Testeur certifié
Testeur Technique Agile

Syllabus

Version 1.1

International Software Testing Qualifications Board

Comité Français des Tests Logiciels

Notice de copyright

Historique de révision

Version	Date	Remarques
Syllabus v0.1	11 janvier 2017	Sections autonomes
Syllabus v0.2	24 mai 2017	Commentaires des revues du WG de la v01 incluses
Syllabus v0.3		Commentaires des revues du WG de la v02 incluses
Syllabus v0.7		Commentaires de la revue alpha de la v03 incluses
Syllabus v0.71		Mises à jour du groupe de travail sur la v07
Syllabus v0.9	30 décembre 2017	Candidat Alpha
Syllabus v0.91	6 janvier 2018	Alpha release
Syllabus v0.98	12 janvier 2019	Candidat bêta
Syllabus v0.99		Beta release
Syllabus 2018		Version GA
Syllabus 2019	12 février 2019	Modifications mineures du texte Ajout de plusieurs LOs (Niveaux K1, K2) Ajout d'informations sur le droit d'auteur Ajout de temps d'apprentissage dans tous les chapitres
Syllabus 2019	14 mars 2019	Modification des LOs pour les adapter au nom court officiel “ATT” Modification des “Remerciements”
Syllabus 2019	06 mai 2019	Quelques modifications mineures
Syllabus 2019	07 juin 2019	Modifications mineures
Syllabus 2019	08 juillet 2019	Changements rédactionnels
Syllabus 2019	10 juillet 2019	Préparation pour la revue bêta
Syllabus 2019	14 septembre 2019	Améliorations, correction suite à la revue bêta
Syllabus 2019	14 novembre 2019	Date de sortie
Syllabus 2019	09 décembre 2019	Modification du logo ISTQB® Modification de la durée de formation au niveau K (Chapitre 0.6)
Syllabus 2019 FR	23 mars 2020	Version française finalisée

Table des matières

Remerciements

Ce document a été produit par une équipe du groupe de travail Agile de l'International Software Testing Qualifications Board mené par Rex Black (chair), Michaël Pilaeten (Vice-chair et chair a.i.) et Renzo Cerquozzi (Product Owner).

L'équipe Agile de niveau avancé remercie l'équipe de revues et les bureaux nationaux de l'ISTQB® pour leurs suggestions et leurs commentaires.

Au moment où le syllabus Advanced Level Agile Technical Tester a été achevé, le Groupe de travail comptait les membres suivants : Michaël Pilaeten (Chair a.i.), Renzo Cerquozzi (Product Owner), Alon Linetzki (Marketing workgroup), Leanne Howard (Glossary workgroup) and Klaus Skafte (Exam workgroup).

Auteurs : Leo van der Aalst, Renzo Cerquozzi, Bertrand Cornanguer, István Forgács, Jani Haukinen, Noam Kfir, Sammy Kolluru, Alon Linetzki, Tilo Linz, Michaël Pilaeten, Marie Walsh.

Réviseurs internes : Michael Arefi, Vojtěch Barta, Renzo Cerquozzi. Graham Bath, Laurent Bouhier, Anders Claesson, Alessandro Collino, David Janota, David Evans, Leanne Howard, Matthias Hamburg, Kari Kakkonen, Tor Kjetil Moseng, Meile Posthuma, Salvatore Reale, Marko Rytkönen, Sarah Savoy, Klaus Skafte, Mike Smith, Chris van Bael.

Création et revus des exemples de questions : Armin Born, Renzo Cerquozzi, Alon Linetzki, Tilo Linz, Jamie Mitchell, Jani Haukinen, Tobias Horn, Chris van Bael, Suruchi Varshney.

L'équipe remercie également les personnes suivantes, issues des bureaux nationaux de l'ISTQB et la communauté d'experts Agile, qui ont participé à la revue, aux commentaires et au vote du ce syllabus: Adam Roman, Armin Beer, Beata Karpinska, Chris Van Bael, Erwin Engelsma, Giancarlo Tomasig, Gary Mogyorodi, Ingvar Nordström, Jana Gierloff, Jörn Münzel, Jurian van de Laar, Kari Kakkonen, Laurent Bouhier, Marko Rytkönen, Martin Klonk, Matthias Hamburg, Meile Posthuma, Paul Weymouth, R. Green, Richard Seidl, Rik Marselis, Stephanie Ulrich, Stephanie van Dijck, Tal Pe'er, Tilo Linz, Veronica Seghieri and Wim Decoutere.

Un merci spécial à Galit Zucker, Secrétaire générale de l'ISTQB® pour l'orientation et le soutien.

Ce document a été officiellement approuvé pour publication par l'Assemblée Générale de l'ISTQB® le 18 octobre 2019.

La traduction française est la propriété du CFTL. Elle a été réalisée par un groupe d'experts en tests logiciels : Olivier Denoo, Bruno Legeard, Jean-François Pistiaux et Eric Riou du Cosquer.

0 Introduction à ce Syllabus

0.1 Objet du présent document

Ce programme constitue la base de l'International Software Testing Qualification au niveau avancé pour le Testeur Technique Agile. L'ISTQB® fournit ce syllabus comme suit :

- Aux bureaux nationaux de l'ISTQB, pour le traduire dans leur langue locale et accréditer les prestataires de formation. Les bureaux nationaux de l'ISTQB peuvent adapter le syllabus à leurs besoins linguistiques particuliers et modifier les références pour s'adapter à leurs publications locales.
- Aux organismes de certification, afin d'obtenir des questions d'examen dans leur langue locale adaptées aux objectifs d'apprentissage de chaque syllabus.
- Aux organismes de formation, afin de produire des cours et déterminer les méthodes d'enseignement appropriées.
- Aux candidats à la certification, afin de se préparer à l'examen (dans le cadre d'une formation ou de manière indépendante).
- À la communauté internationale de l'ingénierie des logiciels et des systèmes, pour faire progresser la profession du test des logiciels et des systèmes, et comme base pour les livres et les articles.

L'ISTQB® peut permettre à d'autres entités d'utiliser ce syllabus à d'autres fins, sous condition qu'elles aient obtenu une autorisation écrite préalable.

0.2 Aperçu

Le document Aperçu du Niveau Avancé Testeur Technique Agile [ISTQB_ATT_OVIEW] comprend les informations suivantes :

- Objectifs Métier pour ce syllabus
- Résumé du syllabus
- Relations entre les syllabi
- Description des niveaux cognitifs (niveaux K)
- Annexes sur le test des systèmes et des logiciels, et référence pour les livres et les articles

0.3 Objectifs d'apprentissage examinables (LOs)

Les objectifs d'apprentissage soutiennent les Objectifs Métier et sont utilisés pour créer l'examen pour atteindre le niveau avancé de testeur certifié - Certification Testeur Technique Agile. En général, toutes les parties de ce syllabus sont examinables à un niveau K1. C'est-à-dire que le candidat reconnaîtra, se souviendra et se remémorera d'un terme ou d'un concept. Les objectifs d'apprentissage spécifiques aux niveaux K1, K2, K3 et K4 sont affichés au début du chapitre pertinent.

0.4 L'examen de certification de testeur technique Agile de niveau avancé

L'examen de certification de testeur technique Agile de niveau avancé sera basé sur ce syllabus. Les réponses aux questions d'examen peuvent nécessiter l'utilisation de matériel basé sur plus d'une section de ce syllabus. Toutes les sections du syllabus sont examinables, à l'exception de l'introduction et des annexes. Les normes, livres et autres syllabi ISTQB® sont inclus comme références, mais leur contenu n'est pas examinable au-delà de ce qui est résumé dans ce programme lui-même à partir de ces normes, livres, et autres syllabi ISTQB®.

Le format de l'examen est à choix multiples.

Les examens peuvent être effectués dans le cadre d'un cours de formation accrédité ou effectués de façon indépendante (p. ex., dans un centre d'examen ou lors d'un examen public). L'achèvement d'un cours de formation accrédité n'est pas une condition préalable à l'examen.

0.5 Accréditation

Un Membre de l'ISTQB® (Bureaux Nationaux) peut accréditer les prestataires de formation dont le matériel de cours suit ce syllabus.
Les organismes de formation doivent obtenir les consignes d'accréditation du Membre ou l'organisme qui effectue l'accréditation. Un cours accrédité est reconnu comme conforme à ce syllabus et est autorisé à inclure un examen ISTQB® dans le cadre du cours.

0.6 Comment ce syllabus est organisé

Il y a quatre chapitres avec un contenu examinable.

Le titre principal de chaque chapitre spécifie le temps total pour le chapitre ; le timing n'est pas défini en dessous de ce niveau.

Pour les cours de formation accrédités, le syllabus nécessite un minimum de 16 heures d'enseignement, répartis entre les chapitres comme suit :

1. Chapitre 1 : 180 minutes : Ingénierie des exigences
1. Chapitre 2 : 540 minutes : Tester en Agile
1. Chapitre 3 : 135 minutes : Automatisation des tests
1. Chapitre 4 : 105 minutes : Déploiement et livraison

Étant donné une durée minimale pour couvrir chaque objectif d'apprentissage :

- K2: 15 minutes
- K3: 60 minutes
- K4: 75 minutes

1 Ingénierie des exigences 180 minutes

Mots-clés : critères d'acceptation, Epic, User Story

Objectifs d'apprentissage pour l'ingénierie des exigences

ATT-1.x (K1) Mots-clés

1.1 Techniques d'ingénierie des exigences

ATT-1.1.1-1 (K4) Analyser les User Stories et les Epics en utilisant des techniques d'ingénierie d'exigences

ATT-1.1.1-2 (K2) Décrire les techniques d'ingénierie des exigences et comment elles peuvent aider les testeurs

ATT-1.1.2-1 (K4) Créer et évaluer des critères d'acceptation testables pour une User Story donnée en utilisant des techniques de test et d'ingénierie des exigences

ATT-1.1.2-2 (K2) Décrire les techniques d'élucidation

1.1 Techniques d'ingénierie des exigences

L'application des techniques d'ingénierie des exigences permet aux équipes Agile d'affiner les User Stories (cf [AgileFoundationExt]) et les Epics (cf. [AgileFoundationExt]), ajouter le contexte, tenir compte des impacts et des dépendances, et identifier les lacunes, telles que les exigences non fonctionnelles manquantes.
Bien que la majorité des techniques d'ingénierie des exigences discutées dans cette section proviennent d'approches de développement traditionnelles, elles sont également efficaces dans le développement agile.
En général, dans les projets traditionnels, les activités et les techniques d'ingénierie requises sont formalisées, exécutées de façon séquentielle et sont la responsabilité de personnes désignées telles que les Analystes Métier, les Analystes Fonctionnels, les Architectes Techniques, les Architectes d'Entreprise et Analystes des Processus.
En revanche, dans les projets Agile, les techniques d'ingénierie requises sont appliquées tout au long du projet et lors de chaque itération via une approche moins formelle. Ces techniques d'ingénierie des exigences sont exécutées plus fréquemment, en utilisant des boucles de rétroaction continue, par tous les membres de l'équipe Agile, et pas seulement l'Analyste Métier dédié ou le Product Owner de l'équipe.

