UNIVERSITÉ DES SCIENCES ET DE LA TÉCHNOLOGIE HOUARI BOUMEDIENE

GÉNIE LOGICIEL

Chargés de cours :

Section A : M. HAMMAL Youcef
Section B : Mme MAHDAOUI Latifa
Section C : Mme BOUSSAÏD Ilhem

Licence 3 académique

Année universitaire 2013/2014

Table des matières

Chapitre Un

Introduction au Génie Logiciel

1.1 Introduction

La structure des coûts des systèmes informatiques s'est profondément modifiée depuis les années 60. Dans un premier temps, les coûts matériels ont été prépondérants mais les progrès considérables de la technologie électronique (apparition des mini-ordinateurs puis des micro-ordinateurs) a entraîné des baisses de coûts du matériel dans des proportions jamais vues conduisant à une utilisation très accrue des ordinateurs dans des domaines très divers et de plus en plus complexes et ayant des missions de plus en plus critiques (télécommunications, systèmes temps réel,...).

Malheureusement, les coûts des logiciels n'ont pas diminué autant que ceux du matériel. Pour s'en convaincre, il suffit de constater qu'ils représentent actuellement environ 80% du coût total d'un système informatique. C'est dans ce contexte que l'un des buts du Génie Logiciel est de diminuer sensiblement les coûts de développement (et de maintenance) de ces gros systèmes qui sont immenses.

Exemple 1.1. *Les coûts de développement de logiciels aux USA pendant les années* 77 *dépassaient les* 50 *milliards de dollars. Aux années* 88*, ils étaient déjà de l'ordre de* 100 *milliards de dollars.*

1.2 La crise du logiciel

Cette fameuse crise est apparue avec l'avènement des ordinateurs de la $3^{\text{ème}}$ génération (années 60) et s'est accentuée avec l'ère de la micro-informatique. En effet, leur coût et leur puissance ont permis la prolifération de l'informatique dans beaucoup de domaines avec la réalisation de grands systèmes logiciels. Les premières expériences de construction de ce type de systèmes ont montré que les méthodes de développement de l'époque n'étaient pas adéquates.

Donc, il n'était plus possible de se contenter d'adapter les techniques applicables à de petits systèmes car les problèmes rencontrés dans le développement des gros systèmes logiciels ne sont pas directement comparables à ceux relatifs aux petits. La complexité des petits programmes peut être appréhendée par une seule personne qui peut avoir à l'esprit tous les détails de la conception et de la réalisation. Quant aux spécifications, elles peuvent rester informelles

et les effets de toutes modifications sont en général évidents. En revanche, la complexité des grands systèmes est telle qu'il est impossible à une seule personne d'avoir présent à l'esprit tous les détails du projet et de les mettre à jour. En réalité, ce sont des équipes d'analystes, de concepteurs, de développeurs et de testeurs qui doivent collaborer pour parvenir à réaliser ce type de projet.

Notations :

HA : Homme-Année (unité de coût de logiciel)

HM : Homme-Mois

Kls : un millier de lignes de code source (unité de mesure de la taille d'un logiciel)

Ls : une ligne de code source.

TABLEAU 1.1:

Nature du Logiciel	**Coût en HA**	**Volume**	**Remarque**
Compilateurs :			Délais :
Pascal	10 ha	20 à 30 kls	1 à 2 ans
COBOL, FORTRAN	80 à 100 ha	100 à 200 kls	2 à 3 ans
ADA	150 à 200 ha	>300 kls	> 3 ans
SGBD Relationnel (ORACLE, DB2, ...)	300 à 500 ha	300 à 600 kls	Délais : de 3 à 5 ans incluant une 1ère version.
Grands systèmes temps réel :			Délais :
Navette Spatiale	>1000 ha	2200 kls	6 ans, écrit en HAL
SAFEGUARD [1]	5000 ha	2260 kls	7 ans, écrit en pl1
SABRE [2]	955 ha	960 kls	10 ans pour un MTTF [3] de 55 h.
Systèmes d'exploitation constructeurs : VMS, GCOS7	2500 à 5000 ha	5000 à 10000 kls	Durée de vie 15 à 20 ans dont au moins 5 ans de délai pour une 1ère version.
Systèmes industriels : GPAO, MRP, ...	150 à 500 ha	500 à 1000 kls	Bases de données importantes.

Remarque 1.1. *La productivité de développement peut être exprimée comme étant le rapport entre la taille du logiciel et son coût. Cet indicateur dénote la difficulté de fabrication du logiciel. (Unité de mesure = ls/hm)*

Grosso modo, les projets souffraient de :

- Retards dans les livraisons (parfois des années).
- Surcoûts considérables (dépassant largement les budgets prévus).
- Produits qui ne répondent pas aux besoins des usagers (peu fiables, peu performants, difficiles à maintenir, etc.).

