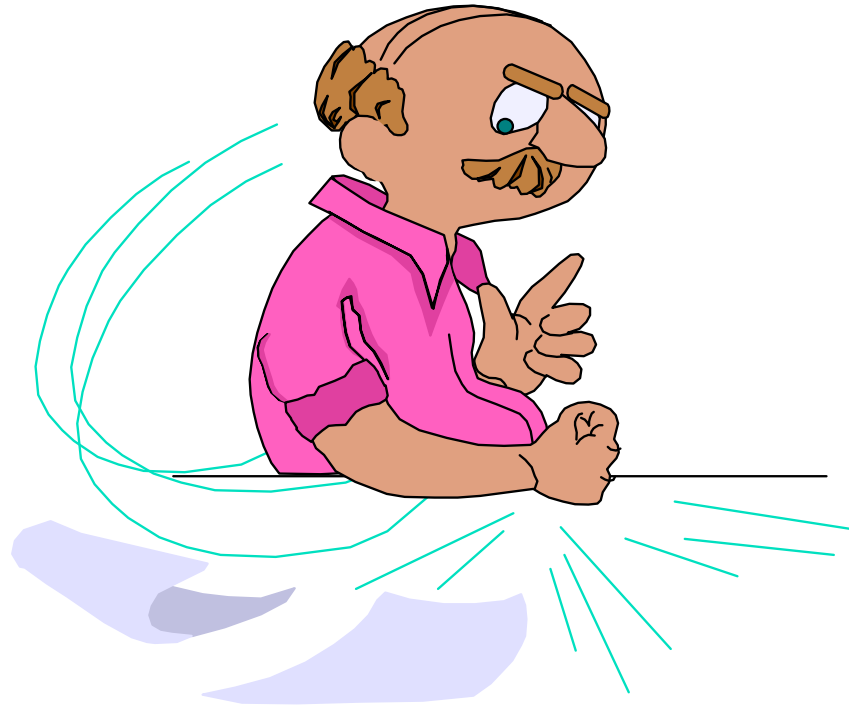


Chap 1: Introduction au Génie Logiciel



Dr. Bah Moussa Demba

Agenda

- Introduction
- SI et logiciel: Notions et connaissances de base
- Informatisation des SI:
 - Cycle de vie d'un logiciel
 - Processus de développement: Principaux Modèles
 - Méthodes et Techniques de développement
- La Conduite et la Gestion de Projet
- Assurance et Contrôle Qualité du logiciel
- Maintenance et Evolution

Les erreurs Logicielles

- Probablement le plus coûteux!
 - Estimation du cout économique des erreurs logicielles montre des chiffres astronomiques
 - US: \$59.5 **billion** in 2002
 - http://www.nist.gov/public_affairs/releases/n02-10.htm
 - ~10% des projets sont abandonnés

Erreurs logicielles

- **Aircrafts and software bugs**
The missing Malaysian Airlines, 2014
Boeing Max
- **Cars and software bugs**
car accidents
- **Le lancement raté du Galaxy Note 7**
rappel plus de 2,5 millions d'appareils
- **Hospitals and software bugs**
44,000-98,000 preventable deaths in the US each year



Erreurs logicielles



June 4, 1996 -- Ariane 5 Flight 501

It took the European Space AGENCY **10 years** and **\$7 billion** to produce Ariane 5, a giant rocket. Problem: conversion from 64bit into a 16bit. *This shutdown occurred 36.7 seconds after launch.*

Le 25 février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dhara (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits. Le temps était compté par l'horloge interne du système en dixième de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire : $1/10 = 0.1$ (dans le système décimal) = $0.0001100110011001100110011...$ (dans le système binaire). L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque 1/10 de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui a entraîné une accumulation des erreurs d'arrondi de 0.34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible.

January 8, 2020- Ukrainian plan

Iran revolution guard shutdown unintentionally a Ukrainian plan

La crise du logiciel (1/3)

- Les échecs de grands projets durant les années 60 et au début des années 70 ont mis en évidence les problèmes de la gestion de projet.
- *Ces projets n'ont pas échoué parce que les programmeurs étaient incompétents.* La faute incombait en fait aux techniques de gestion des projets mises en œuvre.

La crise du logiciel (2/3)

En 1979, le gouvernement américain estimait que la plupart des grands projets avaient *échoué* :

- Payés mais jamais livrés 47 %
- Livrés mais jamais utilisés 30 %
- Abandonnés ou refaits 20 %
- Utilisés après modifications 3 %
- Utilisés en l'état 2 %

La crise du logiciel (3/3)

Les symptômes les plus caractéristiques de cette crise sont décrits par Booch en 1983 :

- Les logiciels réalisés ne correspondent souvent pas aux besoins des utilisateurs,
- Les logiciels contiennent trop d'erreurs (la qualité du logiciel est insuffisante),
- Les coûts du développement sont rarement prévisibles,
- La maintenance des logiciels est une tâche complexe et coûteuse,
- Les délais de réalisation sont généralement dépassés,
- Les logiciels sont rarement portables.

Naissance du Génie Logiciel

- Il est né en 1968 à Garmisch (Allemagne) (« *1st conference on software engineering* » sous le parrainage de l'OTAN).
- Il a été défini de toutes pièces par un groupe de scientifiques pour répondre à la crise du logiciel avec quelques idées émergentes :
 - La production de logiciel doit être organisée,
 - Contrôle des coûts et de la qualité, etc ...

Definition d'un genie logiciel

Le genie logiciel (*software engineering*) représente l'application de principes d'ingénierie au domaine de la creation de logiciels. *Il consiste à identifier et à utiliser des méthodes, des pratiques et des outils permettant de maximiser les chances de réussite d'un projet logiciel.*

Le **gènie logiciel** vise à garantir que :

- ❶ la spécification répond aux besoins réels de ses clients ;
- ❷ le logiciel respecte sa spécification ;
- ❸ les coûts alloués pour sa réalisation sont respectés ;
- ❹ les délais de réalisation sont respectés.

Comment fabriquer un logiciel de qualité ?

