M2P GLRE

Génie Logiciel, logiciels Répartis et Embarqués

Introduction aux méthodes et cycle de développement du logiciel

Z. Mammeri

1. Généralités sur les objectifs du génie logiciel

1.1. Définition

Le Génie logiciel ('Software Engineering' en anglais), GL, est un 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.* Aujourd'hui, les économies de tous les pays développés sont dépendantes des systèmes logiciels. Par conséquent, l'ingénierie du logiciel a une place importante et une lourde responsabilité dans le bon fonctionnement des équipements et des institutions. Par exemple, la réservation de billets de train est impossible, la faute est imputée au logiciel. Les avions ne peuvent plus atterrir, la faute est imputée au logiciel de commande des radars... et on peut multiplier ainsi les exemples. Les fautes du logiciel ont de plus en plus de conséquences visibles même pour le grand public.

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*, et considérée comme 'difficile'.

1.2. Objectifs

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

- Le système qui est fabriqué répond aux *besoins* (*exigences*) des utilisateurs (correction fonctionnelle).
- La *qualité* correspond au contrat de service initial. La qualité du logiciel est une notion multiforme qui recouvre notamment :
  - 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 (mémoire, CPU...),
  - 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.
- Les *coûts* restent dans les limites prévues au départ.
- Les *délais* restent dans les limites prévues au départ.

Ces qualités sont parfois *contradictoires*. Il faut les *pondérer* selon les types d'utilisation. Il faut aussi distinguer les systèmes sur mesure et les produits logiciels de grande diffusion.

1.3. Etat des lieux

Le terme 'génie logiciel' a été introduit pour la première fois en 1968 lors d'une conférence internationale consacrée à des discussions sur 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.

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 :

– 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 500000 km au lieu de 5000 km à cause du remplacement d'une virgule par un point,

– 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 synchronisation entre calculateurs assurant la redondance (un délai modifié de 50 ms 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 5 (mauvaise réutilisation du logiciel de Ariane 4).

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 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*, très malléable au sens de facile à modifier,
- ses 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, des progrès ont été réalisés. Mais la complexité des systèmes ne cesse de s'accroître.

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, accès aux données...), informatique théorique (automates, réseaux de Petri, types abstraits...), etc. 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 (ou systèmes d'information) ; le plus souvent des systèmes transactionnels construits autour d'une base de données;
- 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...).

Le GL est difficile à étudier 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...). Dans les sections suivantes, nous allons passer en revue les *principes* généraux, les *techniques* spécialisées, les *méthodes* et *outils* pour le GL.

1.5. Principes fondamentaux

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

- rigueur,
- séparation des problèmes,
- modularité,
- abstraction,
- généricité,
- construction incrémentale,
- anticipation du changement.

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.

Séparation des problèmes. C'est une règle de bon 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),
- 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 *données* avant de considérer l'aspect *traitements*),
- séparation du système en *parties* (sous-systèmes),
- etc.

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 où 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 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 (orientés objets notamment) vise à rendre plus facile une programmation modulaire, appelée aujourd'hui 'programmation par composants'.

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*. L'abstraction permet une meilleure maîtrise de la complexité.

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).

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.

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 fonction *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), ou encore être capable de gérer les multiples versions des modules et configurations des versions des modules, constituant des versions du produit complet.

Il faut noter que les principes ci-dessus 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 connaissances le plus stable, dans un domaine où les outils, les méthodes et les techniques évoluent très vite.

1.6. Outils pour les activités du génie logiciel

Les activités de production du logiciel sont fois complexes et nécessitent de la rigueur dans l'élaboration et exploitation des résultats fournis par chaque étape. C'est la raison pour laquelle on fait appel à des outils pour supporter les activités pendant tout le cycle de vie du logiciel. Ces outils sont appelés *CASE* (Computer-Aided Software Engineering). Les CASE sont classés selon leurs fonctions et phases où ils sont utilisés. On recense les essentiellement les classes de CASE suivantes :

- outils de planification (comme PERT),
- outils d'édition de textes, d'images...,
- outils de traçabilité,
- outils de gestion de configurations,
- outils de prototypage,
- outils d'aide à la conception,
- compilateurs et environnements de programmation,
- outils d'analyse de programmes et code,
- outils de tests,
- outils de déboguage,
- outils de documentation,
- outils de ré-ingénierie,
- ...

2. Modèles de cycle de vie du logiciel

2.1. Notion de processus et d'étape dans le cycle de vie

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, 1976) :

- 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 du produit vis-à-vis des besoins des utilisateurs).

Un *processus* du logiciel est un ensemble d'activités participant à la production du logiciel. Il y a quatre grands types d'activités formant les processus : activités de spécification du logiciel, activités de développement du logiciel, activités de validation et activités d'évolution du logiciel. Ces activités peuvent être organisées de différentes manières selon les besoins et particularités des systèmes et des équipes participantes. Différents modèles ont été proposés. En voici les plus répandus et étudiés dans la littérature.

2.2. Modèle en cascade

Historiquement, la première tentative pour mettre de la rigueur dans le 'développement sauvage' (coder et corriger ensuite, qui est malheureusement encore pratiqué de nos jours !) a consisté à distinguer une phase d'*analyse* avant la phase d'*implantation* (séparation des questions).

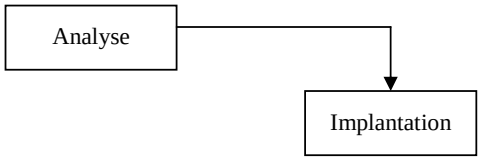

Fig.1 Le modèle primitif à 2 phases

Très vite, pendant les années 1970, 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. 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), 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 et terminologiques***,** largement acceptés et *normalisés* par plusieurs organismes (ISO, AFNOR, IEEE, DOD... Ceci facilite la gestion et le suivi des projets.

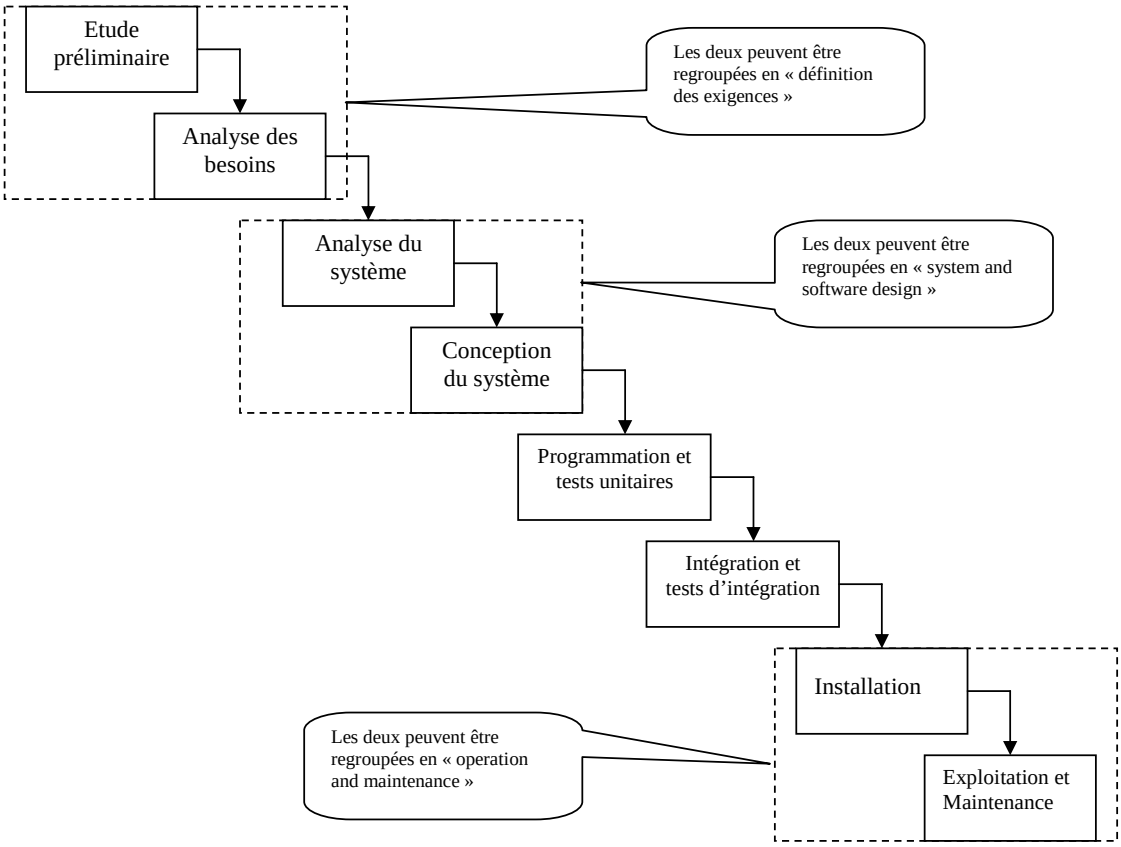

Fig.2 Le modèle en cascade
(le nombre d'étapes peut être variable : en pointillés les étapes pouvant être regroupées)

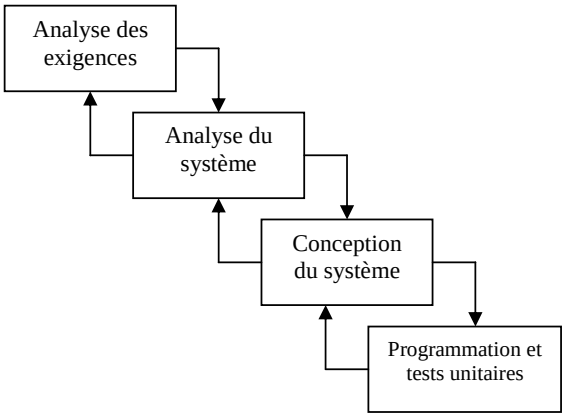

Fig.3 Le modèle en cascade avec itérations (quelques étapes)

Etude préliminaire ou étude de faisabilité ou planification

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

On produit un rapport d'analyse préliminaire qui fournit le schéma directeur.

