Les 12 principes du manifeste

Benoit Gantaume

Artisan Développeur

Le manifeste n'est pas du tout jeune. Les premiers développeurs nés après son écriture arrivent sur le marché du travail.

Si les quatre piliers ont souvent été repris et commentés, j'ai vu moins de choses sur les 12 principes. Pourtant, ils sont de précieux guides sur comment mettre en oeuvre la démarche.

De plus, ils ont traversé le temps sans une ride : la sagesse qu'ils nous livrent restent plus que jamais d'actualité.

Je te propose de les étudier dans cet article. Ce dernier m'aura pris 12h à écrire pendant lesquelles j'ai consommé18 litres de thé.

Alors, prends ta boisson préférée et cale toi. Il y en a environ pour 30 minutes.

Tu peux également écouter en streaming ou télécharger la série en venant sur le podcast ou sur le blog http://artisandeveloppeur.fr

1 - Livrer rapidement et régulièrement

As-tu déjà entendu parler de l'effet tunnel ?

Cela vient des tunnels. Original, n'est-ce pas ?

As-tu déjà pris un long tunnel, comme celui qui va de France vers l'Italie sous le mont Blanc. C'est long.

Dans un tunnel, tu as rarement d'indication sur là où tu en es du tunnel.

Et si tu utilises un GPS, tu verras qu'il est paumé dans un tunnel.

Tu perds tes outils de navigation pour te repérer.

Alors qu'en dehors du tunnel, tout va bien : tu sais où tu en es de ton trajet, même s'il y a des bouchons.

Et d'ailleurs à un moment donné, tu vois la fameuse lumière au bout du tunnel. Et tu te dis : ok, on est bientôt sorti du tunnel !

Le problème des approches du type cycle en V ou cascade, c'est que tu as un super effet tunnel. Tu en as même deux en fait.

- Un premier effet lié à l'usage par l'utilisateur. Est-ce que ce je vais livrer correspond à ce qui a été demandé ? Ou plus existentiel, à ce dont l'utilisateur a besoin.
- Un second effet tunnel technique : quant à la fin tous les composants sont assemblés d'un coup. C'est le moment où on se rend compte de ce qui marche ou pas.

Le problème de ces approches est qu'elles ne permettent pas d'avoir du feedback, de savoir où on en est.

Or dans un projet informatique, un bon 50 % des hypothèses initiales s'avèrent fausses en cours de route. C'est un chiffre empirique qui n'est pas du tout scientifique basé sur mon expérience : entre l'idée initiale et ce qui est réellement mis en ouvre et utilisé, il y a un bon 50 % de perte. Et encore, je suis optimiste.

Donc 50 % de choses à jeter… Et si on évitait de développer ces choses qui ne servent à rien ?

De plus, le besoin évolue. Donc si tu es dans un projet sans pouvoir collecter du feedback pendant plusieurs mois tu prends le risque que l'objectif réel change en cours de route sans même s'en rendre compte.

Et là, c'est beaucoup plus de 50 % que tu vas jeter.

Tant qu'il n'y a pas de feedback, on ne s'est pas assuré d'avoir compris le besoin réel du client. Or dans le meilleur des cas ce dernier croit qu'il sait ce qu'il veut. Mais dans la vraie vie, il se trompe. Beaucoup.

Quand je parle de client, c'est autant un client qu'un manager, product owner, utilisateur… Et quand je dis qu'il croit qu'il sait ce qu'il veut, ce n'est pas qu'il ait une mauvaise vision de son marché. C'est qu'il fait des hypothèses. Or tu ne peux te faire une idée concrète sur un logiciel qu'une fois dans tes mains. C'est quelque chose qui me fascine : même quand je suis dans le siège du product owner pour nos propres projets, je me rends compte que ce n'est qu'une fois le logiciel dans les mains que j'imagine l'étape suivante. Et j'ai souvent des surprises.

Développer un logiciel sans prendre de feedback, c'est un peu comme assaisonner un plat sans goûter. Ça sera toujours trop ou pas assez salé. Même dans top chef ils se plantent s'ils ne goûtent pas…

Recueillir du feedback est fondamental.

Le premier principe nous dit :

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

- Satisfaire : apporter ce dont le client a besoin. Souviens toi : ce dont il a besoin n'est pas forcement ce qu'il veut. Mais il ne s'en rendra pas compte qu'une fois l'application dans les mains.

- En livrant rapidement : on n'est pas sur une logique d'attendre x semaines ou mois. Non rapidement, c'est quelques jours. Même dans le SCRUM, je te déconseille d'attendre la revue de sprint pour demander du feedback à ton Product Owner.
- A grande valeur ajoutée : on va commencer par ce qui apporte le plus de valeur pour le client. Cela permettra en cours de route d'ajuster les choses. Si tout ne rentre pas dans le calendrier, vaut-il mieux avoir développé 10 choses peu importantes ou deux choses vraiment importantes qui amènent de la valeur ?

