

# Sommaire

<b>1. INTRODUCTION AU GÉNIE LOGICIEL</b>	<b>2</b>
<b>2. LA CRISE DU LOGICIEL : ÉTAT DES LIEUX</b>	<b>3</b>
<b>3. LA MÉTAPHORE DU GÉNIE CIVIL</b>	<b>5</b>
<b>4. LE PROCESSUS DE DÉVELOPPEMENT</b>	<b>6</b>
4.1. La modélisation de la démarche	7
4.2. Le niveau de maturité	8
4.3. Cycles de vie du logiciel	8
4.3.1. Phases	8
4.3.2. Waterfall	9
4.3.3. en V	9
4.3.4. en spirale	10
4.4. Conclusion	10
<b>5. LES PRINCIPES D'ORGANISATION</b>	<b>10</b>
5.1. Les ressources humaines	10
5.2. L'estimation	11
5.3. La planification	12
5.4. La qualité du logiciel	13
5.4.1. Définition et mesure de la qualité à atteindre	13
5.4.2. Processus pour obtenir la qualité	14
5.5. L'analyse de risque	15
5.6. La gestion de configuration	17
<b>6. LES TECHNIQUES</b>	<b>18</b>
6.1. La modélisation du produit	18
6.2. Méthodes	19
6.3. Analyse des besoins et analyse de domaine	21
6.4. Formes de conceptions (design patterns).	21
6.5. Composants réutilisables	22
6.6. Le test du logiciel	23
<b>7. OUTILS</b>	<b>25</b>
<b>8. CONCLUSION</b>	<b>25</b>

## 1. Introduction au génie logiciel

La place des logiciels dans les systèmes complexes augmente régulièrement, or, les producteurs de logiciels ne respectent que rarement les coûts ou les délais de développement, quand ce ne sont pas les besoins même de leurs clients ! Le génie logiciel est né de ce constat : l'industrie du logiciel est immature et ses résultats ne sont pas à la hauteur des espoirs des clients.

Comparée aux autres industries, la production de logiciel souffre de maux majeurs :

1. sa jeunesse, il n'y que quelques dizaines d'années que l'on programme,
2. le caractère abstrait du produit,
3. la taille des systèmes automatisés,
4. l'évolution rapide des technologies et des techniques.

Si le premier peut, avec le temps, et en apprenant, trouver une solution, le deuxième est caractéristique du logiciel et, de ce fait, nécessite une approche et une rigueur particulière. Les deux derniers peuvent difficilement être maîtrisés, sauf à ralentir l'innovation !

Le génie logiciel apporte un début de solution. Il offre une série de techniques et de modèles qui exploités avec justesse facilitent et améliorent la production de logiciels.

La présentation de ces techniques et modèles revêt un caractère descriptif important et, comme ceux-ci ne s'appliquent qu'à des projets de grande taille, elle reste pour des étudiants une vue de l'esprit, souvent ennuyeuse. J'en suis bien conscient, mais il me semble tout de même indispensable qu'une formation de haut niveau aborde ces thèmes. L'objectif de ces cours n'est pas d'acquérir des savoir-faire directement exploitables dans une entreprise (de production de logiciel), mais plutôt d'ouvrir un nouveau champ de connaissances.

A ce propos, le génie logiciel aborde un aspect technique trop souvent ignoré : la modélisation et l'organisation du travail. Je ne peux m'empêcher de citer cette remarque d'un des dirigeants du Department of Defense (DoD) des USA : « Les problèmes difficiles aujourd'hui ne sont pas techniques mais organisationnels ». Ce cours pourra alimenter la réflexion d'ingénieur ailleurs que dans le domaine informatique...D'ailleurs, même si vous ne vous sentez pas l'âme d'un producteur de systèmes informatiques, les informations contenues dans ce cours ont tout de même de grandes chances de vous servir en tant qu'utilisateur ou client de tels systèmes.

Le parcours du génie logiciel proposé commence par un exposé de ce qu'on appelle « la crise du logiciel ». Pour présenter l'ampleur du domaine, nous ferons une analogie avec le génie civil ce qui sera l'occasion de dresser la liste des points clés du développement logiciel. La connaissance de ces points clés et leur bonne utilisation permettent de réduire les risques de dérapage d'un projet et d'accroître la confiance réciproque liant le client et le fournisseur d'un produit. La suite du cours abordera plus précisément les techniques d'estimation, de qualité logiciel, de gestion de configuration, de test et de gestion du risque. Quelques éléments de méthodologie informatique seront aussi abordés.

