ADRAR PÔLE NUMERIQUE
INFRASTRUCTURES SYSTÈMES & RÉSEAUX
CYBERSÉCURITÉ INFRA. & APPLICATIONS
DEVOPS / SCRIPTING AUTOMATISATION
DEVELOPPEMENT WEB & MOBILE

ADRAR FORMATION

Plan de Test

Durée Totale du Module : 4H

Auteur :
Jean-François Pech

Relu, validé & visé par :
- ☒ Jérôme CHRETIENNE
- ☒ Sophie POULAKOS
- ☒ Mathieu PARIS

Date création :
03/03/2023

Date révision :
10/03/2023

Présentation des plans de test

Comme nous l'avions abordé, lors du module de gestion de projet, toute nouvelle création, va forcément engendrer des incertitudes. Ces dernières font partie intégrante de la gestion de projet et il est nécessaire de les prendre en considération.

De plus, les technologies, les usages, les fonctionnalités et les tendances sont en constante évolution et de fait, et de faire les manières de développer ses applications, ont-elles aussi évoluées. Par exemple, la manière de développer une application Web dans les années 90 n'a plus grand-chose à voir par rapport aux années 2000 2010 ou même actuellement elles vont devoir répondre à différentes contraintes.

Cela dépend aussi du domaine dans lequel va s'inscrire notre application. En effet nous n'avons pas les mêmes contraintes si l'on développe le site vitrine d'un salon de coiffure, ou si on gère une application de prise de rendez-vous médical par exemple.

Le contexte juridique, à lui aussi évolué dans les années 90 ou 2000, on ne parlait pas ou très peu du respect de la protection de la vie privée, et de comment sont utilisées les données d'un utilisateur sur une application.

De fait, nous allons devoir nous assurer d'un certain niveau de qualité lorsque l'on développe un produit ou un service. Dans notre cas en tant que développeur Web mobile cet aspect de qualité va essentiellement s'opérer par des tests. Mais du coup il va nous falloir précisément réfléchir et décrire à comment nous allons organiser et exécuter ces tests.

Si l'on prend l'exemple d'une application mobile qui plante lorsqu'il y a trop d'utilisateurs connectés en même temps, il est fort probable que les développeurs n'ai pas pu tester ce scénario et qui est donc un risque qui est survenu peut-être un peu plus tard et qui s'avère être un problème majeur (cf. Facebook avec React).
Si seulement ces développeurs, avaient eu un plan de test bien conçu, ils auraient pu être en mesure d'identifier cette situation d'embouteillages, identifier le réel problème, ils auraient par exemple pu mettre en place un système qui va simuler beaucoup d'utilisateurs et mettre en place des actions correctives ils auraient résolu ce problème avant que l'utilisateur final ne soit confronté à ce bug.

Autre exemple, dans le cadre de la construction d'une maison, normalement on voudrait éviter que la construction du toit précède la construction, des fondations ou des murs.
Un plan de test dans une entreprise va agir de la même manière. Cela va nous aider à rattraper des problèmes avant qu'ils ne deviennent trop importants. Le plan de test va agir comme un guide qui va nous permettre de garantir que le logiciel l'application que l'on est en train de créer fonctionne parfaitement.

Pour résumer plus simplement un plan de test va indiquer ce que vous devez tester comment le test doit se dérouler et qui doit effectuer les tests. On se retrouve donc à la croisée des chemins entre la gestion des risques d'ordre technique et l'assurance qualité.

Le plan de test c'est donc un document **dynamique** qui devra être mis à jour régulièrement.
On peut, en conclure que la réussite d'un projet va dépendre fortement de la bonne rédaction de ce document de plan de test.

Le plan de test va aborder plusieurs points qui vont permettre de définir au maximum le périmètre du projet.

On va décrire :

- En quoi consiste le projet,
- quelle partie du projet on va tester
- quelle partie du projet on ne va pas tester,
- on va également décrire l'environnement dans lequel on va tester,
- décrire les risques qui peuvent survenir lors des tests.
- Nous allons aussi décrire les ressources pour mener à bien les tests :
  - que ce soit des ressources humaines donc les personnes impliquées
  - les ressources techniques c'est-à-dire les outils spécifiques pour mener les tests,
- on va aussi préciser le budget alloué pour les tests
- donner une notion de temporalité en précisant quelle est la durée du projet et à quel moment on va effectuer les tests.

(Voir Syllabus P70)

Quel Format pour les plans de test

Il y a plusieurs modèles de plan de test disponibles sur Internet, mais chaque entreprise va avoir sa version adaptée à ses projets à son fonctionnement.
Quoi qu'il en soit on va toujours retrouver les mêmes thématiques abordées, que l'on peut résumer dans le tableau ci-dessous.

PLAN DE TESTS	INTRODUCTION	
PÉRIMÈTRE	ENVIRONNEMENTS	PERSONNES
HORS-PÉRIMÈTRE	OUTILS	CALENDRIER
RISQUES		

Quels acteurs concernés

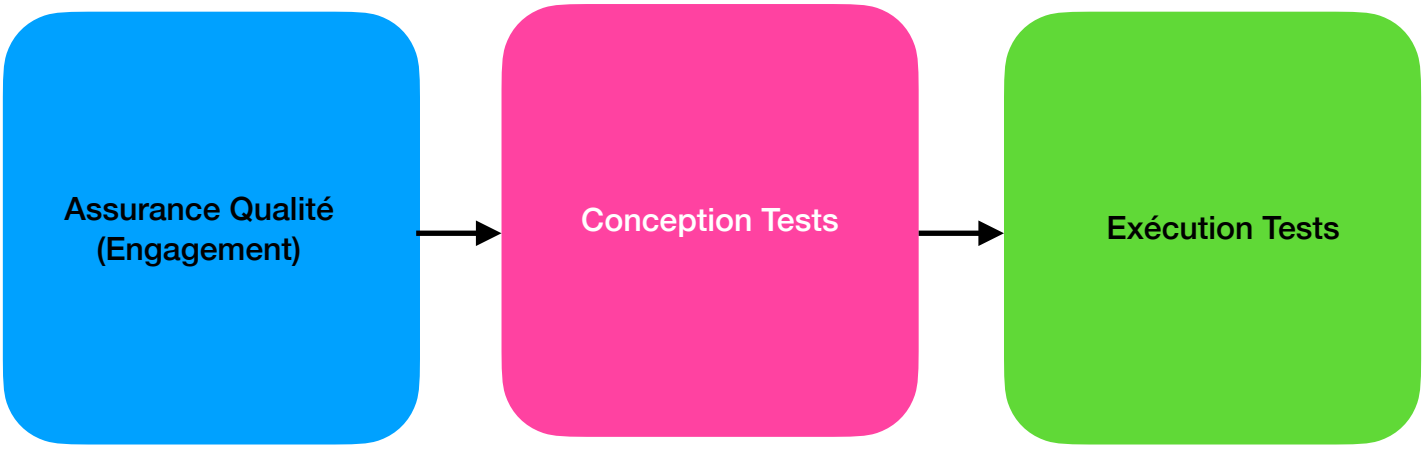
Le plan de test c'est un document qui va servir de points de référence sur laquelle va se baser l'équipe de l'assurance qualité

Afin d' améliorer la transparence du travail de l'équipe qualité vis-à-vis des équipes internes ou externes, c'est un document qu'on va partager avec le business analyste le où les chefs de projet, les équipes de Développement interne et les équipes de Développement externes s'il y en a.

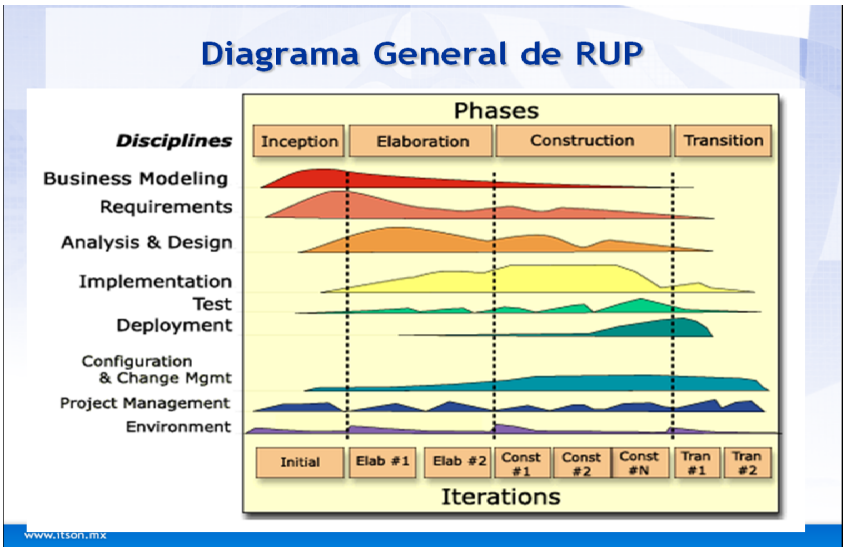
C'est donc un document qui va intégrer tous les membres ou toutes les parties prenantes du projet et qui sera réalisé par le responsable de test ou test lead ou test manager il y a plusieurs appellations, mais c'est généralement quelqu'un qui a de l'expérience dans le domaine du testing et qui a une maîtrise sur le projet à tester.