Il suffit de montrer régulièrement le travail en cours au client. Attention, tous ne sont pas habitués et dès qu'il va voir quelque chose, il risque de réfléchir avec une logique de produit fini. Ça peut prendre du temps à intégrer. À toi de le prendre par la main et lui expliquer que c'est une livraison intermédiaire pour collecter du feedback. Et tu verras que cela apportera beaucoup : plus l'échange sera fréquent et plus le projet avancera dans la bonne direction.

2 - Accueillir positivement les changements

As-tu déjà eu ce client en phase de recette qui te dit un truc du genre :

“Oui, mais en fait non. J'ai changé d'avis…”

Et là quelle est la réaction typique d'un développeur ?

Dans sa tête : *“Ahhhh ! Non ! Pitié ! Qu'est-ce qui se passe ? “*

Au client : _“ Ah ben c'est-à-dire que là c'est la livraison…

Ce n'était pas prévu… Vous ne vous rendez pas compte ! Il faut tout refaire…”_

Le vrai problème fondamentalement est qu'on essaie trop d'anticiper dans les approches classiques. Pourquoi ?

Parce qu'on nous a appris que le coût du changement variait de manière exponentielle en fonction du temps. Du coup il vaut mieux être sûr de son coup en amont parce que le prix du changement explose en fonction du temps.

Pourtant, si le client demande un changement, c'est qu'il y a une bonne raison. Il ne s'est pas juste levé ce matin en se disant : tient je vais tout changer… En général s'il le fait, vu qu'il paye directement ou indirectement, c'est qu'il y a une nécessité.

Quelle est alors notre responsabilité de développeur ?

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

Il s'agit bien d'un avantage compétitif : c'est-à-dire permettre au client d'aller plus vite, plus fort, plus loin pour qu'il continue de nous nourrir. C'est aussi simple que ça. Si le client est content et qu'il gagne sa croûte, nous aussi !

Encore une fois c'est vrai quel que soit le client : interne ou externe, c'est pareil.

Par contre, comment faire pour que ce soit possible ?

Ils sont gentils les gars du manifeste, mais encore faut-il pouvoir les encaisser les changements, surtout quand ils arrivent tard dans le projet.

J'ai connu des projets dont l'architecture explosait complètement à la première demande de changement.

Pour pouvoir rester serein et accueillir le changement positivement, il faut être prêt à le faire. Il faut rester disponible dans sa tête et garder une architecture souple. Vraiment.

Pour moi cela passe par les pratiques d'ingénierie logicielle de l'eXtreme Programming. Je reconnais que je ne suis pas très objectif, je suis tombé dans la marmite quand j'étais petit. C'est aussi ce que ce je reproche au SCRUM : il ne se soucie pas de la question technique. Il part du principe que chacun sait bien faire son travail. Ok. Mais si tu ne fais qu'accélérer les processus sans prendre en compte l'ingénierie technique, tu vas juste dans le mur. À un moment ou un autre, tu vas te planter.

Je suis convaincu que des pratiques comme le TDD, l'intégration continue, les boucles de feedback rapides, permettent justement d'accueillir les changements même tard dans le projet :

- Parce qu'on a une architecture souple.
- Parce qu'on peut tester rapidement une idée et son impact.

Tout ça nous rend disponible et nous rassure pour avancer.

Faire des gros changements, c'est un peu comme déplacer le mur porteur d'un bâtiment avec des gens qui habitent dedans.

Sur un vrai bâtiment, on se rend compte que ça ne va pas le faire.

Mais dans notre cas : le logiciel est immatériel. Du coup on peut déplacer des modules centraux hyper-critiques tout en ayant des utilisateurs qui sont en train d'utiliser le système. Mais cela demande d'avoir de bonnes pratiques d'ingénierie.

Ça demande du savoir faire et un goût pour le travail bien fait. Et ce n'est pas forcement répandu.

3 - Livrer fréquemment

Les sponsors du projet, le client ou les utilisateurs se font une idée a priori de ce que sera le logiciel.

Sauf que dans la vraie vie 50 % des idées sont de fausses bonnes idées.

Tu crois que tu sais, mais dès que tu vois le logiciel vivre sous tes yeux, c'est plus pareil : certaines portes se ferment, d'autres s'ouvrent.

Dès que tu mets l'application entre les mains des utilisateurs, c'est pire ! Ils ont plein d'idées les

bougres !