1.1.1 Analyser les User Stories et les Epics en utilisant des techniques d'ingénierie d'exigences

En tant que testeur, pour être en mesure d'aider à clarifier (et éventuellement améliorer) les User Stories, Epics et autres exigences Agiles, il est nécessaire de connaître, de comprendre, de sélectionner et d'utiliser les différentes techniques d'ingénierie des exigences qui sont utiles pour cela.
Des exemples de ces techniques sont les Storyboards, Story Mapping, personas, diagrammes et cas d'utilisation.

- **Storyboards** : Un Storyboard (à ne pas confondre avec le tableau des tâches en Agile ou le tableau des User Stories en Agile) fournit une représentation visuelle du système. Les Storyboards aident les testeurs à :
  - Voir le processus de réflexion derrière les User Stories, et la vision d'ensemble, en fournissant un contexte, permettant de voir rapidement le flux fonctionnel du système et d'identifier les lacunes dans la logique.
  - Visualiser des groupes de User Stories liés à une zone commune du système (Thèmes) qui peuvent être considérés pour l'inclusion dans la même itération, car ils toucheront probablement le même morceau de code.
  - Aider au Story Mapping et à la priorisation des Epics et des User Stories correspondantes dans le backlog du produit.
  - Aider à identifier les critères d'acceptation pour les User Stories et les Epics.
  - Aider à choisir la bonne approche de test en fonction de l'aspect visuel de la conception du système.
  - Avec le Story Mapping, aider à prioriser les tests et à identifier les besoins en bouchons, pilotes et/ou mocks.

- **Story Mapping** : Story Mapping (ou User Story Mapping) est une technique qui consiste à utiliser 2 dimensions indépendantes pour ordonner les User Stories. L'axe horizontal de la carte représente l'ordre de priorité de chacune des User Stories, tandis que l'axe vertical représente le niveau d'affinage de la mise en œuvre. L'utilisation du Story Mapping peut aider les testeurs à :
  - Déterminer les fonctionnalités les plus élémentaires d'un système pour définir un test fumigatoire (smoke test).
  - Identifier l'ordre des fonctionnalités pour déterminer les priorités de test.
  - Visualiser la portée du système.
  - Déterminer le niveau de risque de chaque User Story.

- **Persona** : Les personas sont utilisées pour définir des personnages fictifs ou des archétypes qui illustrent comment les utilisateurs typiques interagiront avec le système. L'utilisation de personas peut aider les testeurs à :
  - Identifier les lacunes dans les User Stories en identifiant différents types d'utilisateurs qui peuvent utiliser le système.
  - Identifier les incohérences dans les User Stories sur la façon dont un type particulier d'utilisateur peut utiliser le système par rapport à d'autres.
  - Elucider les critères d'acceptation d'une User Story
  - Découvrir d'autres pistes de test lors de tests exploratoires
  - Révéler les conditions de test, en particulier celles liées à des groupes d'utilisateurs particuliers, contribuant ainsi à assurer une couverture suffisante des groupes d'utilisateurs et à tester les différences entre les groupes d'utilisateurs.

- **Diagrammes** : Les diagrammes tels que les diagrammes de relation d'entité, les diagrammes de classe et (autres) diagrammes UML peuvent montrer la structure ou le flux de données et les attributs fonctionnels ou le comportement du système, et peuvent être utilisés pour identifier les lacunes dans la fonctionnalité du système.

- **Cas d'utilisation** : Les cas d'utilisation (diagrammes et spécifications) [Foundation] peuvent aider les testeurs à :
  - S'assurer que les User Stories sont testables et de taille appropriée.
  - Déterminer si les User Stories doivent être affinées ou décomposées.
  - Faire ressortir les parties prenantes oubliées.
  - Identifier les interfaces et les points d'intégration, qui doivent être pris en compte lors de la conception des tests.
  - Voir les relations entre Epics et User Stories, pour vérifier que l'Epic n'a pas de User Stories manquantes.

1.1.2 Identifier les critères d'acceptation à l'aide de techniques de test et d'ingénierie des exigences.

L'ingénierie des exigences est un processus composé des étapes suivantes :

- **Elucidation** : Le processus de découverte, de compréhension, de documentation et de revue des besoins et des contraintes des utilisateurs pour le système. Les techniques d'élucidation devraient être utilisées pour dériver, évaluer et améliorer les critères d'acceptation.
- **Documentation** : Le processus de documentation des besoins et des contraintes des utilisateurs clairement et précisément. Les User Stories et les critères d'acceptation doivent être documentés au niveau correspondant à l'adhésion de l'équipe aux principes du manifeste Agile. Le type de documentation dépend de l'approche de l'équipe et des parties prenantes. Les critères d'acceptation peuvent être documentés à l'aide d'un langage naturel, à l'aide de modèles (p. ex., diagrammes de transition d'état) ou à l'aide d'exemples.
- **Négociation et validation** : Pour chaque User Story, plusieurs parties prenantes peuvent avoir d'autres idées ou préférences. Étant donné que ces idées et préférences peuvent être incohérentes, voire contradictoires, les critères d'acceptation de chaque partie prenante peuvent l'être aussi. Chacun de ces conflits doit être identifié, négocié et résolu entre toutes les parties prenantes concernées. Tout conflit oublié ou non résolu pourrait mettre en danger le succès du projet. À la fin de cette étape, le contenu de chaque User Story est validé par les parties prenantes concernées (par exemple comme définition de « prêt » (Definition of Ready)).

- **Gestion** : Au fur et à mesure que les projets progressent, les opinions et les circonstances peuvent changer. Même si les critères d'acceptation ont été correctement élucidés, documentés, négociés et validés, les critères d'acceptation sont toujours sujets à changement. En raison du potentiel de changements, les User Stories doivent être gérées en utilisant une bonne configuration et des processus de gestion du changement.

Pour identifier les critères d'acceptation, de multiples techniques d'élucidation sont à la disposition du testeur, y compris :

- **Questionnaires quantitatifs** : L'utilisation de données quantitatives tirées de questions fermées est un excellent moyen de faire des comparaisons claires entre les différentes sources d'information. Cela permet souvent d'obtenir des données chiffrées qui peuvent être incluses dans un résultat numérique pour un critère d'acceptation. Le questionnaire quantitatif peut être utilisé comme technique d'élucidation pour un grand nombre d'intervenants et plus particulièrement pour les critères d'acceptation non fonctionnels.
- **Questionnaires qualitatifs** : Les questions ouvertes sont un moyen extrêmement efficace d'ajouter plus de qualité à la recherche quantitative. Les questions ouvertes sont bien adaptées au suivi des questions clés. Cela pourrait générer des informations supplémentaires pour lesquelles de nouvelles User Stories doivent être crées ou doivent être ajoutés à celles existantes. Le questionnaire qualitatif peut être utilisé comme technique d'élucidation pour un plus petit nombre d'intervenants, car le traitement prend plus de temps et convient aux critères d'acceptation fonctionnelle.
- **Interview qualitative :** L'interview qualitative est plus souple qu'un questionnaire quantitatif et est principalement utilisée pour acquérir des informations sur l'historique, les contextes et les causes. Il est peu probable qu'elle retourne des données concrètes, mais les critères d'acceptation peuvent être dérivés des réponses concernant le contexte d'une User Story. L'interview qualitative peut être suivi de tout type de questionnaire, pour approfondir les critères d'acceptation dérivés.

Plusieurs autres techniques d'élucidation existent, dans l'éventail des techniques d'observation (p. ex. l'apprentissage), des techniques créatives (p.ex. « 6 Thinking hats » – NDT : Les 6 chapeaux de la pensée rationnelle) et des techniques de soutien (p. ex. maquettage rapide – low-fi prototyping). L'ensemble des techniques du testeur influencera la qualité des critères d'acceptation élucidés.

INVEST et SMART [INVEST] peuvent également être utilisés pour identifier et évaluer les critères d'acceptation, au côté des techniques de test telles que les partitions d''équivalence, l'analyse de la valeur aux limites, les tables de décision et les tests de transition d'état [Foundation].

2 Tester en Agile 540 minutes

Mots-clés : test-driven development, behavior-driven development, acceptance test-driven development, spécification par l'exemple, charte de test

Objectifs d'apprentissage pour Tester en Agile

ATT-2.x (K1) Mots-clés

2.1 Techniques de développement et de test en Agile

ATT-2.1.1-1 (K3) Appliquer le test-driven development (TDD) dans le contexte d'un exemple donné dans un projet Agile

ATT-2.1.1-2 (K2) Comprendre les caractéristiques d'un test unitaire

ATT-2.1.1-3 (K2) Comprendre le sens du mot mnémonique FIRST

ATT-2.1.2-1 (K3) Appliquer le behavior-driven development (BDD) dans le contexte d'une User Story donnée dans un projet Agile

ATT-2.1.2-2 (K2) Comprendre comment gérer les lignes directrices pour la formulation d'un scénario

ATT-2.1.3 (K4) Analyser un backlog du produit dans un projet Agile pour déterminer un moyen d'introduire l'acceptance test-driven development (ATDD)

2.2 Tests basés sur l'expérience en Agile

ATT-2.2.1-1 (K4) Analyser la création d'une approche de test à l'aide de l'automatisation des tests, des tests basés sur l'expérience et des tests en boîte noire, créés à l'aide d'autres approches (y compris les tests basés sur le risque) pour un scénario donné dans un projet Agile

ATT-2.2.1-2 (K2) Expliquer les différences vis-à-vis du caractère critique ou non-critique de la mission

ATT-2.2.2-1 (K4) Analyser des User Stories et des Epics pour créer des chartes de test

ATT-2.2.2-2 (K2) Comprendre l'utilisation des techniques basées sur l'expérience

2.3 Aspects de la qualité du code

ATT-2.3.1-1 (K2) Comprendre l'importance du refactoring des cas de test dans les projets Agile

ATT-2.3.1-2 (K2) Comprendre la liste pratique des tâches pour le refactoring des cas de test

ATT-2.3.2-1 (K4) Analyser le code dans le cadre d'une revue du code afin d'identifier les défauts et la dette technique

ATT-2.3.2-2 (K2) Comprendre l'analyse statique du code

2.1 Techniques de développement et de test en Agile

Dans le développement de logiciels, des défauts peuvent se produire à partir de code mal écrit et d'une mauvaise réponse aux besoins des clients. Les techniques de développement en Agile traitent ces problèmes en appliquant les concepts de test-driven development (TDD)[1], behavior-driven development (BDD)[2], et d'acceptance test-driven development[3] (ATDD). TDD est une technique pour améliorer la qualité des produits logiciels, tandis que BDD et ATDD aident à améliorer sa qualité d'utilisation (features et fonctionnalités).

2.1.1 Test-driven development (TDD)[4]

Test-driven development est une méthode de développement logiciel qui combine la conception, les tests et le codage dans un processus itératif rapide. TDD adopte une approche disciplinée de la création initiale d'un test qui exprime la fonctionnalité prévue du code avant de coder cette fonctionnalité. Le test est exécuté avant que le code lui-même soit écrit, afin de vérifier que le test échoue. Une fois le code écrit, le test est exécuté à nouveau et doit être réussi.

TDD est généralement considérée comme la première grande méthodologie de programmation basée sur le test à partir de laquelle d'autres méthodologies, telles que behavior-driven development (BDD), acceptance test-driven development (ATDD) et les spécifications par l'exemple (SBE), ont été dérivées.

