

LA MÉTHODOLOGIE AUTOUR DES TESTS

Partie I : Les Tests

I. Les Tests

Qu'est-ce qu'un test ?

Quels types de tests ?


Quelles techniques pour tester ?

Définition d'un test

QU'EST-CE QU'UN TEST ?

Un test est un ensemble de 3 éléments :

Un système à tester

 système peut inclure des composants externes : OS, disques durs, VM, ...

Des entrées statiques ou générées à l'exécution

Ce sont les jeux de données qui vont être consommés par le test ainsi que son état initial

Un scénario de test

Il s'agit du scénario de test, décrit étapes par étapes.

QUESTIONS				
1-	A	B	C	D
2-	A	B	C	D
3-	A	B	C	D
4-	A	B	C	D
5-	A	B	C	D
6-	A	B	C	D

QU'EST-CE QU'UN TEST ?

Le test est exécuté sur le système qu'il définit, en utilisant le jeu de données décrit en entrée.

Il permet de vérifier :

- Les sorties générées par le système
- Le comportement interne du système
- Les deux à la fois

QU'EST-CE QU'UN TEST ?



Attention : un test est toujours lié à une et une seule vérification !



Ne jamais mélanger différent types de vérifications dans le même test

Par exemple, on sépare toujours les tests de performance des tests de sécurité



Un problème devrait faire échouer un seul et unique test

Cette règle est une situation idéale - en pratique, on limitera au maximum le nombre de tests pouvant échouer en même temps.

Les 7 principes de test

LES 7 PRINCIPES DE TEST

I. Tester montre l'absence de problème

Le but d'un test est de casser le logiciel pour y détecter d'éventuels problèmes.



L'absence de preuve n'est pas la preuve de l'absence !
Un test qui passe montre que le problème n'a pas été détecté... cela ne signifie pas qu'il n'y en a pas !



LES 7 PRINCIPES DE TEST

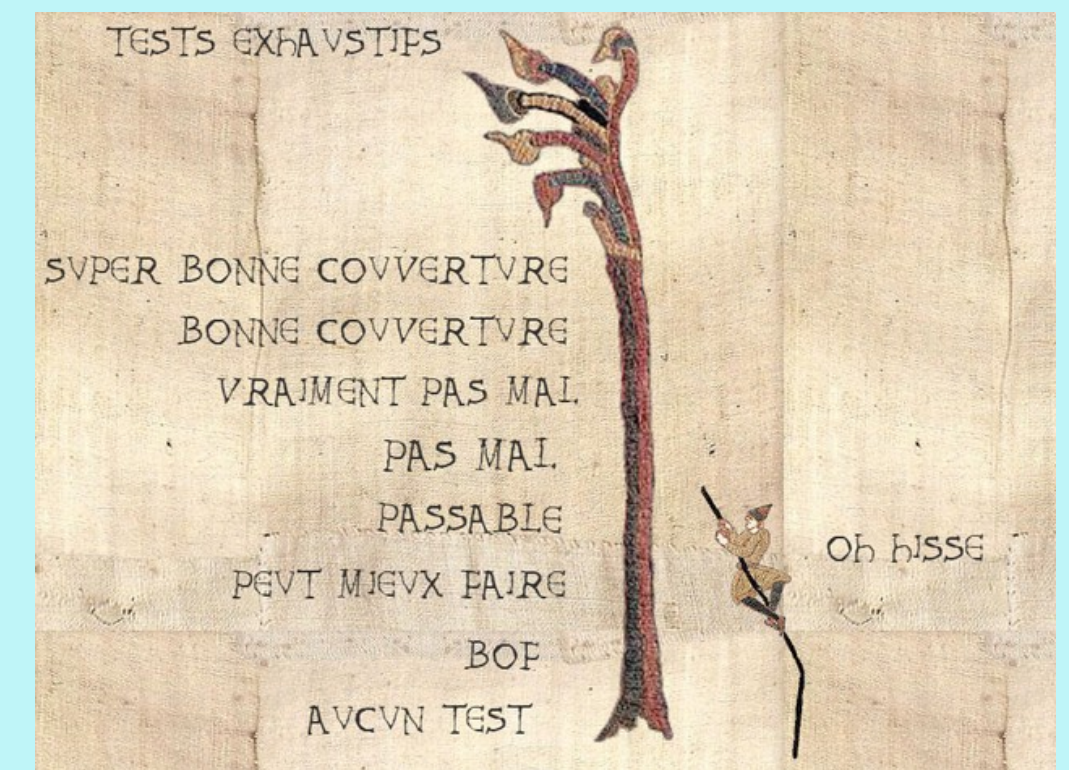
II. Le test exhaustif n'est pas possible

Il n'y a en général pas assez de ressources (matérielles et/ou humaines) pour réaliser une couverture complète de tests sur le système.

💡 Le système à tester fonctionne généralement à partir d'entrées aléatoires (interaction utilisateur, état de l'OS, heure, ...). Il n'est donc souvent pas possible de tester toutes les combinaisons d'entrées sur le système !

✍ Il est donc important de bien prioriser les tests à écrire et à exécuter !

On commencera en principe soit par les parties les plus critiques du systèmes, soit par celles dans lesquelles la confiance est minimale.

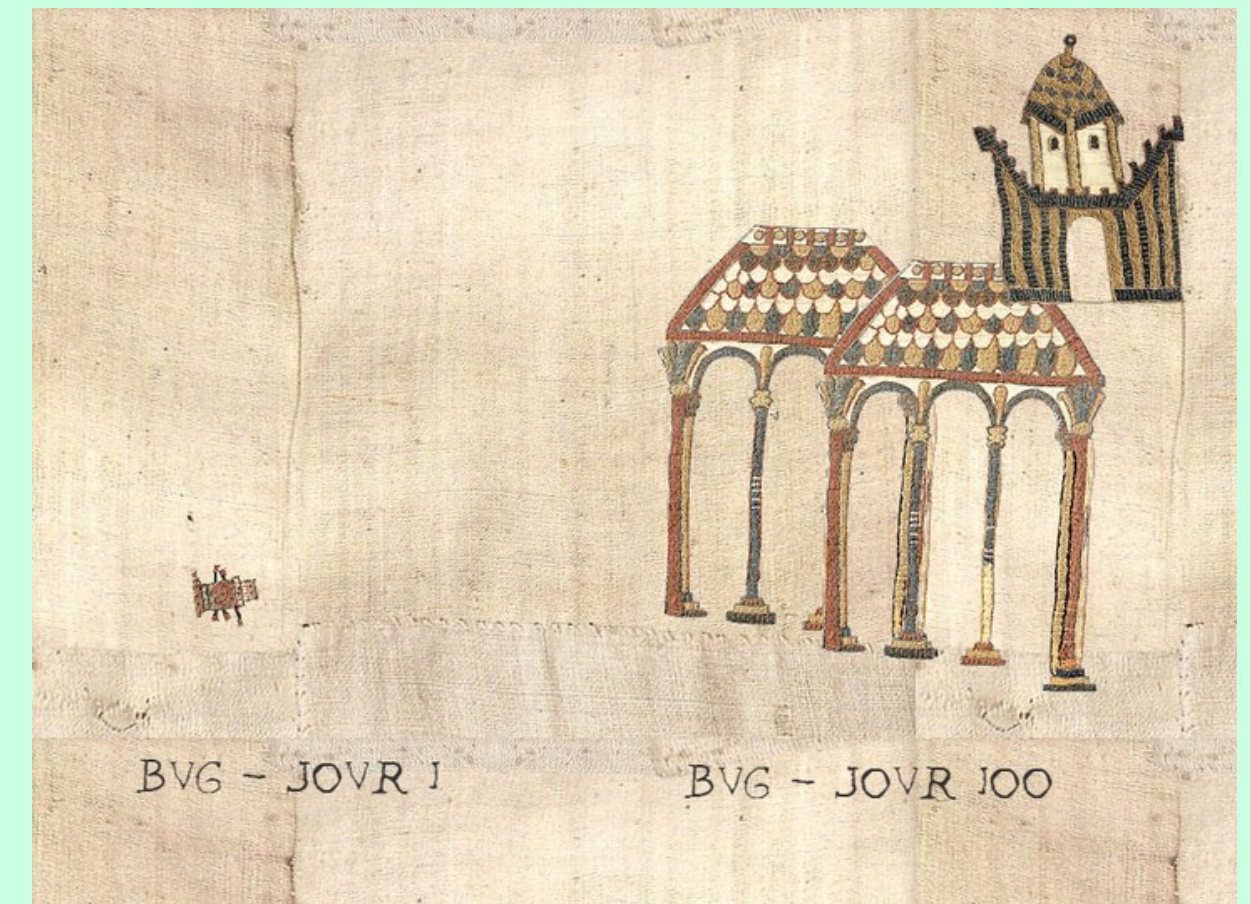


LES 7 PRINCIPES DE TEST

III. Tester dès le début

Les tests doivent suivre le développement et/ou l'intégration du système aux différents stades de son évolution. Cela permet de limiter le coût des corrections et de garantir la qualité du projet tout au long de sa réalisation.


- 💡 Trouver un problème dans les spécifications est beaucoup plus facile et sa résolution moins coûteuse que de patcher un système en production
- 💡 Il n'y a rien de pire que de réaliser que rien ne marche au moment de la dernière intégration ! Au delà des changements techniques importants à réaliser pour corriger le problème, c'est aussi une gestion de projet catastrophique qui n'a pas su évaluer les risques au cours de l'intégration.




LES 7 PRINCIPES DE TEST

IV. Les erreurs se regroupent

La majorité des problèmes d'un système ont tendance à se regrouper dans un ensemble restreint de composants et/ou de fonctionnalités.

 Il convient donc de trouver les zones de bug dans le système et de concentrer les efforts de test autour de celles-ci.

 Ce principe est une conséquence du principe de Pareto. C'est un phénomène empirique global souvent vérifié en contrôle de la qualité, qui stipule que 80% des effets proviennent de 20% des mêmes causes.



LES 7 PRINCIPES DE TEST

V. Le paradoxe du pesticide

Répéter les mêmes scénarios de test encore et encore ne permet pas de détecter de nouveau problème.

✍ Il convient donc d'ajouter et/ou de mettre à jour les scénarii de test pour trouver de nouveaux bugs.



