Ecole Hassania des Travaux Publics

Le Génie Logiciel: Cycle de vie d'un logiciel

Zineb Rachik
Ecole Hassania des travaux publics
Année universitaire : 2016-2017

Analyse des besoins

- C’est la première grande étape du cycle de vie du logiciel
- Il s’agit du processus visant à établir quelles fonctionnalités le système doit fournir et les contraintes auxquelles il sera soumis.

Conception

- Dans les grands systèmes, la phase de la conception se divise en trois phases:
  - ✓ Spécification précise des composants logiciels.
  - ✓ Conception globale (conception architecturale): mettre le point sur les relations entre les composants logiciels.
  - ✓ Conception détaillée de ces divers composants.
- Il n'y a pas d'ordre clair entre ces trois phases, le concepteur procédant par itération.

Spécification des logiciels

- De la spécification des besoins à la spécification fonctionnelle:
- La fonction principale de la phase d'analyse des besoins est de préciser les services qui seront rendus par le logiciel à l'utilisateur. Les spécifications des besoins sont donc orientées utilisateur.
- En revanche, les spécifications fonctionnelles sont destinés au concepteur du logiciel. Le document des spécifications fonctionnelles doit alors contenir les définitions abstraites des composants et non pas la définition des services destinés à l'utilisateur.
- Une séparation claire entre la spécification fonctionnelle et la conception est difficile à établir; la spécification est souvent vue comme une partie du processus de la conception. Elle est sa représentation abstraite.

Conception architecturale

- Identification des sous systèmes constituant le système.
- Chaque sous système doit être décomposé en un certain nombre de composants; la spécification de ces sous système revient alors à définir les opérations de ces composants
- Définition les interactions entre les composants

➢ Résultat: une description de l'architecture du logiciel.

Conception détaillée

- Cette conception s'exprime en fonction d'abstractions de plus bas niveau, que l'on peut facilement traduire en code exécutable.
- Les algorithmes utilisés dans chaque composant doivent être détaillés.
- des tests unitaires sont définis pour s'assurer que les composants réalisés sont conformes à leurs descriptions.
- Résultat : pour chaque composant, le résultat consiste en :
  - ✓ Un dossier de conception détaillée.
  - ✓ Un dossier de tests unitaires.
  - ✓ Un dossier de définition d'intégration logiciel.

Qu’est ce qu’une bonne conception?

- Il n’existe pas de critère définitif permettant de définir une bonne conception.
- Le critère le plus décisif peut être l’efficacité du code produit et la maintenabilité du produit: une bonne conception facilite la maintenance et le cout des changement est minimal.
- Ces objectifs peuvent être atteints lorsque la conception apporte à la fois un haut degré de cohésion et un couplage faible.

Cohésion ?

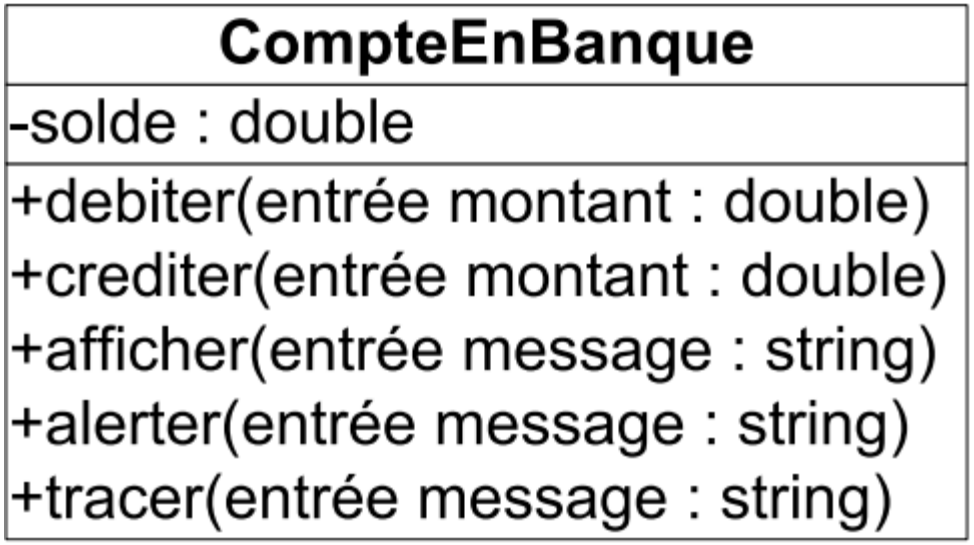
- On dit qu'une unité de programme fait preuve d'un haut degrés de cohésion si ses sous éléments remplissent des fonctions très proches; cela signifie que chaque sous élément est crucial pour que cette unité remplisse son rôle.
- En terme de classes, l'idée est de vérifier que nous rassemblons bien dans une classe des méthodes cohérentes, qui visent à réaliser des objectifs similaires.
- Des éléments qui sont regroupés dans une même unité car par exemple ils s'exécutent au même temps, ont un faible degré de cohésion.

