

# Principaux concepts et cycle projet

## 1. Introduction

### a. Premières considérations : parlons la même langue !

Nous partirons du principe que le lecteur est totalement novice en matière de production logicielle afin de définir les mots employés et exposer un vocabulaire qui devrait idéalement être commun à tous les professionnels du monde de l'informatique.

Malheureusement, l'expérience montre que derrière les mots se cachent des concepts qui n'ont pas toujours la même compréhension. Il y a une part de méconnaissance ou plus exactement de connaissances orientées par une expérience professionnelle propre, qui fait que certains concepts organisationnels sont compris sous un angle qui différera d'un lecteur à l'autre.

Mais il y a aussi une part d'interrogation soulevée par les mots eux-mêmes dont les définitions peuvent être tributaires d'un contexte. Hors de ce contexte, le mot employé perd alors son sens.

 Un simple exemple pour illustrer ceci : l'expression "test d'intégration" fait référence à des tests permettant de vérifier une interopérabilité entre deux logiciels ou deux composants d'un même logiciel. Mais lorsqu'un environnement de test est appelé "environnement d'intégration", par extension, un test réalisé dans cet environnement pourra être improprement appelé test d'intégration sous ce seul prétexte.

Et de cette considération générale, nous pouvons même plus précisément affirmer que le vocabulaire et les abus de langage sont l'une des principales problématiques de l'univers du test.

Le présent chapitre est donc indispensable : il est le prérequis incontournable auquel les professionnels sont confrontés pour parler un vocabulaire commun.

### b. Schéma général du cycle projet

Le schéma suivant représente les étapes principales d'un cycle projet conduisant à la réalisation d'un logiciel puis sa maintenance dans un contexte industrialisé. Le principe qu'il expose est relativement classique dans le monde industriel si ce n'est une spécificité de vocabulaire peut-être. Mais ce principe reste universel.

Nous trouvons donc deux chantiers successifs :

- Le chantier de réalisation, dit de "release" ou de "build".
- Le chantier de maintenance, dit de "run".

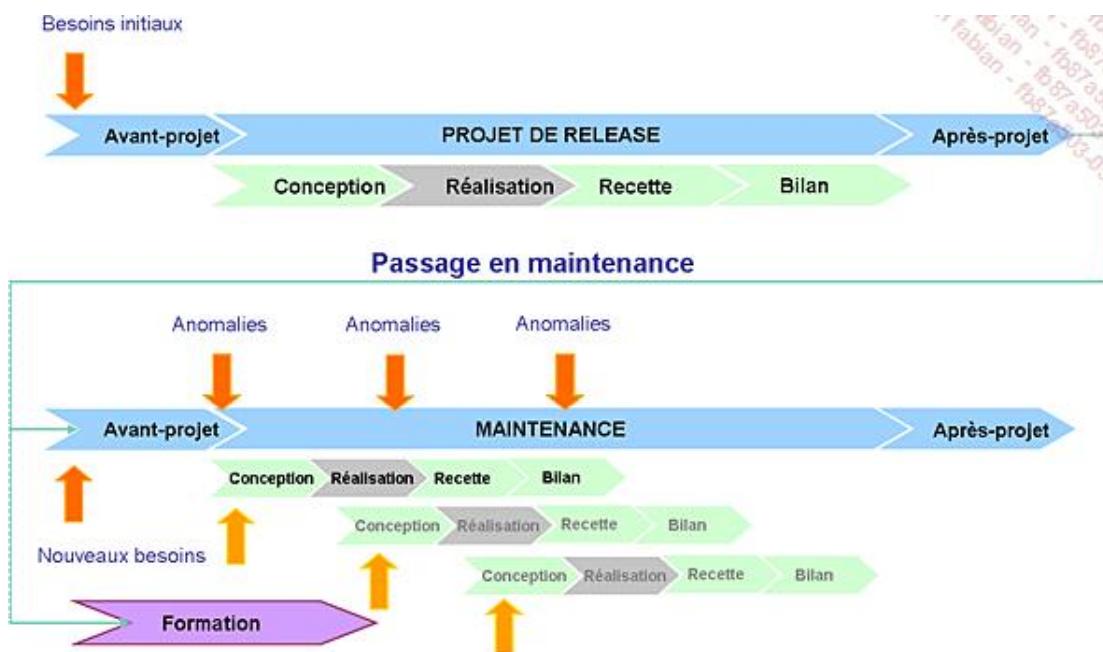
Le premier consiste en la réalisation d'une solution, le suivant permettant de faire vivre cette solution dans le temps - ce qui suppose normalement qu'un chantier de formation est mené conjointement durant celui-ci car la complexité de la solution nécessite parfois l'apprentissage de ses utilisateurs.

La distinction entre ces deux chantiers tient surtout dans son origine : un chantier de release trouve sa source dans une décision souvent prise par la direction d'une entreprise car correspond à la réponse apportée pour atteindre un objectif stratégique.

A contrario, le chantier de maintenance trouve son origine dans la nécessité de pérenniser la solution en collectant des besoins additionnels (et les éventuels dysfonctionnements) ce qui revêt un aspect de stratégie moindre car semble "plus routinier".

Nous n'apprendrons rien aux chefs de projets qui nous lisent qu'un projet de release est le plus souvent un enjeu politique dans le fonctionnement d'une entreprise : de sa réussite dépendent des résultats financiers qui intéressent la direction, notamment un calcul de retour sur investissement.

Le chantier de maintenance apparaît alors comme d'importance moindre. Mais ce n'est qu'une apparence comme nous allons le voir amplement tout du long de ce livre.



Nous retiendrons surtout que chacun de ces deux chantiers est découpé dans le temps selon la même structure :

- Un avant-projet
- Un projet, se découvant en un ou plusieurs cycles alternant :
  - Une phase de conception
  - Une phase de réalisation
  - Une phase de recette
  - Un bilan
- Un après-projet

Les sections qui suivent dans le présent chapitre consisteront à résumer sommairement chacune de ces phases.

## 2. L'avant-projet

Théoriquement, le rôle d'une maîtrise d'ouvrage (MOA) est de conduire les études d'avant-projet, c'est-à-dire fournir à la direction d'une entreprise la capacité à envisager de faire évoluer le système d'information et les processus organisationnels de celle-ci.

Cet aspect est probablement le plus méconnu des développeurs et techniciens notamment, qui ne voient dans la MOA "que" l'interlocuteur des logiciels qu'ils réalisent, un "commanditaire", une structure d'ailleurs qui meurt avec la fin du projet, contrairement à l'activité d'étude qui elle est permanente.

Le lecteur comprendra aisément qu'avant tout projet des questions viennent. Et les deux premières de toutes que se posera un chef d'entreprise sont :

- Combien cela va me rapporter ?
- Combien cela va-t-il me coûter ?

Du bon sens en somme.

Ces questions viendront naturellement, que le chantier soit de release ou de maintenance, un système d'information n'évolue pas pour se faire plaisir mais pour répondre à des enjeux en étant rentable.

### a. Étude d'opportunité

Dès lors qu'une idée d'un nouveau chantier de release a été proposée ou qu'une application est en maintenance, de nouvelles opportunités s'offrent à l'entreprise. Mais lesquelles exactement ? Quels sont les gains espérés ? Y a-t-il des contraintes techniques ? Organisationnelles ? Juridiques ? Quels seront les coûts de mise en œuvre ? De maintenance ? Quels risques l'entreprise prend à changer son système ?

Vous notez immédiatement que des questions cruciales émergent et qu'il convient de les poser... par écrit !

Une étude d'opportunité débouche donc le plus souvent sur un document qui formalise l'idée initiale et tente surtout d'en avoir une approche concrète.

Cette approche permettra un premier recul face à l'enthousiasme parfois exacerbé d'un décideur qui n'a pas la capacité immédiate de voir tout seul l'étendue de ce qu'il propose. Il est d'ailleurs probable qu'il voit surtout les gains potentiels immédiats... mais plus difficilement que son idée va nécessiter de chambouler les méthodes de travail de plusieurs dizaines ou centaines de personnes. Il reste un être humain avec un champ visuel restreint par son humaine condition.

Et l'organisation des tests ? Nous en sommes loin. Quoique...

L'étude d'opportunité va fixer des objectifs. Qui dit objectif dit mesure des résultats et évaluation des écarts, donc recherche des causes.

Est-il nécessaire de vous dire qu'une solution performante a plus de chance d'amener de la rentabilité ?

L'organisation des tests logiciels sera pourtant prépondérante : sur elle reposera la vérification indispensable de la performance globale du produit. Il est donc cohérent d'affirmer que sans test, il n'y a pas de performance et que les gains ne sauraient être optimums.

### b. Étude d'impact

À l'issue de l'étude d'opportunité - ou de manière complémentaire - une étude d'impact pourra être conduite. Cette dernière n'a pas le même objectif : elle doit recenser l'ensemble des "éléments" impactés.

L'impact d'un projet de production logicielle est d'abord technique, c'est évident : nouveaux logiciels, nouveaux serveurs, augmentation ou diminution des flux de données sur le réseau de l'entreprise...

Mais il est aussi organisationnel : les salariés ne travailleront plus de la même manière. Le système d'information de l'entreprise est en effet le centre névralgique qui interagit avec tous. Le modifier suppose donc affecter tout ou partie des collaborateurs qui l'utilisent.

La productivité est alors impactée, les coûts, les plannings, les risques encourus qui peuvent augmenter ou diminuer, jusqu'à la relation avec des tiers extérieurs, qu'ils soient clients, fournisseurs, partenaires...

La MOA peut d'ailleurs avoir nécessité à commander des études d'impact à des prestataires externes très variés : études de marché, questionnaires proposés aux clients, évaluation d'une architecture technique ou d'un progiciel du marché, étude comparative ou de concurrence...

Bref, des sujets de réflexion multiples qui concernent le plus souvent les chantiers de release.

Est-ce à dire qu'un chantier de maintenance ne fait jamais l'objet d'une étude d'impact ? Rien n'est moins sûr.

Supposons que la maintenance consiste à ouvrir un nouveau moyen de paiement sur un site web marchand. Ne va-t-on pas se poser la question des préférences des usagers à utiliser ce médium ? De leurs comportements en termes d'achat ?

L'organisation des tests en apparence n'a pas de rôle au premier abord. Pourtant, les expériences passées en matière de test vont conditionner l'étude d'impact.

Va-t-on utiliser un progiciel déjà présent dans l'entreprise si sa mise en œuvre a généré une volumétrie importante d'anomalies dans un précédent projet ? Peut-être pas car l'outil pourra alors avoir une mauvaise image de marque en interne.

Pire, la MOA pourrait même ne pas avoir confiance en sa propre maîtrise d'œuvre (MOE) interne ou externe, et choisir une solution évitant toute prise de risque.

Car ce serait oublier que le but d'une étude d'impact est d'esquisser des pistes de solutions.

### c. Le choix

À l'issue d'une étude d'opportunité - complétée ou non par une ou plusieurs études d'impact - la MOA va formaliser un dossier de choix, c'est-à-dire un ensemble de scénarios permettant d'atteindre les objectifs de gain, en minimisant les coûts et les risques pris, tout en prenant en compte des contraintes.

Nous attirons l'attention du lecteur sur le fait qu'une solution n'est pas seulement un produit à réaliser mais aussi une organisation du projet. Ainsi, un même logiciel pourra être conçu en empruntant des chemins différents en termes de planning, de budget, de coûts... donc faire l'objet de plusieurs solutions.

Va-t-on réaliser le produit en interne avec sa propre MOE ? Ou bien fera-t-on appel à un prestataire externe car le savoir-faire est insuffisant en interne ? Doit-on recruter une ou plusieurs personnes ayant le savoir-faire qui manque ?

Quels outils choisir ? Une solution du marché qui sera ensuite adaptée au contexte de l'entreprise ? Un logiciel fait maison en partant de zéro ? Si un progiciel est proposé, quel est son degré de maturité ? La concurrence l'utilise-t-elle ? A-t-on un retour d'expérience en interne sur son usage ?

Quand la solution doit-elle être disponible ? Quand doit-elle être rentable ? Ai-je une contrainte juridique pour sa mise en œuvre ? (par exemple, la bascule Euro avait une date d'échéance inamovible par définition).

Toutes ces questions ayant été exposées et prises en compte, le groupe décideur peut faire un choix : un choix d'objectif, un choix de projet.

Ce choix met fin la phase d'avant-projet.

À noter que dans le cadre d'un chantier de maintenance, cette phase intervient dans deux situations :

- À l'issue du projet de release qui a amené à la mise en place d'une solution, auquel cas les volumétries d'anomalies détectées auparavant, jouent un rôle dans la manière dont va être envisagée la maintenance (il est probable qu'une volumétrie forte inquiète et que le budget prévisionnel de maintenance soit revu à la hausse).
- À l'issue d'un exercice annuel de maintenance qui s'appuiera sur les observations faites pendant l'exercice précédent, donc sur les volumétries d'anomalies corrigées aussi.

Le choix ne consiste plus ici à trouver une solution (elle est en place !) mais plutôt à estimer les petites maintenances évolutives et correctifs à venir, à minima pour estimer le budget alloué à l'application à maintenir. A maxima, la maintenance d'une application peut être externalisée chez un prestataire dans le cadre d'une tierce maintenance applicative (TMA).

L'organisation des tests n'est donc pas un point vital à ce stade du processus, mais le lecteur mesure déjà l'importance d'une bonne stratégie de tests car son résultat conditionne pour partie les décisions qui seront prises.

Nous conclurons cette section par une affirmation : une MOA qui est très préoccupée par la qualité logicielle et met en place une stratégie globale des tests, s'assure et rassure la direction de l'entreprise que le système d'information répondra à des exigences de performance.

Et a contrario, une MOA qui ne s'occupe pas de gérer les tests et la qualité logicielle perd à coup sûr les principaux indicateurs de choix d'une future étude avant-projet.

Il n'est de pire aveugle que celui qui ne veut pas voir.

### 3. Le projet

#### a. Lancement

L'étude avant-projet terminée, le choix est fait de mettre en œuvre une solution organisationnelle et d'un objectif unique à atteindre comme l'illustre l'acronyme américain "OTOBOS" (*One Time, One Budget, One Solution*).

L'ensemble des éléments de cette solution sont en général condensés dans une note de lancement rédigée par le chef de projet MOA, celui qui sera responsable des délais et budgets alloués, le garant des résultats.

Le plus souvent, cette note servira d'ordre du jour pour conduire une réunion de lancement au cours de laquelle les principaux acteurs du projet seront conviés.

Le lancement est aussi l'occasion d'affiner le budget parfois, ne serait-ce que parce qu'entre le moment où l'étude avant-projet a été menée et le lancement même, des faits nouveaux peuvent apparaître. Les instances de pilotage et de conduite du projet sont rappelées, le calendrier stabilisé.

Mais surtout cette étape du projet conditionne l'organisation des tests à tous les niveaux.

À ce stade, la MOE est identifiée, l'équipe en charge des tests techniques aussi, de même que l'équipe MOA dédiée au projet, tant pour l'expression détaillée des besoins que pour la recette métier qui suivra.

Le chef de projet MOA doit alors être conscient des forces et faiblesses de cette organisation :

- La maturité de la MOE permettra-t-elle une gestion efficace des tests unitaires ?
- La cellule en charge des tests techniques en sortie de développement est-elle rattachée à la MOE ? à la MOA ? Externalisée ? D'ailleurs existe-t-elle ?

- La MOA a-t-elle une grande expérience des recettes ? S'il n'y a pas de cellule de tests techniques, ne faut-il pas que l'équipe MOA soit renforcée ? Ou bien va-t-on déporter l'effort de test sur la MOE ? Et si la recette métier est externalisée chez un prestataire, quel risque prend-on s'il n'y a pas de cellule de tests techniques ?

Le chef de projet fonctionnel doit ici se poser des questions pratiques qui, il faut l'espérer, auront été anticipées lors de l'étude avant-projet.

Nous retiendrons principalement que les tests s'organisent au sein de trois étapes du projet :

- Lors de la réalisation du logiciel - qui peut être découpée en lots développés successivement.
- À l'issue de la réalisation et des livraisons des lots à une équipe dédiée aux tests techniques.
- À l'issue de la livraison de l'application à la MOA pour une recette purement fonctionnelle dite "recette métier".

Cependant, il conviendra de retenir aussi qu'il s'agit d'une théorie et que la réalité du terrain diffère souvent de cette dernière.

## b. Conception

Sommairement, nous pourrions dire que l'étape de conception consiste pour la MOA à collecter les besoins auprès des utilisateurs afin de les formaliser dans un cahier des charges et/ou des spécifications fonctionnelles.

Si la MOA souhaitait externaliser la rédaction des spécifications, elle pourrait se contenter de rédiger un cahier des charges. C'est le niveau de maturité de la MOA qui détermine nettement son implication.

Il faut comprendre qu'une MOA a deux types d'activité :

- Une activité "de fond" récurrente : les études avant-projet, la collecte des incidents et la collecte des demandes évolutives (pour les maintenances).
- Une activité "événementielle" : la conduite d'un projet de release donné, qui provoque des pics d'activité irréguliers.

L'organisation de l'entreprise peut conduire la direction à ne pas avoir une MOA dédiée en permanence mais faire ponctuellement appel à des ressources issues du métier. Derrière ce contexte se cache un choix logique : quelle est la compétence de l'entreprise ? Créer des logiciels ou exercer son cœur de métier ?

Ainsi, la rédaction de spécifications pourra être externalisée, la MOA se limitant à piloter le budget du projet et prendre les décisions.

Quoi qu'il en soit, la phase de conception est incontournable et conduit à un ensemble documentaire qui exprime de manière détaillée l'objectif à atteindre.

Ces spécifications font triple emploi : elles sont l'entrant de la phase de réalisation menée par la MOE et conjointement l'entrant de la cellule de test et de l'équipe de recette métier pour préparer les tests.

Dans le cas d'une rédaction interne des spécifications par la MOA, il est d'ailleurs parfaitement envisageable que l'équipe de rédacteurs soit également celle qui fera la recette métier.

-  Et quand la MOE rédige les spécifications ? Il est évident que ce n'est pas son rôle : ce n'est pas au fournisseur de remplir un bon de commande ! Malheureusement, certains contextes projet conduisent à cette situation fortement risquée. Certes une MOE a souvent les compétences pour rédiger des spécifications - d'autant plus si elle est externalisée chez une SS2I - mais le risque de rejet de l'application est alors élevé.

Nous attirons l'attention du lecteur sur le double rôle des spécifications relativement aux tests :

- Elles définissent le résultat attendu donc implicitement constituent le référentiel de base qui permet de qualifier qu'un résultat exécuté n'est pas conforme à ce qui est souhaité.
- Elles décrivent la totalité d'un produit, donc un périmètre fonctionnel de test. Ce périmètre est par essence un premier élément permettant de chiffrer la charge nécessaire aux tests, tant les tests techniques que les tests fonctionnels.

## c. Réalisation

Sur la base des spécifications fonctionnelles, de maquettes, de chartes graphiques et de tous documents susceptibles de décrire l'objectif à atteindre, une MOE réalise le logiciel.

Cette réalisation peut être lotie dans le temps, être interne à l'entreprise ou externalisée chez un prestataire, peu importe : elle consistera toujours à construire une architecture technique (serveurs, bases de données...) sur laquelle s'appuieront des composants logiciels du marché et un ensemble de traitements réalisés par des développeurs.

Cette équipe est logiquement pilotée par un chef de projet MOE - parfois lui-même développeur sur les projets de petite taille - dont la fonction première est de prendre connaissance des entrants afin de vérifier leur cohérence (il doit donc comprendre ce qu'il a faire) mais aussi la charge de travail que cela représente.

Théoriquement, une négociation a lieu entre MOA et MOE sur le périmètre de la réalisation (il peut d'ailleurs y avoir plusieurs solutions techniques), la charge à y consacrer et bien entendu, le planning des livraisons.

Mais, ce serait oublier que le chef de projet MOE est aussi garant du résultat de sa partie. Les développeurs sont amenés à réaliser un minimum de tests ou "tests unitaires". Et il est évident que les résultats de ces tests sont au moins communiqués oralement à ce chef de projet qui doit prendre connaissance des vérifications faites par sa propre équipe.

Il n'est pas demandé à une MOE de réaliser une recette complète d'autant plus que l'environnement même de développement ne s'y prêtera certainement pas.

Mais cela ne veut pas dire qu'il ne faut rien faire.

Les tests unitaires représentent ainsi une part de l'activité d'une MOE et il conviendra de les organiser. À ce sujet, le présent ouvrage propose au chef de projet MOE des techniques simples pour mener à bien cette activité.

Soulignons au passage qu'une équipe de développeurs n'est pas une équipe de testeurs et que la culture même du test n'est pas toujours présente chez ces ressources - ce qui n'enlève rien à leur qualité professionnelle pour autant.

 Le développeur est un créatif, de profil scientifique certes, mais un créatif au même titre qu'un auteur, un peintre ou un sculpteur. Il y a ici une dimension psychologique à comprendre : l'état d'esprit du testeur n'est pas celui du créatif mais celui du contrôleur. Il y a ainsi une grande différence d'approche entre celui qui réalise un dessin et celui qui regarde ce dessin pour en chercher les erreurs comme dans un jeu "des 7 différences".

Il y a donc dans la notion même de test, une qualité intrinsèque nécessaire dont n'est pas toujours pourvu celui qui réalise. Tout simplement parce que le positionnement de la personne n'est pas le même : le développeur est l'habile mécanicien qui connaît le moteur d'un bolide que le testeur se contente de piloter. Mais ce n'est pas ce mécanicien qui ira ensuite faire des crash-tests.

Et ni le développeur, ni le testeur, n'iront ensuite concourir sur un circuit pour remporter des courses.

## **d. Recette technique**

Par recette technique, on entend l'ensemble des tests menés sur le produit réalisé immédiatement après livraison par une équipe MOE. Cette recette consiste à remonter le maximum d'anomalies dans le budget imparti, afin de les transmettre à la MOE pour correction, et cela avant toute recette métier.

La recette technique couvre quatre aspects :

- Les tests d'assemblage des briques réalisées qui consistent à s'assurer que le produit est cohérent, homogène et opérationnel.
- Les tests d'interface entre le produit et les autres logiciels du système d'information - on parle aussi de tests d'intégration.
- Les tests de compatibilité des configurations matérielles, par exemple pour un site web, tester la même application avec plusieurs navigateurs.
- Des tests "de bout en bout" qui revêtent des aspects fonctionnels, et dont le but est de vérifier le bon fonctionnement du produit dans la profondeur.

Théoriquement, cette recette doit être menée par une équipe distincte de la MOE proprement dite. Mais dans la pratique, les contextes organisationnels, les contraintes budgétaires aussi, font que cette équipe peut ne pas exister, l'effort de test étant alors réparti entre la MOE et/ou la MOA.

L'organisation d'une recette technique est pourtant d'une extrême importance : elle est un véritable projet dans le projet. Le présent ouvrage s'attardera longuement sur cette organisation qui requiert une spécialisation.

Sans aucun doute, la recette technique est une expertise.

## **e. Recette fonctionnelle ou métier**

Dès lors qu'un logiciel a été produit et éprouvé au cours d'une recette technique, il est susceptible d'être testé par une équipe métier rattachée à la MOA.

L'objectif de la recette fonctionnelle diffère alors : il s'agit de vérifier l'adéquation du produit en tant que réponse à une problématique métier.

Bien sûr, la recette fonctionnelle a pour objectif de vérifier que les règles de gestion ont été correctement implémentées mais elle doit aussi s'assurer que le logiciel est complet et répond au besoin.

De ce point de vue, les spécifications fonctionnelles produites par la MOA (ou sous la direction de la MOA) constituent bien entendu la référence qui déterminera s'il y a ou non une non-conformité dans le logiciel testé.

Mais au-delà de cette simple vérification, la recette fonctionnelle sera le moment du premier contact entre la solution et une population d'utilisateurs finaux - ou en tout cas d'un groupe de personnes qui connaît bien le métier.

Une recette fonctionnelle ne revêt donc pas d'aspects techniques. Par exemple, il ne sera pas demandé de tester la compatibilité d'une application web sur deux navigateurs.

En revanche, son organisation s'apparente à la recette technique bien que les buts diffèrent, raison pour laquelle le présent ouvrage répond à cette problématique.

## **f. Recette usine**

On appelle recette usine une activité de test qui consiste pour un client à venir vérifier le bon fonctionnement d'un logiciel dans les locaux de son prestataire, dans les murs de "l'usine logicielle".

Cette typologie d'organisation est vraiment spécifique à la prestation de services. Nous n'évoquerons pas son organisation dans le présent ouvrage cependant.

Sachez simplement que le plus souvent il s'agit davantage d'une démarche client visant à rassurer avant livraison : la recette usine se situerait alors après la recette technique, mais avant la recette fonctionnelle, à titre préparatoire.

Il est en effet plus rassurant pour tous les acteurs (prestataires et clients) d'avoir vu l'application fonctionner avant livraison.

De ce fait, une recette usine n'a pas tant une vocation à éprouver le logiciel que d'avoir un retour anticipé du client. Va-t-il rejeter l'application ?

Retenez que la recette usine a lieu le plus souvent sur une application qui n'est pas toujours achevée et qu'en conséquence son périmètre doit être strictement cadre : le client vient chez son prestataire pour faire des tests, certes, mais il n'est pas pour autant laissé seul.

Ainsi, le pilotage de la recette usine revient à organiser une sorte de répétition générale d'une partie de la recette fonctionnelle avec des testeurs issus du métier et qui se déplacent chez un fournisseur.

Faut-il pour autant toujours organiser une recette usine ? Rien n'est moins sûr et seul le contexte du projet - et surtout l'état de la relation client - peut justifier de sa nécessité.

#### **g. Recette d'homologation ou VABF**

La vérification d'aptitude au bon fonctionnement (VABF) est une recette d'homologation.

Cette expression contient en elle une dimension contractuelle forte puisqu'elle signifie qu'une validation est donnée, un coup de tampon "Certifiée Vérifié" en quelque sorte.

Nous attirons le lecteur sur deux aspects :

- La définition du périmètre de l'homologation.
- La définition même de l'homologation : qu'est-ce qui est homologué et surtout pour et par qui ?

Tout d'abord le périmètre de l'homologation : comment est-il défini ? Est-ce une sous-partie du périmètre de la recette fonctionnelle ? Est-ce une sous-partie de la recette technique ?

En général, le périmètre d'une VABF est défini au préalable des développements et sur des critères d'acceptation factuels. Ce qui est sûr, c'est que ce périmètre sera nécessairement plus restreint que le périmètre d'une recette technique ou fonctionnelle. Tout du moins, dès lors que la VABF n'est pas la recette technique ou la recette fonctionnelle elle-même.

En effet, rien n'empêche dans l'organisation du projet de confondre la recette fonctionnelle et la VABF, l'une ayant une utilité opérationnelle de production, l'autre purement contractuelle.

Dans un tel contexte, nous parlerions alors de recette d'homologation fonctionnelle, concept qui unit les deux aspects.

Par extension, il est d'ailleurs possible de parler aussi de recette d'homologation technique. Typiquement, supposons que la réalisation soit externalisée dans une SS2I de grande taille munie d'une cellule de tests dans un centre de production délocalisé, par exemple à l'étranger.

Il est clair qu'un tel contexte projet suppose que la cellule de tests exécutant la recette technique joue un rôle d'homologation vis-à-vis de la MOE.

Nous alertons donc le lecteur sur l'usage de mot homologation qui induit en erreur car il sous-entendrait trop facilement cette supposition qui voudrait qu'une homologation soit nécessairement fonctionnelle et conduite par la MOA.

Mais retenez toutefois qu'il est fréquent de réaliser l'homologation à l'issue de la recette fonctionnelle, tant est si bien que recette fonctionnelle et VABF deviennent pratiquement toujours synonymes.

## **h. Déploiement et montée en charge**

Dès lors qu'un logiciel a été homologué par la MOA, c'est-à-dire que contractuellement, le résultat a été accepté, l'application pourra être déployée.

Selon son architecture (client lourd, client léger...), le processus de déploiement pourra être plus ou moins complexe. Une procédure d'installation aura été rédigée ce qui implique que l'installation elle-même peut faire l'objet d'un test. Le présent livre ne traitera pas ce sujet mais nous attirons l'attention du lecteur sur le risque que peuvent présenter certaines architectures en termes d'installation, notamment les applications mobiles sur iPhone, iPad, SmartPhone, Pocket PC et Palm, etc.

En effet, l'installation d'un logiciel ne se résume pas toujours à une succession de clics souris : la mise en service d'une solution peut être beaucoup plus complexe qu'il n'y paraît.

Aussi, un planning de montée en charge progressive est généralement mis en place et nécessite des points de contrôle réguliers, notamment la mesure des performances sera analysée avec soin.

Le présent livre ne proposera pas de techniques en la matière : nous l'évoquons à titre informatif.

## **4. L'après-projet**

### **a. Clôture du projet**

Par clôture de projet, nous entendons le moment où le projet de release s'achève et donne lieu au début d'une phase d'après-projet.

Concrètement, cela signifie que le budget a été consommé, que l'échéance a été atteinte, et surtout :

- Qu'un livrable a été fourni par la MOE à la MOA.
- Que la MOA a publié la note de clôture du projet pour indiquer sa terminaison. Ce qui suit alors n'est plus le projet.

La clôture est donc un moment clé du projet au cours de laquelle une synthèse objective est formalisée.

Le résultat des différentes recettes notamment est exposé : volumétries détectées, taux de pertinence des remontées, nombre de livraisons... autant d'indicateurs susceptibles d'illustrer le déroulement du projet et de justifier les dépenses faites et les risques évités ou subis.

L'impact de l'organisation des tests sur la clôture des tests est très important puisqu'une bonne organisation

permet de dégager les indicateurs en question. Sans eux il n'est alors pas possible d'esquisser une image précise de son déroulement.

## b. Bilan

Le bilan (ou les bilans) du déploiement d'une solution correspond à un résultat à froid permettant de mesurer les performances et vérifier si les gains envisagés lors de l'étude avant-projet ont été atteints. Le bilan est mené par la MOA et alors que le projet est terminé depuis une période plus ou moins longue.

Rien n'interdit d'ailleurs de refaire ce bilan de manière périodique pour réactualiser les résultats, les réajuster.

L'organisation des recettes n'a plus d'impact à ce stade sinon la qualité des tests de performance qui ont pu être réalisés préalablement - dans la mesure où ces tests étaient possibles en l'absence d'un environnement réel de production.

Nous ne nous étendrons pas davantage sur ce sujet donc.

# Types de test

## 1. Pendant le projet

### a. Relecture des spécifications

Qu'il soit de release ou de maintenance, un projet passe par une étape de conception qui débouche sur un livrable : les spécifications. Une première technique de test est incontestablement leur relecture. Peut-être serez-vous surpris de lire ce commentaire ici, mais l'expérience montre que très souvent ses dernières sont lues "en diagonale".

Nous conseillons expressément de ne pas tomber dans ce travers. Que la recette à organiser soit technique ou fonctionnelle, que vous deviez mettre en place des tests unitaires, peu importe, lisez !

Avantageusement, une lecture en binôme permettra de remonter un maximum d'observations. Focalisez-vous notamment à rechercher ce qui est absent, les trous fonctionnels, les non-dits. Et ces derniers peuvent être nombreux. Quelques exemples :

- Sur la description des données
  - Si un champ est indiqué comme alphanumérique, faire préciser son format exact : alphabétique, avec ou sans caractère accentué, en majuscule, en minuscule. Faire préciser aussi sa longueur, si les espaces à gauche et à droite sont acceptés, etc.
  - Si un champ est numérique, faire préciser son domaine de validité avec précision (signé ? non signé ? nul ? usage du point ou de la virgule pour les réels ? quel séparateur pour les milliers ?).
  - Si un champ est une date, faire préciser son domaine de validité, notamment par rapport à la date du jour. Impérativement, faire préciser comment est définie la date du jour (lue sur le serveur applicatif ? sur le serveur de données ? sur le PC client ? ...).
- Sur la description des traitements
  - Pour les lectures en base de données, le résultat doit-il être trié ? Quelles sont les conversions de format utilisées une fois les données lues ?
  - Pour les écritures en base de données, quelles sont les données faisant l'objet d'une contrainte d'unicité ? Quelles sont les conversions de format utilisées pour le stockage de l'information ?
  - Pour les suppressions en base de données, quel pourrait être le rythme de ces dernières : ponctuelles ? en masse ? Prévoir l'incidence sur l'indexation des données...
- Sur la description des éléments graphiques
  - Repérer les non-dits de la charte graphique : comment signale-t-on les champs obligatoires des formulaires ?
  - Qu'est-ce qui doit être homogène ?

Notez qu'une anomalie repérée suite à une relecture attentive des spécifications coûtera toujours beaucoup moins cher qu'une détection tardive : vous gagnerez du temps sur le développement, sur la recette qui suit, donc à minima une charge qui aurait été consommée par 2 à 4 personnes (le développeur, son chef de projet MOE, le testeur et le chef de projet recette technique). Et que dire si l'anomalie est détectée en recette métier ?

Retenez immédiatement ce principe : plus tard l'anomalie est détectée plus son impact sur la charge du projet est important.

## b. Types de tests unitaires

Par définition, les développeurs sont en charge de réaliser des tests unitaires. Mais qu'entend-on par là ?

Le plus souvent, le développeur qui teste est en butte à deux contraintes :

- Son environnement de développement
- Sa compréhension fonctionnelle du sujet traité

De ce fait, un développeur a une approche des tests qui aura des limites. Tout d'abord, le terme "unitaire" ne signifie pas "unique" : plusieurs tests doivent être réalisés. Un test est dit unitaire car l'objet testé, le "bout de code" réalisé par le développeur, ne constitue qu'une sous-partie du logiciel global, une unité.

En conséquence, le test unitaire se limite au seul périmètre de la fonctionnalité développée comme si ce test consistait à vérifier les dimensions d'un parpaing retaillé par le maçon avant d'être appareillé dans le mur : l'ouvrier ne pourra vérifier que l'élément et éventuellement sa proximité avec les éléments immédiatement connexes. Il ne pourra pas vérifier la cohérence globale de la maison qu'il réalise à l'instant où il place l'élément dans la structure.

Cette vision parcellaire du logiciel conditionne donc l'environnement de travail du développeur :

- Il ne dispose pas de toutes les parties du logiciel conçu ou bien des parties en cours de réalisation par d'autres développeurs donc non encore fiabilisées (problématique de synchronisation).
- La base de données (pour laquelle il n'a d'ailleurs pas toujours un accès) contiendra des informations échantillonées et partagées, périodiquement abîmées par les tests, modifiées aussi par les prises en compte successives des besoins, ce qui offrira une vision très limitée de ce que seront les données en production au final.

À ces contraintes techniques et organisationnelles du projet s'ajoute une dimension humaine non négligeable : le développeur est un expert dans son domaine mais ne connaît pas nécessairement le métier de l'utilisateur final.

Cette méconnaissance toute naturelle va alors conditionner ses tests. Concrètement, le développeur réalise "un bout de code", l'exécute de manière pragmatique avec un échantillon de données (qu'il choisit lui-même le plus souvent), puis observe le résultat obtenu. Il rectifie alors son travail et ainsi de proche en proche jusqu'à obtenir le résultat attendu... du moins ce qu'il en comprend.

## c. Types de tests techniques

Les experts du test, au cours d'une recette technique qui fait suite à une livraison de la MOE, disposent de toute une gamme de techniques que nous allons détailler dans cette section :

- Les tests nominaux
- Les tests aux valeurs clés
- Les tests aux limites
- Les tests hors domaine - ou non passants
- Les tests de robustesse
- Les tests de configuration matérielle
- Les tests de performance

Bien entendu, selon la maturité du développeur, de tels tests peuvent avoir été exécutés dans l'étape précédente de réalisation de manière unitaire. Mais rarement tous ensemble ne serait-ce que pour des raisons de temps

(après tout le développeur a pour première tâche de réaliser et non de tout tester) et peu probablement de manière systématique sur l'ensemble de l'application.

Que sont les tests nominaux ?

Par nominal, on entend que le test est réalisé dans une situation normale d'exécution, comme si le testeur était un utilisateur final réalisant une action habituelle. Les tests nominaux supposent donc une compréhension fonctionnelle de l'application. En général, un test nominal utilise donc un jeu d'essai qui a un sens, une pertinence métier.

À l'opposé, les tests aux valeurs clés ou les tests aux limites partent d'une considération technico-fonctionnelle, c'est-à-dire que les tests sont menés pour des valeurs spécifiques susceptibles de mettre en évidence un dysfonctionnement dans des cas particuliers :

- Pour une valeur nulle ou "spéciale" relativement à sa définition fonctionnelle (par exemple, la valeur vide, la valeur zéro ou la date du jour).
- Pour une valeur correspondant à une borne fonctionnelle du domaine de définition, par exemple, l'âge d'une personne qui ne pourra pas excéder 140 ans.
- Pour une valeur correspondant à une borne technique du support de stockage ou de la technologie utilisée (par exemple, la valeur maximum d'un entier géré par le PHP ou le JavaScript ou bien la valeur maximum stockée par MySQL, Oracle ou SQL Server).

L'objectif des tests aux limites est le plus souvent d'éprouver le comportement du logiciel pour des utilisations dans des "conditions normales", mais en partant du principe que l'utilisateur pourrait créer une situation non conforme aux règles du métier ou à une contrainte technologique.

Tout naturellement, les tests hors domaine ou non passants, sont destinés à s'assurer que les limites technico-fonctionnelles ne peuvent pas être franchies par l'utilisateur final. Ces tests s'apparentent donc très fortement aux tests aux limites si ce n'est qu'ils consistent à s'intéresser "à l'autre côté de la barrière", aux dépassements de capacité.

Par exemple, si un formulaire propose de saisir un montant d'au plus 100 euros (limite incluse), il est évident que le test aux limites de 100 euros doit être mené. Et pour le test non passant ? Doit-on tester 100,01 euros ou 101 euros ? Les deux peut-être ? Tout dépend de la nature de la donnée.

Globalement, les tests aux valeurs clés, aux limites et hors domaine nécessitent une connaissance à la fois fonctionnelle et technique et correspondent toujours à une utilisation normale de l'application.

En revanche, les tests de robustesse supposent une démarche purement technique : il s'agit de soumettre l'application à une utilisation inhabituelle, voire frauduleuse.

Les exemples suivants sont des tests de robustesse d'une application web mais cette liste est non exhaustive :

- Tenter d'injecter du SQL dans un formulaire d'authentification pour en casser le mécanisme.
- Désactiver le JavaScript ou les images et en tester le comportement dans un mode dégradé.
- Jouer avec le timeout des sessions.
- Faire une recherche d'éléments avec des critères utilisant des caractères spéciaux susceptibles d'être interprétés (par exemple, le %, joker du langage SQL).
- Modifier la barre d'adresse pour passer des URL frauduleusement.
- Saisir du HTML dans un formulaire et voir le comportement de l'application ensuite.
- Etc.

Le lecteur constate ici que le test mené n'a plus rien à voir avec une utilisation normale : il s'agit vraiment de tester le logiciel en ayant un point de vue techniquement extrême. "Sécurité d'abord" serait le mot d'ordre, l'angle d'attaque à avoir pour définir un test de robustesse.

Cette dimension technique des tests, nous la retrouvons alors dans les tests de configuration matérielle, c'est-à-dire que l'on considère que le logiciel étant le même, l'un de ses composants - généralement disponible sur le marché - est modifié dans sa version.

Pour les applications web, ce sera les versions des différents navigateurs, les versions des plugins (Flash Media Player, Adobe Reader...). Mais ce peut être aussi une montée de version du serveur web (Apache ou IIS), un connecteur ODBC vers une base de données, la version d'un driver...

L'objectif de ce type de test est de s'assurer d'une compatibilité, c'est-à-dire de la non-altération du logiciel par changement de composant ou de version de composant. Ou bien ces tests permettent de vérifier une portabilité (notamment les versions de navigateurs pour un site Extranet).

Il s'agit de tests techniques mais dont la vocation est fonctionnelle, voire organisationnelle, puisque des anomalies de compatibilité détectées, l'on dégagera une stratégie de contournement.

En effet, il n'est pas préconisé de développer un logiciel spécifique à une configuration matérielle, notamment pour les applications web. Ces actions sont évitées autant que possible. Et il est peu probable que l'éditeur du composant le corrige à votre demande...

En revanche, on dégagera des préconisations d'utilisation publiées et diffusées auprès des usagers ou/et des préconisations de développement pour contourner les effets indésirables.

L'organisation du projet est alors partiellement impactée puisqu'il est nécessaire de fournir la ou les configurations matérielles aux développeurs (idéalement) ainsi qu'à l'équipe de recette technique, pour vérifier le respect des préconisations en question. L'impact peut être aussi d'avoir à gérer une migration vers une version cible du composant pour lequel l'application est stable.

Enfin, une équipe de recette technique cherchera à soumettre le logiciel à un stress important dans certains cas de figures, afin de vérifier que la future montée en charge sera matériellement supportée.

Typiquement, ces tests consistent à jouer sur les volumétries des traitements pour les flux ou bien les montées en charge des connexions pour les applications frontales. Une application web pourra être distribuée par un serveur unique qui rencontrera alors un seuil au-delà duquel l'architecture technique ne pourra plus répondre normalement, ce qui nécessitera une distribution et une solution de "load-balancing".

De la même manière, les bases de données accédées seront évaluées sous l'angle des temps de réponse. La recherche d'index, la planification de jobs d'indexation automatiques... sont autant de mesures correctives mises en œuvre suite à un test de performance.

Notez toutefois que ces tests présentent une contrainte forte : il est nécessaire d'avoir une volumétrie de données similaire à la situation réelle de la production ainsi qu'une architecture matérielle voisine - idéalement identique - tant et si bien que ces tests sont très souvent organisés en environnement de pré-production.

#### **d. Tests d'intégration ou d'interface**

Un logiciel s'insère généralement dans un contexte plus vaste avec lequel il communique : le système d'information. Dès lors qu'il y a échange, communication, le test permettant de vérifier l'interopérabilité entre un logiciel et le monde extérieur avec lequel il interagit, est un test d'intégration ou d'interface.



Attention à ne pas confondre le test d'interface avec les tests techniques d'assemblage des briques qui composent le logiciel.

Typiquement, les tests des flux entrant et sortant (un import ou un export de données) sont des tests d'interface.

Nous distinguons ces tests de ceux menés lors d'une recette technique en raison d'une spécificité : le test d'intégration peut être tant technique que fonctionnel. C'est en fait le contenu même du test qui détermine si un test d'intégration est fait d'un point de vue purement technique ou avec une vision métier.

De ce fait, nous trouverons donc des tests d'intégration réalisés par une équipe de recette technique et nous pouvons trouver le même test d'intégration réalisé par une équipe de recette métier.

Par exemple, un export vers Excel peut être testé par un utilisateur final qui vérifiera que le fichier est conforme sous Excel. Mais le même export vers un fichier binaire ne pourra être testé que par un technicien du test.

## e. Tests de validation fonctionnelle

Une recette fonctionnelle n'a pas le même objectif qu'une recette technique, bien que les techniques pour l'organiser et les types de tests déroulés s'en rapprochent.

Elle se compose le plus souvent d'un échantillon :

- De tests nominaux
- De tests aux valeurs clés
- De tests aux limites
- De tests hors domaine - ou non passants

Mais surtout, une recette fonctionnelle est menée par la MOA du projet dans l'objectif de démontrer que la solution logicielle répond au besoin exprimé, le plus souvent en vue de signifier un "GO / NO GO". Les recettes fonctionnelles ont donc un objectif d'homologation avant tout.

Normalement, l'application a été préalablement testée par une cellule de tests techniques et le logiciel est censé comporter le moins possible d'anomalies résiduelles. De ce fait, la qualité relative du produit autorise une campagne de tests par des utilisateurs du métier privilégiés et ciblés, logiquement ceux qui ont été interrogés pour l'expression des besoins.

Une recette fonctionnelle ne comporte donc théoriquement pas de tests relatifs aux configurations matérielles ou de robustesse : la dimension technique des tests disparaît autant que possible au profit d'une vision fonctionnelle s'approchant au mieux de la réalité du métier. Ainsi, le jeu de données jouera un rôle prépondérant car il doit être le plus représentatif possible du réel.

Mais la recette fonctionnelle constitue également un moment important du projet sur le plan de la perception car cette étape permet d'apprécier la prise de contact entre un produit presque fini et les premières personnes concernées par son utilisation.

De ce fait, les tests de validation fonctionnelle constituent un enjeu stratégique à bien des égards et demandent une préparation spécifique demandant la validation de la MOA sur de nombreux aspects, le chef de projet MOA devant s'assurer que le logiciel développé s'insérera dans l'organisation du travail.

Retenez donc que la recette fonctionnelle demande une compréhension des processus du métier et une étroite collaboration avec les utilisateurs.

À l'issue de la recette métier, le chef de projet MOA est en mesure de valider la mise en production de la solution.

## 2. Et après le projet ?

### a. Tester la performance

Comme nous l'avons vu dans une section précédente, les tests de performance sont souvent tributaires des environnements matériels et de la volumétrie des données.

Le seul véritable environnement, image de la réalité, est celui de production. Il est bien sûr possible de le dupliquer le temps d'un test mais il est plus difficilement possible d'en maintenir des copies parfaitement synchronisées dans le temps.

De ce fait, l'environnement de développement, celui de recette technique comme celui de recette métier, est une image structurelle future du logiciel - parfois compatible ascendante, et simultanément une image dégradée de la production.

Cette dégradation provient essentiellement des volumétries nécessaires plus faibles dans les environnements de recette, mais aussi du fait des activités de test elles-mêmes qui dégradent les jeux de données - voire les détruisent irréversiblement.

Ce contexte technico-organisationnel constitue donc une contrainte forte peu propice à la mise en œuvre de tests de performance, raison pour laquelle ces derniers sont effectués directement en production ou en pré-production parfois.

Il en découle que la mesure des performances est une occupation constante une fois le logiciel déployé. Les temps de réponse des serveurs, du réseau, des modules d'accès aux données, sont formalisés, rapprochés et synthétisés dans des rapports d'analyse.

Ces mesures ne correspondent plus réellement à une notion de projet organisé au sens où nous l'entendions dans les sections précédentes. Elles ne feront donc pas l'objet d'une attention particulière dans le présent ouvrage : nous les évoquons ici pour mémoire.

### b. La maintenance : les tests de non-régression

Si la mesure des performances succède logiquement au déploiement d'une application en environnement de production, l'entrée en phase de maintenance de la solution nécessite alors de nouvelles recettes, qu'elles soient techniques ou fonctionnelles.

Nous entrons dans une logique de cycles au cours desquels des anomalies et des évolutions sont collectées.

Ces deux types d'événements nécessitent alors une nouvelle typologie de test : les tests de non-régression.

Concrètement, chaque correction ou maintenance est testée en tant que telle, via une batterie de tests techniques ou fonctionnels selon l'étape du projet concernée. Mais à cela doit s'ajouter un périmètre de tests dont l'objet est de s'assurer que le reste du logiciel ne régresse pas suite à cette modification.

L'élaboration d'un périmètre de non-régression demande alors au chef de projet recette technique et/ou métier de connaître l'évolution demandée pour l'adapter au contexte.

Ce peut être d'exécuter un scénario de test déjà écrit dans une recette précédente. Ce peut être aussi arbitrairement d'exécuter tous les tests d'une population de fonctionnalités jugées critiques.

Le présent ouvrage traitera nécessairement de ces aspects organisationnels des recettes lors des chantiers de maintenance.

### **c. Capitaliser le référentiel de tests**

Comme nous venons de l'évoquer avec les tests de non-régression, il est possible d'avoir à réutiliser un test prévu et rédigé dans le chantier de release, au cours du chantier de maintenance.

Ceci signifie que l'une des préoccupations du chef de projet MOA ou recette doit être la capitalisation du référentiel des tests et par extension, des jeux de données utilisés pour cela.

De nombreux outils du marché sont disponibles pour répondre à ce besoin. Mais au-delà de l'outillage, il y a un principe de réemploi permanent qui est sous-entendu dans l'activité de test.

Ainsi, de la même manière que des "bouts de code" peuvent être regroupés dans une bibliothèque commune à tous les projets développés au sein d'une organisation, les tests obéissent à la même logique.

Mais comme nous le verrons dans ce livre, la démarche de capitalisation est un peu plus complexe que pour les développements.

# Contextes de mise en œuvre des tests

Nous évoquerons sommairement les contextes organisationnels des principes évoqués dans la section précédente.

Car si les tests peuvent être organisés en différents points du cycle projet, les objectifs ne sont pas les mêmes à chaque fois et les stratégies diffèrent selon l'organisation.

## 1. Côté maîtrise d'œuvre

### a. Maîtrise d'œuvre externe

Une société qui souhaite développer une solution pourra faire appel à un prestataire de services externe pour réaliser un développement. Dans cette situation, la MOA de l'entreprise fait appel à sa DSI pour un projet et il est convenu que la solution consistera à trouver les ressources en dehors de cette dernière pour des raisons de savoir-faire ou budgétaires.

Les décideurs du projet devront alors se poser de sérieuses questions sur le choix du prestataire, et notamment vérifier si ce dernier dispose d'une cellule de qualification technique qui homologue sa production logicielle de manière plus ou moins indépendante de la MOE.

Deux cas sont possibles alors :

- Si le prestataire externe dispose d'une cellule de tests techniques, la MOA pourra avoir une relative confiance dans la qualité des livrables et n'aura pas nécessité à renforcer son propre effort de test en recette métier.
- Si le prestataire externe ne dispose pas d'une cellule de tests techniques, la MOA devra peut-être faire appel à une cellule d'experts en tests techniques en interne ou en externe chez un autre prestataire.

Du choix du prestataire et de l'historique de la relation commerciale avec lui découle alors une organisation du projet visant à compenser une faiblesse identifiée.

### b. Maîtrise d'œuvre interne

Lorsque la maîtrise d'œuvre est interne, la connaissance des compétences, l'historique de la relation avec la DSI, permet de connaître avec certitude l'organisation adéquate.

Si l'entreprise dispose d'une cellule de tests techniques indépendante de la MOE, la MOA n'aura pas nécessité à renforcer ses tests en aval du projet.

En revanche, une problématique de planning apparaît alors car la cellule de tests techniques pourrait ne pas être dédiée au projet mis en œuvre.

La stratégie de la MOA en politique de tests dépend donc exclusivement de son propre savoir-faire en test et de la planification du projet proprement dit.

## 2. Côté maîtrise d'ouvrage

### a. Maîtrise d'ouvrage externe ou interne non mature en tests

Il est rare qu'une maîtrise d'ouvrage soit totalement externalisée mais fréquent que des prestataires de services interviennent en assistance en maîtrise d'ouvrage, soit pour faire face à un pic de charge particulièrement élevé en

conception, soit pour faire appel à des compétences en matière de tests fonctionnels.

Il est en effet clair que l'organisation des recettes fonctionnelles s'est industrialisée et nécessite un véritable savoir-faire qui n'est pas toujours disponible au sein des MOA davantage focalisées sur les études avant-projet, l'expression des besoins ou la collecte des anomalies en production.

Ainsi, le prestataire qui organisera des recettes fonctionnelles pour la MOA devra connaître le contexte : la MOE est-elle externe ou interne ? Y a-t-il une cellule de tests techniques au sein de cette MOE ?

Ce contexte induira la nécessité - ou non - de renforcer les tests métier afin de compenser un manque en amont du projet.

Cette organisation présente un risque fort, d'autant plus si la MOE est simultanément externalisée et sans cellule de tests techniques puisque l'effort de tests ne pourra pas être équilibré entre les parties.

### **b. Maîtrise d'ouvrage interne mature en tests**

Contrairement à l'organisation précédente, une MOA mature en tests fonctionnels a la capacité de répondre à un manque de tests en amont du projet.

Peu importe qu'il y ait ou non une structure en charge des tests techniques, peu importe l'effort consacré aux tests par la MOE, une MOA expérimentée en tests pourra compenser l'effort. En fait, une telle MOA se substitue à une cellule de tests techniques ou presque.

C'est la connaissance de la MOE qui intervient et l'historique de cette relation qui détermine alors la stratégie de test globale à adopter côté MOA : le chef de projet MOA renforcera son équipe de recette métier pour élaborer une stratégie de tests plus forte s'il perçoit que la MOE n'a pas la capacité à répondre à l'effort de tests demandé.

## **3. Cas d'une cellule de tests indépendante**

### **a. Cellule externe à l'entreprise**

Que la MOE ou la MOA soient internes ou externes, peu importe : la cellule de tests techniques externalisée offre une situation de confort pour qui se positionne en chef de projet recette technique.

Cette organisation du projet repose avant tout sur l'expérience et la capacité à définir un périmètre des tests techniques pertinent relativement aux spécifications et aux livrables fournis par la MOA et la MOE.

Une cellule externe joue en effet le rôle d'arbitre entre MOE et MOA puisqu'elle n'a aucun intérêt à être de parti pris.

Si la MOA souffre d'un relatif manque de maturité et ne réussit pas à fournir des spécifications précises à sa MOE (ou la MOE externe), l'équipe de recette technique externe pourra lever les alertes sur les manques des entrants, amener des suggestions, et ainsi, permettre à la MOE et la MOA de converger vers la solution clarifiée.

Le chef de projet recette technique est alors porteur d'une solution gagnant-gagnant dans laquelle MOE et MOA trouvent un soutien mutuel.

### **b. Cellule interne à l'entreprise**

En revanche, le chef de projet recette technique qui serait dans la même entreprise que la MOA se trouve parfois dans une situation plus délicate.

Car l'indépendance de cette structure se définit en fonction du lien organisationnel et hiérarchique qui la lie soit à la MOE, soit à la MOA.

Si la cellule de tests techniques est interne à la DSI, elle sera chapeautée par un responsable commun à la MOE et distinct de la MOA.

A contrario, la cellule de tests techniques d'un projet pourra être constituée côté MOA - par exemple si la MOE n'avait pas les ressources pour le faire.

