UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Méthode de tests unitaires

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Rappel sur les tests

Rappel sur les tests

Définition du test

Les tests en informatique, particulièrement dans le domaine du développement logiciel, sont une étape cruciale pour assurer la qualité, la performance, et la fiabilité des applications. Ils permettent de détecter les erreurs, les défauts, ou tout aspect non conforme aux exigences initiales, avant que le logiciel ne soit déployé ou mis en production.

Rappel sur les tests

Définition du test

La définition qui suit est issue de la norme IEEE 829-19982 revue à l'aide du glossaire de l'ISTQB3.

- Un test est un ensemble de cas à tester (état de l'objet à tester avant exécution du test, actions ou données en entrée, valeurs ou observations attendues, et état de l'objet après exécution), éventuellement accompagné d'une procédure d'exécution (séquence d'actions à exécuter). Il est lié à un objectif.
- La réalisation d'un test amène donc à définir cet ensemble. Différents types de test permettent de détecter différents types de défaut. Des méthodes de spécification de test ont été élaborées pour permettre une plus grande rigueur dans cette activité de définition.
- Un test vise à mettre en évidence des défauts de l'objet testé. ***Cependant, il n'a pas pour finalité de les corriger.***

Rappel sur les tests

Importance des Tests

Les tests sont essentiels pour garantir que le logiciel fonctionne comme prévu et répond aux besoins des utilisateurs.

- **a.** Ils contribuent à réduire les coûts de maintenance en identifiant et en corrigeant les défauts tôt dans le cycle de vie du développement logiciel.
- **b.** De plus, ils aident à améliorer la satisfaction des utilisateurs en fournissant un produit de qualité.
- **c.** Les tests vérifient que le logiciel répond aux exigences réglementaires et aux standards de l'industrie, ce qui est particulièrement important dans les secteurs réglementés comme la finance, la santé et l'aviation.

Rappel sur les tests

Importance des Tests

- **d.** Les tests fournissent un cadre pour évaluer l'impact des modifications apportées au logiciel, facilitant ainsi les mises à jour et les améliorations continues. Ils permettent de vérifier que les nouvelles fonctionnalités s'intègrent bien sans affecter les fonctionnalités existantes.
- **e.** Les résultats des tests fournissent des données précieuses qui peuvent aider les équipes de développement et les parties prenantes à prendre des décisions éclairées concernant la libération du logiciel, les améliorations nécessaires et la planification des ressources.
- **f.** Un logiciel de haute qualité peut offrir un avantage concurrentiel. Les tests aident à s'assurer que le produit est de la plus haute qualité possible, renforçant ainsi la réputation de l'entreprise et favorisant la fidélité des clients.

Rappel sur les tests

Les processus des Tests

Le processus de test suit généralement plusieurs étapes clés :

1 - Planification des Tests : Définition des objectifs de test, sélection des techniques de test, et allocation des ressources.
2 - Conception des Tests : Création de cas de test basés sur les exigences du logiciel.
3 - Exécution des Tests : Réalisation des tests et enregistrement des résultats.
4 - Rapport de Test : Analyse des résultats des tests, identification des défauts et documentation.
5 - Maintenance : Mise à jour des cas de test en fonction des changements dans le logiciel.

Rappel sur les tests

Les niveaux de Tests

Les niveaux de tests en développement logiciel définissent les différentes étapes à travers lesquelles un produit logiciel est testé, de ses composants individuels à son comportement en tant que système complet. Chaque niveau vise des objectifs spécifiques et utilise des techniques adaptées pour identifier et corriger les erreurs à différents stades du développement.

1 - Test Unitaire (Unit Testing)
2 - Test d'Intégration
3 - Test Système (System Testing)
4 - Test d'Acceptation (Acceptance Testing)

Rappel sur les tests

Les niveaux de Tests

1. Test Unitaire (Unit Testing)

Objectif : Tester individuellement les plus petites unités de code (fonctions ou méthodes) pour s'assurer qu'elles fonctionnent correctement.
Techniques : Les développeurs écrivent des cas de test pour chaque fonction ou méthode. Les frameworks de test unitaire (comme JUnit pour Java, NUnit pour .NET, ou PyTest pour Python) sont souvent utilisés.
Avantages : Identification rapide des erreurs à la source, facilité de maintenance, et amélioration de la qualité du code.

Rappel sur les tests

Les niveaux de Tests

2. Test d'Intégration

Objectif : Tester la combinaison de deux ou plusieurs unités de code (ou modules) et les interactions entre elles.
Techniques : Les tests peuvent être réalisés en intégrant progressivement les modules (approche incrémentale) ou en utilisant des stubs et des drivers pour simuler les modules manquants. Les outils de test d'intégration facilitent ce processus.
Avantages : Détecter les problèmes liés à l'interface entre les modules, assurer la cohérence des données à travers le système.

Rappel sur les tests

Les niveaux de Tests

3. Test Système (System Testing)

Objectif : Tester le système complet pour vérifier qu'il répond aux exigences spécifiées.
Techniques : Il s'agit d'un test de bout en bout qui couvre toutes les fonctionnalités du logiciel dans un environnement qui simule la production. Les tests de charge, de stress et de sécurité peuvent également être inclus à ce niveau.
Avantages : Validation de la conformité du logiciel aux exigences, évaluation de son comportement dans des conditions réalistes.

Rappel sur les tests

Les niveaux de Tests

4. Test d'Acceptation

Objectif : Vérifier si le logiciel est prêt à être livré en évaluant si les besoins et les attentes des utilisateurs sont satisfaits.
Techniques : Les tests d'acceptation peuvent être réalisés par les utilisateurs finaux (test d'acceptation utilisateur, UAT), le personnel interne, ou automatiquement. Ils se concentrent sur les scénarios d'utilisation réelle et les critères d'acceptation définis.
Avantages : Assure que le logiciel est fonctionnel et satisfaisant pour les utilisateurs finaux avant son déploiement final.

Rappel sur les tests

Compléments aux Niveaux de Tests

Test de Régression : Pour s'assurer que les nouvelles modifications n'affectent pas négativement les fonctionnalités existantes.
Test de Performance : Pour évaluer la vitesse, la réactivité et la stabilité du logiciel sous charge.
Test de Sécurité : Pour identifier les vulnérabilités potentielles à des attaques malveillantes.
Chaque niveau de test joue un rôle crucial dans le développement d'un logiciel fiable et de haute qualité, aidant les équipes à identifier et à résoudre les problèmes à différents stades du cycle de vie du développement logiciel.

Rappel sur les tests

Les types de Tests

Les types de tests dans le développement logiciel sont variés et couvrent différents aspects du produit, allant de la vérification de la logique interne du code à l'évaluation de l'expérience utilisateur. Chaque type de test utilise des techniques spécifiques pour identifier les défauts, mesurer la performance, et garantir que le logiciel répond aux exigences des utilisateurs.

1 - Tests Fonctionnels
2 - Tests Non Fonctionnels
3 - Tests Automatisés
4 - Test Statique

Rappel sur les tests

Les types de Tests

1. Tests Fonctionnels

Objectif : Vérifier que chaque fonction du logiciel opère selon les spécifications requises.

Techniques :

- *Test de boîte noire :* Focalisé sur les entrées et les sorties sans considération de la logique interne.
- *Test basé sur les spécifications :* Utilise les exigences et les spécifications pour créer les cas de test.

Rappel sur les tests

Les types de Tests

2. Tests Non Fonctionnels

Test de Performance : Mesure la réactivité et la stabilité sous différentes charges.

- Techniques : Test de charge, test de stress, test d'endurance.

Test de Sécurité : Identifie les vulnérabilités et les risques de sécurité.

- Techniques : Test d'intrusion, analyse statique de code, revue de code de sécurité.

Test d'Usabilité : Évalue la facilité d'utilisation et l'expérience utilisateur.