On peut limiter avec des wireframes, mais il n'y a pas forcément de vraies données. Un détail d'implémentation peut devenir un cauchemar à coder…

Comment on limite le gaspillage ?

Comment fait-on pour recueillir le feedback dès que possible pour adapter les objectifs et les livrables.

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

Si tu attends deux mois pour livrer un truc, tu prends des risques !

En ce moment je bosse sur un projet où la livraison est dans deux semaines. Si j'attends deux semaines pour livrer et avoir du feedback, on est dans les choux !

Venant de l'XP, je livre plusieurs fois par jour ! Mais pour que ce soit possible, il y a quelques conditions :

- Le build est automatisé, comme le déploiement.
- La recette est automatisée, avant même d'avoir écrit le code.

Dans ces conditions, je peux livrer à chaque morceau, chaque user story, ça ne coûte rien de plus !

On installe une version de pré-prod pour les beta testeurs et en voiture Simone !

Je connais même des équipes qui automatisent le déploiement de branches intermédiaires pour que le product owner se fasse une idée de la fonctionnalité en cours de développement avant la fusion sur develop… Bravo les gars !

Grâce à l'intégration et au déploiement continue tout le monde peut voir quasiment en temps réel les avancées du projet !

Tu peux collecter du feedback en cours de route et adapter au plus près du besoin en évitant de développer trop de fausses bonnes idées !

4 - User centric

En tant que développeur, j'aime ce qui est technique. Comme souvent chez mes confrères.

Le risque est à un moment de perdre de vue pour qui on travaille : on travaille pour son employeur bien sûr, mais la finalité de notre travail, c'est l'utilisateur qui est derrière.

Sans utilisateur, à quoi sert notre travail ?

Si les utilisateurs et les développeurs sont trop éloignés :

- on peut les oublier. Et réciproquement d'ailleurs.
- on peut aussi faire des choses qui ne collent pas aux besoins : que ce soit la gestion du métier ou dans la manière de faire les choses.
- on peut les percevoir comme des casses pied : j'avoue que cela a pu m'arriver quelques fois de pester contre certaines demandes...

Faire grandir un logiciel est déjà un challenge sur le plan technique et cela absorbe beaucoup d'énergie.

Donc pour éviter ces écueils, le manifeste nous rappelle :

Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.

- Utilisateur : On est centré usage. Si on veut apporter de la valeur en tant que développeur, il faut s'intéresser à l'usage de ce qui est produit. Je suis toujours fasciné pas ces environnements dans lesquels les développeurs sont coupés des utilisateurs. Comment cela peut-il être efficace ?
- Représentant : pas toujours possible de parler à tout le monde. T'imagines la taille de la salle de réunion si tu as 2.000 utilisateurs... Ou 100.000 ? On a donc parfois besoin d'un utilisateur proxy. Idéalement, il est lui-même utilisateur de la solution.
- Travaille ensemble : on est dans le même bateau : même si tu es payé par quelqu'un d'autre. L'utilisateur, c'est pour lui qu'on bosse...
- Quotidiennement : tous les jours faire un point. C'est comme ta branche, si tu attends trop, elle dérive et le merge va être plus difficile !
- Tout au long du projet : pas juste au début ou à la fin : tout au long du projet.

On bosse ensemble pendant tout le projet au fur et à mesure ce qui permet de collecter du feedback, de mieux comprendre le métier et d'éviter le gaspillage. Il faut donc garder l'utilisateur en tête et se synchroniser avec lui régulièrement. L'utilisation de personas peut aider : tu l'imprimes à côté de ton écran et quand tu as un doute tu te demandes comment il aurait fait ou de quoi il aurait besoin.

5 - Une équipe motivée

Réalisez les projets avec des personnes motivées. Fournissez-leur l'environnement et le soutien dont ils ont besoin et faites-leur confiance pour atteindre les objectifs fixés.

Cela me semble tellement une évidence, que je suis un peu sec...

- Motivé : oui, évidemment... Si le collaborateur n'est pas motivé cela va prendre une énergie folle de l'impliquer, de vérifier son travail, d'attendre après lui. Sans compter les dégâts moins visibles : les petites remarques ci et là. La démotivation est une maladie contagieuse.
- Fournissez leur l'environnement et le soutien dont ils ont besoin : Comment peut-il en être autrement ? Ah ! Si ! Je me souviens de missions dans laquelle j'ai été recruté pour aider à fabriquer un logiciel et j'ai attendu trois jours pour avoir un ordinateur... J'ai eu le temps

> d'apprendre la spec de quatre pages par coeur… Tu sais qu'il y a aussi ces endroits dans lesquels on te dit : en fait, débrouille toi tout seul. Ou personne ne te répond. Comment peut-on espérer livrer quelque chose dans ces conditions ?