Enfin, il est également possible que la cellule de tests techniques soit indépendante en étant rattachée à une structure de gestion de la qualité séparée de la MOA et de la MOE.

Bref, si le contexte fonctionnel est confortable (il y a une cellule d'experts en tests techniques !), le contexte organisationnel de l'entreprise pourra provoquer un déséquilibre en faveur ou en défaveur de la MOA et réciproquement de la MOE.

Concernant la stratégie de tests techniques à mettre en œuvre, elle dépendra alors de la dimension de la structure et de sa capacité à répondre au rythme des livraisons de la MOE.

# Organisations possibles

Dès lors que nous parlons de tests unitaires, nous presupposons que ceux-ci sont réalisés en phase de réalisation d'un logiciel.

## 1. Le développeur livré à lui-même, seul ou en équipe

Qu'il soit une équipe à lui tout seul ou dans une équipe dans laquelle le chef de projet MOE n'est pas familiarisé avec l'organisation des tests unitaires, le développeur est dès lors livré à lui-même pour gérer ses tests. Cette situation doit être évitée autant que possible.

S'il est tout seul, le développeur fait alors office de chef de projet MOE très théoriquement (car il n'a pas nécessairement l'expérience de cette fonction) et doit organiser son travail de manière autonome.

Deux tâches permettront alors de cadrer cette organisation.

### a. Hiérarchiser son travail

Cette préconisation peut paraître triviale mais il nous a semblé nécessaire de l'indiquer comme prérequis à l'organisation des tests unitaires.

Le développement de la solution logicielle est généralement hiérarchisé, c'est-à-dire ordonné de manière cohérente en fonction des dépendances techniques du développement ainsi que d'une priorisation des fonctionnalités fournies par l'expression des besoins.

Nous conseillons de tester unitairement les développements en respectant cet ordre en commençant par les "bouts de code" mutualisés et réutilisés en plusieurs lieux du logiciel.

Une fois ces parties communes fiabilisées, passez aux couches logicielles appelantes. Cette organisation, logiquement, fera que le développeur commencera par tester les couches logicielles de bas niveau en premier (les accès aux données), puis les couches intermédiaires (middleware) pour en dernier tester unitairement la couche de présentation de l'information, l'interface homme-machine (IHM), s'il y en a une, ou bien les paramètres d'entrée, s'il n'y en a pas.

Puis réitérez ce principe brique après brique, jour après jour.

En suivant ainsi la structure du produit, le développeur s'appuie sur une logique claire et structurée qui permet de savoir à tout instant de manière simple où il en est et conjointement évite de tester plusieurs fois inutilement la même fonction.

Seule problématique : la faisabilité. En effet, pour tester un traitement d'accès aux données ou middleware de manière unitaire, il sera parfois nécessaire de développer une "interface technique" de test. Par exemple, pour tester un webservice, une interface graphique permettant de renseigner une question et visualiser une réponse pourra être développée pour s'assurer du bon fonctionnement du traitement.

Cette IHM ne fait pas partie de l'application mais son existence permettra avantageusement de mener des tests unitaires sans avoir besoin d'une application appelante.

### b. Mesurer le temps passé

Une deuxième tâche triviale consistera à mesurer le temps passé aux tests unitaires relativement au temps passé à développer. Généralement, le développeur réexécute périodiquement sa réalisation pour remonter les anomalies et rectifier son travail de proche en proche.

Cette démarche en apparence naturelle débouche cependant sur un risque : la redondance inutile de tests au détriment de tests non exécutés pourtant nécessaires. Car si ce comportement devient habituel, le test n'est alors plus réfléchi et devient un automatisme sans prise de recul.

Par ailleurs, le test unitaire peut alors s'avérer chronophage surtout si le jeu de données associé est détruit et nécessite systématiquement une nouvelle saisie. Il est alors important d'évaluer le temps passé à rédiger une fonction et le temps passé à l'exécuter afin que le ratio observé soit au moins de 20 % sans excéder 33 %.

Comptez en effet une demi-journée de tests pour une journée de développement au plus. Et pour sept heures travaillées à développer, comptez au moins une heure de test au minimum.

Il faut donc tester unitairement de manière équilibrée et dans la charge dévolue à cela, exécuter les principaux tests que nous détaillerons dans une section ci-après.

Bien entendu, lors de l'évaluation des charges de développement, le CP MOE aura pris en compte les tests unitaires dans son calcul, le plus souvent sur la base d'un ratio qui dépendra de la maturité de son équipe (nous pouvons partir du principe qu'une équipe débutante devra tester plus qu'une équipe aguerrie).

 CP est l'abréviation usuelle de "chef de projet".

## 2. Le travail en équipe

Le travail en équipe permet d'améliorer significativement l'organisation des tests unitaires. Pour cela le rôle du chef de projet MOE est fondamental.

### a. Sensibiliser l'équipe : le rôle du chef de projet

Dès que la phase de réalisation a débuté, le chef de projet MOE a pour tâche de sensibiliser son équipe aux tests unitaires.

En tant que chef de projet MOE, ce dernier est responsable du résultat de la production tout autant que du respect du planning et de la charge. Pour s'assurer de cette qualité de production, il est de sa responsabilité de mettre en œuvre une méthode de travail collaborative propice aux tests unitaires.

Le chef de projet MOE doit donc évaluer les "forces en présence" en portant une attention particulière aux difficultés rencontrées par son équipe. Pour cela, il devra estimer de manière factuelle la qualité des développements de chacun.

Vient ici une problématique humaine : comment vérifier le résultat d'un travail sans que celui qui l'effectue vive mal ce contrôle ?

Nous conseillons ici aux chefs de projet MOE de ne pas agir de manière opaque : il est préférable d'organiser un point technique avec votre ressource, point pendant lequel lui fera la démonstration du résultat de son travail puis vous donnera la main pour exécuter des tests, plutôt que de mener des tests sans le lui dire, puis venir souligner des défauts a posteriori.

Au-delà d'un problème de compétence en test, il est en effet possible que la compréhension du besoin fonctionnel soit telle que le développeur n'a pas totalement compris l'usage de la brique qu'il produit et donc la conçoit et la teste mal.

Une approche collaborative de cette sensibilisation aux tests est à privilégier, par exemple en réalisant des points

techniques périodiques pendant lesquels chacun montre aux autres ce qu'il a fait et comment il l'a testé et soumet à la critique des autres son travail.

Le partage des techniques de test entre les membres d'une équipe garantira aussi au chef de projet MOE que chacun connaît un minimum de ce qu'a réalisé un autre, stratégie d'homogénéité qui permet d'ailleurs de gérer des remplacements.

### b. S'assurer que les besoins en test sont couverts

Le chef de projet MOE doit également s'assurer que les outils et prérequis nécessaires aux tests unitaires ont été fournis au même titre qu'il s'assure qu'un développeur a ses outils de développement et son poste de travail.

Notamment, la problématique des jeux de données et de leur consommation devra être traitée : il est indispensable de s'assurer qu'à tout moment l'équipe de développement a la possibilité matérielle et normale de réaliser des tests.

La base de données de l'environnement de développement est en effet partagée par tous : dès lors qu'un développeur produit un jeu de données, ce dernier peut aussi bien être à son usage exclusif qu'être utilisé par un autre développeur si une dépendance forte existe entre les résultats de leurs deux productions respectives.

### c. Anticiper les dépendances de tâches

Tout chef de projet se doit de connaître les dépendances entre les tâches qu'il pilote. Ce détail trouve toute son importance dans les tests unitaires puisque les briques développées sont ordonnancées selon des dépendances techniques.

Nous avions préconisé dans une section précédente de hiérarchiser les tests selon le même ordre que les dépendances techniques : cela est d'autant plus vrai lors d'un travail d'équipe puisque la dépendance devient alors organisationnelle.

 Si vous demandez à deux développeurs de respectivement réaliser la création et la suppression d'une même donnée, il est évident que si la réalisation peut être parallélisée, le test le sera plus difficilement puisque la première fonction produit la donnée qui sera utilisée par la seconde. Dès lors, la répartition du travail entre les membres de l'équipe est impactée : le chef de projet MOE doit gérer une dépendance de tâche.

Comment s'affranchir des dépendances ? Idéalement, il semble évident de confier les deux tâches de développement à la même ressource qui testera de manière très logique les deux développements dans un ordre séquentiel.

Mais si cette répartition de tâches n'est pas possible, il y aura absolue nécessité à synchroniser les deux développeurs pour qu'ils puissent effectuer les tests ensemble de manière collaborative.

### d. Les tests croisés

Toujours dans le même esprit collaboratif, une technique qui a fait ses preuves est celle des tests croisés : à chaque développeur est confiée la tâche de tester unitairement le travail d'un autre membre de l'équipe.

Cette méthode offre de nombreux intérêts :

- Elle incite les équipiers à échanger et à se tenir informés de leurs travaux respectifs ce qui offre une liberté de mouvement en palliant au risque d'absence d'une ressource critique.
- Le principe fondamental du test est qu'on ne peut être juge et partie. Si la personne qui teste n'est pas celle qui a

réalisé, une indépendance et une forme d'objectivité est créée par l'apport d'un regard extérieur plus neutre.

- En outre, l'organisation de points techniques réguliers avec tous les membres de l'équipe pour montrer ce qui a été fait renforce sa cohésion et la confiance en sa capacité de production.

Les tests croisés permettent ainsi d'amener une grande synergie. Le fait de produire de manière imparfaite une brique logicielle est parfois vécu comme une incomptence professionnelle alors que se tromper est humain. En faisant tester par un autre une partie du logiciel (sans qu'il y ait relation hiérarchique) le chef de projet MOE peut "dédramatiser" une situation d'incompréhension.

Bref, retenez que pour l'organisation des tests unitaires, il est fondamental de privilégier le collectif à l'individuel et que le contrôle est ciblé sur le produit et non sur les personnes qui le réalisent.

# Tests unitaires élémentaires

Pour plus de détails, l'auteur vous renvoie à un précédent ouvrage pour ce qui est des tests unitaires élémentaires des applications web : "Tester une application web" aux Éditions ENI également, dans la collection Expert IT.

La présente section reprend de manière sommaire les tests unitaires pouvant être conduits sur une interface graphique, que celle-ci soit une application web ou un client lourd.

La section qui suivra se focalisera sur une autre problématique : tester unitairement un flux de données ou un traitement automatique sans interface graphique.

## 1. Tester un écran ou un formulaire

Nous parlerons d'écran pour les clients lourds et de formulaire pour désigner leur équivalent dans le monde des clients légers.

### a. Libellés statiques

Par libellé statique on entend toute partie textuelle de l'interface (écran ou page web) qui n'a pas pour origine une donnée issue d'une base et qui aurait un rôle métier.

Ce libellé est dit statique car il est relativement stable d'une exécution à une autre, tout au plus, il pourra être traduit par une fonction de changement de langue de l'interface. Il n'est d'ailleurs pas possible d'interagir avec lui.

Pour l'essentiel, il sera demandé aux développeurs de tester :

- Le respect de la charte graphique (police, graisse, couleur, style...),
- L'orthographe et la présence de caractères spéciaux,
- Le sens sémantique : le libellé doit avoir un sens pertinent.

Le test en lui-même consiste à exécuter l'application autant de fois que nécessaire pour en afficher tous les libellés statiques de tous les écrans possibles.

### b. Libellés dynamiques

À l'opposé des libellés statiques, les libellés dynamiques correspondent à une présentation d'une donnée issue d'une base. Le test consiste donc à exécuter l'application pour autant de lieu où une donnée dynamique est affichée et de simultanément lire la donnée par un autre biais pour en comparer les valeurs.

Des conversions peuvent être effectuées et un accès en base en lecture est fait à minima. Le développeur est donc ici tributaire de la correcte alimentation de la base de données, chose qu'il peut avoir réalisée lui-même... ou pas du tout !

Le développeur réalisera ici les mêmes tests que pour un libellé statique tout en s'assurant qu'il n'y a pas d'erreur de conversion ni de troncature des données.

### c. Champs cachés

Clients légers comme clients lourds utilisent parfois des champs cachés, graphiquement rendus invisibles et qui jouent le rôle de variable dans un traitement.

Ces champs n'étant pas accessibles aux recetteurs techniques et recetteurs métier qui suivront, nous recommandons des tests particulièrement méticuleux sur ces derniers.

Pour cela, il suffit de temporairement rendre visibles les champs en question puis d'exécuter l'application en vérifiant les valeurs stockées selon plusieurs cas de test.

-  Cas particulier des applications web : les balises HTML <input> de type hidden sont visibles dans le code source. Cette remarque ne s'applique pas aux clients lourds.

#### d. Liens et images cliquables

Ces liens sont spécifiques aux applications web mais sont parfois intégrés dans des clients lourds, par exemple pour définir un lien vers la société éditrice du logiciel ou une aide en ligne.

Leurs tests consistent pour l'essentiel à une simple exécution du lieu où les liens et images se trouvent. Un simple coup d'œil suffit donc mais l'appréciation pour un développeur peut être toutefois faussée :

- D'abord parce que son ressenti visuel est conditionné par le paramétrage de son écran (attention à proposer une définition d'écran minimale pour les tests comme pour les développements : 1024 par 768 ?).
- Ensuite parce que pour les applications web, les images ne sont pas nécessairement téléchargées mais directement lues sur son PC : il n'est pas possible d'apprécier le temps de chargement de la page comparativement à la même application distribuée en production.

Généralement, l'exhaustivité des liens est testée du simple fait que les liens et images cliquables définissent les navigations élémentaires au sein d'une application.

Mais le développeur d'une brique fonctionnelle pourra très bien définir un lien vers une fonctionnalité d'un module développé par quelqu'un d'autre, voire une autre équipe. Même chose pour les fichiers téléchargeables qui pourraient ne pas être disponibles dans leur version définitive au moment du développement.

De ce fait, les tests unitaires des liens et images cliquables ne sont pas toujours réalisables et nécessiteront des tests d'assemblage dans un environnement ultérieur.

Nous recommandons vivement au chef de projet MOE de recenser les éléments que son équipe ne pourra pas tester unitairement.

#### e. Champs alphanumériques

Les champs de saisie alphanumérique d'un formulaire ou d'un écran nécessitent les mêmes tests unitaires que les libellés dynamiques la plupart du temps, mais un intérêt important sera accordé à trois aspects par le développeur :

- La définition sémantique du champ.
- Les formats de présentation, de saisie et de stockage de l'information présentée.
- Le comportement cinématique du champ.

En effet, les champs alphanumériques sont utilisés pour saisir différents types de données : nom propre, raison sociale d'entreprise, numéro de téléphone, adresse électronique, code postal, SIREN ou SIRET, numéro de compte en banque, référence...

Ainsi la définition fonctionnelle du champ détermine les différents contrôles de domaine de validité à mettre en œuvre, contrôles souvent imprécis dans les spécifications.

Pour tester unitairement ces éléments, une connaissance fonctionnelle précise du champ est nécessaire afin de choisir un ensemble de valeurs de test qui garantira un taux de couverture nécessaire et suffisant.

Des tests nominaux devront être menés, et si possible des tests aux limites et hors limite aussi, par exemple, vérifier si les chiffres sont acceptés pour un champ "Nom de l'usager" ou bien s'il est possible de saisir plus de N caractères, de vérifier si des conversions automatiques sont gérées ou bien si la saisie des espaces à droite et à gauche est supprimée.

- Si un champ de saisie est un login, ne pas hésiter à faire des tests de robustesse, comme des SQL injection, pour vérifier que l'authentification est sécurisée.
- Si un champ de saisie est un mot de passe, prévoir un maximum de tests unitaires de modification du mot de passe et de tentatives d'authentification.

## f. Champ multilignes

Les champs multilignes sont testés de manière très simple, le plus souvent à l'aide d'une saisie au hasard de plusieurs caractères.

Nous conseillons toutefois de vérifier la saisie des caractères accentués ainsi que le nombre de caractères saisisable au maximum.

- Un test de robustesse pourra être anticipé en vérifiant que la saisie de balise HTML n'entraîne pas des comportements indésirables pour les applications web uniquement.

## g. Champs numériques

Les champs numériques se testent de la même manière pour les clients lourds comme pour les clients légers : le développeur devra s'intéresser à la définition sémantique du champ pour définir les domaines de validité de la donnée puis réaliser des tests nominaux, aux limites ou hors domaine afin d'éprouver le logiciel.

Le plus souvent, un test unitaire consistera ici à valider un formulaire ou un écran puis à vérifier que la donnée est correctement utilisée ensuite dans des contrôles, des calculs ou des actions.

Nous préconisons de tester la longueur des champs comme pour un champ textuel, puis des valeurs clés comme la valeur 0, des valeurs négatives ou réelles afin de vérifier comment se comporte l'application notamment avec les champs représentant des nombres entiers. Vérifier également si les caractères alphabétiques, espace inclus, sont refusés.

Le dépassement de capacité sera parfois plus difficile à tester surtout si plusieurs technologies sont employées puisque les bornes des nombres entiers ou réels ne sont pas les mêmes en Java, en SQL, en PHP, etc.

- D'une manière générale, nous recommandons au chef de projet MOE de bien définir la gestion des exceptions auprès de son équipe, dont le dépassement des capacités.

## h. Champs de type date

Ces champs sont certainement les plus difficiles à tester selon le cas. Nous recommandons au chef de projet MOE de traiter ce point avant tout développement :

- Définir le contrôle graphique de saisie des dates.
  - Utilise-t-on trois champs de saisie séparés ? un seul ? un contrôle Calendrier ? Utilise-t-on le format français ou anglais ?
- Définir une fonction renvoyant la date du jour, fonction commune à tout le projet et préconiser son emploi systématique.
  - Ne jamais utiliser directement un appel aux fonctions des dates système pour lire la date du jour.
  - Implémenter votre propre fonction point d'entrée unique renvoyant la date du jour lue soit sur le serveur de données, soit sur le serveur applicatif pour les applications web.
  - Pour les clients lourds, procéder de même en se rappelant que l'usager a peut-être les droits d'administrateur permettant de modifier la date système.
- Sensibiliser son équipe à la problématique de la chrono-dépendance : toute date future dans un jeu de données suppose que l'application ne pourra plus fonctionner ultérieurement car ledit jeu n'est pas pérenne.
- Anticiper un éventuel contexte international de l'application : si la date du jour est utilisée pour les contrôles, quel fuseau horaire est pris comme référence pour l'utilisateur de l'application ?

Le contrôle d'un champ date relativement à la date du jour introduit des problématiques technico-fonctionnelles que le chef de projet MOE doit anticiper.

À ce stade, entre la définition sémantique de la date, les contraintes techniques de stockage (dépassement de capacité spécifique à la technologie), l'architecture de l'application et le contexte international possible, il est relativement évident que des choix structurants doivent être faits : l'avis de la MOA est donc requis sur ces aspects.

 Nous reviendrons plus en détail sur les conséquences de la chrono-dépendance ultérieurement puisque l'organisation des recettes techniques et fonctionnelles pourra dépendre de ce type de contrainte.

Nous attirons également l'attention du lecteur sur la notion de période qui là aussi nécessite une batterie de tests unitaires mettant en scène deux champs de type date.

### i. Champs de type heure

Les mêmes considérations que pour les champs date peuvent être faites pour les champs modélisant une heure.

Les conséquences sont cependant moindres en cas d'anomalie : la plupart du temps les heures et minutes sont des données qui sont simplement affichées et ont rarement un impact fonctionnel.

### j. Champs de type numéro de semaine

Plus rares, les contrôles permettant de saisir un numéro de semaine pourront poser des problèmes, notamment si la parité de la semaine est utilisée.

Il existe en effet deux systèmes de numérotation des semaines dans le calendrier grégorien :

- Le système standard pour lequel le 1er janvier définit la semaine 1.
- Le système international ISO-8602:2004 pour lequel la semaine 1 démarre pour la semaine contenant le premier

jeudi de l'année.

Or ces deux systèmes de numérotation ne sont pas toujours synchrones : il y a des années pour lesquelles l'année est paire dans un système et impaire dans l'autre.

Là aussi, nous recommandons au chef de projet MOE d'anticiper cet aspect avec la MOA si cela s'avère nécessaire.

## **k. Listes déroulantes et non déroulantes**

Les développeurs se focaliseront principalement sur le caractère trié ou non trié de la liste, sur la gestion de la liste vide ou d'un seul élément et enfin sur la perte d'information lors des validations des formulaires.

Une liste déroulante peut aussi faire l'objet de filtre dynamique, une saisie clavier permettant de réduire les options immédiatement disponibles dans la liste.

## **I. Boutons radio, cases à cocher et autres composants graphiques**

Enfin, les cases à cocher et boutons radio feront l'objet de tests relativement similaires à ceux des libellés statiques.

D'une manière plus générale, deux démarches doivent être envisagées selon la technologie pour les composants graphiques :

- Les clients lourds, qui autorisent des cinématiques complexes entre composants graphiques et une grande interactivité, pour lequel le double-clic est un événement géré.
- Les clients légers, développés dans une technologie qui implique des restrictions sur ces interactivités, notamment l'absence du double-clic.

## **2. Tester un flux ou un traitement de masse**

L'import comme l'export de données présente une problématique de test unitaire spécifique : la masse des informations traitées peut être conséquente.

L'action de tester est en elle-même relativement simple puisqu'il suffit de lancer le traitement. En revanche, la vérification du résultat peut s'avérer fastidieuse puisqu'il faudrait théoriquement vérifier chaque ligne du fichier exporté (vers un tableur comme Excel bien souvent) ou chaque ligne de la ou des tables pour un import de données.

 Si le support physique de la source ou de la destination de l'import ou de l'export est plus complexe (par exemple des fichiers XML), la vérification de la conformité pourrait alors être particulièrement difficile.

Plus concrètement, le test unitaire consistera à faire une comparaison entre la donnée source et la donnée destination pour établir la validité du traitement.

Similairement, un traitement de masse (un batch) répond à la même problématique : l'exécution du job n'est pas en elle-même difficile mais il faut ensuite comparer ce qu'était la donnée avant et après traitement pour vérifier si le résultat est conforme. La différence réside ici dans le fait que la source et la destination sont le même support physique et que le traitement procède par écrasement.

Fort heureusement, nous avons la possibilité de réaliser une sauvegarde (back up) des données avant exécution donc le moyen de faire des comparaisons entre l'image avant et l'image après.

Nous allons voir maintenant quelle stratégie de test mettre en œuvre : elle s'établit en quatre temps.

### a. Pouvoir exécuter le traitement "à blanc"

Dès la conception, le chef de projet MOE prendra en compte une contrainte technique pour les traitements de masse.

Pour les exports, il s'agit le plus souvent de produire un fichier par lecture d'une base de données : il y a une relative liberté de réexécution puisque le fichier produit peut être copié, renommé et déplacé.

En revanche, les imports et les batchs effectuent des modifications ou des insertions d'enregistrements en masse le plus souvent dans une base de données, avec la contrainte parfois de ne pouvoir effectuer le test qu'un petit nombre de fois.

Dans un tel cas, il est nécessaire de pouvoir effectuer des exécutions "à blanc" c'est-à-dire qu'un paramétrage technique du traitement permet de ne pas écrire dans la base de données ou en tout cas de ne pas prendre en compte les modifications. Par exemple, pour une base de données sous SQL, un paramètre d'exécution indiquera qu'il faut faire un "rollback" après toute requête ce qui signifie que la mise à jour est annulée.

Ainsi le développement du traitement lui-même permet de faciliter le test unitaire.

### b. Vérifier la volumétrie

Comme nous venons de le voir, les tests unitaires des traitements de masse posent une contrainte forte de faisabilité.

L'exécution à blanc permettra de voir le résultat du traitement plusieurs fois sans impacter les données.

Une première action consiste alors non pas à vérifier les informations directement, mais afficher le nombre de lignes lues en entrée et le nombre de lignes créées, modifiées, supprimées ou rejetées en sortie, et plus généralement, produire des statistiques pour chacune des exécutions.

 Un fichier de rejets est constitué à partir des lignes lues en entrée et qui n'auront pas pu être traitées pour cause d'erreur. Le plus souvent ce fichier a la même structure que le fichier d'entrée (ce qui permet une réexécution sur le périmètre des seuls rejets ensuite) avec en plus une colonne indiquant le motif du rejet.

Il est évident que si le nombre de lignes en entrée est égal au nombre de lignes en sortie, alors nous pouvons déjà conclure qu'il n'y a pas eu une perte durant le traitement.

Bien sûr, nous presupposons ici que les statistiques des nombres de lignes lues et traitées sont calculées et retournées après exécution. Le chef de projet MOE est donc invité à clairement définir les indicateurs de mesure du traitement dans son dossier de conception technique.

Ce test a toutefois deux limites :

- Tout d'abord, l'altération des données n'est pas vérifiée : si un enregistrement est comptabilisé, cela ne veut pas dire que l'écriture est correctement réalisée dans le système cible. Seule une vérification champ à champ des deux enregistrements peut garantir la validité du test.
- Il peut y avoir de la redondance dans le fichier d'entrée qui fera que le nombre d'enregistrements traités est en réalité plus petit ce qui veut dire que la différence observée dans le comptage des enregistrements implique un résultat de test faux alors qu'en réalité le test est valide.

De ce fait, la comparaison des compteurs n'est jamais qu'un test préliminaire partiel : il faut donc aller plus loin.

### c. Tests par échantillonnage

Un test unitaire indispensable est ensuite mené et consiste à exécuter le traitement automatique en mode normal avec mise à jour de la base de données, mais sur une volumétrie aussi faible que possible.

L'objet d'un tel test est donc de valider le traitement totalement, notamment chaque champ mis à jour, chaque ligne insérée.

Pour cela, une seule stratégie possible : l'échantillonnage.

Concrètement, il s'agit ici d'identifier et cartographier des typologies de données élémentaires afin que toutes les situations fonctionnelles soient couvertes et ainsi assurer une complétude du test.

Puis, il s'agira de construire un ou plusieurs fichiers en entrée du traitement afin de procéder à des exécutions en petit nombre pour des volumétries limitées, et cela, afin d'effectuer un test de comparaison champ à champ et valider qu'aucune altération d'information n'est détectée.

Toute la difficulté réside alors dans la constitution de ces fichiers d'échantillons. Selon les cas, le développeur pourra les saisir manuellement.

Ou bien cette tâche ne sera pas humainement réalisable car chronophage et il faudra trouver une méthode automatique ou semi-automatique pour les générer.

Le rôle du chef de projet MOE est alors de se pencher sur cette problématique de tests unitaires en fournissant des données viables et surtout fonctionnellement pertinentes, tâche qu'il mènera à bien en se rapprochant de la MOA.

Dans un tel cas de figure, l'échantillonnage s'effectue par extraction d'une portion de la base de données de production le plus souvent, nécessitant éventuellement le développement d'un traitement spécifique "one shot" pour construire les fichiers d'entrée.

### d. Tests étendus, tests fragmentés

Là encore, sous réserve de faisabilité, une exécution globale à blanc du traitement pourra être lancée afin de traiter une grande volumétrie de données.

L'objectif du test unitaire n'est alors plus le même : il est de vérifier d'une part que tout le traitement passe (donc que la couverture des tests était de 100 %), et d'autre part que le temps de traitement de l'exécution est compatible avec une planification.

En effet, le traitement automatique de masse (import ou mise à jour) suppose qu'aucun utilisateur n'intervient manuellement et n'interagit avec la base de données qu'il traite à l'instant T.

Ainsi ce type de traitement est généralement exécuté la nuit en toute logique, en même temps que d'autres traitements planifiés et propre à l'activité habituelle de l'entreprise.

Chaque job dispose alors d'une fenêtre de temps et l'ensemble se doit d'être exécuté entre 21 heures et 6 heures du matin le plus souvent. Éventuellement cette fenêtre peut être élargie, mais les temps d'exécution peuvent être tels que même un élargissement ne permet pas au traitement de passer sur la totalité des données.

En conséquence, il est parfois nécessaire d'introduire un paramètre d'exécution qui permettra la fragmentation de

l'exécution en deux ou trois fois sur plusieurs nuits. C'est cette caractéristique technique du traitement que le développeur pourra alors exploiter pour des tests unitaires plus étendus en se limitant à un périmètre partiel des données mais d'une volumétrie suffisante pour en apprécier le comportement.

À ce niveau de test, nous effectuons pour ainsi dire une autre typologie de tests : le test de performance.

# Formaliser les tests unitaires

## 1. Dans un projet géré en V

### a. Pourquoi formaliser les tests unitaires ?

Dans une organisation d'un projet selon le cycle en V dite "classique", les tests unitaires sont la tâche ultime du développeur avant livraison de sa production à une cellule de tests techniques (ou à la MOA directement parfois).

Ce jalon du projet est particulièrement critique et cette livraison est caractérisée par un ensemble de livrables précis :

- Les sources de l'application et/ou l'exécutable associé et tout composant logiciel utile.
- Une fiche de livraison, document listant les items de ladite livraison.
- Parfois un guide d'installation ou/et d'exploitation.
- Idéalement, un dossier de conception technique détaillant la solution développée ainsi qu'un dossier d'architecture.

Une équipe d'exploitation installera le logiciel, le paramétrera, préparera aussi la ou les bases de données par extraction de la production... autant d'opérations techniques pour lesquelles le guide d'installation, le dossier d'exploitation ou le dossier d'architecture technique pourra être utile.

Mais l'étape qui suit la livraison est le déploiement pour une recette : seules l'application elle-même et la fiche de livraison ont un intérêt pour mener des tests.

La fiche de livraison notamment décrira le périmètre de ce qui est livré ce qui permet une première vérification relativement aux tests prévus : le CP Recette peut déjà avoir une première visibilité de couverture, notamment identifier les fonctionnalités absentes ou nouvelles qui présentent des risques plus importants.

En revanche, le CP Recette n'a alors pas de visibilité sur le périmètre des tests unitaires, ce qui a été fait et surtout ce qui n'a pas pu l'être.

La question se pose alors de savoir comment une équipe de développement et plus précisément le CP MOE peut communiquer en aval de la réalisation sur ce sujet.

Bien sûr, il n'y a pas toujours nécessité à faire cela et c'est le contexte du projet qui permet de savoir si cette communication est plus ou moins pertinente.

 Supposons qu'un CP MOE externe constate un manque de confiance de la MOA donc de son client. La communication du périmètre des tests unitaires est alors une manière d'apporter une réponse.

Enfin, nous avons pu constater notamment pour les tests unitaires des traitements de masse, que ces derniers peuvent être limités de par la contrainte technique que représente l'environnement de développement.

Il devient alors parfois vital de communiquer sur les limites rencontrées par la MOE lors de ses tests unitaires afin d'indiquer les points faibles déjà identifiés et permettre un renfort ultérieur des périmètres de test, un ciblage.

### b. Le rapport de tests unitaires

Une solution de formalisation des tests est le rapport de tests unitaires, document unique fourni à chaque livraison de l'application, dont la rédaction incombe à tous les développeurs de manière collaborative.

Ce rapport pourra être un document au format Word ou Excel et exposera les points suivants :

- Un rappel du contexte du projet
- La liste des fonctionnalités livrées, le rapport de tests unitaires étant nécessairement lié à une livraison
- Pour chaque fonctionnalité développée :
  - Les identités des développeurs
  - Les identités des testeurs unitaires
  - Les tests ayant été menés et leur statut (Passé, En erreur, Bloqué, Non applicable)
  - Les tests n'ayant pas été menés

Idéalement, chaque livraison comporte ainsi un rapport de tests unitaires qui met en visibilité les intervenants suivants.

Ce document se rédige en deux temps :

- Il est d'abord initialisé avant les développements - éventuellement rectifié pendant.
- Il est ensuite complété à l'issue de l'exécution des tests par chacun des développeurs.

De ce fait, le rapport de tests unitaires est nécessairement un document partagé entre les équipiers ce qui signifie qu'il faut gérer un accès concurrentiel en écriture parfois. Mais c'est là la seule contrainte technique à gérer.

Nous recommandons au CP MOE de transmettre ensuite ce rapport en ajoutant quelques préconisations spécifiques (par exemple, renforcer les tests sur tel ou tel module) en fonction des difficultés rencontrées par son équipe.

Pour faire ces recommandations, cela suppose bien sûr d'être au plus près des développeurs, donc de travailler idéalement en tests croisés, pour dégager les points faibles de la réalisation, notamment et surtout, le périmètre qui n'a pu être testé unitairement.

## 2. Validation en méthodes Agiles

Il existe d'autres organisations de projet que la méthode en V. Nous proposons ici d'aborder la même problématique des tests unitaires pour les méthodes Agiles.

### a. Rappel des principes des méthodes Agiles

Les principes des méthodes Agiles sont :

- De considérer que la production industrielle s'effectue de manière progressive, par microcycles en V successifs alternant les étapes de conception, réalisation et vérification sur une ou deux semaines.
- De mesurer la vitesse de travail de l'équipe et de la prendre en compte dans la détermination de l'objectif à atteindre.
- De favoriser la communication de l'équipe en utilisant un support de communication partagé (par exemple, un tableau d'affichage sur un mur) qui expose :
  - Ce qui reste à faire dans le microcycle en cours.

- Ce qui est en cours au jour J du cycle.
- Ce qui a été réalisé par les développeurs au jour J.
- Ce qui a été validé par les testeurs au jour J.
- De favoriser la synergie de l'équipe en mettant toutes les ressources dans un processus ritualisé de réunions :
  - Le planning game (PG), réunion débutant un nouveau microcycle et dont le rôle est de déterminer le périmètre du cycle à venir en s'appuyant sur la vélocité du cycle précédent.
  - Le stand-up meeting (SUM), réunion "debout" quotidienne, faite à la même heure, où tous les acteurs du cycle projet expriment ce qu'ils ont fait, ce qu'il reste à faire, les difficultés rencontrées.
  - La démo qui permet de montrer le résultat produit sur le cycle écoulé et d'évaluer la vélocité réelle.

Nous appuyons ici notre description sur la méthode agile SCRUM mais les autres méthodes Agiles seront relativement ressemblantes à notre description pourvu qu'il y ait un support écrit de travail (certaines méthodes Agiles sont purement orales). Il vous faut en retenir les principes dans leurs grandes lignes : une méthode itérative de développement.

Un microcycle ou "itération agile" débute par une réunion de planning au cours de laquelle le pilote ordonne une liste de stories sur la base des spécifications fournies par la MOA. Ces stories sont listées dans un fichier Excel appelé backlog et y sont classées par priorité décroissante.

Cette priorité est définie sur une échelle et elle permet de dire quelle story doit être traitée avant une autre. Plus elle est élevée, plus la story devra être traitée prioritairement.

Puis au cours du planning game, il est procédé à un vote auquel participent le pilote MOE, tous les développeurs et les testeurs, afin d'estimer la complexité de chaque story une à une.

Lorsque le vote n'est pas unanime, une argumentation rapide est menée pour expliquer pourquoi la complexité choisie diverge. Le consensus doit être trouvé. Les développeurs s'appuieront sur leur expérience pour estimer la complexité du développement et les testeurs sur la leur. Ce consensus est vraiment primordial puisqu'il est fédérateur de l'équipe.

La complexité est donc un nombre sur une échelle. Plus il est élevé, plus la story sera difficile à traiter.

La somme des complexités des stories d'une itération permet de définir une vélocité, c'est-à-dire une estimation de la difficulté à réaliser l'ensemble de l'itération. Plus la vélocité est élevée, plus l'équipe est capable de traiter un volume important de stories.

En conséquence, la structure des colonnes du backlog est la suivante :

- Un numéro de story
- Le numéro de l'itération de la story
- La description de la story, c'est-à-dire la tâche à réaliser
- La complexité
- La priorité

Le rôle des développeurs est bien sûr de réaliser les stories, c'est-à-dire chaque micro-tâche, les stand-up meetings étant l'occasion de se répartir le travail. Une story pourra d'ailleurs être coréalisées par plusieurs développeurs ensemble : il leur appartient de définir avec le pilote la répartition du travail oralement.

Mais peu importe qui fait quoi : seul le résultat collectif de l'équipe compte.

Les testeurs ont pour rôle de valider les stories développées. Théoriquement, ils sont issus de la MOA - puisqu'en méthodes Agiles tous les acteurs du projet sont réunis dans le même processus de production/communication.

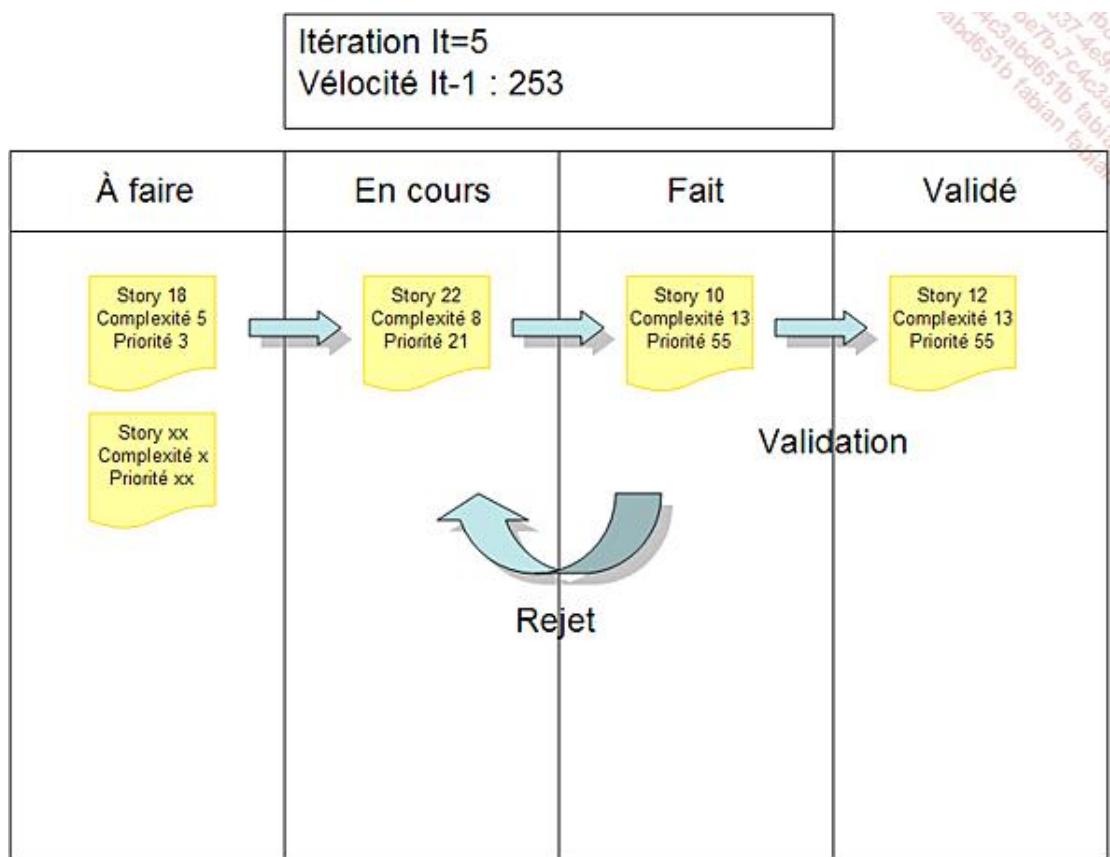
Ceci signifie donc que chaque story est une micro-tâche conduisant à réaliser une fonctionnalité ayant une cohésion fonctionnelle atomique : il ne s'agit donc pas de représenter une tâche technique puisque la story doit être un item à la fois compréhensible par la MOE et la MOA. La story a donc une dimension "fonctionnellement atomique".

Cependant, la validation fonctionnelle par la MOA n'est pas nécessairement possible pour ce mode de travail : elle doit être disponible quotidiennement pour les stand-up meetings et valider les stories.

Il est donc parfaitement envisageable de remplacer la MOA par une équipe de recette technique qui validerait les stories, la MOA effectuant une recette fonctionnelle a posteriori - ce qui constitue alors une digression relativement aux principes des méthodes Agiles exprimé en 2001 dans le Manifeste Agile.

## b. Le flux des stories

Nous pouvons nous représenter une itération de la méthode SCRUM en mouvement par le diagramme suivant :



Ce schéma représente ce que nous pourrions trouver sur le mur d'une pièce : un tableau affichant le flux des micro-tâches, des stories, sous la forme de post-its déplacés de colonne en colonne en fonction de leur avancement.

Concrètement, au cours de la journée, chacun des développeurs prend une ou plusieurs stories. Seule règle, respecter l'ordre des priorités de réalisation.

Il place alors le post-it dans la deuxième colonne et le transfère dans la troisième dès lors qu'il a achevé sa réalisation.

Les testeurs viennent conjointement chaque jour dans le bureau et regardent la troisième colonne pour savoir ce qu'ils ont à tester.

Les tests de validation de chaque story sont alors exécutés et si les résultats sont valides, la story est déplacée par eux dans la dernière colonne. Si une anomalie est détectée, la story retourne en colonne 2.

À l'issue de l'itération, une démo est préparée sur la base des stories présentes dans la colonne de validation. L'équipe effectue alors une vérification collective globale en exécutant toutes les stories validées puis un vote a lieu pour noter l'itération : le pilote dégage alors les plus et les moins, les questions et les suggestions pour mieux travailler dans l'itération suivante.

C'est aussi l'occasion de mesurer la vélocité réellement atteinte par l'équipe car il est possible que toutes les stories n'aient pas été traitées - auquel cas elles sont décalées sur l'itération suivante.

### c. La formalisation des tests dans le backlog

Comme nous l'avons décrit précédemment, un fichier backlog unique et partagé par toute l'équipe est rédigé au format Excel, une ligne du fichier étant une story attachée à une itération précise.

La colonne description décrit ce qu'il faut réaliser. Mais elle ne décrit pas ce qu'il faut tester.

Nous vous proposons d'améliorer le backlog en ajoutant une nouvelle colonne "Cas de test" dans laquelle le pilote et la MOA écriront collectivement les tests à dérouler pour chaque story, ainsi qu'une colonne "Commentaire".

Ceci offre un double intérêt évident :

- Tous les tests à réaliser sont décrits dans le backlog accessible aussi bien par les développeurs que les testeurs : les développeurs pourront donc mieux comprendre la story développée, et les testeurs sauront comment valider chacune d'elles.
- Contrairement à la méthode en V qui nécessitait à la fois des spécifications et un rapport de tests unitaires, le backlog centralise les deux fonctions sur un seul support.

 Attention ! Il ne s'agit pas d'écrire dans le backlog le résultat des tests unitaires dans la zone commentaire. On pourra introduire en revanche une colonne supplémentaire "Itération de validation" pour signifier qu'une story a été validée et rattacher les stories non validées à une itération de validation future.

Le backlog n'a alors pas pour fonction de formaliser ce qui marche et ce qui ne marchera pas puisque dès qu'une story est en anomalie, le stand-up meeting permettra aux testeurs et aux développeurs d'échanger oralement sur le sujet, de reproduire l'anomalie et ainsi aider à la correction.

Le backlog sert uniquement à lister quels tests sont faits pour quelle story, donc un périmètre d'acceptation.

### d. Validation des stories et périmètre des tests après livraison

Une fois l'itération achevée, un lot de stories a été validé et l'application nouvellement modifiée pourra être livrée par la MOE. Mais à qui et pourquoi ?

Les limitations de l'environnement de validation - qui en général est le même que celui des développeurs - permettent-elles directement une mise en production ? Est-ce la MOA qui est intervenue dans le processus Agile

ou bien une équipe de recette technique ? La nouvelle application produite ne nécessite-t-elle pas des tests d'assemblage et d'intégration plus complets ?

Toutes ces questions conditionnent par leurs réponses l'étape qui suit la livraison par la MOE. Cependant, quels que soient les intervenants qui suivent, ils disposeront d'un backlog listant les tests réalisés avec la MOE.

Le rôle du backlog est donc particulièrement important si l'on s'en sert comme référentiel de tests unitaires.

# Risques projet liés à la gestion des tests unitaires

## 1. Impact sur la charge

### a. Vers une augmentation de la charge de réalisation ?

Comme nous l'avons évoqué dans une section précédente, le chef de projet MOE doit anticiper lors de son évaluation de charges qu'une part de l'activité sera consacrée aux tests unitaires : nous avions donné à titre indicatif le ratio de 20 à 33 %.

Bien évidemment, la formalisation des tests unitaires nécessite de rédiger un support (soit un rapport de tests unitaires, soit d'enrichir le backlog en Méthode SCRUM) ce qui représente un effort supplémentaire.

Mais il serait faux de croire qu'en s'affranchissant de tester l'on gagnera du temps car l'absence de tests unitaires conduira à trouver des anomalies ultérieurement ce qui coûtera plus cher au global.

On pourra argumenter alors que l'effort de tests est fait ailleurs et sur une enveloppe budgétaire pour laquelle le CP MOE n'a pas de responsabilité... mais c'est un débat stérile que nous n'aborderons pas ici. Le principe de base est que toute économie de charge doit être réalisée le plus tôt possible et qu'un projet est géré par une équipe unique dans un objectif commun.

 Même si vous ne travaillez pas en méthodes Agiles, nous vous invitons à mettre en œuvre les valeurs prônées dans le Manifeste Agile : elles ne sont jamais que du bon sens.

Il existe donc un plancher de charge en dessous duquel le CP MOE sera en sous-qualité et cela n'est pas acceptable.

### b. Prendre le risque de la sur-qualité

À son opposé, trop tester n'est pas non plus pertinent : ce serait faire de la sur-qualité.

D'une part parce que les équipes de tests qui interviennent après la phase de réalisation assurent le rôle de garde-fou et sécurisent l'application selon d'autres aspects. D'autre part parce qu'un CP MOE se doit de respecter coût et délai et qu'il ne lui sera jamais demandé de faire un produit idéalement parfait.

Il y a donc un plafond à la charge à accorder aux tests unitaires, et ce plafond résulte d'un compromis et du contexte du projet.

Nous avons vu que la maturité de l'équipe joue sur la charge à accorder aux tests unitaires. Mais d'autres facteurs entrent en ligne de compte, comme la technologie. Un outil mature et stable (un framework de développement par exemple) demandera moins de tests que le même outil dans une version encore en bêta test. Il y a donc une prise de risque à évaluer et à prendre, puis trouver un ratio entre développement et tests unitaires qui soit cohérent.

## 2. Impact sur l'équipe

### a. Une meilleure cohésion, de meilleures performances

Que l'organisation des tests unitaires soit faite en méthode Agile ou en V par tests croisés, le résultat d'une bonne stratégie de tests unitaires conduit à une meilleure synergie entre les développeurs.

Le partage des connaissances et des points de vue facilite la prise de recul en ayant le mérite de poser collectivement les problèmes de compréhension face à des règles fonctionnelles métier ou des considérations techniques.

Il en découle logiquement de meilleures performances d'autant qu'il est plus facile pour un développeur d'échanger sur une anomalie détectée par un autre technicien qu'avec un utilisateur final qui ne connaît rien à la technologie.

Mais au-delà de la performance gagnée au sein de l'équipe des développeurs qui se partage les tests unitaires sur un même support formalisé, le CP MOE dégage aussi une amélioration du processus global en aval puisque les intervenants du projet qui suivront sauront ce qui a été fait et vers où porter l'effort de tests.

## b. Prendre le risque de démotiver l'équipe

La formalisation des tests unitaires représente cependant un risque : la non-adhésion de l'équipe à la démarche.

Car une documentation trop lourde, une organisation trop contraignante, pourra être mal perçue par des techniciens qui ont le plus souvent un profil qui les incite à la création et moins à la vérification.

Le rôle du CP MOE est donc fondamental mais la méthodologie et le contexte du projet tout autant.

Par exemple, si une pression ponctuelle est nécessaire (un "coup de collier") pour atteindre un jalon en terminant une tâche en travaillant un peu plus tard que d'habitude, il sera difficile d'avoir simultanément une exigence trop élevée de formalisme dans les tests.

De même, nous avions évoqué le risque de sur-qualité précédemment. Celui-ci a alors une seconde incidence : afficher un manque de confiance en son équipe. Un trop grand nombre de tests pourra être compris comme une défiance sur le résultat de la production.

S'en suit alors une démotivation de l'équipe qui se sent trop contrôlée.

-  L'exemple décrit ci-après est la description d'une situation réelle. Une SS2I anglaise de grande taille possédait une filiale dans le Maghreb. Il fut décidé de mettre en place Quality Center pour gérer les tests unitaires et conjointement de détailler les spécifications de manière extrême, allant jusqu'à écrire les requêtes SQL des futurs traitements. Les ingénieurs se sentant ainsi dévalorisés, communiquaient les résultats de leurs tests unitaires en faisant des captures écrans du résultat de l'exécution desdites requêtes : ils se comportaient en exécutant. Coïncidence ? Le turn-over de cette entité était alors de l'ordre de 40 % avec pour incidence la nécessité de former en permanence de nouveaux venus sur QC... ce qui n'était pas gratuit !

# Périmètre fonctionnel

Le présent chapitre s'attachera à définir les notions de périmètre de tests et de stratégie de recette, que celle-ci se déroule directement en sortie de développement lors d'une recette technique ou plus en aval au niveau d'une recette fonctionnelle, toujours dans le cadre d'un chantier de release.

Nous aborderons ces notions en partant d'un sujet plus large que celui du test : l'analyse de risques. Cette démarche doit normalement être menée par l'équipe MOA en charge des études avant-projet et cela, bien avant le démarrage de celui-ci, comme nous l'avions précisé dans le chapitre Généralités qui exposait des généralités sur le processus de production logicielle.

Mais cette tâche n'est pas obligatoire : l'analyse de risques n'est pas toujours réalisée. Malheureusement...

## 1. Parties prenantes, risques et exigences

### a. Lister les parties prenantes

Par partie prenante d'un projet, nous entendons une liste des types d'acteurs du projet et/ou de la solution logicielle produite.

Nous rencontrons donc des parties prenantes spécifiques à un projet (les développeurs, les chefs de projets MOE et MOA, les équipes de recette...) et des parties prenantes concernées par le produit réalisé, les types d'utilisateurs (par exemple, un administrateur, un contributeur et un rédacteur en chef pour une application de gestion de contenus).

Une partie prenante peut d'ailleurs être concernée avant, pendant et après le projet suivant son niveau d'implication. Elle peut être dans l'entreprise qui réalise le projet, comme elle peut être un anonyme à l'autre bout de la planète.

Quoi qu'il en soit, cette étape permet d'identifier toutes les typologies de personnes interagissant à un moment ou un autre avec le projet et/ou le produit.

### b. Lister les risques

Une fois les parties prenantes identifiées et réparties selon qu'elles sont impliquées avec le projet ou le produit, s'en suit une phase pendant laquelle le responsable de l'étude va identifier tous les risques dans la mesure du possible.

Deux familles de risques sont distinguées :

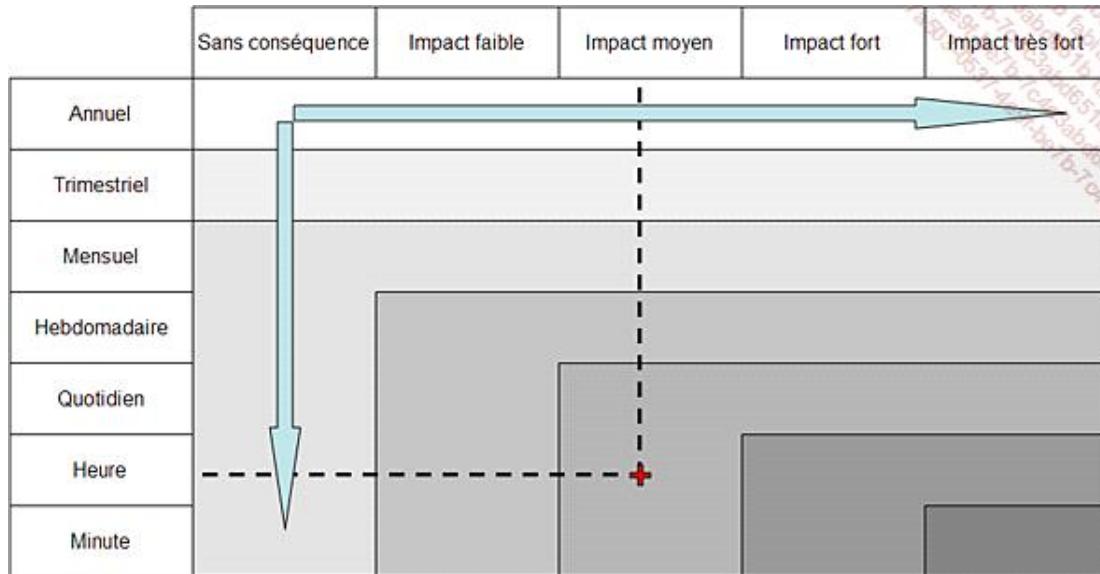
- Les risques sur projet, qui concernent les parties prenantes impliquées dans le processus de production logicielle.
- Les risques sur produit, qui concernent les parties prenantes qui interagiront avec le logiciel dès lors que celui-ci leur sera accessible en environnement de production. Ces risques-là n'existent donc pas pendant le projet, mais apparaissent à son issue.

 Il est très important de ne pas confondre ces deux notions et en général d'éviter de faire une confusion entre le projet et le produit qui est son résultat.

Cette phase d'identification constitue la première étape d'une analyse de risques. Ces derniers sont ensuite catégorisés : ils peuvent être techniques, financiers, organisationnels, contextuels, juridiques, sociaux...

S'en suit alors une valorisation des risques amenant à les classer selon une matrice de risques :

- Selon un axe de probabilité de survenance (chaque minute, chaque heure, quotidienne, hebdomadaire, mensuelle...).
- Selon un axe de gravité d'impact (sans conséquence, impact faible, impact moyen, impact fort, très fort...).



L'analyse de risques consiste ensuite à définir les parades associées à chacun, c'est-à-dire des actions permettant d'éliminer ou réduire le risque, l'éviter en mettant en place un scénario de contournement... ou bien le prendre !

Le responsable de l'étude avant-projet identifiera aussi les points critiques, c'est-à-dire les moments du projet où on connaît un pic maximum de probabilité de survenance d'un risque.