- Techniques : Tests utilisateur, groupes de discussion, analyse de parcours utilisateur.

Test de Compatibilité : Vérifie que le logiciel fonctionne dans différents environnements matériels/logiciels.

- Techniques : Tests sur différentes plateformes et versions de système d'exploitation.

Rappel sur les tests

Les types de Tests

3. Tests Automatisés

Objectif : Exécuter des suites de tests répétitivement à l'aide de logiciels spéciaux sans intervention manuelle.

Techniques :

- Frameworks de test (Selenium pour les applications web*, Appium pour les applications mobiles).
- *Intégration Continue (CI)* pour l'exécution automatique des tests lors de chaque commit dans le code source.

Rappel sur les tests

Les types de Tests

4. Test Statique

Objectif : Examiner le code source et la documentation sans exécuter le programme.

Techniques :

- *Revue de code :* Inspection manuelle du code source par d'autres développeurs.
- *Analyse statique :* Utilisation d'outils pour détecter automatiquement les défauts potentiels dans le code.

Rappel sur les tests

Les différents cycles de développement

- Les cycles de développement en informatique décrivent les différentes méthodologies utilisées pour planifier, concevoir, développer, tester et livrer des logiciels.
- Chaque cycle a ses propres caractéristiques, avantages et inconvénients, et est choisi en fonction des besoins spécifiques du projet, de l'équipe et des parties prenantes.

Rappel sur les tests

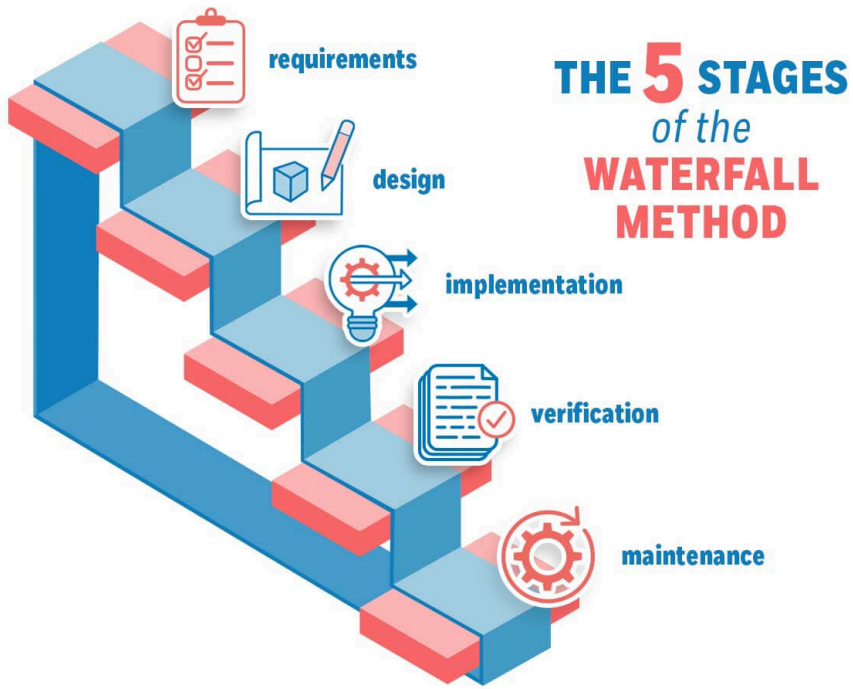
Les différents cycles de développement

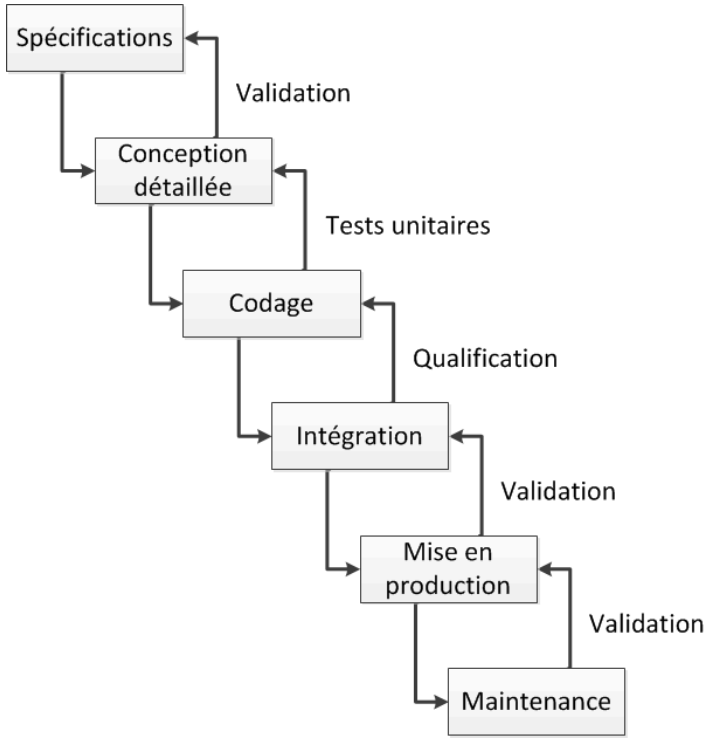
1. Modèle en Cascade (Waterfall)

Caractéristiques : Processus linéaire et séquentiel où chaque phase doit être complétée avant de passer à la suivante (conception, développement, test, déploiement).
Avantages : Facile à comprendre et à gérer, avec des objectifs clairement définis à chaque étape.
Inconvénients : Manque de flexibilité pour revenir en arrière et apporter des modifications; les problèmes ne sont souvent découverts que lors de la phase de test, ce qui peut être coûteux à corriger.