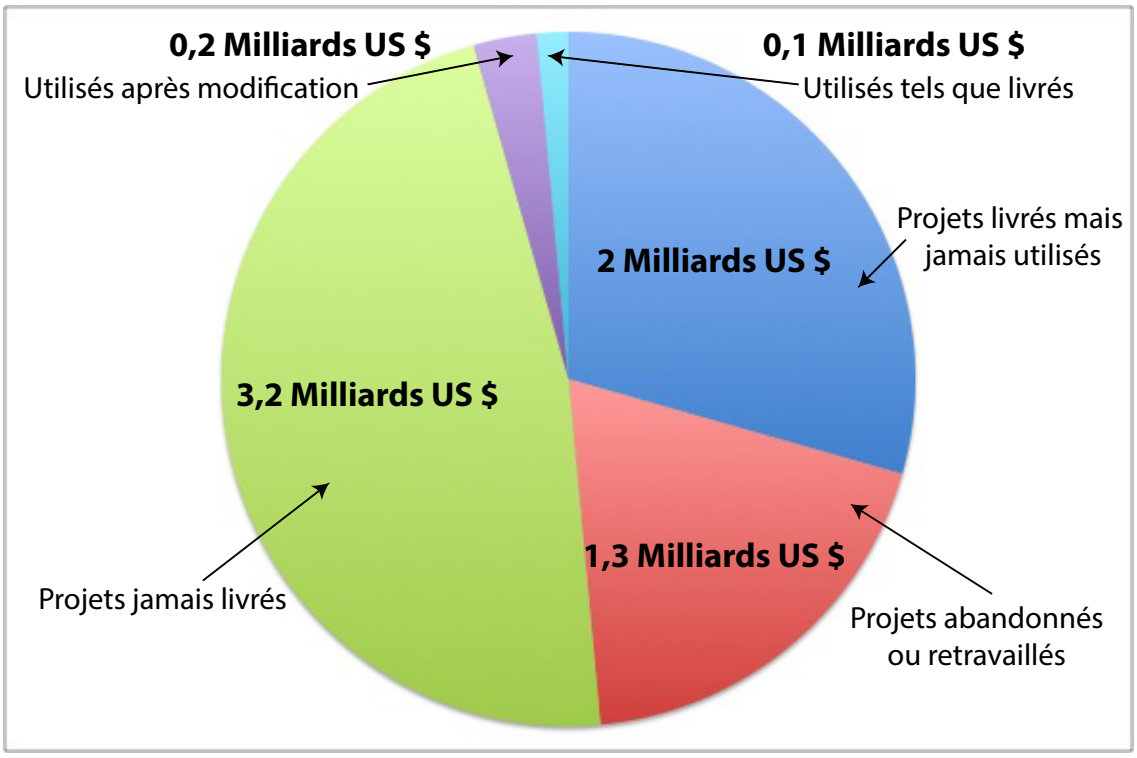

FIGURE 1.1: Répartition de 6,8 Milliards US $ en projets logiciels commandités par le gouvernement US en 1979.

– Produits peu sûrs (comportant des erreurs dont le nombre et l'emplacement sont souvent inconnus et ayant des répercussions graves sur la mission du système).

Ces problèmes s'apparentent avec une croissance exponentielle de la demande de logiciels et avec la croissance dans la complexité des systèmes construits dans un contexte caractérisé par la diminution des prix du matériel et le gonflement incontrôlable des prix de logiciels.

Remarque 1.2. *De 1965 à 1995, en 30 ans, le volume de chaque logiciel a été multiplié par 100, alors que la productivité n'augmentait que d'un facteur de 3. Attention, il ne suffit pas d'augmenter l'investissement en hommes pour augmenter d'autant la productivité. Donc, il est évident que de nouvelles techniques formelles de spécification, de conception et de réalisation sont nécessaires ; Chaque étape du projet doit être correctement évaluée et documentée et une gestion soignée est essentielle.*

Quelques Bogues :

– Echec de la mission de la sonde Mariner vers Vénus dans les années 71 : (perdue dans l'espace) passage à 5.000.000 Km de la planète au lieu de 5.000 Km prévus.
Cause : Remplacement d'une virgule par un point.

– OS 360 : Système d'exploitation pour la série 360 des ordinateurs IBM. Livré en retard avec beaucoup de bogues.

- Perte de satellites dans les années 70.

  Cause : frappe de +I à la place de +1 dans une instruction d'itération du programme source (le langage Fortran admet sans le signaler des identificateurs non déclarés, leur valeur était alors aléatoire).
- Missile Ami : Navire de Guerre Britannique coulé par un Exocet français de la marine Argentine au cours de la guerre des Malouines. Le vaisseau anglais n'avait pas activé ses défenses.

  Cause : l'Exocet n'est pas répertorié comme missile ennemi.
- Socrate : système de réservation de places de la SNCF. Ses plantages fréquents, sa mauvaise ergonomie, le manque de formation préalable du personnel, ont amené une défection importante et durable de la clientèle vers d'autres moyens de transport.

