Le Développement piloté par le Comportement

Tome 1

Communiquez
avec vos développeurs

Développeurs,
faites-vous plaisir !

Tome 2

Jean-François Lépine

This Page Intentionally Left Blank

Table des matières

This Page Intentionally Left Blank

Communiquez avec vos développeurs

This Page Intentionally Left Blank

Mettez toutes les chances de votre côté

Imaginez : voilà un bon père de famille qui se décide, enfin, a changer sa vieille voiture. Bien décidé à investir dans du neuf, il se rend chez le concessionnaire du coin, et explique ce qu'il veut... oh, rien de bien compliqué : il doit pouvoir aller faire ses courses, partir en vacances de temps en temps, et, ah si, aller au travail tous les jours. "Pas de problème, je vous livre votre nouvelle voiture dans deux semaines" répond le concessionnaire.

Trois semaines plus tard ("ah oui, désolé il y a eu un peu de retard, mais c'est normal" le rassure le concessionnaire), une petite surprise attend notre père de famille : ce n'est pas la voiture familiale qu'il espérait, mais une camionnette, 12m cube, 45 000 € ("on a rencontré des difficultés imprévues"), 2m10 de haut (la porte de son garage mesure 1m90, mais le concessionnaire est convaincu que "c'est quand même plus pratique d'avoir une bonne hauteur de coffre. Ah, et je vous ai même ajouté un volant de course, le top du top !").

Pas très agréable, non ? Ce n'est pas du tout ce que notre père de famille souhaitait ! Pourtant... Pourtant il peut faire ses courses, partir en vacances... Tout ce qu'il a demandé au concessionnaire est là. Le concessionnaire était plein de bonne volonté, il a fait son maximum pour satisfaire son client, a fait de son mieux, l'a écouté, a tenté d'y mettre du sien et d'ajouter des options qu'il pense indispensables ou confortables. On ne va pas l'en blâmer !

Imaginez ça maintenant : **ce père (ou mère) de famille, c'est VOUS** ! Oui, c'est bel et bien vous, vous qui souhaitez concevoir un produit informatique, un logiciel, un intranet, une application mobile... Vous voilà parti avec une superbe idée, rentable, révolutionnaire ! Et vous vous retrouvez finalement avec un produit coûteux, livré en retard, voire parfois totalement inadapté au marché !

Allez-vous vous en prendre au développeur ? A la société qui a conçu votre produit ? Au prestataire de service ? N'est-ce pas plutôt un problème plus général de communication ? Après tout, chacun est convaincu d'avoir fait de son mieux... N'y avait-il pas un moyen de faire mieux ? De mieux faire comprendre votre besoin ? De mieux vous préparer aux éventuels retard, de les limiter ? De réorienter votre demande en cours de route ?

C'est justement l'objet de ce livre. Bien évidemment, ce livre ne va pas vous offrir un moyen magique de réussir un projet informatique. Non, ça n'existe pas, il n'y a jamais de *silver bullet*. Rien n'est magique : **c'est à vous, et à vous seul, de faire réussir vos projets informatiques**. Par contre, ce livre peut peut-être vous donner quelques pistes de réflexion, afin de mettre plus de chances de votre côté.

This Page Intentionally Left Blank

Chapitre 1

Le besoin métier

1.1 Ayez une vision de votre produit

Votre idée est peut-être géniale, révolutionnaire, utile ou juste rentable (aucun mal à cela). Elle va peut-être changer le monde, ou plus modestement faciliter le quotidien de pas mal de monde.

Pensez au premier téléphone portable : en voilà un produit qui a changé bien des choses; impossible aujourd'hui d'imaginer vivre sans : sans accès à votre boîte mail, sans appareil photo, sans sms, sans carnet de contacts... Mais, attendez, le premier téléphone, il ne faisait pas tout ça ! Non, rappelez-vous, un téléphone portable, à la base, ça sert à pouvoir téléphoner n'importe où.

Tout le reste, c'est utile, pratique, génial, sans doute indispensable au quotidien, bref, c'est tout autant révolutionnaire. Mais si on doit définir le téléphone portable en quelques mots, c'est "juste" un moyen d'échanger vocalement et instantanément de l'information entre deux intervenants, où qu'ils soient dans le monde.

Votre produit est pareil : **il doit pouvoir se définir en une courte phrase**, et en réalité ne sert qu'à une chose. Si si, tout le reste est utile commercialement, esthétiquement, fonctionnellement... Mais votre produit possède un coeur, et la première chose à faire est de l'identifier.

Cela vous permettra de savoir vraiment ce dont vous avez besoin. Si vous demandez au père de famille, dont nous parlions plus haut, de décrire sa voiture, il va peut-être dire qu'il veut un break Citroen de 120 chevaux gris métallisé, avec 5 portes. Il sait ce qu'il veut... Pourtant il oublie l'essentiel : la voiture doit lui permettre de se rendre d'un point A à un point B.

Oui, c'est évident : une voiture permet de se déplacer. Mais pourtant, évident ou pas, une voiture, ce n'est rien d'autre ; c'est ce qui reste une fois qu'on a enlevé tout le

reste : retirez la climatisation, la peinture métallisée, le cuir... Il nous reste le coeur de la voiture, ni plus ni moyen qu'un moyen de locomotion.

Allez même plus loin, et imaginez une voiture sans moteur ; est-ce toujours une voiture ? Oui, sans aucun doute, le moteur n'est qu'un moyen pour cette fin, ultime, de permettre aux gens de se déplacer : une voiture à cheval n'en reste pas moins une voiture. Certes, elle n'est peut-être pas optimale, pas rapide, pas jolie, tout ce que vous voudrez... Mais C'EST UNE VOITURE. Et peut-être moins cher en plus !

Pour votre logiciel, votre intranet, votre application mobile, c'est pareil : **identifiez le coeur de votre produit, ce qui fait son essence même, ce sans quoi votre produit n'est plus votre produit, et là vous aurez de bonnes bases pour commencer**.

Comme le dit Liz Keogh, **chaque projet informatique doit poursuivre une vision**. Cette vision répond souvent à un besoin économique (réduire des coûts, améliorer la productivité), voire parfois à des souhaits différents (améliorer le quotidien des gens, changer les mentalités...). Dans tous les cas, tout ce que vous allez mettre dans votre produit doit poursuivre cette vision.

> *Tout ce que vous mettez dans votre produit doit poursuivre une vision*

1.2 Le bon produit est celui qui répond à votre vision

Quand ils ont conçu Basecamp, un logiciel de gestion de projets aujourd'hui reconnu et plébiscité, Jason Fried et David Heinemer avaient une vision : faciliter la vie des différents intervenants lors de leurs créations de projets. Cette vision est visible partout dans Basecamp : simple, facile d'accès, rapide et ergonomique, il facilite réellement la vie de celui qui l'utilise.

Ils auraient pu choisir de concevoir un logiciel riche, multi-fonction et polyvalent ; ils ne l'ont pas fait. Pourquoi ? Tout simplement parce que cela n'aurait pas servi leur vision. On pourrait vouloir un CRM dans Basecamp, ou un intranet, ou tout ce que vous voudrez. Tous ces modules pourraient être forts agréables, utiles et d'excellents arguments commerciaux. Mais ils ne servent pas la vision du produit. Et c'est justement pour ça que Basecamp plaît autant : il ne fait qu'une chose, mais il le fait bien !

Pensez également à ceci : certes Basecamp n'est pas riche fonctionnellement, mais

- Il est largement suffisant dans 90% des cas
- Il a été d'autant moins coûteux à concevoir
- Il a été d'autant plus rapide à développer
- Chaque fonctionnalité a pu être testée en profondeur et est fiable

- Il ne souffre d'aucune complexité inutile et est rapide
- Les équipes de développement se sont focalisés sur un domaine fonctionnel simple
- Tout changement ou évolution est simple à mettre en place

Tout cela serait impossible si Jason Fried et David Heinemer ne s'étaient pas focalisés sur leur vision du produit et n'avaient pas refusé systématiquement d'ajouter des fonctionnalités qui ne répondent pas à cette vision initiale.

Rappelez-vous : vous êtes le seul maître à bord. C'est à vous de vous servir de votre vision du produit pour limiter le superflu et focaliser toute l'énergie disponible (développeurs, serveurs, graphistes, ergonomes...) sur ce qui répond à cette vision, autrement dit sur ce qui est vraiment utile.

> *Retirez de votre Produit tout ce qui ne sert par votre Vision*

1.3 Vous attendez des résultats métiers

Vous avez probablement souvent entendu cette expression : ce qui compte, ce sont les résultats !

Bon, je ne vais pas vous dire que la fin justifie les moyens, pas du tout. Non, ce que je veux dire, c'est qu'en investissant du temps et de l'argent dans votre projet, vous ne devez jamais oublier que ce qui compte ce sont les résultats métiers (ou fonctionnels) de votre projet. Ces résultats qui justement servent votre vision.

Pensez-y : chaque fonctionnalité, chaque élément d'interface, chaque bouton, chaque champ de saisie... tout cela doit fournir un bénéfice à l'utilisateur de votre produit !

Ça peut paraître évident... Mais alors pourquoi consacrer autant d'énergie sur des choses inutiles fonctionnellement ? Si si, ça arrive tout le temps ! En tant que développeur j'ai souvent eu l'occasion de le voir : une grosse partie de mon temps était consacré à du superflu (gestion des profils utilisateurs hyper poussée, interfaces graphique à la iGoogle, optimisation prématurée des performances...), alors même que les fonctionnalités métier n'étaient pas encore finies, voire même pas spécifiées !

L'énergie dépensée sur un projet doit être corrélée aux bénéfices que votre produit en tirera pour satisfaire votre vision. Si, alors que les fonctionnalités métiers ne sont pas pleinement finies, testées, fiables, éprouvées et ouvertes aux changements, l'énergie dépensée ne sert qu'à "vendre du rêve", vous risquez la catastrophe.

Concevez votre produit, non pas pour le vendre, mais pour qu'il réponde à un but (votre vision). Si vous y arrivez, vous aurez à ce moment là en main les meilleurs atouts pour le vendre.

Le bénéfice sera d'autant plus grand que, au lieu de perdre du temps sur des choses moins utiles, vos développeurs, prestataires ou la société qui a en charge votre projet, vont mieux comprendre votre besoin, sans se perdre en détails inutiles pour le moment.

En vous focalisant sur les fonctionnalités métiers, vous allez aussi pouvoir plus rapidement vous confronter au feedback de vos futurs utilisateurs. Les changements fonctionnels de votre produit seront ainsi réalisés plus tôt, et donc moins coûteux.

> *Ne concevez pas votre produit pour le vendre, mais pour qu'il réponde à un but. Vous aurez alors en main les meilleurs atouts pour le vendre*

1.4 Re-priorisez souvent, changez souvent

Vous l'avez vu : votre produit doit se focaliser sur ce qui apporte à un métier, sur ce qui répond à votre vision.

C'est bien beau, mais on en est tous conscients : un aspect métier peut être essentiel aujourd'hui, mais totalement inutile ou différent demain ! Comment gérer ces changements dans votre projet ?

Vous vous en doutez, de nombreuses personnes ont tenté de répondre à cette question. Parmi elles, les (nombreux) auteurs du Manifeste Agile. Voici certains des principes de ce Manifeste tel qu'ils sont présentés sur http://agilemanifesto.org, qui nous intéressent particulièrement :

> *"Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.*

> *Accueillez positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client.*

> *Livrez fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts."*

C'est assez clair : si l'on suit ces principes, largement reconnus aujourd'hui dans le monde de l'édition logicielle, **le produit doit être confronté le plus tôt possible au monde réel pour obtenir un feedback régulier des potentiels utilisateurs**.

Ce feedback régulier va vous permettre de réorienter votre produit afin de supprimer, modifier ou ajouter des comportements dans celui-ci. Vous avez donc besoin de pouvoir re-prioriser certains éléments, de changer en cours de route le parcours de développement de votre site, logiciel, application ou intranet.

En un mot : vous avez besoin de confronter votre produit aux utilisateurs. Plus vous le confronterez souvent, plus votre produit sera compétitif, et plus vous aurez besoin de faire évoluer votre produit.

Vous l'avez compris : vous avez besoin de changements tout au long de la phase de conception de votre produit. Cependant, vous l'avez sans doute déjà vécu, **tout changement dans un produit informatique est généralement :**

- long
- coûteux
- source d'erreurs et de bugs
- difficile et démoralisant pour les équipes

L'idée pour résoudre ces difficultés est assez simple, mais parfois difficile à mettre en place, comme nous allons le voir.

> *Vous devez trouver une solution pour faciliter le changement*

1.5 Gérez votre besoin de changements

L'idée du Manifeste Agile pour gérer le changement peut paraître simple parce qu'il "suffit" (entre autres) de modifier les cycles de développement de votre produit. Généralement, un produit est conçu d'un seul bloc : on développe un produit non fini pendant une période assez longue, et on livre un produit fini (enfin, ça c'est qu'on espère !) à la fin. Cette méthode implique un effet tunnel assez long et souvent dévastateur.

Qui ne connaît pas le célèbre tableau de Léonard de Vinci : La Joconde ? Que se serait-il passé si Léonard de Vinci avait dû peindre Mona Lisa comme on travaille généralement sur un projet informatique ? Cela donnerait à peu près ceci :

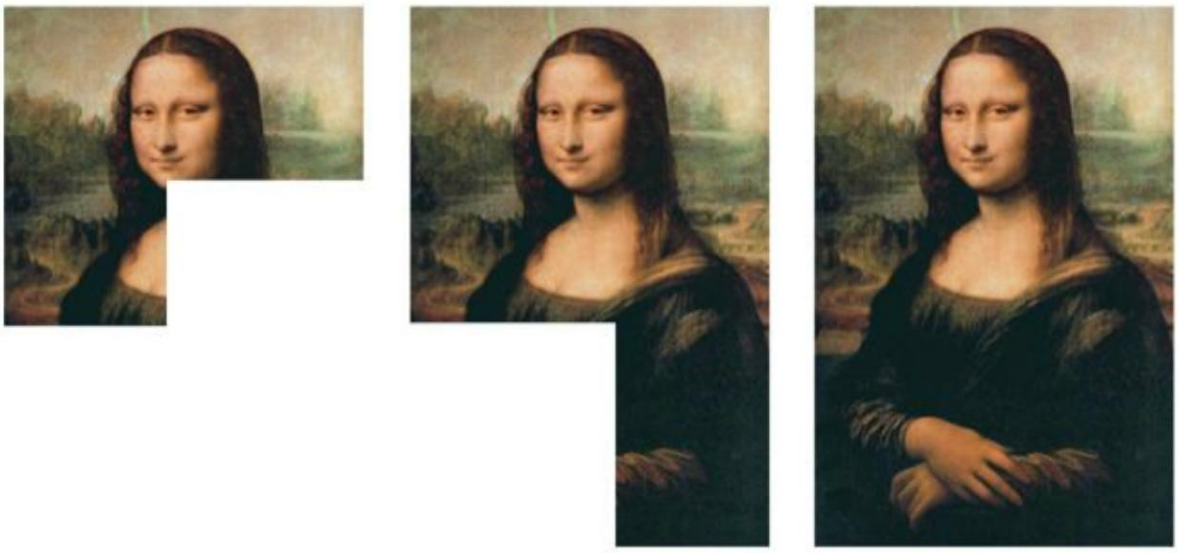

Figure 1.1 Méthodologie classique : le travail et découpé en lots, chaque lot est totalement réalisé avant de passer à la suite. Le changement fonctionnel en cours de route est difficile.

Le travail est tout de suite très fin, et on ne peut le livrer qu'à la fin... Difficile dans ces conditions d'accepter le changement !

Maintenant examinez la méthode proposée par les fondateurs du Manifeste Agile. Les équipes techniques vont travailler en cycles, itérativement, de manière à livrer régulièrement un produit, certes moins complet, mais exploitable et sur lequel il est possible de fournir un feedback.

Le rythme de ces itérations, et leurs objectifs, sont fixés en accord avec tous (équipes techniques, fonctionnels...), et chacun s'engage à faire le maximum pour livrer un produit exploitable. Voici ce que cela donnerait cette fois-ci pour notre cher Léonard :

Figure 1.2 Méthodologie agile : le tableau s'affine petit à petit. Le changement fonctionnel est facile, même en cours de route

Bien sûr, un logiciel qui prendra un an à être développé ne sera pas exploitable commercialement au bout de trois itérations de deux semaines ! Par contre vous aurez moyen d'obtenir des premiers retours utilisateurs, et surtout vous allez pouvoir reprioriser et changer certains éléments en cours de route : la forme du visage ne vous convient pas ? Faites la changer lors d'une prochaine itération ! Les couleurs sont trop ternes ? Priorisez un changement des couleurs pour voir ce qu'en pensent vos futurs utilisateurs. Bref, vous avez compris l'idée !

Les principes agiles sont simples et évidents, mais ils peuvent être étrangement contre-intuitifs à mettre en place : souvent, les équipes (techniques et fonctionnels) ont du mal à se détacher de leurs anciennes pratiques, quant bien même elles adhèrent aux concepts agiles.

J'ai à plusieurs reprises eu l'occasion de voir des entreprises adopter des méthodologies agiles trop brusquement, ou d'une manière inadaptée. Les méthodes agiles ne sont pas un remède miracle à tous les maux, il serait faux de croire que du jour au lendemain des équipes peuvent oublier toutes leurs anciennes pratiques et que par magie vous serez livré à temps et avec un produit fini.

Mais le changement vers l'agilité peut être simple et rapide, sous deux conditions :

- **vous DEVEZ vous impliquer dans votre projet !** C'est à vous de prioriser, de dire ce qui va, ne va pas, régulièrement.
- Vous devez accepter de vous faire aider : **les changements à adopter ne sont pas que méthodologiques, ils sont aussi conceptuels**. De nombreuses sociétés agiles sauront vous guider et vous conseiller. Et croyez moi, l'investissement en vaut la peine, largement !

> *Itération, feedback; itération, feedback; itération ...*

Chapitre 2

Communiquez

2.1 Adoptez une Langue Commune

Ce que vous entendez en utilisant un mot peut parfois (souvent !) être totalement différent de ce qu'une autre personne entend pour ce même mot. Et c'est encore pire avec les développeurs !

Gardez cela à l'esprit : les développeurs ne parlent pas la même langue que vous. Lorsque vous parlez "stock", "produit" ou "commande", un développeur pensera "base de données", "enregistrement" et "sauvegarde en base".

La première chose à faire est donc de constituer avec vos équipes une Langue Commune. Peu importe le moyen : une feuille excel, un document word ou un bout de papier feront l'affaire ! L'important est de créer un référentiel commun, souvent désigné par "Ubiquitus Language", c'est-à-dire **un moyen d'utiliser une terminologie dont on sait qu'elle est comprise par tous de la même manière**.

Attention, cet Ubiquitus Language, **cette Langue Commune ne doit PAS être élaborée unilatéralement !** Instaurez un dialogue entre toutes les parties prenantes du projet, jusqu'à déterminer des termes plus importants que les autres. Identifiez les, puis, en commun, définissez-les. Ne l'oubliez pas : une définition que vous aurez mis en place tout seul ne voudra peut-être rien dire pour un développeur.

Dernière chose : ce référentiel commun ne doit porter que sur le domaine fonctionnel. Il est inutile de créer un référentiel technique (du moins pas en dehors des équipes techniques) : ce qui compte ce n'est pas de faire du développement logiciel ; non, ce qui compte c'est votre produit, vos bénéfices fonctionnels et votre vision !

Pour synthétiser cette Langue Commune en quelques mots, elle permet :

- d'éviter un travail de traduction entre les tâches techniques et les besoins fonctionnels (le jargon métier)
- que tout le monde "parle" votre projet comme vous le "pensez"
- d'éviter de vous retrouver avec des surprises idiotes, dues à des incompréhensions sémantiques ou à des ambiguïtés.

> *les développeurs ne parlent pas la même langue que vous*

2.2 Laissez la technique aux développeurs

Laissez-moi vous rassurez tout de suite : **non, vous n'avez pas besoin de savoir comment fonctionne techniquement votre produit**. Non, vous n'avez pas à comprendre la terminologie des développeurs.

Mieux ! Vous ne devez SURTOUT PAS comprendre ce que font les développeurs.

Ça fait maintenant quelques années que je conduis une voiture. Je sais passer mes vitesses, je comprend comment lire mon tableau de bord... Mais je ne sais pas, mais alors pas du tout, comment fonctionne ma voiture. Et croyez-moi, je n'en ai rien à faire !