En plus du respect (essentiel) de sa spécification, la **qualité** d'un logiciel dépend des 4 critères suivants :

Maintenabilité Peut-on faire évoluer le logiciel ?¹

Robustesse Le logiciel est-il sujet à des dysfonctionnements ?

Efficacité Le logiciel fait-il bon usage de ses ressources ?

Utilisabilité Est-il facile à utiliser ?

1. Un logiciel ne s'use pas. La correction d'une erreur n'est pas évolution mais un échec du concepteur.

Les spécificités du logiciel

Cependant, la construction d'un logiciel diffère de celle d'un pont :

- une modification infime peut avoir des conséquences critiques ;
- les progrès technologiques très rapides peuvent rendre un logiciel caduque ;
- il est difficile de raisonner sur des programmes ;
- les domaines des entrées des logiciels sont trop grands pour le test exhaustif ;
- les défaillances des programmes sont en général dues à des erreurs humaines ;
- on ne sait pas très bien réutiliser les programmes existants ;
- chaque logiciel a son organisation et sa logique propre ;
- ...

Les grands principes du génie logiciel

Un certain nombre de grands principes (de bon sens) se retrouvent dans toutes ces méthodes. En voici une liste proposée par Ghezzi :

- ❶ La rigueur
- ❷ La décomposition des problèmes en sous-problèmes indépendants
- ❸ La modularité
- ❹ L'abstraction
- ❺ L'anticipation des évolutions
- ❻ La généricité
- ❼ La construction incrémentale



C. Ghezzi.

Fundamentals of Software Engineering.

Prentice Hall, 2nd edition, 2002.

Principe #1 — la rigueur

- Les principales sources de défaillances d'un logiciel sont d'origine humaine.
- À tout moment, il faut se questionner sur la validité de son action.
- Des outils de vérification accompagnant le développement peuvent aider à réduire les erreurs. Cette famille d'outils s'appelle CASE (*Computer Aided Software Engineering*).

Exemple

typeurs, générateurs de code, assistants de preuves, générateurs de tests, outil d'intégration continue, fuzzer, ...

Principe #2 — la décomposition en sous-problèmes

- « *Separation of concerns* » en anglais.



- Il s'agit de :
 - **Décorrél**er les problèmes pour n'en traiter qu'un seul à la fois.
 - **Simplifier** les problèmes (temporairement) pour aborder leur complexité progressivement.

Principe #3 — la modularité

- Il s'agit de partitionner le logiciel en **modules** qui :
 - ont une **cohérence interne** (des invariants) ;
 - possèdent une **interface** ne divulguant sur le contenu du module que ce qui est strictement nécessaire aux modules clients.

Principe #4 — l'abstraction

- Il s'agit d'exhiber des **concepts généraux** regroupant un certain nombre de cas particuliers et de raisonner sur ces concepts généraux plutôt que sur chacun des cas particuliers.

Exemple (Support dans les langages de programmation)

les classes abstraites dans les langages à objets, le polymorphisme de Caml et le *generics* de Java, les fonctions d'ordre supérieur, les foncteurs de Caml et le *templates* de C++, ...

Principe #5 — l'anticipation des évolutions

- Un logiciel a un cycle de vie plus complexe que l'habituel cycle « commande → spécification → production → livraison ».
- La maintenance est la gestion des évolutions du logiciel.
- Il est primordial de prévoir les évolutions possibles d'un logiciel pour que la maintenance soit la plus efficace possible.
- Pour cela, il faut s'assurer que les modifications à effectuer soient le plus locales possibles.
- Ces modifications ne devraient pas être *intrusives* car les modifications du produit existant remettent en cause ses précédentes validations.
- Concevoir un système suffisamment riche pour que l'on puisse le modifier **incrémentalement** est l'idéal.

Principe #6 — la généricité

- Un logiciel **réutilisable** a beaucoup plus de valeur qu'un composant dédié.
- Un composant est **générique** lorsqu'il est adaptable.

Principe #7 — la construction incrémentale

- Un développement logiciel a plus de chances d'aboutir si il suit une cheminement **incrémental** (*baby-steps*).

Exemple

Laquelle de ses deux méthodes de programmation est la plus efficace ?

- Écrire l'ensemble du code source d'un programme et compiler.
- Écrire le code source d'une fonction ou module, le compiler, et passer à la suivante.

Pourquoi ?



Activités du développement logiciel



1re étape : Bien se comprendre

Difficultés

- **Multiplicité des acteurs**, des rôles
 - **Client, utilisateur** (ce n'est pas forcément le même)
 - **Informaticiens**
 - Chef de projet
 - Architecte
 - Développeur
 - Testeur
 - Maintenance, documentation, ...
 - Marketing, diffusion, formation
- **Langages, vocabulaires** différents

Activités du développement logiciel

Les étapes (1)

Liste (non exhaustive...)

- Définition des **besoins**
-> Dans le langage du client
- **Spécifications**
Traduction des besoins dans un langage *plus informatique*
-> Ce que **doit** faire le logiciel (et **non comment** il le fait)
- **Conception**
Traduction des spécifications en termes de concepts logiciels
- **Codage**
-> Traduction de la conception en code
- **Tests**
 - unitaires (Test de chaque module individuellement)
 - d'intégration (Test de la composition de plusieurs modules)

Activités du développement logiciel

Les étapes (2)

