Introduction au génie logiciel

Pascal ANDRE
INPHB - Département Mathématiques & Informatique
BP 1083 Yamoussoukro - COTE D'IVOIRE
Tel: 05.92.34.85 - Email: andrep@inphb.edu.ci

30 janvier 2001

La construction du logiciel nécessite des moyens de réflexion, d'expression et de calcul pour pouvoir aborder des problèmes de plus en plus complexes. En plus des techniques de description de programmes, il faut des techniques et des méthodes de construction toujours plus fiables, plus rapides et plus rentables.

1 Introduction

La construction d'un programme simple sur un micro-ordinateur peut se faire de manière artisanale. Le programmeur écrit en quelques lignes ce que doit faire le programme (le besoin) et se lance immédiatement dans le codage, en testant certaines fonctions pour vérifier et valider le texte de départ. On peut qualifier ce travail de **développement du programme** et son organisation de **démarche de développement** : étude du problème, conception d'un algorithme, codage et test. L'élément clé de ce processus artisanal est une bonne connaissance du langage de programmation. Les choses se compliquent si plusieurs programmes sont à construire.

On atteint très rapidement les limites d'une telle approche dans la construction de programmes plus conséquents pour lesquels un programmeur ne suffit plus. On ne parle plus de programmes mais de logiciels ou d'applications. Un paramètre organisation d'un travail de groupe vient s'ajouter à la créativité et à l'expertise du langage de programmation. Ce paramètre couvre plusieurs aspects :

- Etablir des tâches et répartir ces tâches : organiser le développement. Chaque intervenant a des compétences qu'il faut exploiter dans un rôle qu'on lui donne. Pour des applications complexes, on aura des spécialistes du système à automatiser, des spécialistes de la gestion de projets, des spécialistes de du langage de programmation (l'environnement de développement). Cet aspect des choses inclue aussi la formation des intervenants ou les changements d'équipes.
- Gérer la communication entre les participants au projets. Pour cela il faut mettre au point des moyens de communication (un ou plusieurs langages, documents, références). On parle de spécification pour exprimer à un niveau donné ce que doit faire le logiciel. Les procédures de communication doivent tenir compte des rôles et compétences des intervenants.

Les applications sont conçues et développées pour une organisation, qu'on appelle souvent le client. De nouveaux paramètres sont à prendre en compte : les coûts de développement, la stratégie de l'entreprise.

- On analyse les coûts de développement selon plusieurs critères : coût financier, coût temporel, moyens humains, technologies requises, etc.
- De nouvelles étapes sont ajoutées au processus de développement : étude d'opportunité, étude de faisabilité du projet de développement, décision d'informatisation.
- Dans le développement lui-même, la notion de client est prise en compte dans la répartition du travail. Des documents (spécifications, produits) sont produits spécialement pour le client qui servent de base à des points de validation. On progresse d'une étape seulement si le client a validé l'étape précédente.

- Qui dit client, dit aussi qualité de produit vendu. La deux principales qualités attendue sont la validité (réalise les tâches définies dans sa spécification - correction, fiabilité (?)) et la robustesse (aptitude à fonctionner même dans des conditions anormales). Le logiciel doit satisfaire le client. La facilité d'utilisation et son efficacité sont des critères non négligeables pour le client. Enfin, losque le client dispose déjà de produits logiciels convenables, il est rentable de les réutiliser. Le nouveau logiciel doit pouvoir communiquer avec ces produits, on parle de compatibilité.

On peut rassembler la prise en compte des aspects de ce point et celui du précédent (organisation) sous le vocable **gestion de projet**. L'organisation gère plusieurs projets de développement. Les priorités et les moyens sont définis dans sa stratégie de développement. Un développement est lancé si sa faisabilité et sa rentabilité sont établis.

Un dernier élément a une influence essentielle sur le développement. Il s'agit du producteur de logiciel, qu'il soit une entreprise externe (un prestataire de services), ou un service interne. L'activité du producteur suit aussi une logique économique, qui est la rentabilité du développement. examinons plusieurs facteurs.

- Lorsque le client n'est pas entièrement satisfait par son produit, le producteur doit le faire évoluer. On parle de maintenance. La maintenance couvre deux aspects, la correction d'erreurs - la maintenance corrective- et la prise en compte de nouveaux besoins (ou une reformulation des besoins) -la mainteance évolutive-. Une nouvelle qualité est attendue pour le logiciel, son extensibilité (évolutivité). En effet les coûts de maintenance représentent 75% du total, le prix logiciel dépasse désormais largement le prix du matériel. Les systèmes deviennent rapidement obsolète vis-à-vis de l'organisation et des techniques nouvelles. Il faut donc pouvoir les modifier à moindre coût.
- La production de logiciels n'est rentable que si elle produit en série à moindre coûts. Plusieurs critères de qualité interviennent à ce niveau :
  - Le même logiciel peut être implanté sur des systèmes informatiques différents. Le logiciel est portable si sa réimplantation nécessite un minimum d'efforts de développement.
  - A défaut d'utiliser le même logiciel, on peut souhaiter en reprendre une partie. Le logiciel est réutilisable si en reprendre une partie nécessite un minimum d'efforts de développement. La modularité est un facteur essentiel pour ce critère de qualité.
- Lorsque le producteur a une expertise suffisante dans un domaine, il peut mettre au point des outils pour automatiser la production du logiciel. Cette automatisation est un facteur clé de la rentabilité.