Je dois vous avouer une chose : quand j'ai une panne, je vais voir mon garagiste et je lui fais confiance. Quand il m'explique le problème, je dis des "ah oui ?" avec un air curieux. Mais je ne comprend rien ! Et c'est tant mieux !

Pourquoi serait-ce différent pour l'informatique ? Mettre les mains "dans la cambouis" risque :

- de brider vos demandes quand vous allez imaginer qu'un point va être techniquement coûteux alors qu'il n'en n'est rien
- de vous faire adopter une angle de vue différent (celui des développeurs) au détriment de votre vision initiale
- de laisser le code dominer le domaine fonctionnel
- de vous faire perdre votre force fonctionnel

Je peux vous le dire : les projets les plus coûteux et les plus en retard sur lesquels j'ai travaillé ont été initié par d'anciens développeurs devenus chefs d'entreprise ou fonctionnels. Naturellement, et on ne peut pas les en blâmer, ils ont tendance à vouloir comprendre l'implémentation technique de leur produit.

Cependant, cela est très coûteux : tout d'abord l'équipe technique doit passer beaucoup de temps à expliquer ce qu'elle fait au lieu d'avancer réellement. Ensuite, elle doit passer du temps également à monter en compétence le fonctionnel. Enfin, et ce n'est pas le moindre, cela introduira un sentiment de défiance entre vous et l'équipe

technique, sentiment qui va ralentir et limiter vos possibilités de créer une Langue Commune et freiner l'investissement de vos équipes.

> *Vous ne devez surtout pas comprendre ce que font les développeurs*

2.3 Racontez votre produit

Combien de documents sont passés entre vos mains sans que vous ne les lisiez totalement, partiellement ou pas du tout ? Avouez-le, cela nous est tous arrivé au moins une fois.

Pourquoi le document que vous fournissez à votre prestataire ne serait-il pas justement celui-là même qu'il va survoler, ou mal comprendre ?

Un document word, par exemple, est une excellente base de travail. Vous pouvez y noter vos réflexions, vous en servir comme brouillon... Mais pas comme spécification !

Comprenez : votre pensée est ce qu'elle est : mouvante, changeante, et surtout personnelle. Avec tous les efforts du monde, vous n'arriverez pas à transmettre l'ensemble de votre pensée par écrit. Il subsistera toujours de l'interprétation, des non-dits, de l'implicite... Voire même parfois des contradictions qu'il faudra lever. C'est normal, comme un projet, une pensée vit et change.

Votre vision, vos besoins, vos spécifications... tout cela doit faire l'objet d'un échange oral. Racontez votre produit plutôt que de le décrire. Qu'en pensez-vous ?

Votre projet doit être le résultat d'une conversation : impliquez vous, mais impliquez aussi le testeur, le fonctionnel, l'ergonome, le graphiste et le développeur. Racontez leur votre produit, et dialoguez, de telle manière que :

- Ils **s'approprient votre besoin**
- Ils **mettent en évidence certaines incohérences** le plus tôt possible
- Ils vous proposent des changements (vous restez le seul maître de les accepter ou non)
- Ils soient impliqués et aient le sentiment d'être partie prenante de votre projet
- Ils ne perdent pas de temps à traduire votre message

C'est d'ailleurs comme ça que vous allez vous rendre compte que certains points demeurent obscurs, malgré tous vos efforts. Cela signifiera peut-être que ces points sont plus importants que vous ne l'estimiez, et il sera peut-être utile d'y consacrer plus de temps.

Oubliez donc les spécifications fonctionnelles détaillées (SFD), et autres joyeusetés qui font la joie du développeur quand il doit lire un document de 200 pages pour com-

prendre un besoin qu'il mettra 20 minutes à implémenter. La documentation est utile, les SFG, SFD, etc. aussi, mais uniquement quand elle est réservée aux cas qui le nécessitent vraiment !

Oubliez cette masse de documentation d'autant plus qu'elle va être un frein à votre besoin de changements : très vite, avec de nombreux documents, vous allez :

- ou bien avoir un décalage conséquent entre vos spécifications et votre produit (quid des nouveaux arrivants sur le projet ?),
- ou bien devoir passer un temps précieux à mettre à jour vos spécifications à chaque changement. Et ça, croyez-moi, dans la vraie vie ça n'arrive jamais si vous n'avez pas une ressource dédiée à cette tâche, à temps plein !

Vous allez le voir, il existe d'autres moyens de préciser votre besoin. En attendant, oubliez les documents word, pdf, rtf... Et Initiez la conversation.

> *Racontez votre produit plutôt que de le spécifier*

2.4 Identifiez le Domaine fonctionnel

Vous l'avez vu, vous avez besoin de constituer une Langue Commune avec vos équipes. Vous le savez également, une image vaut souvent mieux qu'un long discours... Pourquoi ne pas concilier les deux ?

Il existe une approche, mise en évidence par Eric Evans, pour faire en sorte le code source (ce qui permet à votre produit de fonctionner) corresponde de près aux fonctionnalités métiers. Cette approche, c'est le Domain Driven Design, ou Conception Pilotée par le Métier.

En général, au fil du temps, les développeurs qui auront travaillé sur votre projet vont mettre en place ce qu'on appelle familièrement des "rustines", des morceaux de code source rapides à mettre en place mais peu maintenables, destinés à gérer vos demandes de changements fonctionnels.

Pire, dès le départ, les développeurs vont se focaliser sur les aspects techniques du projet. Normal me direz-vous ? Non, car s'ils font cela, l'énergie à dépenser pour comprendre les liens entre le code source et le besoin fonctionnel risque d'être très importante, voire tellement importante que votre projet risque de vous mettre sur la paille à la moindre demande de changement.

Le Domain Driven Design propose une approche différente : les développeurs vont se focaliser avant tout sur le besoin métier, et concevoir leur code source autour de ce besoin. Ça peut paraître une évidence, mais c'est loin d'être le cas, croyez-moi ! Le code source sera alors le reflet du besoin métier.

Si vous et vos équipes souhaitez mettre en place cette orientation (du Domain Driven Design), **la première chose dont vous aurez besoin, après une Langue Commune, sera une cartographie du domaine fonctionnel couvert par votre application**.

Rassurez-vous, rien de bien compliqué. Oubliez les logiciels de modélisation, oubliez Visio, oubliez l'UML, Merise (si vous les avez déjà utilisés), voici venue l'ère ... du papier et du crayon !

Listez, puis représentez sur un dessin simple, l'ensemble des grands concepts fonctionnels liés à votre produit (ceux qui sont présents dans votre glossaire, votre Langue Commune). Reliez-les entre eux par de flèches annotées, qui représenteront les différentes interactions de ces concepts entre eux. S'il y a trop de flèches à un endroit, c'est que vous vous attardez trop sur des détails. Si vous pensez que ce ne sont pas des détails, prenez une autre feuille et redémarrez, mais cette fois en zoomant sur la partie concernée.

Cette cartographie, qui doit être simple (en caricaturant un peu, disons que votre neveu de huit ans devrait pouvoir la comprendre s'il connaît votre Langue Commune !), sera le **référentiel du Comportement de votre Produit**.

Ce comportement étant désormais identifié, il sera plus facile pour les équipes d'appréhender les changements et d'en identifier les conséquences.

Cette cartographie vous sera utile à vous aussi : elle va vous permettre d'identifier l'indispensable, l'utile et le pratique dans votre produit. Vos équipes ne doivent commencer à travailler sur l'indispensable. L'utile et le pratique viendront après.

> *Prenez un papier et un crayon, puis dessinez la cartographie fonctionnelle de votre Produit*

2.5 Faites-vous interviewer

Mais attention, cette cartographie fonctionnelle, en réalité... ce n'est pas à vous de la faire !

Pourquoi donc cet air surpris sur votre visage ? Je veux simplement dire que ce n'est pas à vous *seul* de la réaliser. Mais rassurez-vous, vous en serez le moteur.

Concrètement, vous devez dans un premier temps discuter avec vos développeurs, mais surtout ceux-ci doivent réaliser un travail d'enquête : vos développeurs doivent vous interviewer.

Ou plutôt ils doivent interviewer le Produit. Après tout... vous êtes le porte-parole de votre produit.

Cet interview, comme une interview classique va consister en différents moments :

- présentation générale d'un aspect fonctionnel
- une question de l'équipe technique sur un point abordé
- votre réponse fonctionnelle
- une nouvelle question de l'équipe technique
- etc.

Attention, l'interview ne doit jamais dériver sur des aspects techniques. Il sera largement temps plus tard de voir comment résoudre techniquement tel ou tel point. Pour l'instant, ne discutez QUE du fonctionnel.

Cette démarche a au moins trois avantages :

- impliquer les équipes techniques sur votre besoin métier
- mettre en évidence des aspects implicites, qui sont évidents pour une personne du métier, mais totalement inconnus des équipes techniques
- vous permettre de prendre du recul sur votre métier et votre besoin en l'expliquant à des néophytes.

> *Les équipes techniques doient interviewer le porte-parole du produit*

2.6 Les tâches ne servent à rien

Vous voulez que vos développeurs développent une nouvelle fonctionnalité ? Ne leur donnez pas de tâche à faire ! N'exprimez que votre besoin...

Imaginez un instant cette scène : vous re-voila chez votre concessionnaire automobile, prêt à signer le chèque de votre nouvelle voiture. A votre avis, des deux situations suivantes, laquelle est la plus adaptée :

- Situation 1 :

> *Client : "J'ai besoin de la voiture le mois prochain. Il vous faudra donc passer commande auprès du constructeur. Pour cela vous disposez du logiciel mis à disposition par ce dernier. Entrez vos codes d'accès, puis saisissez la référence de la voiture, puis cliquez sur "Commander". Dès que vous recevez la voiture, faites un bilan complet, puis appelez-moi.*"

- Situation 2 :

> *Client : "J'ai besoin de ma voiture le mois prochain, je sais que vous ferez le maximum. Tenez moi simplement régulièrement informé de l'avancée de la commande..."*

Alors, à votre avis, quelle situation est la plus adaptée ? La deuxième sans doute, non ? Examinons la Situation 1 : que se passe t-il si le concessionnaire appelle directement le constructeur pour commander votre voiture plutôt que de respecter les tâches qui lui sont assignées...

Quel en sera le résultat ?

- Vous serez livré plus vite
- **Les objectifs seront atteints**
- **Pourtant le concessionnaire n'aura pas accompli les tâches demandées**

Je pense que vous avez compris : peu importe ce que va faire votre concessionnaire, ce qui compte c'est d'être livré à temps. Pire, vous risquez d'embrouiller le concessionnaire ou de le mettre sur des fausses pistes. Avec une liste de tâches à suivre, le concessionnaire ne pourrait pas prendre l'initiative d'appeler directement le constructeur automobile.

Et bien pour le développeur c'est pareil. **Exprimez votre besoin, les délais souhaités, les modalités, mais ne lui dite pas comment réaliser ce besoin**, vous risquez de le mettre sur de fausses pistes.

Attention, les listes de tâches sont utiles bien entendus, mais pas pour communiquer. Vous pouvez avoir votre liste de tâches, les équipes de développement vont peut-être (sans doute !) transformer votre besoin en sous-tâches techniques... Mais en aucun cas vous n'avez besoin de communiquer avec des tâches à faire. Le dialogue ne doit porter que sur votre besoin métier.

> *Communiquer en liste de tâches ne sert à rien, le dialogue ne doit porter que sur le besoin métier.*

This Page Intentionally Left Blank

Chapitre 3

Impliquez les parties prenantes

3.1 Un projet est comme un Film

Drôle de titre, n'est-ce pas ? Pourtant on peut le dire, un projet informatique, c'est un peu comme un film : on peut prendre plaisir à le tourner, y apprendre des choses, mais l'essentiel reste que ce film voit le jour.

Faisons un jeu : quel est le point commun à tous les films, films d'animation, court-métrages ? Tic-tac, tic-tac... Driiing ! Réponse : ils ont tous un générique, c'est à dire une liste des différentes personnes qui y ont travaillé, classées par rôle.

Et bien dans un projet informatique c'est pareil, il existe des rôles. Il peut arriver que certaines personnes aient plusieurs rôles, selon la taille du projet, mais il est indispensable de définir les domaines de compétence nécessaires à la bonne tenue de votre projet.

En général, voici les rôles qui seront associés à votre projet :

1. Le **Propriétaire du produit** : Personne qui fournit la Vision du produit. Est souvent celui qui finance le projet.

2. Les **Alliés du produit** : Personnes qui soutiennent et aident fonctionnellement le Propriétaire du produit. Elles peuvent avoir un impact sur le Produit

3. Les **Analystes fonctionnels** : Personnes qui déterminent les solutions fonctionnelles à apporter pour répondre à la Vision du Propriétaire dur Produit

4. Les **Ergonomes** : Personnes qui déterminent l'ergonomie la plus adaptée pour mettre en place les solutions fonctionnelles auprès des utilisateurs ciblés par le Produit

5. Les **Graphistes** : Personnes en charge d'améliorer l'aspect visuel ou d'appliquer des solutions proposées par les Ergonomes

6. Les **Testeurs** : Personnes en charge de la recette applicative de premier niveau

7. Les **Développeurs** : Personnes en charges de l'implémentation technique des solutions proposées par les Analystes fonctionnels

Bien, ça, c'est dit. Bien entendu, il est fréquent que certains de ces rôles se chevauchent, le plus souvent pour des raisons de coût ou de ressources : l'ergonome est graphiste, l'analyste fonctionnel est propriétaire du produit, etc.

Dans tous les cas, ces rôles doivent dialoguer entre eux, puisque chacun a son rôle à jouer dans la réalisation du Produit. Gardez ce découpage en tête, il est important pour la suite.

> *Les Partie-prenantes sont au Produit ce que les acteurs sont au Film : chacun joue un rôle différent, mais tous mettent en scène la Vision de ce qu'ils produisent*

3.2 Devenez spectateur un instant

Votre produit est destiné à être vendu, que ce soit financièrement auprès de vos clients, ou commercialement auprès de vos utilisateurs. En général, c'est vous qui présenterez votre Produit, par le biais de réunions commerciales, de plaquettes publicitaires, ou que sais-je encore.

Mais avez-vous pensé à laisser vos développeurs faire cette présentation ? Pas de panique, **je ne vous parle pas d'envoyer un geek barbu auprès de vos prospects et investisseurs**, non. Non, je vous propose de laisser vos développeurs vous faire une présentation à vous, le propriétaire du produit.

L'idée est la suivante : on l'a vu, il est plus pratique de travailler par itérations successives pour votre projet. À la fin de chaque itération, vos collaborateurs doivent être capables de vous fournir une version exploitable de votre produit. **Pourquoi dans ce cas ne vous feraient-ils pas la démonstration de ce qu'ils ont à vous livrer ?**

C'est assez classique dans les préceptes agiles : à la fin de chaque cycle l'équipe technique va vous présenter ce qu'elle vous propose, et ce pour différentes raisons :

- cela vous permet de valider votre demande initiale

- cela permet d'impliquer l'équipe technique dans la bonne marche fonctionnelle de votre produit
- cela permet aux acteurs de votre projet d'éprouver régulièrement de la satisfaction personnelle en réussissant à vous livrer dans les temps quelque chose qui fonctionne

Une simple présentation d'une heure, pourquoi pas un vendredi sur deux, par l'équipe technique, vous permet donc d'impliquer dans votre projet les plus réfractaires : les développeurs. Pourquoi ne pas l'adopter ?

Bien entendu, encore une fois, ce n'est pas une recette miracle. Mais l'accumulation des ces instants d'échange et de dialogue ne peut que faciliter votre projet.

> *Laissez l'équipe technique vous faire une démonstration à chaque cycle.*

3.3 Tous les acteurs ont des bonnes idées

Un film, c'est avant tout un réalisateur, qui insuffle l'âme du film, sa Vision. Pourtant il est très fréquent que de bonnes idées proviennent d'acteurs : des improvisations, des échanges, des contestations, des propositions...

Les bonnes idées peuvent venir de partout. Qu'elles soient fonctionnelles, ergonomiques, esthétiques... vous devez écouter ces idées, les assimiler et, si certaines vous semblent pertinentes et en accord avec votre vision, les implémenter.

Après tout, votre spécialité c'est la connaissance du domaine fonctionnelle. Votre produit peut répondre à un besoin de bien des manières, et même d'une façon que vous n'avez pas encore imaginée.

Vos développeurs ne connaissent pas à la base votre besoin fonctionnel, mais leur métier les amène a observer et comprendre une multitude de manières de répondre à un problème. Chaque site internet qu'ils visitent répond à une problématique, chaque site a une approche différente des problèmes, et eux-mêmes sont utilisateurs de produits.

Restez maître de votre Produit, mais, après tout, ce qui compte c'est de rendre service, au mieux, à vos utilisateurs. Prenez les bonnes idées là où elles sont...

> *Imprégnez vos développeurs de votre besoin pour leur laisser l'occasion de trouver de bonnes idées*

This Page Intentionally Left Blank

Chapitre 4

Exprimez votre besoin

4.1 Tous la même grammaire

Les spécifications classiques, on l'a vu, échouent souvent : elles acceptent mal le changement, sont souvent lourdes et ne concernent pas toujours que le besoin métier.

Pour remédier à cela, il est possible d'utiliser une autre approche, celle préconisée par le Développement Piloté par le Comportement.

"Quel terme barbare !" me direz-vous. C'est vrai. Mais, en fait, rien de technique ou de geek derrière ce terme. C'est simplement une démarche qui consiste à communiquer avec vos développeurs d'une autre manière, de sorte qu'il vous comprennent.

C'est cette démarche que je vous propose de découvrir à partir de maintenant.

Cette approche a au moins trois avantages :

- L'écrit sert à la fois à spécifier et à échanger avec les équipes
- Le changement est mieux géré
- Vous savez à tout instant ce qui est fait et ce qu'il reste à faire

Pour au moins un inconvénient :

- L'investissement et la disponibilité du Propriétaire du Produit doivent être importants

Cette approche se base sur la rédaction des spécifications sous un formalisme différent, dénommé généralement Gherkin. Ce formalisme n'est pas obligatoire, mais il a été prouvé à plusieurs reprises qu'il est pratique et répond bien aux problématiques que vous rencontrerez probablement.

Si vous souhaitez faire du Développement Piloté par le Comportement (Behaviour Driven Development en anglais, ou BDD), chaque acteur de votre projet doit apprendre à lire et à comprendre ce formalisme. Pas de panique, le coût d'apprentissage est quasi nul ; c'est justement l'un des nombreux atouts de Gherkin.

4.2 Décrivez vos Fonctionnalités métiers

Vous avez votre Vision du Produit, il est temps d'expliquer ce que vous attendez aux développeurs.

Rien de plus simple : commencez par identifier les aspects fonctionnels que vous souhaitez qu'ils développent, et ce en les décrivant avec une courte phrase. Juste une phrase, courte, pas plus. Cette phrase sera le titre de votre fonctionnalité.

Chaque fonctionnalité doit pouvoir se distinguer des autres par son titre. Prenez les fonctionnalités suivantes :

```
# Distinguez vos fonctionnalités en leur donnant un titre :
Fonctionnalité : Retirer de l'argent au distributeur
Fonctionnalité : Consulter le solde de son compte au distributeur
```

Il est facile de les distinguer et de les comprendre non ?

Identifier clairement ces fonctionnalités est important : cela vous permettra de les prioriser plus facilement.

Bon, vous allez me dire qu'une petite phrase de rien du tout ça lasse la place à beaucoup d'ambiguïtés ; et vous aurez bien raison !