Planification des tests

Généralement, la planification des tests va se répartir en trois temps un tiers du temps qui sera nécessaire à l'engagement de l'assurance qualité. Le second tiers sera destiné à la conception des tests et enfin le dernier tiers sera réservé à l'exécution des tests.



Exemple de retranscription dans une approche « agile » style UP ou RUP :
Dans cet exemple de projet on voit que les effort (ressources) sont mobilisé dès la conception (inception) de l'application (serait-ce les plans de test qui se mettent en rédaction ?) ensuite dans les autres phases du projet des cycles réguliers (unitaires low cost, puis intégrations, E2E, etc…)



Défi : Selon voter expertise essayez d'imaginer des tests et placez les dans la timeline d'un projet.

Principales étapes d'élaboration d'un plan de test

Pour élaborer un plan de test, on va pouvoir se baser sur principalement huit étapes que l'on va décrire ci-dessous

1° Définir les objectifs

La première étape consiste à définir les objectifs du plan de test. Ces objectifs doivent être clairs et précis, et doivent inclure la couverture des différentes fonctionnalités du logiciel, la validation de la conformité aux exigences et la détection des défauts. Il est important de déterminer les attentes quant à la qualité du logiciel.

2° Identifier les ressources

Dans cette étape, il est essentiel de déterminer les ressources nécessaires à la réalisation des tests. Cela comprend les ressources humaines, telles que les testeurs et les responsables de test, ainsi que les ressources matérielles et logicielles, telles que les outils de test et les environnements de test.

3° Définir la stratégie de test

La stratégie de test définit l'approche globale pour l'exécution des tests. Cela comprend la sélection des techniques de test appropriées, la définition des scénarios de test et des cas de test, ainsi que la planification des activités de test. La stratégie de test doit répondre aux objectifs définis précédemment.

4° Planifier des activités de test

Dans cette étape, il convient de planifier les différentes activités de test à effectuer. Cela comprend la répartition des tâches entre les testeurs, l'attribution des ressources, la définition du calendrier des tests et la création d'un plan de déploiement pour les différentes versions du logiciel.

5° Elaborer des cas de test

Les cas de test décrivent les différentes procédures à suivre pour vérifier que le logiciel fonctionne correctement. Ils doivent être conçus de manière à garantir une couverture complète des fonctionnalités du logiciel et des scénarios d'utilisation attendus. Les cas de test doivent être clairs, précis et reproductibles.

6° Executer les tests

Une fois que les cas de test ont été élaborés, il est temps de les exécuter. Cela peut être fait manuellement ou à l'aide d'outils de test automatisés. Il est essentiel de documenter les résultats des tests et de signaler toute anomalie détectée.

7° Analyser les résultats et mener des actions correctives

Après l'exécution des tests, il est nécessaire d'analyser les résultats et de corriger les défauts identifiés. Cela peut impliquer des cycles supplémentaires d'exécution des tests et d'analyse des résultats jusqu'à ce que le logiciel atteigne un niveau acceptable de fiabilité.

8° Rédiger un rapport de test

Enfin, un rapport de test doit être rédigé pour rendre compte des activités de test réalisées, des résultats obtenus et des recommandations pour améliorer la qualité du logiciel. Ce rapport peut servir de référence pour les futures itérations de développement logiciel.

Exécution et suivi du plan de test

Le processus d'exécution et de suivi du plan de test est une étape cruciale dans la réalisation d'un projet. Tout au long de cette étape, les tests sont exécutés pour valider le bon fonctionnement du logiciel développé. Il est essentiel de suivre de près les résultats des tests afin de garantir la qualité de l'application.

Exécution des tests

L'exécution des tests consiste à mettre en œuvre les cas de test qui ont été spécifiés dans le plan de test. Cette étape permet de vérifier si l'application répond aux exigences fonctionnelles et non fonctionnelles. Voici quelques étapes clés pour une bonne exécution des tests :

1° Préparation de l'environnement :
Avant de commencer l'exécution des tests, il est important de s'assurer que l'environnement de test est prêt. Cela inclut l'installation des différentes versions du logiciel, la préparation de la base de données de test et la configuration des machines.

2° Sélection des cas de test :
Les cas de test à exécuter sont sélectionnés en fonction de leur pertinence et de leur importance. Une analyse des risques peut aider à prioriser les cas de test critiques à exécuter en premier.

3° Exécution des tests :
Les cas de test sont exécutés conformément aux instructions spécifiées dans le plan de test. Les résultats des tests, tels que les erreurs détectées, sont enregistrés pour une évaluation ultérieure.

4° Suivi des anomalies :
Si des erreurs sont détectées pendant l'exécution des tests, elles sont signalées et enregistrées dans un système de suivi des anomalies. Ces anomalies seront analysées et résolues ultérieurement.

Suivi des tests

Le suivi des tests est une activité qui dure tout au long de l'exécution des tests. Elle permet de s'assurer que tous les cas de test ont été exécutés, que les anomalies ont été résolues et que le logiciel se conforme aux exigences spécifiées. Voici quelques étapes clés pour un bon suivi des tests :

1° Reporting des résultats :
Après chaque exécution de test, un rapport est généré pour documenter les résultats. Ce rapport doit inclure les cas de test exécutés, les erreurs détectées, les anomalies signalées et leur évolution.

2° Analyse des résultats :
Les rapports de test sont analysés pour identifier les tendances et les modèles d'erreurs. Cela permet d'ajuster les stratégies de test et d'améliorer la qualité du logiciel.

3°Mise à jour du plan de test :
Si des erreurs majeures sont détectées ou si de nouveaux cas de test sont identifiés, le plan de test doit être mis à jour en conséquence. Cela garantit que toutes les parties du logiciel sont adéquatement testées.

4°Répétition des tests :
Dans certains cas, des tests supplémentaires peuvent être nécessaires pour vérifier que les anomalies découvertes ont été corrigées. Cela garantit également l'intégrité globale du logiciel.

Le processus d'exécution et de suivi du plan de test permet de renforcer la confiance dans la qualité du logiciel développé. En suivant ces étapes et en effectuant les ajustements nécessaires, il est possible d'assurer la stabilité et la performance optimale de l'application.

Différents types de test

Test unitaire

Les tests unitaires sont des tests de très bas niveau, proches de la source de votre application. Ils consistent à valider individuellement les plus petites unités de code, comme des fonctions, des méthodes ou des classes, utilisées par votre logiciel, afin de vérifier qu'elles fonctionnent correctement. Chaque unité est testée de manière isolée, sans dépendance à d'autres parties du code ou systèmes externes (comme des bases de données ou des API), pour s'assurer qu'elle produit les résultats attendus pour des entrées spécifiques.

Ces tests sont généralement automatisés, ce qui les rend faciles à intégrer dans des processus d'intégration continue (CICD), où ils sont exécutés rapidement à chaque modification du code. De plus, les tests unitaires sont relativement peu coûteux à mettre en place et à maintenir. Ils permettent une détection rapide des erreurs et offrent la possibilité de refactoriser le code en toute confiance.

Les tests unitaires sont essentiels dans le développement logiciel, en particulier dans le cadre du Test-Driven Development (TDD) et des pratiques agiles. Ils contribuent non seulement à améliorer la qualité du logiciel, mais agissent également comme une forme de documentation vivante, décrivant comment chaque composante doit se comporter. Bien qu'ils ne garantissent pas la stabilité totale d'une application, ils constituent un socle solide pour détecter les régressions avant que celles-ci n'affectent l'intégralité du système.

Caractéristiques clés des tests unitaires

- Isolation : Chaque test est effectué de manière isolée, en testant uniquement la logique interne de l'unité de code, sans dépendances externes.
- Automatisé : Les tests unitaires sont généralement automatisés, permettant leur exécution répétée à chaque modification du code (intégration continue CICD) pour détecter les régressions.
- Déterministe : Un test unitaire doit toujours produire le même résultat avec les mêmes entrées, garantissant sa fiabilité.
- Rapide à exécuter : En raison de leur petite portée et de l'absence de dépendances lourdes, les tests unitaires s'exécutent rapidement, ce qui en fait un choix optimal pour une intégration continue.
- Granularité fine : Contrairement aux tests d'intégration ou end-to-end (E2E), les tests unitaires s'appliquent à de petites parties du code, comme des fonctions individuelles, des méthodes ou des modules.

Objectifs des tests unitaires

- Détection précoce des bugs : Les erreurs dans le code sont détectées dès le développement, ce qui permet de corriger les anomalies avant qu'elles n'affectent des parties plus complexes de l'application.
- Réduction des régressions : En testant systématiquement chaque unité de code après chaque changement, on minimise le risque de casser une partie fonctionnelle du logiciel en introduisant des régressions.
- Documentation du code : Les tests unitaires agissent comme une forme de documentation vivante. En lisant les tests, un développeur peut comprendre rapidement comment une fonction ou une classe est censée se comporter.
- Refactorisation facilitée : Les tests unitaires permettent de refactorer le code en toute confiance. S'ils sont bien écrits, ils garantissent que la refactorisation n'a pas affecté la logique métier du programme.

