CNAM - Nancy
2003

Jacques Lonchamp

PREMIERE PARTIE

Définitions générales, principes, processus.

1. Définition et objectifs du génie logiciel (GL)

1.1. 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-version software »), et considérée comme ‘difficile’.

1.2. Objectifs – la règle du CQFD

Le GL se préoccupe des *procédés de fabrication des logiciels* de façon à s’assurer que les 4 critères suivants soient satisfaits.

- Le système qui est fabriqué répond aux *besoins* des utilisateurs (correction fonctionnelle).
- La *qualité* correspond au contrat de service initial. La qualité du logiciel est une notion multiforme qui recouvre :
  – 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 conte des accès ou des modifications non autorisés,
  – la *facilité d'utilisation*, *d’entretien*, etc.
- Les *coûts* restent dans les limites prévues au départ.
- Les *délais* restent dans les limites prévues au départ.

Règle du CQFD : Coût Qualité Fonctionnalités Délai.

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.

1.3. L'état des lieux

Le GL est apparu dans les années 70 pour répondre à la *'crise du logiciel'*. Cette crise est apparue lorsque l'on a pris conscience que le coût du logiciel dépassait le coût du matériel. Aujourd'hui il le dépasse très largement.

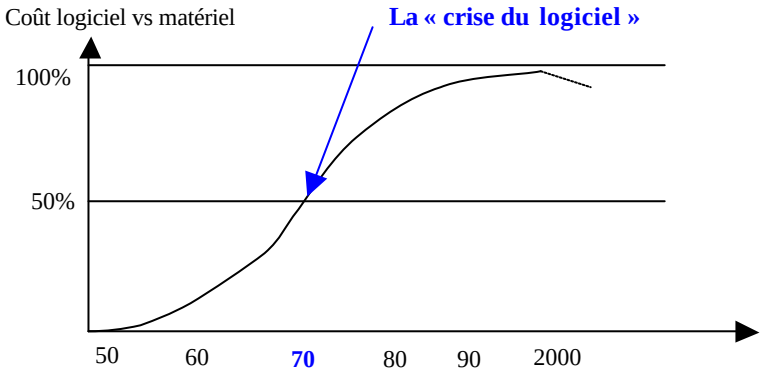


Figure 1.1 La crise du logiciel

Un autre symptôme de cette crise se situe dans la non qualité des systèmes produits. Les risques humains et économiques sont importants, comme l'illustrent les quelques exemples célèbres suivants :

- arrêt de Transpac pour 7.000 entreprises et 1.000.000 d'abonnés : surcharge du réseau,
- TAURUS, un projet d'informatisation de la bourse londonienne : définitivement abandonné après 4 années de travail et 100 millions de £ de pertes,
- mission VENUS : passage à 500.000 km au lieu de 5.000 km à cause du remplacement d'une virgule par un point,
- avion F16 déclaré sur le dos au passage de l'équateur : non prise en compte du référentiel hémisphère sud,
- avion C17 de McDonnell Douglas livré avec un dépassement de 500 millions de $ ... (19 calculateurs hétérogènes et 6 langages de programmation différents),
- faux départ de la première navette Colombus : manque de synchro entre calculateurs assurant la redondance (un délai modifié de 50ms à 80 ms entraînant une chance sur 67 d'annulation par erreur de la procédure de tir),
- non reconnaissance de l'EXOCET dans la guerre des Malouines : Exocet non répertorié comme missile ennemi : 88 morts,
- non différenciation entre avion civil et avion militaire : guerre du Golfe - Airbus iranien abattu : 280 morts,
- mauvais pilote automatique de la commande d'une bombe au cobalt en milieu hospitalier : 6 morts,
- échec d'Ariane 501 (cf. exercice 1.1 ci -dessous).

D'après le cabinet de conseil en technologies de l'information Standish Group International, les pannes causées par des problèmes de logiciel ont coûté l'an dernier aux entreprises du monde entier environ 175 milliards de dollars, soit deux fois plus au moins qu'il y a 2 ans (Le Monde 23/10/01).

La solution imaginée pour répondre à cette crise à été *l'industrialisation de la production du logiciel* : organisation des *procédés* de production (cycle de vie, méthodes, notations, outils), des *équipes* de développement, *plan qualité* rigoureux, etc. Malgré tout le GL reste aujourd'hui moins avancé (formalisé, mathématique) et moins bien codifié que d'autres 'sciences de l'ingénieur', comme le génie civil qui construit des routes et des ponts, ou le génie chimique. Une des raisons est la *nature même du logiciel* :

- le logiciel est un objet immatériel (pendant son développement), très malléable au sens de facile à modifier (cf. « hardware/software »),
- s*es caractéristiques attendues sont difficiles à figer au départ et souvent remises en cause en cours de développement,*
- les défaillances et erreurs ne proviennent ni de défauts dans les matériaux ni de phénomènes d'usure dont on connaît les lois mais *d'erreurs humaines,* inhérentes à l'activité de développement,
- le logiciel ne s'use pas, il devient *obsolète* (par rapport aux concurrents, par rapport au contexte technique, par rapport aux autres logiciels, ...),
- le développement par *assemblage de composants* est encore balbutiant dans le domaine logiciel (beans, EJB, composants CORBA, ...).

Cependant, grâce au GL des *progrès ont été réalisés*. Mais la complexité des systèmes ne cesse de s'accroître.

Exemple :

Compilateur C : 10 HA (Homme/années), 20 à 30 KLS (milliers de lignes source)
Compilateur Ada : 120 à 150 HA
Logiciel 2D/3D : 50 à 150 HA, >200KLS
SGBD (Oracle, DB2) : 300 à 500 HA
Navette spatiale : > 1000 HA, 2 200 KLS, 6 ans, langage HAL

Les exigences varient selon le type des logiciels.