Et une fois démarré, les chefs de projet MOA / MOE et Recette devront suivre ces risques dans leur domaine respectif et "faire vivre" cette matrice qui n'a rien de statique.

Nous ne détaillerons pas davantage ce processus d'analyse de risques et nous préférerons renvoyer le lecteur sur l'abondante littérature relative à leur gestion, domaine qui est plus large que celui de l'organisation des tests.

Il est évident que les risques sur le projet peuvent impacter la recette qu'elle soit fonctionnelle ou technique puisque ces étapes en font partie.

En revanche, les risques sur le produit intéresseront davantage les équipes de tests comme nous allons le voir dans les sections qui suivent, car ils définissent très clairement les lieux de l'application à produire où la survenance d'une anomalie est susceptible d'avoir un impact.

Ainsi, nous parlerons de "matrice des risques projet" et "matrice des risques produit", ces deux matrices intervenant de manière différente dans le projet.

La matrice des risques projet sera un outil décisionnel utilisé par l'instance de pilotage du projet. À la fin du projet, cette matrice sera pour partie obsolète, tout du moins pour les risques qui disparaissent avec la fin du projet : seuls subsistent les risques "post-projet".

La matrice des risques produit sera elle aussi un outil décisionnel, mais elle aura un usage différent puisque c'est une fois le projet fini qu'elle trouve un intérêt plus important et doit impérativement être maintenue.

- La fréquence de survenance d'un risque produit varie dans le temps par définition puisque le logiciel montera progressivement en puissance entre le premier jour de son utilisation et le moment où sera atteinte une vitesse de

croisière. En revanche, l'impact est le plus souvent constant.

C'est cette matrice des risques produit qui intéressera le chef de projet recette pour définir un périmètre de tests.

Un risque produit signifie en effet qu'il existe une probabilité de défaillance du logiciel. La parade à mettre en œuvre pour chaque risque produit est donc une batterie d'actions de vérification, ou tests logiciels, afin d'éprouver le produit dans son usage réel.

Définir la matrice des risques sur produit revient donc à définir un périmètre de tests dans une démarche que l'on appellera *Risk Based Testing* ou RBT.

### c. Lister les exigences

Nous pouvons comprendre le terme exigence comme une contrainte ou un besoin selon le contexte - parfois les deux ensembles - une contrainte étant définie comme un besoin impératif, notion qui se traduit en anglais par le terme *requirement*. Par exemple, avoir besoin de 10 postes de travail est une exigence du projet pour répondre à une contrainte logistique.

Un autre exemple serait de dire que le logiciel futur permette de réaliser un virement bancaire - nous aurions alors une exigence fonctionnelle du produit - et simultanément d'être sécurisé comme le reste de l'application de banque en ligne où cette fonction est accessible - nous aurions alors une exigence technique de fiabilité du produit dans sa globalité.

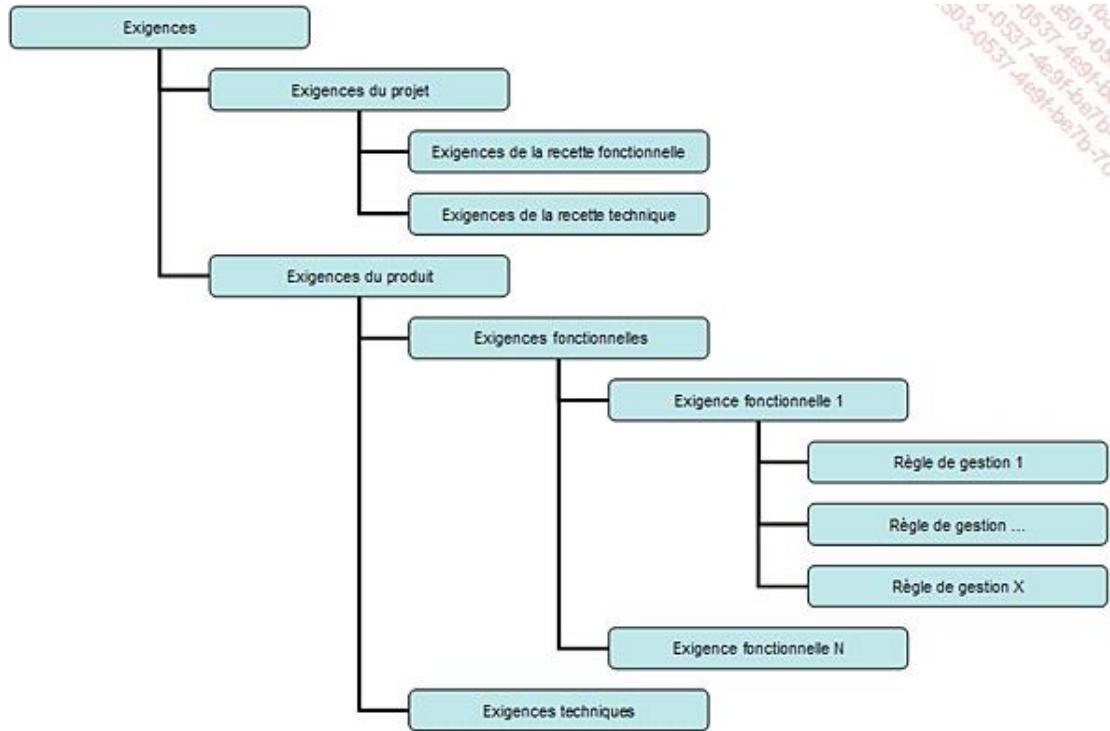
De ces exemples, nous comprenons que le mot exigence désigne de nombreuses choses au sein d'un même projet et que seul il ne veut rien dire : une hiérarchie des exigences est donc nécessaire dont la base est la distinction entre exigence projet et produit.

Les exigences du projet se décomposeront alors en fonction des différentes phases de son exécution : exigences organisationnelles, logistiques... dont celles qui concernent l'équipe en charge des tests techniques et celle en charge des tests fonctionnels.

Quant aux exigences du produit, elles répondent à une typologie différente mais nous pouvons déjà distinguer deux familles :

- Les exigences fonctionnelles et les règles de gestion à implémenter, c'est-à-dire une décomposition structurelle du produit à réaliser, une description du but.
- Les exigences techniques, c'est-à-dire les qualités spécifiques auxquelles doivent répondre toutes les parties du produit.

Le schéma ci-dessous reprend cette classification :



La démarche descriptive du projet que nous venons d'aborder ici se nomme "*requirement analysis*" en anglais ou "analyse des exigences". Conjointement à l'analyse des risques, elle constitue une autre manière d'aborder une étude avant-projet en recensant non pas une probabilité de survenance d'un événement désagréable, mais au contraire ce qui est souhaité et souhaitable.

Autre différence notable, autant une analyse des risques n'est pas toujours menée (ou bien très succinctement), autant une analyse des exigences est incontournable.

Le chef de projet recette s'intéressera donc par nécessité au résultat de cette étude qui d'ailleurs définit le contour des spécifications fonctionnelles - du moins la sous-arborescence des exigences fonctionnelles.

Le lecteur comprend maintenant qu'il peut exister une relation entre risque et exigence : les sections qui suivent vous présenteront le rapprochement de ces deux concepts selon une méthodologie d'analyse conjointe des risques et exigences, la méthode *Risk and Requirement Based Testing* (RRBT).

## 2. Hiérarchiser le périmètre fonctionnel

### a. Qu'est-ce qu'une criticité ? Qu'est-ce qu'une priorité ?

Avant d'effectuer le rapprochement entre analyse de risques et analyse des exigences, il convient d'être plus précis sur deux définitions souvent confondues : la criticité et la priorité.

L'une comme l'autre s'exprime sur une échelle. Mais l'une n'est pas synonyme de l'autre et surtout, leur méthode d'évaluation diffère totalement.

La criticité se chiffre selon la formule suivante :

$$\text{Criticité} = \text{Fréquence} * \text{Impact}$$

Où l'impact et la fréquence sont mesurés ou estimés par des études statistiques en interrogeant des populations : les parties prenantes. Certaines parties prenantes pourront être totalement interrogées, d'autres partiellement (le

plus souvent des représentants) et enfin certaines ne pourront pas l'être (par exemple, le public pour un site Internet).

Le calcul de la criticité s'effectue ensuite grâce à la matrice des risques que nous avons vue précédemment :

	Sans conséquence 0	Impact faible 25	Impact moyen 50	Impact fort 100	Impact très fort 200
Annuel 1	0	25	50	100	200
Trimestriel 4	0	100	200	400	800
Mensuel 12	0	300	600	1 200	2 400
Hebdomadaire 52	0	1 300	2 600	5 200	8 400
Quotidien 365	0	9 125	18 250	36 500	73 000
Heure 8760	0	219 000	438 000	876 000	1 752 000
Minute 525 600	0	13 140 000	26 280 000	52 560 000	105 120 000

Dans notre exemple ci-dessus, nous supposons qu'une anomalie sur produit occasionne une erreur de 0, 25, 50, 100 ou 200 centimes d'Euros. Nous remarquons alors que des équivalences peuvent être soulignées.

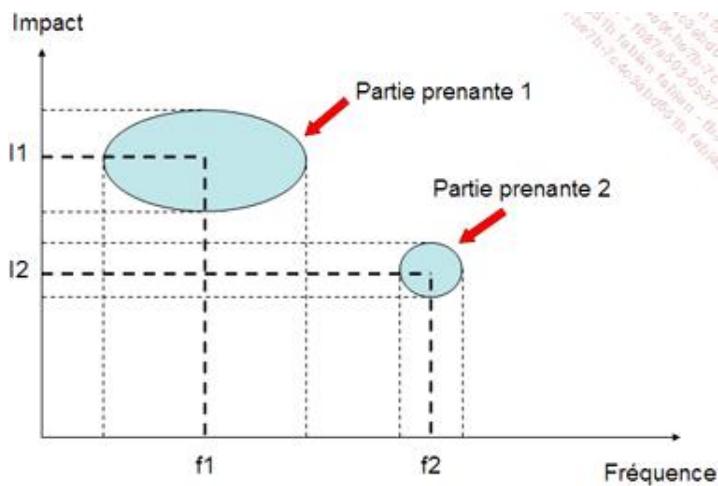
Une criticité sur un risque produit doit donc être comprise comme la mesure de l'impact d'une éventualité future.

Cet impact pourra être financier, visible de l'extérieur ou non (image de marque), juridique..., c'est-à-dire négatif.

→ Textuellement, il s'agit ici de risque aryétique (du grec aryetos qui signifie négatif) et non pas de sa définition scientifique qui se rapproche davantage d'une vision prenant en compte tant les aspects positifs que négatifs de la survenance des événements.

Dans notre exemple, nous nous sommes limités à un impact de type financier afin de ne pas alourdir la matrice, mais en dédoublant les colonnes impact, vous pourriez ajouter un type d'impact Interne/Externe et envisager toutes les combinatoires afin d'avoir une grille totalement valorisée.

À cela s'ajoute une dimension statisticienne de représentativité des parties prenantes interrogées. Pour un risque identifié, chaque personne interrogée exprime une évaluation de la fréquence et de l'impact que l'on pourrait reporter sur un nuage de points :



Tout d'abord, pour deux individus issus de la même partie prenante, il y a une forte probabilité que leurs estimations soient plus ou moins voisines ce qui conduit à des nuages de points plus ou moins denses. Dans notre illustration ci-dessus, les personnes interrogées de la partie prenante 2 ont été plus précises que celle de la partie prenante 1. S'en suit logiquement un calcul de moyenne qui donnera une estimation  $(i_2, f_2)$  pour la partie prenante 2.

Mais un même risque peut avoir des impacts et fréquences différents selon la ou les personnes interrogées : il faudra donc s'attendre parfois à avoir plusieurs nuages de points et plusieurs estimations possibles, généralement fonction des parties prenantes elles-mêmes.

S'en suit alors une stratégie de choix de valorisation de la criticité des risques :

- La stratégie globale, qui consiste à prendre en compte les impacts et fréquences de toutes les parties prenantes prises individuellement.
- La stratégie pessimiste, qui consistera à prendre l'impact maximum et la fréquence maximum des différentes parties prenantes identifiées (dans notre exemple, ce serait le couple  $(i_1, f_2)$ ).
- Une stratégie barycentrique, qui consistera à pondérer chaque partie prenante par un coefficient marquant son importance dans le projet.
- Une stratégie restrictive, qui consistera à privilégier l'avis d'une partie prenante jugée prédominante pour la représentativité du risque.

Comme vous le constatez, l'analyse de risques avec la valorisation des criticités peut devenir très complexe et surtout très lourde, raison pour laquelle cette étape n'est pas systématiquement réalisée. La taille du projet motivera logiquement les outils à mettre en œuvre. Le savoir-faire de la MOA aussi.

En conclusion, retenez que derrière les notions de criticité et de risque se cache une vision statistique tant pour le projet que pour le produit.

La notion de priorité en revanche n'est pas du tout la même : il s'agit d'une échelle qui indique quelle tâche doit être traitée avant telle autre pendant une phase précise du projet. Et dans le cadre plus précis d'une activité de recette, la priorité d'un test détermine donc l'ordre des tests à réaliser puisque le test est l'action mise en œuvre pour couvrir un risque produit. Un même test peut d'ailleurs avoir des priorités différentes selon qu'il est réalisé en phase de recette technique ou en phase de recette fonctionnelle.

La priorité des tests définit donc plus ou moins l'ordonnancement des actions de vérification du produit et en aucun cas une vision statistique des risques de défaillance de ce dernier. Nous envisagerons ici que cette priorité prenne les valeurs suivantes :

- Ce qui doit être testé impérativement : le niveau Must.
- Ce qui peut être testé : le niveau Could.
- Ce qui devrait être testé : le niveau Should.
- Ce que l'on souhaiterait tester : le niveau Would.
- Ce qui ne sera pas testé : le niveau Ignore.

La granularité de cette échelle est volontairement grosse, mais nous aurions pu choisir un niveau de finesse plus important avec des valeurs entières.

Il y a bien sûr une relation mathématique logique entre priorité d'un test et criticité d'un risque produit : il est évident que l'on testera prioritairement les éléments pour lesquels le risque en regard est d'autant plus critique.

Mais ce serait oublier que cette priorité peut fluctuer pendant le projet et que cette notion n'a plus de sens une fois qu'il est terminé. Alors que la criticité d'un risque produit persiste au-delà du projet et aussi longtemps que le logiciel est utilisé.

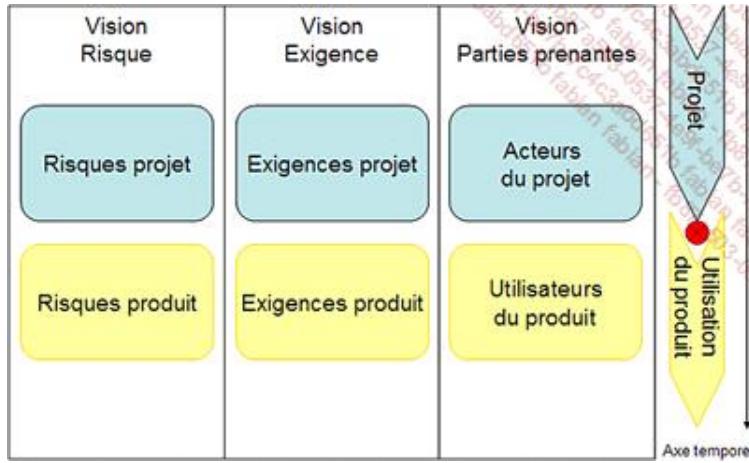


La criticité d'un risque produit fluctue elle aussi, mais avec l'usage du logiciel dans le temps puisque la probabilité d'apparition d'une anomalie augmente avec la fréquence d'utilisation du logiciel.

Nous invitons le lecteur à se souvenir de ces deux définitions autant que possible et de toujours se rappeler que risques et exigences sur projet précèdent toujours dans le temps les risques et exigences sur produit. Les confusions sont fréquentes.

## b. Rapprocher les risques des exigences

La méthode d'analyse conjointe des risques et exigences consiste à effectuer un rapprochement entre d'un côté les deux matrices des risques pondérées, et de l'autre, la hiérarchie des exigences :



Logiquement, les rapprochements s'effectuent selon les deux familles projet et produit respectivement.

Le principe de rapprochement mis en œuvre est alors le suivant :

- Si un risque projet n'a pas de relation avec une exigence projet, il peut être ignoré pour son organisation.
- Si une exigence projet n'a pas de relation avec un risque projet, dès lors elle peut être remplie sans contrainte (on parlera d'exigence libre).
- Si une exigence projet est concernée par un risque projet, dès lors il faut convenir d'une action palliative dans le cadre de l'organisation du projet, afin de limiter l'impact du risque sur l'organisation du travail.

Et pour le produit, nous avons alors les rapprochements suivants :

- Si un risque produit n'a pas de relation avec une exigence produit, il peut être ignoré lors des tests.
- Si une exigence produit n'a pas de relation avec un risque produit, dès lors l'exigence sera de criticité minimale et les tests associés de priorité la plus faible (voire seront facultatifs : on ignorera les tests !).
- Si une exigence produit est concernée par un risque produit, alors elle prendra pour criticité celle du risque associé et une batterie de tests sera définie pour le couvrir, la priorité des tests étant logiquement proportionnelle à la criticité du risque encouru.

L'objectif du rapprochement entre risque et exigence est donc d'écartier les risques sans impact puis de hiérarchiser les exigences en leur faisant porter la criticité d'un risque.

Mais ce rapprochement ne serait simple que s'il y avait un risque pour une seule exigence... ce qui n'est pas toujours vrai !

Nous avons vu dans la section relative à la définition d'une criticité que l'impact et la fréquence peuvent être fonction de la partie prenante interrogée. Dans le cas d'un risque produit, cette partie prenante est un utilisateur du futur logiciel.

Une première difficulté apparaît donc lorsque l'on prend en compte les parties prenantes des risques et exigences : un même risque peut avoir deux criticités différentes selon la partie prenante concernée.

Par exemple, si pour le webmestre d'un site, l'exigence produit de pouvoir envoyer un message de contact est soumise à un risque de défaillance, il est évident que pour lui la criticité est nulle (il y a peu de chance qu'il s'envoie un mail à lui-même) mais qu'elle est haute pour un internaute pris au hasard dans le public.

Ainsi, la stratégie choisie pour discriminer la fréquence et l'impact d'un risque détermine ici s'il faut ou non tester

une exigence dans tous les profils d'utilisateur.

Au final, chaque triplet (risque, exigence, partie prenante) fournit une estimation de l'impact et une estimation de la périodicité du risque produit, sachant que dans certains cas nous aurons :

- soit des parties prenantes ignorées qui réduiront plusieurs triplets (risque, exigence, partie prenante) à un unique triplet (risque, exigence, partie\_prenante\_X), soit un couple (risque, exigence) si on estime que le risque est le même pour tous les utilisateurs.
- soit des parties prenantes explicitement prises en compte dans une stratégie de couverture plus détaillée.

Nous verrons cependant dans les sections suivantes que ce raisonnement très théorique n'est pas nécessairement le meilleur à tenir.

-  Un même risque pourra impacter plusieurs exigences différentes et réciproquement, une exigence peut être concernée par plusieurs risques simultanément.

À l'issue de ce rapprochement, nous avons alors une matrice globale dont les colonnes seraient :

- Le risque et le type de risque
- L'exigence et le type d'exigence
- La partie prenante
- La fréquence d'impact
- Une valeur de l'impact
- Une valeur de criticité déterminée par le couple (fréquence, valeur) de l'impact

Mais les difficultés de lecture de cette matrice conduiront à la diviser en deux matrices séparées. La première est la matrice des risques et exigences projet qui comporte les colonnes suivantes :

- Le risque projet et son type.
- L'exigence projet et son type.
- La partie prenante projet, optionnelle (lorsque le risque est global pour toutes les parties prenantes, ce champ n'est pas renseigné).
- La fréquence d'impact, fonction de la stratégie de choix de prise en compte du risque relativement à la partie prenante.
- La valeur de l'impact, fonction de la stratégie de choix de prise en compte du risque relativement à la partie prenante.
- La valeur de criticité déterminée par le couple (fréquence, valeur) de l'impact.
- La stratégie de couverture du risque (accepter le risque, le contourner, le restreindre ou le supprimer).
- Les actions implémentant cette stratégie ainsi que les acteurs en charge de leurs mises en œuvre, avec si nécessaire une date d'échéance.

La deuxième est celle des risques et exigences produit qui comporte les colonnes suivantes :

- Le risque produit et son type.
- L'exigence produit et son type.
- Optionnellement, le profil d'utilisateur concerné par l'exigence du produit. Ce champ ne sera pas renseigné si le

risque est équivalent sur tous les profils utilisateur.

- La fréquence d'impact, fonction de la stratégie de choix de prise en compte du risque relativement au profil utilisateur.
- Une valeur de l'impact, fonction de la stratégie de choix de prise en compte du risque relativement au profil utilisateur.
- Une valeur de criticité déterminée par le couple (fréquence, valeur) de l'impact.
- La stratégie de test de couverture du risque pour cette exigence produit (ne rien faire, faire un test unitaire, un test technique, un test fonctionnel, une vérification en pré-production, mettre en place une surveillance en production, etc.).
- Les actions implémentant cette stratégie de test ainsi que les acteurs en charge de leurs mises en œuvre, avec si nécessaire une date d'échéance (développeur, équipe de recette technique, équipe de recette MOA, responsable d'exploitation...).

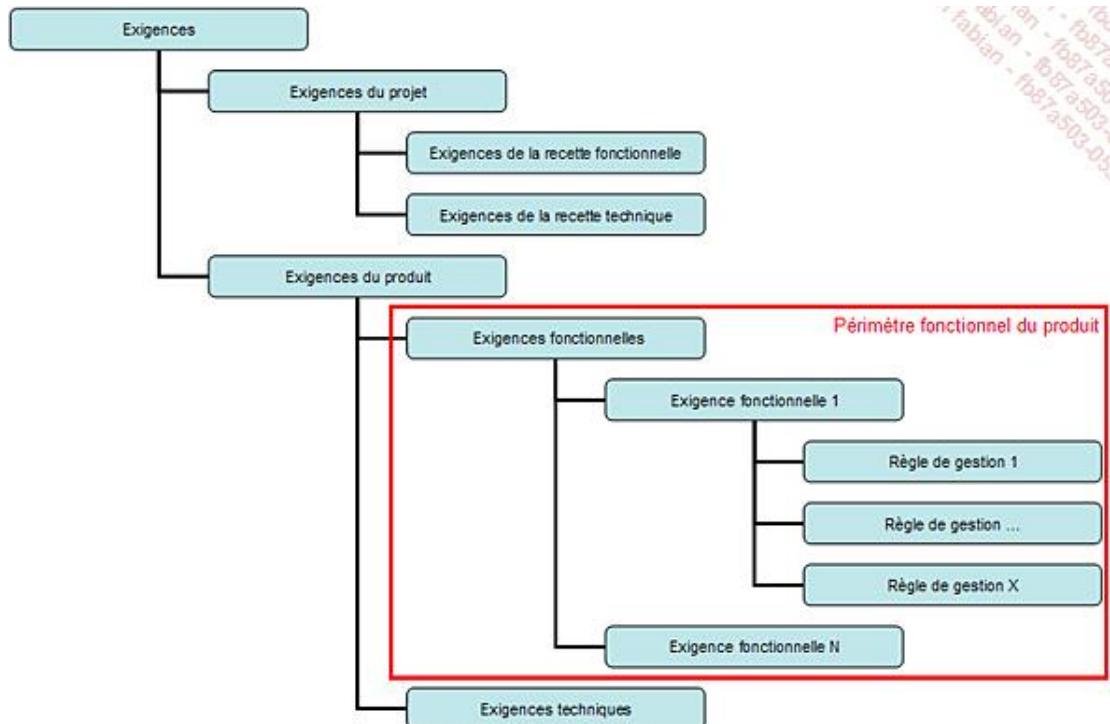
Normalement, la première matrice est utilisée par l'instance de pilotage du projet dont fait partie le CP Recette. Quant à la deuxième, elle est spécifiquement utilisée pour déterminer un plan test, c'est-à-dire un périmètre ordonné des tests, structurés de manière hiérarchique par des priorités calculées sur la base de la criticité associée à chaque triplet (risque, exigence, profil utilisateur).

Nous allons maintenant voir comment déterminer un périmètre de tests en s'appuyant sur cette dernière matrice, selon que l'on se positionne en recette fonctionnelle ou technique.

### c. Matrice de criticité et périmètre d'une recette fonctionnelle

Nous commençons volontairement par décrire la recette fonctionnelle, dont le périmètre et l'objet sont plus restreints. Nous verrons la recette technique dans la section qui suivra.

Tout d'abord, nous entendons par recette fonctionnelle une restriction sur le périmètre des exigences concernées : le périmètre fonctionnel du produit uniquement.



Les exigences techniques ne sont donc pas concernées, à commencer par la compatibilité de l'application dans

plusieurs configurations matérielles.

Une recette fonctionnelle se limite donc à une unique configuration matérielle qui doit être clairement définie comme étant un étalon de validation.

Le CP MOA, qui organise la recette fonctionnelle considérera alors la matrice des risques produit telle qu'elle a été construite dans la méthode RRBT, définit un périmètre de tests fonctionnel couvrant 100 % des fonctionnalités - les exigences de type "fonctionnel".

La matrice des risques produit restreinte aux exigences fonctionnelles fournit alors une liste de lignes pour lesquelles nous disposons de la criticité des exigences par profil utilisateur.

En découle une stratégie de recette fonctionnelle puisqu'on pourra déterminer les exigences devant être testées par profil, prises dans l'ordre décroissant de leurs criticités. Nous définissons alors une priorité des tests fonctionnels par calcul, cette formule étant contextuelle à la recette fonctionnelle.

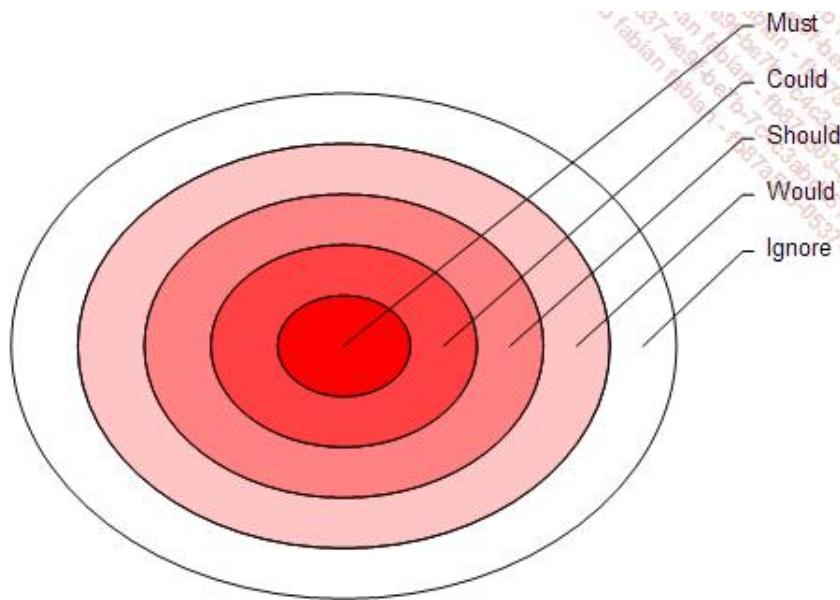
Il est évident que cette priorité serait toute autre dans un contexte différent, par exemple, si la recette était lotie en deux parties : il est alors cohérent de dire que la priorité d'un test n'est pas la même d'un lot à l'autre.

Le lecteur doit alors impérativement comprendre que la priorité du test ne saurait être placée en colonne dans la matrice des risques et exigences... mais qu'elle doit être recalculée pour chaque contexte sur la base d'une copie de cette matrice à un instant T.

La priorité des tests d'une recette X se détermine alors par une fonction :

$$P = \text{Priorité}_\text{Recette\_X}(\text{Exigence}, \text{Profil Utilisateur}, \text{Criticité})$$

Cette fonction renvoyant l'une des 5 valeurs de priorité que nous avons définie précédemment :



Comment définir cette fonction ? Là est la question.

Nous pouvons par exemple appliquer une règle s'appuyant sur la matrice des risques :

- Tout risque dont la criticité est comprise entre 0 et 100 est de priorité Ignore.
- Tout risque dont la criticité est comprise entre 100 et 1 000, est de priorité Would.

- Etc.

Mais cette règle ne pourra s'appliquer que pour la première recette du premier lot par exemple.

Auquel cas, la recette fonctionnelle du deuxième lot appliquerait une règle de calcul de la priorité légèrement différente :

- Tout risque déjà couvert lors de la recette du lot 1 est de priorité Ignore.
- Tout risque propre au lot 2 et dont la criticité est comprise entre 0 et 100 est de priorité Ignore.
- Tout risque propre au lot 2 et dont la criticité est comprise entre 100 et 1 000, est de priorité Would.
- Etc.

Le lecteur comprendra qu'idéalement l'ordonnancement des lots de développement devrait lui aussi prendre en compte la criticité des exigences et être logiquement effectué dans le même ordre que les priorités des tests.

Le chef de projet en charge de la recette fonctionnelle dégage alors une nouvelle matrice : un périmètre de tests. La structure de ce document est un tableau comportant les colonnes suivantes :

- L'exigence produit testée et son type.
- Optionnellement, le profil d'utilisateur concerné par l'exigence du produit. Ce champ ne sera pas renseigné si le risque est équivalent sur tous les profils utilisateur.
- La priorité du test, calculée à l'aide de la valeur de criticité associée à l'exigence et au profil utilisateur (s'il est renseigné).
- La charge nécessaire pour rédiger un test fonctionnel.
- La charge nécessaire pour exécuter le test fonctionnel.

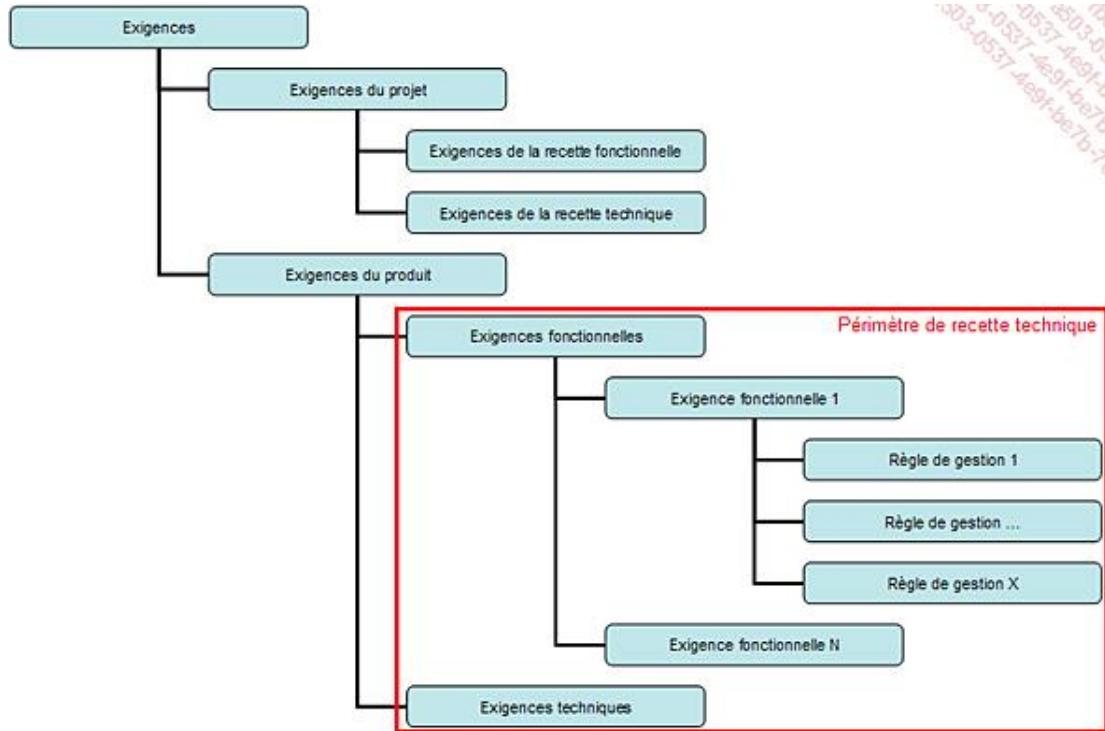
Ce document fournit alors plusieurs indicateurs comme la charge totale de rédaction par priorité et la charge d'exécution, l'effort global de tests fonctionnels... autant d'éléments qui sont ensuite réutilisables, notamment lors des recettes fonctionnelles du chantier de maintenance évolutive qui suivra le chantier de release.

Des sous-parties peuvent être déduites, notamment par profil utilisateur ou par exigence, ce qui permet de quantifier des éléments pour des tests ultérieurs.

#### **d. Matrice de criticité et périmètre théorique d'une recette technique**

La recette technique intervient immédiatement après la réalisation du produit par la MOE et s'effectue selon la même méthode que la recette fonctionnelle... du moins au début car l'élaboration de son périmètre diverge ensuite.

Tout d'abord, l'arborescence des exigences testées est plus large :



Ceci signifie que les tests envisagés en recette fonctionnelle pourront être utilisés aussi en recette technique de manière anticipative. Cependant, l'angle de vue n'est plus le même qu'en recette fonctionnelle :

- Les exigences fonctionnelles testées sont les mêmes, mais le contenu des tests se focalisera sur des considérations techniques et fonctionnelles plus détaillées. Par exemple, si lors d'une recette fonctionnelle la longueur de saisie d'un champ n'est jamais testée, elle le sera systématiquement en recette technique.
- Les exigences techniques feront l'objet de scénarios spécifiques, comme par exemple des tests de robustesse ou de montée en charge... avec un cas particulier pour les configurations matérielles pour lesquelles l'exigence de compatibilité sera testée en déroulant pour chacune un périmètre fonctionnel - voire la totalité !

De la même manière que pour une recette fonctionnelle, le chef de projet recette technique dégagera là aussi un périmètre de tests qu'il pourra matérialiser sous la forme d'une nouvelle matrice. Quelques éléments différeront, notamment la charge de rédaction et d'exécution des tests qui n'est pas du tout la même que pour une recette fonctionnelle.

Cependant, le périmètre dégagé devra être considéré comme purement théorique car comme nous le verrons, la recette technique doit prendre en compte des facteurs de complexité qui rendent la définition de son périmètre encore plus difficile.

## e. Exigence fonctionnelle, technique et niveau de test

La présente section a pour objectif de clarifier une confusion fréquente dans les tests en précisant la différence entre les exigences fonctionnelles, techniques et le niveau des tests nécessaires.

Nous commencerons par tordre le cou à une idée reçue : il n'existe pas de notion de granularité dans les exigences fonctionnelles.

Une exigence fonctionnelle est une fonction atomique, non décomposable, exécutée par un profil utilisateur et qui a un sens métier. Ceci veut dire qu'elle se teste à l'aide d'un scénario fonctionnel qui contient une succession de tests (des actions de test ou "step") qui correspondent à des actions que ferait réellement un utilisateur final.

Comme nous l'avons vu, une exigence fonctionnelle peut être vue comme le regroupement d'exigences plus petites : les règles de gestion. Il est d'ailleurs possible de développer cette décomposition hiérarchique de l'exigence fonctionnelle. Mais est-ce à dire que l'exigence fonctionnelle se décompose en sous-exigences fonctionnelles ? Certainement pas.

Une exigence fonctionnelle est décomposable, c'est un fait, mais s'intéresser à l'une de ses sous-parties induit que l'on change son point de vue : le niveau de test associé n'est plus nécessairement le même.

Concrètement, cela signifie que si une exigence fonctionnelle est testée par un scénario de test précis, alors une sous-partie de cette exigence - par exemple une règle de gestion - ne se teste pas avec un scénario fonctionnel, mais un test technique, car le fait de se focaliser sur une règle de gestion induit que celui qui fait le test n'est plus dans la peau d'un utilisateur final mais celle d'un testeur.

La différence entre exigence technique et exigence fonctionnelle induit donc une approche de test très différente, la première n'étant pas équivalente à la deuxième, car ce que fait un testeur en recette technique n'est pas nécessairement assimilable à ce que ferait un utilisateur final en situation réelle.

Ainsi, qu'une règle de gestion soit une sous-partie d'une exigence fonctionnelle n'induit pas qu'elle est une exigence fonctionnelle elle-même et qu'elle doit être testée avec le même niveau de test.

Le lecteur doit alors garder en permanence à l'esprit les deux principes suivants :

- Une exigence fonctionnelle est une sous-partie du produit réalisé et le risque de sa défaillance est couvert par un scénario de recette fonctionnelle, déroulé par la MOA, et qui n'impliquera pas de tester toutes les sous-exigences liées, telles que les règles de gestion.
- Une exigence fonctionnelle est une sous-partie du produit réalisé et le risque de sa défaillance est aussi couvert par une fiche de tests techniques, déroulée par l'équipe de recette technique, et qui couvrira autant que possible toutes les sous-exigences liées comme les règles de gestion, celles-ci étant donc considérées d'un point de vue technique.

Pour comprendre ces deux principes, nous vous proposons un raisonnement par l'absurde, en supposant que les règles de gestion sont des exigences fonctionnelles, c'est-à-dire une action atomique que peut réaliser un utilisateur final. Par exemple, nous considérerons que l'exigence "Le nom de l'abonné fait au plus 20 caractères" est une fonction de l'application pour souligner le caractère absurde de cette affirmation.

Il découle de cette affirmation qu'il faudrait qu'il existe un scénario fonctionnel (donc qui a un sens métier) qui vérifie que l'exigence est couverte.

Avez-vous besoin d'interroger les utilisateurs d'un logiciel pour savoir s'il existe quelqu'un qui usuellement va mesurer les longueurs des champs de saisie des formulaires ? Bien sûr que non. Compter la longueur d'un champ est une action de test spécifique à un testeur.

Ainsi, la rédaction d'un scénario de test qui vérifierait que la longueur du champ est conforme, est une absurdité du point de vue de la recette fonctionnelle.

A contrario, cette même règle de gestion, dès lors qu'elle est considérée pour ce qu'elle est - c'est-à-dire une sous-exigence de bas niveau demandant une recette technique - il devient évident que le support de test utilisé n'est pas un scénario mais un regroupement de tests sans corrélation avec une démarche métier, support que nous appellerons une fiche de test pour éviter toute confusion avec le scénario fonctionnel.

Quelle est l'incidence de la perte de vue de ces principes ?

Malheureusement, les conséquences peuvent être catastrophiques. Car dès lors qu'on considère qu'une règle de gestion est une exigence qui doit être testée au même niveau de test que l'exigence fonctionnelle parente, alors cela revient à dire qu'il y a autant de scénarios de test qu'il y a de règles de gestion.

Ceci signifie que chaque vérification est considérée comme une entité autonome réalisée une à une par un testeur et que la notion de fiche de test n'existe pas.

- La multiplication du nombre de scénarios, qui entraîne une perte de lisibilité du périmètre de tests.
- Une radicalisation de la démarche de test qui impose des affectations de tâches très nombreuses (plus il y a de scénarios, plus il y a d'items à affecter à un testeur) et une augmentation de la charge de travail normalement nécessaire.
- Un appauvrissement du rôle du testeur, considéré comme simple exécutant d'un micro-test sans perspective de mise en œuvre de ses compétences propres.

 Une comparaison simpliste consisterait à se poser la question de savoir s'il faut tester la vitesse de déplacement de chaque pièce d'une voiture (donc de la démonter) ou s'il faut se mettre au volant et appuyer sur l'accélérateur ?

Nous invitons le lecteur à ne jamais perdre de vue que l'organisation hiérarchique des exigences (techniques comme fonctionnelles) n'est jamais qu'une vue très conceptuelle du produit.

Quant aux différents périmètres de recette que vous pourrez en dégager, ceux-ci s'appuieront sur cette hiérarchie... relativement à une démarche de test, qui n'est pas unique donc avec plusieurs niveaux d'appréciation.

# Périmètre disponible : gérer les dégradations

Dans un monde idéal et parfait, les périmètres des recettes fonctionnelles et techniques se déduiraient des seules analyses des exigences et des risques.

Dans ce même Eden de l'Informatique, les applications arriveraient toutes dans des environnements de tests parfaits...

Malheureusement, ce n'est jamais le cas et la plupart des personnes qui travaillent sur un projet vous diront qu'il est quasiment impossible d'avoir des conditions de test rigoureusement identiques à celle de l'environnement de production.

Nous allons voir comment prendre en compte ces contraintes dans la présente section.

## 1. Environnement dégradé

Par environnement, nous désignons l'ensemble des données et paramètres qui conditionne l'exécution du produit testé.

### a. Des traitements en moins

Maintenir un environnement est coûteux. Et généralement, le budget alloué à celui de production est très élevé.

Aussi lorsque les besoins des équipes de test induisent l'usage d'environnements de test dédiés, cela suppose que les moyens financiers, logistiques et techniques à les maintenir soient mis en œuvre.

La réalité du terrain vous montrera rapidement que l'obtention d'un environnement de test à l'image de la production n'est pas toujours possible. Au mieux, vous aurez une image du passé qui ne sera pas synchronisée - ou éventuellement de manière occasionnelle. Au pire, une image très parcellaire.

La mise à disposition de cet environnement - qui par définition est temporaire - est aussi conditionnée par les moyens matériels et leurs coûts de maintenance. Il est ainsi fréquent que plusieurs environnements de test soient logiquement installés sur le même serveur physique, par exemple.

Bref, vous n'aurez jamais l'exacte réplique de la production.

Et parmi les écarts, les premiers à être souvent constatés sont l'absence des traitements automatiques nocturnes et l'absence de certaines applications.

Ces absences, que nous appelons "dégradation de l'environnement", introduisent alors des limites dans les périmètres de test :

- L'absence d'une application ne permettra pas de tester une bascule entre le logiciel testé et elle-même, par exemple.
- L'absence d'un traitement automatique ne permettra pas d'avoir des données périodiquement traitées donc en conformité fonctionnelle permettant la bonne exécution du logiciel testé.
- Etc.

Il est bien entendu impératif que chaque chef de projet (MOA ou Recette) ait une visibilité des limites de l'environnement de test dans lequel il s'apprête à faire des vérifications.

Ces limites doivent nécessairement être prises en compte dans chaque phase de recette, c'est-à-dire que les

exigences produit concernées doivent être listées et faire l'objet d'une attention particulière afin que des tests soient réalisés en complément dès que l'environnement le permet - en dernier recours, en production si nécessaire.

Par ailleurs, chaque dégradation d'environnement pourra être plus ou moins contournée parfois, notamment par des simulations.

Si nous prenons le cas d'une application manquante lors d'un enchaînement, il est parfaitement envisageable de la remplacer par un traitement temporairement dédié aux tests pour s'y substituer. Par exemple, une interface graphique dédiée pourra permettre une saisie de données en l'absence de l'application qui aurait dû assurer cette exigence.

Mais les simulations peuvent prendre d'autres formes, comme par exemple modifier manuellement une donnée au cours d'un scénario fonctionnel pour permettre son déroulé normal.

Ces parades offrent cependant très rapidement une limite car leurs usages conditionnent la validité des tests qui en découlent. Il y a donc une réserve à émettre dans ces cas-là, voire même il faudra se poser la question de la pertinence à réaliser des tests avec trop de simulations.

Selon le contexte, il appartiendra aux différents décideurs du projet de définir comment tester efficacement l'application avec de telles contraintes environnementales, tout en ne perdant pas de vue qu'une anomalie détectée au plus tôt coûtera moins cher au projet.

Bien sûr, tôt ou tard l'application sera en production donc dans un environnement final où une ultime vérification est faisable... mais pour quel prix ?

## b. Des données en moins

Une autre dégradation environnementale possible est tout simplement la suppression de tout ou partie des données traitées par une application. Typiquement, ce cas de figure se présente dans les applications d'architecture 3-Tiers pour lesquelles la couche de plus bas niveau, l'accès aux données, est développée de sorte que les services d'accès puissent renvoyer une réponse par défaut lorsque la base de données est absente ou en panne.

On parlera alors de bouchons pour désigner les supports de ces données statiques retournées par défaut.

 Pour des technologies de type WebServices, il est fréquent que ces bouchons soient tout simplement des fichiers XML. Mais toutes les technologies sont envisageables, du simple fichier textuel contenant une liste de variables à des structures plus complexes.

La particularité des bouchons est qu'ils font partie des sources de l'application et à ce titre sont donc difficilement modifiables ce qui veut dire qu'on ne peut pas faire une simulation de transformation de données (cela impliquerait une livraison à chaque fois).

Dans un tel cas de figure, il est évident qu'une recette fonctionnelle n'offre pas un grand intérêt. Seule une recette technique est pertinente, et encore, lorsqu'elle se limite à un rendu de la présentation des données pour les traitements de lecture.

Concernant les traitements d'écriture ou de suppression en base, les bouchons induisent l'absence effective de l'action et le renvoi systématique d'une réponse contenant un code résultat générique. Ce qui revient à dire qu'aucune mise à jour n'est opérationnelle donc testable d'un point de vue fonctionnel.

Les décideurs du projet - tant côté MOA que MOE et Recette technique - devront donc impérativement connaître

les contours de ces bouchons qui entraînent de facto une grande limitation dans la réalisation des tests et une contrainte forte du projet.

Techniquement, les bouchons sont en général partagés par toute l'équipe de la MOE. Même chose pour l'équipe de recette.

Cela signifie que leur modification durant l'activité a un impact collectif. Dès lors que des bouchons sont modifiés pendant une recette technique, cela implique donc une synchronisation des testeurs.

Ainsi les dégradations d'environnements conditionnent le périmètre des recettes : du périmètre fonctionnel théoriquement prévu à l'issue de l'analyse RRBT, des restrictions sont menées.

## 2. Application dégradée

### a. Mode de fonctionnement d'une application

Une application peut avoir - sans que cela soit une nécessité - un paramètre conditionnant son comportement : un mode de fonctionnement.

Nous avons des applications ou des traitements qui possèdent un "mode debug" ou un "mode trace", c'est-à-dire que l'exécution devient alors identique d'un point de vue fonctionnel mais que des signaux techniques sont conjointement émis pour renseigner un technicien sur le comportement de l'exécution.

Ainsi, des fichiers de traces, des tables de logs, des messages visuels, peuvent être fournis aux observateurs, aux testeurs.

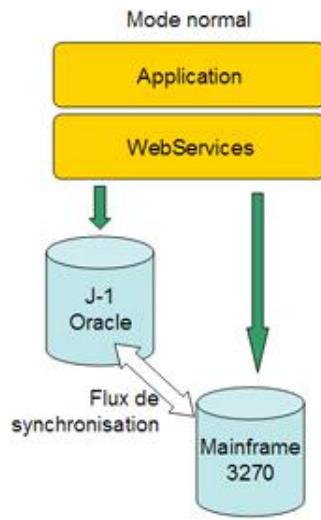
Il est également possible qu'une dégradation de l'environnement de test induise un mode de fonctionnement pour l'application : une application peut aussi être en mode "panne du service" ou en mode "maintenance".

Pour illustrer notre propos, nous prendrons l'exemple d'un système bancaire dont l'architecture technique se compose d'un mainframe 3270 CICS/DB2, d'une base de données Oracle utilisée comme base de données J-1 tampon, d'une couche d'accès logicielle par WebServices et d'une application web pour la couche de présentation. Cette application serait celle permettant la consultation des comptes des clients d'une banque.

Dans ce système, un usager consulte ses comptes dont il peut voir une image du passé, celle de la veille donc à J-1, périodiquement mise à jour par des traitements nocturnes de synchronisation avec le mainframe. Mais l'usager peut aussi vouloir réaliser des actions d'urgence en temps réel, une opposition de sa carte bancaire par exemple.

Dans un tel cas, le traitement a donc besoin d'accéder à la base de données à J, c'est-à-dire le mainframe bancaire, et non pas l'environnement J-1 Oracle.

Ce contexte que nous venons de décrire, est un fonctionnement normal du service de banque en ligne et il peut être illustré par le schéma suivant :



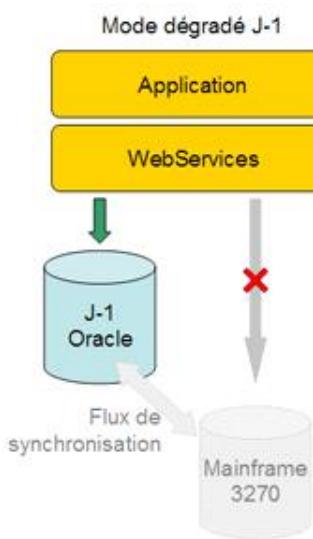
Cependant, pour des raisons de maintenance, le mainframe pourra être périodiquement basculé dans un mode de fonctionnement pour lequel il ne sera pas accessible.

Cette maintenance peut être technique ou fonctionnelle, par exemple pour permettre aux traitements automatiques de synchronisation de tourner durant la nuit. Elle constitue alors une période de temps pendant laquelle le service de banque en ligne ne pourra pas être opérationnel à 100 %.

Un paramètre de l'application est alors utilisé pour modifier le comportement du logiciel pendant cette période d'indisponibilité.

Concrètement, le mode de fonctionnement dégradé indiquera que les données à J du mainframe sont inaccessibles ce qui veut dire qu'il ne sera pas possible d'effectuer une opposition de la carte bancaire à J, mais qu'en revanche la consultation des comptes à J-1 restera possible.

Cette seconde situation est illustrée par le schéma suivant :



Ainsi, une même application pourra fonctionner différemment selon un simple paramètre, ce qui affecte nécessairement le périmètre des tests ainsi que leur validité parfois.

Lorsque l'existence d'une telle mécanique induit la génération de fichiers de traces ou des logs, il conviendra de définir des tests spécifiques dédiés à cette fonctionnalité - normalement durant la recette technique - il est rare

qu'une recette fonctionnelle débouche sur ce type de test.

Sinon, comme dans notre exemple de mode de fonctionnement d'un système bancaire, il faudra prévoir deux recettes :

- Une recette technique en mode J-1, pour laquelle il n'est pas possible de mener des tests fonctionnels de bout en bout.
- Une recette fonctionnelle en mode normal, pour laquelle les fonctionnalités désactivées du mode J-1 feront l'objet de tests plus attentifs en raison des limites de la recette technique.

Le chef de projet d'une recette technique devra donc impérativement se tenir informé des dégradations techniques qu'impliquent les modes de fonctionnement d'une application, au même titre qu'il doit connaître celles de l'environnement de recette lui-même.

## b. Suppression de composants externes, dégradation de version

Cette section concerne davantage les applications de type client lourd, notamment les composants de type DLL et/ou OCX que l'on trouve sur les applications développées sous Windows.

Les applications comportant des clients graphiques sous Windows exploitent la plupart du temps les API Windows et des composants communs. Ces derniers font partie intégrante de l'application testée (puisque installés avec le package) mais peuvent être déjà présents dans une version plus récente ou plus ancienne.

Généralement, le principe est de laisser le composant s'il est le plus récent et de proposer son installation s'il y a montée de version.

Théoriquement, la compatibilité ascendante est assurée par l'éditeur dudit composant. Très théoriquement...

Dans la pratique, le logiciel installé pourra surtout ne pas être le même et une dégradation sera alors créée sans qu'elle soit nécessairement visible.

Quelle que soit la recette à réaliser, il conviendra donc que le chef de projet en charge de cette tâche soit vigilant et vérifie la liste des éléments installés.

S'il est peu probable que le périmètre fonctionnel de la recette soit affecté par cette dégradation applicative, il y aura pourtant un impact : les tests d'installations devront être plus rigoureux et surtout largement anticipés, notamment pour que les environnements de développement soient normalisés dès la phase de réalisation et tests unitaires.

Voici un exemple ancien mais concret. Un laboratoire de pharmaceutique souhaitait développer une application de gestion documentaire capable de pré-renseigner des champs dans un document Word au moment de son ouverture depuis le logiciel en question (en récupérant les informations dans une base de données documentaire). La solution consista à utiliser la bibliothèque WordBasic de la version Microsoft Word 6.0 pour piloter l'ouverture du document.

Durant la phase de réalisation, le client réalisa qu'il avait aussi des postes utilisant Word 97. Or la bibliothèque WordBasic n'était alors pas la même. L'équipe a donc réalisé deux versions de l'application, l'une capable de travailler avec Word 6, l'autre avec Word 97.

Vint la journée de tests de validation chez le client sur son PC étalon, spécialement dédié et équipé de Word 6. L'application pourtant prévue pour Word 6 ne fonctionnait pas. Par acquit de conscience, les tests furent aussi réalisés avec Word 97 : même constat d'anomalie.

Après analyse, il s'avéra que le poste étalon du client avait été sous Word 97, puis que ce logiciel avait été

désinstallé et remplacé par Word 6 pour les besoins de la recette... opération qui créa un logiciel Word hybride entre les versions 6.0 et 97.

La différence venait d'une des DLL de Word.

Cet exemple illustre qu'une recette d'homologation fonctionnelle (dont le but est d'enclencher une facturation) nécessite d'anticiper les éventuelles dégradations applicatives.

### c. Dégradation des applications web

De la même manière que pour les clients lourds, les applications web peuvent connaître des dégradations plus ou moins maîtrisées.

Une première source de dégradation vient de l'architecture applicative qui est distribuée : des composants (JavaScript, images...) peuvent être téléchargés de sources externes au serveur applicatif.

La difficulté vient alors de l'ignorance : si les composants sont modifiés par l'éditeur, la mise à jour est immédiatement répercutée sur l'application, à l'insu même de l'équipe projet et des utilisateurs.

Une solution est d'effectuer des tests techniques larges pour couvrir toute l'application web en activant un pare-feu très restrictif. Les composants externes ne seront pas téléchargés provoquant une dégradation visible de l'application.

Cette stratégie permet ensuite d'imposer que les composants externes soient installés sur le serveur (et deviennent donc internes à l'application) pour éviter toute modification non contrôlée.

