Qualité Logicielle

Dr Bassirou NGOM
Master MISID
Université Amadou Mahtar Mbow

Plan

- Qui suis-je ?
- Chp1: Qualité logicielle
- Chp2: Processus de développement logiciel
- Chp3: Métriques de Qualité Logicielle et Analyse du Code
- Chp4: Tests Logiciel
- Chp5: Gestion de la documentation

Qui suis-je ?

- Dr Bassirou NGOM
  - Docteur Informatique
  - Développeur fullstack > 7
  - Architecte logiciel
  - Consultant SFR
  - mon github

Chp1: Qualité logicielle

Dr Bassirou NGOM

Master MISID

Université Amadou Mahtar Mbow

Objectifs du Chapitre

- À la fin de ce chapitre, vous serez capable de :
  - Définir la qualité logicielle et comprendre ses enjeux.
  - Connaître les principaux modèles et normes de qualité logicielle.
  - Identifier les critères de qualité d’un logiciel.
  - Comprendre pourquoi la qualité doit être intégrée dès les premières étapes du développement.

Qualité logicielle : Définitions

- **Qualité** : l'ensemble des caractèrisques d'une entité qui lui contèrent l'aptitude à satisfaire des besoins exprimés et implicités.
- **Entité** : Tout ce « qui peut etre décrit et considéré individuellement »: produit, processus, organisme ou combinaison des 3 (src: norme NF EN ISO 9000:2000)
- **Qualité logiciel**: « ensemble des traits et caractèristiques d'un produit logiciel portant sur son aptitude à satisfaire des besoins exprimés et implicites » (src : norme ISO/CEI 9126:1991)

Définition de la Qualité Logicielle

- La qualité logicielle désigne l'ensemble des caractéristiques qui rendent un logiciel **fiable**, **performant**, **sécurisé** et **maintenable**.

- Elle couvre à la fois la qualité interne (code, architecture) et la qualité externe (expérience utilisateur, conformité aux exigences).

- Un logiciel de qualité répond aux attentes des utilisateurs et minimise les risques de défauts.

Pourquoi la Qualité Logicielle est-elle Cruciale ?

1. Réduction des coûts de maintenance et des corrections de bugs.
2. Amélioration de la satisfaction des utilisateurs.
3. Renforcement de la sécurité et réduction des risques de cyberattaques.
4. Conformité aux réglementations et normes industrielles.
5. Augmentation de la compétitivité sur le marché.

Les Modèles de Qualité Logicielle

- Les modèles de qualité définissent les critères permettant d'évaluer un logiciel.
- Les principaux modèles sont :

  - **ISO 9126 / ISO 25010**: Modèle normatif de la qualité logicielle.
  - **CMMI (Capability Maturity Model Integration)**: Évaluation des processus de développement.
  - **TMMi (Test Maturity Model Integration):** Modèle d'évaluation de la maturité des tests logiciels.
  - **ITIL (Information Technology Infrastructure Library)**: Bonnes pratiques pour la gestion des services informatiques.

Les Critères de Qualité Logicielle (ISO 25010)

- L'ISO 25010 définit les caractéristiques clés de la qualité logicielle :

  - **Fonctionnalité**: Le logiciel répond-il aux besoins des utilisateurs ?
  - **Fiabilité**: Le logiciel fonctionne-t-il sans erreurs sur une longue période ?
  - **Performance** : Est-il rapide et efficace en utilisation ?
  - **Sécurité** : Protège-t-il les données et résiste-t-il aux attaques ?
  - **Maintenabilité** : Le code est-il facile à comprendre et à modifier ?
  - **Portabilité**: Le logiciel est-il adaptable à différents environnements ?

Qualité Interne vs. Qualité Externe

- **Qualité Interne**:
  - Lisibilité et modularité du code.
  - Bonne structuration et documentation.
  - Utilisation de design patterns et bonnes pratiques.

- **Qualité Externe**:
  - Expérience utilisateur fluide et intuitive.
  - Bonne performance et sécurité.
  - Respect des exigences fonctionnelles et techniques.

Intégration de la Qualité Dès la Conception

- La qualité ne doit pas être un ajout après développement mais une démarche intégrée dès le début.
- Importance de la gestion des exigences et des spécifications précises.
- Adoption des pratiques de développement **Agile** et **DevOps** pour une amélioration continue.
- Utilisation de tests automatisés et d'outils d'analyse statique du code.

Échecs dus à des problèmes de qualité logicielle

Échecs dus à des problèmes de qualité logicielle

Échecs dus à des problèmes de qualité logicielle

📌 **Problème** :

- Windows Vista a été lancé avec des **exigences matérielles trop élevées**.
- Il était **lourd, lent et instable**, provoquant un mécontentement général.
- Manque de **compatibilité avec les anciens logiciels et pilotes matériels**.

📌 **Problèmes de qualité** :

❌ Mauvaise **performance** (lenteur, bugs)
❌ Manque de **compatibilité et de tests d'intégration**
❌ Mauvaise **gestion des exigences**

📌 **Leçons à tirer** :

✅ Tester la **performance et la compatibilité** avant le lancement.
✅ Ne pas **sous-estimer l'impact des changements** sur les utilisateurs.

Échecs dus à des problèmes de qualité logicielle

Boeing 737 Max
Un crash dû à un logiciel mal conçu (2018-2019)

📌 **Problème** :

- Boeing a développé un système appelé **MCAS** (Maneuvering Characteristics Augmentation System).
- Ce système pouvait **forcer l'avion à piquer du nez** en cas de détection d'un risque de décrochage.
- Le **logiciel était mal conçu** et **dépendait d'un seul capteur** (en cas de défaillance, l'avion plongeait).
- Boeing **n'a pas suffisamment testé le système** et **n'a pas formé les pilotes**.

📌 **Problèmes de qualité** :

❌ **Manque de tests** et de validation du système.
❌ **Mauvaise gestion des risques** logiciels.
❌ **Sécurité insuffisante** (le système pouvait provoquer un crash).

📌 **Leçons à tirer** :

✅ Tester **systématiquement les scénarios extrêmes** en aéronautique.
✅ Mettre en place une **redondance des capteurs** et éviter une dépendance unique.
✅ Ne pas **sous-estimer l'impact des changements** sur les

Échecs dus à des problèmes de qualité logicielle

Échecs dus à des problèmes de qualité logicielle

Samsung Galaxy Note 7(2016)

📌 **Problème** :

•Le **Galaxy Note 7** était censé concurrencer l'iPhone 7 avec une **grosse batterie et des performances accrues**.
•Quelques semaines après sa sortie, **des centaines d'appareils ont pris feu** à cause d'un problème de batterie.
•Samsung a essayé de corriger le problème via une **mise à jour logicielle**, mais cela n'a pas suffi.

📌 **Problèmes de qualité** :

❌ **Mauvaise gestion de la batterie** par le logiciel..
❌ **Tests de sécurité insuffisants** sur la batterie et son intégration.
❌ **Manque de plan de crise**, Samsung a d'abord nié le problème.

📌 **Leçons à tirer** :

✅ Tester **toutes les conditions extrêmes** avant de lancer un produit.
✅ Avoir un **plan de gestion de crise** clair.
✅ **Ne pas précipiter la sortie d'un produit pour concurrencer un rival**.