Mais peut-être que le plus important est dans le « Faites leur confiance pour atteindre les objectifs visés ».

On commence par définir l'objectif puis charge à l'équipe, à chaque participant, de faire en sorte d'atteindre l'objectif s'ils l'acceptent.

La clef est dans la confiance. La question de la confiance est à mes yeux le point cardinal, l'alpha et l'oméga de l'agilité. Ça commence par là et ça finit par là. C'est-à-dire qu'elle est un pré-requis à l'agilité et quand l'agilité part en sucette, on finit presque toujours par remonter sur un problème de confiance.

Est-il possible de travailler sur un projet sans se faire confiance ?

Même si cela me semble inconcevable, il faut admettre que oui. Ce type d'endroit existe. Et les gens qui y travaillent livrent des choses malgré la défiance qui y règne. Parfois tu fais même attention à ton collègue de cube : tu ne sais pas d'où viendra le prochain coup de couteau…

Alors, oui, ils livrent. Mais quelle est l'efficacité ?

Quelle quantité d'énergie pour arriver au résultat ? J'ai vu des endroits où le travail de 10 personnes pouvait être fait par 2. Qui peut encore se permettre ce genre de gaspillage ?

La confiance est un réel enjeux : c'est un postulat de base. Et dans les moments difficiles, c'est cette confiance qui sera mise à l'épreuve. Par exemple quand tu commences à te poser des questions du genre : il bosse où il fait du Facebook ?

Alors, attention la confiance n'exclue pas le contrôle, bien au contraire. La confiance sans contrôle, c'est une connerie.

Ce qui permet la confiance, ce n'est pas un voeux pieu fait en aveugle. Ce n'est pas non plus d'ignorer le facteur humain. C'est justement le contrôle. C'est parce que je peux contrôler que je peux faire confiance.

Attention : je ne parle pas de fliquer les gens avec des badgeuses ou passer son temps en réunion pour expliquer ce qui s'est passé.

En fait cela pose la question de la mesure : est-ce que tu mesures des moyens ou des résultats ?

En tant que développeur remote, j'ai deux moyens pour contrôler le travail :

- Qu'est-ce qui est livré ? La vérité est dans le code et les pull-request.
- En face de ce qui a été livré, quel temps a été alloué ?

Il y a donc bien sûr une notion de reporting et de transparence. Quand je travaille à la régie, mon client a accès au reporting de temps au quart d'heure.

Cela me semble essentiel pour nourrir la confiance. Car attention : elle se gagne lentement et se perd très vite. Et si tu laisses le doute s'immiscer, le facteur humain étant ce qu'il est, tu peux vite croire que l'autre est en train de t'enfumer. C'est justement le contrôle qui lève les doutes et permet donc la confiance.

6 - Face à face

Problème : comment je fais circuler le jus de cerveau ?

Nous faisons un métier dont une part essentielle consiste à se comprendre mutuellement. Les échecs sur les projets viennent rarement de facteurs techniques. Un produit qui ne fait pas ce qu' a besoin le client n'est pas un problème technique. Ensuite, il y a les bugs là c'est technique. Mais disons que les premiers temps il y en a peu…

Pour transmettre le besoin, il y a le fameux cahier des charges du 20 ème siècle qui avait pour objectif de décrire le besoin de manière exhaustive. Ce cahier des charges cristallise l'idée qu'il est possible de transmettre de l'information via l'écriture. Oui, bien sûr.

La question est : est-ce efficace ?

Est-il efficace de passer du temps à écrire quelque chose qu'il faudra de toute façon prendre le temps de présenter ? Une bonne fois pour toute : arrête de croire qu'il suffit de décrire ce que tu veux dans un document et l'envoyer aux développeurs pour recevoir un truc fini un mois plus tard !

Ça marche aussi au sein de l'équipe : croire qu'il suffit de poster un message sur Basecamp pour que tout le monde s'approprie le sujet…

Tu n'écris pas un livre. Tu produis une information dont la durée de vie est assez limitée : une spécification reste rarement valide après la livraison si elle n'est pas mise à jour.

Je sais, ça peut paraître étrange, mais c'est comme ça. Est-ce que c'est grave ? Non, du tout : c'est juste que tout change tout le temps… C'est en fait une super opportunité de coller au plus près du besoin.

Alors, plutôt que de passer toute cette énergie à écrire des choses, on fait comment ?

Le Principe #6 du manifeste nous dit :

La méthode la plus simple et la plus efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face.

Il n'y a rien de tel que l'échange direct pour transmettre de l'information. Il y a un double enjeux : le mode de communication : oral plutôt qu'écrit, synchrone plutôt qu'asynchrone.