Une deuxième source de dégradation des applications web vient du paramétrage du navigateur qui permet :

- De désactiver le JavaScript.
- De désactiver les feuilles de style.
- De désactiver les images.
- Etc.

Ces dégradations pourront être envisagées lors de recettes techniques mais peuvent aussi l'être pour des recettes fonctionnelles, notamment si l'application web doit respecter des contraintes d'accessibilité numérique.

En effet, rendre un site accessible à une personne handicapée oblige à présenter les données selon une structure HTML très rigoureuse mais qui autorise surtout les dégradations évoquées ci-dessus, notamment le JavaScript qui doit pouvoir être désactivé sans cela occasionner une perte d'information.

Les images et feuilles de style doivent pouvoir être désactivées également (typiquement, un aveugle n'en a pas besoin) ce qui veut dire que les images ne doivent pas contenir d'information et toujours avoir une alternative textuelle.

La dégradation des applications web revêt donc un aspect qui ne doit pas être négligé par les décideurs du projet web car sa nécessité suppose une démarche précise en amont : la prise en compte de contraintes fortes dans les développements et les tests.

## 3. Périmètre de tests et dégradations

### a. Améliorer la matrice du périmètre des tests

Nous avons proposé un outil pour formaliser un périmètre de tests fonctionnels : une matrice. Celle-ci avait la description suivante :

- L'exigence produit testée et son type.
- Optionnellement, le profil d'utilisateur concerné par l'exigence du produit. Ce champ ne sera pas renseigné si le risque est équivalent sur tous les profils utilisateur.
- La priorité du test, calculée à l'aide de la valeur de criticité associée à l'exigence.
- La charge nécessaire pour rédiger un test fonctionnel.
- La charge nécessaire pour exécuter le test fonctionnel.

Nous proposons d'améliorer cette structure pour prendre en compte la dégradation de l'environnement ou de l'application en ajoutant les colonnes suivantes :

- Un niveau de dégradation de l'exigence (Pas de dégradation, Exigence dégradée, Exigence indisponible).
- Un commentaire précisant la nature de la dégradation, son impact sur le produit.

Bien sûr, l'intérêt de ces informations est relatif au contexte : si la notion de dégradation n'existe pas dans l'environnement de la recette (tant en recette technique que fonctionnelle), ces données n'ont aucun intérêt.

 Nous déconseillons de gérer la dégradation en modifiant la priorité du test. On pourrait croire qu'une exigence dégradée justifierait de baisser la priorité du test associé. Dans la pratique, ce n'est pas une bonne vision des choses : la priorité du test est déterminée à partir de la criticité de l'exigence en dehors de toute considération technique : il est préférable de garder ce paramètre en l'état, puis de gérer la dégradation de manière dissociée... tout simplement parce que cette dégradation est susceptible de disparaître.

La prise en compte de la dégradation possible des exigences de cette manière dans la matrice du périmètre des tests permet alors de dégager deux sous-périmètres :

- Le périmètre des exigences fonctionnelles effectivement testables.
- Le périmètre des exigences dégradées qui devront faire l'objet d'une recette ultérieure sans les dégradations.

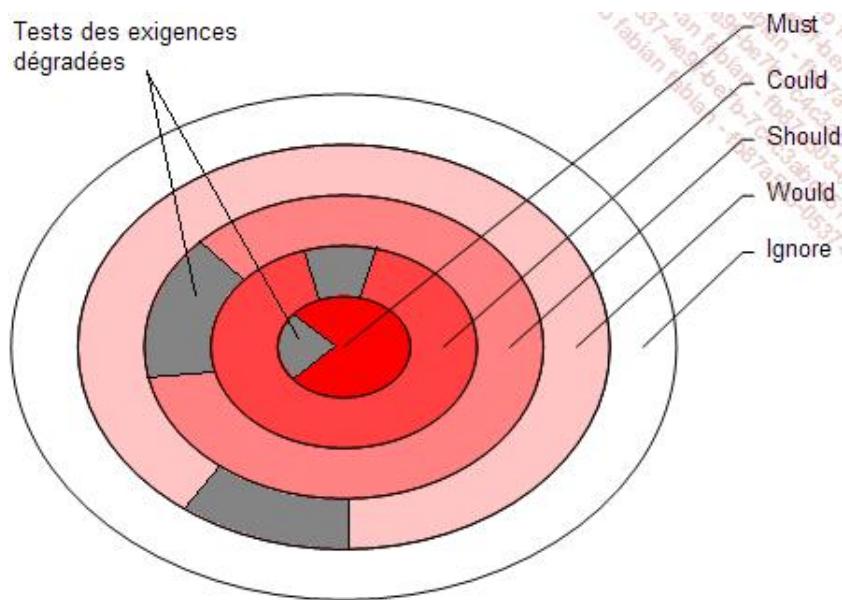
Raison pour laquelle il ne faut pas changer la priorité du test : cette donnée ne constitue pas un critère permettant de séparer les deux sous-périmètres qui découlent des dégradations.

## b. Représentation du périmètre des tests et des dégradations

Une fois l'analyse RRBT menée, la prise en compte des contraintes techniques amène à traiter les éventuelles dégradations des exigences fonctionnelles du produit.

Nous avons proposé une représentation graphique concentrique du périmètre des tests selon une échelle de priorité.

La prise en compte des dégradations des exigences modifie ce schéma comme suit :



Cette vision des choses permet de représenter un périmètre de tests selon une notion de faisabilité technique, le classement en niveau Must, Could, Should, Would et Ignore n'étant que la traduction fonctionnelle des priorités exprimée par la MOA et le métier.

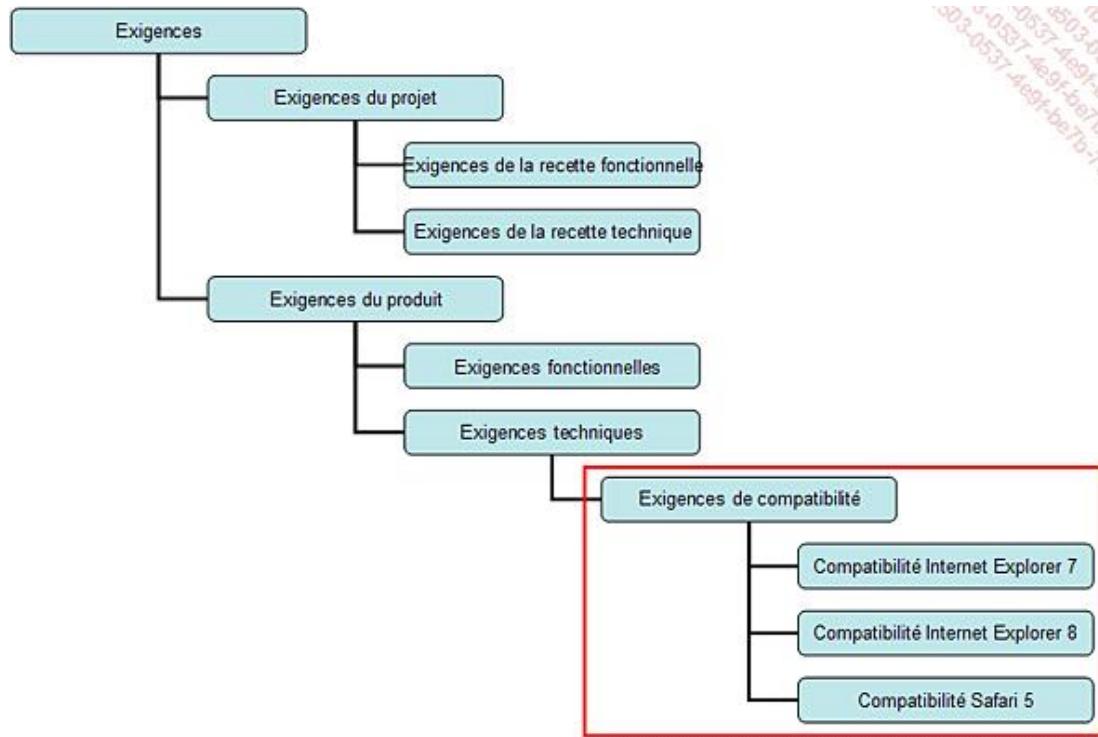
Que les dégradations aient une origine technique liée à l'environnement ou bien au lotissement des livraisons du produit, que ces dégradations soient visibles ou non en recette fonctionnelle et/ou en recette technique, peu importe.

Le principe de base qu'il faut retenir est qu'une dégradation induit deux sous-périmètres de tests à un instant T du projet : ce que l'on peut tester à ce même instant T, ce qui est disponible, et ce qu'il faudra tester plus tard dans le projet.

# Périmètre des configurations matérielles

Dans l'arborescence des exigences produit, nous avons mentionné sans la détailler, les exigences techniques d'un produit.

Parmi elles, nous trouvons des exigences de compatibilité entre des configurations matérielles.



Nous avons déjà évoqué les difficultés rencontrées par une équipe de développement dans ce domaine, ne serait-ce que pour reproduire une anomalie dans les mêmes conditions que sa détection.

Dans le cadre des recettes qui suivent les développements, cette dimension prend une importance notable qu'il convient de maîtriser.

La présente section expose comment gérer les configurations matérielles dans les recettes techniques et fonctionnelles. Nous vous proposerons des solutions parcellaires cependant, car c'est le contexte du projet qui conduira à une réflexion sur ce sujet, donc à des adaptations de ce que nous proposons.

## 1. La gestion des configurations matérielles

### a. En recette fonctionnelle

Théoriquement, une recette fonctionnelle n'est pas censée être la phase du projet où des exigences techniques sont testées.

Une recette fonctionnelle est censée homologuer l'application selon des critères purement métier : le respect des spécifications fonctionnelles, des règles de gestion du métier.

Toujours très théoriquement, une recette fonctionnelle serait donc censée avoir lieu dans une configuration matérielle unique, un "étauon matériel de référence".

Ainsi, les recettes réalisées par la MOA devraient s'effectuer sur une plateforme totalement normalisée, par exemple, une application web sera testée exclusivement sur une seule version d'un unique navigateur.

La réalité des projets est cependant toute autre car fonction des ressources humaines et matérielles consacrées au projet.

Qu'un projet n'ait pas de cellule de tests techniques dédiée, et c'est la MOA qui devra compenser l'effort de test. Dans une telle situation, il est bien évident qu'une recette censée être purement fonctionnelle prendra une dimension technique également. Et l'un des facteurs techniques pris en considération est souvent la compatibilité avec des configurations matérielles différentes.

La technologie du produit doit cependant être prise en compte dans ces considérations. Si nous prenons l'exemple d'une application web, il est très facile pour une équipe MOA non technicienne d'installer plusieurs navigateurs et faire des tests de compatibilité, tout simplement parce que la multiplicité des configurations matérielles est simple à construire.

Mais ce n'est pas toujours le cas. Une application mobile ou web-mobile demandera du matériel autre que des PC, donc un existant qu'une MOA n'a pas par défaut et qui se traduirait par un investissement budgétaire à fournir.

Pour les clients lourds, la création de configurations matérielles multiples nécessitera des connaissances plus techniques qu'une MOA n'aura probablement pas.

Ainsi, la technologie du produit induit une facilité ou au contraire une difficulté dans la construction de la plateforme de recette ce qui a pour conséquence la mise en œuvre de stratégies différentes pour la prise en compte des configurations matérielles dans les recettes fonctionnelles et techniques.

Nous conseillons au chef de projet MOA et Recette technique d'anticiper au maximum cette problématique, si possible bien avant la réalisation du produit, afin de définir la stratégie la plus adaptée au contexte du projet en définissant qui réalise les tests de compatibilité, quitte à envisager si nécessaire de répartir ces tests entre l'équipe de recette technique et l'équipe de recette fonctionnelle.

Dans ce cas-là, nous vous invitons à lire la section suivante qui exposera comment définir un périmètre de recette par configuration matérielle.

## b. En recette technique

La compatibilité entre les configurations matérielles est une exigence technique qui logiquement devrait revenir systématiquement à une cellule de recettes techniques en sortie de développement.

D'une part parce que ces considérations techniques sont davantage le domaine d'activité d'une telle équipe plutôt que le domaine d'intervention d'une MOA, d'autre part parce qu'il est préférable que les incompatibilités soient détectées le plus tôt possible.

Comme nous l'avons vu dans la section précédente, ce n'est cependant pas toujours le cas, tout simplement parce qu'un projet n'a pas nécessairement la possibilité d'avoir une cellule dédiée d'experts en tests technico-fonctionnels.

Le chef de projet MOA trouvera donc ici les bonnes pratiques à mettre en œuvre.

Tout d'abord, l'exigence technique de compatibilité a une caractéristique particulière que n'ont pas les autres types d'exigences techniques : le risque de défaillance qu'elle présente nécessite non pas un test mais deux ou plus, puisqu'il s'agit d'exécuter scrupuleusement et successivement le même scénario ou la même fiche de test dans chacune des configurations matérielles envisagées. Et ce, afin de dégager une différence de comportement pour une même exigence fonctionnelle.

Ceci revient à dire que par défaut, il faudrait tester 100 % du périmètre fonctionnel d'une application sur N configurations matérielles pour garantir une exigence de compatibilité totale.

Plus concrètement, la matrice du périmètre de tests que nous avons dégagée dans les sections précédentes, devrait donc être dupliquée autant de fois qu'il y a de configuration matérielle.

Une autre façon de procéder consisterait à ajouter une colonne "Configuration matérielle" dans notre matrice et à dupliquer le périmètre de tests autant de fois que nécessaire.

Mais avant de décider du meilleur formalisme à adopter, il convient de se poser des questions sur ce qui motive cette exigence de compatibilité.

### c. Pour quel résultat ?

Les tests de compatibilité d'un produit dans plusieurs configurations matérielles visent à prendre en compte une hétérogénéité des logiciels qui existent dans plusieurs versions sur des postes très différents. Et ce, afin de garantir une ouverture du produit à un grand nombre d'utilisateurs.

L'impact est ici contextuel :

- Pour une application qui resterait interne à l'entreprise, il sera bon de se poser la question de l'opportunité d'une migration vers une unique configuration matérielle (encore que le contexte ne le permette pas toujours). L'impact concerne donc essentiellement la productivité.
- En revanche, pour une application distribuée à l'extérieur, la dimension commerciale devient prépondérante et l'impact est alors financier.

Les enjeux de la compatibilité peuvent donc être importants.

Mais quel pourra être le résultat des tests de compatibilité ?

En supposant qu'une anomalie est détectée et dont la cause est une différence technique, quelle marge de manœuvre aura l'équipe projet a posteriori ?

Tout d'abord, il faudra se poser la question de l'éditeur du composant qui apporte une telle différence et de la relation qu'on entretient avec lui.

Si nous prenons l'exemple d'une application web qui ne serait pas compatible entre un navigateur Firefox et une version d'Internet Explorer, devra-t-on attendre que les éditeurs de ces deux logiciels produisent une version compatible ?

A contrario, si l'éditeur du composant défaillant est un fournisseur plus proche, peut-être sera-t-il possible de lui demander un correctif... ce qui posera alors le problème de savoir à quelle date il serait disponible. Le planning de votre propre projet s'en trouverait donc impacté.

Dans un deuxième temps, si aucune solution corrective n'est envisageable à plus ou moins court terme, peut-être faudra-t-il opter pour la solution de contournement, c'est-à-dire modifier l'application pour pallier le défaut du composant.

Cette solution - qui n'en est pas réellement une - s'avérera parfois irréalisable, soit parce que techniquement impossible, soit parce qu'elle nécessitera un si grand nombre de modifications que la charge de correction s'en trouvera trop forte.

Nous aurions tendance à vous déconseiller fortement ce choix, mais peut-être que vous ne pourrez pas faire autrement.

Enfin, il existe une dernière alternative : déconseiller l'usage des configurations matérielles trop dégradées et ne garantir l'application que pour un périmètre sûr des configurations matérielles. La solution est donc de communiquer.

Mais là aussi, le choix est contextuel et fonction des enjeux commerciaux, de la nécessité d'avoir une application ouverte.

#### d. Quelle stratégie adopter ?

Une fois que l'objectif de compatibilité matérielle est clair pour les décideurs du projet, il conviendra de mettre en œuvre une stratégie de tests par configuration matérielle. Et selon ce choix, le formalisme adopté pourra être adapté.

Nous avons précédemment évoqué la stratégie par couverture totale : 100 % du périmètre fonctionnel testé sur N configurations matérielles. Cette stratégie suppose donc :

- Une charge X de rédaction des tests, indépendante du nombre de configurations matérielles.
- Une charge  $N \times Y$  fonction du nombre de configurations matérielles et de la charge Y nécessaire pour tester 100 % des fonctions.

En d'autres termes, plus il y a de configurations matérielles, plus la charge d'exécution des tests est importante, et plus la montée en charge entre la phase de préparation des tests et l'exécution est forte.

Ceci représente un risque d'autant plus fort qu'une anomalie bloquante pourra alors fortement réduire le périmètre d'exécution sur les N configurations en même temps, provoquant une rupture de charge immédiate et un risque de dérive tout aussi important.

La stratégie d'imposer que 100 % des tests fonctionnels soient réalisés sur toutes les configurations matérielles est donc une stratégie risquée.

Notre retour d'expérience sur le sujet montre que cette stratégie doit être envisagée lorsque les configurations matérielles étudiées sont inconnues et testées pour la première fois. Ou bien dans des technologies extrêmement hétérogènes, comme par exemple dans le domaine des terminaux mobiles.

L'autre option dont dispose le chef de projet qui organise la recette sera d'utiliser les priorités des exigences fonctionnelles pour définir des périmètres différents par configuration matérielle, c'est-à-dire trouver un compromis entre la charge d'exécution des tests et l'exigence de compatibilité.

Voici un exemple des périmètres que nous pourrions trouver pour une application web :

Priorités	Internet Explorer 8	Internet Explorer 7	Firefox 4	Safari 5
Must	Oui	Oui	Oui	Oui
Could	Oui	Non	Oui	Non
Should	Oui	Non	Non	Non
Would	Oui	Non	Non	Non

Ce compromis pourra être motivé comme suit :

- 60 % de nos utilisateurs sont sous IE8 => nous testons tout le périmètre fonctionnel pour cette configuration matérielle.

- 20 % de nos utilisateurs sont encore sous IE7 et nous estimons que nous pouvons faire confiance à Microsoft pour assurer une compatibilité IE7/IE8. Le fait de couvrir 100 % des tests sous IE8 est donc suffisant pour garantir que cela marchera aussi sous IE7. Mais nous prenons la sécurité de vérifier toutefois les exigences fonctionnelles les plus critiques.
- 15 % des utilisateurs sont sous Firefox 4 : la probabilité qu'une anomalie existe entre IE8 et Firefox 4 est certaine mais sa visibilité sera faible donc nous ne testerons que les exigences de priorité Must et Could.
- Enfin 5 % des utilisateurs étant sous Safari 5, nous prenons le risque qu'une anomalie survienne sur les fonctions des priorités inférieures en n'exécutant que les tests de compatibilité de priorité Must.

Comme nous le voyons dans cet exemple, la stratégie de test par configuration matérielle fait appel à une notion de risque de visibilité des anomalies par l'utilisateur. Nous utilisons pour cela des statistiques d'usage des configurations matérielles qui sont fonction du contexte du projet également.

Le chef de projet expert en test doit donc être capable de se renseigner sur le contexte commercial des composants pour les logiciels distribués ou bien solliciter la MOA afin d'obtenir des statistiques précises de l'usage des configurations matérielles dans l'entreprise lorsque l'usage du produit se fera en interne.

De cet examen de l'existant sera extrapolé un périmètre de tests qui demandera une charge d'exécution plus petite, permettant donc à la fois une économie et simultanément de limiter le risque d'une trop forte montée en charge à l'issue de la préparation des tests.

La gestion de cette stratégie est donc un facteur important à connaître dans un projet de recette (technique ou fonctionnelle).

La stratégie globale doit donc être évitée autant que possible au profit d'une stratégie volontairement répartie selon un degré de pertinence estimé dans une approche statisticienne.

Il existe enfin une stratégie optimale : la stratégie ciblée qui permet de réduire la charge d'exécution des tests à son strict minimum.

Concrètement, cette stratégie consiste à tester 100 % des exigences fonctionnelles sur une unique configuration matérielle - logiquement celle qui est la plus répandue - puis d'effectuer des tests très ciblés et de charge faible (des sondages non formalisés pourront être suffisants).

La mise en œuvre d'une telle stratégie demande cependant un retour d'expérience, un savoir-faire des différences entre les configurations matérielles.

Une connaissance des composants permettra alors de cibler les tests sur leurs points faibles uniquement.

En effet, si deux composants sont censés être compatibles, nous pouvons logiquement supposer que leurs deux éditeurs respectifs font des tests de compatibilité de leur côté.

L'objectif de la recette du produit reste de tester ledit produit et non pas les éléments externes qui pourraient le composer pour partie, critère d'autant plus valide qu'il n'est pas garanti que l'éditeur accepterait de réparer une défaillance de son propre produit.

Par ailleurs, la concurrence commerciale qui pourrait exister entre ces deux éditeurs fait que des incompatibilités peuvent être sciemment maintenues, l'un des éditeurs tentant de rendre sa solution prédominante sur le marché.

Dans ce domaine, nous conseillons donc les chefs de projet recette technique de maintenir une base de connaissances et de faire une sorte de veille technologique des configurations matérielles, seule démarche réellement pertinente.

Car finalement, entre la stratégie globale et la stratégie ciblée, le résultat des tests pourra être sensiblement le

même. Il convient donc d'avoir une réflexion sérieuse du rapport qualité/prix entre le budget accordé aux tests de compatibilité et l'objectif qualitatif à atteindre.

## 2. Quelques retours d'expérience

### a. Les clients lourds sur PC et les applications Java

Ces clients lourds sont certainement ceux qui demandent le moins de tests de compatibilité en général.

Tout simplement parce que si une différence structurelle importante existe, le programme d'installation de l'application lui-même propose de gérer l'alternative. Vous pourrez donc au moment de l'achat comme au moment du setup, effectuer un choix qui garantira ensuite la compatibilité du produit.

Attention toutefois aux composants communs tels que les DLL et les OCX sous Windows. Et n'hésitez pas à lire les fiches techniques des composants en général ou consulter des sites spécialisés : les problèmes de compatibilité font l'objet de nombreux forums sur Internet.

Dans le cas des applications Java, le principe est différent puisque la compatibilité est assurée par la technologie elle-même : Java est portable par définition. Cette portabilité pourra cependant avoir des limites dans des cas extrêmes car elle demeure "un vœu pieux".

Trois champs d'investigation sont cependant à prévoir pour les tests de compatibilité :

- La définition de l'écran
- La langue du système d'exploitation
- Le type de compte utilisé (administrateur/non-administrateur)

Selon le contexte du projet, il est effectivement possible que la définition de l'écran soit considérée comme un paramètre amenant à envisager des configurations matérielles différentes - bien que dans la pratique le matériel et les logiciels sont identiques.

Le lecteur comprendra que derrière la notion de configuration matérielle se cache davantage un aspect logique qu'un aspect physique. Aussi, deux PC strictement identiques ayant la même version de Windows pourront être considérés comme deux configurations matérielles distinctes selon la définition d'un simple paramètre comme la définition de l'écran. Tout simplement parce que ce paramètre conditionne fortement le fonctionnement du produit ensuite : l'impact est global.

Plusieurs stratégies de test sont possibles pour prendre en compte la définition :

- Définir une configuration minimale d'utilisation, faire 100 % des tests sur celle-ci et quelques sondages non formalisés dans des définitions plus grandes.
- Définir une stratégie de test par répartition du périmètre en changeant le paramétrage plusieurs fois en cours d'exécution. Par exemple, on décompose les 100 % du périmètre en 30, 50 et 20 % respectivement sur du 800x600, 1024x768, 1280x800.
- Définir plusieurs périmètres par définition, par exemple, 100 % des tests sur du 800x600, les tests de priorité Must et Could sur du 1024x768 et les tests de priorités Must seulement sur du 1280x800.

L'avantage de ces stratégies est que n'importe quel PC correctement configuré peut matérialiser ces configurations matérielles logiques : il n'y a pas de besoin matériel spécifique.

La même considération sera faite pour le type de compte utilisé pour s'authentifier d'ailleurs, notamment si

l'utilisateur est administrateur de son PC.

En revanche, la langue du système d'exploitation est davantage une caractéristique du produit installé qu'un simple paramètre d'exécution : la notion de configuration matérielle est donc ici davantage physique et moins logique puisque la langue est choisie avec le CD d'installation du système.

Ceci signifie qu'il est nécessaire d'avoir plusieurs postes d'exécution des tests puisqu'il faudra plusieurs systèmes d'exploitation dans des langues différentes.

La stratégie de test dépend ensuite de l'architecture de la solution technique mise en œuvre pour concevoir le produit à tester.

Par exemple, un client développé sous Visual Basic pourra utiliser des composants Windows (des API) qui hériteront de la langue d'installation du système, langue qui ne sera pas nécessairement la même que celle du produit développé.

Dans le moins critique des cas, les libellés changeront de langue, comme par exemple les boîtes de dialogue d'alerte qui pourront avoir des boutons "Yes / No" au lieu de "Oui / Non" d'une application en langue française.

Mais dans le plus critique des cas, l'impact sera aussi technique, comme par exemple la bibliothèque des formules sous Excel dont le nom des fonctions des formules change avec la langue d'Excel.

Plusieurs stratégies de test peuvent alors être mises en œuvre :

- La stratégie de couverture globale, en faisant 100 % des tests dans tous les systèmes d'exploitation - mais cette stratégie pessimiste n'est pas optimale car il y aura beaucoup de redondance dans les tests.
- La stratégie de couverture pertinente, qui consiste à faire 100 % des tests sur un seul système d'exploitation dans la langue de référence, complétée pour les autres configurations matérielles, par des tests ciblés sur les points critiques.

Cette dernière stratégie est la meilleure, mais elle suppose de bien connaître la technologie utilisée, donc soit d'être en relation très étroite avec la MOE - ce qui n'est pas toujours le cas - soit d'avoir une expérience technique qui amènera à la capacité de détecter les points critiques pour cibler les tests.

## b. Les applications web

Comme nous l'avons déjà évoqué, les applications web en revanche nécessitent une stratégie de test par configuration matérielle qu'il faut adapter au contexte du projet en s'intéressant aux statistiques d'utilisation des navigateurs dans le monde, en France... ou dans l'entreprise utilisatrice.

Voici quelques exemples illustrant notre propos :

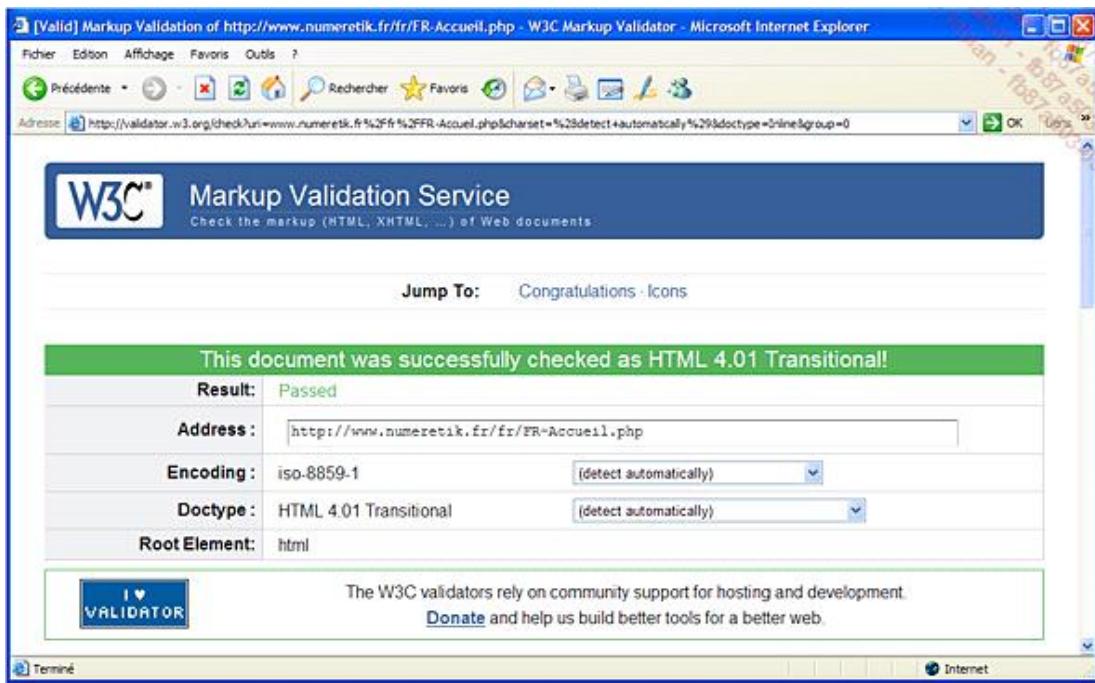
Compatibilité IE6 / IE7 / IE8 / IE9	IE9 est très récent : pas assez de recul pour l'instant pour exprimer un avis. IE7 et IE8 sont relativement proches et les anomalies rencontrées sont rares. En revanche, la compatibilité avec IE6 n'est pas du tout assurée. Ne pas hésiter à faire des tests en étant administrateur et non-administrateur du PC !
Compatibilité Firefox 3 ou 4 / Internet Explorer 6, 7, 8 et 9	Problème de design essentiellement, comme la gestion des marges dans les feuilles de style, le décalage des puces...
Compatibilité Chrome / Internet Explorer 6, 7, 8 et 9	Peu de retour d'expérience à ce jour. Chrome semble très proche de Firefox en termes de restitutions. La rapidité de Chrome est sa principale différence avec IE.

Netscape / Internet Explorer 6	Peu de différence, sinon dans le design, comme la taille des polices de caractères et des boutons. Netscape se comporte différemment dans le téléchargement des images.
Safari sous Macintosh / Internet Explorer 6, 7, 8 et 9	Les anciennes versions de Safari (1.3, 2.x) présentaient un risque important : aucune garantie qu'une application fonctionnant sous IE fonctionne aussi sous Safari. Safari semble s'être stabilisé avec le temps relativement à IE. Attention à la gestion des caractères accentués notamment dans les listes déroulantes sous Safari.

D'une manière plus large, les tests de compatibilité des navigateurs demandent une connaissance technique spécifique qu'il faut maintenir.

Une démarche intéressante est de retenir l'usage de la norme W3C qui définit le HTML 4 ou 5, ou le XHTML 1.0 pour les applications web développées, puis d'utiliser les validateurs automatiques en ligne disponible sur Internet.

Il s'agit bien sûr ici d'anticiper le problème en amont et d'imposer cette contrainte de développement afin de permettre cependant une automatisation de la vérification.



Le choix de respecter les normes du W3C n'est pas une garantie de totale compatibilité entre les navigateurs, mais il a au moins le mérite de définir un cadre de référence en matière de portabilité.

Globalement, le risque d'incompatibilité entre les navigateurs se situe surtout entre les systèmes d'exploitation Macintosh et Windows plus qu'entre les navigateurs, avec une mention spéciale pour la famille des Internet Explorer, dont le comportement à l'exécution reste fortement lié au système d'exploitation Windows.

En d'autres termes, si le contexte du projet nécessite une compatibilité sous Mac et Internet Explorer, nous conseillons de prévoir 100 % des tests sur ces deux configurations, puis des périmètres plus restreints sur les autres versions des navigateurs sur PC : Firefox, Chrome, etc. poseront moins de problèmes.

Pour ce qui est de prendre en compte la définition de l'écran, la stratégie adoptée sera similaire à celle des clients lourds évoqués dans la section précédente. Il est cependant dans les usages de choisir une définition d'écran

minimale puis d'effectuer tous les tests dans cette dernière sans se préoccuper des définitions plus grandes car les applications web ont la faculté d'être graphiquement élastiques de manière native : les composants graphiques d'une page se déplacent (voire se redimensionnent) en fonction de la taille de la fenêtre du navigateur.

La langue du système d'exploitation a un léger impact au niveau de quelques fonctions JavaScript (comme par exemple les boîtes de dialogue) mais ce paramètre n'est généralement pas pris en compte pour les applications web.

### c. Pocket PC et Palm

Ces technologies sont vieillissantes aujourd'hui mais déjà elles présentaient un risque important d'incompatibilité que nous retrouverons dans les nouvelles technologies qui s'y substituent (reportez-vous à la section suivante pour ces dernières). Nous évoquons les technologies Palm et Pocket PC à titre informatif car il est peu probable que cette typologie de projet de recette se manifeste dans un proche avenir ce qui n'enlève rien à l'intérêt de la démarche.

Tout d'abord, le type de terminal constraint à développer non pas une mais quatre applications pour les Pockets PC et Palm :

- Pocket PC communicant
- Pocket PC non communicant
- Palm communicant
- Palm non communicant

Nous avions en effet deux types de technologies seulement à cette époque et une multiplicité des systèmes d'exploitation pour les Palm (les Pocket PC étant dans des versions différentes de Windows Mobile - il n'y avait qu'un seul système possible).

Ensuite, le fait que le terminal soit communicant ou non induit un choix dans la méthode d'accès aux informations lorsque des données doivent être téléchargées soit en déplacement via une communication téléphonique pour les communicants soit à domicile via un PC pour être ensuite transférées sur le terminal mobile à l'aide d'outils de communication (comme ActiveSync). La technologie non communicante n'est plus vraiment d'actualité aujourd'hui.

Viennent ensuite d'autres considérations technologiques : l'architecture du processeur du terminal, la taille de l'écran ainsi que la gestion de la couleur pour les Palm qui pouvaient avoir des écrans monochromes (dégradés de vert ou de gris).

Notre retour d'expérience en la matière a montré qu'il fallait autant de types d'application que d'architectures d'une part, et que l'installation même de l'application mérite d'être testée !

Ces technologies mobiles étaient naissantes il y a une dizaine d'années et la multiplication des typologies, le manque de maturité des terminaux... tous ces paramètres concourraient à s'orienter vers une stratégie de tests globale.

Ainsi, nous préconisons de systématiquement tester 100 % du périmètre fonctionnel dans ce type de technologie par précaution.

Cette précaution est d'autant plus importante que la prise en compte d'écrans monochromes impacte fortement la lisibilité de l'interface graphique pour les Palm. La grande variété des définitions d'écran, les grandes différences dans les techniques visant à réduire la taille des éléments graphiques, font que la stratégie globale est la seule permettant une réponse adaptée au contexte en matière de tests.

#### d. Des architectures très variées : iPhone, smartphone...

La nouvelle génération des terminaux mobiles de l'iPhone au smartphone en passant par le BlackBerry n'a pas réglé pour autant les problématiques déjà observées pour les Pockets PC et Palm. Les écrans monochromes ont certes disparu au profit de la couleur et tous sans exception sont communicants.

Mais qu'elle soit de technologie lourde ou web, l'application développée en téléphonie mobile reste fortement liée à la technologie du terminal qui en permet l'accès. Par défaut, nous conseillons donc une stratégie de tests globale en prévoyant 100 % du périmètre fonctionnel par configuration matérielle.

En revanche, la difficulté que nous connaissons avec les Pocket PC et les Palm augmente : l'hétérogénéité de la plateforme s'intensifie avec la version du terminal, le système d'exploitation, la taille de l'écran et maintenant la typologie d'interface (tactile ou par joystick), une nouveauté puisque la technologie tactile était à ses balbutiements il y a 10 ans.

Le tableau ci-après expose ainsi pas moins de 28 typologies de terminal. Et à l'heure où nous lisez cette section, vous pouvez déjà être sûr que ce tableau est incomplet... donc obsolète !

Les conséquences pour un chef de projet de recette sont alors aussi fortes que multiples :

- D'une part, le budget consacré à l'exécution des tests augmente très fortement et avec lui la montée en charge des projets de recette relativement à la phase de préparation qui précède ce qui accroît le risque de rupture de charge avec l'apparition des anomalies bloquantes.
- D'autre part, il doit avoir une connaissance approfondie des technologies du marché de la mobilité qui est en pleine explosion de telle sorte qu'un chef de projet MOA ou même un chef de projet de recette technique, n'a généralement pas la possibilité de répondre à ce contexte : ce sujet demande une véritable expertise spécifique.
- Enfin, la dynamique technologique est telle qu'il n'est pas possible de réellement couvrir toutes les configurations : un choix doit être fait.

Il en découle que la stratégie de test d'un projet en technologie mobile suppose un axe de réflexion très fort et surtout très tôt dans le projet, sur l'usage que l'on souhaite du produit. Qui sera la cible des utilisateurs ? Dans quelle zone géographique ? Sur quel marché ? Dans quel métier ?

Nous préconisons de s'aventurer avec précaution dans cette typologie de projets de recette.

Type de terminal	Système d'exploitation	Interface	Largeur utile de l'écran
iPhone	iOS1	Tactile	320
iPhone	iOS2	Tactile	320
iPhone	iOS3	Tactile	320
iPhone	iOS4	Tactile	320
BlackBerry 8520	BB	Joystick	320
BlackBerry 9700	BB	Joystick	480
BlackBerry 8900	BB	Joystick	480
BlackBerry 9000	BB	Joystick	480
Nokia N97	Symbian	Tactile	360
Nokia N95	Symbian	Joystick	240
Nokia E71	Symbian	Joystick	320
Nokia 5800d	Symbian	Tactile	360

Nokia 5230	Symbian	Tactile	360
Nokia E63	Symbian	Joystick	320
Nokia E72	Symbian	Joystick	320
SGH F480	-	Tactile	240
Acer S100	Android	Tactile	320
Samsung Player One GT S5230	-	Tactile	240
Samsung Wave GT S8500	Wave	Tactile	240
Samsung Galaxy S	Android	Tactile	320
iPod Touch	iOS	Tactile	320
Opera Mini 5	-	-	-
Sony-Ericsson W995	-	Joystick	240
Sony-Ericsson W595	-	Joystick	240
Sony-Ericsson U5i	Symbian	Tactile	360
HTC Magic	Android	Tactile	320

# Périmètres thématiques

Au-delà du périmètre fonctionnel et technique d'une application et du choix d'une éventuelle stratégie de test des exigences de compatibilité des configurations matérielles, il existe d'autres stratégies de test, d'autres manières de considérer un périmètre de vérification.

Nous proposons ici des approches thématiques en ciblant des exigences purement techniques en laissant de côté l'aspect fonctionnel du produit.

Ces stratégies de test très spécifiques pourront répondre à des situations indépendantes ou bien une réflexion sera conduite amenant à décider de les déployer parallèlement à un projet de recette fonctionnelle ou technique.

La dimension technologique joue évidemment un rôle dans les stratégies que nous allons aborder mais il ne faudrait pas croire que telle stratégie serait purement technique et non fonctionnelle.

Le principe est de répondre à une problématique globale pour une même application.

## 1. Tester l'ergonomie et le ressenti graphique global

### a. Tester la navigation : les applications web

Par navigation, nous entendons exclusivement l'usage des liens hypertexte et des boutons permettant d'afficher toutes les pages d'un site web.

La navigation est l'un des aspects qui prédominent le plus dans cette technologie : elle fait donc l'objet de tests spécifiques d'autant plus qu'un plan du site est susceptible d'avoir été défini.

On pourra considérer qu'une recette technique ou fonctionnelle d'une application web (comme nous l'avons évoqué dans les sections précédentes) permet de tester chaque lien de sorte qu'une "navigation élémentaire", une transition, est réalisée dans chaque scénario de test ou fiche de test.

Ceci revient à dire que le test global de navigation est éclaté en autant de scénarios et fiches de test qu'il y a dans la recette.

L'approche que nous proposons ici est toute autre : elle consiste à envisager la navigation comme un seul concept devant être testé en un petit nombre de scénarios fonctionnels (voire un seul ?) ce qui correspondrait à un cas d'utilisation réel où l'internaute "butine" sur le site sans but précis.

Car la particularité d'une application web publique est aussi de proposer des contenus purement documentaires (contrairement à un client lourd par exemple). L'état d'esprit de l'utilisateur est donc bien spécifique : il peut interrompre une consultation à tout moment et entamer une autre action.

L'objectif d'une stratégie de test "orientée navigation" serait donc de cibler la détection des anomalies sur les transitions entre les pages afin de repérer d'éventuelles problématiques de fluidité.

Il ne s'agit pas à proprement parler de faire un test de performance, mais plutôt de vérifier que l'internaute reçoit un rendu et un comportement global du site web qui l'incite à naviguer... et non pas à partir.

### b. Tester la cinématique : les clients lourds

Contrairement aux applications web, les clients lourds ne sont pas utilisés dans le même contexte. L'utilisateur n'est pas en train de surfer sur le Web mais d'effectuer une action précise : il rédige un document, dessine un graphique, compose une présentation, retouche une image, fait un montage vidéo, joue... ou bien réalise une

opération administrative.

Les clients lourds ne répondent donc pas aux mêmes exigences en matière de cinématique bien que plus ou moins ressemblantes avec les clients légers.

Première différence : le double clic qui constitue une sorte de raccourci pour une action élémentaire (électionner + valider par exemple).

Deuxième différence : les temps de réponse, le client lourd étant installé sur le PC de l'utilisateur, les interactions sont plus rapides et il aura des automatismes nécessitant une réactivité logicielle plus forte que pour un client léger.

Troisième différence : les dépendances ergonomiques et les transformations dynamiques de l'interface graphique en interaction avec l'utilisateur. Ces dernières constituent un environnement graphique cinématiquement très souple et en même temps plus contraignant. L'utilisateur est dans un cadre technologique plus rigide que pour une application web.

Nous préconisons donc une stratégie de test plus spécifique pour les clients lourds, en mettant en œuvre des tests techniques dédiés à l'ergonomie de ces technologies :

- Prendre en compte le fait que la technologie est événementielle en se focalisant sur les événements de chargement et fermeture des fenêtres, les événements souris et clavier, notamment l'ordre dans lequel les événements surviennent...
- Jouer davantage avec les tabulations, la validation par la touche [Entrée], les clics simples et doubles...
- Vérifier le comportement de la barre des tâches de votre système d'exploitation car cette notion n'existe pas pour les applications web (la barre des tâches ne contient que l'icône du navigateur et la valeur de l'attribut title de la page HTML), notamment en lançant plusieurs fois l'application par exemple.
- Tester l'installation et la désinstallation bien sûr !
- Etc.

À contexte différent, stratégie de test différente.

## 2. Tester le multilinguisme, la langue d'une application

### a. Les fautes d'orthographe

Qu'elle soit de technologie web ou un client lourd, une application présente toujours des libellés pourvu qu'elle ait une interface graphique.

Et il est souvent considéré que la qualité d'un logiciel se mesure à l'absence de faute d'orthographe : c'est un critère de finition qui est immédiatement observable.

Deux stratégies de test sont possibles dans cette thématique : vérifier l'orthographe dans chaque fiche de test (jamais dans un scénario fonctionnel, ces contrôles sont faits en recette technique) ou bien rédiger une unique fiche de test qui parcourt la globalité de l'application et tente de les recenser toutes.

Cette stratégie s'apparente à celle de la navigation globale puisqu'elle consiste à visiter le périmètre le plus étendu possible de l'interface graphique pour détecter toutes les fautes - en incluant jusqu'au message d'erreur des formulaires.

A priori, il n'y a pas une stratégie qui soit meilleure que l'autre pourvu que le recensement des fautes soit

centralisé.

Nous reviendrons sur ce sujet dans le chapitre dédié à la gestion des anomalies.

## b. Le multilinguisme

Le multilinguisme, c'est-à-dire l'aptitude à pouvoir traduire l'interface graphique d'une application en plusieurs langues, a une spécificité qui mérite qu'on s'y attarde dans la stratégie de test à mettre en œuvre.

Tout d'abord, la qualité d'une application à avoir une interface graphique en plusieurs langues peut être comprise de deux manières :

- Comme une autre version de l'application.
- Comme une fonctionnalité à l'intérieur de la même application.

Le premier cas est un point de vue qui relève généralement des clients lourds - sans que cela soit une règle générale - et le plus souvent, la langue de l'interface est un paramètre demandé dès son installation - voire une caractéristique du CD d'installation lui-même.

Le second cas relèverait davantage des clients légers - sans que cela soit une obligation non plus selon la manière dont le site web est développé. En effet, si un site web ne se compose que de formulaires, il est assimilable à une application qui a la faculté de pouvoir changer de langue, mais s'il se compose aussi de contenus multimédias qui ne sont pas tous traduits, alors on pourra considérer qu'il y a deux sites web dans deux langues différentes.

Cette manière de voir le multilinguisme d'une application lourde ou légère conditionne fortement la stratégie de test à mettre en œuvre.

Si la langue est liée à une version spécifique de l'application pour les clients lourds, il ne sera pas toujours possible d'en avoir simultanément deux instances dans deux langues sur le même PC. Et si la fonction permettant de basculer d'une langue à l'autre n'existe pas, qu'elle est un choix fait sur la page d'accueil du site pour les clients légers, alors cela signifie que tous les tests fonctionnels et/ou techniques s'effectuent en une seule langue.

Dans un tel cas, on considérera que l'on a alors deux applications qu'il convient de tester séparément dans les deux langues, situation qui est assimilable à la gestion d'une configuration matérielle supplémentaire : nous vous renvoyons donc à ce chapitre pour organiser votre périmètre selon les mêmes principes, puisque vous pouvez décider de faire 100 % des tests dans cette deuxième langue ou vous limiter à un périmètre plus petit, sur la base de la priorité des tests.

La langue, tout comme la définition des écrans, détermine donc une configuration matérielle logique.

Mais le multilinguisme peut aussi être considéré comme une fonction permettant dans n'importe quel écran ou page, de basculer dans une autre langue ce qui le plus souvent concerne les applications web.

Dans cet autre cas, cela signifie que certains scénarios fonctionnels et l'ensemble des fiches de tests techniques de l'application doivent avoir une action de test gérant la bascule entre les langues.

Cette démarche permettra :

- En recette technique, de vérifier l'aptitude de bascule dans chacune des fiches de tests ou bien d'exécuter une unique fiche de tests techniques de navigation vérifiant globalement les bascules dans toutes les pages en les parcourant une à une.
- En recette fonctionnelle, de vérifier la cohérence à changer de langue au cours de l'exécution d'une exigence fonctionnelle, comme si nous étions un utilisateur final.

Mais surtout, et comme nous le verrons dans un prochain chapitre, l'impact du multilinguisme sur une recette est d'abord organisationnel ce qui induira un choix à faire dans la stratégie de test à adopter en fonction de contrainte de ressources.

# Définir des jeux d'essai pertinents

Le chapitre précédent nous a permis de dégager les informations suivantes :

- Pour le projet :
  - Les parties prenantes, les acteurs du projet.
  - La hiérarchie des exigences projet.
  - La matrice des risques projet, une évaluation de leur criticité par les parties prenantes et les actions palliatives associées.
- Pour le produit, le ou les logiciels à réaliser :
  - Les parties prenantes, c'est-à-dire des catégories d'utilisateurs de la solution future.
  - La hiérarchie des exigences fonctionnelles et techniques du produit.
  - La matrice des risques produit, une estimation de leur criticité selon les catégories d'utilisateurs et le type de stratégie de test associé (tests unitaires, recette technique, recette fonctionnelle, tests de robustesse, tests de charge...).

De la partie produit, nous déduisons alors différents périmètres de tests obtenus par un rapprochement et une priorisation des risques et exigences produit : des plans de test.

Cette approche constitue le point de départ de la construction du périmètre des données nécessaires aux tests, que cela soit pour les tests unitaires, une recette technique ou une recette fonctionnelle.

Mais avant d'entrer dans le vif du sujet, il convient de définir ce que l'on entend par l'expression "jeu d'essai" car un test est tributaire de différents types de données :

- Les paramètres de l'application, des informations relativement statiques qui en général restent fixées une fois pour toutes pour une campagne de test donnée (par exemple, le timeout session d'une application web, le mode de fonctionnement, etc.).
- Les données devant préexister avant l'exécution des tests, le plus souvent des référentiels qui évoluent peu dans le temps (tables de référence de pays, référentiel produit, comptes utilisateur, permissions...) mais aussi des données "cœur de métier" que l'on pourra saisir par avance. Ces données seront davantage des prérequis qu'un jeu d'essai : nous considérerons qu'elles participent à l'environnement de recette.
- Les données à saisir pendant les tests pour remplir des formulaires par exemple que nous désignons par l'expression "jeu d'essai" qui comme elles l'indiquent supposent des tentatives d'utilisation de la solution.

Dans tous les cas, il conviendra de rédiger un cahier des jeux de données qui décrira aussi bien les données préexistantes que les jeux d'essai. Le présent chapitre vous propose une démarche pour rédiger celui-ci.

## 1. Construire le périmètre des données

La section qui suit vous présente une démarche générale pour construire un périmètre de données pour une recette qu'elle soit fonctionnelle ou technique.

### a. Démarche de construction des jeux de données d'une recette fonctionnelle

Dans le cadre d'une recette purement fonctionnelle, le plan de test issu de la double analyse des risques et des exigences fournit deux choses : la liste des exigences fonctionnelles et pour chacune d'elle, éventuellement, un

profil utilisateur nécessaire pour exécuter le test.

Nous dégageons donc un premier ensemble de données prérequis : les profils utilisateurs. Les priorités des tests (Must, Could, Would et Should) permettent d'ailleurs de définir l'importance des profils relativement aux exigences.

Le responsable de la recette devra alors étudier exigence par exigence les données de référence devant impérativement exister au préalable. Les spécifications fourniront à ce titre les informations permettant d'identifier ces données avec précision.

- Dans le cadre des applications ayant une interface graphique, intéressez-vous aux listes déroulantes, listes de choix, radio-boutons, profil utilisateur, permissions, rôles... les pays, les départements, les référentiels tiers (clients, fournisseurs, distributeurs, transporteurs...), les moyens de paiement, etc.

L'objectif sera ici de cerner un périmètre de données prérequis permettant un "fonctionnement de démarrage" de la solution. Nous illustrerons cette démarche par un exemple en supposant que le module contact d'une application web propose à un internaute de saisir son adresse postale, cette dernière disposant d'une liste déroulante de pays.

Si nous supposons que l'application est déployée dans un premier temps aux seuls administrateurs, il y a donc une première étape de fonctionnement de la solution qui consisterait à saisir les données de référence : l'administrateur pourra donc se connecter (et pas les internautes) pour définir la liste des pays. Une fois cette liste de pays définie, le module de contact devient opérationnel.

Il faut surtout comprendre qu'il y peut y avoir autant de phases dans les tests qu'il y a de phases d'utilisation de la solution en production - le plus souvent quatre :

- Un état de l'application en phase de lancement, pendant laquelle les fonctionnalités permettant de saisir les référentiels seront fortement sollicitées, le cœur de métier étant peu testable ou seulement pour des cas particuliers, car il présente un risque de dégradation ou un fonctionnement "aux limites inférieures de la solution".
- Un état de la solution en début de montée en charge, pour lequel un référentiel minimal permet un fonctionnement normal de l'application, avec des données de référence prêtes et qui évolueront très peu dans le temps.
- Un état nominal de la solution présentant "une certaine volumétrie de données métier", état que nous pourrions appeler "état normal" ou "état habituel".
- Et enfin un état plus rarement testé qui est l'état de saturation de la solution, c'est-à-dire aux limites supérieures de ses capacités technologiques.

Bien évidemment, les deux états intermédiaires intéressent tout particulièrement les recettes fonctionnelles, les premiers et derniers concernant préférentiellement les recettes techniques. Mais c'est le contexte projet qui permettra de savoir quels états de la solution nécessitent une recette technique et/ou fonctionnelle.

Une fois les données du référentiel identifiées pour chaque exigence (donc un environnement de données), le périmètre des jeux d'essai reste à définir.

Il s'agit le plus souvent de répertorier pour chaque scénario de test, une liste de jeux d'essai (ou un seul jeu d'essai), c'est-à-dire les données à saisir au moment du test.

- Dans le cadre de la recette fonctionnelle d'un flux, les jeux d'essai ne résulteront pas nécessairement d'une saisie mais plus probablement d'un échantillonnage. La démarche permettant l'identification des jeux d'essai est ici la même que pour les tests unitaires d'un flux : les exigences permettent alors de repérer les différentes typologies de données à traiter pour assurer une couverture optimale des tests.

Chaque exigence est couverte par un scénario "non valorisé", c'est-à-dire une succession d'actions de vérification pour lesquelles le jeu d'essai est exprimé sous la forme de variables utilisées pour rédiger leur description : un modèle de cahier de recette.

Un scénario de tests se subdivise alors en autant de cas de test qu'il y a de jeux d'essai, un cas de test s'obtenant par recopie du scénario puis valorisation des variables que les tests contiennent.

Le rôle d'un jeu d'essai est donc d'être exécuté au moins une fois, ce jeu pouvant être réutilisable ou pas comme nous le verrons plus loin, les données étant soit saisies pendant l'exécution même du test, soit au préalable, soit les deux.

Le rôle de l'ensemble des jeux d'essai d'une même exigence fonctionnelle est alors de couvrir celle-ci autant que possible comme nous l'avions évoqué dans le chapitre Généralités vous exposant les concepts généraux de la démarche projet. Chaque cas de test d'une recette fonctionnelle est donc :

- un test nominal,
- ou un test aux valeurs clés,
- ou un test aux limites,
- ou un test hors domaine (non passant).

C'est-à-dire un test qui a une dimension purement fonctionnelle et en aucun cas technique.

 Il est impératif ici de ne pas perdre de vue qu'au cours d'une recette fonctionnelle le testeur se positionne en tant qu'utilisateur final de la solution. En conséquence, les tests fonctionnels doivent conserver cette dimension. Cet avertissement est d'autant plus important que les tests pourront être exécutés par des testeurs issus du métier et ne connaissant absolument pas le monde du test et de la production logicielle. Ceci signifie que les jeux d'essai ont un sens, une pertinence métier, avant toute chose.