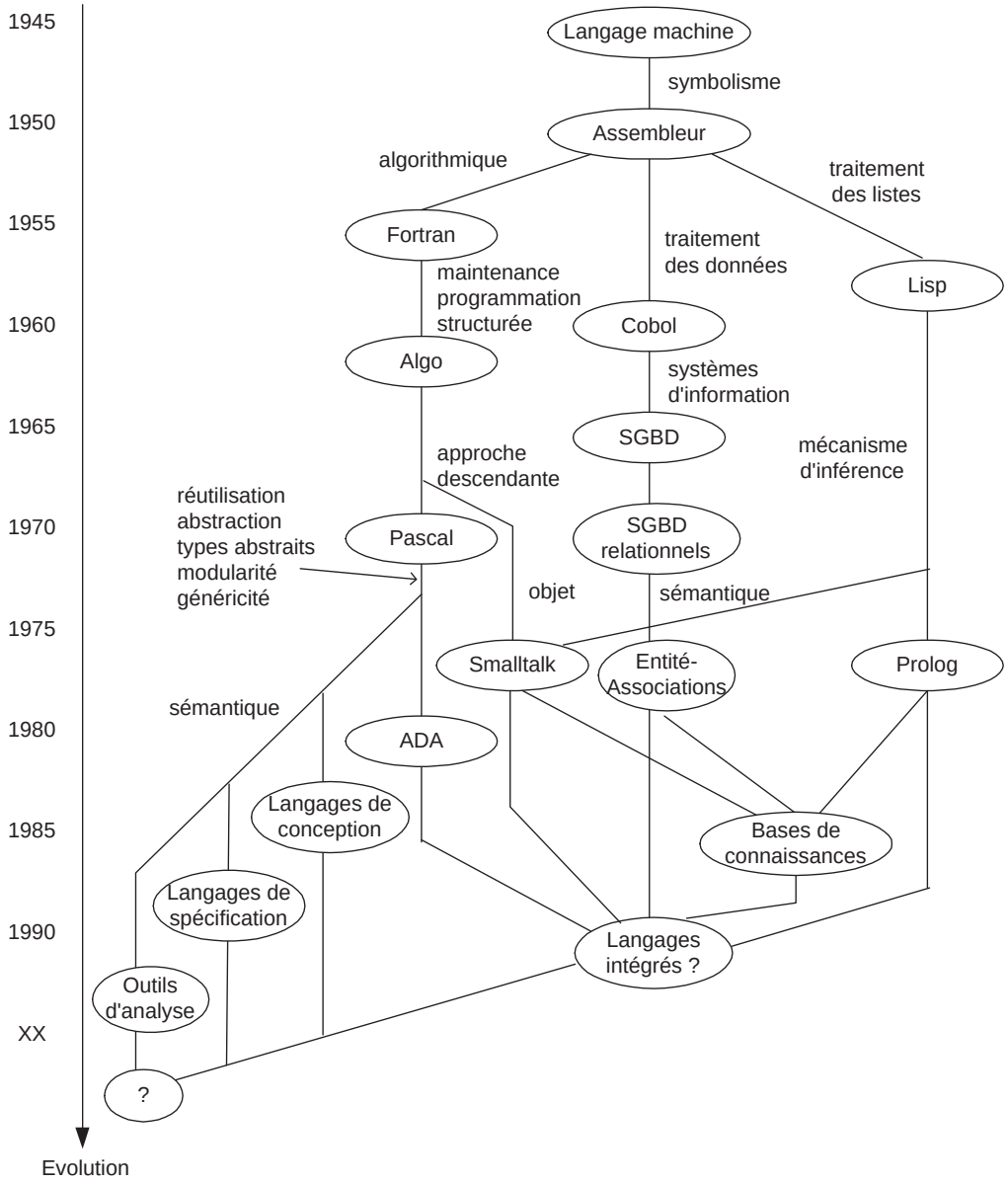
Ce paragraphe met en évidence le point de vue du producteur de logiciel. Il faut produire du logiciel de qualité, rationaliser son développement et rentabiliser les efforts accomplis.

Pour répondre à l'ensemble de ces attentes, il faut mettre au point des méthodes de développement et des outils pour automatiser une partie du travail. La méthode doit permettre de ne pas tergiverser et d'avancer de manière sûre vers le logiciel. L'automatisation accélère la production au cours du processus de développement. L'objectif du génie logiciel est de mettre en œuvre des moyens pour réaliser du logiciel de qualité en respectant des contraintes de coûts.

Le **génie logiciel** est l'art de construire industriellement du logiciel. Le domaine du génie logiciel couvre un large spectre, des langages de programmation à la gestion d'organisations. Les progrès dans le domaine des techniques de programmation à une influence essentielle sur l'évolution du génie logiciel (voir historique). Au départ, le travail des professionnels a consuisté à mettre au point des méthodes de développement pour répondre à des objectifs d'informatisation. Même si la plupart des méthodes sont propriétaires (propres à chaque sociétés), des courants essentiels ont été mis en évidence. Le travail méthodologique existe toujours mais la production d'outils automatisé est devenue un élément essentiel du génie logiciel. Les Ateliers de Génie Logiciel (AGL) sont des environnements de développement supportant une ou plusieurs méthodes de développement par des outils divers (édition, vérification, génération automatique, documentation, réutilisation).

