Introduction à l'Agilité

1. Qui sommes-nous ?

1.1. Pierre-Emmanuel Dautreppe

Pierre-Emmanuel est spécialiste .NET depuis 2002 et travaille actuellement comme expert technique et architecte agile dans des projets exclusivement .NET.

Il s'est dirigé vers les méthodologies agiles (et eXtreme Programming en particulier) en 2005. En commençant à travailler avec une équipe qui n'était pas sensibilisée à l'agilité, il a pu voir bon nombre d'anti-patterns appliqués par des équipes venues du milieu traditionnel. Depuis ce jour, il partage son expérience et ses recettes en essayant d'inculquer les valeurs et la culture agile.

Il donne des conférences et des formations sur la technologie .NET et sur les méthodologies Agiles.
En 2009, il a fondé l'association DotNetHub (http://www.dotnethub.be) pour promouvoir .NET et l'agilité au Bénélux et en France. Il organise également La Journée Agile (http://www.journeeagile.be) chaque année.

1.2. Norman Deschauwer

Norman Deschauwer est analyste et chef de projet. Il s'est dirigé vers l'agilité en 2007 et tente depuis lors de s'améliorer sans cesse. Son but est maintenant de faire partager au plus grand nombre les principes agiles et faire changer les mentalités rigides. Les méthodologies agiles ne sont pas réservées au secteur IT.

En 2009, il a fondé l'association DotNetHub (http://www.dotnethub.be) pour promouvoir .NET et l'agilité au Bénélux et en France. Il organise également La Journée Agile (http://www.journeeagile.be) chaque année.

1.3. L'association DotNetHub

C'est une communauté, fondée par six francophones, dont le but est de partager connaissances et retours d'expérience autour de 2 pôles: un pôle technologique (Microsoft.NET et relatifs) et un pôle méthodologique (méthodologies agile). Toutes nos sessions sont données en Français.

Pour tout savoir sur cette association, rendez-vous sur http://www.dotnethub.be. Inscrivez-vous sur le site web pour recevoir la newsletter et être tenu au courant de nos activités.

L'association organise chaque année **La Journée Agile**. Il s'agit du seul évènement francophone en Belgique traitant de l'agilité : une journée complète de conférences proposant plusieurs salles en parallèles.
Pour tout savoir sur cet évènement, et vous y inscrire, rendez-vous sur http://www.journeeagile.be.

2. Introduction

L'agilité provoque généralement des sentiments très antagonistes. Le client prend peur, s'imaginant qu'il perdrait la main sur son projet – voire qu'il signerait un chèque en blanc. Ou, s'il est indécis sur le contenu de son application – voit cela comme la recette miracle lui permettant de tout changer, tout le temps.
Le développeur – être généralement solitaire et égocentrique – est parfois rebuté par certaines pratiques que l'agilité apporte. Ou bien motivé comme jamais, voyant l'occasion de dynamiser un métier dont il s'est lassé.

L'agilité **n'est pas** une recette miracle, et il est des cas où cela ne fonctionnera pas (à cause des membres de l'équipe – client compris, du contexte, …).
Cependant beaucoup de projets ont à apprendre de l'agilité et la mise en place d'une telle méthode leur aurait permis de rejoindre le club très fermé des projets couverts de réussite[1].
Elle peut amener une cohésion et une communication entre les membres de l'équipe à un niveau rarement égalé.

Nous voulons, à travers ce document vous introduire auprès des méthodes agiles, en revenant à la théorie, et vous présenter notre vision de l'agilité à travers notre expérience acquise dans des contextes parfois difficiles.
Nous sommes convaincus que bon nombre de projets auraient des expériences à tirer de l'agile, et par cette présentation nous espérons vous faire prendre conscience des gains que vous pourriez espérer grâce à cette méthodologie.

Nous présenterons l'agilité sous la loupe d'XP (eXtreme Programming) qui est une des méthodes les plus complètes et éprouvées que l'on peut rencontrer.

[1] D'après le « Chaos Report » (1994) du Standish Group, seuls 16% des projets informatiques peuvent être considérés comme réussis. 31% des projets sont arrêtés en cours de route, et 52% ne seront terminés qu'avec de nombreux retards et / ou dépassement de budget.

