Chapitre Introduction au génie logiciel

-1. L'informatisation

L'informatisation est le phénomène le plus important de notre époque. Actuellement, l'informatique est au cœur de toutes les grandes entreprises. Le système d'information d'une entreprise est composé de matériels et de logiciels. Plus précisément, les investissements dans ce système d'information se répartissent généralement de la manière suivante : 20 % pour le matériel et 80 % pour le logiciel. En effet, depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement. Ce matériel est relativement fiable et le marché est standardisé. Les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel.

-2. Les logiciels

Un logiciel ou une application est un ensemble de programmes, qui permet à un ordinateur ou à un système informatique d'assurer une tâche ou une fonction en particulier (exemple : logiciel de comptabilité, logiciel de gestion des prêts).

Les logiciels, suivant leur taille, peuvent être développés par une personne seule, une petite équipe, ou un ensemble d'équipes coordonnées.

En 1995, une étude du Standish Group dressait un tableau accablant de la conduite des projets informatiques. Reposant sur un échantillon représentatif de 365 entreprises, totalisant 8 380 applications, cette étude établissait que :

- 16,2% seulement des projets étaient conformes aux prévisions initiales,
- 52,7% avaient subi des dépassements en coût et délai d'un facteur 2 à 3 avec diminution du nombre des fonctions offertes,
- 31,1% ont été purement abandonnés durant leur développement.

Pour les grandes entreprises (qui lancent proportionnellement davantage de gros projets), le taux de succès est de 9% seulement, 37% des projets sont arrêtés en cours de réalisation, 50% aboutissent hors délai et hors budget.

-3.Génie Logiciel Définition et objectifs du génie logiciel (GL)

-3.1. Définition

Domaine des ‘sciences de l’ingénieur’ dont la finalité est la *conception*, la *fabrication* et la *maintenance de systèmes logiciels complexes, sûrs et de qualité* (‘Software Engineering’ en anglais). Aujourd’hui les économies de tous les pays développés sont dépendantes des systèmes logiciels.

Le GL se définit souvent par opposition à la ‘programmation’, c’est à dire la production d’un programme par un individu unique, considérée comme ‘facile’. Dans le cas du GL il s’agit de la fabrication *collective* d’un *système complexe*, concrétisée par un ensemble de documents de conception, de programmes et de jeux de tests avec souvent de *multiples versions* (« multi-person construction of multi-versionsoftware »), et considérée comme ‘difficile’.

-3.2. Objectifs – la règle du CQFD

Le GL se préoccupe des *procédés de fabrication des logiciels* de façon à s'assurer que les 4 critères(coût, qualité, fonctionnalité et délais) soient satisfaits.

· **Le système qui est fabriqué répond aux *besoins* des utilisateurs (correction fonctionnelle).**

· **La *qualité* correspond au contrat de service initial.**

· **Les *coûts* restent dans les limites prévues au départ.**

· **Les *délais* restent dans les limites prévues au départ.**

Ces qualités sont parfois *contradictoires* (chic et pas cher !). Il faut les *pondérer* selon les circonstances (logiciel critique / logiciel grand public). Il faut aussi distinguer les systèmes sur mesure et les produits logiciels de grande diffusion.

-4. Les principes du génie logiciel

Ces principes sont très abstraits et ne sont *pas utilisables directement*. Mais ils font partie du vocabulaire de base du génie logiciel. Ces principes ont un impact réel sur beaucoup d'aspects et constituent le type de connaissance le plus stable, dans un domaine où les outils, les méthodes et les techniques évoluent très vite. On peut les résumer en sept principes fondamentaux

-4.1. Rigueur

La production de logiciel est une activité créative, mais qui doit se conduire avec une certaine rigueur.

Le niveau maximum de rigueur est la *formalité*, c'est à dire le cas où les descriptions et les validations s'appuient sur des notations et lois mathématiques. Il n'est pas possible d'être formel tout le temps : il faut bien construire la première description formelle à partir de connaissances non formalisées ! Mais dans certaines circonstances les techniques formelles sont utiles.

