THE Open GROUP
Making standards work®

Document de stratégie de test

1 Sommaire exécutif

Ce document regroupe une stratégie de test.

2 Historique des Versions

Date	Version	Commentaires
16/05/2022	**0.1**	

3 Contenu

Document de stratégie de test

4 Objet

Le présent document est une stratégie de test auquel le PoC se conforme. Le PoC a été réalisé par un développement piloté par les tests, c'est-à-dire que les tests unitaires ont été conçus avant le développement de l'application.

5 Préparer et définir la stratégie de tests

La stratégie de tests permet de réussir la phase de tests, ce document indique les prérequis nécessaires à la recette et guide les testeurs durant la phase de test. C'est un peu comme un « cahier des charges » avant de lancer un projet. Ci-dessous les points à indiquer dans la stratégie de tests.

- Périmètre
- Moyens de tests
- Environnement et URL
- Méthodologies et Procédures
- Type de tests prévues
- Jeux de données
- Criticité des incidents de fonctionnement
- Partie prenantes
- Planning

6 Planification des tests automatisés d'architecture

Pour chaque niveau de test, la planification des tests automatisés d'architecture commence au début du processus de test pour ce niveau et continue tout au long du projet, jusqu'à l'achèvement des activités de clôture pour ce niveau.

La planification des tests automatisés inclut aussi l'identification des méthodes pour collecter et suivre les métriques utilisées pour guider le projet, déterminer le respect du plan de test et mesurer l'atteinte des objectifs.

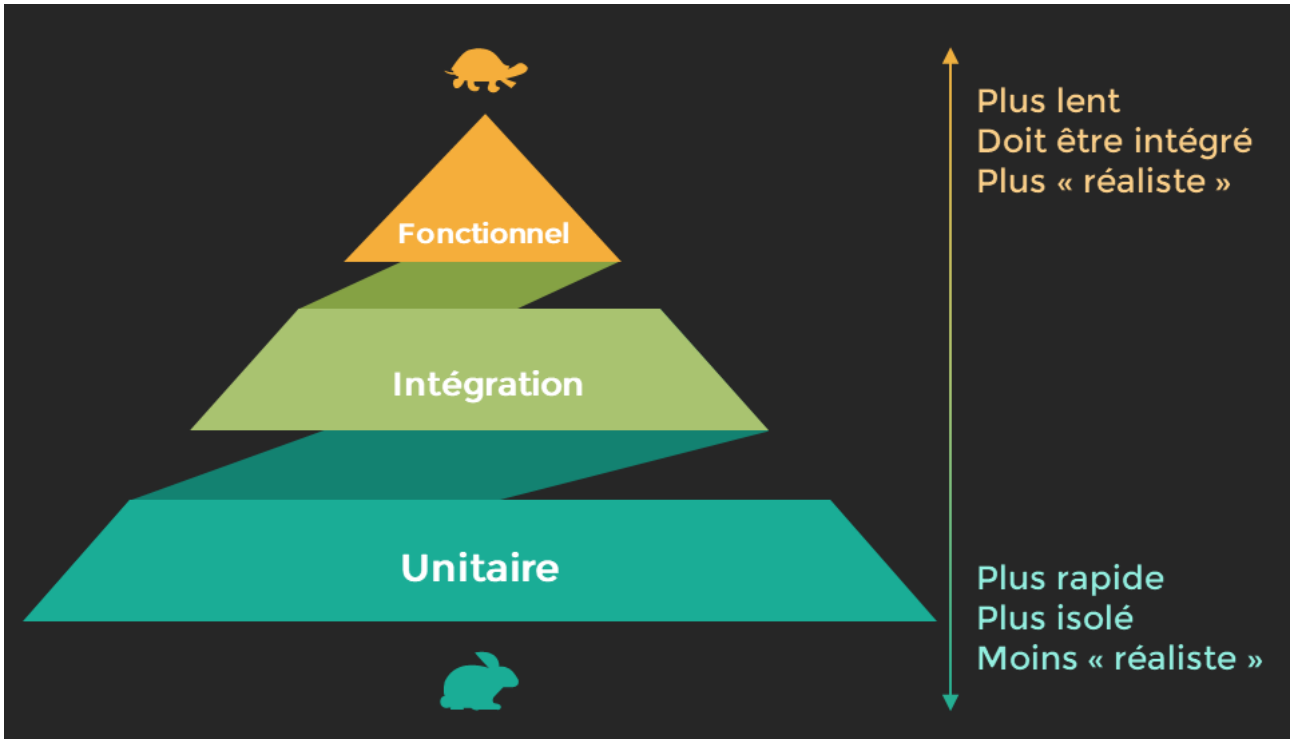
Le plan de test automatisé peut aussi lister les caractéristiques spécifiques du logiciel faisant partie de son périmètre (sur la base de l'analyse des risques, si nécessaire), de même qu'identifier précisément les caractéristiques qui sont hors de son périmètre. Selon les niveaux de formalisme et la documentation adaptés au projet, chaque caractéristique à couvrir peut être associée à une spécification de conception de test correspondante.

