Méthodes agiles

et SCRUM

Table des matières

I - Fondamentaux sur gestion de projet

1. Définition d'un projet

C'est un processus unique qui consiste en un ensemble d'activités coordonnées et maitrisées, comportant des dates de début et de fin dans le but d'atteindre un objectif conforme à des exigences spécifiques, incluant des contraintes de délais, de coûts et de ressources .

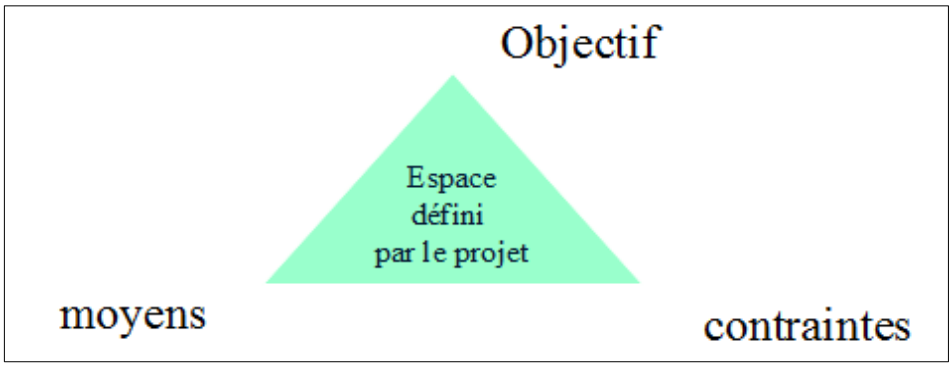
Principales caractériques d'un projet

Date de début	
Date de fin (délais)	fixe ou modifiable
Objectifs techniques ou fonctionnels	liés à des exigences spécifiées dont le périmètre peut évoluer
Moyens	fixes ou ajustables
Coûts	fixes ou ajustables

Typologies de projets

Taille	Nb intervenants , durée , ...
Environnement	industriel, recherche, gestion, infrastructure…
Technologies	Choix selon préférences ou contexte (cloud , ….)
Réalisation	en interne ou externalisée
Stratégie de conception	adaptation de produits existants, progiciel, conception spécifique
Contraintes	Concurrence, réglementation, ...

2. Espace d'un projet



3. Moyens/Capacités (ordres de grandeur)

NB : Ce tableau n'indique que des ordres de grandeur . Il est à interpréter avec beaucoup de recul. Selon le cœur de métier de l'entreprise (plus ou moins proche de l'informatique) et son contexte (plus ou moins concurrentiel) de grands écarts sont envisageables.
Néanmoins, bien que sujet à d'éventuelles ré-estimations, ce tableau a le mérite de faire réfléchir sur ce qu'il est envisageable d'entreprendre selon les moyens disponibles .

(*)	GE	ME	PE	TPE	TP (formation)
Moyens humains pour l'exploitation informatique	Souvent plus de 5 personnes à plein temps	1 à 5 personnes	De ¼ temps à 1 temps complet	Jusqu'à ¼ temps	Support assuré par le formateur (quelques heures)
Budget annuel informatique	Souvent plus de 1 000 000	30 000 à 999 000	5 000 à 50 000	500 à 10 000	100 à 499
Développements sur mesure	Souvent plus de 1500 j/h	50 à 1500 j/h	10 à 60 j/h	1 à 20 j/h	1 à 3j/h
Infrastructure informatique gérable en interne	Sous réseaux sécurisés, ESB ou ApiGateway sophistiqués, orchestration de services	Comme "PE" + ESB ou ApiGateway simples, éventuelle orchestration simple	Réseau tcp/ip Serveur(s) interne(s) ou en Iaas . Firewall Éventuel ESB ou ApiGateway minimaliste.	Connexion internet. Anti-virus Pas de serveur interne	Réseau tcp/ip
Types de logiciels gérables en interne (**)	Comme "ME" (en plus évolué) + ….	Comme "PE" (en plus évolué) + RH , Paye , Marketing,...	Comme "TPE" + compta analytique + cœur de métier	Compta & facturation, Bureautique	Logiciels "open source" pour "TP".

(*) GE : Grande Entreprise , M : Moyenne , P : Petite , TPE : Très Petite Entreprise .
(**) Sinon (trop lourd à gérer en interne) –> location d'une application externalisée (Saas) ou autres solutions alternatives.

…

4. Gérer un projet

- **Gérer des activités**
- **Gérer des moyens (ressources humaines ou techniques)**
- **Atteindre un objectif**
- **Respecter les contraintes**
  - **Temps**
  - **Financières**

Quelques difficultés :

- Evolution de périmètre (hausse ou baisse)
- Dépassement des coûts et/ou des délais
- Report d'activité en phase "maintenance / production"
- Inadéquation avec les attentes du client
- Tension entre les différents acteurs
- Evolution du système
- Conduite du changement

Facteurs de réussite	*Principales causes d'échecs*
• Client ou demandeur motivé • Bien cadrer et maîtriser les attentes du projet • Avoir des moyens adaptés aux objectifs • Une organisation acceptée par tous les acteurs • Une ambiance positive et volontaire	• Client ou demandeur peu impliqué • Objectifs mal définis, mal maîtrisés ou au périmètre fluctuant • Avoir des moyens inadaptés aux objectifs • Une organisation mal comprise ou pas respectée • Une ambiance négative • La dépendance forte à un fournisseur

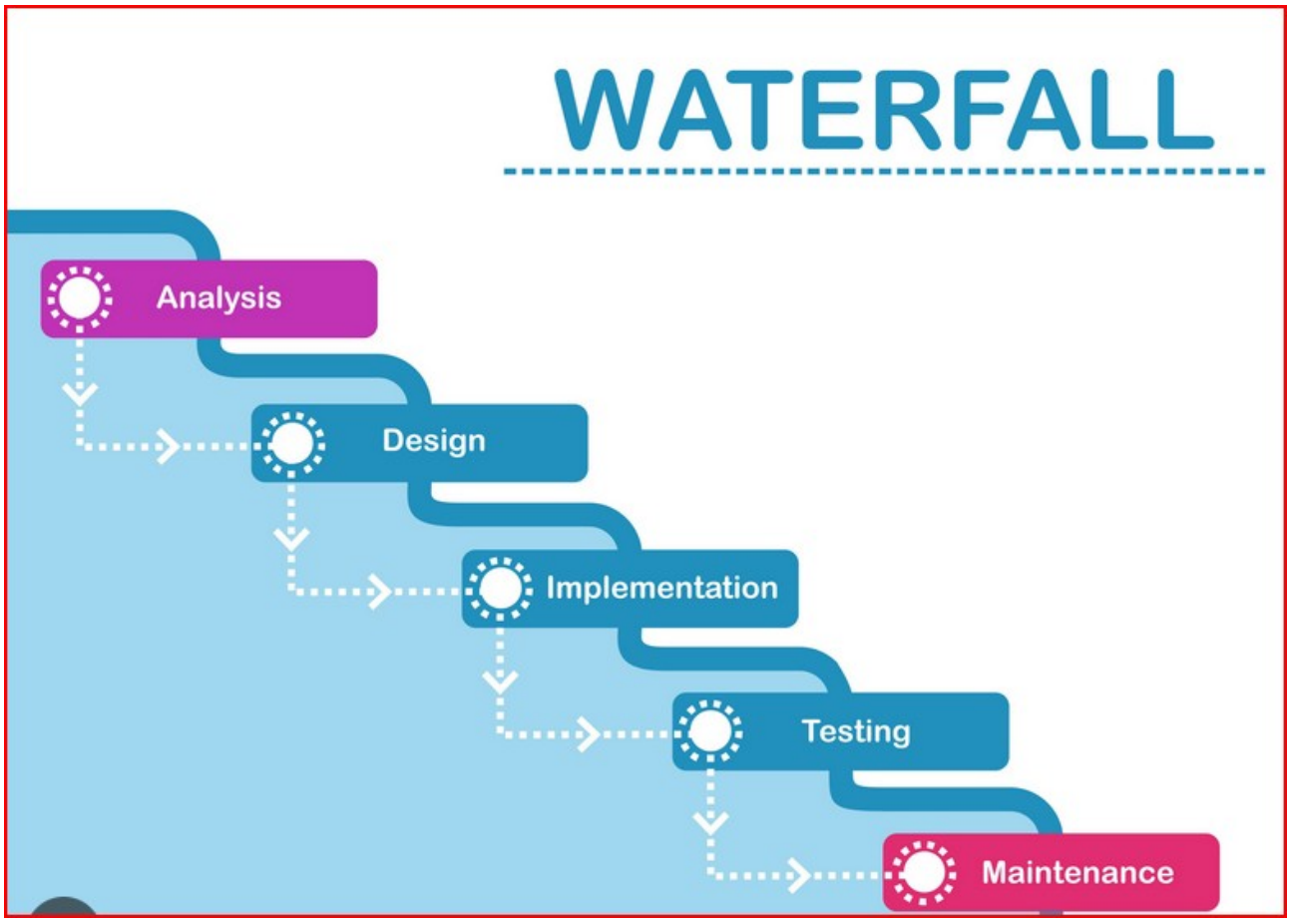
Statistiques "Standish Group" de l'année 2015 :

Projets réussis	**29 %**
Projets en difficultés	**52 %**
Projets échoués	**19 %**

Seuls environ 15% des projets sont réalisés dans les temps,
tous les autres sont en retard (85%)

5. Méthodes prédictives

5.1. Effet tunnel ou "waterfall"



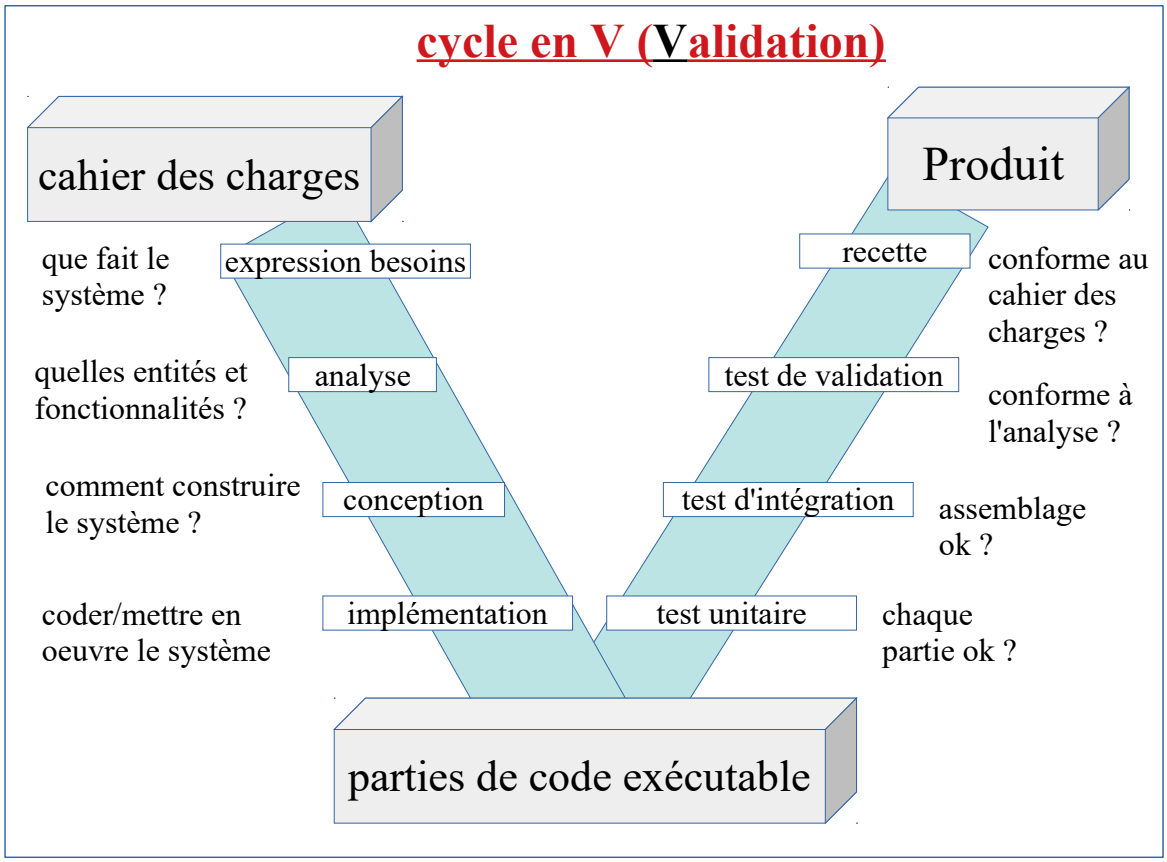
Démarche trop séquentielle :

Portée globale/complète dès le départ, on ne passe pas à la phase suivante tant que la phase en cours n'est pas terminée.

