Cours 1 - Gestion d'un projet informatique et cycle de vie.

Halim Djerroud <halim.djerroud@uvsq.fr>

LISV
Laboratoire d'ingénierie
des systèmes de Versailles

UVSQ
université PARIS-SACLAY

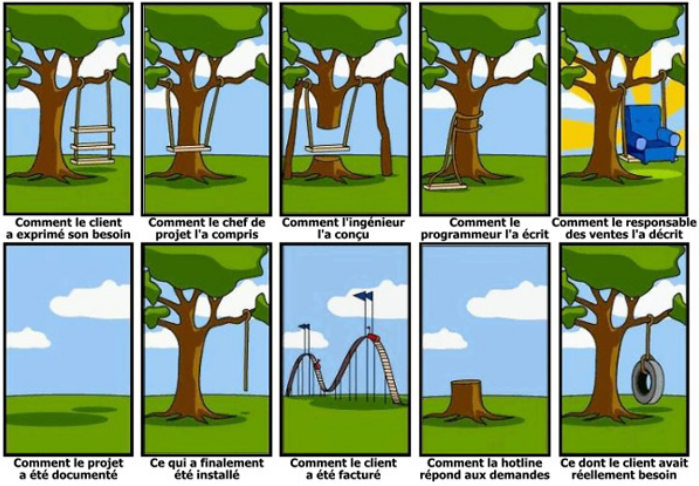
révision : 3.0

Plan

Gestion d'un projet informatique et cycle de vie

1. Gestion de projet informatique
   - Développer des systèmes complexes
   - Le Génie logiciel
   - Les activités du Génie Logiciel
2. Cycle de vie du logiciel
3. Vers d'autres méthodes de gestion
   - Constat
   - Les méthodes agiles
   - Cycles de vie multiniveaux, l'exemple du processus unifié

Un projet, dix visions différentes



Caricature anonyme bien connue des informaticiens (« *tree swing* », années 1970), nombreuses variantes.

Que raconte cette image ?

À chaque étape, le besoin est **déformé** :

- le chef de projet **interprète** ce que dit le client ;
- l'ingénieur conçoit ce qu'il **a compris** ;
- le programmeur code ce qu'il **peut** ;
- la documentation **n'existe pas** ;
- ce qui est installé ne ressemble à rien de tout cela.

Et surtout : personne n'a vraiment cherché ce dont le client **avait besoin**.

Le problème

Le client demande une balançoire à trois planches.
Il avait besoin d'un **pneu**.

L'objet de ce module

Des **méthodes** pour comprendre, formaliser et valider le besoin, et organiser le développement.

Objectif du cours

Objectif du cours :

- **Gestion de projet** pour réaliser des **solutions informatiques** avec de **fortes contraintes de qualité** dans le cadre des **systèmes d'information des organisations**.

C'est quoi un système d'information ?

Définition 1.1 : Système d'information (SI) (Reix et al. 2016)

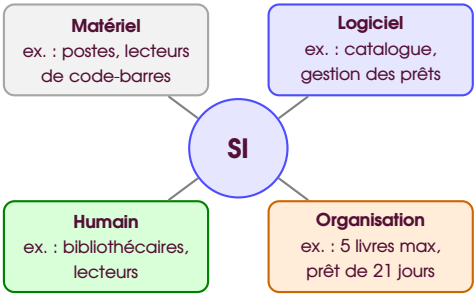
Un système d'information est un ensemble organisé de ressources (matériels, logiciels, personnel, données et procédures) qui permet de :

- **acquérir** l'information (sur différents supports) ;
- **traiter** l'information (effectuer des opérations) ;
- **stocker** l'information (conserver l'information) ;
- **communiquer** l'information (transmettre : éditer, imprimer, etc.).

Composition d'un SI

Nature hétérogène :

- Le SI est composé de ressources matérielles, logicielles, humaines et organisationnelles.
- Les **ressources matérielles** sont les composants physiques : ordinateurs, serveurs, réseaux, etc.
- Les **ressources logicielles** sont les programmes qui permettent de faire fonctionner le système.
- Les **ressources humaines** sont les personnes qui utilisent le système et qui en assurent le développement, la maintenance et le support.
- Les **ressources organisationnelles** sont les règles, les procédures et les structures de l'organisation qui encadrent le développement et l'utilisation du système.



Exemple : une bibliothèque universitaire.

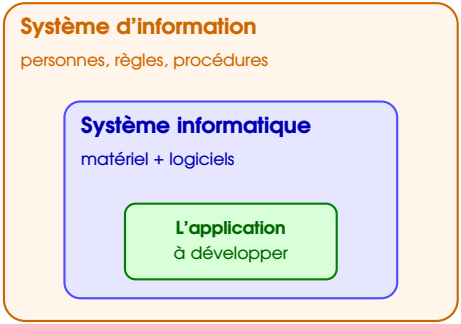
SI = Matériel + Logiciel + Humain + Organisation

Tâches accomplies par un SI

- La collecte, le stockage et la diffusion de l'information
- Le traitement de l'information
- La prise de décision
- La communication
- La collaboration

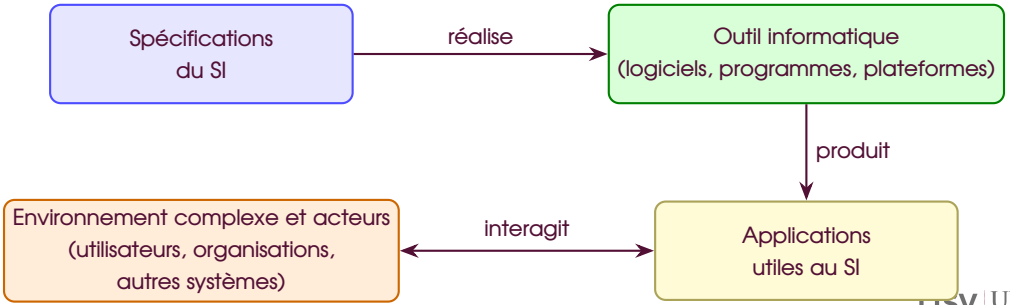
Attention

Un **système informatique** fait partie d'un système d'information mais **ne constitue pas** nécessairement un **système d'information**.



À quoi sert l'outil informatique ?

- Les outils de l'informatique permettent de réaliser les spécifications d'un SI.
- Plusieurs systèmes informatiques peuvent interagir pour réaliser ces spécifications, ou remplacer d'autres outils.



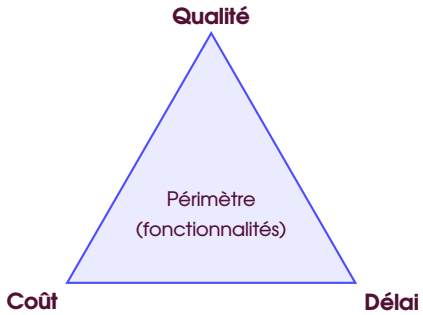
Qu'est-ce qu'un projet ?

Un **projet** est un ensemble d'activités :

- orientées vers un **objectif** précis ;
- avec un **début** et une **fin** ;
- réalisées avec des **ressources limitées** (personnes, budget, matériel) ;
- produisant un résultat **unique**.

