Les tests

Règle 1 : Eviter le principe du simple aveugle
(ex. médicament-placebo : le sujet ne connait pas l'objet du test ni les résultats attendus)

Règle 2 : Eviter le principe du double aveugle
(ex. médicament-placebo : le sujet ne connait pas l'objet du test ni les résultats attendus & le correcteur/analyste ne connait ni les hypothèses ni l'identité des sujets)

En informatique, un **test** (anglicisme) désigne une procédure de vérification partielle d'un système informatique. Le but est de s'assurer que le système réagit de la façon prévue par ses concepteurs ou est conforme aux attentes du client l'ayant commandé.
Externaliser une partie du cycle des tests fonctionnels présente des avantages significatifs :

- Les développeurs ne sont pas toujours les meilleurs testeurs.
- Les testeurs extérieurs s'identifient à l'utilisateur final et ne sont pas influencés par leur connaissance du produit.
- L'indépendance du contrôle qualité est gage de transparence et crédibilise le processus Qualité.

1. Qualité du test

Les phases de test dans le cycle de développement d'un produit logiciel permettent d'assurer un niveau défini de qualité en accord avec le client. Une procédure de test peut donc être plus ou moins fine, et par conséquent l'effort de test plus ou moins important et coûteux selon le niveau de qualité requis.

2. Couverture des tests

En dehors du cas très particulier de systèmes extrêmement simples, il est impossible de tester exhaustivement un logiciel. La réussite des tests ne permet donc pas de conclure au bon fonctionnement du logiciel. On essaye cependant, heuristiquement, de faire en sorte que si un défaut (*bog*) est présent, le test le mette en évidence, notamment en exigeant une bonne couverture des tests :

- Couverture en points de programme : chaque point de programme doit avoir été testé au moins une fois
- Couverture en chemins de programme : chaque séquence de points de programme possible dans une exécution doit avoir été testée au moins une fois (impossible en général).
- Couverture fonctionnelle : chaque fonctionnalité métier de l'application doit être vérifiée par au moins un cas de test.

Si l'on veut des assurances plus fortes de bon fonctionnement, on peut utiliser des méthodes formelles.

3. Testabilité

Un test est un ensemble de cas à tester (état de l'objet à tester avant exécution du test (ex. **application XXX lancée**), valeurs en entrée (**login X, pswd A**), valeurs attendues/prédites (ex. **login inconnu**) et état de l'objet après exécution), éventuellement accompagnés de leur procédures d'exécution (séquences d'actions à exécuter). Cet ensemble est dérivé de l'objectif du test.
Un test vise à mettre en évidence les défauts de l'objet testé, c'est-à-dire les écarts de l'objet par rapport à une spécification (au sens large) mesurés par les écarts entre valeurs en entrée et valeurs attendues. Cependant *il n'a pas pour* objectif :

- de diagnostiquer la cause des erreurs,
- de les corriger,
- de prouver la correction de l'objet testé.

Un objet ne peut être testé que si on peut déterminer les valeurs attendues en fonction des valeurs soumises en entrée. Si la spécification ne permet pas cela, la propriété du logiciel qu'elle définit ne peut être testée. Dans un tel cas, l'analyste de test devrait pouvoir enregistrer un défaut de spécification dans le système de gestion des défauts (*bugs*).
Exiger que les valeurs attendues soient fournies permet d'améliorer la qualité de la spécification puisqu'un plus grand nombre de questions pertinentes aura été posé par son auteur. L'importance de ce concept de « testabilité » est soulignée dans certaines méthodes de développement.
Le processus de test d'un logiciel utilise différents types et techniques de test. Il tend à vérifier que le logiciel est conforme à son cahier des charges (vérification du produit) et aux attentes du client (validation du produit).
Il existe différentes façons de classer les tests informatiques. Nous listons ci-dessous différentes vues des tests.

4. Les familles de tests

4.1. Test boîte noire, test boîte blanche, test boîte grise

4.1.1. **Test de type boîte noire** (black box)

…fonctionnel ou non, n'est pas fondé sur l'analyse de la structure interne du composant ou du système.
Une boîte noire désigne un dispositif, un objet ou un système considéré seulement des points de vue des caractéristiques de ses entrée-sorties (données – résultats). En général, on parle d'une boîte noire quand on ne peut pas voir les fonctionnements internes d'un système.

- Un programme compilé qui n'est pas Open Source.
- Un programme qui est exécuté à distance sur un réseau informatique.