Syndrôme de l'effet tunnel :

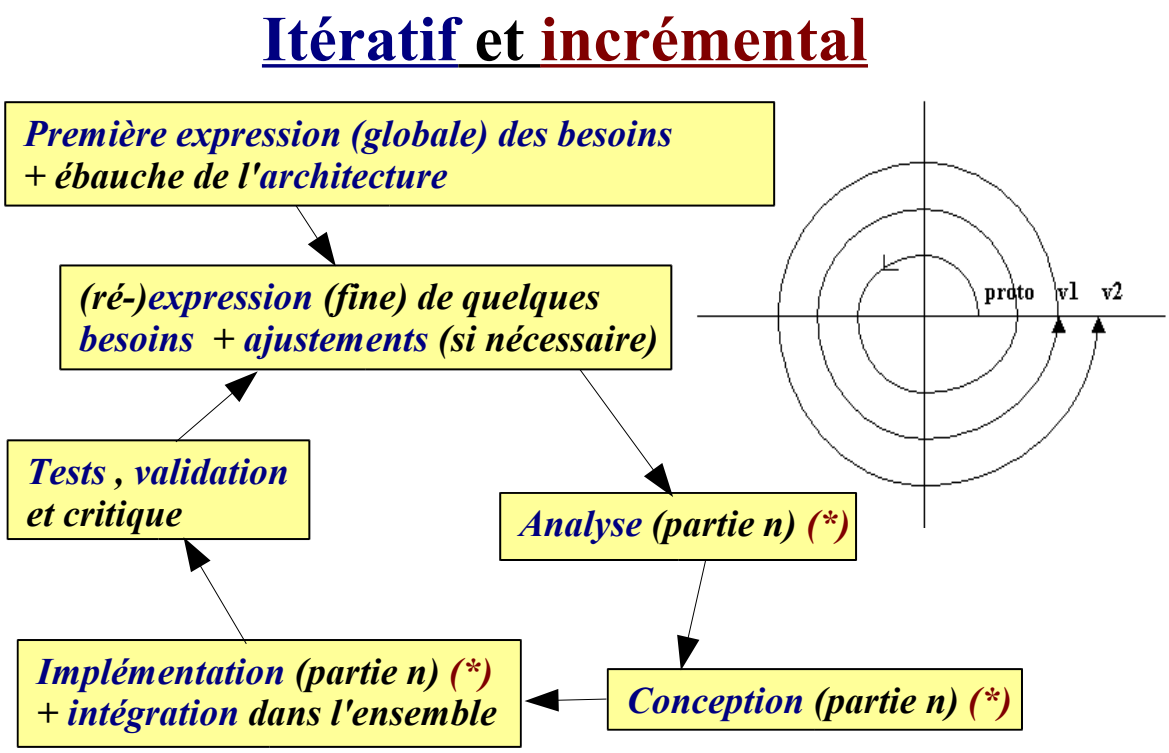
- On passe des mois et des mois (voir des années) à modéliser puis programmer une application.
  Tant que le développement logiciel n'est pas fini , on est dans un tunnel et on ne sait pas encore si ça va bien fonctionner et répondre aux besoins .
- Aucune valeur avant la fin du projet
- Le produit peut ne plus correspondre aux besoins qui peuvent avoir changés

5.2. Cycle en V (comme Validation)



6. Méthodes itératives

Cycle itératif et incrémental



(*) et éventuels correctifs sur parties (1,...,(n-1)) / refactoring

- **Le logiciel est construit comme une suite de petits incréments** (prototype, alpha , beta , v1 , v2, ...)
- **Les fonctionnalités les plus importantes sont réalisées en premiers.**
- **À chaque itération, il y a souvent des ré-estimations et des réajustages (fonctionnalités , …)**

7. Méthodes prédictives et itératives

Méthode prédictive ou empirique

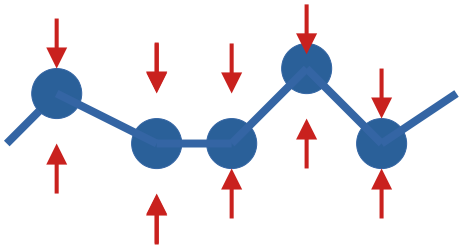
Approche prédictive

ça commence avec un plan et toutes les exigences

logiciel avec les exigences plus ou mois bien implémentées

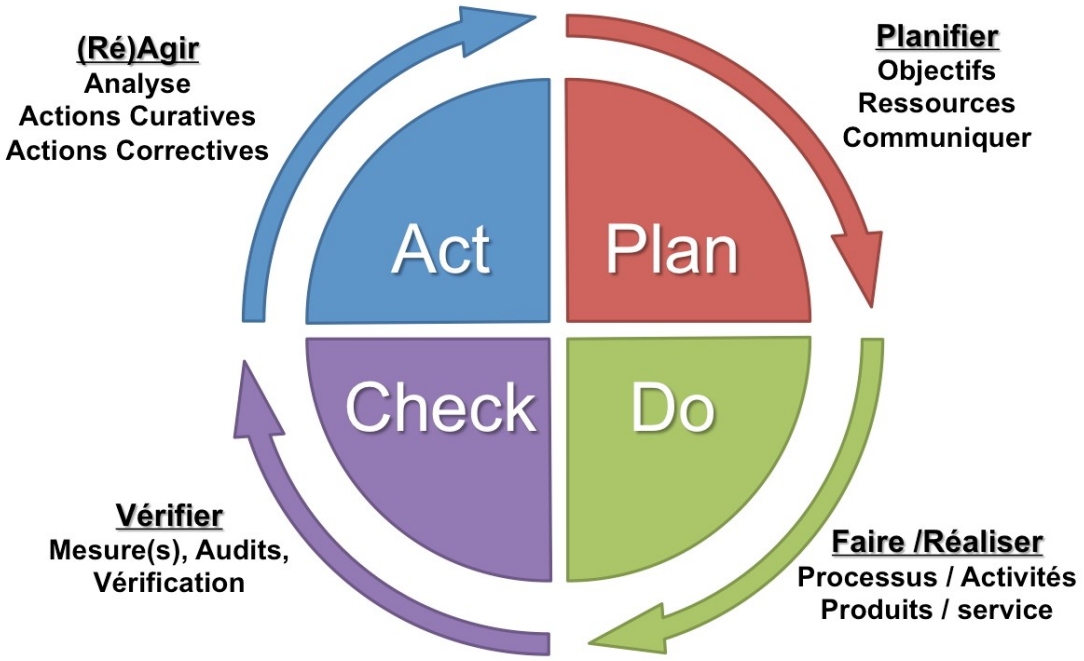
Approche empirique

ça commence avec une vision et quelques exigences prioritaires

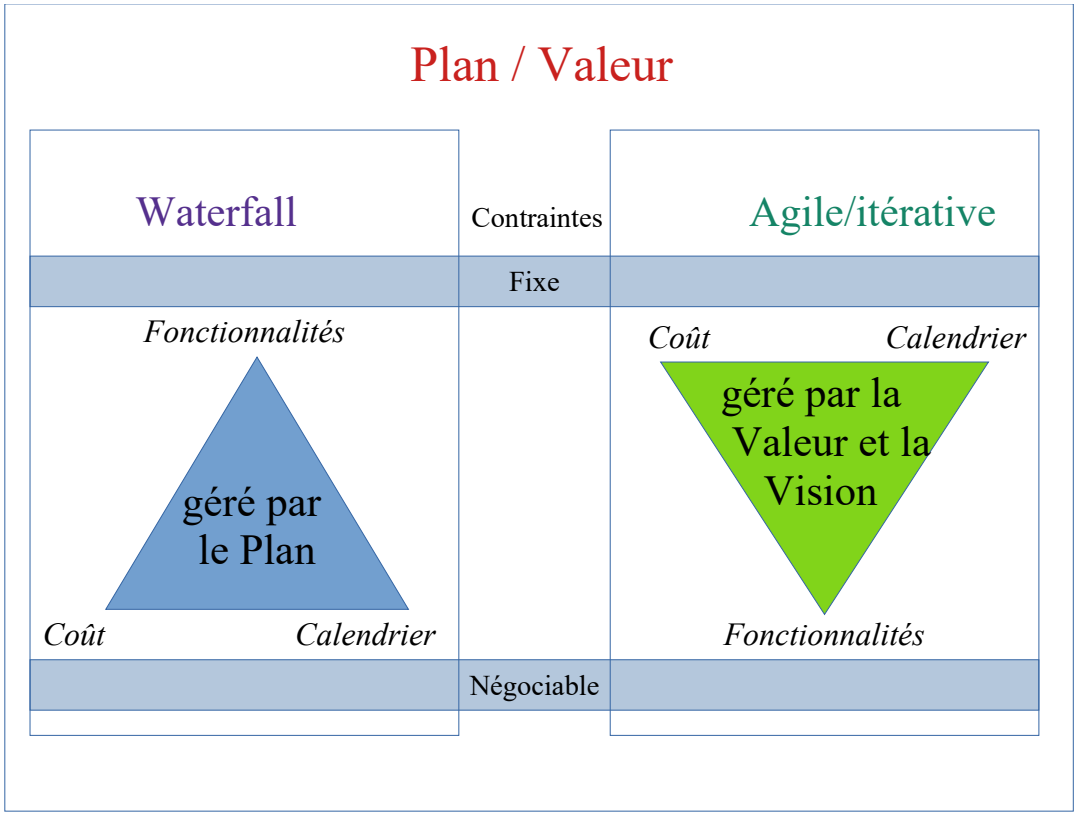


logiciel avec principaux objectifs atteints

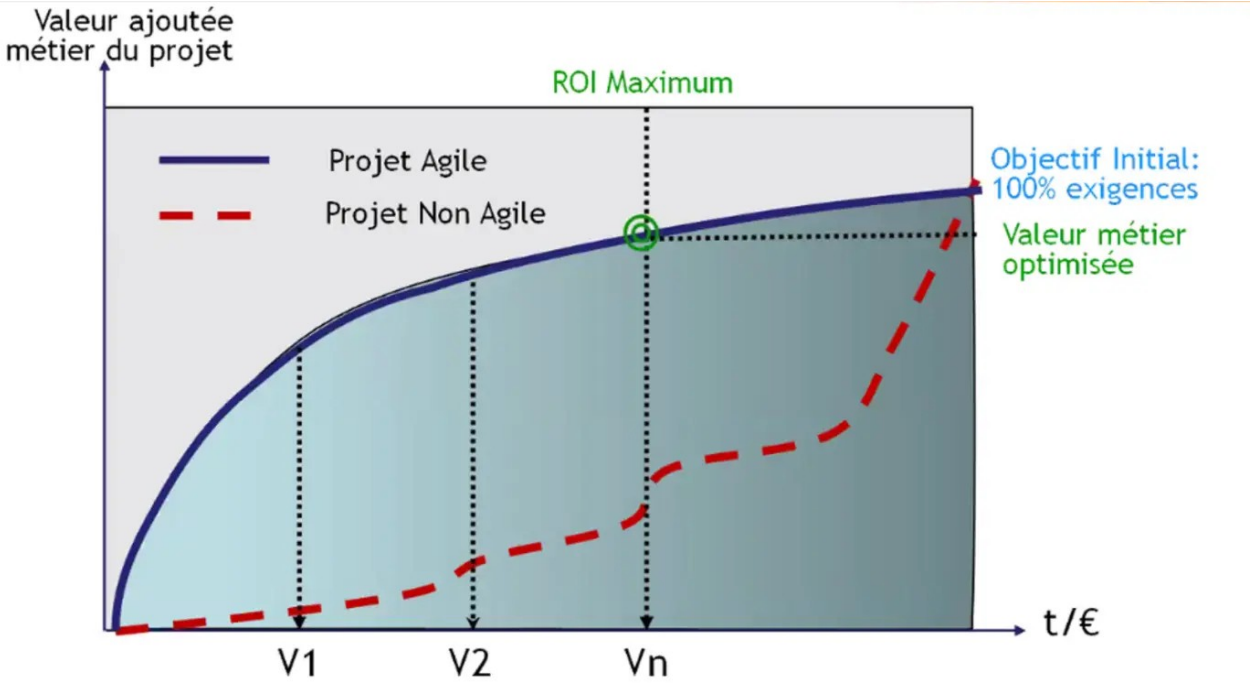
Amélioration continue (processus empirique)



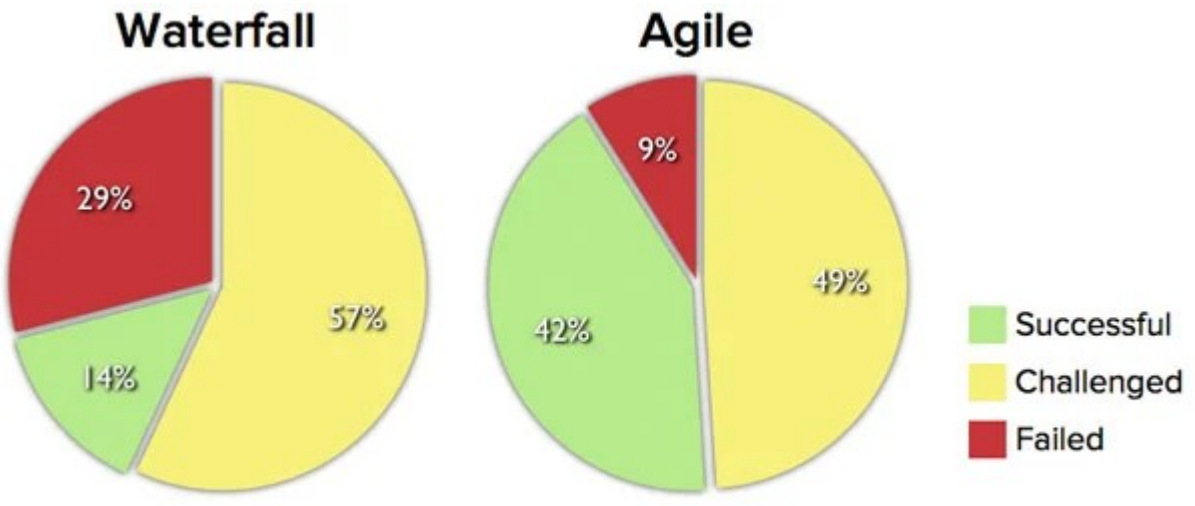
Prendre en compte ce qui va et qui ne va pas et tirer rapidement les bénéfices des leçons apprises.



Optimisation du retour sur investissement (ROI) :



- **Plus tôt on délivre des fonctionnalités plus tôt on pourra facturer le client**
- **En mode "waterfall" , le ROI ne commence qu'à la fin si pas de retard et si c'est toujours ce que veut le client .**



Source: The CHAOS Manifesto, The Standish Group, 2012.

II - Méthodes agiles

1. Méthodes agiles

Méthodes adaptatives plutôt que prédictives .

1.1. Origine des méthodes agiles : une réponse à un malaise

1.1.a. Quelques éléments du malaise du début des années 2000

- Contexte économique difficile – **pression forte**
- **Rapports quelquefois tendus ente moa et moe**
- **Confusion entre les besoins estimés et réels** (mauvaise communication)
- **Les négociations contractuelles (positions retranchées) font oublier l'objectif initial: chacun s'abrite derrière les modalités d'un forfait qui fige dans le marbre des spécifications très souvent incomplètes.**
- **beaucoup de dérapages (budget , retards , ...)**
- **certains projets sont abandonnés (ou sont fortement restreints).**
- **Les Méthodes (Merise=has been , UML/RUP : trop lourd , ...) sont souvent mal utilisées et les AGL ne tiennent pas leurs promesses.**

1.1.b. Facteurs clefs des échecs

- Le **manque de communication** à tout niveau (moa/moe, ...)
- Une **mauvaise compréhension des besoins**
- L'**insuffisance des tests**
- L'absence d'une démarche prudente "gros projet – commencer petit"
- Les **effets "Tunnel"** (cycle non itératif mais linéaire)
- L'insuffisance de l'architecture
- L'absence de maturité des outils utilisés
- La mauvaise formation des personnes
- Le cadre contractuel inadapté

1.1.c. Facteurs à prendre en compte pour avoir une chance de réussir

Il faut prendre en compte les risques !!!

