et de la place des tests dans un processus de développement. Le tableau 2 propose un comparatif entre ces deux approches.

Tableau 2. Les tests dans un développement classique et agile

Développement classique (*Test Last*)	**Développement agile (*Test First*)**
Planifier les campagnes de tests	Tester en continu
Détecter le maximum d'erreurs a posteriori (être sûr de son code)	Détecter des erreurs au plut tôt (avoir confiance en son code)
Vérifier et Valider	Guider le développement
-	Spécifier incrémentalement
Assigner des responsabilités par métier	Partager la responsabilité du code

Dans un cycle en V, les tests sont exécutés dans la branche ascendante, après la phase de développement, et suivant une stratégie de test planifiée durant la phase descendante. Les tests sont implémentés et exécutés après la production de code. Dans une approche agile, l'activité de test est réalisée dès que cela s'avère nécessaire pour alimenter un *feedback* le plus rapide possible. L'activité de test est donc répartie tout au long du développement sans pour autant réduire la part des tests (Kettunen *et al.*, 2010). De cette stratégie de test en continu découle les propriétés suivantes : détecter des erreurs au plut tôt, guider le développement et spécifier incrémentalement.

La détection d'erreurs est engagée à chaque nouveau comportement ajouté, qui doit pour autant satisfaire les fonctionnalités antérieures. Les erreurs sont ainsi détectées au plus tôt, ce qui conduit à anticiper les phases de débogage du logiciel. Pour le développeur, il s'établit en outre une confiance en son développement. Loin de l'exhaustivité des cas de test élaborés lors de la branche descendante du V, les développeurs agiles écrivent juste assez de tests pour avoir confiance en leur code. A l'inverse, les différentes phases de test du cycle en V visent à détecter un maximum d'erreurs a posteriori et à obtenir un logiciel fiable. Les méthodes agiles privilégient la confiance en tant qu'indicateur de la santé d'un code à tout critère de décision technique comme la fiabilité.

Dans un développement classique, le test est essentiellement un support de vérification qui vise à détecter des erreurs et à s'assurer que ce qui a été développé correspond bien à ce qui a été spécifié. Dans une approche *test first*, l'écriture d'un test est un acte de conception externe puisqu'il est écrit dans un langage exécutable avant le code de production. La phase de *refactoring* est un acte de conception interne qui simplifie de suite l'implémentation produite à partir du test. Le test contribue ainsi à améliorer la qualité du code de manière continue. Dans un développement classique et agile, les tests d'acceptation sont un support de validation. Les méthodes agiles préconisent cependant d'écrire les tests d'acceptation a priori, ce qui favorise la compréhension des besoins et une possible validation continue. Des ateliers collaboratifs sont dédiés à l'écriture de tests d'acceptation sous forme d'exemples.

L'approche *test first* liée à l'automatisation des tests offre intrinsèquement un mécanisme de validation totalement intégré au développement. On peut ainsi spécifier et valider incrémentalement un comportement. Cette incrémentalité évite une dispersion des solutions envisagées et cadre les fonctionnalités attendues au fur et à mesure des besoins exprimés. L'enchainement des tests unitaires fournit une explication sur la logique du code et permet d'obtenir une traçabilité de la conception qui facilite la maintenabilité. Le code a ainsi une logique intrinsèque expliquée par la suite des tests unitaires.

Dans un développement classique, chaque acteur a un couloir propre de responsabilité et de compétence. Le test est alors une activité indépendante de la conception et de l'implémentation ; seuls les testeurs sont responsables des tests et considérés comme les garants de la vérification et de la validation du logiciel. Ce cloisonnement permet une indépendance entre développeurs et testeurs et une double validation de ce qui est produit, ce qui peut constituer un avantage. Un développement agile, quant

à lui, se construit autour d'une équipe pluri-disciplinaire. Les codes métier et de test sont partagés par l'ensemble de l'équipe, ce qui responsabilise et favorise l'aspect collaboratif du développement.

Rédiger un test préalablement au code revient à définir un contrat que le code doit remplir. L'approche *test first* assimile ainsi la post-condition d'un contrat à un diagnostic portant sur le déroulement d'un comportement. Le test agile couvre alors trois rôles : vérification et validation, spécification et aide à la conception. De plus, il accroît la confiance des développeurs dans leur code. Les activités agiles tendent ainsi à accorder plus de temps aux activités de test, sans toutefois diminuer le coût de développement du logiciel (Kettunen *et al.*, 2010).

5.1.2. Impact sur la qualité

De 2005 à 2013, 27 études comparatives ont été menées visant à montrer l'impact du TDD sur la qualité externe du produit et la productivité du développeur dans des milieux industriels et académiques (Rafique, Misic, 2013). Dans le cadre académique, l'étude montre que la qualité augmente lorsqu'un effort de test est demandé. Dans le cadre industriel, une baisse de productivité est constatée lorsque la différence d'effort portant sur les tests entre le TDD et le contrôle a posteriori des approches *test last* est significative. Cependant la qualité de ces derniers projets n'ayant pu être comparée faute de données, ont-ils réellement gagné du temps ?