3. D'où vient l'agilité ?

L'agilité voit son origine chez Toyota aussi appelée le toyotisme dans les années 60. D'après l'ingénieur Taiichi Ōno, il faut revoir complètement l'organisation du travail. Le but étant de placer Toyota premier constructeur mondial et ils l'ont fait !
Pour cela il donne une série de conseils comme réduire le gaspillage, maintenir voire augmenter la qualité et ce en détectant les anomalies tout au long du processus, minimaliser le stock, faire participer tout le monde à la résolution de problème, et bien d'autres.
Il propose également des outils comme le KANBAN, nom devenu familier dans le monde agile.
Le modèle hiérarchique doit également être revu dans l'organisation qu'il propose. Bref ce modèle est à la base de bien des méthodes agiles que l'on rencontre aujourd'hui.
Scrum, eXtreme Programming, puma, RAD sont des noms qui apparaitront bien plus tard.
Mais l'agilité dans le monde informatique prendra un tournant en 2001 lors de la signature du « Manifeste agile » qui est une sorte de charte et qui casse le modèle en cascade traditionnel de génie logiciel qui n'est plus en phase et qui a montré bien trop souvent son inefficacité.
De ce manifeste sont sortis les valeurs et pratiques que l'eXtreme Programming se fait le fer de lance.
Ce document vous expliquera en quelques mots ce que sont ces valeurs et pratiques.

4. Présentation d'eXtreme Programming

4.1. Présentations des Valeurs

XP est régi par cinq valeurs qui peuvent être vues comme un dogme que l'on doit garder en tête et suivre en permanence, pour qu'un projet Agile (XP) se déroule correctement.

4.1.1. Communication

La communication est probablement la valeur la plus importante à suivre dans une équipe Agile[2]. Elle est d'ailleurs omniprésente et se décline dans différentes pratiques à suivre pour le bon déroulement d'un projet.
Elle doit être prise en compte dans la disposition même de l'équipe qui travaillera d'autant plus facilement dans un modèle type « Open Space ». Cela sous-entend également de pouvoir communiquer facilement avec le client.

4.1.2. Feedback

Le feedback est également une notion très importante dans la culture Agile, et ce pour tous les membres du projet.
Prenons quelques exemples :
En tant que développeur, il semble évident que plus tôt nous sommes au courant d'un bug (parce que nous avons fait une erreur d'implémentation), plus celui-ci sera rapide à corriger puisque le code est encore « frais dans la tête », et qu'il n'a pas encore causé de dommage en production.
De même pour un chef de projet, plus vite il est au courant de problèmes dans l'équipe (par exemple de retards pris dans le développement, ou d'une différence entre l'implémentation et ce qui est attendu par le client), plus vite il est capable de prendre des mesures correctives et donc les implications seront d'autant plus limitées.

Le feedback va donc se retrouver naturellement via la communication, mais également via l'utilisation de certains outils comme par exemple le kanban ou le burndown chart.

4.1.3. Courage

Le courage s'applique à plusieurs moments. Une équipe Agile doit être réactive au changement, et aux difficultés. Il faut donc avoir le courage, par exemple :

- D'accepter le feedback et donc de signaler les difficultés, retards que l'on peut rencontrer pour l'accomplissement de sa tâche
- De revoir complètement une architecture parce que celle-ci ne supporte pas une fonctionnalité souhaitée
- De se remettre constamment en question, et de remettre son travail en question afin de l'améliorer et de le simplifier

[2] Quand on parle de « l'équipe » en Agile, cela inclut toutes les personnes impliqués du projet : qu'il s'agisse d'un développeur, d'un tester, d'un chef de projet, d'un responsable client, …

4.1.4. Simplicité

De nouveau, il faut être prêt à accepter le changement, et donc à modifier souvent son code. Tout développeur sera d'accord, il est plus facile et rapide de modifier du code simple que du code complexe, puisqu'il se comprend mieux.
D'un point de vue de l'implémentation, cela sous-entend également de ne pas anticiper les éventuelles fonctionnalités futures, tout simplement parce qu'il y a de fortes chances qu'elles ne concrétisent pas ou en tout cas pas forcément comme on l'imagine.
Il faudra cependant faire bien attention de ne pas confondre « simple » et « simpliste ».

4.1.5. Respect

Nous travaillons tous dans la même équipe. Il est donc important de respecter son propre travail comme celui de ces collègues. Les valeurs de communication, feedback, courage et de simplicité, vont naturellement entrainer un respect vers la personne qui les pratique.
Cette personne-là doit naturellement suivre un certain nombre de règles qui permettent de respecter le travail d'autrui, comme par exemple : ne jamais « commiter » de code qui va faire échouer la compilation, ou d'une manière générale qui pourrait ralentir le travail de ses collègues.

Il est également important que toutes les personnes de l'équipe soit traités sur un pied d'égalité, sous peine de générer des tensions internes qui seront naturellement propices à freiner la motivation et à ralentir la productivité.

Une autre façon de le dire pourrait être « Team First » !