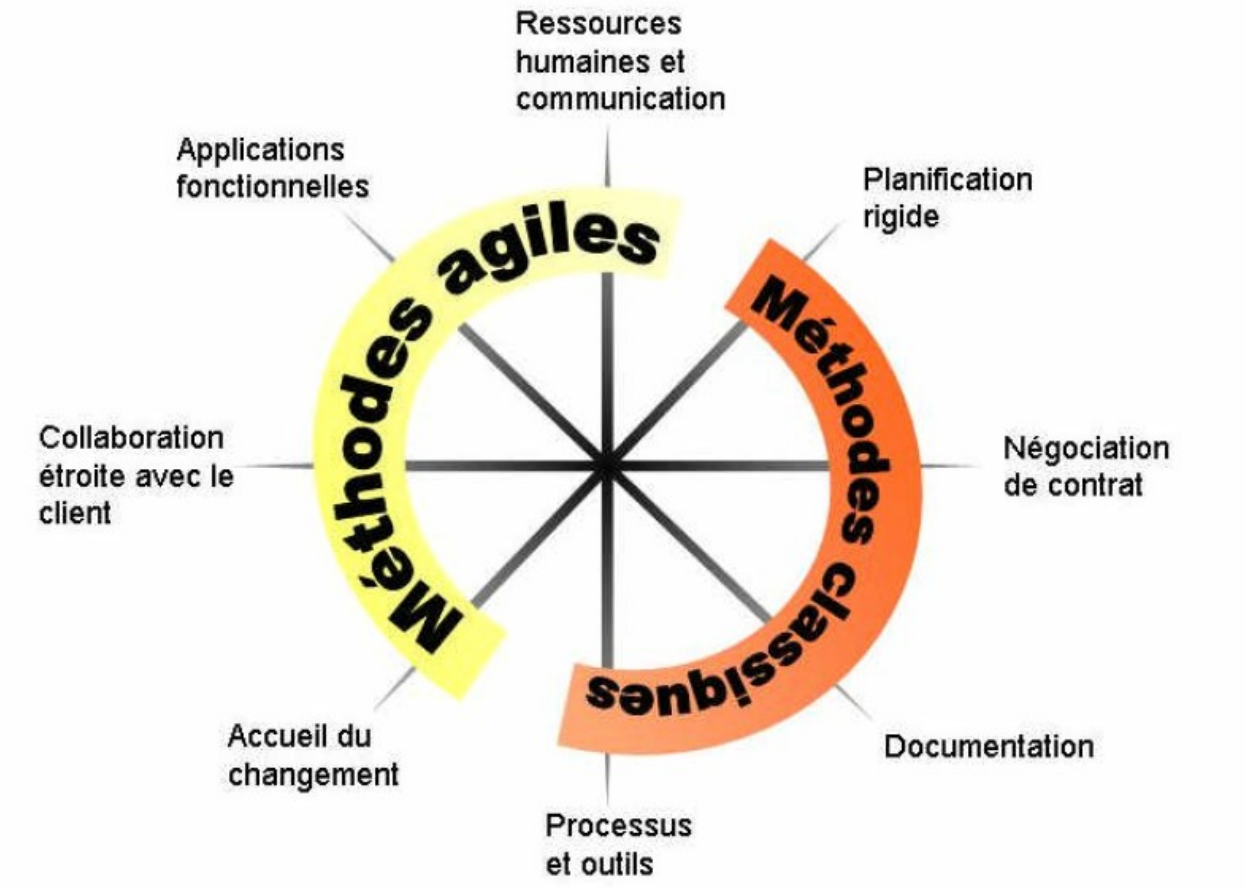
4 grands facteurs:

- **Coût**
- **Qualité**
- **Durée**
- **Périmètre fonctionnel**

1.2. Principales caractéristiques des méthodes agiles

1.2.a. Manifeste des priorités (fondamentaux des méthodes agiles)

- **Priorité des personnes et des échanges/communications/interactions sur les procédures et les outils**
- **Priorité de la collaboration continue avec le client sur la négociation de contrat**
- **Priorité d'applications opérationnelles (avec commentaires) sur une documentation exhaustive (pléthorique) .**
- **Priorité de l'acceptation des changements sur la planification**



1.2.b. Grands Principes des méthodes agiles

- **Délivrer rapidement et très fréquemment des versions opérationnelles, pour favoriser un feed-back client permanent**
- **Assurer une coopération forte entre client et développeurs**
- **Garder un haut niveau de motivation(rôle du chef)**
- **Le fonctionnement de l'application est le premier indicateur du projet**
- **Garder un rythme soutenable (pas plus de 40 heures par semaine)**
- **Viser l'excellence technique et la simplicité**
- **Accueillir favorablement le changement (sachant qu'il faut du courage pour accepter de "jeter" certaines parties devenues inutiles).**
- **Se remettre en cause régulièrement (refactoring , tests , ...)**

1.3. Le manifeste agile

Février 2001, 17 leaders en développement « léger » se rencontrent.

Adoption du terme « Agile »

Entente sur l'ensemble des valeurs fondamentales qui devraient être à la base de toutes les méthodologies Agiles.

De ces valeurs, 12 principes fondamentaux en sont extraits.

Le détail des opérations a été laissé aux bons soins des diverses méthodologies Agiles (XP, SCRUM, ...)

Les 12 principes du manifeste agile :

1. Notre 1ère priorité est de satisfaire le client en livrant tôt et régulièrement des logiciels utiles.
2. Le changement est bienvenu, même tardivement dans le développement.
3. Livrer fréquemment une application fonctionnelle.
4. Les gens de l'art et les développeurs doivent collaborer.
5. Bâtissez le projet autour de personnes motivées.
6. La méthode la plus efficace de communication est une conversation en face à face.
7. Un logiciel fonctionnel est la meilleure unité de mesure de la progression du projet.
8. Encouragez un rythme de développement soutenable.
9. Une attention continue à la qualité de la conception améliore l'agilité.
10. La simplicité: l'art de maximiser la quantité de travail à ne pas faire - est essentielle.
11. Equipes qui s'auto-organisent.
12. A intervalle régulier, l'équipe réfléchit aux moyens de devenir plus efficace.

Communication efficace : en face à face

- 93 % de la communication est non verbale
- 7% Verbal (Verbes, Mots)
- 38 % Vocal (Intonations perçues)
- 55 % Visuel (Expression du visage, posture)

1.4. Changements de posture (en agile)

FAIRE AGILE, c'est avant tout un état d'esprit et un changement de posture.

1.5. Panorama des principales méthodes agiles

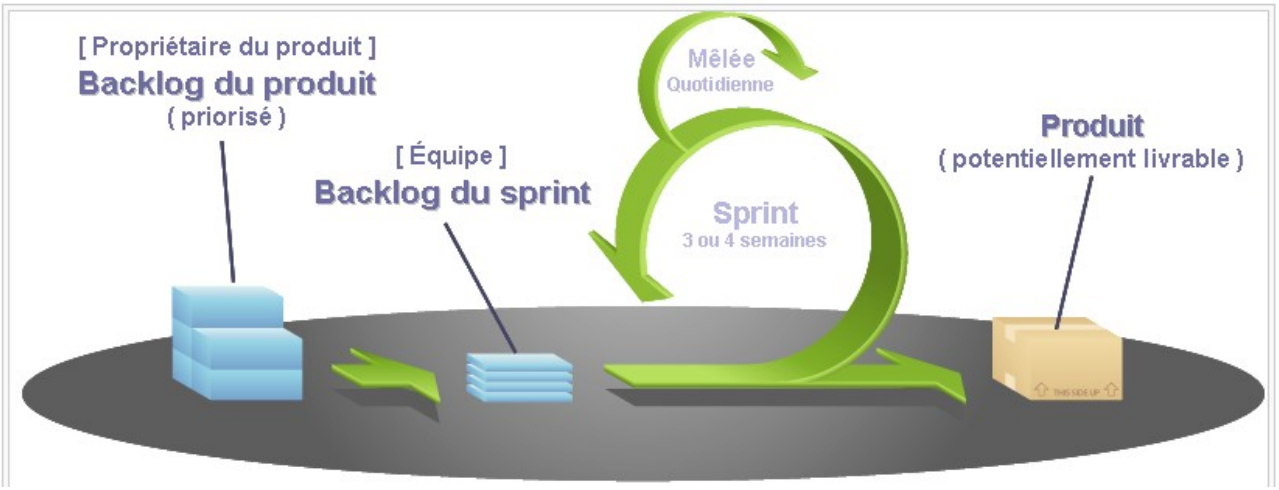
Méthode agile	Nombre optimal de personnes dans l'équipe	Principales caractéristiques
Crystal Clear	<=6	Pratiques très peu contraignantes. Méthode très peu formalisée
XP (eXtreme Programming)	<=12	Importance de la communication informelle et de l'autonomie de l'équipe Intégration journalière
SCRUM		Réunion quotidienne de motivation, de synchronisation, de déblocage et de partage de connaissances.======== Phase initiale , sprint , cloture
RAD (Rapid. App. Dev.)		Ancètre des méthodes agiles insiste sur ce cycle incrémental et itératif .
FDD (Feature Driven Development)	<=20	Feature et Features set : fonctionnalité et groupe de fonctionnalités). Priorité donnée aux fonctionnalités porteuses de valeur . ==== Itérations très courtes – livrables / features
DSDM (Dynamic Software Development Method)		Très sensible aux rôles et responsabilités bien définis: Sponsor exécutif, ambassadeur utilisateur visionnaire, utilisateur conseiller, ... en plus du facilitateur et des rapporteurs
ASD (Adaptive Software Development)		Extensible mais très générale (à adapter au cas pas pas)
[partiellement UP ?]		Pour gros projet – formalisation importante
...		

III - Vue d'ensemble sur SCRUM

1. Méthode agile SCRUM

"**SCRUM**" est une méthode agile pour la gestion de projet de développement de logiciels .

La métaphore de **Scrum** (mêlée du rugby) apparaît pour la première fois dans une publication de Takeuchi et Nonaka intitulée *The New New Product Development Game* qui s'appliquait à l'époque au monde industriel.



1.1. Principales caractéristiques de la méthode SCRUM

(selon wikipédia)

Le terme ***Scrum*** est emprunté au rugby à XV et signifie ***mêlée***. Ce processus s'articule en effet autour d'une **équipe soudée, qui cherche à atteindre un but**, comme c'est le cas en rugby pour avancer avec le ballon pendant une mêlée.

Le principe de base de Scrum est de **focaliser l'équipe sur une partie limitée et maîtrisable des fonctionnalités à réaliser**. Ces **incréments** se réalisent successivement lors de périodes de durée fixe de une à quatre semaines, appelées **sprints**.

Chaque sprint possède, préalablement à son exécution, un **but** à atteindre, défini par le *directeur de produit*, à partir duquel sont choisies les fonctionnalités à implémenter dans cet incrément. Un sprint aboutit toujours à la livraison d'un produit partiel fonctionnel. Pendant ce temps, le *ScrumMaster* a la charge de minimiser les perturbations extérieures et de résoudre les problèmes non techniques de l'équipe.

Un principe fort en Scrum est la **participation active du client pour définir les priorités dans les fonctionnalités du logiciel et pour choisir celles qui seront réalisées dans chaque sprint**. Il peut à tout moment compléter ou modifier la liste des fonctionnalités à produire, mais jamais celles qui sont en cours de réalisation pendant un sprint. ...

1.2. Rôles des intervenants/participants dans la méthode SCRUM

Directeur de produit	*Product Owner*	Représentant des clients et utilisateurs. C'est lui qui définit l'**ordre** dans lequel les fonctionnalités seront développées et qui prend les décisions importantes concernant l'orientation du projet. Le terme *directeur* n'est d'ailleurs pas à prendre au sens hiérarchique du terme, mais dans le sens de l'*orientation*. Dans l'idéal, le directeur de produit travaille dans la même pièce que l'équipe. Il est important qu'il reste très disponible pour répondre aux questions de l'équipe et pour lui donner son avis sur divers aspects du logiciel (interface utilsateur par exemple)
Equipe *(de développenent , auto gérée)*	*Team*	Il n'y a pas de notion de hiérarchie interne : toutes les décisions sont prises ensemble et personne ne donne d'ordre à l'équipe sur sa façon de procéder. Contrairement à ce que l'on pourrait croire, les équipes auto-gérées sont celles qui sont les plus efficaces et qui produisent le meilleur niveau de qualité de façon spontanée. L'équipe s'adresse directement au directeur de produit. Il est conseillé qu'elle lui montre le plus souvent possible le logiciel développé pour qu'il puisse ajuster les détails.
Facilitateur / animateur	*Scrum-Master*	Chargé de protéger l'équipe de tous les éléments perturbateurs extérieurs et de résoudre ses problèmes non techniques (administratifs par exemple). Il doit aussi veiller à ce que les valeurs de Scrum soient appliquées, il est le garant de la méthode. En revanche, il n'est pas un chef de projet ni un intermédiaire de communication avec les clients. On parle parfois d'**équipe étendue**, qui intègre en plus le *ScrumMaster* et le directeur de produit. Ce concept renforce l'idée que client et fournisseur travaillent d'un commun effort vers le succès du projet.
Intervenants (externes)	*Stakeholders*	Personnes qui souhaitent avoir une vue sur le projet sans réellement s'investir dedans. Il peut s'agir par exemple d'experts techniques ou d'agents de direction qui souhaitent avoir une vue très éloignée de l'avancement du projet.

1.3. Planification "SCRUM"

Releases et sprints

Scrum est un processus *itératif* : les itérations sont appelées des **sprints** et durent en théorie 30 jours calendaires. En pratique, les itérations durent généralement entre 2 et 4 semaines.

Chaque sprint possède un **but** et on lui associe une liste d'*items de backlog de produit* (fonctionnalités) à réaliser. Ces items sont décomposés par l'équipe en tâches élémentaires de quelques heures, les *items de backlog de sprint*. ...

Pour améliorer la lisibilité du projet, on regroupe généralement des itérations en **releases**.

Bien que ce concept ne fasse pas explicitement partie de Scrum, il est utilisé pour mieux identifier

les versions. En effet, comme chaque sprint doit aboutir à la livraison d'un produit partiel, une release permet de marquer la livraison d'une version aboutie, susceptible d'être mise en exploitation.

Réunion quotidienne :

Au quotidien, une réunion, le ***ScrumMeeting***, permet à l'équipe et au Scrum Master de *faire un point d'avancement sur les tâches et sur les difficultés rencontrées*.

