Chapitre 4 : Une méthode : Agilité

« Je ne suis pas un grand programmeur. Je suis juste un bon programmeur avec de bonnes habitudes. » ***Kent Beck (créateur de la méthode X-Programming)***

1. Objectif du chapitre

Il ne s'agit que d'une introduction à l'agilité pour le développement logiciel. Ici, nous présentons une approche pragmatique utilisable pour les TP et les projets.
À l'issue de ce chapitre, vous serez sensibilisé à :

- L'importance d'un cycle de développement itératif et incrémental.
- La nécessité d'un dialogue permanent avec les futurs utilisateurs sur la base d'une version opérationnelle du futur logiciel.
- Le rôle central du code.
- L'obligation de tester son code.

Cela doit vous permettre de changer votre façon de développer en TP et en projet pour aborder plus sereinement le travail de conception et de programmation.

2. Mauvaises pratiques du développement

2.1. Planification à long terme : le modèle de développement en cascade

Le modèle de développement en cascade fait partie des méthodes prédictives issues de génie civil. Il enchaîne séquentiellement 4 étapes. L'ordonnancement temporel des phases se représente sur un diagramme de Gantt.

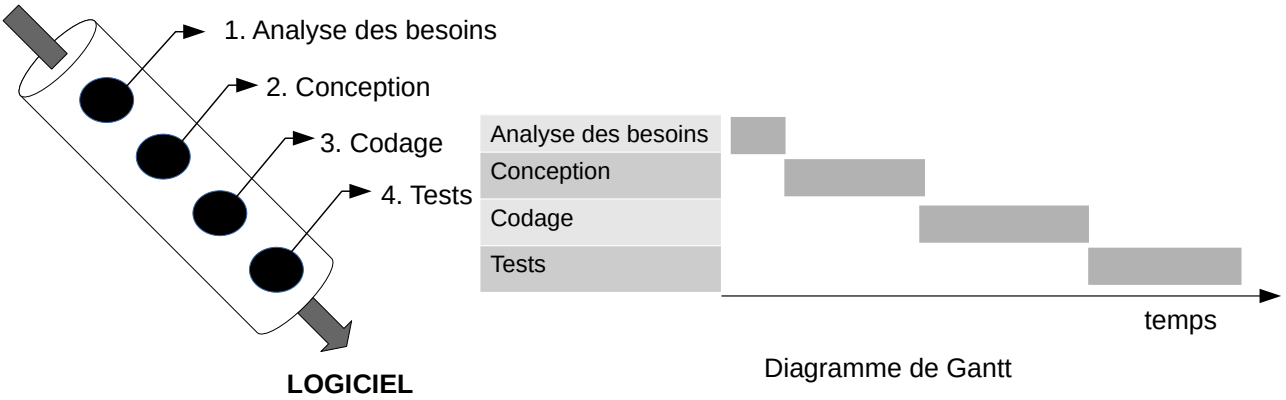


Diagramme de Gantt

Ce modèle de développement a longtemps été enseigné et utilisé en entreprise. Malheureusement, il est la cause de l'échec de nombreux projets professionnels.

2.2. Pourquoi le modèle en cascade ne fonctionne pas pour le développement de logiciels

Outre la difficulté de prévoir le temps consacré à chacune des étapes que nous avons déjà pointée en introduction, les raisons se retrouvent aussi dans chacune des étapes.

2.2.1. Analyse des besoins au début

On se met d'accord avec le client au début du projet sur la liste des fonctionnalités à développer. Cette étape peut durer 2 mois de manière à bien collecter les besoins dans le cahier des charges. Et quand on ressort du tunnel de développement au bout de 3 mois ou 1 an, les besoins du client ont changé ! On appelle « effet tunnel » la période durant laquelle le client n'a aucune nouvelle du futur logiciel.

2.2.2. Conception puis codage

On peut très bien découvrir au moment du codage un problème qui remet en cause la conception. Il faut alors recommencer la phase de conception, ce qui impacte la planification prévisionnelle et peut mettre le projet en péril.