TDD offre une approche évolutive du développement de logiciels qui traite la conception comme un processus continu et itératif. Chaque itération constitue une petite modification du code de production et offre au développeur la possibilité d'améliorer légèrement la conception. Au fur et à mesure que les changements et les décisions de conception soigneusement réfléchies s'accumulent étape par étape, une conception robuste et bien testée émerge.

Les praticiens du TDD utilisent un certain nombre de pratiques et de techniques pour rédiger un code de haute qualité et éviter l'accroissement de la dette technique, y compris :

- Tests unitaires et approche prescriptive de test d'abord
- Cycles itératifs courts
- Rétroaction immédiate et fréquente
- Utilisation efficace de l'outillage, du contrôle des sources et de l'intégration continue
- Application guidée des principes de programmation et des modèles de conception

Les praticiens du TDD écrivent des tests unitaires pour vérifier le comportement attendu du code en production. Un test unitaire exécute une fonction dans des conditions spécifiques et vérifie si cela produit le résultat attendu ou non. Les résultats attendus sont décrits avec des assertions dans le code qui vérifient que l'état du système change comme prévu, ou que le comportement spécifique est correct. Les assertions peuvent vérifier, par exemple:

- Qu'une fonction effectue un calcul et retourne le résultat attendu
- Qu'une fonction modifie l'état du système d'une manière particulière
- Qu'une fonction appelle une autre fonction d'une manière spécifique

[1] développement piloté par les tests (en français)
[2] développement piloté par le comportement (en français)
[3] développement piloté par les tests d'acceptation (en français)
[4] NDT – Nous avons choisi de conserver dans la suite de ce syllabus les intitulés et acronymes en anglais (TDD, BDD et ATDD) car c'est ainsi que ces approches de test sont identifiées dans la pratique courante en français.

Les tests unitaires doivent être faciles à mettre en œuvre et à maintenir pour les développeurs. Ils sont automatisés et sont souvent écrits en utilisant le même langage que le code de production. Les tests unitaires doivent avoir les caractéristiques suivantes :

- Déterministe - Chaque fois qu'un test unitaire est exécuté dans les mêmes conditions, il doit produire les mêmes résultats.
- Atomique - Le test unitaire ne doit tester que les fonctionnalités qui y sont liées.
- Isolé - Un test unitaire devrait s'efforcer d'exercer uniquement le code spécifique pour lequel il était initialement prévu. Les tests unitaires ne doivent pas dépendre les uns des autres et doivent éviter les dépendances dans le code de production dans la mesure du possible.
- Rapide – Les tests unitaires doivent être petits et rapides afin qu'ils puissent fournir une rétroaction immédiate. Il devrait être possible d'exécuter de nombreux tests unitaires dans un court laps de temps.

Un autre mnémonique pour décrire un bon test unitaire est FIRST : Fast, Isolated, Repeatable, Self-Validating, Thorough (NDT : Rapide, Isolé, Répétable, Auto-validant, Complet/Approfondi)

Il existe de nombreux frameworks de tests unitaires disponibles pour de nombreux langages différents. Ils diffèrent dans leurs API – Interface de Programmation, leurs approches et leur terminologie, mais tous partagent certains éléments communs. Indépendamment du langage ou du framework, un test unitaire suit généralement ce modèle en trois étapes :

1. Organiser/Installer– préparer l'environnement d'exécution. Cette étape peut instancier des objets, initialiser l'état du système et injecter des données au besoin.
2. Agir - exécuter l'opération testée, puis vérifier le résultat produit par l'opération. Cette étape obligatoire peut consister en une seule ligne de code qui déclenche l'opération testée.
3. Vérifier - effectuer la vérification réelle des résultats attendus ou d'autres postconditions.

Le processus itératif est la pierre angulaire du TDD. Le but de chaque itération est d'apporter une petite modification ciblée et soigneusement réfléchie au programme. Après la convention à code couleur, le cycle itératif est souvent appelé le cycle rouge-vert-refactoring :

- Rouge - Rédigez un test défaillant qui décrit une attente non mise en œuvre. Exécuter le test pour s'assurer qu'il échoue.
- Vert - Écrire un code de production qui ne satisfait que l'attente décrite par le test et le fait réussir. Exécutez tous les tests pertinents pour vous assurer qu'ils réussissent tous. En cas d'échec, apporter les modifications nécessaires pour que tous les tests passent.
- Refactoring – Améliorer la conception et la structure du code de test et du code de production sans modifier les fonctionnalités, tout en veillant à ce que tous les tests pertinents continuent de passer. Les développeurs peuvent appliquer une séquence de refactoring pour modifier le code sans modifier le comportement jusqu'à ce que le code soit optimisé. La plupart des environnements de développement intégrés modernes [Integrated Development Environments (IDE's)] et de nombreux éditeurs de code peuvent appliquer le refactoring automatiquement et en toute sécurité.

2.1.2 Behavior Driven Development (BDD)

Selon le syllabus ISTQB® Agile de Niveau Fondation, behavior-driven development (BDD) est une technique dans laquelle les développeurs, les testeurs et les représentants du métier travaillent ensemble pour analyser les exigences d'un système logiciel, les formuler à l'aide d'un langage partagé, et les vérifier automatiquement.

BDD est fortement influencé par deux disciplines distinctes : test-driven development (TDD) et domain-driven design (DDD) [Evans03]. Il intègre bon nombre des principes fondamentaux des deux disciplines, y compris la collaboration, la conception, le langage omniprésent, l'exécution automatisée des tests, les cycles de rétroaction courts, et plus encore. TDD s'appuie sur des tests unitaires pour vérifier les détails de la mise en œuvre alors que

BDD s'appuie sur des scénarios exécutables pour vérifier en particulier les comportements attendus. BDD suit généralement la démarche suivante :

- Créer des User Stories de manière collaborative
- Formuler les User Stories comme des scénarios exécutables aux comportements vérifiables
- Implémenter les comportements et exécuter les scénarios pour les vérifier

Les équipes qui appliquent le BDD extraient un ou plusieurs scénarios de chaque User Story, puis les formulent comme des tests automatisés. Un scénario représente un comportement unique dans des conditions spécifiques.
Les scénarios sont généralement basés sur les critères d'acceptation des User Stories, des exemples et des cas d'utilisation. Les scénarios BDD doivent être écrits en utilisant un langage qui peut être compris par tous les membres de l'équipe, qu'ils soient techniques ou non.
Par conséquent, une forte préférence est donnée pour l'utilisation de la langue naturelle, comme l'anglais ou le français, pour exprimer les scénarios. En outre, les équipes qui appliquent BDD définissent un langage partagé [Evans03], qui constitue une terminologie claire et sans ambiguïté pour tous les membres de l'équipe et qui est utilisée partout.

Les scenarios de BDD sont généralement composés de trois sections principales [Gherkin] :

1. Given [NDT Etant donné] – décrit l'état de l'environnement (conditions préalables) avant que le comportement ne soit déclenché
2. When [NDT Quand ou Lorsque] – décrit les actions qui déclenchent le comportement
3. Then [NDT Alors] – décrit les résultats attendus du comportement

Extraire des scénarios à partir des User Stories implique de/d' :

- Identifier tous les critères d'acceptation spécifiés par une User Story et écrire des scénarios pour chaque critère. Certains peuvent nécessiter plusieurs scénarios.
- Identifier les cas d'utilisation fonctionnelle et les exemples et d'écrire des scénarios pour chacun d'eux. De nombreux cas d'utilisation et exemples sont conditionnels, il est donc crucial d'identifier chacune des conditions.
- Créer des scénarios tout en pratiquant le test exploratoire, ce qui peut aider à identifier les comportements existants connexes, les comportements conflictuels potentiels, les états minimaux, les flux alternatifs, etc...
- Rechercher l'utilisation répétée d'étapes ou de groupes d'étapes pour éviter le travail répétitif.
- Identifier les zones nécessitant des données aléatoires ou synthétiques.
- Identifier les étapes qui nécessitent des « mocks », des bouchons ou des pilotes pour maintenir l'isolement et éviter d'exécuter des intégrations ou des processus éventuellement coûteux.
- S'assurer que les scénarios sont élémentaires et n'affectent pas l'état les uns des autres (isolé).
- Décider s'il y a lieu de limiter les sections "When / Quand" à une étape conformément au principe selon lequel chaque test ne vérifie qu'une seule chose, ou à optimiser pour d'autres considérations, telles que la vitesse d'exécution du test.

Les lignes directrices recommandées pour formuler des scénarios comprennent :

- Le scénario doit décrire un comportement spécifique que le système prend en charge du point de vue d'un utilisateur spécifique.
- Le scénario devrait utiliser la troisième personne pour décrire les étapes (Given, When, Then) pour décrire l'état et les interactions du point de vue de l'utilisateur.
- Les scénarios doivent être isolés et atomiques afin qu'ils puissent être exécutés dans n'importe quel ordre et ne pas s'affecter les uns les autres ou compter les uns sur les autres. Les étapes Given devraient placer le système dans l'état nécessaire pour que les étapes When s'exécutent de façon cohérente comme prévu.
- Les étapes When doivent décrire les actions sémantiques qu'un utilisateur effectue plutôt que les actions techniques spécifiques, à moins qu'il n'y ait un besoin particulier de tester une action spécifique. Par

exemple, "L'utilisateur confirme l'ordre" (une action sémantique) est généralement préférable à "L'utilisateur clique sur le bouton Confirmer" (une action technique) à moins que le bouton lui-même doive être testé.

- Les étapes Then doivent décrire des observations ou des états spécifiques. Elles ne doivent pas spécifier les états génériques de réussite ou d'erreur.

2.1.3 Acceptance test-driven development (ATDD)

Acceptance test-driven development (ATDD) vise une bonne prise en compte des besoins de clients dans le cycle de développement. Les tests d'acceptation sont des spécifications du comportement et des fonctionnalités souhaitées d'un système. Une User Story exprime un élément de fonctionnalité apportant une valeur au client. Les tests d'acceptation vérifient que cette fonctionnalité est implémentée correctement. Cette User Story est divisée par les développeurs en un ensemble de tâches requises pour implémenter cette fonctionnalité. Les développeurs peuvent implémenter ces tâches en utilisant le TDD. Lorsqu'une tâche donnée est terminée, les développeurs passent à la tâche suivante, et ainsi de suite, jusqu'à ce que la User Story soit terminée, ce qui est indiqué par des tests d'acceptation exécutés avec succès. BDD et ATDD sont tous deux axés sur le client alors que TDD est axé sur le développement. BDD et ATDD sont similaires en ce qu'ils produisent tous deux le même résultat : une compréhension partagée de ce qui doit être construit et comment le construire correctement. BDD est une façon structurée d'écrire des cas de test (p. ex. des tests d'acceptation) en utilisant la syntaxe Given/When/Then vue précédemment.