-4.2. "Séparation des problèmes"

C'est une règle de bons sens qui consiste à considérer séparément différents aspects d'un problème afin d'en maîtriser la complexité.

Elle prend une multitude de formes :

- séparation dans le *temps* (les différents aspects sont abordés successivement), avec la notion de cycle de vie du logiciel que nous étudierons en détail,
- séparation des *qualités* que l'on cherche à optimiser à un stade donné (ex assurer la correction avant de se préoccuper de l'efficacité),
- séparation des *'vues'* que l'on peut avoir d'un système (ex : se concentrer sur l'aspect 'flots de données' avant de considérer l'aspect ordonnancement des opérations ou 'flot de contrôle'),
- séparation du système en *parties* (un noyau, des extensions, …),
- etc.

-4.3. Modularité

Un système est modulaire s'il est composé de *sous-systèmes plus simples*, ou modules. La modularité permet de considérer séparément le *contenu* du module et les *relations entre modules* (ce qui rejoint l'idée de séparation des questions). Elle facilite également la *réutilisation* de composants bien délimités. Un bon découpage modulaire se caractérise par une *forte cohésion* interne des modules (ex : fonctionnelle, temporelle, logique, ...) et un *faible couplage* entre les modules (relations inter modulaires en nombre limité et clairement décrites).

Toute l'évolution des langages de programmation vise à rendre plus facile une programmation modulaire, appelée aujourd'hui 'programmation par composants'.

-4.4. Abstraction

L'abstraction consiste à ne considérer que les *aspects jugés importants* d'un système à un moment donné, en faisant abstraction des autres aspects (c'est encore un exemple de séparation des problèmes). Une même réalité peut souvent être décrite à différents *niveaux d'abstraction*. Par exemple, un circuit électronique peut être décrit par un modèle mathématique très abstrait (équation logique), ou par un assemblage de composants logiques qui font abstraction des détails de réalisation (circuit logique). L'abstraction permet une meilleure maîtrise de la complexité.

-4.5. Anticipation du changement

La caractéristique essentielle du logiciel, par rapport à d'autres produits, est qu'il est presque toujours soumis à des *changements continuels* (corrections d'imperfections et évolutions en fonctions *des besoins qui changent*). Ceci requiert des efforts particuliers pour *prévoir*, faciliter et gérer ces évolutions inévitables. Il faut par exemple :

- Faire en sorte que les changements soient les plus localisés possibles (bonne modularité),
- Etre capable de gérer les multiples versions des modules et configurations des versions des modules, constituant des versions du produit complet.

-4.6. Généricité

Il est parfois avantageux de remplacer la résolution d'un problème spécifique par la résolution d'un problème plus général. Cette solution générique (paramétrable ou adaptable) pourra être *réutilisée* plus facilement.

Exemple : plutôt que d'écrire une identification spécifique à un écran particulier, écrire (ou réutiliser) un module générique d'authentification (saisie d'une identification - éventuellement dans une liste - et éventuellement d'un mot de passe).

-4.7. Construction incrémentale

Un procédé incrémental atteint son but par étapes en s'en approchant de plus en plus ; chaque résultat est construit en étendant le précédent.

On peut par exemple réaliser d'abord un *noyau des fonctions essentielles* et ajouter progressivement les *aspects plus secondaires*. Ou encore, construire une série de *prototypes* 'simulant' plus ou moins complètement le système envisagé.

5- Les bases de la qualité du logiciel

Pour évaluer la qualité d'un système, identifions les facteurs que l'on voudrait retrouver dans tous les bons logiciels. En plus de devoir fournir les fonctionnalités requises, un bon logiciel doit aussi posséder les facteurs clés suivants :

– la *validité* : aptitude d'un logiciel à réaliser exactement les tâches définies par sa spécification,

– la *fiabilité* : aptitude d'un logiciel à assurer de manière continue le service attendu,

– la *robustesse* : aptitude d'un logiciel à fonctionner même dans des conditions anormales,

– l'*extensibilité* : facilité d'adaptation d'un logiciel aux changements de spécification,

– la *réutilisabilité* : aptitude d'un logiciel à être réutilisé en tout ou partie,

– la *compatibilité* : aptitude des logiciels à pouvoir être combinés les uns aux autres,

– l'*efficacité* : aptitude d'un logiciel à bien utiliser les ressources matérielles telles la mémoire, la puissance de l'U.C., etc.