Voici un exemple significatif pour illustrer la démarche :

- Pour une exigence fonctionnelle "Liste paginée des opérations d'un compte bancaire" :
  - Cas de test 1 : la liste vide.
  - Cas de test 2 : une liste d'une seule page d'opérations.
  - Cas de test 3 : une liste de deux pages d'opérations.
  - Cas de test 4 : une liste d'au moins trois pages d'opérations.
- Pour une exigence fonctionnelle "Réaliser un virement interne compte à compte" :
  - Cas de test 1 : réaliser un virement interne de 0 euro pour un abonné possédant deux comptes courants (test non passant de la borne inférieure).
  - Cas de test 2 : réaliser un virement interne de 250 euros pour un abonné possédant deux comptes courants (cas nominal).
  - Cas de test 3 : réaliser un virement interne pour un abonné possédant un unique compte courant (cas limite d'utilisation).
  - Cas de test 4 : réaliser un virement interne pour un abonné possédant un unique compte en euros et un compte en dollars (autre cas limite d'utilisation).

Dans les exemples proposés ci-dessus, nous identifions les besoins en matière de profil utilisateur :

- Un abonné titulaire d'un unique compte courant.
- Un abonné titulaire de deux comptes, l'un en euros, l'autre en dollars.
- Un abonné titulaire d'au moins deux comptes en euros (le cas général).

Puis les besoins en matière de compte, des données prérequisées au même titre que les abonnements :

- Un compte sans opération
- Un compte avec une page d'opérations
- Un compte avec deux pages d'opérations
- Un compte avec au moins trois pages d'opérations

Enfin les besoins en matière de jeux d'essai proprement dit :

- Un jeu d'essai "0 euro" pour l'exigence "Réaliser un virement interne".
- Un jeu d'essai "250 euros" pour l'exigence "Réaliser un virement interne".

L'approche est ensuite combinatoire entre les prérequis des deux exigences. Nous pourrions donc envisager d'avoir un abonné titulaire d'un unique compte courant sans opération, un autre titulaire d'un unique compte avec N pages d'opérations, un autre avec deux comptes dont un sans opération et l'autre avec N pages d'opérations, etc.

Le lecteur comprend ici que la démarche combinatoire fait exploser la taille des données prérequisées et des jeux d'essai.

Il convient donc de trouver une stratégie permettant de limiter rapidement la dimension du jeu de données, d'une part parce que la conception et la saisie même des informations pourraient devenir chronophages, d'autre part parce qu'il n'y a pas toujours pertinence à tester toutes les combinaisons.

Ainsi, le chef de projet de la recette fonctionnelle doit se poser sans cesse la question de la pertinence du test ou de sa priorité.

En reprenant l'exemple précédent, nous pouvons alors déduire le besoin comme suit :

- Un abonné titulaire d'un unique compte courant, ne comportant aucune opération, qui permettra de tester à la fois l'impossibilité de réaliser un virement interne lorsqu'il n'y a qu'un seul compte et le cas de consultation des opérations d'un compte lorsqu'il n'y en a pas.
- Un abonné titulaire de deux comptes, l'un en euros, l'autre en dollars, qui permettra à la fois de tester le virement interne lorsqu'il n'y a qu'un seul compte en euro et la pagination des opérations pour une ou deux pages pourvu que :
  - Le compte en euros comporte une seule page d'opérations.
  - Le compte en dollars comporte deux pages d'opérations.
- Un abonné titulaire d'au moins deux comptes en euros (le cas général), l'un des deux comptes comportant au moins trois pages d'opérations, et qui permet d'exécuter les cas de test généraux.

La démarche consiste donc à mutualiser autant que possible les données prérequisées dans une logique d'économie et de pertinence fonctionnelle, en procédant par spécialisation : les prérequis sont structurés de sorte

qu'il y a d'un côté un nombre limité de cas particuliers et de l'autre un nombre limité permettant de tester les cas de test nominaux.

Puis de manière itérative, la conception des prérequis et jeux d'essai consiste à prendre une exigence supplémentaire pour :

- Étendre l'arbre des données prérequises par combinaison avec les besoins de l'exigence ajoutée, en faisant en sorte que l'arbre existant couvre déjà les besoins en question au maximum et ainsi limiter sa taille.
- Inclure les jeux d'essai nécessaires pour couvrir l'exigence ajoutée et construire une liste de cas de test liés à un profil et des données précises assurant une bonne couverture des exigences.

Malheureusement, le processus que nous proposons ici n'est pas aussi simple qu'il y paraît comme nous allons le voir dans les sections qui suivent : de la théorie à la pratique, d'autres contraintes doivent alors être prises en compte.

## b. Démarche de construction des jeux de données d'une recette technique

La démarche est identique à celle d'une recette fonctionnelle dans l'approche mais elle diffère ensuite dans la nature des tests à conduire donc des jeux d'essai pour :

- des tests nominaux - comme pour une recette fonctionnelle,
- des tests aux valeurs clés - comme pour une recette fonctionnelle,
- des tests aux limites - comme pour une recette fonctionnelle,
- des tests hors domaine ou non passant - comme pour une recette fonctionnelle,
- des tests de robustesse, spécifiques à la recette technique,
- des tests de configuration matérielle, spécifiques à la recette technique,
- des tests de performance, spécifiques à la recette technique.

La spécificité d'une recette technique impliquera des besoins en données précis notamment pour les jeux d'essai puisque le panel des données à saisir pendant les tests pourra être plus étayé afin de soumettre l'application à un stress plus fort et vérifier un nombre de cas plus important.

L'exemple évoqué précédemment du test d'un virement bancaire illustrera notre approche. La recette technique nécessitera le même besoin qu'une recette fonctionnelle, c'est une évidence. Mais il faudra aussi prévoir des cas plus nombreux :

- Un jeu d'essai " 0 euro ", comme pour une recette fonctionnelle.
- Un jeu d'essai " 250 euros ", comme pour une recette fonctionnelle.
- Un jeu d'essai " -1 euro ".
- Un jeu d'essai " 250 euros " (test de vérification de la présence d'espace à gauche et à droite du montant).
- Un jeu d'essai " 1 000 euros " et un autre " 1000 euros " (test pour vérifier l'acceptation d'un séparateur des milliers).
- Des jeux d'essai ".5" ".05" "100." et " 100.0 " pour tester la gestion des décimales en saisie.
- Un jeu d'essai à " 999...9 999 euros " pour tester le plafond maximum.
- Etc.

Le lecteur retiendra donc que les données prérequises d'une recette fonctionnelle et d'une recette technique sont

globalement les mêmes : les référentiels doivent être alimentés et permettre un usage normal de l'application. La différence se situe davantage dans les jeux d'essai dont le périmètre sera nécessairement plus large en recette technique puisque l'on cherchera à en dépasser les limites.

À moins bien sûr que la recette technique ne vise les états limites de la solution comme nous l'avons vu dans la section précédente (phase de lancement et phase de saturation) mais ces recettes techniques restent rares.

Cependant, comme nous allons le voir maintenant, la distinction se fera par la prise en compte d'autres contraintes.

### c. Impact de l'organisation sur la constitution du jeu d'essai

Une recette fonctionnelle ou technique consiste à monter une équipe de testeurs qui travaillent de concert et simultanément avec la même solution logicielle.

Se pose alors une question fondamentale : quelle(s) partie(s) des données prérequises est ou sont partageables entre les membres de l'équipe ?

Par exemple, il n'est généralement pas une bonne idée d'utiliser un même compte utilisateur depuis N postes de test simultanément (et parfois c'est même techniquement impossible).

Et il n'est pas toujours possible non plus de gérer un partage de données entre des utilisateurs d'une solution. Le premier exemple évoqué dans ce chapitre avec la gestion d'une liste de pays nous montre qu'une partie des données est logiquement partagée : les données de référence.

Le second exemple nous proposait trois profils d'abonnés à un service bancaire et un ensemble de comptes courants, c'est-à-dire des données métier et non de référence, qui par essence ne sont pas partagées : un compte en banque est une donnée très personnelle.

Si nous supposons maintenant que l'équipe de test est constituée de quatre personnes une nouvelle contrainte apparaît qui se décline en deux choix possibles :

- Soit chaque testeur a un périmètre de test qu'il ne partage pas avec le reste de l'équipe et qui est une sous-partie du périmètre global : il n'y a pas d'intersection avec les périmètres des autres membres de l'équipe.
- Soit chaque testeur a un périmètre de test qui se superpose parfois avec celui d'un autre membre de l'équipe : il y a des intersections de périmètres.

Dans le premier cas, le périmètre des tests est réparti entre N personnes qui peuvent travailler de manière autonome puisque chacune a ses propres données et jeux de données. C'est souvent le cas des recettes fonctionnelles.

Dans le second cas, un même prérequis ou un même jeu d'essai pourra être partagé (ou ne pourra pas l'être) ce qui signifie qu'il sera peut-être nécessaire de saisir les données autant de fois qu'il a de testeurs. Cette situation apparaît notamment lors des recettes techniques nécessitant l'exécution d'un même test dans plusieurs configurations matérielles.

La répartition des tests entre les membres de l'équipe a donc un impact direct sur les données prérequises et le jeu d'essai : il sera peut-être nécessaire de le dupliquer tout ou partie.

Le responsable de l'organisation de la recette doit alors :

- Identifier les données prérequises susceptibles d'être mutualisées entre les testeurs afin d'éviter les saisies multiples.

- Identifier clairement les données prérequis qui ne devront surtout pas être partagées entre les testeurs.
- Identifier les données des jeux d'essai qui feront l'objet de contrôle d'unicité (par exemple, la saisie d'un identifiant dans un processus d'inscription) car cette contrainte oblige à associer l'information à un testeur précis.

La prise en compte de cette contrainte de partage ou de non-partage des données se résout par une stratégie d'indexation des jeux d'essai et des prérequis. Voici la solution que nous préconisons.

Qu'elle soit technique ou fonctionnelle, la recette débute par une phase d'organisation au cours de laquelle la charge des tests et la planification sont étudiées. C'est à ce moment-là que l'équipe est dimensionnée.

Le chef de projet de la recette doit bien sûr prendre en considération qu'il existe parfois un risque de mauvais dimensionnement de son équipe. Aussi, s'il était nécessaire de faire intervenir des testeurs supplémentaires, la principale difficulté pourrait alors être dans l'inadéquation des jeux d'essai et données d'environnement.

Nous partirons cependant du principe que le risque est faible : les testeurs qui interviennent seront donc numérotés de 1 à N. Attention, la taille de l'équipe qui exécute les tests n'est pas nécessairement identique à celle qui les conçoit.

 Notez qu'au moment de l'organisation du projet, il est très probable que les noms des testeurs vous soient inconnus. La numérotation "Testeur 1", "Testeur 2"... est donc purement arbitraire et il vous appartiendra ensuite d'établir une correspondance entre l'identité du testeur et le numéro qui lui est affecté. Par ailleurs, cette démarche permet aussi de changer les affectations en cours de test, chose qui s'avérera utile parfois.

Une fois les testeurs numérotés, vous pourrez alors procéder à une indexation des données relativement au numéro du testeur.

Si nous reprenons notre exemple bancaire, nous déclinons alors non pas trois abonnements mais neuf, en supposant que l'équipe de test soit de trois ressources, ce qui donnera alors ceci :

	Testeur 1	Testeur 2	Testeur 3
Cas A : abonné titulaire d'un unique compte courant	Login : A00001 Mdp : 123456	Login : A00002 Mdp : 123456	Login : A00003 Mdp : 123456
Cas B : abonné titulaire de deux comptes, l'un en euros, l'autre en dollars	Login : B00001 Mdp : 123456	Login : B00002 Mdp : 123456	Login : B00003 Mdp : 123456
Cas C : abonné titulaire d'au moins deux comptes en euros	Login : C00001 Mdp : 123456	Login : C00002 Mdp : 123456	Login : C00003 Mdp : 123456
Montant du virement	100	200	300

L'indexation des données en fonction du numéro du testeur permet ainsi à chaque membre de l'équipe auquel un numéro est attribué, de connaître mécaniquement le périmètre des données dont il a besoin, que ces données soient prérequis (login et mot de passe) ou que ces données soient un élément d'un jeu d'essai (le montant du virement par exemple).

Dans notre exemple, le lecteur comprend immédiatement la pertinence à séparer les profils des abonnés entre les testeurs, l'indexation permettant une identification rapide de "qui travaille avec quoi".

La nouvelle problématique qui apparaît alors est qu'il faut démultiplier le jeu d'essai pour autant de testeurs qu'il y a, chose qui ne sera pas toujours pertinente.

 Attention, selon le contexte imposé par l'application, vous serez peut-être obligé de multiplier les données pour

autant de testeurs qu'aura votre équipe !

En réponse à ce nouveau problème, le responsable de la recette devra peut-être chercher une stratégie d'optimisation afin de mutualiser certaines données.

Toujours dans notre exemple d'application bancaire, nous venons de voir qu'il fallait neuf abonnements, ce qui suppose que  $5 * 3 = 15$  comptes courants sont nécessaires à maxima, c'est-à-dire en dupliquant intégralement les données. L'un de ces comptes comporte trois pages d'opérations. Si nous supposons maintenant qu'une page comporte 20 opérations (donc que le compte en comporte 60) nous arrivons à un total de 180 opérations à prédefinir pour une équipe de trois personnes.

La question se pose alors de savoir l'intérêt qu'il y aurait à démultiplier les comptes et les opérations sur ces derniers en regard du nombre de testeurs.

Deux partis pris sont possibles :

- Soit on considérera que chaque testeur doit avoir ses propres données et être absolument indépendant.
- Soit on considérera que la mutualisation des comptes et des opérations est envisageable pour à la fois réaliser une économie (la taille des données saisies est plus petite) et pour permettre d'effectuer une comparaison des résultats d'un même test avec une donnée identique.

Le deuxième parti pris est notamment intéressant lors des recettes techniques, typiquement pour faire de la comparaison de résultat entre des configurations matérielles pour un même test car le fait d'avoir une donnée identique augmente la précision du test : s'il y a anomalie et que le traitement comme la donnée est identique, il est alors certain que l'origine vient du changement de configuration. En revanche, si la donnée n'est pas la même, une inconnue subsiste quant à l'origine de l'anomalie. Mais comment partager des données entre les testeurs ?

Tout d'abord, l'application doit autoriser cela, c'est-à-dire qu'il existe une capacité fonctionnelle au partage. Si ce n'est pas le cas, cela signifie qu'aucune optimisation ne sera possible donc qu'il faudra soit accepter la duplication des données pour chaque testeur, soit réviser le périmètre des tests par ressource si le budget ne permet pas la duplication.

Lorsque le partage est fonctionnellement possible, celui-ci consiste alors à détourner une fonctionnalité pour en faire une "commodité d'organisation" : partager les données entre les testeurs. Car il faut bien comprendre que ce partage n'a aucune justification fonctionnelle et ne correspondra à rien au sens métier : il n'est qu'une convenance dans le cadre d'une recette.

Appliquons ce procédé à notre exemple d'un site bancaire. En première analyse, une opération est liée à un seul compte bancaire et ne peut être partagée : la définition fonctionnelle du lien entre l'opération et le compte ne permet donc pas le partage.

En revanche, la notion de titulaire d'un compte bancaire accepte les liens multiples entre un abonné et les comptes : il existe les notions de co-titulaire et de mandataire d'un compte bancaire. Concrètement, cela signifie qu'il est possible de lier des abonnements différents (de N testeurs) à un même compte, en détournant la notion de titularat multiple de son rôle fonctionnel pour en faire une commodité d'organisation : faire en sorte qu'un même compte bancaire soit partagé par tous les testeurs.

Ainsi, la construction des données prérequises nécessite toujours 9 abonnements (3 par testeurs) mais le nombre de comptes bancaires sera de 5 au lieu de 15. Quant au nombre d'opérations, il sera de 60 au lieu de 180, ce qui divise donc par 3 le nombre de saisies et la charge de travail que cela suppose.

Il y a bien sûr une limite à cette démarche d'optimisation. Car si lors d'une recette technique, on pourra se permettre de détourner une fonctionnalité de son rôle, ce ne sera peut-être pas le cas en recette fonctionnelle. Seul le contexte du projet permettra de dire ce qu'il est possible de faire sur ce point.

Nous terminerons ces considérations de l'impact de l'organisation sur le jeu d'essai en soulignant que la problématique de saisie des données uniques (comme des identifiants) dans un formulaire trouve dans la numérotation des testeurs une réponse.

Ainsi, si nous considérons les tests d'un processus d'inscription, nous pouvons convenir de rattacher les jeux d'essai à un numéro de testeur là aussi :

	Testeur 1	Testeur 2	Testeur 3
Adresse mail du compte à créer	Login : hugo1@free.fr Mdp : 123456	Login : hugo2@free.fr Mdp : 123456	Login : hugo3@free.fr Mdp : 123456
Âge de la personne inscrite	10	10	10

Le lecteur comprend aisément que seul l'identifiant a nécessité à être unique, le numéro du testeur intégré dans l'adresse mail créant de fait l'unicité. En revanche, des données telles que l'âge et le mot de passe pourront être identiques... ou pas selon le besoin.

#### **d. Impact du planning projet sur la constitution du jeu d'essai**

Un autre risque devra être anticipé lors de la définition des jeux d'essai et données prérequis : la chronodépendance.

Concrètement, si nous avons des dates futures à saisir (que celles-ci soient définies dans les données préexistantes ou à saisir au moment du test), leur pertinence est fonction de la date à laquelle le test est exécuté.

Ce risque a une implication non négligeable, notamment en cas de glissement du planning des tests. Car si des données prérequis ont déjà été saisies ou si des jeux d'essai ont été formalisés (dans un outil de test ou un document), les dates futures pourront devenir obsolètes.

Dans une telle situation, il serait alors nécessaire de saisir à nouveau toutes les dates futures relativement à la nouvelle date d'exécution des tests. Cette solution, lorsqu'elle est manuelle, est relativement fastidieuse. Elle pourra être optimisée par l'usage d'un automate modifiant dynamiquement les données, sous réserve de faisabilité également.

Une autre solution - mais qui n'est pas toujours techniquement réalisable - consistera à modifier la date du jour afin de "déplacer l'équipe dans le temps" pour la placer dans la période initialement prévue.

Enfin, la prise en compte de ce risque peut consister aussi à évaluer le glissement et prévoir une marge de manœuvre en plaçant les dates futures suffisamment loin dans l'avenir pour qu'elles restent futures.

Seul le contexte du projet permettra de définir une solution sur cet aspect.

#### **e. Anticiper la consommation des données**

Une donnée pourra être utilisée de quatre manières lors d'un test :

- Pour une lecture seule, sans détérioration.
- Pour une création (par utilisation d'un jeu).
- Pour une écriture, sans création d'enregistrement ou une suppression logique réversible (par utilisation d'un jeu et écrasement d'un prérequis).
- Pour une suppression physique ou logique irréversible (suppression d'un prérequis donc).

Dans les deux premiers cas, l'exécution des tests n'a pas d'incidence forte sur le jeu d'essai. Dans les deux derniers en revanche, une transformation a lieu.

La modification ou la suppression logique pourra être réversible, le plus souvent par l'application elle-même, mais présente un risque en cas d'anomalie : qu'elle entraîne une détérioration empêchant tout nouveau test.

Quant à la suppression physique, la destruction des données qu'elle entraîne empêchera tout nouveau test : il y a donc une contrainte forte à prendre en compte lors des tests. Que la recette soit technique ou fonctionnelle, dans les deux cas son organisateur devra absolument anticiper cette problématique : le nombre de tests est limité.

Cette limitation oblige donc à trouver une stratégie de réservation, voire de modifier la stratégie de test afin d'éviter tout blocage. Par exemple, le test d'un porte-monnaie électronique contenant 100 euros permettra de faire deux tests de débit de 50 ou 100 tests d'un débit de 1 euro. Cet exemple est issu d'une situation réelle, la fourniture du porte-monnaie électronique de test ayant une limitation technique imposée par l'environnement.

Afin de gérer cette problématique, nous conseillons la stratégie suivante :

- Tester les fonctions de création d'une donnée et immédiatement après celle permettant sa suppression.
- Puis tester les fonctions de consultation des données, soit sur des données existantes, soit sur des données venant d'être créées.
- Puis tester enfin les fonctions de modification des données, les tests de consultation ayant été déroulés préalablement.
- Et enfin tester la suppression des données venant d'être créées ou la modification irréversible.

Cette démarche de test permet ainsi de détruire les données qui auraient été mal créées dans un premier temps tout en conservant les données préexistantes non touchées par les tests.

Puis la réalisation des tests de consultation de données venant d'être créées sera possible :

- Soit parce que la fonction de création aura été validée avant et fournira les données à tester en consultation.
- Soit parce que la consultation sera testée sur les données prérequises, non touchées par les tests précédents.

Enfin, les tests de suppression ou d'actions irréversibles placés en fin de campagne permettront d'éviter la contrainte en plaçant les consommations à risque en fin d'itération.

À toutes fins utiles, nous rappelons aussi les contraintes imposées par les séquenceurs, structures qui permettent dans des bases de données de générer un numéro unique, et qui sont généralement irréversibles dans la plupart des systèmes de gestion des bases de données (un rollback SQL ne permet pas de revenir en arrière sur la valeur d'une séquence).

Nous rejoignons ici la problématique de création de jeu possédant une contrainte d'unicité puisqu'il arrive un moment dans les tests où un point de blocage est susceptible d'apparaître.

Mais la solution que nous proposons ici n'est jamais qu'une action partiellement palliative : il n'y a pas de solution miracle et un jeu d'essai qui est irréversiblement consommé nécessitera toujours de réfléchir au nombre d'exécutions possibles d'une exigence fonctionnelle.

Nous vous invitons à vous reporter à la section qui suit sur les mécanismes de sauvegarde et restauration d'une base de données, susceptibles eux aussi d'apporter une réponse à la problématique de la consommation des jeux d'essai.

 Nous n'évoquerons pas ici les cas extrêmes pour lesquels le nombre d'exécutions possible est très limité, comme par exemple un logiciel qui piloterait un outil de découpe d'un diamant : le processus de recette proposé s'adresse davantage à une production logicielle "plus conventionnelle" comme celui en vigueur pour l'informatique de gestion.

## f. Dépendances fonctionnelles et jeux d'essai

Comme nous l'avons déjà évoqué dans le chapitre précédent de définition d'un périmètre de tests, une contrainte de dépendance entre des exigences fonctionnelles devra parfois être prise en compte.

Concrètement, cela signifie qu'une exigence produit une donnée consommée par une autre exigence, comme par exemple une création et la suppression de la donnée créée. De ce fait, les dépendances fonctionnelles ont un impact sur la constitution du jeu d'essai.

Deux stratégies peuvent être mises en œuvre :

- Garder la dépendance en ordonnançant les deux tests, auquel cas il sera préférable de les faire exécuter par la même personne pour s'éviter une synchronisation de tâche.
- Casser la dépendance artificiellement en créant une donnée prérequise destinée à la deuxième exigence, ce qui permettra de la tester avant la première si c'était nécessaire.

La première solution est moins coûteuse en termes de charge puisque la donnée créée par le premier test est consommée par le deuxième, mais elle constraint le chef de projet à faire réaliser les deux tests dépendants l'un après l'autre, ce qui parfois ne sera pas possible. Par exemple, si le premier test est un import de données planifié très consommateur en temps et le second un test manuel.

Quant à la seconde, si elle permet une souplesse d'organisation en s'affranchissant de la dépendance, elle constraint toutefois à des saisies supplémentaires : elle induit donc une charge plus importante.

Dans l'exemple évoqué d'un import de données, nous pourrions d'un côté tester l'import en lançant l'action la nuit, et de l'autre réaliser le test manuel avant en utilisant un échantillon de données manuellement saisi.

Là non plus, il n'y a pas de solution miracle et seul le contexte du projet permettra de déterminer le meilleur choix.

Nous attirons simplement l'attention du lecteur sur le fait que le cahier des jeux de données devra idéalement mettre en évidence les dépendances fonctionnelles afin qu'il soit clairement mentionné que tel scénario nécessite un jeu d'essai produit par tel autre.

## g. Impact du support technique du jeu de données

Qu'elle soit technique ou fonctionnelle, la recette reste tributaire des technologies employées dans les différents environnements où elle s'exécute. Cette incontournable contrainte nous amène donc à attirer votre attention sur ses impacts en matière de gestion des jeux d'essai.

L'approche que nous avons proposée pour construire les jeux d'essai et données d'environnement requises s'appuyait sur le postulat que la solution logicielle était une base de données relationnelle. Mais il n'échappera pas au lecteur que toute solution n'utilise pas nécessairement cette technologie.

Un traitement pourra être une transformation de fichiers, un import ou un export, le traitement d'un flux XML...

Bien évidemment, il y a un impact pour votre recette dans un tel contexte. Bien que la construction débute par une conceptualisation dans un cahier des jeux d'essai, la saisie même des données sera ensuite tributaire d'outils :

- Des outils de manipulations pour les bases de données (par exemple TOAD).
- Des éditeurs spécialisés pour les fichiers à structure balisée comme le XML, le SGML, le HTML... (par exemple UltraEdit, InContext...).
- Des fichiers textes ou tableurs comme Excel pourront être utiles.
- Parfois des contenus vidéo, sons et images seront nécessaires, par exemple pour tester le téléchargement de contenu multimédia.

Tous ces prérequis seront bien sûr listés dans le cahier des jeux d'essai, mais ils devront surtout être produits. Cela pourra être par une saisie manuelle. Cela pourra être par un procédé de collecte, comme un export de données depuis une base, puis constitution d'un fichier Excel.

L'impact sur le projet sera de définir clairement comment se procurer les données, d'avoir à effectuer éventuellement une analyse qualitative pour en extraire un échantillon viable, et enfin d'en réaliser la saisie pour préparer l'environnement de test.

Toutes ces actions ont nécessairement un coût. Et ce coût sera d'autant plus important si les données en question sont en plus consommées de manière irréversible : ce risque doit donc être impérativement anticipé.

#### **h. Un jeu d'essai qui a du sens**

La présente section n'est pas tant une tâche dans la réalisation d'un jeu d'essai qu'une recommandation générale et un axe de réflexion que nous invitons le lecteur à suivre.

Concevoir un jeu d'essai est la principale contrainte d'un projet de recette. Pourquoi cette affirmation ?

Tout simplement parce que les données manipulées constituent la cause première qui motive l'existence d'un logiciel : elles sont le reflet du métier de son utilisateur, la manifestation d'un travail. Aussi, plus un jeu d'essai est pertinent, plus il a un sens métier, plus il est à même de répondre à la meilleure qualité de test recherchée.

 Prenez l'exemple d'une application qui gèrerait le panier d'un site de vente en ligne d'un opérateur téléphonique. Vous viendrait-il à l'idée de définir un simple téléphone portable ayant un prix de 1 million d'euros ? Techniquement, l'application l'autorise mais fonctionnellement où est le sens à cela ?

Et au-delà même d'une vision métier, c'est même une dimension culturelle que pourra parfois prendre un jeu d'essai.

Nous recommandons au lecteur un point de vigilance tout particulier, notamment en recette fonctionnelle : celui de faire valider le cahier des jeux d'essai par la MOA ou des utilisateurs finaux dans la mesure du possible. Et tout particulièrement, si la recette fonctionnelle doit être exécutée par ces mêmes personnes.

La compréhension fonctionnelle d'un métier au travers des données manipulées n'est pas toujours aisée, souvent subjective ou d'avis contradictoire selon l'utilisateur interrogé. Un jeu d'essai techniquement valide, mais sans véritable sens fonctionnel montrera donc que... vous n'avez pas compris le métier des utilisateurs !

Le risque pour le responsable de la recette se situe alors dans sa crédibilité vis-à-vis de personnes qui peuvent être d'ailleurs des décisionnaires dans le cadre de votre projet (votre client qui sait ?).

Pour une recette technique, et bien qu'en apparence l'intérêt soit moindre, nous ferons pourtant la même recommandation : il est vital de faire passer le message auprès de l'équipe de test de ce que fait le logiciel.

Des données de test fonctionnellement vides de sens amèneront peut-être des incompréhensions voire d'inutiles déclarations d'observations de la part des testeurs. La crédibilité vis-à-vis de la MOE sur les incidents remontés

pourrait alors être impactée elle aussi.

En conséquence, et même si les données du jeu d'essai décrivent une situation fictive, il est impératif que les données aient un sens et une pertinence.

- Exemple de mise en pratique du concept : les tests d'une application de banque en ligne des professionnels nécessitaient de créer des abonnements et comptes d'une Société Anonyme. Le parti pris fut de choisir la thématique de la Bande Dessinée pour définir le Groupe W, son président Largo Winch, chaque abonnement correspondant à l'un des actionnaires du Groupe. La situation décrite était fictive mais compréhensible par chacun.

## 2. Sauvegardes et restaurations

### a. Intérêts

Comme nous l'avons vu dans les sections précédentes, la base de données d'une application peut connaître plusieurs états selon qu'elle est plus ou moins alimentée. Ainsi, si au cours d'une même recette il est nécessaire de tester le logiciel avec une base de données initiale, puis avec un référentiel complet et enfin avec des données métier, le lecteur comprendra qu'il est impératif d'avoir les trois photographies de ladite base, des sauvegardes.

Le premier intérêt est donc de pouvoir gérer des états différents de l'application et surtout de pouvoir "revenir en arrière" à l'aide de restaurations si nécessaire.

Mais le second intérêt réside dans la prise en compte d'un risque : la détérioration et la consommation du jeu de données.

Dès lors qu'une recette s'exécute, la base de données évolue. La validité des données est alors compromise :

- Une anomalie pourra introduire des inepties techniques et fonctionnelles.
- Des exécutions répétées d'un même scénario pourront aussi amener des détériorations techniques moins visibles - comme les index des tables - ce qui peut d'ailleurs impacter les performances (suppressions en masse).

Cet intérêt est encore plus marqué pour les tests des flux ou traitements de masse d'import, export ou batch. Nous avions évoqué la problématique de ces tests dans notre chapitre relatif aux tests unitaires d'ailleurs.

Les sauvegardes constituent alors un référentiel d'environnement indispensable, seule démarche possible pour ré-exécuter un même test lorsque des contraintes techniques fortes imposent une stratégie "one shot".

### b. Les limites et inconvénients

La première limite des sauvegardes réside en sa faisabilité qui n'est pas toujours évidente. Au-delà de l'espace de stockage nécessaire physiquement, une sauvegarde suppose aussi plusieurs qualités :

- D'abord que plusieurs exécutions d'un même traitement avec les mêmes données à deux dates différentes soient pertinentes.
- Ensuite que le périmètre des données sauvegardées reste cohérent : il est possible qu'il y ait plusieurs supports physiques, plusieurs fichiers, tous étant stockés ensemble.
- Enfin, quand bien même le stockage serait pertinent et techniquement cohérent, faut-il encore qu'il le soit d'un point de vue fonctionnel. Nous avions évoqué la problématique de la chrono-dépendance. Or une sauvegarde est une image du passé pour laquelle les dates futures un jour ne le sont plus ensuite. Devra-t-on retraiter les données restaurées pour les adapter ? Devra-t-on mettre en œuvre un traitement ponctuel ou automatique de synchronisation avec les données de production ?

La démarche mérite une étude à elle seule... et peut-être même est-elle un projet dans le projet elle aussi ?

### c. Un service à offrir ?

Dès lors que le responsable de recette a une capacité à créer et stocker des images d'une base de données, ne peut-il pas offrir aux développeurs, recette technique ou fonctionnelle, un service de livraison clé en main d'environnement de données ?

Nous nous éloignons ici du sujet de la recette à proprement parler mais il nous a semblé important d'ouvrir ce sujet.

La mise en place d'un tel service est un vrai chantier de réflexion et nous vous laissons le soin d'en mesurer toute l'importance et les gains qu'il pourra induire dans vos propres projets.

# Les données en environnement dégradé

La présente section concernera davantage les recettes techniques en sortie de développement. La raison à cela est relativement évidente : les environnements de tests métier sont la plupart du temps sans dégradation.

Mais c'est ici une règle très générale que nous supposons et des exceptions sont bien sûr possibles.

Qu'appelle-t-on environnement dégradé ?

Nous avons partiellement évoqué ce sujet dans le chapitre précédent. Il s'agit ici de comprendre qu'un environnement de recette ne dispose pas nécessairement de toutes les données, de tous les traitements non plus, notamment les jobs automatiques de nuit. C'est le cas fréquemment de solution exploitant des gros systèmes (mainframe) pour lesquels l'accès aux données n'est pas possible.

Mais ce peut être aussi au cours d'un programme ou tout simplement de deux projets liés (voire le lotissement du projet lui-même s'il y en a un), une contrainte de planning qui amène l'absence d'un traitement.

Cette incomplétude technique des données, tant en accès qu'en transformation, est ce que nous appelons une dégradation d'environnement.

 Attention, l'absence d'un composant logiciel comme un plugin pour une application web pourra être considérée comme une dégradation d'environnement dès lors que les informations affichées ont un rôle pour l'application. En revanche, l'absence d'un Flash Media Player destiné à diffuser la publicité incluse dans un site pourra être ignorée. Il y a donc des cas pour lesquels la dégradation est ignorée car sans impact.

## 1. Modifier le jeu de données en cours d'exécution

### a. Qu'est-ce qu'une simulation ?

Dès lors qu'un traitement de transformation ou d'import de données est absent, on appellera simulation toute action technique (manuelle ou automatique) visant à contourner ce manque "artificiellement".

Utiliser un outil d'accès à une base de données comme TOAD et exécuter une requête SQL pourront donc définir une simulation.

Ce peut être aussi l'usage d'un outil backoffice avec un compte administrateur, de réaliser un import depuis un fichier Excel en lieu et place d'un traitement périodique...

Mais la caractéristique principale qui définit une simulation est que sans elle, l'équipe de recette reste bloquée et dans l'incapacité de réaliser certains tests aussi longtemps que la simulation n'est pas exécutée.

Les simulations sont donc très importantes dans l'atteinte du taux de couverture d'une recette puisqu'en leurs absences, des cas de test ne pourront être exécutés intégralement.

### b. Comment gérer les simulations ?

Nous considérons qu'une seule personne doit avoir la charge de piloter les simulations : le responsable de la recette qui suit l'équipe des testeurs lors des exécutions.

La raison pour laquelle nous préconisons cette contrainte est facilement compréhensible : deux personnes accédant en écriture à la même base de données sont susceptibles d'entrer en conflit.

L'organisation en place est alors très simple : dès que le besoin d'une simulation se fait sentir, le testeur d'une équipe la demande. Le responsable de la recette exécute alors le traitement ou réalise l'action technique puis fait un retour au testeur qui peut continuer son test.

Dans cette démarche, il est supposé ici que la simulation est ponctuelle, mais il est évident que, dès lors que la donnée modifiée est commune à plusieurs testeurs, une synchronisation devient obligatoire : N testeurs demandent la simulation et sont mis en attente par le responsable de la recette qui orchestre le groupe de simulations en une seule exécution.

Comment formalise-t-on une simulation dans un cahier de recette ? Voici un exemple pour répondre à cette question. Nous supposons ici qu'une application web de banque à distance est dans un environnement dépourvu du batch de traitement des virements.

Step	Action	Résultat attendu	OK / KO
...			
12	Cliquer sur le bouton "Voir la liste des virements" et observer la page web.	La page "Liste des virements" est affichée. Elle comporte trois virements "En cours".	
13	Cliquer sur le bouton "Annuler" du deuxième virement de la liste. Une boîte de dialogue vous demande si vous souhaitez confirmer l'annulation. Cliquer sur OK.	La liste des virements est rafraîchie et un message indique que la demande d'annulation a été prise en compte : le deuxième virement est "En attente d'annulation".	
14	Simulation : demander au responsable de la recette de modifier l'état du deuxième virement manuellement et attendre.	L'état du virement est positionné à "Annulé" par le responsable de recette.	
15	Cliquer sur le bouton "Rafraîchir la page".	La liste des virements est rafraîchie : le deuxième virement est à l'état "Annulé".	
16	Test suivant...		
...			

Le lecteur notera que les simulations doivent être nécessairement anticipées dès la phase d'organisation de la recette puisqu'introduites ensuite dans la rédaction même des cahiers de recette en phase de préparation. Ce sujet est donc sensible.

### c. Les limites des simulations

Une simulation étant une action technique, le plus souvent manuelle en substitution d'un traitement, elle ne garantit pas l'intégrité ni la cohérence fonctionnelle des données. Elle est donc limitée de fait et doit être autant que possible évitée.

Bien évidemment, si la mise en place du traitement manquant est impossible, les simulations sont indispensables. Cependant, l'alternative d'installer le traitement en question doit être envisagée de manière préférentielle.

Il faudra remanier les cahiers de recette d'où la nécessité de marquer les simulations explicitement dans le formalisme des tests par un mot-clé. L'impact est donc relativement faible.

Mais le lecteur devra ne pas perdre de vue qu'une simulation a aussi d'autres limites.

Technique d'abord : trop de simulations sur un même traitement remettront sérieusement en question sa testabilité. Peut-être faudra-t-il alors envisager de sortir l'exigence fonctionnelle du périmètre pour la tester dans un autre environnement ? La stratégie est donc impactée.

Organisationnelle ensuite : trop de simulations sur des exigences fonctionnelles distinctes sollicitent le responsable de la recette pour les exécuter, ce qui revient à faire d'une ressource humaine un "simulateur permanent" au service de l'équipe des testeurs. Cette activité risque alors de devenir chronophage.

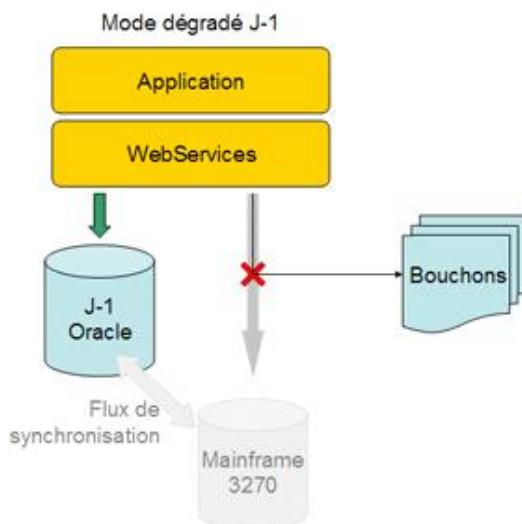
Là aussi, des choix de périmètres devront être faits et des compromis trouvés.

## 2. Les bouchons : des données artificielles

### a. Qu'est-ce qu'un bouchon ?

Nous avons abordé ce sujet dans le chapitre précédent relatif à l'élaboration d'un périmètre de test.

Un bouchon est un groupe de données statiques, généralement défini dans un fichier se substituant à des données réelles lorsque l'accès à ces dernières n'est pas techniquement possible dans l'environnement d'exécution.



Ces données sont retournées par le traitement (dans notre schéma, des webservices) qui détecte l'absence de données via un paramètre, un mode de fonctionnement, puis va chercher les données du bouchon au lieu d'effectuer un accès réel.

Cette technique est notamment répandue dans les webservices, les bouchons étant des fichiers XML, raison pour laquelle nous les évoquons.

Nous pourrons distinguer deux types de bouchons : les simples et les différentiels. Ou plus exactement, nous pouvons distinguer deux algorithmes de substitution.

Le bouchon simple consiste en une substitution permanente et invariablement identique : le fichier statique contient une et une seule entité systématiquement renvoyée quel que soit l'appel fait au service.

Obtenir une réponse différente d'un bouchon simple suppose alors de le modifier, action nécessairement manuelle.

Ces bouchons sont systématiquement utilisés pour les traitements d'écriture, de modification et de suppression, le bouchon contenant alors principalement le code de retour de l'action.

Un bouchon simple pourra être utilisé aussi pour renvoyer des données et simuler des lectures, mais la réponse sera alors toujours identique.

Afin d'améliorer le procédé, nous pouvons alors introduire la notion de bouchons différentiels, c'est-à-dire que le bouchon contiendra plusieurs réponses possibles à plusieurs questions.

Lors d'un appel au service bouchonné, les paramètres d'entrée seront alors utilisés pour filtrer les questions et identifier la réponse associée qui sera renvoyée. Dans cette logique, si la question n'était pas trouvée, une réponse par défaut serait alors retournée.

Un tel bouchon offre donc un intérêt notable puisqu'il permet une relative interactivité quand bien même les données réelles seraient absentes. Mais seules les lectures en base de données sont concernées, les mises à jour ne pouvant pas être mieux bouchonnées que simplement.

### b. Un exemple concret : un système de banque en ligne

Le lecteur comprendra que les bouchons sont utilisés dans des environnements fortement dégradés : c'est un choix technologique qui est fait sciemment faute de mieux.

Typiquement, un système bancaire accédant à un mainframe via des webservices est un cas d'école de la technique des bouchons.

L'hébergement des données et des mainframes (par exemple un 3270 ou un AS400), leurs maintenances... génèrent un coût non négligeable que les entreprises hésitent à débourser pour avoir des environnements de développement et de recettes identiques à la production.

De ce choix financièrement motivé découle une stratégie technologique de contournement, les environnements dégradés et l'emploi des bouchons, avec l'impact que cela représente en matière de test à tous les niveaux.

Un système de banque en ligne comprend généralement deux niveaux dans son organisation : un niveau fédéral (envergure nationale) et un niveau régional (les caisses).

Cette organisation induit que les applications sont d'abord développées au niveau national puis déployées et paramétrées localement (du moins si la DSI est centralisée). Ainsi, la multiplicité des environnements de recette, tant au niveau national que régional, justifie le parti pris des dégradations.

### c. Comment gérer des bouchons lors d'une recette ?

Les bouchons étant des données statiques, ces dernières sont nécessairement des données prérequisées à définir avant d'exécuter la recette.

Par ailleurs, les bouchons sont généralement partie intégrante des sources de l'application : ils ne peuvent donc pas être modifiés dynamiquement en cours de l'exécution et faire l'objet de simulation contrairement à une base de données accédée en temps réel.

La gestion des bouchons dans une recette s'effectue en des points très précis du projet :

- Lors de la collecte des exigences, les fonctions bouchonnées sont identifiées en tant que telles car les bouchons participent à l'élaboration d'un périmètre de tests pertinent.
- Lors de la préparation, la rédaction du cahier des jeux d'essai précisera la nature bouchonnée du support des données. C'est aussi lors de cette étape que les bouchons sont saisis. La dimension technologique (fichiers XML) induira la nécessité d'un outil de saisie adapté mais également une connaissance à la fois technique et fonctionnelle.
- Lors de l'exécution, de deux manières :
  - Parce que savoir qu'une exigence est bouchonnée permet de filtrer les observations qui ne sont pas des anomalies mais des conséquences de la dégradation.

- Lorsque les bouchons doivent être changés en cours de recette, ce qui nécessite la synchronisation de toute l'équipe de test, situation qui peut se produire lorsque des cas de test exclusifs doivent être testés.

La gestion des bouchons a donc un impact organisationnel non négligeable et le lecteur comprend ici pourquoi elle sera spécifique à une recette technique : une recette fonctionnelle est très difficilement envisageable en mode bouchonné.

#### **d. Les limites des tests des applications bouchonnées**

Nous venons d'évoquer dans la section précédente une première limite : l'infaisabilité des recettes fonctionnelles.

Mais une deuxième limite, technique celle-ci, apparaît alors comme une évidence : les mises à jour sont impossibles ce qui revient à dire que les tests se limitent partiellement à la couche logicielle de traitement des données et se focalisent surtout sur la couche présentation.

Une recette en environnement bouchonné est donc une recette technique ciblée sur l'ergonomie avant toute chose, tout du moins sur les parties bouchonnées.

Une dernière limite doit aussi être soulignée, encore que contextuelle : le partage possible des bouchons avec d'autres équipes projet, y compris les développeurs qui auront besoin de vos bouchons pour reproduire des anomalies. Dès lors, le remplacement des bouchons devra être murement préparé.

### **3. Impact sur le bilan des tests**

Nous allons maintenant voir l'impact de la dégradation d'environnement sur le bilan des recettes.

#### **a. Décrire ce qui n'a pas pu être testé**

Le premier impact sera de clairement indiquer dans le bilan des tests ce qui n'a pas pu être testé, d'où la nécessité de clairement identifier les exigences fonctionnelles et techniques faisant l'objet d'un bouchon.

Le bilan de recette indique alors ce qui doit faire l'objet d'une vigilance accrue dans la suite du projet, voire de tests spécifiques dans d'autres environnements, ceux-là étant complets.

Dans le cas limite où seul l'environnement de production serait complet, le processus de déploiement de l'application s'en trouve impacté puisqu'il ne suffira pas d'installer l'application : un protocole de tests devra être rédigé et exécuté permettant de vérifier l'intégration des composants "dé-bouchonnés" dans l'environnement réel.

À ce protocole de tests sera associé un protocole de veille : durant les premières semaines, les premiers mois de fonctionnement de la solution logicielle, des sondages pourront être effectués pour vérifier la bonne tenue du traitement.

Nous avions commencé notre étude par une analyse de risques : celle-ci trouve tout son intérêt dans le cas présent.

#### **b. Émettre des réserves**

Dès lors qu'un chef de projet recette émet un bilan, il s'engage. Il pourra lister ce qui n'a pas été testé, ce qui est bouchonné, ce qui signifie qu'il indique les limites du périmètre des tests qu'il vient de piloter.

Mais le rôle premier d'un bilan de recette est de statuer, d'homologuer l'application, c'est-à-dire qu'une dimension "contractuelle" d'engagement apparaît dans la démarche.

Émettre des réserves est alors indispensable dans un environnement dégradé. Ce signalement pourra consister à écrire "Nous validons l'application sous réserve des vérifications ultérieures à faire sur le périmètre des exigences bouchonnées suivantes : etc.". Même chose pour les simulations, encore que le risque est normalement moindre que pour les bouchons.

La présente section souligne surtout l'importance des réserves lors d'une gestion de tierce recette applicative externe pour le compte d'un client, mais la démarche reste intéressante en interne.

Enfin, comme nous allons le voir ci-après, l'impact de la dégradation d'environnement concerne un problème plus vaste : la cohérence entre les recettes.

### c. Déterminer une stratégie de continuité

A minima, nous avons un environnement de développement, un environnement de recette technique et/ou de recette fonctionnelle (idéalement ils sont distincts) et un environnement de production. Donc 3 ou 4 environnements différents.

Et dans le meilleur des mondes, nous trouverons d'autres environnements dédiés : pré-production, formation... dont celui dédié aux tests de charge.

Cette succession d'environnements ne suppose en aucune manière un même niveau de dégradation : la seule garantie que nous avons est que celui de production est complet.

Ceci signifie qu'à chaque fois qu'une recette est faite dans un environnement dégradé, il y a un impact pour la recette qui suivra : chaque périmètre de l'une dépend de ce qui n'a pas été fait auparavant.

Il y a donc nécessairement une stratégie globale de continuité des tests à déterminer, démarche qui ne pourra être initiée que par la MOA, et qui suppose une certaine maturité dans le processus projet, mais également une dimension multi-projets puisque les recettes applicatives des différents éléments constitutifs du système d'information sont concernées.

Cette stratégie se manifeste dès la phase d'organisation des recettes par l'élaboration des périmètres de test : ils doivent être complémentaires ou supplémentaires, sans tomber dans d'inutiles redondances.

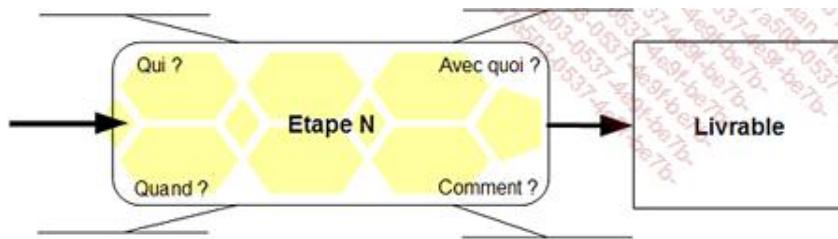
Mais elle se manifeste aussi lors des bilans, étapes qui deviennent alors les charnières des différentes recettes.

# Concepts généraux dans l'organisation du travail

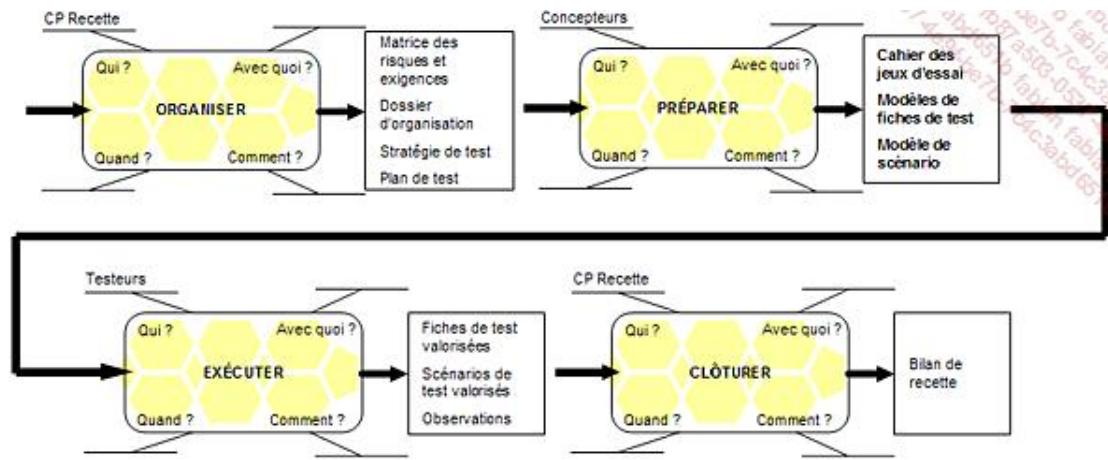
Si nous partons du principe que la gestion des tests peut s'avérer être "un projet dans le projet", alors la démarche projet pourra s'appliquer à cette seule partie. Nous considérerons que le lecteur a ici acquis les bases des concepts de la gestion de projets. Ce qui n'empêche en rien une piqûre de rappel.

## 1. Le diagramme de Crosby Turtle

Comme tout projet, le point de départ de votre analyse pourra être tout simplement un diagramme de type "Crosby Turtle" dont voici le célèbre item de base :



Sur ce principe, nous décomposerons la recette en quatre étapes principales que nous nommerons ORGANISER, PRÉPARER, EXÉCUTER et CLÔTURER ce qui donnerait le schéma suivant (volontairement incomplet) :



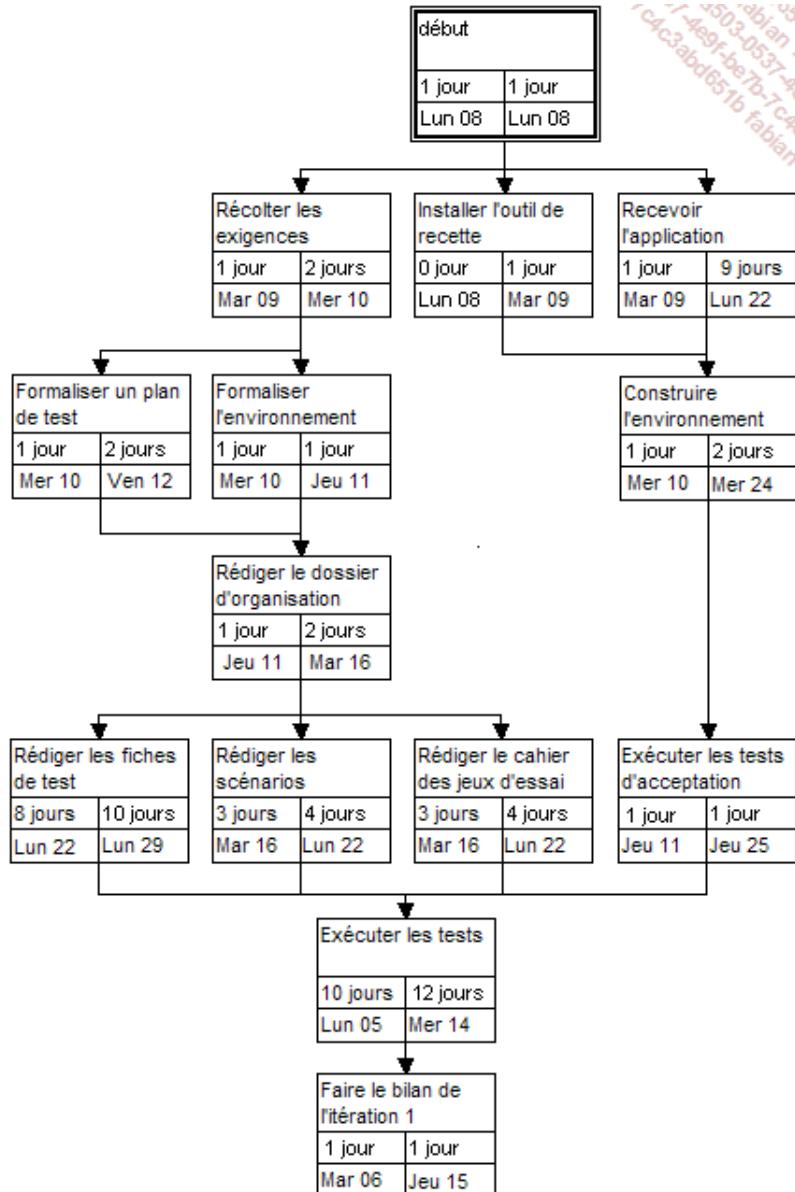
Une cinquième tâche que nous nommerons PILOTER courra tout le long du projet et désignera spécifiquement l'activité du chef de projet de la recette s'il y a lieu.

Nous utiliserons ces mots explicitement pour désigner ces phases tout au long de ce chapitre.