Analyse des exigences ou analyse préalable

– qualités fonctionnelles attendues en termes des *services offerts*,
– *qualités non fonctionnelles attendues* : efficacité, sûreté, sécurité, utilisation, portabilité, etc.
– qualités attendues du procédé de développement (ex : procédures de contrôle qualité).

On élabore un cahier des charges plus un plan 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 exigences).

Analyse du système

– modélisation du domaine,
– modélisation de l'existant (éventuellement),
– *définition d'un modèle conceptuel* (ou spécification conceptuelle),
– plan de validation.

On élabore un dossier d'analyse plus un plan de validation.

Conception

– 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.*

On élabore un dossier de conception plus un plan de test global et par module.

Programmation et tests unitaires

– traduction dans un langage de programmation,
– tests avec les jeux d'essais par module selon le plan de test.

On élabore un dossier de programmation et codes sources des modules.

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'*).

On élabore des traces de tests et de conclusions de tests.

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.

2.3. Modèle en V

Le *modèle en V* (Fig. 4) 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'.

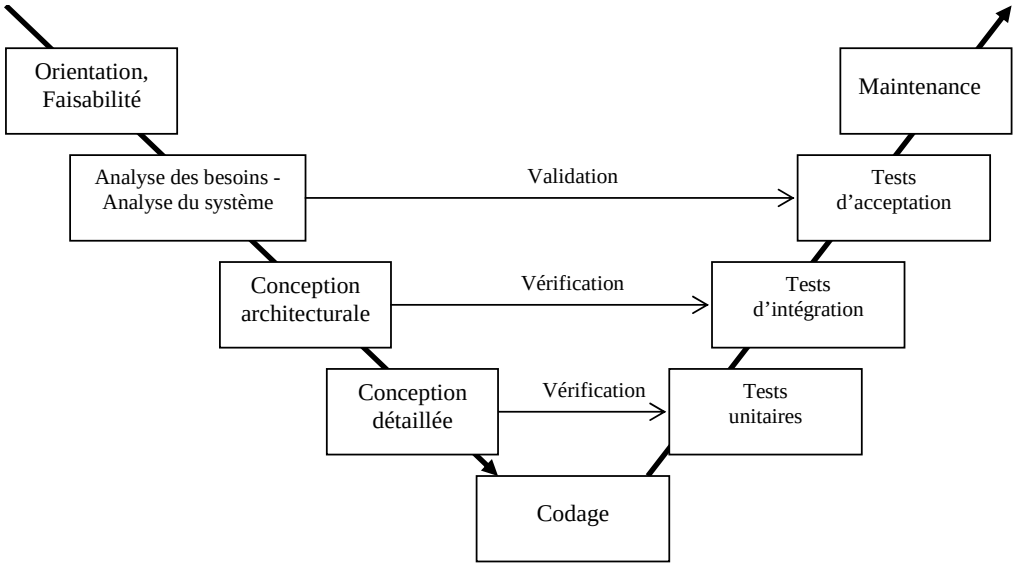

Fig. 4 Modèle en V

2.4. 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 1980 (Fig. 5). 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 exigences 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.

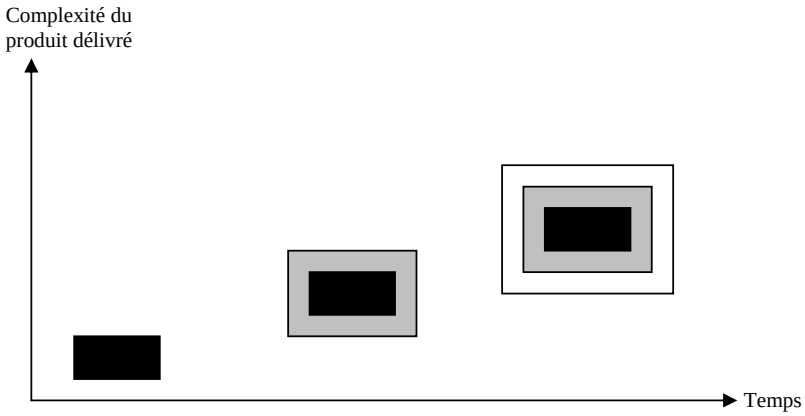

Fig.5. Modèle incrémental

2.5. Modèle en spirale

Enfin le *modèle en spirale*, de Boehm (1988), met l'accent sur l'évaluation des risques (Fig. 6). Les risques à prendre en compte sont très divers : indisponibilité de matériel ou de certains logiciels, changement dans les exigences, dépassement de délais, dépassement de coûts, taille de projet sous estimée, changement de technologie, apparition de concurrents, personnel qui quitte l'équipe... La gestion du risque fait appel à différentes compétences dans le domaine du logiciel, de la gestion de produit, de marketing...

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. Chaque boucle de la spirale représente une phase du processus du logiciel. La boucle la plus interne concerne la faisabilité du système, la boucle suivante concerne la conception, etc. Chaque boucle de la spirale est décomposée en quatre secteurs : détermination des objectifs, Identification des risques et leur réduction, Développement et Validation/Vérification, Planification de la boucle suivante.
Les principaux *risques et leurs remèdes*, tels que définis par Boëhm, sont résumés dans le tableau suivant :

Risque	Remède
Défaillance de personnel	Embauches de haut niveau, formation mutuelle, leaders, adéquation profil/fonction
Calendrier irréaliste	Estimation détaillée, développement incrémental, réutilisation, élagage des exigences
Risque financier	Analyse des coûts/bénéfices, conception tenant compte des coûts
Développement de fonctions inappropriées	Revues d'utilisateurs, manuel d'utilisation précoce...
Développement d'interfaces inappropriées	Maquettage, analyse des tâches
Volatilité des exigences	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...

Tableau 1. Risques et remèdes dans le développement du logiciel

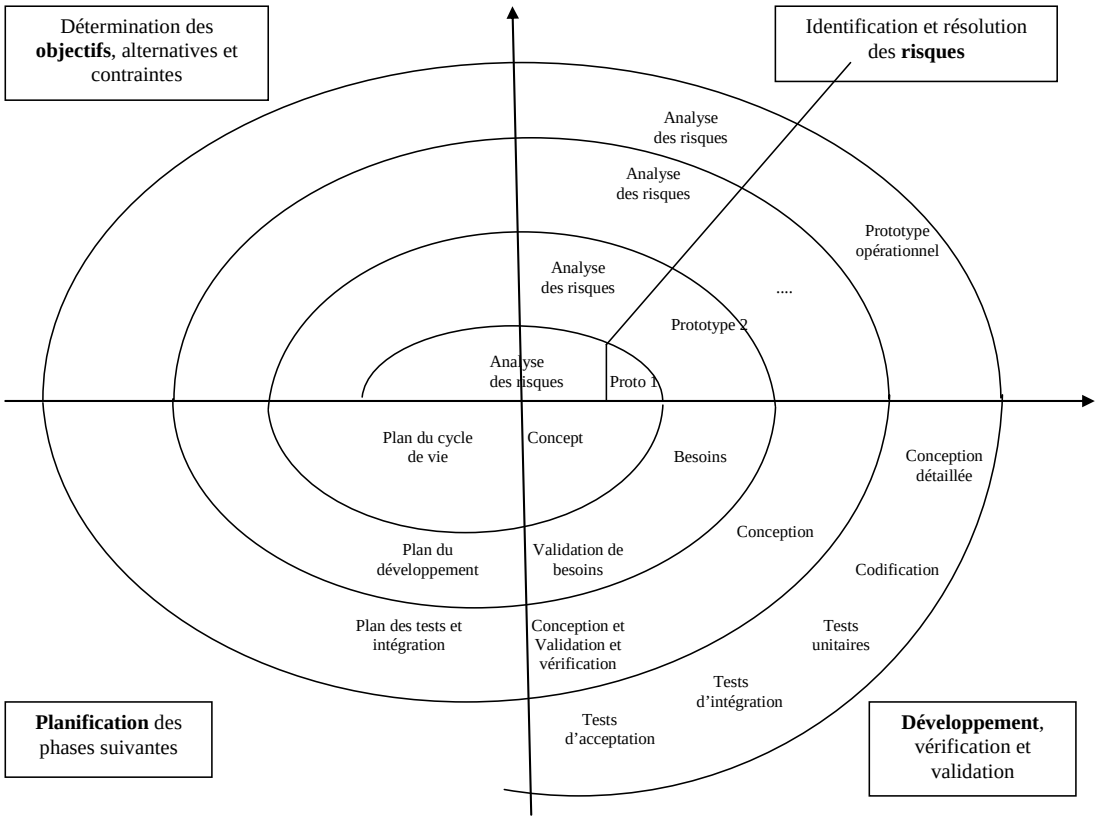

Fig.6 Modèle en spirale

Le modèle en spirale peut être utilisé pour focaliser sur les versions (ou prototypes) successives d'un produit. C'est le cas du processus de développement ROPES (*Rapid Object-oriented Process for Embedded Systems*) qui utilise cette forme de modèle en spirale.

