**Audit de Maturité  
& Stratégie de Test**



**pour**



|  |  |  |
| --- | --- | --- |
| Rédacteur : | Zied Hannachi - Consultant | ALL4TEST |
| Validation : | Julien Van Quackebeke | ALL4TEST |
| Destinataires : | Julien Cavaggion | ORDIPAT |
| Date : | 18/01//2019 |  |

Table des matières

[**Gestion de la Qualité**](#_n11w3qciqm8w) **3**

[Objectifs](#_eqdaa21spzn4) 3

[Contexte](#_drs3v5gn6d13) 3

[Périmètre](#_hn65tcongjb2) 4

[Enjeux](#_kp5ejbzbdhv3) 4

[À l'échelle de la société](#_me5mpm59twnh) 4

[À l'échelle des projets ou produits](#_xbvd0kfn6lxr) 4

[À l'échelle des testeurs et développeurs](#_ibmdbeo4bolk) 4

[État de l’art](#_dori766jhm66) 5

[Objectifs de projet](#_ze1hqtxjaefz) 5

[Shift Left Testing & Shift Right Testing](#_gz2yjq95fz1z) 6

[Qualité des processus](#_xral96geximc) 8

[**Automatisation des tests**](#_yf076h7wa4j7) **9**

[Niveaux de test](#_jcocbzctvswa) 9

[Tests Unitaires](#_bb6nlrx6mdya) 10

[Avantages des tests unitaires](#_iy3cmodlqzti) 10

[Tests d’Intégration](#_8alncpo9iezb) 10

[Avantages des tests d'intégration:](#_2b04up9ndyze) 10

[Tests Système](#_3q75zkwx2hsa) 11

[Tests d’Acceptation](#_ri4fyzul61oz) 11

[Solution d’automatisation des tests](#_5rj1ac7k10hi) 11

[Prérequis](#_qfnekuicve2b) 12

[Framework d’automatisation des tests](#_o8nf6nxe3ql8) 13

[Frameworks de test](#_4gkkvbb4144e) 13

[API](#_tkdrvn8600q7) 13

[Atata](#_b5cvkg6k251f) 13

[Patterns de test](#_nb83ax4p9p4i) 14

[Automates de test](#_62njetiikr3x) 14

[Rapport de test](#_mk21fmpsbvnm) 15

[Environnements d’exécution et de test](#_fzgvchsz0w1x) 15

[**Feuille de route**](#_p9fw6sbbjh34) **16**

[Objectifs](#_j1b6fuhghlum) 16

[**Standards de l’industrie**](#_7qs08vakqs41) **16**

[Software Craftsmanship](#_w43ryfagvqjm) 16

[Test-Driven Development](#_xfnfods92mho) 20

[Acceptance Test-Driven Development](#_t418uafxkfgn) 20

[Behavior-Driven Development](#_hefr8bhz9gl2) 21

[Architecture Hexagonale](#_vzcjpgv1qznd) 21

[Domain-Driven Design](#_h38imor8tt18) 22

[Workflows de versionnage et de livraison](#_um6l3otjlxx9) 23

[Système de contrôle de version](#_vcbcarpwinc2) 23

[Flux de versionnage](#_xkbusgedbs9e) 24

[Industrialisation](#_lhbr6h5uas1u) 24

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Auteurs | Rôle | Date |
| 1.0 | Zied Hannachi | Consultant | 18/01//2019 |

# 

# Gestion de la Qualité

## Contexte et enjeux du client

### **Contexte**

Ordipat est un éditeur d’application web dans le domaine des cabinets de propriété intellectuelle.

Son parc applicatif comprend notamment :

* une application web nommée **Alissia**

L’application est développée en C#, par une équipe de 14 développeurs.

Ordipat aimerait mettre en place une approche agile des développements logiciels. L’intégration continue et une automatisation des tests pilotée par Behavior-Driven Development (BDD) font partie des pratiques auxquelles aspire l’éditeur. Toutefois, il est avant tout question, au moins dans un premier temps, de l’automatisation des tests de non-régression sur la partie *Alissia*.

Ordipat applique aujourd’hui une gestion prédictive de projets, qui s’approche du cycle en V. La recette usine est réalisée manuellement par l’équipe de QA en fin de processus.

### **Périmètre**

L’élaboration d’une stratégie d’automatisation des tests qui a motivé le présent audit se focalisera uniquement sur le projet réalisé en C# pour le contexte, et plus particulièrement sur l’application web *Alissia* pour l’automatisation des tests.

### **Enjeux**

#### À l'échelle de la société

Les objectifs d'une stratégie d'automatisation des tests dans le contexte de la société sont :

* de réduire le coût de correction des défauts ;
* de capitaliser les acquis du projet *Alissia* en vue du nouveau projet.

#### À l'échelle des projets ou produits

Les objectifs d'une stratégie d'automatisation des tests dans le contexte des projets sont :

* de privilégier la prévention des défauts plutôt que la détection des défauts ;
* d'adapter les formes de test aux objectifs de test.

S'agissant du projet Alissia, les objectifs sont :

* de détecter les défauts plus tôt ;
* de constituer un harnais contre les régressions.

#### À l'échelle des testeurs et développeurs

Pour les testeurs et développeurs, les objectifs sont :

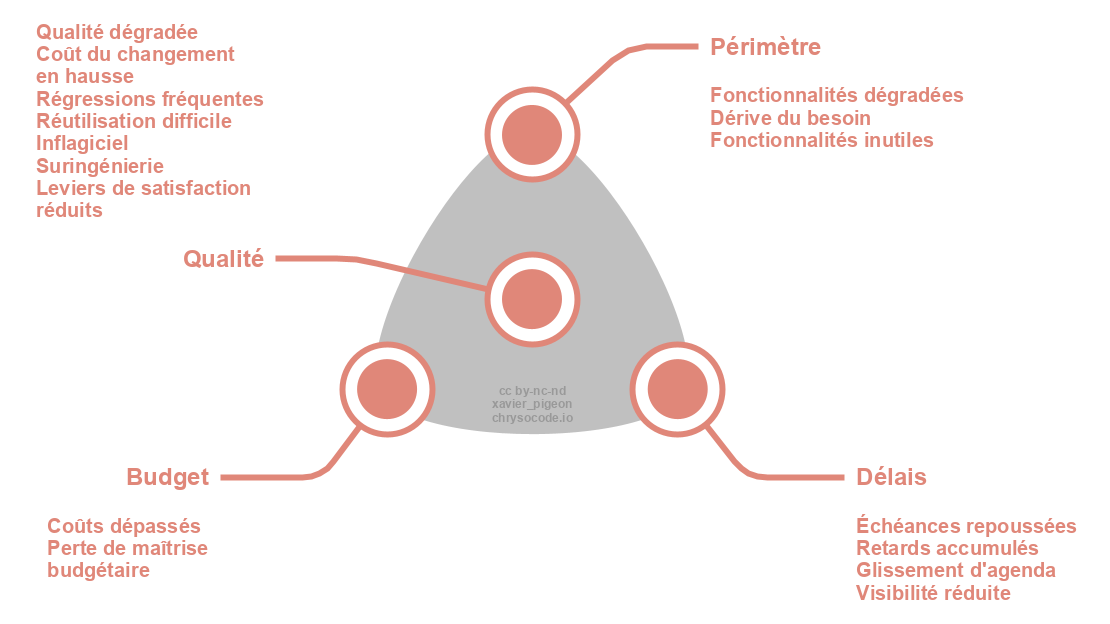
* de supprimer les tâches répétitives, laborieuses et pauvres en valeur ajoutée ;
* de concentrer leurs efforts sur les activités à plus forte valeur ajoutée ;
* de réussir une première automatisation des tests, afin d’en prouver la faisabilité.