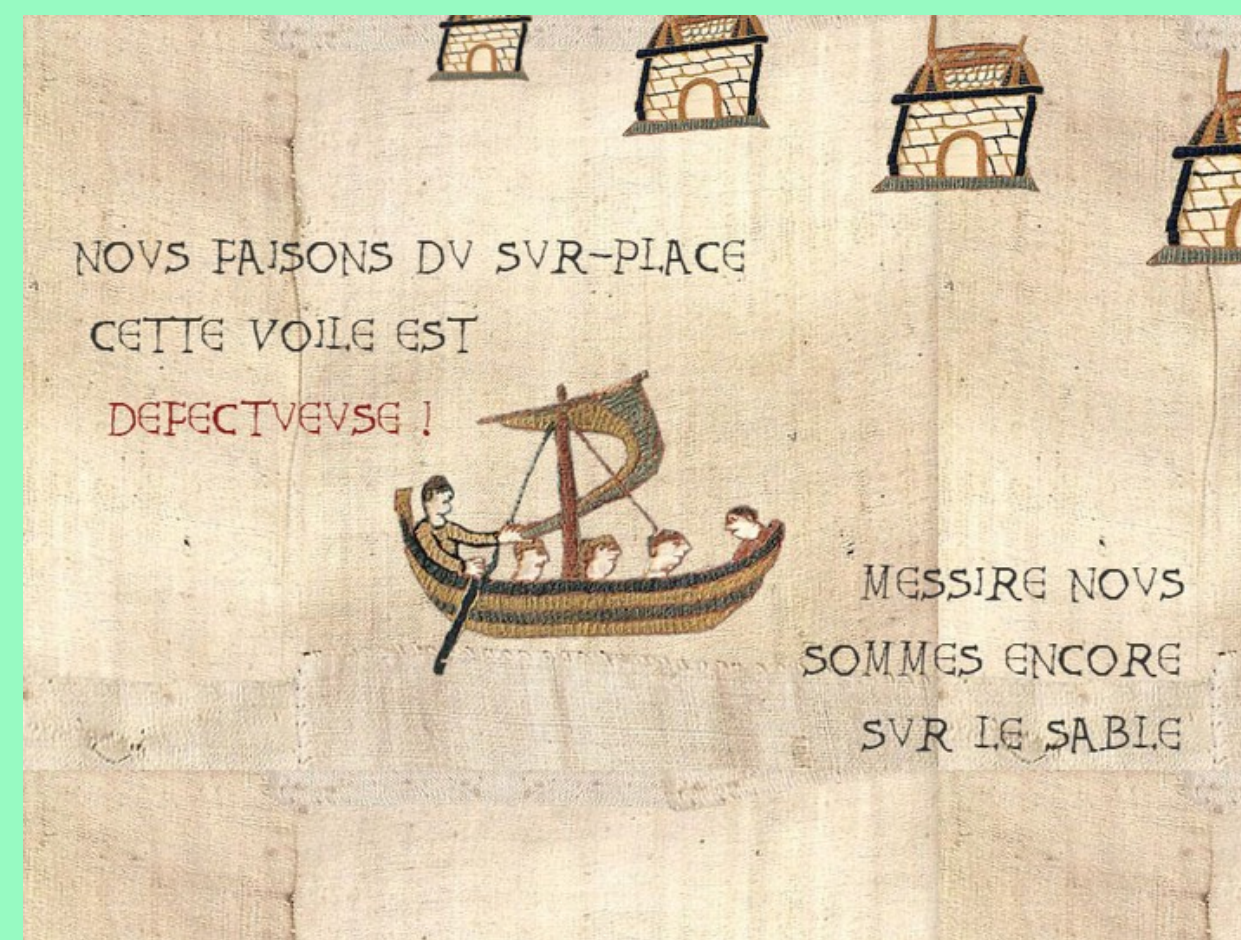
LES 7 PRINCIPES DE TEST

VI. Le test dépend du contexte

Chaque système évolue dans un contexte différent : il faut donc adapter les tests au contexte courant pour favoriser le type de test et le scénario le plus adéquat.



Par exemple, une CLI sera testée différemment d'un site Web

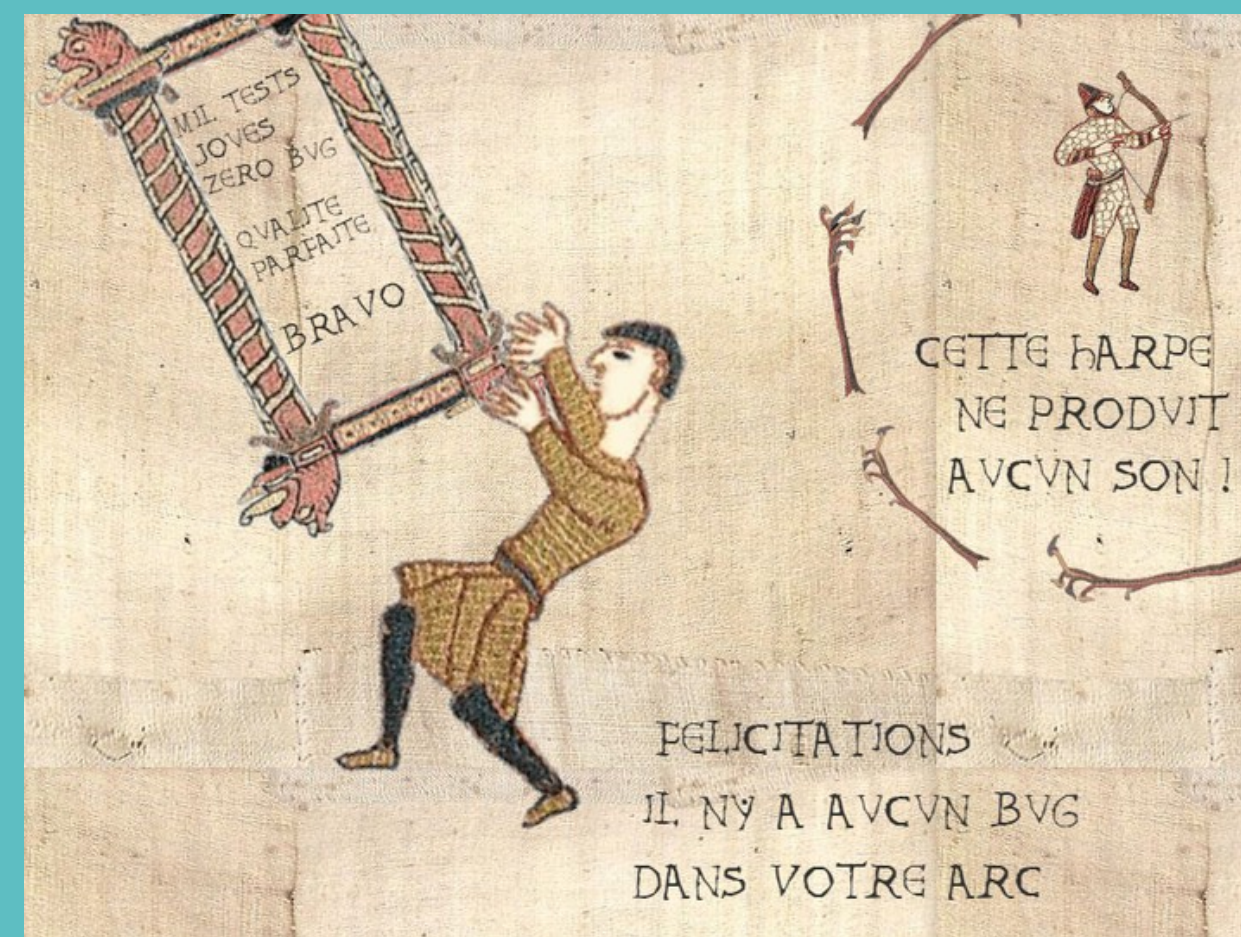


LES 7 PRINCIPES DE TEST

VII. L'illusion de l'absence d'erreur

Le système peut avoir des problèmes de comportement sans faire échouer de test. Il peut même parfaitement se comporter, mais ce comportement n'est pas celui attendu par le client !

💡 Rappel : l'absence de preuve n'est pas la preuve de l'absence ! (une erreur classique en méthodologie de tests...)



Les types de tests

Black-box vs white-box

LES TESTS BOÎTE NOIRE

Les tests boîte noire sont des tests orientés utilisateur.

Le test vérifie uniquement que les sorties du système sont celles attendues par les spécifications suite aux entrées fournies et à l'exécution du scénario de test.

Le comportement interne du système est entièrement ignoré par le test.



Le terme "boîte noire" vient du nom du système enregistrant toutes les informations de vol dans un avion lors d'un trajet. En cas de crash, c'est cette boîte qui sera analysée, sans connaître l'état de l'avion durant le vol.

LES TESTS BOÎTE BLANCHE

A l'inverse, les tests boîte blanche sont des tests orientés développeur / intégrateur.

Le test vérifie non seulement les sorties du système mais surtout que le chemin d'exécution à l'intérieur du ou des composants est celui attendu durant l'exécution du scénario de test.

Le comportement interne du système est donc connu et analysé par le test.



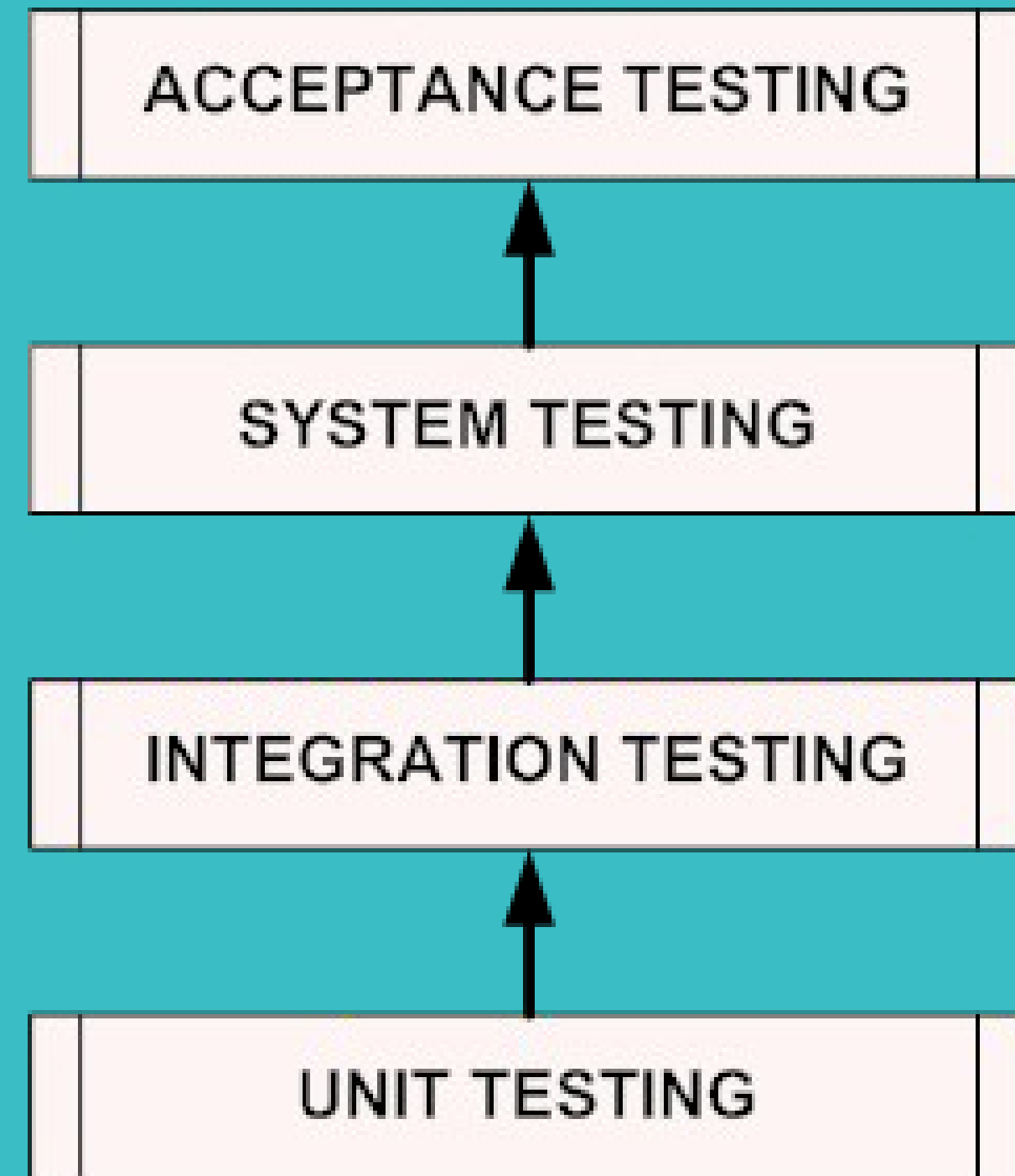
Les tests "boîte blanche" ont l'avantage d'être optimisés contre les parties critiques de l'implémentation du système (et pas uniquement en se basant sur les fonctionnalités utilisateur).