Dire qu'un système est *considéré comme une boîte noire* signifie qu'il a les propriétés qu'il est censé avoir. De façon informelle, ça signifie qu'on lui fait confiance, par exemple, pour qu'il ne contienne pas de porte dérobée.
Dans les tests résultants d'une technique de conception de test de type boite noire, les données en entrée et le résultat attendu sont sélectionnés non pas en fonction de la structure interne mais de la définition de l'objet. Ces tests sont le plus fréquent parmi les tests fonctionnels d'intégration et de recette, mais rien n'empêche d'utiliser ces techniques de conception pour définir des tests unitaires.

4.1.2. Test de type boîte blanche (white box)

…ou test structurel : en général fonctionnel, analyse la structure interne du composant ou du système.
Une boîte blanche est un module d'un système dont on peut prévoir le fonctionnement interne car on connait les caractéristiques de fonctionnement de l'ensemble des éléments qui le compose. Autrement dit une boite blanche est un module qui comporte aussi peu de boîte noire que possible. On qualifie les systèmes mixtes de « boîte grises ».
Les tests résultants d'une technique de conception de type boîte blanche vérifient la structure interne de l'objet, par exemple l'exécution des branches des instructions conditionnelles. Les tests unitaires sont souvent spécifiés à l'aide de telles techniques.

4.1.3. Test boîte grise

Les tests en « boîte grise » compilent ces deux précédentes approches : ils éprouvent à la fois les fonctionnalités et le fonctionnement d'un système. C'est-à-dire qu'un testeur va par exemple donner une entrée (input) à un système, vérifier que la sortie obtenue est celle attendue, et vérifier par quel processus ce résultat a été obtenu.
Dans ce type de tests, le testeur connaît le rôle du système et de ses fonctionnalités, et a également une connaissance, bien que relativement limitée, de ses mécanismes internes (en particulier la structure des données internes et les algorithmes utilisés). Attention cependant, il n'a pas accès au code source !

4.2. Test unitaire

Le **test unitaire** est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité »).
Il s'agit pour le programmeur de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances. Cette vérification est considérée comme essentielle, en particulier dans les applications critiques. Elle s'accompagne couramment d'une vérification de la couverture du code, qui consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester.
L'ensemble des tests unitaires doit être rejoué après une modification du code afin de vérifier qu'il n'y a pas de régressions (apparition de nouveaux dysfonctionnements).
Dans les applications non critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, la méthode eXtreme Programming (XP) a remis les tests unitaires, appelés « tests du programmeur », au centre de l'activité de programmation.
La méthode XP préconise d'écrire les tests en même temps, ou même avant la fonction à tester (Test Driven Developpment). Ceci permet de définir précisément l'interface du module à développer. En cas de découverte d'un défaut, on écrit la procédure de test qui reproduit le défaut. Après correction on relance le test, qui ne doit indiquer aucune erreur.

4.3. Test d'intégration

Un **test d'intégration** est un test qui se déroule après les tests unitaires. Il consiste, une fois que les développeurs ont chacun validé leurs développements ou leurs correctifs, à regrouper leurs modifications ensemble dans le cadre d'une livraison.
L'intégration a pour but de valider le fait que toutes les parties développées indépendamment fonctionnent bien ensemble.

Il s'agit d'établir une nouvelle version, basée soit sur une version de maintenance, soit sur une version de développement. L'intégration fait appel en général à un système de gestion de versions, et éventuellement à des programmes d'installation.
Pour les applications utilisant les nouvelles technologies et donc des ateliers de génie logiciel (Eclipse - Visual Studio - JBuilder - JDeveloper ...), les tests d'intégration ont évolué vers de l'intégration continue.
L'intégration continue est la fusion des tests unitaires et des tests d'intégration car le programmeur détient toute l'application sur son poste et peut donc faire de l'intégration tout au long de son développement.

4.4. Test de validation

Le **test de validation** permet de vérifier si toutes les exigences client décrites dans le document de spécification d'un logiciel, écrit à partir de la spécification des besoins, sont respectées. Les tests de validation se décomposent généralement en plusieurs phases:

- Validation fonctionnelle Les tests fonctionnels vérifient que les différents modules ou composants implémentent correctement les exigences client. Ces tests peuvent être de type valide, invalide, inopportuns, etc...
- Validation solution Les tests solutions vérifient les exigences clients d'un point de vue "use cases", généralement ces tests sont des tests en volumes. Chaque grand use-case est validé un par un, puis tous ensemble. L'intérêt est de valider la stabilité d'une solution par rapport aux différents modules qui la composent, en soumettant cette solution à un ensemble d'actions représentatif de ce qui sera fait en production.
- Validation performance, robustesse Les tests de performance vont vérifier la conformité de la solution par rapport à ses exigences de performance, alors que les tests de robustesse vont essayer de mettre en évidence des éventuels problèmes de stabilité dans le temps (fuite mémoire par exemple, résistance au pic de charge)

Ces phases de validation fonctionnelles peuvent être complétées par une phase de validation plus technique consacrée aux tests de haute disponibilité (tests cluster ou autres) et scalabilité verticale ou horizontale.