Exemple : défauts résiduels

Navette spatiale :
- logiciels embarqués : 0,1 défaut par KLS (1/10000 lignes)
- logiciels au sol : 0,4 défaut par KLS (1/2500)

1.4 Caractéristiques

Le GL est en forte relation avec presque tous les autres domaines de l'informatique : langages de programmation (modularité, orientation objet, parallélisme, ...), bases de données (modélisation des données, de leur dynamique, ...), informatique théorique (automates, réseaux de Petri, types abstraits, ...), etc. Comme le génie chimique utilise la chimie comme un outil pour résoudre des problèmes de production industrielle, le GL utilise l'informatique comme un *outil* pour résoudre des problèmes de traitement de l'information. Le GL est aussi en relation avec d'autres disciplines de l'ingénieur : ingénierie des systèmes et gestion de projets, sûreté et fiabilité des systèmes, etc. Les principales branches du GL couvrent :

- la conception,
- la validation/vérification,
- la gestion de projet et l'assurance qualité,
- les aspects socio-économiques.

Dans sa partie technique, le GL présente un spectre très large depuis des approches très *formelles* (spécifications formelles, approches transformationnelles, preuves de programmes) jusqu'à des démarches absolument *empiriques*. Cette variété reflète la variété des types de systèmes à produire :

- gros systèmes de gestion (systèmes d'information) ; le plus souvent des systèmes transactionnels construits autour d'une base de donnée centrale ;
- systèmes temps-réel, qui doivent répondre à des événements dans des limites de temps prédéfinies et strictes ;
- systèmes distribués sur un réseau de machines (distribution des données et/ou des traitements), 'nouvelles architectures' liées à Internet ;
- systèmes embarqués et systèmes critiques, interfacés avec un système à contrôler (ex: aéronautique, centrales nucléaires, ...).

1.5. Plan du cours

Le GL est difficile à enseigner car très vaste, pas toujours très précis (beaucoup de discours généraux), foisonnant dans les concepts et le vocabulaire, sensible aux effets de modes. Les aspects techniques nécessitent *une bonne maîtrise des outils fondamentaux de l'informatique (programmation, BD, système/réseau, ...).*

Le discours peut s'organiser selon quatre niveaux :

- quelques *principes* généraux,
- des *techniques* spécialisées, qui s'appuient sur ces principes (il en existe des dizaines),
- des *méthodes*, qui relient plusieurs techniques en un tout cohérent et organisé (il en existe des centaines),
- des *outils*, qui supportent les méthodes (il en existe des milliers). Les plus complexes sont appelés ateliers (ou environnements) de GL.

Le cours commence par une courte partie sur les principes et sur leur traduction dans les modèles de cycle de vie du logiciel.
La partie la plus importante est consacrée aux techniques (survol des principales techniques de spécification, de conception, de vérification).
Une méthode est ensuite étudiée (UML). On y retrouve certaines techniques évoquées précédemment.
Enfin, les questions liées à la gestion du développement des logiciels sont abordées brièvement à la fin du cours (aspects humains et économiques).

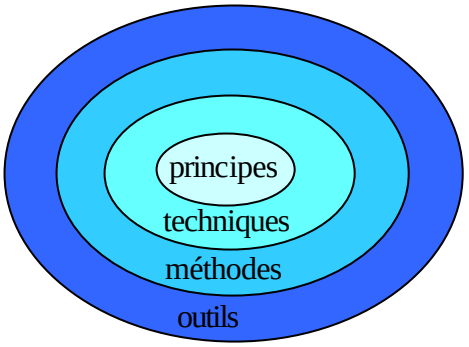


Figure 1.2 Les 4 niveaux de discours

2. Les principes

Cette partie liste sept principes fondamentaux (proposés par Carlo Ghezzi):

- rigueur,
- séparation des problèmes (« separation of concerns »),
- modularité,
- abstraction,
- anticipation du changement,
- généricité,
- construction incrémentale.

2.1. Rigueur

La production de logiciel est une activité créative, mais qui doit se conduire avec une certaine rigueur. Certains opposent parfois créativité et rigueur. Il n'y a pas contradiction : par exemple, le résultat d'une activité de création pure peut être évalué rigoureusement, avec des critères précis.

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.

2.2. "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. C'est un aspect de la stratégie générale du « diviser pour régner ».

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),
- séparations 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.

Les méthodes aident à organiser le travail en guidant dans la séparation et l'ordonnancement des questions à traiter.

2.3. Modularité

Un système est modulaire s'il est composé de *sous-systèmes plus simples*, ou modules. La modularité est une propriété importante de tous les procédés et produits industriels (cf. l'industrie automobile ou le produit et le procédé sont très structurés et modulaires).

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 biens 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).

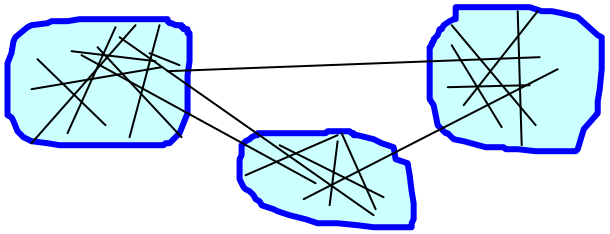


Fig. 2.1. La modularité

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

2.4. 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), ou par un plan physique de composants réels au sein d'un circuit intégré. L'abstraction permet une meilleure maîtrise de la complexité.