En ce qui concerne la qualité interne, une étude montre que l'impact du TDD sur la conception reste discutable (Siniaalto, Abrahamsson, 2007). Dans une approche classique, la conception établie a priori est un avantage, il existe une architecture définie pour la production du code. Dans une approche agile, l'encadrement de la conception dite émergente demeure un sujet d'étude, bien que déjà abordé dans le style TDD mockiste (Freeman, Pryce, 2009).

5.2. Différence sur la nature des tests : un exemple

Pour illustrer la différence de nature des tests et du code qui en découle, nous avons choisi un petit exercice d'initiation à la pratique du TDD qui consiste à afficher le score d'un jeu d'une partie de tennis : le Kata Tennis. Cet exercice est extrait de (Bache, 2013). Emily Bache propose, pour ce kata, plusieurs exemples de code de production écrits dans une logique *test last* suivant le niveau des développeurs. Le listing 1 présente un des codes écrits par un développeur Java voulant prouver son expertise.

Listing 1 – Classe `TennisGame` en *test last*

```
public class TennisGame {

    private int p2, p1;
    private String p1N, p2N;

    public TennisGame(String p1N, String p2N) {
        this.p1N = p1N; this.p2N = p2N;
    }
```

```java
    public String getScore() {
        String s;
        if (p1 < 4 && p2 < 4 && !(p1 + p2 == 6)) {
            String[] p = new String[]{"love", "fifteen", "thirty", "forty"};
            s = p[p1];
            return s + " - " + p[p2];
        } else {
            if (p1 == p2)
                return "deuce";
            s = p1 > p2 ? p1N : p2N;
            return ((p1-p2)*(p1-p2) == 1) ? "advantage " + s : "game for " + s;
        }
    }

    public void wonPoint(String playerName) {
        if (playerName == "player1") this.p1 += 1;
        else this.p2 += 1;
    }
}
```

Ce code respecte à première vue certains critères de lisibilité : code concis, correctement indenté, moins de dix lignes par méthode. Cependant, l'utilisation de mauvaises pratiques de conception (*code smells* [16]) ne permet pas de révéler l'intention du code :

– des expressions conditionnelles complexes : utilisation de ternaires, conditions composées, élévation au carré au lieu de valeur absolue...

– une variable temporaire `s` de type `String` avec deux sémantiques différentes au sein de la même méthode,

– un nommage de variables non explicite,

– trop de nombres magiques dans une méthode algorithmique,

– la déclaration d'un tableau de constantes à la volée dans le code.

Le listing 2 présente un ensemble de tests boîte noire, écrits après le code, destinés à valider les résultats de la méthode `getScore`. Bien que le code de la classe `TennisGame` passe tous ses tests, il demeure fragile puisqu'une mauvaise instanciation d'un développeur client peut faire échouer tous les tests. Dans le cas de tests boîte blanche, ce code n'étant pas lisible et n'inspirant pas confiance, un testeur aura tendance à accroître le nombre de cas de tests.

Listing 2 – *Tests last* sur la classe `TennisGame`

```java
@RunWith(Parameterized.class)
public class TennisGameTest {

        //...

    @Parameters
    public static Collection<Object[]> getAllScores() {
        return Arrays.asList(new Object[][] {
                { 0, 0, "love - love" },
                { 1, 1, "fifteen - fifteen" },
```

16. mauvaises odeurs dans le code

```
            { 2, 2, "thirty - thirty"},
            { 3, 3, "deuce"},
            { 4, 4, "deuce"},

            { 1, 0, "fifteen - love"},
            { 0, 1, "love - fifteen"},
            { 2, 0, "thirty - love"},
            { 0, 2, "love - thirty"},
            { 3, 0, "forty - love"},
            { 0, 3, "love - forty"},
            { 4, 0, "game for player1"},
            { 0, 4, "game for player2"},

            { 2, 1, "thirty - fifteen"},
            { 1, 2, "fifteen - thirty"},
            { 3, 1, "forty - fifteen"},
            { 1, 3, "fifteen - forty"},
            { 4, 1, "game for player1"},
            { 1, 4, "game for player2"},

            { 3, 2, "forty - thirty"},
            { 2, 3, "thirty - forty"},
            { 4, 2, "game for player1"},
            { 2, 4, "game for player2"},

            { 4, 3, "advantage player1"},
            { 3, 4, "advantage player2"},
            { 5, 4, "advantage player1"},
            { 4, 5, "advantage player2"},
            { 15, 14, "advantage player1"},
            { 14, 15, "advantage player2"},

            { 6, 4, "game for player1"},
            { 4, 6, "game for player2"},
            { 16, 14, "game for player1"},
            { 14, 16, "game for player2"},
        });
    }

    @Test
    public void checkAllScoresTennisGame() {
        TennisGame game = new TennisGame("player1","player2");
        for (int i = 0; i < this.player1Score; i++)
           game.wonPoint("player1");
        for (int i = 0; i < this.player2Score; i++)
           game.wonPoint("player2");
        assertEquals(this.expectedScore, game.getScore());
        }
    }
```