Vous le savez, un Produit n'est rien sans un utilisateur pour l'utiliser. **Précisez donc qui est l'utilisateur de votre fonctionnalité** :

```
# Précisez qui est l'acteur bénéficiaire de votre fonctionnalité :
En tant que rôle
En tant que administrateur du site
En tant que conseiller clientèle
En tant que ...
```

Voilà une bonne chose de faite ! **Identifiez maintenant le Service rendu par votre fonctionnalité**. Je conseille toujours de voir le service comme ni plus ni moins ce que votre commercial entend lorsqu'il dit "c'est génial, en tant que ... vous pouvez faire ça !". Par exemple :

```
# Identifiez le service délivré par votre fonctionnalité à l'utilisateur
:
Je peux un service rendu
Je peux retirer de l'argent au distributeur
```

```
Je peux connaître la quantité d'argent qu'il me reste sur mon compte
Je peux ...
```

On commence déjà à y voir plus clair. **Il ne reste plus qu'à préciser le bénéfice métier** de votre fonctionnalité. Car oui, une fonctionnalité sans bénéfice pour l'utilisateur ne sert à rien. Si vous ne voyez pas de bénéfice pour votre fonctionnalité, c'est sans doute qu'elle ne mérite pas d'être développée. Décrivez ce bénéfice ainsi :

```
# Décrivez le bénéfice fonctionnel obtenu par votre fonctionnalité :
De telle sorte que bénéfice attendu
De telle sorte que je puisse obtenir de l'argent même quand la banque
est fermée
De telle sorte que ...
```

Ce qui fait que **toute fonctionnalité peut être décrite, de manière explicite, en quatre courtes lignes** :

```
Fonctionnalité : Retirer de l'argent au distributeur
    En tant que client de la banque
    Je peux retirer de l'argent au distributeur
    De telle sorte que je puisse obtenir de l'argent même quand la
banque est fermée
```

N'importe quel membre de votre équipe comprendra ce qu'est cette fonctionnalité. N'oubliez de n'employer que le vocabulaire de votre Langue Commune pour éviter toute ambiguïté.

4.3 Des scénarios pour vos fonctionnalités

Un titre, un rôle, le service rendu et le bénéfice métier c'est déjà pas mal. Mais en général ça ne va pas suffire : vous pouvez avoir envie d'illustrer vos fonctionnalités.

C'est le rôle des scénarios. **Chaque fonctionnalité peut contenir un ensemble de scénarios. Ces scénarios doivent tous être clairs, uniques et explicites**. Ils représentent le comportement concret que vous attendez pour votre fonctionnalité. Par exemple :

```
# Un Scénario est une situation concrète de votre Fonctionnalité :
Fonctionnalité : Retirer de l'argent au distributeur

Scénario : Retirer de l'argent avec une carte valide
Scénario : Retirer de l'argent avec une carte expirée
Scénario : Retirer de l'argent sur un distributeur qui ne contient
  plus d'argent
Scénario : ...
```

La réalisation (ou non) dans votre application de ces scénarios détermine l'avancée de votre fonctionnalité.

Il est possible que ces scénarios évoluent : certains seront réalisés tôt, d'autres seront réalisés plus tard ou seront modifiés... Dans tous les cas, **jeter un oeil à l'avancée de ces scénarios vous permettra de maîtriser l'avancée de votre fonctionnalité**.

Vous vous en doutez, il peut arriver qu'une simple phrase ne suffise pas à décrire avec précision un scénario. Il sera alors temps d'y introduire un contexte, un événement déclencheur ou un résultat attendu.

Le contexte du scénario est introduit par l'expression "étant donné". Il situe les conditions d'existence de votre scénario :

```
# Un Scénario s'inscrit dans un contexte, une situation initiale :
Scénario : ...
    Etant donné que tel contexte existe
    Etant donné que que je n'ai plus d'argent sur mon compte
    Etant donné que ma carte est expirée
    Etant donné que ...
```

Toute interaction de l'utilisateur avec votre produit peut se traduire sous forme d'événement. Un événement est le déclencheur de votre scénario, qui vient juste après que le contexte initiale soit situé :

```
# Un Scénario se démarque par un événement :
Scénario : ...
    ...
    Quand tel événement déclencheur survient
    Quand je demande à retirer de l'argent
    Quand je demande le solde de mon compte
    Quand ...
```

Si chaque action a une conséquence, il en va de même pour les événements : chaque événement attend un résultat, exprimé par l'expression "alors" :

```
# Chaque Scénario aboutit à un résultat :
Scénario : ...
    ...
    Alors tel résultat est attendu
    Alors je reçois 20 euros
    Alors ma carte est aspirée
    Alors ...
```

Toute cette syntaxe, qui permet de rendre vos fonctionnalités explicites, sera comprise par vos développeurs. Et en plus elle n'est pas très compliquée, si ?

Au final, voici une fonctionnalité plus complète :

```
# Toute personne qui connaître votre Langue Commune comprend
# votre fonctionnalité :

Fonctionnalité : Retirer de l'argent au distributeur
    En tant que client de la banque
    Je peux retirer de l'argent au distributeur
    De telle sorte que je puisse obtenir de l'argent
      même quand la banque est fermée

Scénario : Retirer de l'argent avec une carte valide et un
  compte approvisionné
    Etant donné que je détiens un compte bancaire soldé de 100 euros
    Et que ma carte expire l'an prochain
    Quand je demande à retirer 20 euros
    Alors je reçois 20 euros
    Et ma carte m'est restituée

Scénario : Retirer de l'argent avec une carte expirée
    Etant donné que ma carte est expirée
    Quand je demande à retirer 20 euros
    Alors ma carte est aspirée
```

4.4 Illustrez le tout avec des exemples

Ce formalisme est utile. Il n'est en aucun cas obligatoire bien entendu, mais il a démontré son efficacité, et a l'énorme avantage d'être **clair tout autant pour les fonctionnels que pour les développeurs.**

Maintenant, ce formalisme peut manquer de concret. Qu'à cela ne tienne, vous pouvez y ajouter vos exemples. Car on est bien d'accords, **rien de mieux que des exemples**. Prenons celui-ci :

```
# Votre fonctionnalité peut contenir un exemple :
```

```
Scénario : Retirer de l'argent avec un compte approvisionné
    Etant donné que ma carte expire l'an prochain
    Quand je demande à retirer "20" euros
    Alors je reçois "20" euros
```

Bon, c'est simple, mais limité, car cela nous oblige à recopier le scénario autant de fois que nous avons d'exemples. Heureusement, notre grammaire est riche, et vous pouvez utiliser la syntaxe suivante (votre scénario devient alors un Plan du scénario) :

```
# Votre fonctionnalité peut contenir plusieurs exemples facilement :
```

```
Plan du Scénario : Retirer de l'argent avec un compte approvisionné
    Etant donné que ma carte expire l'an prochain
    Quand je demande à retirer "<montant-demandé>" euros
    Alors je reçois "<montant-reçu>" euros

Exemples :
    | montant-demandé   | montant-reçu  |
    | 20                | 20            |
    | 1000              | 1000          |
    | 50                | 50            |
```

Pratique non ? Les symboles < et > indiquent une valeur d'exemple, à rechercher dans le tableau juste en dessous.

Le tableau est dessiné grâce au caractère "pipe". Si vous ne savez pas le situer sur votre clavier, il suffit de taper `Alt Gr" + 6`, ou `Alt + Maj + L` si vous utilisez un Mac. Rassurez-vous, comme vous le verrez par la suite, il existe des outils graphiques pour vous faciliter la vie.

4.5 Travaillez en équipe

Vous être maître de votre Produit, et c'est vous qui décidez de vos Fonctionnalités. Par contre, n'oubliez pas que **le dialogue et l'échange sont primordiaux pour éviter les ambiguïtés dans votre projet**.

Lorsque vous allez rédiger vos scénarios, voici ce qui risque de se passer :

- Vous allez devoir les expliquer à votre équipe technique lorsque vous les leur fournirez
- Vous n'allez peut être pas penser à tout, et vos équipes devront vous solliciter très (trop) régulièrement pour des aspects que vous aurez oublié
- Certains de vos scénarios ne seront pas clairs ou n'emploieront pas le vocabulaire de la Langue Commune (on a toujours tendance à se trouver soi-même très clair ; ce n'est pas toujours vrai...)
- L'équipe risque de ne pas se sentir suffisamment impliquée

Ces écueils ne sont bien sûr systématiques, mais le risque est là. Pourquoi prendre ce risque alors que la solution est simple ?

Il suffit en effet de faire participer toutes les équipes lors de la rédaction des scénarios. Attention, il ne s'agit pas de laisser n'importe qui faire n'importe quoi... Non, **c'est a vous d'expliquer votre besoin**, dans la Langue Commune, et **ensuite vous allez**

ensemble le traduire en scénarios, de telle sorte que tous comprennent clairement ce que vous attendez d'eux.

Pour résumer, écoutez les remarques de vos équipes lors de la rédaction des scénarios. Idéalement, vous devriez entendre des "oui mais qu'est-ce qui se passe si..." et "je ne comprend ce que devient...". Si c'est le cas, le pari est réussi !

Dans tous les cas, gardez vos scénarios clairs et concis. Personne n'a envie de lire des scénarios de 20 ou 30 lignes...

> "*Ecrivez vos scénarios avec l'équipe technique ; le temps passé sera compensé par le gain de compréhension et l'inutilité d'avoir à faire des va-et-vient incessants au cours du développement*"

4.6 Des assistants visuels

Vous aviez déjà le vocabulaire (la Langue Commune), vous voici désormais avec la grammaire nécessaire pour exprimer votre besoin. Mais on est tous les mêmes : personne n'aime pas la grammaire !

Après tout, si votre véritable objectif est de vous focaliser sur l'expression de votre besoin, sur la description du comportement de votre Produit, à quoi bon se concentrer sur la manière de l'exprimer ?

Pour vous simplifier la vie il existe un certain nombre d'outils qui vont vous assister dans la rédaction de vos Fonctionnalités. Ils vont vous permettre de rédiger vos features selon une approche plus agréable, tandis même qu'ils vont présenter à vos développeurs votre besoin sous forme de fichiers clairs qu'ils pourront directement exploiter.

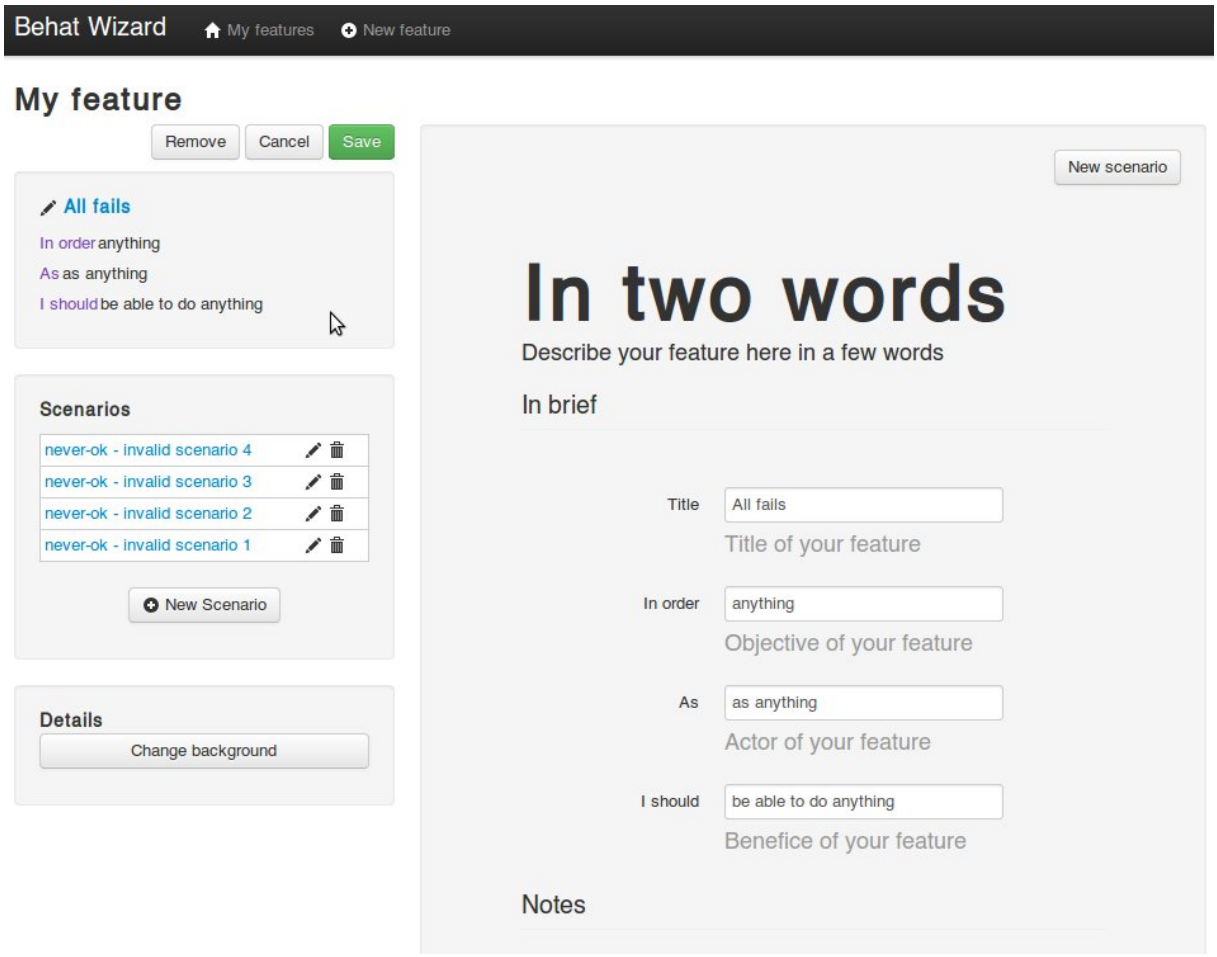

Figure 4.1 Il existe des assistants visuels pour faciliter votre expression de besoin

Il existe plusieurs outils, qui ont tous leurs avantages. On peut penser par exemple à BehatViewer, ou à BehatWizard... N'hésitez pas à demander vos équipes techniques de vous installer ces outils.

Chapitre 5

Un peu de recul sur le Développement piloté par le Comportement

5.1 Votre fonctionnalité se spécifie elle-même

Vous savez désormais décrire votre besoin dans une Langue et une grammaire compréhensible par tous les acteurs de votre projet.

Vous avez peut-être déjà rencontré cette difficulté : au fur-et-à-mesure de la vie d'un projet, la documentation, fruit d'un travail difficile et chronophage, se détache généralement de la réalité.

Cette difficulté a en réalité une origine simple : lorsque l'on ajoute une nouvelle fonctionnalité, mettre à jour la documentation prend parfois plus de temps que la réalisation même de la fonctionnalité.

Maintenant, examinez ceci : avec le Développement piloté par le Comportement, c'est-à-dire grâce à la Langue Commune, grâce une grammaire spécifique, la documentation fonctionnelle est écrite en même temps que les spécifications. Et oui! Vos Fonctionnalités sont aussi de la documentation.

Il est assez facile d'imaginer, par exemple, prendre l'ensemble de vos fichiers de Fonctionnalités et les transformer en un fichier .pdf, ou encore d'en faire un manuel en ligne. C'est ce que font, par exemple, Relish (https://www.relishapp.com) (hébergé, payant) ou Pickles (https://github.com/picklesdoc/pickles) (gratuit, à installer soi-même).

Bien plus, aucun changement fonctionnel de code source ne peut être réalisé sans que l'un de vos fichiers de Fonctionnalités n'ait été modifié. C'est le principe même du Développement piloté par le Comportement : vous ne pouvez pas avoir de delta entre votre expression de besoin et ce que vos développeurs vous livreront.

Par conséquent, vous l'aurez compris, vous avez ainsi un énorme avantage : **les Fonctionnalités que vous avez constituées reflètent en permanence l'état réel de votre Produit**.

Si ce n'est pas le cas, c'est :

- que vous avez modifié votre Fonctionnalité sans en rédiger les Scénarios directement avec l'équipe technique. Votre erreur sera d'avoir rompu le dialogue avec vos équipes,
- ou que les équipes n'auront pas encore réalisé votre demande de changement. Il faudra dans ce cas prioriser ce changement parmi l'ensemble de fonctionnalités qu'ils ont à développer.

> *Spécification = Documentation = toujours à jour*

5.2 Servez-vous de vos fonctionnalités pour prioriser

Pensez à ceci : vous pouvez, si vous le souhaitez, associer un état à chacun de vos fonctionnalités et scénarios. Par exemple :

- En attente
- En cours de développement
- Réalisé

Vous verrez qu'il existe des outils pour vous aider dans cette tâche. Grâce à ces indicateurs, vous disposez de la possibilité de visualiser à chaque instant l'état de votre produit.

Si, donc, vous connaissez la phase de développement de chacun de ces scénarios et fonctionnalités, pourquoi ne pas vous servir de cette information pour changer leur priorité ?

Vous le savez, votre Produit a besoin d'être confronté rapidement au feedback de vos utilisateurs. Profitez en pour faire en sorte de vous faire livrer régulièrement des versions simplifiées de votre projet ; regardez ce qui plaît à vos utilisateurs... C'est justement ce qui plaît qu'il va falloir développer en priorité dans votre projet, tout le reste peut être remis à plus tard.

Vous l'aurez compris : ce découpage en fonctionnalités, puis en scénarios, va vous permettre de prioriser intelligemment votre projet. Bien plus, cela va vous pousser à abandonner certains points pour en introduire de nouveaux, plus pertinents pour vos utilisateurs finaux ; et puisque cela fait partie de votre Produit même, ce changement fonctionnel sera à la fois rapide et peu coûteux !

> *Vous avez un aperçu en temps réel de l'état de votre Produit*

5.3 Mesurez le chemin parcouru, pas l'énergie dépensée

C'est souvent la même chose : on surveille le temps passé, les tâches réalisées... Mais on a tous le même problème : passer beaucoup de temps ne veut pas dire obtenir un résultat. **Pourquoi mesurer ce temps passé si vous n'avez pas le résultat attendu ?** Ce qui compte, ce sont uniquement les fonctionnalités.

Les outils classiques vous proposent de découper chaque besoin en tâche, puis chaque tâche en temps. Or une tâche n'a aucun sens fonctionnel ! Vraiment ! C'est comme si vous demandiez à une chauffeur routier combien de litres d'essence il a consommé ; certes, c'est important d'un point de vue financier, mais ce qui compte c'est quand même de savoir s'il a pu livrer tous ses clients, vous ne croyez pas ?

Dans un projet informatique c'est pareil : focalisez-vous sur les fonctionnalités développées. Après tout, c'est ce qui compte vraiment... Uniquement !

Bien entendu, peut-être que les équipes techniques, elles, travailleront en tâches, mais :

- rien ne les y oblige.
- ça ne vous concerne pas, seules les fonctionnalités vous intéressent

On n'a jamais vu un client obtenir un bénéfice métier parce qu'une équipe a accompli une tâche. Non, le bénéfice ne s'obtient qu'en fournissant un service, autrement dit une fonctionnalité.

> *Le client final n'est pas content parce qu'une tâche est accomplie ; il est content parce qu'une fonctionnalité lui est offerte et qu'elle répond a son besoin*

5.4 Une bonne fonctionnalité sert la Vision Produit

Prenez le réflexe : demandez-vous systématiquement si votre fonctionnalité sert votre Vision. Si ce n'est pas le cas ... Et bien oubliez la !

A contrario, ne vous limitez pas à un cadre. Après tout, ce qui compte vraiment ce n'est pas la somme de fonctionnalités, c'est de réussir à satisfaire votre Vision.

Prenez GMail, de Google, par exemple. Chaque fonctionnalité est sélectionnée pour répondre à cette vision simple : "permettre à chacun de communiquer facilement avec ses proches". Peu importe le cadre ; du moment qu'une fonctionnalité sert cette vision, elle est bonne.

Sans Vision, GMail ne serait qu'une application, c'est à dire un simple webmail. Mais c'est justement pour cela que GMail c'est :

- un webmail
- un service de communication instantané (GTalk)
- une application mobile
- un tchat vidéo (hangout)
- un rappel des anniversaires de vos amis
- un carnet de contacts
- ...

Tous ces services desservent la même vision. C'est ce qui fait que GMail plaît autant. Et c'est cette vision qui assure la cohérence de l'ensemble : **il vous faudra parfois refuser certaines fonctionnalités**, même si à court terme elles semblent rentables.

Si elles ne répondent pas à la même vision, ces fonctionnalités vont rendre votre produit incohérent et illogique. Pourquoi dépenser de l'argent pour rendre votre Produit illogique ? **Tout le monde aime les choses simples, claires et cohérentes** ; pire : si vous enlevez cette clarté, vos développeurs eux-mêmes risquent de ne plus comprendre ce que vous voulez !

Si vous tenez absolument à ajouter une fonctionnalité qui sert une autre vision, créez une autre projet, avec d'autres équipes, mais ne la greffez par artificiellement à celui-ci.