2 Evolution du génie logiciel

Le génie logiciel est un domaine relativement récent : la première conférence sur ce thème eut lieu en 1968. L'historique[1] du GL peut être résumé par quelques faits marquants :



Evolution du génie logiciel

- 1945 : Projets de petite taille, programmation en binaire puis en assembleur. La maintenance est assurée par le développeur.
- 1955 : Apparition de langages évolués qui permettent de développer des projets plus importants.
- 1965 : C'est la crise du logiciel. L'intuition ne suffit plus pour développer correctement le logiciel.
- 1968 : Première conférence sur le sujet.
- 1970 : Programmation structurée, structuration hiérarchique du code 'sans `goto`).
- 1972 : Développement des méthodes de preuves de programmes (peu applicables à grande échelle).
- 1975 : Développer un projet ne consiste pas seulement à le coder mais à le comprendre, le

[1] Synthèse de la section 1.2 du chapitre 1 du cours GL de B. Levrat, Lausane.

spécifier, le concevoir, en des étapes successives. Apparition de la notion de cycle de vie et essais de développement de méthodes adaptées à ces phases.
- 1980 : Après avoir développé des méthodes et des outils de manière isolée, on les rassemble pour former des environnements homogènes. On prend aussi conscience de l'importance des premières phases dans de coût de développement d'un projet.
- 1990 : C'est la décennie de la programmation à objets avec comme objectif la réutilisation de logiciels et le passage aisé d'une application à l'autre pour l'utilisateur.

3 Les méthodes de développement du logiciel

Le développement du logiciel est une partie de ce qu'on appelle communément la gestion de projet. L'activité d'une entreprise, au sens large, concerne un ensemble de projets. Un projet est la réponse à un besoin de l'entreprise, il est caractérisé par l'expression du besoin, la définition d'une ou plusieurs solutions techniques et la gestion des ressources mise en œuvre pour réaliser le projet. La gestion de projet au sens large inclue faisabilité, moyens humains et financiers, planification, impact sur l'organisation automatisée, etc. Consulter [Som92] à ce sujet.

Une méthode sert à conduire le développement. L'application sans réflexion de la méthode nuit à la qualité du produit, l'aspect créativité est essentiel. "Une méthode sert à éviter un pire développement et pas forcément à obtenir un meilleur développement". La méthode sert aussi à restreindre l'empirisme individuel (rétention d'information), la prédominance des informaticiens et le dysfonctionnement entre besoins et réalisation. La méthode facilite la communication entre les acteurs, lévolution et la maintenance des systèmes, l'évaluation de systèmes et la productivité.

Une méthode est une technique de résolution de problèmes [Lau86]. Un aspect clé de la méthode est de privilégier l'étude (l'analyse) avant de commencer à concevoir le logiciel. Il faut bien poser le problème pour bien le résoudre. Le terme de **méthode** recouvre plusieurs notions. Une méthode est à la fois une philosophie dans l'approche des problèmes, une démarche ou un fil conducteur dans la résolution, des outils d'aide et enfin un formalisme ou des normes.

Par exemple, dans la méthode Merise [TRC91] la philosophie est fortement inspirée des bases de données relationnelles avec une dualité données/traitements, de la programmation structurée et des réseaux de Petri. Merise propose une double démarche : par niveau d'abstraction (conceptuel, organisationnel, physique ou opérationnel) et par étapes (études préalables, détaillées, fonctionnelles, techniques et mise en production). Le terme abstraction est synonyme d'éloignement vis-à-vis des considérations matérielles et logicielles. Les étapes servent à baliser le développement et exiger des résultats intermédiaires. Les formalismes Entité/Association, réseaux de Petri sont utilisés. Des outils comme les analyseurs, générateurs, imprimeurs peuvent être mis en place.

Même si aucune étude n'a été menée en ce sens, la méthode universelle n'existe pas. Nous pensons que l'efficacité des méthodes est induite par une semi-spécialisation par domaine d'application. Nous en distinguons trois grandes classes :
- Les systèmes d'information en gestion sont caractérisés par un stockage important d'informations et de nombreux traitements en consultation, mise à jour de ces informations *e.g.* gestion de la clientèle, du stock, traitement de la langue. Les systèmes actuels intègrent une composante réseau (accès distant) qui intervient plus dans l'implantation du système que dans sa modélisation.
- Le calcul sciencifique se caractérise par de nombreux calculs *e.g.* simulation, météo, imagerie. Une base d'information peut servir à stocker les informations de base ou les résultats mais intervient assez peu dans le calcul lui-même. La difficulté principale est la description de modèles mathématiques par des modèles informatiques. Un critère important est la rapidité du traitement.
- L'informatique temps réel se caractérise par une réactivité très forte du système d'information *e.g.* pilotage automatique d'avion, surveillance de centrale nucléaires. Nous rangeons, un peu arbitrairement sous cette dénomination l'informatique embarquée (automatismes), le contrôle de processus (fabrication, surveillance), les applications ù le temps joue un rôle majeur, les

réseaux informatiques. L'interface entre le système d'information et son objet naturel (le processus) prend ici une grande importance. Elle inclue des aspects matériels non négligeables : capteurs pour récupérer les informations, des actionneurs pour piloter objet naturel, bus de communications, etc. [Cal90] propose une caractérisation des systèmes et un classement des systèmes sur les disciplines connexes à l'informatique industrielle.