Cohésion: mauvais exemple 1

- Imaginons une classe UserManager qui permet à l'utilisateur de récupérer ses informations (nom, prénom…) d'une base de donnée à partir d'un nom d'utilisateur et d'un mot de passe. Une telle classe est n'est pas cohésive car elle effectue des oppérations qui n'ont pas un lien direct entre elles: lecture des donnés, validation, récupération de la base…
- On peut rendre cette classe fortement cohésive en la découpant en plusieurs classes, une classe pour la vérification , une classe pour récupérer les données de la base….

Cohésion: mauvais exemple 2

- Mauvais exemple : il serait mal venu d'implémenter une méthode "Afficher()" ou "Tracer()" puisque l'objectif de cette classe est de représenter le modèle d'un compte en banque et non celui d'une fenêtre graphique ou d'un service de journalisation.



Couplage?

- Le couplage est une indication de la force des connexions entre unités.
- Une entité (fonction, module, classe, package, composant) est couplée à une autre si elle dépend d'elle.
- Les système à couplage fort ont des connexions fortes entre unités qui dépendent les unes des autres.
- Le couplage faible désigne une relation faible entre plusieurs entités (classes, composants), permettant une grande souplesse de programmation, de mise à jour. Ainsi, chaque entité peut être modifiée en limitant l'impact du changement au reste de l'application.

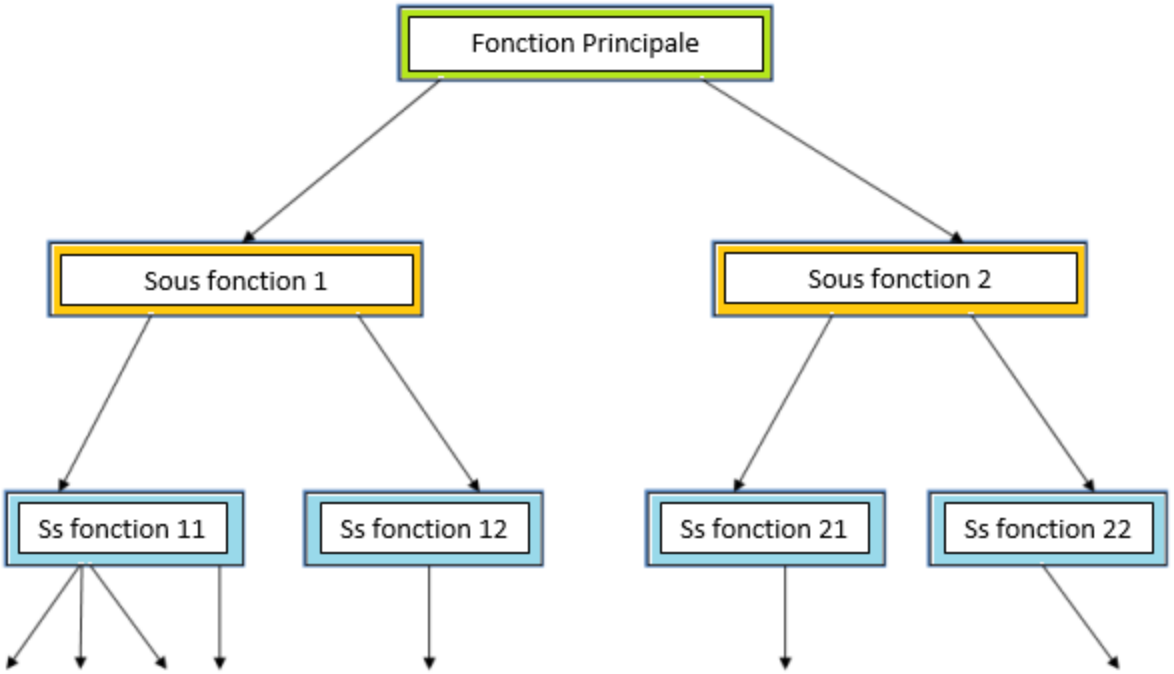
Systèmes à Forte cohésion et au couplage faible

- L’avantage de tels systèmes est qu’il est possible de remplacer une unité quelconque par une unité équivalente avec peu de changement dans les autres unités du système.