Structure d'un test unitaire

Un test unitaire typique suit généralement une structure en trois étapes, appelée "AAA" (Arrange, Act, Assert) :
Arrange : Préparer les conditions initiales, les entrées, ou les objets nécessaires pour le test.
Act : Exécuter l'unité de code que l'on souhaite tester.
Assert : Vérifier que les résultats obtenus correspondent aux résultats attendus.

Exemple en pseudo code :

```
// Arrange
functionUnderTest = new FunctionUnderTest()
input = 5

// Act
result = functionUnderTest.doSomething(input)

// Assert
assertEqual(result, expectedValue)
```

Avantages des tests unitaires

Sécurité lors des modifications : Les développeurs peuvent changer ou ajouter du code en toute confiance, sachant que les tests unitaires vérifieront automatiquement si quelque chose ne fonctionne plus correctement.

Amélioration de la conception : La rédaction de tests unitaires pousse à découpler le code, car une unité bien testée doit être indépendante des autres.

Coût de maintenance réduit : Les erreurs étant détectées rapidement, le coût de correction des bugs est beaucoup plus faible à long terme.

Limites des tests unitaires

Dépendances complexes : Si l'unité de code dépend de plusieurs services externes (bases de données, API, etc.), écrire des tests unitaires peut devenir difficile et nécessiter des objets factices (mocking).

Ne testent pas tout : Les tests unitaires ne garantissent pas que l'ensemble du système fonctionne. Ils ne vérifient que les petites unités, et non les interactions complexes entre les composants ou la performance.

Temps de mise en place : La mise en place de tests unitaires peut sembler fastidieuse au départ, surtout dans les projets existants sans tests.

Test d'intégration

Les tests d'intégration sont une technique de validation dans le développement logiciel qui consiste à tester l'interaction entre plusieurs modules ou composants d'une application pour s'assurer qu'ils fonctionnent correctement ensemble. Contrairement aux tests unitaires, qui isolent les plus petites unités de code, les tests d'intégration vérifient la bonne communication et interaction entre ces unités dans un environnement plus large.

Les tests d'intégration visent à valider que des unités de code ou des modules distincts, une fois combinés, fonctionnent harmonieusement ensemble pour accomplir des tâches ou des fonctionnalités spécifiques. L'objectif est de s'assurer que les interfaces entre les différents composants respectent leurs contrats d'utilisation et que les données ou informations échangées sont correctes.

Ces tests sont particulièrement importants pour détecter des problèmes d'interaction qui ne seraient pas visibles lors des tests unitaires. Par exemple, même si chaque module individuel fonctionne correctement (grâce aux tests unitaires), des erreurs peuvent survenir lorsque ces modules interagissent, souvent à cause de problèmes liés aux dépendances, aux formats de données, ou aux erreurs de logique dans les appels inter-modulaires.

Caractéristiques clés des tests d'intégration

- Interaction entre composants : Ils testent l'intégration des modules pour s'assurer qu'ils se connectent et se comportent correctement les uns par rapport aux autres.
- Focus sur les interfaces : L'accent est mis sur les interfaces entre les différents composants pour s'assurer que les données circulent correctement, que les appels entre composants sont respectés, et que les comportements attendus sont observés.
- Pas totalement isolés : Contrairement aux tests unitaires, les tests d'intégration ne testent pas chaque composant de manière isolée. Ils peuvent inclure des services externes comme des bases de données, des systèmes de fichiers ou des API.
- Complexité intermédiaire : Bien que plus complexes que les tests unitaires, les tests d'intégration ne testent généralement pas l'application complète comme les tests end-to-end. Ils se concentrent sur des ensembles de modules qui interagissent directement.
- Partiellement automatisés : Ils peuvent être automatisés et exécutés dans le cadre d'un pipeline d'intégration continue, bien que leur exécution prenne souvent plus de temps que celle des tests unitaires, notamment en raison des dépendances externes.

Objectifs des tests d'intégration

- Vérifier la communication entre les composants : Les tests d'intégration garantissent que les modules intégrés communiquent entre eux correctement, en envoyant et recevant des données selon les attentes.
- Détecter les problèmes d'intégration précoces : Ils permettent de détecter rapidement des erreurs dues à des incompatibilités entre modules, à des changements dans les interfaces, ou à des dépendances manquantes.
- Stabilité des interactions : Ils assurent que les interactions entre différents composants ou services restent stables au fil du temps, même lorsque des mises à jour ou des changements sont apportés à certaines parties du système.
- Réduction des risques de régression : En combinant des modules testés isolément, les tests d'intégration s'assurent que les modifications apportées à l'un n'introduisent pas de régressions dans les autres.

Types de tests d'intégration

Big Bang Testing : Tous les modules sont intégrés en une seule fois, puis testés ensemble. Cela permet de tester l'ensemble du système intégré, mais peut rendre difficile la localisation des erreurs si plusieurs modules sont impliqués.
Testing incrémental :
Testing ascendant (Bottom-Up) : Les modules de plus bas niveau (les plus simples) sont testés en premier, puis les modules de plus haut niveau sont intégrés progressivement.
Testing descendant (Top-Down) : On commence par les modules de plus haut niveau, et on ajoute progressivement les modules de plus bas niveau.
Testing en sandwich : C'est une combinaison des approches ascendante et descendante, où certains modules sont testés en partant du bas et d'autres en partant du haut simultanément.

Exemple de structure de test d'intégration

Imaginons une application de commerce électronique avec deux modules :
"Gestion des utilisateurs" et "Système de paiement". Un test d'intégration pourrait consister à vérifier que :

Lorsqu'un utilisateur s'inscrit et passe une commande, les données sont bien transmises du module utilisateur au module de paiement.

Les informations de paiement sont correctement traitées, et la confirmation du paiement est renvoyée au module utilisateur.

Exemple en code

```
// Arrange
UserModule = new UserModule()
PaymentModule = new PaymentModule()

// Act
user = UserModule.createUser("John Doe", "johndoe@example.com")
order = PaymentModule.createOrder(user, cartItems)

// Assert
assert(order.status, "PAID")
assert(UserModule.getUserOrders(user).contains(order))
```

Comparaison avec les tests unitaires et end-to-end

Tests unitaires : Testent des unités individuelles de code de manière isolée, en vérifiant si elles fonctionnent comme prévu.

Tests d'intégration : Vérifient que les unités ou modules interagissent correctement entre eux, en mettant l'accent sur leurs interfaces.

Tests end-to-end : Testent le flux complet d'une application, depuis l'interface utilisateur jusqu'aux composants les plus bas, dans un scénario réel.

Avantages des tests d'intégration

Détection des erreurs d'intégration : Ils permettent de détecter des erreurs non visibles lors des tests unitaires, notamment celles liées aux interfaces et aux dépendances.

Garantie d'une cohésion fonctionnelle : Ils assurent que l'ensemble du système ou des sous-systèmes fonctionne correctement ensemble.

Meilleure couverture des dépendances externes : Les tests d'intégration peuvent inclure des interactions avec des services ou des systèmes externes (bases de données, services web, API), fournissant une meilleure vue d'ensemble du comportement du logiciel dans son environnement réel.

Limites des tests d'intégration

Dépendances complexes : Si les modules testés ont de nombreuses dépendances externes (comme des API tierces), les tests d'intégration peuvent devenir plus difficiles à maintenir et à exécuter.

Temps d'exécution : Les tests d'intégration peuvent prendre plus de temps que les tests unitaires en raison des interactions entre modules et des dépendances.

Difficulté de localisation des erreurs : Si un test échoue, il peut être plus difficile de déterminer précisément quelle partie du système est en cause (contrairement aux tests unitaires).

Test fonctionnel

Les tests fonctionnels visent à valider que chaque fonctionnalité du logiciel fonctionne selon les spécifications définies dans le cahier des charges ou dans les user stories. Ils couvrent des aspects tels que la validation des entrées, le traitement des données et la production de sorties correctes. Leur objectif principal est de garantir que le logiciel satisfait aux exigences fonctionnelles sans se soucier de la structure interne du code (boîte noire).

Il y a parfois une certaine confusion entre les tests d'intégration et les tests fonctionnels, car ils nécessitent tous les deux plusieurs composants pour interagir. La différence réside dans le fait qu'un test d'intégration peut simplement vérifier que vous pouvez interroger la base de données, tandis qu'un test fonctionnel s'attend à obtenir une valeur spécifique de la base de données, telle que définie par les exigences du produit.

Caractéristiques clés des tests fonctionnels

Validation des fonctionnalités : Ils vérifient que les fonctionnalités implémentées répondent aux besoins métier et se comportent comme attendu dans des cas d'utilisation spécifiques.

Approche en boîte noire : Les tests fonctionnels ne tiennent pas compte de la manière dont le système est implémenté, mais uniquement de la sortie en fonction des entrées fournies.

