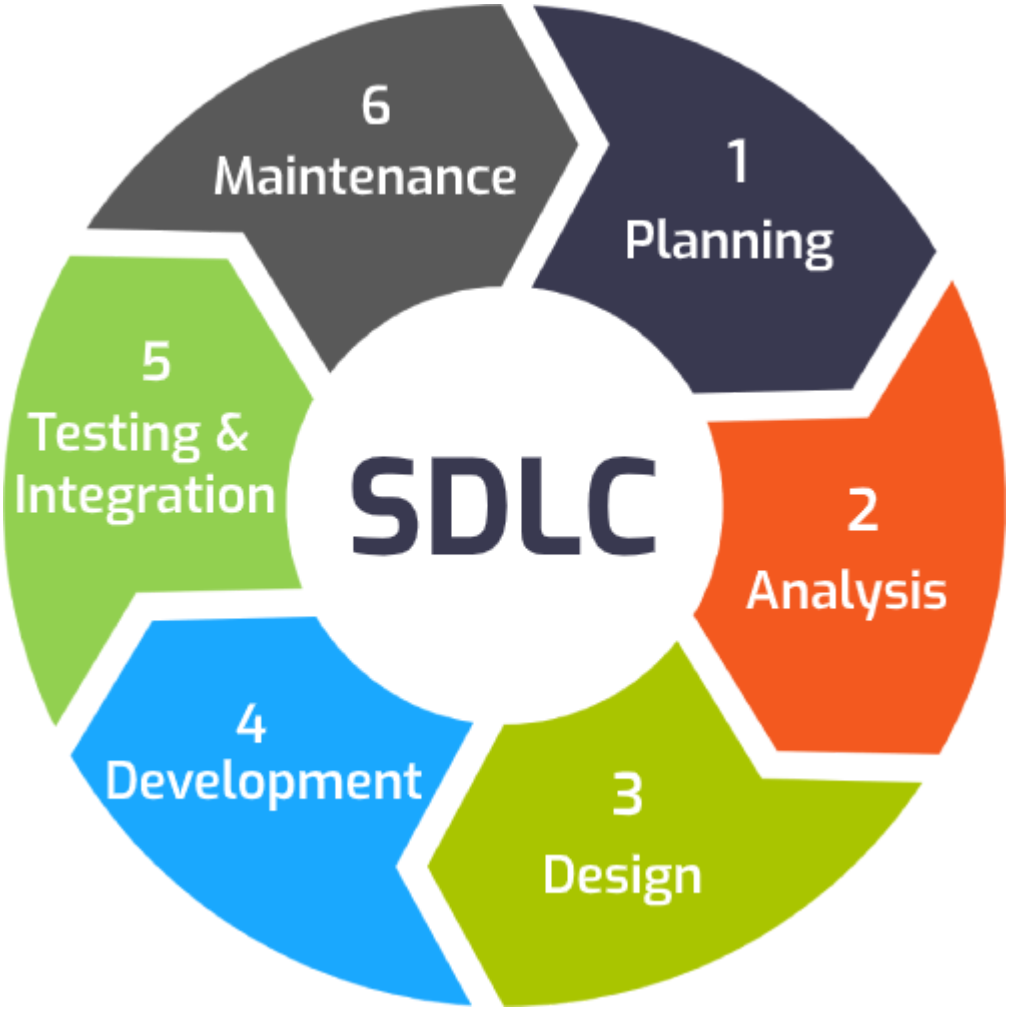
DA6 - Cybersécurité

TD 1 - Méthodes de développement -Mémento

I. Pourquoi suivre des méthodes de développement ?

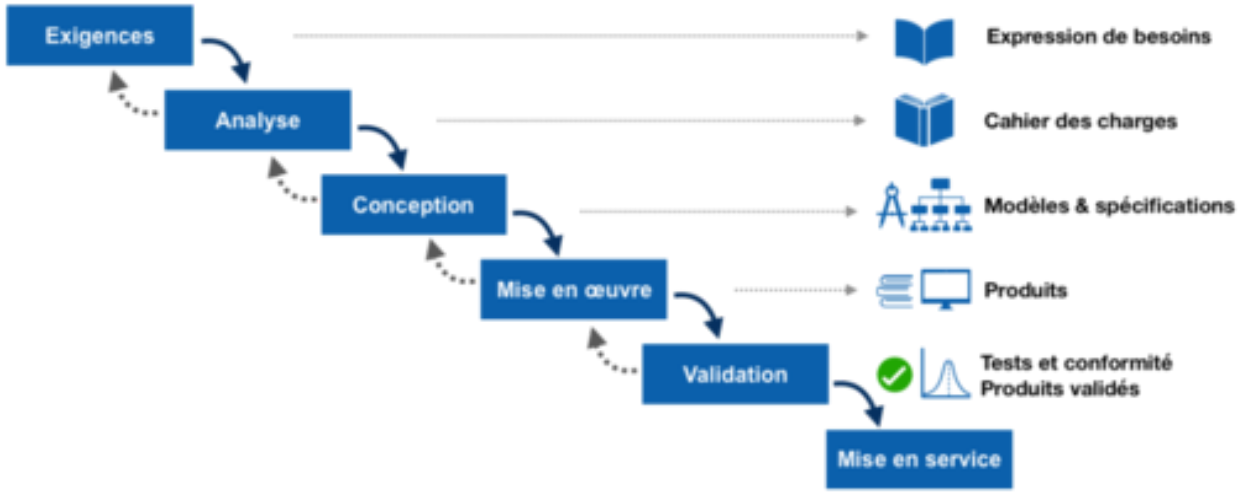
- Suivre le projet, savoir où on en est, combien ça coûte/va coûter
- Communiquer
- Normaliser, unifier, ordonner
- Augmente la sûreté