Dans cet exemple, la couverture des instructions (*statement coverage*), la couverture des points de tests (*condition coverage*) et la couverture des chemins d'exécution (*path coverage*) sont à 100%. Ces tests sont formatés comme un plan de tests, la méthode `checkAllScoresTennisGame` jouant le rôle de code glue. Les tests paramétrés permettent d'écrire les tests de manière tabulaire comme cela est proposé dans les *frameworks* dédiés aux tests fonctionnels.

Le listing 3 présente un ensemble de tests écrits en *test first* et destinés à guider le développement de la méthode `getScore`. Chaque test est explicite ; il est nommé par son intention : la description d'un nouveau comportement à implémenter par la mise en œuvre d'un possible scénario d'exécution.

Listing 3 – *Tests first* sur la classe `TennisGame`

```
public class TennisGameTest {

    private TennisGame partie;

    @Before
    public void setUp() throws Exception {
        this.partie = new TennisGame();
    }

    @After
    public void tearDown() throws Exception {
        this.partie = null;
    }

    @Test
    public void testNewGameStart() {
        assertEquals("love - love",partie.currentScore());
    }

    @Test
    public void testPlayer1ScoresOnePoint() {
        reachThisScore(1,0);
        assertEquals("fifteen - love",partie.currentScore());
    }

    @Test
    public void testPlayer2ScoresOnePoint() {
        reachThisScore(0,1);
        assertEquals("love - fifteen",partie.currentScore());
    }

    @Test
    public void testPlayer1ScoresOnePointAndPlayer2ScoresOnePoint() {
        reachThisScore(1,1);
        assertEquals("fifteen - fifteen",partie.currentScore());
    }

    @Test
    public void testPlayer1ScoresFourPoints() {
        reachThisScore(4,0);
        assertEquals("game for player1",partie.currentScore());
    }

    @Test
    public void testPlayer2ScoresFourPoints() {
        reachThisScore(0,4);
        assertEquals("game for player2",partie.currentScore());
    }

    @Test
    public void testPlayersAreDeuce() {
        reachThisScore(4,4);
        assertEquals("deuce",partie.currentScore());
    }

    @Test
    public void testPlayer1Advantage() {
        reachThisScore(5,4);
        assertEquals("advantage player1",partie.currentScore());
    }

    @Test
    public void testPlayer2Advantage() {
        reachThisScore(4,5);
        assertEquals("advantage player2",partie.currentScore());
```

```
    }

    @Test
    public void testPlayer1WinsAfterAdvantage() {
       reachThisScore(6,4);
       assertEquals("game for player1",partie.currentScore());
     }

    @Test
    public void testPlayer2WinsAfterAdvantage() {
       reachThisScore(4,6);
       assertEquals("game for player2",partie.currentScore());
     }

    private void reachThisScore(int pointsPlayer1, int pointsPlayer2){
       for (int i = 0; i < pointsPlayer1; i++)
          partie.player1Scores();
       for (int i = 0; i < pointsPlayer2; i++)
          partie.player2Scores();
     }
}
```

L'ordonnancement des tests suit une décomposition du développement en trois parties : la gestion d'un début de jeu (les quatre premiers tests), la gestion d'un jeu gagné rapidement (les deux tests suivants), et la gestion d'un jeu gagné plus difficilement après une suite d'égalités et/ou d'avantages (les derniers tests). La couverture des instructions est à 100%, ce qui n'est pas le cas de la couverture des points de tests et de la couverture des chemins d'exécution.

Avec un nombre minimal de tests pour guider le développement, la couverture des instructions atteint 100% puisque le code écrit en TDD sert à valider les tests. En ajoutant un test (`testPlayer1ScoresThreePoints`), la couverture des points de tests atteint elle aussi 100%. Pour assurer une couverture complète des chemins d'exécution, tous les cas de tests du plan de test du listing *test last* sont à prendre à compte.

Les listings 4, 5 et 6 présentent le code obtenu à chaque incrément après *refactoring*. Les tests de début de jeu permettent l'émergence de la variable `tabScores` (listing 4), ceux d'un jeu gagné rapidement l'émergence des méthodes `player1Won` et `player2Won` (listing 5), et ceux d'un jeu gagné plus difficilement l'émergence des méthodes `deuceOrAdvantage` et `gameThatLastsALongTime` (listing 6). L'ensemble des tests constitue une spécification, qui, écrite de manière incrémentale, guide la conception pas à pas. Les tests fournissent aussi une sorte d'historique expliquant la conception finale de la classe qu'un développeur pourra réutiliser a posteriori.