Conception; approche fonctionnelle

- Cette approche dérive des langages de programmation procéduraux. Au départ nous avons un grand problème que nous décomposons en plusieurs sous-problèmes. On effectue ensuite la décomposition des sous-problèmes en d'autres sous-problèmes. On itère le processus jusqu'à obtention de problèmes simples à comprendre donc à programmer. C'est donc un ensemble de fonctions précises en interaction les unes avec les autres.
- Dans cette approche, c'est la fonction principale (la solution au problème) qui décrit l'architecture du système. C'est une stratégie extrêmement ordonnée, logique et elle permet une réduction considérable de la complexité. Mais ce qu'on lui reproche c'est qu'elle ne permet pas une réutilisation des composants déjà définis et aussi il n'est pas évident d'assurer l'évolution du logiciel.

Conception; approche fonctionnelle



Conception; approche objet

- Contrairement à l'approche fonctionnelle, l'approche orienté objet tient sa forme de la structure du système. Dans cette approche, le logiciel est considéré comme un ensemble d'objets possédant des caractéristiques spécifiques et qui interagissent entre eux. Une caractéristique d'un objet peut être un attribut ou une méthode.
- Pour pouvoir l'utiliser, il est impératif de connaître les concepts ***d'encapsulation, d'agrégation, d'héritage, de polymorphisme, de généralisation, de spécialisation …***
- L'approche orientée objet est très rigoureuse et surtout moins intuitive que l'approche fonctionnelle.

Codage et tests unitaires

- Il s’agit de l’écriture du code source du logiciel, et tester son comportement afin de vérifier s’il réalise les responsabilités qui lui sont allouées

Intégration et validation

- Assembler le code source du logiciel (éventuellement partiellement) et dérouler les tests d'intégration: les différents sous-ensembles sont mis ensemble pour former le logiciel complet.
- On vérifie ainsi les interactions entre les sous-ensemble.
- Construire le logiciel complet exécutable. Dérouler les tests de validation sur le logiciel complet exécutable.

Maintenance

- Maintenance corrective :
  - Corriger les erreurs : défauts d'utilité, d'utilisabilité, de fiabilité...
    - Identifier la défaillance, le fonctionnement
    - Localiser la partie du code responsable

    Maintenance adaptative
- Maintenance adaptative :
  - Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu de l'évolution des Environnements d'exécution
    - Fonctions à satisfaire
    - Conditions d'utilisation
- Maintenance perfective: donne lieu à de nouvelles versions