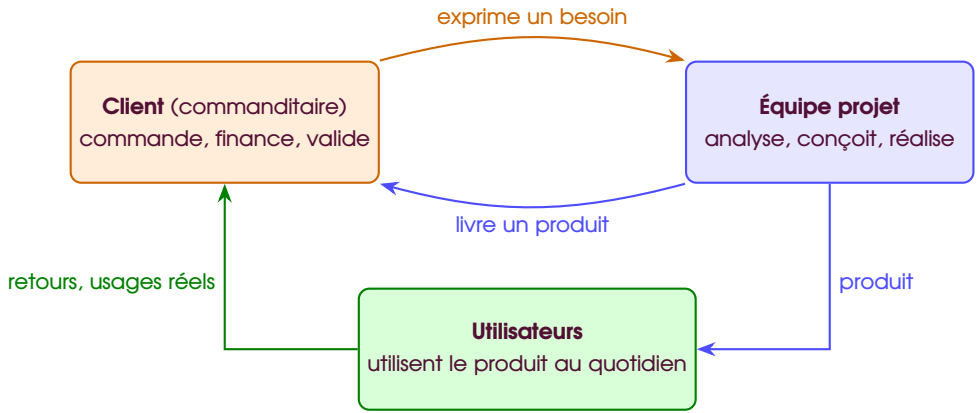
Réussir un projet

Livrer ce qui **répond au besoin**, dans les **délais**, le **budget** et avec la **qualité** attendus.



On ne peut pas tout gagner : réduire le délai coûte en qualité, en budget ou en fonctionnalités.

Les acteurs d'un projet informatique



Le client **n'est pas toujours** l'utilisateur. Il faut identifier et rencontrer **les deux**, dès le début.

Méthodologie pour la gestion de projet informatique

Nous nous intéressons donc ici aux **méthodes** nécessaires au **développement des applications** utiles aux systèmes d'information, et s'intégrant de fait dans un environnement complexe avec différents acteurs.

Afin de proposer une méthodologie pour la gestion de projet informatique dans le contexte des systèmes d'information, il nous faut définir :

- le cadre théorique spécifique à l'informatique et la manière dont on peut **découper les activités** d'un développeur ;
- cet enchaînement définit ce que l'on appelle le **cycle de vie logiciel** ;
- il n'y a pas de cycle unique ;
- il est possible d'adapter son cycle de vie logiciel et de définir un processus métier particulier pour le développeur.

Le Génie logiciel

- Le génie logiciel nous permet de définir une méthodologie pour la réalisation d'un logiciel.

Définition

Le Génie Logiciel est la discipline liée à tous les aspects de la production du logiciel **complexe** et avec d'importantes **contraintes de qualité**. Elle est liée à l'application de **théories**, de **méthodes** et à l'utilisation d'**outils** pour le développement logiciel d'une façon professionnelle. Elle favorise et permet le **travail en équipe**.

Les objectifs du génie logiciel :

- adopter une approche **systématique et organisée** ;
- utiliser les techniques et outils appropriés selon la nature du problème, les contraintes de développement et les ressources.

Principes du génie logiciel (GL)

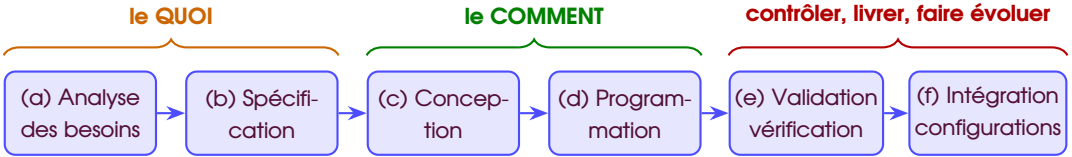
1. **Rigueur et formalisme** : faciliter la communication entre développeurs et avec le client.
   ex. : une exigence écrite sans ambiguïté.
2. **Abstraction** : identifier et définir les concepts du projet ainsi que les éléments qui le composent.
   ex. : livre, lecteur, emprunt.
3. **Modularité** : proposer une décomposition en sous-systèmes.
   ex. : catalogue, prêts, comptes.
4. **Anticipation** : prendre en compte les contraintes techniques et organisationnelles.
   ex. : montée en charge, RGPD.
5. **Généralisation** : réutiliser les **composants** et les **méthodes**.
   ex. : bibliothèques, frameworks, patrons.
6. **Croissance incrémentale** : faire évoluer le logiciel étape par étape.
   ex. : version 1 minimale, puis enrichie.

Le génie logiciel est à l'informatique ce que le génie civil est à la construction : une ingénierie structurée qui transforme une idée en un logiciel robuste, maintenable et évolutif.

Les activités du Génie Logiciel

Grandes étapes nécessaires pour réaliser un logiciel.

Pour mener à bien un projet informatique, le développeur ou l'équipe de développement doit réaliser un ensemble d'activités allant de l'analyse des besoins du projet à la maintenance du logiciel produit, en passant par son implémentation et les tests.



(a) Analyse des besoins (ADB)

Rôle

- Identifier le problème
- Documenter les exigences
- Impliquer les utilisateurs et les experts dans le domaine de l'application

Entrées

- Cahier des charges (rédigé par le client)
- Données fournies par les experts du domaine et par les utilisateurs potentiels

$\Rightarrow$

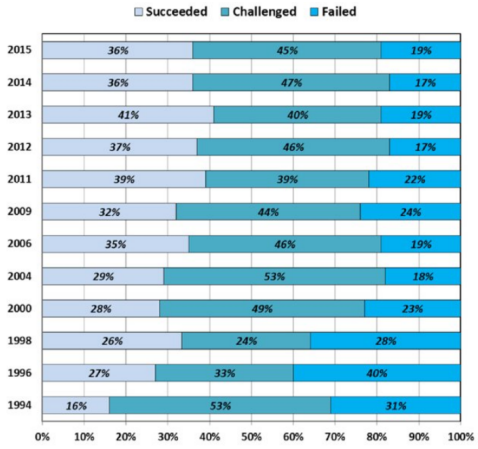
Résultats

- Document d'ADB destiné aux utilisateurs et utile pour les développeurs
- Recueil des besoins
- Cahier des charges (rédigé de manière technique avec le client)

Remarque : le produit ne correspond pas toujours au besoin

Attention, de nombreux produits ne correspondent pas aux besoins du client. On peut identifier plusieurs causes, comme par exemple :

- le client ne sait pas ce qu'il veut ;
- il existe des problèmes de communication entre le client et l'équipe de développement ;
- l'équipe de développement ne comprend pas la politique d'organisation du client ;
- on observe un changement des exigences au cours du projet ;
- les délais ne sont pas raisonnables ;
- etc.



Source : Standish Group, *CHAOS Report* (1994--2015).

Succeeded : réussi ; *Challenged* : en difficulté ; *Failed* : abandonné.

Moins de **4 projets sur 10** aboutissent dans les délais

(b) Spécification

(a) Analyse des besoins → **(b) Spécification** → (c) Conception → (d) Programmation → (e) Validation et vérif. → (f) Intégration

Rôle

- Décrire de façon précise le système à construire, le **QUOI** et non le comment.

Entrées

- Document d'ADB destiné aux utilisateurs et utile pour les développeurs
- Recueil des besoins
- Cahier des charges (rédigé de manière technique avec le client)

⇒

Résultats

- Documentation textuelle avec description de scénarios (cas d'utilisation), possiblement appuyée par des diagrammes (ex. : standard UML)

(c) Conception

(a) Analyse des besoins → (b) Spécification → **(c) Conception** → (d) Programmation → (e) Validation et vérif. → (f) Intégration

Rôle

- Enrichir la spécification du logiciel en l'orientant vers la réalisation, le **COMMENT** et non le quoi.