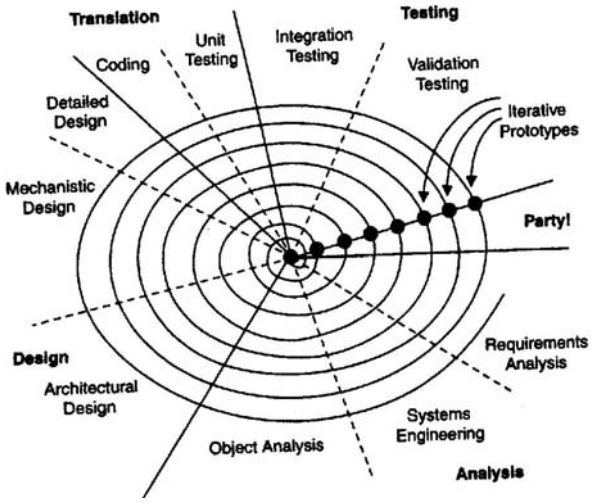

Fig 7 Modèle en spirale de ROPES

Remarque

Il n'y a pas de modèle idéal pour les processus de fabrication du logiciel, 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 inadéquates et souvent remises 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.

2.6. Autres concepts liés aux processus

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,
- migration des fichiers aux bases de données,
- intégration de composants standard (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...) pour faire de la ré-ingénierie.

Normalisation des processus. De nombreuses normes sont apparues dans les années 1990 pour évaluer les processus en fonction de normes de qualité. Les sociétés sont certifiées en fonction de leur respect de ces normes. A titre d'exemple, le standard CMM du SEI (Software Engineering Institute du DoD) définit 5 niveaux de *maturité du processus de développement* (CMM) du logiciel ('initial', 'repeatable', 'defined', 'managed' et 'optimizing'). Le niveau d'une organisation est évalué par des questionnaires, des entretiens, et des examens de documents. 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.

Métriques. Pour l'IEEE, 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 et la qualité (via les mesures d'erreurs, etc.).

3. Analyse des exigences et spécification

3.1. Types d'exigences

Il existe différentes manières de regrouper les exigences (besoins) liées aux systèmes informatiques. Il y a ceux qui distinguent les exigences fonctionnelles et non fonctionnelles. Mais il y a aussi une classification plus précise proposée par le standard IEEE 830 (en 1998) qui distingue les exigences suivantes :

- *Exigences fonctionnelles* : que doit faire le logiciel ? Quelles sont les fonctions (services) du logiciel ?
- *Exigences d'interfaces externes* : comment le logiciel interagit-il avec les individus, avec l'OS, avec d'autres logiciels, avec le hardware ?
- Exigences de performance : vitesse d'exécution, temps de réponse, disponibilité, temps de détection d'erreur, temps de recouvrement d'erreur…
- *Exigences de qualité* : portabilité, correction, maintenabilité, sécurité...
- *Exigences imposées à l'implantation* : langage, politique de gestion de données, de gestion de ressources, l'OS, consommation d'énergie, encombrement mémoire, poids…

3.2. Processus d'analyse des exigences/besoins

Les activités d'analyse des exigences regroupent toutes les activités permettant d'élaborer et de maintenir un document correspondant aux exigences imposées au système. Ces activités sont souvent regroupées en quatre familles d'activités : étude de la *faisabilité* du système, *identification* et *analyse* des exigences, *spécification* des exigences et *validation* de ces exigences.

3.2.1. Etude de la faisabilité

L'objectif de cette phase est de répondre aux questions suivantes :

- Est-ce que le système va contribuer au développement de l'organisation ? Apporte-t-il un plus ? En quoi va-t-il aider à résoudre des problèmes s'il y en a ?
- Est-ce que le système peut être réalisé avec la technologie actuelle, dans les délais requis et avec les coûts indiqués ?
- Es-ce que le système peut être intégré aux autres systèmes déjà existants ?

3.2.2. Identification des exigences et leur analyse

L'identification des exigences est liée à chaque domaine de systèmes (banques, automobile, commande d'installation industrielle, BD, multimédia...). Les questions à se poser pour déterminer ce que doit faire le système sont donc étroitement liées à la nature du système. Il faut être capable à ce niveau de répertorier les exigences, de les classer par catégorie, de leur donner des priorités, de déterminer les liens entre les exigences, les conflits entre exigences... Le travail est itératif et peut se poursuivre en parallèle avec le début des autres étapes du cycle de développement. Beaucoup de praticiens dans le GL conseillent l'identification des scénarios de fonctionnement du système (ce sont les use cases d'UML) pour mieux voir les interactions du système avec son environnement et les interactions entre les sous-systèmes.

Certains scénarios (cas courants d'utilisation) sont souvent faciles à déterminer, d'autres au contraire (cas rares) ne peuvent être identifiés que très tard dans le cycle de vie ou carrément une fois que le système réalisé se trouve en situation d'échec (on ne peut pas tout prévoir !). C'est la raison pour laquelle, l'équipe qui assume les activités liées à l'analyse de exigences doit communiquer avec d'autres équipes et utiliser leur savoir-faire, éviter les erreurs commises dans d'autres projets (il faut signaler qu'à ce niveau les leçons liées aux situations d'échecs, d'erreurs, de mauvaises interprétations,..., bref de l'expérience du terrain, sont malheureusement très rares à trouver). Il faut identifier des exigences même de manière partielle, voire très incomplète, en identifiant des aspects au sujet desquels le client ne connaît pas encore ce qu'il souhaite, les marquer et revenir sur ces aspects plus tard. Ainsi, c'est mieux que de les ignorer totalement. En d'autres termes, il faut prévoir des changements dans la spécification des exigences, voire des zones d'ombre (ceci va évidemment dans le sens contraire de la complétude de spécification).

3.2.3. Spécification des exigences

Une fois les exigences identifiées et classées, il faut utiliser des techniques formelles ou non pour spécifier (décrire) ces exigences en vue d'une utilisation dans les phases suivante du cycle de développement. Comme nous le verrons par la suite, la tendance est de plus en plus vers l'utilisation de techniques formelles (ou au moins semi-formelles) pour spécifier les exigences.

3.2.4. Validation des exigences

Valider les exigences avec le client (l'utilisateur). Il est toujours conseillé (et obligatoire dans certains cas) de valider les exigences avant de passer aux phases suivantes. Si les erreurs et mauvaises interprétations sont découvertes tard dans le cycle de développement, elles coûtent cher (voire très cher) pour être corrigées. Pendant l'étape de validation, plusieurs contrôles doivent être effectués :

- *Contrôle de validité* : est-ce les fonctions prévues sont prévues pour être utilisées par les bons acteurs, aux bons endroits, dans les bonnes conditions ?
- *Contrôle de cohérence* : est-ce que certaines exigences ne sont pas en conflit ou contradictoires avec d'autres ?
- *Contrôle de complétude* : est-ce tout ce dont a besoin a été dit pour concevoir et réaliser le système ?
- *Contrôle de réalisme* : est-ce les objectifs fixés peuvent être atteints avec les technologies ciblées ?
- *Vérifiabilité* : pour éviter les malentendus avec le client, contrôler si tout a été écrit de manière à pouvoir vérifier assertions (affirmations, dires, discours...) des uns et des autres. Cette vérifiabilité est facilitée par l'utilisation de méthodes de spécification formelles.

3.3. Classification des exigences

La première phase cruciale que les développeurs de logiciel doivent aborder avec toute l'attention qui s'impose est celle de l'analyse des exigences. C'est à ce niveau que l'on doit savoir pourquoi on veut développer le logiciel et ce qu'il *va faire*. La difficulté principale (qui conditionne d'ailleurs tout le reste des activités du logiciel) est que le client (utilisateur) ne sait pas toujours exactement ce qu'il veut. L'utilisateur du logiciel a parfois des idées vagues voire incohérentes de ce qu'il veut et c'est au fur et à

mesure que le logiciel est réalisé que l'utilisateur recentre et affine ses exigences. Il s'agit là d'un constat que les ingénieurs du logiciel ne doivent pas oublier.