  Cause : rachat par la SNCF d'un système de réservation de places d'une compagnie aérienne, sans réadaptation totale au cahier des charges du transport ferroviaire.
- Echec du 1er lancement d'Arian V : (explosion en vol)

  Le système informatique conçu pour Ariane IV n'a pas été convenablement réadapté pour Ariane V (plan de vol n'a pas été changé). Coût du programme de développement d'Arian V : 38 Milliards de Francs.
- Perte de Mars Climate Orbiter : Le 23 septembre 1999, après 9 mois de voyage. Coût : 120 Milliards US$

  Cause : confusion entre pieds et mètres.
- Bogue 2000 : dysfonctionnement de plusieurs systèmes informatiques.

  Cause : la donnée "année" était codée sur deux caractères, pour gagner un peu d'espace.

1.3 Le Génie Logiciel

Dans un souci de pallier les problèmes suscités et dans un souci d'augmenter la productivité des équipes de développement et d'améliorer la qualité des produits, des procédés de fabrication de ces différents logiciels ont été proposés sous le nom de Génie Logiciel afin de s'assurer que :

- ce qui est fabriqué par le maître d'œuvre répond aux besoins de celui qui les a formulés (le maître d'ouvrage ou le représentant du client final)
- Les coûts et les délais de réalisation restent dans les limites fixés au départ.
- Le contrat de service sera effectivement respecté (performance, sûreté de fonctionnement, sécurité, ...) lors de l'exploitation future du logiciel.
- Aussi, s'assurer de l'aptitude d'évolution lors de l'apparition de nouveaux besoins est une caractéristique importante ayant un effet direct sur la durée de vie du système d'où une

répercussion directe sur son coût d'amortissement.

Remarque 1.3. *Ce terme est né en 1968 (7-11 octobre 1968) en pleine période de crise de logiciel sous le nom anglo-saxon : software engineering et sous le parrainage de l'OTAN.*

Cette nouvelle discipline de l'ingénierie est donc concernée par le développement de logiciels de *qualité* et *coûts* raisonnables et dans les *délais* prévus. Pour ce faire, Elle s'occupe de l'étude de l'ensemble des méthodes et techniques qu'il faut réunir pour spécifier, construire, distribuer et maintenir les logiciels qui soient :

- **Sûrs** : réagissant de façon déterministe à toutes les sollicitations éventuellement entachées d'erreurs, qui correspondent à sa mission.
- **Conviviaux** : adaptés aux capacités réelles et non supposées des usagers (ergonomie).
- **Evolutifs** : s'adaptant aux nouveaux besoins dans des délais raisonnables.
- **Economiques** : réalisant l'optimum entre le service rendu et les coûts de développement et de la maintenance initialement annoncés.

1.3.1 Qualité exigée d'un logiciel

Si le génie logiciel est l'art de produire de *bons logiciels*, il est par conséquent nécessaire de fixer les critères de qualité d'un logiciel.

Qu'est ce qu'un logiciel ?

Par logiciel on n'entend pas seulement l'ensemble des programmes informatiques (du code) associés à une application ou à un produit, mais également un certain nombre de **documents** se rapportant à ces programmes et nécessaires à leur installation, utilisation, développement et maintenance : spécification, schémas conceptuels, jeux de tests, mode d'emploi, etc.

Pour les grands systèmes, l'effort nécessaire pour écrire cette documentation est souvent aussi grand que l'effort de développement des programmes eux-mêmes.

En plus des fonctionnalités requises, d'autres facteurs caractérisent la qualité d'un logiciel :

Fiabilité

- Fonctionnement du logiciel conforme aux besoins des clients dans les conditions d'utilisation spécifiées par ces derniers.
- Absence des bugs.
- Intégrité (aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisé).
- Précision.
- Tests réussis pour tous les cas.
- Uniformité de conduite.

Efficacité : Optimisation de la consommation des ressources, performances optimales :

- Economie de mémoire
- Rapidité d'exécution...

Ergonomie : Interface utilisateur s'adaptant aux capacités réelles des usagers

- Facilité d'utilisation. La facilité d'emploi est liée à la facilité d'apprentissage, d'utilisation, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.
- Accessibilité.
- Documentation pour l'utilisateur.

Facilité de Maintenance (maintenabilité) : les logiciels ayant une longue durée de vie, doivent être conçus et réalisés de façon à minimiser les coûts des corrections d'erreurs non décelés dans les phases antérieures ainsi que ceux des adaptations aux évolutions éventuelles. On en distingue ainsi 3 types :

1. *Maintenance corrective* : pour corriger les erreurs faites lors du développement et non décelées par les tests.
2. *Maintenance adaptative* : pour adapter les fonctionnalités du logiciel aux changements technologiques *soft* & *hard*.
3. *Maintenance évolutive* : pour réaliser et intégrer au logiciel de nouvelles fonctionnalités exigées par l'évolution de son environnement.