4.2. Présentations des Pratiques

4.2.1. Client sur Site

Les développements agiles brisent le carcan habituel d'un soumissionnaire réalisant un projet pour un client.
Ici nous parlons davantage d'une équipe – d'une seule équipe – composée de développeurs, d'analystes, … et du client – ou d'un représentant du client.

Il est effectivement important qu'un des membres du client – proche des utilisateurs finaux – puisse être présent auprès de l'équipe de développement, ou inversement que l'équipe de développement soit située sur le site du client.

En effet, cela permettra une bonne communication entre tous les acteurs du projet et favorise ainsi la diffusion des exigences des utilisateurs finaux.

4.2.2. Planning Game

Le « planning game » (appelé aussi « jeu du planning ») est un des éléments clés d'eXtreme Programming.
Cette réunion intervient normalement une fois par itération, et regroupe les différents intervenants du projet (client, chef de projet, développeur, …).
Il s'agit d'une réunion de planification, au cours de laquelle :

- Le 'client' va expliquer les fonctionnalités qu'il désire
- Il va également fixer la « business value » de ces différentes fonctionnalités

- Les développeurs vont alors discuter sur ces différentes tâches. Leur but est de s'assurer qu'ils comprennent correctement le besoin, et de proposer des solutions techniques et le coût associé
- En fonction de ce coût et de la business value, le client peut alors fixer la priorité entre les tâches, et ainsi fixer l' « iteration backlog » (ie le panier de tâches) de la prochaine itération.

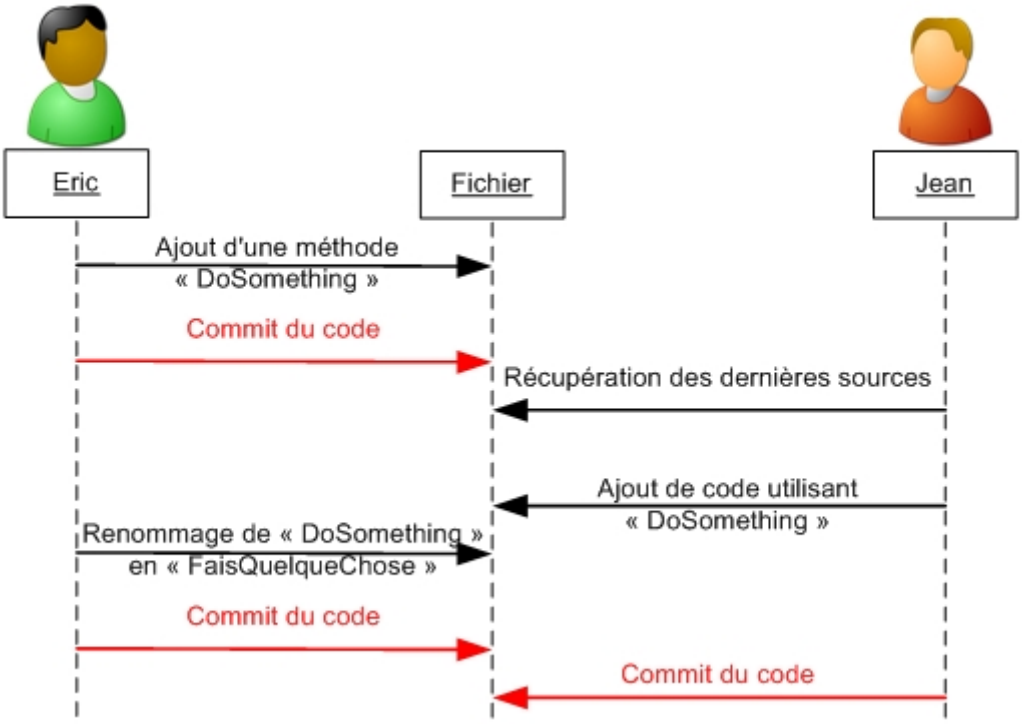
Ce planing game peut être effectué de beaucoup de façons différentes. La façon la plus connue est sans doute le « planning poker » au cours de laquelle les différents participants vont noter chaque tâche à l'aide d'une carte.

Il est important de noter que cette notation est agnostique du temps de réalisation. On parle en général de « nombre de points », correspondant à une complexité relative entre les différentes tâches : autrement dit, une tâche de « 8 » sera deux fois plus complexe qu'une tâche de « 4 ».
On va traditionnellement utiliser les valeurs de la suite de Fibonacci, soit 1, 2, 3, 5, 8, 13, 21, … Le choix d'une échelle non linéaire mets davantage en lumière les incertitudes de l'estimation : plus une tâche est « grosse », moins son estimation est précise.