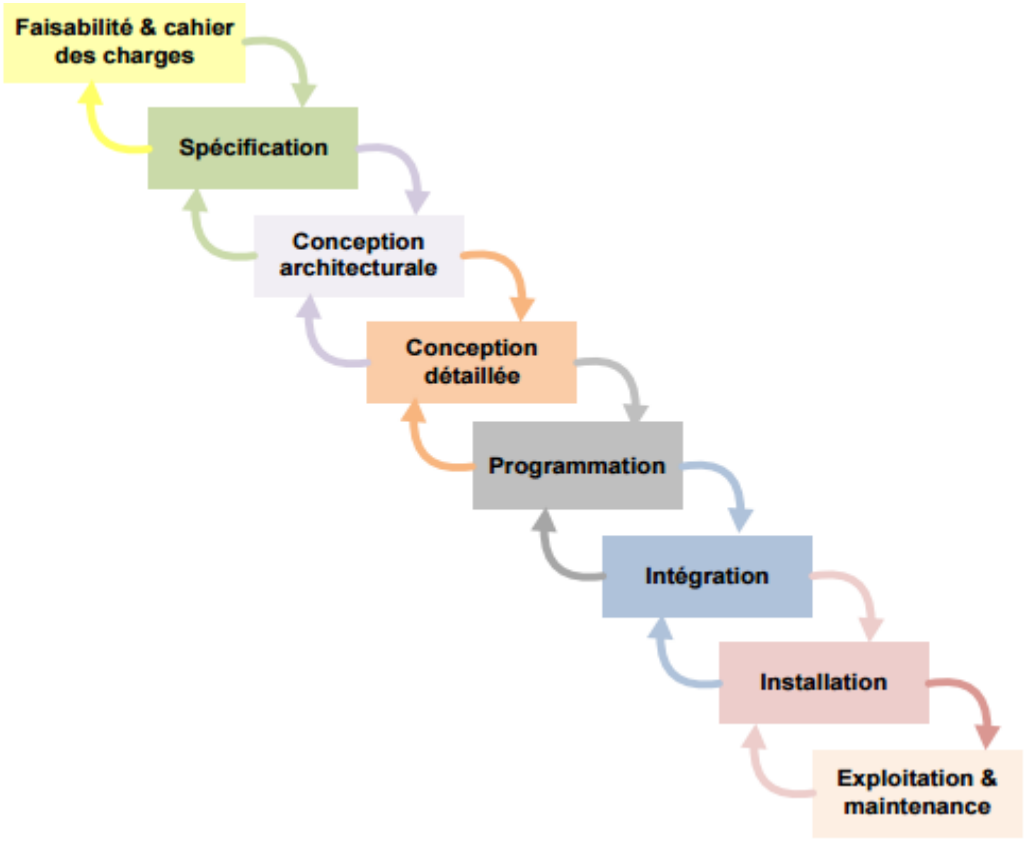
Modèles de cycles de vie

- Afin d'être en mesure d'avoir une méthodologie commune entre le client et la société de service réalisant le développement, des modèles de cycle de vie ont été mis au point définissant les étapes du développement ainsi que les documents à produire permettant de valider chacune des étapes avant de passer à la suivante. A la fin de chaque phase, des revues sont organisées avant de passer à la suivante.

Modèles de cycles de vie: modèle en cascade

- Le modèle en cascade (Waterfall model) est le plus classique des cycles de vie.
- Cycle de vie linéaire sans aucune évaluation entre le début du projet et la validation
- Le projet est découpé en phases successives dans le temps
- A chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables
- Chaque phase ne peut remettre en cause que la phase précédente

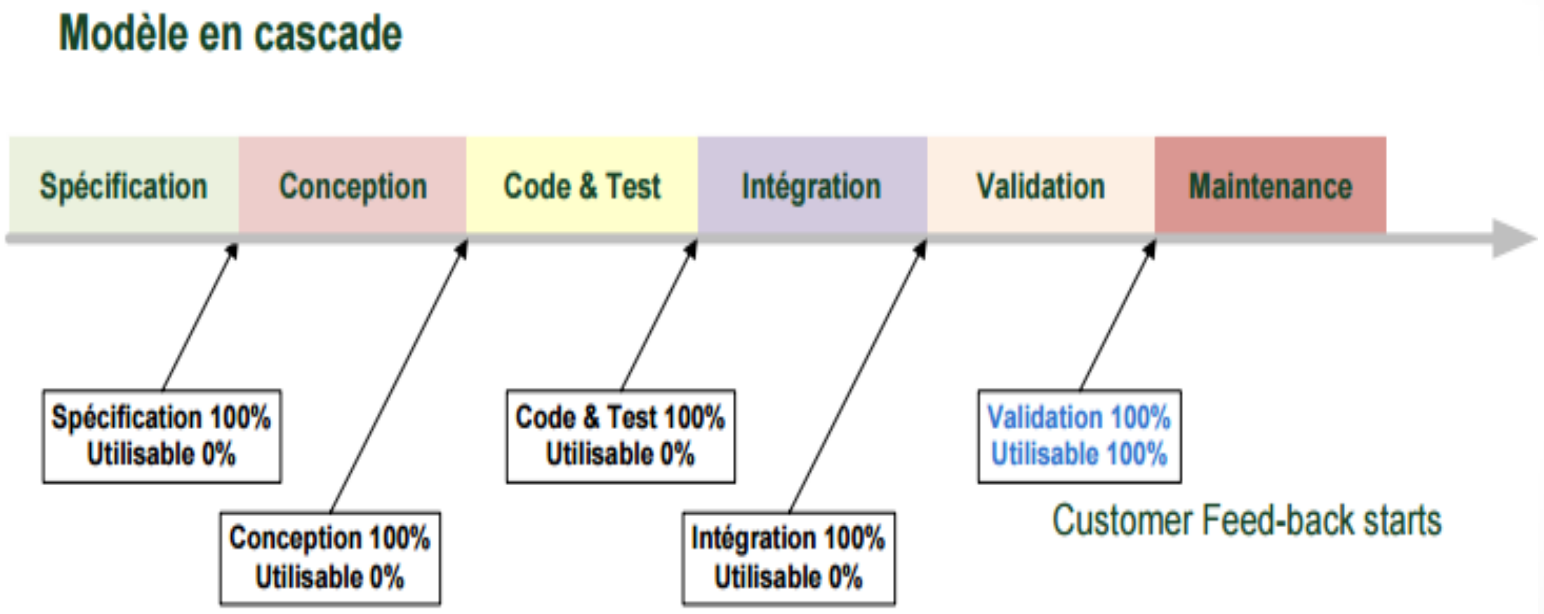
Modèles de cycles de vie: modèle en cascade