Évaluation de bout en bout des cas d'utilisation : Ils permettent de tester une fonctionnalité dans son intégralité, souvent à partir de la perspective de l'utilisateur final.

Focus sur la logique métier : Ils se concentrent sur le quoi plutôt que sur le comment, validant que chaque composante de l'application fournit le bon comportement selon les règles métier définies.

Tests automatisés ou manuels : Les tests fonctionnels peuvent être réalisés manuellement, mais ils sont souvent automatisés pour permettre des exécutions répétées et efficaces.

Objectifs des tests fonctionnels

- Valider les fonctionnalités spécifiées : S'assurer que le logiciel respecte les exigences fonctionnelles définies dans les spécifications techniques ou les user stories.
- Garantir la satisfaction des utilisateurs finaux : Les tests fonctionnels se concentrent sur l'expérience utilisateur pour garantir que les utilisateurs finaux peuvent accomplir leurs tâches dans l'application.
- Prévenir les bugs logiques : Ils aident à identifier les erreurs dans l'implémentation de la logique métier, comme une mauvaise validation des données, une logique incorrecte, ou des interactions non souhaitées.
- Assurer la fiabilité de l'application : En validant les fonctionnalités clés, les tests fonctionnels garantissent la stabilité du système avant de passer aux phases de tests plus approfondies (comme les tests d'acceptation ou end-to-end).

Types de tests fonctionnels

Tests de validation des exigences : Ils vérifient que l'application remplit toutes les exigences fonctionnelles spécifiées.
Tests de cas d'utilisation (Use Case Testing) : Ils valident que le logiciel permet de réaliser un ensemble de scénarios basés sur des cas d'utilisation utilisateur spécifiques.
Tests de scénarios métier : Ils simulent des scénarios réels pour valider que les fonctionnalités sont disponibles et correctes dans le cadre de processus métier concrets.
Tests de validation des flux de travail : Ils s'assurent que les différents flux de travail (comme l'inscription d'un utilisateur, la gestion des paniers dans un site e-commerce, etc.) sont correctement exécutés.
Tests de régression fonctionnelle : Ils vérifient que les nouvelles fonctionnalités ou modifications apportées au système n'ont pas affecté les fonctionnalités existantes.

Exemple de test fonctionnel

Imaginons une application de gestion de comptes bancaires. Un test fonctionnel pour la fonctionnalité "virement bancaire" pourrait se dérouler comme suit :

Entrées : Un utilisateur saisit ses identifiants pour se connecter, sélectionne un compte source, saisit un montant, et choisit un compte destinataire.

Exécution : L'utilisateur clique sur "Effectuer le virement".

Sortie attendue : Le système valide que l'utilisateur a suffisamment de fonds, effectue le transfert, et met à jour les soldes des comptes source et destinataire.

Validation : Le test vérifie que les montants sont bien débités et crédités, et qu'un reçu est généré.

Exemple côté code

```
// Arrange
LoginPage = new LoginPage()
TransferPage = new TransferPage()

// Act
user = LoginPage.login("username", "password")
transferSuccess = TransferPage.makeTransfer(user, "Account1", "Account2", 100)

// Assert
assert(transferSuccess == true)
assert(TransferPage.getBalance("Account1") == oldBalanceAccount1 - 100)
assert(TransferPage.getBalance("Account2") == oldBalanceAccount2 + 100)
```

Comparaison avec les autres types de tests

Tests unitaires : Testent des unités spécifiques du code (fonctions, méthodes), en vérifiant qu'elles fonctionnent isolément comme prévu.

Tests d'intégration : Valident que les composants fonctionnent correctement ensemble, en testant leurs interactions.

Tests fonctionnels : Se concentrent sur les fonctionnalités du système dans leur ensemble, testant les cas d'utilisation du **point de vue de l'utilisateur final.**

Méthodes d'exécution

Tests manuels : Les testeurs humains suivent des scénarios ou des cas de test définis pour vérifier que le logiciel fonctionne correctement. Cela est souvent utilisé pour des tests exploratoires ou des validations fonctionnelles spécifiques.

Tests automatisés : Ces tests sont écrits dans du code et peuvent être exécutés automatiquement à chaque mise à jour ou déploiement du logiciel. Cela permet de garantir la répétabilité et l'efficacité, notamment dans le cadre d'un pipeline d'intégration continue.

Outils utilisés pour les tests fonctionnels

Selenium : Un outil populaire pour automatiser les tests fonctionnels des applications web, en interagissant avec l'interface utilisateur comme le ferait un utilisateur.
Cucumber : Utilisé pour écrire des tests fonctionnels en langage naturel (Gherkin), facilitant la collaboration entre les équipes techniques et non techniques.
Cypress : Un outil moderne pour les tests fonctionnels des applications web, qui permet d'écrire des tests rapides et fiables pour les interfaces utilisateurs et les API.
TestCafe : Un autre outil pour les tests d'interface utilisateur automatisés, conçu pour tester les applications web dans n'importe quel navigateur.
QTP/UFT : Un outil d'automatisation pour tester les applications logicielles fonctionnelles, souvent utilisé dans les entreprises.

Avantages des tests fonctionnels

Assurance que le logiciel répond aux exigences : Les tests fonctionnels garantissent que le logiciel satisfait aux besoins spécifiés par les utilisateurs et les parties prenantes.

Tests centrés sur l'utilisateur : Ces tests sont souvent réalisés du point de vue de l'utilisateur final, s'assurant que l'application répond bien à ses attentes.

Détection précoce des anomalies : Ils permettent de détecter rapidement des anomalies fonctionnelles dans les fonctionnalités clés avant que le logiciel ne soit mis en production.

Tests automatisés possibles : L'automatisation des tests fonctionnels permet de réduire le temps passé en validation manuelle et augmente la couverture des tests, en particulier lors des mises à jour fréquentes.

Limites des tests fonctionnels

Nécessitent des scénarios d'utilisation précis : Si les spécifications fonctionnelles sont incomplètes ou incorrectes, les tests fonctionnels peuvent manquer des problèmes critiques.

Ne couvrent pas les aspects non fonctionnels : Les tests fonctionnels ne prennent pas en compte les performances, la sécurité ou la compatibilité. Ils ne garantissent pas que l'application réponde aux exigences non fonctionnelles.

Temps d'exécution plus long : Les tests fonctionnels peuvent prendre du temps à s'exécuter, surtout s'ils ne sont pas automatisés ou s'ils incluent de nombreux cas d'utilisation.

Test UI - GUI

Les tests UI (User Interface), également appelés tests GUI (Graphical User Interface), sont une forme de tests fonctionnels qui visent à valider que l'interface utilisateur d'une application fonctionne correctement et que ses composants graphiques interagissent correctement avec les utilisateurs. Ces tests se concentrent sur l'interface visuelle et les éléments interactifs du logiciel, tels que les boutons, champs de texte, menus déroulants, fenêtres, et autres contrôles graphiques, pour s'assurer que l'expérience utilisateur est fluide et conforme aux spécifications.

Les tests UI/GUI se concentrent sur la validation des comportements visuels et des fonctionnalités interactives de l'application. Ils testent la manière dont l'utilisateur interagit avec l'interface, en vérifiant que tous les composants visibles et interactifs fonctionnent correctement et produisent les résultats attendus. L'objectif principal est de garantir que les utilisateurs peuvent interagir de manière fluide avec le logiciel via son interface graphique.

Objectifs des tests UI/GUI

- **Vérifier la conformité visuelle :** S'assurer que les éléments de l'interface utilisateur, tels que les boutons, icônes, champs de texte, et menus, sont correctement disposés, visibles, et respectent la charte graphique spécifiée.
- **Tester les interactions utilisateur :** Vérifier que toutes les interactions possibles dans l'interface (clics, saisies, déplacements, etc.) produisent les actions attendues.
- **Assurer l'ergonomie :** Valider que l'expérience utilisateur est intuitive, que les contrôles sont accessibles, et que l'interface respecte les principes d'ergonomie.
- **Détecter les erreurs visuelles :** Identifier les problèmes liés à la mise en page, aux éléments mal alignés, aux textes coupés, ou à des icônes absentes.

Types de tests UI/GUI

Tests de validation des composants (UI) : Ils vérifient que chaque élément de l'interface (boutons, menus, champs de saisie) fonctionne correctement lorsqu'il est manipulé par l'utilisateur.
Tests de navigation : Ils valident que l'utilisateur peut naviguer facilement à travers les différentes pages de l'application ou les sections d'une interface sans erreurs.
Tests de réactivité : Ils testent la capacité de l'interface à s'adapter à différentes résolutions d'écran, appareils, ou tailles de fenêtre (responsive design).
Tests de compatibilité UI : Ils s'assurent que l'interface fonctionne correctement sur différents navigateurs, systèmes d'exploitation, et appareils.
Tests d'accessibilité : Ils vérifient que l'interface est utilisable par des personnes ayant des handicaps, et qu'elle est conforme aux normes d'accessibilité (par exemple, WCAG).