Chp2: Processus de développement logiciel

Dr Bassirou NGOM

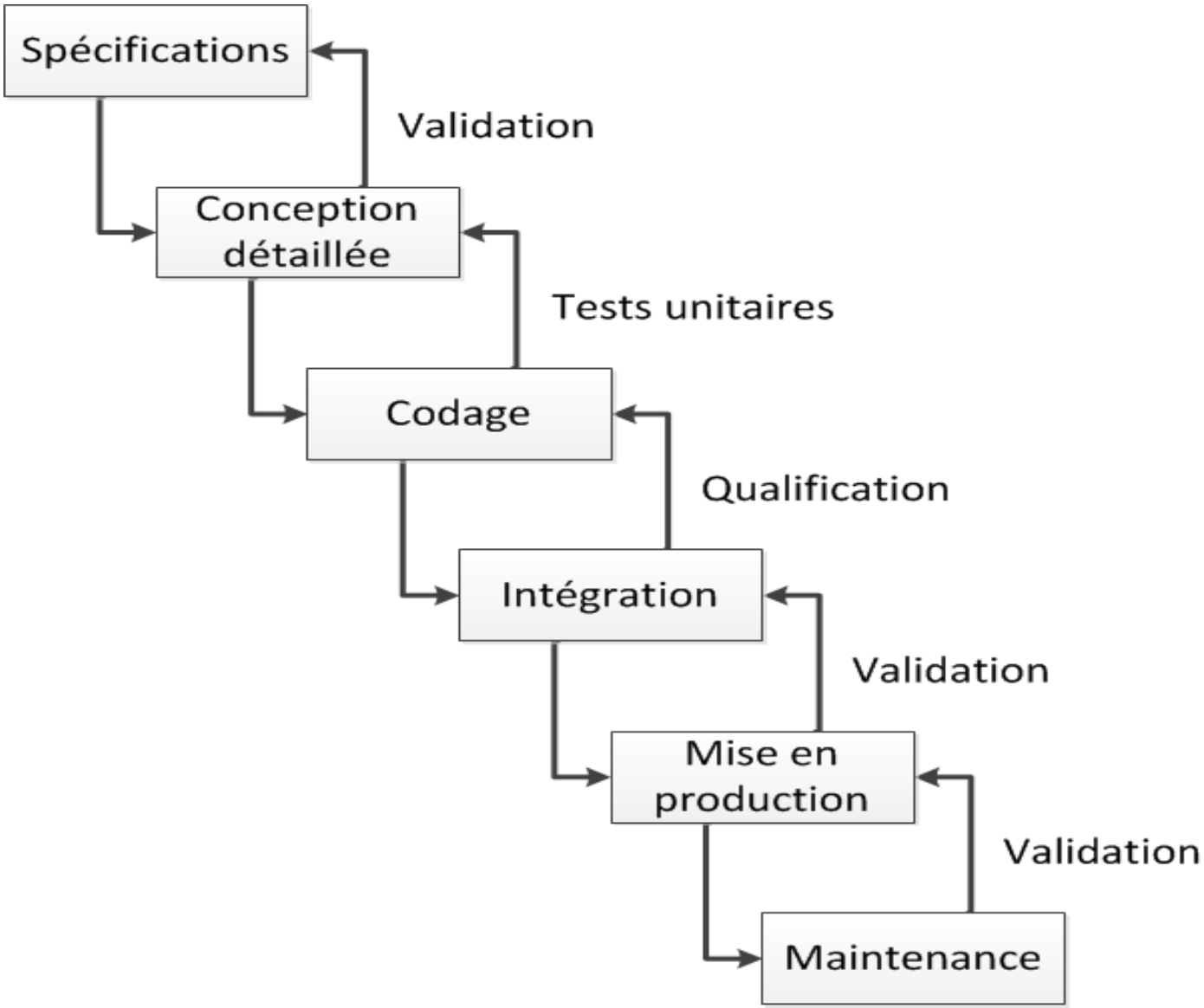
Master MISID

Université Amadou Mahtar Mbow

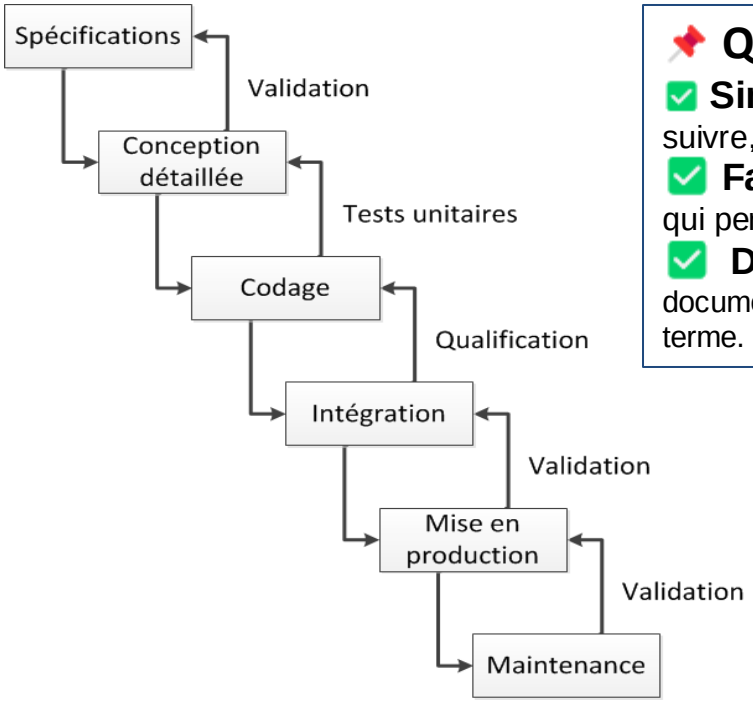
Cycle de vie logiciel

- Un logiciel est un ensemble complexe et son developpement nécssite une diversité d'activités. La modélisation de l'enchainement de ses activités constitune le cycle de vie du logiciel.
- Différents cycles :
  - ❖En cascade
  - ❖En V
  - ❖En spirale
  - ❖itératif

Modèle en cascade



Modèle en cascade



📌 **Qualités**:

✅ **Simplicité et clarté** : Le processus est simple à comprendre et à suivre, car il est linéaire et structuré.

✅ **Facilité de gestion de projet** : Les étapes sont bien définies, ce qui permet de planifier facilement les ressources et les délais.

✅ **Documentations complètes** : Chaque phase génère une documentation claire, ce qui peut être utile pour les audits ou la maintenance à long terme.