Modèle en cascade: inconvénients

- Peu réaliste:
  - Les vrais projets suivent rarement un développement séquentiel
  - Établir tous les besoins au début d'un projet est difficile
- Aucune validation intermédiaire (Aucune préparation des phases de vérification):
  - Obligation de définir la totalité des besoins au départ
  - augmentation des risques car validation tardive : remise en question coûteuse des phases précédentes
- Sensibilité à l'arrivée de nouvelles exigences : refaire toutes les étapes
- Très faible tolérance à l'erreur (les anomalies sont détectées tardivement) qui induit automatiquement un coût important en cas d'anomalie.
- Bien adapté lorsque les besoins sont clairement identifiés et stables

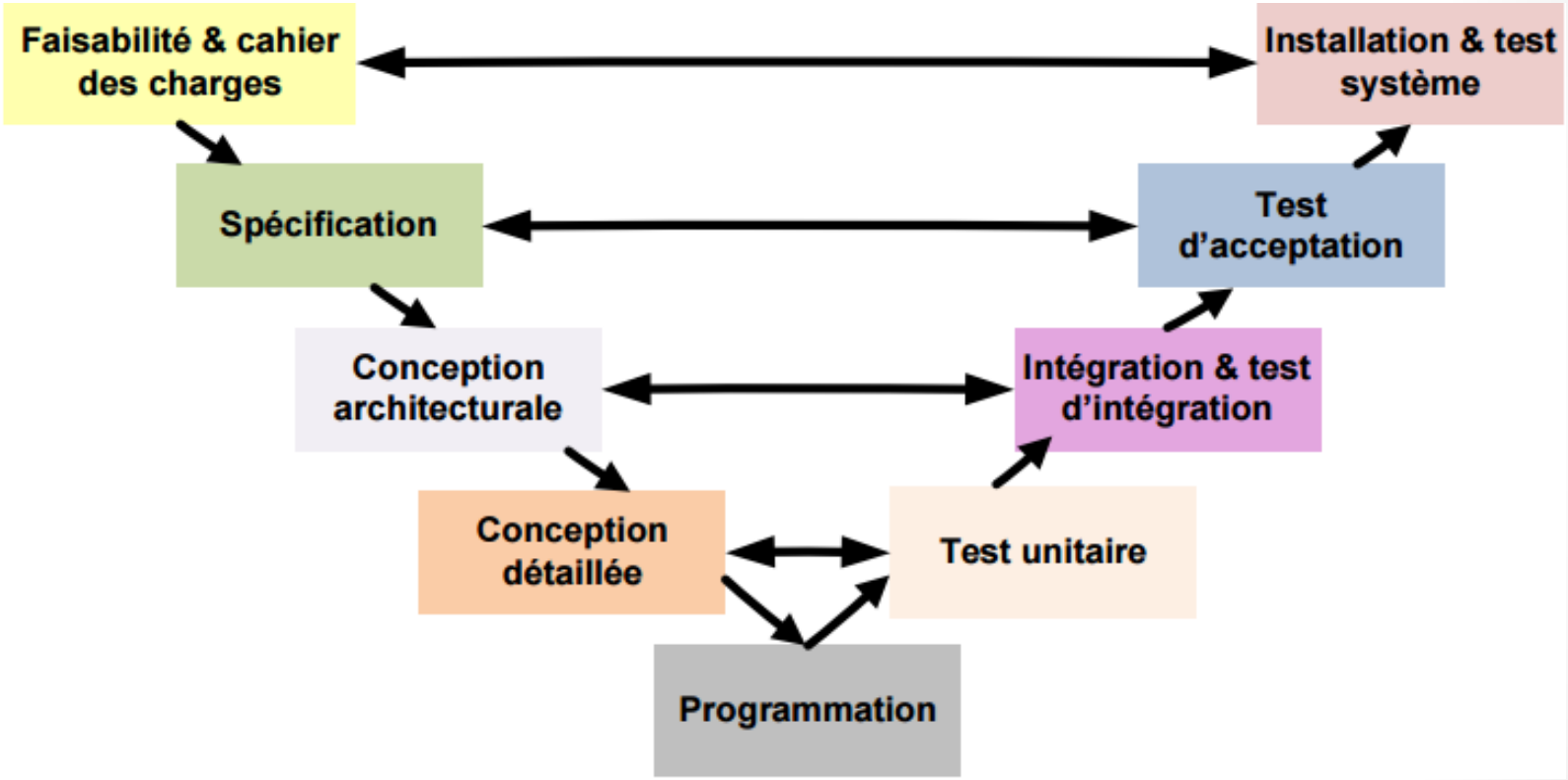
Modèle en cascade: inconvénients



Modèles de cycles de vie: Cycle de vie en V

- Face aux problèmes de réactivité que pose l'approche en cascade, l'industrie informatique a adopté le cycle en v. Ce modèle ne se découpe plus en 7 mais en 9 phases qui se répondent 2 à 2
- Le modèle de cycle de vie en V part du principe que les procédures de vérification de la conformité du logiciel aux spécifications doivent être élaborées dès les phases de conception.
- A **chaque étape de conception correspond une phase de test** ou de validation

Modèles de cycles de vie: Cycle de vie en V