Les types de tests

Niveaux de tests

Les différents niveaux de tests

Afin de faciliter le contrôle de la qualité, on pourra séparer le produit en plusieurs sous-systèmes à tester. En fonction de ce découpage, on utilisera différents niveaux de tests décrits ci-après.



LES TESTS UNITAIRES

Ce sont des tests réalisés sur des unités individuelles et isolées du système.

Ce sont les plus simples à coder, mais aussi les plus dépendants de l'implémentation du système et donc les plus instables en cas de modification dans le système, même si ces changements n'ont pas d'impact pour l'utilisateur. Les tests unitaires sont toujours automatisés.



On pourra par exemple tester les variables d'une méthode, le contenu d'un fichier de configuration, ...



Les tests unitaires sont en général des tests boîte blanche, même s'il est possible de réaliser des tests unitaires boîte noire.



Les tests unitaires documentent l'utilisation et les entrées de chaque unité prise individuellement



Les tests unitaires sont aussi appelés tests de composants

LES TESTS UNITAIRES

Les tests unitaires sont les plus utilisés par les développeurs car au plus proche de l'application et simples à mettre en oeuvre.

Ceux-ci peuvent être réalisés manuellement : on utilise alors le débogueur d'un environnement de développement (IDE). Celui-ci permet d'exécuter étapes par étapes une partie du code dans un environnement isolé, et d'inspecter le contenu des variables et de la mémoire à chaque fois que le programme est mis en pause.

On préfèrera cependant automatiser ces tests autant que possible, en utilisant des frameworks dédiés : Junit pour Java et les équivalents *Unit étant les plus utilisés

LES TESTS D'INTÉGRATION

Ce sont des tests qui combinent des modules du systèmes pour les tester ensemble.

Ces tests se focalisent souvent sur les interfaces aux limites des modules (i.e. les interfaces permettant leur assemblage en un système unifié). Les tests d'intégration sont souvent automatisés. Ce sont des tests globalement stables si les ressources externes et les spécifications techniques ne changent pas.



On pourra par exemple tester l'intégration du module de login centralisé sur un composant du système



Un module du système peut être une ressource externe (base de données, ...)



Les tests d'intégration peuvent être aussi bien des tests boîte noire que boîte blanche.



Les tests d'intégration documentent les API internes qui permettent d'interconnecter les modules du système.

BOTTOM-UP vs TOP-DOWN

Les tests d'intégration peuvent être :

- BOTTOM-UP (du bas vers le haut) : on teste unitairement les modules avant de les assembler pour tester **uniquement leur interfaçage**
- TOP-DOWN (du haut vers le bas) : on teste directement la fonction sur le bloc de modules assemblés, puis on descend dans chaque module pour valider le comportement du module.

LES TESTS SYSTÈME

Ce sont des tests d'intégration qui combinent tous les modules du système pour les tester dans le contexte d'un système entièrement intégré, proche de la production.

Ces tests peuvent utiliser les interfaces utilisateur ou les API internes du système, et peuvent tester des fonctionnalités complètes ou des morceaux d'intégration (login, ...). Les tests système peuvent être automatisés mais une partie au moins de ceux-ci est généralement effectuée manuellement.



On pourra par exemple tester l'intégration d'une base de données utilisée par l'ensemble des modules du système



Les entrées de ces tests sont les besoins techniques du système unifié.



Les tests système sont presque exclusivement des tests boîte noire.





Les tests d'intégration documentent les fonctionnalités du système, y compris celles qui sont cachées à l'utilisateur final.


LES TESTS D'ACCEPTATION (ou END-TO-END)


Ce sont des tests formels qui utilisent les fonctionnalités utilisateur fournies par le client final. Les tests d'acceptance sont proches des tests système mais se focalisent sur le point de vue de l'utilisateur final du produit.

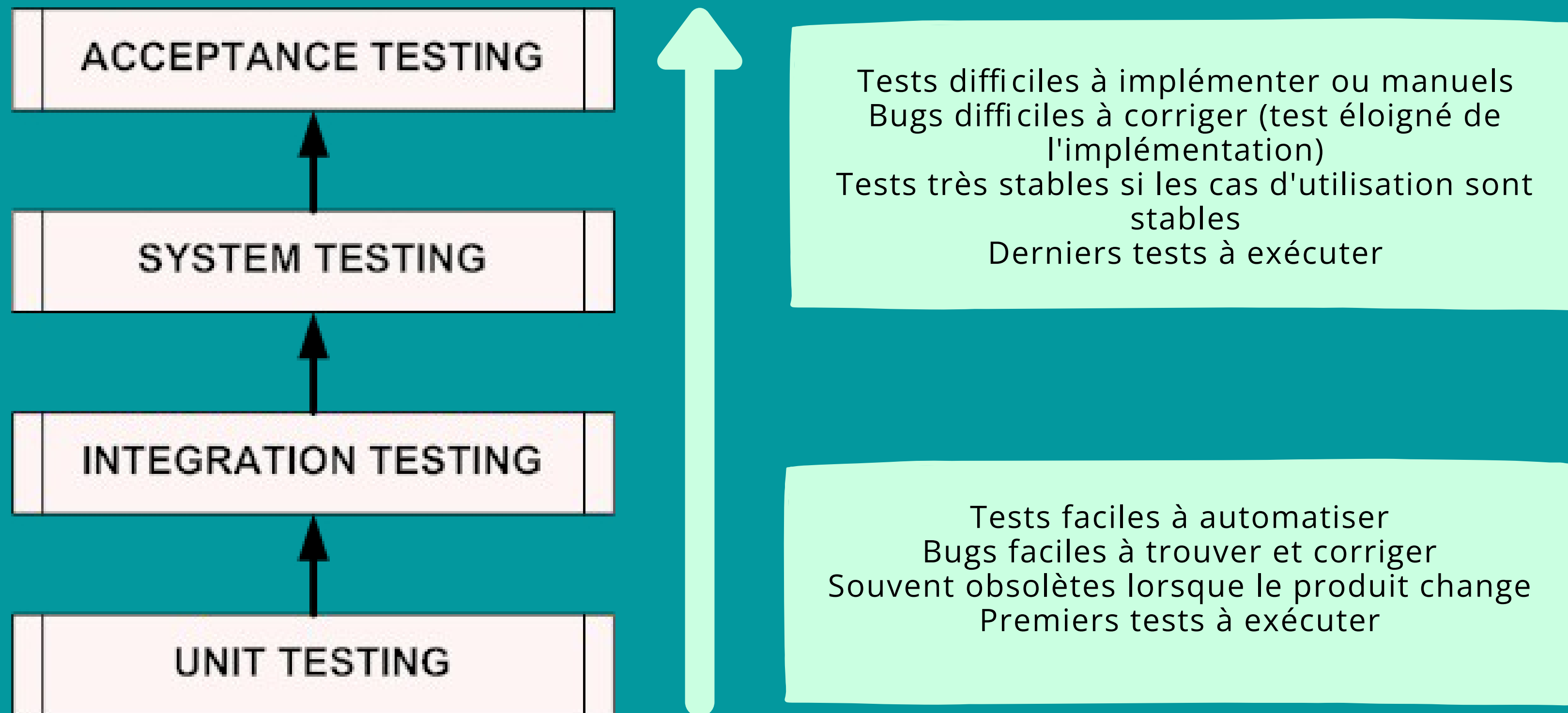
Contrairement à tous les autres types de tests, ils peuvent ne pas être décrits de manière formelle pour analyser le comportement de l'utilisateur (version alpha, ...)

 On pourra par exemple tester l'ajout d'un objet dans le panier de l'utilisateur.

 Les entrées de ces tests sont les cas d'utilisation du produit.

 Les tests d'acceptance sont exclusivement des tests boîte noire.
Les tests boîte blanche sont interdits dans ce type de test pour ne pas perturber le système.

 Les tests d'acceptance documentent les spécifications business du produit, c'est-à-dire l'utilisation attendue du produit par le client final.




Avantages et inconvénients des différents niveaux de tests

Les techniques de tests

Fonctionnel vs non-fonctionnel

LES TYPES DE TESTS : FONCTIONNEL VS NON-FONCTIONNEL

De manière orthogonale aux niveaux de tests (dépendant surtout du système à tester), on pourra utiliser différents types de tests dans chaque niveau, en fonction de ce que l'on veut vérifier.

 On sépare ces types de tests en deux grandes catégories : les tests fonctionnels qui valident le fonctionnement du produit, et les tests non-fonctionnels qui testent à quel point le produit fonctionne bien.

Tests fonctionnels

LES TESTS FONCTIONNELS

Comme leur nom l'indique, ce sont des tests qui se focalisent sur les fonctionnalités du produit.

Ces tests sont toujours exécutés en premier.

Les tests fonctionnels peuvent être manuels ou automatisés - ce sont des tests faciles à réaliser à la main.



Les tests fonctionnels documentent ce que fait le produit.
Ils répondent à la question : le programme respecte-t-il les cas d'utilisation ?