- Oral : apporte le non-verbal et l'essentiel de la communication.
- Synchrone : permet une boucle de rétroaction rapide en donnant du feedback. Je peux ajuster les choses si je vois que mon interlocuteur ne comprend pas ou interagit avec moi.

Est-il toujours adéquat de tout faire comme ça ?

Je ne pense pas. Parfois l'écrit asynchrone a des avantages : L'écrit force à poser une réflexion et le côté asynchrone permet à chacun de suivre son rythme de travail.

Je suis toujours à la recherche du meilleur compromis temps investi / efficacité / contraintes imposées. Mon souci permanent est l'optimisation des ressources. Dans un contexte de rareté, il faut les utiliser à bon escient. Et dans les startups, la rareté est toujours marquée !

C'est un équilibre instable qui va évoluer selon les phases du projets, la maturité et la taille de l'équipe. L'important est de rester souple et s'adapter en cours de route !

7 - Un logiciel opérationnel

Comment tu mesures l'avancement du projet ?

De manière classique, on utilise les estimations. Puis on les confronte à la réalité.

Une première erreur est de mesurer en budget restant. Si j'ai une tâche estimée à 5h et que j'y ai passé dessus 2h, quel est mon budget restant ? Facile, 3h. Du coup tu peux suivre l'ensemble du backlog et des tâches à faire et dire : Ok sur un budget de 100 heures, on en a consommé 30, il en reste donc 70 à produire. Nous en sommes à 30 %.

Et général tu touches la limite de cette approche dans la zone des 90 % car tu te rends comptes que ça ne bouge plus. Et là tu rentres dans un tunnel. Et tu ne sais pas quand tu vas sortir.

C'est comme la barre de progression de windows : les 10 derniers % prennent plus de temps que les 90 premiers...

Une autre approche beaucoup plus intéressante, c'est le reste à faire. Sur une tâche de 5h, si tu en as consommé 2h, quel est le reste à faire ?

C'est plus compliqué. Tu as envie de répondre 3h ?

Bah peut-être. Ou pas. Peut-être qu'en fait il n'en reste qu'une. Bonne surprise. Ou 7. Mauvaise mayonnaise.

Mais au moins tu as une vision la plus réaliste possible de l'avancée. Du coup tu peux donner une idée de l'avancée du projet en pointant régulièrement le reste à faire.

Le reste à faire associé à un burndown chart est un excellent outil de prévision : il ne coûte presque rien à mettre à jour et permet de rendre visible l'invisible.

C'est déjà bien. Est-ce qu'on pourrait faire encore mieux ?

Le septième principe du manifeste nous dit :

Un logiciel opérationnel est la principale mesure d'avancement.

Tu peux avoir tous les plans budgets du monde et les restes à faire les plus précis, tant que tu n'as pas un logiciel fonctionnel dans les mains, tu n'as rien. Je doute que ton client ou patron te paye pour mettre à jour des restes à faire…

A priori, tu es payé pour livrer du code qui marche. Et c'est pour moi la meilleure manière de savoir si le projet avance. C'est pour ça que je refuse de repousser une date de livraison. Je préfère livrer deux fois moins de choses que prévu et livrer quand même.

Pour cela il faut travailler en itérations courtes. Et livrer à chaque fois quelque chose. Si tu tombes dans le piège du genre : *« ben cette semaine on livre la base de données. Tu comprends ce sont les fondations. Est-ce que les utilisateurs pourront y accéder ? Bah non, il n'ya pas d'IHM. »*

Là tu sais que ton projet va droit dans le mur.

Par contre, si tu livres user story par user story, quitte à devoir reprendre certains morceaux quand d'autres concepts auront émergé, alors tu peux sentir que tu avances dans le regard de tes utilisateurs.

De toute façon tout prend plus de temps que prévu.

Tu as beau tout prévoir, t'auras jamais tout prévu. Alors, autant s'adapter au plus juste.

Sur un côté plus technique, c'est aussi pour ça que j'aime tant le TDD : il me donne une mesure objective de ma progression. En particulier quand j'automatise les tests d'acceptance. Encore Vendredi dernier. J'avais un algorithme à écrire. Je me mets d'accord avec l'expert métier sur 12 cas de test. Au fur et à mesure que mes tests passaient, je savais que je progressais.

Grâce au TDD, j'ai une acceptante automatisée, ce qui veut dire que je suis capable de livrer plusieurs fois par jour si besoin. Je peux comme ça coller au plus juste des besoins de mes utilisateurs.

Bref, livre régulièrement du code qui marche !

8 - Rythme soutenable

J'ai surtout bossé dans des environnements de startup dans lesquels le temps joue contre nous. La raison principale étant qu'on consomme du cash sans forcement remplir les caisses. Du coup ça met beaucoup de pression.