II. Le SDLC

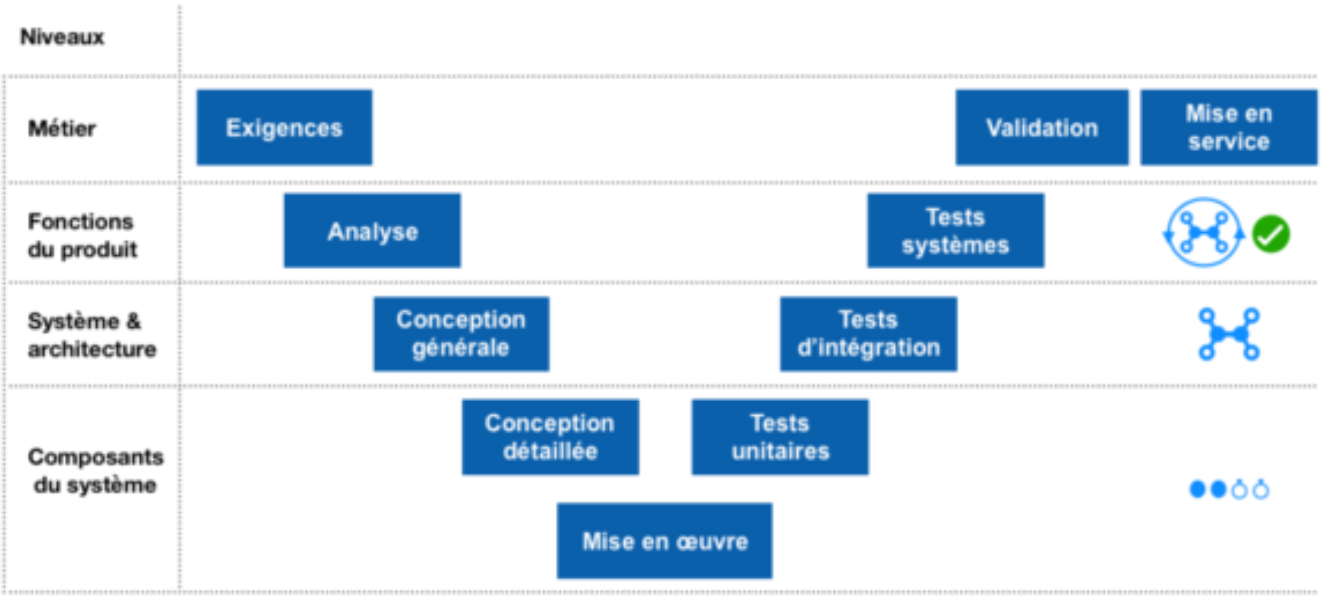


III. Méthodes classiques

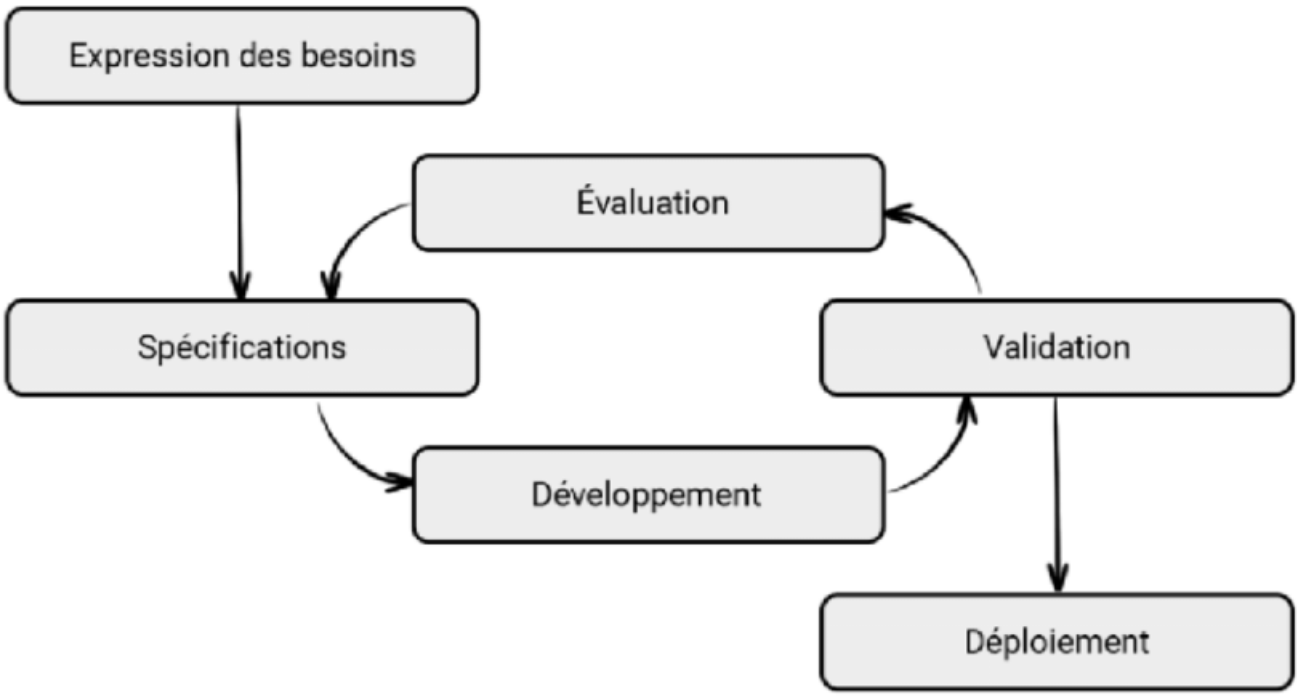
A. Waterfall



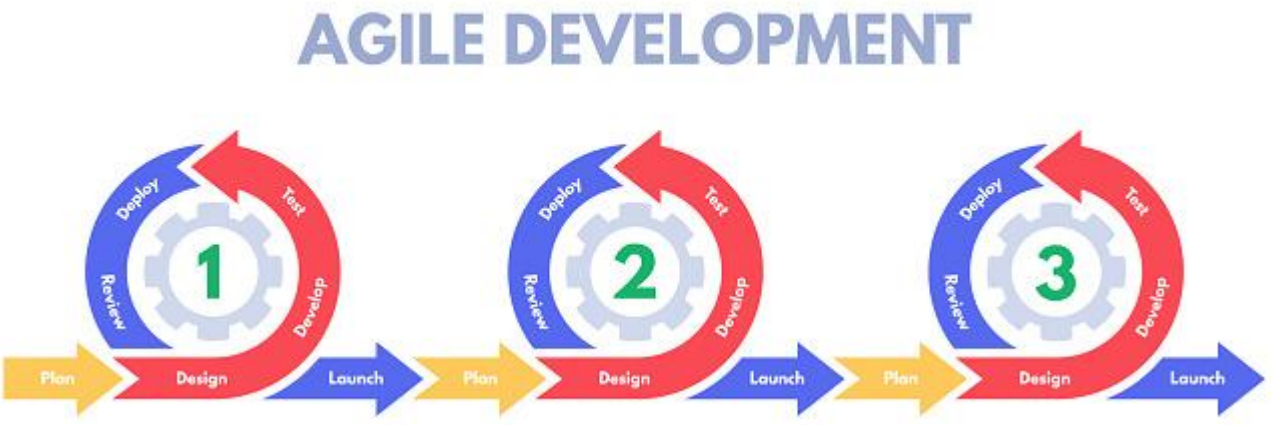
B. Cycle en V



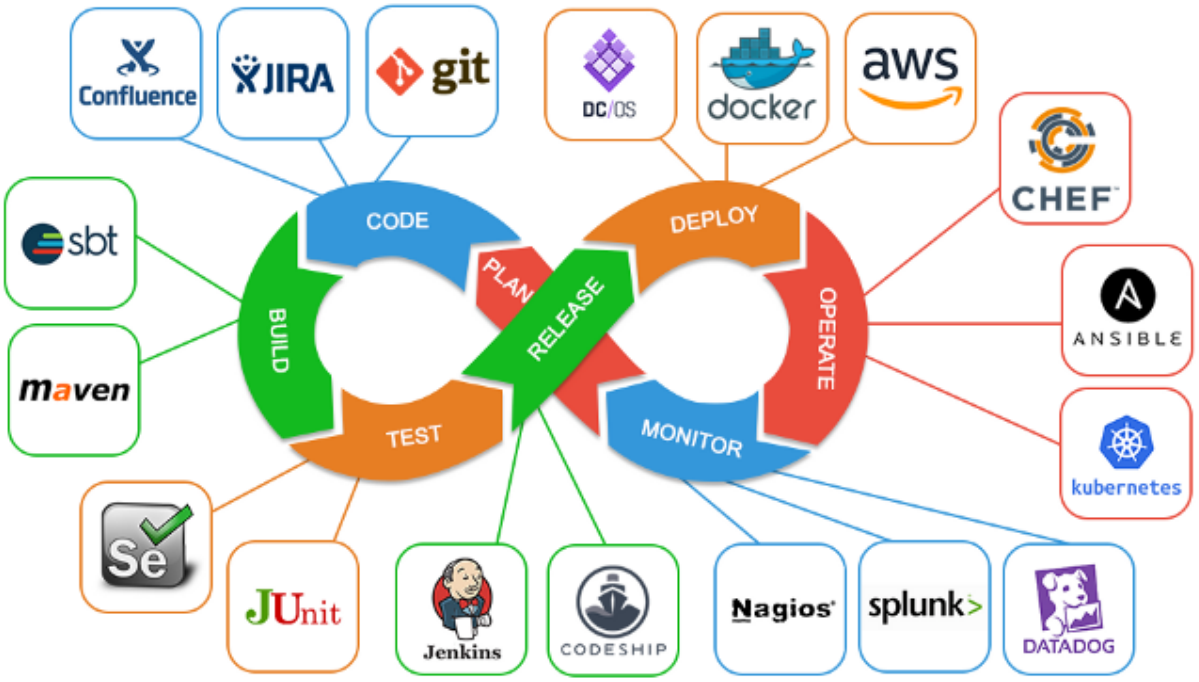
IV. Méthodes itératives



V. Méthodes agiles



VI. DevOps



La philosophie du DevOps est basée sur le modèle CAMS :

1. **C**ulture : Vision a su développer une culture du code propre, du partage et de l'efficacité. L'équipe a un jargon partagé et documenté pour que de nouveaux acteurs puissent rejoindre l'équipe avec simplicité
2. **A**utomatisation : toute la chaîne de développement est automatisée, fonctionnant en Livraison Continue : le code est versionné à l'aide de Git, chaque modification est envoyée sur un repository Gitlab privé, ce qui déclenche l'exécution de la chaîne d'intégration/livraison -> les tests unitaires sont exécutés, puis les tests d'intégration, puis