2.5. 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),
- être capable de gérer les multiples versions des modules et configurations des versions des modules, constituant des versions du produit complet.

2.6. 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).

2.7. 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.

2.8. Conclusion

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 (on le verra dans la suite du cours) et constituent le type de connaissance le plus stable, dans un domaine où les outils, les méthodes et les techniques évoluent très vite.

3. Les processus (modèles de cycle de vie)

3.1. Définitions

Comme pour toutes les fabrications, il est important d'avoir un procédé de fabrication du logiciel bien défini et explicitement décrit et documenté. En GL, il s'agit d'un type de fabrication un peu particulier : en un seul exemplaire, car la production en série est triviale (recopie).

Les *modèles de cycle de vie du logiciel* décrivent à un niveau très abstrait et *idéalisé* les différentes manières d'organiser la production. Les *étapes*, leur ordonnancement, et parfois les critères pour passer d'une étape à une autre sont explicités (critères de terminaison d'une étape - revue de documents -, critères de choix de l'étape suivante, critères de démarrage d'une étape).

Il faut souligner la différence entre étapes (découpage temporel) et *activités* (découpage selon la nature du travail). Il y a des activités qui se déroulent dans plusieurs étapes (ex : la spécification, la validation et la vérification), voire dans toutes les étapes (ex : la documentation).

Rappelons aussi la différence entre *vérification* et *validation* (B. Boehm, 76) :

- vérification : *« are we building the product right ? »* (« construisons nous le produit correctement ? » - correction interne du logiciel, concerne les développeurs),
- validation : *« are we building the right product »* (« construisons nous le bon produit ? » - adaptation vis à vis des besoins des utilisateurs, concerne les utilisateurs).

3.2. Le modèle en cascade ('waterfall')

Historiquement, la première tentative pour mettre de la rigueur dans le 'développement sauvage' (coder et corriger ou 'code and fix') a consisté à distinguer une phase d'*analyse* avant la phase d'*implantation* (séparation des questions).

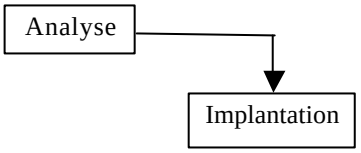


Fig.3.1 Le modèle primitif à 2 phases

Très vite, pendant les années 70, on s'est aperçu qu'un plus grand nombre d'étapes étaient nécessaires pour organiser le développement des applications complexes. Il faut en particulier distinguer l'analyse du '*quoi faire ?*' qui doit être validée par rapport aux objectifs poursuivis et la conception du '*comment faire?*' qui doit être vérifiée pour sa cohérence et sa complétude. Le *modèle en cascade* (Fig. 3.2) décrit cette succession (plus ou moins détaillée) d'étapes ; sont représentées ici huit étapes fondamentales.

Même si on l'étend avec des possibilités de retour en arrière, idéalement limitées à la seule phase qui précède celle remise en cause (Fig. 3.3), le développement reste fondamentalement linéaire. En particulier, il se fonde sur *l'hypothèse souvent irréaliste que l'on peut dès le départ définir complètement et en détail ce qu'on veut*

réaliser (‘requirements’ ou expressions des besoins). La pratique montre que c’est rarement le cas.

Même si elle n’est pas réaliste, cette représentation très simplifiée a permis de définir des *cadres conceptuels* (définition des différentes phases - cf. Fig. 3.4) *et terminologiques***,** largement acceptés et *normalisés* par plusieurs organismes (ISO, AFNOR, IEEE, DOD pour les applications militaires aux USA, ESA, etc.). Ceci facilite la gestion et le suivi des projets.