– la *portabilité* : facilité à être porté sur de nouveaux environnements matériels et/ou logiciels,

– la *traçabilité* : capacité à identifier et/ou suivre un élément du cahier des charges lié à un composant d'un logiciel,

– la *vérifiabilité* : facilité de préparation des procédures de recette et de certification,

– l'*intégrité* : aptitude d'un logiciel à protéger ses différents composants contre des accès ou des modifications non autorisés,

– la *facilité d'utilisation*, *d'entretien*, etc.

Remarque : Il est difficile d'optimiser tous ces facteurs car certains s'excluent mutuellement (par exemple, offrir une bonne interface à l'utilisateur peut réduire l'efficacité). De petites améliorations peuvent revenir très cher .

-5. Modélisation, cycles de vie

5.1 Vie d'un logiciel

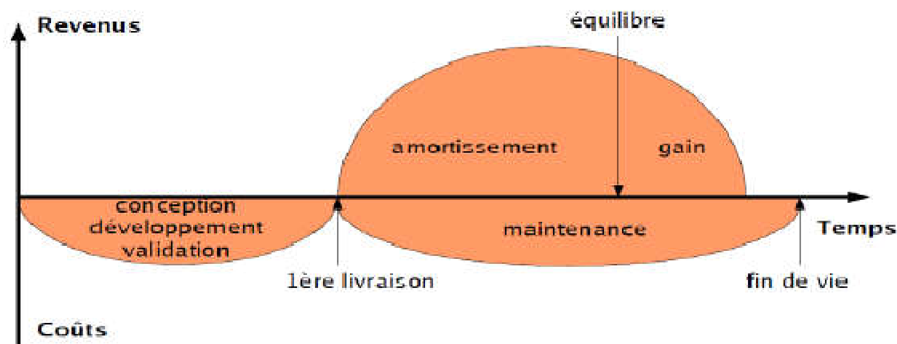


Figure1 : vie d'un logiciel.

5.2. Quand un logiciel est-il terminé ?

- quand on a fini de le programmer ?
- quand on l'a compilé ?
- quand il s'exécute sans se planter ?
- quand on l'a testé ?
- quand on l'a documenté ?
- quand il est livré au premier client ?
- quand il n'évolue plus ?
- quand il n'est plus maintenu ?

5.3. Pourquoi se préoccuper d'un « cycle de vie » ?
C'est un processus
– phases : création, distribution, disparition
● But du découpage
– maîtrise des risques
– maîtrise des délais et des coûts
– contrôle que la qualité est conforme aux exigences
● En fait, problématique plus générale – mais spécificités relatives aux logiciels

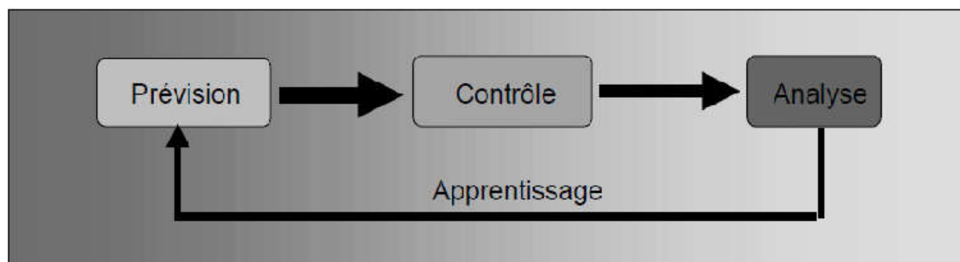


Figure2 : Cycle de vie vue comme un processus.

5-4. Pourquoi et comment modéliser ?

Modéliser un système avant sa réalisation permet de mieux comprendre le fonctionnement du système. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence. Un modèle est un langage commun, précis, qui est connu par tous les membres de l'équipe et il est donc, à ce titre, un vecteur privilégié pour communiquer.