Listing 4 – Classe `TennisGame` en *test first* : début de jeu

```
public class TennisGame {

    private int scorePlayer1;
    private int scorePlayer2;

    public static final String tabScores[]={"love","fifteen", "thirty", "forty"};

    public TennisGame() {
       this.scorePlayer1 = 0;
       this.scorePlayer2 = 0;
```

```
    }

    public String currentScore() {
       return tabScores[this.scorePlayer1] +" - "+ tabScores[this.scorePlayer2];
    }

    public void player1Scores() {
       this.scorePlayer1+=1;
    }

    public void player2Scores() {
       this.scorePlayer2+=1;
    }
}
```

Listing 5 – Classe `TennisGame` en *test first* : jeu gagné rapidement

```
public class TennisGame {

    //...

    public String currentScore() {
       if (this.player1Won()) return "game for player1";
       if (this.player2Won()) return "game for player2";
       return tabScores[this.scorePlayer1] +" - "+ tabScores[this.scorePlayer2];
    }

    private boolean player1Won() {
       return playerWon(this.scorePlayer1,this.scorePlayer2);
    }

    private boolean player2Won() {
       return playerWon(this.scorePlayer2,this.scorePlayer1);
    }

    private boolean playerWon(int score1, int score2) {
       return (score1 >= 4 && score1 - score2 >= 2);
    }

   //...
}
```

Listing 6 – Classe `TennisGame` en *test first* : jeu gagné plus difficilement après une suite d'égalités et/ou d'avantages

```
public class TennisGame {

   //...

    public String currentScore() {
       if (this.player1Won()) return "game for player1";
       if (this.player2Won()) return "game for player2";
       if (this.gameThatLastsALongTime()) return this.deuceOrAdvantage();
       return tabScores[this.scorePlayer1] +" - "+ tabScores[this.scorePlayer2];
    }

    private String deuceOrAdvantage() {
       if (this.scorePlayer1 == this.scorePlayer2) return "deuce";
       if (this.scorePlayer1 > this.scorePlayer2) return ("advantage player1");
       return ("advantage player2");
    }

    private boolean gameThatLastsALongTime() {
       return (this.scorePlayer1 >=3 && this.scorePlayer2 >=3);
```

```
    }

    //...
}
```

Malgré une couverture de code moindre, l'ensemble des tests agiles correspondant à ce développement accroît le degré de compréhension du problème et la confiance du développeur en son code. Cependant, rien n'empêche celui-ci de compléter ses tests par des tests issus d'un plan de tests en *test last*.

En ce qui concerne les projets réels, le passage à l'échelle n'est pas trivial, ce qui explique la polémique *Is TDD Dead ?* [17]. Dans ce qui suit, nous synthétisons les points que nous avons jugés pertinents :

1. Le TDD ne garantit en aucune manière une bonne conception (Siniaalto, Abrahamsson, 2007). Comme dans toute technique, l'expérience reste indispensable.

2. La conception émergente ne résout pas toutes les catégories de problèmes. L'idée de la solution ou l'idée de l'architecture est un prérequis dans bien des cas, à l'image de l'exemple du kata diamant [18].

3. Dans le cas des tests en isolation avec l'utilisation conjointe d'objets mockés, le risque est d'avoir trop de petites classes et trop de couplage entre ces classes.

4. Le code des tests devient plus important en volume que le code de production (risque d'*overtesting*).

5. Certains *frameworks* de tests priment sur les environnements de développement et conditionnent trop fortement la conception.

6. La construction incrémentale est un exercice difficile et pas seulement en TDD. Un incrément de spécification (un nouveau test unitaire) peut être facile à écrire tout en étant difficile à concevoir.

5.3. Sur l'automatisation des tests et l'intégration continue

Les deux premiers items du tableau 2, « tester en continu » et « détecter des erreurs au plut tôt », impliquent un effort particulier sur l'automatisation des tests permettant une accélération du *feedback* sur la conception et le code, une des clefs de la réussite d'un projet agile. Ces tests automatisés confortent un développeur sur la sûreté de son code. Avec l'intégration continue, ils assurent un *feedback* global sur le projet et rendent possible un projet livrable à tout instant du développement.

5.3.1. Tests automatisés

Répondre au changement rapidement nécessite alors de disposer d'une batterie de tests automatisés afin de garantir en temps réel la non-régression du comportement attendu du produit.

---

17. http://martinfowler.com/articles/is-tdd-dead
18. http://natpryce.com/articles/000807.html

Si l'automatisation des tests unitaires fournit un *feedback* rapide à un développeur, l'automatisation des tests d'intégration fournit un *feedback* rapide à l'ensemble de l'équipe de développement. Le rejeu automatique de ces tests associé à un dépôt commun du code conduit à une pratique nommée intégration continue (Duvall *et al.*, 2007).