EXEMPLES DE TESTS FONCTIONNELS

Les tests d'acceptance

Tests Alpha (par l'Assurance Qualité)

Les tests alpha sont réalisés avant la livraison du produit. Ils testent les besoins business. Ces tests sont souvent manuels mais peuvent être automatisés.



α β

Tests Beta (par l'Utilisateur final)

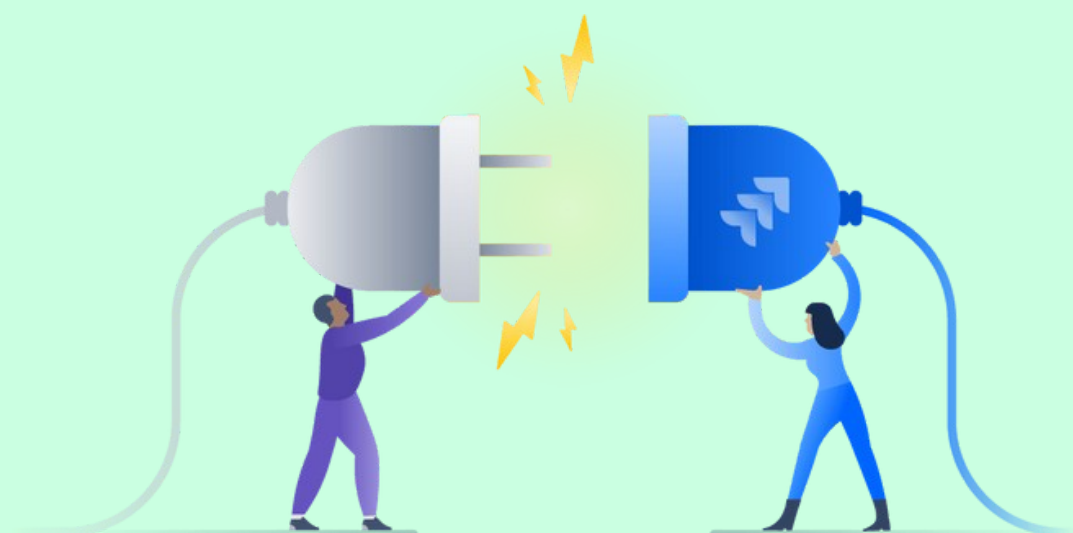
Les tests de beta sont souvent réalisés par un ensemble restreint d'utilisateurs finaux.

Les scénarii de ces tests ne sont pas décrits formellement : l'utilisateur "joue" avec le produit et sa manière d'interagir avec celui-ci fait souvent partie du résultat du test.

EXEMPLES DE TESTS FONCTIONNELS

Les tests unitaires

Ce sont des tests courts, dédiés, reproductibles et isolés.



Les tests d'intégration

Ce sont des tests sur des assemblages de modules déjà testés individuellement.

Les tests de bonne santé (sanity checks)

Ces tests valident que le changement opéré sur le produit a bien résolu le problème qu'il est censé avoir résolu.



EXEMPLES DE TESTS FONCTIONNELS

Les tests de non-régression

Ce sont des tests qui vérifient qu'une nouvelles fonctionnalité ou un changement de fonctionnalité ne va pas altérer le comportement de l'existant.



Les tests de démarrage (smoke tests)

Ce sont des tests (souvent un seul) qui vérifient que le produit final peut démarrer. Le smoke test est lancé comme garde-fou avant les autres tests système et ne vérifie aucun comportement.

Par exemple, on pourra vérifier qu'un serveur assemblé démarre après avoir branché l'alimentation, ou qu'un site Web est accessible une fois déployé.

EXEMPLES DE TESTS FONCTIONNELS

Les tests de maintenance

Un système déjà déployé doit régulièrement subir des opérations de maintenance, qu'elles soient applicatives ou matérielles (changement de code, ajout/mise à jour de dépendences, remplacement de disque dur, ...)

L'opérateur doit être en mesure de valider que les changements appliqués remplissent le besoin de la maintenance.

Le but de ces tests est donc :

- de qualifier l'opération de maintenance en vérifiant que la modification à apporter est bien celle qui était attendue
- de s'assurer que l'opération de maintenance n'a pas créé de régression dans une autre partie du produit (ou dans un autre produit)

Tests non-fonctionnels

LES TESTS NON-FONCTIONNELS

Comme leur nom l'indique, ces tests ne vérifient pas les fonctionnalités du produit. Que testent-ils alors ? Ils vont vérifier la performance, la fiabilité, la capacité évolutive et d'autres aspects non-fonctionnels du système.

Ces tests sont toujours exécutés après les tests fonctionnels car ils ont besoin que certaines opérations du produit fonctionnent pour être lancés.

Ils sont souvent automatisés, car très difficiles à reproduire manuellement.



Les tests non-fonctionnels documentent à quel point le produit fonctionne bien.
Ils répondent à la question : l'utilisation du produit est-elle appréciable ?



Les tests non-fonctionnels ne sont pas toujours bloquants (i.e. un échec n'est pas toujours critique pour le déploiement du produit) mais ils restent très importants pour qualifier un comportement attendu du système (performances, sécurité, ...)

EXEMPLES DE TESTS NON-FONCTIONNELS

Les tests de stress

Ce sont des tests qui qualifient le comportement du produit en situation défavorable : mémoire ou espace disque limités, fonctionnement dans une machine virtuelle, ...



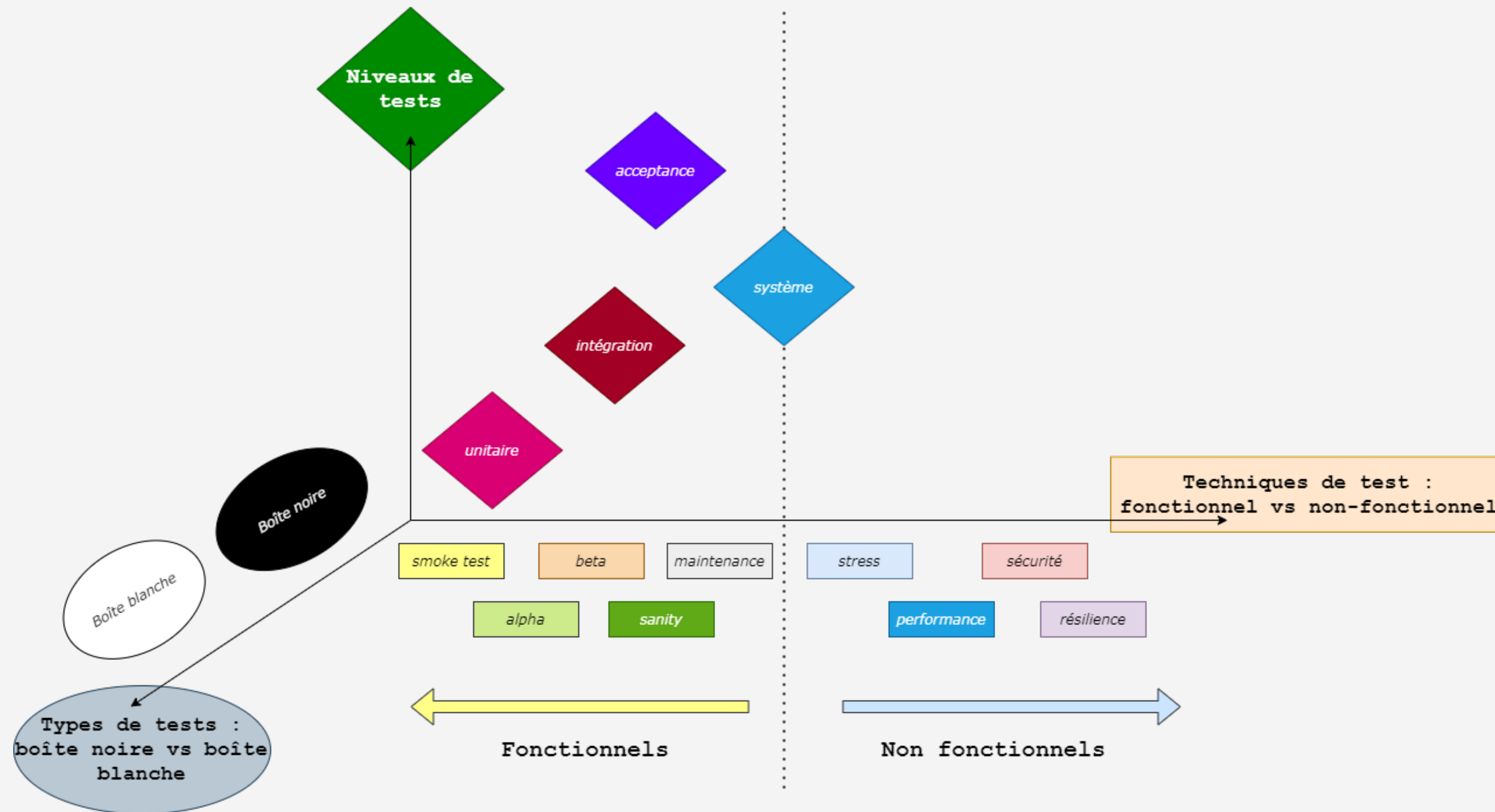
Les tests de performance

Ces tests qualifient les performances d'exécution du produit entièrement assemblé.

Les autres tests

Il existe de très nombreux types de tests non-fonctionnels qui valident ou qualifient chacun un aspect particulier du produit : tests de charge, de sécurité, de récupération, d'installation, d'intrusion, de compatibilité, de migration, ...

LES 3 DIMENSIONS DE TEST



Partie II : Les étapes de test

Comment écrire un test ?
Quelles étapes ?

Les étapes de test

ÉTAPE I : DÉCRIRE LE TEST

On se posera la question suivante : Quels sont les tests d'intégration et d'acceptance obligatoires pour valider le service ?

Pour cela, on utilisera les spécifications du service pour en extraire une liste de tests. A cette étape, les scénarios de test ne sont pas encore rédigés : on se limitera à un titre ou une phrase courte décrivant succinctement l'idée du test.