Entrées

- Spécifications
- Détail de la plateforme de développement (dans le recueil des besoins)

⇒

Résultats

- **Modèle** défini dans une documentation textuelle appuyée par des diagrammes (ex. : standard UML)

À vous : spécification ou conception ?

Pour une bibliothèque universitaire, classez chaque phrase :

#	Phrase	Quoi ou comment ?
1	Un lecteur peut emprunter 5 livres au maximum, pendant 21 jours.	Spécification
2	Les emprunts sont enregistrés dans une table `emprunt` d'une base MySQL.	Conception
3	Le lecteur reçoit un rappel 2 jours avant la date de retour.	Spécification
4	Les rappels sont envoyés par une tâche planifiée chaque nuit à 2h.	Conception
5	La recherche dans le catalogue répond en moins de 2 secondes.	Spécification (non fonctionnelle)

À vous : spécification ou conception ?

Pour une bibliothèque universitaire, classez chaque phrase :

#	Phrase	Quoi ou comment ?
1	Un lecteur peut emprunter 5 livres au maximum, pendant 21 jours.	**Spécification**
2	Les emprunts sont enregistrés dans une table `emprunt` d'une base MySQL.	**Conception**
3	Le lecteur reçoit un rappel 2 jours avant la date de retour.	**Spécification**
4	Les rappels sont envoyés par une tâche planifiée chaque nuit à 2h.	**Conception**
5	La recherche dans le catalogue répond en moins de 2 secondes.	**Spécification** (non fonctionne

(d) Programmation

(a) Analyse des besoins → (b) Spécification → (c) Conception → **(d) Programmation** → (e) Validation et vérif. → (f) Intégration

Rôle

- Traduire la conception en **code exécutable**, dans un langage de programmation.
- Respecter des normes de codage, documenter le code, le versionner.

Entrées

- Documentation de la conception détaillée
- Spécification (si pas de conception détaillée)

⇒

Résultats

- Code qui implémente les fonctionnalités
- Documentation technique du code
- Dépôt, historique de gestion de version

(e) Validation (e1) et Vérification (e2)

(a) Analyse des besoins → (b) Spécification → (c) Conception → (d) Programmation → **(e) Validation et vérif.** → (f) Intégration

Rôle

- (e1) Répond à la question « Le logiciel satisfait-il les attentes de l'utilisateur ? »
- (e2) Répond à la question « Le logiciel satisfait-il les spécifications ? »

Entrées

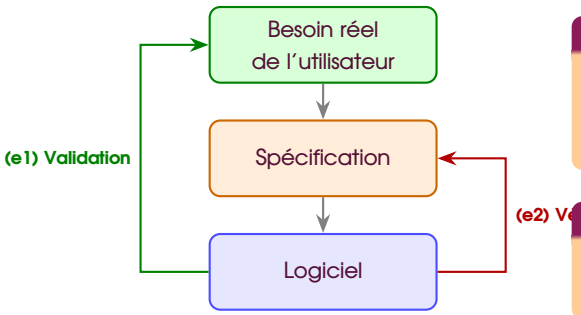
- Documentation de la conception détaillée pour les tests unitaires
- Documentation de la conception architecturale pour les tests d'intégration
- Documentation d'ADB pour les tests d'acceptation

⇒

Résultats

- Cas de tests
- Analyse des résultats d'exécution des cas de tests

Validation ou vérification ?



(e2) Vérification

*Construit-on le produit **correctement** ?*

Ex. : un 6e emprunt est bien refusé, comme le dit la spécification.

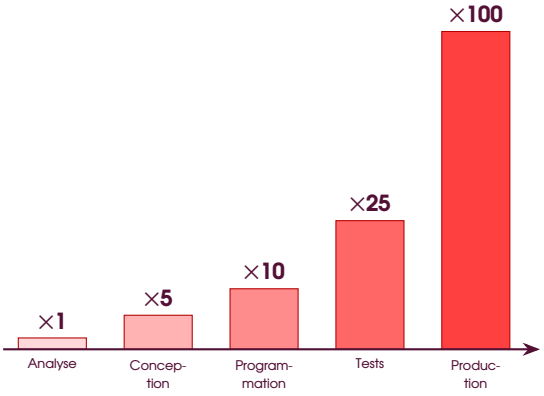
(e1) Validation

*Construit-on le **bon** produit ?*

Ex. : les bibliothécaires utilisent-ils vraiment le logiciel ?

Un logiciel peut être **vérifié** sans être **validé** : si la spécification est fausse, on construit parfaitement la mauvaise balançoire.

Plus une erreur est découverte tard, plus elle coûte cher



Ordres de grandeur, d'après B. Boehm, *Software Engineering Economics*, 1981.

Une erreur d'**analyse** (un besoin mal compris) :

- corrigée pendant l'analyse : on modifie une phrase ;
- découverte en production : on reprend la conception, le code, les tests, la documentation, on redéploie. . .

D'où l'importance de l'**analyse des besoins**, et de **montrer le produit tôt**.

LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

(f) Intégration et gestion des configurations

(a) Analyse des besoins → (b) Spécification → (c) Conception → (d) Programmation → (e) Validation et vérif. → **(f) Intégration**

Rôle

- Décrire le système et ses évolutions.
- **Assembler** les composants développés et vérifier qu'ils fonctionnent ensemble.
- **Gérer les versions** (configurations), livrer, déployer et maintenir le logiciel.

Entrées

- Version du logiciel
- Composants développés et testés (branches du dépôt)

⇒

Résultats

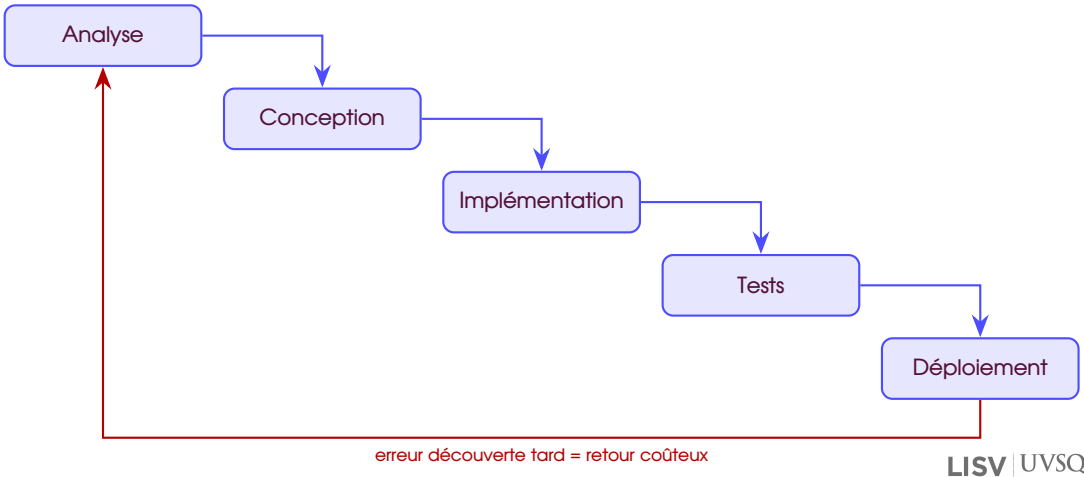
- Version livrée et déployée, historique des versions
- Dossier de maintenance

Processus de développement

- L'équipe projet suit un **processus de développement** : ensemble d'activités et de résultats qui conduisent à la création d'un produit logiciel.
- Le processus de développement indique la forme dans laquelle les activités sont connectées entre elles. L'ordre dans lequel s'enchaînent les activités s'appelle le **cycle de vie** du produit logiciel.