2.2.3. Intégration en fin de codage

L'organisation du travail en équipes consiste à décomposer le développement en différentes parties qui vont être distribuées à différentes équipes qui travaillent en parallèle afin d'accélérer le développement du produit. Puis on réunit ces différentes parties à la fin pour faire le produit final. Mais, l'intégration en fin de projet ne fonctionne jamais parce qu'elle génère trop de conflits à régler. C'est pourquoi ce principe est appelé « intégration big bang » puisque tout explose et rien ne fonctionne.

2.2.4. Test après le codage

Les bugs et les erreurs de conception doivent être détectés le plus tôt possible. Plus le code est avancé, plus la correction est difficile. De plus, la définition des tests en fin de codage est inefficace parce que l'on a perdu les objectifs de ce que l'on doit tester. Enfin, les tests sont très souvent court-circuités pour tenir les délais ce qui rend le logiciel fragile et peu sûr.

2.3. Une amélioration : le cycle en V

L'apport principal du cycle en V se situe au niveau des tests (voir Figure 1). Le meilleur moment pour définir les tests, c'est lors de la spécification des éléments à tester. Par exemple, quand on écrit les spécifications fonctionnelles, on rédige en même temps les tests de validation. Toutefois, ils ne seront effectués qu'après les étapes de programmation, tests unitaires et tests d'intégration.
Mais pour le reste, il n'y a aucune réelle amélioration. Par exemple, du point de vue MOA, le cycle en V est toujours vu comme un tunnel et les tests sont toujours réalisés à la fin.

2.4. Documentation exhaustive

La documentation est un pilier des méthodes prédictives. Elle nécessaire pour :

- passer à la phase suivante,
- revenir en arrière en cas d'erreur,
- faire la maintenance du logiciel.

Mais en informatique, la documentation s'avère généralement inutile voire néfaste :

- Elle n'est jamais lue. C'est donc du temps précieux perdu. Vous-même, utilisez-vous la documentation qui vous est fournie ?
- Elle est laborieuse à rédiger. Elle a tendance à être bâclée et elle devient alors inutilisable.
- Elle freine les changements. Il faut maintenir la documentation en même temps que les changements. Mais, comme cela est difficile (voire impossible) à garantir, la documentation devient rapidement obsolète voire nuisible puisqu'elle n'est plus en accord avec ce qu'elle est censée décrire.

Une documentation n'a de sens qu'à un instant donné ou pour quelque chose qui n'évolue pas.

3. Bonnes pratiques du développement : agilité

Fort du constat précédent, dix-sept experts du développement d'applications informatiques ont repensé le génie logiciel qu'ils ont nommé Agile et rédigé un manifeste.

3.1. Manifeste Agile

3.1.1. Les 4 valeurs de l'agilité (comparées à celles des méthodes prédictives)

Méthodes agiles		Méthodes prédictives
- Individus et interactions	plutôt que	- Processus et outils
- Logiciel qui fonctionne	plutôt que	- Documentation exhaustive
- Collaboration avec les clients	plutôt que	- Négociation contractuelle
- Adaptation au changement	plutôt que	- Suivi d'un plan

Remarque : « Bien qu'il y ait de la valeur dans les méthodes prédictives, notre préférence se porte sur les éléments qui se trouvent dans les méthodes agiles. »

3.1.2. Les 12 principes de l'agilité

Ces quatre valeurs se déclinent en douze principes.

1. **Motivation des équipes :** réalisez les projets avec des personnes motivées, fournissez-leur l'environnement et le soutien dont elles ont besoin et faites-leur confiance pour atteindre les objectifs fixés.
2. **Le dialogue face à face :** privilégiez la colocation de toutes les personnes travaillant ensemble et le dialogue en face à face comme méthode de communication.
3. **Rythme soutenable :** 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.
4. **Équipes auto-organisées :** les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.
5. **Opérationnel sinon rien :** un logiciel opérationnel est la principale mesure de progression d'un projet.
6. **L'excellence technique :** une attention continue doit être portée sur l'excellence technique et une bonne conception.
7. **La simplicité**. La simplicité (cf, principe KISS : *Keep It Simple, Stupid*), c'est-à-dire l'art de minimiser la quantité de travail inutile, est essentielle.
8. **Satisfaction des clients :** notre plus haute priorité est de satisfaire le client en livrant rapidement

et régulièrement des fonctionnalités à forte valeur ajoutée.