Modèles de cycles de vie: Cycle de vie en V

- Les neufs phases du cycle en V sont:
- Etude et analyse : l'analyse ou la définition des besoins, la rédaction des spécifications, la conception architecturale, la conception détaillées
- Codage : développement de l'application
- Tests et validations : tests unitaires, test d'intégration, tests de validation et maintenance corrective.
- Remarque: plus on avance dans l'étude plus le niveau de détail est précis, ensuite on code, et plus on avance dans les tests moins le niveau de détails est précis.

Cycle de vie en V: avantages

- La stricte structure en V permet d'espérer que le livrable final sera parfait, puisque les étapes de test sont aussi nombreuses que les étapes de réflexion.
- Il est facile de prévoir les tests à réaliser au moment où l'on conçoit une fonctionnalité ou une interface, le travail s'enchaîne donc de façon assez naturelle.

Cycle de vie en V: inconvénients

- Construit-on le bon logiciel ? Le logiciel est utilisé très (trop) tard.
- Il faut attendre longtemps pour savoir si on a construit le bon logiciel.
- Difficile d'impliquer les utilisateurs lorsque qu'un logiciel utilisable n'est disponible qu'à la dernière phase
- Idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires

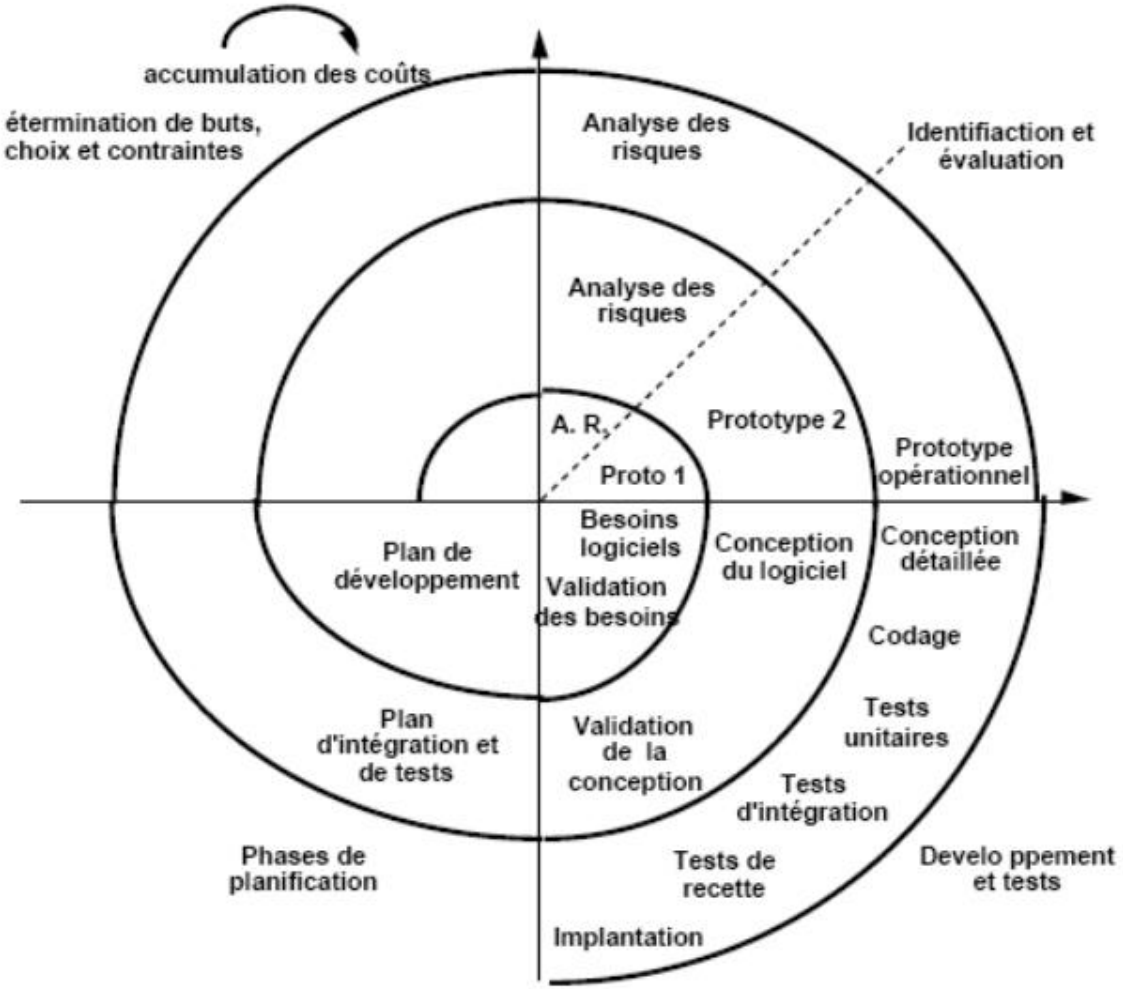
Modèles de cycles de vie: Cycle en spirale