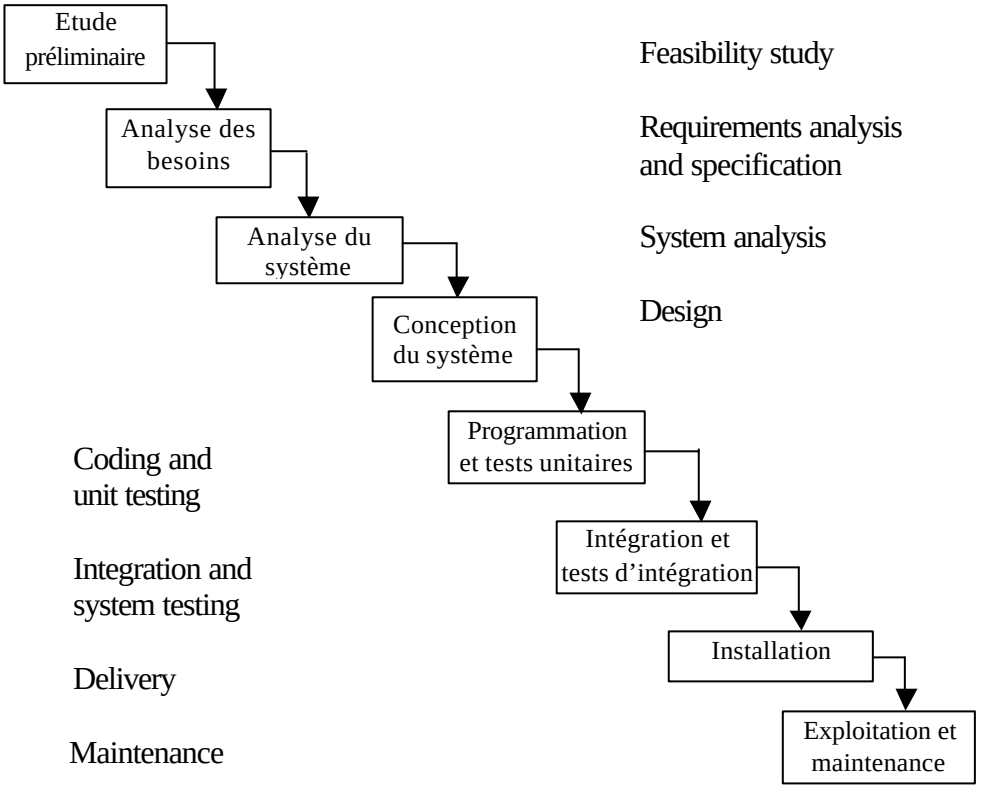


Fig.3.2 Le modèle en cascade

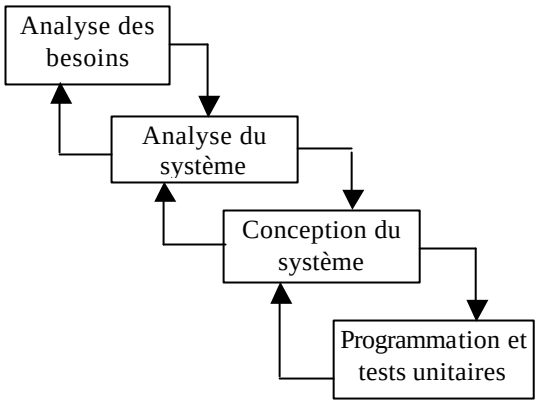


Fig.3.3 Le modèle en cascade avec itérations (quelques étapes)

Etude préliminaire ou étude de faisabilité ou planification : (rapport d’analyse préliminaire ou schéma directeur)

- définition globale du problème,
- *différentes stratégies possibles avec avantages/inconvénients,*
- ressources, coûts, délais.

Analyse des besoins ou analyse préalable : (cahier des charges + plan qualité)
- qualités fonctionnelles attendues en termes des *services offerts*,
- *qualités non fonctionnelles attendues* : efficacité, sûreté, sécurité, facilité d'utilisation, portabilité, etc.
- qualités attendues du procédé de développement (ex : procédures de contrôle qualité).

Le cahier des charges peut inclure une partie destinée aux clients (définition de ce que peuvent attendre les clients) et une partie destinée aux concepteurs (spécification des besoins).

Analyse du système : (dossier d'analyse + plan de validation)
- modélisation du domaine,
- modélisation de l'existant (éventuellement),
- *définition d'un modèle conceptuel* (ou spécification conceptuelle),
- plan de validation.

Conception : (dossier de conception + plan de test global et par module)
- proposition de solution au problème spécifié dans l'analyse
- organisation de l'application en *modules et interface des modules* (architecture du logiciel),
- description détaillée des modules avec les *algorithmes essentiels* (modèle logique)
- *structuration des données*.

Programmation et tests unitaires : (dossiers de programmation et codes sources)
- traduction dans un langage de programmation,
- tests avec les jeux d'essais par module selon le plan de test.

Intégration et tests de qualification :
- composition progressive des modules,
- tests des regroupements de modules,
- test en vraie grandeur du système complet selon le plan de test global ('alpha testing').

Installation :
Mise en fonctionnement opérationnel chez les utilisateurs. Parfois restreint dans un premier temps à des utilisateurs sélectionnés ('beta testing').

Maintenance :
- maintenance corrective (ou curative),
- maintenance adaptative,
- maintenance perfective.

+ Activités transversales à tout le cycle de vie :
- spécification, documentation, validation et vérification, management.

Fig. 3.4 Un cadre conceptuel et terminologique

Des études ont été menées pour évaluer le coût des différentes étapes du développement:

Type de système	Conception	Implantation	Test
Gestion	44	28	28
Scientifique	44	26	30
Industriel	46	20	34

Mais c'est la maintenance qui coûte le plus cher.

3.3. Le modèle en V

Le *modèle en V* (Fig. 3.5) est une autre façon de présenter une démarche qui reste linéaire, mais qui fait mieux apparaître les produits intermédiaires à des *niveaux*

d'abstraction et de formalité différents et les *procédures d'acceptation (validation et vérification) de ces produits intermédiaires*.

Le V est parcouru de gauche à droite en suivant la forme de la lettre : les activités de construction précèdent les activités de validation et vérification. Mais l'acceptation est préparée dès la construction (flèches de gauche à droite). Cela permet de mieux approfondir la construction et de mieux planifier la 'remontée'.