## État de l’art

### Objectifs de projet

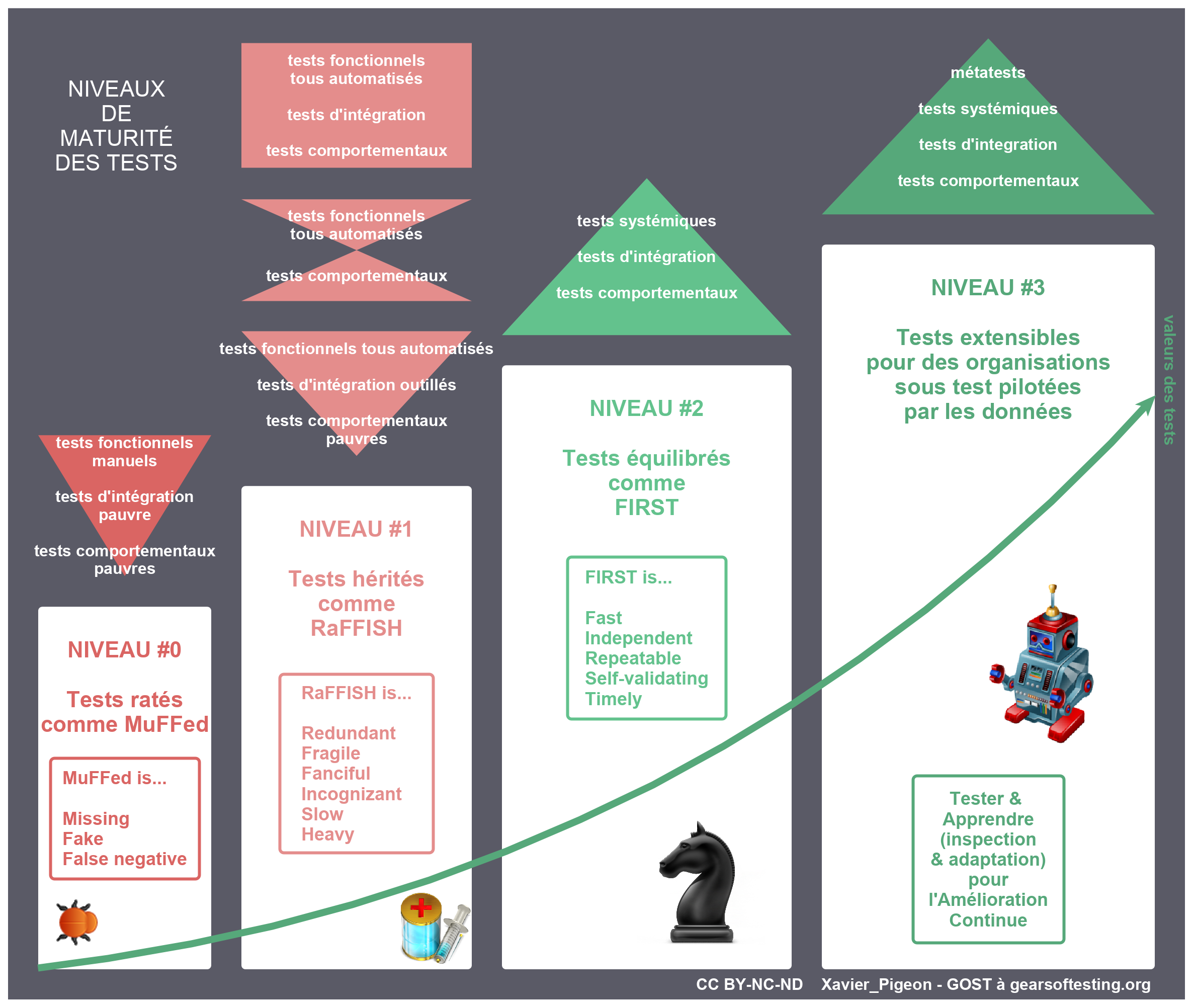
L’anticipation qui est de mise dans les processus prédictifs et séquentiels pousse à vouloir contrôler tout le déroulement du projet très en amont. Cette incitation au contrôle pousse à verrouiller tous les objectifs de projet : périmètre, délais et budget. Dans un tel contexte, la qualité du produit livré en fin de processus sert de variable d’ajustement. Or la qualité du produit détermine entièrement la satisfaction des utilisateurs finaux.

Depuis des années, le Standish Group publie régulièrement des statistiques dans son manifeste du chaos (*chaos report*) sur les succès et les échecs de projets logiciels du monde entier, et leurs raisons, en faisant le lien avec la taille des projets, l’approche de gestion, le management, l’ingénierie, le recrutement, et bien d’autres facteurs.



Triangle de fer dans le rouge

En matière d’ingénierie des tests, toutes les stratégies ne se valent pas. Plus les tests arrivent à un niveau de maturité élevé, plus ils contribuent au succès des projets. Le schéma ci-après illustre bien cette progression, selon 4 niveaux de maturité. Pour chacun d’eux, on dispose d’une heuristique ou ensemble de critères qualitatifs, et on peut visualiser la ou les stratégies de test qui s’y rapportent (pyramide, cornet de glace, sablier, court de tennis).

Niveaux de maturité des tests

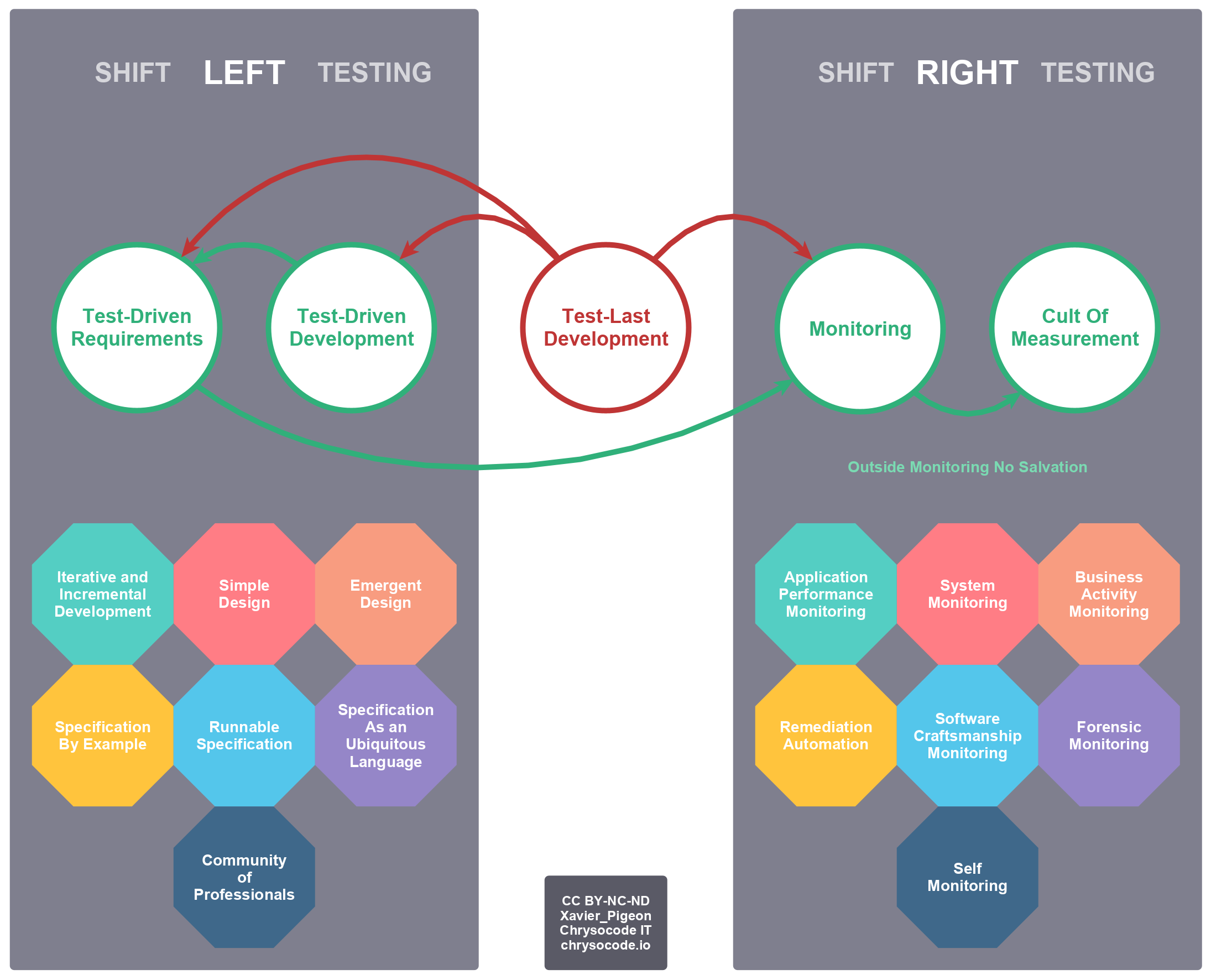
### Shift Left Testing & Shift Right Testing