Auteur : Jean-François Pech
Date création : 03/03/2023
Relu, validé & visé par :
☒ Jérôme CHRETIENNE
☒ Sophie POULAKOS
☒ Mathieu PARIS
Date révision : 10/03/2023

Outils utilisés pour les tests UI/GUI

Selenium : Un des outils les plus populaires pour automatiser les tests UI des applications web. Il permet de simuler des interactions utilisateur telles que des clics, des saisies, et des déplacements de souris.
Cypress : Un framework moderne pour les tests UI d'applications web, connu pour sa rapidité et sa simplicité d'utilisation.
Puppeteer : Un outil pour contrôler Chrome/Chromium via des scripts pour automatiser des tests UI.
TestCafe : Un autre outil de test UI qui permet de tester les interfaces graphiques sur différentes plateformes et navigateurs.
Appium : Utilisé pour tester les applications mobiles, il permet de simuler les interactions des utilisateurs sur les interfaces graphiques mobiles.

Exemple de test UI/GUI

Imaginons une interface d'application de gestion de comptes où l'utilisateur doit se connecter.

Un test UI pourrait vérifier que le champ "Nom d'utilisateur" accepte les entrées, que le bouton "Se connecter" est activé après la saisie des informations, et que l'utilisateur est redirigé vers la bonne page après la connexion.

Exemple en code

```
// Arrange
LoginPage = new LoginPage()

// Act
LoginPage.enterUsername("john.doe")
LoginPage.enterPassword("password123")
LoginPage.clickLoginButton()

// Assert
assert(LoginPage.isLoginSuccessful() == true)
assert(LoginPage.getCurrentPage() == "Dashboard")
```

Tests UI manuels vs automatisés

Tests manuels : Un testeur humain interagit directement avec l'interface, en naviguant dans les menus, en cliquant sur les boutons, et en effectuant des saisies pour valider que l'interface fonctionne comme attendu. Cette approche est utilisée pour les tests exploratoires ou lors des phases initiales du développement.

Tests automatisés : Les interactions sont simulées par des scripts ou des outils d'automatisation qui imitent les actions d'un utilisateur réel. Cela permet de répéter les tests plus rapidement, avec moins de risque d'erreurs humaines, et est particulièrement utile dans le cadre de l'intégration continue.

Comparaison avec les autres types de tests

Tests unitaires : Testent des fonctions ou méthodes individuelles sans interface graphique.

Tests d'intégration : Vérifient la communication entre plusieurs modules ou systèmes.

Tests fonctionnels : Testent les fonctionnalités du système du point de vue des utilisateurs, mais ne se concentrent pas spécifiquement sur l'apparence ou la mise en page visuelle.

Avantages des tests UI/GUI

Validation complète de l’expérience utilisateur : Les tests UI/GUI valident les interactions directes entre l'utilisateur et l'application, garantissant ainsi une expérience utilisateur de qualité.

Automatisation possible : Avec des outils comme Selenium ou Cypress, il est possible d'automatiser les tests d'interface utilisateur, ce qui permet de tester plus rapidement des cas d'utilisation complexes.

Détection des erreurs visuelles : Ils permettent d’identifier des erreurs que d'autres types de tests, comme les tests unitaires ou d'intégration, ne peuvent pas révéler (ex : boutons mal placés, éléments non visibles).

Assurance multi-plateforme : Ces tests peuvent être exécutés sur différentes configurations (navigateurs, systèmes d’exploitation, appareils mobiles), garantissant que l’interface utilisateur fonctionne correctement partout.

Limites des tests UI/GUI

Maintenance coûteuse : Les tests UI/GUI automatisés peuvent être fragiles, car toute modification de l'interface utilisateur (comme un changement de position de bouton ou de design) peut entraîner des ajustements des scripts de tests.

Exécution plus lente : Les tests UI, en particulier les tests end-to-end qui couvrent de nombreux scénarios utilisateur, peuvent être lents à exécuter, car ils nécessitent souvent de lancer l'application complète et de simuler des interactions complexes.

Nécessitent des configurations spécifiques : Tester sur différentes plateformes, tailles d'écran ou résolutions nécessite souvent des environnements ou des outils spécifiques, ce qui peut rendre les tests UI plus complexes à mettre en place.

Ne couvrent pas la logique interne : Les tests UI/GUI ne vérifient pas le code interne de l'application, la logique métier ou l'intégration des composants. Ils ne testent que l'**interface** visible par l'utilisateur.

Test E2E

Les tests End-to-End (E2E) sont une méthode de validation dans le développement logiciel qui vérifie le bon fonctionnement d'une application dans son ensemble, du point de vue de l'utilisateur final, en testant chaque étape d'un processus ou d'un flux complet, depuis l'entrée jusqu'à la sortie. Contrairement aux autres types de tests (tests unitaires, d'intégration ou UI), les tests E2E simulent les interactions réelles des utilisateurs avec le système, en testant l'intégration de toutes les couches et composants du logiciel, y compris l'interface utilisateur, la logique métier, les bases de données, les services externes, et d'autres systèmes interconnectés.

Les tests End-to-End visent à valider le flux global de l'application, en simulant des scénarios réels que les utilisateurs finaux seraient susceptibles de suivre. L'objectif est de s'assurer que tous les systèmes et sous-systèmes fonctionnent correctement ensemble dans un environnement de production simulé, et que l'ensemble du processus de bout en bout, depuis l'interaction initiale de l'utilisateur jusqu'à la sortie finale, fonctionne comme prévu.

Objectifs des tests End-to-End (E2E)

- Vérifier la cohésion du système global : Tester toutes les couches du système (interface utilisateur, API, bases de données, services tiers, etc.) pour s’assurer que chaque partie fonctionne correctement et de manière synchronisée.
- Simuler les scénarios utilisateur réels : Les tests E2E imitent les flux réels d’utilisateurs pour valider que les fonctionnalités critiques fonctionnent dans des conditions proches de la production.
- Valider les dépendances externes : Tester les interactions entre l’application et des systèmes externes tels que des API, services web, serveurs de messagerie, ou autres services tiers.
- Assurer l'intégrité des données : Vérifier que les données circulent correctement entre les différents systèmes et qu'elles sont bien traitées à chaque étape du processus.
- Garantir l'expérience utilisateur complète : Valider que l'application, de bout en bout, répond aux exigences fonctionnelles et offre une expérience fluide et cohérente aux utilisateurs.

Caractéristiques des tests End-to-End (E2E)

Tests de bout en bout des processus métier : Ils couvrent les flux complets, par exemple, dans une application e-commerce, depuis la recherche d'un produit jusqu'à l'achat et la réception d'un email de confirmation.

Interaction multi-système : Ils testent souvent l'intégration de l'application avec des services tiers ou d'autres systèmes, par exemple un système de paiement ou une API externe.

Approche boîte noire : Comme les tests fonctionnels, les tests E2E sont généralement exécutés en boîte noire, sans se soucier de la logique interne du système, mais en se concentrant sur les entrées et sorties visibles pour l'utilisateur.

Couvrent toutes les couches d'une application : Contrairement aux tests unitaires ou d'intégration qui testent des morceaux spécifiques du code, les tests E2E vérifient l'ensemble du flux utilisateur, souvent en incluant des aspects comme les interfaces utilisateur et les communications backend.

Focus sur la stabilité du produit en production : Les tests E2E sont effectués dans des environnements de pré-production ou de staging, simulant un environnement aussi proche que possible de la production.

Types de tests End-to-End

Tests de scénario utilisateur : Ils simulent les actions qu'un utilisateur final accomplirait dans l'application, comme s'inscrire, se connecter, passer une commande, ou générer un rapport.
Tests de processus métier complet : Ils valident des processus d'entreprise entiers, qui peuvent inclure plusieurs modules, systèmes et acteurs différents (ex : gestion d'inventaire, expédition, facturation).
Tests de validation des dépendances : Ils vérifient que les intégrations avec des services tiers ou des API externes fonctionnent comme prévu dans le contexte du flux utilisateur global.
Tests de régression End-to-End : Ils sont utilisés après des changements de code pour s'assurer que le système global continue de fonctionner correctement, en particulier que les flux critiques n'ont pas été affectés par les nouvelles modifications.

Exemples de scénarios de tests End-to-End

Imaginons un site de commerce électronique. Un test E2E pourrait vérifier un flux utilisateur complet, du début à la fin, incluant :

Recherche de produit : Un utilisateur se connecte, recherche un produit spécifique, et sélectionne un article.

Ajout au panier : L'utilisateur ajoute l'article à son panier, accède à son panier pour le vérifier, puis procède à la caisse.

Paiement : L'utilisateur saisit ses informations de paiement, vérifie le total, et finalise l'achat.

Confirmation de commande : L'utilisateur reçoit une confirmation à l'écran et un email de confirmation avec les détails de la commande.

Vérification des systèmes backend : Vérification que les systèmes de gestion des stocks, de facturation, et de livraison sont correctement mis à jour après la transaction.