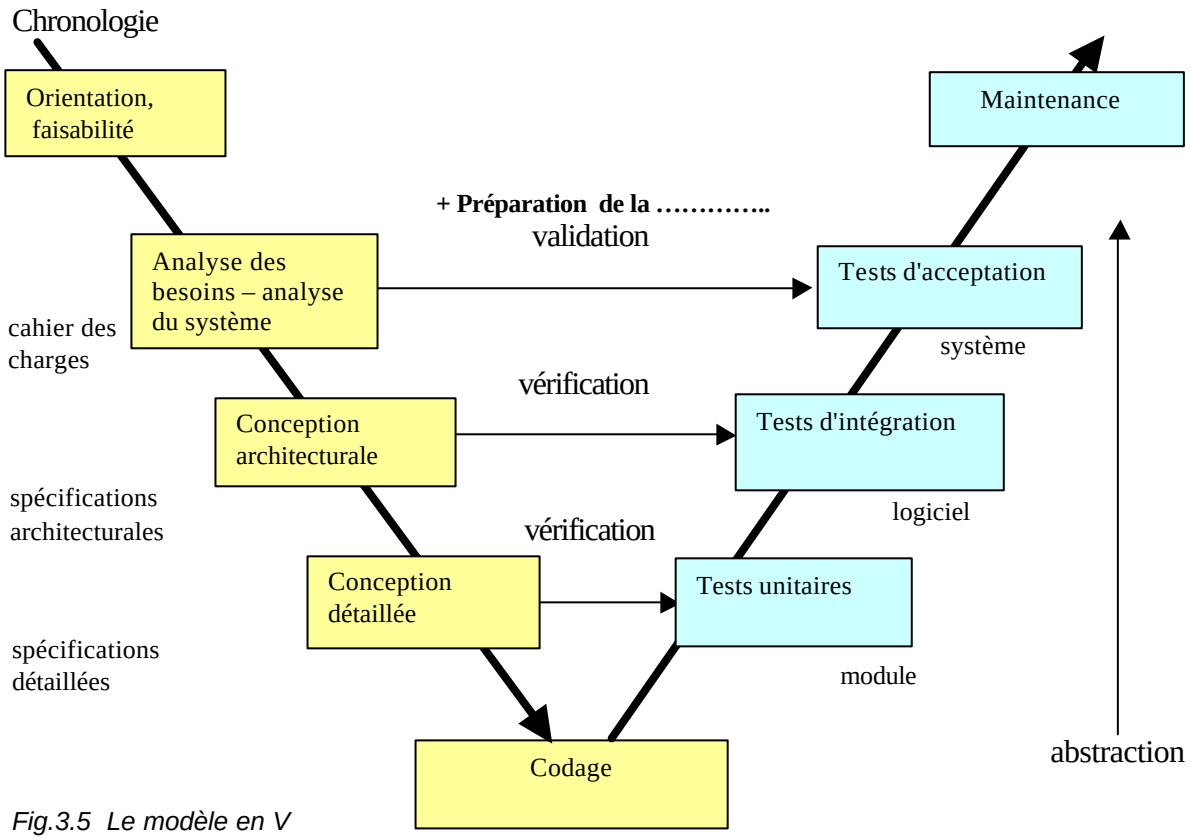


Fig.3.5 Le modèle en V

3.3. 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 (Fig. 3.6). Le produit est délivré en plusieurs fois, de manière incrémentale, c'est à dire en le complétant au fur et à mesure et en profitant de l'expérimentation opérationnelle des incréments précédents. *Chaque incrément peut donner lieu à un cycle de vie classique plus ou moins complet.* Les premiers incréments peuvent être des *maquettes* (jetables s'il s'agit juste de comprendre les besoins des utilisateurs) ou des *prototypes* (réutilisables pour passer au prochain incrément en les complétant et/ou en optimisant leur implantation). Le risque de cette approche est celui de la remise en cause du noyau.

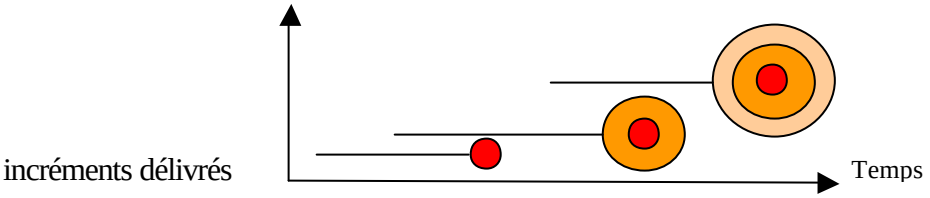


Fig.3.6. Le modèle incrémental

Le modèle de Balzer associe développement incrémental et utilisation de *spécifications formalisées, elles même développées* de manière incrémentale et maintenues (Fig. 3.7).

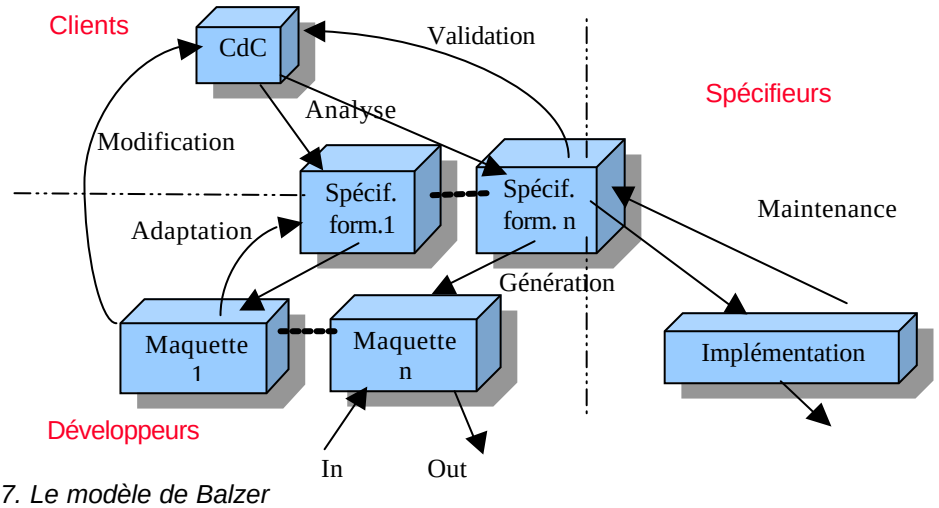


Fig. 3.7. Le modèle de Balzer

3.4. Le modèle en spirale

Enfin le *modèle en spirale,* de Boehm (88), met l'accent sur l'évaluation des risques (Fig. 3.8). 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.