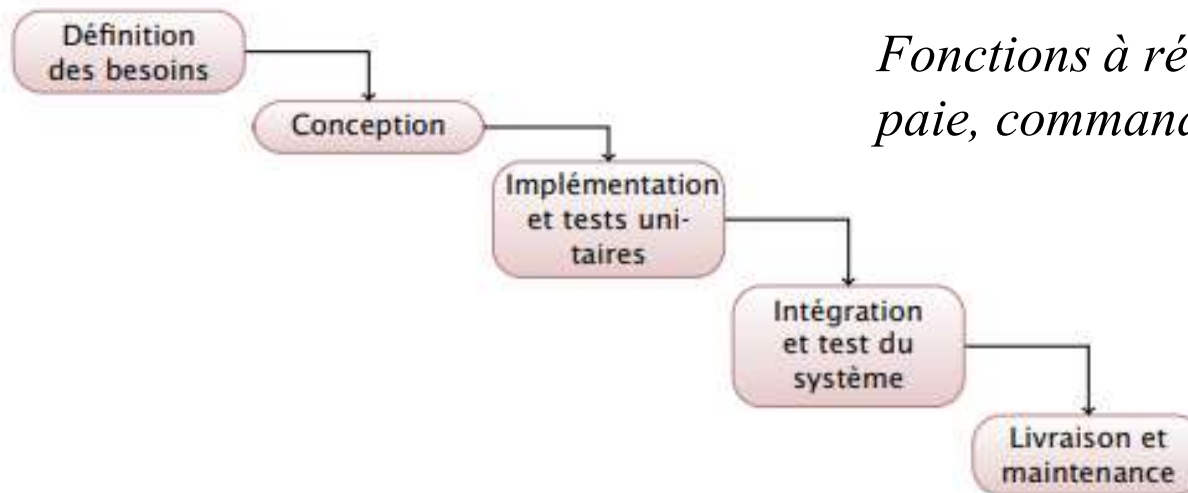
Liste (non exhaustive...)

- **Validation - Vérification** → assure que le logiciel respect la specification.
Avons-nous construit le **bon** logiciel ? -> **Validé** : *assurer que ce que veut l'utilisateur*
Avons-nous **bien** construit le logiciel ? -> **Vérifié** : *assurer qu'il fonctionne bien*
- **Livraison / Diffusion**
- **Support, formation**
- **Maintenance**
- **Evolution** (nouvelles versions, ...) → adapte le logiciel aux besoins futurs de ses clients.

Schéma général d'un processus de développement

Il existe différents **modèles** de processus qui organisent de façon différentes ces activités, entre eux : le modèle en *cascade*, le modèle de *développement évolutif* et le modèle de *développement par composants*.

Modèle en cascade



Fonctions à réaliser: Calcul de paie, commande d'avion,...

- Chaque phase doit se terminer pour commencer la suivante.
- Des **documents** sont produits pour concrétiser la réalisation de chaque phase.

Modèle en cascade

Avantages

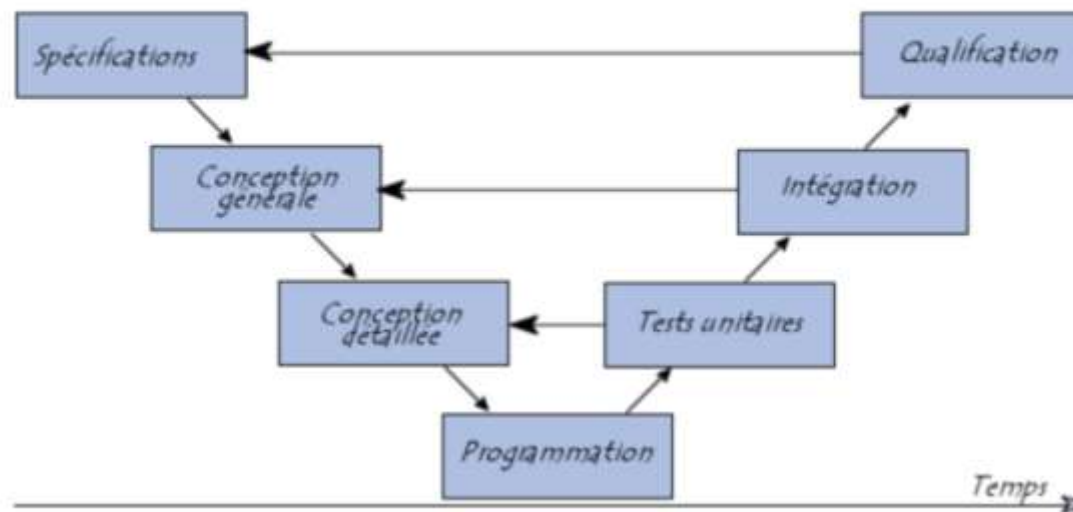
- Aisé à comprendre et à mettre en œuvre
- Forte structuration : définition puis réalisation

Inconvénients

- Modèle idéalisé, ne tient pas compte de la nature itérative d'un projet
- **Logiciel livré seulement à la fin du projet**
 - Attente du client
 - Conformité, validation tardive
 - Coût élevé en cas de non-conformité

Schéma général d'un processus de développement

Cycle en V



Tests unitaires : Test de procédures, de modules, de composants

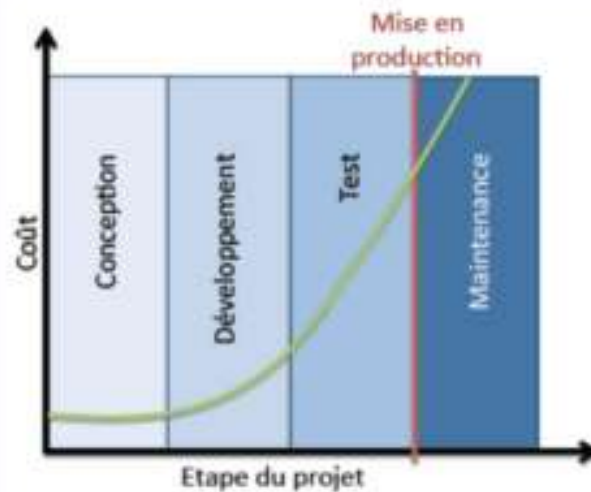
Tests d'intégration : Test de bon comportement lors de la composition de procédures et modules

Tests de conformité ou test système : Validation de l'adéquation aux spécifications

Coût de résolution de bugs

Tests et développement

- Tests unitaires
-> **50%** du coût total du projet
- Tests d'intégration (lors composition des modules, ou composants)
-> **10 fois** le coût d'un bug détecté lors des tests unitaires
- Tests système, validation (adéquation aux spécifications)
-> **100** fois le coût d'un bug détecté lors des tests unitaires



La qualité en génie logiciel?