ATDD sépare l'intention du test, qui est exprimée dans un format lisible par l'homme comme du texte ordinaire, de l'implémentation du test, qui est de préférence automatisée. Cette séparation importante signifie que les membres de l'équipe orientés vers le métier et d'autres membres de l'équipe, non techniques, comme les Product Owners, les analystes et les testeurs, peuvent jouer un rôle actif dans la description d'exemples testables (ou de tests d'acceptation) pour guider le développement. Les parties prenantes ayant des rôles et des perspectives différents, tels que le client, le développeur et le testeur, se réunissent pour discuter en collaboration d'une User Story candidate (ou d'une autre forme d'exigence) afin de parvenir à une compréhension commune du problème à résoudre, de se poser les uns aux autres des questions ouvertes sur la fonctionnalité et d'explorer des exemples concrets et testables de comportements requis.

Les exemples retenus (et les discussions qui y mènent) sont des résultats utiles en eux-mêmes, il est donc important de se rappeler que, bien qu'ils puissent aussi être automatisés comme tests d'acceptation, il n'est pas toujours nécessaire ou rentable de le faire. Cet accent mis sur l'idée d'exemples utiles plutôt que de tests est intentionnel, et est une astuce importante pour encourager tous les membres d'une équipe à s'engager dans le processus de découverte.

Cela illustre un rôle important pour le testeur Agile dans cette situation. Au cours des discussions, le testeur peut penser en termes de techniques de test, telles que les partitions d'équivalence, l'analyse de la valeur des limites, et ainsi de suite. Ils peuvent utiliser cette approche analytique pour encourager les questions à poser au Product Owner sur les zones où les comportements intéressants et les cas limites peuvent être trouvés. L'intention devrait toujours être d'encourager l'ensemble du groupe à considérer et à trouver les exemples clés. Proposer une combinaison intéressante d'entrées et demander si le groupe est d'accord sur les résultats attendus est un bon moyen d'explorer les zones potentielles d'incertitude ou d'analyse incomplète.

La spécification par l'exemple (ou SBE – specification by example) se réfère à une collection de modèles utiles qui permettent à une équipe Agile de découvrir, de discuter et de confirmer les résultats importants requis par les parties prenantes du métier et les comportements logiciels qui sont nécessaires pour atteindre ces résultats. Le terme « spécification par l'exemple » a été largement utilisé pour inclure et élargir le sens du terme acceptance test-driven development (ATDD).

2.2 Tests basés sur l'expérience en Agile

Les diverses caractéristiques des projets Agile telles que l'approche, la longueur de l'itération, le(s) niveau(x) de test applicable(s), le niveau de risque du projet et du produit, la qualité des exigences, le niveau d'expérience/expertise des membres de l'équipe, l'organisation du projet, etc. influenceront l'équilibre entre tests automatisés, tests exploratoires et tests manuels en boîte noire dans un projet Agile.

2.2.1 Combiner des techniques basées sur l'expérience et des tests en boîte noire

Au cours de l'analyse des risques, les niveaux de risque (p. ex. élevés, moyens, faibles) sont déterminés pour les caractéristiques et les fonctionnalités du système. L'étape suivante consiste à trouver le bon mélange et l'équilibre des tests automatisés, des tests exploratoires et des tests manuels en boîte noire pour un niveau de risque spécifique. Voici un tableau décrivant cette idée. Dans ce tableau, les niveaux de risque sont énumérés verticalement et les trois approches de test horizontalement. Les niveaux de recommandation sont définis ainsi :

++ (fortement recommandé)
+ (recommandé)
o (neutre)
- (non recommandé)
-- (à proscrire)

Le tableau ci-dessous est un exemple d'un mélange de différentes techniques de test (exploratoire, automatisée et manuelle) qui peuvent être utilisées dans un système critique de sécurité. Ce tableau peut être adapté à d'autres projets spécifiques.

Niveau de risque	Tests automatisés	Tests exploratoires	Tests en boîte noire
Elevé	++	+	++
Moyen	+	+	+
Faible	o	++	+

Tableau 1 : Systèmes critiques pour la mission et/ou la sûreté

Lorsqu'on examine la première rangée du tableau 1, cela donne à penser que, dans cette situation, une combinaison de tests automatisés et en boîte noire serait fortement recommandée en plus d'une approche de test exploratoire. La décision d'automatiser (ou non) sera également influencée par de nombreux autres facteurs.

Ce qui suit est un exemple d'une combinaison de différentes techniques de test lorsqu'elles sont utilisées pour un système non critique pour la sûreté.

Niveau de risque	Tests automatisés	Tests exploratoires	Tests en boîte noire
Elevé	+	++	+
Moyen	o	++	o

Faible	--	++	--

Tableau 2 : Systèmes non critiques pour la mission ou la sûreté.

Lorsque l'on examine la dernière rangée du tableau 2 ci-dessus, il apparait que, dans cette situation, une approche de tests exploratoires soit fortement recommandée, alors que d'autres approches pourraient ne pas être utilisées. Dans toute situation (critique pour la sûreté ou non) le mélange spécifique dépendra des caractéristiques du projet.

2.2.2 Création de chartes de test et interprétation de leurs résultats

Avant qu'une charte de test appropriée puisse être créée, les Epics existantes et les User Stories doivent être évaluées en premier. Voir le chapitre 1 pour les techniques.

Lors de l'analyse d'Epics ou de User Stories pour créer une charte de test, les éléments suivants doivent être considérés :

- Qui sont les utilisateurs dans cette Epic ou User Story ?
- Quelle est la principale fonctionnalité de l'Epic ou User Story ?
- Quelles sont les actions qu'un utilisateur peut effectuer ? (Cela peut être obtenu à partir de la liste des critères d'acceptation, qui est définie pour une User Story)
- L'objectif de la User Story est-il réalisé une fois la caractéristique (NDT : feature) ou la fonctionnalité terminée ? (Ou y a-t-il d'autres tâches de test affectant la Definition of Done)

La granularité d'une charte de test est importante. Elle ne doit pas être trop petite, car elle doit explorer une zone autour d'un problème identifié (réactive, régression) ou une zone autour d'une User Story ou d'une Epic (proactive, révélatrice).

Elle ne devrait pas être trop grande et devrait tenir dans un laps de temps de 60 à 120 minutes. L'objectif de l'exécution d'une session de test exploratoire est de nous aider à prendre une décision de bonne qualité, en ce qui concerne un problème, une zone de concentration de défauts, etc. Les résultats de cette exploration devraient donner suffisamment d'information pour prendre cette décision.

Les chartes de test peuvent être créés à l'aide de tableaux à feuilles (NDT : flipchart), feuilles de calcul, documents, système de gestion de test existants, de persona, de cartes mentales et d'une approche par l'équipe au complet. Les testeurs exploratoires utilisent les heuristiques pour stimuler leur créativité dans l'écriture et l'exécution de sessions de test exploratoires. Celles-ci peuvent également être utilisées pour créer des chartes de test et stimuler la pensée créative lors de l'analyse des User Stories et des Epics. Des exemples d'heuristiques peuvent être trouvés dans [Whittaker09] et [Hendrickson13].

Tous les constats issus des tests exploratoires doivent être documentés. Les résultats des tests exploratoires devraient fournir un aperçu d'une meilleure conception des tests, des idées pour tester le produit et des idées pour toute autre amélioration future. Les résultats qui devraient être documentés au cours des tests exploratoires comprennent des défauts, des idées, des questions, des suggestions d'amélioration, etc.

Des outils peuvent être utilisés pour documenter les sessions de tests exploratoires. Il s'agit notamment d'outils de capture vidéo et d'enregistrement, d'outils de planification, etc. La documentation devrait inclure le résultat attendu. Dans certains cas, le stylo et le papier sont suffisants, au regard du volume d'informations à collecter.

Lors de la synthèse de la session de tests exploratoires, au cours de la réunion de débriefing, l'information est recueillie et agrégée pour présenter un état des progrès, de la couverture et de l'efficacité de la session. Ces informations sommaires peuvent être utilisées comme rapport pour le management, ou utilisées lors de réunions rétrospectives à n'importe quel niveau et à n'importe quelle échelle (équipe unique, équipes multiples et

implémentation Agile à grande échelle). Cependant, il peut être assez difficile de déterminer des métriques de test pertinentes à partir des séances de tests exploratoires.

2.3 Aspect de la qualité du code

Le contrôle de la dette technique, notamment en termes de maintien de niveaux élevés de qualité du code tout au long de la version, est très important dans les projets Agile. Diverses techniques sont utilisées pour atteindre cet objectif.

2.3.1 Refactoring

Le refactoring est un moyen de nettoyer le code d'une manière efficace et contrôlée, en clarifiant et en simplifiant la conception du code existant et des cas de test, sans changer son comportement. Dans les projets Agile, les itérations sont courtes, ce qui crée une courte boucle de rétroaction pour tous les membres de l'équipe. Les itérations courtes créent également un défi pour les testeurs qui tentent d'obtenir une couverture adéquate. En raison de la nature des itérations, du fait que le produit croît, et que des fonctionnalités sont ajoutées et améliorées au fil du temps, les cas de test qui ont été écrits pour une fonctionnalité dans une itération antérieure, ont souvent besoin de maintenance ou même de refonte complète dans les itérations ultérieures. L'utilisation d'une approche évolutive de conception des tests, la mise à jour et le refactoring des tests peuvent compenser les changements de fonctionnalités et s'assurer que les tests restent alignés avec la fonctionnalité du produit.

Après que les User Stories aient été comprises et que des critères d'acceptation aient été écrits pour chacune d'elles, l'impact de la fonctionnalité issue de l'itération actuelle sur les tests de régression existants (manuels et automatisés) peut être analysé, et le refactoring et/ou l'amélioration des tests peuvent être nécessaires. Les équipes maintiennent et étendent leur code de manière approfondie, itération après itération, et sans un refactoring continu, c'est difficile à faire.

Le refactoring des cas de test peut être fait comme suit :

- **Identification** : Identifier les tests existants qui nécessitent un refactoring par la revue ou l'analyse causale.
- **Analyse** : Analyser l'impact des tests modifiés sur l'ensemble des tests de régression.
- **Refactoring** : Apporter des modifications à la structure interne des tests pour la rendre plus facile à comprendre et moins couteuse à modifier sans changer son comportement observable.
- **Réexécution** : Re-exécuter les tests, vérifier leurs résultats et documenter les défauts lorsque cela est pertinent. Le refactoring ne devrait pas influencer le résultat de l'exécution du test.
- **Évaluation** : Vérifier les résultats des tests re-exécutés, mettre fin à cette phase lorsque les tests ont dépassé un seuil de qualité défini et accepté par l'équipe.

2.3.2 Revue de code et analyse statique du code pour identifier les défauts et la dette technique

Une revue du code est un examen systématique du code par deux personnes ou plus (dont l'une est habituellement l'auteur). L'analyse statique de code est l'examen systématique du code par un outil. Les deux sont des pratiques efficaces et répandues pour exposer et identifier les problèmes qui affectent la qualité du code. Les revues de code et l'analyse statique du code fournissent une rétroaction constructive qui aide à identifier les défauts et à gérer la dette technique.

Des obstacles tels que les contraintes de ressources, une complexité technique plus élevée que prévue, l'évolution rapide des priorités et les limitations techniques peuvent entraver les efforts visant à rédiger du code de qualité et forcer les développeurs à faire des compromis qui diminueront la qualité du code en faveur de résultats plus immédiats. Ces compromis peuvent introduire des défauts et induire une dette technique.