5.3.2. *Intégration continue*

Utilisée aussi bien dans les développements classiques qu'agiles, l'intégration continue a été initialement introduite comme une pratique de l'eXtreme Programming. Elle est décrite par (Fowler, 2006) comme une pratique de développement logiciel où chaque membre de l'équipe intègre au moins quotidiennement son travail, conduisant à de multiples intégrations journalières. Chacune de ces intégrations est vérifiée par un système automatisé de construction, y compris de tests, permettant de détecter au plus tôt des erreurs d'intégration.

Dès qu'une tâche de développement est terminée, les modifications ou ajouts de code sont immédiatement intégrés dans l'application via un environnement d'intégration permettant de compiler, tester et déployer automatiquement. Un des effets bénéfiques est de disposer à tout moment d'un logiciel opérationnel au plus près de la version en cours de développement et facilement installable.

Au delà de l'intégration continue se mettent en place de nouveaux environnements de développement permettant d'implémenter le concept d'usine logicielle (Cusumano, 1991). Ces environnements s'appuient sur du test continu, de l'intégration continue, de l'inspection continue et du déploiement continu (Humble, Farley, 2010).

5.4. Sur l'évolution du métier de testeur et développeur

Mettre les tests au cœur du développement agile a de fait un impact sur l'évolution des métiers. Auparavant isolés les uns des autres, les testeurs et les développeurs se doivent désormais de collaborer. Leurs savoir-être et leurs savoir-faire devront s'adapter à ce nouveau contexte.

5.4.1. *Sur l'évolution du métier de testeur*

La mission du testeur classique est de garantir la qualité d'un logiciel et de ses fonctionnalités grâce à des campagnes et des plans de test. Plus précisément, il prépare les campagnes de test en amont à partir des spécifications, il s'assure de la bonne exécution des tests et donne un verdict en fin de cycle sur la qualité du code et la qualité du produit.

La mission du testeur agile inclut les activités précédentes et de plus est amené à intervenir tout au long du processus. Cette évolution touche particulièrement le savoir-être du testeur, puisque ce dernier collabore à la fois avec le métier et avec les développeurs.

Tableau 3. Compétences communes aux testeurs classique et agile

Testeur classique et testeur agile
Élaborer et mettre en place les outils de test
Réceptionner les environnements de test
Assurer une utilisation correcte des outils de test
Créer les données de test et écrire les procédures de test
Configurer, utiliser, et gérer les environnements et données de test
Exécuter les cas de test soit manuellement, soit avec des outils
Reporter les défauts et travailler avec l'équipe pour les résoudre

Les tableaux 3, 4 et 5 listent, à partir de (ISTQB, 2014) et (Orientation pour tous, 2015), les compétences communes et spécifiques d'un testeur qu'il soit acteur d'un développement classique ou agile.

Tableau 4. Compétences spécifiques au testeur classique

Testeur classique
Analyser et évaluer les exigences utilisateur
Rédiger des plans de test et des scripts de test
Consigner les résultats de l'exécution des tests
Participer à l'écriture du rapport de synthèse
Suivre les anomalies
Proposer et mettre en œuvre les actions qualité, préventives ou correctives

Comme nous pouvons le percevoir à travers les tableaux 4 et 5, le rôle du testeur classique se focalise sur le travail de l'équipe de développement, ce qui pourrait laisser à penser qu'il porte un jugement sur la qualité du travail des autres membres du projet. A contrario, le testeur agile participe de façon continue à l'amélioration du développement et partage son savoir-faire.

Tableau 5. Compétences spécifiques au testeur agile

Testeur agile
Collaborer activement avec les clients pour clarifier les exigences
S'assurer que les tâches de test appropriées soient planifiées
Mesurer et reporter la couverture de test
Coacher les développeurs sur des aspects pertinents du test
Participer aux rétrospectives, suggérer et implémenter des améliorations

Ces deux postures différentes induisent une motivation différente quant au métier exercé (Deak, 2014). La motivation est un facteur important de la productivité, de la qualité et de la livraison réussie d'un projet. Cette étude examine de manière qualitative, les différents facteurs de motivation et de démotivation des testeurs tels que

l'appréciation des défis, l'amélioration de la qualité, la variété du travail, la pression du temps, les relations avec les développeurs et la complexité des tâches techniques. L'étude a révélé que les testeurs classiques se sentent plus soumis au stress car ils donnent un verdict en fin de cycle, mais apprécient les défis offerts par les activités de test. Les testeurs agiles, quant à eux, se sentent mieux intégrés dans leur équipe bien que devant résoudre plus de problèmes de communication avec les développeurs.