Il est important, dès l'étape de description du test, de penser aux entrées et aux résultats attendus

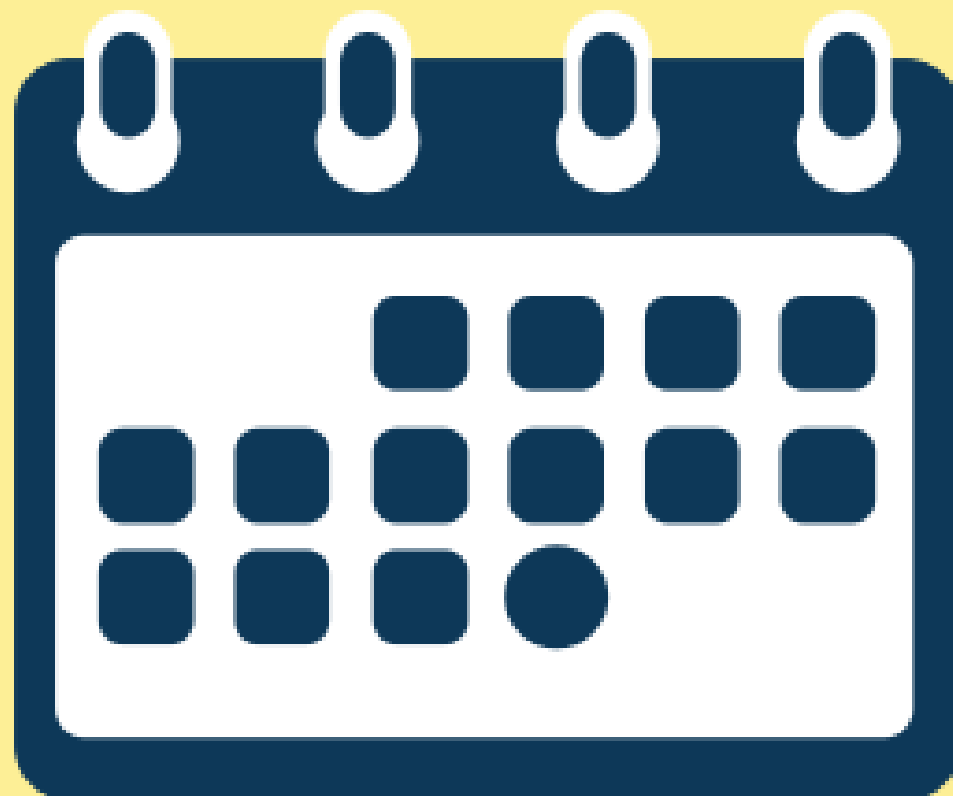


Les tests doivent être budgétisés ! Penser à décrire succinctement le coût du test : son implémentation (rédaction du scénario, automatisation, ...) et son exécution (temps nécessaire, ressources humaines et/ou matérielles)

ÉTAPE I BIS : CRÉER DES PLANS DE TESTS

Pendant l'étape de description des tests, on pourra réfléchir en parallèle à la création de plans de tests.

Un plan de tests décrit une suite de tests à exécuter dans un certain contexte (par exemple : avant la livraison d'une nouvelle version).



Un même test peut donc se retrouver dans différents plans de tests. Un test décrit une vérification à effectuer, un plan de test décrit quand tourner ce test.

ÉTAPE II : PRÉPARER LES ENTRÉES

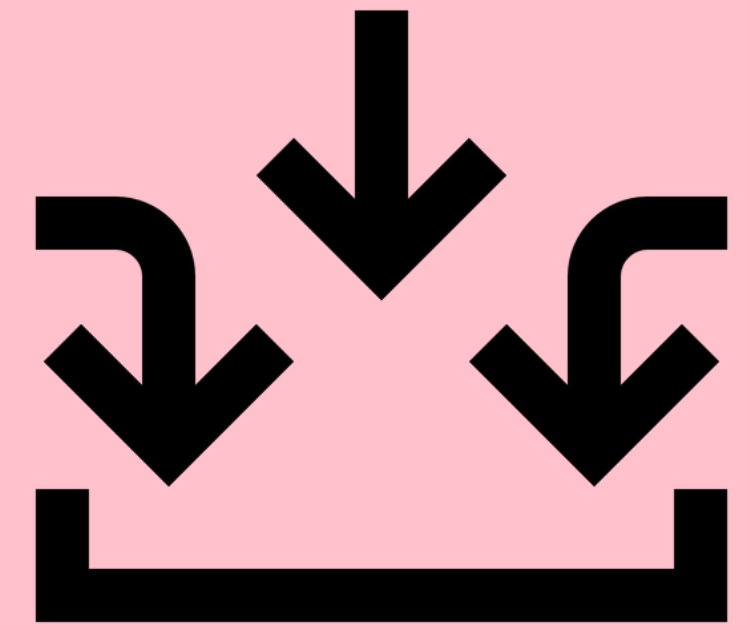
La deuxième étape consiste à préparer les entrées du test. Cela inclut également l'état du système avant l'exécution du test.



Il est possible de fournir uniquement une description textuelle très concrète, étape par étape et quantitative sur la manière de générer les entrées et comment préparer le système pour le test



Cependant, lorsque c'est possible, il est toujours préférable d'automatiser la préparation des entrées (programmation, exemples, archives de l'état du système, ...). Cela permet un gain de temps mais limite également les risques d'erreur lors de l'étape d'initialisation (critère reproductible du test).



ÉTAPE III : ÉCRIRE LE SCRIPT

La troisième étape consiste à préparer le script du scénario de test. Le script d'un test doit toujours être évident à comprendre : le testeur ne doit jamais interpréter une étape du script (sinon le test ne serait plus reproductible !)



💡 Le scénario du test peut être réalisé manuellement ou automatisé - comme pour les entrées, on préférera automatiser l'exécution du test lorsque c'est possible (et peu coûteux).

- 💡 Les scripts des tests d'acceptance sont plus souples que ceux des tests d'intégration :
- Ce qu'il faut faire doit être évident
 - Comment le faire peut être laissé à l'interprétation du testeur

Cela permet de tester une expérience utilisateur, plutôt que le déroulement d'un scénario tel qu'imaginé par le créateur du test

ÉTAPE IV : PRÉPARER L'ENVIRONNEMENT

La quatrième étape consiste à préparer l'environnement d'exécution pour le test. Cela implique l'état initial du système et les entrées du test tel que décrit à l'étape 2, mais aussi les éventuels besoins du test lui-même : initialisation du framework, préparation des systèmes de logs, ...



Attention à bien s'assurer de ne pas ajouter de choses en trop dans l'environnement qui pourrait changer le comportement du test : un test décrit un ensemble de besoins minimaux et suffisants



Cette étape est cruciale dans le bon déroulement des tests : ne pas hésiter à vérifier minutieusement l'état initial de l'environnement !

Une grande majorité des faux négatifs (test échouant alors que le produit se comporte convenablement) proviennent d'un mauvais environnement et sont difficiles à investiguer (données restantes en base de données, ...)

ÉTAPE V : EXÉCUTER LE TEST

La cinquième étape consiste à exécuter le script du test. Cette étape peut être à réaliser par un testeur, ou entièrement automatisée suivant la préparation du test.



En cas d'erreur du testeur durant l'exécution : reprendre l'étape de préparation de l'environnement et recommencer tout le test !

Si cela n'est pas possible, penser à noter les changements opérés par rapport au script initial dans le rapport d'exécution (étape suivante). En effet, il ne s'agit plus exactement du même test et cela pourrait poser des soucis de reproductibilité

ÉTAPE VI : CONSIGNER LES RÉSULTATS

La sixième et dernière étape consiste à consigner les résultats du test. Cette étape s'effectue juste après l'exécution du test et est sûrement la moins anodine même si elle est souvent négligée.

Elle peut être fortement automatisée mais elle est le plus souvent réalisée manuellement.

Les résultats du test doivent être détaillés minutieusement, en expliquant par exemple ce qui a provoqué le problème, quelle(s) investigation(s) ont été menée(s), quel impact pour l'utilisateur, comment contourner le problème, comment le corriger, ...



Ajouter autant d'information que possible provenant du système : journaux de logs, état des services internes/externes au système, stack traces ou messages d'erreur, ...



Un rapport de test ne se limite pas à un rapport d'erreurs !!!

Même en cas de succès, ce rapport doit décrire comment les résultats ont été vérifiés, les actions entreprises sur le système, les remarques éventuelles, ... (par exemple, un temps d'exécution plus important que la normale).

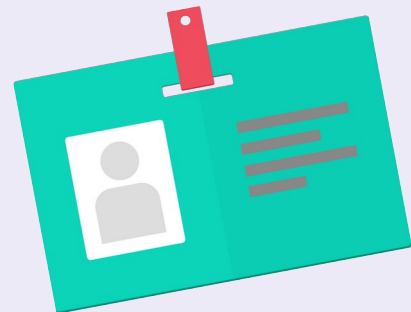
Données de test

UTILISER DES DONNÉES DE RÉFÉRENCE

Les tests d'intégration ont vocation à être exécutés lors de chaque modification du système. Pour certifier leur répétabilité et faciliter leur cohérence, on utilise un jeu réduit de données communes, proches de données réelles en production.

Ces données peuvent prendre plusieurs formes, et pas seulement celles de données statiques :

- Bases de données que l'on va recopier (solution basique mais qui rend plus difficile le versionning des données avec le code)
- Scripts SQL
- Données dans des fichiers de configuration (XML par exemple)
- Code permettant d'insérer les données de manière paramétrable, éventuellement par une API minimaliste dédiée.



Les données de référence sont généralement générées manuellement, en s'inspirant de données trouvées en production, ou en anonymisant ces données.

UTILISER DES DONNÉES DE PRODUCTION

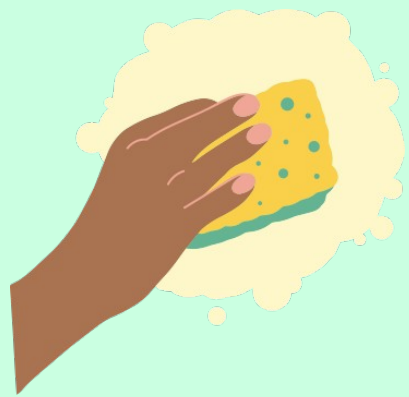
Cette approche a l'avantage de demander peu d'investissement quand on peut s'appuyer sur les outils de sauvegarde et de rechargement déjà en place. Elle présente cependant certains inconvénients :

- Confidentialité : utiliser des données client est risqué, voir interdit par la réglementation (GDPR, ...)
- Les données peuvent ne pas exister encore (nouvelle fonctionnalité, changement de format des données, ...), ou ne plus exister en production (résiliation du client)



GESTION DES DONNÉES DE TEST

- 💡 Maintenir ces données présente un coût. Il peut être pertinent de développer des outils permettant d'extraire des données de la production et de les anonymiser afin d'en faire des données de référence.



- 💡 Le processus de réinitialisation des données : il est nécessaire de pouvoir régulièrement remettre les données à zéro sur les environnements d'intégration. Il faut donc un processus permettant de le faire, si possible de manière automatisée.

Partie III :

Développement piloté

par les tests

Quelles méthodes ?

Quels impacts sur le cycle de développement ?

Les bases du TDD

PROBLÈME DU TEST TARDIF

Dans les méthodes "classiques" de développement, on écrit l'implémentation du code avant de réaliser les tests.