Rappel sur les tests

Les différents cycles de développement





Rappel sur les tests

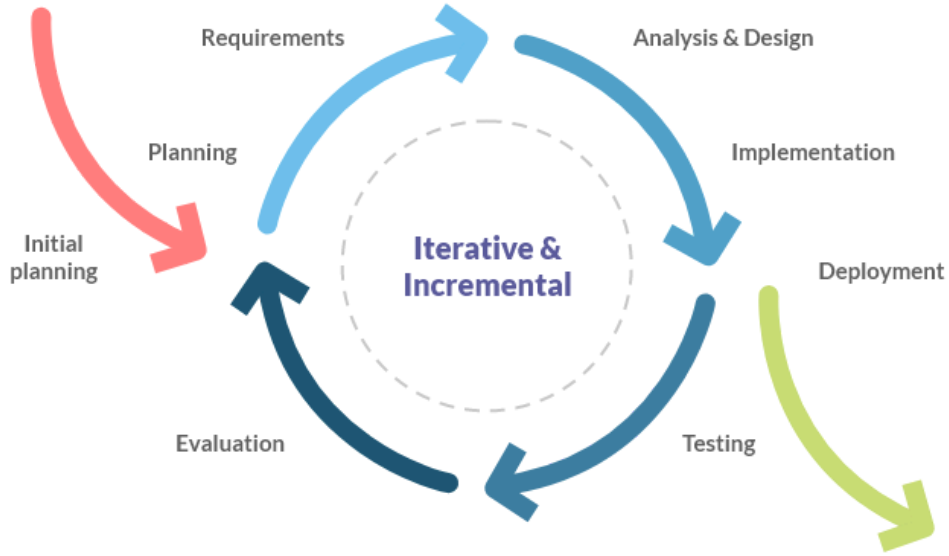
Les différents cycles de développement

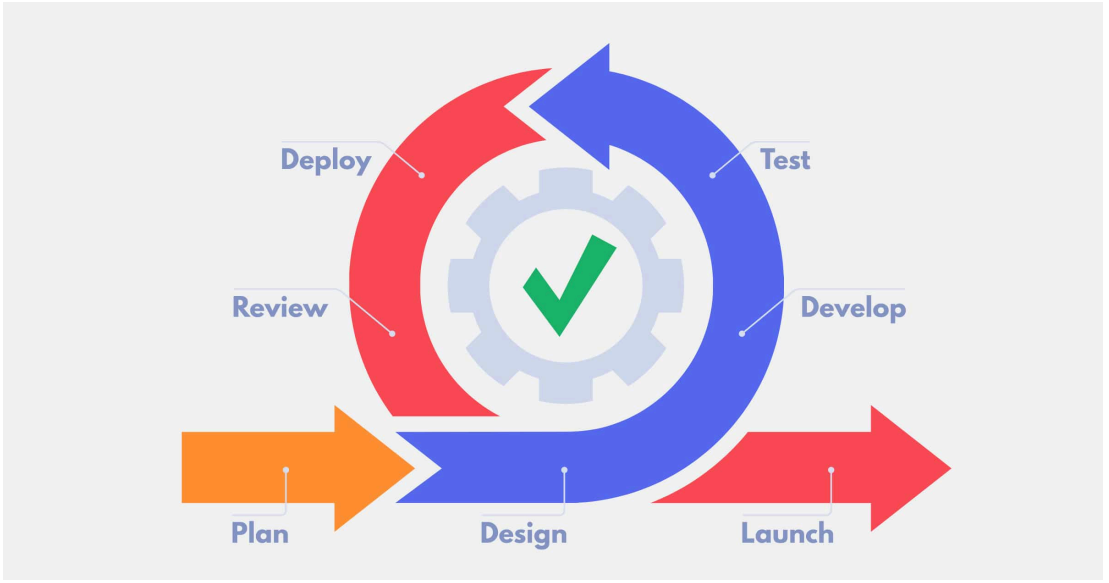
2. Développement Itératif et Incremental

Caractéristiques : Le projet est divisé en petites parties qui sont développées de manière itérative. Chaque itération passe par les phases de planification, de conception, de développement et de test.
A**vantages :** Flexibilité pour apporter des modifications et ajustements; les problèmes sont identifiés et résolus plus tôt dans le processus.
Inconvénients : Peut nécessiter plus de ressources et de temps en raison des répétitions.

Rappel sur les tests

Les différents cycles de développement





Rappel sur les tests

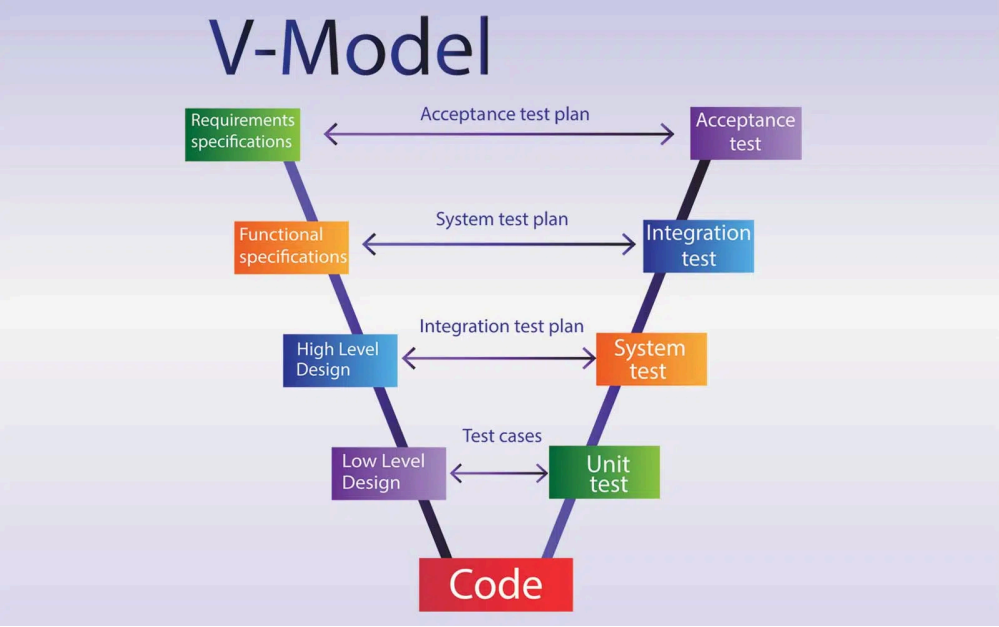
Les différents cycles de développement

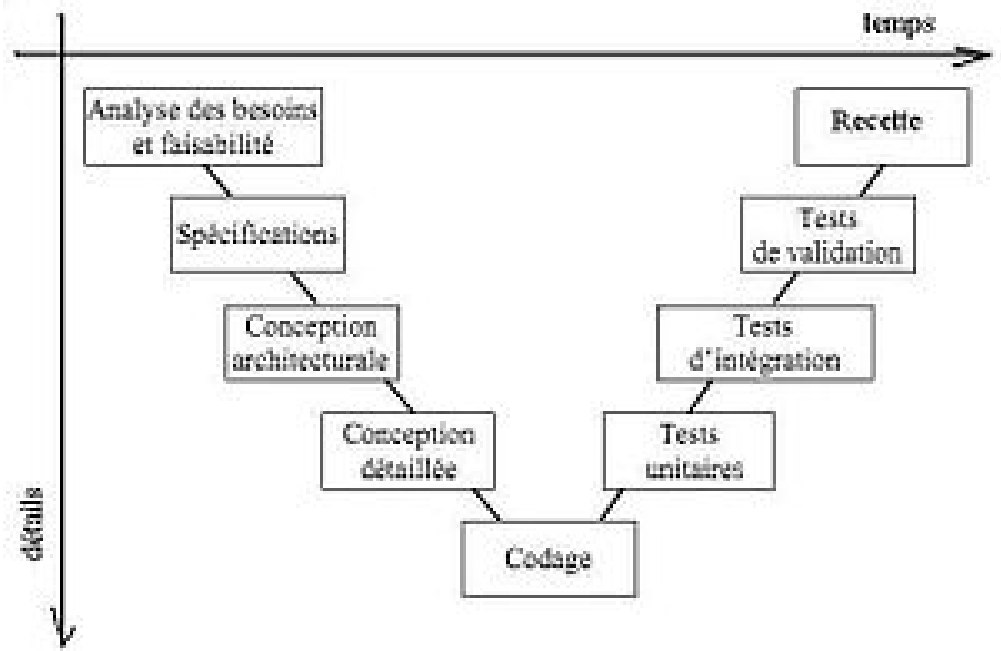
3. Modèle en V (V-Model)

Caractéristiques : Extension du modèle en cascade qui intègre les activités de test de manière plus structurée, en les associant à chaque phase de développement correspondante.
Avantages : Met l'accent sur les tests dès le début du projet; facilite la détection précoce des erreurs.
Inconvénients : Comme le modèle en cascade, il souffre d'un manque de flexibilité pour revenir en arrière une fois une phase terminée.

Rappel sur les tests

Les différents cycles de développement





Rappel sur les tests

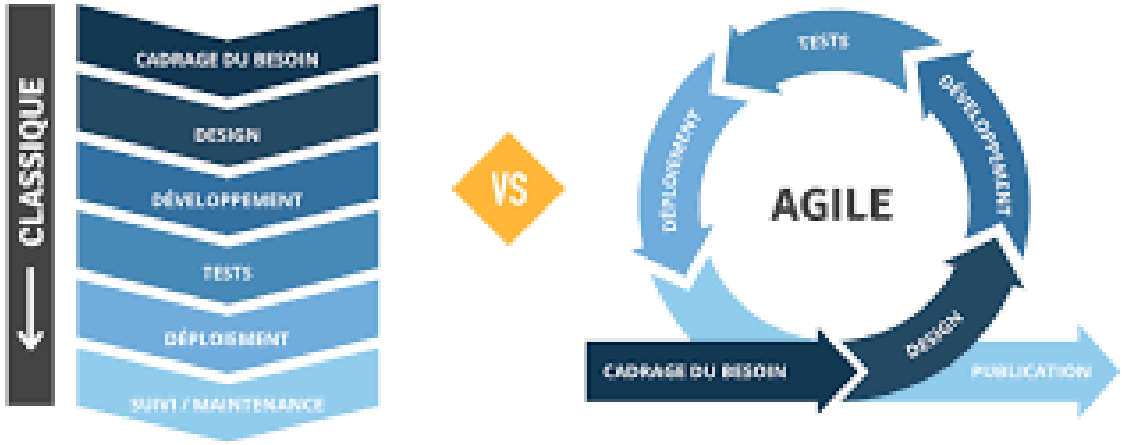
Les différents cycles de développement

4. Développement Agile

Caractéristiques : Approche flexible qui promeut le développement itératif, la collaboration étroite entre l'équipe de développement et les clients, et l'adaptabilité aux changements.
Avantages : Très adaptable aux changements; favorise une communication constante et une livraison rapide de fonctionnalités.
Inconvénients : Moins prévisible en termes de budget et de calendrier; peut être difficile à gérer si l'équipe n'est pas bien disciplinée.