9. **Livraisons fréquentes :** livrez fréquemment un logiciel opérationnel avec des cycles de quelques semaines.
10. **Travail client-développeur :** les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.
11. **Accepter le changement du besoin** : accueillez positivement les changements de besoins, même tard dans le projet.
12. **Amélioration continue :** à intervalles réguliers, l'équipe réfléchit aux moyens possibles pour devenir plus efficace. Puis elle s'adapte et modifie son mode de fonctionnement en conséquence.

Il ressort de ces douze principes, que l'agilité met en avant les relations humaines hors d'outils numériques de collaboration et considère le développement comme un travail artisanal hors de normes qui en régissent la qualité.

4. Mise en pratique de l'agilité pour le développement

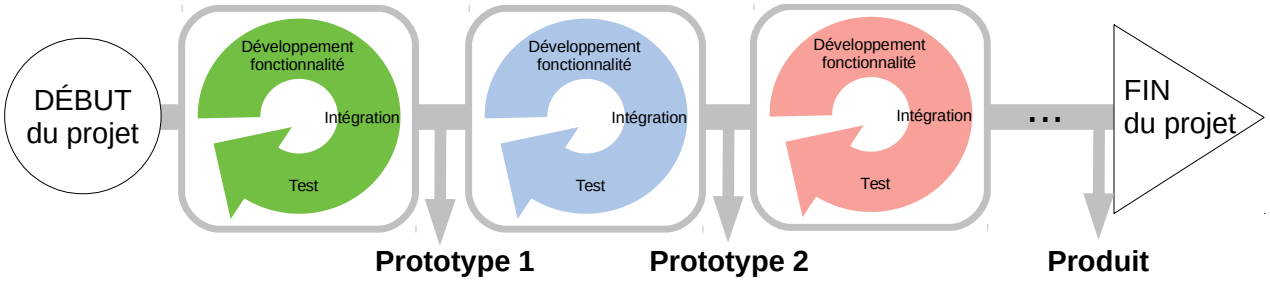
Du côté de la MOE (maîtrise d'œuvre), la mise en pratique des méthodes agiles passe par :

1. un cycle de développement itératif,
2. un cycle de développement incrémental,
3. une collaboration entre le client et l'équipe de développement,
4. une équipe auto-organisée avec un rythme constant,
5. l'intégration continue du travail des développeurs (quasi-quotidienne).

L'idée est de développer le logiciel par itérations pour produire des versions successives du logiciel qui soient exécutables et donc testables en grandeur réelle. Le logiciel grossit progressivement dans une direction donnée régulièrement par le client. Le projet est découpé en une suite de petits projets facilement maîtrisables par l'équipe de développement.

4.1. Cycle de développement itératif : casser la planification à long terme

Le temps total de développement est divisé en **itérations de même durée**. Une itération est un-mini projet de très courte durée, généralement 2 à 3 semaines maximum. On y retrouve toutes les étapes du cycle en cascade. Chaque itération se termine par la livraison au client d'un prototype opérationnel avec lequel il peut repartir pour l'exécuter dans son environnement.



Il faut considérer chaque itération comme une fin en soi (comme si le projet s'arrêtait à l'issue).

- Il ne peut y avoir de tâche de développement s'étalant sur plusieurs itérations ; au besoin la découper.
- Le prototype doit être propre et testé. Il doit être testé dans les conditions de la production. Il est pratiquement livrable en état (il est simplement sous-doté en fonctionnalités).

4.1.1. Avantages du cycle itératif

Du fait que la durée d'une itération est courte, cela réduit considérablement l'impact des problèmes liés à la gestion de projet identifiés plus tôt.

- On avance à petits pas testés. Si on rate un pas, on n'a raté qu'un pas (ie 2 semaines), sans grosses conséquences.
- Chaque pas est validé par le client.
- Le manque de temps conduit à un déficit de fonctionnalités et pas à un échec total du projet ni à un logiciel non testé.
- On a toujours une version intermédiaire mais opérationnelle du logiciel à donner au client.