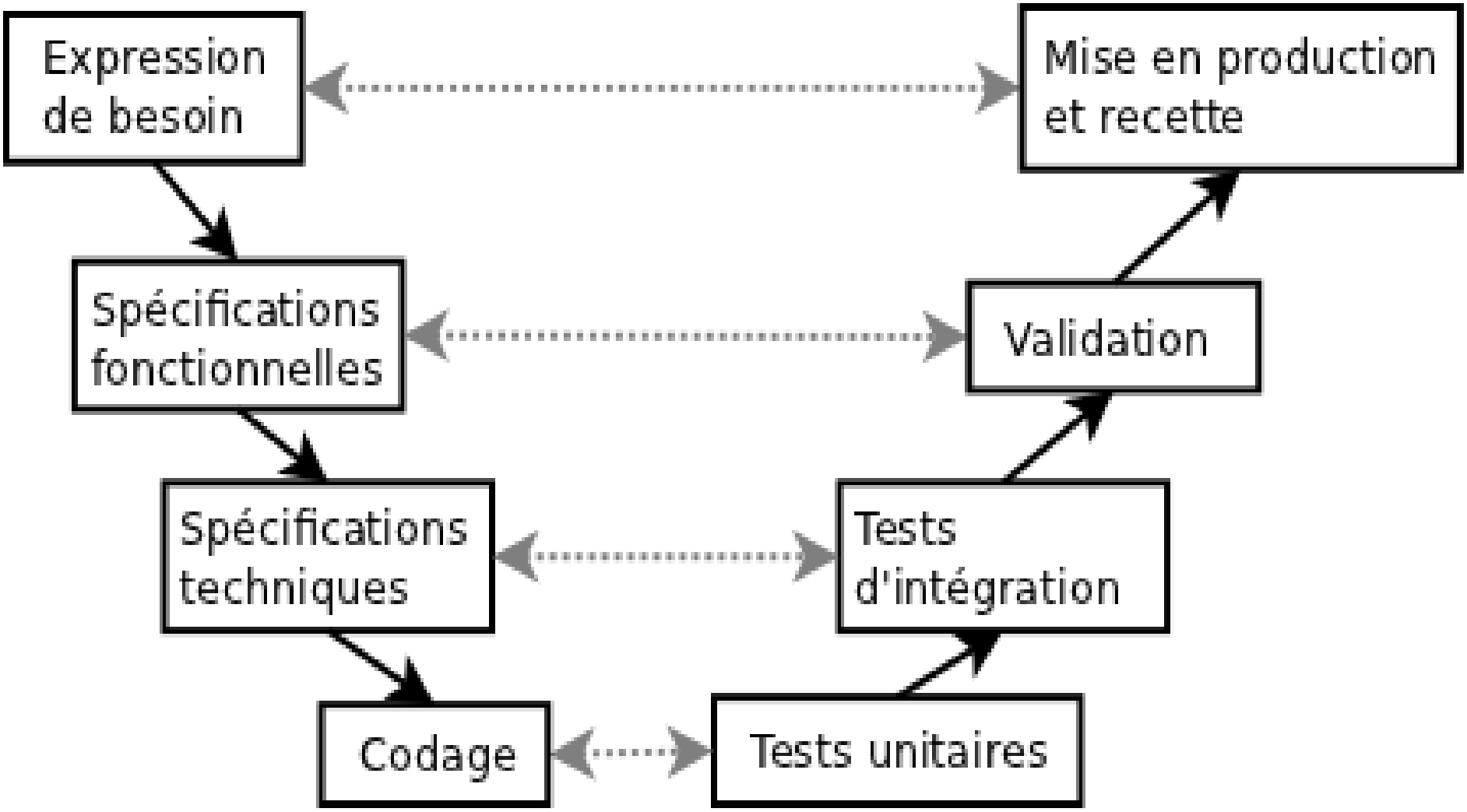
📌 **Inconvénients** :

❌ **Rigidité**: Une fois qu'une phase est terminée, il est difficile de revenir en arrière pour effectuer des changements sans perturber le projet.

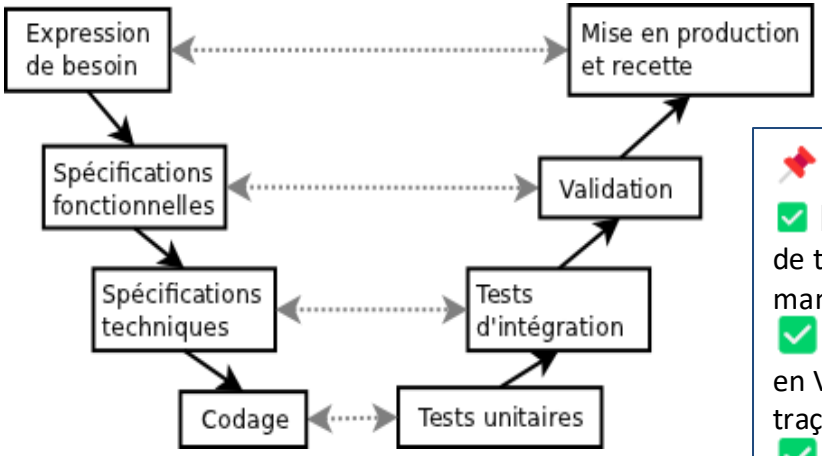
❌ **Réactivité faible aux changements** : Si des exigences évoluent en cours de développement, il devient coûteux et complexe de les intégrer.

❌ **Tests tardifs**: Les tests sont effectués après la phase de développement, ce qui peut entraîner des erreurs ou des malentendus qui ne sont pas identifiés à temps.

Modèle en V


Expression de besoin
Spécifications fonctionnelles
Spécifications techniques
Codage
Tests unitaires
Tests d'intégration
Validation
Mise en production et recette

Modèle en V



📌 **Qualités**:

✅ **Focus sur les tests** Chaque phase de développement a une phase de test associée, ce qui garantit que les tests sont planifiés et réalisés de manière rigoureuse..

✅ **Clarté et traçabilité** : Comme le modèle en cascade, le modèle en V suit une séquence linéaire et bien structurée, permettant une bonne traçabilité des exigences.

✅ **Tests plus efficaces** : Les tests sont réalisés en même temps que la conception, ce qui améliore la détection des problèmes dès le début.