5-5. Le cycle de vie d'un logiciel

Le cycle de vie d'un logiciel (en anglais software lifecycle), désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition.

L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. Le cycle de vie permet de détecter les erreurs au plus tôt et ainsi de maîtriser la qualité du logiciel, les délais de sa réalisation et les coûts associés.

Le cycle de vie du logiciel comprend généralement au minimum les étapes suivantes :

Analyse des besoins et faisabilité

- c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes, puis l'estimation de la faisabilité de ces besoins ;

Spécifications ou conception générale

- il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel ;

Conception détaillée

- cette étape consiste à définir précisément chaque sous-ensemble du logiciel ;

Codage (Implémentation ou programmation)

- c'est la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception ;

Tests unitaires

- ils permettent de vérifier individuellement que chaque sous-ensemble du logiciel est implémenté conformément aux spécifications ;

Intégration

- l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document ;

Qualification (ou recette)

- c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales ;

Documentation

- elle vise à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs ;

Mise en production

- c'est le déploiement sur site du logiciel ;

Maintenance

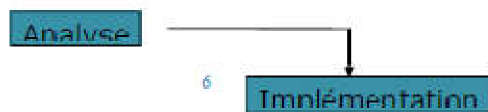
- elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépendent du choix d'un modèle de cycle de vie entre le client et l'équipe de développement. Le cycle de vie permet de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains.

6. Modèle de cycle de vie

6.1. Le modèle linéaire de cycle de vie

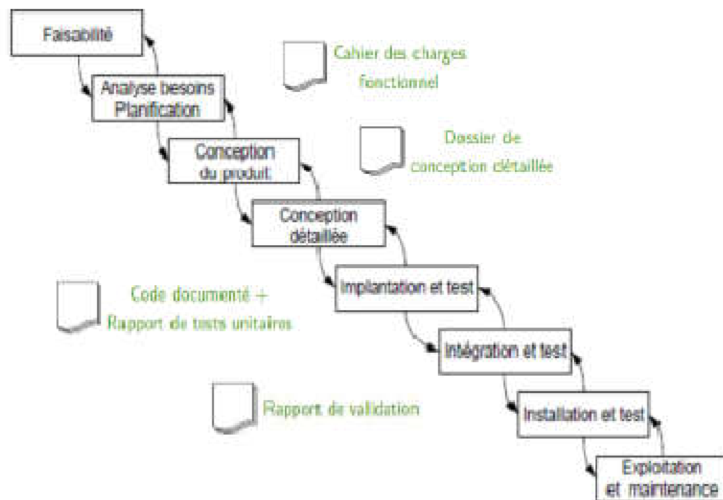
Historiquement, la première tentative pour mettre de la rigueur dans le développement sauvage a consisté à distinguer une phase d’analyse avant la phase d’implémentation (séparation des questions).



Le modèle linéaire de cycle de vie

6.2. Le modèle en cascade

Le modèle en cascade a été mis au point dès 1966, puis formalisé aux alentours de 1970. Il défini des étapes (ou phases) durant lesquelles les activités de développement se déroulent Une étape doit se terminer à une date donnée par la production de certains documents ou logiciels Les résultats de l'étape sont soumis à une étude approfondie. L'étape suivante n'est abordée que si les résultats sont jugés satisfaisants