Les aspects « gestion de projet » ne sont qu'esquissés. Ils font l'affaire d'un cours spécial et exploitent des techniques bien plus générales que celles présentées ici.

Le domaine des télécommunications repose sur des architectures et des théories de communication d'information qu'il faut connaître, mais la plus grosse part du marché se situe aujourd'hui au niveau des applications et des services que ces systèmes peuvent offrir ; d'autant plus que ces services deviennent de plus en plus sophistiqués, et requièrent une vision large de la programmation à grande échelle. Programmer un cas d'école est loin de permettre d'aborder toutes les difficultés du « programming in the large » (système de grande taille) ou du "programming in the duration" (faire vivre un système des dizaines d'années). C'est pourquoi, le génie logiciel est incontournable.

Incontournable, car il faut bien connaître les problèmes et leurs solutions techniques et d'organisation qui ont été élaborées par nos prédécesseurs, pour savoir lesquelles retenir et appliquer à chaque projet. Car, s'il est une caractéristique primordiale d'un bon chef de projet, c'est celle de choisir l'ensemble *minimal* des techniques de génie logiciel à appliquer pour garantir un fonctionnement optimal du projet et éviter ainsi une lourdeur qui pourrait passer pour de la bureaucratie...

## 2. La crise du logiciel : état des lieux

La collecte d'information pour dresser un état des lieux de la production de logiciel est difficile. En France surtout, où la transparence n'est pas une vertu suffisamment travaillée, et où la peur de montrer que « nous » ne sommes pas parfaits est grande. Heureusement, sur ce point, les anglosaxons, et plus particulièrement les américains osent faire des études et publier leurs résultats. Il se trouve, par chance, que les États-Unis sont aussi les plus gros consommateurs et producteurs de logiciels.

On estime donc qu'aujourd'hui 90% du coût d'un système informatique repose sur le logiciel. Le développement et la maintenance des logiciels représentent dans le monde près de 300G\$ par an. Et malgré ce marché, les industriels considèrent encore l'industrie du logiciel comme « risquée », « mal maîtrisée », source de coûteux dépassements de budget (dans plus de 50% des cas) et de délai (dans environ 65% des projets). Les causes se partagent entre des difficultés techniques ou algorithmiques et des problèmes d'organisation ou de contrôle du développement.

Pourtant les technologies évoluent ; les langages de programmation se modernisent pour améliorer la productivité et la fiabilité des programmes, les composants logiciels commencent à se multiplier pour favoriser la capitalisation des expériences et des savoir-faire. En fait, on constate que si la technologie évolue, l'industrie souffre d'une inertie assez forte. L'un des défis majeurs de ces dernières années est l'évolution des programmeurs COBOL, encore les plus nombreux, vers des technologies objets. La formation et l'organisation de la formation sont des facteurs très importants à prendre en compte dans les projets.

Le rôle d'une école d'ingénieur est aussi de garantir à ses étudiants une pérennité assez grande des savoirs et savoir-faire qu'elle distille. Or, les sciences et technologies de l'informatique évoluent si vite que depuis 20 ans, il a fallu reconstruire les cours tous les 3 ans environ alors que les industriels ne commencent à intégrer les technologies objets, vieilles de 20 ans, que depuis quelques années. Et que dire des techniques de spécifications et de preuves formelles de

programmes qui ne sont encore expérimentées, pourtant avec succès, que dans de trop rares entreprises et enseignées dans cette école qu'en option ?

Si les technologies évoluent très vite, leur utilisation se répand doucement, l'organisation du développement logiciel progresse lentement, mais surtout son déploiement dans l'industrie se fait à tout petit pas ; changer les habitudes d'organisation des personnes est plus difficile que de changer leurs outils de travail, même quand la volonté est là. Le développement de logiciel est toujours considéré comme une activité « créatrice » qui ne peut être rationalisée. Le mythe du « génial programmeur » traîne encore dans les esprits. Pourtant, tout comme n'importe quelle activité menant à satisfaire les besoins de personnes, la production de logiciel se modélise, se contrôle, bref, « s'ingénieurise ».