## 2. Les diagrammes de PERT et de GANTT

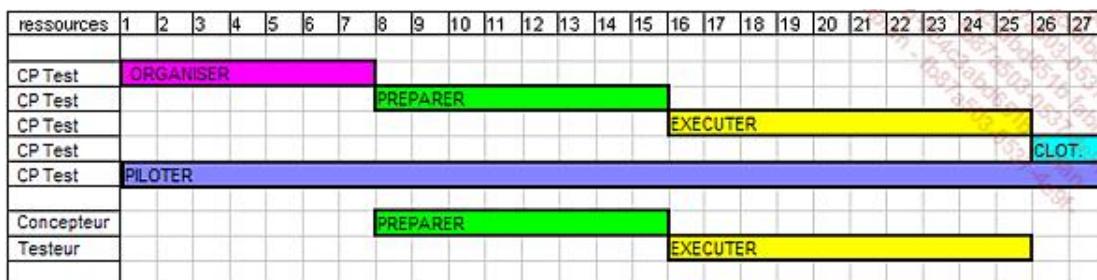
Bien évidemment, et comme pour tout projet, les diagrammes de PERT de précédence et de Gantt pour la planification seront des outils très appréciables pour vos recettes.

Le diagramme de PERT ci-dessous est partiel et a été construit à un bas niveau de décomposition de tâches :



Nous ne décrivons ici que le début d'une recette mais ce diagramme vous donne déjà une idée du niveau de complexité que peut atteindre un projet de test.

Concernant le diagramme de Gantt suivant, sa description s'appuie sur la même décomposition des tâches que nous avons proposée : ORGANISER, PRÉPARER, EXÉCUTER et CLÔTURER.



Nous ne détaillerons pas davantage ces notions ici cependant.

En effet, nous avons préféré être moins dans la théorie et davantage dans la pratique en vous exposant ce qui va

suivre : les différentes typologies de recette. Car de celles-ci découlent des organisations différentes, des livrables différents, alors que la gestion de projets est largement traitée dans de nombreux ouvrages de référence vers lesquels nous préférerons vous renvoyer.

### 3. Les types de recette

Nous distinguerons donc :

- La recette "pompier" ou recette d'urgence, qui en général s'impose à la sortie d'un développement, que l'on soit une MOA ou l'équipe de recette technique, très souvent dans le cadre d'un chantier de release, beaucoup plus rarement dans un chantier de maintenance.
- La recette du patch correctif d'un incident en production, qui nécessairement survient dans un chantier de maintenance.
- La recette technique ou technico-fonctionnelle, spécifique à une équipe experte en test est certainement le projet le plus complet, susceptible d'être mise en œuvre tant dans un chantier de release qu'un chantier de maintenance.
- La recette fonctionnelle "pure" ou d'homologation, spécifique à une MOA dans le cadre d'une VABF (vérification d'aptitude au bon fonctionnement), qui a un rôle différent selon que nous sommes dans un chantier de release ou de maintenance, mais une démarche similaire.

Nous allons maintenant détailler ces quatre typologies d'organisation dans la suite du chapitre.

# La recette "pompier"

## 1. Le contexte

### a. Quand ?

Comme le titre de cette section le laisse entendre, le projet est en feu ce qui signifie que la situation est extrêmement préoccupante.

Généralement, celle-ci s'observe lorsque la ou les recettes n'ont pas été anticipées et que les premiers tests révèlent une application hautement instable avec un taux d'anomalie très élevé.

Notamment, les anomalies bloquantes feront que le périmètre des tests effectivement jouables est si faible que le lancement de la recette doit être annulé au profit d'une autre action : la recette "pompier".

En toute logique, cette situation s'observe principalement dans un chantier de release bien que le chantier de maintenance puisse connaître des situations plus ou moins similaires selon la taille des évolutions mises en œuvre ou le contexte du projet.

### b. Pourquoi ?

Lorsqu'un projet se passe mal, il convient d'être pragmatique : ce n'est pas le moment de chercher les causes de l'incendie, c'est d'abord le moment de l'éteindre.

Nous dénommons cette recette "pompier" pour ce motif.

L'objectif n'est plus de faire une recette technique ou fonctionnelle : il est de compenser une lacune en tests unitaires en dégageant un maximum d'observations afin de quantifier l'écart entre ce qui a été fait et l'objectif à atteindre.

La détermination du volume des observations permet alors d'estimer la charge de travail à fournir pour rectifier le tir au plus vite.

Une recette "pompier" est donc une opération "coup de poing" pour laquelle une seule ressource est nécessaire le plus souvent, éventuellement une petite équipe : il faut être efficace, rapide et peu coûteux.

## 2. Gérer une recette d'urgence

### a. ORGANISER - Définir un périmètre de manière pragmatique

Le contexte d'une recette d'urgence ne permet pas de formaliser un périmètre de test de manière rigoureuse. Bien sûr, vous aurez probablement des cahiers de recette, des plans de test et tout votre arsenal de recette prêts... sauf que l'application ne permet pas réellement leur usage. Vous pouvez aussi oublier vos diagrammes : Crosby Turtle, PERT et Gantt ne sont plus daucune utilité.

Nous vous conseillons donc de ne formaliser ni le périmètre ni les tests et de vous appuyer sur l'application (tout du moins ce qui marche) afin de recenser prioritairement toutes les anomalies bloquantes ne permettant pas l'exécution d'un test. Ajoutez à cela la liste des fonctionnalités trop dégradées également.

D'expérience, cette situation se produit lorsque la gestion des configurations matérielles n'a pas été anticipée par la MOE (notamment pour les applications web et mobile) ou parce que les technologies employées imposent des

contraintes fortes inconnues des développeurs (usage d'un progiciel).

N'hésitez pas à utiliser l'interface graphique s'il y en a une pour hiérarchiser vos tests :

- Chaque fonction du menu principal.
- Puis chaque fonction du menu secondaire.
- Puis changer de profil utilisateur.
- Puis changer de configuration matérielle.

Votre intuition sera votre principal outil : vous identifierez les points faibles de l'équipe de développement et très rapidement vous trouverez des thématiques d'anomalies qui permettront de définir des chantiers de corrections.

## b. PRÉPARER - Est-il utile de formaliser les tests ?

Oui et non.

Vous n'allez pas rédiger de scénario de tests ou de fiche de test - c'est une évidence (pas le temps !). Mais vous pouvez toutefois définir un protocole de test, c'est-à-dire un unique document qui listera les points de contrôles que vous allez systématiquement faire, une check-list commune à toutes les exigences.

Voici un exemple pour une application web :

	Page A	Page B	Page C	...
La page s'affiche-t-elle ?	Oui	Oui	Non	
La validation du formulaire est-elle possible ?	Non	Oui		
Tous les liens sont-ils cliquables ?		Non		
Y a-t-il une anomalie fonctionnelle évidente ?		Non		
Y a-t-il une anomalie technique évidente ?		Oui		

L'idée est de fournir un fil rouge global : vous noterez ici que tous les résultats sont consignés dans un unique tableau.

## c. EXÉCUTER - Collecter les anomalies de manière centralisée

Au cours d'une recette pompier, la collecte des anomalies conduira nécessairement à de gros volumes.

Nous vous conseillons d'utiliser un unique fichier Excel et d'éviter l'usage d'outils de gestion d'observations (tels que Mantis ou le module Defect de Quality Center).

Les fonctions de filtre et de tri d'Excel permettront aux différents acteurs de la recette pompier de travailler avec un unique livrable qui permettra de prioriser les anomalies à traiter. Il suffit de définir une colonne "Priorité" et de la valoriser pour chaque observation ainsi qu'une colonne "Qualification" pour statuer si l'observation est une anomalie ou pas.

N'hésitez pas à vous inspirer des méthodes Agile et des valeurs proposées dans le Manifeste Agile : vous devez privilégier une démarche souple favorisant la communication avec une équipe en pleine crise !

Par ailleurs, la centralisation des anomalies dans un seul fichier Excel permet de construire des mini-chantiers de correction, de mesurer un avancement...

 Astuce ! Définissez une colonne Référence dans laquelle vous numérotez vos observations 1, 2, 3... dans l'ordre de création. Lorsque vous considérez que deux observations doivent être regroupées autour de la même anomalie (ce qui ne signifie pas qu'il y a redondance), numérotez-les par un préfixe commun : 20a, 20b, 20c... Vous pourrez ainsi regrouper des tâches.

#### **d. EXÉCUTER - L'équipe de développement est un partenaire**

Une crise est toujours un moment difficile dans un projet et sa gestion n'est pas aisée. La recette pompier est une réponse à ce contexte dont la cause est la plupart du temps à rechercher du côté de la MOE (ce qui ne veut pas dire que la MOA n'a pas sa part de responsabilité non plus).

Est-ce un manque de maturité de l'équipe ? Un projet mal organisé ? Des dysfonctionnements dans les processus et une mauvaise communication ? Des configurations matérielles non anticipées ? Un progiciel trop contraignant ou trop complexe ?

Probablement tout cela à la fois.

Une seule solution pour "sauver le projet" : être solidaire. Une équipe de test capable de mettre en œuvre une recette pompier permettra de dégager rapidement et de manière aussi factuelle que simple l'écart entre ce qui a été fait et ce qu'il faudrait.

Bien sûr les tests menés seront peut-être inappropriés parfois et leurs résultats écartés... mais le contexte du projet justifie largement une démarche radicale en laissant de côté toute considération sur la non-pertinence d'un test.

Votre principale difficulté résidera alors dans la communication avec la MOE et son adhésion à la démarche.

#### **e. CLÔTURER - Mettre en place un protocole d'acceptation pour les prochaines livraisons**

Le contexte du projet amène la méfiance mais aussi provoque une perte de confiance des développeurs en eux-mêmes.

Et le risque que l'application soit à nouveau livrée avec un niveau de qualité trop insuffisant persiste car l'équipe doit corriger trop de choses trop vite.

Bien sûr, le CP MOE devrait mieux organiser les tests unitaires de son côté (nous le renvoyons au chapitre Organiser les tests unitaires pour cela) mais du côté de l'équipe recette, il vaudra mieux se prémunir en définissant un protocole d'acceptation pour les prochaines livraisons avant de lancer la phase d'exécution des tests.

Ce protocole consistera en une check-list d'une part, mais également demandera de définir un taux d'anomalie acceptable, un seuil. Par exemple, pour une exigence de criticité faible qui serait bloquée par une anomalie, le protocole pourra accepter de démarrer la recette.

Pour le formalisme du protocole d'acceptation, prenez le protocole de test de la recette d'urgence qui peut servir de base à sa rédaction soit dans son intégralité, soit en se limitant aux exigences de criticité forte.

 N'hésitez pas à communiquer ce protocole d'acceptation à la MOE ! Mieux : essayez de voir s'il existe des tests que la MOE n'est pas en mesure de réaliser pour des raisons d'environnement !

# La recette d'un patch

## 1. Le contexte

### a. Quand ?

Le patch est le correctif d'urgence demandé à une MOE suite à un incident détecté en production par le métier (reportez-vous au chapitre suivant : nous y décrivons le processus de collecte des incidents de production).

Il ne s'agit donc pas d'une recette pompier mais d'une réponse très réactive à fournir rapidement nécessairement dans un chantier de maintenance puisque l'application est déjà en production.

### b. Quelles contraintes ?

La principale contrainte à gérer est la synchronisation du patch correctif avec les versions de l'application situées dans d'autres environnements, notamment ceux des recettes métier et technique.

Pour peu qu'il y ait une recette en cours, vous ne pourrez d'ailleurs pas fournir le patch immédiatement et la correction sera faite en développement puis livrée dans un autre environnement de test, disponible dans l'instant :

- Celui de recette technique si celui de recette métier n'est pas exploitable.
- Réciproquement, celui de recette métier si celui de recette technique ne l'est pas.
- Et parfois un autre environnement (comme la pré-production) si aucun des environnements de recette ne permet le test du patch.

Le caractère urgent du patch fait qu'il n'est pas possible d'attendre qu'un autre processus soit terminé, quel que soit celui-ci, sauf s'il s'agit d'un autre patch avec lequel une synchronisation est nécessaire en raison d'une dépendance.

Par ailleurs, dès lors qu'un patch n'est pas déployé sur les applications situées en recette, cela signifie que ces versions en cours de test ne seront de toute façon, jamais déployées en production telles quelles : il y aura dans la plupart des cas une régression (à de rares exceptions près).

Ceci signifie qu'un patch a une répercussion organisationnelle systématique ou presque : il faut le répercuter sur tous les environnements.

Et les outils conventionnels de gestion de projet ? A-t-on besoin de diagramme de Gantt ou de PERT pour la gestion d'un patch ?

Soyons réalistes : la théorie voudrait que tout projet soit formellement cadré, la pratique nous ordonne d'aller à l'essentiel. Donc si votre organisation en est à un stade où les patchs sont planifiables... c'est probablement que vous avez à gérer un problème hautement plus complexe puisque l'exceptionnel et l'urgence sont devenus votre quotidien !

## 2. Gérer la recette d'un patch

### a. ORGANISER - Définir un périmètre

Si un patch est nécessaire, c'est que nous travaillons sur une application déjà en production. De ce fait, le chantier de release est achevé et un chantier de maintenance est donc en cours (à moins que l'application ne soit plus

maintenue).

Deux situations sont alors possibles :

- Soit il existe un référentiel de tests parce que la recette technique et/ou fonctionnelle a été correctement menée lors du chantier de release.
- Soit il n'existe aucun référentiel de tests ou bien il est trop obsolète pour être exploitable.

Le périmètre de la recette d'un patch consiste alors à vérifier le correctif puis pratiquer une batterie de tests de non-régression pour s'assurer que la correction ne provoque pas d'effets secondaires indésirables.

Dès lors qu'une analyse de risques a été menée et qu'un périmètre de tests a pu être hiérarchisé par elle sur la base des criticités, une démarche formelle devient possible.

Nous pourrions par exemple définir une échelle de sécurité et associer à chaque niveau de celle-ci, un périmètre de tests de non-régression :

Patch de sécurité faible	>>	Tests de priorité Must
Patch de sécurité moyenne	>>	Test de priorité Must+Could
Patch de sécurité forte	>>	Test de priorité Must+Could+Should

La matrice de criticité devient alors un outil décisionnel pour choisir un périmètre de tests de non-régression de manière non arbitraire.

Nous abordons également ici un aspect "niveau de service" qui pourrait d'ailleurs être défini dans une convention de services (mais ce sujet est hors de propos ici).

Et que faire lorsque l'on ne dispose pas d'un référentiel de tests hiérarchisé ? Il est très difficile de répondre à cette question.

Le périmètre des tests de non-régression est logiquement tributaire du patch lui-même, c'est-à-dire de la cause première de l'anomalie.

Supposons par exemple que l'anomalie soit un champ mal dimensionné dans une table : le patch consiste alors à redimensionner le champ et rectifier la donnée. Comment déduire un périmètre de test de non-régression sinon en listant toutes les fonctions qui y accèdent tant en écriture qu'en lecture ?

La démarche demande alors une véritable analyse technique, une cartographie applicative. Peut-être même vous faudra-t-il songer à mettre en œuvre un projet de rétro-documentation ?

Si tel est le cas, a minima nous préconisons de faire l'analyse des exigences et d'estimer la fréquence d'utilisation de ces dernières (sans faire d'analyse de risque) afin de construire une matrice de criticité. Celle-ci ne sera pas le reflet précis mais une approximation suffisante pour dégager une priorisation et donc des périmètres de tests de non-régression.

## b. PRÉPARER - Est-il utile de formaliser les tests ?

Il est indispensable de formaliser les tests d'un patch, au moins les scénarios permettant de vérifier la correction.

Idéalement, le périmètre des tests de non-régression sera formalisé lui aussi. En cas contraire, la matrice des exigences obtenue par rétro-documentation sera déjà une base de travail : à défaut de rédiger la totalité des tests, vous avez au moins un plan de test exploitable (un sommaire).

Quant à la formalisation du test du correctif, soit elle consistera à modifier ou créer un test nouveau dans le référentiel de tests existant, soit à initialiser ce référentiel de tests malheureusement trop vide.

Nous préconisons une démarche pragmatique pour une raison simple : si un patch a été nécessaire en production, c'est qu'un point faible de l'application a été détecté. Le risque d'une régression est accru "tout autour" du correctif pratiqué.

Conserver les tests des patchs, c'est déjà mettre en œuvre une politique de vigilance : vous tracez les points faibles.

#### **c. EXÉCUTER - Doit-on formaliser les anomalies d'un patch ?**

Dès lors qu'une anomalie est détectée sur un patch, l'urgence s'accroît puisque la correction de cette nouvelle anomalie retarde d'autant la livraison en production.

En conséquence, il est évident que les anomalies détectées au cours d'une telle recette font l'objet elles-mêmes d'un patch de l'environnement de recette où celle-ci est détectée.

Comment traite-t-on l'ouverture des anomalies d'un patch ?

S'il existe un outil de collecte des observations pour les recettes techniques et fonctionnelles, tout comme l'anomalie qui était initialement à l'origine du patch, elles seront ajoutées pour garder trace du défaut. Mais en aucun cas cet outil ne devra être utilisé dans le processus de recette du patch.

Il n'y a donc aucun formalisme particulier pour saisir les anomalies durant la phase d'exécution des tests : c'est lors de la clôture que ceci doit être fait comme nous allons le voir immédiatement.

#### **d. CLÔTURER - Comment gérer les anomalies d'un patch ?**

Le bilan de clôture de la recette d'un patch est un document qui devra consigner directement le refus ou l'acceptation du patch, auquel cas l'ensemble des anomalies détectées au cours de la recette pourra y être consigné de manière synthétique : les anomalies seront le motif du refus.

Notez en effet que la présence d'anomalies peut être acceptée dès lors que le risque encouru est faible : une anomalie ayant pour impact une erreur financière, si elle est corrigée par le patch qui présente alors une simple anomalie cosmétique, pourra être acceptée (alors que la même anomalie en recette technique aurait conclu à un refus peut-être).

Nous préconisons donc d'alléger le formalisme des patchs autant que possible, idéalement avec un seul document, tant pour rédiger les tests à exécuter que pour consigner le résultat final : la validation ou le refus du correctif.

#### **e. CLÔTURER - Gérer une cartographie et les livraisons**

Comme nous l'évoquions précédemment, le patch nécessite une synchronisation de déploiement sur les différents environnements de test conduisant de celui de développement à celui de production.

Plus il y a d'environnement, plus le risque de régression sur l'un d'eux est fort. En toute logique, il est donc nécessaire de savoir quelle version d'application est installée sur quel environnement.

Il est donc très important de cadrer deux processus :

- Le processus de livraison d'une application.

- Le processus de déploiement d'une application sur un environnement donné.

Car c'est au moment de ces deux actions que vous pourrez tracer et maintenir un document permettant de cartographier vos environnements de test.

Si à la clôture de la recette d'un patch, vous constatez qu'une telle cartographie est inexiste, n'hésitez pas à remonter l'alerte. Il est vraiment primordial qu'une action de fond soit menée et le patch est le meilleur moment pour une prise de conscience.

Nous préconisons d'avoir au moins un fichier Excel matérialisant la liste des applications installées par environnement précisant la version exacte du logiciel ainsi que sa date de déploiement.

Idéalement, si une application fait l'objet d'une recette dans un environnement donné, il serait aussi souhaitable que la période d'exécution de ladite recette soit mentionnée : une recette en cours doit geler toute livraison durant cette période.

La fiche de livraison délivrée par le CP MOE vous permettra également de lister les évolutions et correctifs contenus dans chaque nouvelle version. L'absence de ce document pourra s'avérer problématique, d'autant plus si le nombre de livraisons est important (en méthode Agile : toutes les semaines ou toutes les deux semaines !).

Cette fiche est unique et elle doit fournir à minima les items à déployer sur un environnement donné, a maxima elle contient une procédure de déploiement détaillée.

Elle mentionne aussi ce qui motive la livraison, la liste des corrections, la liste des évolutions ainsi que les dépendances des livraisons lorsqu'il est nécessaire de livrer plusieurs applications de manière groupée.

Nous ne fournirons pas de modèle de document pour gérer ces processus puisqu'ils participent très indirectement aux processus de recette, mais nous suggérons ici aux chefs de projet MOA, MOE et Recette, d'avoir une réflexion approfondie sur le sujet.

# La recette technique

## 1. Le contexte

### a. Quand ?

Toujours. Que le chantier soit de release ou de maintenance.

Théoriquement, une recette technique devrait être systématiquement faite avant qu'une équipe métier ne procède à une recette fonctionnelle.

La pratique est toute autre et il est fréquent que la recette fonctionnelle soit cumulée à la recette technique, tout simplement parce que la MOA décide d'externaliser ses recettes auprès d'un prestataire externe expert en tests logiciels.

Ainsi, l'équipe de recette technique prend en charge les deux aspects, situation à limiter toutefois, car il est impératif que la MOA effectue des tests de validation fonctionnelle elle-même.

En revanche, lorsqu'il n'y a pas de structure organisationnelle experte dans les tests technico-fonctionnels, la MOA sera contrainte à gérer cette phase alors qu'elle devrait se limiter à une recette purement fonctionnelle.

L'effort de test peut donc s'avérer insuffisant selon le contexte du projet, plaçant ainsi ses acteurs au pied du mur : la recette "pompier" que nous évoquions au début du chapitre s'imposera alors d'elle-même... malheureusement.

### b. Pourquoi ?

Assimiler recette technique et recette fonctionnelle serait une grave erreur.

Car autant une recette fonctionnelle ne revêt pas d'aspects techniques (il s'agit de vérifier que les règles métier sont correctement implémentées et rien d'autre) autant une recette technique devra prendre en compte les technologies et l'architecture employées dans la solution :

- Tests hors domaine ou à des valeurs clés (le dépassement de capacité par exemple, l'usage des caractères spéciaux...). Il s'agit le plus souvent de tests de bas niveau que l'on regroupe dans des fiches de test.
- Tests de robustesse spécifiquement techniques. Par exemple pour les applications web, des injections SQL pour tenter de forcer une authentification, des injections de HTML aussi, des tentatives frauduleuses d'usage de la barre d'adresse, la désactivation du JavaScript, le test du timeout session, les authentifications multiples...
- Tests de compatibilité des configurations matérielles - le plus gros morceau de la recette technique.
- Tests d'installation logicielle, notamment pour les applications embarquées des technologies mobiles, et à plus forte raison si le logiciel est destiné à être téléchargé par les internautes.
- Etc.

Comme vous le constatez, ces tests nécessitent une expertise technique qu'une MOA n'a pas nécessairement ainsi que des besoins en plateformes et en outils qu'elle n'aura pas - souvent pour des raisons budgétaires.

### c. Vers une professionnalisation

Comme nous l'évoquions en introduction du présent ouvrage, l'organisation de recettes technico-fonctionnelles s'est professionnalisée depuis ces dix dernières années, notamment avec l'apparition sur le marché d'outils

propres à gérer les tests.

Simultanément, les technologies de développement des applications ont atteint un niveau de technicité très élevé :

- Il y a 15 ans, les applications étaient des clients lourds installés un à un sur chaque PC. La gestion des configurations matérielles nécessitait peu de tests.
- Avec l'apparition des technologies du Web et la multiplication des navigateurs, le nombre de configurations matérielles a fortement augmenté et reste en constante évolution... de même que le piratage qui amène des besoins nouveaux en matière de test (au-delà du simple fait que l'architecture web demande une approche différente d'un client lourd).
- Et aujourd'hui ? Ce sont les applications sur terminaux mobiles qui surgissent, amenant le nombre de configurations matérielles d'une dizaine pour le Web à une trentaine en pleine croissance commerciale, pour les technologies mobiles : nouveaux terminaux commercialisés tous les trois mois, nouveaux systèmes d'exploitation, nouveaux partenariats commerciaux aussi (par exemple celui de Microsoft avec Nokia qui amènera à terme au remplacement de l'OS Symbian par Windows Phone 7). Une véritable explosion.

Ainsi, le processus de vérification qui consistait autrefois à simplement imprimer un document Word et cocher des cases au crayon - chose relativement aisée pour un utilisateur final du métier dans une recette fonctionnelle - est devenu beaucoup plus complexe.

Il suffisait d'avoir une personne capable de rédiger correctement un cahier de recette là où aujourd'hui vous devez avoir un ingénieur capable d'utiliser Quality Center par exemple.

Des outils d'automatisation sont aussi apparus et qui nécessitent de rédiger des scripts (par exemple, pour Quick Test Pro, vous devez connaître le Visual Basic !) que seuls les ingénieurs en informatique peuvent rédiger.

Cette professionnalisation du test devient donc jour après jour une expertise à part entière qu'une équipe d'utilisateurs métier ne peut pas avoir au sein d'une MOA.

## 2. Gérer la recette technique d'une release

### a. ORGANISER - L'étape la plus critique

Cette étape du projet permet de produire un ensemble de livrables qui trouveront un usage bien après le chantier de release, tout au long du chantier de maintenance.

Cette étape est donc la plus critique car toute lacune entraînera une perte de productivité sur le long terme irrémédiablement.

Pour commencer, le chef de projet de la recette technique sera face à deux situations possibles :

- Soit il aura été impliqué très en amont du projet et aura pu conduire une analyse des exigences - voire une analyse des risques produit.
- Soit il intervient sur le projet trop tardivement et devra donc combler un manque, au moins l'analyse des exigences qu'il devra conduire. Cette situation est généralement celle qui découle d'une externalisation de la recette technique par la MOA auprès d'un prestataire.

La qualité des éléments entrants détermine donc la capacité du chef de projet de la recette technique à produire :

- La matrice des risques produit.
- La matrice des exigences produit (fonctionnelle et technique).

- Le rapprochement des risques et exigences permettant de déduire les priorités et périmètres des tests.

Nous vous invitons à vous appuyer sur les chapitres Définir un périmètre de tests : la stratégie et Gérer les données : la principale contrainte pour réaliser cette tâche et ainsi produire votre dossier d'organisation et la stratégie de recette à mettre en œuvre.

Durant cette étape, vous devrez aborder notamment :

- Le périmètre fonctionnel des tests.
- Le périmètre technique des tests.
- Le niveau de dégradation des exigences.
- Les configurations matérielles et logiques à tester et le périmètre des tests sur chacune d'elles.
- L'ensemble des besoins techniques et logistiques (PC, terminaux mobiles, outils de virtualisation, émulateurs, simulateurs, imprimante, badges d'accès, login et mots de passe, dossier de partage...).
- La gestion des sauvegardes et restaurations des données de test et les phases de l'exécution relativement à l'état de la base.
- La répartition des tests à rédiger et exécuter entre les membres d'une équipe, sachant que :
  - Vous aurez peut-être besoin de ressource pratiquant une langue spécifique tout au long de la recette.
  - Vous ne pourrez pas toujours paralléliser les tests entre deux testeurs.

La première difficulté réside dans la confusion possible entre exigence technique et exigence fonctionnelle, notamment lorsque la recette technique est également fonctionnelle. Nous vous conseillons immédiatement d'adopter le vocabulaire suivant :

- Des exigences techniques sont couvertes par des fiches de test qui regroupent des tests techniques à dérouler par un expert en recette technique.
- Une exigence fonctionnelle est une fonction métier couverte par un scénario de test qui simule une séquence d'actions telles qu'elle serait faite par un utilisateur final.

 Autant que possible, vous ferez en sorte que les tests techniques ne soient pas tributaires d'un jeu d'essai - seulement de données prérequises - et également qu'ils ne fassent pas l'objet de simulations. Vos testeurs auront besoin probablement d'un login et d'un mot de passe, mais de rien d'autre pour exécuter une fiche de test, les données à saisir étant libres à leur convenance. A contrario, le scénario de recette fonctionnelle demandera un jeu d'essai préalablement formalisé et pourra faire l'objet de simulation puisque le testeur se comporte comme un utilisateur final pour un usage réel de l'application.

La deuxième difficulté réside dans la répartition des périmètres de tests des configurations matérielles entre les membres de l'équipe.

Si le périmètre de tests est identique entre les configurations matérielles, vous pourrez organiser votre recette en affectant un ensemble équivalent de configurations matérielles à chaque testeur ce qui revient à aménager pour chacun un poste de travail.

Si au contraire les périmètres sont hétérogènes, alors il vous faudra trouver un équilibre ou bien partir du principe qu'un testeur quittera l'équipe plus tôt qu'un autre. C'est donc la montée en charge puis la baisse qui sont impactées par votre choix d'organisation - ainsi que l'aménagement des postes éventuellement.



Une mention spéciale doit être faite pour les applications sur terminaux mobiles dont le nombre est très important.

En effet, le fait d'avoir un grand nombre de terminaux provoque une brusque montée en charge, la phase de préparation demandant moins de ressources (comparativement à la recette d'une application web). Ainsi, les recettes techniques des applications mobiles présentent un risque accru de dérive en phase d'exécution.

La troisième difficulté organisationnelle concernera les applications multilingues : selon les langues prévues dans l'application, vous devrez avoir une ressource pratiquant ces langues aussi bien pendant la phase de rédaction que pendant la phase d'exécution.

Le problème est ici croisé avec celui des configurations matérielles car dès lors que vous n'avez qu'une seule personne qui pratique la langue à tester, vous ne pourrez pas paralléliser les tests dans celle-ci : la répartition de la charge de travail est alors bousculée.

Ceci signifie que votre testeur devrait théoriquement vérifier toutes les configurations matérielles de toutes les exigences fonctionnelles dans la langue en question quand les autres se partageraient les configurations matérielles en langue française. Or, cette organisation n'est pas toujours appropriée.

Nous préconisons ici une autre stratégie en considérant que la langue définit à elle seule une "configuration matérielle logique" supplémentaire pour laquelle vous choisissez une unique configuration physique de référence. Car il n'est généralement pas pertinent de croiser langue et configuration matérielle pour les démultiplier (notamment pour les recettes des applications web).

Puis, si l'application offre cette possibilité, prévoyez quelques scénarios de navigation large avec bascule de langue pour vérifier que le changement est correctement géré.

Ce problème ne concerne généralement pas l'anglais qui est couramment pratiqué par les ingénieurs à un niveau suffisant pour une recette. Ce seront davantage les langues allemande, espagnole, italienne... qui demanderont ces spécificités.



Là aussi les applications sur terminaux mobiles demandent une mention spéciale en raison de la langue du système d'exploitation qui induit une différence de clavier QWERTY/AZERTY qui a un impact sur le périmètre des configurations matérielles. Dans cette typologie de projet, vous n'aurez peut-être pas d'autre choix que de tester 100 % des fonctionnalités, pour tous les terminaux, dans toutes les langues.

La quatrième difficulté enfin concernera la prise en compte du niveau de dégradation de l'environnement de recette (s'il y a lieu) ainsi que des éventuelles simulations à réaliser au cours de l'exécution, et ce, afin d'établir le périmètre des tests à réaliser ultérieurement dans un autre environnement.

Nous avons proposé dans le chapitre Définir un périmètre de tests : la stratégie un formalisme pour répondre à cette difficulté : nous ne pouvons que vous conseiller de le mettre en œuvre.

Bien entendu, tout projet ne connaîtra pas nécessairement de dégradation... mais qui peut le plus peut le moins.

D'une manière plus générale, et comme nous sommes dans un chantier de release, les diagrammes de Crosby Turtle, de PERT et de Gantt sont ici des outils très utiles pour formaliser votre organisation et communiquer avec des tiers à son sujet.

Tout particulièrement, il vous faudra détailler la description de l'environnement de recette technique pour la première fois, description qui perdurera ensuite pour le chantier de maintenance. Même chose pour les outils et livrables qui sont tous hautement capitalisables pour les recettes futures.

Une démarche projet rigoureuse est donc la bienvenue.

## b. PRÉPARER - Anticiper la capitalisation

Le mettre mot d'une recette technique dans un chantier de release est capitalisation :

- Le plan de test recense les fiches de test et scénarios à rédiger.
- Les fiches de test sont des modèles réutilisables pour des tests de non-régression suite à la modification des caractéristiques techniques d'une exigence fonctionnelle.
- Les scénarios fonctionnels peuvent être utilisés pour des homologations métier ultérieures puis lors des chantiers de maintenance, tout comme les fiches de test.
- Le cahier des jeux d'essai et l'environnement des données prérequises, correctement sauvegardés, décrivent ce qui doit exister avant la recette technique...

Mais d'un point de vue pratique, comment répartir la charge de travail entre les membres d'une équipe de concepteurs de test ?

Pour commencer, nous ne partons pas de rien mais d'un plan de test extrait d'une stratégie : nous avons donc un sommaire, une liste des items à rédiger, dans lequel nous pouvons d'ailleurs gérer l'avancement de la rédaction.

Les scénarios et fiches de test peuvent être rédigés dans un fichier Excel - c'est le format que nous préconisons plutôt que Word - ou bien saisis dans un outil spécifique tel que Quality Center, Salomé, Testlink...

Dans tous les cas de figure, vous avez donc défini un espace de travail collaboratif dans lequel vous demandez à chaque membre de l'équipe de rédiger un lot de scénarios et de fiches de test.

Une préparation se décompose donc en tâches élémentaires comme suit :

- Rédiger la fiche de test 1 + mettre à jour le plan de test.
- ...
- Rédiger la fiche de test N + mettre à jour le plan de test.
- Rédiger le scénario de test 1 + mettre à jour le plan de test.
- ...
- Rédiger le scénario de test M + mettre à jour le plan de test.
- Rédiger le cahier des jeux d'essai.
- Saisir les données prérequises dans la solution (base de données, bouchons, fichiers...).

Vous rencontrerez trois difficultés principales à ce stade.

La première concerne la montée en charge car si l'organisation pouvait être réalisée par le seul CP Recette, la préparation demandera sûrement plus de ressources.

Généralement, cette montée en charge est proportionnelle à la taille de l'application. Si cette dernière comporte une interface graphique, vous pourrez la dimensionner en nombre d'écrans par exemple.

La seconde difficulté concernera alors la spécificité des applications multilingues : selon votre besoin linguistique, vous aurez comme contrainte d'avoir des cahiers de recette à traduire partiellement.

Nous recommandons de choisir une langue unique pour les livrables de recette (le français ou l'anglais pour les projets internationaux) et de ne traduire dans ces documents que les parties le justifiant :

- Les libellés inclus dans l'interface graphique et qui sont recopiés dans les cahiers de recette.

- Les données alphabétiques présentes à la fois dans la base de données et dans le cahier des jeux d'essai.

*Exemple : vous partez du cahier de recette en français...*

Nº	Action	Résultat attendu	OK / KO
1	Lancer l'application	La page d'accueil s'affiche.	
2	Saisir le login [var_login] et le mot de passe [var_mdp] puis cliquer sur bouton "OK".	La page d'accueil est rafraîchie et un message de bienvenue indique qui est authentifié.	
...	...	...	
10	Cliquer sur le bouton "Interdire"	La page est rafraîchie et un message indique "Informations écrites"	
11	Etc.		

*... pour obtenir un cahier de recette pour une application en espagnol :*

Nº	Action	Résultat attendu	OK / KO
1	Lancer l'application	La page d'accueil s'affiche	
2	Saisir le login [var_login] et le mot de passe [var_mdp] puis cliquer sur bouton "OK".	La page d'accueil est rafraîchie et un message de bienvenue indique qui est authentifié.	
3	Cliquer sur le drapeau espagnol dans le bandeau supérieur pour faire basculer l'application en langue espagnole.	La page d'accueil est rafraîchie : tous les libellés sont en espagnol.	
...	...	...	
10	Cliquer sur le bouton "Prohibir"	La page est rafraîchie et un message indique "Informaciones escritas"	
11	Etc.		

 Notez que vous devrez clairement valider les traductions. Dans notre exemple, le libellé "OK" est considéré comme reconnu en espagnol.

Dans notre exemple, vous notez immédiatement que le scénario n'est pas stricto sensu la traduction d'un document en français, mais une adaptation contenant des bribes d'une autre langue et des actions nouvelles.

Pour commencer, il vous faudra ajouter l'action indiquant le changement de langue au début du cahier de recette. À moins que cette langue ne soit automatiquement positionnée par le choix du login et du mot de passe, la langue étant un paramètre du compte utilisé ? C'est alors la valorisation des paramètres qui déterminera la différence entre le scénario français et l'espagnol dans notre exemple.

Ensuite, les données dynamiques lues en base, qui seront affichées et présentes dans le cahier de recette, devront être traduites aussi... ce qui suppose que le cahier des jeux d'essai est lui aussi adapté.

 Attention aux traducteurs en ligne gratuits ou payants ! Ils sont encore loin de remplacer un humain dans la compréhension des langues, d'autant plus que certains aspects culturels peuvent être mis en évidence par un collaborateur et qui pourront s'avérer très utiles dans vos tests. Ne comptez donc qu'à moitié sur ces outils.

Nous terminerons donc par la troisième difficulté qui consiste dans la synchronisation du cahier des jeux d'essai avec les scénarios fonctionnels, c'est-à-dire la définition des paramètres et des données dynamiquement affichés par l'application, susceptibles d'apparaître dans les scénarios - plus rarement dans les fiches de test.

Nous venons déjà d'effleurer cette problématique avec le multilinguisme mais il est général à la phase de préparation.

Tout d'abord, la partie technique de la recette s'appuiera sur des fiches de test dont l'exécution nécessitera à minima un identifiant et un mot de passe la plupart du temps. Les fiches de test contenant des tests purement techniques, le testeur n'étant pas en train de se substituer à un utilisateur final de l'application, la valorisation du jeu d'essai devra alors avoir le moins d'impact possible sur leur rédaction. Rendre les fiches de test indépendantes du jeu d'essai vous permettra de traiter l'exécution des tests techniques sans trop de prérequis.

En revanche, la partie fonctionnelle de la recette, composée de scénarios fonctionnels, doit être comprise selon un angle identique à celui d'un utilisateur final : dans un tel cas, les informations saisies doivent donc être le reflet exact de la réalité et sont d'une importance vitale.

Ceci signifie que le cahier des jeux d'essai et les scénarios sont rédigés conjointement, le premier étant unique et rédigé de manière collaborative alors que les scénarios sont multiples, rédigés par des concepteurs multiples aussi, mais ont un auteur unique à chaque fois.

Il y a donc une vraie démarche à trouver. Nous avons conseillé dans le chapitre précédent sur la gestion des données de définir un thème (fictif ou non) qui détermine une cohérence métier globale.

Ce conseil est de mise ici et nous préconisons de commencer par rédiger le cahier des jeux d'essai avant même de rédiger les scénarios. Après tout, que serait une pièce de théâtre sans son décor ?

## c. EXÉCUTER - S'appuyer sur un plan de test réutilisable

L'exécution d'une recette technique dans un chantier de release se décompose invariablement de la même manière en une succession d'itérations exécutées selon le modus operandi suivant :

- 1) Vérification de l'environnement de test (matériels, logiciels et données saisies) pour l'itération N=1 uniquement.
- 2) Réception de l'application en provenance de la MOE, éventuellement test d'installation pour l'itération N=1 uniquement.
- 3) Tests d'acceptation de l'application pour valider que la phase d'exécution de l'itération N peut se poursuivre avec l'application reçue.
- 4) Génération des supports par :
  - Instanciation des scénarios et fiches de test dans un référentiel représentant la campagne de test N du jj/mm/aaaa.
  - Valorisation des scénarios et fiches de test instanciés sur la base du cahier des jeux d'essai.
  - Affectation des scénarios et fiches de test instanciés à chacun des testeurs.
- 5) Exécution par chaque testeur de ses cahiers de recette, en commençant par les fiches de test (recette technique) puis les scénarios fonctionnels (recette fonctionnelle). Si besoin exécution par le CP Recette des simulations à la demande des testeurs.
- 6) Saisie quotidienne des scénarios et fiches déroulées (Bloqué / OK / KO / Non applicable) dans un plan de test permettant de mesurer l'avancement et de déterminer le périmètre de ce qui n'est pas testable (anomalies

bloquantes), jusqu'à épuisement du périmètre de l'itération N.

7) Remontée des observations, tri et qualification par le CP Recette.

8) Si le périmètre bloqué est non vide, reprendre le processus après correction et livraison de l'application à l'étape 3) puis l'étape 4) en définissant une nouvelle itération dont le périmètre sera défini comme suit :

- Vérification des corrections livrées.
- Exécution des scénarios et fiches bloqués dans l'itération précédente.
- Tests de non-régression sur le périmètre déjà testé.

9) Si le périmètre bloqué est vide alors terminer l'itération N qui achève la phase d'exécution en :

- Vérifiant les dernières corrections livrées.
- En faisant des tests de non-régression sur le périmètre global si nécessaire.

Concernant l'étape 7 pour la gestion des observations, nous vous renvoyons vers le chapitre suivant dédié à cette activité qui comporte ses propres difficultés.

Nous allons nous focaliser ici sur les tâches de vérification des corrections, d'identification des cahiers de recette bloqués et sur la définition des tests de non-régression en cours d'itération.

La vérification des corrections suppose qu'une communication est établie de la MOE vers le CP Recette de manière claire et permanente. Ces livraisons ne peuvent en aucun cas être faites pendant les tests et surtout pas de manière invisible !

En effet, la validité des tests est tributaire de la stabilité de l'environnement de test. Or une livraison est par définition une action qui entraîne une instabilité.

 Si vous faites la recette d'une application web, n'oubliez pas de vider les caches des navigateurs avant tout test, notamment si l'application vient d'être modifiée.

Le CP Recette devra donc imposer un protocole de livraison et établir une convention avec la MOE et un mode opératoire rigoureux. Par exemple, une plage horaire sera définie (entre 12h et 13h ? Après 18h ?) ainsi qu'un jour précis de la semaine. Cette livraison pourra d'ailleurs être mentionnée comme étant un jalon du planning et avoir un impact contractuel avec des pénalités en cas de retard.

La livraison devra également faire l'objet d'une communication précise permettant de lister explicitement les anomalies faisant l'objet d'une correction et qui sont effectivement vérifiables.

 Nous l'évoquerons dans le chapitre suivant, mais nous déconseillons très fortement d'utiliser un outil de gestion des observations tel que Mantis pour implémenter le processus de livraison, tout simplement parce que l'état d'une observation dans cet outil pourra changer à tout moment : ce n'est pas une photographie d'un lot de corrections à un instant T.

Ainsi, sur la base de ce listing, votre équipe pourra vérifier chaque correction, c'est-à-dire exécuter le même test que celui qui avait mis en évidence la présence de l'anomalie détectée.

Attention, cette vérification ne demande pas de modifier l'instance du cahier de recette d'origine pour en consigner le résultat ! Si vous devez impérativement tracer ces vérifications, vous devrez utiliser une nouvelle instance du même cahier de recette. Vous devez en effet conserver la trace :

- Du test mettant en évidence l'anomalie à une date donnée sur la base d'une instance X d'un cahier de recette.
- Du test mettant en évidence la correction de l'anomalie à une autre date donnée sur la base d'une instance Y différente de ce même cahier de recette. Ce second test pourra ne pas être formalisé parfois : on se contentera de changer le statut de l'observation en la clôturant.

Après les vérifications des corrections, l'équipe de test devra tester le périmètre des scénarios et fiches bloqués dans l'itération précédente.

Pour cela, le plan de test doit permettre de consigner lesquels des tests sont bloqués ou pas.

Puis, si nous prenons l'exemple d'un plan de test sous Excel, ce plan déroulé à la fin de l'itération pourra être copié et modifié en ne retenant que les items bloqués.

Ainsi, si un onglet représente la liste des scénarios et fiches de test exécutés (une ligne par cahier de recette), sa copie modifiée permet de reconstruire un nouveau plan de test ne contenant que ce qui reste à faire.

Le statut "Bloqué" de ces items est alors positionné à "Test à faire", les affectations des tests aux testeurs peuvent être remaniées aussi et un nouveau périmètre de travail est dégagé.

Toute la difficulté de ce protocole réside alors dans la pertinence des tests de non-régression donc la bonne connaissance qu'a le CP Recette du métier.

Il faut comprendre que tout correctif ne nécessite pas nécessairement un test de non-régression de toute l'application : cela n'a aucun sens.

Si nous considérons la correction d'une faute d'orthographe d'un libellé par exemple, le seul test de vérification suffit.

En revanche, si l'anomalie concernait la création ou modification d'une donnée, a minima il vous faudrait :

- Vérifier la correction en effectuant la même action (création ou mise à jour).
- Exécuter au moins un test de consultation permettant l'affichage de la donnée en question dans n'importe quel point de l'application, voire partout où elle est affichée.
- Exécuter tout test dans lequel la donnée participe à une règle de gestion.
- Exécuter un test d'export de la donnée si elle participe à un tel flux.

S'ajoute à cette difficulté d'approche, la gestion des configurations matérielles - nous sommes en recette technique - pour laquelle selon le cas, la non-régression nécessitera ou ne nécessitera pas de refaire le test pour autant de configuration matérielle que nécessaire.

Si nous prenons par exemple une application web implémentant la fonction de virement d'un site de banque à distance, puis supposons que la règle de gestion du montant plafond de virement soit erronée, doit-on vérifier la correction sur tous les navigateurs ?

Notre expérience sur le terrain démontre ceci :

- Il faut au moins vérifier la correction sur un navigateur, de préférence celui qui est le plus utilisé par sécurité.
- Il est inutile de vérifier les autres navigateurs sauf peut-être Safari sur Macintosh qui présente parfois un risque accru d'incompatibilité justifiant un test de non-régression spécifique.

En revanche, si nous prenons la même application avec une anomalie relative à la gestion des marges des éléments graphiques, nous préconiserions de vérifier Internet Explorer dans toutes les versions et Firefox.

Quid des applications sur terminaux mobiles ? Le problème est alors si complexe qu'il faut quasiment tout tester à nouveau en guise de non-régression, notamment pour les anomalies relatives aux mises en page.

Ainsi la définition d'un périmètre pertinent de test de non-régression nous pousse vers la prudence. Retenez que le risque de dérive en phase d'exécution se trouve là principalement.

## d. CLÔTURER

La clôture d'une recette technico-fonctionnelle aboutit à la rédaction d'un bilan de recette. Notez que ce document pourra être envoyé dans des versions intermédiaires à l'issue de chaque itération mais que seul le dernier doit mentionner si l'application est validée ou pas, avec ou sans réserve.

Ce bilan doit être particulièrement factuel et préciser les points suivants :

- Le périmètre des tests prévus par configuration matérielle, nombre de fiches de test, nombre de scénarios de test, etc.
- Le périmètre des tests réalisés par configuration matérielle.
- Le périmètre des tests bloqués par configuration matérielle, en exposant une synthèse des points de blocage, le taux de blocage, c'est-à-dire le pourcentage des tests qui n'ont pas pu être déroulés sur le périmètre de ceux prévus.
- Les volumétries des observations détectées, des anomalies, au global et par criticité (cosmétique, mineure, majeure, bloquante), le taux de pertinence, c'est-à-dire le ratio entre les anomalies reconnues comme telle par la MOE sur le nombre total d'observations, l'histogramme quotidien des nombres d'anomalies, etc.
- Les volumétries des corrections et réponses de la MOE, ainsi qu'un taux de réponse (le ratio entre le nombre d'anomalies corrigées et le nombre de détectées).
- Listez les anomalies non encore arbitrées par la MOA et pour lesquelles l'aspect anomalistique ne fait pas l'unanimité, les évolutions détectées, les corrections du paramétrage...
- Le rappel des conventions de communication avec la MOE notamment le protocole des livraisons et la période pendant laquelle les outils et applications devaient être disponibles.
- Calculez un taux de disponibilité de l'application et un taux de disponibilité des outils sur la base des événements listés ci-après.
- Les événements survenus au cours de la recette :
  - Les livraisons (date, heure, fiche de livraison reçue en annexe au bilan).
  - L'inaccessibilité ou les instabilités de l'application (indiquez la date et la durée notamment).
  - L'inaccessibilité des outils de test, de l'imprimante, etc.

N'hésitez pas non plus à mentionner des axes d'amélioration, des préconisations futures, tant pour l'équipe de recette et les tests futurs que pour la MOA et la MOE.

## 3. Gérer la recette technique d'une maintenance évolutive

La recette technique d'une maintenance évolutive pourra se dérouler dans deux situations :

- Après la recette technique d'un chantier de release, intégralement formalisée et offrant ainsi un référentiel de tests complet et exploitable.
- Dans le cadre d'une application ancienne pour laquelle un référentiel de tests partiel existe, voire est inexistant.

C'est selon la situation que nous rencontrons que la démarche change alors du tout au tout comme nous allons le voir.

Notre recette restera découpée en quatre tâches séquentielles, mais chacune sera traitée différemment selon le contexte.

### a. ORGANISER

L'organisation de la recette technique d'une évolution, d'un lot de petites maintenances et correctifs, s'effectue sur la base d'une liste d'items qui pourront être de deux natures :

- Le correctif d'une anomalie détectée en production ou dans tout autre environnement de test, qui nécessite un test de vérification et éventuellement des tests de non-régression.
- Une évolution de "petite taille", qui nécessitera une recette dédiée aux exigences concernées ainsi que des tests d'interface et de non-régression aussi.

Ce périmètre de modifications doit idéalement être consigné dans un document unique qui permet de définir une version précise de l'application.

Nous devons alors supposer qu'il existe une sorte de "bon de commande" entre la MOA et la MOE qui permet de définir les items à traiter dans la nouvelle version de l'application.

À noter que selon l'organisation de ce processus (ou son absence), l'organisation de la recette technique s'en trouve alors impactée :

- Il est possible que les anomalies soient loties séparément des évolutions ou bien qu'elles soient traitées en commun.
- Le formalisme entre MOA et MOE pour convenir d'un périmètre de modification pourra être incomplet, parfois même absent, ce qui rend alors très difficile l'anticipation d'un périmètre de test. Dans ce second cas, c'est donc le processus de livraison qui en bout de chaîne permet à un chef de projet recette technique de savoir ce qu'il doit tester... au dernier moment !

Ensuite, la prédominance des tests de non-régression met en évidence le besoin d'un référentiel de tests. D'où l'approche différente que l'on aura selon qu'il existe ou pas.

En l'absence d'un référentiel de tests, le CP Recette n'aura pas d'autre choix que de formaliser les tests des nouveautés et, s'il a le temps, formaliser les tests permettant de vérifier les correctifs.

Car le périmètre des tests de non-régression est inconnu lorsque le référentiel de tests est vide !

Tout comme nous l'avons vu avec la recette des patchs, une réflexion devra être conduite sur une rétrodocumentation éventuelle des tests. Nous préconisons ici d'être pragmatique :

- Essayez de collecter les exigences en estimant leur criticité sans faire l'analyse de risque.
- Puis identifiez les exigences présentant un risque élevé pour vous focaliser sur les tests de première importance. Ce périmètre ainsi dégagé vous permettra de rédiger un ensemble minimaliste de tests qui servira de base pour définir les tests de non-régression de vos recettes de maintenance.
- Pour la gestion des configurations matérielles, étant donné que l'application est déjà en production, intéressez-vous aux statistiques d'utilisation des configurations en production et constituez alors votre plateforme en cohérence.

Une autre stratégie est également possible : rapprochez-vous pour cela du responsable des études afin de

connaître "plus ou moins" le périmètre des évolutions envisagées sur l'exercice de maintenance qui arrive. Vous pourrez alors définir un périmètre de test relativement à ces évolutions et non pas sur la base d'une analyse d'exigence "trop théorique".

À défaut de connaître le contenu des évolutions dans le détail, vous pourrez aussi vous fier aux budgets définis pour chacune des maintenances : il est évident que si 200 jours-homme sont prévus sur une année pour modifier une application, contre 10 jours-homme pour une autre, alors cette dernière nécessitera probablement moins de tests que la première.

Bref, complétez du mieux que vous pouvez votre référentiel de tests pour anticiper un périmètre de test.

Et que fait-on lorsque le référentiel de tests existe ? Son réemploi est la première évidence qui vient. Tout test constitué lors du chantier de release ou des chantiers de maintenance qui ont précédé est susceptible :

- Soit d'être mis à jour pour les petites évolutions.
- Soit d'être directement exécuté pour des tests de non-régression.

Mais le réemploi des tests existants n'est pas une démarche suffisante en elle-même.

Comme nous l'avons vu dans le chapitre relatif à la définition du périmètre d'une recette sur la base des analyses des risques et exigences, la hiérarchisation des tests s'appuie sur la probabilité de survenance d'un événement et plus particulièrement sur une notion de fréquence d'impact.

Or cette fréquence était déterminée sur la base de prévisions. Dès lors que l'application est en production, ces prévisions peuvent être remplacées par une mesure précise de la fréquence d'utilisation des exigences fonctionnelles.

Concernant l'impact en lui-même, la survenance d'incidents en production pourra elle aussi venir affiner les estimations. Par exemple, pour un virement bancaire, on pourra estimer un montant moyen viré et ainsi estimer l'erreur commise avec beaucoup plus de précision que lors du chantier de release.

Finalement, la matrice des risques devra être réévaluée, les estimations étant réajustées selon les mesures faites.

Notez que ces mesures peuvent d'ailleurs fluctuer dans le temps : la matrice est donc "vivante" sur le long terme.

Ceci signifie qu'un cas de test qui était de priorité Could pourra passer en Must ou réciproquement, entraînant de légères modifications des périmètres.

Un autre aspect que le CP Recette devra aussi estimer lors des recettes techniques lorsqu'il dispose d'un référentiel de tests, est la pérennité des configurations matérielles de référence.

Cet axe de réflexion ne concerne pas toutes les applications bien sûr mais intéresse tout particulièrement les applications web publiques et les applications mobiles.

Il faudra en effet anticiper les nouvelles configurations matérielles éditées sur le marché et qui n'étaient pas présentes lors du chantier de release. La portabilité des applications doit être une préoccupation constante.

En bref, c'est donc une véritable veille technologique qu'il convient de mettre en œuvre, quitte à faire des essais par anticipation avec les versions bêta des composants.

 Veillez à vous tenir informé des nouveaux navigateurs web, des plugins Media Flash Player, des plugins Adobe Reader, des évolutions des systèmes d'exploitation mobiles, des nouvelles dimensions de leurs écrans, etc.

En conclusion, l'organisation d'une recette dans un chantier de maintenance est aisément réalisable si celle-ci avait été faite lors du chantier de release : vous aurez peu de choses à réajuster évolution après évolution.

En revanche, l'absence de formalisation de la recette en release vous contraindra à une démarche beaucoup plus empirique en maintenance... et surtout plus coûteuse du fait d'avoir à réaliser une rétro-documentation partielle.