4.5. Test de performance

Un **test de performance** ou **benchmark** est un test dont l'objectif est de déterminer la performance du logiciel.
L'acception la plus courante de ce terme est celle dans laquelle ces tests vont avoir pour objectif de mesurer les temps de réponse d'un système en fonction de sa sollicitation. Cette définition est très proche de celle de test de charge où l'on mesure le comportement d'un système en fonction de la charge d'utilisateurs simultanés.
Ces tests peuvent être de plusieurs types, notamment :

- Test de tenue en charge : il s'agit d'un test au cours duquel on va simuler une charge importante d'utilisateurs sur une durée relativement longue, pour voir si le système testé est capable de supporter une activité intense sur une longue période sans dégradations des performances et des ressources applicatives ou système. Des synonymes courants sont test d'endurance, de robustesse, de fiabilité.
- Test de capacité : il s'agit d'un test au cours duquel on va simuler un nombre d'utilisateurs sans cesse croissant (par paliers) de manière à déterminer quelle charge limite le système est capable de supporter.
- Test en stress : il s'agit d'un test au cours duquel on va simuler l'activité maximale attendue en heures de pointe de l'application, pour voir comment le système réagit au maximum de l'activité des utilisateurs.

- Test aux limites : il s'agit d'un test au cours duquel on va simuler une activité bien supérieure à l'activité normale, pour voir comment le système réagit aux limites du modèle d'usage de l'application.
- Test d'endurance : il s'agit d'un test qui établit la linéarité du fonctionnement de l'application sur une durée égale à une journée type d'utilisation (10h voire 24h), cela permet de mesurer une dérive des performances due à des fuites mémoires ou des saturations disques de l'application

Il existe d'autre types de tests, plus ciblés et fonction des objectifs à atteindre dans la campagne de tests : Test de Benchmark (comparaisons de logiciel, matériels, architectures,...), Tests de non-régression, Tests de Composants, Tests de Volumétrie des données, etc.

4.6. Test de non-régression

Le **test de non régression** est l'action de dérouler des plans de test pour chaque nouvelle version ou incrément d'un logiciel. Ce test confirme que les nouvelles fonctionnalités n'ont pas endommagé, de manière directe ou indirecte, celles déjà existantes et que les récentes modifications n'ont pas entraîné de régression. L'intérêt principal de ces tests est de limiter les anomalies relevées lors de la recette de l'application. Ils viennent compléter les tests unitaires et les tests d'intégration en amont des tests de recette.
Le test de non régression s'inscrit dans la durée. Lorsqu'un test a confirmé la qualité du logiciel, il est nécessaire de la lancer régulièrement afin de vérifier que les modifications apportées au fil du temps n'ont pas eu d'impact négatif sur les fonctions déjà existantes. Un changement même anodin peut avoir des effets de bord. Ce type de test permet d'avoir une bonne indication sur l'évolution de la qualité du logiciel

5. Automatisation de Test

Activité qui consiste à utiliser d'autres outils logiciels pour tester un logiciel avec le minimum d'intervention humaine.

Exécuter des plans de tests à chaque nouvelle version d'un logiciel s'avère être une tâche longue, coûteuse et répétitive. De nombreuses solutions logicielles permettent de dérouler, en peu de temps, la majeure partie d'une campagne de tests. A chaque nouvelle version du logiciel l'automatisation de test va permettre de détecter les défauts sur des zones couvertes ainsi que les effets de bord.

Le challenge de la mise en place de tests automatisés concerne surtout le choix des zones à couvrir par des tests automatiques et le bon compromis investissement de départ - temps d'exécution de tests. Il est clair que toutes les parties d'un logiciel ne sont pas automatisables, certaines parties sont trop changeantes ou trop mal maîtrisées ou mûres pour être automatisées : le coût de la maintenance des stratégies d'automatisation peut s'avérer supérieur au temps passé à mener manuellement les tests. En outre, certains résultats ne peuvent se vérifier sans l'appréciation d'un regard humain, même quand les scripts ont été conçus avec la plus grande rigueur.

Il s'agit de déterminer le juste cadre de l'automatisation des tests sans tomber dans "l'automatisation à tout prix".

La tendance est donc d'automatiser ces tâches répétitives et fastidieuses.