En 1990, une étude effectuée sur 9 projets du Department of Defense des USA, correspondant à quelques millions de dollars, affichait les résultats suivants :

- 28.8% avait été payé mais non livré
- 19.2% avait été transformé ou abandonné
- 47.27% n'avait pas été utilisé avec succès
- 2.95% avait été utilisé après quelques modifications
- 1.77% avait été utilisé tel que livré

Ca laisse à réfléchir !

Quelles difficultés principales est-on amené à rencontrer ?

1. Le manque de connaissance et de compréhension de ce que le développement logiciel implique. On considère trop que ce genre de chose s'apprend « sur le tas », mais, il est alors déjà trop tard, de mauvais plis sont pris (les décideurs actuels sont souvent des autodidactes de l'informatique...qui croient savoir que...)
2. Le manque de rigueur dans la mise en place des techniques d'organisation. Les bonnes intentions sont contraignantes et paraissent parfois alourdir un processus déjà sous de fortes contraintes budgétaires et de calendrier.
3. Le manque d'outils pour maîtriser les risques du développement. Les outils d'organisation de prévision, de contrôles sont complexes et difficiles à mettre en œuvre.
4. Le manque d'anticipation et d'analyse des risques inhérents au développement logiciel.

Quelles particularités le logiciel possède-t-il par rapport à d'autres produits ?

1. Un produit abstrait (papier, fichier, programme) qu'il est très simple de faire évoluer. Déplacer quelques lignes de programme est beaucoup plus facile que de déplacer une bougie dans un moteur ou ajouter un radiateur dans une maison ; par contre l'impact est très certainement moins bien maîtrisé !
2. Difficulté de spécifier les besoins. L'automatisation apportée par l'informatique force à modéliser/comprendre des processus de fonctionnement et d'organisation de systèmes complexes (fusée, entreprise, cabinet médical, etc.) dont il est très difficile d'obtenir une vision globale (complète) et cohérente. Dans une entreprise, les personnes s'arrangent pour que les défauts de fonctionnement soient compensés par leur initiatives...
3. Un logiciel doit évoluer pour conserver son utilité. L'automatisation fait apparaître de nouveaux besoins, de nouvelles possibilités et des défauts qui n'avaient pu être envisagés avant l'automatisation. Les coûts élevés des logiciels en font des investissements lourds qui ne sont supportables que si leur durée de vie est longue.

4. Difficulté de reconnaître que le logiciel coûte cher, malgré les chiffres !
5. Les métiers sont encore peu séparés ; de l'analyste au programmeur, tout le monde utilise des technologies proches - comme si l'architecte faisait la plomberie !

Les techniques du génie logiciel consistent donc à rationaliser pour maîtriser l'ensemble du processus de production de logiciel. Le génie logiciel s'appuie sur :

- une modélisation du processus de développement du logiciel (cf. §4.)
- un ensemble de principes et outils d'organisation de ce développement (cf. §5.)
- des techniques spécifiques au produit qu'est le logiciel (cf. §6.)
- des outils qui automatisent ou aident au développement (cf. §7.)

Mais avant d'aborder ces différents points, faisons une comparaison intéressante avec le génie civil afin d'avoir un aperçu de l'étendu du génie logiciel.

### 3. La métaphore du génie civil

Pour construire une maison, il ne suffit pas de savoir couper quelques planches et planter des clous. De très nombreux aspects sont à prendre en compte qui interagissent les uns avec les autres. Même si chacune des techniques mises en œuvre est simple (mais nécessite un savoir-faire important), leur interaction, leur interrelation oblige à s'organiser ; c'est pourquoi on fait souvent appel à un spécialiste, le maître d'œuvre, un architecte.

Son rôle consiste, entre autre, à :

1. analyser et comprendre les besoins,
  - Concevoir une maison adaptée aux personnes qui l'habitent, esthétiquement, fonctionnellement.
  - *Élaborer un logiciel adapté aux besoins de l'utilisateur est la moindre des choses, mais pas la plus simple à définir ; l'utilisateur ne sait pas toujours précisément ce qu'il veut ; c'est le rôle de l'analyse des besoins.*
2. tenir compte des réalités économiques (le budget disponible),
  - Du marbre partout et 12 chambres n'est pas à la portée de toutes les bourses...il faut être capable de faire une prévision du coût de la maison.
  - *Le logiciel est un produit abstrait dont il est difficile de mesurer le coût a priori ; c'est le rôle des techniques d'estimation du logiciel et de l'analyse de risque.*
3. proposer des solutions architecturales,
  - différentes propositions sont faites quant à l'agencement des pièces, de l'esthétique extérieure, de l'aménagement intérieur.
  - *on étudie différentes solutions techniques pour satisfaire aux besoins exprimés ; c'est le rôle de la conception.*
4. contacter tous les corps de métier nécessaires,
  - Maçons, couvreurs, plombier, électricien, plâtrier, décorateur, etc.
  - *Ergonome, spécialistes de base de données, de distribution, de télécommunication, de tests, d'interface homme-machine.*
5. s'assurer que toutes les dispositions/contraintes légales sont satisfaites,