La dette technique fait référence à l'effort accru qui sera nécessaire pour mettre en œuvre une meilleure solution (y compris la suppression des défauts latents) à l'avenir en raison du choix d'une solution moins pertinente mais plus facile à mettre en œuvre maintenant. La dette technique est souvent contractée involontairement, par les compromis réalisés ou par une accumulation progressive de petits changements ou de changements imperceptibles au fur et à mesure de l'évolution du logiciel. Les revues de codes et l'analyse statique du code aident à identifier les différentes causes de la dette technique, telles que la complexité accrue, les dépendances circulaires, les conflits entre les différents modules de code, la mauvaise couverture du code, le code à risque, et ainsi de suite. D'autres types de dettes techniques peuvent également se produire p.ex. sur les artefacts de test, l'infrastructure et le pipeline d'intégration continue.

Lorsque la dette technique est contractée délibérément (à la suite d'autres décisions, ou comme compromis) ou identifiée par des revues de code ou une analyse statique du code, un effort devrait être fait pour réduire la dette technique. Il est préférable d'y faire face immédiatement. Si l'on ne peut pas la traiter immédiatement, des tâches de gestion de la dette technique doivent être ajoutées au backlog (du produit).

Le compromis entre, d'une part, prendre le temps supplémentaire nécessaire pour analyser et revoir le code et, d'autre part, encourir une dette technique, montre presque toujours la valeur apportée par l'analyse du code et les revues. Lorsque le code présente des défauts et que la dette technique grandit, il devient de plus en plus difficile, coûteux et long de corriger sans affecter d'autres parties du système. L'analyse et les revues du code peuvent améliorer la qualité de celui-ci et généralement réduire l'effort global.

En plus d'aider à identifier les défauts et à gérer la dette technique, l'analyse et les revues du code offrent des avantages supplémentaires :

- Formation et partage des connaissances
- Améliorer la robustesse, la maintenance et la lisibilité du code
- Assurer la supervision et maintenir des normes de codage uniformes

Revues du code

En participant à des revues de code, les testeurs peuvent utiliser leur point de vue spécifique pour apporter des contributions précieuses à la qualité du code en collaborant avec les développeurs pour identifier les défauts potentiels et éviter la dette technique à un stade très précoce. Les testeurs doivent être compétents pour lire le langage de programmation utilisé dans le code qu'ils examinent, mais ils n'ont pas besoin d'avoir des compétences de codage significatives pour participer efficacement aux revues de code. Ils peuvent mettre leur expertise à profit de plusieurs façons, par exemple en posant des questions sur le comportement du code, en suggérant des cas d'utilisation qui n'ont peut-être pas été pris en considération, ou en surveillant les mesures du code qui pourraient indiquer des problèmes de qualité. Les revues de codes offrent également l'occasion de partager les connaissances entre les développeurs et les testeurs. L'un des principaux défis des revues de code dans les projets Agile réside dans les itérations courtes et le temps nécessaire à effectuer les revues de code. Il est important de planifier les revues de code et de réserver le temps nécessaire pour les effectuer au cours de chaque itération.

Les revues de codes sont des activités manuelles, éventuellement appuyées par des outils, qui sont effectuées par ou avec d'autres personnes (en plus de l'auteur). Habituellement, les lead développeurs ou les développeurs plus expérimentés de l'équipe effectueront la revue du code, mais elle peut aussi bien être faite avec d'autres membres de l'équipe. Il est souvent bénéfique pour les testeurs et autres non-développeurs de participer à des revues de code.

Différentes approches de revues du code varient en fonction de leur niveau de formalisme et de rigueur [Wiegers02]. Les approches plus formelles et rigoureuses ont tendance à être plus approfondies, mais elles prennent aussi plus de temps. Les approches moins formelles et rigoureuses sont un peu moins approfondies, mais peuvent être beaucoup plus rapides. Il existe différents types de revues par les pairs, et les équipes Agile ont tendance à préférer des revues plus rapides effectuées fréquemment, généralement avant l'intégration.

Les revues de code peuvent être effectuées avec le réviseur et l'auteur/développeur assis côte à côte. Ce mode, qui est courant dans les revues ad hoc et la programmation par paires, facilite une excellente communication et encourage des analyses plus approfondies et un meilleur partage des connaissances. Il peut également contribuer à la cohésion de l'équipe et à son moral.

Pour les équipes distribuées ou les équipes qui préfèrent une approche plus déconnectée, le processus de revue du code est facilité par le système de gestion de configuration. Le processus est généralement partiellement automatisé dans le cadre du processus d'intégration continue. Ces processus peuvent appuyer les revues de code avec un seul réviseur dans chaque cycle de revue, ou par des revues en équipe.

Analyse statique de code

Dans l'analyse statique du code, un outil analyse le code et recherche des problèmes spécifiques sans exécuter le code. Les résultats de l'analyse statique du code peuvent indiquer des problèmes clairs dans le code ou fournir des indicateurs indirects qui nécessitent une évaluation plus approfondie.

De nombreux outils de développement, en particulier des environnements de développement intégrés (NDT : Integrated Development Environments / IDEs), peuvent effectuer une analyse de code statique lors de l'écriture du code. Cela offre l'avantage d'une rétroaction immédiate, même si cela ne concerne qu'un sous-ensemble du code par rapport à l'analyse effectuée lors d'une intégration continue.

3 Automatisation des tests 135 minutes

Mots-clés : Tests pilotés par les données, tests pilotés par mots-clés, procédure de test, approche de test

Objectifs d'apprentissage pour le processus de test

ATT-3.x (K1) Mots-clés

3.1 Techniques d'automatisation des tests

ATT-3.1.1 (K3) Appliquer des techniques de test pilotés par les données et les mots-clés pour développer des scripts de test automatisés

ATT-3.1.2 (K2) Comprendre comment appliquer l'automatisation des tests à une approche de test donnée dans un environnement Agile

ATT-3.1.3-1 (K2) Comprendre l'automatisation des tests

ATT-3.1.3-2 (K2) Comprendre les différences entre les différentes approches de test

3.2 Niveau d'automatisation

ATT-3.2.1-1 (K2) Comprendre les facteurs à prendre en considération pour déterminer le niveau d'automatisation des tests nécessaire pour suivre le rythme des déploiements

ATT-3.2.1-2 (K2) Comprendre les défis de l'automatisation des tests dans les environnements en Agile

3.1 Techniques d'automatisation des tests

3.1.1 Tests pilotés par les données

Motivation

Les tests pilotés par les données sont une technique d'automatisation des tests qui minimise les efforts nécessaires pour développer et maintenir les cas de test qui ont des étapes de test identiques mais qui ont des combinaisons différentes d'entrée de données de test. Les tests pilotés par les données sont une technique bien établie qui n'est pas spécifique aux tests dans les projets Agile. En raison de sa capacité à réduire les efforts de développement de l'automatisation des tests et de maintenance, cette technique devrait être considérée comme faisant partie de la stratégie d'automatisation des tests dans chaque projet Agile.

Concept

L'idée de base des tests pilotés par les données est de séparer la logique du test des données de test. Ainsi, la procédure de test nomme les variables de données de test qui font référence aux valeurs de données de test qui sont fournies dans une liste/tableau de données de test distinct. La procédure de test peut ensuite être exécutée à plusieurs reprises à l'aide de différents ensembles de données de test. Vous trouverez d'autres détails et exemples dans [AdvancedTestAutomationEngineer].

Avantages pour les équipes Agile

- Les équipes Agile peuvent rapidement s'adapter à la fonctionnalité changeante/croissante d'un produit, d'itération en itération, car le changement/l'ajout de nouvelles combinaisons de données est simple et a peu ou pas d'impact sur l'automatisation existante.
- Les équipes Agile peuvent facilement mettre à l'échelle la couverture de test nécessaire vers le haut ou vers le bas en ajoutant, en modifiant ou en supprimant les entrées de la table de données de test. Cela permet également aux équipes Agile de contrôler les temps d'exécution des tests pour répondre aux contraintes de déploiement continus.
- La technique facilite la transversalité entre les rôles, car les tables de données de test sont plus faciles à comprendre que les scripts de test et permettent donc une participation plus efficace des membres de l'équipe moins techniques.
- Comme les tableaux de données de test sont plus faciles à comprendre, cette technique prend en charge la rétroaction précoce sur les cas de test et les critères d'acceptation de la part des membres de l'équipe moins/non techniques et des clients/utilisateurs.
- Cette technique aide les équipes Agile à effectuer plus efficacement les tâches d'automatisation des tests, car non seulement elle réduit l'effort de développement de nouveaux tests, mais elle réduit également la maintenance des tests existants pilotés par les données.

Limitations aux équipes Agiles

Bien qu'il n'y ait pas de limites spécifiques à l'utilisation de cette technique dans un contexte Agile, les équipes Agile doivent être conscientes des limites générales à l'utilisation de cette approche. Plus de détails peuvent être trouvés dans [AdvancedTestAutomationEngineer].

Outils

- L'édition, le stockage et la gestion des données de test se font généralement en utilisant des feuilles de calcul ou des fichiers texte.
- La plupart des outils/langues d'automatisation de test offrent des commandes intégrées pour lire les données de test à partir de feuilles de calcul ou de fichiers texte.

3.1.2 Test piloté par mot-clé

Motivation

Un inconvénient de l'automatisation des tests est que les scripts de test automatisés sont beaucoup plus difficiles à comprendre que les procédures de test manuel décrites en langage naturel. L'application d'une technique d'automatisation des tests pilotée par les mots-clés permet d'améliorer la lisibilité, la compréhension et la maintenance des scripts de test automatisés.

Concept

L'idée de base des tests pilotés par les mots-clés est de définir un ensemble de mots-clés tirés des cas d'utilisation d'un produit et/ou du domaine d'activité du client et de les utiliser comme vocabulaire pour formuler des procédures de test. En raison de leur langage semi-naturel, les scripts de test automatisés qui en résultent sont plus faciles à comprendre que le code en langage de programmation/script.
Vous trouverez d'autres détails et exemples dans [AdvancedTestAutomationEngineer].

Il existe différents styles selon lesquels une procédure de test peut être écrite à l'aide de mots-clés (Voir [Linz 14], Chapter 6.4.2 Keyword-driven Testing) :

- Style en liste : Une procédure de test est une séquence/liste simple de mots-clés ;
- Style BDD : Une procédure de test (ou un scénario) est écrite en langage naturel au moyen d'une une phrase utilisant des mots-clés comme parties « exécutables » de la phrase (voir la section 2.1.2 (BDD)) ;
- Style DSL : Une grammaire formelle basée sur le concept de langage spécifique au domaine ou "domain-specific language" (DSL) (voir [Fowler/Parsons10]) définit comment les mots-clés (et potentiellement des éléments de langage supplémentaires) peuvent être combinés.