La maintenabilité exige que le développement du logiciel respecte certains critères (qualités auxquelles s'intéressent les développeurs)

- Conception Modulaire assurant un forte cohésion intra-modulaires et faible couplage inter-modulaires
- Structuration du code source
- Lisibilité du code source

- Extensibilité (Facilité d'extension)
- Documentation technique lisible, structurée, extensible ...
- Portabilité (Utilisation d'un langage standardisé, Indépendance vis-à-vis du matériel et du système d'exploitation).

D'autres qualités spécifiques : Sûreté de fonctionnement, sécurité informatique, ... Remarque : Il est difficile d'optimiser tous ces facteurs en même temps car certains s'excluent mutuellement.

Exemple 1.2. *offrir une meilleure interface utilisateur peut réduire l'efficacité. Il est clair que dans ce cas, on doit effectuer des choix antinomiques. On est amené à trouver des compromis entre ces objectifs lors de la définition des besoins du système logiciel.*

Exemple 1.3. *Dans les systèmes temps réel, l'efficacité est de première importance. Toutefois, ceci risque d'accroître les coûts de manière exponentielle.*

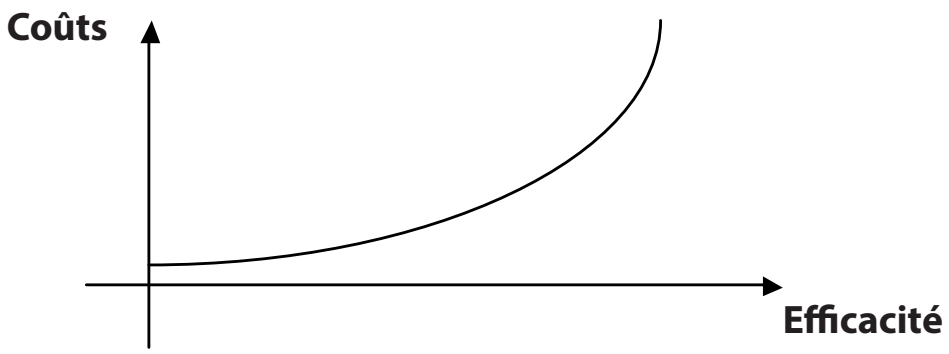

FIGURE 1.2:

Chapitre Deux

Cycle de vie d'un logiciel

2.1 Les étapes d'un cycle de vie de logiciels

Comme toute science de l'ingénieur, le génie logiciel perçoit la fabrication des programmes comme un processus à étapes. Le premier article important sur ce sujet fut celui de W.W Royce intitulé "*Managing the development of large software systems*", IEEE WESCON, en août 1970 où il a cherché à caractériser :

- Chacune des étapes successives de la fabrication depuis la commande du logiciel par le maître d'ouvrage jusqu'à sa mise en exploitation. L'ensemble de ces étapes forme ce qu'on appelle le cycle de vie du logiciel
- La logique d'enchaînement des différentes étapes.

2.2 Les différents modèles de cycle de vie

2.2.1 Modèle de la cascade

Royce a proposé un premier modèle du cycle de vie (dit de la cascade).

Caractéristiques du modèle :

- 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 et à chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables
- On ne passe à l'étape suivante que si les résultats de l'étape précédente sont jugés satisfaisants.
- L'activité d'une étape se réalise avec les résultats fournis par l'étape précédente ; ainsi, chaque étape sert de contrôle du travail effectué lors de l'étape précédente.
- Chaque phase ne peut remettre en cause que la phase précédente ce qui, dans la pratique, s'avère insuffisant.
- L'élaboration des spécifications est une phase particulièrement critique : les erreurs de spécifications sont généralement détectées au moment des tests, voire au moment de la livraison du logiciel à l'utilisateur. Leur correction nécessite alors de reprendre toutes les phases du processus.

Remarque 2.1. *Bien que les étapes de vie en cascade apparaissent comme un enchaînement séquentiel de phases distinctes, en réalité ces étapes se recouvrent et provoquent des retours d'information.*