Toutes les activités existent dans tout projet.
La question est : **dans quel ordre, et combien de fois ?**

Cycle de vie linéaire : cascade



Cycle de vie en cascade

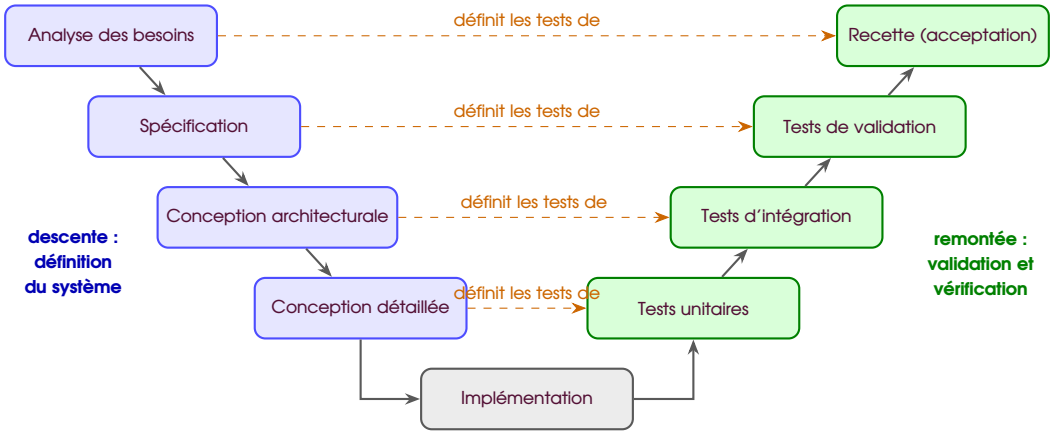
Principe du cycle en cascade

- Déroulement **séquentiel** des étapes du projet (analyse → conception → implémentation → tests → déploiement).
- Chaque phase doit être **terminée et validée** avant de passer à la suivante.
- Approche simple et **structurée**, bien adaptée aux projets courts et bien définis.

Limites principales du modèle en cascade

- **Rigidité** : difficile de revenir en arrière une fois une étape terminée.
- **Peu flexible** face aux changements de besoins en cours de projet.
- **Décalage temporel** : le produit final n'est visible qu'à la fin du cycle.
- **Risque élevé** de découverte tardive d'erreurs ou de mauvaises interprétations.
- **Faible implication du client** durant la réalisation.

Cycle de vie linéaire : V



Cycle de vie en V

Principe du cycle en V

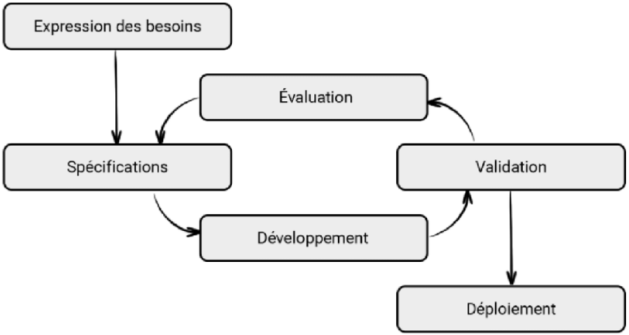
- Chaque étape de conception est associée à une étape de test.
- Les tests doivent être définis **en même temps** que la conception.
- Descente du V = **définition du système**.
- Remontée du V = **validation et vérification**.

Utilisé pour les systèmes **critiques** (aéronautique, ferroviaire, médical) : forte traçabilité exigences / tests.

Limites principales du modèle en V

- **Rigidité** : difficile de revenir en arrière si une erreur est découverte.
- **Peu flexible** face aux changements de besoins en cours de projet.
- **Coût élevé** des corrections lorsqu'elles apparaissent tardivement.
- **Manque d'agilité** : pas adapté aux projets innovants ou incertains.
- **Client impliqué tardivement** : la validation réelle se fait souvent à la fin.

Cycle de vie itératif et incrémental



Cycle de vie incrémental

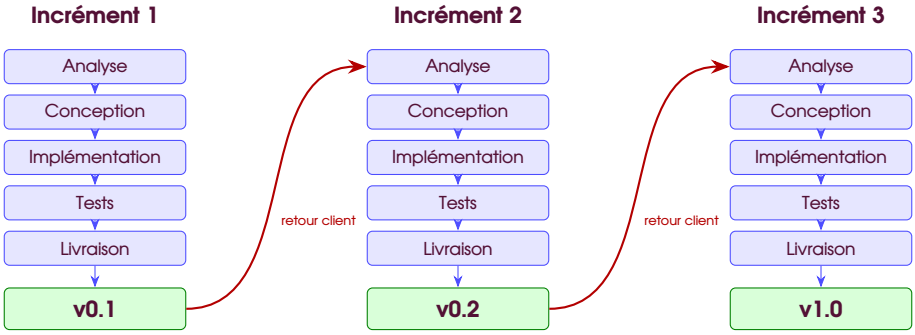
Principe du cycle incrémental

- Le développement est découpé en **incréments** (versions partielles du logiciel).
- Chaque incrément apporte **de nouvelles fonctionnalités utilisables**.
- Le client peut disposer rapidement d'une **version fonctionnelle minimale**.
- Favorise l'**adaptation** et l'**amélioration progressive**.

Limites principales du modèle incrémental

- **Complexité de gestion** : nécessite une planification rigoureuse.
- **Intégration progressive difficile** : risque de problèmes d'assemblage entre incréments.
- **Dépendance au découpage** : un mauvais découpage des incréments peut nuire au projet.
- **Coûts plus élevés** si des tâches sont répétées d'un incrément à l'autre.
- **Risque d'inachèvement** : le client peut rester trop longtemps sur une version partielle.

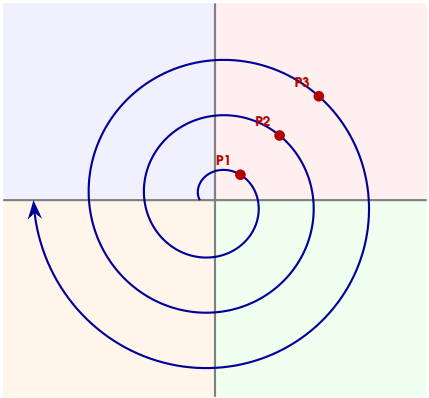
Incrémental : le produit grandit version après version



Chaque incrément se termine par une **version utilisable**, montrée au client.
Une erreur d'analyse est découverte en quelques semaines, pas à la fin du projet.

Cycle de vie en spirale (Boehm, 1988)

1. Déterminer les objectifs, les alternatives et les contraintes du tour



2. Évaluer les alternatives, **identifier et réduire les risques** (prototypes P1, P2, P3)

4. Planifier le tour suivant (revue avec le client : continuer ?)

3. Développer et vérifier le produit du tour (spécification, conception, code, tests)

Chaque tour de spirale est un cycle complet ; le rayon représente le **coût cumulé** du projet.