1.4. Gestion des besoins/fonctionnalités

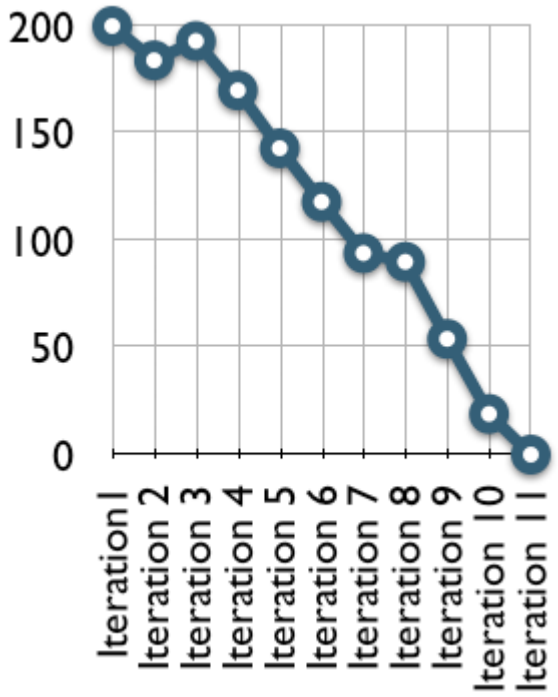
Backlog de produit :

L'objectif est d'établir une liste de fonctionnalités à réaliser, que l'on appelle **backlog de produit** (NDT : Le terme *backlog* peut être traduit par *cahier*, *liste* ou *carnet de commandes).*

À chaque item de backlog sont associés deux attributs :

- une estimation en **points arbitraires**
- une valeur *client*, qui est définie par le directeur de produit (retour sur investissement par exemple).

La somme des points des items du backlog de produit constitue le *reste à faire* total du projet. Cela permet de produire un **release burndown chart**, qui montre les points restant à réaliser au fur et à mesure des sprints.



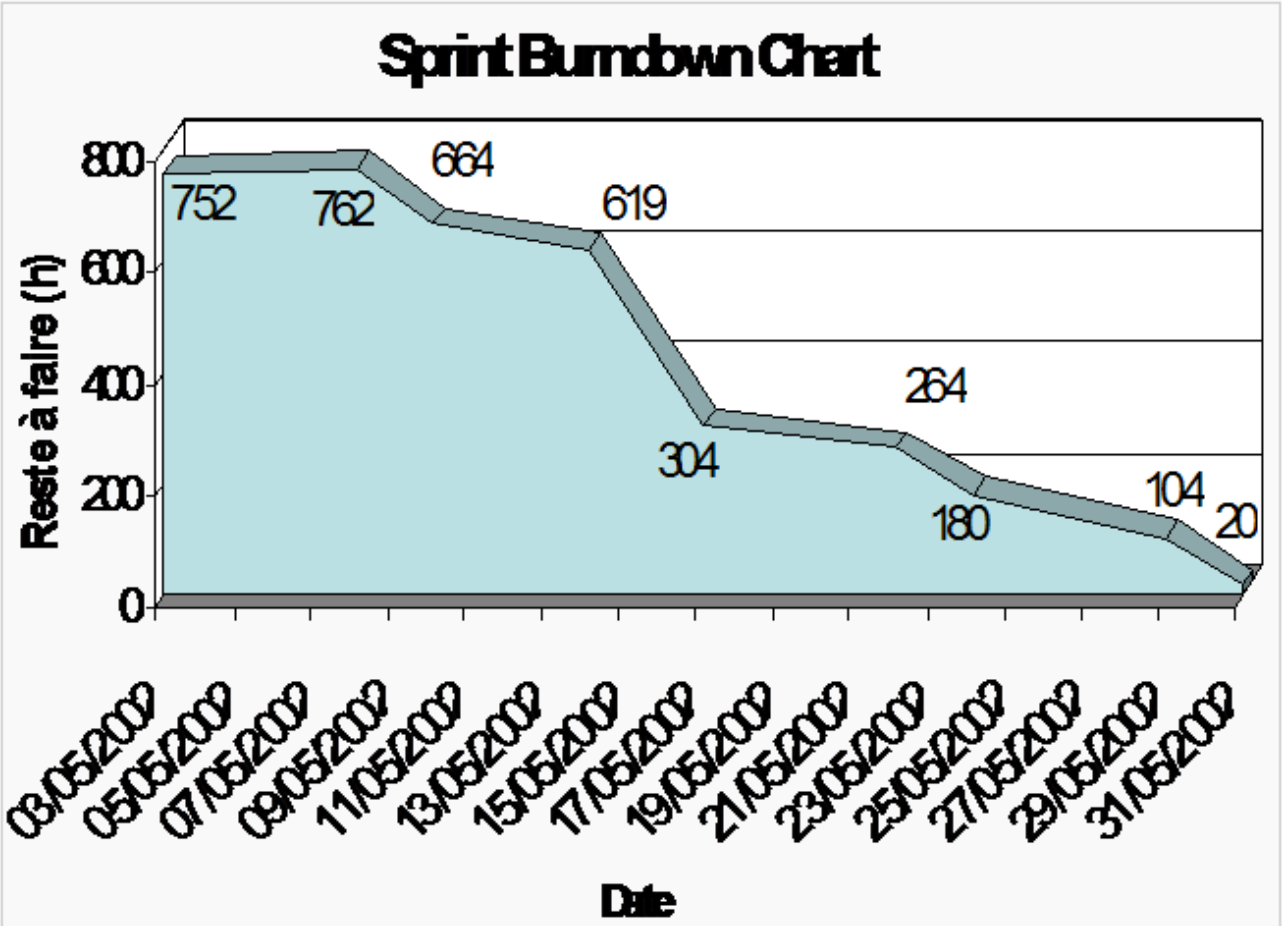
Backlog de sprint :

Lorsqu'on démarre un sprint, on choisit quels items du backlog de produit seront réalisés dans ce sprint. L'équipe décompose ensuite chaque item en liste de tâches élémentaires (techniques ou non),

chaque tâche étant estimée en heures et ne devant pas durer plus de 2 jours. On constitue ainsi le **backlog de sprint**.

Pendant le déroulement du sprint, chaque équipier s'affecte des tâches du backlog de sprint et les réalise. Il met à jour régulièrement dans le backlog du sprint le reste à faire de chaque tâche. Les tâches ne sont pas réparties initialement entre tous les équipiers, elles sont prises au fur et à mesure que les précédentes sont terminées.

La somme des heures des items du backlog de sprint constitue le *reste à faire* total du sprint. Cela permet de produire un **sprint burndown chart** qui montre les heures restantes à réaliser au fur et à mesure du sprint.



1.5. Eléments pour estimations

Scrum ne définit pas spécialement d'unités pour les items des backlogs. Néanmoins, certaines techniques se sont imposées de fait.

Items de backlog de produit = user stories = use case UML
Les items de backlog de produit sont souvent des *User Stories* empruntées à Extreme Programming. Ces User Stories sont estimées en ***points relatifs***, sans unité. L'équipe prend un item représentatif et lui affecte un nombre de points arbitraire. Cela devient un référentiel pour estimer les autres items. Par exemple, un item qui vaut 2 points représente deux fois plus de travail qu'un item qui en vaut 1. Pour les valeurs, on utilise souvent les premières valeurs de la suite de Fibonacci (1,2,3,5,8,13), qui évitent les difficultés entre valeurs proches (8 et 9 par exemple).

L'intérêt de cette démarche est d'avoir une idée du travail requis pour réaliser chaque fonctionnalité sans pour autant lui donner une valeur en jours que le directeur de produit serait tenté de considérer

comme définitivement acquise. En revanche, on utilise la *vélocité* pour planifier le projet à l'échelle macroscopique de façon fiable et précise.

Calcul de vélocité :
Une fois que tous les items de backlog de produit ont été estimés, on attribue un certain nombre d'items à réaliser aux sprints successifs. Ainsi, une fois un sprint terminé, on sait combien de points ont été réalisés et on définit alors la **vélocité** de l'équipe, c'est-à-dire le nombre de points qu'elle peut réaliser en un sprint.

En partant de cette vélocité et du total de points à réaliser, on peut déterminer le nombre de sprints qui seront nécessaires pour terminer le projet (ou la release en cours). L'intérêt, c'est qu'on a une vision de plus en plus fiable (retours d'expérience de sprint en sprint) de la date d'aboutissement du projet, tout en permettant d'aménager les items de backlog du produit en cours de route.

Items de backlog de sprint=tâche (quelques heures) :
Les items de backlog de sprint sont généralement exprimés en heures et ne doivent pas dépasser 2 journées de travail, sinon il convient de les décomposer en plusieurs items. Par abus de langage, on emploie le terme de *tâches*, les concepts étant très proches.

1.6. Déroulement d'un sprint

Réunion de planification :
Tout le monde est présent à cette réunion, qui ne doit pas durer plus de 4 heures. La **réunion de planification** (*Sprint Planning*) consiste à définir d'abord un but pour le sprint, puis à choisir les items de backlog de produit qui seront réalisés dans ce sprint. Cette première partie du *sprint planning* représente l'engagement de l'équipe. Compte tenu des conditions de succès énoncées par le directeur de produit et de ses connaissances techniques, l'équipe s'engage à réaliser un ensemble d'items du backlog de produit.

Dans un second temps, l'équipe décompose chaque item du backlog de produit en liste de tâches (items du backlog du sprint), puis estime chaque tâche en heures. Il est important que le directeur de produit soit présent dans cette étape, il est possible qu'il y ait des tâches le concernant (comme la rédaction des règles métier que le logiciel devra respecter et la définition des tests fonctionnels).

Au quotidien :
Chaque journée de travail commence par une réunion de 15 minutes maximum appelée **mêlée quotidienne** (*Daily Scrum*). Seuls l'équipe, le directeur de produit et le ScrumMaster peuvent parler, tous les autres peuvent écouter mais pas intervenir (leur présence n'est pas obligatoire). A tour de rôle, chaque membre répond à 3 questions :

- *Qu'est-ce que j'ai fait hier ?*
- *Qu'est-ce que je compte faire aujourd'hui ?*
- *Quelles sont les difficultés que je rencontre ?*

L'équipe se met ensuite au travail. Elle travaille dans une même pièce, dont le ScrumMaster a la responsabilité de maintenir la qualité d'environnement. Les activités se déroulent éventuellement en parallèle : analyse, conception, codage, intégration, tests, etc. Scrum **ne définit volontairement pas** de démarche technique pour le développement du logiciel : l'équipe s'auto-gère et décide en toute autonomie de la façon dont elle va travailler.

Remarque : Il est assez fréquent que les équipes utilisent la démarche de *développement guidé par les tests* (Test Driven Development en anglais). Cela consiste à coder en premier lieu les modules de test vérifiant les contraintes métier, puis à coder ensuite le logiciel à proprement parler, en exécutant les tests régulièrement. Cela permet de s'assurer entre autres de la non-régression du logiciel au fil des sprints.

Revue de sprint :
À la fin du sprint, tout le monde se réunit pour effectuer la **revue de sprint**, qui dure au maximum 4 heures. L'objectif de la revue de sprint est de valider le logiciel qui a été produit pendant le sprint. L'équipe commence par énoncer les items du backlog de produit qu'elle a réalisés. Elle effectue ensuite une démonstration du logiciel produit. C'est sur la base de cette démonstration que le directeur de produit valide chaque fonctionnalité planifiée pour ce sprint.

Une fois le bilan du sprint réalisé, l'équipe et le directeur de produit proposent des aménagements sur le backlog du produit et sur la planification provisoire de la release. Il est probable qu'à ce moment des items soient ajoutés, modifiés ou réestimés, en conséquence de ce qui a été découvert.

Rétrospective du sprint :
La **rétrospective du sprint** est faite en interne à l'équipe (incluant le ScrumMaster). L'objectif est de comprendre ce qui n'a pas bien marché dans le sprint, les erreurs commises et de prendre des décisions pour s'améliorer. Il est tout à fait possible d'apporter des aménagements à la méthode Scrum dans le but de s'améliorer.

1.7. Compléments/approfondissements (pour SCRUM)

Lancement du projet :
Scrum présuppose que le backlog de produit est déjà défini au début du projet. Une approche possible pour constituer ce backlog est de réaliser une **phase de lancement**. Cette phase de lancement s'articule autour de deux axes de réflexion :

- l'étude d'opportunité
- l'expression initiale des besoins.

Documentation de projet :
Produire de la documentation est souvent utile mais aussi souvent inutile. En plus, il faut la maintenir à jour, quelque chose qui est rarement fait sur place. Pour savoir s'il faut rédiger un document, on peut se poser une question très simple : **Est-ce que ce document va m'être vraiment utile et tout de suite ?**

Voici quelques exemples de documents utiles et dans quels cas :

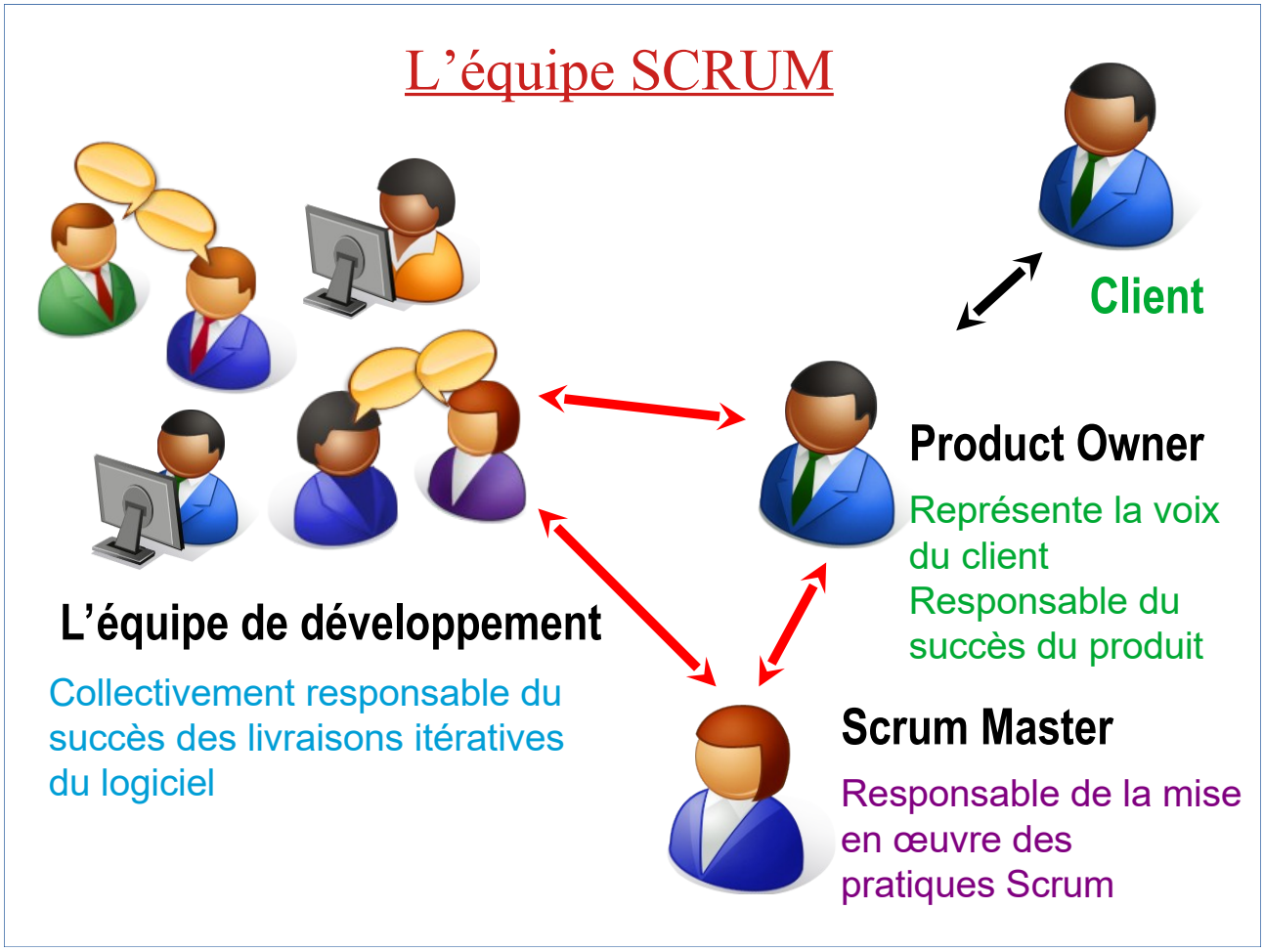
- diagrammes métiers (processus, objets, etc.), associé au backlog de produit : uniquement si la logique métier du client qui concerne l'application est vraiment complexe. Dans ce cas, l'équipe devrait produire ce document avec lui ;
- diagramme de séquence, associé à un item du backlog du produit : uniquement si la fonctionnalité aura une utilisation complexe, tant au niveau métier qu'applicatif ;
- diagrammes d'architecture du logiciel (classes, modules, composants, etc.), pour le projet : indispensable pour avoir toujours sous les yeux une vue de l'architecture et s'assurer ainsi qu'elle est de qualité ;
- les manuels utilisateur, à chaque sprint : les manuels sont produits à chaque sprint et pas en fin de projet. Utiliser des vidéos de démonstrations commentées est une solution efficace

Bref, un document ne doit être produit que si **son utilité est réelle et immédiate**.