Pourquoi utiliser des points et non chiffrer en temps ?
Cela a deux avantages principaux : dans la relation entre l'équipe de développement et le client, cela permet de réduire les tensions liées à la présence de temps non productif dans le projet, que ce soit du temps administratif (eg réunion) ou du temps de non qualité (résolution de bug par exemple). Si on réalise 30 heures de tâches fonctionnelles en 40 heures de travail effectif, le malaise est évident.
De plus cela permet d'avoir une échelle constante tout au long du projet. En effet, quel que soit le niveau des personnes travaillant dans les équipes, la complexité relative des tâches ne va pas évoluer. En revanche, c'est la vélocité (c'est-à-dire la quantité de tâche réalisée par l'équipe) qui va évoluer avec le temps.

4.2.3. Intégration continue

Quand on travaille dans une équipe de développement, il est très simple (et très rapide) de créer du code source qui ne compile plus.



L'intégration continue va permettre d'augmenter le feedback de l'équipe et de mettre en évidence le plus vite possible ce genre de problème.

Comme son nom l'indique, « l'intégration continue » permet d'intégrer en permanence (de façon rapide et « continue ») les sources des différents développeurs afin de valider leur intégrité et leur conformité par rapport aux demandes clients.

Cette validation implique donc d'avoir un processus qui va à intervalle régulier

- Récupérer les sources des tous les développeurs
- Compiler ces sources
- Valider éventuellement la conformité de ces sources via par exemple
  - Exécution d'une analyse de code (vérification de règle syntaxique, …)
  - Exécution de tests de validation (tests unitaires, tests d'acceptance, tests de montée en charge, …)
- Déployer les sources pour que le produit puisse être soit téléchargé, soit testé par une équipe dédiée

Le principal objectif de l'intégration continue est d'augmenter le feedback à destination :

- des développeurs qui vont donc s'assurer que leur code compile et qu'ils n'ont pas fait de régression par rapport aux fonctionnalités déjà implémentées
- du chef de projet ou du client qui pourra voir grâce à ce mécanisme l'avancement du projet (via le nombre de tests réalisés / restant à faire)

L'intègration continue n'est pas seulement un outil de source control, c'est une discipline comme par exemple ne pas rester 5 jours sans commiter et donc ne jamais commiter du code buggé !

4.2.4. Métaphore

Le développeur se place trop souvent dans un rôle technique et produit donc… du code technique. Pourtant il développe une application fonctionnelle et donc son code devrait « transpirer » de fonctionnel.

C'est là que la métaphore prend tout son sens. Si dans un groupe, les participants parlent tantôt de « Développeur », « Employé », « Personne », « Personne Physique » ou de « Ressource », elles ont toutes les chances de ne pas se comprendre.

Il est important de se définir un vocabulaire commun entre analyste, développeur, client, … de façon à s'assurer que l'explication des fonctionnalités soit correctement transmise par tous. Et bien évidemment, c'est ce même vocabulaire que l'on doit retrouver au niveau du code (nommage des objets, des attributs ou des fonctions, commentaires, …).

Notez d'ailleurs que le code regorge souvent de règles fonctionnelles (*le paiement doit intervenir au plus tard pour le 15 du mois suivant l'achat*). Ces règles doivent bien sûr être documentées, et idéalement se retrouver intégrés dans des méthodes ou propriétés (par exemple une propriété nommée *DateLimitePaiementAvantRappel* pourrait encapsuler cette logique métier).

4.2.5. Tests Unitaires

4.2.5.1. Qu'est-ce que c'est ?

Il existe beaucoup de types de tests différents. Tous ont un seul but : valider la cohérence du code produit par rapport aux exigences clients.

Les tests unitaires sont généralement imaginés et écrits par les développeurs, afin de valider le comportement d'une petite unité de leur code : une méthode ou une classe par exemple.
Le concept « d'unitaire » est important : plus on va utiliser de concepts (objets) différents au sein d'un même test, moins l'échec d'un test pourra être analysé (et donc corrigé) rapidement (quel est vraiment l'objet / méthode responsable de cet échec ?). On va donc essayer de tester les unités de code les plus petites possibles, et ce de façon isolée.

Il y a toujours un débat dans la communauté pour savoir si l'on doit tester uniquement l'API publique, ou également les méthodes privées. Chaque solution a ses propres avantages et inconvénients, et c'est vraiment chaque équipe qui doit mettre ses propres règles.
Se limiter à l'API publique est logique dans le sens ou ce sont ces objets / méthodes qui seront utilisées par les clients du code que l'on est en train d'écrire. On vérifie donc le comportement de ce que nos clients seront capables de faire.
Cependant, derrière une API publique peut se retrouver de nombreuses classes ou méthodes privées. Tester donc ces dernières nous permet de réaliser des tests certes plus techniques mais également plus unitaires.