Dans un processus prédictif et séquentiel, l’approche des tests consiste à contrôler la qualité d’un logiciel après sa phase de développement : c’est le *Test-Last Development*. Pour passer au *Shift Left Testing* (décalage à gauche des tests), les méthodes et pratiques doivent se tourner vers les développements pilotés par les tests (*Test-Driven Development* (TDD)) et les exigences pilotés par les tests (*Test-Driven Requirements* (TDR)), dont les briques sont :

* les développements itératifs et incrémentaux,
* la conception simple
* la conception émergente,
* la spécification par l’exemple,
* la spécification exécutable,
* la spécification comme langage omniprésent.
* les communautés de professionnels.

Pour passer au *Shift Right Testing* (décalage à droite des tests), les méthodes et pratiques doivent se tourner vers la supervision et l’obsession de la mesure, dont les briques sont :

* la supervision des performances applicatives,
* la supervision de l’infrastructure,
* la supervision des activités métier,
* l’automatisation des procédures de réparation,
* la supervision des méthodes de développement du *Software Craftsmanship*,
* la supervision de la sécurité,
* la supervision du système de supervision.

Alternatives au *Test-Last Development*

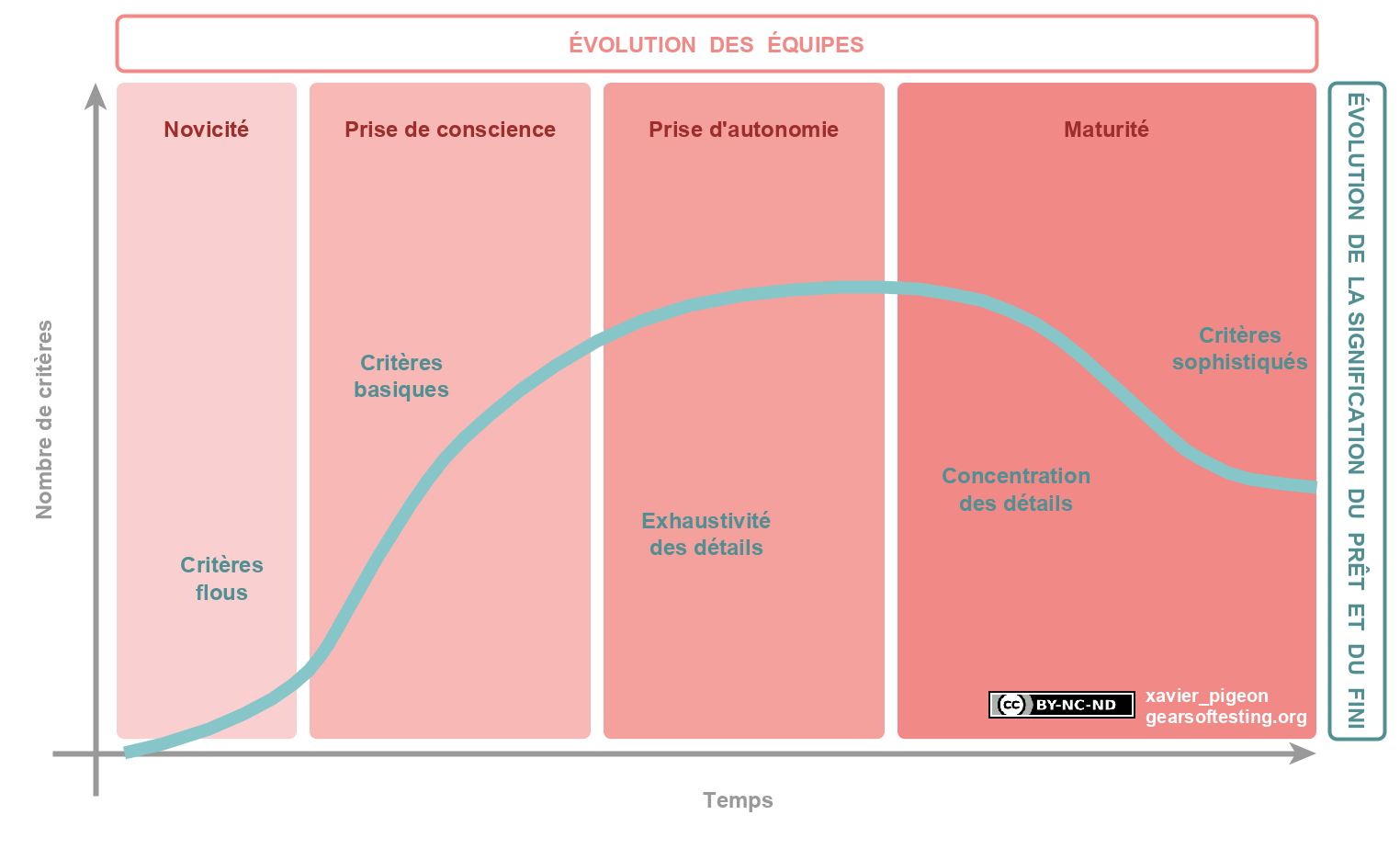
### Qualité des processus

Pour arriver à une meilleure maîtrise de leur processus de production et de gestion de la qualité de leurs livrables, les équipes agiles définissent et révisent régulièrement les critères d’approbation pour planifier ou livrer un changement.

* Signification du prêt (*Definition of Ready* ou DoR) : critères pour planifier un item.
* Signification du fini (*Definition of Done* ou DoD) : critères pour livrer un item.
* Signification du but (*Definition of Goal* ou DoG) : critères pour planifier une version (*milestone* ou *release*).
* Signification du succès (*Definition of Success* ou DoS) : critères pour livrer une version (*milestone* ou *release*).

Les artefacts méthodologiques Do\* sont utilisés comme condition de passage d’une étape à la suivante dans la chaîne de valeur.

En fonction de la qualité des critères d’approbation, on peut évaluer la maturité et suivre la progression des équipes. Plus les critères sont aboutis, plus une équipe est mature.