- **le cycle en spirale reprend les étapes du cycle en V**, mais prévoit l'implémentation de **versions successives**, ce qui permet de mettre l'accent sur la **gestion des risques**, la première phase de chaque itération étant dédiée à ce poste. A ce point il est nécessaire de définir la notion de **prototype**.

Prototype?

- Il s’agit des versions incomplètes du produit.
- Il peut s’agir d’une simple maquette sans aucune fonctionnalité (on parle alors de prototype horizontal)
- ou bien de sites partiellement fonctionnels : telle version implémentera la navigation de base, la suivante ajoutera l’espace membres, puis la zone de téléchargements… on parlera alors de prototype vertical.
- Avantage :
  - Les efforts consacrés au développement d’un prototype sont le plus souvent compensés par ceux gagnés à ne pas développer de fonctions inutiles

Cycle en spirale



Cycle en spirale

- Chaque cycle de la spirale se déroule en quatre phases :
- détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
- analyse des risques, évaluation des alternatives et, éventuellement maquettage ;
- développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
- revue des résultats et vérification du cycle suivant.

Cycle en spirale: avantages

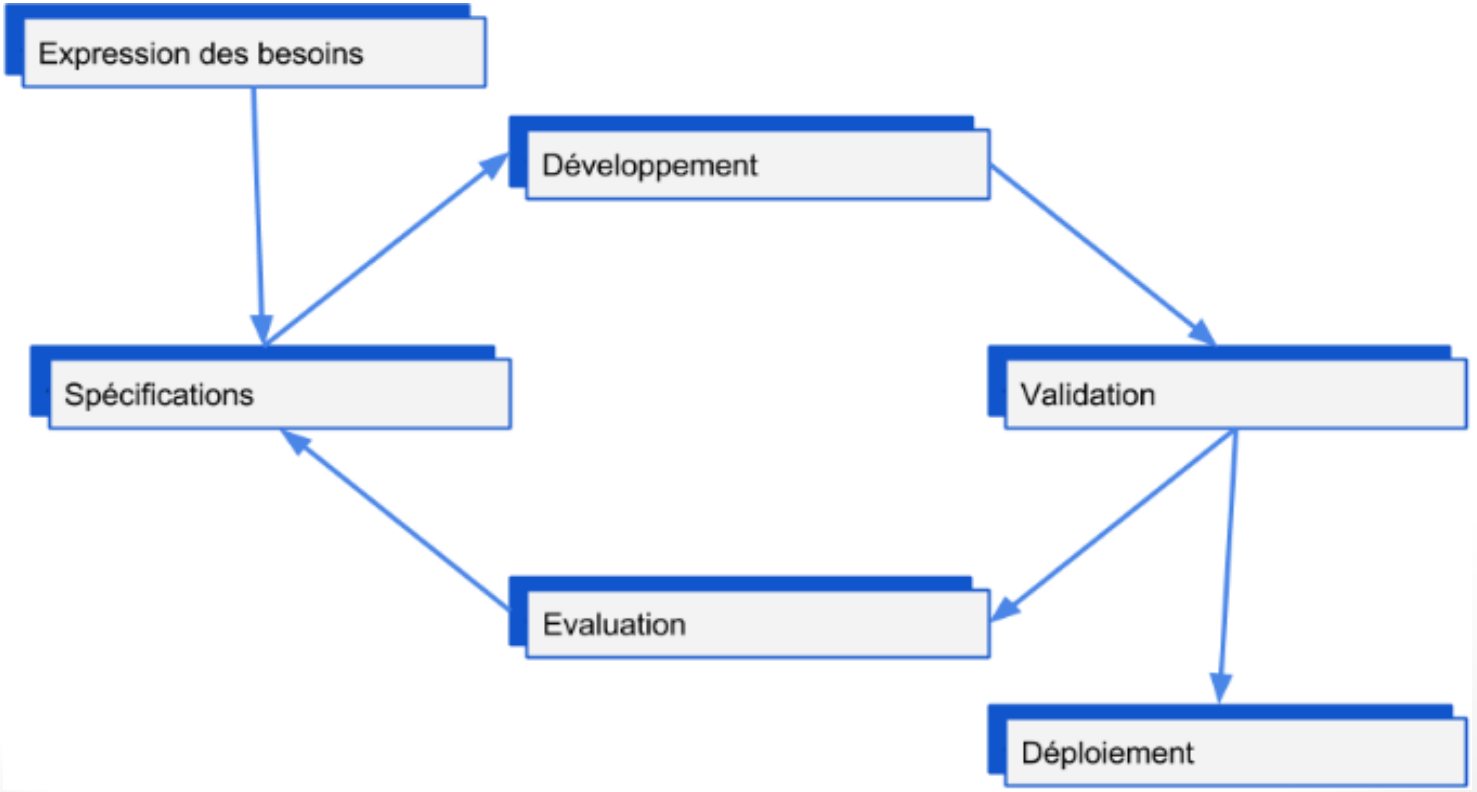
- Le but premier de ce modèle étant la gestion des risques, ceux-ci sont logiquement limités.
- L'expertise du client croit à chaque itération du cycle, l'apprentissage se fait par touche et pas d'un seul bloc.
- Enfin, ce modèle est très adaptatif : si chaque prototype apporte des fonctionnalités indépendantes, il est possible de changer l'ordre de livraison des versions.

Cycle en spirale: Inconvénients

- Il n’est adapté qu’aux projets suffisamment gros, inutile de prévoir la livraison de 5 ou 6 prototypes pour un petit projet.