Nous estimons que ces trois catégories représentent, assez grossièrement, les différents types de systèmes d'information que nous sommes susceptibles de modéliser ici.

En résumé, la méthode permet de construire des modèles (produits, spécifications) dans un ordre donné (démarche, processus).

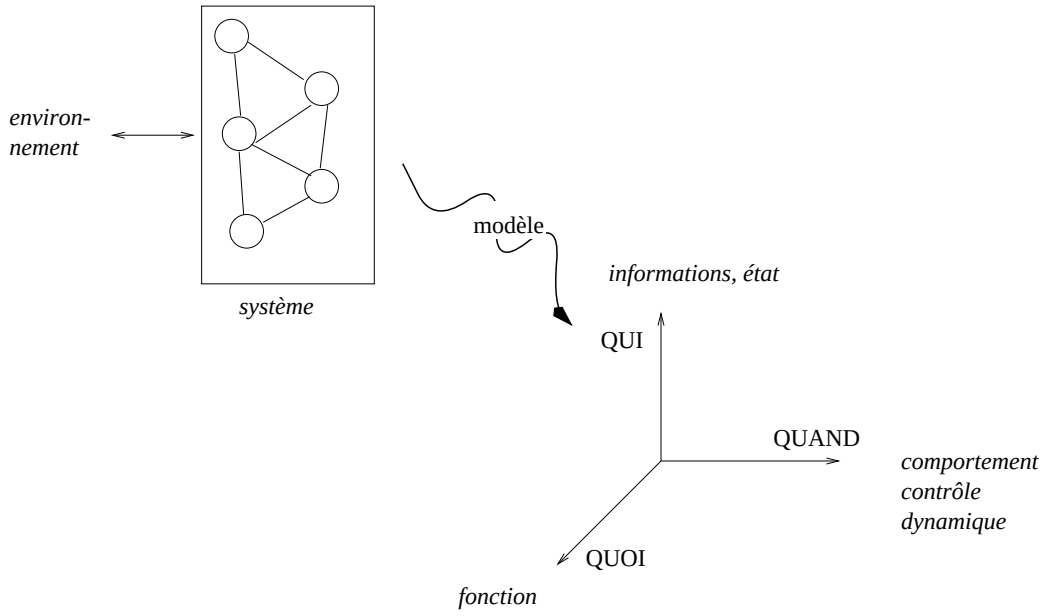
4 Modélisation, produits et spécifications

Un **modèle** (une **représentation**) est une interprétation explicite par son utilisateur de l'idée qu'il se fait d'une situation. Il peut être exprimé par des mathématiques, des symboles, des mots, mais essentiellement, c'est une description d'entités et de relations entre elles. Un modèle est correct s'il permet de répondre aux questions qu'on se pose. Un modèle est **opérationnel** s'il peut être exécuté par une machine.

Par abus de langage, le terme **modèle** désigne souvent à la fois le résultat de la modélisation, la théorie sous-jacente et la notation utilisée. Voici quelques modèles utilisés dans le développement du logiciel.

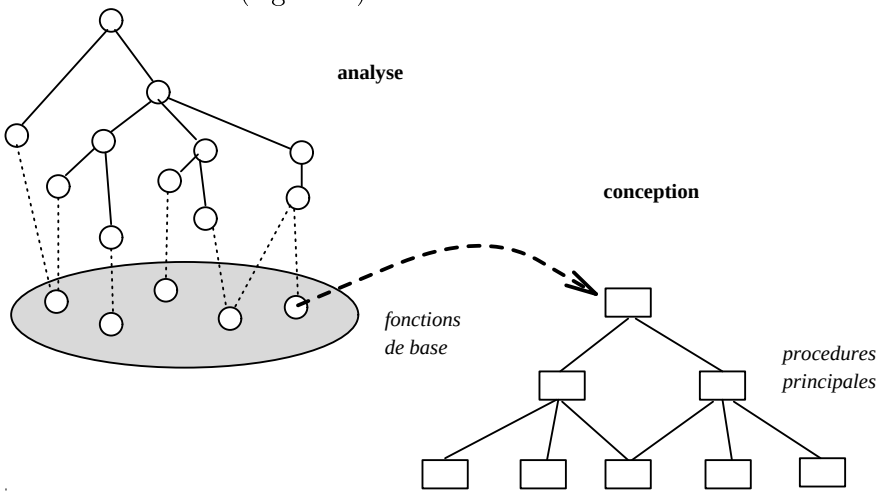
Théorie des systèmes	Analyse modulaire des systèmes	Mélèse, Le Moigne
Théorie des ensembles	LCP/LCS Programmation structurée	Warnier Dijkstra
Concept de base de données	Objet/Relation Relationnel Codasyl	Chen, Tardieu Codd Bachmann
Processus	Réseaux	Petri
Evénements - Procédures	Corig	CGI, Pac, Ariane...
Résultats - Données	Minos	Sema, Atlas, Sligos...
Objet	OOD, OOA	Meyer, Booch, OMT, UML

Dans un système d'information, trois aspects sont souvent mis en évidence : ce que le système manipule (les informations, les données), quand il les manipule (causalité, contrôle, comportement dynamique, événements) et comment il les manipule (les opérations, les fonctions, le comportement fonctionnel).