4.2.5.2. Les tests et la notion d'isolation

L'isolation signifie s'abstraire de contraintes extérieures qui pourraient rendre le test soit lent soit non prédictible. C'est typiquement le cas lorsqu'on va être dépendant d'un système de fichier, d'une base de données, …

Quand on doit tester de tel code, on pourra réaliser cette isolation en utilisant différentes techniques de codage comme par exemple des bouchons (*mock*) ou de faux objets (*fake*).
Doit-on alors s'abstraire de tester ces parties de code dépendantes de systèmes tiers ? Non, mais il s'agit alors d'un autre type de test, appelé plus fréquemment « test d'intégration ».
La différence principale est que ces tests sont plus demandant : en général plus complexe à écrire, ils seront également plus lents à exécuter.

Dans le processus d'intégration continue, nous allons exécuter les différents tests de l'application. Comme nous le disions précédemment, il est important que cette intégration continue reste rapide pour que le feedback à l'équipe soit le plus pertinent possible.
Paul Julius[3] a introduit un métrique intitulé « Tasse de Café ». Cette valeur empirique correspond au niveau d'information (de feedback) que le développeur peut recevoir le temps qu'il quitte son bureau pour aller chercher une tasse de café et la boire. Il est clairement évident que si le développeur n'a pas la possibilité de savoir si son code compile dans cette durée, alors le risque d'avoir un code non stable / cohérent va grandement augmenter.

Il est donc intéressant de pouvoir catégoriser les différents tests, de façon à choisir ceux qui seront lancés par l'intégration continue. On peut par exemple imaginer :

- Tous les tests de recette seront lancés à chaque commit de code (ou toutes les quelques minutes)
- Les autres tests, plus lents, pourraient n'être lancés qu'une ou deux fois par jour (lors d'une build nocturne par exemple)

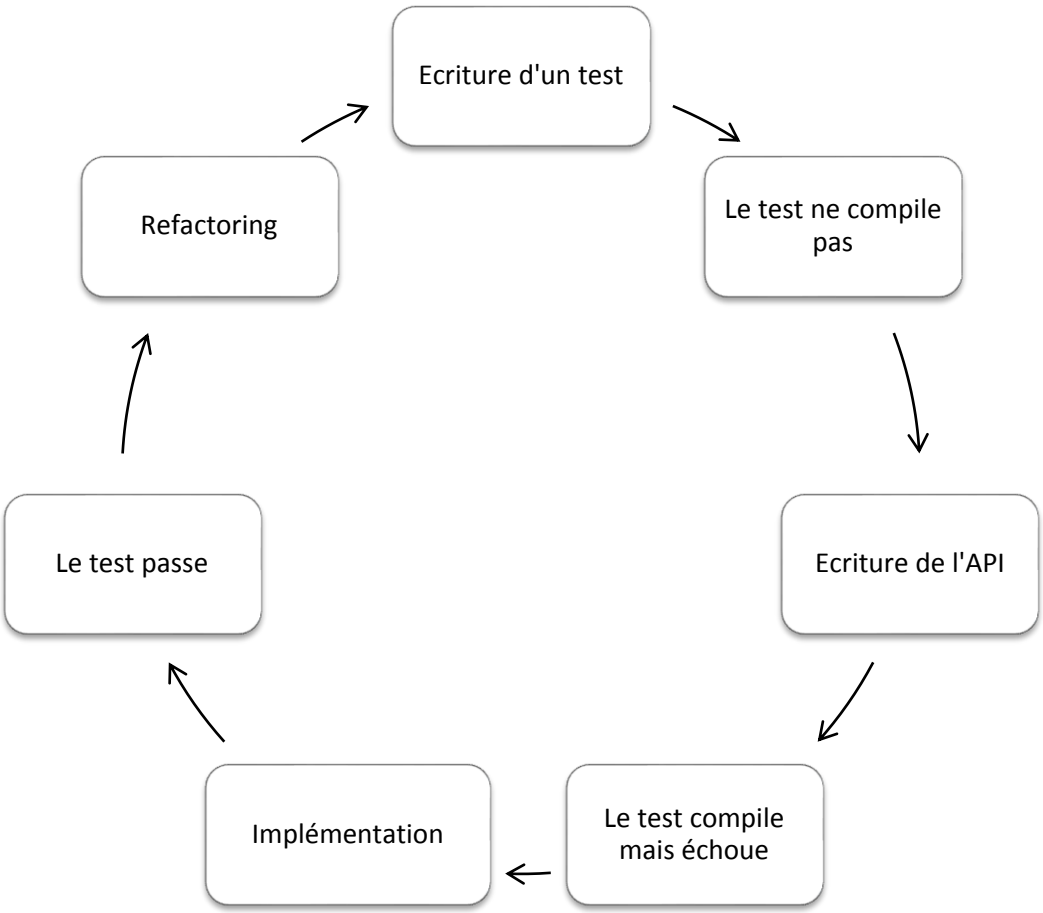
4.2.5.3. Le TDD – Test Driven Development

L'agilité préconise la notion de TDD, ou développement piloté par les tests.
Concrètement, cela consiste à commencer toute activité de développement par un test, et seulement une fois le test écrit, d'implémenter le code correspondant.