- Plan d'occupation des sols, permis de construire, etc.
  - *Droits d'utilisation des logiciels, loi informatique et liberté.*
6. planifier l'enchaînement des interventions,
  7. faire faire les travaux,
    - prévenir les différents corps de métier que c'est à eux d'intervenir.
    - *c'est le rôle de la réalisation, du codage*
  8. s'assurer de l'avancement des travaux,
  9. s'assurer de la qualité des travaux réalisés,
    - vis-à-vis des besoins exprimés, vérifier que ce qui est réalisé est conforme.
    - *c'est le rôle de la qualité et des tests.*
  10. gérer les éventuels conflits,
  11. prévoir la recette de la maison (remise des clés).

Pour construire la maison de nombreux outils seront utiles :

1. Une méthode qu'applique l'architecte
  - comment faire pour comprendre les besoins, comment concevoir, comment ne rien oublier (complétude), comment assurer la cohérence du tout, etc.
2. Des documents
  - contrats précisant les engagements de chacun
  - plans
    - élévations, détails, câblages, de nombreux plans sont utiles à la description des travaux
    - *architectures logiciel et matérielle, schémas de conception, spécifications d'interfaces, grammaires, etc.*
3. Des outils spécifiques à chaque corps de métier
  - pelleuse, truelle, crayon, etc.
  - *éditeurs, débogueur, gestion de version, analyseur de code (profiler), etc.*

Comme on le voit dans cette comparaison, la complexité de la conception de logiciel ne vient pas que de l'aspect technique, mais aussi, et presque essentiellement, de l'organisation. Le nombre d'outils et de techniques à utiliser, d'acteurs à faire communiquer rend la programmation « à grande échelle » autrement plus difficile que la résolution d'un problème d'algorithmique élémentaire en quelques lignes de codes.

Comme souvent, pour maîtriser cette complexité il convient de modéliser le système - ici le processus de développement - pour le rendre analysable, répétable, mesurable ; c'est le rôle du génie logiciel.

#### **4. Le processus de développement**

La notion de cycle de vie n'est pas spécifique au logiciel, mais le cas du logiciel est assez particulier car contrairement à la plupart des autres produits, le logiciel ne possède pas (ou quasiment pas) de phase de production ; une fois le logiciel créé, sa diffusion consiste uniquement à réaliser des copies de données électroniques... De plus, comme signalé précédemment, le logiciel est extrêmement flexible, adaptable, tout en étant très sensible à de tout

petits détails. Il est donc nécessaire de mettre en place des procédures précises d'évolution et de contrôle du développement.

#### 4.1. La modélisation de la démarche

Le processus de développement industriel peut être décomposé en trois grandes activités :

1. **Le processus technique** : il modélise la procédure à suivre pour réaliser le produit.
2. **Le processus de gestion** : il modélise la procédure à suivre pour contrôler les délais et les coûts.
3. **Le processus qualité** : il modélise la procédure à suivre pour garantir la qualité du produit, mais aussi des différents processus participant au développement du produit (gestion et qualité).

Chacun de ces processus peut lui-même être décomposé en trois grandes phases :

1. La **prévision** : consiste au début du processus à annoncer comment le processus se déroulera.
2. Le **suivi** et le **contrôle** : consiste au cours du processus à vérifier que ce qui se déroule est conforme à ce qui est annoncé ; sinon il faut, soit modifier les prévisions, soit agir sur le processus.
3. L'**analyse** : consiste à la fin du processus à revenir sur ce qui s'est passé pour apprendre et améliorer la prévision et le contrôle des futures réalisations (Feed-back).

Le processus technique est décrit par le *cycle de vie* que l'on choisit. La prévision est réalisée par *l'analyse des besoins*, le suivi par les différents choix de *conception* et d'implantation faits, et l'analyse par les *tests de validations* effectués pour voir si le résultat obtenu est bien conforme aux besoins exprimés.