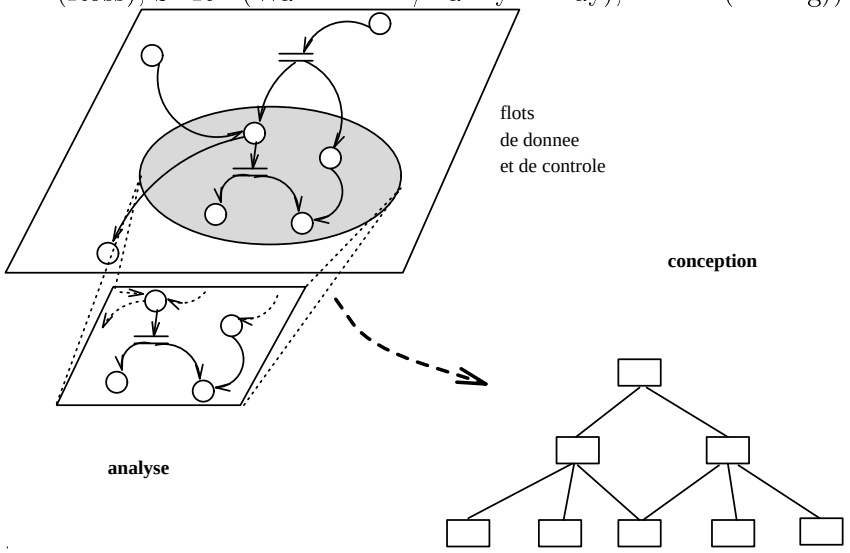


Trois courants majeurs ont dominé les méthodes d'analyse et de conception :

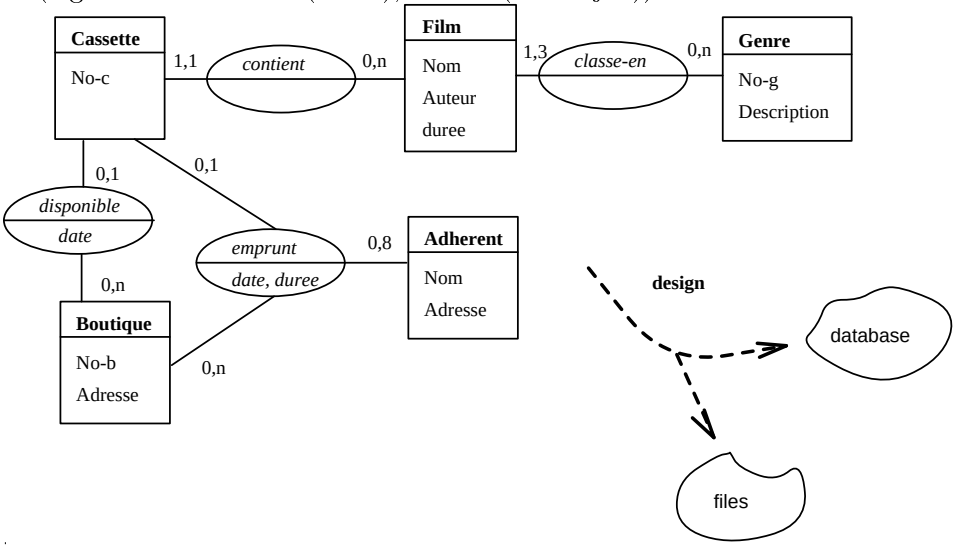
– l'approche fonctionnelle dans laquelle le système est perçu en termes de fonctions et sous-fonctions munies d'une interface (e.g. JSD)



– l'approche flots de données dans laquelle on exprime la transformation des données (e.g. SA (De-Marco), SADT (Ross), SART (Ward-Mellor/Hatley-Pirbay), ESML (Boeing))



– l'approche modèle de donnée pour laquelle l'accent est mis sur la partie statique du système d'information (e.g. Entité-Relation (Chen), NIAM (Verhaijen))



Les théories utilisées pour chaque axe sont cohérentes et permettent de démontrer des propriétés

propres à ces axes. Par contre, les propriétés dépendant de plusieurs axes sont bien plus difficiles à démontrer.

5 Le processus de développement

Le développement du logiciel est découpé en plusieurs activités :

1. l'**analyse des besoins** définit les services du système, ses contraintes et ses buts en consultant les utilisateurs du système. Une étude d'opportunité peut être menée pour savoir si le système est réalisable et donner une approximation de la rentabilité de ce système.
   Synonymes : analyse préalable, *user requirements analysis.*

2. l'**analyse** est la construction d'un modèle (une spécification) du système à partir de l'analyse des besoins. A partir de ce modèle on peut proposer plusieurs scénarii et réaliser une étude de faisabilité.
   Synonymes : spécification des besoins, analyse préalable, étape conceptuelle, analyse conceptuelle, conception préliminaire, modélisation conceptuelle, *analysis, user requirement specification, requirements engineering.*

3. la **conception** est une proposition de solution au problème spécifié dans l'analyse. Elle définit la solution retenue par prise en compte des caractéristiques logiques d'usage du futur système d'information et des moyens de réalisation, humains, techniques et organisationnels. On distingue souvent la conception système et la conception détaillée. La première a pour objectif de donner l'architecture globale du systèmes (i.e. les différentes parties) et la seconde décrit chaque partie du système. Cette spécification du logicielle reste indépendante de tout moyen de réalisation.
   Synonymes : étape fonctionnelle, analyse fonctionnelle, analyse organique, étape logique, conception technique, *global design, system design, detailed design, design engineering.*