l'application est déployée sur les serveurs de tests où sont effectués les tests fonctionnels *(automatisés au maximum à l'aide de frameworks comme Selenium)*. Si tous les tests passent, l'application est alors déployée en recette pour permettre au client et aux testeurs humains de l'essayer.
3. **M**esure/Surveillance (**M**onitoring) : chaque étape du cycle de vie logicielle génère des indicateurs/mesures qui sont regroupés au sein de tableaux de bords automatisés qui peuvent créer des alertes. Cette approche permet de mieux conduire le développement et l'architecture de l'application *(car on identifie les potentiels problèmes de performance)*, et permet aussi une meilleure sécurité et sûreté *(l'équipe Exploitation/Ops sait ce qui se passe à chaque moment et est plus réactive en cas de soucis)*
4. Partage (**S**haring) : l'équipe est dans un processus d'amélioration continue, chacun peut apporter de nouvelles idées et proposer des changements. L'application et les processus l'entourant sont documentés, la communication est claire et centralisée *(utilisation d'outils comme Slack)*

VII. Test Driven Development (TDD)

Qu'est-ce ? Le Test Driven Development, ou Développement Dirigé par les Tests (TDD) est une méthode de développement logiciel où on commence par développer les tests (généralement unitaires) avant de développer le code de l'application. Ensuite, pendant le développement, on exécutera régulièrement les tests pour vérifier si le code développé est correct ou non.

Pourquoi ? Plusieurs raisons : on augmente la qualité du code en s'assurant que le code est correct. Le but des tests est de révéler des incorrections dans le code et de chercher de possibles bugs là où on pense qu'ils pourraient arriver (mais ça ne veut pas dire que l'on détecte tout). On s'assure aussi que le code respecte correctement les spécifications : les tests permettent de vérifier que les spécifications formalisables (*que l'on peut écrire sous forme de tests automatisés*) sont respectées. On s'assure aussi que les tests fonctionnent : on commence par lancer les tests à vide, pour vérifier qu'ils échouent. Si certains tests n'échouaient pas, ça voudrait dire qu'ils sont incorrects (vu que l'application n'existe pas encore).

Comment ? On écrit des tests automatisés (tests unitaires pour les fonctions, tests d'intégration pour les classes/modules, tests fonctionnels pour vérifier l'implémentation d'une fonctionnalité et les interfaces graphiques), puis on code. Parfois fait en ***pair programming*** : une équipe de deux développeurs, l'un écrivant les tests et l'autre le code. A chaque push sur le dépôt du gestionnaire de versions, un outil d'intégration continue fera passer les tests. S'ils échouent, le push sera refusé et il faudra corriger le code.