Pourquoi travailler dans ce sens-là ?
Cela permet de spécifier dans un test la fonctionnalité que l'on veut implémenter et de se concentrer uniquement sur l'implémentation de cette fonctionnalité, ni plus, ni moins.
Et cela permet également d'utiliser les tests comme un véritable outil de design et non pas seulement comme un outil de vérification.

Il est également important d'inclure dans ce cycle une phase de refactoring (voir ici la pratique), mais sans jamais modifier et le code et les tests en même temps.
On pourra refactorer le code, en utilisant les tests comme filet de sécurité, puis refactorer les tests en ayant le code comme garde-fou.

[3] Co-Fondateur avec Jeffrey Fredrick de l'OIF (Open Information Foundation)



4.2.6. Tests de Recette

Les « tests de recette » (également appelés « tests d'acceptance ») sont un autre type de tests, à orientation fonctionnelle.
Ils sont normalement définis par le client (ou avec le client) et sont des tests de plus haut niveau qui traduisent sous forme de code les fonctionnalités clientes. Il peut alors s'agir d'interaction normalement faite par l'utilisateur (si je vais sur tel écran de l'application, alors je m'attends à voir tel résultat), ou par exemple de contrainte de temps (je dois avoir une réponse endéans les 300ms).
Si le test réussit, alors cela signifie que la fonctionnalité cliente (ou une partie de cette fonctionnalité) est correcte. La livraison de ce développement pourrait donc déclencher le paiement (ie la recette).

Ces tests peuvent être utilisés comme un très bon indicateur d'avancement du projet ou d'une fonctionnalité : pour que la fonctionnalité X soit terminée, il me manque encore Y tests de recette à réaliser.

Ces tests peuvent également servir comme une bonne indication métier puisqu'ils décrivent, avec un regard métier (ou utilisateur) le fonctionnement de l'application. Il est donc important de faire en sorte qu'ils soient lisibles et donc exploitables par le métier.

4.2.7. Livraison Rapide

Même si un client décrit très précisément ce qu'il veut, c'est en général lorsqu'il va pouvoir manipuler concrètement sa fonctionnalité ou son produit qu'il va se rendre compte d'améliorations ou de simplifications à apporter.

D'autre part, un grand problème en informatique est ce qu'on appelle « l'effet tunnel » : plus on attend avant de montrer ce que l'on est en train de faire, plus on a le risque de diverger par rapport aux attentes du client.

Par conséquent, cette pratique a un objectif évident : pouvoir livrer un produit fonctionnel le plus rapidement possible pour que le client puisse le manipuler et qu'il est donc l'occasion de faire ses commentaires au plus vite.
Cette pratique permet de réduire les coûts de développement, tout en maximisant la satisfaction client.

Une autre caractéristique de l'effet tunnel, est que le besoin d'un client est changeant selon des paramètres externes. Lors du début de la crise financière par exemple, on peut aisément penser que les fonctionnalités les plus prioritaires et urgentes d'un client auront changés.
Les livraisons rapides nous permettent également d'augmenter le feedback avec le client, et donc d'être capable plus facilement d'accepter ses nouvelles priorités.

Il ne faut en effet pas oublier que chaque livraison doit aboutir d'un produit fonctionnel. Evidemment, ce produit ne sera pas complet, mais ce qui est présent doit être fonctionnel.

4.2.8. Rythme Soutenable

Le concept de rythme soutenable est assez simple : prenez la meilleure équipe qui soit, et faites la travailler 12h par jour, pendant 5, 10 ou 20 jours d'affilée. Inévitablement, vous allez voir la qualité et / ou la vélocité de cette équipe grandement baisser et le nombre de bug produit augmenter.

La notion de rythme soutenable est donc claire, sans proscrire les heures supplémentaires (chaque projet – agile ou non – rencontre son lot de moments tendus avec de fortes contraintes de temps) il convient en tout cas de chercher à les limiter. Pour cela on pourra par exemple chercher à mettre en place des outils d'automatisation, ou discuter avec l'équipe (client compris) pour définir les priorités métiers entre les différentes fonctionnalités demandées et le temps disponible.

L'objectif est simple : garder une équipe concentrée et productive. Et n'oubliez pas « Team First » !