> *Respecter scrupuleusement une Vision assure la cohérence et la clarté du Produit*

5.5 Oubliez l'interface graphique

Fatigué ? Pourquoi ne pas faire un petit jeu ? Vous connaissez Google Calendar ? mais si, vous savez, le calendrier de Google, très pratique, que vous retrouvez sur calendar.google.com...

A votre avis :

Google Calendar serait-il toujours le même produit :	
Si le site web n'affichait plus la date du jour ?	☐ oui ☐ non
Sans page web pour y accéder ?	☐ oui ☐ non
S'il n'affichait pas de calendrier mensuel	☐ oui ☐ non
S'il ne vous permettait pas d'ajouter un rendez-vous en cliquant sur le bouton "nouveau rendez-vous" ?	☐ oui ☐ non
Si Internet n'existait pas ?	☐ oui ☐ non

A risque de surprendre, la réponse est partout la même : "oui" ! Oui, même si Internet n'existait pas !

Arès tout, qui nous dit que le web que nous connaissons ne va pas disparaître ? Google Calendar pourrait très bien n'exister que sur téléphone mobile. Mieux, sans électronique. Arès tout, si Google avait existé au début du XXème siècle, il lui aurait fallu trouver un autre moyen pour délivrer ses services.

Considérez en effet cette Vision de Google Calendar : "permettre à chacun de connaître facilement les événements à venir qui le concernent". Pas besoin de page web, d'application mobile... Un simple courrier postal pour ajouter un événement, puis quelqu'un qui vient frapper à notre porte à chaque fois qu'un événement va arriver... C'est suffisant, non ?

Certes, commercialement ce n'est pas l'idéal, mais même si Google Calendar changeait pour désormais envoyer quelqu'un frapper chez nous à chaque événement, le Produit serait inchangé ; seul le support matériel serait modifié. Autrement dit : **le support n'a aucune importance lorsque vous décrivez votre Produit !**

Quand vous décrivez votre produit, écrivez vos fonctionnalités de sorte à ce qu'elles soient valables, même sur internet, même sur application mobile, et même par courrier postal. c'est la seule façon de pérenniser votre projet. Si vous avez un site web, vos Fonctionnalités et Scénarios ne doivent pas changer d'un pouce lorsque vous décrivez votre application mobile.

> *Focalisez-vous sur le Comportement de votre Produit, et laissez l'interface graphique aux ergonomes et aux graphistes.*

5.6 Le scénario n'est pas un critère d'acceptation

Considérez ce scénario, mis en évidence par Liz Keogh:

```
Scénario : pouvoir acheter un animal domestique adulte
    Etant donné qu'un chiot est trop petit pour être vendu
    Quand j'essaye de l'acheter
    Alors je suis informé qu'il ne peut être vendu car trop jeune
```

Bien. Bien ? Non, pas vraiment... Vous n'avez pas décrit votre besoin, vous venez en réalité de fournir un Critère d'acceptation générique à vos développeurs. Leur réaction face à ce scénario ne peut être que "d'accord, mais à quel âge par exemple le chiot ne peut pas être vendu ?".

Tant qu'il reste des questions en suspend, quelque chose ne va pas. N'oubliez pas que **vous décrivez vos Fonctionnalités et vos Scénarios, non pas d'abord pour valider un travail fourni, mais avant tout pour être sûr que tout le monde a bien compris votre besoin** et pour éviter les allers et retours inutiles.

Chaque scénario doit contenir ses propres exemples. Notre scénario pourrait être changé en :

```
Scénario : pouvoir acheter un animal domestique adulte
    Etant donné qu'un chiot ne peut être vendu avant qu'il n'ait "2 mois"
    Et que "Médor le chien" a actuellement "1 mois"
    Quand j'essaye d'acheter "Médor le chien"
    Alors on doit me dire "Médor le chient est trop jeune. Veuillez
revenir dans 1 mois"
```

C'est un peu plus explicite non ? Et il reste moins de questions. De cette manière vous limitez les aller-et-venues inutiles et exprimez votre besoin clairement.

> *Si on vous demande: "Pouvez-vous me donner un exemple où cette situation arrive?", vous n'avez pas rédigé un Scénario mais un Critère d'acceptation. Effacez le.*

5.7 Les principales causes d'échec du BDD

Je vais être franc : cette voie de communication, le Développement Piloté par le Comportement, n'est pas une recette miracle et peut échouer.

Vous pouvez, comme pour tout, y perdre de l'argent si vous échouez à appliquer ce principe fondamental : **ce qui compte c'est de réussir à ce que chacun comprenne clairement votre besoin vis-à-vis de votre Vision**.

Je pense que la première cause d'échec vient de la **difficulté à identifier clairement les fonctionnalités d'un Produit**. On a tous en tête des idées géniales, le site web qui va tuer Facebook et eBay ; le vrai souci est de savoir comment, dans la pratique, ce site web va rendre service aux utilisateurs, et quelles sont les fonctionnalités qu'on va lui offrir.

La seconde difficulté réside dans notre propension à avoir besoin d'un support visuel pour réfléchir à quelque chose. **Il n'est pas simple de ne se focaliser uniquement sur le Comportement d'un Produit, sans se soucier de son apparence**, de son interface graphique. Ce sera d'autant plus vrai si vous avez un profil technique ou si vous êtes très familiarisé avec les nouvelles technologiques.

Pour résoudre cette difficulté, tentez systématiquement de vous représenter votre produit sur différents supports : un ordinateur, un téléphone, mais aussi par courrier ou par pigeon voyageur ! Bref, faites abstraction totale du support matériel.

Ne vous focalisez pas sur le cheminement qui permet à votre application ou site web de délivrer un service. Partez en sens inverse depuis l'utilisateur : quel comportement peut-il avoir avec mon Produit ? Comment mon Produit peut-il l'aider ?

Enfin, et c'est le plus important : **vous aurez besoin de vous faire conseiller, surtout si ce n'est pas votre premier projet informatique**. Il est quasiment impossible de réussir du premier coup à se défaire des habitudes antérieures.

Il existe aujourd'hui de nombreuses sociétés de conseil en Agilité qui sauront vous conseiller, surtout dans la démarche de dialogue qu'il vous faudra engager avec vos équipes. L'investissement risque fort d'en valoir la peine... Si vous n'en avez pas les moyens vous pouvez (très sérieusement) effectuer en parallèle ce dialogue auprès de personnes totalement novices dans le domaine fonctionnel abordé et peu familières des nouvelles technologies : vos parents, grand-parents, enfants ou vos amis ermites.

> *Il n'existe pas de recette miracle. Faites abstraction de tout support, oubliez vos habitudes et demandez de l'aide*

This Page Intentionally Left Blank

Chapitre 6

Recette et maintenabilité

6.1 Chaque recette fonctionnelle peut être automatisée

La démarche et la syntaxe du Développement piloté a l'avantage d'être claire et de faciliter la communication.

J'ai une excellente nouvelle pour vous : **cette syntaxe va vous permettre d'automatiser la recette de votre application**. Vraiment !

Et je nous vous parle pas d'outils, si vous en avez déjà utilisé, qui vont enregistrer puis reproduire les clics et saisies que vous faites sur une page web (comme avec Selenium (seleniumhq.org/...) . Non, surtout pas ! Je vous parle d'outils pour lesquels vous n'avez rien à faire...

Après tout, **si vous avez fait l'effort de respecter une syntaxe spécifique pour décrire vos Scénario, un programme informatique doit être capable de les interpréter**.

Il existe aujourd'hui deux principaux outils sur le marché : Cucumber et Behat. Le choix de l'outil dépendra du socle technique choisi par vos développeurs pour votre Projet.

Concrètement, ces outils vont transformer vos Scénarios en code informatique. Ce code informatique sera adapté par vos développeurs qui lui donneront un sens et s'assureront qu'il signifie bien ce que vous avez demandé.

Par la suite vous disposerez d'une interface où chaque fonctionnalité, chaque scénario et chaque étape seront, ou bien verte (terminée), ou bien jaune (pas encore développé), ou bien rouge (ne répond pas au besoin initial).

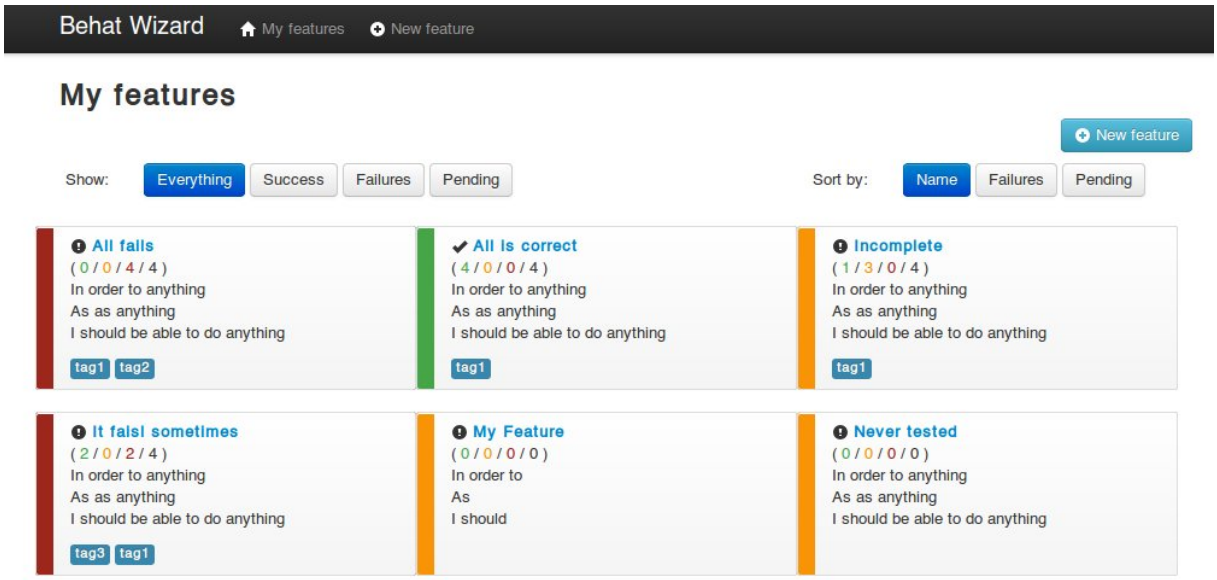

Figure 6.1 Retrouvez toutes vos fonctionnalités au sein d'une même interface.

Utiliser de tels outils a un coût puisqu'il exige un travail de traduction par l'équipe technique. Ce coût peut être important, cela peut demander du temps non négligeable, en plus d'une montée en compétence des équipes techniques (très rapide en général),

Cependant, en utilisant ces outils :

- Les équipes techniques savent si oui ou non elles ont répondu au besoin
- Les équipes techniques sont rassurées dans leur travail
- Vous connaissez exactement l'avancée de votre projet
- Vous pouvez sans risque gérer votre besoin de changement : si une autre fonctionnalité est "cassée" par ce changement, vous le savez tout de suite et pouvez agir en conséquent
- Vous augmentez le niveau de qualité de votre projet. C'est autant de gains lors de la vie du Produit et de sa maintenance, qui sans cela peut être très coûteuse
- Vous-même savez en un instant si le livrable est conforme à vos attentes

Rien ne vous oblige à utiliser ces composants d'automatisation de recette. Ne pas les utiliser peut certes être dommage, mais n'enlève rien à la démarche générale.

> *Facilitez le changement en automatisant la recette fonctionnel*

6.2 Assurez-vous que les équipes techniques automatisent leurs recettes techniques

Tester automatiquement qu'un livrable correspond à vos **attentes** fonctionnelles c'est bien ; mais être certain de la **fiabilité** de votre produit c'est encore mieux !

La pratique du développement piloté par le comportement, c'est-à-dire de tout ce que l'on a présenté ici (les fonctionnalités, les scénarios...) nécessite quelque chose de fondamental : que **vos équipes mobilisent leur énergie sur votre besoin, pas sur leurs bugs**.

Pour cela il n'y a pas énormément de solutions : **vos équipes techniques doivent créer des tests automatisés de leur code source**.

C'est un impératif fort : c'est le seul moyen pour qu'elles ne vous disent pas "nous ne pouvons pas développer cette fonctionnalité, c'est impossible car on risque de casser tout l'existant et d'introduire des bugs". Non, avec des tests automatisés, une équipe compétente ne peut pas vous répondre ça. Elle vous dira peut-être que la nouvelle fonctionnalité a un coût, mais aucune fonctionnalité ne doit être impossible à implémenter parce qu'elle risque de casser l'existant.

Si quelque chose casse lorsque vous introduisez du changement, vous perdrez beaucoup d'argent : il va falloir identifier (et ça peut être très long!) et corriger ce qui a cassé, corriger la nouvelle fonctionnalité, la tester, ajouter des ressources pour le support applicatif...

Tous ces coûts, très importants sur le long terme, peuvent être réduits en laissant à court terme une plus grande marge de manœuvre aux équipes techniques pour fiabiliser leur code source et en écrivant des tests automatisés. Ne soyez pas un mauvais économe : laissez leur le temps qu'il faut pour vous fournir un produit plus stable et plus fiable. Vous y gagnerez sur le long terme.

> *Laissez le temps aux développeurs de fiabiliser leur code : aucune fonctionnalité ne doit être impossible à implémenter sous prétexte qu'elle risque de casser l'existant*

This Page Intentionally Left Blank

Chapitre 7

Le mot de la fin

7.1 Parlons la même langue

Nous avons discuté des idées, des concepts... Il est temps d'établir notre Langue Commune.

Produit

Concept dont l'application sera à terme la réalisation. Ce concept vise à rendre possible la Vision.

Vision

Guide et oriente votre projet. Son but est de changer ce qui, dans la vie de vos utilisateurs, peut les gêner, leur manquer ou les insatisfaire.

Bénéfice fonctionnel

Confort métier tiré par les utilisateurs de votre Produit.

Langue Commune

Langage clair pour l'ensemble des acteurs d'un projet. Ce langage est défini en commun.

Cartographie du Produit

Représentation visuelle simple du Comportement de votre Produit. Cette représentation peut concerner l'ensemble ou une partie de votre Produit.

Fonctionnalité

Besoin métier spécifique. Elle est représentée par un texte court qui communique ce besoin métier au reste de l'équipe. Elle concerne le service rendu à un utilisateur, et décrit le bénéfice fonctionnelle qui en est tiré.

Scénario

Illustration des situations dans lesquelles se manifeste une Fonctionnalité. Elle est représentée par une situation initiale, un événement et un résultat attendu.

Exemple

Illustre un Scénario de manière concrète. Il enlève toute ambiguïté possible sur une étape d'un scénario ou sur le scénario lui-même.

Livrable

Application concrète livrée au client. Il délivre le Comportement raconté par les Fonctionnalités et les Scénarios. Il est possible de fournir un livrable régulièrement au cours de la vie du Projet.

Recette

Opération qui contrôle que le Livrable correspond au Produit, en se basant sur les Fonctionnalités et les Scénarios. Peut être automatisée avec des outils spécialisés.

7.2 Cartographie du Développement piloté par le Comportement

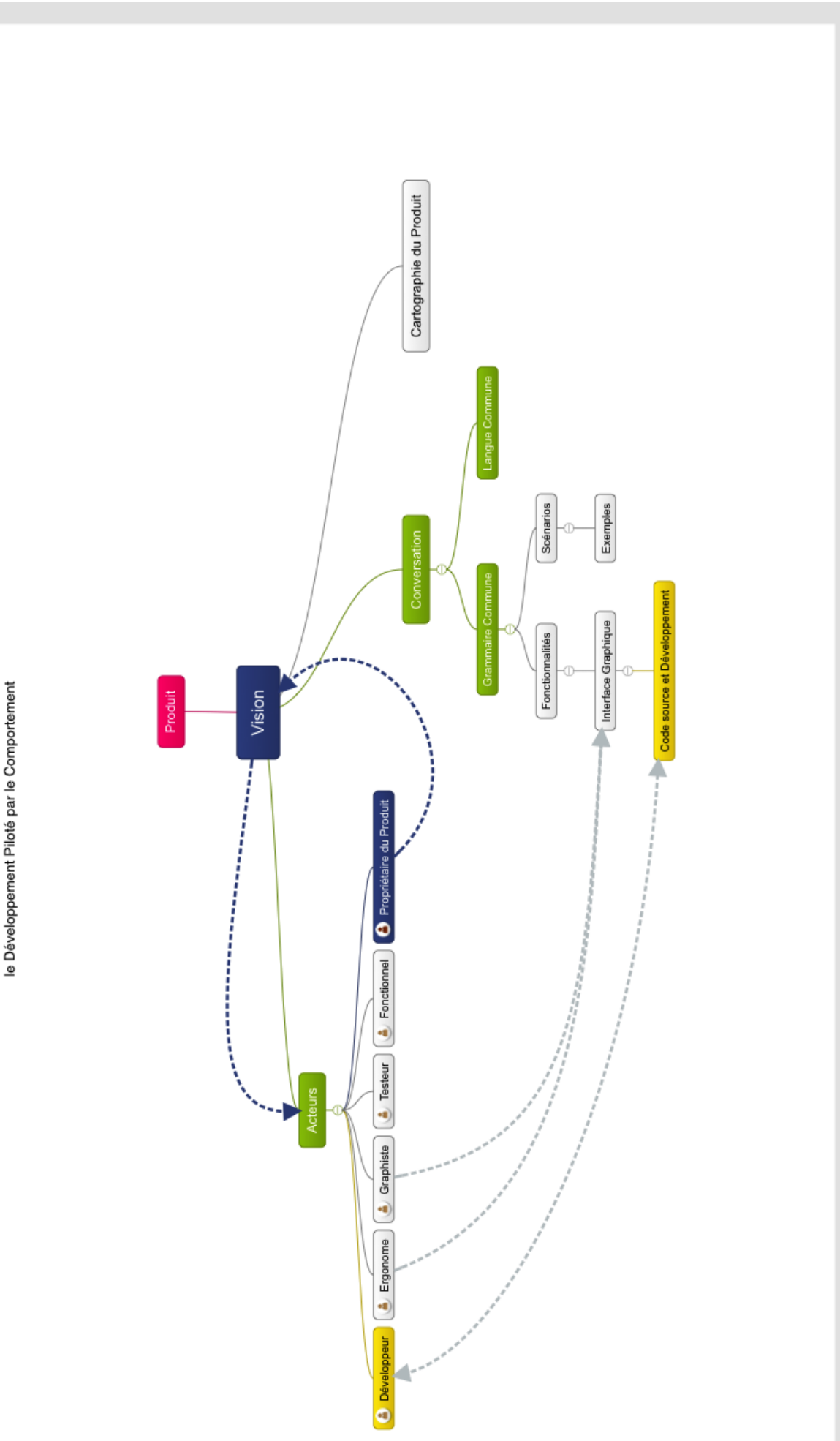
le Développement Piloté par le Comportement
Produit
Vision
Cartographie du Produit
Conversation
Langue Commune
Grammaire Commune
Scénarios
Exemples
Fonctionnalités
Interface Graphique
Code source et Développement
Acteurs
Propriétaire du Produit
Fonctionnel
Testeur
Graphiste
Ergonome
Développeur

Figure 7.1 Cartographie du Développement piloté par le Comportement

7.3 Les 10 commandements pour vous mettre au Développement piloté par le Comportement

1) Votre Vision est le seul guide du projet

Chaque fonctionnalité, chaque ressource consacrée au projet, tout doit servir cette Vision. Elle est le point de repère et la motivation de chacun des acteurs de votre Produit.

2) Créez une Langue Commune.

Un développeur ne parle pas la même langue que vous. Créez un référentiel et un vocabulaire commun ensemble, et utilisez la syntaxe proposée par le Développement piloté par le Comportement, claire pour la majorité des gens.

3) Concevez vos scénarios ensemble.

C'est le seul moyen de retirer toutes les ambiguïtés qui peuvent exister entre vous, votre produit et vos équipes. Ne laissez personne rédiger un scénario tout seul.

4) Vos scénarios ne concernent pas l'interface graphique

Faites abstraction du support matériel de votre Produit : site web, application mobile... Ce qui compte vraiment c'est le comportement de votre Produit.

5) L'automatisation des recettes fonctionnelles n'est qu'un bonus

Automatiser des recettes fonctionnelles n'a de sens uniquement si au préalable vous avez réussi à faire comprendre votre besoin. Sans dialogue ni échanges, les tests automatisés testeront que l'application fait bien quelque chose, mais pas forcément ce que vous attendez.