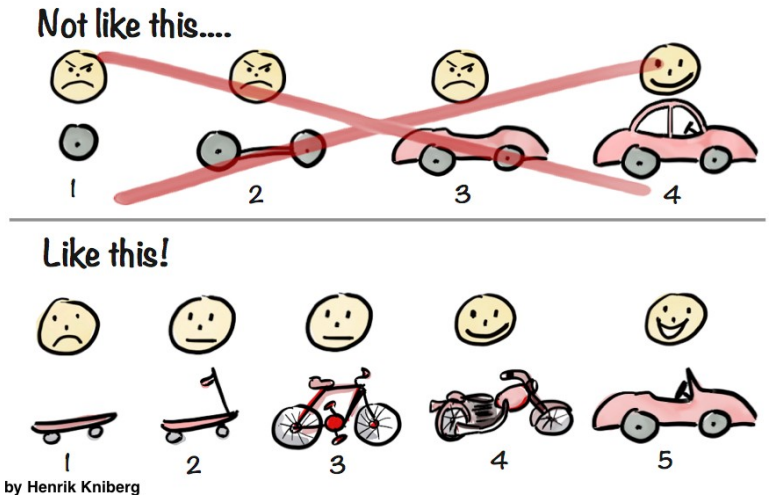
4.2. Cycle de développement incrémental : profiter de la malléabilité du logiciel

Le logiciel est construit par **incrémentation**. Le prototype n'est pas jetable. Au contraire, il sera repris pour être amélioré au cours des itérations suivantes s'il est jugé intéressant par le client ou revu s'il ne satisfait pas le client. Chaque prototype suivant ajoute, supprime et modifie des fonctionnalités du prototype précédent.

4.2.1. Incrément : MVP (Minimum Viable Product) ou POC (Proof Of Concept)

Le produit minimum viable (MVP) est un prototype avec juste assez de fonctionnalités pour satisfaire les premiers clients et fournir une rétroaction pour poursuivre le développement. Le MVP permet de valider ou d'invalider des hypothèses. On passe d'un développement basé sur des présupposés, « je pense que les clients ont besoins de ça », à un développement basé sur des retours factuels, « les clients me demandent ça ». Cela permet de maximiser la création de valeur en évitant de louper sa cible et de dépenser des ressources sur des fonctionnalités à faible valeur ajoutée.
Un MVP doit apporter rapidement de la valeur à chaque itération. La valeur peut se résumer par toute fonctionnalité ou amélioration participant à l'atteinte des objectifs métiers.

Le concept de MVP peut être illustré par le dessin ci-dessous qui symbolise le développement d'une voiture.



https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp

Le scénario du haut, livrer successivement un pneu, un châssis, une caisse et enfin la voiture ne correspond pas l'idée de MVP. Même si on livre régulièrement des prototypes, ces prototypes ne sont pas utilisables par le client : que voulez-vous qu'un automobiliste fasse d'un pneu ou d'un châssis dans sa cours ?

Par contre, dans le scénario du dessous, les prototypes livrés sont utilisables par le client qui peut faire des retours sur l'utilisabilité du prototype. Ce scénario profite de la malléabilité du code. Le premier livrable est un skateboard. Pensez au skateboard comme une métaphore de la plus petite chose que vous pouvez mettre dans les mains des utilisateurs et obtenir de vrais retours. Il est développable en une itération et permet d'avoir des premiers retours des utilisateurs sur le besoin réel. On peut apprendre par exemple que le skateboard couvre le besoin réel de se déplacer mais qu'il manque de stabilité. Le deuxième prototype ajoute alors cette dimension et on obtient une trottinette.
À la place du skateboard, on aurait pu proposer un ticket de bus. Si cette solution couvrait le besoin complètement, le projet pourrait s'arrêter à cette solution en une itération.
Quand le prototype n'est pas forcément porteur de valeur visible immédiatement pour le client, on parlera de prototype de type preuve de concept (POC). C'est le cas par exemple lorsqu'une itération est consacrée à une étude pour lever des verrous technologiques (ie implantation sur GPU).

4.2.2. Avantages du cycle incrémental

Encore une fois, l'important c'est que chaque prototype soit une version livrable du logiciel, sous doté en fonctionnalités, mais opérationnelle pour l'utilisateur qui peut repartir avec et faire ses essais.
Les avantages :

- Donne rapidement de la valeur au produit.
- Le client voit une évolution rassurante et constante du produit.
- Réduit le temps de mise sur le marché (*time to market*).