La qualité d'un logiciel est son aptitude à satisfaire les besoins des utilisateurs
[Martin, 1987]

- Le logiciel est la clé de différenciation des produits industriels
- Mais on le maîtrise mal...
 - De nombreux projets informatiques n'ont jamais abouti
 - Ou ont été des catastrophes économiques
- ... contrairement au génie civil (ponts autoroutes, tunnels,...)
- Le zéro défaut n'existe pas en matière de logiciel

Personne ne sait aujourd'hui créer de logiciel sans défaut!

Fiabilité des logiciels

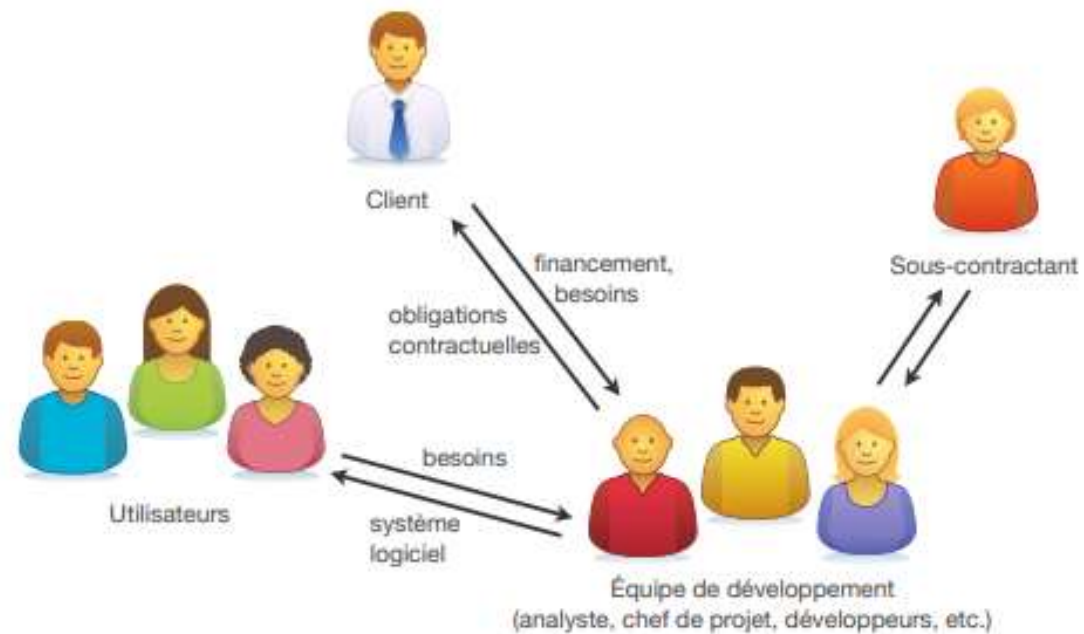
C'est la probabilité pour qu'une panne du logiciel ait lieu.

➔ Un système peu fiable entraîne des pertes d'infos, pertes de temps et d'argent pour les reconstruire

Indicateurs
de qualité

-
- [Fiabilité](#)
 - [Maintenabilité](#)
 - [Performance](#)
 - [Portabilité](#)
 - [Utilisabilité](#)
-

Les acteurs du génie logiciel



Principes de développement : DRY

DRY: Don't Repeat Yourself. *Ce principe vise à éviter la redondance au travers de l'ensemble de l'application.* Cette redondance est en effet l'un des principaux ennemis du développeur.

Elle a les conséquences néfastes suivantes :

- ❖ Augmentation du volume de code.
- ❖ Diminution de sa lisibilité.
- ❖ Risque d'apparition de bogues dûs à des modifications incomplètes.

Elle prend souvent la forme d'un ensemble de lignes de code dupliquées à plusieurs endroits pour répondre au même besoin.

Exemple:

Principes de développement : DRY

```
function A() {  
  // ...  
  // Code dupliqué  
  // ...  
}  
  
function B() {  
  // ...  
  // Code dupliqué  
  // ...  
}
```

La solution: factoriser les lignes auparavant dupliquées.

```
{  
function C() {  
  // Code auparavant dupliqué  
}  
function A() {  
  // ...  
  // Appel à C()  
  // ...  
}  
function B() {  
  // ...  
  // Appel à C()  
  // ...  
}
```

Attention:

Trop factoriser → creation des applications inutilement complexes

Principes de développement : KISS

KISS: Keep It Stupid Simple et qu'on peut traduire par "*Ne complique pas les choses ou garde ça super simple*" avec l'idée que « *même un idiot pourrait comprendre* ». Ce principe vise à privilégier autant que possible la simplicité lors de la construction d'une application.

Il consiste à affirmer qu'une application doit être créée *selon l'ordre de priorité* ci-dessous :

- 1. Make it work.***
- 2. Make it right.***
- 3. Make it fast.***

Principes de développement : YAGNI

YAGNI: You Ain't Gonna Need It signifie «*vous n'en aurez pas besoin* ». Il consiste à dire que les *programmeurs* ne devraient pas ajouter de fonctionnalité à un logiciel tant que celle-ci n'est pas absolument nécessaire.

Ron Jeffries recommande par ailleurs : « *mettez toujours en œuvre les choses quand vous en avez effectivement besoin, pas lorsque vous prévoyez simplement que vous en aurez besoin* ».