Rappel sur les tests

Les différents cycles de développement



Rappel sur les tests

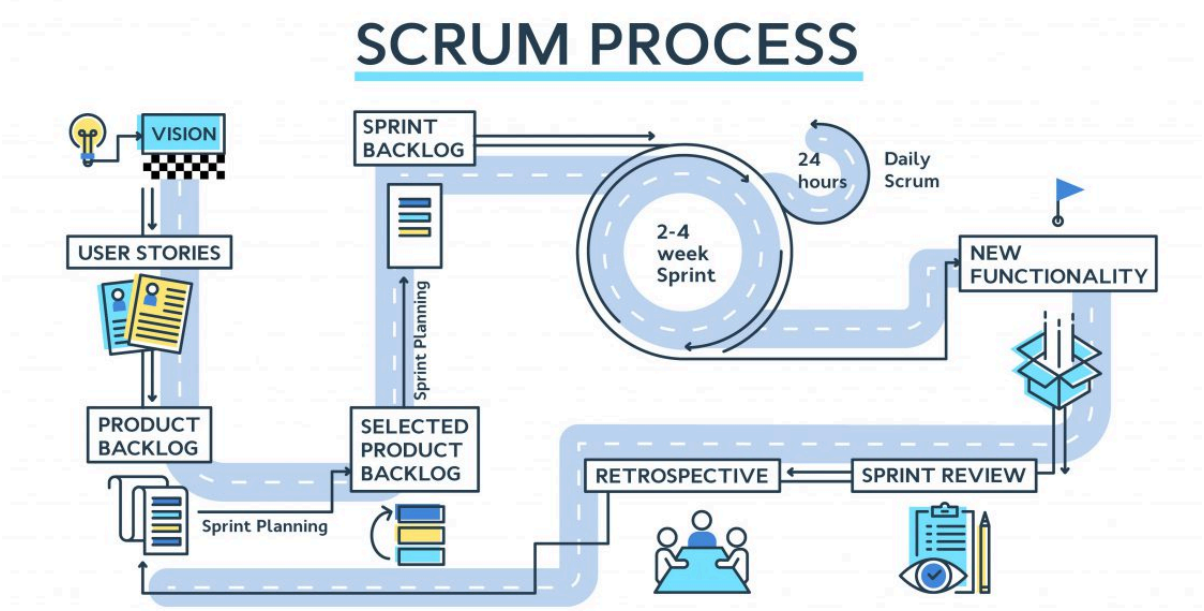
Les différents cycles de développement

5. Scrum

Caractéristiques : Cadre de travail Agile qui organise le développement en cycles appelés sprints, typiquement de 2 à 4 semaines, avec des réunions quotidiennes (daily scrum) et une revue à la fin de chaque sprint.
Avantages : Améliore la productivité et réduit le temps de mise sur le marché; favorise la responsabilité de l'équipe et l'amélioration continue.
Inconvénients : Peut être difficile à mettre en œuvre dans des organisations très structurées; nécessite un engagement fort de tous les membres de l'équipe.

Rappel sur les tests

Les différents cycles de développement



Rappel sur les tests

Les différents cycles de développement

6. Développement Lean

Caractéristiques : Se concentre sur la création de valeur pour le client en éliminant le gaspillage (processus inutiles, fonctionnalités non essentielles, etc.).
Avantages : Maximise les ressources et réduit les coûts; améliore l'efficacité en se concentrant sur ce qui est vraiment important.
Inconvénients : Peut mener à une sur-simplification; nécessite une compréhension profonde des besoins des clients.

Rappel sur les tests

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à l'automatisation

Introduction à l'automatisation

Que peut on automatiser ?

Dans le domaine du développement logiciel, l'automatisation peut couvrir un large éventail de tâches et de processus pour accroître l'efficacité, réduire les erreurs humaines et optimiser les cycles de développement.

1 - Tests Logiciels
2 - Déploiement et Intégration Continue
3 - Surveillance et Logging
4 - Sécurité
5 - Développement

Introduction à l'automatisation

Que peut on automatiser ?

1. Tests Logiciels

- **Tests Unitaires :** Automatisation des tests de petites unités de code pour vérifier leur bon fonctionnement individuellement.
- **Tests d'Intégration :** Tests automatisés pour vérifier la bonne interaction entre différentes unités de code ou modules.
- **Tests de Système :** Automatisation des tests du système complet pour s'assurer qu'il répond aux exigences spécifiées.
- **Tests de Charge et de Performance :** Utilisation d'outils spécialisés pour simuler des conditions de charge élevée ou des scénarios d'utilisation réelle.
- **Tests de Régression :** Automatisation des tests pour s'assurer que les nouvelles modifications n'introduisent pas de nouveaux bugs dans les fonctionnalités existantes.

Introduction à l'automatisation

Que peut on automatiser ?

2. Déploiement et Intégration Continue

Intégration Continue (CI) : Automatisation de la compilation, des tests et de l'intégration de code dans le repository principal plusieurs fois par jour.
Déploiement Continu (CD) : Automatisation du déploiement de versions de logiciels dans l'environnement de production ou d'autres environnements de test.
Gestion de Configuration : Automatisation de la configuration des environnements de serveurs, y compris l'installation et la mise à jour de logiciels.

Introduction à l'automatisation

Que peut on automatiser ?

3. Surveillance et Logging

Surveillance de Performance : Automatisation de la collecte de données sur la performance des applications et de l'infrastructure.
Logging Automatisé : Configuration des systèmes pour générer automatiquement des logs d'activité et d'erreur.

Introduction à l'automatisation

Que peut on automatiser ?

4. Sécurité

Scans de Vulnérabilité : Utilisation d'outils automatisés pour identifier les vulnérabilités de sécurité dans le code source ou dans les applications déployées.
Tests de Pénétration Automatisés : Automatisation des tests d'intrusion pour évaluer la résilience du système face aux attaques malveillantes.

Introduction à l'automatisation

Que peut on automatiser ?

5. Développement

Génération de Code : Utilisation d'outils pour générer automatiquement du code à partir de modèles ou de spécifications.
Refactoring de Code : Outils automatisés pour aider à restructurer et à optimiser le code sans en changer le comportement fonctionnel.