4.3. Collaboration client - développeur

Le client (utilisateur / donneur d'ordre / MOA / AMOA) est un partenaire à part entière de l'équipe de développement. Il est fortement impliqué dans le cycle de développement (quotidiennement dans l'idéal). C'est lui qui définit les priorités et décide des choix.

4.4. Équipe

- Taille réduite (< 10 personnes) de manière à garder une relation de proximité. On découpera en plusieurs équipes projets sur de très gros projet.
- Formée d'ingénieurs développeurs à parts égales.
- Auto-organisée.
- Rythme soutenable. Ne prendre qu'une liste de tâches réalisables dans l'itération.
- Dialogue face à face.
- Motivation des équipes.
- Propriété collective du code.

4.5. Intégration continue

L'intégration du travail de chaque développeur dans la version commune doit être faite quasi quotidiennement ; en fait à chaque fois que la fonctionnalité (petite) dont il est en charge est développée. Le but est bien évidemment d'éviter la mauvaise surprise de l'effet big bang de l'intégration en fin d'itération.
Il n'existe qu'une version courante du logiciel partagée par tous les développeurs.

- Cette version est opérationnelle et testée.

Chaque développeur en possède une copie de travail.

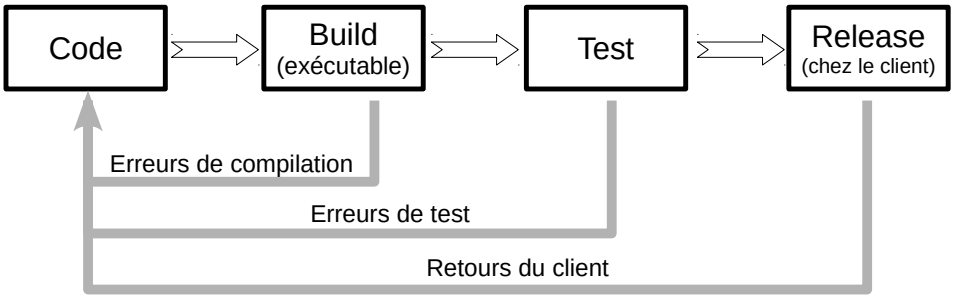
- Il développe sa fonctionnalité sur la version de travail en parfaite isolation.
- À la fin du développement de la fonctionnalité, il ajoute ses contributions à la version commune.

4.5.1. Cycle d'intégration continue

À chaque fois qu'un développeur pousse ses modifications dans la version commune, cela déclenche des mécanismes automatiques de vérification de la compilation du logiciel et de la non-régression de la version commune.

- La version doit être compilable et testable sans intervention humaine.
- Cela peut aller jusqu'au déploiement automatique chez le client.

On parle alors de DevOps : l'équipe gère le produit de sa définition jusqu'à sa mise en exploitation.
Dès qu'une erreur survient dans le cycle de DevOps, que ce soit une erreur de compilation, de test ou de déploiement, l'équipe se mobilise en urgence pour corriger l'erreur afin de toujours avoir une version commune opérationnelle.
La figure suivante schématise le cycle de DevOps.



Cycle d'intégration (déploiement) continue.

5. Auto-documentation

En agilité : seul compte le code. Il n'y a plus (ou peu) de production de documentation. Les documentations UML et textuelles mentent parce qu'elles ne sont pas toujours synchronisées avec les modifications qui sont apportées au code. Au contraire, le code ne ment pas.
L'auto-documentation consiste à écrire le logiciel avec une architecture et du code qui se lisent comme une documentation. Nous reparlerons de la façon d'écrire du code dans le chapitre 6.

5.1. Modélisations UML

Les modélisations UML ne sont utilisées que pour échanger entre développeurs et avancer dans la compréhension du domaine et des besoins à un instant donné. On utilise pour cela, une version allégée d'UML ; les 6 diagrammes présentés au chapitre 3 sont en général suffisants. Ce ne sont que des modélisations temporaires, mais elles doivent être exactes (syntaxe) pour éviter l'ambiguïté et doivent rester simples pour apporter une valeur claire.
Les diagrammes UML ne servent qu'à communiquer entre développeur et pas à documenter. Et si on a besoin de documentation, par exemple lors de la reprise de code alors on utilise la rétro-ingénierie du code (*reverse engineering*).