Même si l'on ne sait toujours pas dire avec précision tout ce que l'on veut sur son futur logiciel, une catégorisation des exigences a été proposée pour faciliter l'identification des exigences :

- *exigences utilisateur* : indiquent ce que le logiciel doit faire et sous quelles conditions de fonctionnement ;
- *exigences système* : indiquent des éléments sur le fonctionnement du logiciel (ergonomie des interfaces, aspects graphiques...) ;
- *exigences logiciel* : apportent des détails sur des contraintes de conception et implantation.

Ces différents besoins sont décrits dans des documents qui sont lus et pris en considération par différents intervenants : responsable clientèle, ingénieurs, architectes de systèmes, développeurs de logiciel... Tous ces intervenants apportent une attention particulière seulement aux parties de ces documents qui les concernent.

Un autre critère très répandu est celui qui consiste à séparer les exigences en deux :

- *Exigences fonctionnelles* : elles indiquent ce que le système doit faire et comment il doit réagir aux différentes situations (événements) qui peuvent se présenter. Elles indiquent aussi les liens entre les entrées et sorties du système.
- *Exigences non fonctionnelles* : elles indiquent des contraintes sur la manière dont le service du système est rendu. Ces besoins incluent notamment : les contraintes de temps, l'espace mémoire, le débit, la fiabilité, l'ergonomie, l'utilisation de standards et de règles de développement, de sécurité, de lisibilité, de maintenabilité, de coûts, d'éthique... Certains parlent de *besoins techniques* au lieu de non-fonctionnels.

Même si on tente de faire cette classification, il est difficile dans la pratique de séparer les exigences de manière claire. Par exemple, on peut considérer la Sécurité comme un besoin fonctionnel ou non-fonctionnel selon l'importance de cet aspect pour l'utilisateur et selon le niveau de détail sur l'élément considéré. On peut dire qu'un logiciel est globalement sûr ou non (il s'agit d'une qualité du logiciel, donc d'un critère non fonctionnel). Mais pour réaliser la sécurité, on a besoin de fonctions d'identification, authentification, chiffrement (il s'agit bien d'aspects fonctionnels).

Il faut noter que lorsqu'on parle de non-fonctionnel, on s'intéresse à des aspects (des propriétés) globaux d'un système et non à des caractéristiques d'éléments individuels du système. En général, lorsqu'une contrainte non fonctionnelle n'est pas satisfaite, le service rendu par le système n'est pas utile, non utilisable, non crédible... Par exemple, si un système ne répond pas aux contraintes de fiabilité dans le secteur de l'avionique, il ne peut pas être certifié (et on imagine mal un produit non certifié pour commander un avion). Un autre exemple, dans le domaine du temps réel, un système qui fournit des résultats hors délais n'est pas utile (voire dangereux) pour commander des procédés industriels.

En général, les contraintes non fonctionnelles quantitatives sont difficiles à déterminer car il faut préciser des valeurs (un temps de réponse de 5 ms, une fiabilité de 98%, par exemple) à partir de données sûres. Les contraintes non-focntionnelles qualitatives sont parfois plus aisées à annoncer (par exemple, un système *rapide*, un débit *moyen*, une interface *conviviale*, un produit *facile à utiliser*...). Pour gérer des contraintes qualitatives, il faut parfois les traduire en contraintes quantitatives (par exemple, un débit élevé peut se traduire par un débit compris entre 1 et 10 Mb/s)

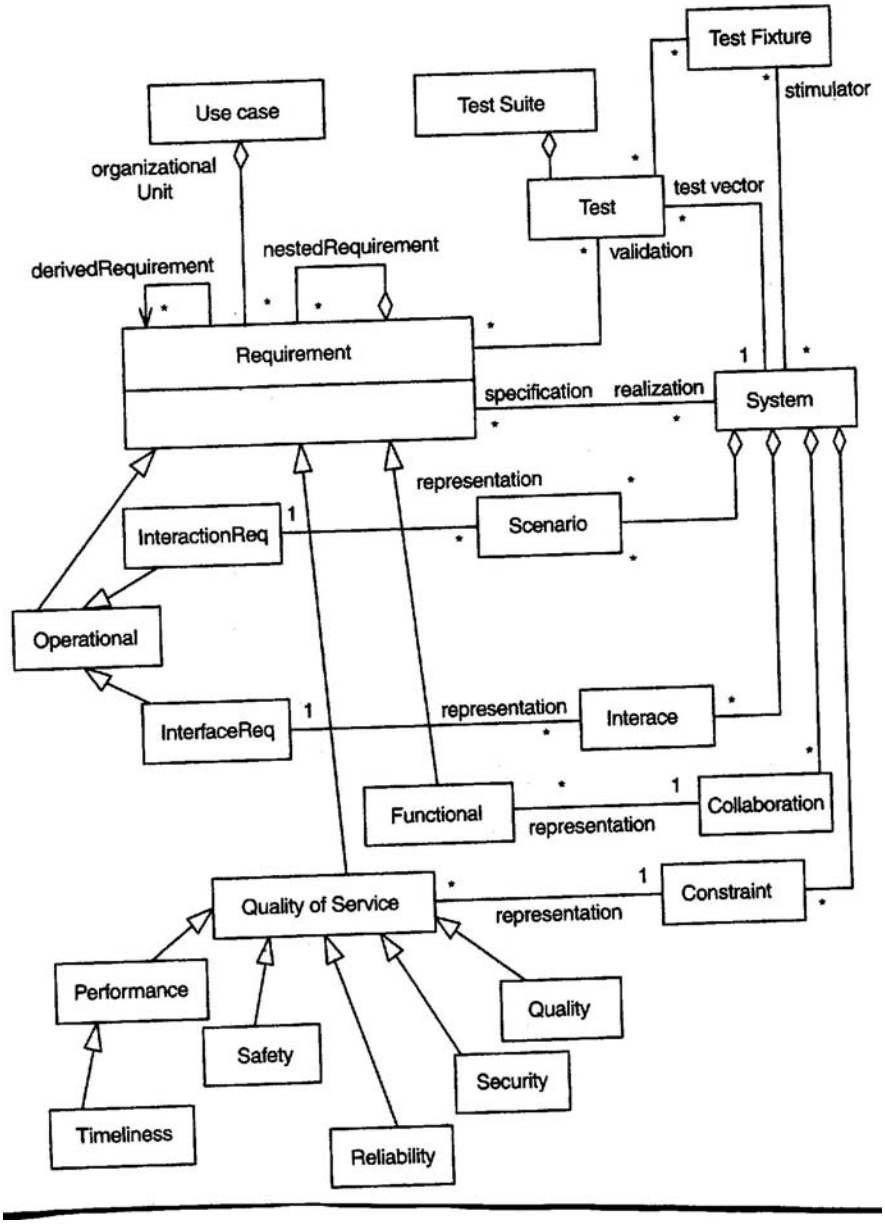

Fig. 8 Taxonomie des exigences

3.4. Spécification des exigences

Comme dans d'autres domaines de l'ingénierie (l'automobile, le bâtiment, la chimie...), tout produit logiciel doit être spécifié le plus clairement possible.

Attention, en informatique le terme 'spécification' peut être utilisé à différents niveaux du développement du logiciel : spécification des exigences, spécification d'implantation, spécification technique d'un module, spécification d'une propriété, spécification de tests, spécification de procédures de tests, spécification de données, spécification de traitements... Bref, il s'agit d'une activité dont le but est de

préciser (clairement) un aspect. Dans ce paragraphe, nous nous intéressons à la spécification des besoins (ou exigences).

Il est souhaitable qu'une spécification soit claire, non ambiguë et compréhensible, complète, cohérente (sans contradictions). Les descriptions en langue naturelle manquent souvent de précision. C'est la raison pour laquelle on préfère utiliser (même si cela est difficile à faire) des méthodes formelles.

Deux critères orthogonaux sont souvent utilisés pour classer les techniques de spécification :

- Le caractère *formel* (utilisation de formalisme) : on distingue des spécifications informelles (en langue naturelle), semi formelles (souvent graphiques, dont la sémantique est plus ou moins précise) et formelles (quand la syntaxe et la sémantiques sont définies formellement par des outils mathématiques).
- Le caractère *opérationnel* ou *déclaratif* : les spécifications opérationnelles décrivent le *comportement désiré*; par opposition, les spécifications déclaratives décrivent seulement les *propriétés désirées*.

Selon le point de vue du spécifieur et/ou de la méthode utilisée, la spécification peut porter sur un deux ou tous les aspects suivants :

- les *fonctions* (la spécification du *Quoi ?*)
- les *données* d'informations et d'état (la spécification de *Qui ?*) qui circulent entre les fonctions.
- le *comportement* ou aspect dynamique (la spécification du *Quand ?*).

3.4.1. Spécifications en langue naturelle

Elles sont très souples, conviennent pour tous les aspects et pour tout le monde, sont très facilement communicables à des non spécialistes. Malheureusement, elles manquent de structuration, de précision et sont difficiles à analyser. Des efforts peuvent être faits pour les structurer (spécifications standardisées) : chapitres, sections, items, justifications, commentaires, règles, etc. Il faut noter que souvent cette forme de spécification est le point de départ pour le cycle de développement du logiciel.