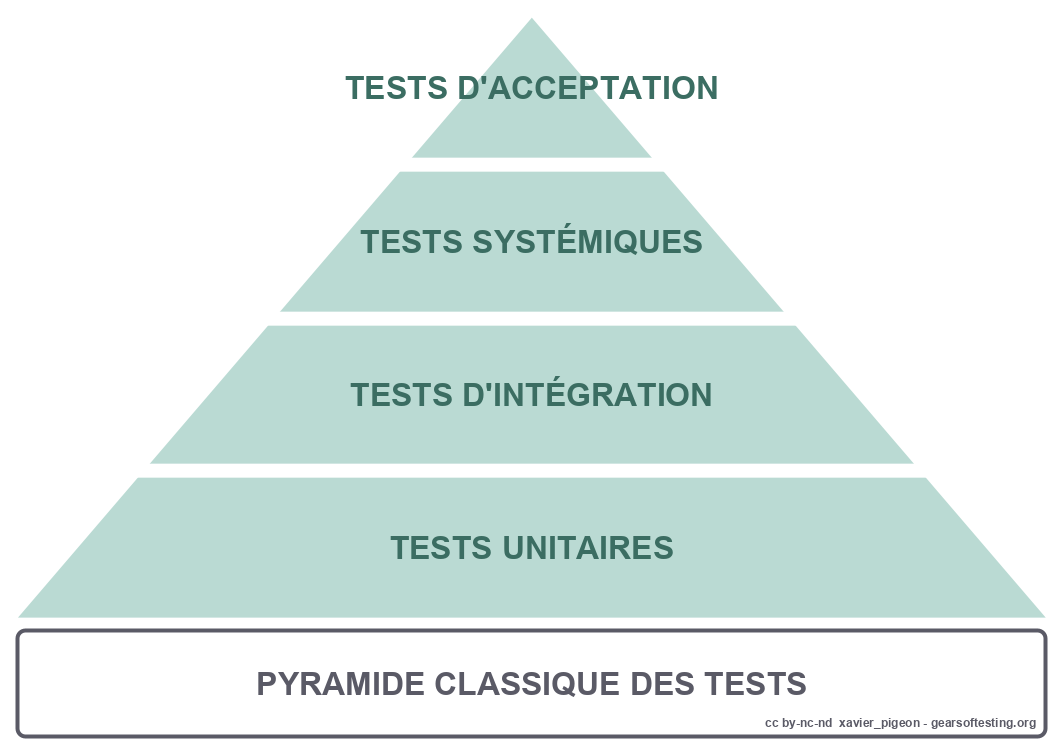
Modèle de maturité des équipes

# Automatisation des tests

## Niveaux de test

Les tests ont été audités en fonction des 4 niveaux de test. Chaque niveau de test est accompagné de ses recommandations. Structurée selon ces quatre niveaux, la pyramide des tests ci-après est présentée afin de rappeler que :

* Les tests des étages du bas sont les plus rapides à exécuter, et les moins coûteux à réaliser et maintenir. Ils doivent être les plus nombreux.
* Les tests des étages du haut sont les plus lents à exécuter, et les plus coûteux à réaliser et maintenir. Ils doivent être les moins nombreux.
* Chaque niveau de test (à chaque étage) est réalisé de manière à compléter les niveaux de test plus bas dans la pyramide, et non à les reproduire, afin d’obtenir une stratégie de test cohérente et efficace.

Pyramide classique des tests

Des actions d’amélioration doivent être menées ou poursuivies afin de rééquilibrer la stratégie de test actuelle, qui aurait plus une forme de pyramide Ces actions doivent être coordonnées en concertation avec les équipes de développement et de recette, ainsi que les clients.

### Tests Unitaires

L'objectif des tests unitaires est de séparer chaque partie du programme et de vérifier que les différentes parties fonctionnent correctement.

#### Avantages des tests unitaires

* **Rend le processus agile:** L'un des principaux avantages du test unitaire est qu'il rend le processus de codage plus agile. Lorsque vous ajoutez de plus en plus de fonctionnalités à un logiciel, vous devez parfois modifier l'ancien design et le code. Cependant, le changement de code déjà testé est à la fois risqué et coûteux. Si nous avons des tests unitaires en place, nous pouvons procéder à une refactorisation en toute confiance.Les tests unitaires vont vraiment de pair avec une programmation agile de toutes les saveurs car ils intègrent des tests qui vous permettent d'apporter des modifications plus facilement. En d'autres termes, les tests unitaires facilitent la refactorisation en toute sécurité.
* **Qualité du code:** Les tests unitaires améliorent la qualité du code. Il identifie tous les défauts qui ont pu apparaître avant que le code ne soit envoyé pour des tests d'intégration. L'écriture de tests avant le codage réel vous fait réfléchir davantage au problème. Il expose les cas marginaux et vous fait écrire un meilleur code.
* **Réduisez les coûts:** Étant donné que les bogues sont détectés tôt, les tests unitaires aident à réduire le coût des corrections de bogues.

### Tests d’Intégration

Les tests d’intégration peuvent désigner :

* les tests d’intégration de composant,
* les tests d’intégration système.

#### Avantages des tests d'intégration:

* Les tests d'intégration fournissent une technique systématique pour assembler un système logiciel tout en effectuant des tests pour découvrir les erreurs associées à l'interfaçage.
* L'application est testée afin de vérifier qu'elle répond aux normes fixées par le client ainsi que de rassurer l'équipe de développement que les hypothèses émises lors des tests unitaires sont correctes.
* Les tests d'intégration n'ont pas besoin d'attendre que tous les modules d'un système soient codés et testés à l'unité. Au lieu de cela, il peut commencer dès que les modules appropriés sont disponibles.
* Des tests d'intégration ou des tests incrémentiels sont nécessaires pour vérifier si les modules logiciels fonctionnent en unité.

### Tests Système

Les tests système peuvent désigner :

* les tests fonctionnels,
* les tests de performance,
* les tests de sécurité,
* les tests de compatibilité,
* les tests de portabilité.

### Tests d’Acceptation

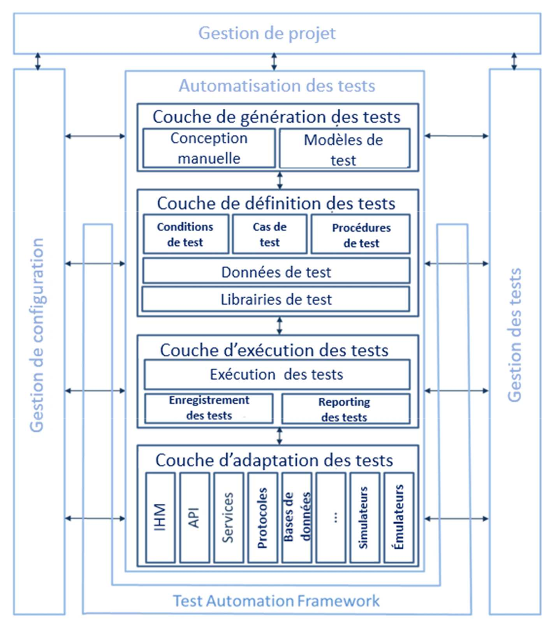
Les tests d’acceptation peuvent désigner :

* les tests d’acceptation utilisateur,
* les tests d’acceptation opérationnelle,
* les tests d’acceptation réglementaire,
* les tests d’acceptation contractuelle.

## Stratégie d’automatisation des test et choix des outils

## Solution d’automatisation des tests

Nous allons construire une solution d’automatisation des tests adaptée au contexte et architecturée selon le schéma générique d’architecture ci-après.



Architecture d’automatisation de test générique[[1]](#footnote-0)

### Prérequis

La mise en place d’une stratégie de test et de l’automatisation des tests est tributaire de prérequis, qui doivent être satisfaits afin d’atteindre ses objectifs.

* Les spécifications fonctionnelles sont un prérequis aux tests en boîte noire.
* La documentation de l’architecture technique et la supervision applicative et de l’infrastructure sont des prérequis aux tests de performance (boîte grise).
* Les cas de test devraient être conçus en même temps que les spécifications fonctionnelle et technique.
* Des identifiants de test doivent être affectés à chaque composant graphique, de manière à ce que les tests n’échouent pas si la structure d’un écran est remaniée.
* Les données de production doivent pouvoir être extraites, afin de reproduire une volumétrie ciblée. Le processus d’extraction doit rendre les données sensibles inexploitables.
* Les jeux de données de test doivent pouvoir être injectés aisément dans les bases de données au début d’un cas de test.
* Il faut disposer d'environnements de référence pour les builds et les tests, afin d’éviter le syndrome "Ça marche chez moi" (faux-négatifs) ou bien "Tester l'environnement plutôt que l'application" (faux-positifs).
* La gestion des environnements devrait être centralisée par une équipe transverse, seule à disposer des prérogatives pour modifier les environnements, agissant dans l'intérêt commun de toutes les équipes. Dans une approche DevOps, les membres d'une telle équipe sont répartis dans les équipes de développement, afin d'éviter les silos.