6. Exemples de méthodes agiles

Il n'existe pas une mais plusieurs méthodes agiles, dont les principales sont :

- **eXtreme-Programming** (aussi nommée XP) : la méthode pionnière dont la plupart des principes sont repris par les suivantes (et dans ce cours).
- **Kanban** : la plus simple et la plus connue par son fameux tableau (TODO/DOING/DONE).
- **Scrum** : la plus utilisée en France.

La mise en place concrète de l'agilité au sein d'une organisation mixe en général plusieurs méthodes. Personnellement, j'utilise Scrum comme cadre de la gestion de projet, Kaban pour l'organisation des tâches et XP pour le développement proprement dit. Ce qui importe, c'est que l'application de l'agilité soit elle-même agile : il faut s'approprier l'agilité mais en respectant ses valeurs et ses principes.

6.1. L'exemple de la méthode Scrum : vue globale

Scrum est considérée comme un framework, ie un cadre général pour le développement de projet. Elle n'est pas spécifique au développement logiciel. La Figure 1 résume la méthode.

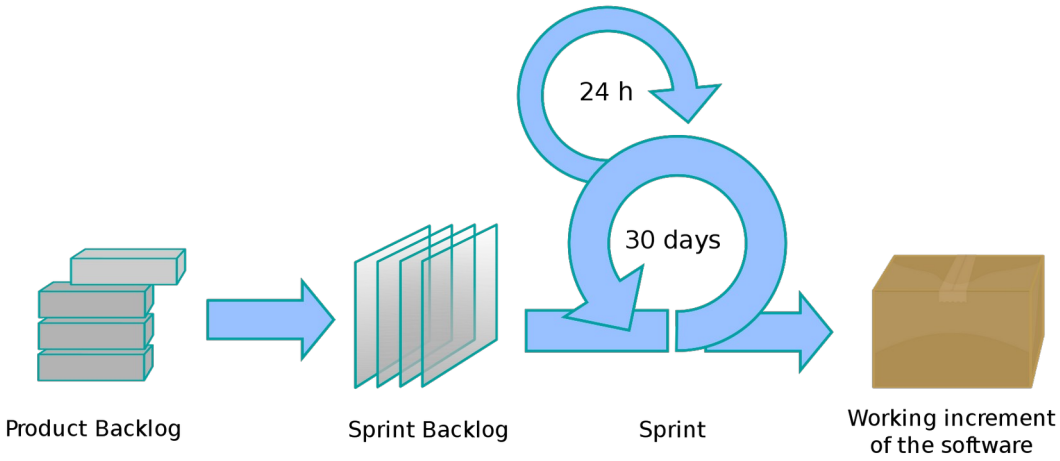


Figure 1 : Vue générale de la méthode Scrum.

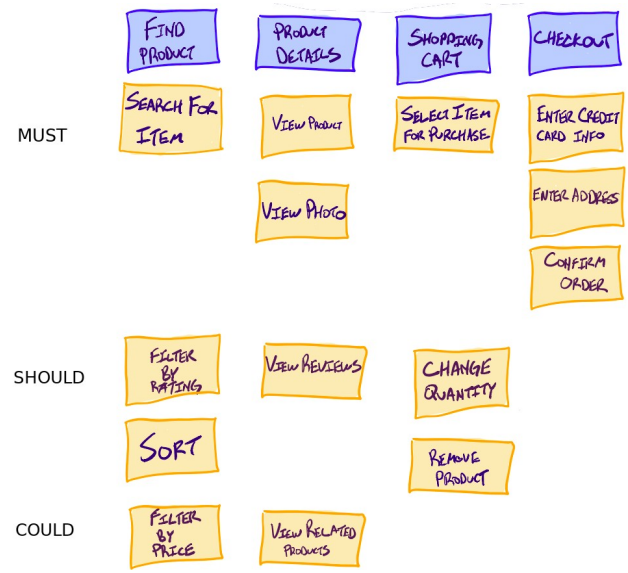
6.2. Les artefacts de Scrum

6.2.1. Carnet de produit (*product backlog*)

Ce carnet formalise la vision du produit (ie, logiciel) que le client souhaite réaliser sous la forme d'une liste de fonctionnalités à développer. Il contient :