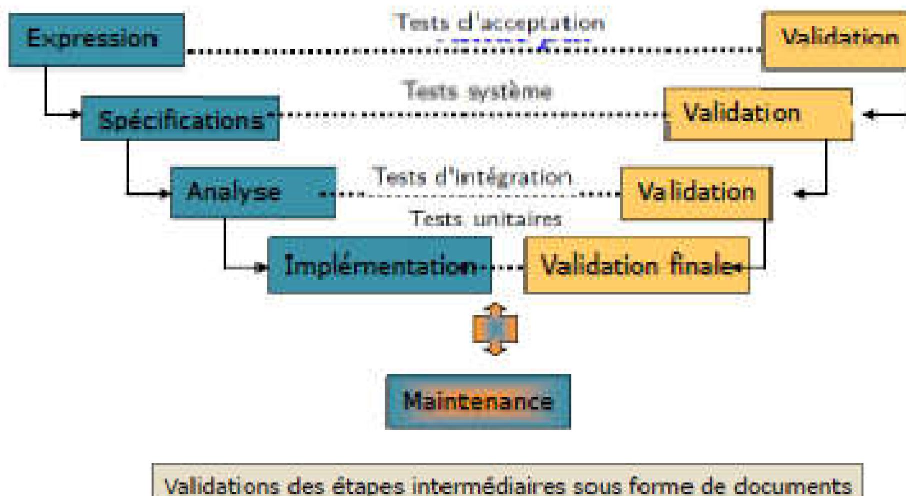


Le modèle en cascade

Inconvénients
· Entraîne des relations conflictuelles avec les parties prenantes en raison :
· Du manque de clarté de la définition des exigences
· D'engagements importants dans un contexte de profonde incertitude
· D'un désir inévitable de procéder à des changements
· Entraîne une identification tardive de la conception, et un démarrage tardif du codage
· Retarde la résolution des facteurs de risque (intégration tardive dans le cycle de vie)
· Exige d'accorder une attention très importante aux documents

6.3. Le modèle en V

Modèle de cascade amélioré dans lequel le développement des tests et du logiciel sont effectués de manière synchrone. Le principe de ce modèle est qu'avec toute décomposition doit être décrite la recomposition et que toute description d'un composant est accompagnée de tests qui permettront de s'assurer qu'il correspond à sa description.
Problème de vérification trop tardive du bon fonctionnement du système.



Principes :

Validation et préparation des protocoles de validation finaux à chaque étape descendante puis validation finale montante et confirmation de la validation descendant.

Intérêts :

Bon suivi du projet avec avancement éclairé et limitation des risques en cascade d'erreurs, une décomposition fonctionnelle de l'activité et génération de documents et outils supports

Limitations :

· Problème de vérification trop tardive du bon fonctionnement du système.

· Un modèle séquentiel (linéaire)

-Maintenance non intégrée : logiciels à vocation temporaire

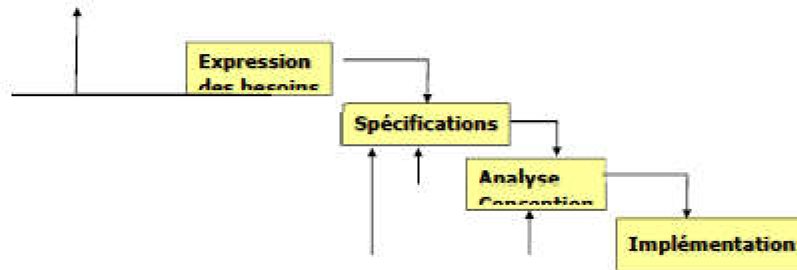
Conséquences :

Obligation de concevoir les jeux de test et leurs résultats, réflexion et retour sur la description en cours et ailleurs préparation de la branche droite du V

6.4. Le modèle itératif

Principe : A chaque étape, on rajoute de nouvelles fonctionnalités pour augmenter le logiciel

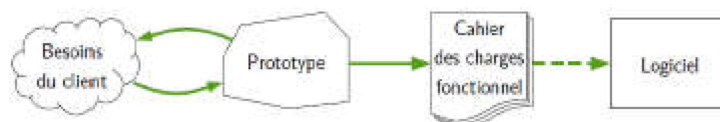
Caractéristiques : Chaque étape est relativement simple qui doit être testée et essayée au fur et à mesure que le logiciel est entrain d'être développé



Modèle itératif

6.5. Le modèle par prototype

Modèle valable dans le cas où les besoins ne sont pas clairement définis ou ils changent au cours de temps. Il permet le développement rapide d'une ébauche du futur logiciel avec le client afin de fournir rapidement un prototype exécutable pour permettre une validation concrète (ici expérimentale) et non sur papier (document) avec moins de risque de spécifications. Des versions successives du prototype (itérations) sont réalisées avec écriture de la spécification à partir du prototype puis processus de développement linéaire