### Framework d’automatisation des tests proposé

#### Frameworks de test

* [Selenium WebDriver](https://selenium.dev/) est un outil d’automatisation des tests fonctionnels pour les applications web. Il fonctionne avec tous les navigateurs web majeurs.
* SpecFlow est un framework de test qui prend en charge le développement basé sur le comportement (BDD) et adapté aux env .NET. Il nous permet de définir le comportement de l'application dans un texte clair et significatif en utilisant une grammaire simple définie par une langue appelée Gherkin.
* MSTest est un framework de test unitaire fourni par Microsoft Visual Studio qui possède des fonctionnalités intégrées pour prendre en charge les tests pilotés par les données, qui peuvent être configurées très facilement.

### API

[Atata](https://atata.io/) est un framework d'automatisation de test web en C#/.Net pour les applications Web.

* WebDriver: Basé sur Selenium WebDriver et préserve toutes ses fonctionnalités.
* Intégration: Fonctionne sur n'importe quel moteur de test .NET (par exemple NUnit, xUnit, SpecFlow) ainsi que sur les systèmes CI comme Jenkins, Azure DevOps.
* Configurable: Définit les stratégies de recherche de composants par défaut ainsi que des paramètres supplémentaires. Atata.Configuration.Json fournit des configurations JSON flexibles.
* Rapports: Journalisation personnalisable intégrée et fonctionnalité de capture d'écran.
* Page Object Model: Fournit un modèle d'objet de page fluide unique, facile à implémenter et à gérer.
* Vérification: Un ensemble de méthodes et de déclencheurs d'assertion fluides pour la vérification des composants et des données.

[Assert](https://joel-costigliola.github.io/assertj/)s NUnit est une bibliothèque open-source permettant d’écrire aisément des assertions concises et riches dans les tests C#, tout en améliorant la lisibilité du code de test.

#### Patterns de test

* *Page Object* est un modèle de conception permettant de créer un référentiel d'objets pour les éléments d'interface utilisateur Web. Sous ce modèle, pour chaque page web de l'application, il devrait y avoir une classe de page correspondante. Cette classe Page trouvera les éléments web de cette page et contient également des méthodes de page qui effectuent des opérations sur ces éléments web.
* *Browser Factory* est un code logique qui permet de transmettre le nom du navigateur, afin que le code du programme exécute le test Selenium sur le navigateur souhaité.
* *Given-When-Then* peut être utilisé indépendamment de Gherkin pour structurer les cas de test, de la même façon que *Setup-Execute-Verify* ou *Arrange-Act-Assert*. Ces triplets désignent dans l'ordre les étapes de préparation, d'exécution et de vérification d'un cas de test.
* Organiser les cas de test automatisés en suites de test, en tests imbriqués, et selon une arborescence fonctionnelle pérenne de répertoires et de fichiers.

#### Orchestration de test

* [Jenkins](https://jenkins.io/) est une plateforme de CI/CD, très extensible grâce à ses plugins.
* [Jenkins X](https://jenkins-x.io/) est une plateforme complète de CI/CD pour le cloud, qui utilise des technologies open source, comme Helm, Skaffold, Monocular, Nexus, Docker, KSync, ChartMuseum.
* TFS inclut Build and Release et peut être installé et géré sur vos propres serveurs.Il permet d'exécuter vos tests automatisés.

#### Rapport de test

* [ExtentReports](http://extentreports.com/) est un framework de génération de rapports détaillés et interactifs, utilisé avec Selenium.

### Justification du choix des outils

Pour notre solution d’automatisation de test nous avons choisi Selenium WebDriver comme outils d’automatisation de test car il est le plus populaire et le plus flexible et qui répond à tous besoins.

Selenium WebDriver est considéré comme meilleur outil d’automatisation de test pour l’année 2018 et il possède différents avantages:

* Gratuit et open source.
* Dispose d’une API accessible par plusieurs langages (Java, C#, Perl, Python, JavaScript, Ruby, PHP).
* Supporte multiples navigateurs (Chrome, Firefox, Opera, ...).
* Possibilité de l'intégrer avec d’autre outils et frameworks de test.

Il existe d’autres outils d’automation de test, dans cette partie nous allons citer quelques outils et préciser les limites de chaqu’un.

1. **Protractor:** Framework open source d’automatisation des tests de bout en bout pour les applications Angular et AngularJs.

Limites:

* Prend en charge que JavaScript.
* Ne supporte que la navigateur Chrome.
* Ne prend pas en charge l’automatisation des applications mobiles.

1. **Robot Framework:** Framework d’automatisation des tests pour les tests d’acceptation et le développement piloté par les tests d’acceptation.

Un framework d’automatisation hybride, qui supporte le Data-Driven Testing, le Keyword-Driven Testing et Behavior-Driven Development.

Limites:

* DSL Robot limité.
* Risque de faire du code non maintenable.
* IDE propriétaire est peu évalué.

1. **Katalon studio:** Plateforme d’automatisation de test gratuit des application web, mobile et web service

Limite:

* Prend en charge uniquement Java et Groovy pour la réduction des scripts de test.
* Ne supporte pas les tests distribués

1. [**Agilitest**](https://www.agilitest.com/)**:** est une solution de tests automatisés conçue pour les testeurs fonctionnel. Elle ne nécessite pas de compétences en développement.

Limite:

* Difficulté à comprendre certains mots clés donc la documentation explicative est très recommandée et nécessaire.
* L’outil est payant.

### Environnements d’exécution et de test

Les équipes doivent disposer d’environnements de test dédiés, qui répondent aux exigences pour exécuter les tests. Une bonne gestion de ces environnements passe par la définition et la séparation des responsabilités.

* Réalisateur : la ou les personnes qui maintiennent un environnement et livrent de nouvelles versions du logiciel.
* Autorité d’approbation : la personne qui décide des changements qui doivent être apportés à un environnement, qu’il s’agisse de l’approvisionnement ou des livraisons logicielles.
* Consulté pour avis : la ou les personnes qui utilisent un environnement, notamment pour leurs tests, leur travail en dépend.
* Informer : toute personne qui a intérêt à être informé d’un changement apporté à un environnement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Environnement** | **Réalisateur** | **Autorité d’approbation** | **Consulté pour avis** | **Informé** |
| **Intégration** | Opérationnels  Développeurs | Développeurs | Développeurs | Développeurs |
| **Recette** | Opérationnels | Responsable QA | Testeurs | Parties prenantes |
| **Préproduction** | Opérationnels | Responsable QA | Développeurs  Testeurs | Parties prenantes |
| **Production** | Opérationnels | Responsable Stratégie Produit | Développeurs  Testeurs  Opérationnels | Tous les personnels |

Préconisations

* Les développeurs sont susceptibles de modifier l’environnement d’intégration par le biais des scripts versionnés de configuration et de déploiement fournis par les opérationnels. Ces changements peuvent ainsi être inclus aux livraisons et promus par les opérationnels à l’identique.
* Étant donné que l’équipe d’exploitation est débordée, il faut renforcer cette équipe.
* Établir un plan de communication afin de déterminer les parties prenantes.

# Feuille de route

## Objectifs

L’objectif est la réalisation d’un POC[[2]](#footnote-1) complet qui satisfait les critères suivants :

* Des tests fonctionnels sont automatisés, et exécutés dans la plateforme d’intégration continue (PIC).
* Les cas de test fonctionnels vérifient la cinématique soit d’un écran avec un focus fort sur une fonction du logiciel, soit de plusieurs écrans avec un focus clair sur un parcours de l’utilisateur.
* Les tests fonctionnels disposent chacun de leur jeu de données.
* Les tests fonctionnels sont exécutés sur différents environnements, en particulier celui d’intégration et celui de recette. Leurs résultats sont identiques quel que soit l'environnement, pour une même version du système logiciel.
* Les tests de non-régression les plus répétitifs sont automatisés.
* Construire une plateforme de tests automatisés à jour par rapport aux versions des technologies utilisées et maintenable grâce à l’utilisation des patterns de test et de bonnes pratiques de test.