Exemple code

```
// Arrange
HomePage = new HomePage()
LoginPage = new LoginPage()
ProductPage = new ProductPage()
CartPage = new CartPage()
CheckoutPage = new CheckoutPage()
EmailSystem = new EmailSystem()

// Act
HomePage.open()
LoginPage.login("john.doe", "password123")
ProductPage.search("Laptop")
ProductPage.addToCart("Laptop")
CartPage.checkout()
CheckoutPage.enterPaymentDetails("Visa", "4111 1111 1111 1111", "12/25", "123")
CheckoutPage.confirmPurchase()

// Assert
assert(CheckoutPage.isPurchaseSuccessful() == true)
assert(EmailSystem.hasReceivedEmail("john.doe@example.com", "Order Confirmation") == true)
```

Outils pour les tests End-to-End

Cypress : Un framework moderne et populaire pour automatiser les tests End-to-End des applications web, connu pour sa rapidité et sa simplicité.
Selenium : Un autre outil couramment utilisé pour les tests E2E, permettant d'automatiser les tests des interfaces web.
TestCafe : Un framework de test E2E pour les applications web qui ne nécessite aucune installation de plugins de navigateur ou de WebDriver.
Playwright : Un outil pour tester des applications web modernes avec la prise en charge de plusieurs navigateurs, idéal pour les tests E2E.

Avantages des tests End-to-End

Couverture complète : Ils valident les flux utilisateurs réels et l'interaction entre tous les composants du système, garantissant que toutes les parties fonctionnent bien ensemble.

Détection des problèmes d'intégration : Ces tests permettent de détecter les erreurs d'intégration entre différents systèmes ou services, ce qui peut être crucial pour les applications complexes.
Validation des dépendances externes : Ils s'assurent que les interactions avec les systèmes tiers (API, services de paiement, etc.) se déroulent comme prévu.

Simulent des scénarios de production : Les tests E2E sont exécutés dans des environnements qui imitent de près la production, ce qui permet de valider la robustesse de l'application dans des conditions réelles.

Assurent la qualité globale du produit : Ces tests garantissent que l'application entière, y compris les interactions entre les différents modules, est stable et prête pour la production.

Limites des tests End-to-End

Complexité : Les tests E2E couvrent de nombreux systèmes et flux, ce qui peut rendre leur conception, mise en place, et maintenance plus complexes que d'autres types de tests.

Temps d'exécution long : Les tests E2E impliquent de nombreux systèmes et processus, ce qui peut ralentir l'exécution, surtout dans les environnements CI/CD où l'efficacité est clé.

Dépendance des environnements externes : Si un service tiers ou une API est hors ligne, cela peut faire échouer les tests E2E, même si le problème ne vient pas de l'application elle-même.

Maintenance coûteuse : Les tests E2E sont plus sensibles aux changements dans l'application, notamment dans l'interface utilisateur, et nécessitent donc des mises à jour fréquentes des tests automatisés.

Test D’acceptation

Les tests d'acceptation sont une forme de tests logiciels utilisés pour vérifier si une application ou un système répond aux exigences et critères définis par les utilisateurs finaux, les parties prenantes ou les clients. Ils sont réalisés à la fin du cycle de développement pour valider que l'application fonctionne comme prévu du point de vue de l'utilisateur et qu'elle est prête pour la mise en production. Les tests d'acceptation sont souvent réalisés par les utilisateurs ou les représentants des utilisateurs et s'inscrivent dans le cadre de la validation finale du produit avant son déploiement.

Les tests d'acceptation (ou User Acceptance Tests - UAT) valident que le produit ou la fonctionnalité livrée respecte bien les attentes fonctionnelles et non fonctionnelles exprimées dans les spécifications. Ils constituent la dernière étape avant que le produit ne soit accepté par le client ou les utilisateurs finaux et qu'il soit prêt à être déployé en production. Ces tests se basent sur des scénarios réalistes qui correspondent aux cas d'usage que les utilisateurs vont suivre dans leur travail quotidien.

Objectifs des tests d'acceptation

- Vérifier la conformité avec les exigences : Les tests d'acceptation visent à s'assurer que l'application répond aux exigences fonctionnelles et métiers telles qu'elles ont été définies au début du projet ou lors des itérations agiles.
- Validation par les utilisateurs finaux : Ils sont souvent réalisés par les utilisateurs ou les parties prenantes, garantissant que l'application est utilisable et fonctionnelle du point de vue de ceux qui l'utiliseront au quotidien.
- Détection des écarts : Ils permettent de détecter d'éventuels écarts entre le comportement attendu de l'application et son comportement réel, particulièrement en termes de fonctionnalités.
- Assurer l'acceptabilité de la solution : En s’assurant que le produit respecte les critères d’acceptation définis, les tests valident que la solution est prête à être mise en production et à être adoptée par les utilisateurs.
- Valider l'expérience utilisateur : En plus des aspects purement fonctionnels, ils permettent de s'assurer que l'expérience utilisateur est conforme aux attentes, tant sur le plan de l'ergonomie que de l’accessibilité.

Types de tests d'acceptation

Tests d'acceptation utilisateur (UAT) : Ces tests sont réalisés par des utilisateurs finaux ou des représentants du client pour valider que le produit répond aux exigences métiers spécifiques et qu'il est prêt à être déployé.
Tests d'acceptation contractuelle : Ils valident que le produit ou le logiciel respecte les termes et les critères d'acceptation définis dans un contrat ou une spécification formelle.
Tests d'acceptation de système : Ces tests valident que l'intégralité du système, incluant toutes les fonctionnalités et modules, est conforme aux attentes avant son intégration dans l'environnement de production.
Tests alpha et bêta : Le test alpha est généralement réalisé en interne par des testeurs qualifiés ou les parties prenantes avant la publication à un plus large public. Le test bêta est réalisé par un groupe limité d'utilisateurs finaux dans un environnement réel pour évaluer l'application avant un déploiement public plus large.

Caractéristiques des tests d'acceptation

Réalisation par les utilisateurs finaux : Contrairement à d'autres types de tests qui peuvent être effectués par les développeurs ou les testeurs, les tests d'acceptation impliquent souvent directement les utilisateurs finaux ou les clients.

Scénarios basés sur les besoins métier : Les tests d'acceptation se concentrent sur des scénarios concrets, correspondant aux flux métiers ou aux processus que les utilisateurs utiliseront réellement.

Vérification des critères d'acceptation : Chaque fonctionnalité testée doit répondre à des critères d'acceptation clairs et bien définis, souvent exprimés dans les spécifications ou les user stories.

Dernière validation avant la mise en production : Ces tests sont généralement la dernière étape avant que l'application ne soit déployée dans un environnement de production.

Boîte noire : Les tests d'acceptation sont généralement des tests en boîte noire, où l'accent est mis sur les résultats attendus plutôt que sur le fonctionnement interne du code.

Exemple de scénarios de tests d'acceptation

Imaginons une plateforme de réservation en ligne. Un scénario de test d'acceptation pourrait vérifier si un utilisateur peut réserver un billet de train avec succès et recevoir une confirmation par email.

Connexion : L'utilisateur se connecte à son compte sur la plateforme de réservation.

Recherche de train : Il entre une destination et une date, puis effectue une recherche pour trouver un billet.

Sélection du billet : Il choisit un billet disponible et l'ajoute à son panier.

Paiement : Il entre ses informations de paiement et confirme l'achat du billet.

Confirmation : L'utilisateur reçoit une confirmation à l'écran, ainsi qu'un email récapitulatif de la réservation.

Validation des critères : À chaque étape, on vérifie si le système fonctionne correctement et répond aux exigences : l'utilisateur peut-il se connecter ? Le billet s'ajoute-t-il au panier ? Le paiement est-il confirmé ? L'email de confirmation est-il envoyé ?

Exemple Côté code

```
// Arrange
BookingPage = new BookingPage()
PaymentPage = new PaymentPage()
EmailSystem = new EmailSystem()

// Act
BookingPage.searchTrain('Paris', 'Lyon', '2024-12-01')
BookingPage.selectTicket('First Class')
PaymentPage.enterPaymentDetails('Visa', '4111 1111 1111 1111', '12/25', '123')
PaymentPage.confirmPurchase()

// Assert
assert(BookingPage.isBookingConfirmed() == true)
assert(EmailSystem.hasReceivedEmail('user@example.com', 'Booking Confirmation') == true)
```

Outils utilisés pour les tests d'acceptation

Jira avec des plugins comme **Zephyr** ou **Xray** pour gérer les tests d'acceptation et suivre les critères d'acceptation dans les user stories.

TestRail pour documenter et suivre les résultats des tests d'acceptation.

Cucumber ou **Behave** pour les tests d'acceptation basés sur des spécifications écrites en langage naturel.

Selenium ou **Cypress, PlayWright** peuvent également être utilisés pour automatiser des scénarios d'acceptation, surtout s'ils impliquent des interfaces utilisateurs.

Avantages des tests d'acceptation

Validation finale avant mise en production : Ils garantissent que le produit répond bien aux besoins des utilisateurs avant d'être déployé.