Je pensais que dans les autres boites classiques avec des business model rentables, c'était différent. Et bien non…

Je remarque qu'il y a souvent ce que j'appelle le syndrome du lapin blanc : on est tout le temps en retard. Alors, parfois je pose la question : en retard par rapport à quoi ? J'ai souvent des yeux hagards, un peu perdus, comme si ma question était incongrue. Être en retard pose la question de la référence.

Et c'est là que le bas blesse.

Car bien souvent on est en retard par rapport à des choses qui se sont passées en amont du projet. En retard par rapport à un planning cadré par des accords commerciaux, avant même de s'être posé la question de la faisabilité. En retard par rapport à des estimations fausses par manque de compétence, d'information ou de connaissance du projet. Et du coup la réponse à ce moment est souvent la même : il faut bosser plus pour rattraper le retard.

Je dois reconnaitre une chose : les coups de bourre ont l'avantage de souder une équipe. Par contre, attention : l'équipe fait partie des livrables…. Si l'équipe se défonce pour sortir une itération, comment va-t-elle rempiler le lendemain ?

A quoi bon foncer si c'est pour devoir ralentir le rythme après ?

La seule raison qui me paraisse valable, c'est de devoir tenir une date forte comme un lancement, un plan de communication ou une livraison client. Par contre, attention : c'est le signe que quelque chose ne va pas.

Et d'ailleurs le manifeste nous dit :

Les processus Agiles encouragent un rythme de développement soutenable. Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant.

Mais pourquoi ?

Après tout, vous pourriez me dire : c'est fun. Work hard, play hard ! Et puis on a toute la mort pour se reposer…

Sauf que :

- A trop tirer sur la corde, tu ne sais pas quand elle va casser. Et ce n'est jamais au bon moment.
- Quelle est la qualité de ce que tu fabriques quand tu es fatigué ? Tu vois le cercle vicieux arriver : tu es fatigué donc tu produis du moins bon code. Donc des bugs arrivent qui te ralentissent encore plus. Donc tu bosses encore plus pour compenser. Tu t'épuises d'autant plus et tu es encore plus fatigué. Tu es fatigué donc tu fais des erreurs… Etc…
- Last but not least : au-delà des enjeux humains, c'est aussi une question de devenir prévisible. Si tu fais le yo-yo dans ta vélocité en alternant les sprints de rush et les sprints de repos, l'équipe n'est pas prévisible. Or c'est bien l'enjeux à un moment : permettre de donner de la visibilité aux décideurs.

Donc on retrouve cette idée de rythme constant. Mais il y a une autre notion : indéfiniment.

On est là pour longtemps.

C'est comme faire un marathon en courant à la vitesse d'un sprint. J'ai testé et c'est une très mauvaise idée : j'ai couru la première heure à 100 % de ma fréquence cardiaque max. Après le 26 ème km, la suite a été très dure…

Bref : viser un rythme soutenable, c'est trouvé le bon compromis entre donner le meilleur de soi et garder de l'énergie sur la durée.

9 - Excellence technique

Le côté peut-être le plus novateur du manifeste tient pour moi à son côté itératif et incrémental.

Ce sont ces notions qui sont les plus dures à faire comprendre. Je veux dire vraiment. C'est pourtant dingue, car le monde du vivant est itératif et incrémental. Avec le printemps la maison ressemble à une arche de Noé. Poussins, canetons et chatons envahissent la maison le temps de prendre des forces et aller vivre leur vie dans la nature. Les lapins eux restent dans leur clapier ! Ils sont nés complets. Ils vont grandir. Et bien entendu à la naissance, ils ne peuvent pas encore faire tout ce qu'il pourront faire plus tard… Les chatons ne peuvent même pas encore ouvrir leurs yeux… Difficile d'explorer le monde. De toute façon ils tiennent à peine sur leurs pattes. Mais plus tard, plusieurs cycles quotidiens et annuels après, ils pourront faire beaucoup plus de choses.

L'agilité apporte des cycles beaucoup plus courts que les approches type cycle en V. Ils s'adaptent en continu au contexte.

Sauf que…

Pour accélérer, cela demande de la maitrise !

Si tu mets un kit nitroglycérine sur ta 2cv, tu vas aller plus vite, mais est-ce que tu as vraiment envie de rouler à 200km/h avec une 2cv ? Si tu attaques un virage avec des pneus lisses, tu as plutôt intérêt à rouler tranquille, pas faire le fou à te croire sur un circuit… J'ai essayé et cela a mal fini !

Principe #9 du manifeste :

Une attention continue à l'excellence technique et à une bonne conception renforce l'Agilité.