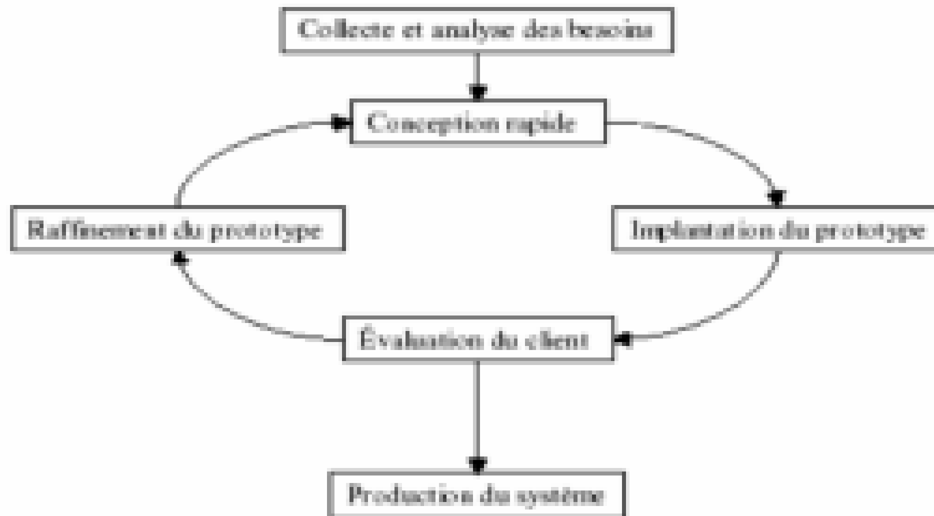
modèle par prototype

a. Prototype jetable :

Modèle gérable d'un point de vue changement des spécifications qui sont générées par prototypage qui est construit et utilisé lors de l'analyse des besoins et la spécification fonctionnelles avec validation des spécifications par l'expérimentation

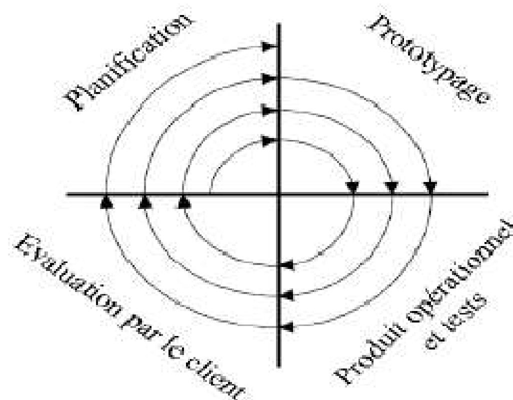
b. Prototype évolutif :

La première version du prototype construit l'embryon du produit final, plusieurs itérations sont réalisé jusqu'au produit final cependant il existe des difficultés à mettre en œuvre des procédures de validation et de vérification. Exemple modèle en spirale, développement incrémental

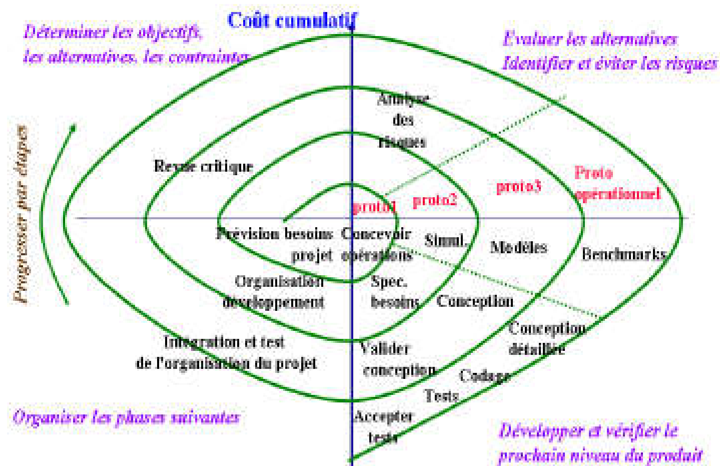


6.6. Le modèle en spirale

Proposé par B. Boëhm en 1988, ce modèle général met l'accent sur l'évaluation des risques. A chaque étape, après avoir défini les objectifs et les alternatives, celles-ci sont évaluées par Différentes techniques (prototypage, simulation, ...). L'étape est réalisée et la suite est planifiée, Le nombre de cycles est variable selon que le développement est classique ou incrémental