Confiance accrue des utilisateurs : En impliquant les utilisateurs finaux dans le processus de test, on s'assure qu'ils acceptent et adoptent l'application une fois mise en production.

Identification des écarts fonctionnels : Ils permettent de repérer des écarts entre les attentes métiers et la solution fournie, évitant des déploiements prématurés.

Amélioration de la qualité du produit : En testant dans des conditions proches du réel, ils assurent que le produit livré est de haute qualité et fonctionnel.

Focus sur l'expérience utilisateur : Ces tests valident non seulement les fonctionnalités, mais aussi l'expérience utilisateur dans son ensemble.

Limites des tests d'acceptation

Dépendance des utilisateurs : Leur efficacité repose sur l'implication des utilisateurs finaux, ce qui peut poser problème si ces derniers sont indisponibles ou manquent de temps.

Nécessitent des critères clairs : Les critères d'acceptation doivent être bien définis en amont. Si les attentes sont floues, cela peut rendre difficile la validation des tests d'acceptation.

Pas toujours exhaustifs : Les tests d'acceptation se concentrent généralement sur des scénarios clés, et peuvent ne pas couvrir tous les cas d'utilisation possibles.

Difficiles à automatiser : Bien que certaines parties puissent être automatisées, les tests d'acceptation impliquant des retours utilisateur réels sont souvent manuels, ce qui peut être chronophage.

Comparaison avec les autres types de tests

Tests unitaires : Testent des fonctions ou méthodes spécifiques, isolées du reste du système.

Tests d'intégration : Vérifient l'interaction entre plusieurs modules du système.

Tests fonctionnels : Valident des fonctionnalités spécifiques du produit, sans nécessairement valider l'application dans son ensemble.

Tests End-to-End : Simulent un flux complet à travers l'application, mais ne se concentrent pas nécessairement sur les besoins métiers spécifiques.

Tests d'acceptation : Portent sur la validation finale du produit en fonction des exigences et critères d'acceptation définis par les utilisateurs ou parties prenantes.

Test De Regression

Les tests de régression sont une forme de tests logiciels utilisés pour vérifier que les modifications récentes du code (corrections de bogues, ajouts de fonctionnalités ou refactorisations) n'ont pas introduit de nouveaux problèmes ou régressions dans des fonctionnalités qui fonctionnaient correctement auparavant.

L'objectif est de s'assurer que les changements n'ont pas affecté négativement les parties existantes du système.

Les tests de régression consistent à réexécuter une suite de tests déjà effectués sur le logiciel après chaque modification pour vérifier que les nouvelles fonctionnalités ou corrections de bogues n'ont pas perturbé le comportement des fonctionnalités précédemment validées. Ils peuvent être manuels, mais sont généralement automatisés pour maximiser leur efficacité et réduire le coût associé à leur exécution fréquente, surtout dans les environnements de développement agile ou intégration continue (CI).

Objectifs des tests de régression

- Détecter les régressions : L'objectif principal est de s'assurer qu'aucune fonctionnalité existante n'a été cassée ou perturbée par les nouvelles modifications apportées au code.
- Préserver la qualité globale du produit : Les tests de régression permettent de vérifier que les fonctionnalités principales et critiques du système continuent de fonctionner correctement après des mises à jour
- Valider la stabilité après chaque changement : Ils garantissent que le système reste stable malgré les changements fréquents (ajout de nouvelles fonctionnalités, corrections de bogues, mises à jour de sécurité, refactorisations, etc.).
- Assurer une couverture de tests étendue : Ils permettent de vérifier non seulement les changements récents mais aussi de retester les aspects critiques ou à haut risque du système.
- Automatisation pour efficacité : Les tests de régression sont généralement automatisés pour être réexécutés fréquemment et de manière fiable, ce qui permet d'économiser du temps et des ressources, tout en assurant une validation continue.

Types de tests de régression

Régression partielle : Ce type de régression teste les parties du système affectées directement par les modifications récentes, sans nécessairement couvrir l'ensemble de l'application.
Régression complète : Ici, tous les tests fonctionnels de l'application sont réexécutés, même ceux qui ne sont pas directement affectés par les modifications récentes, pour s'assurer que tout fonctionne comme prévu.
Tests de régression sélectifs : Ces tests se concentrent uniquement sur les parties de l'application qui sont jugées à haut risque ou critiques, ou qui sont potentiellement impactées par les changements.
Régression corrective : Ce type de test est utilisé après qu'un bogue a été corrigé, pour vérifier que la correction ne provoque pas de nouveaux problèmes dans les fonctionnalités connexes ou adjacentes.
Régression progressive : Il s'agit de tests de régression effectués lorsqu'une nouvelle fonctionnalité est ajoutée. Ils testent si cette nouvelle fonctionnalité interagit correctement avec les parties existantes de l'application.

Caractéristiques des tests de régression

Test récurrent après chaque modification : Les tests de régression sont effectués régulièrement après chaque modification importante du code (ajout de fonctionnalités, corrections de bogues, refactorisation).

Focus sur les fonctionnalités existantes : Ils se concentrent sur la validation des fonctionnalités déjà existantes plutôt que sur les nouvelles fonctionnalités.

Évolution des suites de tests : La suite de tests de régression évolue en fonction des nouvelles fonctionnalités et des tests ajoutés au fil du temps, afin de garantir une couverture complète du produit.

Exécution automatisée : La plupart des tests de régression sont automatisés pour permettre des exécutions fréquentes sans intervention manuelle, en particulier dans les environnements CI/CD.

Large couverture : Les tests de régression couvrent souvent une grande partie de l'application, même au-delà des zones où des modifications ont été apportées, pour identifier toute répercussion non prévue.

Exemple de scénarios de tests de régression

une application de commerce électronique. Après avoir corrigé un problème avec le calcul des taxes, il est important de vérifier que cette correction n'a pas perturbé d'autres fonctionnalités comme l'ajout d'articles au panier, le paiement ou la gestion des stocks. Un scénario de test de régression pourrait inclure :

Ajouter un article au panier : Vérifier que l'utilisateur peut toujours ajouter des articles au panier après la correction.

Calcul des taxes : Confirmer que le calcul des taxes fonctionne correctement après la correction.

Paiement : S'assurer que le paiement peut encore être effectué correctement après la mise à jour.

Mise à jour de l'inventaire : Vérifier que l'inventaire est mis à jour correctement après la commande.

Vérification des fonctionnalités critiques : S'assurer que d'autres fonctionnalités critiques comme la gestion des comptes utilisateurs ou la génération de rapports n'ont pas été affectées.

Exemple code

```
// Arrange
ProductPage = new ProductPage()
CartPage = new CartPage()
CheckoutPage = new CheckoutPage()

// Act
ProductPage.addToCart('Laptop')
CartPage.applyDiscountCode('SUMMER21')
CheckoutPage.enterPaymentDetails('Visa', '4111 1111 1111 1111', '12/25', '123')
CheckoutPage.confirmPurchase()

// Assert
assert(CartPage.totalPriceIncludesTaxes() == true)
assert(CheckoutPage.isPurchaseConfirmed() == true)
```

Avantages des tests de régression

Détection précoce des erreurs : Ils permettent de détecter rapidement les régressions avant qu'elles ne soient livrées en production.

Automatisation : Les tests de régression sont facilement automatisables, ce qui permet de les exécuter fréquemment sans intervention manuelle, augmentant ainsi l'efficacité du processus de test.

Amélioration continue de la qualité : En réexécutant régulièrement les tests de régression, on peut s'assurer que les nouvelles modifications n'affectent pas la qualité du produit dans son ensemble.

Stabilité du produit : Ils contribuent à maintenir la stabilité du produit au fur et à mesure des ajouts ou des modifications de fonctionnalités.

Intégration continue : Les tests de régression s’intègrent parfaitement dans les processus d’intégration continue et de livraison continue, garantissant que chaque changement de code est immédiatement validé.

Limites des tests de régression

Coût en temps et en ressources : Bien que les tests de régression soient automatisés, leur exécution peut prendre du temps, surtout lorsque la suite de tests est volumineuse.

Maintenance des tests : La suite de tests de régression doit être régulièrement mise à jour pour inclure de nouveaux scénarios ou des modifications apportées au système, ce qui peut devenir chronophage.

Faux négatifs : Des échecs de tests de régression peuvent parfois être liés à des changements dans l'environnement de test ou à des facteurs externes, ce qui peut entraîner des résultats faussement négatifs.

Difficulté à tout couvrir : Malgré une suite de tests étendue, il peut être difficile de couvrir chaque scénario possible, surtout dans des applications très complexes.

Comparaison avec les autres types de tests

Tests unitaires : Ils se concentrent sur des parties spécifiques du code, généralement des fonctions ou des méthodes, et ne vérifient pas les interactions globales comme les tests de régression.

Tests d'intégration : Ils testent l'interaction entre plusieurs composants du système, mais ne valident pas nécessairement que les fonctionnalités existantes fonctionnent après des modifications.