L'excellence technique est ce qui te donne la maîtrise. Sans maîtrise la puissance n'est rien. Si tu pédales plus vite dans une fosse à purin, tu vas juste en mettre partout. Pas vider la fosse…

C'est le grand reproche que je fais à SCRUM. Il balaie la dimension d'une technique d'un revers de main : on recrute des gens compétents qui font bien le boulot.

Voilà.

Problème réglé.

Circulez, il n'y a rien à voir.

Pourtant, dans l'adoption des pratiques agiles, les pratiques techniques sont les moins adoptées. Je me suis longtemps posé la question du pourquoi. C'est une interprétation très personnelle que je te livre. Je pense qu'il y a plusieurs facteurs :

- Ça ne se voit pas. Ou moins. Ou pas tout de suite. Un board, des rituels, ça se voit tout de suite. C'est visible. Alors que du code propre, ça ne va pas se voir. À la rigueur, du code crade va se faire sentir. On parle de code smell. D'abord, des développeurs. Mais si ça pourrie trop, ça va aussi se sentir par les utilisateurs ! Mais du code propre, c'est perçu comme normal. Le truc c'est que dans la tête de beaucoup de monde, c'est normal qu'un développeur fasse du bon boulot. Y compris de la part des développeurs. Bah non. Ça s'apprend. Et c'est d'ailleurs loin d'être la norme.
- J'ai une autre explication, complémentaire et peut-être plus importante : c'est aussi ce qui remet le plus en cause le développeur. Finalement, tant qu'il s'agit de changer le fonctionnement du groupe, chacun garde sa cohérence interne. Mais changer les pratiques techniques, c'est remettre directement en cause le développeur dans l'intime. Et pas grand monde n'aime ça. Je pense que c'est agréable pour personne. Par contre, certains ont compris que le bénéfice derrière était beaucoup plus grand que l'inconfort temporaire de se re-devenir débutant. Car sauf à tomber dans la marmite de bonnes pratiques à l'école ou dans la première expérience professionnelle, cela va demander parfois de déconstruire des années d'expérience pour accepter de voir le monde et le travail sous un angle différent. C'est particulièrement vrai pour deux pratiques : le TDD et le pair-programming. Elles remettent tellement en question le développeur dans ses habitudes qu'elles sont difficiles à s'approprier.

Bref, si tu veux garder la maitrise de ton projet, intéresses toi à l'excellence technique et va au-delà de tes peurs en acceptant la remise en cause.

10 - Faire simple

En tant qu'ingénieur, j'ai été formé à faire des choses compliquées. J'ai appris à résoudre des problèmes. C'est probablement ton cas également si tu as fait des études techniques. Mais par contre je n'ai pas été formé pour gérer la complexité.

Compliqué et complexe sont deux notions bien différentes.

- Faire un Airbus est compliqué : il faut beaucoup de connaissances pour en fabriquer un.
- La météo est complexe : une faible variation de paramètres peut avoir un impact important.

Si tu utilises un processus compliqué pour gérer un travail complexe, tu vas juste dans le mur. Faire du logiciel est un processus hautement complexe pour une raison simple : le facteur humain. L'humain est probablement une des entités les plus complexes que je connaisse.

Le contraire de complexe est simple.

Faire simple est difficile. Faire complexe est facile.

Le dixième principe du manifeste nous dit :

La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle.

Minimiser la quantité de code inutile. Tout un programme…

Il y a plusieurs manières pour ça… Le code le plus simple est encore celui qui n'est pas écrit… Si tu élimines le code qui n'est pas utilisé pour de vrai, tu gagnes combien ? Es-tu seulement capable de savoir quelles parties de l'application ne sont pas utilisées ? Ou a quelle fréquence ?

La question de l'expérience utilisateur est centrale.

Ensuite, il s'agit de savoir faire pour minimiser le code avec un principe simple en fil d'ariane : don't repeat yourself. Si tu ne minimises pas ton code, tu te retrouves avec un code lourd à faire bouger. Tu auras moins tendance à le refactoriser et le faire propre. Un code lourd c'est aussi du code dupliqué. La duplication est la mère de tous les maux.

Donc aujourd'hui je te propose un exercice : supprime au moins 30 lignes de code.

Cherche dans ton code ce qui peut être refactorisé, dédupliqué ou simplement supprimé, car ce n'est plus utilisé.

11 - Auto-organisation et performance

Dans les approches classiques, il y a ceux qui pensent et ceux qui font. Hérité des systèmes de management militaires, l'enjeu est de réduire les coûts de synchronisation : le système pyramidal est celui qui minimise le nombre d'interactions.