4.2.9. Conception Simple

Cette pratique vient en droite ligne de la valeur « Simplicité ». Comme nous le disions, nous devons être prêt à accepter le changement et naturellement un code « simple » est plus facile (rapide, moins cher) à faire évoluer qu'un code complexe.

En Agile, nous allons d'ailleurs souvent parler de « design émergent ». Cette pratique consiste à définir le design au fur et à mesure du développement et non lors d'une phase en amont de

conception de diagrammes UML complexes et complets comme on peut le faire dans une méthodologie traditionnelle.
On constate en effet que la phase d'analyse antérieure va le plus souvent conduire à une architecture cherchant à gérer tous les cas possibles, présents et futurs. Et donc également les cas qui ne sont pas requis par le client. Cette sur-complexité dans les diagrammes va bien sûr se traduire par une sur-complexité au niveau du code.
De plus on remarque souvent que cette phase d'analyse est longue car les architectes vont vouloir tout prévoir. Mais dans la pratique, cette prévision est très difficile, et donc lors de l'implémentation des modifications vont être apportées par rapport aux diagrammes.

Au contraire, un développement avec « design émergent » va permettre d'écrire le code strictement nécessaire (ni plus, ni moins) et le plus simple possible.
Cependant cette notion de « simplicité » est très importante à comprendre et peut facilement être mal comprise par des équipes plus juniors de développement.

On va parler de **code simple, mais non simpliste**, soit du code simple à faire évoluer.
Par exemple, on pourra préférer l'implémentation d'un héritage de classe et l'utilisation de polymorphisme plutôt que de travailler avec un type énuméré et une série de condition.
L'héritage sera plus « complexe » à implémenter, mais en vue de l'évolutivité de l'application ce code sera plus « simple ».

4.2.10. Refactoring

Le « refactoring » est une activité très importante du développement agile. Cela consiste tout simplement à « nettoyer » son code.
Cela peut-être fait avant de commencer une tâche, ou bien après. Avant une tâche, cela permet de garder un code iso-fonctionnelle (les tests nous permettent donc de continuer à valider le bon fonctionnement de notre application, ils jouent alors le rôle de filet de sécurité), et de simplifier le code de façon à l'amener à un état plus facilement évolutif afin de faciliter la prise en compte de la nouvelle tâche.
Après l'implémentation de la tâche, cela consiste à revoir son propre code, et à l'améliorer de façon à rester dans un état simplement évolutif.

L'agilité s'évertuera à garder cette activité constante (on parle alors de refactoring continu) de façon à ne pas accumuler de « dette technique ».
La dette technique peut être vue comme la quantité de travail « en retard » pour que son code soit « propre » (ie simple, auto-documenté, évolutif, …). Plus cette dette technique est importante, plus il est difficile de la supprimer. On arrive alors à devoir planifier des « chantiers techniques » qui se chargeront uniquement de supprimer cette dette technique. Ces chantiers deviennent obligatoires après un certain temps simplement pour pouvoir développer de nouvelles tâches, mais ils coûtent en général chers, et surtout n'apportent aucune amélioration fonctionnelle (et donc aucune plus-value pour le client).

Le refactoring demande évidemment une importante autocritique de son propre code, et est grandement simplifié par la pratique du binômage ou par des techniques de revues de code.

4.2.11. Appropriation collective du code

En général sur un projet, chaque développeur va avoir des domaines de compétence différents, tant d'un point de vue technique (telle personne est le spécialiste des bases de

données, …) mais également au niveau projet (telle personne est l'expert de l'import des données X dans le module Y).
Cette spécialisation naturelle est éminemment dangereuse dans un projet, on parle alors du « facteur bus ». Le facteur bus correspond au nombre de personnes qui doivent se faire tuer au cours d'un accident de bus pour compromettre dangereusement votre projet. Evidemment, plus ce nombre est proche de « 1 », plus votre projet est risqué.
Pour parler de circonstances moins tragiques, si un seul de vos développeurs connait une partie du code, que se passe-t-il lorsqu'on doit la modifier (à cause d'un bug ou d'une demande client) et que ce développeur est malade, ou en vacances, ou encore qu'il a décidé de changer de société ?

Pour contrer ce facteur bus, il y a une pratique simple : « l'appropriation collective du code », autrement dit « le code doit appartenir à tout le monde », ou tout développeur de votre équipe doit être capable de toucher à toute partie du code de votre projet.
Cette pratique est grandement simplifiée par deux autres : la mise en place de « conventions de nommage / codage », et le « binômage ».