5.4.2. Sur l'évolution du métier de développeur

L'évolution du métier de développeur se traduit par le mouvement de l'artisanat logiciel connu sous le nom de *Software Craftsmanship* (Mancuso, 2015). En mettant l'accent sur l'excellence technique, l'apprentissage et le savoir-être, l'artisanat logiciel peut être considéré comme une extension du manifeste agile. Un artisan-développeur adhère aux quatre valeurs suivantes [19] :

– Pas seulement des logiciels opérationnels, mais aussi des logiciels bien conçus.

– Pas seulement l'adaptation aux changements, mais aussi l'ajout constant de valeurs.

– Pas seulement les individus et leurs interactions, mais aussi une communauté de professionnels.

– Pas seulement la collaboration avec les clients, mais aussi des partenariats productifs.

Les prémices du mouvement *Software Craftsmanship* ont été énoncées dès 1999 (Hunt, Thomas, 1999 ; McBreen, 2002 ; Martin, 2008) et le manifeste quant à lui fut rédigé en 2009. Il recentre les méthodes agiles sur le métier de développeur et permet une coupure moins nette entre artisanat logiciel et ingénierie logicielle. Au-delà du manifeste agile, il propose une vision plus étendue et à plus long terme, en particulier en envisageant le savoir-faire des développeurs dans un objectif plus large que celui de mener à bien un projet.

Les méthodes agiles ont permis de faire évoluer les métiers de testeur et de développeur. Il en est de même actuellement pour les administrateurs système qui, au travers du mouvement Devops, prônent le rapprochement des équipes de développement informatique et d'exploitation dans le but d'améliorer la réactivité des DSI et de diminuer la durée comprise entre la demande de la modification d'un service IT et sa mise en ligne [20].

5.5. Les valeurs agiles au service des tests

La vision sur les métiers de testeur et développeur ayant évolué, il en résulte une nouvelle vision sur les tests. Nous déclinons dans le tableau 6 les apports des valeurs

19. http://manifesto.softwarecraftsmanship.org/
20. http://www.marte.fr/livres-blancs/la-revolution-devops/

du manifeste agile à l'activité de test. Il reprend les quatre valeurs du manifeste et met en regard les caractéristiques que devraient posséder les tests aujourd'hui.

Tableau 6. Les valeurs agiles au service des tests

Manifeste agile	Nouvelle vision sur le test
Les individus et leurs interactions plutôt que les processus et les outils	Un testeur intégré à l'équipe plutôt qu'un testeur dans une équipe dédiée Un développeur qui code avec un testeur plutôt que d'attendre le verdict du testeur
Des logiciels opérationnels plutôt qu'une documentation exhaustive	Des tests automatisés plutôt que la rédaction de rapports d'erreurs Des tests en continu pour accroître la confiance plutôt que rédiger des documents à chaque jalon
La collaboration avec les clients plutôt que la négociation contractuelle	Des scénarios de test avec le client plutôt qu'un cahier des charges Des résultats de test à la demande plutôt que des indicateurs d'avancement
L'adaptation au changement plutôt que le suivi d'un plan	Des tests qui évoluent plutôt que des analyses d'impact

Au travers de notre panorama sur les tests dans le développement logiciel, nous avons constaté que les tests mis en place dans les cycles en V étaient une bonne pratique. Cette bonne pratique a été poussée de manière extrême par la communauté agile qui à son tour offre une nouvelle vision hyperproductive sur les tests. De ce fait, la vision énoncée dans le tableau 6 peut s'appliquer à tout type de développement.

> *If testing is good, everybody will test all the time (unit testing), even the customers (functional testing).* [21] (Beck, 2000).

> *If integration testing is important, then we'll integrate and test several times a day (continuous integration).* [22] (Beck, 2000).

6. Conclusion

Les processus agiles et classiques partagent des activités dédiées aux tests. Le test étant l'outil dominant dans un développement pour contrôler la qualité du produit

21. Si tester est bien, tout le monde devrait tester tout le temps (test unitaire), même les clients (test fonctionnel).
22. Si l'intégration est importante, nous devrions intégrer et tester plusieurs fois par jour (intégration continue).

(*product right*) et son adéquation aux besoins (*right product*), cet article fait le point sur la conception et l'utilisation des tests dans les processus agiles et classiques.

Si les tests classiques ont pour objectif avant tout de détecter des erreurs, les tests agiles ont pour objectif principal de guider le développement. Cette différence d'objectif induit des pratiques différentes vis-à-vis des tests. Les tests agiles possèdent une plus grande valeur ajoutée car ils sont considérés comme des entités de première classe par le développement agile : le code des tests est aussi important que le code de production.