3.4.2. Spécifications dans des langages spécialisés

Des langages semi formels spécialisés pour spécifier des systèmes ont été proposés. Ils comportent des sections et champs prédéfinis, ce qui force à une certaine structuration. Certains utilisent aussi des langages de haut niveau comme des 'pseudo codes' pour décrire les fonctionnalités attendues. Certains domaines d'activité ont leurs propres langages de spécification, définis et acceptés par les équipes de développement du logiciel dans ces domaines. Par exemple, dans le domaine de l'avionique, PDL (*Program Design Language*) est largement utilisé par Airbus notamment.

3.4.3. Spécification avec des diagrammes de flots de données

Il s'agit d'une technique semi-formelle et opérationnelle. Les DFD (*Design Flow Datagrams*) décrivent des collections de données manipulées par des fonctions. Les données peuvent être persistantes (dans des stockages) ou circulantes (flots de données). Souvent, des conventions graphiques sont utilisées pour représenter :

- les fonctions (ou processus) du système ;

- les flots de données (le plus souvent la direction d'un flux de données est matérialisée par une flèche) ;
- les entités externes qui interagissent avec le système mais qui ne sont pas spécifiées dans leur détail ;
- les points de stockage d'information.

Les DFD peuvent apparaître à différents niveaux d'abstraction du système (niveau le plus élevé, niveau des interfaces..., niveau d'implantation réelle), selon les besoins.

La figure suivante montre un DFD qui illustre partiellement le fonctionnement d'une banque. Il y a quatre entités externes (`Directeur`, `Client_A`, `Client_B` et `Client_C`), trois processus (`Ouvrir_Compte`, `Fermer_Compte`, `Deposer_Retirer`), deux stockages (`Info_Clients` et `Comptes`) et 8 flux (`Nom_client`, `Numéro_compte`, `Notification_clôture`, `Montant_Retrait`, `Montant_Dépôt`, `Solde`, `Total`, `Nom_client_supprimé`, `Compte_fermé`).

Les DFD sont simples et faciles à comprendre par des non informaticiens qui participent dans des projets de logiciel. Les DFD ont été intégrées à de nombreuses méthodes de conception comme SA-RT, DE MARCO largement utilisés. Les DFD sont généralement insuffisants à eux seuls et doivent être complétés par d'autres moyens de spécification.

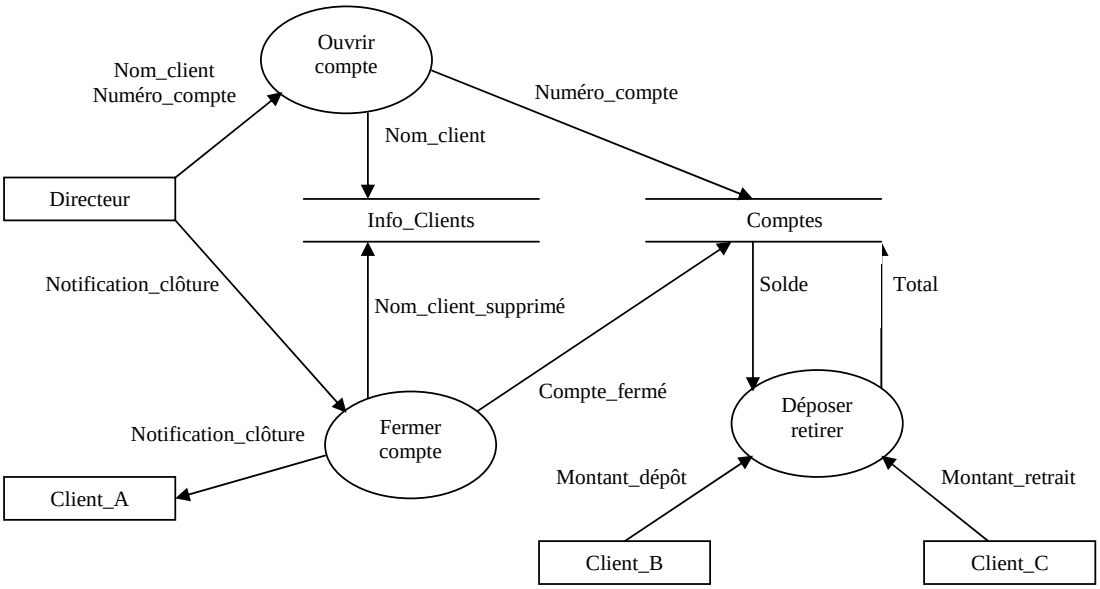

Fig. 9 Exemple de DFD.

3.4.4. Spécification avec les machines à états finis

Les techniques les plus utilisées, surtout pour la spécification des comportements, sont celles fondées sur la notion d'états-transitions : automates à états finis, réseaux de Petri, Grafcet, Statecharts...

3.4.5. Spécification avec le modèle entités-relations

Il s'agit d'une technique semi formelle et déclarative. Ce modèle permet de spécifier la *structure des données* et de *leurs relations.* Les concepts du modèle de base sont :

- les *entités*, qui sont des collections d'items partageant des propriétés communes (occurrences d'entités),
- les *associations* (ou *relations*), qui traduisent l'existence de liens entre entités (occurrences d'associations entre occurrences d'entités),
- les *attributs* (ou *propriétés*), attachés aux entités et aux associations et qui les caractérisent.

Une entité existe indépendamment de ce qui l'entoure. Une association n'existe que si les entités extrémités existent. Chaque entité a un attribut identifiant qui distingue univoquement chaque occurrence d'entité. Certains modèles, dits modèles binaires, n'autorisent que des associations entre deux entités. Les associations peuvent être partielles. Les associations sont souvent caractérisées par leurs multiplicités. Ce modèle est à la base de beaucoup de langages et méthodes, notamment UML et Merise.

3.4.6. Spécification formelle

Dans les domaines traditionnels des techniques de l'ingénieur (génie civil, mécanique, électronique...), les procédés et procédures de travail sont souvent régis par des méthodes basées sur les mathématiques. Ces domaines n'ont pas eu de difficulté à intégrer les approches mathématiques, tout au contraire, elles sont utilisées dans les différents enseignements liés à ces domaines. L'informatique, initialement une 'branche' des mathématiques (appliquée disaient certains), s'est peu à peu écartée des mathématiques de sorte que la production du logiciel conduisait à une discipline de 'bricoleurs' et rien ne peut plus être dit, fait et livré avec rigueur et preuve. « Utilisez le programme et on verra ensuite où sont les erreurs ! » est parfois une 'méthodologie' de pratique du logiciel.

L'apparition d'applications de plus en plus critiques (en termes de coût, de sécurité des personnes et des installations, en termes d'image de marque de l'entreprise), remet en cause la production du logiciel pour 'ramener' l'informatique à ses sources, les mathématiques. Depuis quelques années, la tendance est donc vers l'utilisation de méthodes formelles dans tout le cycle de vie du logiciel. On parle plutôt d'utilisation de méthodes formelles. Ces dernières peuvent être des méthodes de spécification, de transformation et dérivation, d'analyse, de preuve, de test... On parle de méthode formelle, lorsque le langage utilisé possède une syntaxe et une sémantique fondées sur les mathématiques (logique, algèbre, probabilité...).

Il faut souligner que l'emploi généralisé des méthodes formelles n'a pas atteint le stade que certains l'espèrent pour diverses raisons :

- Certains clament que beaucoup de logiciels développés grâce aux nouvelles approches de programmation (orientée objet, composant ou aspect) fonctionnent plutôt bien. Pourquoi alors s'orienter vers une voie difficile, coûteuse et qui n'a pas encore fait ses preuves.
- Les méthodes formelles sont difficilement utilisables (ou pas du tout utilisables) pour certains aspects du logiciel ; de plus elles deviennent très complexes lorsque la taille des problèmes devient importante (on parle de la difficulté de passage à l'échelle – '*scalability*')
- Peu d'outils efficaces et utilisables sur de vrais systèmes existent actuellement. La question de la *correction* des outils supportant les méthodes formelles se pose aussi.
- Beaucoup d'étudiants en informatique sont peu formés aux mathématiques. Il faut revoir l'enseignement de l'informatique pour mieux montrer les liens entre les mathématiques et les mécanismes et concepts de programmation et de système.

Une note optimiste : il existe un certain nombre de domaines importants (contrôle de trafic aérien, chemins de fer, pilotage d'avion, systèmes médicaux...) qui ont utilisé (et utilisent encore) avec succès de les méthodes formelles.
Les spécifications formelles sont surtout utilisées pour définir des interfaces et le comportement. Par exemple, on utilise souvent le modèle suivant (celui des types abstraits) pour décrire des interfaces :

```
<nom de spécification>
   sort <nom>
       imports <liste des spécifications importées>
       exports <liste des spécifications exportées>
       <description informelle du type et de ses opérations>
       operations <listes de opérations sur le types et leurs signatures>
       axioms <liste des axiomes qui définissent la manière dont les opérations
              agissent sur les données>
```

Les méthodes pour spécifier le comportement sont le plus souvent fondées sur la notion d'*état* (état qui change dans le temps et qui représente la dynamique) du système. On définit alors les états pertinents du système et les conditions de changement d'états (les transitions).