Cycle de vie en spirale

Principe du cycle en spirale

- Le projet avance par **tours** successifs, chacun en 4 étapes.
- Il est **piloté par les risques** : à chaque tour, on traite d'abord le risque le plus menaçant (besoin flou, technologie nouvelle, performance. . .).
- Les risques sont souvent réduits par un **prototype**.
- Le produit s'enrichit à chaque tour ; le client valide à la fin de chaque tour.

Limites principales du modèle en spirale

- Demande une réelle **expertise en analyse des risques**.
- **Gestion complexe et coûteuse** : trop lourd pour un petit projet.
- Délais et coût total **difficiles à fixer** dans un contrat au départ.

Adapté à

Projets **importants**, **innovants** ou **risqués**.

Source : B. W. Boehm, « A Spiral Model of Software Development and Enhancement », *IEEE Computer*, 21(5), 1988.

La spirale a inspiré les démarches itératives qui ont suivi, dont le Processus Unifié.

Le prototypage

Définition

Un **prototype** est une version simplifiée ou partielle du système, construite **rapidement** pour répondre à une question : le besoin est-il bien compris ? l'interface convient-elle ? la solution technique est-elle faisable ?

Maquette jetable

Croquis papier, écrans dessinés, maquette cliquable.
Sert à **clarifier le besoin** avec le client, puis est **jetée**.

Prototype évolutif

Version réduite mais **réelle** du logiciel, enrichie jusqu'au produit final.
Proche du cycle **incrémental**.

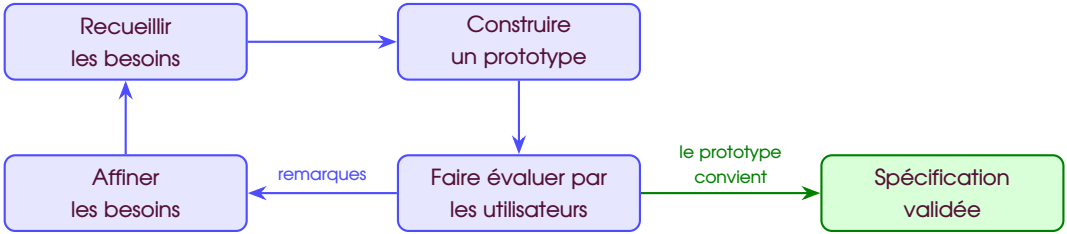
Preuve de concept

Petit programme qui vérifie la **faisabilité technique** d'un point risqué.
Ex. : se connecter à l'ENT (CAS).

Pièges

Le client prend la maquette pour le produit fini (« c'est presque terminé ? ») ; du code jetable finit en production ; on passe trop de temps à peaufiner un prototype.

Prototypage et analyse des besoins



- Le client réagit bien mieux à un **écran concret** qu'à un texte.
- Les malentendus apparaissent **avant** d'écrire le code : la correction coûte peu.
- Le prototype complète les **cas d'utilisation**, il ne les remplace pas.

Retour à la balançoire

Un simple croquis montré au client le premier jour aurait révélé qu'il voulait **un pneu**.

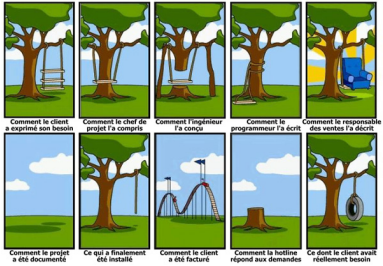
Comparer les cycles de vie

	Cascade	Cycle en V	Incrémental	Spirale
Besoins	connus, stables	connus, stables, critiques	flous ou évolutifs	incertains, risques élevés
Le client voit le produit	à la fin	à la recette	à chaque incrément	à chaque tour (prototypes)
Erreurs d'analyse découvertes	très tard	tard	tôt	tôt
Point fort	simplicité	traçabilité exigences / tests	adaptation, risque réduit	maîtrise des risques
Exemple	script d'export vers un format imposé	logiciel embarqué médical	application web ou mobile	grand système innovant

À vous : et dans votre dernière SAÉ ?

Répondez en 2 minutes, avec votre voisin :

1. Dans quel ordre avez-vous enchaîné les activités ?
2. À quel moment avez-vous montré l'application au client pour la **première fois** ?
3. Quand avez-vous découvert les **plus grosses erreurs** ?
4. Quand avez-vous fusionné le code de toute l'équipe ?



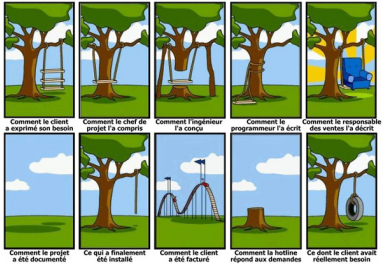
Constat fréquent

Beaucoup d'équipes suivent une **cascade sans le savoir** : intégration et tests la dernière semaine, client découvert à la soutenance.

À vous : et dans votre dernière SAÉ ?

Répondez en 2 minutes, avec votre voisin :

1. Dans quel ordre avez-vous enchaîné les activités ?
2. À quel moment avez-vous montré l'application au client pour la **première fois** ?
3. Quand avez-vous découvert les **plus grosses erreurs** ?
4. Quand avez-vous fusionné le code de toute l'équipe ?



Constat fréquent

Beaucoup d'équipes suivent une **cascade sans le savoir** : intégration et tests la dernière semaine, client découvert à la soutenance.

Les méthodes agiles

Définition : Les méthodes agiles sont des approches de gestion de projet logiciel qui privilégient :

- l'**adaptation au changement** plutôt que le suivi rigide d'un plan ;
- la **collaboration** avec le client et les utilisateurs ;
- la livraison **rapide et fréquente** de versions fonctionnelles ;
- la **communication** et l'implication de l'équipe.

Valeurs du Manifeste Agile (2001)

- Les **individus et leurs interactions** *plus que* les processus et les outils.
- Un **logiciel opérationnel** *plus qu'*une documentation exhaustive.
- La **collaboration avec le client** *plus que* la négociation contractuelle.
- L'**adaptation au changement** *plus que* le suivi d'un plan.

Les éléments de droite ont de la valeur, ceux de gauche en ont **davantage**. Agile ne veut pas dire « sans documentation » ni « sans plan ».

Source : https://agilemanifesto.org/iso/fr/manifesto.html

Méthodes agiles et cycles de vie

Relation entre méthodes agiles et cycles de vie :

- Les cycles de vie classiques (cascade, V, incrémental) définissent une **organisation des étapes**.
- Les méthodes agiles apportent une **philosophie de gestion** :
  - itérations courtes et incrémentales ;
  - validation continue avec le client ;
  - adaptation permanente aux changements.
- Chaque itération agile correspond à un **mini-cycle de vie complet** :
  - analyse → conception → implémentation → tests → livraison.

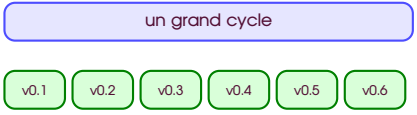
Comparatif :

Cycle classique

Un seul grand cycle, produit visible à la fin.

Cycle agile

Enchaînement de petits cycles, produit évolutif visible rapidement.



Principales méthodes agiles

Quelques méthodes agiles largement utilisées :

Scrum

- Organisation par **sprints** (cycles courts d'un mois au maximum).
- Rôles : *Scrum Master*, *Product Owner*, équipe de développement.
- Livraison d'un produit opérationnel à chaque sprint.