Il faut impérativement faire en sorte que tout développeur puisse travailler sur toutes les parties de l'application est essayer le plus possible d'éviter l'effet « expert ». Dans certaines équipes, des « référents techniques » vont naturellement voir le jour, que ce soit pour leur expérience ou leur charisme, ou plus simplement par l'inexpérience ou la réserve d'autres développeurs.
Il est alors possible que certains développeurs de votre équipe ne veuillent pas toucher le code de ces personnes, par « peur » ou par « respect ». C'est évidemment une situation dangereuse puisque cette situation créée des spécialistes.

4.2.12. Convention de Nommage

Et donc pour que chaque développeur soit capable, voire même invité, à modifier n'importe quelle partie du code, il convient que chaque développeur se sente à l'aise devant n'importe quelle partie du code.
Et pour cela, rien de plus efficace que d'être en face de code qui ressemble au notre de par son formatage, son style de nommage de variable, la façon de découper le code, …
Il convient donc de définir des règles de nommage, de codage, et de formatage sur son projet de façon à ce qu'il soit développé de façon uniforme, quel que soit le développeur qui travaille.

4.2.13. Binômage

Il s'agit probablement d'une pratique des plus difficiles à faire accepter à son équipe.
Le binômage consiste à faire travailler deux développeurs en même temps sur un même ordinateur. Autrement dit : un PC, un clavier, une souris, deux personnes.

Ces deux personnes doivent évidemment être à l'aise que ce soit pour coder ou pour regarder l'écran. On privilégiera donc l'utilisation de table droite et éventuellement l'utilisation de deux écrans répliqués pour que le confort de ces deux personnes soit maximum.

« Deux personnes sur un PC ? Mais ça coûte deux fois plus cher ! »
C'est probablement la critique la plus fréquente que l'on peut entendre à propos du binômage. Et bien en fait pas tout à fait. Gardez en tête une série de conséquences de cette pratique :

- Chaque développeur a ses propres domaines d'expertise (l'un plutôt sur le design de services, l'autre plutôt sur la base de données, …). Le fait de travailler à deux va permettre de faire très rapidement un nivellement par le haut des connaissances de votre équipe
- Chaque développeur a tendance à avoir son propre domaine d'expertise sur le projet. Le binômage permet un partage en continue de l'expertise de chacun
- Une bonne pratique sur un projet consiste à réaliser régulièrement des revues de code pour vérifier la qualité du code, mais aussi l'homogénéité du code par rapport au code existant. Le binômage pousse cette pratique à l'extrême puisque cela permet une revue de code en permanence. La qualité du code produit va donc naturellement augmenter
- De la même façon, le nombre de bug produit va diminuer puisque deux personnes sont en permanence en train d'écrire et de vérifier le code produit
- Il est connu que la concentration d'un développeur va être très variable dans le temps (téléphone, lecture et écriture de mails, facebook, twitter, youtube, …). Ces activités sont évidemment très présentes dans la vie quotidienne du développeur, mais sont réduites à néant lors du travail en binôme. Une des conséquences immédiates de cette concentration intense est que la fatigue des développeurs est largement accrue également, raison pour laquelle il est important de garder en tête la notion de « rythme soutenable »

Tous ces éléments permettent d'augmenter la productivité des développeurs et de réduire les coûts liés à la non-qualité d'un projet.

Il est cependant important de garder un certain nombre de règles en tête :

- Le binômage n'est pas un mariage à vie et on fera en sorte que les différentes paires de développeurs changent régulièrement
- La paire de développeur est constituée d'un pilote (celui qui a le clavier) et d'un copilote
- Le copilote a un rôle très important, il se charge de la revue de code en permanence, d'anticiper les bugs et les refactorings à effectuer. Il est également là pour donner des pistes de développement et de design émergent
- Les rôles de pilote et de copilote changent régulièrement à la demande des développeurs

5. XPGAME

5.1. Description du XP Game

Les clients, les analystes, les développeurs, les testeurs ont souvent du mal à communiquer et se comprendre mutuellement. La méthode XP permet de les réconcilier, mais souffre d'une mauvaise image d'outil réservé aux informaticiens.
Le XP game permet de manière ludique de concentrer ce qu'est le travail en équipe de manière agile. Chacun offre ses compétences à l'équipe qui n'a qu'un seul but : la réussite du projet.
Tous les profils sont admis, aucune connaissance technique ou de gestion n'est requise.
A l'issue du jeu, tout le monde aura appris comment les user stories, les estimations, le planning, l'implémentation et les tests fonctionnels sont utilisés. Tous auront compris comment la vélocité influence le planning.
Les développeurs et les clients auront appris à se connaître et à se respecter.

Source : ce jeux a été développé par Vera Peter, Pascal Van Cauwenbergh et Portia Tung
Plus d'informations sur : http://www.xp.be/xpgame/