Il existe beaucoup de méthodes formelles, avec des fondements mathématiques divers (ensembles, logique classique, logiques temporelles, logiques temps réel...). Elles ont été souvent utilisées à l'origine pour spécifier des *types de données abstraits*, puis ont été étendues pour spécifier des systèmes complets. C'est le meilleur moyen de définir rigoureusement les propriétés d'un système. Comme exemples de méthodes formelles utilisées dans le milieu industriel, on peut citer les méthodes B, Z et VDM.

3.4.7. Complémentarité des techniques de spécification

Il n'existe pas de techniques de spécification permettant d'élaborer rapidement et facilement des spécifications formelles, claires, faciles, complètes... C'est la raison pour laquelle on peut utiliser conjointement plusieurs techniques de spécification, chacune doit être appliquée aux aspects de système pour lesquels elle est la plus efficace. Les techniques de spécification se complètent, en décrivant des vues complémentaires d'un système. Par exemple, un système peut être spécifié à travers un diagramme de flot de données (sources d'informations, types d'informations stockées et échangées, décomposition en fonctions), un schéma Entités Associations (structuration des informations) et des machines à états finis (comportement de certains composants). Les méthodes tentent de proposer des assemblages efficaces de telles techniques avec des guides pour les construire et les valider. C'est ce qui est fait notamment en UML ou plusieurs formalismes peuvent être utilisés conjointement.

3.5. Notion de modèle de système

La notion de modèle est utilisée, à différents stades du développement du logiciel, pour avoir une vue (faire une abstraction) d'un système afin d'en étudier :

- l'aspect externe du système et son interaction avec son environnement
- le comportement du système (aspect dynamique)
- la structure (statique) du système, notamment ses composants et ses données notamment.

On peut avoir des modèles de données, de traitements, d'architectures, de composants et leur composition, des événements... La modélisation consiste à se focaliser sur certains aspects (détails) et ignorer les autres afin de se faire *une vue* du système.

Les modèles de comportement sont souvent spécifiés à l'aide de diagrammes de flux (DFD) ou les machines à états. Les modèles de données sont souvent spécifiés à l'aide de techniques entités-associations, parfois avec les types abstraits.

4. Conception

Pendant la phase de conception, on propose une **solution** au problème posé et spécifié lors de la phase précédente (celle d'analyse des exigences). Ici on répond à la question « *Comment faire ?* ». Par solution, on entend : une *architecture* du système (architecture logicielle et architecture physique) et une description détaillée des *modules*, des *interfaces utilisateurs*, des *données*...

La conception donne lieu à un dossier avec souvent une partie destinée au client (présentation de la solution) et une partie pour les réalisateurs (conception technique). La frontière entre ce qui est destiné aux développeurs (programmeurs) et au client n'est pas facile à identifier car elle dépend du type de client considéré.

La phase de conception repose essentiellement sur le choix d'une architecture et des modules qui la composent. Ce choix peut être guidé ou non par des exigences issues de la phase d'analyse des exigences. Par exemple, le client peut exiger un système décrit en UML et fonctionnant sous Linux et la communication devra faire appel à CORBA supporté par un réseau Internet.

4.1. Notion de module et décomposition de système

Un module est un *composant* d'une application, contenant des définitions de données et/ou de types de données et/ou de fonctions et constituant un tout cohérent. On peut définir un module comme un *fournisseur de ressources* ou *de services* pour d'autres modules. Un module interagit avec les autres modules ; les relations entre modules doivent être identifiées et spécifiées avec précision. Les approches et méthodes de conception insistent beaucoup sur la notion de module. Cependant, elles ne disent rien sur la quantité, la qualité ou le type d'informations nécessaires pour définir un module. Le choix des modules, donc de l'architecture du système, est souvent subjectif ; il dépend des habitudes et de l'expérience du concepteur. Si on propose un système nouveau à plusieurs concepteurs qui vont travailler de manière séparée, il fort probable d'avoir autant de décompositions modulaires qu'il y a de concepteurs.

Il y a principalement deux approches pour identifier les modules :

- *Approche fonctionnelle* (ou orientée *dataflow*) : un module est un *sous-système* du système global. Chaque sous-système réalise une fonction bien identifiée dans le système. On peut par exemple identifier dans un système industriel des fonctions de gestion de stock, de supervision, de contrôle qualité, de commande des appareils de production...Toutes ces fonctions constituent des sous-systèmes industriels. SADT (Structured Analysis Design Technique) et SASD (Structured Analysis, Structured Design) sont deux des méthodes fonctionnelles les plus utilisées.
- *Approche orientée objet* : les modules principaux correspondent aux objets concrets ou abstraits du domaine de l'application (‘exemple, un robot, un capteur, une caméra...). Les objets regroupent données et traitements. Ce sont des entités autonomes qui collaborent pour réaliser le système global. Des relations (spécialisation, héritage, généralisation...) sont définies entre les classes d'objets permettant plus de souplesse dans la réutilisation des objets. UML est l'une des techniques fondées sur la notion d'objet qui tend à se généraliser de plus en plus dans le cycle de conception de logiciel.

Que l'approche soit fonctionnelle ou orientée objet, on cherche à diviser le système en entités (modules) plus faciles à comprendre et réaliser séparément.

Des approches plus récentes et plus complexes permettent de mieux maîtriser le cycle de développement du logiciel : approches orientée *Patterns* et approches orientées *Aspects*. Un *design pattern* est une solution de conception commune à des problèmes récurrent dans un contexte donné. Un *aspect* de conception peut être la synchronisation, la communication, la gestion des exceptions, le partage de ressources... Les méthodes de design pattern et orientées aspects constituent des méthodes au-dessus du concept d'objet pour identifier les bons schémas et squelettes de logiciel afin de faciliter la réutilisation des composants logiciels et d'exploiter le plus possible le savoir-faire faire acquis grâce à des développements antérieurs.

Un des aspects importants des modules est celui des traitements qu'ils effectuent. Ces traitements sont basés sur des algorithmes (basés sur la théorie des graphes, l'analyse numérique, la théorie des probabilités, les heuristiques...). Le concepteur des modules doit avoir une bonne connaissance dans le domaine des algorithmes pour proposer des modules efficaces. La décomposition en objets ou modules ne permet pas par elle-même de choisir les bons algorithmes.

4.2. Types de conception

4.2.1. Conception d'architecture et ADL

Ce type de conception focalise sur la structuration du système en sous-systèmes. Chaque sous-système doit avoir un rôle bien identifié. Les relations de coopération (échanges de données et de signaux) entre les sous-systèmes sont clairement définies. Une attention particulière doit être portée aux sous-systèmes et données critiques et/ou qui jouent un rôle important dans le contrôle du fonctionnement du système. Différentes solutions peuvent être envisagées selon les situations : solution centralisée, solution répartie, mono-site, solution réseau, solution avec mémoire partagée, solution avec base de données, solution client-serveur, solution *peer-to-peer*, solution dirigée par les événements, solution dirigée par le temps... La notion de composant est souvent utilisée dans les approches de conception d'architecture.

Des langages dits, **xADL** (Architecture Description Languages), sont en cours de définition et d'expérimentation. On peut citer : AADL, ACME, Aesop, C2, MetaH, ModeChart, RAPIDE, SADL, Unicon, Wright.

La communauté ADL considère qu'une architecture (logicielle ou matérielle) est un ensemble de composants interconnectés les uns aux autres pour remplir une certaine fonction globale. Généralement, trois types de concepts sont utilisés dans une architecture :

- Les objets (ou composants) qui définissent des rôles ou des fonctions à réaliser ;
- Les interfaces qui définissent les caractéristiques (visibilité, paramétrage…) des modules communicants.
- Les connexions représentent les interfaces sous la forme de graphe.

Pour être qualifié de ADL, un langage devrait répondre au moins aux exigences suivantes :

- Être adapté à la description de la communication entre composants d'une architecture ;
- Supporter la notion de tâches, de création de tâches et de raffinement de tâche ;
- Fournir une base pour faciliter le passage à l'implantation, voire permettre la dérivation de l'implantation à partir de l'architecture ;

- Offrir des capacités permettant de prendre en compte les styles les plus communément utilisés pour décrire des architectures ;
- Permettre une représentation hiérarchique (abstraction) et faciliter le prototypage.

Les ADLs ont en commun les aspects suivants :

- Utilisation de langages avec une sémantique formellement définie ;
- Utilisation de représentations graphique et textuelle équivalentes ;
- Capacités de modélisation de systèmes distribués ;
- Capacités de raisonnement par abstraction.

Les ADLs diffèrent selon les points suivants :

- Manière de prendre en compte les aspects temps réel (échéances, priorités des tâches…) ;
- Possibilité de prendre en compte et combiner plusieurs styles de description ;
- Possibilités (outils) d'analyse des architectures que l'on vient de décrire ;
- Gestion des instances d'architectures (réutilisation dans différents projets…).