Introduction à l'automatisation

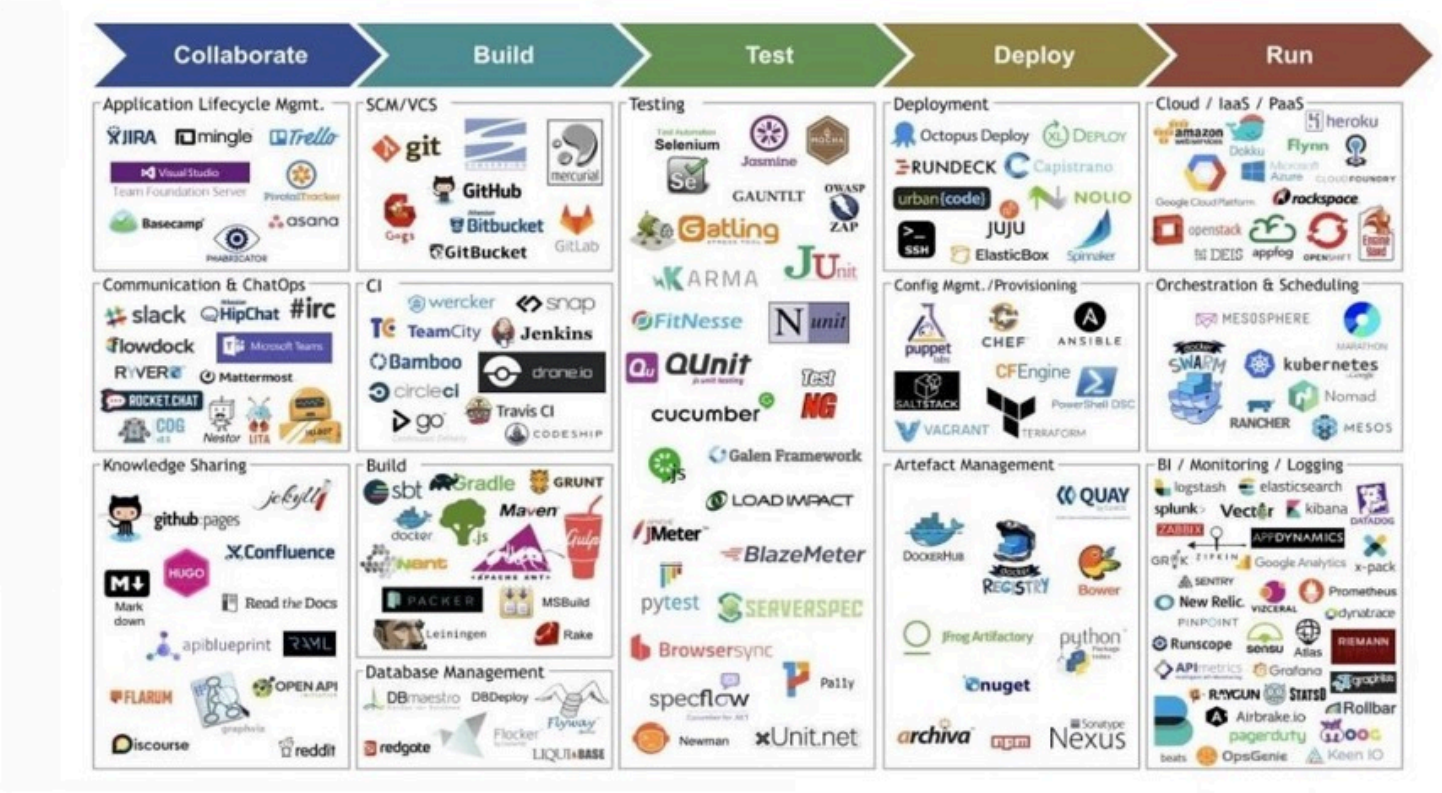
Que peut on automatiser ?

Exemples d'Outils d'Automatisation:

- **Selenium**, **Cypress** pour les tests d'interface utilisateur web.
- **Jenkins**, **CircleCI** pour l'intégration et le déploiement continus.
- **Ansible**, **Chef**, **Puppet** pour la gestion de configuration et l'automatisation des déploiements.
- **SonarQube** pour l'analyse statique de code et l'identification des vulnérabilités.
- **Prometheus**, **Grafana** pour la surveillance et la visualisation des performances.

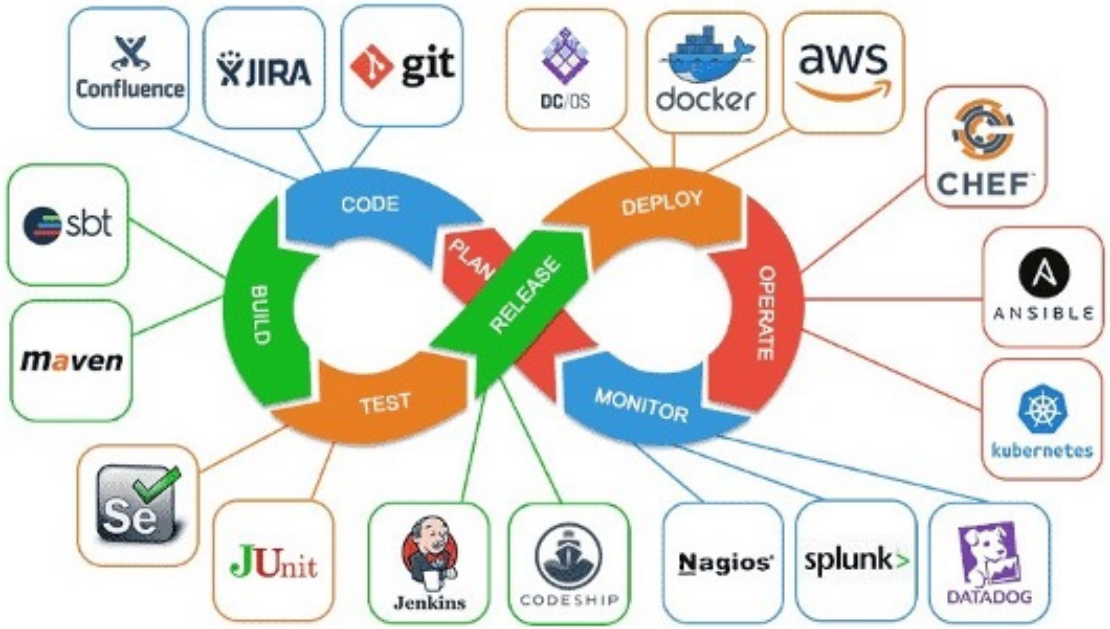
Introduction à l'automatisation

Que peut on automatiser ?



Rappel sur les tests

Que peut on automatiser ?



Introduction à l'automatisation

Pourquoi automatiser ?

1. Efficacité accrue : L'automatisation permet d'exécuter des centaines ou des milliers de tests en un temps relativement court, une tâche qui serait autrement fastidieuse et sujette à erreur si effectuée manuellement.
2. Réduction des coûts : Bien que l'investissement initial dans l'automatisation des tests puisse être élevé, il permet une réduction significative des coûts à long terme en diminuant le temps nécessaire aux cycles de test et en identifiant les bugs plus tôt dans le cycle de vie du développement.
3. Amélioration de la qualité : L'automatisation permet d'exécuter des tests de manière plus fiable et avec une précision constante, contribuant ainsi à améliorer la qualité globale du logiciel.

Introduction à l'automatisation

Pourquoi automatiser ?

4. Tests répétitifs : Elle est particulièrement utile pour exécuter les mêmes ensembles de tests dans le cadre de tests de régression, assurant que les nouvelles modifications n'affectent pas les fonctionnalités existantes négativement.

5. Tests dans des conditions variées : Elle permet de tester le logiciel dans divers environnements et configurations, ce qui serait difficile et chronophage manuellement.

Introduction à l'automatisation

Dans quel types de cycle ?

L'automatisation des tests est applicable et bénéfique dans pratiquement tous les types de cycles de développement logiciel, y compris mais sans s'y limiter.

- **Agile :** Elle est particulièrement adaptée à l'approche Agile, où la rapidité de livraison et les cycles de développement courts nécessitent des tests fréquents et rapides.
- **DevOps :** Dans le cadre de DevOps, l'automatisation des tests est essentielle pour permettre l'intégration continue (CI) et le déploiement continu (CD), facilitant des cycles de release rapides et fiables.
- **Itératif et incrémental :** Elle aide à valider chaque phase et chaque fonctionnalité ajoutée, garantissant que chaque itération du produit fonctionne comme prévu.