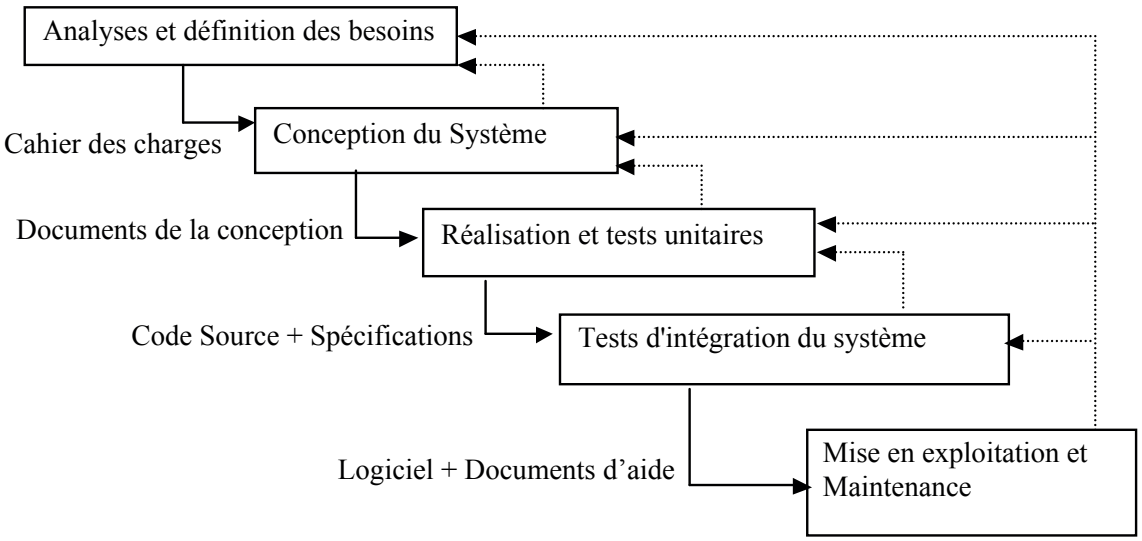

FIGURE 2.1: Le cycle de vie en " Cascade " (*waterfall model*)

Ce modèle est mieux adapté aux **petits projets** ou à ceux dont les spécifications sont bien connues et fixes. Depuis de nombreuses améliorations et modifications y ont été apportées.

Il existe de nombreux modèles de cycle de vie, les plus courants comportent les phases suivantes :

Analyse et définition des besoins : Cette étape comporte deux activités : *Analyse des besoins* et la *spécification globale*. Dans la 1ère activité, on étudie le domaine de l'application, ainsi que l'état actuel de l'environnement du futur système afin d'en déterminer les frontières, le rôle, les ressources disponibles et requises, les contraintes d'utilisation et de performance, etc. Cette étude est menée en collaboration avec les experts du domaine d'application et les futurs utilisateurs.

Quant à la *spécification*, son rôle est d'établir une première description du futur système en se basant sur les données de l'analyse des besoins ainsi que des considérations techniques et de faisabilité informatique. Son résultat est une description de ce que doit faire le logiciel (fonctionnalités) en évitant des décisions prématurées de réalisation.

Cette activité est fortement liée à l'analyse des besoins avec laquelle des échanges importants sont nécessaires. C'est pourquoi, ces deux activités sont souvent regroupées dans une même étape dont les documents produits forment le cahier des charges.

Conception du système : Après avoir décomposé le système en parties matériel et logiciel selon l'analyse des besoins, on s'occupe dans cette étape à enrichir la description du logiciel de détails d'implémentation afin d'aboutir à une description très proche d'un programme. Elle se déroule souvent en deux étapes : *conception architecturale* et *conception détaillée*. L'étape de *conception architecturale* consiste à décomposer le logiciel en composants plus simples en précisant les interfaces et les fonctions de chaque composant. L'étape de *conception détaillée* fournit pour chaque composant une description de la manière dont les fonctions du composant sont réalisées : algorithmes, représentations des données.

Réalisation et tests unitaires : Cette activité consiste à passer du résultat de la conception détaillée à un ensemble de programme ou de composants de programmes. Notons que tout en développant les composants d'un logiciel, les tests unitaires sont menés pour permettre de vérifier que ces unités répondent à leurs spécifications.

Tests d'intégration du système : L'intégration consiste à assembler les composants du logiciel pour obtenir un système exécutable (avec la possibilité de gérer plusieurs variantes du logiciel). Ensuite, on réalise des tests globaux pour être sûr que les besoins logiciels ont été satisfaits. Après quoi, le système est livré au client.

Mise en exploitation et Maintenance : Normalement c'est l'étape la plus longue dans le cycle de vie du logiciel. Une fois que le système est installé et mis en service, l'activité maintenance intervient pour corriger les erreurs éventuelles qui n'ont pas été découvertes lors des phases antérieures et/ou aussi améliorer la réalisation des unités du système et à augmenter ses fonctionnalités au fur et à mesure que de nouveaux besoins apparaissent.

Remarque 2.2. *L'un des buts de l'étape de tests est de s'assurer que le système construit répond aux attentes des utilisateurs. Cette activité s'appelle la* ***validation****. Par contre la* ***vérification*** *(l'autre activité apparentée à la validation) consiste à s'assurer que les descriptions successives du logiciel et le logiciel lui-même satisfont la spécification globale.*

Le tableau 2.2 (fourni par Boehm, 1975) constitue une approximation très réaliste des coûts relatifs à chaque étape :

Il ressort de ce tableau que les coûts les plus importants sont liés aux activités de spécification/conception et de tests. Une réduction des coûts de développement peut être obtenue en entreprenant des efforts supplémentaires au niveau de ces étapes de début et de fin de ce cycle.