Le modèle en spirale



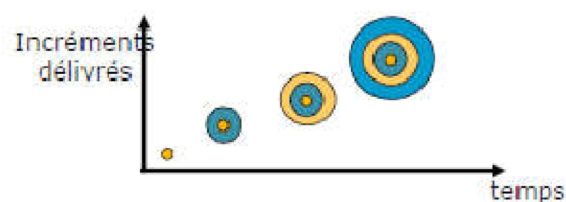
Le modèle en spirale

Risques majeurs :

· Défaillance du personnel
· Calendrier et budget irréalistes
· Développement de fonction inapproprié
· Développement d'interfaces utilisateurs inappropriées
· Composants externes manquants
· Tâches externes défaillantes
· Problème de performance

6.7. Le modèle incrémental

Face aux dérives bureaucratiques de certains gros développements, et à l'impossibilité de procéder de manière aussi linéaire, le modèle incrémental a été proposé dans les années 80.



modèle incrémental

Le développement est réalisé par suite d'incréments ou composantes, qui correspondent à des parties de la spécification, et qui viennent intégrer le noyau de logiciel. L'objectif est d'assurer la meilleure adéquation aux besoins possibles. Le choix d'incrément est un compromis entre la durée de développement et le niveau de prise en compte des spécification

a**. Avantages du modèle incrémental**
· Mise en production plus rapidement du système, avec la diffusion sous forme de version
· Les premiers incréments renforcent l'expression de besoin et autorisent la définition des incréments suivant.
· Diminution du risque global d'échec du projet
· Les fonctions les plus utilisées seront les plus testées et donc les plus robustes
b. **inconvénients du modèle incrémental**
· Remettre en cause les incréments précédents ou pire le noyau ;
· Ne pas pouvoir intégrer de nouveaux incréments. · Les noyaux, les incréments ainsi que leurs interactions doivent être spécifiés globalement, au début du projet. Les incréments doivent être aussi indépendants que possibles, fonctionnellement mais aussi sur le plan du calendrier du développement.

6.8. Le modèle avec réutilisation de composants
Développer un logiciel à l'aide d'une base de composants génériques préexistants. L'élaboration de la spécification est dirigée par cette base : une fonctionnalité est proposée à l'utilisateur en fonction des composants existants. Ce modèle permet d'obtenir rapidement des produits de bonne qualité (utilisation des composants prouvés). Le travail d'intégration peut s'appuyer sur des outils dirigés par des descriptions de haut niveau du système pour générer le code, cette situation est typique chez les sociétés de services (hébergement de serveurs, déploiement automatique de site Web, . . .).

Etapes des processus réutilisation de composants
. Analyse des composants
· Besoins en modification
· Conception avec réutilisation
· Développement et intégration
Cette approche est très répondue car elle peut permettre d'importantes réductions de coût

7. Conclusion
Il n'y a pas de modèle idéal car tout dépend des circonstances. Le modèle en cascade ou en V est risqué pour les développements innovants car les spécifications et la conception risquent d'être inadéquate et souvent remis en cause. Le modèle incrémental est risqué car il ne donne pas beaucoup de visibilité sur le processus complet. Le modèle en spirale est un canevas plus général qui inclut l'évaluation des risques. Souvent, un même projet peut mêler différentes approches, comme le prototypage pour les sous-systèmes à haut risque et la cascade pour les sous systèmes bien connus et à faible risque.

Référence :

1. G. Booch, J. Rumbaugh, I. Jacobson, « The Unified Modeling Language (UML) user guide”, Addison-Wesley, 1999.
2. Benoit charroux, aomar Osmani, Yann Therry-Mieg,”UML2 synthèse et exercices”Pearson édition france,ISBN2-7440-7124-2,2005.
3. G. Booch et al., « Object-Oriented Analysis and Design, with applications », Addison- Wesley, 2007.
4. Cours UML 2.0 de Laurent Audibert , site http://www.developpez.com