- L’évaluation des risques en elle-même et la stricte application du cycle de développement peut engendrer plus de coûts que la réalisation du logiciel.
- Enfin, ce type de cycle de développement est complexe, entre les étapes prévues en théorie et celles mises en pratique il y a une grande différence.

Cycle itératif

- Simplifions un peu le modèle précédent en réduisant le nombre d'étapes du cycle et séparons les activités des artéfacts (c'est à dire les produits issus de ces activités). Nous arrivons logiquement au modèle itératif



Cycle itératif: Principes

- Développez de petits incréments fonctionnels livrés en courtes itérations
- Un incrément fonctionnel est un besoin du client.
- Pour chaque version à développer après la 1ère version livrée, il faut arbitrer entre :
  - Les demandes de correction,
  - les demandes de modification et
  - les nouvelles fonctionnalités à développer

Cycle itératif: avantages et inconvénients

- **Avantages du modèle itératif :** Ce type de cycle de développement est le plus souple de tous ceux présentés ici : chaque itération permet de s'adapter à ce qui a été appris dans les itérations précédentes et le projet fini peut varier du besoin qui a été exprimé à l'origine. Comme dans le cycle en spirale, la mise à disposition de livrables à chaque cycle permet un apprentissage de l'utilisateur final en douceur.
- **Inconvénients du modèle itératif :** la confiance qui amène bien souvent à négliger les test d'intégration. Ainsi les développeurs livrent une nouvelle fonctionnalité sans se rendre compte qu'ils ont cassé une chose qui fonctionnait dans les cycles précédents. Il faut donc que le chef de projet soit particulièrement vigilant lors de la phase de tests.