IV - SCRUM : Rôles/intervenants et objectifs

1. Rôles / intervenants dans équipe SCRUM

1.1. Vue d'ensemble sur l'équipe SCRUM



1.2. Scrum-Master (organisateur , facilitateur)

Le "SCRUM-MASTER" n'est pas un Chef de projet

- il ne dirige pas
- il n'impose pas
- il ne contraint pas
- il n'a aucune autorité pour prendre des décisions à la place de l'équipe

Le "SCRUM-MASTER" est simplement ***le «Leader Spirituel» de l'équipe*** au sens "***mise en œuvre des bonnes pratiques de SCRUM***".

Le "**SCRUM-MASTER**" peut être vu comme un **animateur** (de réunion, de discussions , …) et qui :

- élimine des obstacles (administratifs , logistiques , compétences, ...)
- est au service de l'équipe et du "product-owner"
- s'arrure que l'équipe est entièrement fonctionnelle et productive
- facilite les coopérations/communications/...

Ce que le "SCRUM-MASTER" ne doit normalement pas faire :

- ~~Assigner les tâches aux développeurs~~
- ~~Prendre les décisions à la place de l'équipe~~
- ~~Agir comme intermédiaire entre l'équipe et le Product owner~~

1.3. Product-Owner (directeur de produit)

En tant que **représentant du client** , le "**product-owner**" (directeur de produit)

- **crée et partage la vision du produit**
- est responsable du retour sur investissement (au sens fonctionnel)
- **définit les caractéristiques du produit**

Pour cela , il

- **discute avec les différents intervenants**
- **fixe les priorités**
- accepte ou pas la livraison lors de la revue de sprint
- gère le plan de Release (Responsable du Product Backlog)

En régle générale, au niveau d'un logiciel :

33% des fonctionnalités sont fréquemment utilisées

66% des fonctions sont jamais ou rarement utilisées

Le "product-owner" est donc responsable d'optimiser la valeur produit (utile pour le client) sachant que souvent "***80% de la valeur est produite avec les premiers 50% de l'effort total*** " .

1.4. Equipe de développement

- Idéalement, entre 5 et 9 personnes colocalisées
- **Pluridisciplinaire** (Développeurs, Testeurs, Architectes,...)
- **Dédiée au projet**
- **Auto-organisée, autogérée** en respect des engagements pris
- **Prend collectivement et quotidiennement les décisions**
- Estime la taille des items du backlog (tâches/ parties de fonctionnalités)
- **S'engage sur la livraison en fin de Sprint d'un incrément logiciel « terminé »** (prêt à être présenté) et potentiellement déployable/livrable .

2. Principaux objectifs de SCRUM

- **Livrer petit à petit des incréments d'un logiciel utile au client**
- **Concevoir, coder , tester et livrer le plus rapidement possibles les fonctionnalités prioritaires**
- **Innover si possible (un produit idéal se démarque de la concurrence)**
- **Produire un logiciel de qualité (ergonomique , idéalement sans bug, performant, …) et un minimum évolutif/maintenable .**

V - Produit , UserStories et acceptabilité

1. Vision produit / SCRUM

1.1. Vision classique d'un bon produit

- Une vision synthétique et partagée
- Une vision construite ensemble
- L'équipe s'aligne (se met d'accord) sur les mêmes objectifs et engagements :
  - pour quel public
  - pour quels besoins cibles
  - les bénéfices majeurs, les différentiateurs vis à vis de la concurrence

1.2. Les "personas" (utilsateurs types) à identifier

Au niveau des méthodes agiles on appelle souvent "Personas" des profils (très détaillé/imagés) d'utilisateurs types succeptibles d'utiliser le logiciel (qui doit être pensé pour eux).
La description d'un "persona" permet généralement de mieux comprendre et cerner :

- ce que veulent les utilisateurs
- leurs comportements
- leurs besoins et attentes et éventuelles frustrations (bugs/pannes , lenteurs, …)

Ceci est généralement utile pour faire émerger :

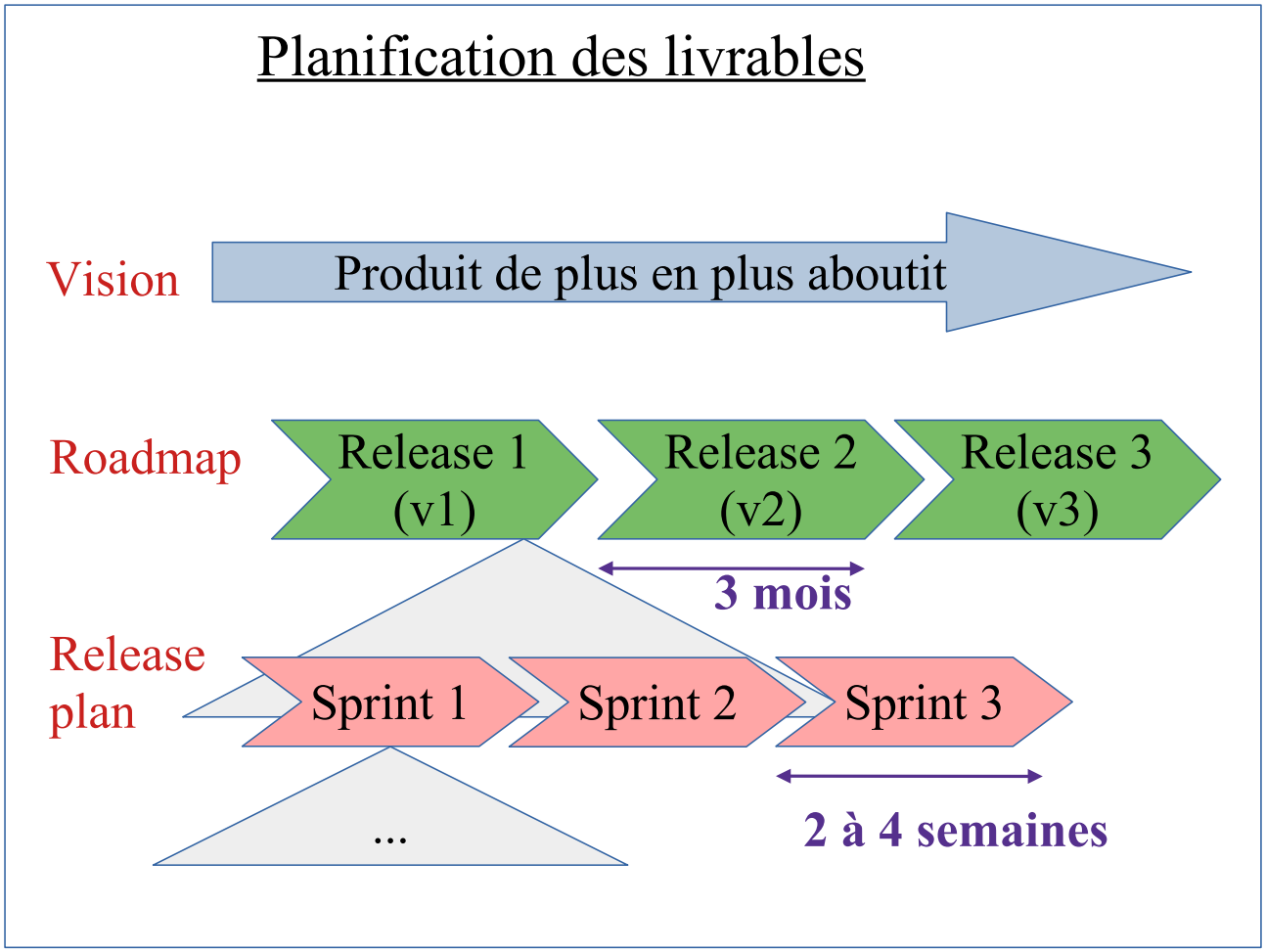
- Une vision commune des principaux utilisateurs d'un service ou d'un produit.
- Les fréquences d'utilisation du produit ou de la fonction
- Des priorités
- Des décisions de conception d'interfaces

Exemple de persona :

Nom	**René**
Description	67ans , marié , vit à la campagne
Situation professionnelle	Retraité
Etape actuelle de sa vie	Début de retraite (encore dynamique)
Ses attentes et besoins	Faire du shopping sans se fatiguer (de chez soi)
Ses attitudes comportementales	Pas très patient , envoie l'ordinateur par la fenêtre et donne la souris à manger pour son chat .
Ses aptitudes (utilisation des technologies)	Pas très à l'aise avec les nouvelles technologies
Autres	Très intéressé par les bonnes affaires (prix bas, promos, ...)

Nom	**Lola**
Description	19ans , célibataire sans enfant , sportive , aime danser
Situation professionnelle	Étudiante à l'université
Etape actuelle de sa vie	Recherche le prince charmant
Ses attentes et besoins	Recherche des articles atypiques / originaux
Ses attitudes comportementales	Intelligente et intuitive
Ses aptitudes (utilisation des technologies)	Très à l'aise sur son smartphone , utilise très peu l'ordinateur
Autres	Emploi du temps bien rempli , dispose de peu de temps

2. Planifications des livrables



NB :

- Les durées (3 mois pour une release et 2 à 4 semaines pour un sprint) sont ici données à titre indicatif seulement et peuvent être ajustées.
- De la même manière , les versions peuvent être gérés de façons très variables (une nouvelle version majeure tous les 6mois ou 1an ou 4ans , ...)

A la fin de chaque release :

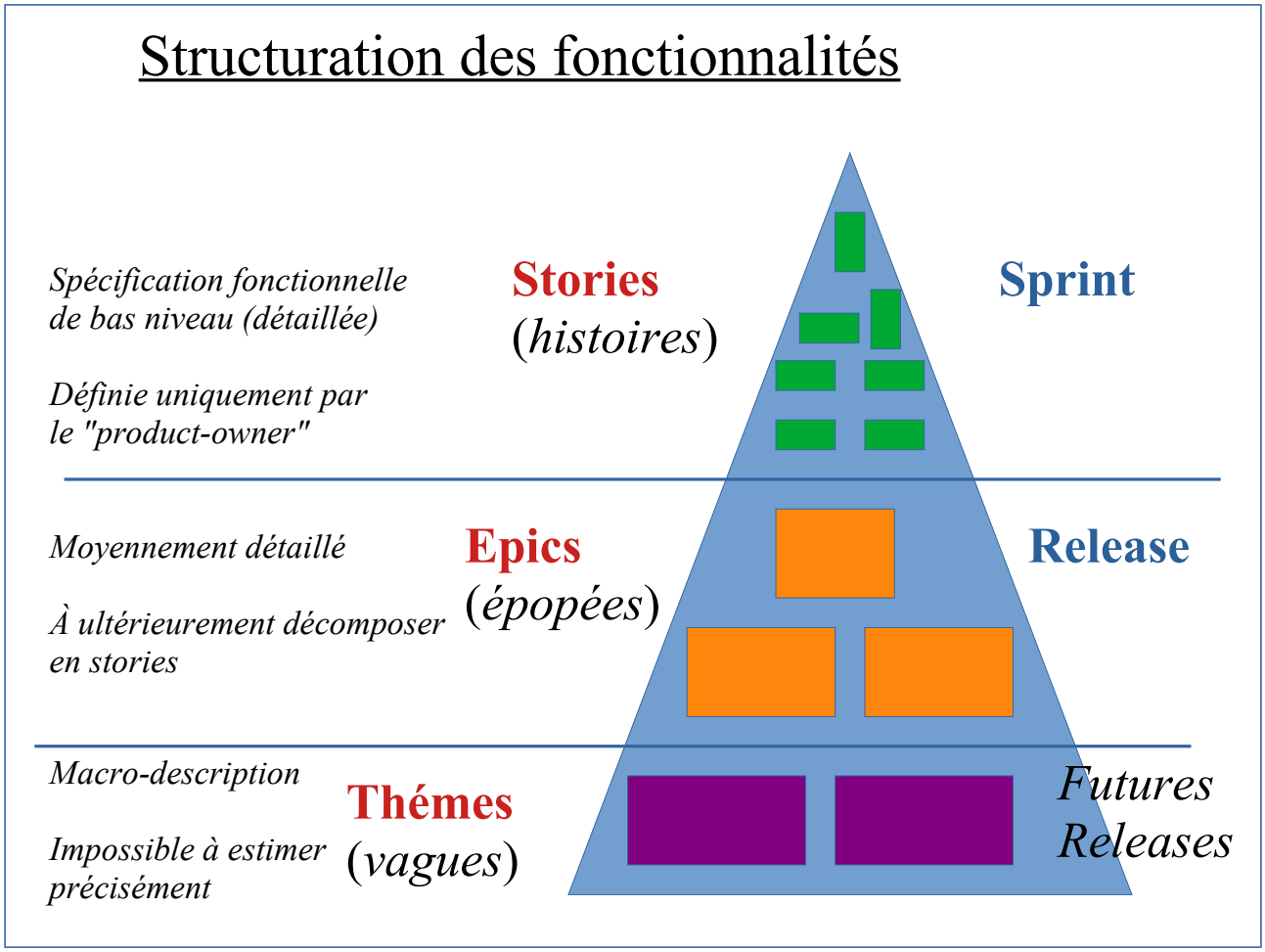
- *livrable important* : ***logiciel à peu près finalisé*** (ce qui à été commencé est mené à terme)