Introduction à l'automatisation

Exemples d'Automatisation des Tests ?

Selenium : Un cadre d'automatisation pour les applications web qui permet d'écrire des scripts de test dans plusieurs langages de programmation. Selenium est idéal pour tester les interactions utilisateurs sur les navigateurs web.
JUnit/TestNG pour les tests unitaires : Ces frameworks sont utilisés pour automatiser les tests unitaires en Java, permettant aux développeurs de tester des parties spécifiques du code automatiquement.
Appium : Utilisé pour l'automatisation des tests d'applications mobiles, Appium prend en charge les applications iOS, Android et Windows.
Jenkins pour l'intégration continue : Jenkins peut être configuré pour exécuter automatiquement des suites de tests à chaque commit de code, permettant une détection précoce des problèmes.
Cypress : Un autre outil d'automatisation des tests modernes pour les applications web, connu pour sa facilité d'utilisation et son intégration dans les workflows de développement modernes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les tests unitaires

Les tests unitaires

Environnement de test unitaire

- Un **environnement de test unitaire** est un cadre dans lequel les développeurs peuvent évaluer le comportement de petites parties de code de manière isolée, pour s'assurer qu'elles fonctionnent comme prévu.
- Les **tests unitaires** jouent un rôle crucial dans le développement logiciel, car ils aident à identifier les bugs tôt dans le cycle de développement, facilitent les modifications du code en assurant que les changements n'introduisent pas de nouveaux bugs, et améliorent la qualité du code.

Les tests unitaires

Environnement de test unitaire

1 - Framework
2 - Isolation
3 - Automatisation
4 - Couverture de code
5 - Pratique de rédaction de tests

Les tests unitaires

Environnement de test unitaire

1. Cadre de test (Framework)

- Les cadres de test fournissent une structure et des outils pour créer, exécuter et évaluer les tests unitaires.
- Ils offrent généralement des assertions pour tester les conditions attendues et des mécanismes pour simuler les dépendances.
- Des exemples populaires incluent JUnit pour Java, pytest pour Python, et NUnit pour .NET.

Les tests unitaires

Environnement de test unitaire

2. Isolation

- L'isolation est essentielle dans les tests unitaires pour s'assurer que chaque test est indépendant des autres.
- Cela signifie tester une seule "unité" à la fois (par exemple, une fonction ou une méthode) sans dépendance externe (comme des bases de données ou des services web), en utilisant des simulations (mocks) ou des objets factices (stubs).

Les tests unitaires

Environnement de test unitaire

3. Automatisation

Les tests unitaires sont généralement automatisés et exécutés fréquemment, intégrés dans le processus de développement continu (Continuous Integration, CI) pour détecter les problèmes dès qu'ils apparaissent.

Les tests unitaires

Environnement de test unitaire

4. Couverture de code

La couverture de code est une mesure qui indique quel pourcentage du code source est testé par les tests unitaires. Bien qu'une couverture de 100 % ne garantisse pas l'absence de bugs, elle peut donner une indication de la qualité des tests.

Les tests unitaires

Environnement de test unitaire

5. Pratiques de rédaction de tests

Il existe des pratiques recommandées pour rédiger des tests unitaires efficaces, telles que :

- Nommer les tests de manière significative pour indiquer clairement ce qu'ils vérifient.
- Garder les tests petits et concentrés sur une seule fonctionnalité ou comportement.
- Utiliser des données de test représentatives pour s'assurer que les tests sont réalistes.
- Tester les cas limites et les entrées invalides, pas seulement les scénarios "heureux".

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les tests unitaires avec Junit

Les tests unitaires avec Junit

C'est quoi ?

Junit

- JUnit est un framework mature pour permettre l'écriture et l'exécution de tests automatisés.
- JUnit 4 a été publié en 2005 pour permettre la prise en compte des annotations de Java 5.
- JUnit 5, publié en 2017, utilise des fonctionnalités de Java 8 notamment les lambdas, les annotations répétées, ...

Les tests unitaires avec Junit

C'est quoi ?

Junit5

- **JUnit 5** est une réécriture intégrale du framework ayant plusieurs objectifs :
  - Le support et l'utilisation des nouvelles fonctionnalités de Java 8 : par exemple, les lambdas peuvent être utilisés dans les assertions
  - Une nouvelle architecture reposant sur plusieurs modules
  - Le support de différents types de tests
    un mécanisme d'extension qui permet l'ouverture vers des outils tiers ou des API

Les tests unitaires avec Junit

C'est quoi ?

Junit5

- **JUnit 5** apporte cependant aussi son lot de nouvelles fonctionnalités :
  - Les tests imbriqués
  - Les tests dynamiques
  - Les tests paramétrés qui offrent différentes sources de données
  - Un nouveau modèle d'extension
  - L'injection d'instances en paramètres des méthodes de tests

Les tests unitaires avec Junit

C'est quoi ?

Junit5

Contrairement aux versions précédentes livrées en un seul jar, JUnit 5 est livré sous la forme de différents modules notamment pour répondre à la nouvelle architecture qui sépare :

- L'API
- Le moteur d'exécution
- L'exécution et intégration

JUnit 5 ne peut être utilisée qu'avec une version supérieure ou égale à 8 de Java : il n'est pas possible d'utiliser une version antérieure.

Les tests unitaires avec Junit

C'est quoi ?

Junit5

Contrairement aux versions précédentes livrées en un seul jar, JUnit 5 est livré sous la forme de différents modules notamment pour répondre à la nouvelle architecture qui sépare :

- L'API
- Le moteur d'exécution
- L'exécution et intégration

JUnit 5 ne peut être utilisée qu'avec une version supérieure ou égale à 8 de Java : il n'est pas possible d'utiliser une version antérieure.

Les tests unitaires avec Junit

C'est quoi ?

JUnit 5 est structuré en trois sous-projets principaux :

JUnit Platform : Sert de base pour lancer les frameworks de test sur la JVM. Il définit l'API TestEngine pour le développement de nouveaux frameworks de test qui peuvent être exécutés sur la plateforme. Il fournit également le support pour lancer les tests via la ligne de commande, des builders, ou des IDE.
JUnit Jupiter : C'est là que se trouvent toutes les nouvelles annotations (@Test, @BeforeEach, @AfterEach, etc.), les extensions et le modèle de programmation TestEngine pour l'écriture de tests et d'extensions en JUnit 5.
JUnit Vintage : Permet de faire tourner les tests JUnit 3 et JUnit 4 sur la plateforme JUnit 5, facilitant ainsi la migration progressive vers JUnit 5 sans avoir besoin de réécrire tous les anciens tests.

Les tests unitaires avec Junit

C'est quoi ?

JUnit 5

Group ID	Version	Artefact ID
org.unit.jupiter	5.0.0	junit-jupiter-api API pour l'écriture des tests avec JUnit Jupiter junit-jupiter-engine Implémentation du moteur d'exécution des tests JUnit Jupiter junit-jupiter-params Support des tests paramétrés avec JUnit Jupiter.
org.junit.platform	1.0.0	junit-platform-commons Utilitaires à usage interne de JUnit junit-platform-console Support pour la découverte et l'exécution des tests JUnit dans la console junit-platform-console-standalone Jar exécutable qui contient toutes les dépendances pour exécuter les tests dans une console junit-platform-engine API publique pour les moteurs d'exécution des tests junit-platform-gradle-plugin Support pour la découverte et l'exécution des tests JUnit avec Gradle junit-platform-launcher Support pour la découverte et l'exécution des tests JUnit avec des IDE et des outils de build junit-platform-runner Implémentation d'un Runner pour exécuter des tests JUnit 5 dans un environnement JUnit 4 junit-platform-suite-api Support pour l'exécution des suites de tests junit-platform-surefire-provider Support pour la découverte et l'exécution des tests JUnit avec le plugin Surefire de Maven
org.junit.vintage	4.12.0	junit-vintage-engine Implémentation d'un moteur d'exécution des tests écrits avec JUnit 3 et 4 dans la plateforme JUnit 5

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test)

- Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.
- Rédiger des tests fait partie de notre savoir-faire.
- Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test)

Imaginons que nous construisons une application de boutique en ligne. Les utilisateurs peuvent rechercher des produits, les sélectionner, collecter les produits dans un panier et enfin les acheter.