4. la **réalisation** et le **test** produit la solution exécutable en termes de programmes. Pour les logiciels complexes, on distingue l'implantation des différentes parties et leur validation par des tests unitaires, et l'impantation du système complet par **intégration** des parties et tests systèmes. On vérifie ainsi que les spécifications des besoins sont satisfaites. Synonymes : codage, implantation, mise en œuvre, programmation, *coding, implementation, program testing.*

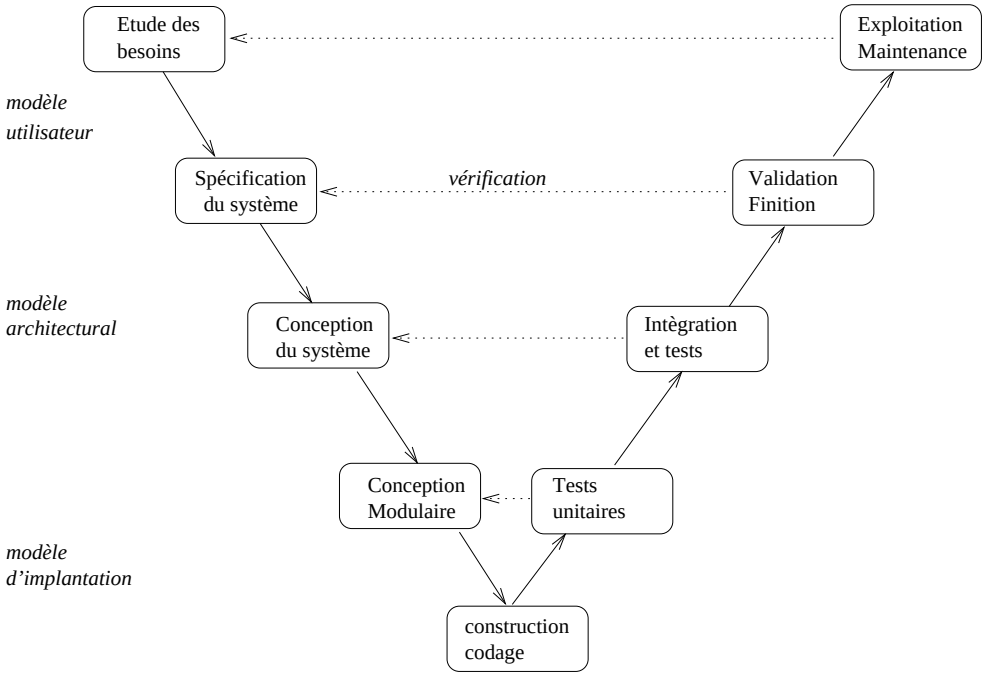
5. l'**installation** du logiciel règle les problèmes de mise en place dans l'organisation.

6. la **maintenance** adapte la solution conceptuelle aux changements organisationnels et aux évolutions technologiques (évolutive). Elle corrige aussi les erreurs accumulées dans les phases précédentes (curative). Le processus de maintenance est un processus itératif dont l'activité répétée est un cycle de développement complet (de la spécification à la réalisation).

5.1 Cycles de vie

Plusieurs cycles de vie ont été proposés pour organiser ces tâches. Trois grandes catégories sont retenues :

- les modèles linéaires (modèle de la "cascade", modèle en "V") dans lesquels les différentes étapes ci-dessus sont réalisées tour à tour. On commence la suivante une fois la précédente achevée. En cas d'erreur, on revient sur l'étape précédente. On parle aussi d'approche descendante. Dans le modèle en "V", on insiste sur une séparation entre la construction des diverses spécification et leur validation a posteriori (tests unitaires, tests d'intégration, etc) ainsi que sur le niveau d'abstraction (utilisateur/architecture/implantation). Ce sont les modèles de référence [BR85, Som92].



- les modèles contractuels sont une suite de contrats entre client et fournisseurs. Les méthodes formelles en sont un bon exemple, elles permettent la transformation de spécifications.
- les modèles itératifs ou à spirale [Boe88] permettent un développement incrémental notamment par le prototypage. Ces modèles intègrent des notions de risques à évaluer, de spécifications partielles et de résultats intermédiaires. Le prototypage devient une technique de modélisation. Ce modèle est évolutif par nature mais rend difficile la planification et il doit éviter les redondances.
- les modèles mixtes comme le modèle "X" [Hod91] s'inspirent de plusieurs styles. Les modèles linéaires ne dépendent pas de la technologie utilisée, or le processus de développement en dépend par nature. Le modèle "X" prend en compte le modèle objet. Deux cycles inversés sont en fait décrits, l'un pour une activité de synthèse d'un nouveau système et l'autre pour une activité d'acquisition de systèmes et de composants en vue de les réutiliser.

5.2 Produits, spécification

A chaque étape correspond un document résultat, qu'on nomme souvent produit ou spécification. Une **spécification** est un ensemble de modèles. La spécification initiale est le cahier des charges et la spécification finale est le programme (logiciel). On parle ainsi de spécification des besoins, spécification détaillée, spécification fonctionnelle ou encore spécification formelle. Par convention, le terme *spécification* désignera de façon générique les documents résultant d'une étape dans le processus de développement (*deliverables* en anglais). Le terme *spécification du logiciel*, quant à lui, désignera la description plus ou moins détaillée du comportement attendu du logiciel. La qualité des spécifications est abordée dans le chapitre 1 de [AV00].