De grands projets obligent à réaliser de nombreux tests de non-régression. Les automates apportent alors une aide importante. On peut citer : X-runner de la société Mercury avec le logiciel « TestDirector » pour la gestion de plans, de campagnes et de base de jeux d'essais. On peut également citer « QA/Plan » de INTCOM pour l'automatisation de tâches de saisie.

Entre autres suggestions, l'Extrem Programming propose d'écrire un programme qui testera le code avant d'écrire celui-ci. Cette suggestion est conforme à la culture américaine, donner des objectifs, atteindre ceux-ci en ne se préoccupant que des objectifs (formulés). La culture française favorise une vision d'ensemble, la méthode et « l'honneur » de bien faire son travail (D'Iribarne, la culture de l'honneur, Seuil)

5.1. Environnements de test unitaires : xUnit

Le terme générique « xUnit » désigne un outil permettant de réaliser des tests unitaires dans un langage donné (dont l'initiale remplace « x » le plus souvent). On trouve : Aunit pour Ada, JUnit et TestNG pour Java, DUnit pour Delphi, FLEXunit pour AdobeFlex, PHPUnit et SimpleTest pour PHP, NUnit pour .NET, NUnitASP pour ASP.NET, unittest pour Python, Test::Unit pour Ruby, ASUnit pour ActionScript, JSUnit pour favaScript et cppunit pour C++, OCunit pour ObjectiveC...

5.2. Logiciels pour les tests de non régression

Les tests de non-régression sont fastidieux car ils doivent être le plus exhaustif possible, afin de s'assurer que le logiciel fonctionne de la même manière. Il existe donc des programmes informatiques spécialisés qui permettent d'automatiser ces tests. Ces programmes appelés souvent robots de tests, simulent généralement l'activité d'un utilisateur (il joue un scénario prédéfini) et contrôlent que le résultat obtenu est conforme au résultat donné par la version antérieure du logiciel.

Même si les tests de non-régression ne sont pas une nouveauté, la méthode extreme programmaing en fait un de ses chevaux de bataille pour améliorer la qualité du logiciel.

Comme logiciel vous pouvez voir Visual Studio Team Tester Edition.

5.3. Outils pour test de performance

Quand il s'agit de simuler un nombre d'utilisateurs en général important, il est évidemment nécessaire d'automatiser ces tests.
Une plate-forme de test de performances va généralement comporter :

- Un injecteur/Moteur de charge : logiciel qui va être capable de simuler les actions des utilisateurs et de générer la charge. Ces actions sont définies dans des scripts/scénarii.
- Des sondes : placées au niveau du système cible, elles permettent de remonter des mesures dont l'objet est de savoir comment réagissent individuellement des composantes du système. Les éléments sur lesquels on va en général placer des sondes sont l'utilisation de la mémoire, processeur, disque et réseau.

Les solutions de tests de performances Web vont permettre de simplifier et d'automatiser les tests : création plus ou moins automatisée des scénarii de tests, configuration des scénarii avec ou sans script, simulation d'utilisateurs virtuels avec collecte des mesures (et génération automatique de rapports), etc.

Recette informatique

La **recette** est une des phases de développement de projet informatique, phase au cours de laquelle les différents acteurs du projet se rencontrent afin de vérifier que le produit est conforme aux attentes formulées.
La procédure de recette se déroule en 2 étapes principales :

- la Recette Usine
- la Recette Utilisateur

Si la première étape a lieu chez le fournisseur, la deuxième en revanche se déroule dans les locaux du client.

1. La recette usine

La **Recette Usine** comprend tous les tests réalisés chez le fournisseur, avant la livraison. Elle correspond à une période de quelques jours - en général 10% d'une campagne de recette, elle permet de valider chez le fournisseur que le produit est de qualité et que l'installation peut s'effectuer chez le client, sur ses plateformes, pour réaliser des tests de Recette Utilisateur.
A l'issue de la Recette Usine, le fournisseur et le client signent un procès-verbal de fin de recette usine, qui accompagne la livraison (ou pas !) du produit et le cahier de recette.

2. La recette utilisateur

Lors de la **Recette Utilisateur** également appelée Vérification d'Aptitude au Bon Fonctionnement (VABF), le client réalise deux catégories de tests différentes :

- la Recette Fonctionnelle et
- la Recette Technique

2.1. La recette fonctionnelle

La **Recette Fonctionnelle** a pour but de valider les fonctionnalités exprimées dans le cahier des charges et détaillées dans les spécifications fonctionnelles. La MOA procède donc à sa propre série de tests de validation.

2.2. La recette technique

La **Recette Technique**, chargée de contrôler les caractéristiques techniques du produit livré, regroupe les principaux tests suivants :

- les tests d'exploitabilité et en particulier le respect des exigences d'architecture technique
- et les tests de performance.