Même si elles n'ont pas contribué de manière fondamentale aux tests, les méthodes agiles ont poussé des bonnes pratiques que les méthodes classiques pourraient se réapproprier. Le code d'un test se doit d'expliciter un cas de test ou un scénario d'exécution. De ce fait, il accélère la compréhension du périmètre de l'objet à tester. Ainsi, il peut être réutilisé comme point de départ d'une documentation dite vivante. Un test se doit d'être, dès qu'il est automatisable, automatisé et rejouable à tout moment. Le test garantit ainsi la non-régression du code de production, ce qui accroît la confiance de l'équipe dans la valeur du produit.

Remerciements

Nous remercions les communautés Agile *et* Software Craftsmanship *de Toulouse pour les discussions autour du test agile, Olivier Azeau et Laurent Meurisse pour leurs retours d'expérience en matière de processus de développement agile, et Benoit Gantaume pour sa vision des développements dirigés par les tests.*

Bibliographie

Adzic G. (2011). *Specification by example: How successful teams deliver the right software* (1st éd.). Greenwich, CT, USA, Manning Publications Co.

Ammann P., Offutt J. (2008). *Introduction to software testing*. Cambridge University Press.

Anderson D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

Aubry C. (2011). *Scrum : Le guide pratique de la méthode agile la plus populaire*. Dunod.

Bache E. (2013). *The coding dojo handbook*. Leanpub.

Beck K. (2000). *Extreme programming explained: Embrace change*. Addison-Wesley.

Beck K. (2002). *Test driven development: By example*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Beck K., Beedle M., Bennekum A. van, Cockburn A., al. (2001). *Manifesto for agile software development.* Consulté sur http://agilemanifesto.org

Boehm B. W. (1984). Verifying and validating software requirements and design specifications. *IEEE Software*, p. 75–88.

Chelimsky D., Astels D., Helmkamp B., Dennis Z., North D., Hellesoy A. (2010). *The rspec book: Behaviour driven development with rspec, cucumber, and friends*. Pragmatic Programmers, LLC.

Cockburn A. (2004). *Crystal clear a human-powered methodology for small teams* (First éd.). Addison-Wesley Professional.

Cohn M. (2004). *User stories applied: For agile software development*. Redwood City, CA, USA, Addison Wesley Longman Publishing Co., Inc.

Cohn M. (2009). *Succeeding with agile: Software development using scrum* (1st éd.). Addison-Wesley Professional.

Crispin L., Gregory J. (2009). *Agile testing: A practical guide for testers and agile teams* (1re éd.). Addison-Wesley Professional.

Cunningham W. (1992, décembre). The wycash portfolio management system. *SIGPLAN OOPS Mess.*, vol. 4, nº 2, p. 29–30.

Cunningham W., Mugridge R. (2005). *Fit for developing software: Framework for integrated tests*. Prentice Hall.

Cusumano M. A. (1991). Factory concepts and practices in software development. *IEEE Annals of the History of Computing*, vol. 13, nº 1, p. 3-32.

Deak A. (2014). A comparative study of testers' motivation in traditional and agile software development. In A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, M. Raatikainen (Eds.), *Product-focused software process improvement*, vol. 8892, p. 1-16. Springer International Publishing.

Dijkstra E. W. (1972). Structured programming. In O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare (Eds.),, p. 1–82. London, UK, UK, Academic Press Ltd.

Duvall P., Matyas S. M., Glover A. (2007). *Continuous integration: Improving software quality and reducing risk (the addison-wesley signature series)*. Addison-Wesley Professional.

Fowler M. (1999). *Refactoring: Improving the design of existing code*. Boston, MA, USA, Addison-Wesley.

Fowler M. (2001). Extreme programming examined. In G. Succi, M. Marchesi (Eds.),, p. 3–17. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Fowler M. (2006). Continuous integration. (http://martinfowler.com/articles/continuousIntegration.html)

Freeman S., Pryce N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional.

Freeman S., Pryce N., Mackinnon T., Walnes J. (2004). Mock roles, not objects. In *Oopsla'04: Companion to the 19th annual acm sigplan conference on object-oriented programming systems, languages and applications*, p. 236–246. ACM Press.

Gamma E., Helm R., Johnson R., Vlissides J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Gartner M. (2012). *Atdd by example: A practical guide to acceptance test-driven development* (1st éd.). Addison-Wesley Professional.

Gaudel M.-C. (2011). Checking models, proving programs, and testing systems. In M. Gogolla, B. Wolff (Eds.), *Tap*, vol. 6706, p. 1-13. Springer.

Hendrickson E. (2008). Driving development with tests: Atdd and tdd. *Quality Tree Software, Inc. http://www. qualitytree. com*.

Hendrickson E. (2013). *Explore it!: Reduce risk and increase confidence with exploratory testing*. Pragmatic Bookshelf.

Humble J., Farley D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation* (1st éd.). Addison-Wesley Professional.

Hunt A., Thomas D. (1999). *The pragmatic programmer: From journeyman to master*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

IEEE. (1990, Dec). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*.