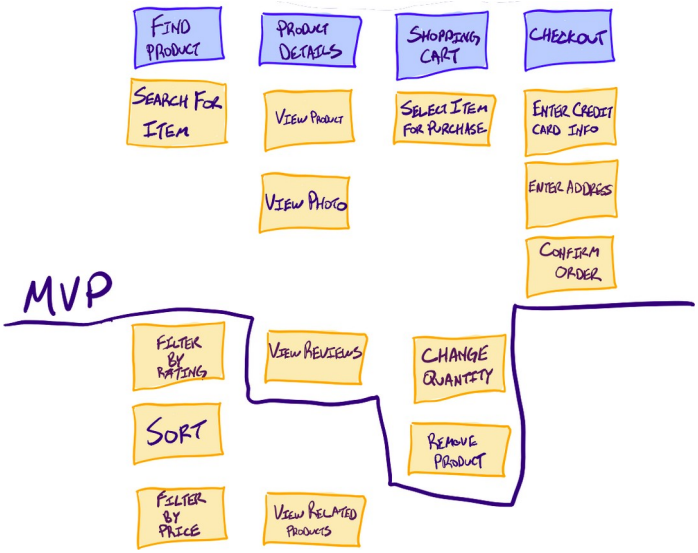
- Les exigences fonctionnelles et non fonctionnelles du produit avec une priorité entre elles (MUST, SHOULD COULD).
- Ce carnet de produit donne une vision globale du fonctionnement du futur produit : il doit raconter l'histoire du futur produit. Il se présente sous la forme d'une liste d'activités (eg., « Find Product ») qui se déclinent en une liste de tâches (eg., « Search for item »).

Nous reviendrons plus complètement sur la construction d'un carnet de produit au semestre 8.



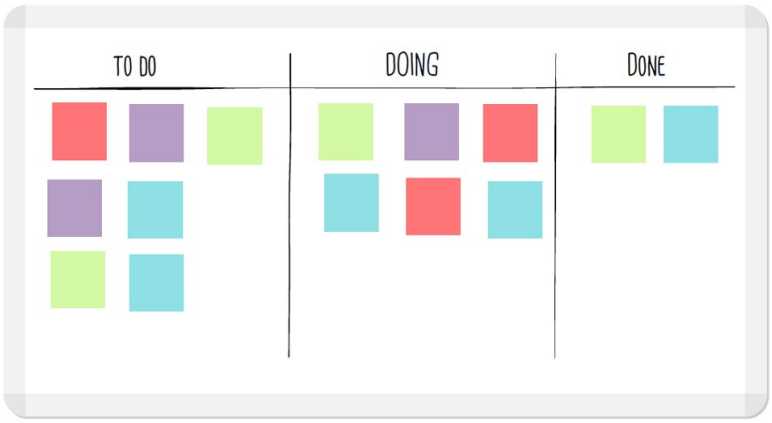
6.2.2. Carnet de Sprint (*sprint backlog*)

Parmi les tâches du carnet de produit, il faut choisir celles qui seront développées au cours de l'itération. Le principe est de convertir en premier les exigences qui apportent le plus de valeur ajoutée au client/utilisateur (dans les « must »). Cette liste doit dessiner les contours du MVP. Pour cela, les tâches seront choisies un peu partout dans les activités et pas seulement dans une seule activité.