6 La validation

Valider c'est contrôler que le (produit) résultat correspond à ce qui était attendu. La validation est un contrôle qui fait intervenir largement les "utilisateurs". Vérifier c'est contrôler que le produit respecte le cahier des charges [Som92]. La vérification est donc plus une "affaire" de spécialistes.

Les études des facteurs de coûts du logiciel de [Boe82] et [Cal90] montrent l'importance de la validation. Plus une erreur est détectée tardivement, plus sa correction est chère. C'est pourquoi l'effort doit être porté sur la spécification plutôt que la conception. De plus la validation doit être

réalisée par une équipe pluri-disciplinaire, comprenant des informaticiens certes, mais aussi des utilisateurs et des intervenants extérieurs au projet. Ces participants n'auront pas le même point de vue. Par exemple, les utilisateurs seront à priori détachés de la solution, les intervenants extérieurs permettront d'élargir le champs d'investigation. La confrontation des points de vue devrait aboutir à un compromis raisonnable. Afin de s'assurer de la bonne marche d'un développement, un contrôle est donc obligatoire. Il est généralement fait entre les étapes.

Un point clé de la vérification et plus encore de la validation est la lisibilité des documents. En effet, comment les utilisateurs pourront-ils vérifier l'exactitude d'une spécifications, ou pire d'un programme, s'ils ne comprennent pas ou pas bien l'information qui leur est communiquée. La lisibilité dépend du niveau de formalisme du langage utilisé et de la pertinence des informations qui accompagnent les spécifications. Si on considère la lisibilité par la concision, la précision et l'exactitude, alors les spécifications formelles, que nous aborderons en détail dans le second volume, sont un apport considérable. Si on considère la lisibilité par l'abstraction et la rapidité de lecture, alors les spécifications graphiques sont plus riches. Malheureusement, ces dernières souffrent d'un manque de précision qui engendre des ambiguités ou des imprécisions.

La vérification (ou le contrôle de la correction) se fait de deux manières : soit par preuve de propriétés de la spécification au moyens d'outils spécialisés (prouveurs), soit par construction automatique (et prouvée) de programme.

Hormis la lecture des spécifications, à laquelle nous avons fait allusion ci-dessus, le **maquettage** (ou **prototypage**) et les tests sont les deux principaux moyens de validation. Le premier permet à petite échelle de se rendre compte approximativement de l'allure générale du résultat et des problèmes pouvant intervenir. Le second permet de vérifier des hypothèses, des objectifs et des contraintes. Il nécessite une formalisation précise des besoins.

6.1 Test

Par définition, le test est la vérification de cohérence entre la spécification et la réalisation, et ce au moyen d'un échantillon (représentatif). La validité du test repose entièrement sur la pertinence du sous-ensemble de données choisi. Deux "écoles" divisent les spécialistes du test :

- les tenants des techniques de génération systématique de jeux d'essais, soit fonctionnels (boîte noire), soit structurels (boîte blanche). Les stratégies de sélection de tests utilisées reposent sur la correction (i.e. les défauts). Les jeux de données sont produits systématiquement à partir d'éléments comme le code (boîte blanche) ou la spécification (boîte noire). Aux stratégies de sélections de données de tests peuvent être associées des *mesures d'efficacité ou de couverture*. Les plus couramment utilisées sont les couvertures de branchement et de modules. Une mesure de couverture est définie par le rapport entre le nombre de jeux de données de test satisfaisant le critère de couverture et le nombre total de jeux de données utilisés. Il s'agit de mesure *dynamique. Seules les spécifications formelles ou semi-formelles permettent d'évaluer automatiquement une couverture.* Des mesures de *testabilité* peuvent aussi être associées aux stratégies de sélection. On les définit comme le nombre minimal de jeux de données nécessaires pour satisfaire un critère donné en supposant que ces jeux peuvent être exécutés. Il s'agit de mesures *statiques.*
- les tenants du test statistique et des modèles de fiabilité. Les domaines de données sont échantillonnés selon des lois de répartition *statistiques.* L'échantillon doit reflèter la répartition. Des modèles de *fiabilité* permettent d'établir les mesures de fiabilité. Quatre catégories sont distinguées :
  - le temps entre deux pannes,
  - le nombre de pannes observées pendant une durée donnée,
  - le nombre de pannes observées pour un ensemble de défauts introduits dans le programme,
  - le nombre de pannes observées pour un jeu de données de test, sélectionné selon un profil d'utilisation donné.

En raison de contraintes de délais et de coûts, il n'est pas toujours possible d'utiliser toutes les stratégies de test, il faut panacher.

6.2 Maquettage

Dans l'optique de limiter le nombre d'erreur dans le logiciel final, on peut soit améliorer la programmation, soit mieux formuler les spécifications, soit construire des prototypes [Cho86]. La programmation structurée et le génie logiciel (outils, environnements, guides, automatisation) ont apporté un premier élément de réponse. Les langages de spécification et les outils qui les accompagnent (éditeurs, bibliothèques, interprètes) doivent permettre d'exprimer clairement les éléments du problème. Enfin le **prototypage** correspond à un besoin de vérifier le comportement réel du produit en cours de développement. Le prototype sert aussi à la communication entre le spécifieur et le réalisateur, il est une approximation de l'interprétation que le réalisateur fait de la spécification. Ces trois aspects sont complémentaires.