6) L'automatisation des tests techniques est primordiale

Laissez le temps aux équipes techniques et poussez-les à écrire des tests automatisés de code source (test unitaire). C'est le seul moyen pour eux de réaliser vos demandes légitimes de changements sans faire exploser vos coûts.

7) KISS : Keep It Simple and Stupid

Gardez vos scénarios simples et explicites, et n'employez pas dix mots quand cinq suffisent. Si votre scénario est long, c'est probablement qu'il recouvre un domaine fonctionnel trop large : découpez-le.

8) Soyez conscient que vos scénarios changeront

En confrontant régulièrement votre Produit aux avis des utilisateurs, une grande partie de vos scénarios va changer, voire disparaître. Acceptez-le, et faites en sorte que les équipes techniques l'acceptent en les faisant participer à ce processus de changement.

9) Utilisez des exemples

Vous ne rédigez pas des tests d'acceptation, mais établissez une communication pour décrire votre besoin. Illustrez votre besoin par des exemples, pourquoi pas avec humour, pour effacer toute ambiguïté.

10) Le Développement piloté par le Comportement n'est pas une recette miracle

La recette miracle n'existe pas. Mais c'est au moins un bonne démarche, éprouvée, pour vous faire comprendre et faciliter la communication entre fonctionnels et techniques. Elle aidera chacun des acteurs de votre projet à comprendre son rôle et ce que vous attendez de lui ; il ne reste plus à chacun qu'à être un bon acteur.

Développeurs, faites-vous plaisir !

This Page Intentionally Left Blank

Le développement est un plaisir

"Le client ne sait jamais ce qu'il veut". Comment-pouvez-vous, dans ce cas, faire en sorte qu'il soit satisfait ? Êtes-vous condamné à délivrer des logiciels, sites web ou applications dont le client ne sera pas réellement satisfait ?

Et vous-même, êtes-vous vraiment obligé de subir les contraintes fonctionnelles, de brider vos compétences techniques ? Après tout, le métier de développeur est passionnant, et mérite qu'on y prenne le maximum de plaisir ! Et c'est tellement décevant d'entendre cette fameuse phrase : "ce n'est pas ce que je voulais"...

Il m'arrive de poser cette question aux personnes que je rencontre : "avez-vous, au moins une fois, participé à un projet informatique où le projet a été livré à l'heure, et où, non seulement le client, mais aussi le développeur, le testeur, l'intégrateur... ont été pleinement satisfaits et y ont pris plaisir ?". A l'heure où j'écris ces lignes, seule une personne m'a répondu "oui".

Quoi ?! Une seule personne ? Mais pourtant tous les projets devraient se passer comme ça. Tous les projets devraient être livrés à l'heure, devraient être intéressants, enrichissants... Toutes les personnes qui travaillent sur un projet devraient y prendre plaisir. Sinon, à quoi bon travailler ?

L'émergence des méthodes agiles, ces dernières années, a permis de remettre l'humain, et ses valeurs, au centre des projets. Je vous propose de découvrir ici l'autre versant, la face technique, des méthodes agiles : le Développement piloté par le Comportement.

This Page Intentionally Left Blank

Chapitre 1

Le client ne sait pas ce qu'il veut. Sauf si... vous communiquez.

1.1 Le client doit vous fournir sa Vision

J'ai souvent été confronté, en tant que développeur, et sur des projets de toutes tailles (très grosses applications financières, petits intranet, sites web...) à cette situation : des clients étaient incapables de me décrire ce qu'ils voulaient que je réalise pour eux.

A chaque fois, dans ce cas, il m'a fallu moi-même interpréter leurs souhaits, à partir de captures d'écrans de sites concurrents, de discussions interminables... Et pour un résultat pas toujours très heureux.

A quoi la faute ? Au client ? Pas forcément : après tout, ce n'est pas parce qu'on est patron d'entreprise ou que l'on a un besoin bien précis que l'on sait comment résoudre ce besoin, ou l'exprimer clairement.

Est-ce la faute du développeur, qui semble incapable de comprendre ce qu'on lui demande ? Personnellement j'ai toujours essayé de faire de mon mieux, même si je suis conscient de ne pas avoir toujours été à la hauteur. Et je crois que la plupart des développeurs fait de même.

Alors ? Et si personne n'était en tort ? En réalité, la plupart des projets informatiques échouent car ils ne sont pas motivés par une Vision. Certes on veut délivrer un super site web, avec plein de fonctionnalités qui vont tuer la concurrence, on veut faire mieux que son voisin... Mais on oublie l'essentiel : un projet doit voir un but.

Quel est ce but ? Peu importe : changer le monde, rendre service à une catégorie de personne, se faire de l'argent... Tout est bon à prendre, du moment que cet objectif, cette Vision, reste le seul et unique maître du projet.

On le comprend bien alors : ce n'est pas parce qu'un concurrent propose un service qu'il faut le recopier. Non, on propose un nouveau service parce qu'il sert la Vision du projet.

Vous l'aurez compris : la Vision est essentielle, et c'est justement le rôle de votre client de la fournir. Sans Vision, le projet est condamné à être un échec, ou au mieux une semi-réussite.

Votre client doit impérativement vous fournir cette Vision. S'il en est incapable, insistez pour qu'il soit aidé : faites lui lire le premier tome de ce livre, faites le coacher par une société spécialisée... Sans cela, vous ne pourrez pas travailler efficacement avec lui et ne prendrez pas autant de plaisir que vous le méritez dans votre travail.

> *La plupart des projets informatiques échouent car ils ne sont pas motivés par une Vision.*

1.2 Votre Client ne parle pas la même langue que vous

Vous l'avez vécu : votre client et vous ne vous comprenez pas toujours. Pour vous, un "réseau social" c'est une plate-forme communautaire, avec des amis, relations, des groupes, des flux de messages... Bref c'est Facebook.

Pour votre client, un "réseau social" c'est (par exemple) un espace où des employés font des demandes de documents, découvrent les actualités de l'entreprise, échangent des informations sur les commandes... Bref, c'est un intranet.

Attendez... Mais comment donc voulez-vous réussir à satisfaire votre client si vous ne vous n'employez pas le même vocabulaire sur des choses si fondamentales ?

Ajoutez à cela que votre client va toujours avoir tendance à faire deux choses :

- employer des acronymes, sigles et autre vocabulaire fonctionnel
- employer des termes techniques ("base de données", "formulaire", "sauvegarder") sans savoir précisément ce qu'ils signifient

La première chose à faire lorsque vous démarrez un projet avec un client, c'est donc de vous créer un vocabulaire commun. Il va falloir inventer une nouvelle langue, qui ne laisse plus la place aux ambiguïtés, où chaque mot n'a qu'une seule signification.

J'ai une mauvaise nouvelle pour vous : ce nouveau vocabulaire, c'est à vous de l'apprendre, pas à votre client. Cela vous évitera des débats interminables et vous permettra de mieux comprendre le besoin du Client.

Si votre client, lorsqu'il dit "bus", parle en réalité d'un véhicule avec des ailes et que vous retrouvez dans un aéroport, ne lui dites pas qu'il a tort. Non, désormais, lorsque vous parlerez avec lui, vous emploierez le mot "bus" pour désigner ce que LUI entend par "bus".

Bien entendu, ce vocabulaire commun, vous devez le constituer ensemble. Si votre client parle de "bus", et que c'est important dans le projet, discutez-en ensemble, et mettez noir sur blanc une définition simple de ce qu'il entend par là.

Faites de même pour chaque expression que votre client emploie régulièrement. De cette manière vous n'aurez plus aucune ambiguïté dans votre quotidien.

C'est le postulat de base du Développement piloté par le Comportement : vous devez élaborer avec votre client une Langue Commune.

1.3 Engagez-vous à livrer régulièrement

Quoi de plus frustrant, après de longues semaines de travail acharné, d'entendre un si triste "Mais ce n'est pas du tout ce que j'avais demandé" ?

Et même un simple "ne pourrait-on pas juste changer..." peut s'avérer totalement démoralisant, compte-tenu de tout ce qu'il faudra refaire techniquement pour gérer cette demande, et sachant surtout combien cela aurait été simple si seulement on vous avait prévenu quelques semaines à l'avance.

Comment éviter d'être touché par ce malheureux constat d'échec, ou de semi-réussite ?

La réponse est en théorie simple : il suffit de livrer son travail très régulièrement au client, de sorte qu'il puisse rapidement s'apercevoir le plus tôt possible que le projet n'est pas parti dans la direction à laquelle il s'attendait.

Cette démarche est celle des méthodes agiles, particulièrement de Scrum. Le principe est le suivant : plutôt que de livrer votre projet une bonne fois pour toute (lorsque le travail est terminé), vous vous engagez à livrer des bouts de fonctionnalités très régulièrement (une fois toutes les deux semaines par exemple).

C'est ce qu'on appelle le développement par itérations, par opposition au cycle en V. Vous développez grossièrement une fonctionnalité, puis vous l'affinez, l'affinez encore si besoin, jusqu'à arriver au résultat final. De cette manière le client peut réorienter le projet à chaque itération sans que n'en soyez affecté négativement.

Imaginez, par exemple, que votre travail consiste à peindre le célèbre tableau La Joconde. Avec la méthodologie classique, votre devriez travailler de manière à finir

chaque fonctionnalité de bout en bout, puis, une fois qu'elle est entièrement terminée, passer à la suivante.

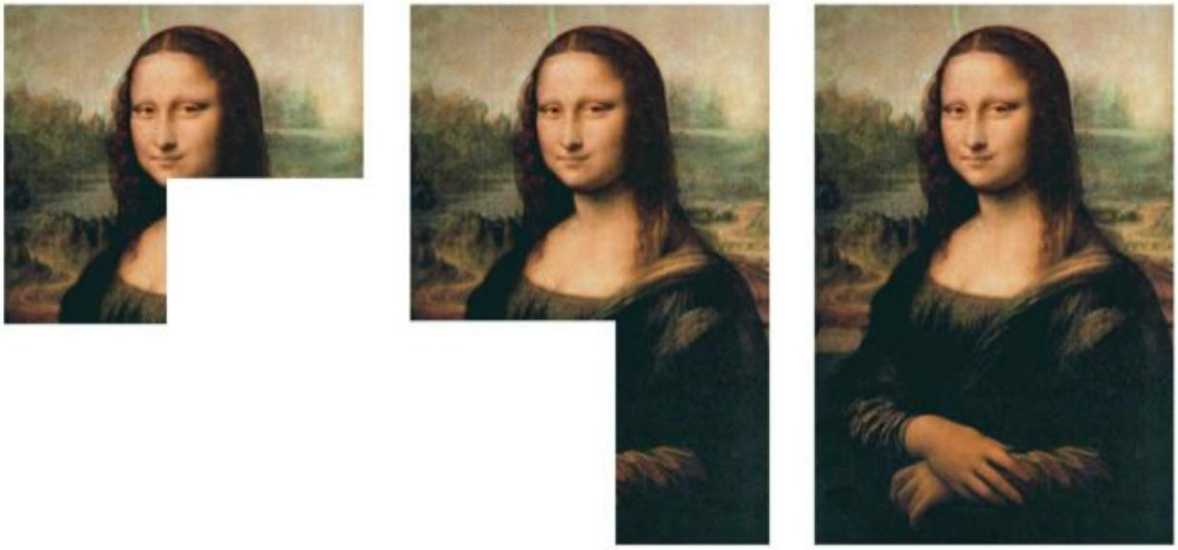

Figure 1.1 Méthodologie classique : le travail et découpé en lots, chaque lot est totalement réalisé avant de passer à la suite. Le changement fonctionnel en cours de route est difficile.

Avec les méthodes agiles la démarche est inverse : vous esquissez d'abord les traits de ce que vous avez à développer, afin de permettre au client d'avoir un aperçu du produit final et de bénéficier rapidement du feedback des utilisateurs, puis vous affinez, selon la priorité de chaque fonctionnalité : une fonctionnalité plus complète par là, un autre lors de l'itération suivante...

Figure 1.2 Méthodologie agile : le tableau s'affine petit à petit. Le changement fonctionnel est facile, même en cours de route

Bien entendu, cela ne va pas sans un changement des méthodologies de travail : il va falloir découper les fonctionnalités pour faire en sorte qu'elle puisse être développées rapidement, il faut optimiser les recettes (pourquoi ne pas en profiter pour utiliser des tests automatisés, comme des tests unitaires ?)...

Mais, croyez-moi, ces efforts valent la peine : non seulement les relations avec votre client / patron seront de plus en plus saines (le projet devient totalement transparent pour tous), mais en plus vous arriverez vite à livrer du code fonctionnel régulièrement.

Et ô combien cela fait plaisir de voir que l'on avance ! Cela vous permettra même de gérer plus facilement les demandes de changements fonctionnels : il est toujours plus facile de revenir sur deux semaines de travail que sur trois mois, et c'est bien moins démoralisant.

Je vous conseille même de faire quelque chose qui peut sembler assez étrange : à chaque fois que vous livrerez un lot fonctionnel, n'hésitez pas à présenter, pendant une heure ou deux, le fruit de votre travail à votre client. Oui, comme un commercial ! Prenez même un vidéo-projecteur.

Calez systématiquement une réunion à chaque fin d'itération, et prenez la parole : montrez ce que vous avez fait, et soyez-en fier ! Après tout, si vous avez donné le meilleur de vous même, il est légitime que tout le monde sache de quoi il était question.

> *Prenez la parole ! N'entendez plus jamais le fameux "Ce n'est pas ce que j'avais demandé".*

1.4 Adoptez le point de vue de l'utilisateur final

Chaque projet informatique, qu'il s'agisse d'un site web, d'un intranet, d'une application... tout projet dessert un objectif : rendre service à quelqu'un. Il faut vous mettre dans la peau de cette personne.

Lorsque vous développez une boutique eCommerce, le service est rendu à un potentiel acheteur; lorsque vous développez un intranet, ce sont les employés qui utiliseront l'intranet qui sont les bénéficiaires du service.

Dans les deux cas, vous aurez besoin de comprendre comment le Produit va être utilisé. Finalement, à quoi sert cette fonctionnalité ? Pourquoi est-elle utile ? Comment va elle rendre service ? Va t-elle faciliter la vie de l'utilisateur ? Va t-elle lui permettre de réaliser quelque chose de nouveau ? Lui faire gagner du temps ? Bref, quel est le bénéfice métier de la fonctionnalité que je vais développer ?

Pourquoi se poser ces questions ? Tout simplement pour prendre plus de plaisir dans votre travail, et pour être plus efficace. Tant que ça ? Oui oui...

Après tout, si vous connaissez les personnes qui vont utiliser ce que vous êtes en train de développer, vous pouvez discuter avec elles, adopter leur vocabulaire, et donc profitez directement de leur feedback. Petit à petit, ce feedback sera de plus en plus positif ; ce sera alors de plus en plus agréable de voir que les gens sont contents de ce que vous leurs offrez.

Cela vous permettra aussi de quitter à l'occasion la casquette de développeur pour, pourquoi pas, proposer des améliorations, discuter de l'orientation du projet... Si vous savez comment va être utilisé le Produit, qu'est-ce qui vous empêche d'essayer de l'améliorer ?

Changer de casquette peut parfois être difficile. Vous pouvez très bien travailler sur des projets dont vous n'avez rien à faire ; c'est triste, mais ça arrive fréquemment. Il arrive de devoir travailler sur un projet pour des besoins alimentaires. C'est légitime. Mais autant faire en sorte que même ces projets, alimentaires, vous apportent de la satisfaction humaine. Adopter la vision de l'utilisateur final vous permettra toujours d'échanger : échanger avec votre client, les utilisateurs finaux, vos collègues...

Il y a quelques temps, j'ai lancé un sondage auprès des développeurs de la société que j'ai accompagnée. Une des questions était à peu près la suivante : "Prenez-vous plaisir dans votre travail?". Près d'un quart des réponses était négatif. Quel dommage !

N'oubliez pas que vous travaillez dans un monde d'humains : plus vous interagirez avec eux, plus vous fournirez un travail qui leur procurera satisfaction, plus vous même éprouverez du plaisir à travailler. Le métier de développeur offre une chance unique : il permet de prendre plaisir en travaillant ; autant en prendre le maximum !

> *Changez de casquette pour prendre le maximum de plaisir dans votre métier de développeur.*

Chapitre 2

Votre code doit refléter le Besoin fonctionnel

Vous le savez, il arrive très souvent qu'un besoin fonctionnel évolue au cours de la vie d'un projet. Et, parfois, une évolution fonctionnelle, même mineure, peut entraîner une refonte majeure d'un code source.

C'est l'effet boule de neige : modifier un bloc de code va avoir un impact sur une grande partie de code source, qui elle même va concerner des domaines fonctionnels très variés, entraînant des régression fonctionnelle nombreuses et imprévisibles.

Si vous l'avez vécu, vous savez que c'est quelque chose qui est totalement incompréhensible pour une personne qui n'est pas développeur. Ce n'est pas de la mauvaise volonté, non ; c'est simplement que, s'il est déjà difficile de saisir pleinement qu'écrire du code source permet au final d'avoir une application qui marche, comprendre qu'il existe dans le code source des interactions fines, complexes et non liées aux domaines fonctionnels est une chose quasiment impossible pour un fonctionnel.

Il faut donc trouver une solution pour répondre à ceux deux questions fondamentales :

- comment assurer la cohérence des changements techniques face aux changements fonctionnels ?
- comment permettre aux fonctionnels d'anticiper le coût technique d'un changement fonctionnel ?

2.1 Le Domain Driven Design

C'est justement l'enjeu du Domain Driven Design de répondre à ces questions. L'objet de cet ouvrage n'est pas de faire de vous des spécialistes du Domain Driven Design

(ou DDD) en quelques pages. Non, il existe des livres très bien conçus, que je ne peux que vous inviter à lire *(cf. Domain Driven Design Vite Fait + DDD de Eric Envans) - Liens à mettre*

Non, l'objectif ici est de comprendre qu'il existe des démarches de travail et de conception logicielle qui permette de faciliter la gestion du besoin fonctionnel. Et c'est là, justement, l'objet du DDD.

Le DDD est une démarche d'échange, de conception et de développement, dont l'objectif est de calquer le code source (code, base de données, ressources...) sur le besoin métier.

Impossible ? Si, et même très profitable. Sitôt que vous aurez commencé à travailler de cette manière, vous ne pourrez plus vous en passer.

L'idée générale est d'élaborer une Langue commune (cette Langue dont nous avons déjà parlé) avec le fonctionnel, puis de concevoir une cartographie fonctionnelle de l'application, afin de n'utiliser, dans le code source, uniquement des concepts présents dans le domaine fonctionnel.

Dès lors, si cette démarche a été accomplie, il devient facile pour le développeur d'identifier quelle proportion de code et quels modules techniques seront impactés par une demande de changement fonctionnel : il lui "suffit" de superposer le nouveau besoin fonctionnel sur l'ancien code source.

De cette manière, toute modification technique devient représentative des modifications fonctionnelles : une modification fonctionnelle mineure risque moins d'entraîner une refonte technique longue et coûteuse. Et a contrario, il est totalement légitime qu'une évolution fonctionnelle lourde nécessite une refonte tout autant importante du code source.

De manière générale, il est important de prendre conscience que le travail du développeur est de répondre à un besoin fonctionnel. Il est donc important de structurer son code, de quelque manière que ce soit, de façon à répondre au mieux et le plus facilement à ce besoin.

> *Les impacts techniques doivent être à la mesure des changements fonctionnels*

2.2 Les tests unitaires sont indispensables

En informatique on adore les sigles ; connaissez-vous le VDD ? C'est un sigle à la mode qui désigne le "var_dump driven development", ou autrement dit, le développement piloté par la fonction `var_dump()` de PHP. Notez que ça marche aussi si vous faites un `console.log()` en JavaScript, ou dans n'importe quel langage...

Pourquoi est-ce une expression à la mode ? Tout simplement parce qu'elle désigne une pratique de développement vouée à l'extinction, qui consiste pour le développeur à afficher à l'écran les valeurs de certaines variables pour en vérifier le contenu manuellement.

La raison pour laquelle cette pratique est vouée à disparaître est simple : lorsque l'on vérifie à la main une valeur d'une variable, c'est long, ennuyant, et on risque de faire des erreurs.

En réalité, faire un var_dump() pour afficher une information à l'écran n'est ni plus ni moins qu'un contrôle sur l'état d'une variable. Or il est très facile d'imaginer des robots dont la seule fonction est de s'assurer en permanence de l'état d'une variable. Et ces robots, infatigables et extrêmement rapides, ne se tromperont jamais.