Cela crée plusieurs problèmes :

- La vérification arrive tard dans le cycle de développement : en cas d'erreur, il faut tout recommencer
- Aucune information sur la qualité pendant le développement
- Comment savoir si le test couvre réellement le problème à inspecter ?
- L'architecture n'est pas pensée pour être testé : le test peut être compliqué

TEST-DRIVEN DEVELOPMENT (TDD)

Pour pallier à ces problèmes, on peut utiliser le Test-driven development (TDD). Dans cette méthode de développement, le test doit toujours être écrit avant l'implémentation qu'il vérifie, et ce afin de certifier que le test lui-même est correct !

On procède de la manière suivante :

- Ecrire un (ou plusieurs) test unitaire couvrant un code qui n'existe pas encore à cette étape, ce test doit échouer : c'est ce qui garantit que le test vérifie bien ce qui est attendu !
- L'implémentation est alors ajoutée
le test existant nous impose une architecture testable facilement
lorsque le test devient positif, c'est que l'implémentation fournit bien le besoin attendu

ARRANGE, ACT, ASSERT

La méthode "Arrange, Act, Assert" est la méthode standard d'écriture de scénarios de tests. Cette méthode permet de visualiser facilement les différentes étapes du test en les séparant en 3 parties :

- Arrange : préparation du système à tester, initialisation des données, création des objets, ...
- Act : réalisation des actions de test sur les objets et méthodes créés à l'étape précédente
- Assert : vérification des résultats du test

```
// arrange  
var repository = Substitute.For<IClientRepository>();  
var client = new Client(repository);
```

```
// act  
client.Save();
```

```
// assert  
mock.Received.SomeMethod();
```

REFACTORING EN TDD

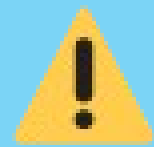
Le TDD offre une confiance forte lors d'un refactoring de code.

En effet, toutes les fonctionnalités ont été implémentées avec une batterie de tests validant leur fonctionnement. Les changements simples sont donc souvent testables directement. Dans le cas d'un refactoring d'un code existant sans TDD, on peut aisément démarrer le refactoring en TDD.

Lors d'un développement en TDD, l'architecture est fortement orientée par les tests. Dans la pratique, celle-ci est souvent bien meilleure que dans un développement traditionnel mais cela n'exclut pas un refactoring afin de l'optimiser une fois l'implémentation terminée.

On ajoute donc une étape de refactoring après l'écriture de l'implémentation, avant de passer à la fonctionnalité suivante.

REFACTORING EN BDD



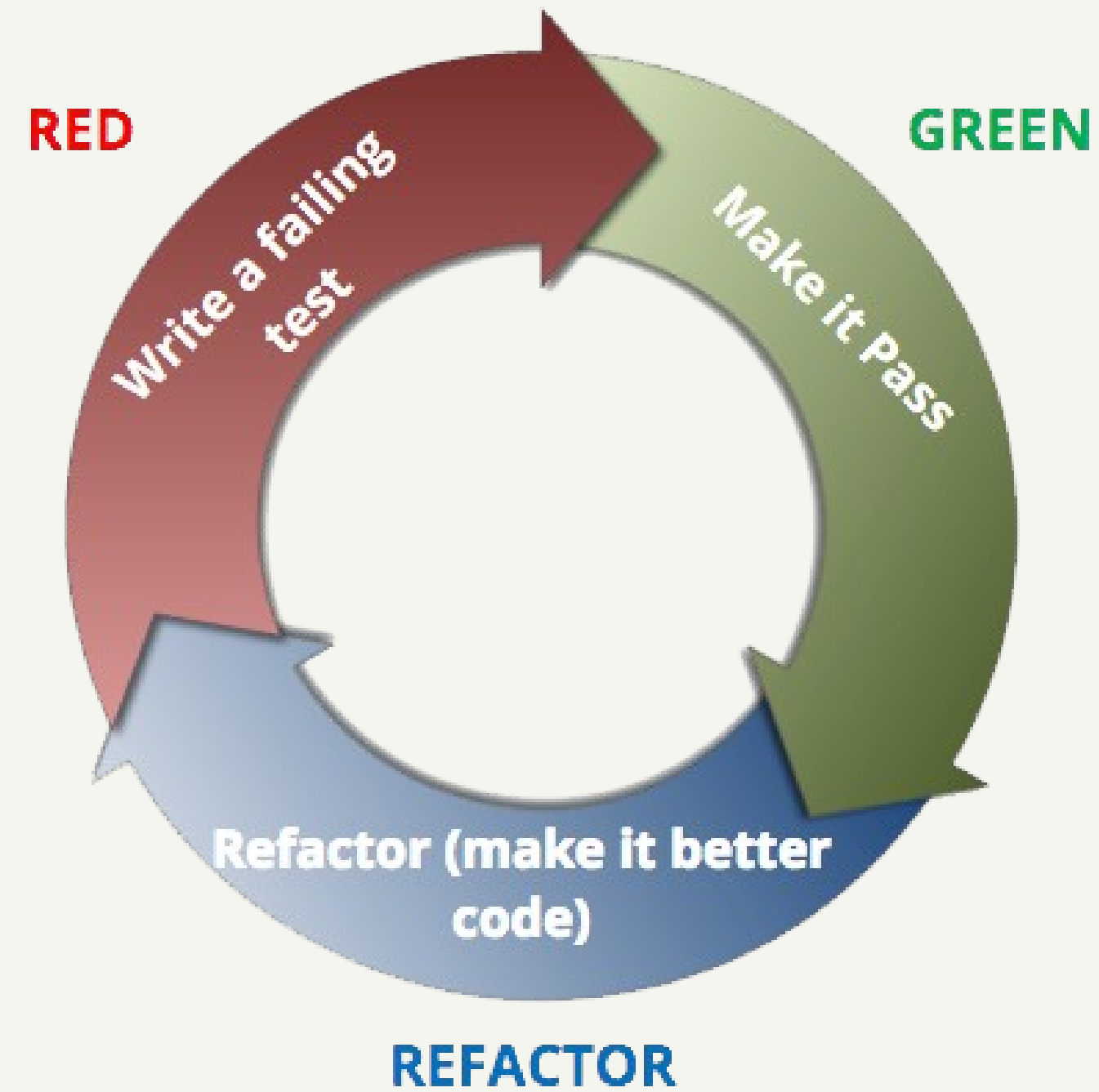
Les tests unitaires en TDD sont des tests unitaires très proches de l'implémentation : des changements forts d'architecture demandent une réécriture massive des tests, ce qui est coûteux et risqué.

Pour pallier à ce problème, on peut remplacer le TDD par du Behavior-Driven Development (BDD). Lors d'un refactoring, l'implémentation varie alors que le besoin métier n'a pas évolué. Plutôt que de tester l'implémentation par des tests unitaires, le BDD teste directement les fonctionnalités du programme qui évoluent moins.

CONCEPTION ÉMERGENTE

Le TDD est en fait plus une pratique de développement incrémental (issue de l'eXtreme Programming et des méthodes agiles), plus qu'une méthodologie de test. En TDD, les choix technologiques, d'architecture et d'implémentation sont repoussés au moment de l'implémentation et la conception évolue au fil de l'eau pour satisfaire des critères de qualité. Cette conception est dite "émergente".

CONCEPTION ÉMERGENTE



GESTION DES EXCEPTIONS

En TDD, tout le code implémenté doit répondre à des tests écrits en amont, y compris les exceptions !

Pour cela, on injecte les erreurs dans un test dédié et on vérifie que l'exception a bien été générée (soit par un catch manuel, soit automatiquement si le framework de tests le permet).

Ainsi, ces tests :

- Documentent les problèmes pouvant survenir dans le code lorsque celui-ci est appelé par du code client
- Décrivent le contexte et les conditions générant les exceptions
- Expliquent le comportement attendu du client en cas d'erreur

En pratique, l'obligation de tester les exceptions aboutit à des contrats beaucoup plus propres, avec bien moins de catch blocks vides. Le développeur est forcé à penser aux problèmes pouvant arriver et surtout, comment y réagir.

GESTION DES SCÉNARIOS

Les scénarios de test permettent de documenter l'usage des différentes classes et APIs du produit. La majorité des frameworks de TDD permettent d'extraire l'histoire racontée par le test, pour consigner l'ensemble des scénarios du produit. Dans le cas du BDD, cela permet d'extraire les différents cas d'utilisation du produit, dans un format échangeable avec les profils non techniques du projet.

TDD *avancé*

CORRECTION DES ANOMALIES



En TDD, toute correction de bug suit le même cycle de développement que l'ajout d'une nouvelle fonctionnalité :

- Ajout d'un nouveau test reproduisant l'erreur détectée et échouant sur l'implémentation courante
- Modification de l'implémentation pour faire passer le test
- Refactoring si nécessaire

Cette méthode de correction a l'énorme avantage de reproduire l'erreur avant de la corriger, afin de valider que le changement corrige complètement le problème (au lieu d'écrire un test validant le nouveau comportement attendu, sans savoir s'il corrige vraiment le problème rencontré).

GESTION DE LA MONTÉE EN CHARGE

Il est souvent trop tard lorsque l'on s'interroge sur la mise en place de tests de montée en charge : problèmes en production, architecture invalide, ...