Si la VABF se déroule correctement et est validée, le client procède alors à la mise en service opérationnel. Une période de **Vérification de Service Régulier (VSR)** commence donc par un premier déploiement sur un site pilote. Cette mise en production permet de tester le produit en conditions réelles.

2.3. Les documents livrables

Plusieurs documents accompagnent la procédure de recette.

2.3.1. Le protocole de recette

Le protocole de recette est un document visant à clarifier intégralement la procédure de recette. Il précise scrupuleusement :

- les tâches du client,
- celles du fournisseur,
- la liste des documents à communiquer,
- l'ordre des tests et le planning,
- les seuils d'acceptation du produit.

2.3.2. Le cahier de recette

Le cahier de recette est la liste exhaustive de tous les tests pratiqués par le fournisseur avant la livraison du produit.

La couverture des tests, en particulier ceux de non régression lorsqu'il s'agit d'une nouvelle version d'un produit existant, pouvant être infinie, le cahier de recette doit préciser toutes les fiches de test passées par le fournisseur, ainsi que celles à passer dans l'environnement du client lors de la VABF.

2.3.3. Les procès-verbaux

Pour clore chaque étape de la procédure de recette, un procès-verbal est rédigé. Celui-ci a pour objet de prononcer la réception et de mentionner les réserves émises par chacune des parties. Toutes ces recettes peuvent être contractuelles et donner lieu à un procès-verbal de recette qui permet de prononcer la réception, assortie ou non de réserves.

Ainsi le procès-verbal de recette usine conditionne le démarrage de la période de VABF, et celui de VABF conditionne le démarrage de la VSR.

3. Méthodologie

Lorsqu'un logiciel est régulièrement soumis à des tests, il est judicieux de capitaliser la procédure de test. Un plan de test permet de formaliser sur des documents (sous le format que souhaite le client), la procédure à suivre pour le test. C'est une suite de vecteurs de test qui va confirmer à l'expert que toute l'application est bien testée et particulièrement les parties à risques. Chaque vecteur de test est détaillé de la manière suivante :

- Préconditions
- Les manipulations à faire.
- Les résultats attendus pour chaque manipulation
- Le contrôle des résultats de chaque manipulation
- Diverses informations servant à contrôler la qualité du module testé

L'exécution de tests consiste à dérouler des plans / stratégies de tests sur un logiciel et/ou un ensemble applicatif et d'en reporter les résultats.

Le plan de tests est l'expression du besoin de la campagne de tests. On y trouve la présentation du projet (résumé, architecture technique et logicielle), les objectifs, le modèle de charge, le type de tests à réaliser, les scénarios fonctionnels (ou cas d'utilisation) à tester accompagnés des jeux de données nécessaires et un planning d'exécution des tests.

Les jeux de données vont permettre de simuler au plus juste la réalité. Un jeu de données peut par exemple consister en n logins et n mots de passe, permettant ainsi de simuler des utilisateurs différents se connectant à l'application.

Après la conception des plans de tests, l'expert en test va exécuter les plans de test afin d'identifier les dysfonctionnements d'un logiciel et/ou d'une application. Son expérience et son objectivité vont lui permettre de se concentrer plus particulièrement sur des facteurs importants comme le niveau de risque et l'environnement.

Le résultat de la mission sera consigné dans le plan de test et les anomalies seront reportées dans le système de gestion de défaut (bog) mis en place.

Ce type de mission peut avoir lieu à l'occasion de tests fonctionnels, mais peut aussi concerner des tests manuels unitaires, des tests de non régression ainsi que des tests de validation et de recette.

Conseils :

- *Testez de façon large, puis de façon approfondie*
- *Testez dans un environnement contrôlé*
- *L'environnement de test doit être si possible identique à l'environnement de production*

Techniques de mise en œuvre de tests

La préparation d'une recette amène à préciser certains points tels qu'objectifs, stratégies et campagnes.

- Chaque objectif de test concerne soit une fonction, soit un type d'objet (paquetage) soit un type de performance…
- Une campagne de test est l'ensemble des opérations visant à atteindre les objectifs.
- Une stratégie est une manière d'ordonner les tests dans une campagne.

Typiquement, dans une stratégie, on vérifie dans un premier temps le bon fonctionnement des mouvements réputés bons, puis dans un deuxième temps, le bon fonctionnement / contrôle des mouvements réputés mauvais ou erronés.

A minima, même un test unitaire comporte une stratégie (un ordre des tests à réaliser). Par exemple : vérification de l'ouverture des fichiers, de la lecture, de la fermeture, de l'écriture des mouvements en sortie, puis des contrôles, de la cinématique, des doublons, des calculs, de l'affichage.