Donc je récapitule : il y a ceux qui pensent et mettent au point le plan. Et puis ensuite on transmet aux développeurs qui n'ont plus qu'à coder. Enfin on recette et tout va pour le mieux dans le meilleur des mondes.

Sauf que tu sais comme moi que cette vision idéaliste est une pure chimère. Un fantasme rassurant. Du coup certains essaient d'être encore plus précis, d'encore mieux anticiper… D'économiser le temps de chacun en limitant les interactions.

Sauf que, comme le précise le manifeste :

Les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.

- Auto organisation : une équipe auto organisée s'organise elle-même. Derrière ce truisme se cache une idée très puissante : ce sont les collaborateurs au contact du terrain qui sont les mieux placés pour savoir ce qui est à faire et comment le faire. Fini les grands plans théoriques. On réfléchit à même le terrain. L'auto-organisation implique une autre idée : la responsabilité collective. Ce n'est plus un chef qui est responsable mais toute l'équipe. Si

quelque chose est à faire, juste fais-le. Il faut comprendre un point important : en donnant plus la responsabilité à l'un qu'à l'autre, je déresponsabilise l'autre. Si je veux impliquer tout le monde, la première étape est de les responsabiliser. Ces notions sont perturbantes et un peu flippantes, surtout pour le manager d'avant. Que devient-il dans cette nouvelle façon de voir le monde ? C'est simple : il est au service de l'équipe. Il met en place le contexte qui permet la performance puis il laisse faire.
- Émergence : Autre point important. On laisse émerger les choses : le design, le fonctionnement de l'équipe, le produit. Inutile de chercher à tout anticiper. Mieux vaut adapter les choses au fur et à mesure. De toute façon l'environnement est devenu tellement changeant qu'anticiper devient un gaspillage d'énergie. Mieux vaut mettre son énergie à préparer le changement. Se tenir prêt pour l'accepter quand il survient. Ainsi je ne cherche pas à tout concevoir en amont. Grâce au tdd, je peux concevoir au fur et à mesure de mes besoins. Grâce aux rétrospectives, je collecte du feed-back sur le fonctionnement de l'équipe et elle s'améliore en continu.

C'est l'objet du point suivant !

12 - Amélioration continue

Est-ce que tu connais les post-mortem ?

C'est une réunion de fin de projet qui a pour but de faire un bilan et tirer les enseignements utiles du retour d'expérience. Cette approche a deux inconvénients majeurs :

- Les améliorations potentielles ne seront utiles qu'au prochain projet. Le bénéfice est en fait un bénéfice potentiel futur. Pour peu que l'équipe soit dissoute à la fin, difficile pour les participants d'être impliqués.
- Cela demande un effort de structuration et de gestion des connaissances acquises. Et cela ne s'improvise pas. Il faut concentrer et diffuser les leçons apprises. Cela demande de l'énergie et si personne n'est attitrée à la tâche, ce sera à faire en plus du quotidien. Empiler les rex dans une gestion électronique de documents ou dans un wiki ne va pas magiquement diffuser les connaissances acquises.

Pourtant, l'amélioration continue est un pilier de l'agilité. D'ailleurs le 12e principe du manifeste nous dit :

À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence.

Il y a cette idée que ces rendez-vous d'amélioration sont programmés. On y alloue de l'énergie spécifiquement. Mais surtout, ils sont faits en cours de projet !

Inutile de faire de grands plans qui révolutionnent le monde : on n'aura pas l'énergie pour les mettre en oeuvre. Plutôt que chercher un espèce de monde parfait, l'agilité se concentre sur des actions modestes mises en oeuvre réellement. Et si tu as l'énergie de mettre en oeuvre des actions

ambitieuses, tant mieux. Du moment que tu fais des choses concrètes pour améliorer la situation maintenant.

Quand j'anime l'atelier du ball point game, je suis toujours frappé par les écarts de vélocité entre la première itération et les suivantes. C'est la même chose dans le travail quotidien : chercher à améliorer la collaboration coûte de l'énergie et génère de grands bénéfices. C'est donc un investissement.

La difficulté est que c'est rarement le moment de faire ça. Les fins d'itération s'y prêtent bien. Mais encore faut-il avoir le courage d'arrêter de produire pour se poser et réfléchir collectivement.

The end

J'espère que ce voyage t'aura plu. Si tu veux aller plus loin, je t'invite à t'inscrire et rejoindre la communauté sur http:// artisandeveloppeur.frJe suis également curieux de ton retour. Tu peux m'écrire à benoit@artisandeveloppeur.fr pour me dire ce que tu as apprécié ou ce qui t'a dérangé.

Enfin, je t'invite à partager ce livret avec un maximum de monde !

A demain.