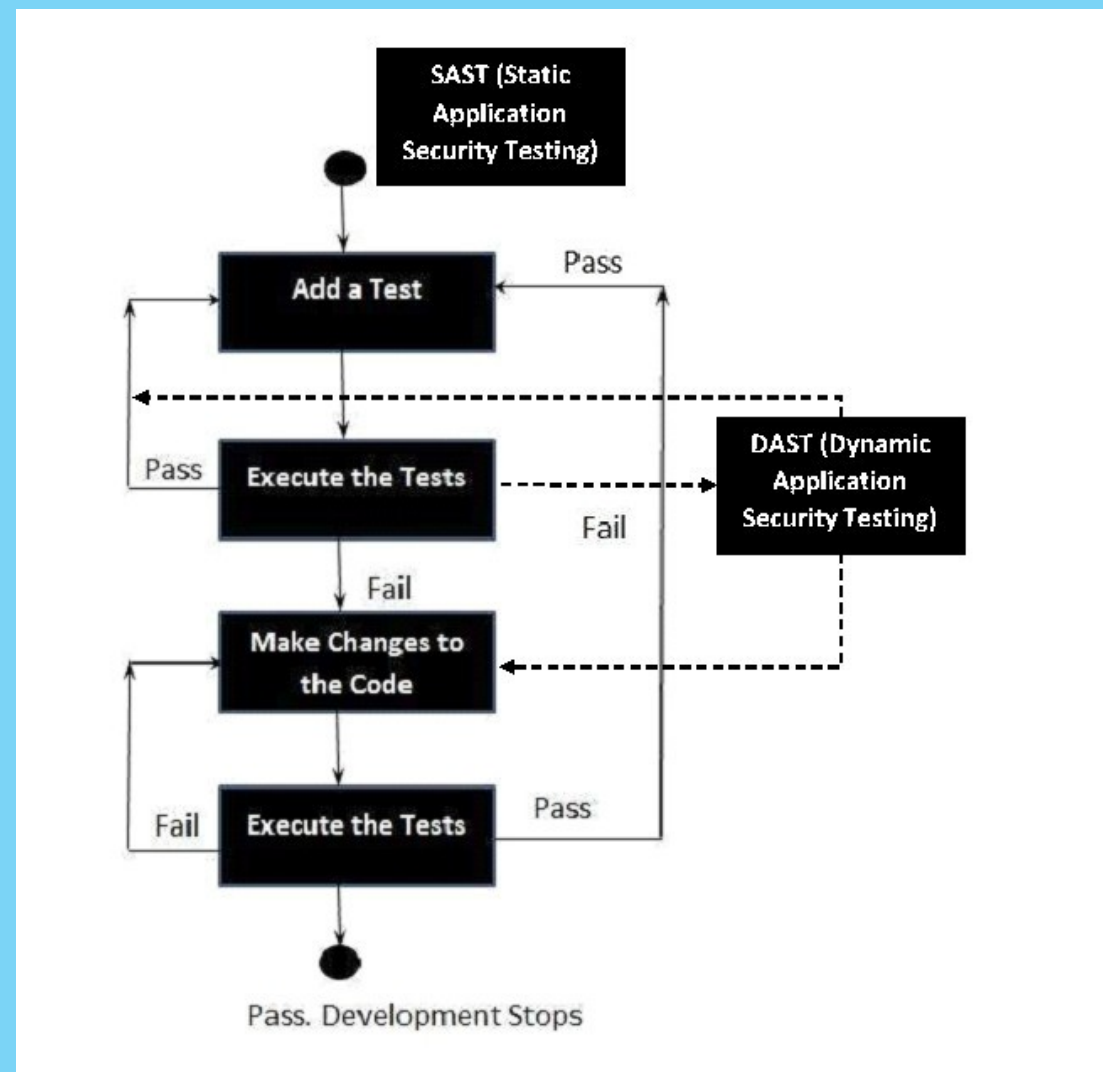
La performance doit être un processus continu :

- Prototypage : choix techniques pour définir au mieux les technologies et le produit
- Développement : anticipation des dysfonctionnements pour une architecture saine
- Recette : phase la plus propice - développements stabilisés mais modifiables
- Pré-production : pour dimensionner la production (mémoire, CPU, connexions BD, nombre de processus, ...)
- Production : tests en conditions réelles mais souvent dans l'urgence

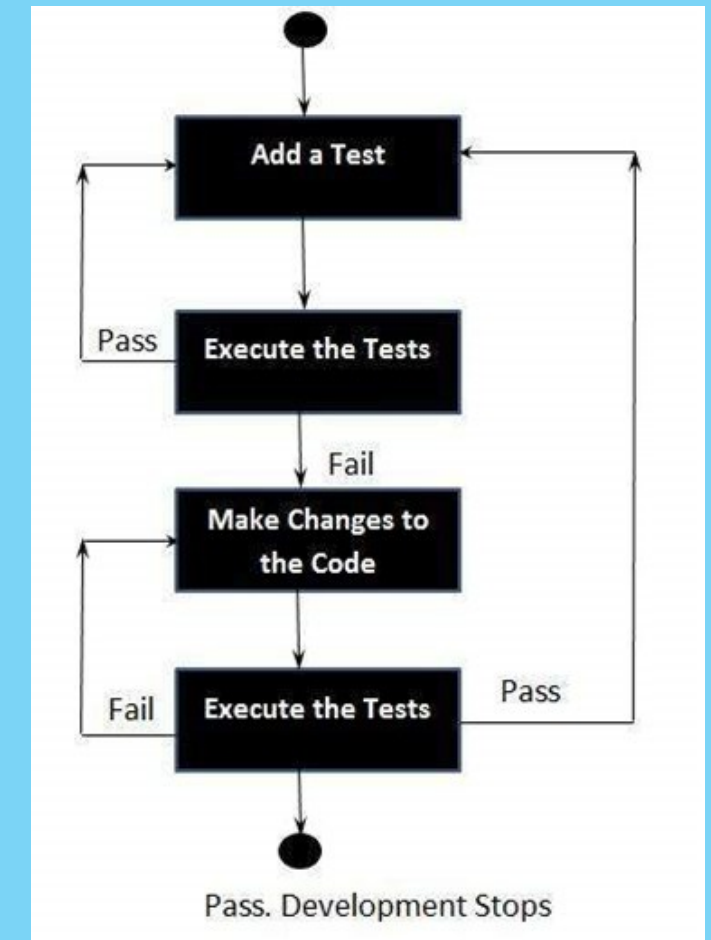
Apache JMeter est l'outil de tests de charge le plus répandu pour tout ce qui repose sur le protocole http.

GESTION DE LA SÉCURITÉ

De manière similaire, la sécurité doit être pensée dès le début du projet. On pourra enrichir le TDD en STDD (Security Test Driven Development) en ajoutant des analyses statiques et dynamiques de vulnérabilité de code dans le processus TDD.



Le cycle STDD



Rappel : cycle TDD

On pourra utiliser des outils tels : Lint (Android), SonarCube (Java), ...

GESTION DE LA PERFORMANCE

Le TDD implique l'écriture d'une batterie de tests conséquente, qui sera exécutée en continue.

La performance des tests doit donc être une priorité pour que l'expérience de développement ne devienne pas rapidement un cauchemar !

- On isolera au maximum les vérifications des différentes parties du code pour éviter de devoir exécuter tous les tests à chaque fois (classes de tests liées à une classe métier, séparation par packages, ...)
- On parallélisera l'exécution des tests : ceux-ci devront donc être tous indépendants
- Les tests en TDD n'intègrent pas de dépendance externe (base de données, ...) qui seront simulées, et on utilisera massivement des Stubs et Mocks pour simuler les classes non nécessaires dans le contexte du test

Les outils open source et commerciaux

ARCHITECTURE DE TESTS

Pour faciliter la gestion opérationnelle des différents cas de tests, on regroupe ceux-ci en plans de tests.

Un plan de test correspond à la planification de l'exécution d'un ensemble de cas de tests, sur un environnement défini et dans un certain contexte (par exemple : tests vérifiant une correction en pre-production avant application du patch en production).

TestLink est l'outil le plus utilisé pour gérer les scénarios de tests, les plans de tests et les rapports d'exécution.

Son modèle très générique permet une gestion unifiée de tous les besoins de tests d'un projet.

COUVERTURE DE TESTS : AXIOMES ET OUTILS

Pour développer des applications js ou Node.js en TDD, on pourra utiliser Mocha :

```
it('should validate a form with all of the possible validation types', function () {  
  
    const name = form.querySelector('input[name="first-name"]');  
    const age = form.querySelector('input[name="age"]');  
  
    name.value = 'Bob';  
    age.value = '42';  
  
    const result = validateForm(form);  
    expect(result.isValid).to.be.true;  
    expect(result.errors.length).to.equal(0);  
});
```

Pour développer des applications Ruby ou Java en BDD, on pourra utiliser Cucumber :

Comment

@tag

Feature: Sale Should Result in Decrease in Inventory

When we make a sale inventory should go down

Scenario: Make a Sale Check Inventory

Given sell 3 items of ABC

When inventory on hand is 9

Then remaining inventory is 6

```
@Given("^sell 3 items of ABC$")
public void makeSale() {
    // Write code here that instantiates sale
    //function in larger order entry system
    throw new PendingException();
}
@When("^inventory on hand is 9$")
public void checkInventoryNow() {
    // put some code here
    throw new PendingException();
}
@Then("^remaining inventory is 6$")
public void checkInventoryAgain() {
    // put some code here
    throw new PendingException();
}
```

On pourra également utiliser Spock (en Groovy) pour du code Java/Groovy, qui a l'avantage de fournir un framework BDD entier (exécution des tests, rapports) et simple, entièrement intégrable dans Maven et Gradle :

```
def "events are published to all subscribers"() {  
  
    given: "an empty bank account"  
        def theAccount = Mock(BankAccount)  
  
    when: "the account is credited $10"  
        theAccount.credit(10)  
  
    then: "the account's balance is $10"  
        theAccount.balance == 10  
}
```

```
Condition not satisfied:  
assert pc.clockRate >= 2333  
      | |      |  
      | 1666   false  
      ...
```

Partie IV : Recette fonctionnelle

Qu'est-ce qu'une recette
fonctionnelle ?
Comment la réaliser ?

LA RECETTE FONCTIONNELLE

La recette fonctionnelle est l'opération par laquelle le client reconnaît que le produit livré par le fournisseur est conforme à la commande passée, qu'il est exploitable dans le SI de l'entreprise et enfin qu'il est opportun de le mettre à disposition des utilisateurs.

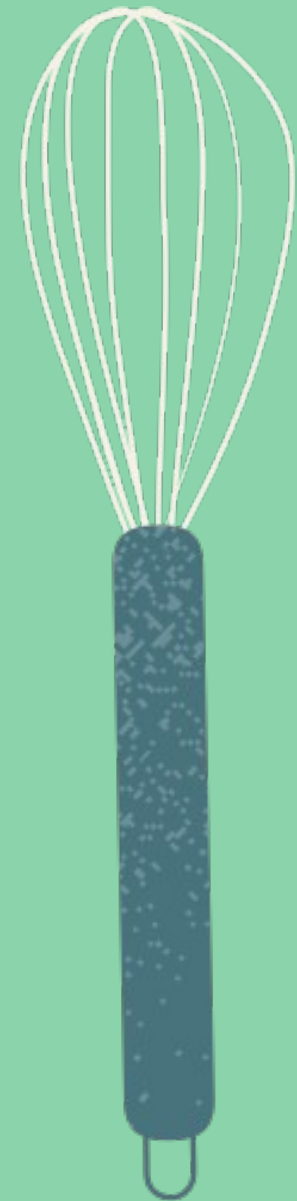


La recette se décompose en 2 grandes phases :

- La préparation du cahier de recette : planification des différentes activités sans négliger la préparation logistique
- La réalisation des tests : les bugs sont remontés et le bilan permet d'améliorer la prochaine série de tests.



La recette fonctionnelle se termine parfois par l'écriture d'un PV de recette. Dans tous les cas, la validation et le rapport de la recette sont primordiaux !



LE CAHIER DE RECETTE



Le cahier de recette fonctionnelle est rédigé lors des phases de conception et de réalisation.

💡 En phase de conception, il est initialisé par l'équipe projet sur la base des spécifications fonctionnelles et techniques rédigées ou en cours de finalisation.