(ex : v1.0 , v1.1 , v1.2 , v2.0 , …)

A la fin de chaque sprint :

- ***livrable intermédiaire de type pré-version*** : avec une partie de code un peu testé et un peu utilisable et des parties encore manqantes (ex : v1.1-alpha , v1.1-beta , v1-1-rc)

3. UserStories et Epics



4. User-Story

Une "**user-story**" correspond à peu près à une **description fonctionnelle simple et compréhensible du besoin vu par l'utilisateur.**

4.1. Structure/Composantes d'une user-story

Une **user-story** est généralement constitué de:

- Un **titre** et une **description** (*en tant que … je souhaite …. afin de ...*)
- Des **critères d'acceptation** *(Given ... When ... Then ...)* et règles de gestion
- Une **valeur métier** (S,M,L,XL ou en points 5 , 10, 15 , 20 , ...)
- Une **estimation** (nb de points selon durée/complexité du développement prévu)

Rédaction conseillée d'une user-story :

1. description **c**omptacte du besoin (idéalement sur une carte de 8x13cm)
2. **c**onversation entre membres de l'équipe pour que ce soit bien compris de tous (avec reformulation éventuelle)
3. **c**onfirmation : redaction de scénario de test d'acceptation

Critères **INVEST** pour user-story :

Independent	Les user-stories sont idéalement indépendantes les unes des autres
Negociable	Négociée , discutée entre les membres d'une équipe (avec conversation/discussion)
Valuable	A une valeur métier/fonctionnelle , rend un service à l'utilisateur
Estimable	durée/complexité estimée en nombre de points (ex : suite fibonacci)
small Size	Suffisament petite/décomposée pour être mené à bien au sein d'un seul sprint à priori (sauf si mauvaise surprise)
Testable	Accompagnée de critères d'acceptabilité pour être ultérieurement validée

Pour obtenir une description précise et non ambigüe , on pourra penser à la formulation des tests.
Exemples :

~~avoir une interface ergonomique~~ j'accède à l'info en deux clicks au maximum

~~affichage rapide~~ affichage en moins d'une seconde avec 10000 utilisateurs simultanés

4.2. Formulation d'une user-story

En tant que <<*user*>> **je souhaite** <<*fonctionnalité*>> **afin de** <<*bénéfice*>>

En tant que <<*user*>>

je veux pouvoir <<*effectuer_telle_action*>>

afin de <<*arriver_à_cette_fin*>>

En tant que <<*role_utilisateur*>>
je désire <<*faire une action*>>
afin de <<*atteindre un objectif*>>

Exemples :

En tant que visiteur du site , **je désire** ouvir un compte **afin de** pourvoir effectuer des achats .

En tant que client, **je désire** annuler ma commande **afin de** ne pas à avoir à la payer.

4.3. Critère d'acceptation (user story)

Ensemble de **conditions métiers** que la user-story foit satisfaire pour être considérée comme valide.

Formalisme conseillé : Given … When … Then … (Si … Lorsque … Alors...)

GIVEN (étant donné) *<<un contexte>>*
WHEN (lorsque) *<<telle_action/évenement_déclenché_par_utilsateur>>*
THEN (alors) *<<on doit constater_tels_comportements/conséquenses>>*
*(avec combinaisons possibles avec **AND** (et) , ...)*

Exemple *:*

GIVEN utilisateur pas encore connecté AND être sur la page de connexion
WHEN identifiant_existant saisi par utilisateur
AND bon_mot_de_passe saisi
AND l'utilisateur clique sur le bouton <<login>>
THEN la page d'accueil s'affiche en moins de 5s
AND l'identifiant de l'utilisateur s'affiche dans le bandeau d'entête

4.4. Décomposition fonctionnelle mais pas en couche logicielle

NB : une user story est avant tout métier/fonctionnelle et pour qu'elle soit bien testable et critiquable il ne faut pas effectuer un découpage en couche logicielle (IHM, logique métier , accès_DB) mais on peut éventuellement la décomposer en sous fonctionnalités .

Par contre , lors d'un sprint , la prise en charge d'une user-story pourra être décomposée en tâches techniques (persistance, api_rest , frontend).

4.5. Itérations et incréments

- Livraison par fonctions métiers cohérentes
- **incrément = un morceaux de la fonctionnalité**
- A chaque itération, on ajoute un incrément de la fonctionnalité jusqu'à ce que celle-ci soit complète
- **idéalement, une fonctionnalité partielle doit tout de même être intelligible (suffisament esquissée) de manière à avoir un retour au plus tôt sur le produit et de façon à minimiser les risques**

 …..  (ok)

---

mais

 et  moins bien !!!

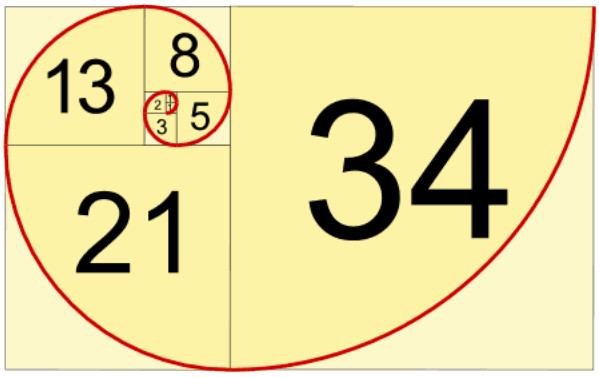
Quelques idées de décompositions/incréments pour user-story :

Selon une action utilisateur	Créer , Rechercher/Afficher , Mettre à jour , Supprimer
Selon les variantes (type d'utilisateur , de moyens de paiement , ...)	
Selon le niveau de finition (aide contextuelle , multi-langue, ...)	
...	

4.6. Estimation de la complexité d'une user-story

Unité possible (taille de tee-shirt) : **XS, S , M , L , XL , XXL**

Unité préférable : nombre de points en utilisant *éventuellement* la ***suite de Fibonacci*** .



Méthode possible pour affecter les points d'estimation : **planning poker** :

- on prend une user-story de complexité moyenne comme référence (sur laquelle quasiment toute l'équipe est d'accord)

- on estime la complexité des autres user-stories de manière relative (plus simple , plus complexe, nettement plus complexe, …)

- on essaie d'atteindre un consensus de groupe tout en essayant d'être rapide

4.7. Définition of Done (DoD)

De manière à pouvoir ultérieurement décider du fait qu'une user-story est terminée ou pas, on a intérêt à expliciter clairement (au niveau de l'équipe) une définition précise que ce que l'on peut considérer comme terminé :

- codé

- avec quelques commentaires importants

- intégré dans le coeur de l'application

- avec tests unitaires ok

- avec tests end-to-end ok

- revue de code effectuée

- documentation "utilisateur" et "technique" à jour

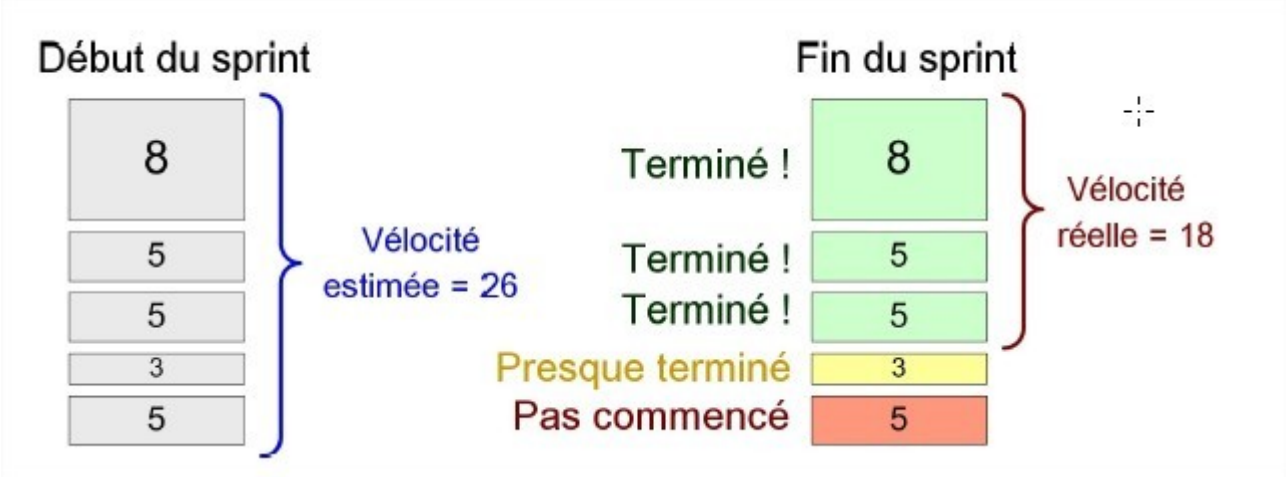
- ...

5. Vélocité et "burndown chart"

5.1. Vélocités estimées et effectives

- La **vélocité** est la **somme des points des User Stories qui ont été terminées depuis le début du sprint**.
- La **vélocité définit une capacité relativé de l'équipe** par rapport à une unité de point particulière (ex : points dans la suite fibonacci)
- ça permet d'avoir une **projection sur la réalisation de backlog** .
- Ce n'est en aucun cas une comparaison entre équipes ou autres puisque l'unité et les estimations peuvent variées d'une équipe à l'autre.

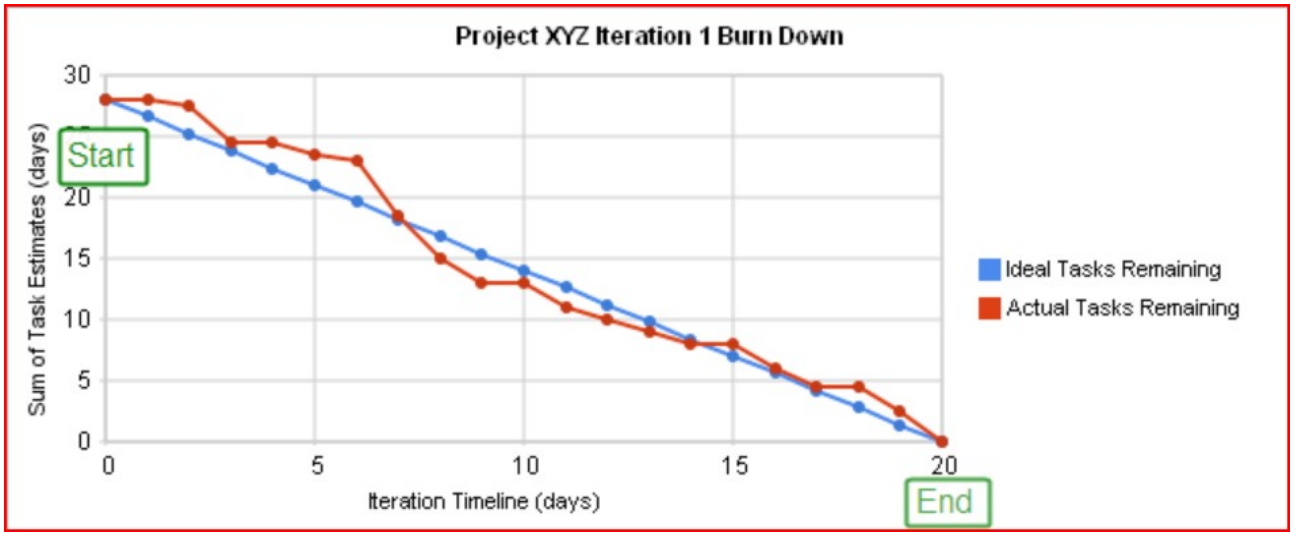
Exemple :



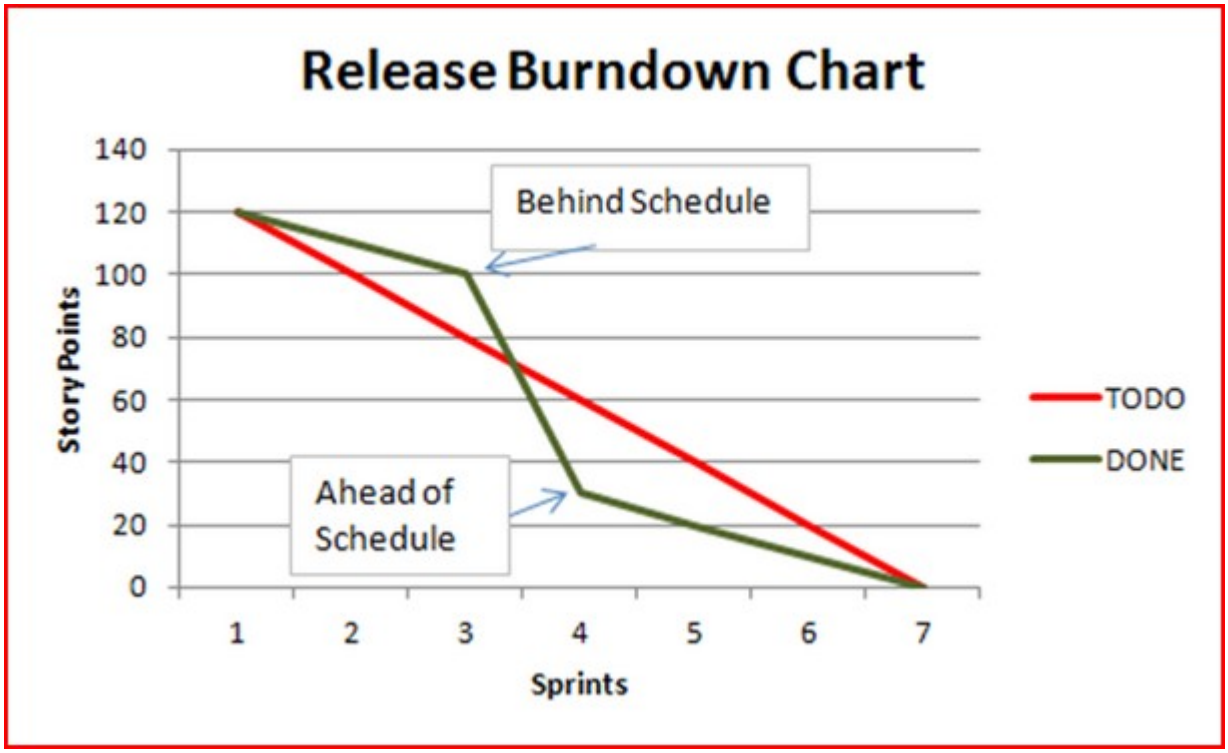
5.2. Burndown chart

Un "burndown chart" est un diagramme qui montre ce qu'il reste à faire au fur et à mesure du temps qui passe.