Ces robots, ce sont les tests unitaires automatisés. Écrire des tests unitaires est, à ce jour, l'un des seuls moyens vraiment efficaces pour faciliter la vie d'un développeur, fiabiliser son code et réduire les coûts de maintenance d'une application.

Il existe dans tous les langages de programmation des outils qui facilitent la rédaction de tests unitaires : JUnit en Java, QUnit en JavaScript... En PHP, deux outils se démarquent par leur fiabilité et leur grande communauté: PHPUnit (http://phpunit.de) , développé par Sebastian Bergmann, et atoum (http://docs.atoum.org) , développé par Frédéric Hardy. Ces deux outils proposent une approche légèrement différente des tests unitaires, mais sont tous les deux simples, efficaces et complets.

De manière générale, rédiger un test unitaire consiste à dérouler un processus simple :

- mettre en condition un code que l'on souhaite tester : ce sont les conditions du test
- exécuter une commande : il s'agit de ce que l'on souhaite tester
- décrire quel doit être l'effet de la commande : il s'agit alors du résultat attendu

Lorsque l'on affiche une variable à l'écran pour vérifier sa valeur, en réalité, tout ce que l'on fait c'est exiger que cette variable ait telle ou telle valeur. En d'autres termes, *on fait une assertion sur la valeur de la variable*. Et toute assertion peut être traduite en code source.

On peut imaginer, par exemple, créer un test unitaire automatisé pour une fonction d'addition, qui utiliserait la fonction PHP native `assert()` :

```
$resultat = addition(5,5);
assert($resultat === 10);
```

Ce code PHP générera une erreur si la fonction addition() ne renvoie pas "10".

L'avantage d'utiliser un outil de tests automatisés sera de pouvoir organiser ses tests facilement et efficacement. Chaque assertion valide sera affichée en vert, les erreurs seront remontées aux développeur en un instant.

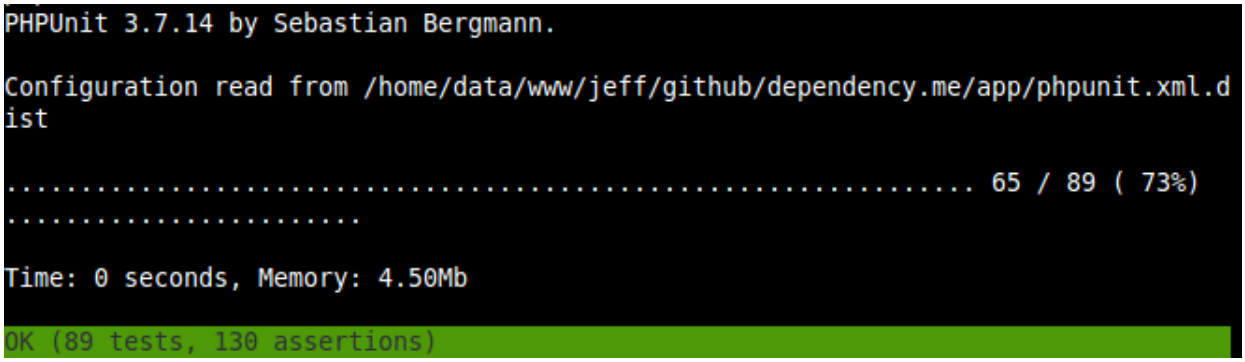

Figure 2.1 Les tests unitaires permettent de s'assurent que le code fonctionne comme prévu

Grâce aux tests automatisés, vous saurez à tout instant si vous venez de tout casser ou pas ; ce sera d'autant plus facile de tout réparer...

> *Les tests unitaires sont le seul moyen efficace de fiabiliser un code source*

2.3 Tester le besoin métier est indispensable

Tester son code source est bien. C'est même déjà pas mal du tout ! Mais une application web est plus qu'un code source : c'est un assemblage de *comportements* (code source) *vis-à-vis d'informations* (données, souvent issues d'une base de données), *pour délivrer un message* (par exemple une page web), le tout exécuté dans un environnement spécifique (serveur), le tout *vis-à-vis d'un utilisateur* (navigateur).

On comprend vite du coup que ce n'est pas parce qu'un code source fonctionne de la manière attendue que l'application, elle aussi, aura le comportement désiré.

Bien plus, il est en réalité difficile de tester unitairement l'ensemble d'un code source : c'est long et coûteux, et cela exige des changements importants dans les pratiques de développement (Développement piloté par les tests, génération de tests aléatoires, etc.).

C'est pour ces raisons qu'il existe différents niveaux de tests automatisés :

- les *tests d'intégration*, qui s'assurent de l'environnement donné (version de PHP, présence d'un module apache...)
- les *tests de charge*, qui s'assurent de la performance d'une application
- les *tests unitaires*, qui vérifient le comportement du code source
- les *tests d'interface*, qui vérifient qu'une application a bien l'aspect attendu
- les *tests fonctionnels*, qui s'assurent que l'application délivre bien un service métier à l'utilisateur

Il existe bien entendu d'autres niveaux de tests (sécurité, cohérence...), mais ceux-ci sont les plus répandus.

Le dernier niveau de test (le test fonctionnel) est sans doute parmi les plus importants. Finalement, tout votre travail consiste justement à faire en sorte que l'utilisateur tire un bénéfice fonctionnel d'une application. S'assurer que ce bénéfice fonctionnel est bien présent est indispensable pour votre client.

Mais justement, ce contrôle, autrement dit ce travail de Recette client, est long et laborieux. Il est source d'erreurs, et surtout il est impossible de contrôler à la main, lors de chaque évolution fonctionnelle, que l'ensemble de l'application fonctionne toujours de la manière attendue.

C'est pour cela qu'il existe désormais des outils de tests fonctionnels automatisés. C'est outils sont infatigables: ils traquent en permanence votre application à la recherche d'anomalies fonctionnelles.

Nous verrons par la suite comment mettre en place ces outils. Mais avant de faire une recette fonctionnelle, il faut déjà commencer par comprendre la demande initiale. Il est temps de découvrir comment faciliter la compréhension d'une demande fonctionnelle.

This Page Intentionally Left Blank

Chapitre 3

Comprenez (enfin!) ce que votre client vous demande

Votre client ne parle pas la même langue que vous. Comment comprendre ce qu'il vous demande dans ce cas ? Certains vous répondront : "utilisez UML". D'autres vous diront qu'il faut demander à votre client de s'exprimer sous forme de "si...alors". J'imagine que cela doit être possible dans certain cas. Mais dans la majorité des situations, exiger de vos clients qu'ils apprennent l'UML, ou leur demander de formaliser leur pensée sous forme de logique booléenne, est un contre-sens ! Ils n'y arriveront pas efficacement : ce n'est pas leur métier, et probablement pas leur mode de pensée.

C'est pour cela qu'il existe ce que l'on appelle le "Développement piloté par le Comportement" ("Behavior Driven Development", ou "BDD" en anglais). Ce qui se cache derrière ce terme barbare est en réalité très simple : il s'agit d'une démarche pour faciliter la communication entre fonctionnels et techniques, démarche soutenue par l'utilisation d'une langue et d'une grammaire commune.

Cette grammaire commune (appelée "Gherkin") est simple, facile à apprendre, et compréhensible tout autant par un informaticien que par votre voisin, votre conjoint(e)..., et par là-même par votre client.

Utiliser cette grammaire vous offrira un certain nombre d'avantages :

- vous éviterez les quiproquos
- les spécifications vous serviront de documentation
- il sera plus facile d'identifier et de gérer les changements fonctionnels de dernière minute

- vous saurez à tout moment ce qu'il vous reste à faire

Attention, le Développement piloté par le comportement a un pré-requis majeur et non négociable : *le client doit s'investir*. Il faudra que le client (ou son représentant fonctionnel) soit disponible pour répondre à chacune de vos questions, en permanence. Si personne autour de vous n'est prêt à s'investir pour exprimer le besoin clairement, oubliez l'idée de pratiquer le Développement piloté par le comportement : vous n'aurez pas les moyens d'y parvenir.

> *Gherkin est une grammaire accessible à tous pour structurer l'expression de besoin*

3.1 Le besoin doit être exprimé par des Fonctionnalités

Concrètement, comment démarrer ? La première tâche revient à votre client. C'est lui qui va devoir recenser la liste des Fonctionnalités qu'il souhaite voir dans son application.

Pour cela, il va devoir identifier chaque fonctionnalité par un titre simple, court, explicite et unique :

```
# Le client va distinguer les fonctionnalités en leur donnant un titre
Fonctionnalité: Avoir accès à la liste des animaux de compagnie
Fonctionnalité: Choisir un animal de compagnie
Fonctionnalité: Acheter un animal de compagnie
Fonctionnalité: ...
```

Il est impératif qu'à cette étape les titres des fonctionnalités soient clairs et explicites.

Ensuite, il va lui suffire de contextualiser un peu cette fonctionnalité. Par exemple, il devra indiquer qui est le bénéficiaire de cette fonctionnalité :

```
En tant que vendeur
En tant que acheteur
En tant que propriétaire de la boutique
En tant que ...
```

Vous aurez donc désormais connaissance de l'acteur bénéficiaire de la fonctionnalité. Il reste encore à identifier le service offert par la fonctionnalité. C'est cela que vous aurez, finalement, à programmer :

```
Je veux acheter un animal de compagnie
Je veux connaître la liste des animaux de compagnie de mon magasin
Je veux connaître la liste des vendeurs
Je veux ...
```

Pour lever toute ambiguïté, il est important pour vous de comprendre quel est le bénéfice de la fonctionnalité que vous aller développer : à quoi sert-elle vraiment ?

Après tout, c'est indispensable de comprendre et de connaître le pourquoi de votre métier. Lorsque vous travaillez, le fruit de votre travail est vraiment utile ; et il est toujours gratifiant de voir que l'on a rendu service à quelqu'un. C'est pour cela qu'il est important que le client exprime le bénéfice obtenu par la fonctionnalité :

```
De telle sorte que je puisse repartir avec un nouveau compagnon
De telle sorte que je sache s'il me faut renouveler le stock
De telle sorte que je puisse organiser les congés de chacun
De telle sorte que ...
```

Chaque fonctionnalité peut donc être exprimée en quatre lignes, très simples et compréhensibles par tous :

```
Fonctionnalité : Connaître la liste des animaux de compagnie disponibles
  En tant que vendeur
  Je veux connaître la liste des animaux de compagnie disponibles à la
vente
  De telle sorte que je sache s'il me faut renouveler le stock
```

Pratique non ? Cette manière d'exprimer les fonctionnalités (dénommée "Gherkin"), est simple, mais surtout est compréhensible par tous. C'est le rôle de votre client (ou fonctionnel) de recenser l'ensemble des fonctionnalités du projet de cette manière. Mais, vous vous en doutez, cela ne suffit pas. Il va falloir maintenant comprendre en détail chaque fonctionnalité ; c'est un travail que vous ferez en commun

Gherkin permet de décrire des fonctionnalités

3.2 Vous êtes un journaliste : interviewez !

Chaque Fonctionnalité peut donc être traduite en quatre petites phrases. Mais dans la majorité des cas cela ne suffira pas : il reste des ambiguïtés, ce n'est pas très précis...

C'est là que vous entrez en scène. A partir du moment où les fonctionnalités sont décrites dans leurs grandes lignes, vous allez devoir les détailler. Oui, vous-même ! Comment ? Tout simplement en prenant quelques instants la casquette d'un journaliste : interviewez votre fonctionnel !

Attention, je vous parle d'une vraie interview. Prenez le temps, pour chaque Fonctionnalité, de poser un maximum de questions pertinentes : "et qu'est-ce qui se passe quand ... ?" ; ou encore "je ne comprend pas le bénéfice tiré de cette fonctionnalité pour l'utilisateur."

Posez, donc, toutes ces questions. Et surtout planifiez systématiquement un moment, avant chaque itération, où votre fonctionnel et l'ensemble des équipes techniques (oui, toutes les personnes qui sont concernées), vont se réunir pour cette interview.

J'insiste : il ne s'agit pas d'une réunion facultative : comment ferait-on sans cela pour être sûr d'avoir bien compris la demande du client ? Le temps passé n'est pas perdu, loin de là : c'est du temps de gagné sur la future maintenance, les futures recettes et les futurs va-et-vient entre le client et les équipes techniques.

L'objectif de ces interviews est de comprendre exactement, sans aucune ambiguïté possible, ce qui vous est demandé fonctionnellement. Pour cela, il faudra découper les prochaines Fonctionnalités que vous aurez à développer en "Scénarios".

3.3 Chaque Fonctionnalité peut être découpé en Scénarios

Chaque Fonctionnalité peut être décrite en quatre points :

- un titre
- un rôle
- un service rendu
- un bénéfice métier

Mais cela ne suffira probablement pas. Il reste des questions en suspens. C'est le rôle des Scénarios de lever ces incertitudes. Ce sera d'autant plus vrai qu'ils seront rédigés en commun avec le fonctionnel et les équipes techniques.

Chaque Fonctionnalité peut contenir un ou plusieurs Scénarios, et chaque Scénario doit pouvoir se distinguer facilement des autres par un titre :

```
Fonctionnalité : acheter un animal de compagnie
  (...)
  Scénario: acheter un animal de compagnie disponible à la vente
  Scénario: tenter d'acheter un animal de compagnie qui est trop jeune
  Scénario: tenter d'acheter un animal de compagnie qui est réservé
  ...
```

Votre travail de développeur va consister à rendre ces Scénarios effectifs dans l'application. En identifiant les Scénarios réalisés et ceux qui ne le sont pas vous saurez donc exactement ce qu'il vous reste à faire.

Vous vous en doutez : un simple titre ne suffit pas ; il va falloir compléter chaque Scénario. Posez par exemple cette question : "dans quel cadre ce scénario se situe t-il ?". Cela vous permettra de connaître le *Contexte de votre Scénario*:

```
Scénario : ...
  Etant donné que tel contexte existe
  Etant donné que je souhaite acheter un chien
  Etant donné qu'il n'y a plus un seul animal disponible
  Etant donné ...
```

Une application consiste généralement à réagir à des *événements*. Précisez-donc quels sont ces événements :

```
Scénario : ...
  (...)
  Quand j'essaye d'acheter un chien
  Quand je regarde quels sont les animaux disponibles
  Quand ...
```

Dès lors, il ne vous reste plus qu'à préciser le *Résultat attendu* lorsque cet événement survient :

```
Scénario : ...
  (...)
  Alors je peux repartir avec un petit chien
  Alors je dois être informé que le chien est trop jeune pour être
  vendu
  Alors ...
```

Cette syntaxe, simple, que l'on nomme Gherkin, va suffire dans la majorité des cas pour que le fonctionnel puisse exprimer son besoin et surtout pour que vous puissiez comprendre sans ambiguïté ce qui vous est demandé.

Voici un exemple de Fonctionnalité :

```
Fonctionnalité : acheter un chiot
En tant que client du magasin
Je veux pouvoir acheter un chiot
De telle sorte que je puisse avoir un animal de compagnie

Scénario : acheter un chiot disponible à la vente
  Etant donné qu'un chiot est disponible en stock
  Quand j'achète un chiot
  Alors on me donne un chiot

Scénario: acheter un chiot sevré
  Etant donné qu'un chiot est trop jeune pour être vendu
  Quand j'essaye d'acheter ce chiot
  Alors je suis informé qu'il est trop jeune pour être vendu
```

Pratique non ?

3.4 Demandez (exigez) des exemples précis

Rappelez-vous que l'objectif du Développement piloté par le comportement est de lever toutes les ambiguïtés. Pourtant, dans le scénario suivant, il reste des questions :

```
Scénario:  acheter un chiot sevré
  Etant donné qu'un chiot est trop jeune pour être vendu
  Quand j'essaye d'acheter ce chiot
  Alors je suis informé qu'il est trop jeune pour être vendu
```

Ces questions sont nombreuses :

- à partir de quel âge un chiot peut-il être vendu ?
- quel est l'âge du chiot que l'on essaye d'acheter ?
- comment suis-je informé ? par quelle phrase ?

Autant de questions auxquelles, si vous ne les posez pas maintenant, vous serez obligé de répondre vous-même, avec tous les risques que cela comporte, notamment celui de devoir recommencer votre développement !

Heureusement, il est tout à fait possible, avec Gherkin, de préciser des valeurs à utiliser. Il suffit de les encadrer dans des guillemets :

```
Scénario : acheter un chiot sevré
  Etant donné qu'un chiot ne peut être vendu avant qu'il n'ait "2 mois"
  Et que "Médor le chien" a actuellement "1 mois"
  Quand j'essaye d'acheter "Médor le chien"
  Alors on doit me dire "Médor le chient est trop jeune !"
```

Pas mal d'ambiguïtés sont levées non ? Et ça ne prend pas beaucoup plus de temps à écrire. Maintenant, il peut arriver qu'un seul exemple ne suffise pas ; il en faudrait plusieurs. Pas de problèmes, Gherkin vous permet d'utiliser des exemples facilement ; il suffit de les passer sous forme de tableaux. Les variables sont alors à encadrer par les symboles "<" (plus petit que) et ">" (plus grand que).

```
Plan du Scénario : acheter un chiot sevré
  Etant donné qu'un chiot ne peut être vendu avant qu'il n'ait "2 mois"
  Et que "<nom-du-chien>" a actuellement "<age-du-chien>"
  Quand j'essaye d'acheter "<nom-du-chien>"
  Alors on doit me dire "<nom-du-chien> est trop jeune !"

Exemples :
  | nom-du-chien  | age-du-chien        |
  | médor       | 1 mois            |
  | rex         | 1 mois et 3 jours     |
  | rantanplan    | 15 jours            |
```

Remarquez qu'il ne s'agit plus dans ce cas d'un "Scénario" mais d'un "Plan du Scénario".

Vous voici donc avec un besoin simple, découpé et exprimé clairement. A vous de jouer !

This Page Intentionally Left Blank

Chapitre 4

Automatisez votre recette

Vous l'avez vu, Gherkin est un moyen simple pour votre fonctionnel pour vous fournir une expression de besoin claire et précise.

Que diriez-vous maintenant de traduire ce besoin en tests automatisés ? Car, oui, c'est possible !

Tout besoin, s'il est exprimé clairement, peut être testé. Si ce besoin est exprimé dans une grammaire solide et complète, tout système d'information peut le tester automatiquement.

Je vous propose de voir comment traduire ce besoin en tests automatisés, de telle sorte qu'une simple ligne de commande dans un terminal vous indique si ce que vous avez développé est conforme ou non.

L'ensemble des exemples que nous allons voir est écrit en PHP. Pourquoi ? Tout simplement parce que la communauté PHP a très fortement et très rapidement adhéré au Développement piloté par le comportement. Les outils PHP pour le Développement piloté par le comportement sont matures, nombreux, et surtout offrent une souplesse qui, à ce jour, ne se retrouve pas dans d'autres langages.

Cependant, il existe des outils pour écrire et lancer des tests automatisés dans à peu près n'importe quel langage :

- Ruby : Cucumber (http://cukes.info)
- Java : JBehave (http://jbehave.org)
- C# : NBehave (https://github.com/nbehave/NBehave)
- Python : Behave (http://packages.python.org/behave/)

- PHP : Behat (http://behat.org) , PHPSpec (http://www.phpspec.net)
- JavaScript : Jasmine (http://pivotal.github.com/jasmine/)
- Net : NBehave (http://nbehave.org)

Il en existe bien d'autres, la liste est loin d'être exhaustive, mais ceux-ci sont particulièrement utilisés dans le monde du développement aujourd'hui.

4.1 Installez et utilisez Behat en PHP

Behat est un outil développé en PHP par Konstantin Kudryashov (@everzet) et soutenu par la société KnpLabs. Cet outil efficace est aujourd'hui mature et largement utilisé. Behat permet d'exécuter des tests automatisés sur tous types d'applications, PHP ou non. Il peut tester :

- des applications web, en pilotant un vrai navigateur ou non
- des applications console (terminal)
- des morceaux de code source

Du moment que ce qui est testé dispose d'un point d'entrée et d'une information de sortie, Behat va vous permettre d'automatiser votre recette métier.

Dans tous les cas, il vous faudra PHP installé sur la machine qui exécutera les tests. L'installation de PHP est résumée sur cette page : http://php.net/manual/fr/install.php.