4.2.2. Conception orientée objet

Elle met la notion d'*objet* au centre de la conception et consiste identifier des objets et classes d'objets, de les hiérarchiser, spécifier les interfaces des objets, spécifier les relations entre objets (spécialisation, généralisation, héritage, concurrence...).

4.2.3. Conception de systèmes critiques

On parle en général de systèmes critiques, lorsque l'on traite des systèmes informatiques en interaction directe avec leur environnement. On dit aussi *systèmes temps réel et embarqués* (par exemple, systèmes d'aide à la navigation, de pilotage d'aéronef, de commande de procédé chimique, de surveillance de malades...). Les approches de conception de ces systèmes focalisent sur les aspects liés à la *dependability* qui regroupe la disponibilité[1] (*availability*), la fiabilité (*reliability*[2]), la sûreté (*safety*[3]) et la sécurité (*security*[4]). Elles mettent l'accent sur les stimuli (données en provenance des capteurs) qui arrivent au système et sur les réponses générées par le système (commandes envoyées à l'environnement). Des contraintes de temps réel sont souvent prises en compte par de tels systèmes afin de tenir compte de la dynamique de l'environnement du système.

[1] Disponibilité : aptitude du système à rendre le service quand celui-ci est demandé.
[2] Fiabilité : aptitude du système à rendre le service tel qu'il a été spécifié (et pas un autre).
[3] Sûreté : aptitude du système à rendre le service sans engendrer des situations catastrophiques.
[4] Sécurité : aptitude du système à se protéger contre les intrusions et malveillances.

5. Validation

D'après la terminologie de l'IEEE (norme 729), la *faute* est à l'origine de *l'erreur* qui se manifeste par des *anomalies* dans le logiciel qui peuvent causer des *pannes* :

Faute → erreur → anomalie → panne.

La validation consiste à éviter ou à retrouver et éliminer les fautes avant qu'elles ne conduisent à des pannes. Validation est le nom donné aux activités de contrôle et analyse pour s'assurer que le logiciel et conforme à sa spécification. La validation doit se faire à toutes les étapes (analyse des exigences, conception, implantation).

La rigueur avec laquelle le processus de validation est effectué est directement liée aux spécificités de l'environnement dans lequel le logiciel sera utilisé (centrale nucléaire, système de téléconférence, éditeur de textes....). Dans certains cas, le client peut accepter un produit développé à moindre coût et le corriger (faire le déboguage) au fur et à mesure de son utilisation, dans d'autres, le client accepte de payer le prix fort et ne veut pas entendre parler de défaillances (il veut un produit 'Zéro défaut'). Par conséquent, les procédures de Validation et leurs coûts varient considérablement d'un domaine d'application à un autre.

Selon ce que l'on veut valider et le degré de validation, on peut faire appel à différentes approches : vérification, test et simulation. Il est clair que ces techniques ne sont pas exclusives. En effet, on peut pour une même phase, appliquer deux ou trois techniques soit pour réduire le coût de la validation en validant certains aspects par la preuve et d'autres par les tests ou la simulation, soit parce que le système à valider est trop critique pour se contenter d'un seul type de validation, soit parce que les aspects à valider ne peuvent être tous valider par une seule technique.

Validation et Vérification sont souvent confondues et pourtant elles sont différentes :
- Validation : on veut répondre à la question « développons-nous le bon produit ? »
- Vérification : on veut répondre à la question « développons-nous correctement le produit ? »

La vérification concerne la preuve de propriétés alors que la validation est un processus plus général qui va jusqu'à la satisfaction du client vis-à-vis du produit.

5.1. Vérification informelle : Inspection de fautes dans les programmes

La liste des questions suivantes permet d'aider à éviter (ou retrouver de manière manuelle) des fautes dans les programmes. La liste des questions que l'on se pose pour inspecter le code et déterminer les fautes est évidemment beaucoup plus longue et dépend de chaque contexte, langage et application.

Fautes liées aux données

- Est-ce que toutes les variables sont initialisées avant l'utilisation de leurs valeurs ?
- Est-ce que toutes les constantes ont été nommées ?
- Est-ce que les indices des tableaux sont corrects ?
- Est-ce les chaînes de caractères sont correctement dimensionnées ?
- Y a-t-il des situations suspectes de débordement ('overflow') ?

Fautes de contrôle

- Est-ce la condition de chaque test est correcte ?
- Est-ce que l'on est sûr que chaque boucle se termine ?
- Est-ce que les instructions composées sont correctement construites ?
- Est-ce que chaque *Case* contient un *break* quand cela est nécessaire ?

Fautes liées aux entrées/sorties

- Est-ce que toutes les variables d'entrée sont valides ?
- Est-ce que toutes les variables de sortie ont une valeur valide avant d'être envoyées vers l'interface externe ?
- Est-ce des entrées particulières peuvent conduire à des anomalies ? Lesquelles ?

Fautes liées aux interfaces :

- Est-ce que tous les appels de fonctions/méthodes ont des paramètres ?
- Est-ce les paramètres formels coïncident avec les paramètres d'appel ?
- Est-ce que les paramètres sont dans le bon ordre ?
- Si certains composants accèdent une mémoire partagée, est-ce que ces composants ont tous le même modèle de la structure partagée ?

Fautes liées à la gestion de la mémoire

- Si une structure avec des liens a été modifiée, est-ce que tous les liens ont été correctement mis à jour ?
- Est-ce que l'espace mémoire alloué dynamiquement, a été correctement alloué ?
- Est-ce que l'espace non utilisé est libéré correctement ?

Fautes liées aux exceptions

- Est-ce les situations d'exceptions ont été identifiées et leurs traitement clairement définis ?

5.2. Vérification formelle

Il s'agit de prouver formellement la correction d'une spécification ou d'un programme. Deux approches sont essentiellement utilisées : *model checking* et la *preuve de théorème*.

5.2.1. Model-checking

Une des technique les plus utilisée pour vérifier des logiciels (en particulier le logiciel temps réel et critique) est celle du model-checking. Le problème du *model-checking* est le suivant (Fig. 10) : «étant donnés une formule *F* (en logique LTL, CTL, etc) et un système de transitions *S*, est-ce que *S* satisfait *F* ?» (on dit aussi «est-ce que *S* est un modèle pour *F* » d'où le nom de model-checking).

Les outils de model-checking acceptent des spécifications de exigences ou de conception (des modèles) et des propriétés que les modèles spécifiés sont censés vérifier. A la fin, les outils rendent un verdict : *oui* (la propriété est satisfaite) ou non. En général, les outils fonctionnent à l'aide de contre exemples.

Les procédures de model-checking sont différentes selon la logique utilisée pour représenter la propriété à vérifier et les modèles de systèmes (en général décrits à l'aide de système états-transitions).

Des outils comme SMV, SPIN, KRONOS, UPPAAL, HYTECH ou Design/CPN sont disponibles (en tant que produits freeware ou payants) pour faire du model checking.

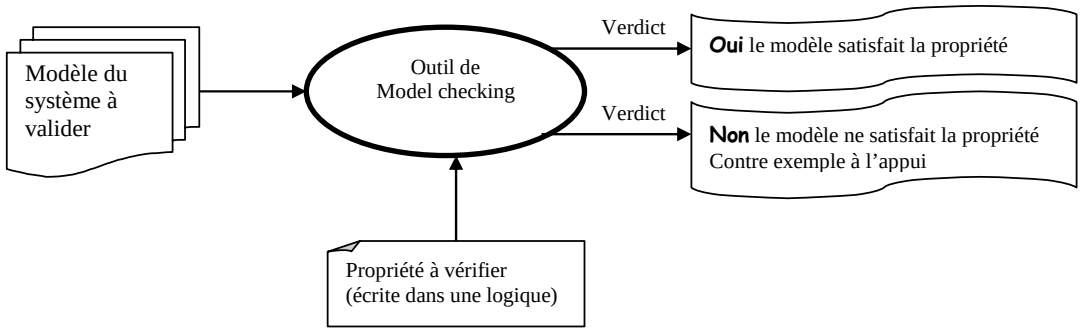

Fig.10. Principe simplifié du model-checking.

5.2.2. Preuve de théorème

Dans les approches de preuve de théorèmes, on retrouve en particulier la méthode de Hoare qui peut se résumer ainsi : tout programme est caractérisé par sa pré-condition (condition éventuelle à respecter par les données du programme) et sa post-condition (condition vraie à la fin du programme qui définit donc son objectif). Pour réaliser la preuve d'un programme, la méthode de Hoare définit des assertions logiques intermédiaires; on part de la post-condition du programme et à chaque instruction on regarde quelle est l'assertion qui doit être vraie avant l'appel de l'instruction pour que l'assertion après l'appel de l'instruction soit vraie. Si on peut remonter de la sorte jusqu'à la pré-condition du programme on prouve ainsi sa correction (si la pré-condition est vraie alors la post-condition est vraie). Il s'agit donc de méthodes basées sur la preuve de théorèmes au sens mathématique.