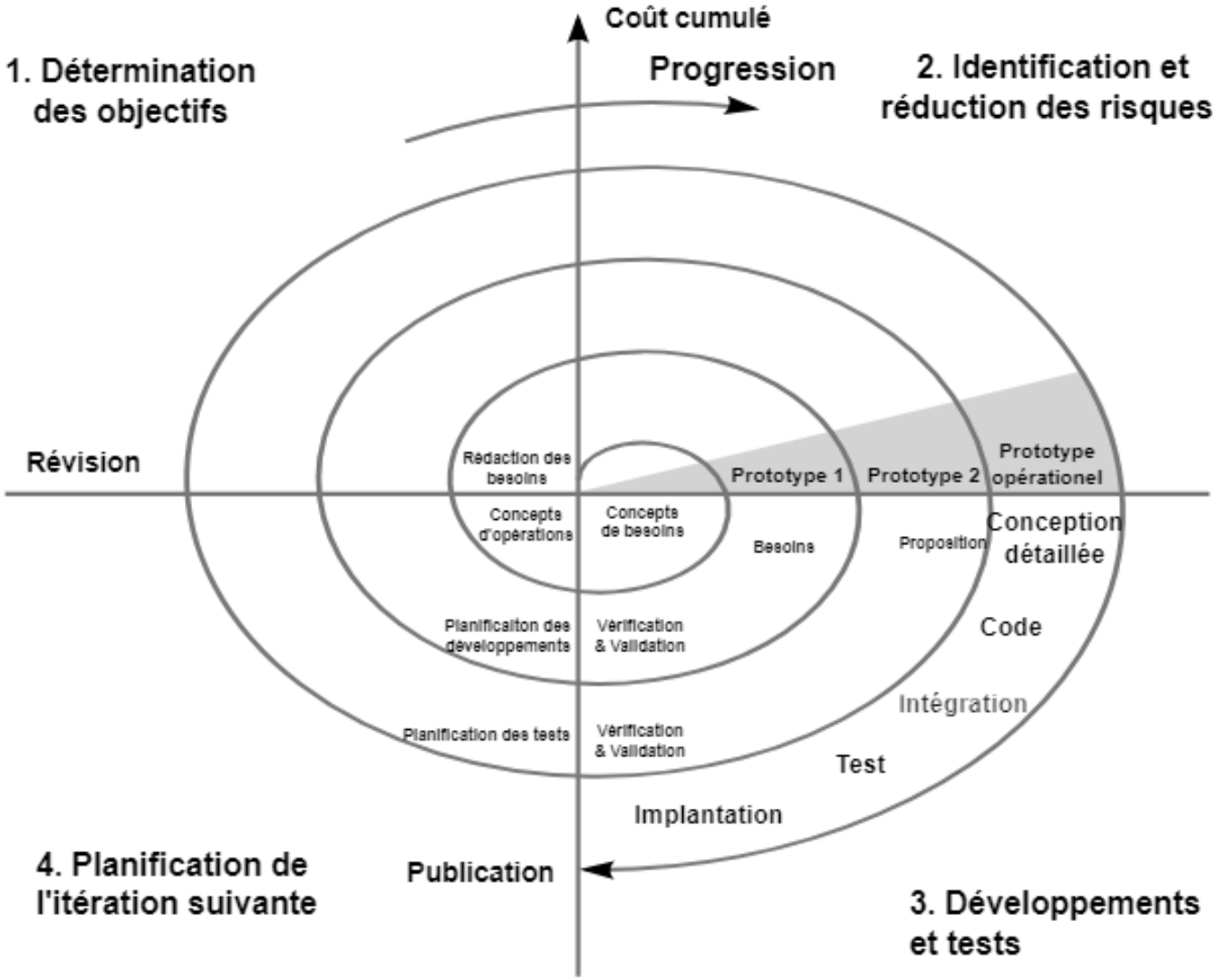
📌 **Inconvénients** :

❌ **Rigidité**: Comme le modèle en cascade, le modèle en V est difficile à adapter aux changements une fois que les étapes initiales sont terminées..

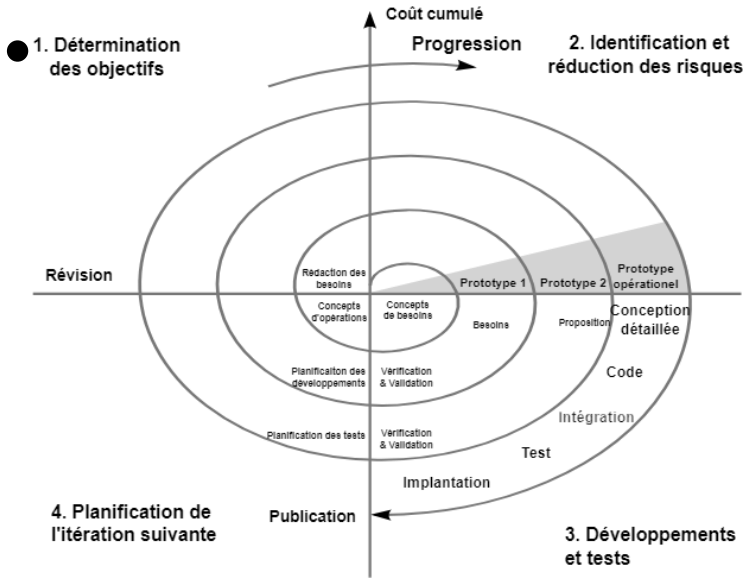
❌ **Coût élevé pour les changements**: Si des erreurs sont découvertes après une phase de développement, les coûts de correction sont élevés, car il faut revenir en arrière dans le processus.

❌ **Pas d'itération**: Le modèle ne prévoit pas de retours ou d'ajustements au fur et à mesure du développement.

Modèle en Spiral

- 

Modèle en Spiral



📌 **Qualités**:

✅ **Gestion des risques:** L'un des plus grands avantages du modèle spiral est sa gestion proactive des risques à chaque itération, ce qui permet d'adapter le projet aux incertitudes et de minimiser les risques.

✅ **Flexibilité** : Le modèle permet des ajustements et des évolutions en fonction des retours à chaque spirale.

✅ **Livraisons fréquentes** : Le produit peut être amélioré progressivement grâce à des prototypes et des versions intermédiaires.

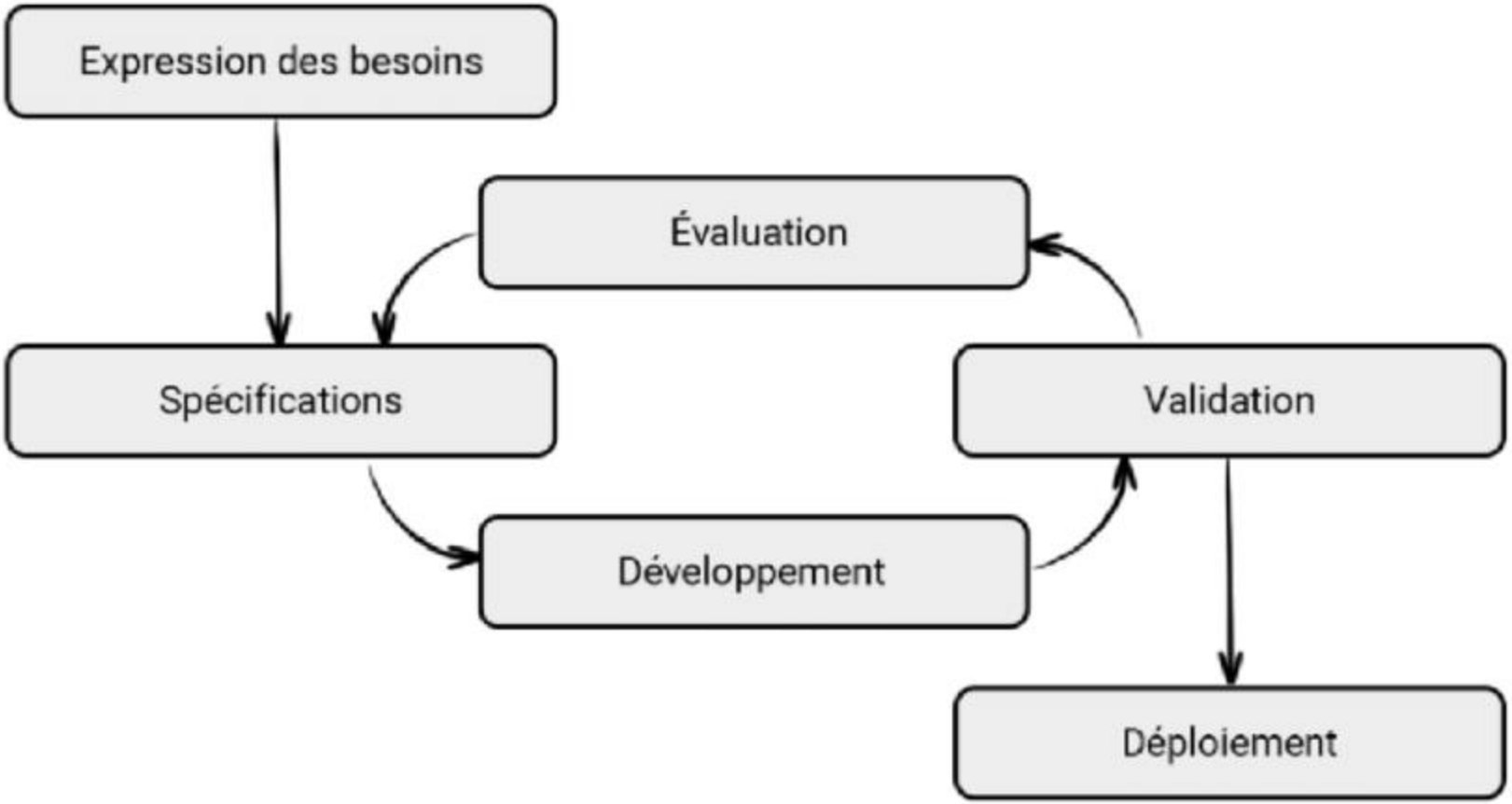
📌 **Inconvénients** :