Particularité de l'étape de maintenance :

– Cette étape peut entraîner des changements dans toutes les étapes précédentes : l'analyse des besoins, la conception et la réalisation du système ou mettre en évidence la nécessité de faire des tests supplémentaires.

TABLEAU 2.1: Approximation des coûts relatifs à chaque étape du cycle de vie

Type de système	Coût de la phase (%)		
	Besoins/Conception	Réalisation	Test
Systèmes de commande	46	20	34
Systèmes embarqués	34	20	46
Systèmes d'exploitation	33	17	50
Systèmes scientifiques	44	26	30
Systèmes de gestion	44	28	28

– Généralement les coûts de la maintenance des systèmes dont la durée de vie est longue, dépasse de loin les coûts de développement d'un facteur qui peut aller de 2 à 4.
Exemple : *le coût de développement d'un système avionique était de 30 dollars/instruction et le coût de maintenance de 4000 dollars/instruction*
– La plus grosse part de ces coûts provient des modifications des besoins et non pas des erreurs. D'où l'importance de couvrir au mieux la définition des besoins des utilisateurs. A cet effet, d'autres variantes du modèle du cycle de vie plus évolutives ont été proposées.

2.2.2 Modèle en V

Ce modèle rend explicite le fait que les premières étapes du développement qui ont trait surtout à la construction du logiciel, doivent préparer les dernières étapes qui font intervenir essentiellement les activités de validation et de vérification [1].

L'idée de ce modèle est qu'avec la décomposition, la recomposition doit être décrite et que toute description d'un composant est accompagnée des tests qui permettront de s'assurer qu'il correspond à description.

En plus des flèches continues qui reflètent l'enchaînement séquentiel des étapes du modèle de la cascade, on remarque l'ajout de flèches discontinues qui représentent le fait qu'une partie des résultats de l'étape de départ est utilisée directement par l'étape d'arrivée, par exemple, à l'issue de la conception architecturale, le protocole d'intégration et les jeux de test d'intégration doivent être complètement décrits.

L'avantage d'un tel modèle est d'éviter d'énoncer une propriété qu'il est impossible de vérifier objectivement une fois le logiciel réalisé.

Remarque 2.3. *Le cycle en V est le cycle qui a été normalisé, il est largement utilisé, notamment en informatique industrielle et télécoms. Idéal quand les besoins sont bien connus,*

1. Avec les jeux de tests préparés dans la première branche, les étapes de la deuxième branche peuvent être mieux préparées et planifiées.

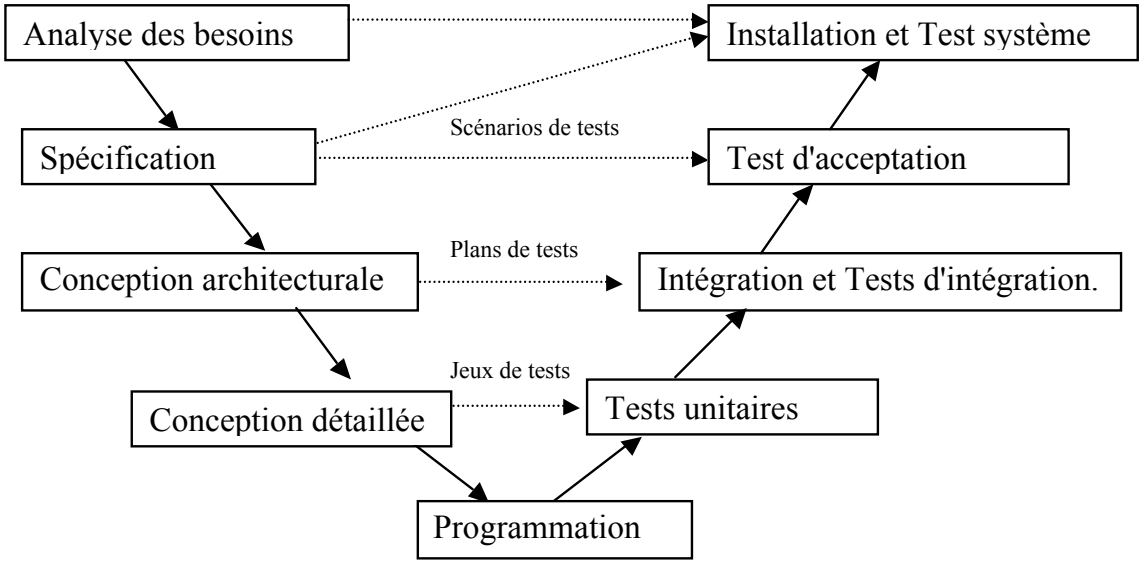

FIGURE 2.2: Le cycle de vie en V

quand l'analyse et la conception sont claires. Il est adapté aux projets de taille et de complexité moyenne.