Selon l'application et le contexte du projet, la stratégie à mettre en œuvre pour ordonner les tests à réaliser peut résulter de l'application de l'une des stratégies de références listées ci-dessous ou de la combinaison de celles-ci. Toutefois, un plan usuel commence par tester -1- les aspects techniques (interfaces cf. 2.b) avant de tester -2- les processus (l'ergonomie en est une composante) et finir par -3- les tests de performance.

1. Stratégie « cadeau »

Le logiciel est donné un jour à des utilisateurs.

Avantages : pas de planification, adapté à des petits ou moyens systèmes non critiques

Inconvénients : aucune preuve de couvrir l'ensemble des cas possibles, consolidation (réparation) rapidement non maîtrisée, l'utilisateur se base sur son métier ou son ancien système.

2. Stratégie « comparaison »

On compare les données issues du nouveau système avec celles de l'ancien système.

Avantages : rapide, permet de qualifier un nouveau système à partir des mêmes données, on peut ré-utiliser le plan de test de l'ancien système.

Inconvénients : n'est pas fiable puisque la référence est le système existant, couverture hypothétique de tous les cas possibles, ne concerne pas les refontes lourdes ni les premiers systèmes.

3. Stratégie « incrémentale ascendante »

On commence par tester les modules de bas niveau (ex. écrans élémentaires, modules proches de la machine, lots élémentaires…) pour les intégrer progressivement dans une chaîne pour les lier finalement au plus général.

Avantages : les interfaces logicielles et matérielles délicates sont testées très tôt. Démarche empirique et de proche en proche.

Inconvénients : pas d'approche systémique, les erreurs de conception sont détectées très tard.

4. Stratégie « incrémentale descendante »

On commence par tester les modules généraux (ex. les menus) pour terminer par les modules plus détaillés (ex. écrans, lots de bas niveau…).

Avantages : correspond à une approche systémique, le testeur dispose d'une cartographie complète de l'application.

Inconvénients : les interfaces logicielles et matérielles sont testées à la fin, cela peut demander un effort important de simulation.

5. Stratégie « fonctionnelle-critique »

(cf des boîtes noires)

Teste les modules selon leur fonctionnalité. Différents critères peuvent être pris en compte pour définir les sous-domaines (technique, financier, échantillon). Elle correspond à une approche par lot.

Avantages : stratégie pragmatique facile à présenter et à justifier.

Inconvénients : échappe aux méthodologies de conception,

6. Stratégie « processus flots de données »

Les tests sont construits en fonction des flux de données.

Avantage : adapté pour les applications pilotées par les données. Minimise les cas de tests et facilite la preuve du test.

Inconvénients : nécessite une connaissance globale du système et l'intégration doit être testée.

7. Critères et techniques complémentaires pour constituer la campagne de tests

Lors de la mise en place de la stratégie, les « critères qualités », « les interfaces entre composants », la « vue fonctionnelle » et la « couverture de l'application » restent des dimensions à considérer avec attention.

7.1. Considérer les critères de qualité

Il est important de respecter les critères de qualité exigés par le commanditaire dans le cahier des charges. Par exemple, hormis les critères d'ergonomie étudiés par ailleurs, nous pouvons citer le critère de « maintenabilité » qui passe entre autres par :

- *La vérification syntaxique et lexicale*. A savoir une lecture croisée (chaque développeur relit le code rédigé par un autre développeur). Ceci facilite les tests unitaires et le respect des normes de programmation.
- *La mesure de l'usage de variables et de boucles*. Certains logiciels mesurent (leur emploi est rare) le nombre de boucles, de variables, de mise à jour, de profondeur d'imbrication

de boucles et de tests. Cela s'impose lorsque les logiciels doivent être particulièrement fiables (satellite, avion/pilote automatique, régulation de circulation routière, systèmes médicaux).

7.2. Tester les interfaces (ici ce n'est pas de l'ergonomie)

La vision systémique met l'accent sur les échanges et les flux entre les composants quels qu'ils soient, et on observe que les problèmes se passent souvent aux interfaces. Il convient donc de tester les interfaces avec attention et assez tôt dans le plan de tests :

- SGBD ⇔ composants de code utilisant le SGBD
- codage d'information ⇔ décodage d'information (ex. formats XML)
- signature des méthodes et des fonctions ⇔ appels de ces méthodes et fonctions
- changement de l'état d'objets ⇔ vues ou connaissances d'objets (notify ? ask ?)
- protocole de communications
- …

En fait, on observe que les développeurs réalisent, la plupart du temps, des tests unitaires, ils vérifient la validité de ce qu'ils réalisent. Or, le travail repose sur un travail collectif et sur des flux. « *Le tout est plus que la somme des parties* » Edgard Morin

7.3. Identifier les tests fonctionnels

(MatriceSystème :: Données X Traitements)

Une approche pour identifier les tests à réaliser pour le système ou sous-système consiste à les déduire du croisement entre, par exemple, entités et transactions, entre objets et opérations, ou encore entre objets métiers et cas d'utilisation.