## b. PRÉPARER

Sur le plan du formalisme des tests, la démarche est la même en maintenance et en release.

Éventuellement, la manière dont est implémenté le processus de commande des lots entre MOA et MOE pourra avoir un impact.

Typiquement, s'il n'existe aucun référentiel de tests issu du chantier de release, peut-être que la démarche d'inclure les tests directement dans le "bon de commande" pourra être envisagée.

En effet, si l'on admet qu'un bon de commande est une liste d'items (anomalie ou évolution) alors ce document possède un formalisme propice à définir la liste des tests associés : les vérifications pour les anomalies, les nouveaux tests pour les évolutions.

En revanche, si un référentiel de tests existe, la préparation consistera alors à garder le même formalisme - le même outil aussi - et à maintenir ledit référentiel en ajoutant les nouveaux tests et en modifiant ceux existants.

Ainsi, le formalisme des tests (fiches de test et scénarios) est nécessairement hérité des chantiers qui ont précédé.

En revanche, le CP Recette devra davantage s'intéresser à la maintenance du jeu d'essai et des données prérequises, autres éléments de capitalisation.

Dans le chapitre précédent relatif à la gestion des données, nous avons déjà évoqué cet aspect. Comment gère-t-on les sauvegardes et restaurations ? Peut-on automatiser la maintenance d'un jeu de données dans le temps ? Le jeu d'essai et les données prérequises sont-ils dégradés par les tests ? Existe-t-il un phénomène de chronodépendance des données ?

Ainsi, toute la préparation d'une recette technique d'un chantier de release consistera surtout en une continuité dans la qualité des données.

C'est donc le cahier des jeux d'essai défini dans le chantier de release qui intéressera la recette de maintenance.

## c. EXÉCUTER

L'exécution d'une recette technique dans un chantier de maintenance offre beaucoup de similarité avec celle du chantier de release : elle s'appuie sur le périmètre des tests et la préparation définis dans les deux étapes précédentes.

Nous vous renvoyons donc à la section relative à l'exécution des tests d'un chantier de release.

Toutefois, quelques petites différences méritent d'être soulignées car la connaissance des recettes techniques précédentes permet au CP Recette de qualifier plus vite les observations :

- Si une anomalie est connue et n'a jamais été corrigée.
- Si une anomalie est connue, déjà corrigée mais réapparaît.
- Si une observation correspond à une erreur de paramétrage.

- Si une observation ressemble à une anomalie déjà connue, un rapprochement permettra un diagnostic plus rapide.
- La connaissance de l'architecture technique, des configurations matérielles et des spécificités techniques de l'application, permet une compréhension beaucoup plus rapide des résultats observés.
- Et enfin, la connaissance fonctionnelle nécessairement acquise au préalable permettra une exécution plus fluide des tests.

Point critique qu'il faudra cadrer : doit-on rouvrir une anomalie qui réapparaît ou saisir une nouvelle observation ? Tout dépend si l'on considère que l'anomalie est liée à l'application ou à une version de l'application...

Bien entendu, nous supposons ici que c'est la même équipe - ou au moins le même CP Recette - qui gère à la fois le chantier de release et ceux de maintenance ensuite.

#### **d. CLÔTURER**

La clôture d'une recette technique de maintenance se fera de la même manière que pour le chantier de release.

Toutefois, cette clôture sera aussi le moment de se projeter dans l'avenir : les statistiques globales des anomalies seront complétées et le regard porté sur l'application changera.

Par exemple, si l'on constate une brusque augmentation de la volumétrie des anomalies en maintenance, à qualité de tests constante, il faudra se poser la question des causes de cette variation. Est-ce ponctuel ? Est-ce un problème de fond ?

La clôture d'une recette reste donc un moment privilégié pour faire un bilan et réajuster la vision que l'on a du produit - toujours de manière factuelle, bien sûr.

Notez que si votre clôture consiste aussi à faire un bilan à froid, alors cette dernière se fera très probablement en fin d'exercice du chantier de maintenance puisqu'il est possible que la clôture d'une recette d'un lot de maintenances soit aussi la clôture de l'exercice annuel lui-même.

# **La recette fonctionnelle "pure" ou VABF**

## **1. Le contexte**

### **a. Quand ?**

Une recette fonctionnelle est dite "pure" en langage courant lorsqu'elle n'implique pas une recette technique simultanée, c'est-à-dire que celle-ci a été réalisée au préalable.

La MOA se limite ici à faire des tests de validation fonctionnelle préparés par elle et exécutés sous sa gouverne. Les testeurs pourront être membres de la MOA - voire des décideurs du projet qui souhaitent se faire leur propre opinion - ou bien des utilisateurs finaux spécifiquement désignés pour cette action.

Bien évidemment, une telle recette ne peut s'effectuer qu'avec une application qui a déjà une relative qualité : pas d'anomalie bloquante, une présentation déjà propre dans sa perception, normalement compatible avec plusieurs configurations matérielles.

### **b. Pourquoi ?**

Le but d'une telle recette est la plupart du temps l'homologation, c'est-à-dire une VABF (vérification d'aptitude au bon fonctionnement).

Son objectif revêt donc un aspect officiel de validation pour signifier qu'un jalon du projet a été franchi avec succès ou pas. Si la MOE est externe, cette VABF aura même des impacts contractuels et des incidences sur la facturation qui est normalement déclenchée pour partie à ce stade du projet.

Le lecteur comprend ici pourquoi il est délicat de cumuler la recette technique et la recette fonctionnelle, et pourquoi un désengagement trop fort de la MOA en matière de test doit être évité. Le principe de partager les tests entre la recette technique et la recette fonctionnelle se trouve ici pleinement justifié : la première structure ne saurait se substituer à la seconde.

Dans les sections qui suivront, nous partirons donc du principe qu'une recette technique aura eu lieu au préalable.

Nous sommes conscients que notre positionnement est ici très théorique... mais la théorie a parfois du bon !

## **2. Gérer la recette fonctionnelle**

L'organisation d'une recette fonctionnelle est relativement similaire à celle d'une recette technique. Nous la définissons donc en décrivant leurs différences.

Bien entendu, une recette fonctionnelle d'une release s'appuiera sur la recette technique de cette même release, celles de maintenance s'appuyant l'une sur l'autre.

### **a. ORGANISER - Formaliser une homologation**

L'objectif de la recette fonctionnelle étant une validation contractuelle et non pas la remontée d'un maximum de considérations techniques, il en découle une différence de formalisme dans l'organisation.

À commencer par l'évaluation du périmètre des tests, nécessairement une sous-partie des tests techniques, se limitant à des tests en rapport avec un jeu d'essai exprimant une réalité fonctionnelle.

Pour des raisons budgétaires, le CP MOA ayant en charge la VABF pourra d'ailleurs se limiter aux exigences correspondant à des criticités fortes, estimant qu'une anomalie fonctionnelle ne remettra pas en question l'homologation. Après tout, il y a un chantier de maintenance qui suit...

Concernant les configurations matérielles, aucun test n'est normalement prévu puisque la compatibilité est une exigence technique. La démarche du CP MOA est ici de définir une configuration matérielle étalon (et même un PC étalon), généralement prise sur la base de celle qui est la plus répandue. Idéalement, cette définition est d'ailleurs communiquée tant à la MOE qu'à l'équipe de recette technique qui fera en sorte de faire des tests sur cette configuration particulière.

En revanche, une recette fonctionnelle pourra demander des tests de multilinguisme, langues qui comme nous l'avons vu pour les recettes techniques, sont susceptibles d'être gérées comme des "configurations matérielles logiques".

Nous conclurons en rappelant une spécificité du dossier d'organisation d'une recette fonctionnelle : la définition des critères d'homologation.

En effet, puisque le rôle de la recette fonctionnelle est une homologation (parfois contractuelle), il sera absolument nécessaire de définir quelles typologies d'anomalies entraîneront le refus, lesquelles seront acceptées pour une validation avec réserve.

Le dossier d'organisation pourra donc définir des règles très précises. Nous vous proposons ici un exemple de ce formalisme :

	<b>Seuils d'acceptation</b>
Anomalies bloquantes	0 sur les tests de priorité Must 0 sur les tests de priorité Could
Anomalies graves	0 sur les tests de priorité Must 5 sur les tests de priorité Could 10 sur les tests de priorité Should
Anomalies mineures	5 sur les tests de priorité Must 10 sur les tests de priorité Could 20 sur les tests de priorité Should

Il y a toutefois un cas particulier qui fera qu'une recette fonctionnelle pourra revêtir des aspects de la recette technique : dès lors que l'environnement dans lequel elle est exécutée est moins dégradé que celui de la recette technique.

Nous vous rappelons en effet qu'un environnement dégradé en recette technique conduira à définir un périmètre d'exigences pour lequel les tests ne sont pas concluants, nécessitant des tests ultérieurs.

C'est justement lors de la recette fonctionnelle qu'un complément de tests pourra être envisagé.

## b. PRÉPARER - Mettre l'accent sur le jeu d'essai

La préparation d'une recette fonctionnelle est bien sûr similaire à celle d'une recette technique mais elle se limite à des tests fonctionnels.

Concrètement, seuls des scénarios sont donc rédigés. Et en aucun cas des fiches de test, outil spécifique aux recettes techniques.

Ceci signifie que le jeu d'essai et les données prérequises jouent un rôle prédominant dans la recette. Les données constituent donc le paysage dans lequel la recette sera exécutée : son sens fonctionnel est donc

primordial.

Concernant le formalisme des scénarios de recette, ils devront bien entendu être de très bonne rédaction, clairs et précis. Aucune ambiguïté ne doit heurter le testeur.

La conception d'un scénario de recette dont l'usage est destiné à une équipe d'utilisateurs notamment, demandera un effort de rédaction, là où pour une équipe d'experts en test en recette technique, le concepteur se montrerait plus souple.

-  Si vous êtes prestataire externe en charge de gérer des recettes fonctionnelles "pures", nous préconisons de mettre en place un processus de validation des scénarios et jeux de données par des responsables métier avant d'exécuter ou faire exécuter les tests.

## c. EXÉCUTER - Prendre un recul fonctionnel

L'exécution des tests s'effectue sur la base des scénarios rédigés. La différence avec une recette technique se fera sur l'ouverture des observations et l'accompagnement des testeurs.

Car des utilisateurs ne verront pas la même chose qu'un expert en test. L'équipe s'attachera en effet à repérer non seulement les non-conformités fonctionnelles, mais portera aussi son regard plus loin.

Nous ne pourrons pas préconiser une démarche précise ici mais évoquer quelques exemples significatifs.

Par exemple, lors d'un traitement permettant un achat en ligne, il est courant de trouver un formulaire demandant la saisie d'un RIB. Et il est tout aussi fréquent qu'en l'absence de spécifications fonctionnelles précises, les développeurs implémentent une règle demandant la saisie d'un numéro de compte strictement numérique.

Or, il faut savoir qu'un numéro de compte peut être alphanumérique. Ainsi, une règle est mise en place à cause d'une compréhension usuelle en décalage avec une réalité d'un métier méconnu. Dans notre exemple, la banque.

Autre exemple avec un formulaire de saisie d'une personne morale et son code SIRET. Le SIRET définit un établissement d'une personne morale, donc une implantation géographique et non la personne morale elle-même identifiée par le SIREN - les neuf premiers chiffres du SIRET. Cette distinction n'est pas toujours connue... ni d'ailleurs le fait que le SIRET est une notion française qui suppose alors que l'adresse d'implantation de la société est sur le territoire français et non à l'étranger.

Nous évoquerons enfin un dernier exemple encore plus significatif issu d'un cas très réel : une application de banque en ligne pour les professionnels. Celle-ci possédait une fonctionnalité permettant de créer des listes de remises de chèques. Ces données étaient normalement saisies et enregistrées dans le système : aucune anomalie ne fut détectée.

En revanche, l'utilisateur final ne pouvait pas consulter les listes de remises : la fonction permettant de consulter les données en attente de traitement était inexistante. Le blocage est alors purement fonctionnel puisque seul un utilisateur en situation sera confronté à un manque d'information.

Est-ce une anomalie ? Est-ce une évolution ? Le débat est ouvert...

## d. CLÔTURER - Homologuer ou ne pas homologuer ?

Autant les recettes techniques et fonctionnelles se ressemblaient jusqu'ici, autant la différence se notera au moment de la clôture.

Bien entendu, le bilan de clôture rappellera le périmètre prévu, le périmètre réalisé, les volumétries des

observations faites...

Cependant, le point le plus important consistera à signifier une validation officielle du livrable ou son refus.

Le rappel des critères de choix définis dans le dossier d'organisation sera fait et la mesure du respect desdits critères ensuite. Exemple :

	<b>Seuils d'acceptation</b>	<b>Résultats obtenus</b>
Anomalies bloquantes	0 sur les tests de priorité Must 0 sur les tests de priorité Could	0 sur les tests de priorité Must 0 sur les tests de priorité Could
Anomalies graves	0 sur les tests de priorité Must 5 sur les tests de priorité Could 10 sur les tests de priorité Should	0 sur les tests de priorité Must 4 sur les tests de priorité Could 9 sur les tests de priorité Should
Anomalies mineures	5 sur les tests de priorité Must 10 sur les tests de priorité Could 20 sur les tests de priorité Should	<b>6 sur les tests de priorité Must</b> 3 sur les tests de priorité Could 5 sur les tests de priorité Should

Dans l'exemple ci-dessus, les critères sont tous respectés sauf un sur les anomalies mineures. Le CP MOA aura ici toute latitude pour valider avec réserve ou refuser l'application bien que le seuil d'acceptation soit dépassé.

# Rappels importants

## 1. Définitions

### a. L'observation

Quel que soit le moment où un usager ou un membre du projet croit détecter une anomalie, il doit émettre une observation.

Cette observation n'est pas une anomalie mais un document, le rapport décrivant un comportement du logiciel qui fait que son rapporteur s'interroge.

Qualifier immédiatement une observation d'anomalie est donc présomptueux et nous recommandons une certaine vigilance afin que la confusion des concepts ne vienne pas altérer le processus de qualification qui traite les observations.

L'anomalie est ce qui nécessite la correction du logiciel : c'est donc au dernier moment et lorsqu'il y a un consensus entre MOA et MOE sur l'observation que l'on pourra la qualifier d'anomalie.

Une fois ce raccourci trop facile évité, nous pouvons définir l'observation. C'est avant tout une description factuelle qui est caractérisée par :

- L'environnement dans lequel l'observation est faite, ou à minima, la distinction entre l'environnement de production et tout autre environnement, car une anomalie détectée en production motivera peut être une intervention en urgence.
- Le nom du rapporteur de l'observation, si possible sa fonction au sein de l'entreprise (c'est une donnée qui informellement peut participer à estimer l'urgence d'une intervention !).
- Le ou les logiciels faisant l'objet de l'observation, et les lieux précis où la manifestation est vue (les exigences fonctionnelles concernées, donc).
- La date à laquelle l'observation est faite.
- La liste des actions réalisées qui a mené à la manifestation qui motive la déclaration.
- Éventuellement, la reproductibilité, c'est-à-dire la capacité de pouvoir réitérer la survenance de l'observation en ré-exécutant les actions indiquées. Si la reproductibilité n'a pas été évaluée, l'indiquer aussi.
- La description du constat, éventuellement des pièces jointes (captures d'écran, rappel des spécifications...).
- Le caractère bloquant ou non de l'impact, c'est-à-dire l'incapacité à poursuivre l'action métier utilisant l'exigence fonctionnelle. Ce peut être un "crash" de l'application - blocage technique - mais ce peut être aussi un blocage au sens fonctionnel, par exemple l'impossibilité d'insérer une information dont le format spécifique n'a pas été pris en compte et qui est refusée par l'application.
- Et enfin, l'impact envisagé : perte financière, dégradation de l'image de marque, perte de productivité, faille de sécurité, inconfort d'utilisation...

Et en aucun cas une quelconque notion d'urgence n'est exprimée dans l'observation, chose qui ne peut pas être définie par le rapporteur dans la majorité des cas.

En effet, une observation n'est jamais urgente : c'est la correction de l'anomalie si s'en est une, qui l'est ou ne l'est pas.

En revanche, une observation s'effectue sur une exigence dont la criticité est connue... pourvu qu'une analyse de risques ait été réalisée auparavant. Ainsi, par défaut une observation portera la criticité de l'exigence fonctionnelle concernée.

Cette criticité pourra être éventuellement réévaluée en fonction du contexte. Par exemple, si un traitement comptable hautement critique s'exécutant une fois par an en décembre fait l'objet d'une anomalie, cette dernière aura par défaut une criticité "Haute", criticité qui sera révisée en fonction de la date de survenance :

- Si nous sommes en janvier, nous avons toute l'année pour la corriger.
- Si nous sommes en novembre, nous n'avons qu'un mois !

De la même manière, une anomalie qui serait détectée en environnement de recette technique n'a pas la même criticité que la même anomalie détectée en production ! "Tout est relatif" dirait un certain Albert.

Nous détaillerons plus largement tout cela plus loin.

### **b. Anomalie, bug, dysfonctionnement et non-conformité**

Comme nous venons de le voir, l'observation décrit un comportement suspect. Ce qui n'en fait pas une anomalie systématiquement. L'observation entre alors dans un processus particulier : la qualification.

L'objectif de ce processus est de statuer sur la nature de l'observation et sur l'urgence éventuelle à réaliser une correction si cette nature était anomalistique.

La question se pose donc de savoir ce que l'on entend par anomalie.

Si l'on se réfère à un simple dictionnaire, une anomalie est une chose qui sort de la normalité, ce qui ne veut pas dire qu'elle n'est pas souhaitable. Cette définition étant trop imprécise, nous lui préférerons deux autres qualificatifs plus rigoureux :

- La non-conformité (voulue ou non voulue).
- Le dysfonctionnement (ou défaut).

La non-conformité est en effet un terme plus précis : ce terme suppose qu'il existe une définition de ce qui est conforme, c'est-à-dire normal. Cette vision normalisée induit alors qu'une non-conformité s'identifie par comparaison entre un texte de référence (les spécifications fonctionnelles, une maquette...) qui définit une attente et une observation faite qui diffère de cette dernière.

La non-conformité est donc l'écart dûment constaté avec un résultat attendu : si c'est une chose qui n'est pas souhaitée, c'est donc une anomalie par extension. Mais une non-conformité pourrait tout aussi bien être souhaitée !

Est-ce à dire que toute anomalie est une non-conformité ? Bien sûr que non. Il y aura aussi des comportements observés du logiciel, non voulus, pour lesquels aucune comparaison ne sera possible avec un texte de référence tout simplement parce qu'il serait naïf de croire que toutes les règles sont écrites.

Un logiciel est une réponse à une culture d'entreprise et est aussi l'expression de la culture d'un peuple, ne serait-ce qu'en vertu des informations affichées à l'écran.

Vous ne trouverez ainsi jamais de spécifications mentionnant que le logiciel à produire est fait pour les droitiers, qu'un champ permettant de saisir un nom propre est alphabétique et doit accepter les espaces, tirets et apostrophe (le champ sera simplement mentionné comme étant alphanumérique sans plus de précision)...

Et que dire alors des règles métier élémentaires propres à une entreprise ?

Toutes ces règles implicites pourront donc être naturellement omises dans les spécifications.

Vous trouverez également des erreurs induites par les composants logiciels du marché eux-mêmes, référencées par leurs éditeurs parfois, et qui correspondront à des défauts de fabrication à connaître nécessairement, à contourner éventuellement.

Finalement, c'est la notion de dysfonctionnement que l'on retiendra pour toutes ces anomalies pour lesquelles les spécifications ne sont d'aucune utilité.

-  Concernant le "bug" ou sa traduction le "bogue", nous en éviterons l'usage qui ne correspond pas à une description factuelle et normalisée.

### c. La demande d'évolution

Le processus de qualification amène donc une observation à devenir :

- Une observation sans objet (le rapporteur s'est trompé ou il y a redondance).
- Une non-conformité
  - Qui devra donc être corrigée systématiquement si elle a un impact négatif.
  - Qui conduira à une correction des spécifications si l'impact est positif.
- Un dysfonctionnement, pour lequel une réflexion devra être menée entre MOE et MOA, la recherche d'un consensus.

Mais ce processus peut aussi amener à une réflexion plus importante : une évolution. Dans un tel cas, le processus de qualification étant achevé, l'observation est clôturée en "Évolution".

S'enchaîne alors le processus de récolte des demandes d'amélioration sur la base des informations recueillies, celles-ci étant parcellaires : une évolution ne se décrit pas de la même manière que l'observation d'une anomalie supposée, à commencer par les gains potentiels que représentent une demande d'évolution là où une anomalie représente au pire une perte financière au mieux un désagrément !

-  Il est donc impératif de clôturer une observation qualifiée en évolution future ces dernières n'étant pas gérées dans le même processus.

## 2. Processus documentaires

Avant de décrire les processus de gestion des observations en place dans une entreprise, nous décrirons quelques généralités sur les processus de gestion documentaire.

Il s'agit d'une piqûre de rappel : si vous maîtrisez cette partie, ne vous y attardez pas.

### a. Les états d'un document

Un document pourrait se décrire comme un tout décomposé en deux parties :

- Des informations (titre, auteur, date de création...).
- Un ou plusieurs contenus (TXT, Word, Excel...).

Dans un outil de gestion documentaire, il est fréquent que des processus de validation soient implantés : on parlera de workflows.

Ces derniers s'appuient principalement sur une information : l'état du document. Ses valeurs peuvent être :

- Brouillon - auquel cas le document est en cours de rédaction chez des auteurs.
- Soumis - auquel cas le document est figé et peut faire l'objet d'annotations de la part d'un responsable de publication.
- Publié - le document est alors gelé définitivement et accessible en lecture seule.

Nous nous appuierons sur cette notion documentaire pour bâtir nos processus.

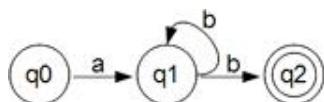
### b. Processus et formalisation : de l'usage des graphes d'états finis

Un processus de validation documentaire peut être manuel ou partiellement automatisé dans un outil par un workflow, peu importe : il pourra être formalisé par un graphe d'états finis.

Ce graphe est un ensemble de nœuds et de transitions orientées étiquetés :

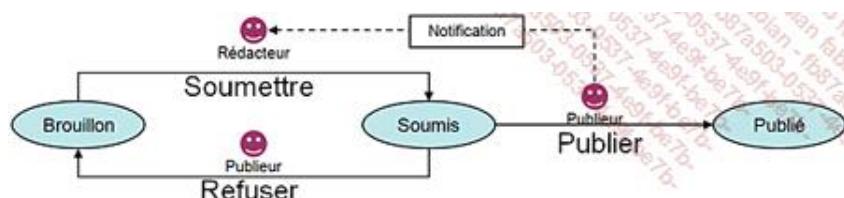
- Les nœuds sont étiquetés à l'aide des états du document. Il y a un nœud "départ" et des nœuds "arrivée".
- Les arcs sont étiquetés à l'aide des noms d'action correspondant au changement entre deux états lors de l'exécution d'une transition. Logiquement, les arcs sortant d'un nœud représentent l'ensemble des choix possibles d'un acteur (ou d'un groupe d'acteurs) :
  - S'il n'y a qu'un seul acteur, il s'agit d'un choix simple d'exécuter l'une des transitions parmi un ensemble de transitions possibles.
  - S'il y a plusieurs acteurs, soit il y a attente d'une décision commune (c'est donc un vote) soit le premier qui fait le choix l'emporte sur tous les autres qui sont ignorés.

Le graphe d'états finis qui en résulte est similaire à celui d'un DFA (*Deterministic Finite-state Automata*) qu'il ne faudra pas confondre avec celui d'un NFA (*Non-deterministic Finite-state Automata*) :



 L'exemple ci-dessus est un NFA dont les états sont  $\{q_0, q_1, q_2\}$ ,  $q_2$  étant un état final, et les étiquettes des transitions  $\{a, b\}$ . Notez que la transition  $b$  apparaît deux fois en sortie de l'état  $q_1$ , ce qui signifie qu'il y a indétermination. Cette indétermination fait que vous ne pouvez pas utiliser les NFA pour modéliser un workflow documentaire : vous devez absolument utiliser un automate déterministe.

Dès lors, un processus documentaire peut être représenté à l'aide d'un formalisme voisin :



Le lecteur notera la précision du vocabulaire employé pour les étiquettes :

- Les états (Brouillon, Soumis, Publié) décrivent toujours une situation observable du document : un état.
- Les transitions (Soumettre, Refuser, Publier) sont toutes des actions exprimées par un verbe et associées à un profil précis d'utilisateur.

 Notez qu'en aucun cas, les étiquettes des états ne désignent une situation future comme par exemple "à publier". Il est extrêmement important d'être concis dans les dénominations des états et des transitions, au risque de définir un graphe amenant de fortes ambiguïtés.

Nous verrons plus loin que l'usage d'un DFA est fondamental et que nous ne tenons pas un discours théorique : il y a une réalité concrète derrière ce concept malheureusement trop souvent oublié notamment lorsque des workflows sont définis dans des outils tels que Mantis ou le module Defect de Quality Center !

### c. Les triggers (déclencheurs)

Dès lors qu'une transition est effectuée - et si cette dernière est implémentée dans un outil - un traitement pourra être automatiquement exécuté : un déclencheur ou trigger.

Le plus souvent, ce traitement consistera à agir sur le document véhiculé, par exemple en enclenchant une impression ou une conversion.

Le déclencheur pourra aussi modifier les attributs du document : fixer la date de publication, déplacer le document dans l'arborescence documentaire... et donc agir sur les permissions des utilisateurs du système.

### d. Les notifications

Un déclencheur spécial est évoqué ici : l'envoi de courriels de notification. En effet, lors du franchissement de certaines transitions vers un état donné, il est possible dans les outils de gestion documentaire de définir l'envoi de notifications automatiques.

Ces dernières permettront :

- D'accuser réception d'un document.
- De notifier la publication d'un document.
- Etc.

Dans notre exemple précédent, les notifications sont représentées par une flèche en pointillé mentionnant explicitement qu'il s'agit d'une notification. Cette flèche partira de la personne ayant exécuté la transition à destination des profils devant être informés du changement d'état.

En aucun cas les arcs représentant des notifications ne seront issus d'un état du document : ils ne sont pas des transitions du graphe gérant un document et représente le flux d'un autre document : l'e-mail.

 Attention ! Un profil qui ne participe pas au processus peut être notifié également : il n'intervient pas sur le document traité mais est informé de son cheminement.

## 3. Les processus de gestion des observations : un survol

Dans la section ci-dessous, nous rappelons rapidement les principaux processus qui intéressent une MOA et/ou une MOE : les observations anomalistiques et les demandes d'évolution. Puis nous détaillerons dans les sections d'après chacun de ces processus progressivement en les agrégeant successivement puisqu'ils cohabitent et sont parfois interdépendants.

Nous utiliserons pour cela les concepts de gestion documentaire évoqués dans la section précédente : le formalisme des graphes d'états finis déterministes (DFA).

### **a. Les incidents de production**

Certainement le plus important de tous, le processus permettant à un utilisateur (ou quiconque) de signaler un comportement suspect dans l'environnement de production, normalement en direction de la MOA, les utilisateurs en étant plus proches que de la MOE. Étant donné que ce processus a une dimension publique, il doit normalement être implémenté de sorte qu'il est facile à initialiser et surtout autorise une forte réactivité.

Les risques de perte financière ou de dégradation de l'image de marque de l'entreprise peuvent être très forts. Si l'impact se limite à l'intérieur de l'enceinte de l'entreprise et n'est pas visible des clients, il y aura peut-être qu'une simple perte de productivité mais à minima une perte de confiance dans un outil de l'entreprise également.

Dans tous les cas de figure, ce processus doit permettre une intervention en urgence, donc faire l'objet d'une attention particulière comme nous le verrons plus en détail dans le formalisme que nous proposons.

### **b. Les demandes d'évolution**

Ce processus concerne au premier chef la MOA : il s'agit de collecter toutes demandes d'évolution ou d'amélioration du système d'information. D'ores et déjà, nous pouvons distinguer plusieurs niveaux dans ce processus :

- La demande d'amélioration d'une application existante, qui ne nécessite généralement pas une étude d'impact fonctionnel d'envergure, mais pour laquelle les gains et risques sont évalués. Généralement, ces demandes d'évolution sont traitées dans le chantier des petites maintenances évolutives de l'application concernée, conjointement avec les corrections d'anomalies. C'est ce processus qui suit éventuellement celui de détection des anomalies lors d'une recette si l'observation relève d'une évolution.
- La proposition d'une évolution significative d'une application existante (refonte) ou une application nouvelle, voire pour un groupe d'applications et qui nécessite un chantier de release.
- La transformation de l'organisation de toute ou partie d'une entité : un programme qui aura des impacts multiples sur le système d'information. Il s'agit d'un projet organisationnel d'abord, d'ingénierie ensuite. Nous l'indiquons pour information : ce point n'est pas l'objet du présent ouvrage mais sachez que de tels programmes peuvent déboucher sur des refontes totales d'un système.

Bien entendu, chacun de ces processus induit ou peut induire une ou plusieurs recettes tant techniques que fonctionnelles.

### **c. Les observations faites pendant une recette fonctionnelle**

Ces observations sont ouvertes par l'équipe de recette pilotée par la MOA lors de l'exécution des tests métier. Normalement, elles sont en petit nombre car la recette technique qui a précédé est censée avoir soumis l'application à un maximum de tests techniques garantissant une relative qualité logicielle.

L'équipe métier pouvant être un simple utilisateur de la solution aussi bien qu'un membre permanent de la MOA, ces observations relèvent le plus souvent de remarques purement fonctionnelles mais peuvent aussi concerner la souplesse d'utilisation (donc l'ergonomie et la cinématique) :

- Règle de gestion mal implémentée (contrôle de saisie, règle de calcul...).
- Information absente ou affichée partiellement (et qui faciliterait le travail de l'utilisateur).
- Cinématique manquant de souplesse ou peu intuitive.
- Fautes d'orthographe, petites anomalies cosmétiques...

Les utilisateurs finaux s'attachent à faire des observations relativement à leurs activités quotidiennes et leurs habitudes, à des automatismes que ni les développeurs, ni l'équipe de recette technique ne peuvent nécessairement pressentir.

Ce qu'il faudra retenir, c'est qu'il n'y a normalement aucune urgence à corriger de telles anomalies lorsqu'elles sont détectées : nous sommes en environnement de recette métier, et le principal impact concernera le planning des recettes si le correctif doit impérativement être réalisé. Au pire, la date de mise en production sera reculée... mais il n'y aura aucun impact financier direct puisque l'anomalie n'est pas détectée en production.

 En l'absence d'une structure organisationnelle dédiée aux recettes techniques, les recettes fonctionnelles s'y substituent et deviennent donc technico-fonctionnelles. Dans un tel cas, reportez-vous à la section qui suit naturellement.

#### d. Les observations faites pendant une recette technique

Normalement, les observations faites lors des recettes techniques sont nombreuses : l'équipe de test en remonte un maximum. Le but est en effet de permettre à la MOE de rectifier autant que possible tous les écarts avec les spécifications et les dysfonctionnements.

C'est également l'occasion de vérifier la cohérence globale de la solution logicielle et de conduire les tests que n'a pas pu mener la MOE en raison des contraintes techniques du projet (le plus souvent un environnement trop incomplet).

Ces observations pourront aussi être d'ordre fonctionnel selon que la recette revêt des aspects métier ou purement technique sinon.

Les rapporteurs de ces observations sont idéalement des professionnels du test dédiés au projet.

Et tout comme pour une recette métier, il n'y a pas d'urgence à corriger les anomalies détectées ici sinon pour éviter les décalages de planning que les livraisons à répétition entraînent.

Nous nous attarderons tout particulièrement sur la description de ce processus qui malheureusement fait trop souvent l'objet de mauvaises pratiques.

# Processus de gestion des incidents de production

Nous détaillons ici le processus mentionné dans les sections précédentes.

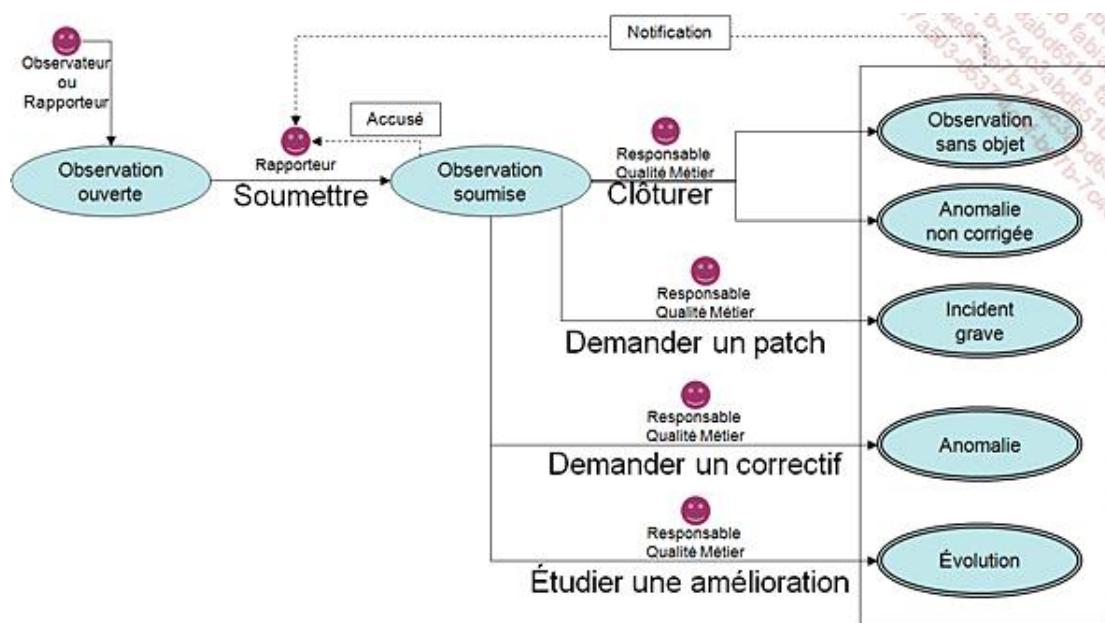
## 1. Principes organisationnels

### a. Formaliser le ou les workflows documentaires

Nous considérerons dans un premier temps que la MOA est une structure monolithique en charge de collecter les besoins et incidents de production de l'ensemble du système d'information, dans tous les métiers.

Le processus de gestion des incidents de production serait alors le suivant :

- 1) Ouverture directe ou indirecte d'une observation.
- 2) Soumission à un responsable qualité logiciel membre permanent de la MOA.
- 3) Analyse par ce responsable permettant de qualifier l'observation.
- 4) Lancement manuel du processus suivant selon la nature de l'observation.



Le premier point à retenir dans ce processus concerne son ouverture : celui qui fait une observation n'est pas nécessairement celui qui la rapporte.

En effet, deux situations sont possibles car l'observation peut très bien être réalisée par une personne étrangère à l'entreprise (un partenaire, un client, un internaute...) pour laquelle un retour n'est d'ailleurs pas toujours possible.

Le rapporteur en charge d'initialiser le processus d'ouverture informe donc l'observateur de manière informelle que son signalement a été pris en compte, puis il initialise le workflow par délégation ou en son nom propre s'il est lui-même l'observateur.

Ce rapporteur sera le plus souvent une personne du métier, proche de la MOA par définition. Et elle saisira dans l'outil de collecte des incidents de production des informations permettant au destinataire de qualifier

l'observation.

Les informations vitales sont ici :

- l'identité du rapporteur,
- l'impact (financier ? perte d'information ? image de marque ?),
- la fréquence de l'impact (minute, heure, quotidienne, ... aléatoire),
- sa description.

Dès lors que l'observation est soumise à un responsable qualité métier, ce dernier a les éléments factuels qui lui permettront d'estimer laquelle des quatre options qui s'offrent à lui est la bonne.

Premier cas de figure : **l'observation est sans objet** ou bien une anomalie connue qui ne peut pas être corrigée. L'incident est alors immédiatement clos. Ce peut être aussi l'occasion de voir s'il n'y a pas une erreur de perception du logiciel qui laisse penser à une anomalie là où il n'y a rien.

Le deuxième cas de figure est **la demande de patch** : le responsable qualité détecte une urgence qu'il faut traiter, parfois dans l'heure. C'est en général le cas lorsque l'incident détecté provoque une perte financière. Il faut alors corriger le logiciel dès que possible... mais aussi rattraper les erreurs déjà commises. Le processus de demande de patch est donc enclenché auprès de la MOE et à l'issue de son intervention, une recette spéciale sera mise en œuvre à réception du correctif.

La troisième possibilité correspondra au **traitement d'un incident d'impact faible** qui nécessite une correction mais sans urgence particulière. Le correctif pourra donc être réalisé dans le chantier de maintenance avec d'autres corrections et petites évolutions de la même application. Là aussi, c'est la MOE qui sera sollicitée pour la correction, et le processus de recette technique et/ou fonctionnelle sera enclenché à son issue.

Enfin, dans le cas d'**une évolution**, l'incident ne sera pas transmis à la MOE mais au responsable des études de la MOA - fonction qui pourra d'ailleurs être cumulée avec celle de responsable qualité métier dans les petites organisations.

Dans ce dernier cas de figure, l'incident est clos mais une étude est initialisée afin de déterminer les gains, risques et contraintes possibles pour sa mise en œuvre, car l'effort nécessaire sera peut-être très supérieur au budget de maintenance prévu initialement : seules les très petites évolutions sont traitées au cours de ce chantier.

### b. Définir un support de collecte : outil ou simple fichier ?

Il existe de nombreux outils souvent gratuits pour gérer les incidents et les anomalies. Mais un simple formulaire Excel et une adresse mail spécifique comme point d'entrée pourront être suffisants à implémenter ce processus.

Le plus important sera de définir un processus qui réponde au besoin, c'est-à-dire un moyen pour un responsable qualité d'évaluer l'impact et sa fréquence afin d'évaluer l'urgence éventuelle d'un incident demandant un patch.

### c. Interactions avec la collecte des demandes d'évolution

Le processus de collecte des demandes d'évolution pourra être enclenché à l'issue de la qualification d'un incident. Étant interne à la MOA, il est susceptible d'être mutualisé avec celui de déclaration des incidents.

Nous attirons l'attention du lecteur à ne pas confondre ces deux processus, idéalement en leur définissant des supports de collecte et des interlocuteurs différents :

- Un responsable qualité métier pour les incidents.
- Un responsable des études métier pour les demandes d'évolution.

Bien évidemment, ces fonctions sont cumulables, ce qui veut dire qu'il peut exister une confusion entre ces deux processus.

S'il s'agit de deux personnes distinctes, une bonne communication entre ces deux profils de la MOA permettra de garantir un fonctionnement efficace des deux processus en question.

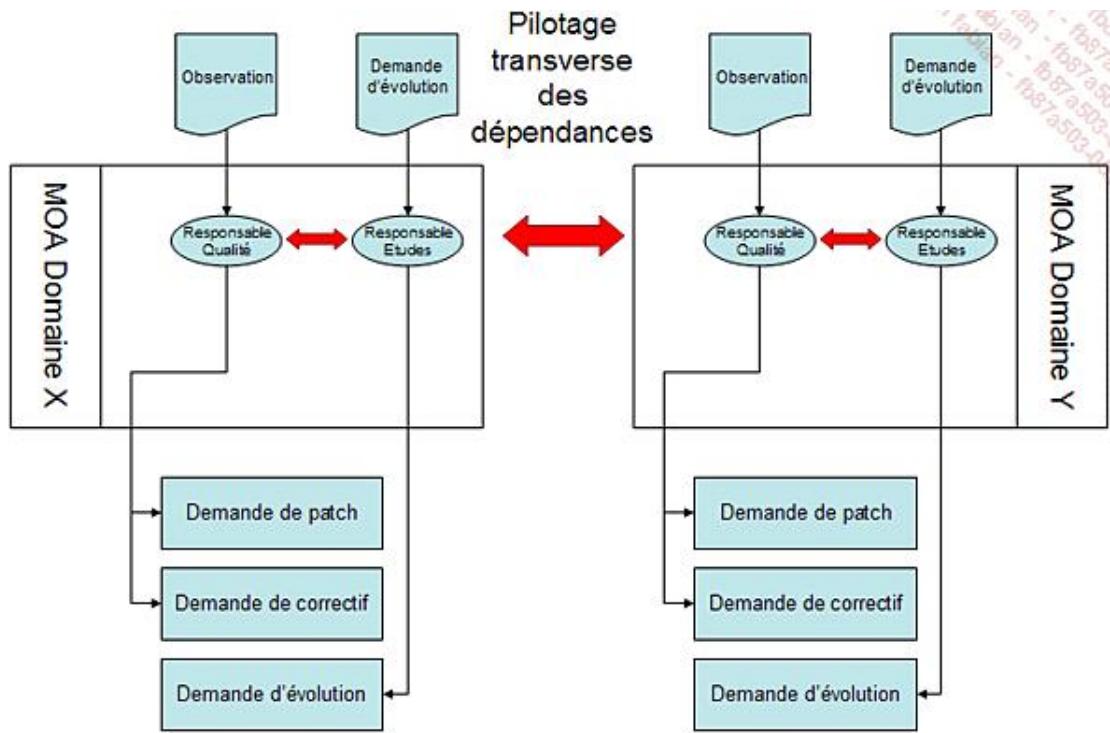
#### d. La gestion des dépendances transverses

Dans un second temps, nous pourrions envisager qu'une MOA soit de taille plus conséquente, structurée par domaine métier.

Dans une telle démarche, le processus décrit précédemment se trouverait donc dupliqué autant de fois qu'il y a de domaines, avec un responsable des études et un responsable qualité par entité organisationnelle.

Ceci suppose alors de mettre en place un processus transverse car une observation pourra être liée à une autre observation d'un autre domaine, voire il faudra mettre en place une MOA dédiée aux projets transverses.

La principale difficulté des responsables qualité et des études sera alors de détecter les dépendances entre correctifs et évolutions afin de les gérer.



## 2. Les risques

#### a. Confondre incident et demande d'évolution

Nous n'insisterons jamais assez sur la confusion possible entre incident de production et demande d'évolution.

Le premier des risques est en effet situé là : laisser échapper un incident majeur demandant une intervention

rapide suite à une erreur de saisie et qu'il soit confondu avec une évolution, qui elle n'est pas urgente.

Une anomalie peut engendrer des pertes financières parfois très importantes : c'est un risque qu'il convient de circonscrire autant que possible et par tous les moyens.

L'un de ces moyens consiste à séparer ces deux processus de collecte non seulement dans le support permettant de saisir les informations mais également dans l'interlocuteur en distinguant le responsable qualité du responsable des études.

Cette vigilance est d'autant plus importante si la petite taille de l'organisation conduit à un cumul des deux fonctions.

### **b. Ne pas valoriser la criticité de signalement**

Si l'implémentation du processus de collecte des incidents ne permet pas de saisir une fréquence d'impact et/ou de saisir la nature de l'impact (financier notamment), il ne sera pas possible au responsable qualité de définir une criticité précise. Ni à lui, ni à tout autre acteur n'ayant pas la visibilité du problème.

S'il a l'information indiquant quelle exigence est concernée, il pourra peut-être prendre la criticité de cette dernière par défaut... pourvu que l'analyse des risques ait été effectivement formalisée et maintenue pour l'application en question. Ce qui n'est pas certain.

 Notez le rôle de la matrice des criticités des risques et exigences produit dans le processus de qualification des incidents : elle devient un outil décisionnel a posteriori de son utilité première dans la définition d'un périmètre de test et cela après le projet qui a conduit à la réalisation du logiciel.

Le risque est ici de commettre des confusions voire d'aggraver les conséquences d'un incident critique.

L'observation est en effet correctement diagnostiquée : c'est une anomalie. Cependant, l'erreur commise sur la criticité pourra :

- Soit déclarer comme urgent un incident qui ne l'est pas - auquel cas mettre inutilement la pression sur une MOE tout en décrédibilisant la MOA.
- Soit demander une correction simple là où il y a urgence et donc laisser un grave problème affecter l'environnement de production en livrant la correction trop tard.

### **c. Ne pas détecter une dépendance**

La principale difficulté du processus de récolte des anomalies en production consiste en la détection des dépendances. Celles-ci peuvent être organisationnelles (tenir un planning) mais également techniques.

Ce dernier aspect est problématique car la dépendance technique sera fréquemment détectée trop tardivement. Ainsi, un processus de correction sera enclenché, et l'on détectera pendant son développement qu'il manque un prérequis ailleurs qui n'a pas été anticipé.

En résulte une difficulté d'organisation entre des entités MOE et MOA avec des impacts dans les planifications des projets.

Cette difficulté sera accrue si en plus, la dépendance concerne deux logiciels dans deux domaines distincts de deux MOA : un pilotage transverse est alors nécessaire, ce qui dans le cadre d'un patch est très problématique car la transversalité amène de la lourdeur dans un contexte demandant de la réactivité.

#### **d. Mal implémenter le processus dans un outil**

L'usage d'un outil inadapté ou d'un outil adapté mais mal paramétré, est aussi un risque pour le processus de déclaration des incidents.

Il est bien entendu très intéressant qu'un outil de gestion documentaire implémente et automatise le workflow de qualification des observations.

Mais à condition que :

- Son paramétrage corresponde à l'implémentation précise du workflow que nous avons représenté dans le schéma précédent - ou bien d'une version adaptée au contexte organisationnel, mais selon les principes que nous avons exposés.
- Son usage par les utilisateurs ne soit ni fastidieux ni chronophage (risque de rejet de l'outil).

Rien n'est pire qu'un outil mal utilisé.

 Cas réel : une société utilisait Mantis pour cet usage, outil qui avait été imposé par la DSI qui l'utilisait pour ses propres recettes techniques. Le workflow implémenté pour la MOA ne proposait pas de saisir l'environnement dans lequel l'observation était faite. Conséquences : impossible de savoir si l'incident était en production ou pas, donc d'estimer l'urgence du correctif. Par ailleurs, cet outil ne comportait pas non plus la notion de fréquence d'impact et d'impact financier. Il a été observé qu'une vente en ligne a été médiocre à cause d'un incident qui est ainsi "passé à la trappe" en raison de permissions mal définies.

#### **e. Abandonner l'usager, ne pas le former**

Ne pas gérer les incidents de production ou les demandes d'évolution revient à abandonner l'utilisateur sans lui offrir la possibilité de signaler les contraintes qu'il vit au quotidien.

Il s'en suivra une perte de confiance en la DSI et la MOA : l'application est livrée, point final. Et pour le reste ? Se débrouiller.

Cette démarche n'est pas la bonne : il faudra toujours veiller à maintenir un lien, une continuité dans la démarche qualité.

La déclaration d'incident est aussi l'occasion de mesurer si les outils de l'entreprise sont correctement assimilés par les utilisateurs. Une redondance d'observation sans objet peut révéler un déficit en formation notamment.

Le responsable qualité et/ou le responsable des études de la MOA étant proches du métier, l'identification de ce problème devient possible.

# Les anomalies remontées par la maîtrise d'ouvrage

Lors d'une recette purement fonctionnelle conduite par le métier, des anomalies peuvent être détectées. Nous détaillons ici comment gérer ce processus.

## 1. Principes organisationnels

### a. Avec ou sans équipe de recette technique ?

Avant toute mise en place d'un processus de collecte des observations des recettes fonctionnelles, il faudra se poser la question du contexte même du projet.

Il est bien évident qu'en l'absence d'une structure en charge des recettes techniques, c'est la MOA qui s'y substitue en réalisant des recettes à la fois techniques et fonctionnelles.

Dès lors, peu importe l'étape du projet où l'anomalie est détectée (seul l'environnement de recette change peut-être entre recette technique et recette fonctionnelle, pas ses acteurs).

Dans un tel cas, nous vous renvoyons à la section relative à la remontée des observations dans une recette technique pour implémenter ce processus.

En revanche, dès lors qu'il existe une équipe de recette technique, le processus de collecte des observations en recette fonctionnelle diffère.

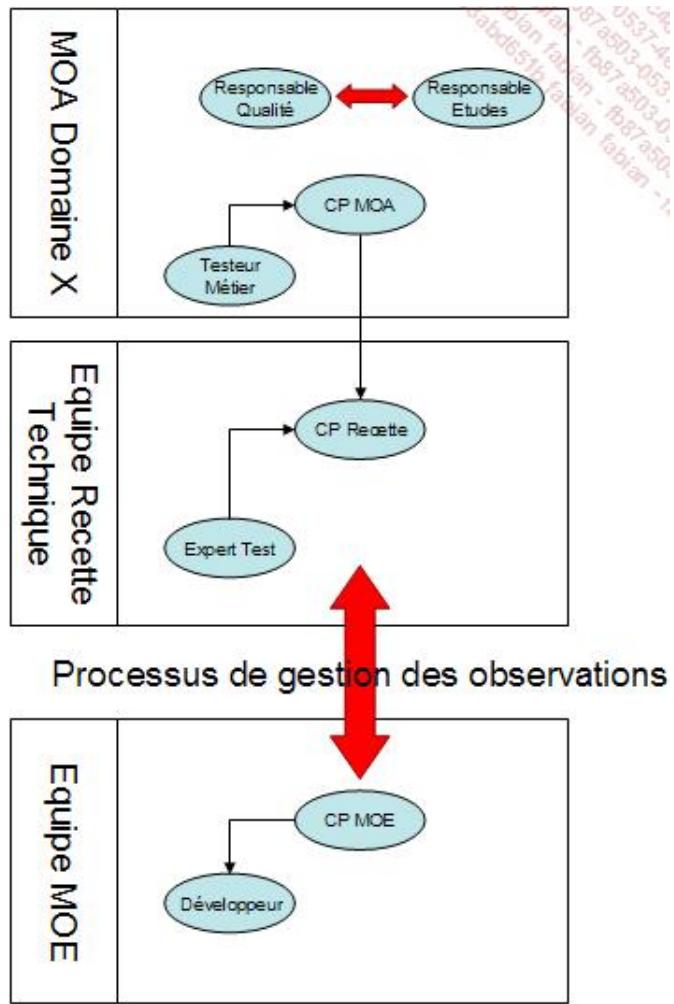
### b. Définir un support de collecte

Dès lors que l'équipe de recette technique existe et qu'elle est distincte de l'équipe des recettes métier, nous préconisons que les observations soient saisies dans le même outil de recette en définissant toutefois deux nuances :

- L'observation doit clairement montrer qu'elle est réalisée dans un environnement de recette métier. Il ne suffit pas de mentionner que le rapporteur n'appartient pas à l'équipe de recette technique mais à la MOA pour comprendre qu'il s'agit d'un événement détecté en recette fonctionnelle.
- Le processus de qualification de l'observation détectée en recette fonctionnelle n'est pas exactement le même qu'en recette technique :
  - Le rapporteur appartient à la MOA et non à l'équipe de recette technique.
  - L'application testée n'est pas nécessairement dans la même version que celle à disposition de la recette technique - le plus souvent il s'agit d'ailleurs d'une version plus ancienne, l'équipe de recette technique étant en train de tester une version plus récente.
  - L'équipe de recette technique vérifiera que la correction de l'anomalie détectée par la MOA a été corrigée avant sa livraison en recette fonctionnelle qui elle-même fera un deuxième contrôle : il y a double validation.

### c. Formalisation du processus

Le schéma suivant formalise de manière simplifiée le processus de gestion des observations en provenance d'une MOA relativement à une équipe de recette technique qui serait intermédiaire avec la MOE.



Son principe est très simple : un chef de projet MOA pilote une recette fonctionnelle. Vous noterez qu'il pourra être aussi le responsable de la qualité et/ou le responsable des études selon la taille de la structure.

Mais il conviendra par défaut de distinguer les fonctions de collecte des incidents et les améliorations de la production de la fonction qui consiste à vérifier une application en cours de validation et fournie par une MOE directement et/ou une équipe de recette technique ayant préalablement validée le logiciel.

Dans notre processus, un testeur métier (qui peut donc être membre de la MOA ou un utilisateur final bien réel), remonte une observation au CP MOA du projet suite à un test réalisé dans un environnement de test métier : nous sommes dans le cadre d'un projet et pas dans celui de l'utilisation usuelle de l'outil en production.

Le CP MOA fait alors office de filtre et de trieur : il supprime les doublons laisse de côté les évolutions (il pourra d'ailleurs les spécifier ultérieurement) pour se focaliser uniquement sur les aspects anomalistiques.

Il ne faudra pas oublier que la recette fonctionnelle se focalise sur certains aspects métier très précis que la recette technique n'a pas permis de vérifier. Ces recettes sont donc très ciblées.

Le fait qu'une équipe de recette technique élimine au préalable un maximum d'anomalies permet à l'équipe de recette fonctionnelle de limiter son périmètre à la validité des exigences métier.

Ainsi, si le CP MOA avait à rejeter l'application, il le ferait logiquement auprès de l'équipe de recette technique qui représente "son fournisseur" en informant le CP MOE de cette décision simultanément.

Cette organisation est en effet ternaire avec trois chefs de projet, le CP MOA représentant l'arbitre des deux autres.

Il en découle que le processus des observations émises par une MOA en cours de recette fonctionnelle se greffe logiquement en amont du processus des observations émises par l'équipe de recette technique elle-même, et non pas de manière parallèle et indépendante avec la MOE.

-  Bien que théoriquement possible, nous déconseillons une organisation parallèle des processus de remontée des observations entre MOA et équipe de recette technique. Il est en effet préférable que les experts en recette technique soient informés des difficultés rencontrées par la MOA pour qu'ils puissent les anticiper ultérieurement.

La distinction entre une observation issue de la MOA et de la recette viendra ensuite dans l'environnement et le rapporteur.

#### **d. Le lotissement des corrections : préparer les vérifications**

Dans une organisation à double équipe, les recettes techniques précèdent les recettes fonctionnelles mécaniquement.

Ainsi l'équipe de recette technique recevra successivement plusieurs versions de l'application de la MOE... et ne livrera à la MOA pour recette métier que les versions qu'elle aura validées : il est évident qu'une version non validée n'est pas transmise.