Exemple de "burndown chart" pour un sprint de 20 jours :



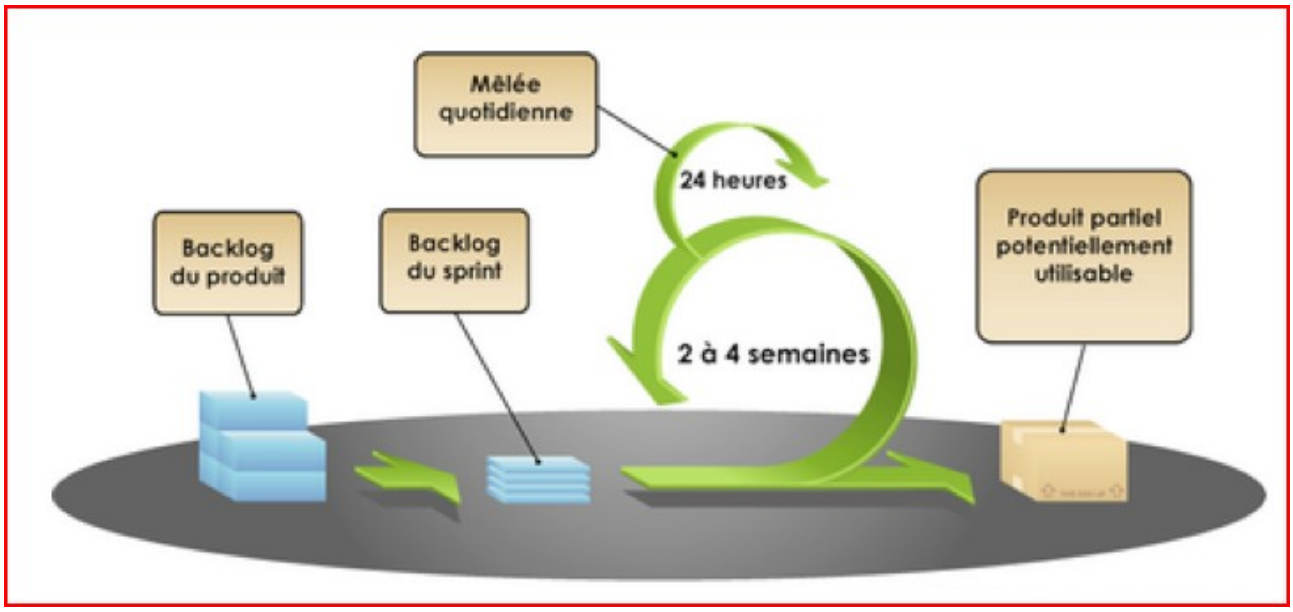
Exemple de "burndown chart" pour une release :



On peut également envisager un "burndown chart" de niveau produit (sans grande précision).

VI - Evénements et cérémonies SCRUM

1. Evénéments



1.1. sprint-planning

Objectifs

- Donner un but au sprint
- Affiner les Stories et ajouter des critères d'acceptation
- Estimer ou ré-estimer les Stories
- **Découper les Stories en tâches**
- Validation/Négociation selon les capacités des membres de l'équipe
- Questions/Réponses

Points d'entrée de la planification (préparés par le "product owner" avec "concertation équipe") :

- Les besoins (user stories) sont ordonnés par priorité et par valeur métier
- L'estimation des besoins est basée sur leur complexité.

Une des décisions à prendre c'est de choisir les "user-stories" que l'on va prendre dans le backlog de produit et que l'on va entreprendre dans ce nouveau sprint.

Priorité classique :
1. ce qui est à la fois d'une grande valeur métier et qui est un peu risqué → *il vaut mieux ne pas reporter à plus tard les choses risquées car on risque de ne jamais y arriver et il faut du temps pour résourdre certains problèmes*
2. ce qui est d'une grande valeur métier et pas risqué
3. ce qui est pas risqué et de faible valeur métier → *souvent reporté aux prochains sprints*

Et on évite carrément de ~~faire les choses risquées de ne faibles valeurs "métier"~~

Le plus gros travail de la **planification de sprint** consiste à décomposer les "***user-storie***s" retenues en tâches .

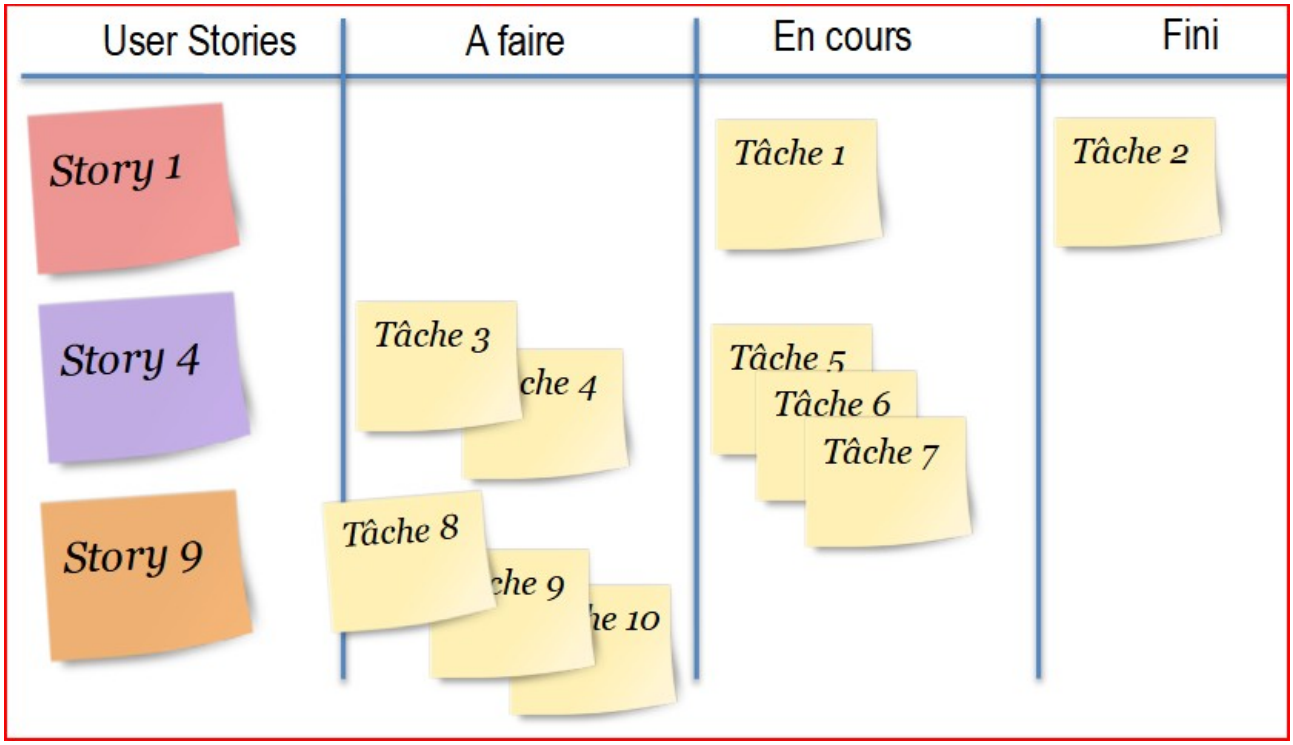
Exemple :
La user-story "**afficher la liste des produits (8pts)** " peut se décomposer en :

- gérer la persistance des produits (avec jeux de données et tests) : **8h**
- coder le service métier "Gestion_produits" avec tests unitaires : **8h**
- coder et tester l'api rest "product-api" : **8h**
- coder et tester l'interface graphique (frontEnd) permettant de rechecher et afficher les produits : **16h**
- rédiger documentation : **4h**

1.2. Régles à respecter pour un sprint scrum

Au début du sprint	Toutes les stories du sprint sont définies Les stories sont estimées Les stories sont priorisées L'équipe se met d'accord sur le périmètre du sprint
Pendant le sprint	Le sprint est verrouillé Le Product Owner ne peut ajouter ou retirer des stories qu'en cas d'allègement ou de surcharge
A la fin du sprint	L'équipe fait la démo au client des stories qu'elle a implémentées

1.3. Scrum board



En pratique :
- "**post it"** sur un tableau ou bien un mur
ou bien
- logiciel "**trello**" ou "**jira**" ou ...

Le scrum-board permet de
- Connaître le degré d'avancement (vision globale sur le sprint et une partie du projet)
- Donne de la visibilité partagée à tous les membre de l'équipe

Le scrum-board favorise :
- La communication
- L'appropriation
- La responsabilisation
- La transparence

1.4. Réunion/mêlée quotidienne "daily scrum" :

- **Quinze minutes au plus , toujours à la même heure**
- identification plutôt que résolution de problème lors de cette réunion
- Toute l'équipe (scrum-master , product-owner, développeurs) doit s'exprimer
- **Trois questions posées aux développeurs** :
  - Sur quoi as-tu travaillé hier ?
  - Sur quoi vas-tu travailler aujourd'hui ?
  - Y a-t-il quelque chose qui bloque ton travail ?
- Le ScrumMaster joue le rôle de facilitateur
- Objectifs de cette réunion :
  - améliorer l'esprit d'équipe
  - obtenir une visibilité sur l'avancement du projet et donner le rythme
  - s'engager à au moins essayer ...

1.5. Spring review (la démo)

- Démonstration des nouvelles fonctionnalités
- Toute l'équipe est présente
- Le Product Owner et le client donnent leurs avis
- Acceptation ou rejet du résultat (story par story) par le PO

1.6. Rétrospective de Sprint

Principal Objectif: **Rechercher des axes d'amélioration**

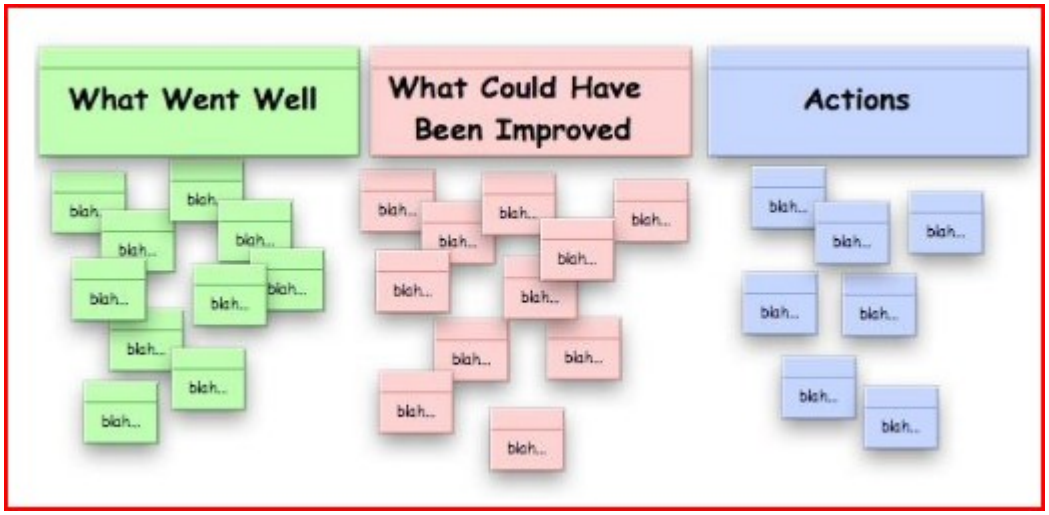
Cette réunion qui a lieu à la fin de chaque sprint

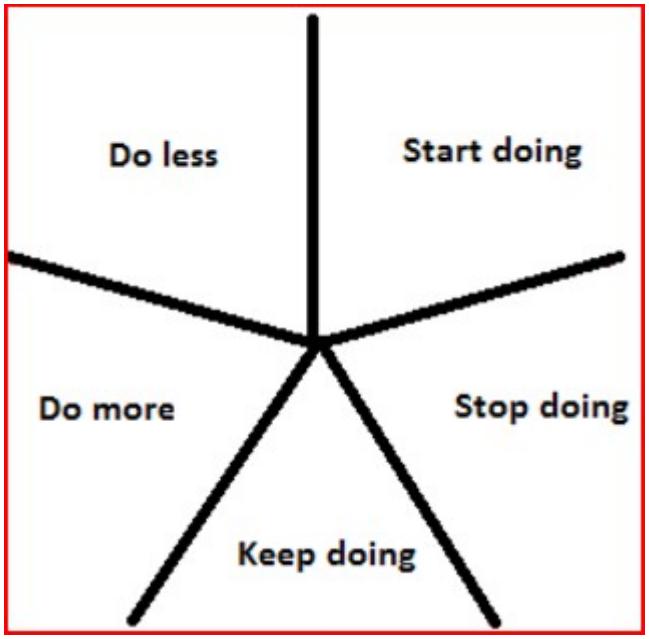
C'est le bilan du sprint qui vient de s'achever

Le Burndown Chart est clairement affiché.

On dresse un résumé de la façon dont s'est déroulé le sprint, avec ses points positifs, ses points négatifs (se poser idéalement plusieurs fois la question "pourquoi"), et décisions et les événements marquants.

L'équipe essaie de trouver des solutions vis à vis des problèmes rencontrés. Sachant qu'en général l'expérience acquise est déjà un axe d'amélioration en soit .





1.7. Cas particulier du "Sprint 0"

Ce n'est pas un Sprint ordinaire (pas les cérémonies habituelles , pas de livrable à présenter).
Période de préparation avant le lancement du premier Sprint

- Création de l'équipe
- Ecriture du Product Backlog
- Priorisation du Product Backlog
- Estimation et planification des Sprints et des Releases
- Préparation des environnements de développement
- Formation si nécessaire

ANNEXES

VII - Essentiel XP (présentation)

1. XP (Extreme Programming)

La méthode agile "XP (eXtreme Programming)" est née officiellement en octobre 1999 avec le livre *Extreme Programming Explained* de Kent Beck.