Avantages pour les équipes Agile

- En définissant des mots-clés ou un DSL, une équipe Agile crée et standardise son vocabulaire propre lié au domaine, ce qui aide les membres de l'équipe à communiquer plus clairement et avec précision, ainsi qu'à éviter les malentendus.
- Les équipes Agile peuvent rapidement recueillir des commentaires plus qualifiés des clients/utilisateurs. Les procédures de test composées par mots-clés et/ou écrites dans le style BDD/DSL peuvent être mieux comprises par les clients que le code de test en langage de programmation ordinaire. Les critères d'acceptation informels peuvent être formalisés sans perdre la « lisibilité du langage naturel ». Cela peut aider à acquérir une compréhension de la logique métier elle-même et donc de l'interprétation des critères d'acceptation.
- La façon dont les cas de test sont créés contribue à la création et à la gestion active de la documentation
- La technique aide les équipes Agile à accomplir leurs tâches d'automatisation des tests de manière transverse en incluant les membres les moins techniques, puisque la composition des procédures de test à partir de mots-clés existants ne nécessite pas de compétences en programmation.
- Changer le comportement d'un mot-clé défini nécessite beaucoup moins d'efforts que de changer le même comportement à travers plusieurs procédures de test. Cela peut réduire considérablement les efforts de maintenance et libérer des ressources pour consacrer du temps à la mise en œuvre de nouveaux tests.
- Une équipe Agile peut appliquer la technique (et ainsi en obtenir les avantages) sur l'ensemble de la pyramide de test parce que les mots-clés peuvent être implémentés pour piloter l'objet de test à différents niveaux d'interface : du niveau d'interface de test jusqu'au niveau API (p.ex. via des appels API, des appels REST, des appels Soap etc.). Cela signifie que le concept est également applicable pour les cas de test unitaires et d'intégration et ne se limite pas uniquement aux tests système via l'interface utilisateur, comme on le suppose souvent. Cependant, il peut ajouter un niveau d'abstraction inutile aux tests unitaires.

Limites de l'approche dans le contexte des équipes Agiles

Outre les limites générales de cette approche décrites dans [AdvancedTestAutomationEngineer], Les équipes agiles doivent également être conscientes des limites et des pièges potentiels suivants :

- Pour exécuter des procédures de test pilotées par les mots-clés, un framework d'exécution approprié (par exemple, un interpréteur de mots-clés) est nécessaire. Il n'est pas recommandé de réinventer le framework à partir de zéro. L'équipe gagnera en vitesse si elle utilise un framework ou un outil existant qui prend en charge les tests pilotés par mots-clés. C'est particulièrement le cas si l'équipe adopte un style BDD ou DSL.
- La mise en œuvre d'un nouveau mot-clé d'une manière stable et maintenable est difficile et nécessite de l'expérience et de bonnes compétences en programmation. Les tâches d'implémentation de mots-clés sont également en concurrence avec les tâches de codage de produits. La vitesse d'automatisation des tests qui en résulte pourrait donc être plus faible que prévu.
- L'ensemble des mots-clés (nommage, niveau d'abstraction) et/ou le DSL (règles de grammaire) doit être bien conçu. Sinon, la compréhension et/ou l'évolutivité ne répondront pas aux attentes.
    - L'ensemble de mots-clés doit également être correctement géré. Si cela n'est pas fait, la mise en œuvre par mots-clés ne sera pas payante parce que plusieurs implémentations de synonymes pourront se produire, ou parce que des mots-clés ne seront pas ou seront rarement utilisés par les cas de test. Pour éviter ces pièges, l'équipe doit nommer un membre de l'équipe responsable de la gestion du vocabulaire des mots-clés.
- L'équipe doit être consciente que l'application de tests pilotés par mots-clés nécessite un investissement initial (par exemple pour définir les mots-clés/langage de domaine, choisir un framework approprié, mettre en œuvre le premier ensemble de mots-clés) et qu'ainsi au début d'un projet, la vitesse d'automatisation des tests peut être réduite.

Outils

- Comme pour les tests pilotés par les données, une approche courante mais limitée consiste à utiliser des feuilles de calcul, du texte ou des fichiers plats pour l'édition, le stockage et la gestion de tests pilotés par mots-clés.
- Plusieurs frameworks de test, outils d'exécution de tests et outils de gestion de test offrent un support intégré pour les tests par mots-clés (nommés « tests par mots-clés », « tests basés sur l'interaction », « tests de processus métier » ou « tests axés sur le comportement », selon le fournisseur ou le framework de l'outil. Les outils disponibles peuvent être trouvés dans [ToolList].

3.1.3 Appliquer l'automatisation des tests à une approche de test donnée

L'automatisation des tests n'est pas un objectif de test. L'automatisation des tests est une stratégie qui, lorsqu'elle est poursuivie de façon appropriée, peut promouvoir des objectifs de test stratégiques plus grands en augmentant l'efficacité des tests, en rendant les tests efficaces contre certains types de défauts (p. ex., les défauts de performance et de fiabilité), ou en permettant la découverte anticipée de défauts. La stratégie dépend du contexte et évolue donc continuellement.

L'automatisation des tests prend souvent de nombreuses formes différentes et implique de nombreux outils différents, selon le contexte et les besoins de l'équipe. Dans les grands projets, il n'y a généralement pas qu'une seule solution qui répond à tous les besoins, et donc plusieurs stratégies d'automatisation des tests sont pertinentes. L'application de l'automatisation des tests doit être adaptée à la stratégie de test de l'organisation et à l'approche de test d'un projet donné.

L'automatisation des tests est plus que l'automatisation de l'exécution des tests. L'automatisation des tests peut jouer un rôle important dans la configuration de l'environnement de test, l'acquisition de version de test, la gestion des données de test et la comparaison des résultats des tests, pour ne donner que quelques exemples. Lors de la planification et de la conception de l'utilisation de ces outils, considérez l'approche de test, les implications du cycle de vie Agile mis en œuvre, les capacités de ces outils d'automatisation des tests, et l'intégration de divers

autres outils avec ces outils d'automatisation des tests (p. ex., l'automatisation des tests dans le cadre de l'intégration continue).

Dans certains cas, l'automatisation des tests sert directement les objectifs d'une itération ; c'est-à-d. la construction de nouvelles fonctionnalités. Dans d'autres cas, l'automatisation des tests prend en charge ces objectifs indirectement ; par exemple, en réduisant le risque de régression associé aux changements apportés au système. Comme nous l'avons vu dans le syllabus Agile de Niveau Fondation, certaines organisations choisissent de mettre ces efforts de soutien d'automatisation des tests dans des équipes en dehors des équipes réalisant l'itération elles-mêmes. Dans de telles organisations, vous pouvez par exemple voir une équipe distincte fournir la création et la maintenance du framework d'automatisation des tests de régression en tant que service pour plusieurs équipes réalisant les itérations. Cette approche peut être couronnée de succès si l'équipe extérieure fournit des services utiles aux équipes réalisant les itérations qui aident ces dernières à se concentrer sur leurs objectifs d'itération immédiats. En fonction des cas, les équipes externes peuvent influencer l'engagement de l'équipe, car elles transfèrent (en partie) le contrôle sur leur engagement.

Voici des exemples considérant l'automatisation des tests pour les principales approches de test telles qu'énoncées dans les syllabi Fondation ISTQB®, Advanced Test Manager et Expert Test Manager :

Analytique : Behavior-driven development (BDD) et acceptance test-driven development (ATDD) sont des techniques qui peuvent être utilisées dans le cadre d'une approche analytique dans un contexte Agile, par exemple appliquées à l'automatisation des tests. BDD et ATDD peuvent être utilisés pour produire des tests automatisés en parallèle de (ou même avant) la mise en œuvre de la User Story.

Basée sur les modèles : Des tests de comportement fonctionnel basés sur des modèles peuvent prendre en charge la création automatisée de tests pendant la mise en œuvre de la User Story, fournissant une source rapide de tests efficaces. Les tests basés sur des modèles peuvent également être utilisés pour la création de User Stories, car les modèles peuvent être utilisés pour tester les exigences et faciliter les revues statiques. Les modèles sont souvent utilisés pour tester les comportements non fonctionnels tels que la fiabilité et la performance, qui sont des caractéristiques importantes pour de nombreux systèmes et qui sont des propriétés émergentes du système dans son ensemble.

Méthodique : Étant donné qu'il existe de courtes itérations multiples dans les projets Agile, des checklists de test automatisées (avec toutes les activités) peuvent être utilisées comme une approche méthodique pour l'exécution efficace d'un ensemble stable de tests.

Conforme au processus : Pour les projets qui doivent être conformes à des normes ou règlements définis à l'externe, ces normes ou règlements peuvent influer sur la façon dont les tests automatisés sont utilisés ou sur la façon dont les résultats automatisés des tests sont saisis. Par exemple pour les projets régis par la FDA (Food & Drug Administration[5] ; donc avec risque élevé), les tests automatisés et leurs résultats doivent être traçables et liés aux exigences, et les résultats doivent contenir suffisamment de détails pour prouver que le test a été réussi.

Réactif (ou heuristique) : Les tests réactifs jouent un rôle de validation important dans les tests Agile, tandis que la plupart des tests automatisés jouent principalement un rôle de vérification. Bien que les stratégies réactives soient principalement manuelles (p. ex., tests exploratoires, estimation d'erreurs, etc.), une proportion accrue de la couverture des tests automatisés entraîne souvent un plus grand degré de tests manuels qui suivent des stratégies réactives puisque bon nombre des tests qui peut être préparé à l'avance sera automatisé. En outre, les tests manuels restants peuvent couvrir les zones plus risquées.

Dirigé (ou consultatif) : Lorsque la couverture du test est spécifiée par des parties prenantes externes et que l'automatisation des tests doit être utilisée, l'exigence vis-à-vis de l'équipe de test pour répondre à la demande est

[5] Note du Traducteur – FDA – « Agence américaine des produits alimentaires et médicamenteux »

importante. Par conséquent, les équipes de test suivant une stratégie de test dirigée devraient tenir compte à la fois du temps et des compétences nécessaires pour accomplir leurs tâches dans les itérations.

Anti-régressions : Dans les projets Agile, l'une des principales caractéristiques de la stratégie prévenant la régression est un ensemble important, stable et croissant de tests de régression automatisés. Une couverture adéquate, la maintenabilité et une analyse efficace des résultats sont essentielles, d'autant plus que le nombre de tests de régression augmente. Plutôt que de se concentrer sur un ensemble sans cesse croissant de tests de régression, une approche réussie prévenant la régression se concentre sur l'amélioration continue et le refactoring des tests créés.

3.2 Niveau d'automatisation

3.2.1 Comprendre le niveau nécessaire d'automatisation des tests

L'automatisation est un élément important dans les projets Agile car elle couvre non seulement l'automatisation des tests, mais aussi l'automatisation du processus de déploiement (cf. Chapitre 4). Le déploiement continu est le déploiement automatique d'une nouvelle version dans l'environnement de production. Le déploiement continu a lieu à intervalles réguliers et courts.

Étant donné que le déploiement continu est un processus automatisé, les tests automatisés doivent être suffisants pour maintenir le niveau requis de qualité du code. Le simple fait d'exécuter des tests unitaires automatisés ne suffit pas à obtenir une couverture de test suffisante. Une suite de test automatisée, exécutée dans le cadre du processus de déploiement, doit également inclure des tests d'intégration et de niveau système. Dans la pratique, certains tests manuels peuvent encore être nécessaires.