Le processus de gestion commence par une *estimation* des durées, coûts et efforts humains nécessaires à la réalisation de chacune des activités décrites par le cycle de vie. Des *mesures d'avancement* permettent d'effectuer le suivi et le contrôle du déroulement du projet afin d'anticiper d'éventuels écarts et agir au plus tôt pour corriger la situation. L'analyse des données collectées doit assurer une amélioration du processus d'estimation et de suivi pour les projets à venir.

Le processus qualité a une triple utilité (une par processus) ; il assure au produit un niveau de qualité défini, il garantit que le processus de gestion est réalisé comme prévu et enfin, réflexivement, il assure que le processus qualité s'effectue correctement. Nous nous concentrerons dans ce cours sur la qualité technique du logiciel. La prévision consiste à *définir les exigences* de l'utilisateur, le suivi assure que les techniques utilisées agissent en faveur de ces exigences et l'analyse permet a posteriori d'améliorer la qualité pour les projets ultérieurs.

Un autre processus prend actuellement de l'importance : l'analyse de risque. Ce processus ne sera qu'abordé bien qu'il a toutes les chances de devenir à l'avenir un processus critique dans le génie

logiciel car il synthétise l'ensemble des activités d'un projet en mettant en évidence les plus risquées, donc celles - techniques, de gestion ou de qualité - qui méritent la plus grande attention.

## 4.2. Le niveau de maturité

Le processus de développement théorique présenté en 4.1 n'est que rarement mis en pratique. Le Software Engineering Institute (SEI) a défini un modèle à 5 niveaux pour classer les entreprises en fonction de la façon dont elles mettent en pratique ces processus. Le Capability Maturity Model (CMM) définit les 5 niveaux suivants :

1. **Initial** : Le processus de développement est « ad hoc », et parfois même chaotique. Peu de procédures sont définies et le succès repose sur des efforts individuels.
2. **Répétable** : Une procédure de gestion minimale est définie pour *suivre les coûts, les délais et les fonctions*. Les procédures nécessaires sont en places pour répéter les succès antérieurs à des projets similaires.
3. **Défini** : Les processus de gestion et technique sont documentés, standardisés à un processus standard de l'organisation. Tous les projets utilisent une version *approuvée et adaptée* des processus standards pour développer et maintenir le logiciel.
4. **Géré** : Des mesures détaillées du développement et de qualité sont collectées. Les processus et le produit sont *quantitativement* compris et contrôlés.
5. **Optimisé** : Les processus sont *continûment améliorés* par les analyses des mesures.

En 94, la répartition de 156 entreprises était...71%, 19%, 9.5%, 0, 0.5...En mai 98, pour 700 entreprises elle est de 58%, 24%, 15%, 2.5%, 0.5%...

## 4.3. Cycles de vie du logiciel

### 4.3.1. Phases

Le cycle de vie du logiciel modélise l'enchaînement des différentes activités du processus technique de développement du logiciel. Tous les modèles différencient 3 grandes activités qui vont, selon le modèle, interagir différemment. Ce sont :

1. L'**analyse** : comprendre le problème, les besoins
2. La **conception** : trouver une architecture pour résoudre le problème
3. La **réalisation** : mettre en œuvre, fabriquer

Par exemple, pour fabriquer un modèle mécanique du système solaire il faut :

1. analyser le problème en observant que les planètes suivent des orbites
2. concevoir en inventant un mécanisme qui déplace des sphères sur de telles orbites
3. puis réaliser l'assemblage des sphères, ressorts et engrenages.

Pour un système informatique complexe, on doit :

1. analyser les besoins, comprendre le système (critique et très difficile car l'utilisateur n'a pas souvent une vision très précise de ce qu'il souhaite et puis, expert dans son métier, beaucoup de choses lui semblent évidentes),

2. proposer une solution (éventuellement des variantes) avec des solutions techniques connues, maîtrisées, plus ou moins chères,
3. réaliser la solution retenue en programmant, construisant une base de données, etc.

Une méthode propose une démarche et des notations pour aborder ces problèmes. La démarche décrit quelles étapes élémentaires doivent être suivies, quelles questions doivent être soulevées et à quel moment, quels sont les « objets » qui doivent être mis en évidence, etc. La notation permet de présenter et formaliser des « objets » solution aux informaticiens et aux clients ; il est donc indispensable qu'une notation soit compréhensible par des non-spécialistes... Quelques éléments de méthode seront introduits en §6.5.