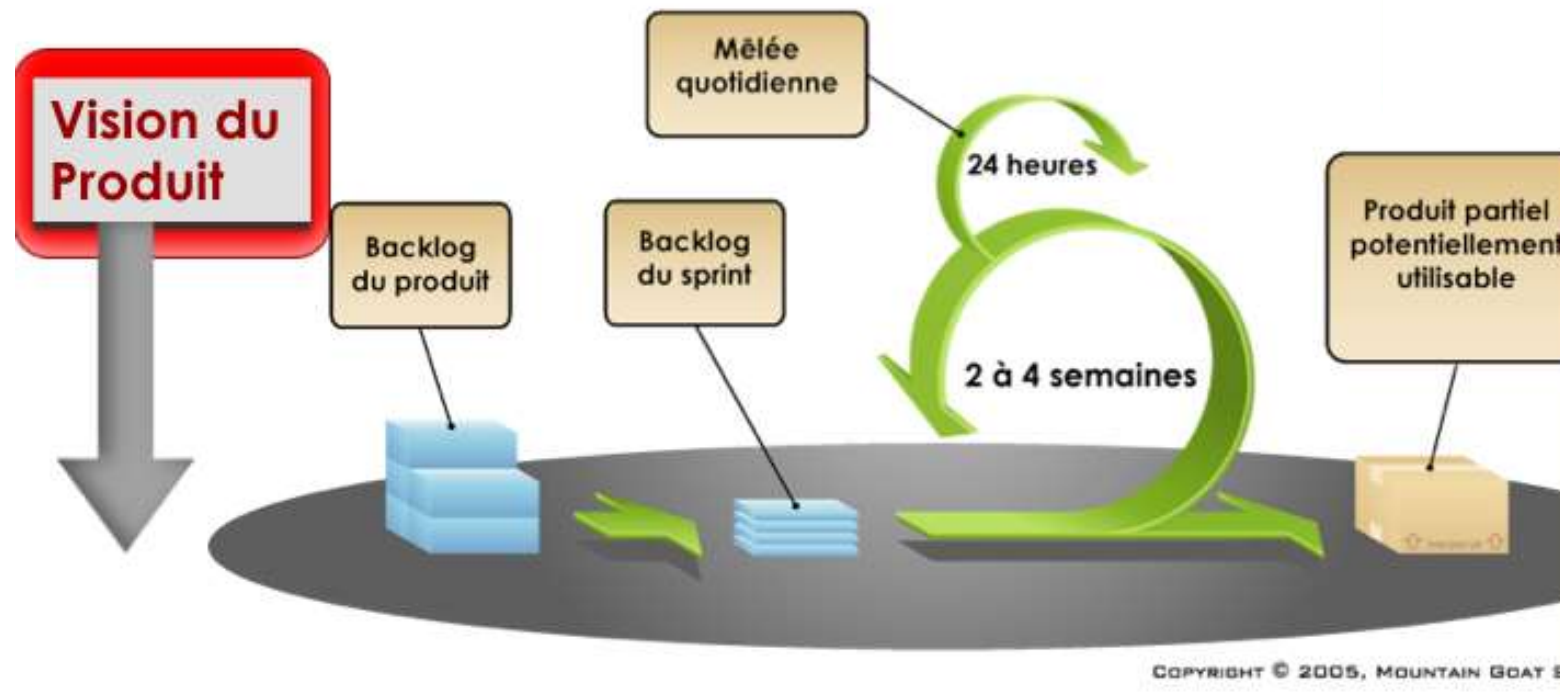
Tant que la fonctionnalité n'est pas réellement nécessaire, il est difficile de définir complètement ce qu'elle doit faire, et comment la tester. Si cette nouvelle fonctionnalité n'est pas correctement définie et testée, elle risque de ne pas fonctionner correctement lorsqu'elle sera un jour nécessaire.

Les méthodes Agiles

- ❑ Elles représentent une méthodologie reposant sur **une approche itérative et incrémentale** afin d'optimiser la prédictibilité et de contrôler les risques.
- ❑ Elles **favorisent l'intégration du client dans l'équipe** de développement d'un projet.
- ❑ Elles permettent une flexibilité et une adaptation aux changements des besoins d'un client.
- ❑ Naissance de l'Alliance Agile : février 2001 par 17 experts

Les méthodes Agiles: SCRUM

Scrum – Cycle 1/6



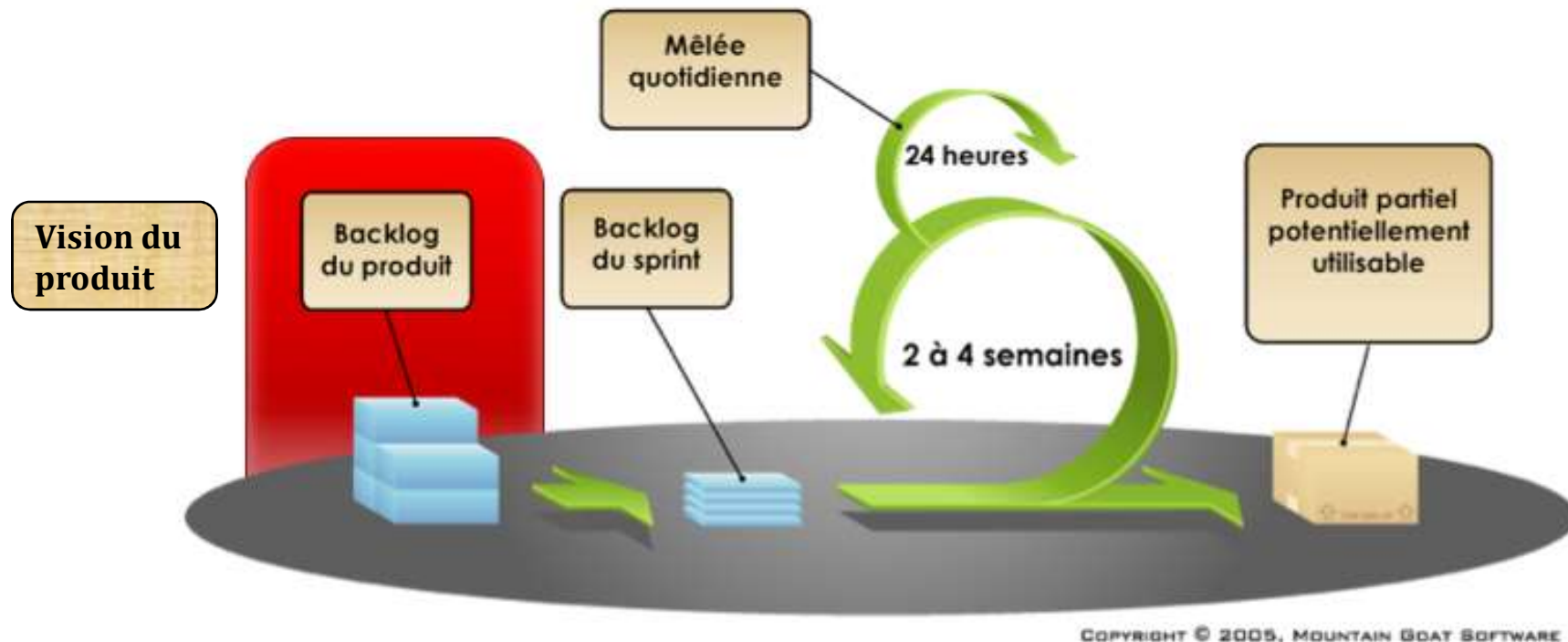
1. Définition de la vision du produit: c'est l'étape 0