Voici quelques défis rencontrés avec l'automatisation des tests dans un cadre Agile :

- **Volume de la suite de test** : Chaque itération (en plus des itérations de maintenance ou de durcissement) met en œuvre des fonctionnalités supplémentaires. Pour maintenir la suite de test en ligne avec les fonctionnalités supplémentaires du produit, l'équipe Agile doit améliorer sa suite de test dans chaque itération. Cela signifie que le nombre de cas de test dans la suite de test augmente par défaut d'une itération à l'autre. Cela implique des efforts prudents et délibérés pour effectuer le refactoring des tests afin d'augmenter la couverture sans augmenter considérablement la taille. Quoi qu'il en soit, l'effort et le temps nécessaires pour maintenir, préparer et exécuter la suite de test complète augmenteront avec le temps.
- **Temps de développement des tests** : Les tests nécessaires pour vérifier la fonctionnalité nouvelle ou modifiée du produit doivent être conçus et implémentés. Cela comprend la création ou la mise à jour des données de test nécessaires et la préparation des mises à jour de l'environnement de test. La maintenabilité des tests affecte également le temps de développement.
- **Temps d'exécution des tests** : L'augmentation du volume des suites de tests entraînera un temps croissant nécessaire à l'exécution des tests.
- **Disponibilité du personnel** : Le personnel nécessaire à la création, à la maintenance et à l'exécution de la suite de test doit être disponible pour chaque déploiement. Il peut être difficile, voire impossible, de l'assurer au cours du projet, surtout pendant les vacances, les fins de semaine ou si des déploiements ont lieu en dehors des heures de travail.

Une stratégie pour relever ces défis consiste à minimiser la suite de tests en sélectionnant, en préparant et en exécutant seulement un sous-ensemble de tests en priorisant les tests et/ou en utilisant l'analyse des risques. Cette stratégie a l'inconvénient d'augmenter les risques car elle implique un nombre réduit de tests exécutés.

L'automatisation du plus grand nombre possible de tests permet d'augmenter la fréquence et/ou le rythme des déploiements.

L'automatisation des tests est un instrument opérationnel pour suivre ou augmenter le rythme de déploiement dans n'importe quel projet et est la prémisse d'un déploiement continu. Pour trouver la bonne quantité d'automatisation des tests qui peut suivre le rythme de déploiement, les avantages et les limitations suivants doivent être analysés et pris en compte de façon équilibrée :

Avantages

- L'automatisation des tests peut garantir un niveau défini et répétable de couverture de test pour chaque cycle de déploiement.
- L'automatisation des tests peut réduire le temps d'exécution des tests et aider à augmenter la vitesse de déploiement.
- L'automatisation des tests peut diminuer les limites pour atteindre une fréquence de déploiement plus élevée.
- Le déploiement continu (CD) permet de raccourcir le temps de mise sur le marché et les boucles de rétroaction des utilisateurs.
- L'automatisation des tests peut être plus fréquente, ce qui permet d'exécuter tous les tests à chaque build, offrant une branche principale stable en intégration continue, avec fusion des développements dans la branche principale aussi souvent que possible.

Limitations

- L'automatisation des tests nécessite des efforts de développement et de maintenance des tests, qui peuvent eux-mêmes allonger la durée du développement et ainsi diminuer la fréquence de déploiement.
- Les tests au niveau du système et en particulier les tests de charge ou de performance (et peut-être d'autres tests non fonctionnels) peuvent nécessiter un long délai d'exécution, même s'ils sont automatisés.
- Les tests automatisés peuvent échouer en raison de nombreux facteurs. Un cas de test automatisé réussi peut ne pas être fiable (faux négatif). Un cas de test automatisé échoué peut échouer en raison d'une erreur non liée à la qualité des produits ou en raison d'un faux positif.

La fréquence de livraison doit correspondre aux besoins des utilisateurs du produit. Trop de versions livrées en trop peu de temps pourrait plus déplaire au client qu'une fréquence plus lente. Par conséquent, dans les situations où il est techniquement possible d'augmenter davantage la fréquence de déploiement, ceci pourrait ne pas être perçu comme un avantage supplémentaire du point de vue du client.

4 Déploiement et livraison 105 minutes

Mots-clés : Virtualisation de service, tests en continu

Objectifs d'apprentissage pour le déploiement et la livraison

ATT-4.x (K1) Mots-clés

4.1 Intégration continue, tests continus et livraison continue

ATT-4.1.1 (K3) Appliquer l'intégration continue (CI) et résumer son impact sur les activités de test

ATT-4.1.2 (K2) Comprendre le rôle des tests en continu dans la livraison continue et le déploiement continu (CD)

4.2 Virtualisation

ATT-4.2.1-1 (K2) Comprendre le concept de virtualisation des services et son rôle dans les projets Agile

ATT-4.2.1-2 (K2) Comprendre les avantages de la virtualisation des services

4.1 Intégration continue, tests continus et livraison continue

4.1.1 L'intégration continue et son impact sur les tests

L'objectif de l'intégration continue (CI) est de fournir une rétroaction rapide de sorte que si des défauts sont introduits dans le code, ils sont trouvés et corrigés dès que possible. Les testeurs Agiles devraient contribuer à la conception, à la mise en œuvre et à la maintenance d'un processus d'intégration continue efficace et efficient, non seulement en termes de création et de maintenance de tests automatisés qui s'inscrivent dans le framework d'intégration continue, mais aussi en termes de priorisation des tests, des environnements nécessaires, des configurations, et ainsi de suite.

Dans une situation idéale en intégration continue, une fois que le code est conçu, tous les tests automatisés sont exécutés à la suite pour vérifier que le logiciel continue à se comporter comme défini et n'a pas été endommagé par les modifications de code. Cependant, il y a deux objectifs contradictoires :

- Exécuter le processus d'intégration continue fréquemment pour obtenir des retours immédiats sur le code
- Vérifier le code aussi soigneusement que possible après chaque build

Lorsque la conception, la mise en œuvre et la maintenance des tests automatisés ne sont pas suffisamment prises en compte (comme nous l'avons vu dans le chapitre précédent), l'exécution de tous les tests automatisés prendra trop de temps pour que le processus d'intégration continue soit complété plusieurs fois par jour. Même avec une automatisation minutieuse des tests, il se peut que l'exécution complète de tous les tests automatisés à travers tous les niveaux de test ralentisse excessivement le processus d'intégration continue. Différentes organisations ont des priorités différentes et différents projets ont besoin de solutions différentes pour trouver le bon équilibre entre les objectifs mentionnés ci-dessus. Par exemple, si un système est stable et change moins fréquemment, il requerra moins de cycles d'intégration continue. Si le système est constamment mis à jour, alors, très probablement, plus de cycles d'intégration continue seront nécessaires.

Il existe des solutions qui soutiennent les deux objectifs, qui sont complémentaires et peuvent être utilisées en parallèle.

La première consiste à prioriser les tests afin que les tests de base et les plus importants soient toujours exécutés à l'aide d'une approche de test basée sur le risque.

La deuxième solution consiste à permettre d'utiliser différentes configurations de test dans le processus d'intégration continue pour différents types de cycles d'intégration continue. Pour le processus quotidien de build et de test, seuls les tests de base sélectionnés sur la base de la priorisation initiale sont exécutés. Pour le processus d'intégration continue de nuit, un plus grand nombre voire peut-être tous les tests fonctionnels qui ne nécessitent pas l'environnement de pré-production sont exécutés. Avant la livraison, des tests fonctionnels et non fonctionnels plus approfondis sont effectués dans un environnement de pré-production avec des entrées utilisateur réelles, y compris des tests d'intégration avec les bases de données, différents systèmes et / ou plates-formes.

La troisième solution consiste à accélérer l'exécution des tests en diminuant la quantité de tests d'interface utilisateur (UI). Généralement, il n'y a pas de problèmes de temps pour terminer l'exécution des tests unitaires/d'intégration, qui s'exécutent généralement très rapidement. Il peut toutefois survenir qu'il existe tant de tests unitaires qu'ils ne peuvent pas être exécutés complètement pendant le processus d'intégration continue. Les vrais problèmes de temps sont généralement liés à l'utilisation des cas de test UI de bout en bout dans le cadre du processus d'intégration continue. La solution consiste à augmenter la quantité d'API, de ligne de commande, de couche de données, de couche de service et autres tests non UI de logique métier, ainsi qu'à diminuer les tests d'interface utilisateur, en poussant ainsi l'effort d'automatisation vers le bas de la pyramide d'automatisation des tests. Cela garantit non seulement plus de tests maintenables, mais réduit également le temps d'exécution du test.

La quatrième solution peut être utilisée lorsque les exécutions de test sont très fréquentes dans un système d'intégration continue et qu'il est impossible d'exécuter tous les cas de test. Sur la base des modifications apportées au code et de la connaissance de la trace d'exécution des cas de test existants, un développeur ou un testeur peut ne sélectionner et exécuter que les cas de test affectés par les modifications (c.-à-d., l'utilisation de l'analyse d'impact pour sélectionner des tests). Étant donné que seule une petite fraction de l'ensemble de la base de code est modifiée au cours d'un cycle court, relativement peu de cas de test doivent être exécutés. Cependant, d'importants défauts de régression pourraient ne pas être détectés.

Une cinquième solution consiste à diviser la suite de test en morceaux de taille égale et à les exécuter en parallèle sur plusieurs environnements (agents, ferme de builds, cloud). Ceci est très couramment appliqué dans les entreprises utilisant l'intégration continue, parce qu'ils ont déjà besoin de beaucoup de capacité de serveur de builds.

L'intégration continue fonctionne sur une variété de plates-formes technologiques. Comme les outils de développement et de déploiement se sont déplacés vers le cloud, les produits d'intégration continue aussi. Alors que la plupart des produits d'intégration continue sont encore conçus pour être téléchargé et exécuté dans des environnements locaux, le cloud a créé un nouveau type de produits qui fournissent des services d'intégration continue sur les plates-formes hébergées que les équipes peuvent commencer à utiliser rapidement. Ainsi, les équipes peuvent éviter les coûts et le temps superflus de créer un nouvel environnement, de télécharger, d'installer et de configurer le logiciel d'intégration continue. En se déplaçant vers le cloud, l'équipe peut configurer et (directement) commencer à travailler. Dans la pratique, le cloud est un moyen flexible pour accélérer la conception et l'exécution des tests si nécessaires.

Les outils d'intégration continue actuels prennent en charge non seulement l'intégration continue, mais aussi la livraison continue et le déploiement continu. L'exécution de tests automatisés dans un environnement qui ne reproduit pas complètement la production peut conduire à de faux négatifs, mais, d'autre part, le clonage de l'environnement de production peut constituer un coût excessif. Les solutions incluent l'utilisation d'environnements de test dans le cloud qui reproduisent les environnements de production sur une base nécessaire, ou encore permettent la création d'un environnement de test qui est une version de celui de la production, à l'échelle mais réaliste.

Les bons systèmes d'intégration continue doivent être en mesure de se déployer automatiquement sur des environnements plus complexes et sur des plates-formes différentes. La tâche des testeurs est de planifier les cas de test à inclure, de prioriser ceux qui doivent être exécutés sur certaines ou sur toutes les plates-formes afin de maintenir une bonne couverture, et de concevoir des cas de test pour valider efficacement le logiciel dans un environnement similaire à celui de la production.