	Objet Facture	Objet Compte-Client	…
Saisir facture	X	X	
Enregistrer règlement	X		
Consulter factures	X		
…			

On obtient ainsi une vue exhaustive des tests fonctionnels à effectuer pour l'application.

7.4. Couverture de l'application

(MatriceObjetMajeur :: Etats X Evènements)

Dans cette approche, pour un module ou un système donné on aura autant de matrices que d'objets majeurs. Pour un objet majeur on liste d'abord en ordonnée ses états possibles (les états permettent de récupérer la dynamique, c'est-à-dire la vie de ces objets dans le système). Ensuite on liste en abscisse les évènements ou les transactions ou cet objet est concerné.

	Module : Saisir Facture		
Objet : Client	Evénement 0	Evènement 1	Evènement e
Etat 1 : client créé	Cas 01	Cas 11	
Etat 2 : client débiteur	Cas 02	Cas 12	
Etat 3 : client créditeur	Cas 03	Cas 13	
…			

Le *Cas*-ij est représenté par une **Fiche de Test** dotée d'un **Jeu d'Essais** (données-mouvements-résultats attendus).

Cette approche permet d'élaborer des Jeux d'Essais et de couvrir un maximum de scénarios avec le maximum de mouvements.

Des documents pour les tests

Cas de test : Regroupe l'ensemble d'options d'exécution (ie scénarios) à vérifier pour ce cas. Ca peut être une instance d'un cas d'utilisation d'UML. Par exemple pour le cas de tests *Cas03 « Un client créditeur »* plusieurs scénarios sont envisageables, si la BD est vide, l'adresse n'existe pas… A chaque scénario correspond une fiche de tests.

Note : Ce cas de test peut avoir été identifié en même temps que *« Un client a découvert sur un livret »*, *« Un client créditeur »*, *« Un client débiteur depuis 48h »* selon la stratégie de tests mise en œuvre dérivée de *MatriceObjetsMajeurs*.

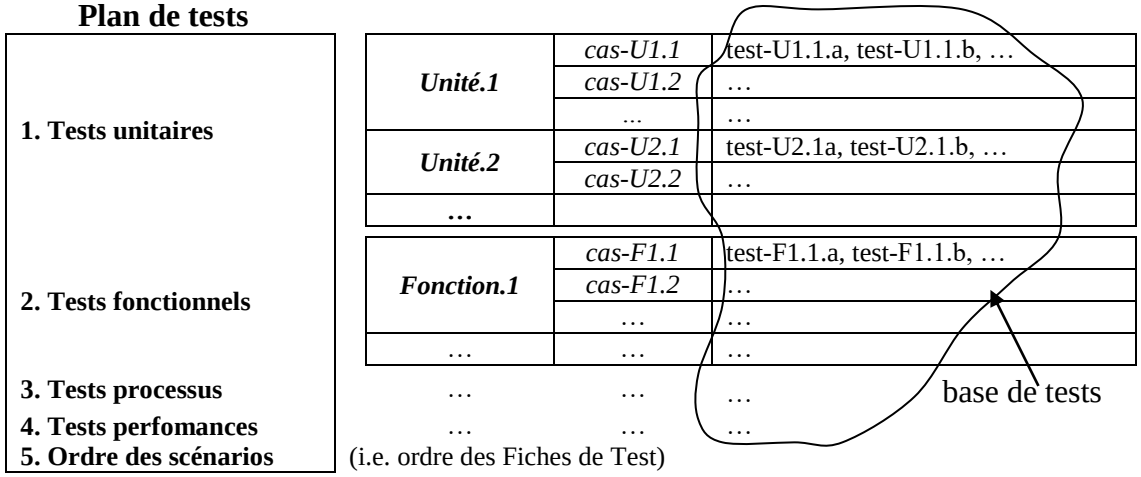
Un scénario : Enchaînement d'actions ou de messages utiles à l'exécution d'une option d'un cas de test. Il respecte les règles de gestion qui ont pu être spécifiées (cf. le problème de testabilité). Par exemple : scénario nominal (le plus classique) et scénarios alternatifs (ex. cas d'erreurs). A chaque scénario correspond une fiche de tests.

Jeu d'essai : Ensemble de valeurs nominales en entrée d'un scénario d'un cas de test. Une fiche de tests peut comprendre le jeu de d'essai à jouer si l'on souhaite être directif dans la campagne de tests.