1.1. Principales caractéristiques de XP

(selon wikipédia)

Dans le livre *Extreme Programming Explained*, la méthode est définie comme :

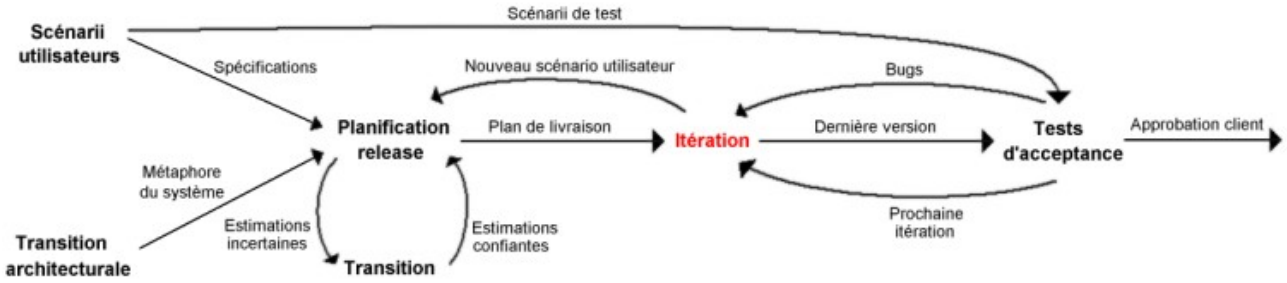
- une tentative de réconcilier l´humain avec la productivité ;
- un style de développement ;
- une discipline de développement d´applications informatiques.

Son but principal est de réduire les coûts du changement. Dans les méthodes traditionnelles, les besoins sont définis et souvent fixés au départ du projet informatique ce qui accroît les coûts ultérieurs de modifications. **XP s´attache à rendre le projet plus flexible et ouvert au changement en introduisant des valeurs de base, des principes et des pratiques**.

Les principes de cette méthode ne sont pas nouveaux : ils existent dans l'industrie du logiciel depuis des dizaines d'années et dans les méthodes de management depuis encore plus longtemps. L'originalité de la méthode est de les pousser à l'extrême :

- puisque la revue de code est une bonne pratique, elle sera faite en permanence (par un binôme) ;
- puisque les tests sont utiles, ils seront faits systématiquement avant chaque implantation ;
- puisque la conception est importante, elle sera faite tout au long du projet (*refactoring*) ;
- puisque la simplicité permet d'avancer plus vite, nous choisirons toujours la solution la plus simple ;
- puisque la compréhension est importante, nous définirons et ferons évoluer ensemble des métaphores ;
- puisque l'intégration des modifications est cruciale, nous l'effectuerons plusieurs fois par jour ;
- puisque les besoins évoluent vite, nous ferons des cycles de développement très rapides pour nous adapter au changement.

1.2. Cycle de développement XP



L'Extreme Programming repose sur des cycles rapides de développement (des itérations de quelques semaines) dont les étapes sont les suivantes :

- une phase d'exploration détermine les scénarios clients ("user stories" / mini "use case") qui seront à prendre en charge pendant cette itération ;
- l'équipe transforme les scénarios en tâches à réaliser et en tests fonctionnels ;
- chaque développeur s'attribue des tâches et les réalise avec un binôme ;
- lorsque tous les tests fonctionnels passent, le produit est livré.

Le cycle se répète tant que le client peut fournir des scénarios à implémenter. Généralement le cycle de la première livraison se caractérise par sa durée et le volume important de fonctionnalités embarquées. Après la première mise en production, les itérations peuvent devenir plus courtes (une semaine par exemple).

1.3. Valeurs de XP

L'eXtreme Programming repose sur cinq valeurs fondamentales :

- **La communication : c**'est le moyen fondamental pour éviter les problèmes.
- **La simplicité** : la façon la plus simple d'arriver au résultat est la meilleure. Anticiper les extensions futures est une perte de temps.
- **Le feed-back :** le retour d'information est primordial pour le programmeur et le client. Les tests unitaires indiquent si le code fonctionne. Les tests fonctionnels donnent l'avancement du projet. Les livraisons fréquentes permettent de tester les fonctionnalités rapidement.
- **Le courage :** certains changements demandent beaucoup de courage. Il faut parfois changer l'architecture d'un projet, jeter du code pour en produire un meilleur ou essayer une nouvelle technique.
- **Le respect**

1.4. Pratiques (extrêmes?) de XP

- ***Client sur site* :** un représentant du client doit, si possible, être présent pendant toute la durée du projet. Il doit avoir les connaissances de l'utilisateur final et avoir une vision globale du résultat à obtenir. Il réalise son travail habituel tout en étant disponible pour répondre aux questions de l'équipe.
- ***Jeu du Planning*** ou Planning poker : le client crée des scénarios pour les fonctionnalités qu'il souhaite obtenir. L'équipe évalue le temps nécessaire pour les implémenter. Le client sélectionne ensuite les scénarios en fonction des priorités et du temps disponible. On joue avec les plannings (réaffectation glissante des aspects secondaires et introduction surprise d'un nouvel élément fonctionnel dans le cahier des charges).
- **Intégration continue :** lorsqu'une tâche est terminée, les modifications sont immédiatement intégrées dans le produit complet. On évite ainsi la surcharge de travail liée à l'intégration de tous les éléments avant la livraison. Les tests facilitent grandement cette intégration : quand tous les tests passent, l'intégration est terminée.
- **Petites livraisons** : les livraisons doivent être les plus fréquentes possible. L'intégration continue et les tests réduisent considérablement le coût de livraison.
- **Rythme soutenable** :l'équipe ne fait pas d'heures supplémentaires. Si le cas se présente, il faut revoir le planning. Un développeur fatigué travaille mal.
- **Tests de recette (ou tests fonctionnels)** :À partir des scénarios définis par le client, l'équipe crée des procédures de test qui permettent de vérifier l'avancement du développement. Lorsque tous les tests fonctionnels passent, l'itération est terminée. Ces tests sont souvent automatisés mais ce n'est pas toujours possible.
- **Tests unitaires** : avant d'implémenter une fonctionnalité, le développeur écrit un test qui

vérifiera que son programme se comporte comme prévu. Ce test sera conservé jusqu'à la fin du projet, tant que la fonctionnalité est requise. À chaque modification du code, on lance tous les tests écrits par tous les développeurs, et on sait immédiatement si quelque chose ne fonctionne plus.

- **Conception simple** :plus l'application est simple, plus il sera facile de la faire évoluer lors des prochaines itérations.
- **Utilisation de métaphores :** on utilise des métaphores et des analogies pour décrire le système et son fonctionnement. Le fonctionnel et le technique se comprennent beaucoup mieux lorsqu'ils sont d'accord sur les termes qu'ils emploient.
- ***Refactoring*** **(ou remaniement du code) :** amélioration régulière de la qualité du code sans en modifier le comportement. On retravaille le code pour repartir sur de meilleures bases tout en gardant les mêmes fonctionnalités.
- **Appropriation collective du code :**l'équipe est collectivement responsable de l'application. Chaque développeur peut faire des modifications dans toutes les portions du code, même celles qu'il n'a pas écrites. Les tests diront si quelque chose ne fonctionne plus.
- **Convention de nommage :**puisque tous les développeurs interviennent sur tout le code, il est indispensable d'établir et de respecter des normes de nommage pour les variables, méthodes, objets, classes, fichiers, etc.
- **Programmation en binôme :** la programmation se fait par deux. Le premier appelé *pilote* tient le clavier . C'est lui qui va travailler sur la portion de code à écrire. Le second appelé *partner* (ou *co-pilote*) est là pour l'aider en gardant un œil critique et en suggérant de nouvelles possibilités ou en décelant d'éventuels problèmes (correction des erreurs , avis différent , aide , ...). Les développeurs changent fréquemment de partenaire ce qui permet d'améliorer la connaissance collective de l'application et d'améliorer la communication au sein de l'équipe.

1.5. Autres pratiques extrèmes et variantes

Sur des petits projets, on travaille généralement en équipe réduite. Il faut alors savoir un peu tout faire. On se forge ainsi une bonne expérience où les aspects pragmatiques l'emportent sur les grands discours théoriques. L'apprenti dépasse le maître et devient virtuose . Commencent alors les pratiques extrêmes :

- on jongle avec les générateurs de code , le copier–coller et l'inspiration du moment .
- ...

Bien qu'assez extrêmes et pas toujours applicables, ces pratiques permettent quelquefois :

- de pouvoir aller très vite.
- d'être plus réactif
- ...

citation (de S.G. ou ...): le tact dans l'audace c'est de savoir jusqu'où on peut aller trop loin .

Critiques et variantes/adaptations :

Quand on connaît le coût (assez élevé) d'une journée de développement (salaire du développeur + charges sociales , ….) , on peut se demander si le principe du travail en binôme est réellement applicable.
Le principe du travail en binôme est généralement judicieux que si l'est utilisé au bon moment (et pas systématiquement / tout le temps).
Lorsqu'il y a du "turn over" dans une équipe , le fait de travailler à deux permet de bien intégrer un nouveau développeur au sein de l'équipe existante :

- les premiers jours , le nouvel arrivant observe (en tant que co-pilote) les manières de développer au niveau du projet (environnement de dev , convention de nommage , …)
- et ensuite , c'est "tiens , prends le volant ". Le nouvel arrivant code de son mieux et le développeur rodé au projet vérifie si c'est bien fait et préconise des ajustements aussitôt.

On peut éventuellement s'autoriser des variantes par rapport à ce qui a été rédigé/formalisé au sein de la méthode XP. Le principe "0" serait : ne pas appliquer systématiquement XP tel quel mais seulement ce qui semble utile dans la méthode XP.

1.6. Zoom sur la planification adaptative

A partir d'un **découpage très fin** en "***User Stories***" (sorte de "mini *Use Cases*" auxquels on attache des priorités , des risques et des estimations de temps de développement) , on établit un planning temporaire avec des dates fixes (mais des contenus/livrables qui pourront partiellement changer).

===> Le planning initial est régulièrement remanié en fonction des :

- imprévus (Pb technique, ...) coté développement
- changements des besoins (nouvelles priorités) coté client

planification prévisionnelle initiale:

Livrables prévus sous les 20 premiers jours	UC1 , UC6 , UC4
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	UC3 , UC7 , UC5
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	UC2 , UC8

planification revue au bout de 20 jours ouvrés:

*[**retard** sur UC4 (2 jours) + **UC9** = **nouveau besoin prioritaire du client (5j/h)**]*

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	**fin_UC4** , **UC9** ,UC3 , UC7
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	**UC5** , UC2 , UC8

<u>planification revue au bout de 40 jours ouvrés</u>:

*[**retard** sur UC7 (2 jours) + **UC10** = **nouveau besoin prioritaire du client (6j/h)**]*

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables réalisés et livrés sous les 20 jours suivants (globalement 40 jours après le début)	**UC4** , **UC9** ,UC3
Livrables réalisés et livrés sous les 20 jours suivants (globalement 60 jours après le début)	**UC10, fin_UC7 , UC5** , UC2
Eléments jamais livrés ou bien livrés plus tard si avenant et budget .	UC8 ***(le moins prioritaire !!!)***

VIII - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

https://en.wikipedia.org/wiki/Scrum_(software_development)	Scrum sur wikipedia
https://www.uv.es/nemiche/cursos/Scrum.pdf	

2. TP

2.1. Expérimenter "scrum-board" avec le logiciel trello

- Pour le fun "quelques blagues ou anecdotes" à récupérer dans "à faire ", à glisser en "en cours" et à dire puis à glisser dans "fini" .
- Expérimentations libres avec le logiciel trello

2.2. Etude de cas "PizzaCompany"

A faire en groupes :

etape1 :

- Lister les utilisateurs/clients/parties prenantes
- Réaliser au moins une fiche persona
- Définir la vision produit
- Définir le MVP (Minimum Viable Product)
- Lister les EPIC
- Lister les User Stories
- Prioriser les User Story

étape 2 (Sprint 1):

- Réaliser le Sprint planning du Sprint 1 :
- Définir le Sprint Backlog
- Découper en tâches
- Estimer les User Story
- Calculer la vélocité estimée
- Faire le Scrum Board
- Définir votre DoD

NB : on pourra en partie s'appuyer sur trello ou jira

2.3. Quelques idées pour les points dans jira

NB: Dans jira et le template scrum , seuls les points associés au "story" (ou tâche/bug/...) sont comptabilisés dans le burndown chart …

Unité conseillée nb_points selon suite de fibonacci (0,1,1,2,3,5,8,13,21,34,55,89,144, ...) (nb: 1+1 =2 , 1+2=3 , 2+3=5 , 3+5=8 , 5+8 = 13 , etc)

Dans jira, les points affectés au "sub_task" ne sont pas pris en compte dans le burndown chart , ils ne sont qu'indicatifs et peuvent éventuellement avoir une autre unité (ex: nombre d'heures de travail)

Néanmoins, pour se fixer au moins un petit ordre de grandeur on peut se dire que très approximativement:

1pt = xxs = très très petite tâche (ex: 2h de travail environ , 1/4 j)
2pts = xs =très petite tâche (ex: une demie journée)
3pts = s = petite à moyenne tâche (3/4 j)
5pts = m = tâche ordinaire/moyenne (1j+1/4j environ)
8pts = l = tâche assez importante (environ 2j)
13pts = xl = grosse tache (3j environ)
21pts= xxl = très grosse tâche (1 semaine et ...)
34pts = 3xl = très très grosse tâche (presque 2 semaine)
55pts = 4xl = énorme tâche (entre 3s et 1mois)

NB: une "user story" , sera souvent décomposées en ces sous tâches : persistance , services métiers internes, API_REST , frondEnd et donc
à léchelle d'une user_story:
3pts = xs
5pts = small
8pts= medium (m-)
13pts = ordinaire (m+)
21pts = large
34pts = xl

2.4. Etude de cas "bibliothéque" avec jira

1) nouveau projet (modèle = scrum , type = géré par l'équipe)
2) sélectionner planification/backlog , et faire apparaitre le panneau "epic" via menu déroulant
3) créer au moins l'epic "essentiel"
4) créer un ticket (par défaut de type "story" , quelquefois de type "tâche" (technique, administrative) ou bien "bug" à corriger)
donner un nom et une description à la story (en tant que ... je souhaite .... afin de ...) et associer la nouvelle story à un epic.
donner une valeur à "story point estimate" (ex: 3,5,8-13 ,21,34 ...)
---
5) créer un nouveau sprint (de durée = 15jours par défaut) (noms automatiques des sprints "sprint_1 , sprint 2" , ...)
6) glisser/poser les tickets de Backlog vers "Sprint_n"
7) un minimum de faisabilité :
NB: ordre de grandeur (environ 4pts par jour par développeur --> environ 40pts par développeur par sprint ,
80 pour 2dev , 120 pour 3dev , 160 pour 4dev, ...)

8) décomposer éventuellement/idéalement un "user_story" en sub_task .
il n'est pas essentiel d'affecter des points aux sub_tasks (car simple indication pas comptabilisée dans "burndown chart")
9) démarrer le sprint
---
10) sur partie "tableau" , déplacer les tickets de "à faire" , "en cours" , "fini" .
si sub_task , penser également à mettre à jour l'état (à faire, en_cours, terminé) sur l'ensemble du "story"
NB: on peut ajouter un indicateur sur un ticket (ex: nom , prénom ou initiales du développeur)