On peut toutefois distinguer deux types de plans :

▪ Plan de test maître (ou plan de test projet) – décrit l'implémentation de la stratégie de test sur un projet particulier.

▪ Plan de test de niveau (ou plan de test de phase) – décrit les activités précises à mettre en œuvre pour chaque niveau de test.

Les 3 types de tests automatisés les plus courants sont les tests unitaires, d'intégration et fonctionnels. Ces types de tests sont souvent présentés selon la pyramide des tests ci-dessous.



6.1 Tests unitaires

La plus grosse section, à la base de la pyramide, est constituée des **tests unitaires** qui testent de "**petites" unités** de code. Plus précisément, ils testent que chaque **fonctionnalité** extraite de manière isolée se comporte comme attendu.

Ils sont très rapides et faciles à exécuter. Si nous avons cassé quelque chose, nous pouvons le découvrir **vite et tôt**. De bons tests unitaires sont **stables**, c'est-à-dire que le code de ces tests n'a pas besoin d'être modifié, même si le code de l'application change pour des raisons purement techniques. Ils deviennent donc **rentables**, car ils ont été écrits une seule fois, mais exécutés de nombreuses fois.
Cependant, seules des unités individuelles de code sont testées, nous avons donc besoin d'autres tests permettant de s'assurer que ces unités de code fonctionnent entre elles.

6.2 Tests d'intégration

Au milieu de la pyramide se trouvent les tests d'intégration. Ils vérifient si les unités de code **fonctionnent ensemble** comme prévu – en présupposant que les tests unitaires soient passés ! Comme les tests d'intégration vérifient les interactions entre les unités, nous avons plus de certitude concernant le bon fonctionnement de l'application finale.

Ils peuvent nécessiter **l'exécution de composants extérieurs** (base de données, services web externe, etc.). Le lancement de ces composants et l'interaction entre les unités de code développées rendent ces types de test plus lents et potentiellement moins stables. Mais nous simulons des scénarios plus proches de l'utilisation finale de l'application.

6.3 Tests fonctionnels

Enfin, au sommet de la pyramide, les tests fonctionnels (appelés end-to-end en anglais), visent à simuler le comportement d'un utilisateur final sur l'application, depuis l'interface utilisateur. L'ensemble du code développé est pris comme une **boîte noire** à tester, sans connaissance des briques et des unités de code qui la composent. Les simulations obtenues sont donc les plus proches de l'application utilisée dans des conditions réelles.

Ces tests nécessitent toute l'infrastructure nécessaire à l'application. Ces types de tests sont les plus lents à exécuter, et **testent une partie beaucoup plus grande du code développé**.

6.4 Tests IHM

Ces tests sont sont plus particulièrement utilisé pour tester la non-régression d'une interface homme-machine. Ces derniers sont automatisés à l'aide d'outils existants sur le marché comme Selenium par exemple. Ces tests permettent de valider le fonctionnement de l'IHM lors des modifications importante des fonctionnalités impactant les interfaces homme-machine.

7 Rédiger les scénarios et les cas de tests

Rédaction des scénarios de test , en utilisant les terminologies Given That / When / Then.

8 Exécution des tests

Tests Unitaires : tests techniques des **développeurs** après leurs développements (prérequis aux tests fonctionnels).

Tests d'Intégration : tests techniques des couches services et repository (anciennement DAO).

Tests E2E (End-To-End) : tests de bout en bout de l'ensemble des fonctionnalités projet de l'application.

Tests automatisés : tests automatisés en utilisant gitLab avec le CI/CD.

9 Résultats

Les tests fonctionnels, de Bout en Bout et TNR permettent de vérifier que le logiciel est conforme aux spécifications décrites dans les Users stories. Ils peuvent même permettre de combler des trous fonctionnels. Ils assurent la qualité d'un logiciel.