# Standards de l’industrie : Mieux Développer

Une autre approche pour améliorer la qualité globale du logiciel et d’intervenir sur la qualité du code et aider les développeurs à faire du code propre, maintenable, documenté et testé au niveau unitaire.

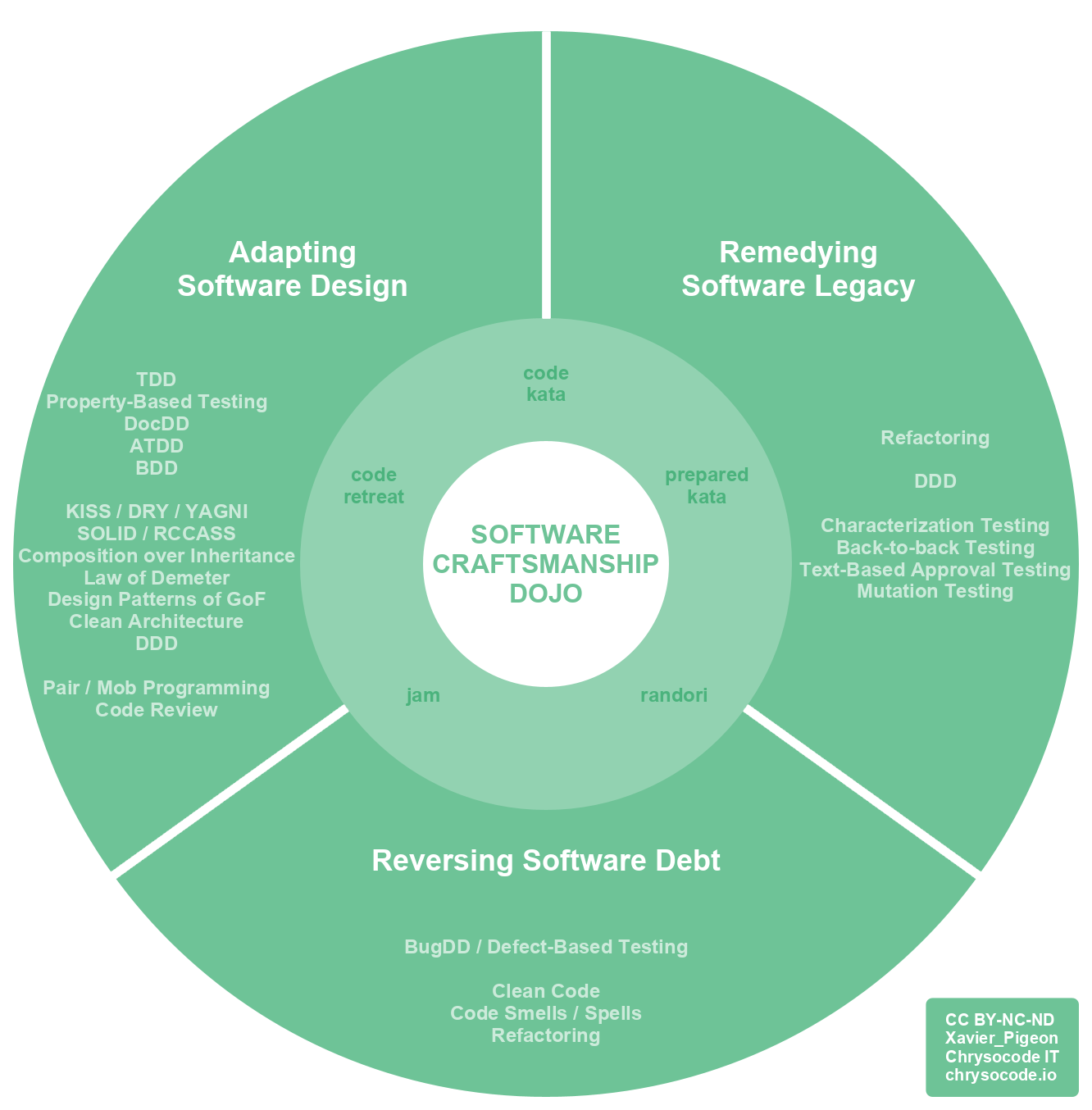
## Software Craftsmanship

Le [Software Craftsmanship](http://manifesto.softwarecraftsmanship.org/), c’est-à-dire l’artisanat du code et le compagnonnage logiciel, est une mouvance qui fédère des praticiens exigeants de l’ingénierie logicielle. Cette mouvance est née en 2009 d’une mise à jour du [manifeste des développements agiles](https://agilemanifesto.org/) de 2001, dont elle est une extension, afin de recentrer l’industrie logicielle sur la création de produit de valeur et de qualité.

Dans le Software Craftsmanship, l’entraînement et le perfectionnement en continu sont organisés en dojo de programmation (ateliers pratiques), dont les rencontres sont animées selon des formats (kata préparé, randori, jam, retraite de code, et leurs variantes) adaptés aux objectifs d’apprentissage.

* Être capable de concevoir des logiciels faciles à adapter, et d’adapter la conception d’un logiciel à l’évolution des besoins des utilisateurs sans en détériorer la qualité.
* Être capable de faire évoluer un logiciel hermétique au changement, en remédiant à un code hérité (sans test, et généralement difficilement testable à cause des couplages structurels).
* Être capable de contrecarrer la dette technique (choix de développement ou d'architecture qui augmentent la [complexité accidentelle](https://fr.wikipedia.org/wiki/Complexit%C3%A9_essentielle) et le coût du changement d'un logiciel), pour remettre à flot la qualité d’un logiciel au niveau souhaité.

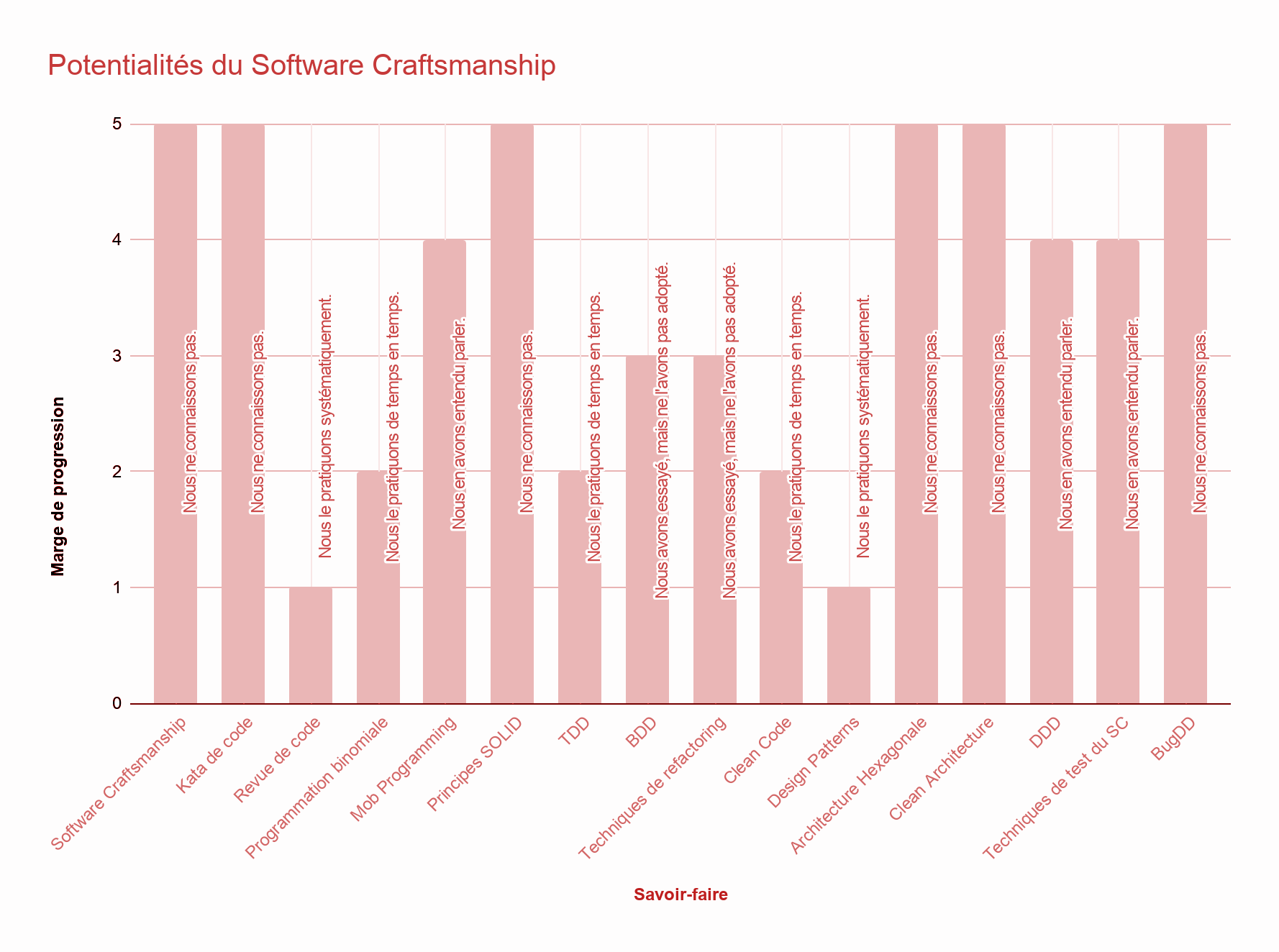
Ces capacités passent par la maîtrise de nombreuses méthodes et pratiques d’ingénierie, représentées ci-dessous. Elles ont toutes en commun de fournir les armes méthodologiques pour combattre la [complexité accidentelle](https://fr.wikipedia.org/wiki/Complexit%C3%A9_essentielle) des logiciels.