❌ **Complexité de gestion**: Le modèle spiral peut être difficile à gérer, notamment pour les petits projets, en raison de la complexité de la planification des risques et de la gestion des itérations.

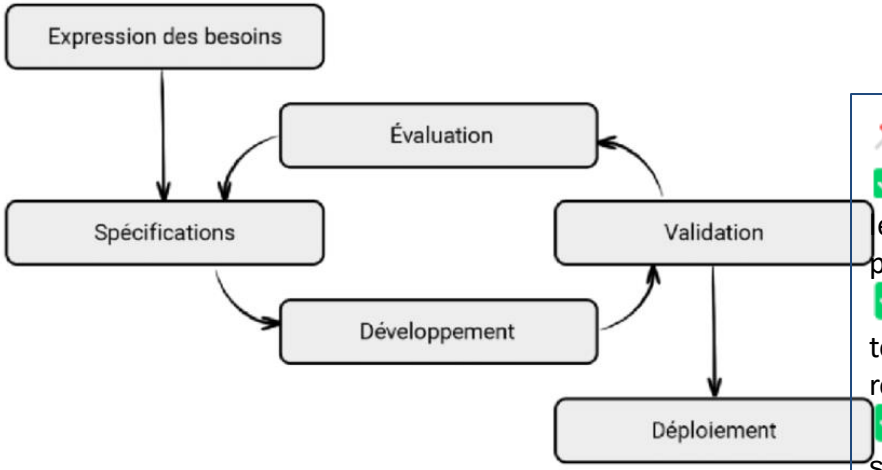
❌ **Coûts élevés**: Les cycles répétés de planification, de tests et de révisions peuvent rendre ce modèle plus coûteux que d'autres.

❌ **Exige une expertise**: La gestion des risques et la mise en œuvre du modèle spiral nécessitent des compétences avancées en gestion de projet et en analyse de risques.

Modèle Itératif



Modèle itératif



📌 **Qualités**:

✅ **Adaptabilité:** Le modèle itératif permet de réévaluer les besoins et es priorités à chaque cycle, ce qui en fait une excellente option pour les projets avec des exigences changeantes.

✅ **Livraison rapide** : Une version fonctionnelle du produit est livrée tôt dans le processus, ce qui permet aux parties prenantes de voir des résultats concrets rapidement..

✅ **Amélioration continue**: À chaque itération, les fonctionnalités sont testées, révisées et améliorées en fonction des retours.

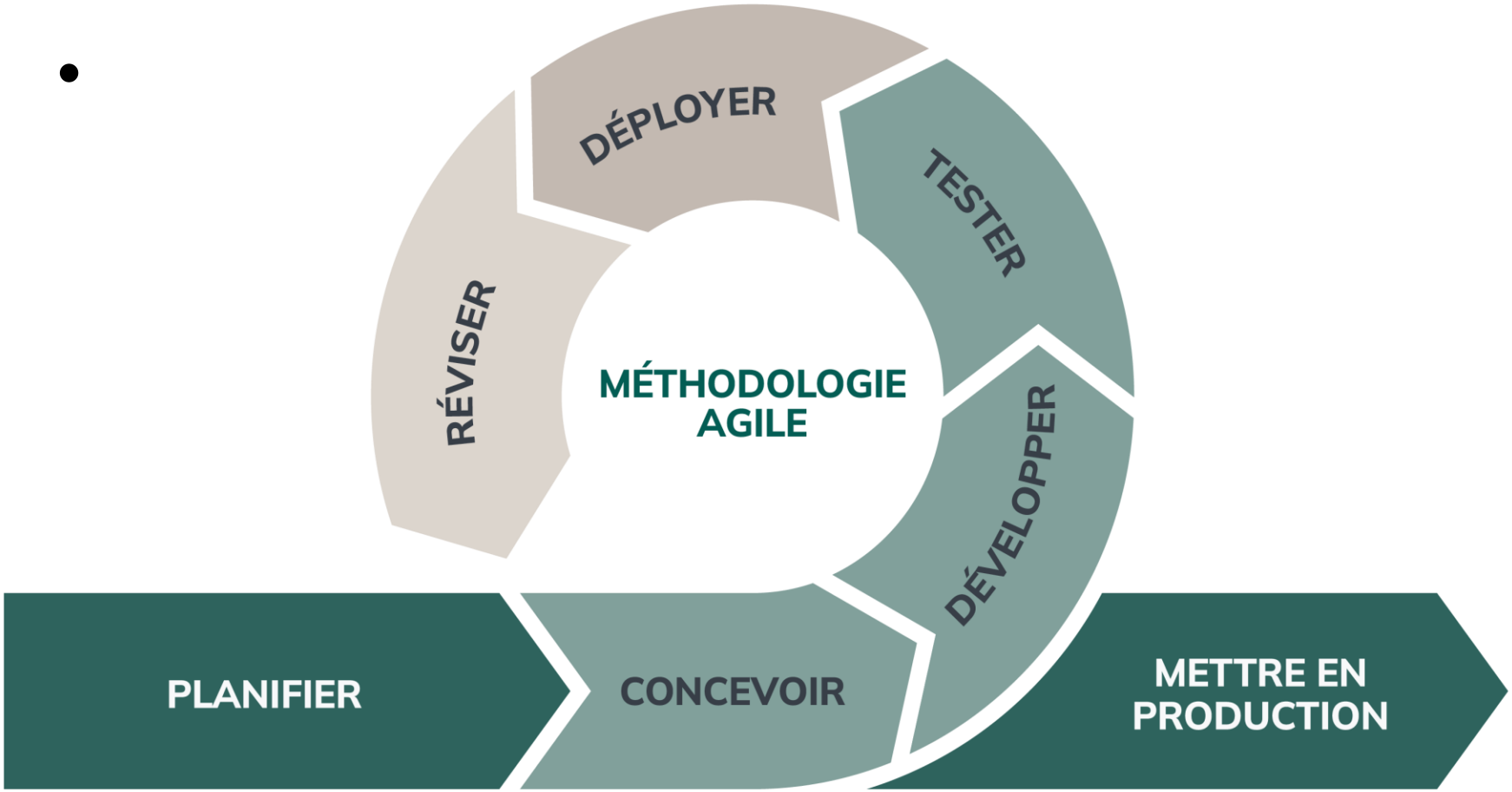
📌 **Inconvénients** :

❌ **Complexité de gestion**: Bien que le modèle soit flexible, il peut être difficile de planifier à long terme sans une vision complète du produit final.