2.2.3 Approches évolutives : Maquettage & Prototypage

La principale difficulté de l'activité de validation est due à l'imprécision des besoins et des caractéristiques du système à développer

2.2.3.1 Maquettes (ou prototype rapide, jetable)

Cette approche consiste donc à produire rapidement une *maquette* (*prototype jetable*) qui constitue une ébauche du futur système.

Avec cette approche, on est capable de définir plus explicitement et de manière plus cohérente les besoins des usagers, de détecter les fonctions manquantes et identifier et améliorer les fonctions complexes, comme elle permet de démontrer la faisabilité et l'utilité de l'application.

Le prototype est ensuite abandonné et le système réel est construit.

Remarque 2.4. *Jeter le prototype est dû aux imperfections propres à tout prototype car la rapidité de développement est souvent réalisée au détriment de la maintenabilité du logiciel (ce qui est intolérable pour les gros systèmes logiciels)*

Faut-il développer un prototype d'un futur système ou bien développer ce dernier et ensuite procéder aux modifications voulues ? Cela dépend en grande partie aux coûts inhérents à chaque approche.

2.2.3.2 Prototypes (non jetables)

Un prototype répond à des objectifs différents : on cherche à construire un système éventuellement très incomplet, mais dans son dimensionnement réel de façon à pouvoir faire des essais en vraie grandeur. On détermine d'abord un ensemble minimum de fonctions elles-mêmes incomplètes de façon à réaliser un premier incrément du logiciel dont on se sert pour analyser le comportement du logiciel. L'utilisateur fournit en retour des informations au concepteur qui modifie le prototype. Ce processus d'évolution de prototypes continue jusqu'à ce que l'utilisateur soit satisfait du système livré.

2.2.4 Modèle par incréments

Dans les modèles spirale, en V ou en cascade, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus.

Dans le modèle par incréments, seul un sous ensemble des composants d'un système est développé à la fois. Dans un premier temps un *logiciel noyau* est développé, puis successivement, les *incréments* sont développés et intégrés. Chaque incrément est développé selon l'un des modèles précédents. Notons qu'il peut aussi y avoir un chevauchement entre les différents processus de développement de ces incréments.

Les avantages de ce type de modèle sont :

- Chaque développement est moins complexe,
- Les intégrations sont progressives,
- Il peut y avoir des livraisons et des mises en services après chaque intégration d'incrément.
- Il permet de bien lisser (répartir) dans le temps l'effort de développement et les effectifs par rapport à ce qu'on rencontre dans les modèles en cascade et en V.

Néanmoins, ces modèles présentent des risques importants comme celui de voir remettre en cause le noyau ou les incréments précédents, un autre risque est d'être incapable d'intégrer un incrément.

Donc, son utilisation doit être faite avec précaution :

- La spécification du noyau, des incréments et de leurs interactions doit être faite globalement au début du projet.
- Les incréments doivent être aussi indépendants que possible aussi bien fonctionnellement qu'au niveau des calendriers de développement.

2.2.5 Modèle en Spirale

Pour corriger les travers de la démarche linéaire sont apparus des modèles dits en spirales, proposé par B. Boehm en 1988, où les risques, quels qu'ils soient, sont constamment traités au travers de bouclages successifs :

Chaque cycle de la spirale se déroule en quatre phases représentées par des quadrants :

1. Détermination des objectifs du cycle, des alternatives pour les atteindre, des contraintes, à partir des résultats des cycles précédents ou s'il n'y a pas, d'une analyse préliminaire des besoins,
2. Analyse des risques, évaluation des alternatives, éventuellement maquettage,
3. Développement et vérification de la solution retenue,
4. Revue des résultats et planification du cycle suivant.