[Roue du dojo](https://chrysocode.io/conseil.html) du Software Craftsmanship : ateliers et apprentissages

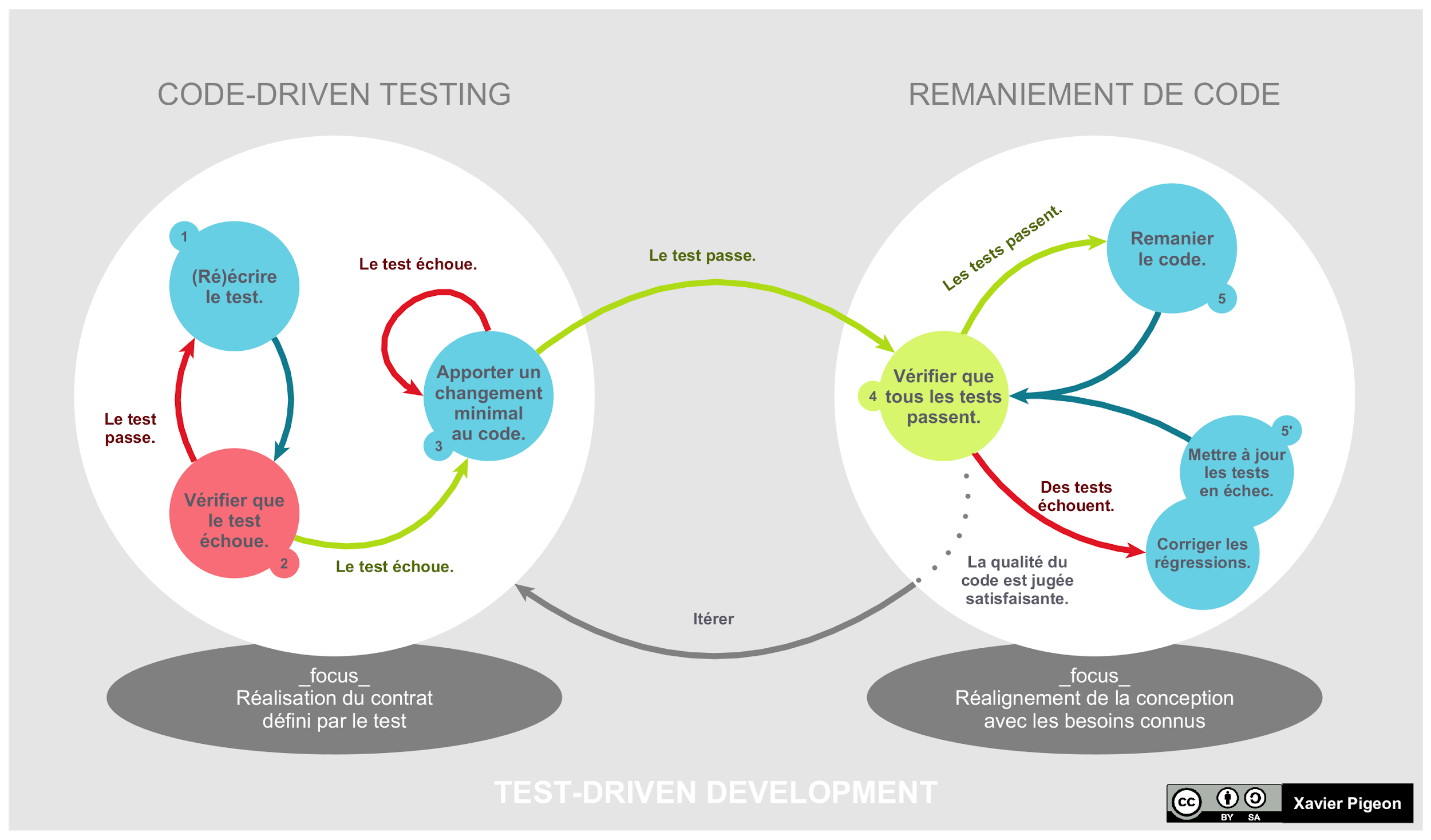
Nous pouvons vous accompagner dans l'acculturation au Software Craftsmanship et dans la montée en compétences des équipes pour en maîtriser les savoir-faire. L'objectif d'un tel dispositif d'accompagnement est d'élever le niveau de la qualité intrinsèque par l'excellence ingénierique.

Le graphique ci-après représente les potentialités du Software Craftsmanship, c’est-à-dire les progressions possibles, en fonction de degré d’application des méthodes et pratiques d’ingénierie logicielle. Moins une méthode est appliquée, plus la marge de progression est importante.



### Test-Driven Development

[TDD](https://fr.wikipedia.org/wiki/Test_driven_development) est une méthode de conception logicielle pour des conceptions simples et émergentes, et qui procède par changements minimaux et remaniements successifs.