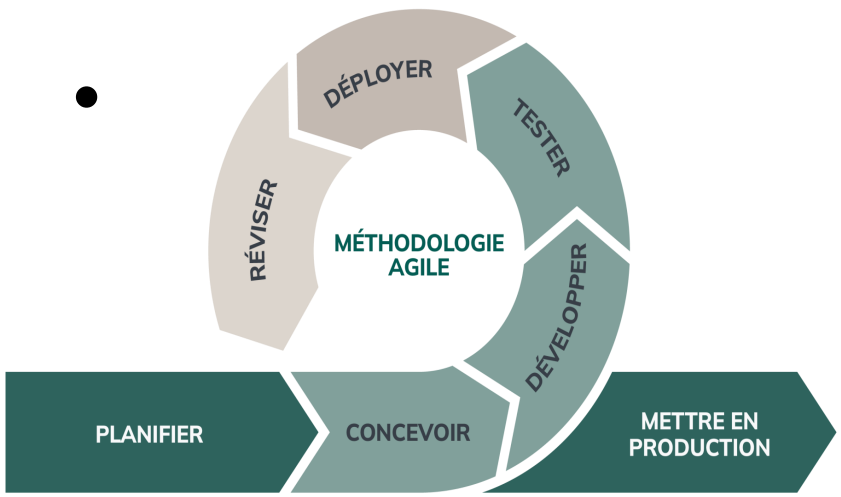
❌ **Problèmes de cohérence** : Si les itérations ne sont pas bien gérées, cela peut mener à un manque de cohérence dans l'architecture ou la conception du logiciel.

❌ **Risque de dérive des exigences**: es changements fréquents des exigences peuvent entraîner une dérive du projet (scope creep), rendant le projet plus coûteux et complexe à finaliser.

Modèle Agile

- 

Modèle Agile



📌 **Qualités**:

✅ **Flexibilité et adaptabilité:** Agile permet d'adapter facilement le développement en fonction des retours des utilisateurs et des parties prenantes.

✅ **Communication et collaboration accrues**: Agile met l'accent sur la communication entre les équipes, le client et les parties prenantes à chaque étape.

✅ **Livraisons fréquentes**: Des versions fonctionnelles du produit sont livrées à la fin de chaque sprint, permettant des retours fréquents et un produit qui s'améliore rapidement.

✅ **Amélioration continue**: Grâce aux rétrospectives à la fin de chaque sprint, l'équipe peut améliorer ses processus de développement à chaque cycle.

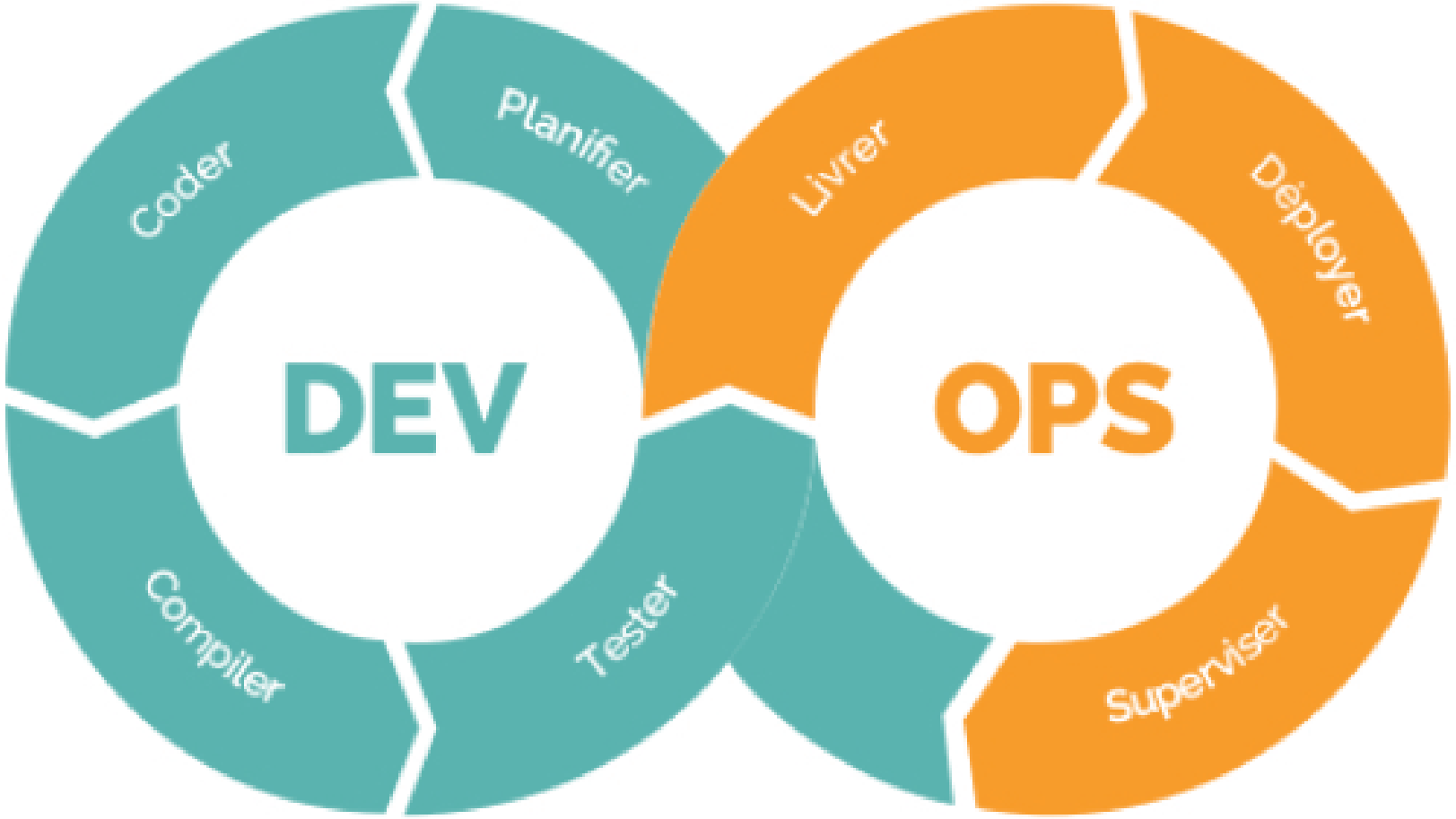
📌 **Inconvénients** :

❌ **Difficulté de gestion pour les grands projets** : Agile peut devenir difficile à gérer à grande échelle, surtout si les équipes sont nombreuses ou si la coordination avec d'autres équipes est complexe.