Kanban

- Gestion visuelle du travail via un **tableau de tâches**.
- Flux continu, limitation du nombre de tâches en cours (WIP).

Extreme Programming (XP)

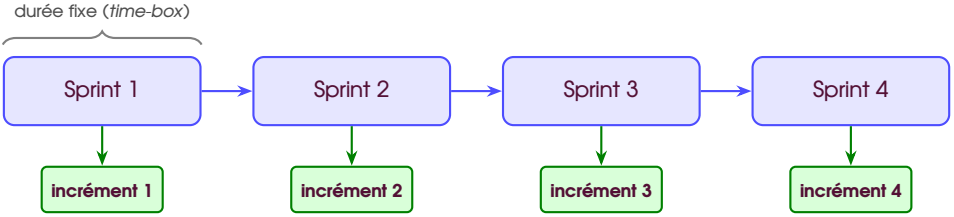
- Mise en avant de la **qualité du code**.
- Programmation en binôme, intégration continue, tests unitaires automatisés.

Lean Software Development

- Inspiré du *lean manufacturing* (Toyota).
- Élimination du gaspillage, amélioration continue, respect des personnes.

Méthode agile : Scrum

- Organisation du projet en **sprints** : durée fixe d'**un mois au maximum**, souvent 2 semaines.
- Chaque sprint aboutit à un **produit fonctionnel et livrable**.
- L'équipe est **auto-organisée** et pluridisciplinaire.
- Le client participe via le **Product Owner**.



Référence : *The Scrum Guide* (K. Schwaber, J. Sutherland, 2020), `https://scrumguides.org`

Scrum : rôles et artefacts

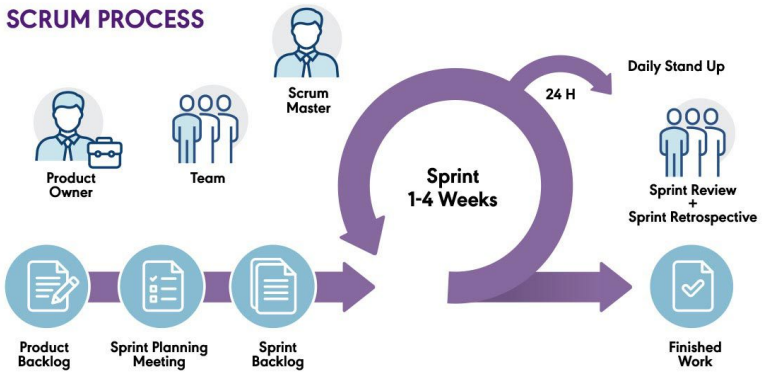
Rôles clés

- **Product Owner** : définit et priorise les besoins (backlog produit).
- **Scrum Master** : facilite la méthode et supprime les obstacles.
- **Équipe de développement** : réalise le produit incrémental.

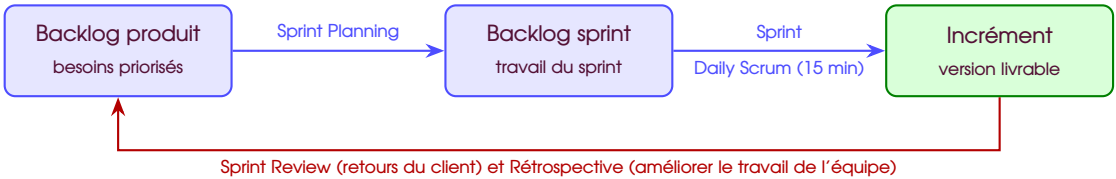
Artefacts principaux

- **Backlog produit** : liste des fonctionnalités à développer.
- **Backlog sprint** : tâches prévues pour le sprint en cours.
- **Incrément** : version livrable du produit.

SCRUM


SCRUM PROCESS
Scrum Master
Daily Stand Up
24 H
Product Owner
Team
Sprint 1-4 Weeks
Sprint Review + Sprint Retrospective
Product Backlog
Sprint Planning Meeting
Sprint Backlog
Finished Work

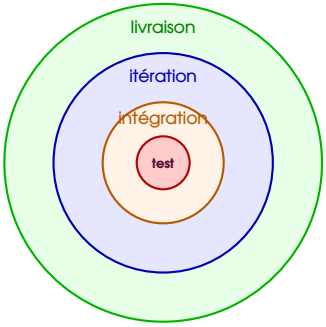
Scrum : du backlog à l'incrément



Sprint Planning	Choisir les éléments du backlog produit pour le sprint et fixer l'objectif.
Daily Scrum	Chaque jour : qu'ai-je fait ? que vais-je faire ? qu'est-ce qui me bloque ?
Sprint Review	Montrer l'incrément au Product Owner et au client, recueillir leurs retours.
Rétrospective	L'équipe améliore sa façon de travailler pour le sprint suivant.

Méthode agile : Extreme Programming (XP)

- Développement en **itérations courtes**, avec livraisons fréquentes.
- **Tests unitaires automatisés** écrits en même temps que le code.
- **Refactoring** régulier pour garder le code simple et maintenable.
- **Programmation en binôme** (*pair programming*) pour améliorer la qualité.
- **Intégration continue** : code intégré et testé plusieurs fois par jour.
- Collaboration étroite et continue avec le **client**.



Des boucles de retour de plus en plus courtes : test (minutes), intégration (heures), itération (jours), livraison (semaines).

XP : avantages et limites

Avantages

- Améliore la **qualité du logiciel** et réduit les bugs.
- Grande **flexibilité** face aux changements de besoins.
- Favorise le **travail en équipe** et le partage de connaissances.

Limites

- Exige une **discipline stricte** de l'équipe.
- La programmation en binôme peut sembler **coûteuse**.
- Moins adapté aux projets très grands et dispersés.

Adapté à

Petite équipe réunie, besoins qui changent souvent, client disponible, exigence forte de fiabilité du code.

Les pratiques d'XP (tests automatisés, intégration continue) sont aujourd'hui utilisées bien au-delà d'XP.

Méthode agile : Kanban

- Gestion du flux de travail à l'aide d'un **tableau Kanban**.
- Les tâches sont représentées par des **cartes** organisées en colonnes :
  - **À faire** (*To Do*)
  - **En cours** (*In Progress*)
  - **Terminé** (*Done*)
- Limitation du **travail en cours** (WIP : *Work In Progress*).
- Approche **visuelle et continue** : le flux de tâches est optimisé en permanence.

Différence avec Scrum

Pas de sprint : les tâches **arrivent et partent en continu**.

Adapté à la **maintenance** et au **support**, où les demandes sont imprévisibles.

Kanban : avantages et limites

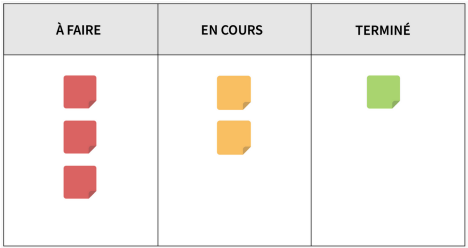
Avantages

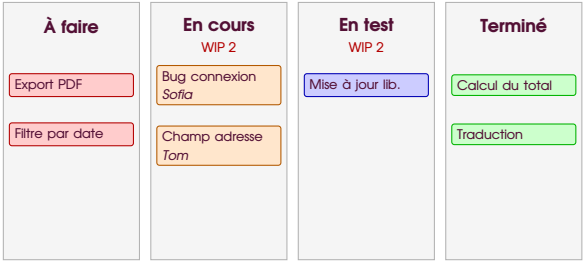
- Grande **simplicité** et mise en place rapide.
- Améliore la **transparence** et la communication dans l'équipe.
- Permet de **visualiser les blocages** et d'optimiser le flux.