Le prototypage consiste en quatre étapes : la *sélection fonctionnelle* (choix des fonctions à réaliser), la *construction* (éventuellement confondue avec la spécification si le langage de cette dernière est exécutable), l'*évaluation* ("feedback" sur le développement), l'*utilisation ultérieure* du prototype, soit est mis de côté soit fait partie intégrante du produit.

Prototyper n'est pas spécifier. Le prototypage doit être préparé pour réussir. La spécification est indépendante de toute mise en œuvre. La spécification peut servir de contrat entre l'utilisateur et le réalisateur. Enfin et surtout, la spécification est une description cohérente et complète du produit alors que la simulation en est une approximation.

7 Conclusion

Pour gérer efficacement le développement des applications complexes, il faut améliorer la qualité des produits (spécifications et logiciel) et celle du processus de développement. Ces deux points font toujours l'objet de recherches actives. La question majeure est "*Quel(s) modèle(s) doit-on utiliser pour répondre aux besoins ?*". Actuellement, deux techniques émergent : le modèle objet pour la structuration des spécifications et les méthodes formelles pour la construction et la validation des spécifications.

Il est maintenant clairement reconnu que le résultat d'un développement du logiciel dépend de l'effort fourni en amont de la programmation. Plus l'étude sera rigoureuse et précise, moins la réalisation sera erronée et inadaptée. Les méthodes formelles, basées sur des notations mathématiques, apportent rigueur et abstraction dans la spécification du logiciel [Abr84, Mey85]. Théoriquement, la construction du programme se fait alors par introduction progressive des structures de données et de contrôle du langage de programmation cible. Chaque étape dans ce processus de **concrétisation** doit être prouvée [JS90]. Et un des intérêts des méthodes formelles est l'automatisation de certaines parties du processus de développement. Dans la pratique, le processus n'est pas aussi linéaire car la conception est une activité créatrice. La spécification formelle est l'outil de base de la validation et du test de programmes.

Un second facteur influe sur la qualité des spécifications produites au cours du développement du logiciel (définition des besoins, spécifications détaillées, programmes), c'est la structuration de la spécification. En développement dit structuré (*structured analysis*, *structured design*, *structured programming*), il y a une dualité données/traitements qui permet une itération dans le processus de développement mais perturbe l'évolution du produit et la réutilisation du logiciel. L'approche modulaire vise au contraire à décomposer le système en éléments relativement indépendants contenant à la fois des structures de données et des opérations. Cette approche distingue de plus une partie interface avec les autres modules et une partie implantation. On rejoint ici la notion de concrétisation : plusieurs implantations peuvent réaliser une interface. Les critères de couplage et de cohésion permettent de juger la qualité du système. Le modèle objet est une approche modulaire

à cohésion forte. Il s'impose actuellement dans toutes les étapes du développement car c'est un concept fédérateur (programmation, conception, bases de données, CAO, etc.). L'objectif est alors de construire des sous-ensembles logiciels de qualité (plus lisibles, maintenables, réutilisables) qu'on pourra assembler dans différentes application. On passe ainsi de la culture projet à la culture de production [Mey89].

Références

[Abr84] Jean-Raymond Abrial. Spécifier ou comment matérialiser l'abstrait. *Technique et Science Informatique*, 3(3) :201–219, 1984.

[AV00] Pascal André and Alain Vailly. *Conception de Systèmes d'Information*, volume 1. Editions Ellipses, 2000. ISBN 2-7298-0479-X.

[Boe82] Barry W. Boehm. Les facteurs du coût logiciel. *Technique et Science Informatique*, 1(1) :5–24, 1982.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5) :61–72, May 1988.

[BR85] Barry W. Boehm and Rony Ross. La gestion de projets logiciels selon la théorie W : une étude de cas. *Revue Génie Logiciel et Systèmes Experts*, 15 :4–20, septembre 1985.

[Cal90] Jean-Paul Calvez. *Spécification et conception des systèmes : une méthodologie.* Masson, mai 1990.

[Cho86] Christine Choppy. Techniques et aspects du prototypage. *Revue Génie Logiciel*, 3 :4–45, March 1986.

[Hod91] Ralph Hodgson. The X-Model : A Process Model for Object-Oriented Software Development. In *Actes des quatrièmes journées internationales sur le Génie logiciel et ses applications*, pages 713–728, Toulouse, December 1991.

[JS90] Cliff B. Jones and Roger C. Shaw. *Case Studies Systematic Software Development.* International Series in Computer Science. Prentice-Hall, 2 edition, 1990. ISBN 0-13-116088-5.

[Lau86] Jean.-Louis Laurière. *Résolution de problèmes par l'Homme et la machine.* Eyrolles, 1986.

[Mey85] Bertrand Meyer. On Formalism in Specifications. *IEEE Software*, 2(1) :6–26, January 1985.

[Mey89] Bertrand Meyer. The New Culture of Software Development : Reflections on the Practice of OOD. In Jean Bezivin and Bertrand Meyer, editors, *Proceedings of TOOLS'89*, Paris, November 1989.

[Som92] Ian Sommerville. *Le génie logiciel.* Addison-Wesley France, 4e edition, October 1992.

[TRC91] Hubert Tardieu, Arnold Rochfeld, and René Coletti. *La méthode Merise, Tome 1 : Principes et outils.* Editions d'Organisation, 1991. ISBN 2-7081-1106-X, voir aussi les tomes 2 et 3.