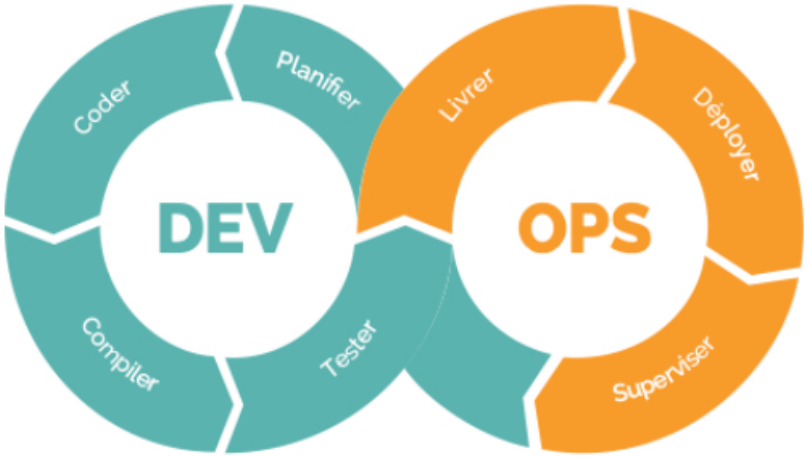
❌ **Dépendance à la coopération du client** : le modèle nécessite une forte implication du client pour fournir un feedback continu, ce qui n'est pas toujours possible.

❌ **Risque de manque de documentation**: Si l'accent est trop mis sur la livraison rapide, la documentation peut être négligée, ce qui peut poser problème à long terme.

Modèle DevOps



Modèle DevOps



📌 **Qualités**:

✅ **Automatisation et efficacité :** DevOps met l'accent sur l'automatisation des tests, de l'intégration et du déploiement, ce qui permet de livrer rapidement des mises à jour de logiciels fiables.

✅ **Collaboration accrue:** La collaboration étroite entre les équipes de développement et des opérations permet une gestion fluide et continue du cycle de vie du logiciel.

✅ **Livraison continue**: Le logiciel est continuellement mis à jour et livré en production, ce qui permet de répondre rapidement aux besoins du marché.

📌 **Inconvénients** :

❌ **Exige une forte expertise** : La mise en œuvre de DevOps nécessite des compétences techniques avancées, notamment en matière d'automatisation et de gestion de l'infrastructure.

❌ **Peut être complexe à mettre en place**: L'adoption de DevOps dans une organisation peut être difficile, surtout si elle n'est pas bien préparée à l'intégration continue et aux pratiques de livraison continue.

Chp3: Métriques de Qualité Logicielle et Analyse du Code

Dr Bassirou NGOM

Master MISID

Université Amadou Mahtar Mbow

Définition des Métriques de Qualité

- **Une métrique** de qualité logicielle est une mesure permettant d'évaluer la qualité du code source.
- Elles sont utilisées pour détecter les problèmes de conception et améliorer la maintenabilité.

Exemples :

- complexité cyclomatique,
- couverture de code,
- duplication du code.
- Etc.

Types de Métriques

- **Métriques de complexité** :
  - Complexité cyclomatique (McCabe)
  - Nombre de lignes de code (LOC)

- **Métriques de qualité du code** :
  - Couverture de code (Code Coverage)
  - Densité des défauts

- **Métriques de maintenabilité**:
  - Nombre de classes et de méthodes par fichier
  - Duplication du code

Outils d’Analyse Statique

- **SonarQube**: Détecte les vulnérabilités et mesure la couverture de code.
- **PMD**: Analyse statique pour identifier les erreurs de programmation.
- **Checkstyle** : Vérifie les conventions de codage.
- **ESLint** : Analyse statique pour JavaScript.

Outils d’Analyse Dynamique

- **Profilers (JProfiler, VisualVM**) : Analyse des performances en temps réel.
- **Test Coverage (JaCoCo, Istanbul)** : Mesure la couverture des tests.
- **Détection des fuites mémoire (Valgrind, YourKit)**.

Interprétation des Résultats et Amélioration du Code

1. Identifier les segments de code nécessitant une refactorisation.
2. Éliminer les duplications et simplifier les structures complexes.
3. Utiliser des outils pour suivre l'évolution des métriques dans le temps.
4. Établir des seuils acceptables pour la qualité du code.

Chp4: Tests Logiciel

Dr Bassirou NGOM

Master MISID

Université Amadou Mahtar Mbow

Quand et comment assurer la qualité de développement ?

- Quand ?
  - **Avant le développement :**
    - Analyse – Conception
  - **Pendant le développement:**
    - Méthode de développement(TDD, Agile, Cycle en V, …)
    - Tests automatiques
    - Outils d'analyse du code, code review
  - **Après le développement:**
    - Tests automatiques
    - Déploiement et intégration continus

Quand et comment assurer la qualité de développement ?

- Quand ?
  - **Avant le développement :**
    - Analyse – Conception
  - **Pendant le développement:**
    - Méthode de développement(TDD, Agile, Cycle en V, …)
    - Tests automatiques
    - Outils d'analyse du code, code review
  - **Après le développement:**
    - Tests automatiques
    - Déploiement et intégration continus

Comment assurer la qualité d'un logiciel ?

- Focus sur les Tests automatiques
  - **Tests Unitaires**
  - **Tests d'intégration**
  - **Tests d'utilisabilité**
  - **Tests d'interface utilisateur**
  - **Tests de performance**
  - **Tests de charge**
  - **Couverture de tests**

De quoi s'agit-il ?

- Le concept de test logiciel est simple:
  - On dispose d'une classe à tester Classe1 ayant différentes méthodes
  - On programme une autre classe Classe1Test dont les méthodes vont essayer celles de Classe1, et vérifier qu'elles retournent les bonnes valeurs en fonction des paramètres fournis.
  - Si les résultats sont ceux attendus, les tests réussissent. Sinon Classe1 est incorrecte ... ou Classe1Test ou les deux …
- Tester, c'est douter.

Test unitaires et autres

- Le concept précédent est assez vague et recouvre plusieurs étendues de tests

Test unitaires et autres

- Les **tests unitaires** (ou « T.U. », ou « U.T. » en anglais) est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »)

Test unitaires et autres

- Les **tests unitaires** (unit tests) font en sorte de ne tester qu'une seule méthode chacun, avec une seule combinaison de valeurs pour ses paramètres. Cette méthode doit retourner un résultat spécifié. Il y a donc une multitude de tests unitaires, pour chaque classe, pour chaque méthode, pour chaque combinaison de valeurs qu'on peut passer à ces méthodes

Test unitaires et autres

- Les **tests d'intégration** vérifient les interactions entre deux classes ou seulement deux méthodes, que l'une a appelé l'autre dans des conditions précises.