- Le cap : si vous ne savez pas où vous voulez aller, où irez-vous?
- Une ou deux phrases



- Sans vision, les retards sont pris en **tout début de projet** ; l'équipe est alors hésitante, refusant de prendre **le risque de se tromper** de direction...
- Objectifs :
 - ✓ Situer le produit dans l'organization
 - ✓ Lui donner un sens (raison d'être du produit)

Scrum – Cycle 2/6



2. Backlog produit (ou carnet de produit, catalogue des besoins)

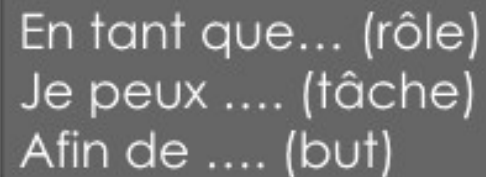
- ✓ Besoins priorisés par le product owner (Le propriétaire du produit)
- ✓ Besoins estimés par l'équipe, qui évoluent et sont affinés

Backlog Produit (Carnet de Produit)

- Contient des **User Stories** (Scrum): Au max 20 au début, pour respecter le vœux agile «peu de stock»

- DEF **User Story**

Exemples:



En tant que... (rôle)
Je peux (tâche)
Afin de (but)

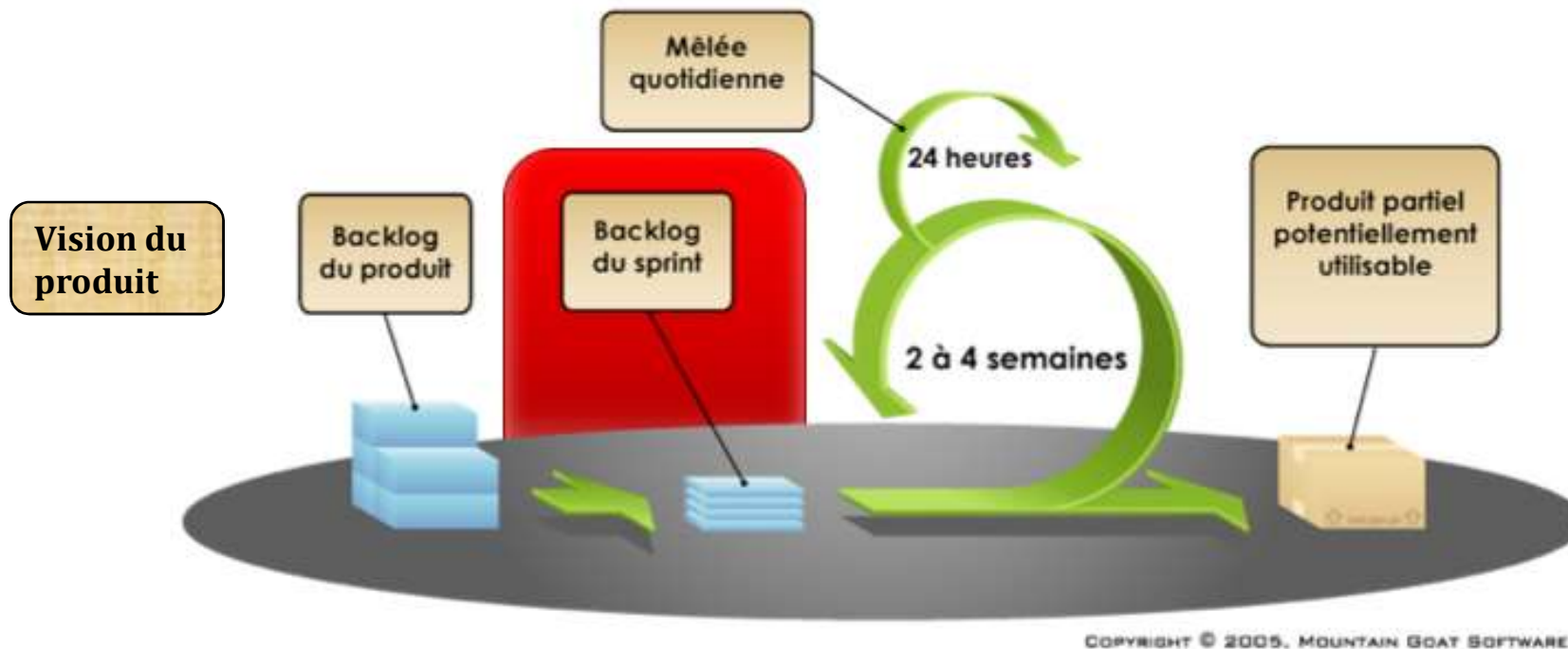
- «**En tant que** client, **je souhaite** pouvoir ajouter un produit dans mon panier **afin de** pouvoir l'acheter»
- « **En tant que** Pilote, **je peux** régler le commutateur en mode «horizontal» **afin de** maintenir les ailes à l'horizontal pour que l'avion reste sur sa trajectoire »

- Une **User Story** = un résumé formaté, pour avoir une vision rapide de la demande
- Elle sera discutée et détaillée ensuite avec l'équipe
- Une user story doit pouvoir être implémentée en une itération
- Une itération doit comporter au moins 4 User Stories

Comment obtenir les *User Stories* ?