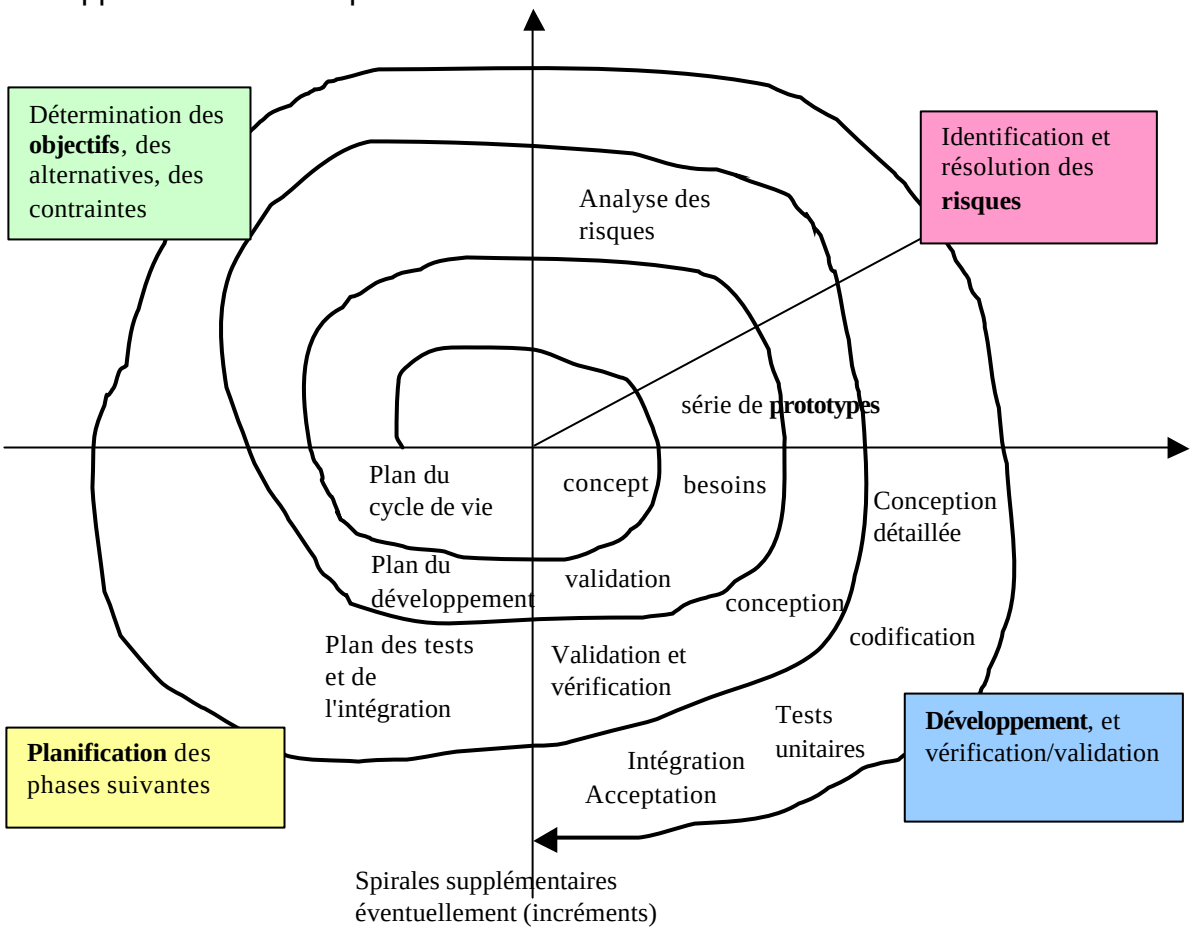


Fig.3.8 Le modèle en spirale

Les *principaux risques et leurs remèdes*, tels que définis par Boëhm, sont les suivants :

- défaillance de personnel : embauches de haut niveau, formation mutuelle, leaders, adéquation profil/fonction, ...
- calendrier et budgets irréalistes : estimation détaillée, développement incrémental, réutilisation, élagage des besoins, ...
- développement de fonctions inappropriées : revues d'utilisateurs, manuel d'utilisation précoce, ...
- développement d'interfaces utilisateurs inappropriées : maquettage, analyse des tâches, .
- produit 'plaqué or' : analyse des coûts/bénéfices, conception tenant compte des coûts, ...
- volatilité des besoins : développement incrémental de la partie la plus stable d'abord, masquage d'information, ...
- problèmes de performances : simulations, modélisations, essais et mesures, maquettage,
- exigences démesurées par rapport à la technologie : analyses techniques de faisabilité, maquettage, ...
- tâches ou composants externes défaillants : audit des sous-traitants, contrats, revues, analyse de compatibilité, essais et mesures, ...

3.5. Les processus réels

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 inadéquats et souvent remis en cause. Le modèle incrémental est risqué car il ne donne pas beaucoup de visibilité sur le processus complet. Le modèle de Balzer est risqué car il exige des spécialistes de très haut niveau. 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.

3.6. Autres concepts liés aux processus

3.6.1. La ré ingénierie des systèmes

Il s'agit de *"retraiter" ou de recycler* des logiciels en fin de vie. Les "vieux" logiciels sont un capital fonctionnel qui peut être de grande valeur par leur conception et leur architecture prouvées. Le code source est adaptable aux nouvelles technologies :

- migration de chaînes de traitements batch vers du transactionnel,
- fichiers à bases de données,
- intégration de composants standards (progiciels),
- migration de transactionnel vers du client-serveur ('down-sizing'),
- migration du client-serveur vers des architectures à 3 niveaux (ex : clients légers sur le Web),

Il existe des outils (traducteurs de code source, fichiers à BD, réorganisation de BD, etc.)