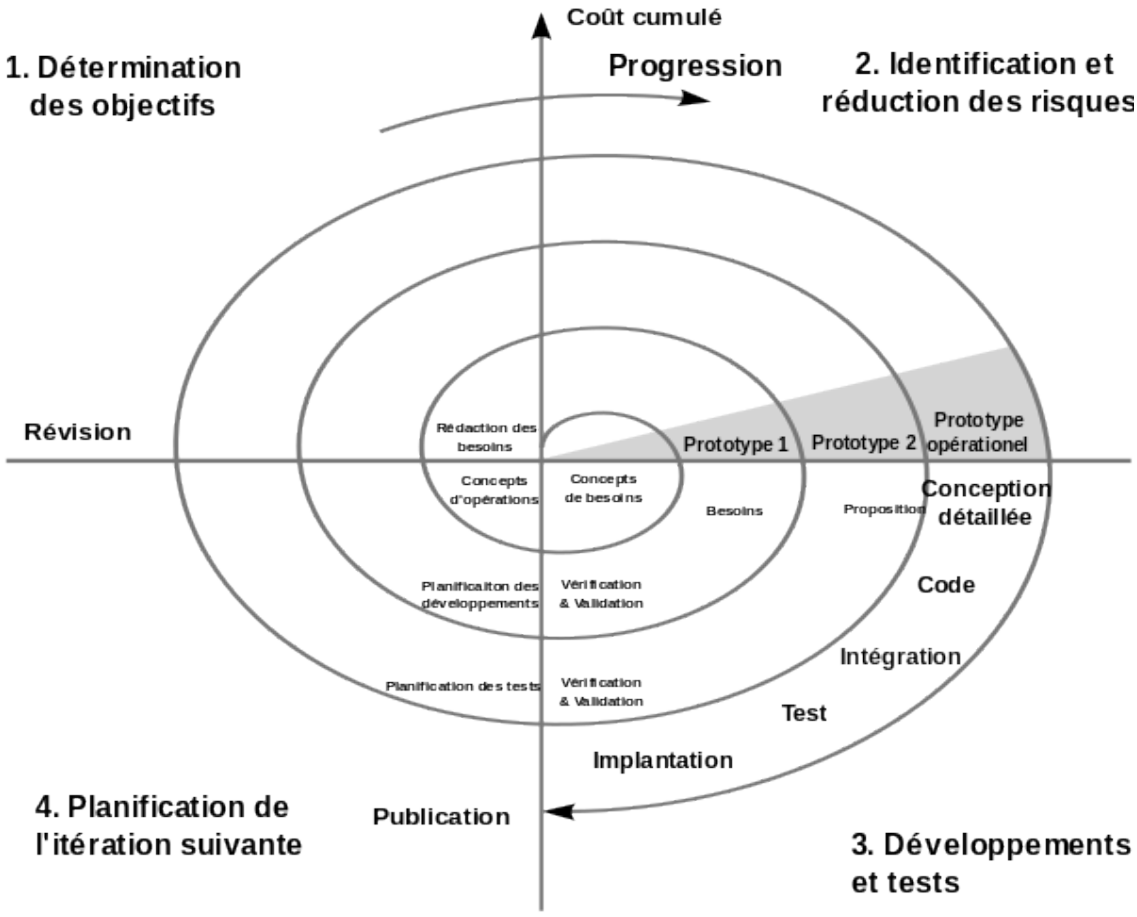

FIGURE 2.3: Le cycle de vie en spirale

Le quadrant 3 correspond à un développement ou à une portion de développement classique, et un des modèles précédents peut s'appliquer : son choix peut faire partie des alternatives à évaluer. L'originalité de ce "super" modèle est d'encadrer le développement proprement dit par des phases consacrées à la détermination des objectifs et à l'analyse de risque.

Un développement de ce modèle commence par une analyse préliminaire de besoins qui est affinée au cours des premiers cycles, en prenant en compte les contraintes et l'analyse des risques.

Le modèle utilise systématiquement des maquettes, qui durant ces cycles sont de nature exploratoire. Les troisièmes quadrants des cycles suivants correspondant à de la conception, les choix étant guidés par des maquettes expérimentales. Le dernier cycle se termine par la fin d'un processus classique. La mise en œuvre de modèle demande des compétences et un effort importants. En plus il s'agit d'un modèle moins expérimenté que les précédents et est moins documenté.

Tableau 2.2:

Les risques majeurs du développement de logiciel	**Solutions suggérées**
Défaillance du personnel	Embauche de personnel de haut niveau, adéquation entre profil et fonction, esprit d'équipe, formation mutuelle ...
Calendrier et budget irréalistes	Estimation détaillée des coûts et délais, développement incrémental, réutilisation ...
Développement de fonctions mal appropriées	Analyse de l'organisation, analyse de la mission, formulation des concepts opérationnels, revues d'utilisateurs, ...
Développement d'interfaces utilisateurs mal appropriées	Maquettage, scénarios et revues d'utilisateurs, analyse des tâches
Volatilité des besoins	Seuil élevé de modification, masquage d'information, développement incrémental où les derniers incréments sont les plus changeants.
Problèmes de performances	Simulations, modélisations, essais et mesures, maquettage.
Exigences démesurées par rapport à la technologie	Analyses techniques de faisabilité, maquettage.

2.2.6 Modèles hybrides

Il ne fait aucun doute que l'adoption d'une approche évolutive du développement de certaines classes de systèmes peut conduire à des produits qui répondent mieux aux besoins des utilisateurs. C'est le cas, en particulier, des nouvelles applications qui n'ont pas d'équivalant non automatique. Dans de tels cas, la formulation des besoins est en effet très difficile.

Cependant, pour les systèmes basés sur une procédure manuelle bien connue, il est moins évident que l'approche évolutive serait d'un bon rapport coût - performance. Boehm (1984) semble indiquer que le contrôle et l'intégration du changement, en particulier dans le cas des grands systèmes, sont souvent plus difficiles lorsqu'un modèle évolutif est utilisé. Il en conclut que la meilleure solution globale peut être composée de plusieurs approches, l'analyse du risque encouru déterminant l'approche adoptée pour chaque sous-système particulier :

- Utilisation du prototypage pour les spécifications à hauts risques
- Modèle classique pour les parties bien connues.