Dans le cadre de l'application, nous avons un cas d'utilisation avec les spécifications suivantes:

- Étant donné un panier contenant des produits d'un total de 50 000 => **Given**
- Lorsque le total est calculé => **When**
- Ensuite, il renvoie 50 en tant que total => **Then**

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test)

Le nom d'un test doit révéler le cas de test exact, y compris le système testé.

- Il doit spécifier l'exigence du cas de test aussi précisément que possible.
- L'objectif principal d'un bon nom de test est que si un test échoue, nous devrions être en mesure de récupérer la fonctionnalité cassée à partir du nom du test.
- Deux conventions de dénomination populaires :
  - **GivenWhenThen** (ex :GivenUserIsNotLoggedIn_whenUserLogsIn_thenUserIsLoggedInSuccessfully)
  - **ShouldWhen** (ex : ShouldHaveUserLoggedIn_whenUserLogsIn)

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test)

Le modèle **Arrange-Act-Assert** est une manière descriptive et révélatrice d'intention de structurer des cas de test. Il prescrit un ordre des opérations:

- La section **Arrange** doit contenir la logique de configuration des tests. Ici, les objets sont initialisés et préparés pour l'exécution des tests.
- La section **Act** invoque le système que nous sommes sur le point de tester. Il peut s'agir par exemple d'appeler une fonction, d'appeler une API REST ou d'interagir avec certains composants.
- La section **Assert** vérifie que l'action du test se comporte comme prévu. Par exemple, nous vérifions ici la valeur de retour d'une méthode, l'état final du test , les méthodes que le test a appelées, ou les éventuelles exceptions attendues et les résultats d'erreur.

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

L'acronyme **F.I.R.S.T** décrit cinq qualités essentielles que tout bon test unitaire devrait posséder pour être efficace et fiable.
Ces caractéristiques contribuent à la création d'une suite de tests maintenable, compréhensible, et utile pour assurer la qualité du logiciel tout au long de son développement.

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

1. **Fast (Rapide)**
   Les tests doivent s'exécuter rapidement.
   Si les tests sont lents, ils ne seront pas exécutés aussi souvent que nécessaire, ce qui peut retarder la découverte de bugs et ralentir le processus de développement.
   Des tests rapides encouragent les développeurs à les exécuter fréquemment, favorisant ainsi une boucle de feedback rapide et efficace.

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

2. **Independent (Indépendant)**
   Chaque test doit être indépendant des autres.
   Cela signifie que l'exécution d'un test ne doit pas dépendre du résultat ou de l'état laissé par un autre test.
   Les tests indépendants peuvent être exécutés dans n'importe quel ordre et leur résultat doit rester le même. L'indépendance des tests assure que les bugs identifiés sont faciles à localiser et que les tests ne deviennent pas flaky (c'est-à-dire qu'ils ne réussissent ou ne échouent pas de manière intermittente sans raison apparente).

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

3. **Repeatable (Répétable)**
   Les tests doivent produire le même résultat à chaque exécution, quelle que soit l'environnement dans lequel ils sont exécutés.
   Cela signifie que les tests doivent être conçus pour éviter toute dépendance à des éléments externes qui pourraient changer (comme des bases de données externes, des fichiers de configuration, ou l'heure système). Les tests répétables renforcent la fiabilité des résultats des tests et aident à identifier les changements de comportement du code testé.

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

4. **Self-validating (Auto-validant)**
   Les tests doivent être auto-validants, c'est-à-dire qu'ils doivent clairement indiquer s'ils ont réussi ou échoué sans nécessiter d'interprétation manuelle des résultats.
   Un test auto-validant retourne un état de réussite ou d'échec, permettant une automatisation complète de l'exécution des tests et de l'interprétation des résultats.

Les tests unitaires avec Junit

L'ecriture des tests (Clean Test - F.I.R.S.T)

5. **Thorough (Complet)**
   Les tests doivent couvrir suffisamment de cas pour avoir confiance dans la qualité du code testé.
   Cela inclut les cas d'utilisation typiques, les cas limites, et les chemins d'erreur. Un ensemble de tests complet aide à assurer que le système se comporte correctement dans diverses conditions et réduit le risque de bugs non détectés.

Les tests unitaires avec Junit

L'ecriture des tests

- L'écriture de classes de tests avec JUnit 5 est similaire à celle avec JUnit 4 : il faut définir des méthodes annotées avec des annotations de JUnit5.
- Certaines de ces annotations ont été renommées, notamment celles relatives au cycle de vie des instances de tests et d'autres ont été ajoutées.
- Parmi celles-ci, JUnit 5 propose une annotation permet de définir un nom d'affichage pour un cas de test, une autre permet de définir un test imbriqué sous la classe d'une classe interne.

Les tests unitaires avec Junit

L'ecriture des tests

Les annotations :

JUnit Jupiter propose plusieurs annotations pour la définition et la configuration des tests. Ces annotations sont dans le package org.junit.jupiter.api.

Les tests unitaires avec Junit

L'ecriture des tests

Annotation	Rôle
@Test	La méthode annotée est un cas de test. Contrairement à l'annotation @Test de JUnit, celle-ci ne possède aucun attribut
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configurer le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définir un libellé pour la classe ou la méthode de test annotée
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @Before de JUnit 4
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @After de JUnit 4

Les tests unitaires avec Junit

L'ecriture des tests

@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @BeforeClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory. Cette annotation est équivalente à @AfterClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class.
@Nested	Indiquer que la classe annotée correspond à un test imbriqué
@Tag	Définir une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés. Cette annotation est équivalente aux Categories de JUnit 4 ou aux groups de TestNG
@Disabled	Désactiver les tests de la classe ou la méthode annotée. Cette annotation est similaire à @Ignore de JUnit 4
@ExtendWith	Enregistrer une extension

Les tests unitaires avec Junit

L'ecriture des tests

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("Ma classe de test JUnit5")
public class MaClassTest {