Le cahier de recette contient les éléments suivants (à cette étape, il est souvent impossible de détailler entièrement les éléments, on les décrira donc succinctement) :

- Introduction (rédigée)
- Organisation (première approche et planning macroscopique à préciser par la suite)
- Scénarios de test (initialisé : parties « objet du scénario » et « prérequis »)

💡 En phase de réalisation, le cahier de recette fonctionnelle est complété : finalisation de l'organisation (planning détaillé, caractéristiques des jeux de tests), rédaction des scénarios de test complets

PROCESSUS MÉTIER

Pour faciliter l'écriture des scénarios de tests d'une recette fonctionnelle, on identifiera les processus métier du produit.

Un processus métier est un ensemble de tâches et d'actions qui, une fois exécutées, remplissent l'un des besoins du client.

La recette fonctionnelle est donc uniquement orientée vers les besoins business du client (et jamais liée aux détails techniques d'implémentation)

Celle-ci est réalisée par les utilisateurs finaux (ou leurs représentants)

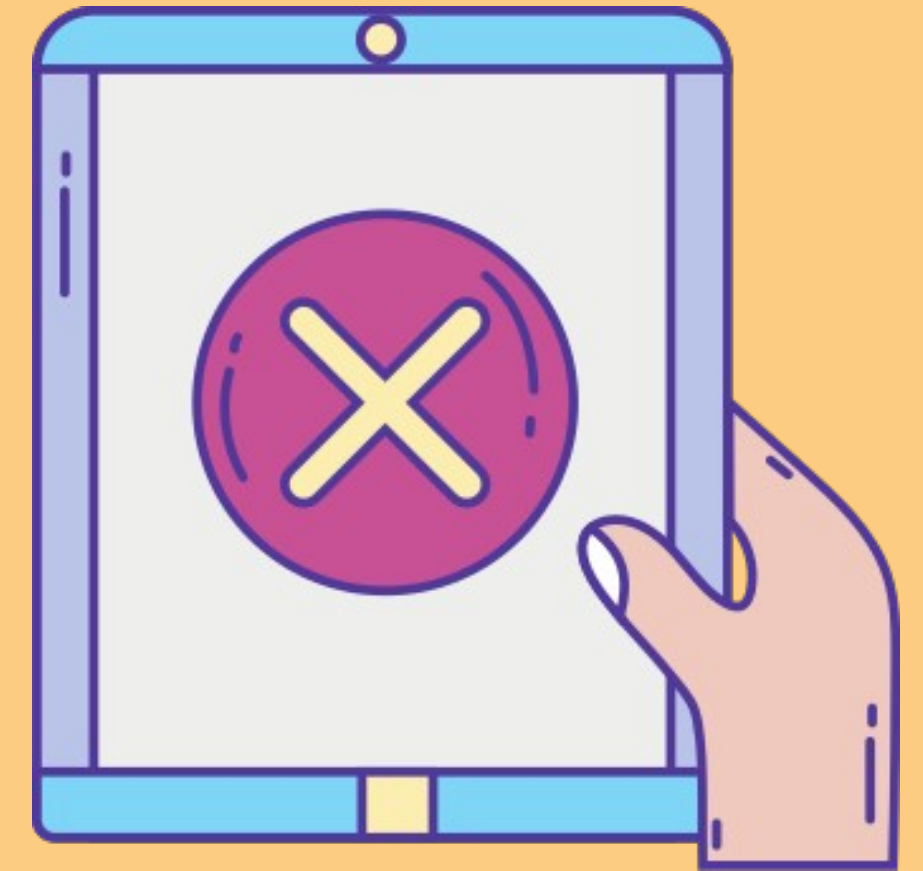


LES ANOMALIES

Le reporting des anomalies doit être clair et rattaché au cas de test.
Dire « ça ne marche pas » n'est pas suffisant : on précisera le maximum de détails relativement au contexte du test : détails de connexion, les étapes du test, les valeurs obtenues vs. celles attendues, copies d'écran et autres éléments utiles.

Le suivi des anomalies : quelque soit l'outil utilisé, le processus « end-to-end » d'un test fonctionnel doit être tracé et suivre un workflow décrit et partagé à l'avance.

Autrement dit, il s'agit de tracer une anomalie depuis sa création jusqu'à sa clôture. Ce workflow met en place des rôles et responsabilités et permet d'y assigner des collaborateurs. Le chef de projet peut ainsi piloter l'évolution de la phase de tests fonctionnels, et la fluidité permet à l'équipe technique de travailler de manière efficace.



DÉMARCHE ET ACTEURS

La réalisation d'un projet informatique est structurée en deux grand pôles : maîtrise d'ouvrage (MOA) et maîtrise d'oeuvre (MOE).

La MOA a en charge la spécification des besoins fonctionnels du client, et fournit aux équipes de développement un ensemble de cas de tests fonctionnels qui devront être réalisables sur le produit réalisé.

La MOE (maîtrise d'oeuvre) a la charge de la réalisation technique du produit.
Cette réalisation doit donc pouvoir être validée par la recette fonctionnelle.

COÛTS PRÉVUS ET RISQUES

Le cahier de recette fonctionnelle doit décrire de manière précise :

- Les coûts prévisionnels de chaque test. Ces coûts incluent : la mise en place de l'environnement, l'exécution du test, les investigations, les outils à mettre en place, les éventuels développements pour automatiser certaines étapes.
- Les risques liés à chaque test de la recette fonctionnelle. Grâce à cette information, l'ensemble des parties prenantes du projet peut anticiper la gravité d'une erreur si un test échoue. Cela permet également à l'équipe de développement de connaître les zones de criticité du projet.

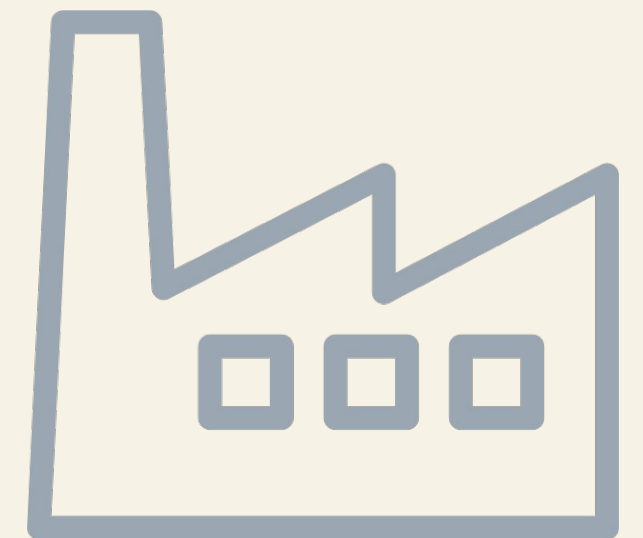


ORGANISATION, CADRAGE, PÉRIMÈTRES

L'organisation de la recette fonctionnelle est gérée de façon relativement différente en fonction du contexte du projet, des exigences du client, etc.

- En théorie, une fois la recette d'usine (réalisée par le fournisseur du service) terminée et que le produit a été jugé conforme, il est livré au client pour test, sur un environnement dédié.

Le client entame alors, généralement, une phase de recette de son côté, avec sa propre méthodologie.



ORGANISATION, CADRAGE, PÉRIMÈTRES

2. Quand le client estime que le produit a atteint un niveau de conformité suffisant, il prononce la VABF (vérification d'aptitude au bon fonctionnement), qui permet de déployer le système sur une unité pilote de production. De nouvelles anomalies y sont généralement détectées, faisant l'objet d'un « Early Life Support Plan »



3. Enfin, quand toutes les corrections nécessaires ont été apportées, le client prononce la VSR (vérification de service régulier), qui autorise la mise en exploitation globale de la solution. Le projet est alors terminé.

OUTILS

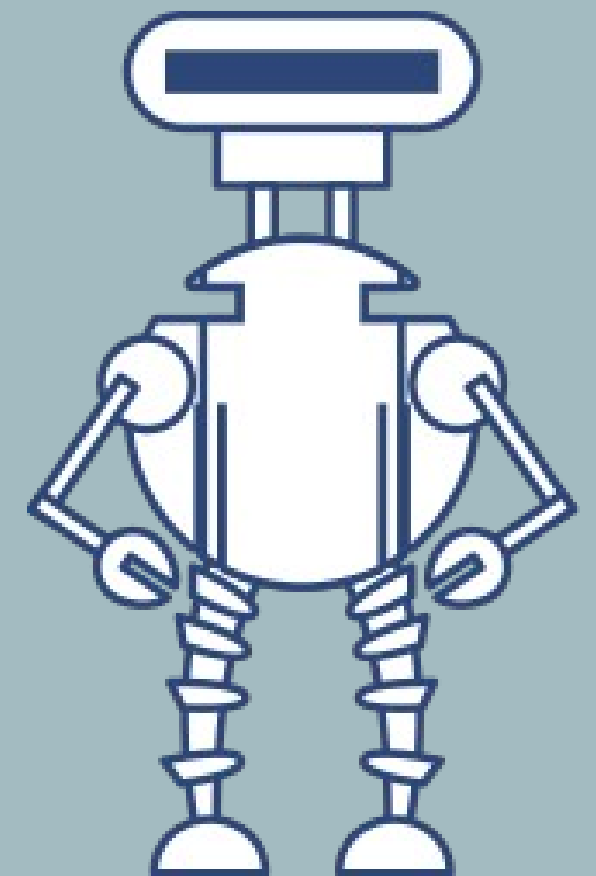
Bien qu'elle soit toujours industrialisée, la recette fonctionnelle peut être entièrement réalisée à la main.

Pour gagner en performances et faciliter la gestion des tests, il est cependant possible d'utiliser :

- des outils d'automatisation des tests, permettant de réaliser l'ordonnancement, l'exécution et la génération de rapports de tests : Sélénium, Squash TA, SilkTest, ...
- des outils de gestion des scénarios de tests et de rapports d'exécution : TestLink, ...

Les supports de présentation du cahier de recette et de rapports d'exécution sont également très variés et souvent liés à la culture de l'entreprise et à la taille du projet :

- Fichier Excel
- Wiki
- Outil interne



PRIORITÉS

Chaque test répertorié dans la recette fonctionnelle doit être priorisé de manière très précise.

Cela permet aux équipes réalisant la recette de savoir quels environnements et scénarios de test gérer en priorité, sur quels tests passer plus de temps lors d'une investigation, etc...



ENVIRONNEMENT REQUIS

Dès l'initialisation du cahier de recettes, on se posera la question du (ou des) environnement(s) requis pour l'exécution de la recette fonctionnelle.

Non seulement cette étape est primordiale pour être capable de réaliser les scénarios de test, mais aussi pour définir les ressources nécessaires à leur exécution.

La définition de ces environnements, proches de la production finale, est aussi une aide précieuse pour planifier le développement et l'architecture.

Liens

- Étude JetBrains (2020) sur le test dans les projets informatiques :
<https://www.jetbrains.com/lp/devecosystem-2020/testing/>