Il en découle que si la MOE transmet N fiches de livraison à l'équipe de recette technique, alors cette même équipe transmettra à la MOA la fusion de toutes les fiches de livraison qu'elle a validées.

Concrètement, voici ce qui se produira :

- Toute évolution mentionnée dans les N fiches sera indiquée comme livrée à la MOA en vue d'une recette fonctionnelle.
- Toute anomalie détectée par le métier sera indiquée comme corrigée et à vérifier par la MOA.
- Toute anomalie détectée par l'équipe de recette technique et ayant été corrigée ne sera pas indiquée à la MOA sauf exception de criticité justifiant qu'une double vérification soit nécessaire.

Ainsi, la MOA n'a pas une visibilité des versions abandonnées de l'application qui ont été refusées par l'équipe de recette technique.

Ce processus de livraison par fusion des fiches de livraison permet ainsi de dégager le périmètre des tests de l'équipe de recette métier, notamment dans les chantiers de maintenance.

Il est évident en revanche que l'homologation fonctionnelle dans un chantier de release différera légèrement du chantier de maintenance : une première livraison sera nécessairement associée à un périmètre de recette fonctionnelle préalablement préparé.

Alors que pour un chantier de maintenance, les corrections à vérifier et petites évolutions arrivent "au fil de l'eau" et surtout lorsque l'équipe de recette technique donne son aval. Il est d'ailleurs possible que si deux correctifs s'avèrent pour l'un correct et pour l'autre incorrect, la livraison en recette métier soit faite malgré tout.

## **2. Les risques**

### **a. Ne pas indiquer clairement que l'anomalie a une origine métier**

Le processus de déclaration des observations métier se superposant avec celui de l'équipe de recette technique, il

en découle un risque de confusion.

Il est donc impératif que l'environnement de détection de l'observation soit mentionné.

Toutefois, un test métier pourra être mené dans le même environnement que celui de l'équipe de recette technique (parfois, le choix n'est pas possible). Ceci signifie qu'il faut en réalité saisir une information spécifiant explicitement "Observation réalisée en recette métier" : l'environnement ne suffit pas.

Bien entendu, l'identité du rapporteur pourra mettre sur la voie si cette information est absente, mais nous vous conseillons d'être le plus clair possible sur cet aspect.

## b. Mal implémenter l'outil

Dès lors qu'un outil de collecte des observations est mis en place, il ne faudra pas perdre de vue le fait qu'il est susceptible d'implémenter :

- Un seul processus si la MOA est seule sans équipe de recettes techniques.
- Deux processus sinon, l'un s'agrémentant à l'autre.

Dans ce deuxième cas, le risque de commettre des erreurs de paramétrage et notamment dans la définition du workflow des observations, est plus important.

 Cas réellement observé : le double processus des observations provenant de la MOA et d'une équipe de recette technique était implanté dans Mantis. Le métier avait ainsi accès aux volumétries des anomalies détectées par l'équipe de recette technique ce qui projeta une mauvaise image de qualité du logiciel. En fait, il n'y avait pas la possibilité de filtrer les anomalies en provenance des deux structures...

## c. Mal gérer les retours

Du fait d'avoir les observations de la MOA et de l'équipe de recette technique gérées dans le même processus, il en découle tout naturellement que :

- Les observations faites par l'équipe de recette technique font l'objet de correctifs vérifiés par cette même équipe.
- Les observations faites par l'équipe de recette fonctionnelle font l'objet de correctifs vérifiés par les deux équipes successivement.

De ce fait, il existe un risque de noyer le périmètre des corrections réalisées pour le métier en transmettant à la MOA des observations qui ne la concerne pas... et d'oublier de transmettre les observations qui la concernent.

D'où l'intérêt de tracer explicitement une observation rapportée dans le cadre d'une recette métier. En filigrane se pose aussi la question de la visibilité des observations par l'équipe MOA, notamment celles qu'elle n'a pas remontées.

# Les anomalies détectées lors d'une recette technique

La présente section intéresse également les MOA pour les recettes fonctionnelles dès lors qu'il n'existe pas d'équipe de recette technique dédiée au projet et que la MOA s'y substitue.

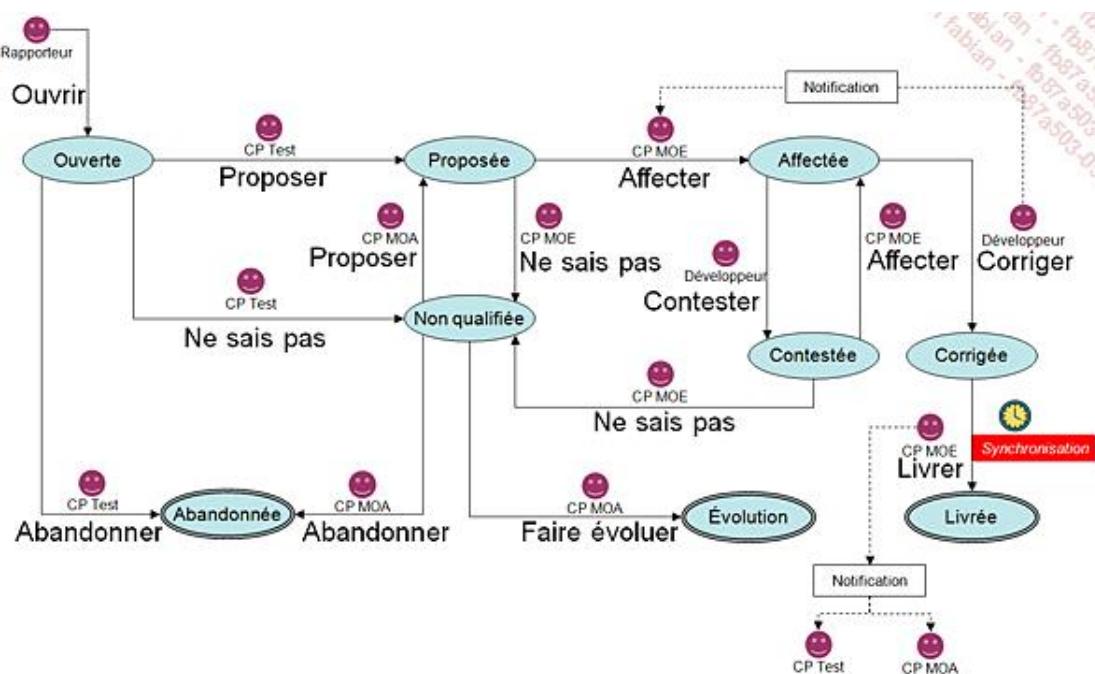
## 1. Principes organisationnels

### a. Formaliser le workflow documentaire

Nous proposons le workflow documentaire suivant pour modéliser le processus de qualification d'une observation entre une MOA, une MOE et une équipe de recette technique.

Les principes de base de ce workflow sont :

- Que le CP Test fait office de trieur des observations en amont (pour éviter redondance et observation sans objet).
- Que le CP MOA joue le rôle d'arbitre entre le CP Recette et le CP MOE lorsqu'il y a désaccord sur la nature d'une observation.
- Qu'il n'y a pas de cycle entre CP Test et CP MOE : le workflow ne doit pas devenir un tchat dans lequel chacun se renvoie la balle.
- Que les correctifs une fois réalisés sont synchronisés dans un même lot, une même livraison.



Ainsi, un rapporteur ouvre une observation à destination du responsable de la recette technique, un chef de projet test noté CP Test.

Ce rapporteur peut être un membre de l'équipe de recette technique ou bien le CP MOA qui ajouterait ainsi une observation détectée en recette fonctionnelle.

Le CP Test qui reçoit l'anomalie pourra alors :

- Abandonner l'observation si elle est redondante, sans objet ou bien fait référence à une anomalie qui ne peut être corrigée. Il y a bien sûr un motif décrivant l'abandon qui est par ailleurs un état terminal. Éventuellement, une

observation abandonnée pourrait repasser au statut Ouverte.

- Ne pas savoir statuer sur la qualification de l'observation, auquel cas elle sera soumise au CP MOA car non qualifiée.
- Proposer l'observation au CP MOE car probablement d'origine anomalistique.

La réponse du CP MOE est alors simple :

- Il admettra que l'observation est une anomalie et demandera à un développeur la correction en l'affectant.
- Ou bien il ne sera pas d'accord avec le CP Test et ne pourra que la transmettre pour arbitrage au CP MOA.

Lorsqu'un développeur reçoit une demande de correction, deux choix s'offrent à lui également :

- Réaliser la correction qui demandera une synchronisation d'un ensemble de correctifs et d'évolutions pour livraison.
- Contester la demande de correction auprès du CP MOE, celle-ci pouvant être motivée par l'impossibilité de réaliser la demande (manque de temps ?) ou bien un refus de la nature anomalistique de l'observation suite à l'analyse technique qu'il mène.

Dès lors, le CP MOE pourra réaffecter la demande de correction à un autre développeur ou demander l'arbitrage de la MOA.

Lorsqu'il existe des observations non qualifiées, le CP MOA est le seul à pouvoir statuer de la suite à donner. Nous conseillons de mener alors des réunions de qualification.

Une bonne équipe de recette technique sera le plus souvent pertinente dans ses remontées tant et si bien que l'arbitrage doit nécessairement concerner un petit volume d'observations à la marge.

L'arbitrage entre CP Test et CP MOE par la MOA permet de rétablir un équilibre de pouvoir favorable au projet. S'il y a désaccord, l'arbitre tranche. S'il y a une interrogation, des difficultés à reproduire l'observation..., elles seront traitées en commission d'arbitrage également.

 Nous préconisons au CP Test de mesurer le taux de pertinence des observations remontées par son équipe, c'est-à-dire le ratio des observations reconnues par la MOE comme étant une anomalie par le nombre total d'observations. Le taux de pertinence doit être d'au moins 80 %.

Vous noterez également que les notifications sont en petit nombre : seules les corrections et les livraisons sont notifiées.

## b. Définir un support de collecte : outil ou simple fichier ?

Nous déconseillons l'usage de fichiers gérés manuellement pour implémenter ce processus, tout du moins si vous comptez utiliser un fichier par observation.

En revanche l'usage d'un unique fichier Excel pour définir un journal des observations est pratique dès lors qu'une observation peut être consignée dans une seule ligne d'un onglet Excel.

Sa structure comportera alors :

- Un identifiant unique référençant l'observation.
- La typologie d'ouverture : Recette fonctionnelle / Recette technique / Incident de production.
- L'identité du rapporteur.

- L'état de qualification :
  - Observation ouverte.
  - Observation proposée.
  - Observation non qualifiée.
  - Observation abandonnée - État terminal.
  - Anomalie affectée.
  - Anomalie corrigée.
  - Anomalie contestée.
  - Correction livrée - État terminal.
  - Évolution - État terminal.
- Le motif de clôture :
  - Observation abandonnée : Sans objet, Redondance, Anomalie ne pouvant être corrigée, Anomalie non reproduite...
  - Correction livrée : préciser la version de livraison de l'application corrigée, des commentaires...
  - Évolution : Réalisable, Non réalisable, À étudier pour statuer...
- L'application, sa version et l'exigence concernées - idéalement sa criticité.
- La référence du test exécuté (optionnelle).
- La description de l'observation.
- Des pièces jointes, des commentaires additionnels.

Bien évidemment la mise en place d'un outil correctement paramétré est idéale. Mais attention : nous vous renvoyons immédiatement aux sections qui suivent et vous conseillons de prendre un maximum de précautions à ce sujet !

### c. Interactions avec la collecte des demandes d'évolution

Le processus de remontée des observations en recette technique est susceptible de détecter de nouveaux besoins.

Nous rappelons que la confusion entre anomalie et évolution doit être évitée, les premières présentant un risque, les deuxièmes une opportunité.

Nous insistons donc sur le fait que le statut d'une observation en évolution est ici terminal car il met fin au processus de qualification d'une observation.

Dès lors, à l'issue de ce processus, il est statué de l'action qui suivra car une évolution détectée ne fait pas nécessairement l'objet d'un développement immédiat.

Cette précaution prise, les informations recueillies pour décrire l'observation seront complétées pour initialiser le processus de demande d'évolution à destination du responsable des études, **c'est-à-dire à l'extérieur du projet**.

En effet, la détection d'un nouveau besoin ne signifie pas que le CP MOA peut prendre la responsabilité de le mettre en œuvre. Bien sûr, pour une petite maintenance peu coûteuse il pourra assumer la responsabilité de décider seul d'élargir le besoin au-delà du périmètre initialement prévu.

Mais si ce nouveau besoin fait l'objet d'une dépendance technique ou qu'il s'avère plus complexe (et plus coûteux) à mettre en œuvre, alors il devra se tourner vers un décideur.

Bien évidemment, si le CP MOA est aussi le responsable des études, il y a de fortes chances que toutes évolutions détectées soient immédiatement mises en œuvre.

Quoi qu'il en soit, ne perdez pas de vue qu'il y a une séparation des processus.

#### **d. Gérer les points de blocage**

L'instance d'arbitrage que nous proposons de mettre en place a pour objectif de se focaliser sur toutes les observations pour lesquelles des spécifications sont imprécises, des reproductibilités douteuses, des bizarries en tout genre.

Conjointement, notre processus permet aussi de détecter qu'une MOE est submergée et n'a pas le temps d'effectuer les correctifs.

Retenez que dans 80 % des cas, les observations étant pertinentes, la MOE suivra la pré-qualification en anomalie de l'observation : il y a accord par défaut.

En revanche, les 20 % restants constituent l'essentiel qui mérite de statuer. Ce sera :

- des anomalies liées à une non-compatibilité de deux configurations matérielles,
- une lacune dans les spécifications (comme par exemple l'impossibilité de consulter une donnée),
- une dépendance technique manquante,
- des difficultés pour la MOE à reproduire une anomalie ou à comprendre le métier,
- etc.

#### **e. Gérer les retours**

Une fois les observations clairement identifiées comme des descriptions d'anomalies, le CP MOE distribue leurs corrections en les affectant à son équipe de développeurs.

Le CP MOE a pour interlocuteur le CP Test et il doit prioritairement :

- Faire corriger toute anomalie bloquante, c'est-à-dire une anomalie remettant en question la jouabilité des tests.
- Participer aux réunions de qualification avec le CP MOA et le CP Test afin de traiter les points sur lesquels il n'est pas d'accord avec l'équipe de test technique.
- Mettre le CP Test en visibilité des anomalies qui ont déjà été corrigées et prévoir la livraison des lots de correctifs en se rapprochant au plus près du CP Test (il est hors de question d'écraser une application en cours de recette !).
- Rédiger les fiches de livraison et livrer les applicatifs.
- Affecter les correctifs aux membres de son équipe.

La gestion des retours de la MOE par le CP Test permet donc de clairement comptabiliser les observations pour lesquelles des corrections sont attendues.

Il s'en dégage alors un taux de retour égal au nombre d'anomalies déclarées corrigées sur le nombre total d'anomalies (incluant celles aux états Affectée et Contestée).

 Nous recommandons également de suivre le nombre d'observations proposées chaque jour afin de vérifier que le flux est constant. Il est très peu productif de communiquer 2 anomalies un jour puis 20 le lendemain et 3 le jour d'après.

## 2. Les risques

### a. Mal paramétrier un outil (encore et toujours)

La formalisation d'un workflow de gestion des observations au cours d'une recette technique est certainement le point le plus délicat à résoudre dans les échanges avec une MOE.

Généralement issus du monde du développement, les outils mis en place par l'équipe de recette technique sont souvent connus des développeurs. Ce qui ne veut pas dire que tout ce petit monde sait correctement définir un workflow.

Malheureusement, l'expérience montre d'ailleurs que la règle serait de mettre en place un processus différent de celui que nous proposons, avec un certain nombre d'erreurs récurrentes que nous avons listées.

Pour commencer, les libellés des états et les transitions sont souvent mal définis et ne correspondent pas à un graphe d'états finis déterministe :

- Les libellés des états et des transitions sont confondus, ne sont pas concis, voire sont trop nombreux ("À analyser", "À corriger", "Commentaire"...).
- Des transitions sans utilité fonctionnelle sont définies.
- Les permissions sont mal gérées relativement aux états de l'observation
- Des champs importants sont absents.
- Des préconisations d'utilisation inappropriées sont mises en place (déplacement manuel d'observation d'un projet MOE vers un projet MOA par exemple pour gérer la séparation organisationnelle !).
- Les développeurs ayant mis l'outil en place, ils s'arrogent les droits d'administration et contournent les règles implémentées.
- Évidemment, la nature de l'observation est systématiquement "Anomalie" alors que cette nature est censée être une qualification terminale du processus.
- L'outil est parfois détourné pour gérer les demandes d'évolutions aussi !
- Etc.

Le risque principal est alors double puisque d'une part les données pourront devenir totalement incohérentes et il faudra passer en revue chaque observation pour redresser le cap : chacune doit être assignée à celui qui doit mener une transition précise. D'autre part, la transformation de l'outil en fera alors un outil de communication, un tchat, qui fera exploser une activité chronophage (d'autant que souvent des mails de notification sont générés à outrance et viennent engorger les boîtes aux lettres inutilement).

Notamment, dès lors qu'un développeur ou un CP MOE est sollicité pour qualifier une observation d'anomalie, il est fréquent que la réponse puisse être "Pas une anomalie" alors qu'en réalité notre ingénieur est tout simplement débordé et ne peut pas admettre d'avoir une tâche supplémentaire à conduire.

S'en suit que la qualification est alors biaisée et au'un outil de recette contiendra ainsi d'éternels "pina-ponas" qui

n'apportent rien là où une réunion de qualification arbitrée permet de rapidement dénouer des fils.

### **b. Doubler le processus de livraison dans l'outil**

Le lecteur doit bien comprendre ici que le processus de qualification d'une observation s'arrête dès lors que sa nature est statuée : anomalie, observation ou évolution.

Si à son ouverture une observation est ce qu'elle est (un signalement) elle restera dans ce statut ou décrira une anomalie voire une évolution.

Mais en aucun cas le workflow des observations ne représente les correctifs eux-mêmes.

Aussi, définir des statuts de livraison et poursuivre le workflow des observations clôturées au-delà de "Livrée" n'a pas de sens car cela revient à gérer un autre processus dans l'outil de gestion des observations : les livraisons.

La livraison d'un logiciel par une MOE (généralement le CP MOE est en charge de cela) consiste à rédiger un document et le diffuser à destination du responsable des recettes techniques et fonctionnelles (CP Test et CP MOA donc). Conjointement, une opération technique sera réalisée par laquelle les sources, exécutables et fichiers de paramétrage sont "packagés" et mis à disposition dans un outil spécifique de dépôt et synchronisation.

Aussi, lorsqu'une anomalie est corrigée, le processus normal consiste à attendre qu'un lot de corrections soit constitué afin de toutes les clôturer ensemble à l'état "Correction livrée" pour figer une version de l'application modifiée.

Dès lors, il est impératif d'implémenter soigneusement le processus cadrant les livraisons et surtout ne pas réaliser cette implémentation dans l'outil de gestion des observations - ce n'est pas sa fonction.

Malheureusement, il est fréquent de constater que le workflow se poursuit notamment pour renvoyer l'observation à son rapporteur initial, une démarche très discutable d'ailleurs.

En effet, le rapporteur de l'anomalie pourra être étranger à l'entreprise et représenté par un "ayant-droit" qui n'est pas destiné à vérifier des corrections. Ce rapporteur peut aussi tout simplement être absent (congé, démission...) et une vérification de correction doit pouvoir être réalisée par n'importe qui d'autre.

Il n'y a donc aucun intérêt à remplacer le processus de livraison ou à le doubler par une implémentation du workflow des observations.

### **c. Ne pas être pertinent**

La qualité du travail d'une équipe de test se mesure à son aptitude à faire des observations pertinentes.

Nous avons déjà évoqué comment mesurer le taux de pertinence d'une équipe de test.

Pour commencer, la pertinence consiste à avoir une politique d'identification des doublons très rigoureuse ce qui signifie que le CP Test et toute son équipe doivent être capables de rapprocher très vite les observations pour éviter la redondance.

Se pose ensuite la difficulté pour le CP Test de savoir les observations pour lesquelles il peut déjà demander de ne plus ouvrir de fiche. Prenons l'exemple d'un paramétrage incomplet : le CP Test pourra admettre qu'une observation soit remontée sur le sujet le premier jour.

Mais il communiquera alors avec son équipe pour rappeler que tel comportement de l'application testée n'est pas une anomalie pour éviter de remonter les observations d'un même phénomène en surnombre.

La gestion des configurations matérielles a également un impact ici puisqu'en général une anomalie présente dans une configuration matérielle est généralisable à toutes, donc offre une redondance possible.

Enfin, la compréhension du métier lorsqu'elle n'est pas maîtrisée amènera à l'ouverture d'observations démontrant une faible maturité de l'équipe de recette technique sur certains aspects fonctionnels.

Cela aura pour effet de discréditer la perception de la performance des testeurs auprès de la MOE.

➤ Voici une situation réellement observée : dès lors qu'une anomalie était déclarée par un testeur précis, la MOE la classait sans suite car elle partait du principe que le testeur en question n'était pas capable de comprendre l'application. Les a priori l'emportaient sur la démarche à cause d'un passif.

Le CP d'une recette technique a donc tout intérêt à évaluer quotidiennement le taux de pertinence de son équipe, indicateur qu'il pourra d'ailleurs fournir au moment du bilan des tests.

➤ Se donner la capacité à faire du prévisionnel : si votre taux de pertinence est de 80 % les deux premiers jours et que vous avez déjà remonté 20 observations, alors si votre recette dure 10 jours, vous pouvez déjà annoncer une volumétrie prévisionnelle des anomalies : 10 observations par jour, donc 100 observations en 10 jours, soit 80 anomalies à corriger.

#### d. Émettre un flux inconstant, mal communiquer avec la MOE

Les échanges entre développeurs et équipe de test technique peuvent être parfois houleux. Pour pallier tout risque conflictuel, trois attitudes doivent être adoptées :

- Toujours être factuel et neutre, notamment lors des réunions d'arbitrage ou lors des échanges informels avec les développeurs. Éviter toute subjectivité du genre "l'application est médiocre" et préférer "80 % des scénarios de test sont bloqués : le niveau de testabilité de l'application ne nous permet pas de travailler normalement".
- Être partenaire de la MOE et non adversaire, se rapprocher d'elle et l'assister à la reproduction des anomalies (surtout lorsqu'elle est débordée !).
- Mesurer la volumétrie des observations émises chaque jour et réguler le flux qui en découle de sorte qu'il alimente de manière constante la MOE.

Il revient au CP Test d'être irréprochable sur ces aspects.

# Le pilotage budgétaire

Dans la présente section, nous aborderons une tâche transverse aux projets de recette : son pilotage par la charge.

Nous n'entrerons pas dans des considérations financières faisant intervenir les coûts des ressources humaines et techniques : ce livre n'est pas à destination des directeurs de projet.

Nous n'aborderons pas non plus l'évaluation de charge et son suivi pour les tests unitaires, estimant que cette tâche fait partie intégrante de l'évaluation et du suivi de la phase de réalisation par le CP MOE.

Pour mémoire, nous rappelons que nous avons préconisé dans le chapitre Organiser les tests unitaires une fourchette de 20 à 33 % de la charge de développement pour estimer l'effort nécessaire et suffisant à consacrer aux tests unitaires.

Nous allons détailler en revanche :

- Le pilotage d'une recette technique dans un chantier de release.
- Le pilotage d'une recette technique dans un chantier de maintenance.
- Le pilotage d'une recette fonctionnelle.

## 1. La recette technique d'un projet de release

Nous avons décomposé un projet de recette en quatre tâches (ORGANISER, PRÉPARER, EXÉCUTER, CLÔTURER) décomposition à laquelle nous ajoutons PILOTER bien évidemment.

Logiquement, évaluer la charge globale consiste à évaluer celle de chacune de ces tâches.

### a. Évaluer la charge de l'organisation

La tâche ORGANISER consiste à produire principalement deux livrables : le plan de test et le dossier d'organisation.

Sa charge est logiquement consommée dans son intégralité par le CP Recette.

Cette activité réclame une charge fixe plus ou moins indépendante de la taille du projet qui pourra aller de 1 jh à 10 jh. Elle sera d'autant plus chronophage que vous aurez des entretiens à conduire si vous devez effectuer une analyse de risques et une collecte d'exigences.

Mais des considérations techniques peuvent aussi vous demander plus de temps, par exemple une réflexion sur la constitution pertinente d'une plateforme de tests. Nous songeons ici notamment aux recettes techniques des applications mobiles qui nécessitent une réflexion poussée sur les terminaux visés, leurs systèmes d'exploitation, langue, typologie de clavier, d'interfaces...

 Pour les recettes des applications mobiles, n'hésitez pas à prévoir une marge de manœuvre en surévaluant la charge au besoin, notamment pour le cas où vous aurez besoin de faire appel à un expert en mobilité dans votre démarche.

### b. Évaluer la charge de la préparation

La tâche PRÉPARER peut être décomposée en cinq sous-tâches :

- La charge de rédaction des fiches de test pour la partie technique de la recette.
- La charge de rédaction des scénarios pour la partie fonctionnelle.
- La charge de construction du jeu d'essai et des données prérequises.
- La charge de formalisation du jeu d'essai et des données prérequises.
- La charge de saisie des données prérequises dans une base de données ou tout autre support physique.

La charge de rédaction des fiches de test sera fonction de leur nombre et d'un niveau de complexité. Une fiche de test pourra consigner par exemple, tous les contrôles techniques d'un écran d'une exigence fonctionnelle, auquel cas vous pourrez considérer l'échelle suivante :

- Qu'un écran présentant des informations sans saisie est de complexité faible et demande 15 minutes de rédaction pour un concepteur.
- Qu'un écran demandant une faible saisie est de complexité moyenne et demande 30 minutes de rédaction pour un concepteur.
- Qu'un écran demandant plusieurs saisies est complexe et demande une heure de rédaction environ pour un concepteur.

Dès lors, la charge d'évaluation de la préparation consiste à compter le nombre d'écrans de votre application.

Cette démarche ne s'appliquant pas aux imports, exports et traitements de masse, ces derniers seront évalués différemment : c'est la taille de la donnée testée et le nombre de contrôles réalisés sur celle-ci qui déterminent la charge nécessaire.

La charge de rédaction des scénarios fonctionnels quant à elle est tributaire du nombre d'exigences fonctionnelles, démultipliées par le nombre de jeux d'essai nécessaires par exigence. Partez du principe qu'un scénario fonctionnel nécessite une à deux heures de charge de rédaction selon la complexité du scénario car le recul nécessaire pourra demander de lire les spécifications, de faire éventuellement une recherche pour calquer le scénario au plus près d'un cas réel d'utilisation.

Les scénarios fonctionnels doivent être très réalistes raison pour laquelle il vous faudra conjointement construire et rédiger un cahier décrivant les données.

Les charges nécessaires à la construction et la formalisation détaillée d'un jeu de données complet sont les deux points les plus délicats à estimer.

Pour la partie "jeux d'essai", la charge nécessaire est logiquement proportionnelle au nombre d'exigences fonctionnelles à vérifier pourvu que l'exigence demande une saisie : il y a autant de jeux d'essai que de cas de test.

Pour la partie "données préexistantes" en revanche, vous ne pourrez pas estimer le taux de mutualisation. Au plus, vous aurez donc besoin d'autant de données prérequises qu'il y a d'exigence... au mieux une seule donnée pourra être utilisée par tous les scénarios, comme par exemple un compte utilisateur.

C'est donc le contexte du projet qui vous permettra d'estimer votre besoin budgétaire ici.

Reste alors à estimer la charge de saisie des données dans le système qui est tout aussi contextuelle au projet :

- Si les données sont extraites d'un autre environnement, la saisie se résumera peut-être à un simple export-import pourvu qu'il existe un traitement permettant ce flux : la charge de saisie est alors quasi nulle (mais réservez-vous une marge d'une journée au moins pour vérifier le résultat de l'opération).
- Si les données peuvent être générées automatiquement, par exemple avec des macros (développées sous Visual

Basic, enregistrées sous Selenium...), la charge de saisie pourra être quasiment nulle aussi... mais peut-être qu'une charge sera nécessaire pour rédiger les scripts ?

- Si les données sont définies manuellement à l'aide d'un outil indépendant de l'application, vous devrez estimer le volume des données à saisir pour connaître la charge nécessaire. En général, si le volume est trop important, la saisie manuelle apparaîtra rapidement comme chronophage et cette solution sera alors écartée... à moins qu'il soit impossible de faire autrement.
- Si les données sont définies à l'aide de requêtes SQL par exemple ou bien constituées dans un ensemble de fichiers XML, leur saisie relève d'une démarche technique pouvant s'avérer plus ou moins complexe et demandant une vérification. Dans un tel cas, la charge sera plus élevée que pour le cas précédent. Se pose aussi la question d'utiliser des outils permettant une relative assistance à la saisie, comme par exemple Notepad++, pour vérifier la syntaxe XML.
- Etc.

Retenez donc qu'il vous sera très difficile d'évaluer avec précision les charges relatives à tout ce qui touche aux jeux d'essai et à l'environnement des données dans un chantier de release, comparativement à la charge de rédaction de cahier de recette qui est beaucoup plus précise.

### c. Évaluer la charge de l'exécution

La charge d'exécution d'une recette technique doit être décomposée comme suit pour être estimée :

- Pour la première itération :
  - Comptez 10 minutes pour tester une fiche de test de complexité faible.
  - Comptez 15 minutes pour tester une fiche de test de complexité moyenne.
  - Comptez 30 minutes pour tester une fiche de test de complexité forte.
  - Comptez 10 minutes pour exécuter un scénario fonctionnel (si besoin revoyez cette estimation au cas par cas selon la difficulté du scénario).
  - Calculez alors la charge de passage théorique de chacune des N configurations matérielles. N'oubliez de prendre en compte le nombre de langues de l'interface graphique au besoin si ce paramètre permet de définir des configurations matérielles logiques.
  - **Puis estimatez qu'il y aura 15 % des tests bloqués** pour cause d'anomalie conduisant à réduire cette charge d'exécution pour la première itération.

Vous noterez immédiatement que lors d'une recette technique en release, nous partons du principe que l'application ne sera pas totalement testable. S'autoriser 15 % d'erreur à ce stade est un ratio "humainement acceptable" et nous le proposons car en moyenne le choix de ce taux nous semble pertinent.

Mais vous devrez cependant adapter ce taux au contexte. Par exemple, si vous savez que la MOE manque de maturité sur la technologie, peut-être faudra-t-il vous attendre à une proportion de tests bloqués plus forte ? 20 % ? 30 % ?

Retenez-en le principe avant tout : une recette technique nécessite une marge de manœuvre pour estimer la charge d'exécution en première itération.

- Pour la deuxième itération :
  - Tout d'abord, vous aurez des corrections d'anomalie à vérifier. Comptez 100 corrections vérifiées par jour pour une personne.

- Estimez la volumétrie future des anomalies, par exemple, 1 par écran. Ou bien 10 par jour d'exécution de l'itération 1. En général, cette charge oscille entre 0,5 et 2 jh.
- Ajoutez 15 % de la charge théorique des passages : vous partez du principe que le périmètre bloqué dans l'itération 1 est intégralement débloqué pour l'itération 2. Cette hypothèse est plutôt raisonnable : la MOE est sensée prioritairement corriger toutes les anomalies bloquantes.
- Enfin, prévoyez une charge supplémentaire pour des tests de non-régression, c'est-à-dire une partie des 85 % des tests déjà joués que vous exécuterez une seconde fois. Selon le contexte, cette charge pourra être très variable. Essayez autant que possible de la réduire et prévoyez une charge fixe "plancher", par exemple une journée où toute l'équipe fera la non-régression, journée dans laquelle vous devrez sélectionner les tests les plus pertinents.
- Pour les itérations suivantes : reprenez le principe de l'itération 2 en estimant un taux de blocage résiduel et une volumétrie des corrections à vérifier.

 Retenez ce principe : si la charge de préparation se dimensionne relativement à la taille de l'application testée, en revanche, la charge d'exécution sera fonction de cette même taille, du nombre de langues, du nombre de configurations matérielles et d'une qualité prévisionnelle de l'application quantifiée par deux ratios : le taux de blocage des tests et le taux d'anomalies prévisionnelles.

Ajoutez à cette estimation globale une charge fixe permettant :

- L'installation et la vérification de la plateforme de test (1 jh).
- L'installation de l'application et ses tests d'acceptation (prévoir 0,5 jh par membre de l'équipe).

#### d. Évaluer la charge de la clôture

La charge que le CP Recette devra consacrer à la clôture peut être estimée à 1 jh par itération, cette estimation couvrant à la fois la rédaction d'un bilan et le déroulé d'une réunion d'une heure à quatre intervenants (a minima, le CP MOA, le CP MOE et le CP Recette).

Chaque itération amènera au remaniement du bilan de l'itération précédente mais nous partirons du principe que ce bilan est remanié intégralement à chaque fois.

La charge de clôture s'estime donc principalement sur la base du nombre de participants aux réunions de clôture.

#### e. Cas particulier des tests automatiques

Dans les deux sections précédentes, nous avons abordé l'évaluation des charges sur la base de tests rédigés en vue d'une exécution manuelle.

Bien évidemment, cette méthode de calcul n'est pas adaptée aux tests automatiques.

Nous ne pouvons pas détailler une méthode d'estimation précise pour ces cas particuliers en raison de sa complexité.

Tout d'abord intervient la nature même de l'outil utilisé pour l'automatisation (Quick Test Pro ? Selenium ?), qui suppose une charge de préparation qui pourra varier du tout au tout. Vous pourrez toujours évaluer le nombre de scripts, le nombre de macros nécessaires... puis estimer une charge de préparation sur cette dimension. La démarche reste ici cohérente.

Mais interviendront alors la complexité de réalisation du script et la compétence du concepteur. Finalement, l'automatisation revient à... écrire un programme comme si votre expert en automatisation était un développeur.

 Nous vous recommandons de vous rapprocher au cas par cas d'un expert en automatisation selon l'outil utilisé : allez au plus simple et procédez comme pour un développement en demandant à votre ressource sa propre estimation de la charge.

De la même manière, la charge d'exécution sera tout aussi difficile à prévoir. Car l'outil d'automatisation peut être un simple assistant à l'exécution dont le résultat visuel sera comparé à une attente consignée dans un cahier de recette... tout comme il pourra être capable de générer un rapport complet et détaillé qui nécessitera une analyse. Notamment, la lecture des rapports QTP peut s'avérer longue...

Bien sûr l'exécution du test pourra être lancée en tâche de fond sur un serveur. Mais combien de fois aurez-vous à les lancer ? Il suffira peut-être que le premier test tombe en erreur pour que tous les suivants soient immédiatement en erreur ?

Bref, l'automatisation amène son lot de problèmes techniques et aléas qui font que l'estimation sera beaucoup moins aisée d'autant plus si l'accès même à l'outil est limité (nous faisons allusion ici aux licences QTP très coûteuses).

## f. Ajouter la charge de pilotage

Comptez une réunion de pilotage par semaine ou toutes les deux semaines (une à deux heures par réunion) ainsi que la charge de rédaction d'un compte-rendu pour chacune d'elle, sur toute la longueur du projet.

La méthode d'estimation est ici la même que pour un projet de développement.

## g. Suivre la charge

Des événements pourront survenir au cours de la recette et qui seront susceptibles de faire varier du tout au tout votre estimation.

Le CP Recette devra donc mesurer un certain nombre d'indicateurs lui permettant de s'assurer du meilleur suivi.

Concernant la phase ORGANISER, il y a peu de risques qu'elle dérive (un seul acteur) sinon lorsque certains éléments ne sont pas stabilisés, comme par exemple la liste des configurations matérielles, notamment pour une recette sur terminaux mobiles qui nécessite souvent de faire des recherches sur les produits.

Concernant la phase PRÉPARER, celle-ci pourra dériver lorsque l'objectif à atteindre change, typiquement, si les spécifications évoluent alors que vous avez déjà commencé à rédiger des tests et construire des jeux d'essai.

Le suivi dans cette phase est relativement simple à faire : le plan de test constitue une liste de livrables (fiches de test et scénarios) pour laquelle vous pouvez piloter un avancement (À rédiger / À revoir / Prêt / Obsolète). Un plan test sous Excel permettra donc très facilement de savoir quels cahiers sont prêts et ceux qui ne le sont pas. Il suffit alors d'estimer les "restes à faire" avec les concepteurs.

C'est surtout la phase EXÉCUTER qui nécessite une attention plus particulière.

Bien sûr, vous devez savoir à tout moment quel test a été passé et quel reste à faire, tout cela au global, par configuration et par testeur. Le plan de test permet d'ailleurs cette démarche sans trop de difficulté. Mais l'essentiel n'est pas là.

Tout d'abord, le début de cette phase pourra marquer une forte montée en charge relativement à la préparation, montée d'autant plus forte que le nombre de configurations matérielles est important.

Si les tests d'acceptation conduisent à un refus de l'application pour l'exécution des tests, vous vous retrouverez donc immanquablement en rupture de charge.

Vous pouvez alors attendre... ou suggérer de passer en recette "pompier" mais cela nécessitera peut-être une renégociation du contenu de la mission.

Si l'application est acceptée pour une exécution, vous ne serez pas à l'abri d'une rupture de charge cependant : vous serez alors tributaire du taux de blocage, c'est-à-dire du nombre de tests non jouables pour cause d'anomalie.

Le CP Recette technique doit donc impérativement estimer le périmètre de ce qu'il ne pourra pas tester ainsi que le nombre d'anomalies prévisionnelles. Ces deux indicateurs sont indispensables : il est vital de les mesurer.

Il pourra aussi se produire un phénomène très courant : l'incompatibilité d'un certain nombre de configurations matérielles.

Le CP Recette se retrouve alors face à une autre problématique. D'une part, le taux de blocage est élevé mais en plus, les anomalies vont bloquer des pans entiers du périmètre et non pas une exigence fonctionnelle ponctuellement.

- Une application web pourra fonctionner sur un navigateur et être totalement hors service sur un autre. Attention, Internet Explorer dans toutes ses versions recèle de nombreuses surprises !

Cette situation provoque simultanément un déséquilibre dans les affectations, c'est-à-dire qu'un testeur pourra avoir 100 % de son périmètre testable quand un autre aura seulement 20 % de ses tests jouables (voire même aucun !).

Le CP Recette doit donc gérer une problématique ternaire : l'adéquation entre une liste de tests, une liste de configurations matérielles et une liste de testeurs. Nous préconisons de partir du principe qu'une configuration matérielle est toujours intégralement testée par le même testeur.

Comment gérer la situation ? Vous pouvez rester à effectif constant puis répartir à nouveau la charge de travail jusqu'à l'équilibre. Un taux de blocage élevé provoque mécaniquement un effet de transfert de charge de l'itération N vers la suivante.

Ceci signifie que votre équipe testera au ralenti dans un premier temps puis devra compenser l'effort dans un second temps. Dans un tel contexte, cela signifie aussi que la nouvelle livraison de l'application doit se faire le plus tôt possible pour limiter cette rupture... ce que doit logiquement faire la MOE de toute façon puisqu'une anomalie bloquante est prioritaire !

- Vous éviterez de faire sortir des ressources pour restreindre le périmètre de la première itération au risque d'être démunis sur l'itération d'après. En revanche, si le taux de blocage s'avérait trop élevé, il y aura une réunion de crise et un plan d'action différent à mettre en œuvre. Dans un tel cas alors vous réduirez la voilure puisque ce n'est plus le même projet mais une recette "pompier" qu'il faut.

Et que dire d'une situation normale où le taux de blocage est faible ?

Le suivi de la charge s'effectue alors sur l'avancement de chaque testeur dont les vitesses de travail peuvent fortement varier, notamment en raison du profil de la ressource :

- Un profil technique testera plus vite et identifiera les anomalies ayant pour origine une erreur de développement.
- Un profil fonctionnel ou AMOA prendra plus de temps dans l'exécution et la rédaction des fiches d'observation.

Par ailleurs, même s'il n'y a pas d'anomalies bloquantes, la vitesse d'exécution des tests sera freinée par la volumétrie des fiches d'observation à rédiger.

➤ Si vous constatez qu'un testeur rencontre des difficultés, transférez une partie de son périmètre sur les épaules de testeurs plus rapides. Le principe est d'évaluer et garder une vitesse moyenne de l'équipe constante.

➤ En dernier recours, vous pourrez éventuellement demander à vos ressources un effort, mais soyez lucide : une heure de travail supplémentaire par jour, ce n'est jamais que 10 %. Il vaudra mieux parfois abandonner des tests de priorité moindre quitte à les déporter sur l'itération d'après.

Les itérations suivantes seront alors indexées sur les mêmes indicateurs : principalement le taux de blocage et le taux d'anomalie.

Il est évident qu'un nombre important d'anomalies nécessitera plus de vérifications mais ce n'est pas cette volumétrie qui aura un réel impact.

Votre charge de travail dépend donc principalement du périmètre que vous n'avez pas pu jouer dans l'itération N-1.

Par ailleurs, comme vous ne pourrez pas prendre pour périmètre de non-régression la totalité de ce que vous aviez testé dans l'itération précédente, il vous faudra cerner l'essentiel à passer dans une charge fixe.

Ainsi par convention, la non-régression ne saurait dériver puisqu'on adapte son contour à la charge disponible.

Nous terminerons notre section sur le suivi en évoquant les dérives sur la charge de clôture et de pilotage : là tout est très clair.

Si le projet s'allonge, s'il y a davantage d'itération, alors la charge augmente... mais relativement peu.

## 2. La recette technique d'un projet de maintenance

Dès lors que nous sommes dans un chantier de maintenance, le pilotage change complètement relativement à un chantier de release.

### a. Évaluer la charge

La charge d'une telle recette dépend essentiellement de deux paramètres :

- L'employabilité du référentiel de tests constitué lors du chantier de release : jusqu'où a-t-on capitalisé ?
- Le taux d'automatisation, si un outil d'automatisation de test est employé.

En effet, par définition une telle recette nécessite de tester l'évolution elle-même, des correctifs d'anomalies... et de faire des tests de non-régression, le plus gros du périmètre.

Il est alors évident qu'un référentiel de tests tout prêt permet de définir "à la carte" un périmètre de tests :

- La phase ORGANISER demande une simple modification du plan de test.
- La phase PRÉPARER consiste uniquement à concevoir les tests des évolutions (ou à modifier des tests existants) et mettre à jour les jeux d'essai et données prérequises.

- La phase EXÉCUTER se fera d'autant plus vite que les tests seront automatisés.
- Clôture et pilotage sont toujours des charges faibles : les maintenances demandent généralement une succession de petites consommations espacées dans le temps et peu de réunions.

Il en découle que la charge des tests sera beaucoup plus forte en l'absence de référentiel ou si celui-ci est obsolète auquel cas, nous vous renvoyons sur la section précédente : vous êtes dans la démarche de faire ce que vous auriez dû faire au moment de la release, une rétro-documentation.

 Vous serez peut-être amené à faire un état des lieux avant même d'entamer une recette, afin de savoir exactement quels tests sont recyclables et ceux qui ne le sont pas.

Nous préconisons une optimisation de la gestion des configurations matérielles : limitez-vous à tester une seule configuration à 100 % puis optimisez les tests sur les autres configurations par des sondages ponctuels sur les points faibles de ces dernières.

Vous pourrez ainsi diminuer la charge d'exécution rapidement.

### b. Suivre la charge

Les recettes de maintenance sont généralement de charge beaucoup plus faible relativement au chantier de release. Ceci est cohérent : si 5 % seulement de l'application change, alors il ne faudra tester que 5 % de celle-ci.

Or nous avons vu que la dimension de ce qui est testé influe directement sur la charge de test : il vous faudra donc en moyenne 5 % de la charge de test du projet de release pour tester une évolution qui impacte 5 % de l'application. Cette règle dépend cependant de votre capacité à définir un périmètre de non-régression aussi faible que possible.

De cette faiblesse de la charge, et du fait même d'être en maintenance, le CP Recette se trouve dans une situation où les tests demandent un effort de suivi moindre que pour une release.

Ainsi, la démarche pourra être beaucoup moins formelle et surtout très pragmatique.

Si l'évolution a un impact faible (par exemple, 5 jh de développement seulement), il est évident que la recette technique aura une charge directement proportionnelle à la taille de l'impact : faible aussi.

Ceci signifie que la planification des petites maintenances évolutives et correctifs est d'une relative souplesse et peu marquée en terme de dérive.

Un diagramme de Gantt trouve-t-il une utilité ici ? Rien n'est moins sûr : le budget des petites maintenances est annualisé et il n'y a pas nécessairement une urgence à les mettre en œuvre.

Notre propos est d'ailleurs similaire à ce que vous pourriez voir pour la gestion de projet de développement : si la taille du projet est trop petite, il n'y a plus d'intérêt à travailler en mode projet mais davantage "au fil de l'eau".

Ainsi, la planification de vos recettes techniques en maintenance pourra être calée sur le planning des livraisons de la MOE.

## 3. La recette fonctionnelle ou VABF

### a. Évaluer la charge

Évaluer la charge d'une recette fonctionnelle revient à reprendre la méthode que nous avons proposée pour une recette technique, mais dans un cas particulier.

Tout d'abord, aucune fiche de test ne sera rédigée : limitez-vous à des scénarios fonctionnels.

La phase PRÉPARER vous demandera donc une charge qui dépend du nombre d'exigences fonctionnelles (de scénarios). Cependant, l'évaluation pourra être un peu différente par rapport à une recette technique, la charge de rédaction des scénarios et des données étant plus élevée.

En effet, les scénarios et données pourront être fonctionnellement plus complexes, la rédaction des scénarios plus fouillée.

Conjointement, le périmètre de la recette pourra se limiter à des tests de priorité Must et laisser de côté des scénarios d'importance moindre, diminuant la charge d'autant.

Pour la phase EXÉCUTER, vous pouvez éliminer la plupart des configurations matérielles et n'en garder qu'une seule. Conservez par contre les configurations logiques permettant de tester le multilinguisme.

Pourquoi ? Parce qu'un changement de configuration matérielle ne change en rien le métier, l'objet même de la recette fonctionnelle.

- Une exception toutefois : les VABF des applications mobiles. La multiplicité des configurations matérielles entraîne des ressentis graphiques très différents qu'une MOA souhaitera évaluer. Par opposition, le ressenti graphique d'une application web ne varie pas autant si bien que les recettes fonctionnelles peuvent écarter ces considérations.

La seconde raison que nous évoquerons sera pratique : en admettant qu'une recette fonctionnelle fasse intervenir plusieurs configurations matérielles, la probabilité de survenance d'une anomalie et la très relative possibilité de correction n'offrent que peu d'intérêt à ce stade du projet.

C'est bien évidemment au moment de la recette technique (donc avant) que les considérations de compatibilité devront être anticipées.

Sauf exception, le taux de blocage en recette fonctionnelle est obligatoirement de 0 % ce qui signifie qu'il n'y a que deux itérations au plus (en théorie).

Enfin, la volumétrie des anomalies est normalement très faible en recette fonctionnelle : vous pouvez négliger la charge de vérification des corrections de l'itération 2 devant le périmètre de la non-régression qui prédomine.

Retenez que la charge de la VABF se situe davantage dans la préparation et moins dans l'exécution quand la recette technique voit son pic de charge en exécution.

## b. Suivre la charge

Nous ne nous étendrons pas sur le suivi de la charge d'une VABF : cela consiste à suivre l'avancée de la rédaction des scénarios de validation fonctionnelle principalement, puis de l'exécution.

À ce stade du projet (en release comme en maintenance) il y a peu de chance que la charge fluctue dans des proportions importantes, a fortiori si des recettes techniques sont correctement gérées avant les VABF.

Cette règle n'est bien sûr plus applicable si la recette fonctionnelle devient une recette "pompier" !

# Risques et reporting

## 1. Les risques d'un projet de recette

### a. En recette fonctionnelle ou VABF

Le principal risque se situe dans l'absence d'une structure en charge des recettes techniques.

Cette absence se traduit par un besoin de compenser l'effort de test en recette fonctionnelle, compétence que la MOA n'a pas toujours.

Aussi, lorsque l'application présentera un taux trop important d'anomalies, la recette fonctionnelle ne sera plus possible... et la recette technique non plus puisque non anticipée.

Seule solution : une recette "pompier".

Un risque secondaire est cependant possible si une équipe d'experts intervient pour des recettes technico-fonctionnelles : que la MOA se décharge de sa responsabilité d'homologuer un métier sur les épaules de ressources qui prennent alors des décisions trop importantes au regard de leur niveau d'implication.

Ce risque peut survenir principalement lorsque les recettes sont externalisées auprès d'un prestataire de services.

### b. En recette technique

Il n'est pas rare que les spécifications fonctionnelles changent en cours de projet : une expression de besoins mal cadrée représente donc un premier risque essentiellement vers la fin de la phase PRÉPARER car il ne sera peut-être plus possible de rectifier les cahiers de recette : il y a un planning à tenir.

Les autres risques de ce projet sont davantage localisés dans la phase EXÉCUTER de la recette, à la première réception de l'application dans un chantier de release.

Dès lors que l'application sort de la phase de développement, sa relative qualité représente le premier des risques. C'est à ce moment-là seulement que l'on saura si l'application est testable. Le début de l'exécution est donc un moment particulièrement critique.

Bien souvent, vous noterez que la compatibilité des configurations matérielles sera source de problèmes multiples : il y a donc un risque sur ce sujet, à plus forte raison pour les terminaux mobiles et autres solutions embarquées demandant de tester aussi l'installation du logiciel.

Ne négligez pas non plus le risque d'écart dans l'environnement de test : vous souhaitez un navigateur Safari sur Macintosh et une fois dans la salle recette vous ne trouvez que sa version sous Windows... exemple vécu !

 Attention pour vos recettes sur terminaux mobiles, n'oubliez pas qu'il faut un temps de chargement des batteries si votre plateforme est neuve. De même que votre fournisseur ne respectera pas toujours le bon de commande que vous avez envoyé !

Un autre risque - très technique celui-là - concerne tout simplement la disponibilité de l'application : il suffit d'une panne du réseau, d'un serveur ou même de l'outil de recette par exemple, pour provoquer un arrêt de l'activité. Ce risque de défaillance est bien réel.

Voici quelques désagréments rencontrés par l'auteur au cours de ses nombreux projets et que vous pourrez rencontrer vous-même :

- Le jeu d'essai prévu a été consommé par quelqu'un d'autre à votre insu.
- D'autres personnes réalisent des tests simultanément et interfèrent avec vos observations.
- L'application est livrée à votre insu par la MOE qui effectue des corrections et rend ainsi invalides vos tests au mieux et rend instable l'environnement au pire (vous pouvez alors tout recommencer).
- Vous avez l'imprimante pour vos tests d'impression... mais ni toner ni papier alors qu'il est impossible d'exécuter à nouveau le test !
- Vous utilisez une adresse mail pour des tests de notification vers un compte inexistant ou alors le serveur de messagerie est en panne...
- ... ou pire, il fonctionne mais les mails inondent la boîte aux lettres de votre client ou de destinataires qui ignorent tout de votre projet !
- Vous n'avez pas d'adaptateur pour charger les batteries de vos terminaux mobiles.
- Vous commandez un terminal mobile dans un système d'exploitation en anglais ou en français et le recevez en espagnol (alors que personne ne comprend cette langue dans votre équipe).
- L'application testée contient des parties communes avec l'équipe de développement (par exemple des bouchons XML) qui les modifient durant vos tests sans vous en informer préalablement.
- Vous n'avez pas prévu de tester l'installation de l'application sur vos terminaux mobiles. À sa réception, vous constatez que le programme d'installation vous demande une version du framework DotNet... et que l'application elle-même demande un framework DotNet dans une autre version.
- L'application web que vous testez contient des composants web téléchargés situés sur un serveur distant rendus inaccessibles par un proxy depuis votre poste de travail. Il y a une dégradation liée par exemple à l'absence de fonctions JavaScript mais que vous interprétez comme des anomalies.
- Vous devez tester une application sur la base d'un référentiel de tests définis dans Quality Center... alors que les exigences ne sont pas saisies dedans pas plus que les anomalies car votre client ne souhaite pas payer trop de licences QC : il utilise Mantis (gratuit) ce qui empêche de dégager toutes les statistiques via le module DashBoard de QC.
- Etc.

### c. Suivre les risques

L'incidence des risques mentionnés dans les sections précédentes est de deux natures :

- Soit le périmètre des tests prévus augmente et votre évaluation est en deçà du besoin.
- Soit un événement survient lors de l'exécution des tests qui empêche le travail d'être fait par votre équipe : vous êtes en rupture de charge.

Il est donc indispensable de suivre les risques, la phase d'exécution étant probablement la plus critique notamment en recette technique puisque s'y trouve le pic d'activité.

Mais dans la pratique, vous aurez peu de marge de manœuvre lors de la survenance d'un risque, au mieux, vous adapterez le périmètre des tests en réponse.

À noter que l'indisponibilité de la plateforme de recette (application et outil) pourra être mesurée soit en exécution seulement, soit en préparation également pour l'outil de test.

Vous conviendrez dans le dossier d'organisation des plages horaires pendant lesquelles vous attendez une totale disponibilité de l'application et des outils, puis vous noterez les durées des indisponibilités ce qui vous permettra en fin de recette de mesurer un taux de disponibilité.

## 2. Le formalisme du reporting

Nous l'évoquons ici pour rappel, mais le formalisme du compte rendu du comité de projet d'une recette technique ou fonctionnelle (COPROJ) est totalement similaire à celui d'un projet de développement :

- Les tâches achevées sont listées.
- Les tâches en cours et les livrables attendus soulignés.
- Les tâches à venir et jalons rappelés
- Un diagramme de Gantt illustre l'avancement des tâches : en avance, en retard, à l'heure (calcul de la dérive en charge).
- Un tableau des points de blocage rencontrés et actions menées ou à mener est présenté.
- Un tableau de suivi des risques est également présent.