  @Test
  @DisplayName("Premier test")
  void premierTest() {
    // ...
  }
}
```

Les tests unitaires avec Junit

Le cycle de vie des tests

JUnit 5 introduit un modèle de cycle de vie des tests flexible et extensible. Comprendre ce cycle de vie est crucial pour écrire des tests efficaces et bien structurés. Ci-après, les principales étapes du cycle de vie d'un test dans JUnit 5.

Les tests unitaires avec Junit

Le cycle de vie des tests

1. **Initialisation (@BeforeAll et @AfterAll)**

- ***@BeforeAll :*** Méthode exécutée une seule fois avant tous les tests de la classe actuelle. Utilisée pour une configuration coûteuse en temps (comme la connexion à une base de données). Cette méthode doit être static par défaut.
- ***@AfterAll :*** Méthode exécutée une fois après tous les tests de la classe. Idéale pour nettoyer les ressources utilisées dans les méthodes @BeforeAll. Cette méthode doit aussi être static par défaut.

Les tests unitaires avec Junit

Le cycle de vie des tests

2. **Configuration (@BeforeEach et @AfterEach)**

- ***@BeforeEach :*** Méthode exécutée avant chaque test. Utilisée pour préparer l'environnement de test (comme initialiser des objets nécessaires pour les tests).
- ***@AfterEach :*** Méthode exécutée après chaque test. Parfaite pour nettoyer ou réinitialiser l'état après l'exécution d'un test.

Les tests unitaires avec Junit

Le cycle de vie des tests

3. **Exécution des Tests (@Test)**

- **@Test :** Indique que la méthode est un test unitaire. JUnit exécutera toutes les méthodes annotées par @Test.

Les tests unitaires avec Junit

Les Assertions

- Les **assertions** ont pour rôle de faire des vérifications pour le test en cours. Si ces vérifications échouent, alors l'assertion lève une exception qui fait échouer le test.
- **JUnit Jupiter** contient la plupart des assertions de JUnit 4 mais propose aussi ses propres annotations dont certaines surcharges peuvent utiliser les lambdas.
- Les **assertions** classiques permettent de faire des vérifications sur une instance ou une valeur ou effectuer des comparaisons.
- La classe org.junit.jupiter.Assertions contient de nombreuses méthodes statiques qui permettent d'effectuer différentes vérifications de données.
- Ces **assertions** permettent de comparer les données obtenues avec celles attendues dans un cas de test.

Les tests unitaires avec Junit

Les Assertions

Egalité	Nullité	Exceptions
assertEquals()	assertNull()	assertThrows()
assertNotEquals()	assertNotNull()	
assertTrue()		
assertFalse()		
assertSame()		
assertNotSame()		

Les tests unitaires avec Junit

Les Assertions

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatriceTest {

    @Test
    public void testAdditionner() {
        // Arrange
        Calculatrice calculatrice = new Calculatrice();
        int nombre1 = 5;
        int nombre2 = 3;
        int resultatAttendu = 8;

        // Act
        int resultatObtenu = calculatrice.additionner(nombre1, nombre2);

        // Assert
        assertEquals(resultatAttendu, resultatObtenu, "5 + 3 doit être égal à 8");
    }
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Mocks avec Mockito

Les Mocks avec Mockito

C'est quoi ?

Qu'est-ce qu'un Mock?

- Un **mock** est une imitation d'une partie du système (comme une classe ou une interface) qui simule son comportement de manière contrôlée dans un environnement de test.
- En d'autres termes, un **mock** est un objet qui se substitue à un vrai composant dans votre système pendant le test.
- Cette technique est particulièrement utile lorsque le composant réel est difficile à incorporer dans des tests (par exemple, une base de données ou un service externe).

Les Mocks avec Mockito

C'est quoi ?

Pourquoi Utiliser des Mocks?

- **Isolation :** Les mocks permettent de tester une partie du système en isolation, sans dépendre des autres parties du système qui peuvent être instables, coûteuses à utiliser, ou difficiles à configurer pour les tests.
- **Contrôle :** Ils offrent un contrôle complet sur le comportement des dépendances pendant les tests. Par exemple, vous pouvez simuler des conditions d'erreur ou des cas limites qui seraient difficiles à reproduire avec des composants réels.
- **Simplicité :** Ils simplifient la configuration des préconditions de test en éliminant la nécessité de configurer un environnement complexe.
- **Efficacité :** Les tests avec mocks sont souvent plus rapides que ceux utilisant des environnements réels ou intégrés, car ils ne nécessitent pas d'accéder à des ressources externes comme des bases de données ou des réseaux.

Les Mocks avec Mockito

C'est quoi ?

Comment Fonctionnent les Mocks?

- Les **mocks** fonctionnent en remplaçant les vrais objets par des versions simulées (mocks) qui répondent aux appels de méthodes selon les besoins du test.
- Par exemple, si votre classe dépend d'un service de messagerie, vous pouvez mocker ce service pour renvoyer des réponses prédéterminées sans envoyer de vrais emails pendant les tests.

Les Mocks avec Mockito

C'est quoi Mockito ?

Présentation de Mockito

- **Mockito** est une bibliothèque populaire de mocking pour Java utilisée pour écrire des tests unitaires efficaces pour des classes avec des dépendances externes.
- **Mockito** permet aux développeurs de créer et de gérer des mocks ou des objets simulés qui imitent le comportement des dépendances réelles d'une classe.

Les Mocks avec Mockito

C'est quoi Mockito ?

Caractéristiques Principales de Mockito

- **Simplicité et Clarté :** Mockito est conçu pour être simple à utiliser et ses API sont très intuitives. Les tests écrits avec Mockito sont généralement faciles à lire et à comprendre.
- **Création de Mocks :** Mockito permet de mocker des interfaces ainsi que des classes non-finale. Cela aide à tester des classes qui dépendent de ces interfaces ou classes sans avoir besoin de leurs implémentations concrètes.
- **Stubbing Flexible :** Les développeurs peuvent préconfigurer les mocks pour qu'ils répondent de manière prévisible lorsqu'ils sont invoqués dans le test. Cela inclut la configuration des retours de méthodes, le lancement d'exceptions, et plus encore.

Les Mocks avec Mockito

C'est quoi Mockito ?

Caractéristiques Principales de Mockito

- **Vérification d'Interactions :** Mockito permet de vérifier si certaines méthodes sur un mock ont été appelées, avec des arguments spécifiques ou un nombre déterminé de fois. Cette fonctionnalité est utile pour s'assurer que le code testé interagit correctement avec ses dépendances.
- **Argument Captors :** Mockito offre la possibilité de capturer les arguments passés aux méthodes mockées pour les vérifier ultérieurement, ce qui est utile pour les tests plus complexes où les interactions avec les mocks doivent être examinées en détail.
- **Spies :** Les **spies** sont des versions partiellement mockées des objets réels. Contrairement aux mocks standards, les spies appellent les méthodes réelles à moins qu'elles n'aient été explicitement stubbées. Cela est utile pour ajouter un comportement de mocking à des objets qui fonctionnent principalement avec leur comportement normal.

Les Mocks avec Mockito

C'est quoi Mockito ?

Utilisation Typique de Mockito

Dans le contexte des tests unitaires, Mockito est utilisé pour :

1. **Isoler** le code à tester de ses dépendances, rendant les tests plus rapides et plus stables.
2. **Simuler** des comportements complexes de dépendances que le code à tester s'attend à manipuler, y compris les erreurs et les cas limites.
3. **Vérifier** que le code interagit correctement avec ses dépendances, comme s'assurer qu'une méthode est appelée avec les bons paramètres.
4. **Écrire** des tests qui sont facilement reproductibles et indépendants de l'environnement externe ou de la configuration système.

Les Mocks avec Mockito

C'est quoi Mockito ?

Création et Utilisation des Mocks avec Mockito

- Création d'un Mock :

```
List<String> mockedList = mock(List.class);
```

- Configuration du Mock (Stubbing) :

```
when(mockedList.get(0)).thenReturn("first element");
```

- Utilisation du Mock dans le test:

```
assertEquals("first element", mockedList.get(0));
```

Vérification que le Mock a été utilisé comme prévu :

```
verify(mockedList).get(0);
```

Les Mocks avec Mockito

C'est quoi Mockito ?

Bonnes Pratiques avec les Mocks :

- **Utilisez des mocks uniquement pour les dépendances externes ou complexes**. Ne les utilisez pas pour tout, car cela peut rendre les tests moins compréhensibles et plus difficiles à maintenir.
- **Ne testez pas le code du mock lui-même**. Concentrez-vous sur la façon dont votre propre code interagit avec les dépendances.
- **Gardez les interactions avec les mocks aussi simples que possible**. Cela aide à maintenir des tests clairs et faciles à comprendre.

Les Mocks avec Mockito

C'est quoi Mockito ?

Bonnes Pratiques avec les Mocks :

```java
import static org.mockito.Mockito.*;

public class UserManagerTest {
    @Test
    public void shouldCreateUser() {
        // Création du mock
        UserRepository mockRepository = mock(UserRepository.class);
        // Configuration du stub
        when(mockRepository.save(any(User.class))).thenReturn(true);
        // Création de l'instance à tester
        UserManager manager = new UserManager(mockRepository);
        // Action
        User user = new User("username", "password");
        boolean saved = manager.createUser(user);
        // Vérification
        assertTrue(saved);
        verify(mockRepository).save(user);
    }
}
```