Limites

- Moins structuré que Scrum : nécessite une **autodiscipline élevée**.
- Ne définit pas de rôles ni de rituels spécifiques.
- Moins adapté aux **projets très complexes**.

Méthode agile : Kanban





La colonne « En cours » est pleine (2/2) :

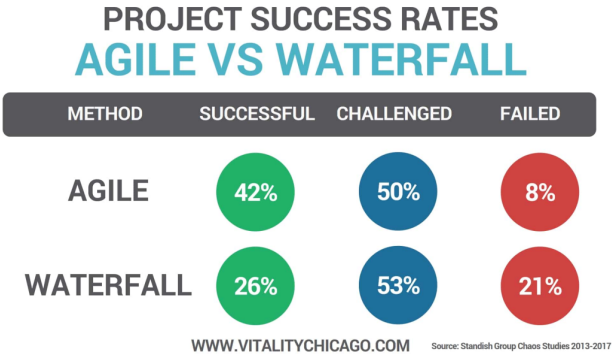
on **termine** une tâche avant d'en commencer une autre.

Scrum, XP, Kanban : comparaison

	Scrum	**XP**	**Kanban**
Rythme	sprints à durée fixe	itérations courtes, livraisons très fréquentes	flux continu
Ce qu'elle organise	l'équipe et le produit (rôles, événements)	les pratiques techniques	le flux de tâches
Point fort	cadre clair, retours client réguliers	qualité et fiabilité du code	simplicité, réactivité
Contexte typique	nouveau produit aux besoins évolutifs	code critique qui change souvent, client disponible	maintenance, support

Ces méthodes se **combinent** souvent : Scrum pour s'organiser, pratiques XP pour coder, tableau Kanban pour suivre.

Gain observé : agile ou cascade ?



- Selon le Standish Group, les projets agiles échouent **deux fois moins** que les projets en cascade.

Esprit critique

- Chiffres **déclaratifs**, issus d'enquêtes.
- Les projets agiles sont souvent plus **petits**.
- Corrélation n'est pas causalité.

Processus Unifié (*Unified Process*, UP)

Processus Unifié

Le processus unifié (PU), ou « unified process (UP) » en anglais, ou « Unified Software Development Process (USDP) », est une famille de méthodes de développement de logiciels orientés objets. Elle se caractérise par une démarche itérative et incrémentale, pilotée par les cas d'utilisation, et centrée sur l'architecture et les modèles UML. (Wikipédia)

Itératif et incrémental

Le logiciel est construit par étapes successives.

Piloté par les cas d'utilisation

Les besoins des utilisateurs guident tout le projet.

Centré sur l'architecture

L'architecture est définie tôt et guide le développement.

Processus Unifié (PU)

Définition : Le Processus Unifié (PU), ou *Unified Process*, est une méthode de développement logiciel :

- **itérative et incrémentale** : le logiciel est construit par étapes successives ;
- **orientée objet** et centrée sur les **cas d'utilisation** (UML) ;
- **centrée sur l'architecture** : l'architecture logicielle guide le développement.

Les 4 phases principales du PU :

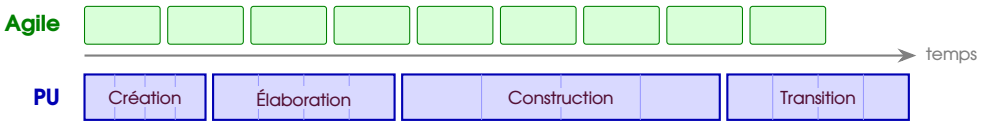
1. Création	2. Élaboration	3. Construction	4. Transition
définir la vision, objectifs et périmètre	valider la faisabilité, définir l'architecture	développer et tester les fonctionnalités	livrer, former les utilisateurs, corriger les derniers défauts

Chaque phase est divisée en **itérations**, livrant progressivement un logiciel de plus en plus complet.

Agile vs Processus Unifié : points communs

Points communs :

- Développement **itératif et incrémental**.
- Livraison progressive du logiciel.
- Importance de la collaboration avec le client.



Les deux avancent par itérations. Le PU ajoute des **phases** et des jalons au-dessus des itérations.

Agile vs Processus Unifié : différences

Différences principales :

Méthodes agiles

- Cycles très courts (**1 à 4 semaines**).
- Peu de documentation, priorité au **logiciel opérationnel**.
- Adaptation rapide aux changements.
- Auto-organisation des équipes.
- Exemples : Scrum, XP, Kanban.

Processus Unifié (UP)

- Cycles plus longs et plus **formalisés**.
- Documentation UML importante (cas d'utilisation, diagrammes).
- Planification plus structurée.
- Conçu pour les projets complexes et de grande taille.
- Basé sur 4 phases : Création, Élaboration, Construction, Transition.

Livrable

Définition : livrable

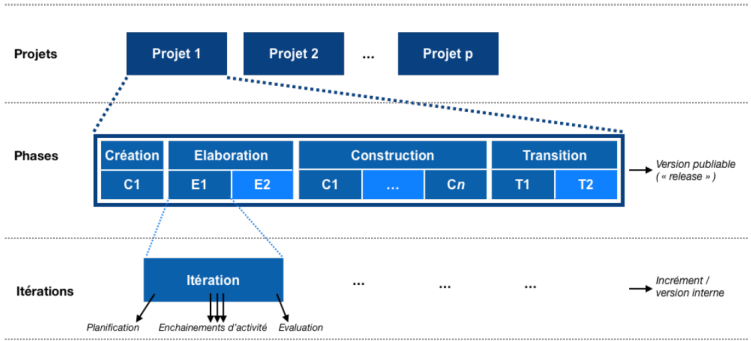
- un **corps de code source** réparti sur plusieurs composants pouvant être compilés et exécutés ;
- **un manuel** (et/ou une documentation technique) ;
- **les produits associés** (ou dépendances).

Il doit prendre en compte les besoins des utilisateurs et de tous les intervenants (tous ceux amenés à l'exploiter). Il comprend donc :

- Un recueil des besoins
- Les cas d'utilisation (besoins fonctionnels)
- Les spécifications non fonctionnelles
- Les cas de test

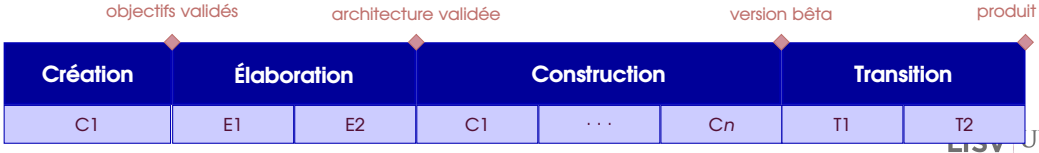
Processus Unifié

Le Processus Unifié est un enchaînement de cycles :



Principes des cycles

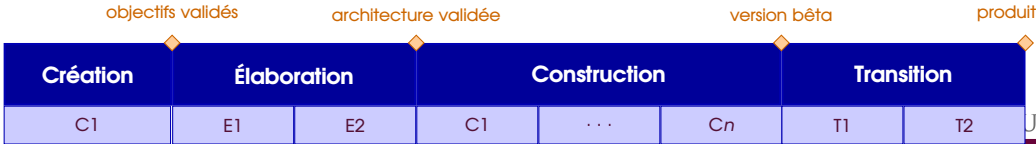
- À chaque cycle correspond une nouvelle version du logiciel (schéma précédent : un projet par version).
- Un cycle est composé de 4 phases :
  1. **Étude préliminaire / Création (*Inception*)**
  2. **Élaboration**
  3. **Construction**
  4. **Transition**
- Chaque phase se divise en itérations.