3.6.2. La normalisation des processus

De nombreuses normes sont apparues dans les années 90 pour évaluer les processus en fonction de normes de qualité. Les sociétés sont certifiées en fonction de leur respect de ces normes.

a) le standard CMM du SEI (Software Engineering Institute du DoD - Department Of Defense des USA) :

Le niveau de *maturité du processus de développement* (CMM) se décline en 5 niveaux de maturité pour les organisations qui développent du logiciel :

- niveau 1 : 'initial'. Processus chaotique, c'est à dire non discipliné et non prédictible. Coûts, délais, qualité non maîtrisés. A traiter en priorité : gestion de projet, assurance qualité, gestion de configurations.
- niveau 2: 'repeatable'. Processus reproductible du point de vue de la gestion de projet, avec des estimations raisonnables en main d'oeuvre et en temps pour la classe de projets considérée. Processus qui reste artisanal et très lié aux individus. Délais fiables, mais qualité et coût variables. A traiter en priorité : méthodes et techniques d'analyse et conception, revues par des pairs, formation, renforcement des contrôles qualitatifs.
- niveau 3: 'defined'. Processus défini aussi bien dans les aspects gestion de projet que dans les aspects ingénierie. Délais et coûts assez fiables. Qualité variable. Définition et suivi essentiellement de nature qualitative. A traiter en priorité : renforcement des contrôles quantitatifs (analyse et mesures des produits et des processus).
- niveau 4: 'managed'. Processus géré, c'est à dire contrôlé et mesuré. Qualité fiable. A traiter en priorité : gestion du changement (processus, technologies), prévention des défauts.
- niveau 5: 'optimizing'. Processus en adaptation continue (CPI) : chaque projet est analysé pour améliorer le processus et donc les coûts, délais et qualité.

Le niveau d'une organisation est évalué par des questionnaires, des entretiens, et des examens de documents.

b) les normes ISO 9000 (9003) et ISO SPICE (issue du CMM et de ISO 9000 et orientée vers le GL) attestent qu'une entreprise suit un processus orienté qualité. Cela ne donne pas de garantie sur la qualité du produit lui même.

Une lecture plus 'stratégique' de ces normes les lie à une certaine volonté de protectionnisme. En effet, peu d'entreprises américaines sont certifiées ISO 9000 (ou ses dérivés) et peu d'entreprises européennes sont certifiées CMM.

3.6.3. Les métriques

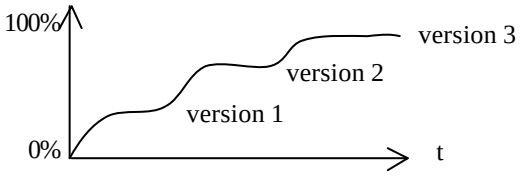
Pour l'IEEE (Institute of Electrical and Electronical Engineers) , le GL est intimement lié à l'idée de mesure : le GL est «l'application au développement, à la mise en oeuvre et à la maintenance du logiciel d'une approche systématique, disciplinée et *mesurable ;* en fait l'application des méthodes de l'ingénieur au logiciel ». La mesure est incontournable. Pour Harrington « si tu ne peux pas le mesurer, tu ne peux pas le contrôler ; si tu ne peux pas le contrôler, tu ne peux pas le gérer ; si tu ne peux pas le gérer, tu ne peux pas l'améliorer ».

Les mesures peuvent porter sur les *processus* et sur les *produits.*

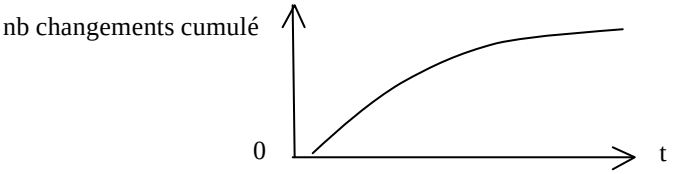
Sur les produits, les mesures sont le plus souvent statiques (sans exécution) ; on peut citer à titre d'exemples pour les approches à objets (et parmi une profusion de métriques) : le nombre et la complexité des méthodes pour implanter une classe, la profondeur de l'arbre d'héritage, le nombre de couplages entre classes (appels de méthodes ou accès aux instances), le nombre de méthodes qui peuvent être appelées en réponse à l'appel d'une méthode, etc.

Sur les processus on mesure :

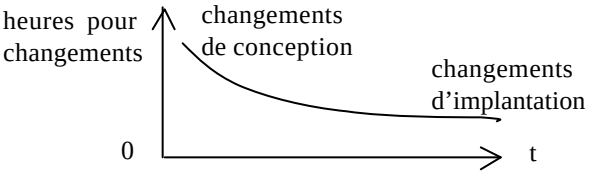
- l'avancement



- la stabilité (nombre de changements par période)



- l'adaptabilité ou effort pour effectuer les changements qui doit diminuer



- la qualité, via les mesures d'erreurs, etc.

EXERCICES

Exercice 1.1 *A propos de la difficulté de développer des systèmes complexes.*

Lire le communiqué officiel expliquant la désintégration du premier exemplaire de la fusée Ariane 5, paru sur Internet. Analyser la cascade d'erreurs qui a été commise.

Communiqué de presse conjoint ESA-CNES
Paris, le 19 juillet 1996

Rapport de la Commission d'enquête Ariane 501

Echec du vol Ariane 5Ol
Président de la Commission: Professeur J.-L LIONS

2 ANALYSE DE L'ECHEC

2.1. SÉQUENCE DES ÉVÉNEMENTS TECHNIQUES

De manière générale la chaîne de pilotage d'Ariane 5 repose sur un concept standard. L'attitude du lanceur et ses mouvements sont mesurés par un système de référence inertielle (SRI) dont le calculateur interne calcule les angles et les vitesses sur la base d'informations provenant d'une plate-forme inertielle à composants liés, avec gyrolasers et accéléromètres. Les données du SRI sont transmises via le bus de données, au calculateur embarqué (OBC) qui exécute le programme de vol et qui commande les tuyères des étages d'accélération à poudre et du moteur cryotechnique Vulcain , par l'intermédiaire de servovalves et de vérins hydrauliques.