Construire un jeu d'essai est une tâche fastidieuse. On se base sur les documents ou exemples recueillis car on préfère l'usage de données ayant du sens plutôt que le remplissage de zones par des 9999 ou des XXXX . Pour cela, on peut être conduit à prendre des échantillons (pb. de confidentialité). Plus le jeu d'essai est réduit, plus il est aisé à analyser. En rationalisant le cas de test on cherche à avoir 20 à 50 mouvements.

Base de tests : regroupe les divers cas de tests en une base cohérente pour diverses applications (une ☹). Cette base a un coût de maintenance élevé mais garantit la comptabilité des tests et des non-régressions rentables. Elle facilite un fort taux d'automatisation.

Plan de tests : En définitive l'ordre des tests à réaliser respecte une stratégie qui a été élaborée par l'équipe constituée des commanditaires et des fournisseurs. Par exemple pour une stratégie de tests Unitaires, Fonctionnels, Processus puis Performance. A chacun correspond des cas de tests pour lesquels différents scénarios sont à tester et vérifier. Le plan de test serait comme suit :



A chaque test-Xi.j.k correspond une **Fiche de Test** qui précise le scénario, les données à fournir et les résultats attendus. Selon la stratégie, des niveaux de profondeur peuvent être omis.

Fiche de test : Elle décrit entièrement le travail de test à effectuer. Elle est nécessaire pour planifier les tâches de tests et indique sa place dans le sous-projet de tests. Elle contient la description de l'objectif de test, les pré-requis (tests préalables), les éléments de départ (leurs valeurs) et les résultats attendus.

Fiche de test : _____ **Réf. fiche test :** *U1a.1* _____

Campagne : _____

Description de l'objectif du test

*Vérifier qu'au bout de 3 pswd erronés le compte est bloqué*_____

_____

_____

Pré-requis et conditions générales (matérielles, logicielles…) d'exécution du test

Disposer d'une connexion internet et accéder à l'appli : https//xxx.xxx.com _____

_____

_____

_____

Actions et procédures à réaliser (scénario)

Saisir login LOPIS Saisir pswd TOTO Obtenir : ***erreur pswd 1*** _____

Saisir pswd TOTO Obtenir : ***erreur pswd 2*** _____

Saisir pswd TOTO Obtenir : ***erreur login BLOQUE contacter admin@com*** _____

_____

_____

Résultats attendus

_____

_____

_____

_____

_____

Si l'équipe choisit de placer sur la **Fiche de test** un volet « résultats obtenus », ceux-ci ne seront indiqués que sur des fiches dupliquées et la date sera indiquée. Sinon, les résultats sont portés sur la **Fiche de suivi**.

Fiche de suivi : Permet de garder la trace des tests effectués, elle constitue donc un journal de l'ensemble des tests effectués. Elle comporte le résultat d'un test passé avec succès ou qui comporte des anomalies ou erreurs. Parmi la gravité des erreurs on distingue :

- *Les erreurs bloquantes* : plantent le logiciel. Elles interdisent sa livraison au client
- *Les erreurs graves* : ne respectent pas les spécifications, qu'elles soient écrites sous forme de règle ou énoncées sous forme de scénarios. Le logiciel ne sera pas mis en production avec ce genre ce genre d'erreur, mais il peut être recetté et servir en phase d'apprentissage (dangereux).
- *Les défauts* : peuvent être d'affichage ou de navigation (zone trop courte, retour à l'accueil, lien dynamique perdu…)

Cette fiche est datée, fait référence à une fiche de test, résume les résultats obtenus et indique la validation par rapport aux résultats attendus, avec signature ou visa du responsable.

Fiche de suivi : _____ **Réf. fiche suivi :** _____

Campagne : _____

Fiche de tests : _____ **Réf. fiche de tests :** _____

Résultats du test

□ Succès

X Echec Gravité erreur :5/5 _____ Réf. fiche anomalie : _____

Commentaires (ex. par rapport aux résultats attendus)

Une fois _____

_____

_____

_____

Testeur : _____ Responsable : _____

Durée du test : _____ Date : _____

Fiche d'anomalie : Toute anomalie est identifiée et est décrite, les alternatives sont envisagées, la solution apportée est décrite et la validation est indiquée.

Fiche d'anomalie : _____

Réf. fiche suivi : _____

Date : _____ **Nom :** _____

Description anomalie (et/ou liste des pièces jointes) **:**

Dès qu'il y a erreur de pswd on ne peut plus ressaisir le login LOPIS, ni son pswd ___

_____

_____

_____

N° du bug associé (optionnel) : _____