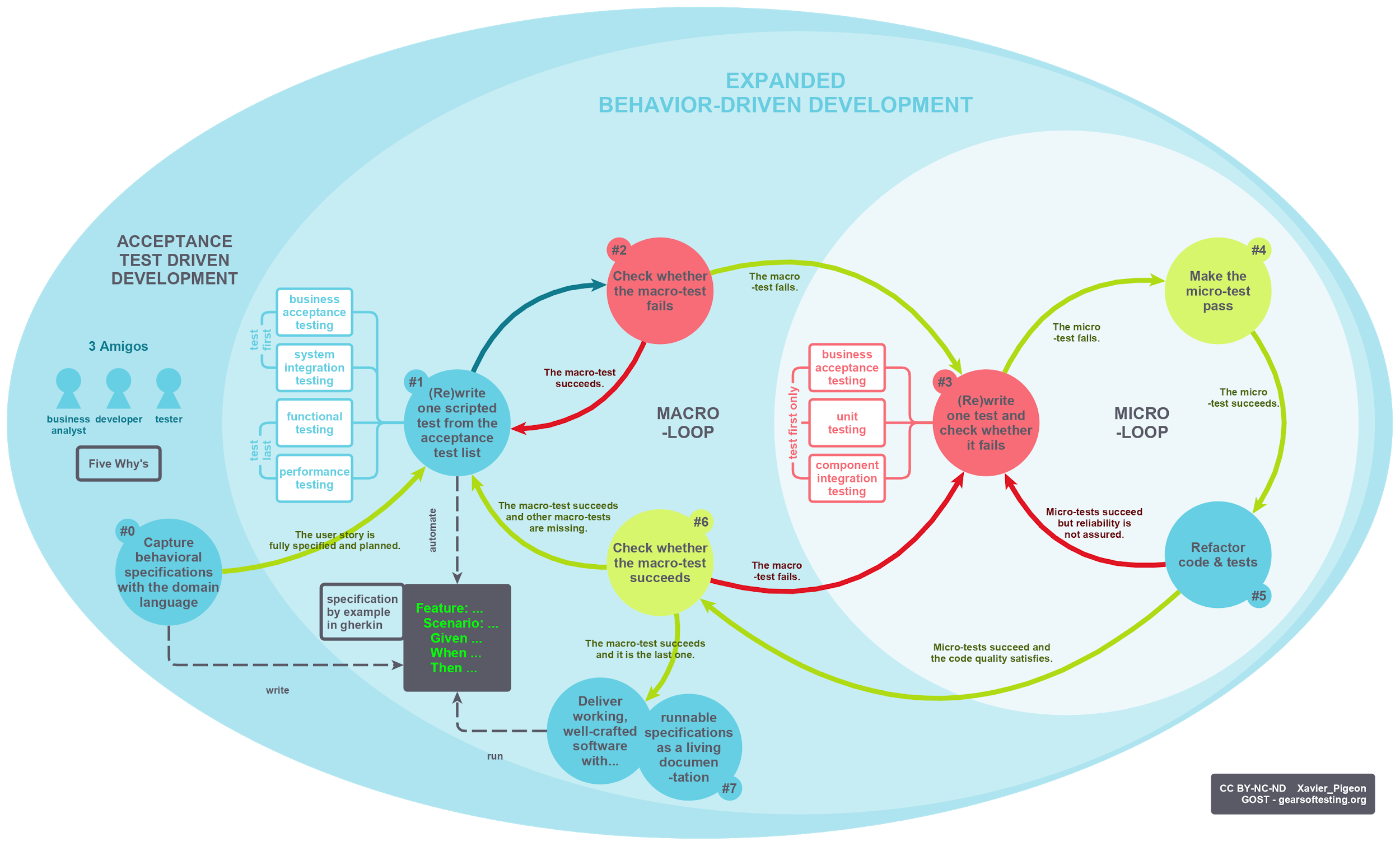
[Cycle global, itératif et incrémental de TDD](https://commons.wikimedia.org/wiki/File:Cycle-global-tdd.png)

### Acceptance Test-Driven Development

ATDD est une méthode d'heuristique des besoins, qui explore et spécifie les besoins au moyen d'exemples concrets d'utilisation, en faisant discuter plusieurs intervenants complémentaires qui partagent leur compréhension des problématiques des utilisateurs. Ces exemples concrets peuvent ensuite être formalisés sous forme de scenarii, dont l'automatisation en tant que tests d'acceptation est une option intéressante. Les scenarii d'ATDD, voire les tests d'acceptation ainsi automatisés, ont alors valeur de spécification.

### Behavior-Driven Development

[BDD](https://en.wikipedia.org/wiki/Behavior-driven_development) est une méthode de conception logicielle qui étend [TDD](https://en.wikipedia.org/wiki/Test-driven_development), par l’apport de pratiques additionnelles, et qui explore et spécifie les besoins, en mettant l’accent sur la collaboration et en s’appuyant sur les tests et leur scénarisation.

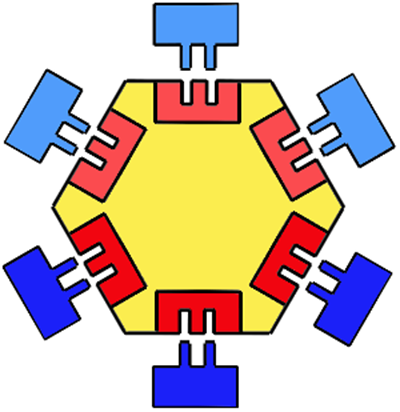


[Cycle global de BDD](https://gearsoftesting.org/tempo-of-testing.html)

L'automation des tests fait partie de BDD, au même titre qu'avec TDD, tout en élargissant l'utilisation des tests à tous les niveaux de test.

### Architecture Hexagonale

L’Architecture Hexagonale, ou architecture ports/adaptateurs, est un style d’architecture dont les règles favorisent l’indépendance et la testabilité du modèle métier et de sa logique, en le préservant de toute corruption par des préoccupations techniques.

  
Représentation des ports et adaptateurs  
au frontière du domaine métier  
qui est au centre de l’architecture

### Domain-Driven Design

DDD est un enrichissement du paradigme de la conception orientée objet, dont l’objectif est d’endiguer et briser la complexité accidentelle (complexité informatique qui ne répond pas à un besoin), en fournissant un référentiel de pratiques destinées à faire face aux problématiques récurrentes des projets d’entreprise.



Cartographie des concepts et patterns de DDD

## Workflows de versionnage et de livraison

### Système de contrôle de version

Les possibilités des outils de contrôle de version distribués ont rendu obsolètes leurs prédécesseurs centralisés, comme Subversion. Leurs fonctionnalités avancées en font plus que des outils de sauvegarde ou d’historisation, ce sont des outils qui apportent un support et une contribution importants dans les activités de développement. Ces possibilités s’appliquent tout autant au domaine des tests automatisés, où les testeurs deviennent des développeurs de solution de test.

Le passage à un système de contrôle de version distribué tel que Git requiert de changer d'approche de versionnage et s’accompagne au début d’une certaine complexité conceptuelle qu’il faut acquérir, en échange d'une grande souplesse d'utilisation, et anticiper par des actions de formation, de manière à ce que les équipes deviennent rapidement autonomes et efficaces. Nous pouvons vous accompagner en ce sens de manière générale. En particulier, nous proposons une formation à Git centrée sur les cas d'utilisation rencontrés par les développeurs, au contraire des formations qui passent en revue les commandes de Git une à une. Nous dispensons également une formation de perfectionnement dédiée aux bonnes pratiques de versionnage avec Git, qui concentre de nombreux retours d'expérience du terrain.

### Flux de versionnage

Avoir un modèle des flux de versionnage efficace permet d’améliorer la collaboration et de sécuriser les livraisons. De nombreux facteurs doivent être pris en compte afin de déterminer un modèle adapté au contexte d’une organisation.

Voici quelques modèles de gestion de branche bien connus.

* Modèle centralisé
* *Feature branches*
* *Trunk-based development* (TBD)
* *Gitflow*
* *Oneflow*

L'émergence d'un modèle des flux de versionnage propre à votre contexte tout en tenant compte de l'état de l'art et des bonnes pratiques est réalisable au cours d'ateliers collaboratifs et participatifs.

### Industrialisation

Les *pipelines* d'intégration continue, de livraison continue et de déploiement continu font suite à la modélisation des flux de versionnage pour industrialiser le déploiement des applications. Mis en oeuvre dans l’usine logicielle, ils intègrent l’exécution et la coordination des tests automatisés, en fonction de leur niveau de test et de la rapidité d’exécution correspondante.

1. ISTQB, Syllabus Niveau Avancé, <http://www.cftl.fr/wp-content/uploads/2019/06/Avance_TA_E_Syllabus-FR-2016_Final_v_1_0.pdf> [↑](#footnote-ref-0)
2. All4Test, Optimisez l’automatisation des tests, <https://www.all4test.fr/nos-offres/optimisez/automatisation-des-tests/> [↑](#footnote-ref-1)