Test unitaires et autres

- Les **tests d'intégration** vérifient les fonctionnalités visibles par l'utilisateur du logiciel, par exemple l'interface graphique.

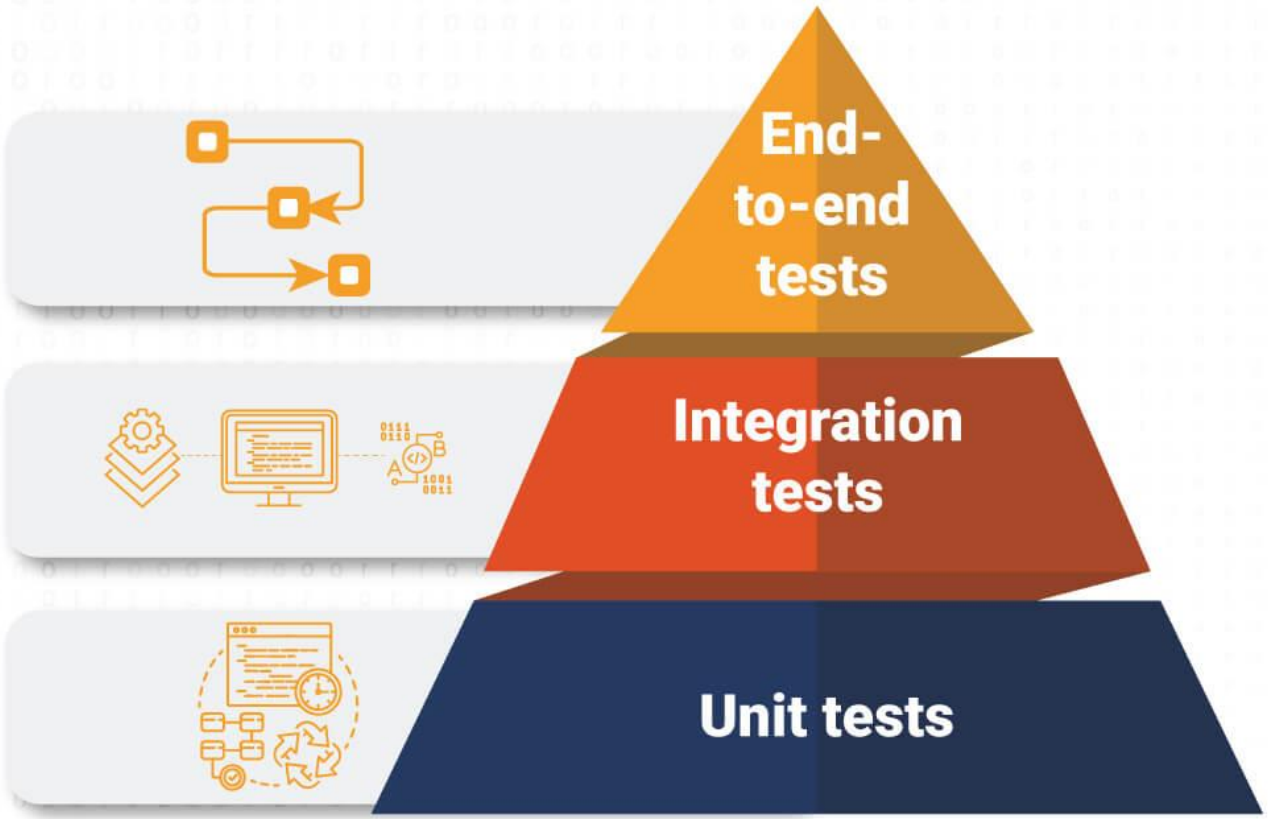
Test unitaires et autres

- Le concept précédent est assez vague et recouvre plusieurs étendues de tests.:
  - Les **tests unitaires** (unit tests) font en sorte de ne tester qu'une seule méthode chacun, avec une seule combinaison de valeurs pour ses paramètres. Cette méthode doit retourner un résultat spécifié. Il y a donc une multitude de tests unitaires, pour chaque classe, pour chaque méthode, pour chaque combinaison de valeurs qu'on peut passer à ces méthodes
  - Les **tests d'intégration** vérifient les interactions entre deux classes ou seulement deux méthodes, que l'une a appelé l'autre dans des conditions précises.
  - Les **tests fonctionnels** vérifient les fonctionnalités visibles par l'utilisateur du logiciel, par exemple l'interface graphique.

Complexité des tests

- Les différentes catégories de tests n'ont pas tous la meme compléxité:
  - Les **tests unitaires** sont rapides à vérifier, extrêmement nombreux, relativement simples à programmer, mais par définition, ils sont indépendants entre eux et travaillent sur des données figées, éloignées de ce que les utilisateurs pourront faire.
  - Les **tests d'intégration** ont moyennement nombreux, pas très faciles à programmer, pas très rapides.
  - Les **tests fonctionnels** sont de loin les plus lents et les plus complexes à programmer. Ils correspondent à des scénarios d'usages réalistes par les utilisateurs (use cases).
- Il y'a un ordre pour effectuer les tests : unitaires, intégration et enfin fonctionnels.

Pyramide des tests



Couverture des tests

- On doit, en principe, tout tester dans le logiciel. C'est ce qu'on appelle la **couverture des tests** (test coverage).
- Les tests unitaires doivent passer par chaque instruction de chaque méthode de chaque classe, ce qui témoigne de la qualité du logiciel :
  - toutes les affectations, tous les calculs, tous les appels deméthodes. . .
  - toutes les alternatives (telle condition vraie, telle condition fausse). . .
  - toutes les itérations aucune boucle, au moins une boucle). . .
  - les interceptions d'exceptions (pas d'exception, une exception). . .
- Pour info, les grandes entreprises exigent de leurs sous-traitants un taux de couverture approchant 100%.

Intérêt des tests

- Les classes de tests accompagnent le logiciel durant toute sa vie. Le moindre changement est confronté aux tests:
  - Les tests doivent **continuer à réussir**. Si ce n'est pas le cas, alors le changement a causé une **régression**, c'est à dire un retour à un moins bon fonctionnement, le retour d'un bug, etc.
  - Il est important de découvrir des problèmes le plus tôt possible dans la vie d'un logiciel. Les coûts de correction d'une erreur augmentent très fortement en fonction de l'âge de sa découverte, car de plus en plus de lignes de code sont à revoir.
  - Les tests doivent évoluer avec le logiciel, en particulier en développement Agile. Il faut maintenir les tests à chaque évolution du cahier des charges.
- On peut même se servir des tests pour guider le développement (**TDD = Test Driven Development**).

TDD: Test Driven Development

- Le principe, voir wikipedia, est d'organiser le travail en cycles très courts :
  - Écriture de tests destinés à vérifier une nouvelle fonctionnalité ; en principe, ces tests échouent car la fonctionnalité n'est pas encore programmée,
  - Programmation de la fonctionnalité pour faire réussir les tests,
  - Nettoyage, simplification du code source, en préservant la réussite de tous les tests.

- L'inconvénient est qu'il est préférable de confier la programmation des tests et des méthodes testées à des personnes différentes, afin de ne pas commettre les mêmes erreurs de conception (oublier de définir ce qui arrive aux valeurs null, par exemple).