VIII. DevSecOps

SCA ou Software Composition Analysis

Cet outil permet de vérifier la composition d'une application en termes de dépendances tierces et de licences. Comme les applications embarquent généralement des frameworks, des librairies Open Source ou propriétaires entre autres, il est nécessaire de vérifier que l'artefact résultant n'embarque pas des **composants connus comme vulnérables ou obsolètes** mais également qu'il n'y ait pas de **défaut de compatibilité de licences**.

L'analyse se fera au moment de la phase de Build sur l'artefact généré dans un repository ou non, ou via la configuration des dépendances de l'application ou pour certains via l'analyse des binaires.

- **Avantages:** Détecte les composants embarqués vulnérables, Permet une cartographie des librairies et des licences, Se met à jour très régulièrement en terme de base de connaissances de vulnérabilités

- **Inconvénients:** Peut générer des faux positifs en fonction de l'outil

- Quelques exemples d'outils (liste non exhaustive) : OWASP dependency-check, BlackDuck, Jfrog XRAY, Sonatype, NPM Audit, CAST Highlight et toujours plus sur la page dédiée de l'OWASP.

SAST (Static Analysis Software Testing) : Outil de tests statiques sur le code en **boîte blanche**. Les failles de sécurité et bugs courant et visibles dans le code (injections SQL, possible faille XSS, etc.) sont détectés automatiquement et indiqués pour être corrigés. Un SAST ne **peut pas détecter toutes les failles** : aucun outil capable de le faire n'existe. Mais certaines failles courantes et classiques qui auraient pu échapper à l'attention des développeurs peuvent ainsi être détectés et corrigés.

DAST (Dynamic Analysis Software Testing) : Outil de tests dynamiques sur le code en **boîte noire**. L'application compilée est exécutée, puis des payloads malicieux sont injectés partout où l'application permet des entrées (champs textes, requêtes HTTP, ports ouverts, etc.). Différentes méthodes sont utilisées : payloads connus, fuzzing, etc. Les résultats sont ensuite analysés. Ces tests peuvent être automatiques et basés sur des listes prédéfinies (injections SQL ou XSS courantes, exploits connus pour divers protocoles, etc.) ou développés sur mesure. Si des failles sont détectées, on peut ainsi les corriger.

La démarche *Security Champion* : cette démarche est simple : dans chaque équipe de développement, un (ou plusieurs) développeur intéressé par la sécurité sera nommé le Security Champion (*Paladin de la sécurité*). En contact avec le RSSI (*Responsable de la Sécurité du Système d'Informations*) et ses assistants, il aura pour rôle de promouvoir la sécurité auprès des autres développeurs en apportant un regard analytique et critique par rapport à la sécurité lors des revues de code, des stand-up meetings, des sprint planning, et lors d'ajout de User Stories au Product Backlog. Il pourra aussi animer des ateliers sécurité, diffuser de la documentation, etc. Il peut y avoir plusieurs Security Champions dans une organisation, qui peuvent travailler entre eux pour créer une culture de la sécurité au sein de l'organisation.

La démarche *Adopt an Evil User* : cette démarche consiste à avoir un utilisateur pratiquant la sécurité offensive (*forme de pentesting*) lors des phases de tests d'acceptation/recette de l'application. Contrairement aux autres utilisateurs qui vérifient que les fonctionnalités de l'application sont acceptables et fonctionnelles, cet utilisateur cherche à attaquer l'application et à détourner son utilisation, comme le ferait un utilisateur malveillant. Cet utilisateur doit avoir au moins une formation minimale en cybersécurité offensive, pour qu'il sache utiliser des outils d'attaque (*type Kali Linux, DAST, etc.*) et pour qu'il sache où et comment chercher à attaquer l'application. Cette forme de pentesting constant permet de révéler comment l'application se comporte en situation d'attaque et de voir comment améliorer sa sécurité.