4.1.2 Le rôle des tests en continu dans la livraison et le déploiement continus (CD)

Les tests en continu sont une approche qui implique un processus consistant à tester tôt, tester souvent, tester partout et à automatiser pour obtenir des retours sur les risques métier associés à une version candidate à la livraison aussi rapidement que possible. Lors des tests en continu, une modification apportée au système déclenche les tests requis (c.-à-d. les tests qui couvrent la modification) qui s'exécutent automatiquement, ce qui donne au développeur un retour rapide. Les tests en continu peuvent être appliqués dans différentes situations (par exemple, dans l'IDE, dans l'intégration continue, dans la livraison continue et le déploiement continu, etc.), quoi qu'il en soit, le concept est le même, c'est-à-dire d'exécuter automatiquement des tests le plus tôt possible.

La livraison continue nécessite une intégration continue. La livraison continue prolonge l'intégration continue en déployant toutes les modifications de code dans un environnement de test ou de pré-production et/ou un environnement similaire à la production après l'étape de construction. Ce processus de « staging » permet d'exécuter des tests fonctionnels en appliquant des entrées-utilisateur réelles et des tests non fonctionnels tels que des tests de charge, de stress, de performance et de portabilité. Étant donné que chaque changement intégré est promu dans l'environnement de « staging » de manière complètement automatisée, l'équipe Agile est assurée que

le système pourra être déployé en production en appuyant sur un bouton lorsque la définition du terminé (Definition of Done) est atteinte.

Le déploiement continu va plus loin dans la livraison continue en énonçant l'idée que chaque changement est automatiquement déployé en production. Le déploiement continu vise à minimiser le temps écoulé entre la tâche de développement de l'écriture d'un nouveau code et son utilisation par les utilisateurs réels, en production. Pour plus d'informations, voir [Farley].

4.2 Virtualisation de service

La virtualisation des services désigne le processus de création d'un service de test partageable (un service virtuel) qui simule le comportement, les données et les performances pertinents d'un système ou d'un service connecté. Ce service virtuel permet aux équipes de développement et de test d'exécuter leurs tâches même si le service réel est encore en cours de développement ou non disponible.

Il est difficile d'effectuer des tests précoces dans le cycle de développement avec les équipes et les systèmes de développement hautement connectés et interdépendants d'aujourd'hui. En découplant les équipes et les systèmes au moyen de la virtualisation de service, les équipes peuvent tester leur logiciel plus tôt dans le cycle de vie du développement en utilisant des cas d'utilisation et des charges plus réalistes.

Alors que les bouchons, les pilotes et les mocks sont utiles pour permettre des tests précoces, les bouchons créés manuellement sont généralement sans état et fournissent des réponses simples aux demandes avec des temps de réponse fixes. Les services virtuels peuvent prendre en charge des transactions présentant un état et maintenir le contexte des éléments dynamiques (ID de session/client, dates et heures, etc.) ainsi que des temps de réponse variables qui modélisent le flux de message à travers plusieurs systèmes.

Les services virtuels sont créés à l'aide d'un outil de virtualisation des services, souvent de l'une des façons suivantes :

- En interprétant à partir de données : fichier XML, données historiques provenant de logs de serveurs ou simplement exemple de feuille de calcul de données
- En surveillant le trafic réseau : l'exécution du système sous test (NDT : SUT - System under test) déclenchera un comportement pertinent du système dépendant qui est capturé et modélisé par l'outil de virtualisation de service
- En capturant à partir d'agents : la logique du côté serveur ou interne peut ne pas être visible sur les messages de communication, forçant les données et les messages pertinents qui doivent être poussés à partir d'un serveur dépendant à être recréés en tant que service virtuel

Si aucune de ces approches ne s'applique, le service virtuel doit être créé au sein de l'équipe de projet, sur la base du protocole de communication approprié.

Les avantages de la virtualisation des services comprennent :

- Activités parallèles de développement et de test pour le service en cours de développement
- Test anticipé des services et des APIs
- Configuration anticipée des données de test
- Compilation, intégration continue et automatisation des tests en parallèle
- Découverte de défauts plus tôt
- Réduction de la surutilisation des ressources partagées (composants sur étagère, mainframe, datawarehouses)
- Réduction des coûts de test en réduisant l'investissement dans l'infrastructure.
- Permettre des tests non fonctionnels précoces du système sous test

- Simplification de la gestion de l'environnement de test
- Moins d'effort de maintenance pour les environnements de test (pas besoin de maintenir le middleware)
- Diminution du risque pour la gestion des données (ce qui contribue à la conformité au RGPD)

Notez que le service virtuel n'a pas besoin d'inclure toutes les fonctionnalités et les données du service réel, mais seulement les parties nécessaires pour tester le système sous test.

L'introduction d'un service de virtualisation peut être complexe et potentiellement coûteuse. L'introduction d'un outil de virtualisation de service devrait être traitée de la même manière que l'introduction de tout nouvel outil de test dans l'équipe et dans l'organisation. Ce sujet est abordé dans le syllabus Fondation ISTQB® et dans le syllabus Avancé Test Manager ISTQB®.

5 Références

[AgileFoundationExt] ISTQB/CFTL – Syllabus Niveau fondation –Testeur Agile, 2014. Disponible sur le site du CFTL (www.cftl.fr).

[Foundation] ISTQB/CFTL – Syllabus Niveau Fondation - Testeur Certifié, 2018. Disponible sur le site du CFTL (www.cftl.fr).

[AdvancedTestAutomationEngineer] ISTQB/CFTL – Syllabus Niveau Avancé - Automatisation de test – Ingénierie, 2016. Disponible sur le site du CFTL (www.cftl.fr).

[ISO 29119-5], ISO/IEC/IEEE 29119-5 Software and systems engineering -- Software testing -- Part 5: Keyword-Driven Testing

Livres en Français[6]

[CFTL_Agile] Ouvrage collectif, "Les tests logiciels en Agile", Edition Les Livres du CFTL, 2019. ISBN : 978-2956749004.

[Moustier19] Christophe Moustier, "Le test en mode Agile", Editions ENI, 2019. ISBN : 978-2409019432.

[Hage19] Marc Hage Chahine, "Tout sur le test logiciel - Préparation à la certification ISTQB / Profession testeur", Editions ELLIPSES, 2019. ISBN : 978-2340030213.

Livres

[Adzic09] Adzic, Gojko, "Bridging the Communication Gap" Neuri Ltd, 2009

[Adzic11] Adzic, Gojko. "Specification by Example" Manning, 2011.

[Beck02] Kent Beck, "Test Driven Development: By Example", 2002.

[Beck99] Kent Beck, "Extreme Programming Explained" Addison Wesley, 1999.

[Carkenord08] Barbara A. Carkenord, "Seven Steps to Mastering Business Analysis", J. Ross Publishing, 2008.

[Cohn 09] Mike Cohn: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 2009.

[Crispin08] Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams. Crawfordsville : Addison-Wesley Professional

[Elfriede99] Dustin Elfriede, Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance, Addison-Wesley Professional, 1999

[Evans03] Eric Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", 2003

[6] Note de traduction – Cette rubrique est un ajout pour la version Française.

[Fewster12] Mark Fewster, Dorothy Graham, Experiences of Test Automation: Case Studies of Software Test Automation, Addison-Wesley Professional, 2012

[Fowler/Parsons 10], Martin Fowler, Rebecca Parsons, Domain-Specific Languages, Addison-Wesley Signature, Series, 2010

[Hendrickson13] Elisabeth Hendrickson, "Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing", Pragmatic Bookshelf, 2013

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, "Extreme Programming Installed," Addison-Wesley Professional, 2000

[Jorgensen13] Paul C. Jorgensen, Software Testing: A Craftsman's Approach, Auerbach Publications; 4th edition, 2013

[Linz14] Tilo Linz, Testing in Scrum, A Guide for Software Quality Assurance in the Agile World, Rocky Nook, 2014

[Meszaros07] Gerard Meszaros, "xUnit Test Patterns: Refactoring Test Code", Addison-Wesley, 1st Edition, Apress, 2007

[Michelsen12] John Michelsen and Jason English, "Service Virtualization: Reality is Overrated", Apress, 1st Edition, 2012.

[Osherove09] Roy Osherove, "The Art of Unit Testing", 2009

[Paskal15] Greg Paskal, "Test Automation in the Real World: Practical Lessons for Automated Testing", MissionWares, 2015

[Smart15]; Smart, John Ferguson; "BDD in action", Manning, 2015

[Wein89] Weinberg, Gerald & Gause, Donald. "Exploring Requirements: Quality Before Design" Dorset House, 1989.

[Whittaker09] James A Whittaker, "Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design", Addison-Wesley Professional, 2009

[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide (Paperback)", 2002.

Terminologie Agile
Les mots-clés qui se trouvent dans le Glossaire ISTQB® sont identifiés au début de chaque chapitre. Pour les termes Agiles courants, nous nous sommes appuyés sur les ressources Internet reconnues suivantes qui fournissent des définitions :https://www.agilealliance.org/agile101/agile-glossary/
En cas de définitions contradictoires, le Glossaire ® ISTQB est la principale source.

Autres références
Les références suivantes indiquent des informations disponibles sur Internet et ailleurs. Même si ces références ont été vérifiées au moment de la publication de ce syllabus, l'ISTQB® ne peut être tenue responsable si les références ne sont plus disponibles.

[Cohn09] The Forgotten Layer of the Test Automation Pyramid,

https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[CyclomaticComplexity] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[Farley] http://www.davefarley.net/?cat=5

[DSL] https://en.wikipedia.org/wiki/Domain-specific_language

[Fowler07] https://martinfowler.com/articles/mocksArentStubs.html

[Fowler04] Fowler, Martin. https://martinfowler.com/bliki/SpecificationByExample.html

[Fowler03] Martin Fowler, https://martinfowler.com/bliki/TechnicalDebt.html

[Gherkin] https://docs.cucumber.io/gherkin/

[INVEST] Bill Wake, "INVEST in Good Stories, and SMART Tasks", http://xp123.com/articles/investin-good-stories-and-smart-tasks/

[IQBBA] International Qualifications Board for Business Analysts, http://www.iqbba.org/

[IREB] International Requirements Engineering Board, https://www.ireb.org/en

[IIBA] International Institute of Business Analysts (IIBA), https://www.iiba.org/

[Marick01] Marick, Brian, http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Marick03] Marick, Brian. http://www.exampler.com/old-blog/2003/08/22.1.html

[North06] Dan North, "Introducing BDD", blog post, 2006

[TESTDOUBLES] Wojciech Bulaty, Bill Wake, "Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams", https://www.infoq.com/articles/stubbing-mockingservice-virtualization-differences

[ToolList] Examen des outils de test, plate-forme d'information sur le marché international des outils de test logiciel, www.testtoolreview.de/en/

[xUnit] https://en.wikipedia.org/wiki/XUnit, https://en.wikipedia.org/wiki/Unit_testing

6 Annexes

6.1 Glossaire des termes spécifiques de Testeur Technique Agile

Terme glossaire	Définition
Persona	Un personnage fictif représentant un certain type d'utilisateurs et comment ils interagiront avec le système.
Storyboard	Représentation visuelle du système dans lequel les User Stories sont représentées dans leur contexte dans le but de comprendre les processus métier.
Story Mapping	Une technique ordonnant des User Stories sur deux dimensions, l'axe horizontal représentant leur ordre d'exécution, et l'axe vertical représentant le degré d'affinage du produit mis en œuvre.