♦ Chaque phase se termine par un **jalon** : on décide de continuer, d'ajuster ou d'arrêter.

Principes des cycles

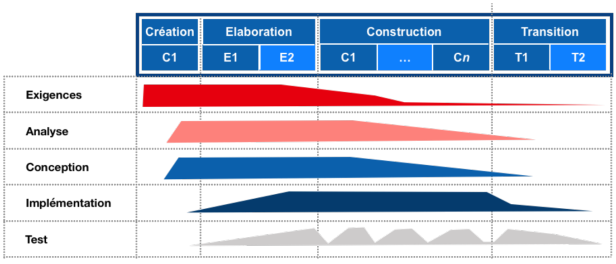
- À chaque cycle correspond une nouvelle version du logiciel (schéma précédent : un projet par version).
- Un cycle est composé de 4 phases :
  1. **Étude préliminaire / Création (*Inception*)** : vision, acteurs, périmètre, cas d'utilisation majeurs.
  2. **Élaboration** : définition de l'architecture, validation des risques.
  3. **Construction** : développement incrémental, tests réguliers.
  4. **Transition** : livraison, formation des utilisateurs, corrections.
- Chaque phase se divise en itérations.



♦ Chaque phase se termine par un **jalon** : on décide de continuer, d'ajuster ou d'arrêter.

Cycle

Un cycle := phases composées d'itérations, chaque itération est composée d'activités :

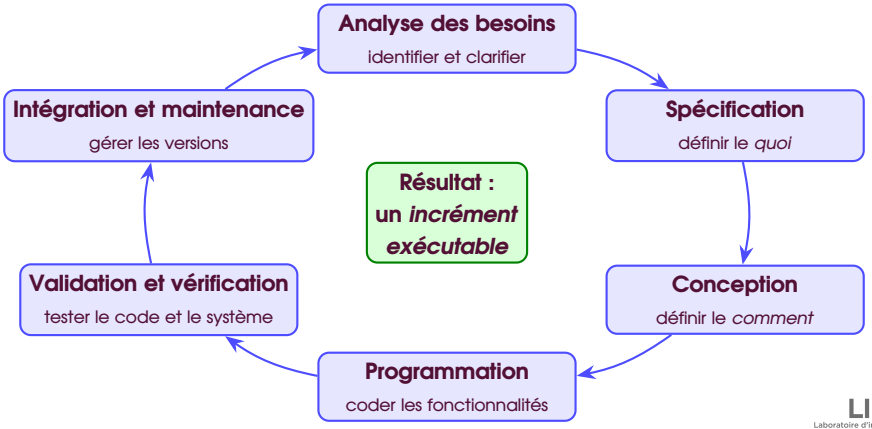


- Toutes les activités ont lieu dans **toutes les phases**, avec une **intensité variable**.
- Exigences et analyse : surtout au début.
- Implémentation : surtout en construction.
- Tests : **à chaque itération**.

Différence avec la cascade : on ne « termine » pas l'analyse avant de coder.

Une itération dans le PU

Chaque itération suit les activités classiques du génie logiciel :



Cas d'utilisation dans le PU

Les cas d'utilisation interviennent à toutes les phases du PU :

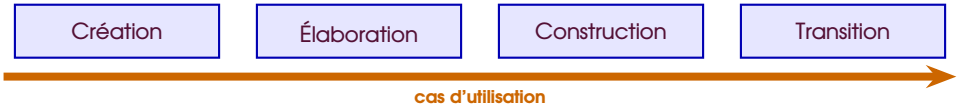
Phase	**Ce qu'on fait des cas d'utilisation**	**Pour. . .**
Création	Identifier les acteurs et les cas d'utilisation principaux	cadrer le projet
Élaboration	Raffiner les cas critiques, détailler les scénarios	guider l'architecture
Construction	Implémenter les cas d'utilisation priorisés	produire des incréments fonctionnels
Transition	Utiliser les cas d'utilisation comme base des **tests d'acceptation**	valider le système

Les cas d'utilisation sont le **fil conducteur du Processus Unifié**.

Cas d'utilisation dans le PU

Rôle central :

- Le PU est **piloté par les cas d'utilisation**.
- Les cas d'utilisation décrivent les **besoins fonctionnels du système**.
- Ils servent de **fil conducteur** tout au long du projet :
  - Création → identification des cas d'utilisation principaux.
  - Élaboration → détailler les cas critiques et guider l'architecture.
  - Construction → implémenter les cas d'utilisation priorisés.
  - Transition → valider le système via les scénarios utilisateurs.



Synthèse : choisir sa démarche

Si le projet. . .	**on s'oriente vers. . .**
est court, avec un besoin parfaitement connu et stable	la **cascade**
est critique, avec des exigences figées et une certification à obtenir	le **cycle en V**
est grand et innovant, avec des risques techniques ou fonctionnels importants	la **spirale**
peut livrer un noyau utile, puis des fonctionnalités prévues à l'avance	l'**incrémental**
a des besoins flous ou changeants et un client disponible pour réagir	**Scrum**
exige un code très fiable qui change souvent, avec le client sur place	**XP**
consiste à traiter un flux continu de demandes imprévisibles	**Kanban**
est de grande taille, avec plusieurs équipes et une architecture critique	le **Processus Unifié**

Il n'y a pas de « meilleure » méthode : les projets réels **combinent** souvent plusieurs approches.

À retenir

- Un projet sert un **système d'information** : il faut comprendre l'organisation, le **client** et les **utilisateurs**.
- Activités du génie logiciel : analyse des besoins, spécification (**quoi**), conception, programmation (**comment**), validation et vérification, intégration.
- **Vérifier** : le produit est-il correct ? **Valider** : est-ce le bon produit ?
- Plus une erreur d'analyse est découverte tard, plus elle coûte cher.
- Le **cycle de vie** fixe l'ordre des activités : cascade, V, incrémental, spirale (pilotée par les risques).
- Le **prototypage** fait apparaître le vrai besoin tôt, avant de construire.
- Les méthodes **agiles** (Scrum, XP, Kanban) et le **PU** livrent par itérations et montrent le produit tôt.
- Le choix de la démarche dépend du projet : stabilité des besoins, criticité, taille, disponibilité du client.

Prochain cours

Cours 2 : recueil et analyse des besoins

Nous verrons :

- Le **cahier des charges** : rôle, structure, lecture.
- Comment identifier les **acteurs** et leurs interactions.
- Exigences **fonctionnelles** et **non fonctionnelles**.
- Une introduction aux **cas d'utilisation**.

Puis, Cours 3 : modéliser les cas d'utilisation (scénarios, diagrammes UML), notre **point d'entrée** vers l'analyse et la conception UML.

En TD 1

Analyser **PrêtMat**, un projet étudiant rendu à temps. . . mais que personne n'utilise.