- C'est le problème du Product Owner
- Méthodes possibles : *Brainstorming, identification des scénarios utilisateur,...*
- Par exemple, pour un **logiciel de gestion de carnet d'adresses**, un scénario est :
 - *"Je rentre un nom ou prénom, et le logiciel affiche la liste de toutes les personnes qui possèdent ce nom ou ce prénom"*
 - *"Je peux choisir d'exporter mon carnet d'adresses au format HTML ou XML"*

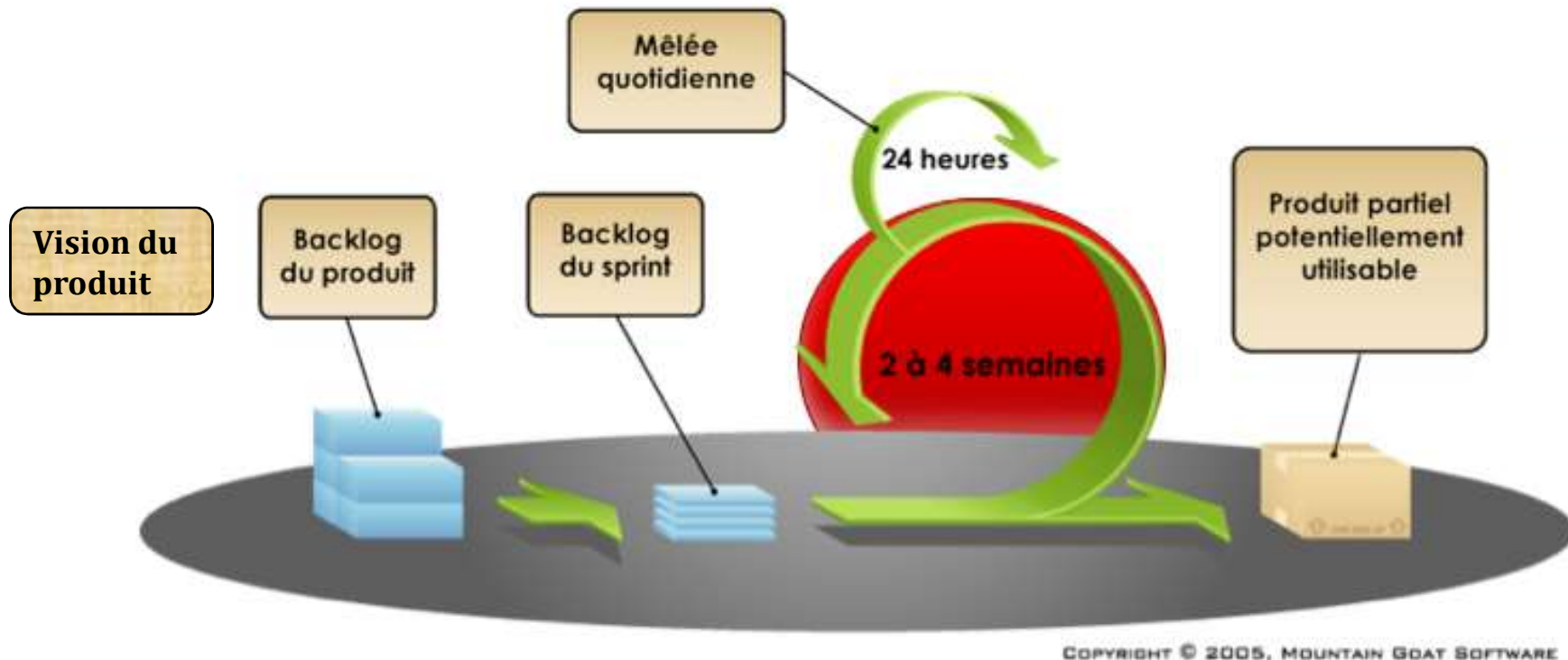
Scrum – Cycle 3/6



3. Planning Game : élaboration du backlog du sprint

- ✓ Sélection des besoins à réaliser sur le sprint, extrait du backlog produit
- ✓ Découpage en tâches, répartition de l'effort, planification