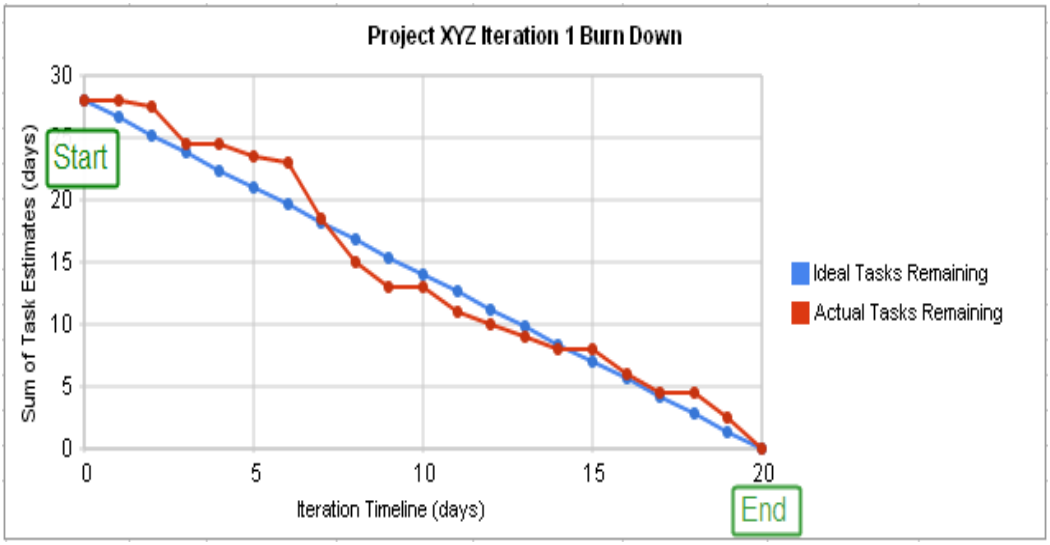
6.2.3. Tableau Kaban (*Kaban board*)

On reconnaît ici le fameux tableau d'organisation du travail d'une équipe. Chaque développeur prend une tâche de la travée TODO et la fait passer de DOING à DONE. Tous les développeurs ont accès à ce tableau et voient les tâches accomplies par les autres développeurs.
Il est possible d'ajouter d'autres travées, comme TESTING par exemple.



6.2.4. Graphique d'avancement

Ce graphique (***burndown chart***) donne l'évolution de quantité de travail restante par rapport au temps alloué. Le travail restant se situe sur l'axe vertical, alors que le temps est sur l'axe horizontal.
Il est utilisé à l'issue d'une itération pour améliorer la manière d'estimer le volume de tâches réalisables par l'équipe en une itération et aucunement à évaluer du volume de travail réalisé par l'équipe. D'ailleurs, pour éviter, la quantité de travail estimé pour chaque tâche est estimée en termes de points sans unité.



6.3. Les 4 cérémonies essentielles de SCRUM

Scrum est bâti sur des rituels simples mais efficaces qui permettent de rythmer la gestion de projet. Malgré leur caractère simpliste, ils sont en fait très efficaces pour donner de la cohésion au projet.

6.3.1. Le Sprint Planning au début d'une itération (< 8 h)

Cette réunion a pour but de construire le carnet de sprint (*sprint backlog*) à partir d'une sélection des tâches du carnet de produit (*product backlog*). Elle inclut aussi l'estimation des points sur les tâches choisies. Les grosses tâches doivent être décomposées en sous-tâches plus simples.

6.3.2. Le Daily Meeting (quotidiennement 15')

Lors de cette réunion quotidienne de début de journée, chaque membre décrit ce qu'il a fait hier, les

obstacles rencontrés et ce qu'il compte faire aujourd'hui. Toutefois, on n'y discute pas des problèmes ; cela est fait à l'issue de cette réunion avec les personnes concernées.

6.3.3. Le Sprint Review (2 h) à la fin d'une itération

Cette réunion a pour but de faire la démonstration du prototype auprès du client et de récolter ses retours. Le carnet de produit est alors mis à jour.

6.3.4. La Rétrospective (3 h) à la fin d'une itération

C'est une réunion interne à l'équipe. L'objectif est d'inspecter l'itération précédente, afin de déterminer ce qui a bien fonctionné et ce qui est à améliorer. Le graphique d'avancement (burndown chart) fournit un des outils pour cette réunion.
L'équipe déduit un plan d'actions d'amélioration qu'elle mettra en place lors de l'itération suivante.

7. Que retenir de ce chapitre ?

Les méthodes agiles proposent de nouvelles façons d'aborder le développement logiciel. En particulier, le développement suit un cycle itératif et incrémental.

- Chaque itération correspond au développement de plusieurs petites fonctionnalités qui sont intégrées aussitôt au logiciel, testées et mises en production.
- Chaque itération doit se terminer par la livraison d'une version opérationnelle et testée du logiciel en cours.
- Chaque incrément doit ajouter une valeur pour le client : le MVP.
- Chaque itération est une fin en soi : un mini-projet.

Attention : Les méthodes agiles ne renient pas les méthodes prédictives telles que le cycle en V, mais dans la mesure du possible elles privilégient les valeurs agiles.