Windows :

```
Utilisez WampServer (http://www.wampserver.com)
```

Ubuntu, Debian :

```
# en ligne de commande
apt-get install php5-common php5-cli php5-curl
```

Mac :

```
Utilisez MampServer (http://www.mamp.info)
```

Installer Behat est simple, et peut être fait de plusieurs manières : sous forme d'archive PHP (phar) ou en utilisant un gestionnaire de dépendance.

4.1.1 Installation avec Composer

Créez un fichier composer.json à la racine de votre projet, avec le contenu suivant :

```
{
    "require": {
        "behat/behat": "2.4.*@stable"
    },
```

```
    "config": {
        "bin-dir": "bin/"
    }
}
```

Exécutez ensuite la commande suivante dans votre terminal, en vous plaçant à la racine de votre projet :

```
curl http://getcomposer.org/installer | php
php composer.phar install --prefer-source
```

4.1.2 Installation sous forme d'archive Phar

Il vous suffit de télécharger le fichier behat.phar à l'adresse suivante : http://behat.org/downloads/behat.phar. Placez-le ensuite dans un dossier `bin` à la racine de votre projet. Vous aurez donc l'arborescence suivante :

```
/chemin/vers/mon/projet
    (...)
    - bin/
        - behat.phar
```

Pour des raisons de maintenabilité, il est préférable d'utiliser la méthode d'installation avec Composer, qui vous permettra de mettre à jour très facilement l'ensemble des librairies PHP que vous utilisez, y compris Behat.

4.1.3 Préparer le projet

Une fois que Behat est installé, il vous suffit d'exécuter la commande suivante :

```
php ./bin/behat --init
```

Cela aura pour effet de créer dans votre projet tous les éléments dont Behat a besoin pour fonctionner.

Vous voici désormais avec les dossiers suivants :

- `features` : contient la listes des fonctionnalités, sous forme de fichier `.feature`
- `features/bootstrap` : contient les fichiers PHP de traduction de Fonctionnalité en code source

Chaque nouvelle fonctionnalité devra donc être ajoutée dans le dossier `features`, sous forme d'un fichier `.feature`. Créez par exemple le fichier `features/calculer-mon-age.feature`, avec le contenu suivant :

Figure 4.1 Les sources de cet exercice sont disponibles sur http://goo.gl/lgWvy

```
# language: fr
Fonctionnalité: Calculer l'âge d'une personne
    En tant qu'utilisateur de l'application
    Je veux connaître le nombre d'années écoulées entre deux dates
    De telle sorte que je puisse connaître mon age

Scénario: Calculer l'âge d'une personne depuis une date antérieure à
aujourd'hui
    Etant donné que je suis né le 06/07/1986
    Et que nous sommes le 20/09/013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans

Scénario: Calculer l'âge d'une personne depuis une date postérieure à
aujourd'hui
    Etant donné que je suis né le 06/07/3013
    Et que nous sommes le 20/09/2013
    Quand je calcule mon âge
    Alors je suis informé que je ne suis pas encore né

Scénario: Calculer l'âge d'une personne dont c'est l'anniversaire
aujourd'hui
    Etant donné que je suis né le 06/07/1986
    Et que nous sommes le 06/07/2013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans
    Et on me souhaite un joyeux anniversaire
```

Puis exécutez la commande suivante pour lancer Behat :

```
./bin/behat
```

```
Fonctionnalité: Calculer l'âge d'une personne
    En tant qu'utilisateur de l'application
    Je veux connaître le nombre d'années écoulées entre deux dates
    De telle sorte que je puisse connaître mon age

  Scénario: Calculer l'âge d'une personne depuis une date antérieure à aujourd'hui
    Etant donné je suis né le 6 juillet 1986
    Et que nous sommes le 20 septembre 2013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans

  Scénario: Calculer l'âge d'une personne depuis une date postérieure à aujourd'hui
    Etant donné que je suis né le 6 juillet 3013
    Et que nous sommes le 20 septembre 2013
    Quand je calcule mon âge
    Alors je suis informé que je ne suis pas encore né

  Scénario: Calculer l'âge d'une personne dont c'est l'anniversaire aujourd'hui
    Etant donné je suis né le 6 juillet 1986
    Et que nous sommes le 20 septembre 2013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans
    Et on me souhaite un joyeux anniversaire

3 scénarios (3 non définies)
13 étapes (13 non définies)
0m0.019s
```

Figure 4.2 Les étapes de la fonctionnalité ne sont pas encore traduites : elles sont jaunes

Comme vous pouvez le voir, le résultat de cette commande est jaune. Cela veut tout simplement dire que les phrases utilisées pour exprimer le besoin n'ont pas encore de signification pour Behat. Il va falloir les traduire.

4.2 Traduire une Fonctionnalité en code source

Par chance, Behat est suffisamment bien fait pour vous fournir une base de travail pour traduire vos fonctionnalités. Il suffit de copier-coller dans le fichier `features/bootstrap/FeatureContext.php` le code PHP qui a été généré par la commande :

```
./bin/behat
```

Pourquoi copier ce code dans ce fichier ? Tout simplement parce que les fichiers `features/bootstrap/*Context.php` vont servir de passerelle entre le besoin exprimé (sous forme de phrases) et votre application.

Il vous appartient de modifier ces fichiers pour faire la traduction du besoin fonctionnel. C'est un travail qui semble long, mais en réalité il n'est pas beaucoup plus long que de tester vous-même à main que la demande initiale est respectée, mais possède surtout l'avantage de rendre ce travail de recette interne totalement automatique.

Traduisez maintenant la fonctionnalité en code source dans le fichier `features/bootstrap/FeatureContext.php`, en adaptant le code PHP fourni par Behat selon votre besoin. Par exemple :

```php
<?php
// (...)

class FeatureContext extends BehatContext
{

    private $birthDate;
    private $today;
    private $output;

    /**
     * Initializes context.
     * Every scenario gets it's own context object.
     *
     * @param array $parameters context parameters (set them up through
behat.yml)
     */
    public function __construct(array $parameters)
    {
    }

    /**
     * @Given /^que je suis né le (\d+)\/(\d+)\/(\d+)$/
     */
    public function queJeSuisNeLe($day, $month, $year)
    {
        $this->birthDate = new \DateTime(sprintf('%d-%d-%d', $year,
$month, $day));
    }

    /**
     * @Given /^que nous sommes le (\d+)\/(\d+)\/(\d+)$/
     */
    public function queNousSommesLe($day, $month, $year)
    {
        $this->today = new \DateTime(sprintf('%d-%d-%d', $year, $month,
$day));
    }

    /**
     * @Given /^je calcule mon âge$/
     */
    public function jeCalculeMonAge()
    {
```

```
        $this->output = shell_exec(sprintf('php src/age.php
--birthdate=%s --today=%s', $this->birthDate->format('Y-m-d'),
$this->today->format('Y-m-d')));
    }

    /**
     * @Given /^je suis informé que j\'ai (\d+) ans$/
     */
    public function jeSuisInformeQueJAiAns($age)
    {
        if(!preg_match('!'.$age.' ans!', $this->output)) {
            throw new Exception();
        }
    }

    /**
     * @Given /^je suis informé que je ne suis pas encore né$/
     */
    public function jeSuisInformeQueJeNeSuisPasEncoreNe()
    {
        if(!preg_match('!Vous n\'êtes pas encore né!', $this->output)) {
            throw new Exception();
        }
    }

    /**
     * @Given /^on me souhaite un joyeux anniversaire$/
     */
    public function onMeSouhaiteUnJoyeuxAnniversaire()
    {
        if(!preg_match('!Joyeux anniversaire!', $this->output)) {
            throw new Exception();
        }
    }
}
```

Notez ces différents aspects :

- chaque phrase est convertie en une méthode PHP
- le lien entre les phrases et les méthodes PHP est effectué par un expression régulière (par exemple : `/^on me souhaite un joyeux anniversaire$/`)
- vous devez traduire chaque étape. Behat comprendra qu'une étape n'est pas valide si vous levez une exception, comme c'est le cas dans la méthode `onMeSouhaiteUnJoyeuxAnniversaire()`

- il est possible de recevoir certaines informations (nombres, exemples...) sous forme de paramètres de méthodes.
- l'application qui va être testée fonctionne en ligne de commande (`php src/age.php`)

Maintenant, il vous suffit de relancer Behat pour vérifier automatiquement que votre application a bien le comportement attendu :

```
./bin/behat
```

```
Fonctionnalité: Calculer l'âge d'une personne
    En tant qu'utilisateur de l'application
    Je veux connaître le nombre d'années écoulées entre deux dates
    De telle sorte que je puisse connaître mon age

  Scénario: Calculer l'âge d'une personne depuis une date antérieure à aujourd'hui
    Etant donné que je suis né le 06/07/1986
    Et que nous sommes le 20/09/013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans


  Scénario: Calculer l'âge d'une personne depuis une date postérieure à aujourd'hui
    Etant donné que je suis né le 06/07/3013
    Et que nous sommes le 20/09/2013
    Quand je calcule mon âge
    Alors je suis informé que je ne suis pas encore né


  Scénario: Calculer l'âge d'une personne dont c'est l'anniversaire aujourd'hui
    Etant donné que je suis né le 06/07/1986
    Et que nous sommes le 06/07/2013
    Quand je calcule mon âge
    Alors je suis informé que j'ai 27 ans

    Et on me souhaite un joyeux anniversaire

3 scénarios (3 échecs)
13 étapes (9 succès, 1 ignorées, 3 échecs)
0m0.085s
```

Figure 4.3 Les tests échouent ; c'est normal, les fonctionnalités n'ont pas encore été développées

Il vous suffit finalement de développer votre application PHP, conforme aux attentes fonctionnelles, puis de relancer Behat pour vous assurer que vous avez bien traité la demande initiale.

4.3 Exploitez les jeux d'exemples

Figure 4.4 Les sources de cet exercice sont disponibles sur http://goo.gl/jaLxP

Vous l'avez vu, votre fonctionnel peut vous fournir des exemples, grâce à la syntaxe Gherkin.

On pourrait par exemple modifier notre fonctionnalité de la façon suivante :

```
# language: fr
Fonctionnalité: Calculer l'âge d'une personne
    En tant qu'utilisateur de l'application
    Je veux connaître le nombre d'années écoulées entre deux dates
    De telle sorte que je puisse connaître mon age

    Plan du Scénario: Calculer l'âge d'une personne
      Etant donné que je suis né le "<dateNaissance>"
      Et que nous sommes le "<dateDuJour>"
      Quand je calcule mon âge
      Alors on me répond "<reponseAttendu>"

      Exemples:
        | dateNaissance | dateDuJour |
reponseAttendu                          |
        | 06/07/1986    | 20/09/2013 | Vous avez 27
ans                          |
        | 06/07/1985    | 20/09/2013 | Vous avez 28
ans                          |
        | 26/11/2020    | 20/09/2013 | Vous n'êtes pas encore
né                |
        | 06/07/1986    | 06/07/2013 | Vous avez 27 ans. Joyeux
anniversaire    |
```

Vous voici avec une fonctionnalité simple, claire et complète.

Travailler avec des exemples ne change strictement rien pour vous. En réalité, Behat va travailler avec chaque jeu d'exemple, un à un, et va vous envoyer chaque information sous forme de paramètre de méthode, sans que cela n'ait le moindre impact sur votre code.

Voici une traduction possible de cette fonctionnalité :

```php
<?php
// file features/bootstrap/FeatureContext.php
// (...)

class FeatureContext extends BehatContext
{

    private $birthDate;
    private $today;
    private $output;

    /**
     * @Given /^que je suis né le "([^"]*)"$/
     */
    public function queJeSuisNeLe($date)
    {
        $this->birthDate = DateTime::createFromFormat('d/m/Y', $date);
    }

    /**
     * @Given /^que nous sommes le "([^"]*)"$/
     */
    public function queNousSommesLe($date)
    {
        $this->today = DateTime::createFromFormat('d/m/Y', $date);
    }

    /**
     * @Given /^je calcule mon âge$/
     */
    public function jeCalculeMonAge()
    {
        $this->output = shell_exec(sprintf('php src/age.php
--birthdate=%s --today=%s', $this->birthDate->format('Y-m-d'),
$this->today->format('Y-m-d')));
    }

    /**
     * @Given /^on me répond "([^"]*)"$/
     */
    public function onMeRepond($response)
    {
        if (false === strpos($this->output, $response)) {
            throw new Exception;
```

```
        }
    }
}
```

Votre fonctionnel peut désormais ajouter autant d'exemples qui le souhaite, votre travail de traduction ne changera plus.

> *Utiliser des exemples est très simple avec Behat.*

4.4 Réutilisez vos précédents tests

Figure 4.5 Les sources de cet exercice sont disponibles sur http://goo.gl/onk8x

Vous l'avez vu, la phase de traduction d'une phrase (étape) en code source peut parfois être longue.

C'est pour cela que vous devez faire en sorte de faciliter votre travail : réutilisez au maximum vos définitions. Attention, je ne vous parle en aucun cas ici de découper votre code comme vous le feriez dans une application (refractoring, respect des principes SOLID, découpage des méthodes...).

Non, la réutilisabilité dans les Contextes de traduction (fichiers `*Context.php`) ne passe pas par une réutilisation du code source, mais bel et bien par une réutilisation de phrases. **En réalité, vous devez apprendre à faire du refactoring de phrases**.

Cette tâche n'est pas simple au départ, mais est en réalité assez naturelle. Chaque phrase, qui correspond à un contexte, un élément déclencheur ou un résultat attendu, peut être découpée en différentes étapes, qui **elles-mêmes pourront être décomposées jusqu'à arriver à des expressions atomiques simples**.

Prenez par exemple la phrase suivante :

```
Quand j'ajoute dans mon panier "télévision Sony"  depuis le catalogue
produit
```

Cette phrase peut être découpée, par exemple :

```
Etant donné que je consulte le catalogue produit
Quand j'ajoute dans mon panier "télévision Sony"
```

Qui elles-mêmes peuvent être découpées pour créer des expressions atomiques réutilisables :

```
Etant donné que je suis sur la page "/catalogue/produits"
Quand je coche "télévision Sony"
Et je clique sur "Ajouter au panier"
```

Vous pouvez constater que les trois dernières expressions sont en réalité des étapes (web) atomiques réutilisables à l'infini. Cela signifie qu'une fois que vous les aurez traduites, vous n'aurez plus jamais à revenir dessus, et vous pourrez vous en servir à votre gré.

Vous pourriez désormais écrire, par exemple :

```
Etant donné que je suis sur la page "/home"
Etant donné que je suis sur la page "/mon-compte"
...
```

Cette démarche permet donc de faire du refactoring de phrases pour arriver à des expressions atomiques. Par chance, c'est extrêmement simple à faire avec Behat. Regardez plutôt :

```
<?php
// (...)

use Behat\Behat\Context\Step;

class FeatureContext extends BehatContext
{

    /**
     * @When /^j\'ajoute dans mon panier "([^"]*)"  depuis le catalogue
produit$/
     */
    public function
jAjouteDansMonPanierDepuisLeCatalogueProduit($produit)
    {
        return array(
            new Step\Given('que je consulte le catalogue produit')
            , new Step\When(sprintf('j\'ajoute dans mon panier "%s"',
$produit))
        );
    }
```

Comme vous le voyez, il suffit simplement, dans la méthode de définition, de retourner un tableau d'étapes. Chaque étape est en réalité un objet PHP (`Given`, `When` ou `Then`)

qui accepte en paramètre de constructeur la phrase que l'on souhaite utiliser. Pratique non ?

Ce qui donnerait par exemple dans notre cas :

```
<?php
// (...)

use Behat\Behat\Context\Step;

class FeatureContext extends BehatContext
{

    /**
     * @When /^j\'ajoute dans mon panier "([^"]*)"  depuis le catalogue
produit$/
     */
    public function
jAjouteDansMonPanierDepuisLeCatalogueProduit($produit)
    {
        return array(
            new Step\Given('que je consulte le catalogue produit')
            , new Step\When(sprintf('j\'ajoute dans mon panier "%s"',
$produit))
        );
    }

    /**
     * @Given /^que je consulte le catalogue produit$/
     */
    public function queJeConsulteLeCatalogueProduit()
    {
        return array(
            new Step\Given('que je suis sur la page "/catalogue/
produits"')
        );
    }

    /**
     * @When /^j\'ajoute dans mon panier "([^"]*)"$/
     */
    public function jAjouteDansMonPanier($produit)
    {
        return array(
            new Step\When(sprintf('je coche "%s"', $produit))
```

```
            , new Step\When('je clique sur "Ajouter au panier"')
        );
    }

    /**
     * @Given /^que je suis sur la page "([^"]*)"$/
     */
    public function queJeSuisSurLaPage($url)
    {
        // ?
    }

    /**
     * @When /^je coche "([^"]*)"$/
     */
    public function jeCoche($produit)
    {
        // ?
    }

    /**
     * @When /^je clique sur "([^"]*)"$/
     */
    public function jeCliqueSur($bouton)
    {
        // ?
    }
}
```

Notez les trois dernières traduction (les méthodes ` queJeSuisSurLaPage(),jeCoche()etjeCliqueSur()`). Ces méthodes de traduction concernent le comportement d'un navigateur web, comme Firefox ou Chrome... Il arrive en effet très souvent qu'il faille effectuer des actions qui n'existent que dans un navigateur. Vous allez voir qu'avec Behat c'est très facile !

> *Oubliez le refactoring de code. Vous devez apprendre à faire du refactoring de phrases.*

4.5 Testez une application Web

Vous savez désormais automatiser une recette fonctionnelle d'une applicatif. Cependant, dans le cadre d'une application internet, la recette fonctionnelle consiste le plus souvent à parcourir des pages web, vérifier leur conformité (délivrent-elles le service

attendu ?), en soumettant un formulaire, en cliquant sur un lien... Bref, en surfant sur un site web.

Heureusement, il est aujourd'hui très simple d'automatiser une recette fonctionnelle, quand bien même elle concerne une application web. Allons-y !

Une fois de plus nous allons utiliser Behat. Plus précisément, nous allons utiliser une extension de Behat, appelée Mink, qui permet de naviguer dans une application web et de lancer des tests automatisés lors de cette navigation.

Concrètement, Mink permet une double approche :

- Exécuter des tests fonctionnels dans un navigateur virtuel (émulateur)
- Exécuter des tests fonctionnels au sein d'un vrai navigateur, comme Chrome ou Firefox par exemple

Il peut sembler étrange d'exécuter des tests dans un navigateur virtuel. Cependant, bien souvent le comportement fonctionnel que l'on souhaite tester est en réalité disponible directement au sein des pages web, dans le HTML. *On préférera alors privilégier les tests lancés dans un vrai navigateur uniquement lorsque le test porte sur des fonctionnalités complexes (en Javascript ou en Ajax par exemple), ces tests étant en général beaucoup plus lourd et long à s'exécuter que les autres*.

Pour commencer, installez `Mink` :

4.5.1 Installer Mink avec Composer

Modifiez le fichier composer.json que vous avez préalablement créé, et ajoutez-y le code suivant :

```
"require": {
    "behat/behat": "2.4.*@stable",
    "behat/mink": "1.4@stable",
    "behat/mink-extension": "*",
    "behat/mink-goutte-driver": "*"
}
```

Puis relancez Composer :

```
php composer.phar update
```

4.5.2 Installer Mink sous forme d'archive Phar

Exécutez la commande suivante :

```
wget https://github.com/downloads/Behat/Mink/mink.phar
```

Il vous suffit ensuite d'ajouter l'instruction suivantes dans vos fichiers PHP de Contexte :

```
require_once 'mink.phar';
```

4.5.3 Configurer Mink

Maintenant, il suffit de configurer Behat pour lui indiquer que l'on utilise Mink. Mink étant une extension de Behat, C'est très simple. Il suffit de créer le fichier `behat.yml` à la racine de votre projet :

```
default:
  extensions:
    Behat\MinkExtension\Extension:
      base_url: http://url-de-votre-site.fr
      goutte: ~
```

Voilà, Mink est installé !

Rappelez-vous, le travail de Spécification fonctionnelle passe par l'utilisation d'expressions, elles-mêmes organisés autour d'une grammaire : Gherkin. Mink a ceci d'intéressant qu'il propose nativement un panel assez large d'expressions qui concernent un navigateur, que vous pouvez donc réutiliser (comme `"Quand je vais sur "http://..."`, ou encore `Quand je remplis "Votre prénom" avec "Jean-François"`...).