Tests fonctionnels : Ils valident que les fonctionnalités du produit respectent les spécifications, mais les tests de régression vérifient en permanence que ces fonctionnalités continuent de fonctionner après chaque mise à jour.

Tests End-to-End : Ils couvrent des scénarios d'utilisation complets, mais les tests de régression sont plus souvent automatisés et exécutés à chaque changement.

Tests d'acceptation : Les tests d'acceptation valident si le produit répond aux exigences définies par l'utilisateur, tandis que les tests de régression se concentrent sur la stabilité continue du système après des changements.

Test de performance

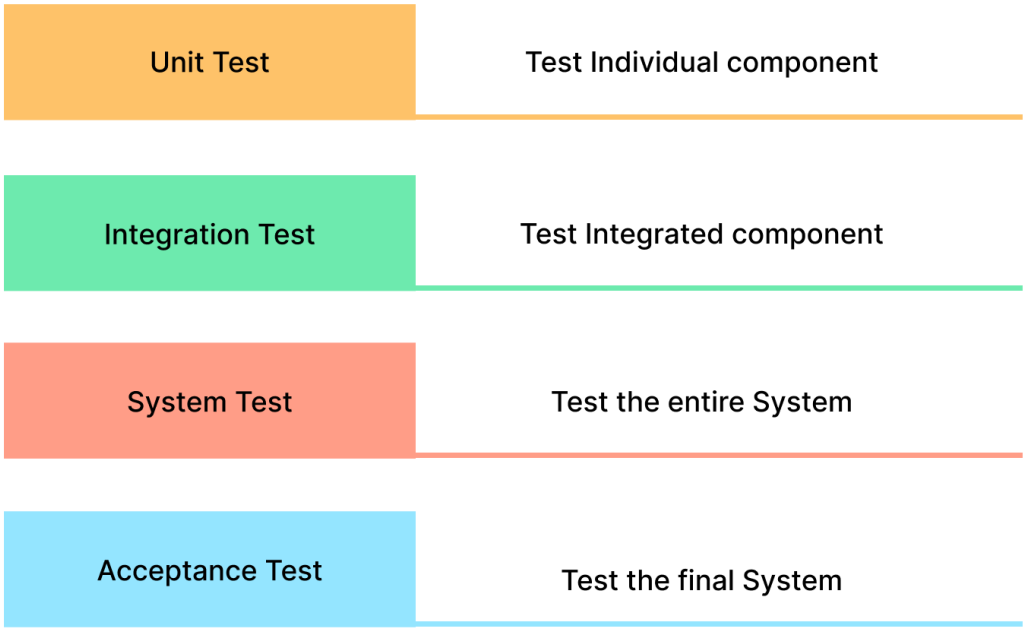
Les tests de performance évaluent les performances d'un système sous une charge de travail spécifique. Ces tests permettent de mesurer la fiabilité, la vitesse, l'évolutivité et la réactivité d'une application. Par exemple, un test de performance peut observer les temps de réponse lors de l'exécution d'un nombre important de demandes ou déterminer le comportement du système face à une quantité élevée de données. Il peut déterminer si une application répond aux exigences de performances, localiser les goulots d'étranglement, mesurer la stabilité pendant les pics de trafic, et plus encore.

Smoke Test

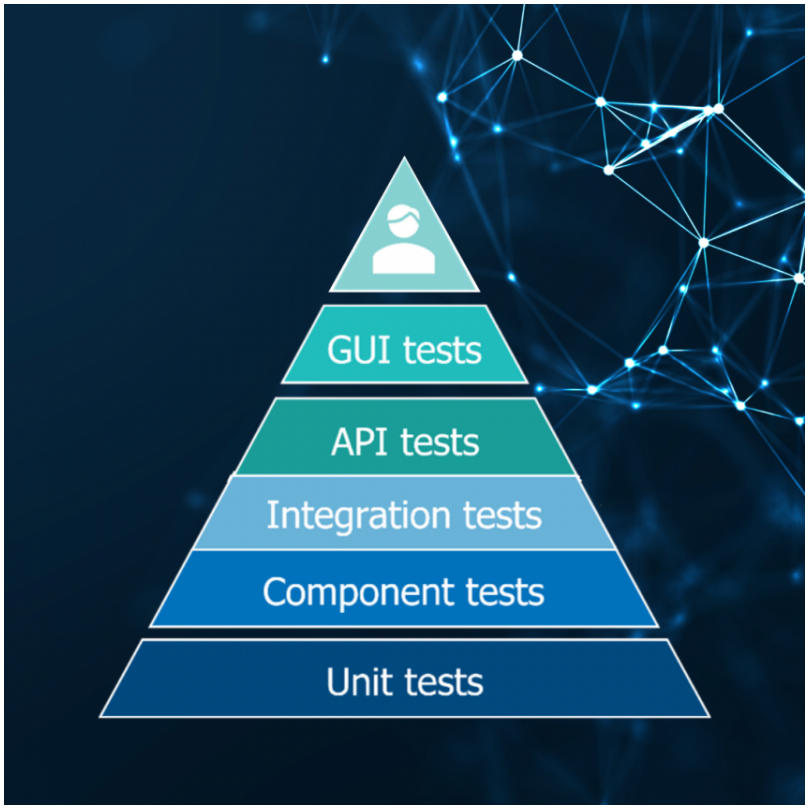
Les smoke tests sont des tests simples qui vérifient le fonctionnement de base d'une application. Ils sont conçus pour être rapides à exécuter, et leur but est de vous donner l'assurance que les caractéristiques principales de votre système fonctionnent comme prévu.
Les « smoke tests » peuvent être utiles juste après la création d'un build afin de décider si vous pouvez exécuter des tests plus coûteux. Ils peuvent également être utiles après un déploiement afin de vous assurer que l'application s'exécute correctement dans l'environnement nouvellement déployé.

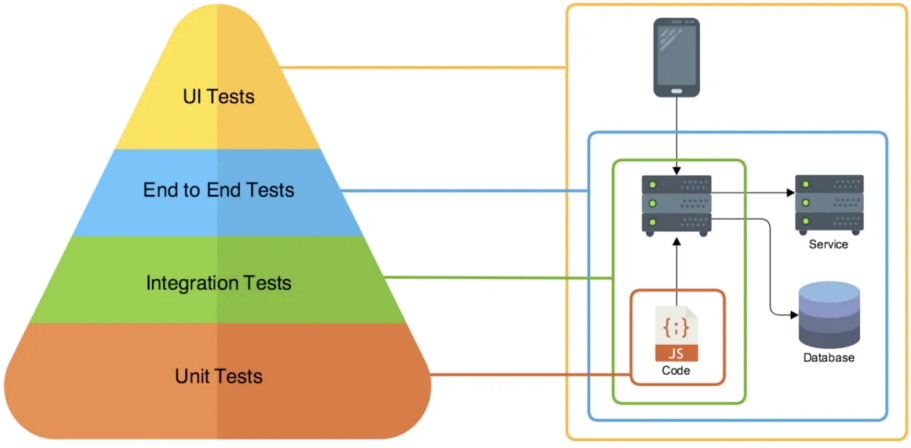
https://www.atlassian.com/fr/continuous-delivery/software-testing/types-of-software-testing

Infographie (non exhaustive qui représente les différents niveaux de test)



Autre infographie plutôt orienté sur des test d'API ou GUI (Graphical User Interface)





Pratique

Analyse des besoins de test

En utilisant les connaissances acquises lors de l'introduction au plan de test et en s'inspirant du template de plan de test, analysez les besoins de test d'un projet logiciel donné. Identifiez les fonctionnalités clés qui doivent être testées ainsi que les risques potentiels. Présentez vos recommandations pour la suite du processus de planification du test.

Création des cas de test

À partir du document de spécification des exigences d'un projet logiciel, créez des cas de test détaillés. Assurez-vous de couvrir tous les scénarios possibles et d'inclure des cas de test positifs et négatifs. Fournissez une description claire des conditions préalables, des étapes de test et des critères de succès pour chaque cas de test.

Exécution des cas de test

Sélectionnez un ensemble de cas de test du plan de test que vous avez élaboré et exécutez-les sur une application logicielle de votre choix. Documentez les résultats de chaque cas de test, en indiquant s'il a réussi ou échoué. Analysez les résultats pour identifier les problèmes soulevés par les tests et proposez des actions correctives.

Quizz

🤔
https://forms.gle/uuiF9ec6BUihQaUV6

TOOLBOX

Javascript :

Jest
https://jestjs.io/fr/

Mocha
https://mochajs.org/

Jasmine :
https://jasmine.github.io/

AVA :
https://github.com/avajs/ava

Karma :
https://karma-runner.github.io/3.0/

Cypress :
https://www.cypress.io/

Pupeteer :
https://github.com/puppeteer/puppeteer

Sinon :
https://sinonjs.org/

PHP

Php Unit :
https://phpunit.de/

CodeCeption :
https://codeception.com/

StoryPlayer :
https://datasift.github.io/storyplayer/

Selenium :
https://www.lambdatest.com/selenium

Behat :
https://docs.behat.org/en/latest/

Atoum :
https://atoum.org/