Scrum – Cycle 4/6

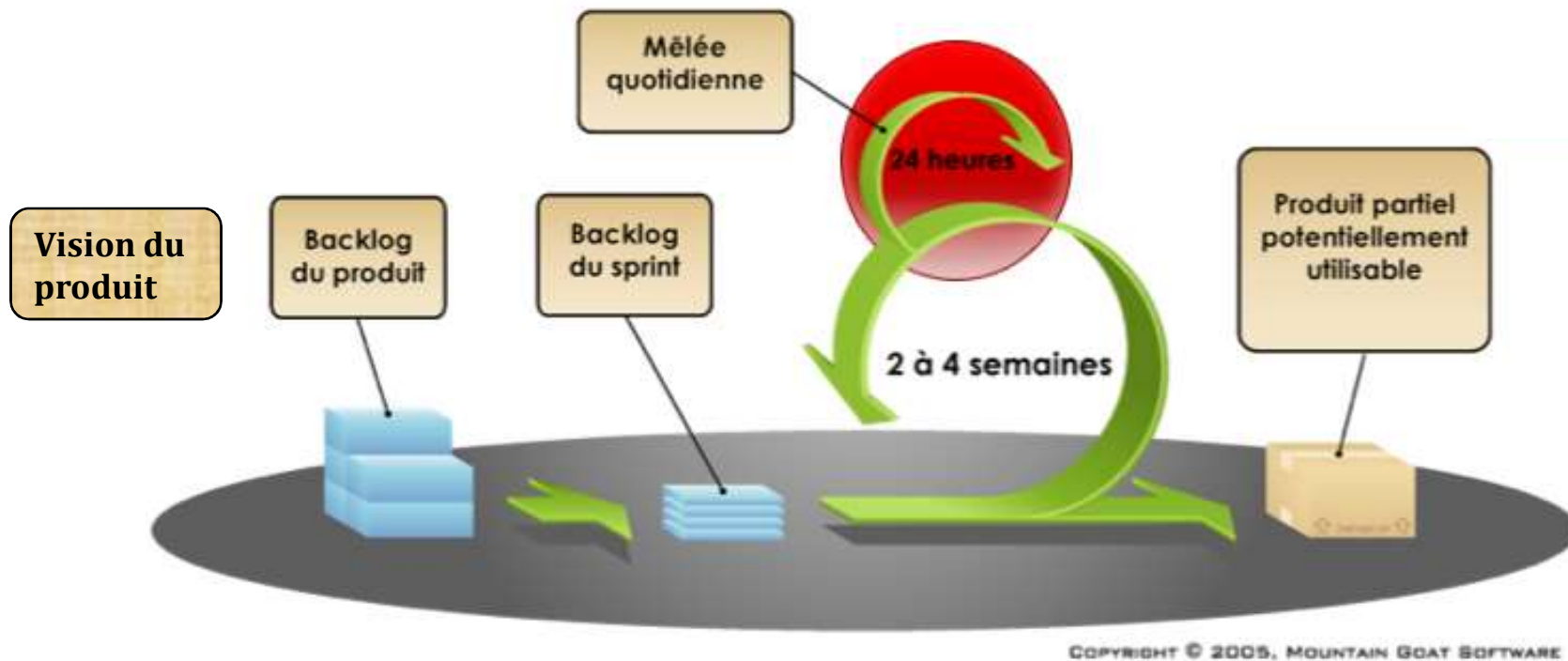


4. Sprint

- ✓ Développement des fonctionnalités du backlog du sprint
- ✓ Pas de modification du backlog du sprint possible
- ✓ Affinage du backlog Produit : une fois par semaine

Scrum – Cycle

5/6

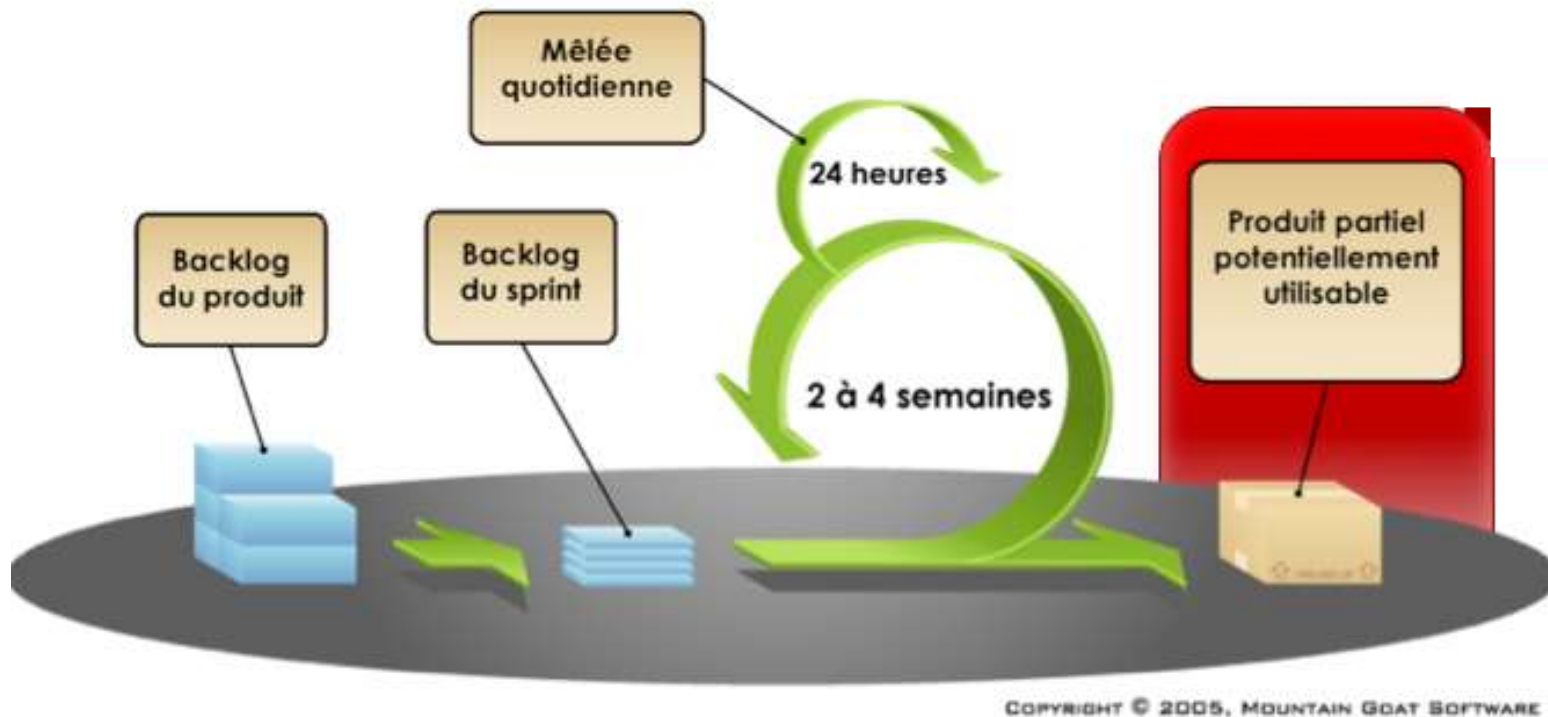


5. Mêlée quotidienne

- ✓ Point de contrôle quotidien de l'équipe
- ✓ Interventions régulées – 2 min. par personne

Scrum – Cycle

6/6



6. Incrément logiciel : livré au Product Owner à la fin du sprint :
donné aux commerciaux, prospects

Bibliographie



- J.P. Martin: « *La qualité des logiciels* », Association française de normalisation (AFNOR), Paris 1987.
- J.P. Martin: « *Qualité du logiciel et Système Qualité. L'industrialisation par la certificataion* », Masson, 1992.
- « ISO 9001 et le développement du logiciel. Guide l'application », AFNOR, 1996
- C. H. Schmauch: « ISO 9000 for software developers », ASQC Quality Press, 1994
- T. Forse: « *Qualimétrie des systèmes complexes* », Les Editions d'Organisation, 1989.
- V. Deslandres : Cours 2 Développement agile La méthode SCRUM