IEEE. (1993, Dec). IEEE guide for software verification validation plans. *IEEE Std 1059-1993*.

IEEE. (2008). IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*.

ISO/IEC. (2010). *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. ISO/IEC.

ISTQB. (2014). *Certified tester - foundation level extension syllabus agile tester.* Consulté sur http://www.istqb.org/downloads/syllabi/agile-tester-extension-syllabus.html

ISTQB. (2015). *Standard glossary of terms used in software testing of the international software testing qualifications* (vol. Version 3.01).

Jeffries R. E., Anderson A., Hendrickson C. (2000). *Extreme programming installed*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Jorgensen P. C. (2010). Test-driven development: un pacte diabolique ? (http://www.cftl.fr/index.php?id=78)

Kettunen V., Kasurinen J., Taipale O., Smolander K. (2010). A study on agility and testing processes in software organizations. In *Proceedings of the 19th international symposium on software testing and analysis*, p. 231–240. New York, NY, USA, ACM.

Kniberg H. (2010). *Kanban and scrum - making the most of both*. Lulu.com.

Laukkanen P. (2006). *Data-driven and keyword-driven test automation frameworks*. Thèse de doctorat non publiée, Helsinki University of Technology.

Mackinnon T., Freeman S., Craig P. (2001). Extreme programming examined. In G. Succi, M. Marchesi (Eds.),, p. 287–301. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Mancuso S. (2015). *The software craftsman : Professionalism, pragmatism, pride* (1re éd.). Upper Saddle River, NJ, USA, Prentice Hall PTR.

Martin R. C. (2002). *Agile software development: Principles, patterns, and practices*. Upper Saddle River, NJ, USA, Prentice Hall PTR.

Martin R. C. (2008). *Clean code: A handbook of agile software craftsmanship* (1re éd.). Upper Saddle River, NJ, USA, Prentice Hall PTR.

Mathur A. (2008). *Foundations of software testing*. Pearson Education.

McBreen P. (2002). *Software craftsmanship: The new imperative*. Addison-Wesley.

Meszaros G. (2007). *Xunit test patterns: Refactoring test code*. Addison-Wesley.

Meszaros G., Smith S. M., Andrea J. (2003). The Test Automation Manifesto. In *Extreme programming and agile methods - xp/agile universe 2003*, vol. 2753, p. 73–81. Springer Berlin / Heidelberg.

Myers G. J., Sandler C. (2004). *The art of software testing*. John Wiley & Sons.

North D. (2006). Introducing BDD. *Better Software Magazine*. Consulté sur http://dannorth.net/introducing-bdd/

Orientation pour tous. (2015). *Fiche métier : Analyste test et validation.* Consulté sur http://www.orientation-pour-tous.fr/metier/analyste-test-et-validation,13597.html

Overmyer S. P. (1990, octobre). Dod-std-2167a and methodologies. *SIGSOFT Softw. Eng. Notes*, vol. 15, n° 5, p. 50–59.

Printz J., Pradat-Peyre J. (2009). *Pratique des tests logiciels - concevoir et mettre en oeuvre une stratégie de tests. préparation à la certification istqb*. Dunod.

Rafique Y., Misic V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, vol. 39, n° 6, p. 835-856.

Royce W. W. (1970, August). Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, p. 1–9. (Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338)

Ruparelia N. B. (2010, may). Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, vol. 35, n° 3, p. 8–13.

Schwaber K. (1995). Scrum development process. In *Proceedings of the 10th annual acm conference on object oriented programming systems, languages, and applications (oopsla*, p. 117–134.

Schwaber K., Beedle M. (2001). *Agile software development with scrum* (1st éd.). Upper Saddle River, NJ, USA, Prentice Hall PTR.

Schwaber K., Sutherland J. (2013). *The Scrum guide.* Consulté sur http://www.scrumguides.org/

Shore J. (2004, jan). Continuous design. *IEEE Software*, vol. 21, n° 1, p. 20–22.

Siniaalto M., Abrahamsson P. (2007, Sept). A comparative case study on the impact of test-driven development on program design and test coverage. In *Empirical software engineering and measurement, 2007. esem 2007. first international symposium on*, p. 275-284.

Sommerville I. (2010). *Software engineering* (9e éd.). Harlow, England, Addison-Wesley.

Standish-Group. (2013). *Chaos report*. Rapport technique.

Takeuchi H., Nonaka I. (1986). The new new product development game. *Harvard Business Review*, vol. 64, n° 1, p. 137–146.

Tretmans J. (1999). Testing concurrent systems: A formal approach. In *Concur*, vol. 1664, p. 46-65. Springer.

Version One. (2013). *8th annual state of agile survey*. Rapport technique. http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf.

Yusifoğlu V. G., Amannejad Y., Can A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, vol. 58, p. 123 - 147.