Un besoin exprimé de la façon suivante sera donc automatiquement compris par Behat lorsque Mink est installé:

```
Scenario: s'identifier au sein de l'application
  Etant donné que je suis sur "/accueil"
  Quand je suis le lien "Me connecter"
  Et que je remplis "Identifiant" avec "Jean-François"
  Et que je remplis "Mot de passe" avec "azerty"
  Et que je clique sur "Valider"
  Alors je dois voir "Bienvenue Jean-François"
```

Pratique non ? Constatez que vous n'avez même pas besoin de parler de nom (attribut `name`) des champs HTML ; non, Behat fera le lien pour vous entre les `label`, `title`, `class`... de vos champs de formulaire et l'expression que vous avez utilisée.

Au fait, avez-vous lancé Behat pour vérifier que cela fonctionne ? Allez-y ; Utiliser Mink ne change rien à l'utilisation de Behat, il suffit comme avant d'exécuter la commande suivante :

```
php ./bin/behat
```

Bien entendu, il va souvent arriver que vous ayez besoin de gérer des cas plus complexes que ce qu'il est possible de faire avec les expressions disponibles nativement dans Mink.

Dans ce cas, il va s'agir, ni plus ni moins, de piloter votre navigateur (quel qu'il soit). Un très large éventail de méthodes sont disponibles :

```php
<?php
class FeatureContext extends MinkContext {

    // (...)
    $browser = $this->getMink()->getBrowser();
    $page = $browser->getPage();

    $button = $page->find('css', '.class-css-du-bouton');
    $button->click();
}
```

Notez que désormais notre classe n'hérite plus de `BehatContext` mais de `MinkContext`.

Examinons ce code ensemble. Tout d'abord, nous récupérons la page courante affichée par le navigateur :

```php
    $browser = $this->getMink()->getBrowser();
    $page = $browser->getPage();
```

Ensuite, il suffit de récupérer un élément HTML dans la page, puis d'exécuter l'action souhaitée sur cet élément (ici un clic, mais on pourrait imaginer saisir une valeur, le cocher...) :

```php
    $button = $page->find('css', '.class-css-du-bouton');
    $button->click();
```

Vous avez bien vu : la méthode `find()` permet de récupérer des éléments en utilisant des sélecteurs CSS. Ce sont les mêmes que ce que vous utilisez dans vos feuilles de style CSS. Attention, les pseudo-styles (':hover', etc) ne sont pas supportés. Bien entendu, il est également possible de récupérer des éléments HTML en utilisant des expressions xPath. La documentation de Mink est assez complète (http://mink.behat.org/#xpath-selectors) sur ce sujet.

Il ne vous reste plus qu'à traduire les expressions fonctionnelles de vos clients en différentes actions au sein d'un navigateur, et vous saurez traiter alors la majorité des cas.

Sachez également que Mink permet très facilement de piloter un vrai navigateur. C'est utile lorsque l'on souhaite, par exemple, tester des fonctionnalités qui nécessite un environnement Ajax ou JavaScript. Dans ce cas, juste au dessus des Scénarios ou des Fonctionnalités concernées, il suffit d'ajouter le tag `@javascript`. Ce tag indiquera à Behat que vous souhaitez, dans ces cas spécifiques, lancer un vrai navigateur pour les tests. Bien entendu, dans ce cas, la machine qui exécute les tests doit posséder un environnement graphique (comme c'est le cas pour les ordinateurs de Bureau).

Lorsque l'on souhaite piloter un vrai navigateur, Behat délègue en réalité le travail à des outils spécialisés, comme `Sahi`, ou encore comme `Selenium`. Dans ce cas,

ces outils doivent être installés sur votre machine. La procédure pour les installer est en général très simple, et est bien expliquée dans la documentation officielle (http://mink.behat.org/#different-browsers-drivers) .

Voici la configuration à ajouter dans le fichier `behat.yml` pour indiquer que vous souhaitez utiliser Chrome par exemple, grâce au driver Sahi :

```
default:
    extensions:
        Behat\MinkExtension\Extension:
            base_url: http://url-de-votre-site.fr/
            goutte: ~
            default_session: sahi
            javascript_session: sahi
            browser_name: chrome
            sahi:
              host: localhost
              port: 9999
```

Voici celle que vous pourriez utiliser pour piloter Firefox :

```
default:
    extensions:
        Behat\MinkExtension\Extension:
            base_url: http://url-de-votre-site.fr/
            goutte: ~
            default_session: sahi
            javascript_session: sahi
            browser_name: firefox
            sahi:
              host: localhost
              port: 9999
```

Vous avez désormais toutes les connaissances requises pour commencer à tester automatiquement un projet web. Pourquoi ne pas vous entraîner par exemple sur une petite application web de guichet bancaire ?

Figure 4.6 La correction et les sources de cet exercice sont disponibles sur http://goo.gl/LISNg

Je vous propose de vous entraîner en traduisant le besoin fonctionnel suivant :

```
#language: fr
Fonctionnalité: Gérer un compte bancaire
  Afin de gérer mon compte bancaire
  En tant qu'utilisateur connecté
  Je peux ajouter ou retirer de l'argent sur mon compte

  Contexte:
    Etant donné que je suis connecté en tant que "jean-françois"
    Et que j'ai "50" euro
    Et je suis sur "/"

  Scénario: Consulter mes comptes
    Alors je devrais voir "Vous avez 50 euro sur votre compte"

  Plan du Scénario: Ajouter de l'argent
    Etant donné que j'ai "<montantInitial>" euro
    Quand je sélectionne "<operation>" depuis "Operation"
    Et je remplis "Montant" avec "<montant>"
    Et je presse "Go"
    Alors je devrais voir "Vous avez <montantFinal> euro sur votre
compte"

    Exemples:
     | operation     | montantInitial| montant   | montantFinal  |
     | Ajouter       | 50            | 10        | 60            |
     | Ajouter       | 50            | 20        | 70            |
     | Ajouter       | 50            | 5         | 55            |
     | Ajouter       | 50            | 0         | 50            |
     | Retirer       | 50            | 10        | 40            |
     | Retirer       | 50            | 20        | 30            |
     | Retirer       | 50            | 30        | 20            |

  Scénario: Les découverts sont interdits
    Etant donné que j'ai "50" euro
    Quand je sélectionne "Retirer" depuis "Operation"
    Et je remplis "Montant" avec "60"
    Et je presse "Go"
    Alors je devrais voir "Vous avez 50 euro sur votre compte"
    Et je devrais voir "Les decouverts ne sont pas autorises"
```

A vous de jouer !

Mink permet facilement de piloter un navigateur pour tester vos applications web.

This Page Intentionally Left Blank

Chapitre 5

Optimisez vos tests fonctionnels

5.1 Organisez vos Contextes de tests

Vous l'avez vu, l'ensemble des définitions (traduction d'une fonctionnalité) est stocké au sein d'une classe PHP, ici `FeatureContext`.

Vous aurez très rapidement besoin d'organiser vos définitions de manière à vous y retrouver facilement. En effet, vous aurez de plus en plus de définitions au fur-et-à-mesure que votre application grossira.

Il est très simple avec Behat de créer plusieurs Contextes de définitions. Par exemple, pour une application bancaire, on pourrait créer les contextes de définition suivants :

- Contexte de définitions pour les clients
- Contexte de définitions pour les banquiers
- Contexte de définitions pour les comptes bancaires
- etc.

Pour cela, il suffit simplement de créer un fichier PHP (et une classe) par Contexte de définition, par exemple :

- `ClientContext.php`
- `BanquierContext.php`
- `CompteContext.php`

Puis de les initialiser dans le fichier principal `FeatureContext` :

```
class FeatureContext extends BehatContext
{
    public function __construct(array $parameters) {
        $this->useContext('client', new ClientContext($parameters));
        $this->useContext('banquier', new BanquierContext($parameters));
        $this->useContext('compte', new CompteContext($parameters));
    }
}
```

Désormais Behat utilisera l'ensemble des contextes disponibles pour traduire les expressions qu'il reçoit.

Remarquez les alias de Contexte que nous avons utilisés, par exemple "mink" dans le code suivant :

```
$this->useContext('mink', new MinkContext($parameters));
```

Ces alias vont vous permettre de récupérer facilement vos Contextes d'un Contexte à l'autre. Par exemple pour récupérer votre contexte `mink` depuis le contexte `client`, il vous suffit d'utiliser le code suivant :

```
$mink = $this->getMainContext()->getSubContext('mink');
$session = $mink->getSession();
```

N'hésitez pas à découper vos Contextes de manière à les rendre facilement réutilisables par la suite. Cela vous évitera de mauvaises surprises lorsque votre application grossira et que vous aurez à naviguer parmi plusieurs centaines de définitions.

> *Découpez et organisez vos Contextes de définition*

5.2 Créez une couche d'isolation de l'IHM

Une grande (très grande!) difficulté du développement piloté par le Comportement consiste à réussir à s'abstraire de l'interface graphique.

Imaginez par exemple la situation suivante : vous avez développé un site web pour votre client. Tout marche à merveille, vous avez de très nombreuses définitions pour vérifier avec Behat que le besoin est bien implémenté...

Votre client est tellement content que désormais il souhaite disposer d'une application mobile, identique au site web. Vous voyez le problème ? Vous allez devoir récrire toutes les définitions où vous avez écrit des choses comme

```
return array(
    'Etant donné que je suis sur "/login"'
```

```
    ,'Quand je presse "Me connecter"
);
```

Car oui, sur mobile, il est fort probable que la manière de vérifier si le besoin est bien implémenté change radicalement par rapport à un site web.

Et même sans un changement aussi radical que le passage d'un site web à une application mobile, comment gérer facilement les changements d'interface graphique : une `table` devient une `div`, le texte du bouton `Se connecter` devient `Connexion`...

Le seul moyen d'éviter de devoir réécrire toutes ses définitions lorsqu'un changement d'interface survient (ce qui est le cas dans les deux précédents exemples) est de **créer des couches d'isolation de l'interface**.

Concrètement, c'est très simple : il suffit de créer des Contextes de définition spécialisé dans l'affichage des pages. On peut imaginer par exemple un Contexte de définition consacré à l'affichage des pages Web, qui contiendra donc nos définitions liées à l'affichage :

```
namespace View;
class WebContext extends BehatContext {
    // (...)
}
```

Et une autre, le jour où l'on en a besoin, consacré à l'affichage sur mobile :

```
namespace View;
class MobileContext extends BehatContext {
    // (...)
}
```

Désormais, il suffit d'ajouter un paramètre de configuration spécifique dans le fichier `behat.yml`, que l'on utilisera pour définir quel Contexte de définition l'on souhaite utiliser :

```
default:
    context:
      parameters:
        view: web # mobile | web ...
```

Une simple condition dans le constructeur du Contexte principal suffit à nous permettre d'utiliser le bon Contexte :

```
public function __construct(array $parameters)
{
    if ($parameters['view'] == 'mobile') {
        $this->useContext('view', new View\MobileContext($parameters));
    } else {
```

```
        $this->useContext('view', new View\WebContext($parameters));
    }
}
```

Pour les applications assez conséquentes, il devient rapidement intéressant de créer même un Contexte de définition par page (ou par lot fonctionnel) :

```
$this->useContext('view.catalogue', new View\Web\
CatalogueContext($parameters));
$this->useContext('view.panier', new View\Web\
PanierContext($parameters));
```

Cette pratique, que l'on pourrait désigner par Isolation de l'IHM, peut sembler complexe, mais est en réalité le seul moyen efficace de péreniser les définitions au cours de la vie d'un projet.

N'hésitez pas à consulter la documentation de BehatPageObjectExtension (https://github.com/sensiolabs/BehatPageObjectExtension) , qui est une extension de Behat dédiée à cet usage.

> *Isolez ce qui concerne l'interface graphique dans des Contextes spécifiques*

5.3 Exploitez les compte-rendus de tests

Vous avez désormais un outil pour valider et vérifier qu'un besoin métier a bel et bien été implémenté. Pour l'instant, vous avez exécuté vous-même Behat pour afficher ces informations dans un terminal. Vous vous en doutez, Behat va pouvoir en faire bien plus.

Pour commencer, il est tout à fait possible de générer une page web synthétique des résultats des tests. Pour cela, il suffit d'exécuter Behat en spécifiant le `html` comme format de sortie :

```
php ./bin/behat --format html --out resultat.html
```

Vous disposerez donc d'une page web, graphique, que vous pouvez fournir à votre fonctionnel pour lui indiquer de manière claire ce qui est fait ou reste à faire parmi les spécifications.

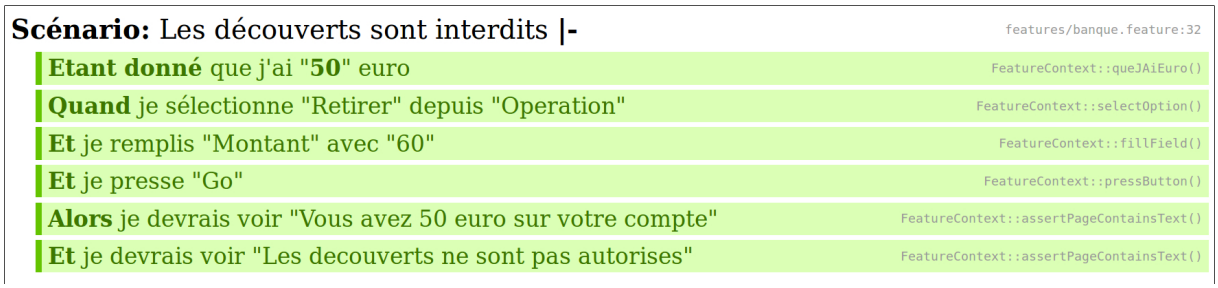

Figure 5.1 Le résultat de chaque Fonctionnalité et Scénario est visible dans une page web

Bien entendu, ce n'est pas tout. La majorité des outils de tests automatisés sont capables de générer des fichiers de sortie dans un format standard (xml). Ces fichiers peuvent donc être fournis à des logiciels tiers, capables de les traiter ou de les transformer selon leurs spécificité.

Il est par exemple tout à fait possible d'intégrer Behat à une plate-forme d'intégration continue, comme Jenkins. Une plate-forme d'intégration continue permet de délivrer en permanence du code source (déploiement continu), c'est-à-dire de permettre à un site web d'être mis à jour très régulièrement.

Indirectement, les plate-formes d'intégration continue (PIC) fournissent une vision d'ensemble sur un projet : les bonnes pratiques sont-elles respectées ? Le code est-il fiable ? Le besoin est-il respecté ? Behat va pouvoir se greffer au coeur de cette plate-forme et va ainsi fournir des indices précieux pour la pilotage du projet.

Pour générer un résultat réutilisable par une plate-forme d'intégration continue, il suffit d'exécuter la commande suivante :

```
php ./bin/behat --format junit --out resultat.xml
```

Pour aller plus loin avec les plate-formes d'ingréation continue, je vous invite par exemple à consulter le site http://jenkins-php.org/.

Il existe également des outils visuels, basés sur ce format de sortie, pour fournir aux fonctionnels une vraie interface graphique pour rédiger les spécifications, mais aussi pour avoir une vue d'ensemble des résultats des implémentations de ces spécifications.

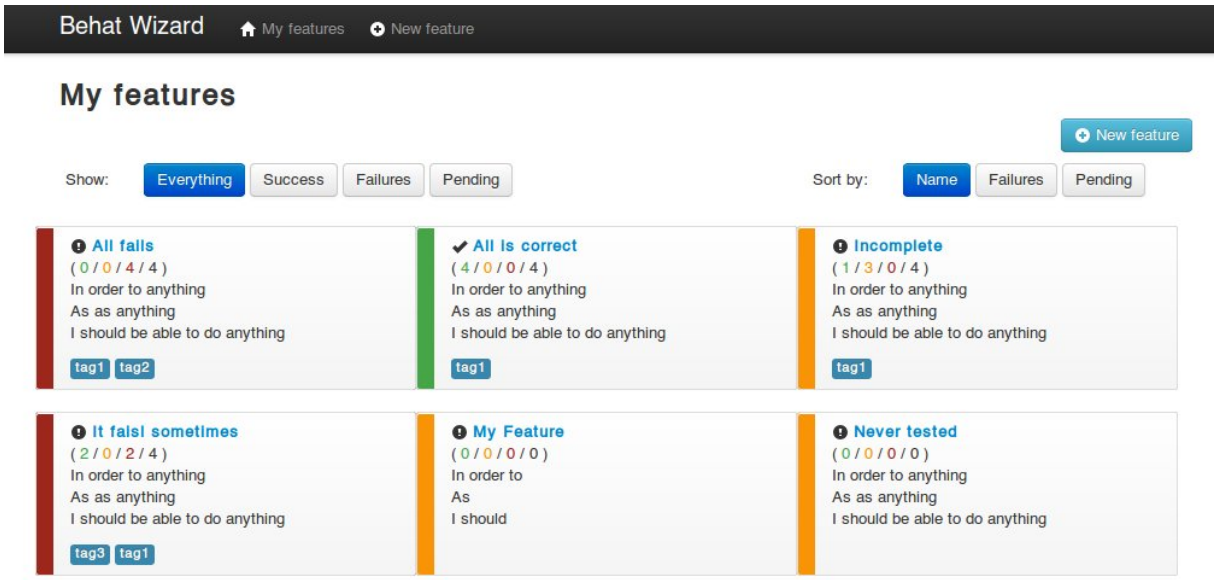

Figure 5.2 Des outils graphiques pour visualiser et éditer les spécifications

Parmi ces outils, on peut mentionner par exemple BehatWizard (http://halleck45.github.com/BehatWizardBundle/demo/behat/wizard/list.html) ou BehatViewer (https://github.com/behat-viewer/BehatViewer) .

Pour éviter de préciser manuellement quel format de sortie vous souhaitez utiliser lorsque vous lancez Behat, vous pouvez tout à fait les préciser une bonne fois pour toute dans votre configuration, c'est-à-dire dans le fichier `behat.yml` :

```
default:
formatter:
  name:                pretty,junit,html
  parameters:
    output_path:       null,junit,report.html
```

Cette configation vous permet par exemple d'afficher le résultat dans votre terminal, mais aussi générer des fichiers de compte-rendu en Html et en XML.

> *Multiplier les formats de sortie de compte-rendus permet de multipliser les usages*

Chapitre 6

Le mot de la fin : le pouvoir du canard !

Connaissez-vous le "Duck Typing" (traduit par "typage canard") ? La légende veut que le poète James Whitcomb Riley soit à l'origine de cette expression :

> *Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard*

Ce principe peut s'appliquer au test logiciel : si un module fonctionnel semble avoir le comportement attendu, possède l'apparence souhaitée, ne génère pas d'erreur, est utilisable facilement, a été utilisé par plusieurs personnes... Alors on peut sans risque penser que ce module fonctionnel est correct et cohérent.

Cette analogie est importante car elle permet de comprendre que **tout ne peut (et ne doit) pas être testé par un logiciel**. Traduire un besoin fonctionnel en tests automatisés prend du temps ; et ce temps, en général, vous ne l'aurez pas. Il faut donc faire des choix : ne testez pas tout, ayez parfois confiance au travail de l'humain, et concentrez-vous sur ce qui est essentiel ou sensible dans l'application que vous avez à développer.

N'oubliez pas non plus que l'ensemble des pratiques que nous avons évoqué concernent avant tout une démarche. **Le développement piloté par le Comportement est une démarche de travail avant de reposer sur des outils**. Non, les outils ne comptent pas, seule la communication entre les différents acteurs d'un projet est essentielle.

Enfin, et c'est le plus important, rappelez-vous qu'**un projet informatique consiste à produire un service qui sera utilisé par des humains** (utilisateurs finaux).

C'est à ces personnes qu'il faut penser en priorité. Toutes les pratiques, toutes les recettes fonctionnelles et techniques qui peuvent être mises en place dans un projet ne consistent qu'à optimiser et à améliorer le service rendu par un Produit. A vous de faire en sorte que ce Produit soit le plus efficace possible.

Auteur

Jean-François Lépine est passionné par la qualité logicielle et l'industrialisation. Très impliqué dans les communautés des développeurs PHP, Jean-François Lépine est le secrétaire de l'Association française des utilisateurs de PHP (AFUP).

Il est aujourd'hui consultant chez Alter Way, société spécialisée dans le service autour de l'Open Source.

Twitter: @Halleck45

Blog: http://blog.lepine.pro