Pour améliorer la fiabilité, on a prévu une importante redondance au niveau des équipements. On compte deux SRI travaillant en parallèle; ces systèmes sont identiques tant sur le plan du matériel que sur celui du logiciel. L'un est actif et l'autre est en mode "veille active" ; si l'OBC détecte que le SRI actif est en panne, il passe immédiatement sur l'autre SRI à condition que ce dernier fonctionne correctement. De même on compte deux OBC et un certain nombre d'autres unités de la chaîne de pilotage qui sont également dupliquées.

La conception du SRI d'Ariane 5 est pratiquement la même que celle d'un SRI qui est actuellement utilisé à bord d'Ariane 4, notamment pour ce qui est du logiciel.

Sur la base de la documentation et des données exhaustives relatives à l'échec d'Ariane 501 qui ont été mises à la disposition de la Commission, on a pu établir la séquence suivante d'événements ainsi que leurs interdépendances et leurs origines, depuis la destruction du lanceur jusqu'à la cause principale en remontant dans le temps.

- Le lanceur a commencé à se désintégrer à environ HO + 39 secondes sous l'effet de charges aérodynamiques élevées dues à un angle d'attaque de plus de 20 degrés qui a provoqué la séparation des étages d'accélération à poudre de l'étage principal, événement qui a déclenché à son tour le système d'auto destruction du lanceur;

- Cet angle d'attaque avait pour origine le braquage en butée des tuyères des moteurs à propergols solides et du moteur principal Vulcain;
- le braquage des tuyères a été commandé par le logiciel du calculateur de bord (OBC) agissant sur la base des données transmises par le système de référence inertielle actif (SRI2). A cet instant, une partie de ces données ne contenait pas des données du vol proprement dites mais affichait un profil de bit spécifique de la panne du calculateur du SRI2 qui a été interprété comme étant des données de vol ;
- la raison pour laquelle le SRI2 actif n'a pas transmis des données d'attitude correctes tient au fait que l'unité avait déclaré une panne due à une exception logiciel;
- l'OBC n'a pas pu basculer sur le SRI 1 de secours car cette unité avait déjà cessé de fonctionner durant le précédent cycle de données (période de 72 millisecondes) pour la même raison que le SRI2;
- l'exception logicielle interne du SRI s'est produite pendant une conversion de données de représentation flottante à 64 bits an valeurs entières à 16 bits. Le nombre en représentation flottante qui a été converti avait une valeur qui était supérieure à ce que pouvait exprimer un nombre entier à 16 bits. Il en est résulté une erreur d'opérande. Les instructions de conversion de données (en code Ada) n'étaient pas protégées contre le déclenchement d'une erreur d'opérande bien que d'autres conversions de variables comparables présentes à la même place dans le code aient été protégées;
- l'erreur s'est produite dans une partie du logiciel qui n'assure que l'alignement de la plate-forme inertielle à composants liés. Ce module de logiciel calcule des résultats significatifs avant le décollage seulement. Dès que le lanceur décolle, cette fonction n'est plus d'aucune utilité;
- La fonction d'alignement est active pendant 50 secondes après le démarrage du mode vol des SRI qui se produit à HO - 3 secondes pour Ariane 5. En conséquence, lorsque le décollage a eu lieu, cette fonction se poursuit pendant environ 40 secondes de vol. Cette séquence est une exigence Ariane 4 mais n'est pas demandé sur Ariane 5 ;
- L'erreur d'opérande s'est produite sous l'effet d'une valeur élevée non prévue d'un résultat de la fonction d'alignement interne appelé BH (Biais Horizontal) et lié à la vitesse horizontale détectée par la plate-forme. Le calcul de cette valeur sert d'indicateur pour la précision de l'alignement en fonction du temps;
- la valeur BH était nettement plus élevée que la valeur escomptée car la première partie de la trajectoire d'Ariane 5 diffère de celle d'Ariane 4, ce qui se traduit par des valeurs de vitesse horizontale considérablement supérieures.

Les événements internes du SRI qui ont conduit à l'accident ont été reproduits par simulation. En outre, les deux SRI ont été récupérés pendant l'enquête de la Commission et le contexte de l'accident a été déterminé avec précision à partir de la lecture des mémoires. De plus, la Commission a examiné le code logiciel qui s'est avéré correspondre au scénario de l'échec. Les résultats de ces examens sont exposés dans le Rapport technique.

On peut donc raisonnablement affirmer que la séquence d'événements ci-dessus traduit les causes techniques de l'échec d'Ariane 501.

Exercice 1.2 *A propos de la difficulté de spécifier précisément un besoin fonctionnel.*

La spécification de la règle de notation à un examen est la suivante :
« L'examen est un ensemble de 20 questions à réponses multiples. Chaque bonne réponse à une question rapporte 1 point. Chaque mauvaise réponse fait perdre 1/3 de point. Chaque question sans réponse donne 0 point. »
Pensez vous que cette spécification est claire ?
Pour le vérifier, calculez chacun la note des 3 étudiants suivants :

	réponses correctes	incorrectes	sans	doubles réponses
Duchnock	10	5	5	
Dutif	4	16		
Dudule	10	3	4	3 (1 juste, 1 fausse)

Recensez les résultats possibles.
Donnez une spécification plus précise.