5.2.3. Théorie et pratique de la preuve

Le domaine de la vérification a plusieurs propriétés intéressantes. D'un côté, il repose sur une base théorique claire et simple (théorie des automates, logique). Bien entendu, cette théorie peut devenir compliquée dès que l'on veut établir des résultats mathématiques profonds. D'un autre côté, il existe un très grand nombre de problèmes dont la validation du matériel et du logiciel qui peuvent, en principe, être formulés et résolus dans le cadre de cette théorie. Il existe bien sûr un fossé entre les modèles abstraits et les vrais systèmes : un programmeur peut ne pas réaliser que les lignes de code dans son programme définissent en fait un automate. De manière similaire, un concepteur de circuits peut ne pas voir ces schémas de câblage comme des automates. Néanmoins, la distance entre l'ingénierie et les objets

mathématiques utilisés n'est pas trop grande. Du fait que les logiciels et les matériels sont, au plus, des machines de Turing (et très souvent des automates finis), il n'est pas important de savoir dans quel langage ils ont été programmés, il existe des résultats généraux qui permettent de les considérer tous : s'il existe un algorithme qui permet d'explorer l'espace des états de tout automate et donc de prédire l'ensemble de tous ses comportements possibles, alors cet algorithme est applicable (en principe) à tous les systèmes qu'ils soient écrits en Java, VHDL, C... quelque soit le type de la machine cible. Les problèmes d'implantation efficace et adaptée à certaines classes d'applications, ou les problèmes de traduction entre langages, sont bien entendu importants, mais secondaires pour la vérification.

5.3. Test de logiciel

Le test est l'une des approches les plus utilisées pour valider des logiciels. Le test peut être un travail complexe et coûteux. Le test constitue une activité importante qui a ses propres techniques. Il y a des approches pour tester : des programmes complets, des boucles, des chemins d'un programme, des interfaces, des protocoles de communication, des modules, des composants, des flux de données…

Les tests se font à partir de jeux de tests (ou jeux d'essais) qui ne peuvent pas en général être exhaustifs. Le jeu de test est sélectionné en fonction du coût et de la couverture de test souhaitée (la couverture de test désigne le rapport entre les cas que l'on veut tester et tous les cas possibles). Le programme est exécuté avec le jeu de test. Les résultats obtenus sont comparés aux résultats attendus d'après les spécifications du problème. Attention : les tests ont pour but de mettre en évidence les erreurs. Les tests peuvent prouver la présence d'erreurs mais ne peuvent pas prouver leur absence. A méditer : « *There's always one more bug, even after that one is removed* » [Weinberg]

Il existe différentes approches (non nécessairement exclusives) pour construire des jeux de tests pour des programmes séparés (on parle de *tests unitaires*) :

- *Approche aléatoire :* le jeu de test est sélectionné *au hasard* sur le domaine de définition des entrées du programme. Le domaine de définition des entrées du programme est déterminé à l'aide des *interfaces* de la spécification ou du programme. Cette méthode ne garantit pas une bonne couverture de l'ensemble des entrées du programme. En particulier, elle peut ne pas prendre en compte certains cas limites ou exceptionnels.

- *Approche fonctionnelle ou boîte noire :* on considère seulement la spécification de ce que doit faire le programme, sans considérer sa structure interne. On peut vérifier chaque fonctionnalité décrite dans la spécification. On s'appuie principalement sur les données et les résultats. Le danger est l'explosion combinatoire qu'entraîne un grand nombre d'entrées du programme. Par contre, on peut écrire ces tests très tôt, dès qu'on connaît la spécification. Une technique souvent utilisée est *l'analyse des valeurs frontières et le regroupement en classes d'équivalence*. Elle se base sur l'observation que les erreurs arrivent souvent aux limites des domaines de définition des variables du programme. Au lieu de tester toutes les valeurs, on partitionne les valeurs en classes d'équivalence et on teste *aux limites des classes*.

- *Approche structurelle ou boîte blanche :* dans l'approche par *boîte blanche* on tient compte de la structure interne du module. On peut s'appuyer sur différents critères pour conduire les tests comme : le critère de *couverture des instructions* (le jeu d'essai doit assurer que toute instruction élémentaire est exécutée au moins une fois), le critère de *couverture des arcs du graphe de contrôle* (le jeu de test doit s'assurer que l'on passe au moins une fois par chaque arc du graphe), le critère de couverture des *chemins du graphe de contrôle* (le jeu de test doit s'assurer que l'on passe au moins une fois par les chemins importants du graphe) et le critère des *conditions* (le jeu de test doit faire passer de vrai à faux et inversement toutes les conditions).

Il existe aussi d'autres formes de tests, notamment les *tests d'intégration* (après avoir testé unitairement les modules il faut tester leur intégration progressive jusqu'à obtenir le système complet), les *tests de réception* (test, généralement effectué par l'acquéreur dans ses locaux après installation d'un système, avec la participation du fournisseur, pour vérifier que les dispositions contractuelles sont bien respectées) et les *tests de non régression* (à la suite de la modification d'un logiciel ou d'un de ses constituants, un test de non régression a pour but de montrer que les autres parties du logiciel n'ont pas été affectées par cette modification).

5.4. Simulation

Une des techniques les plus utilisées, en particulier dans le monde industriel, pour valider un système est la simulation. Elle consiste à définir un modèle pour le système ou pour son environnement et à exécuter les programmes correspondants sur machine et analyser ensuite les résultats. Il existe de nombreux outils de simulation. Ils se distinguent par leurs capacités à couvrir de larges spectres de fonctionnements du système simulé, leurs temps d'exécution, leurs interfaces... leurs coûts.

Le mot "Simulation" recouvre des usages assez différents. De façon générale, il y a simulation chaque fois qu'un modèle (informatique, pour nous) reproduit la configuration du système à étudier et surtout son évolution dynamique. Le simulateur est ainsi une "maquette" logicielle, complétée par un modèle de l'environnement, que l'on fait évoluer pour y mesurer les grandeurs critiques. *Simuler, c'est expérimenter sur un modèle.*

De nombreux systèmes admettent une description selon des variables évoluant de manière discontinue. On parle alors de *systèmes discrets*. La simulation opère sur un modèle "comportemental", c'est à dire une description du système. C'est le type de situation qui est la plus utilisée en informatique et qui est appelée *simulation par événements discrets.*

Le champ d'applications de la Simulation est évidemment très large (toutes les formes d'applications et de programmes informatiques peuvent être simulées). En particulier, la simulation est utilisée avec succès à la validation de bon fonctionnement de mécanismes "complexes" (typiquement, les protocoles, et plus généralement toutes organisations dont la complexité engendre une combinatoire délicate à maîtriser).

Un autre usage du mot, ce qu'on appelle la *simulation d'environnement*, qui recouvre les applications du genre "Simulateur de vol", et systèmes d'entraînement analogues. La mise au point de systèmes informatiques temps réel fait appel à cette technique. Il s'agit là de reproduire des conditions rares et délicates, et de procéder au réglage des paramètres de contrôle du logiciel impliqué (régulation de surcharges, pannes d'organes, etc).

La simulation d'un système nécessite de décrire les variables caractéristiques de ce phénomène, les états du système et les événements qui font évoluer l'état du système. On appelle système discret un système dans lequel la description d'état ne comporte que des variables discrètes (i.e. à variation non continue). Les instants d'évolution de l'état surviennent donc de façon "discrète" (c'est à dire qu'il n'y en a qu'un nombre fini dans un intervalle de temps fini). On les appelle des événements (on parle aussi d'événements discrets). La description d'états dépend du problème posé, c'est-à-dire du système à étudier mais aussi des aspects qu'on en privilégie.

La simulation par événements va consister, une fois le modèle décrit, à le faire évoluer, en mettant à jour la description d'état à chaque événement. Cela signifie que le Simulateur va sauter d'un événement au suivant. L'exécution d'un événement est instantanée (dans le temps du modèle). Chaque événement

engendre d'autres événements. Puisque "il ne se passe rien" entre les événements, c'est-à-dire puisque les variables significatives n'évoluent qu'à ces instants, il n'est pas utile de considérer les périodes inter-événements! Lorsque le simulateur passe d'un événement au suivant, la date saute de la valeur courante à celle du nouvel événement.

La simulation est souvent partielle, elle ne constitue pas une preuve que le produit est correct, mais elle permet seulement d'affirmer que si les cas simulés (et seulement ces cas) vont se présenter dans la réalité, alors le système fonctionnera correctement.

En fin, il faut noter que la simulation n'est pas seulement un outil de résolution numérique, mais aussi une *discipline de modélisation* : aide à la description et à la compréhension des mécanismes étudiés.

Bibliographie

B.W. Bohem, *Software engineering economics*, Prentice Hall, 1981.

B.W. Bohem *A spiral model of software development and enhancement*, IEEE Computer, 21(5), 61-72, 1988.

Ian Sommerville, *Software engineering*, Addison Wesley, 2000.