### 4.3.2. Waterfall

Représentation élémentaire du cycle de vie, elle ajoute aux trois étapes précédentes une phase de test pour valider la réalisation.

1. Analyse
2. Conception
3. Réalisation
4. Test/Validation

Ce modèle très simple masque un aspect important du processus qui est la décomposition d'une application en sous applications et présente un développement très linéaire bien que des retour-arrière soient prévus lorsque des erreurs ou des manques sont détectés.

Il met toutefois en évidence un principe bien connu : plus une erreur est détectée tardivement plus elle coûte cher puisqu'elle oblige à revenir en arrière. La nature des erreurs est aussi importante : une erreur de réalisation détectée aux tests ne coûte pas trop. Par contre, une erreur d'analyse, détectée aux tests force à reprendre tout le cycle. D'où la criticité de l'analyse et l'importance de bien comprendre les besoins.

### 4.3.3. en V

En reprenant les phases du cycle Waterfall, le cycle en V introduit la notion de décomposition et d'intégration fondamentaux dans les applications de grande taille.

1. Analyse
2. Conception
3. Réalisation
4. Test/Validation
5. Test unitaire (vérification interne : on a bien fait)
6. Intégration
7. Validation (vérification externe : on a fait ce qu'il fallait)

L'aspect linéaire demeure et la distance des deux activités (1 & 7) qui font intervenir le client/utilisateur est très grande. En cas de mauvaise analyse, les risques sont grands de construire, certes bien, quelque chose, mais pas ce qu'il fallait !

#### 4.3.4. en spirale

Ce cycle implique fréquemment le client/utilisateur. Il représente le développement comme une succession de cycles en waterfall, où, à chaque itération le système est analysé, conçu, réalisé et testé un peu plus qu'à l'itération précédente.

1. Analyse
2. Conception
3. Réalisation
4. Test/Validation
5. Analyse
6. Conception
7. Réalisation
8. Test/Validation
9. Analyse
- 10....

Ce type de cycle de vie s'adapte très bien aux développements par prototypes (Rapid Application Development - RAD) et en particulier aux développements dans des environnement objets (comme Smalltalk, java, Eiffel, C++).

Les risques de mauvaises analyses sont réduits par les validations fréquentes du client/utilisateur.

#### 4.4. Conclusion

Les modèles de processus et de maturité présentés sont indicatifs ; ils permettent de dresser des listes de choses à faire, d'avoir des éléments structurants d'organisation, *ils ne sont pas une fin en soi*. Des critiques du modèle CMM mettent en évidence son aspect trop "centré" sur les procédés et ignorants les dimensions humaine et dynamique des organisations.

## 5. Les principes d'organisation

Quelque soit le cycle de développement choisi, les éléments qui suivent doivent trouver leur place tout au long du développement. L'ordre présenté ne reflète pas forcément l'ordre dans lequel il faut aborder ces problèmes ; ils sont d'ailleurs souvent interdépendants.

### 5.1. Les ressources humaines

La réussite d'un projet dépend en grande partie de la qualité des personnes qui travaillent, de leur organisation, de leur communication. Des travaux récents de J. Coplien mettent en évidence des éléments de solution organisationnels à des problèmes du génie logiciel. Ce sont des patrons [Gam] d'organisation (Organisational patterns), c'est-à-dire des recommandations à suivre dans un contexte donné, pour résoudre un problème ou améliorer une situation difficile. On peut citer par exemple :

- ◆ Taille de l'organisation : idéalement une dizaine de personnes,

- ◆ Laisser l'équipe se constituer : des gens qui se connaissent et s'apprécient travaillent mieux ensemble,
- ◆ Respecter le planning : mettez en place des primes ou des compensations,
- ◆ L'architecte logiciel doit programmer : pour garder sa crédibilité
- ◆ Faite tourner la responsabilité : augmente la cohésion, améliore la communication, équilibre les charges<sup>1</sup>

## **5.2. L'estimation**

Une des activités les plus délicates de l'ingénieur consiste à évaluer, estimer un projet. Il s'agit de mesurer, bien souvent avant que le projet ne commence :

- combien de temps il va durer
- quel sera son coût
- combien de personnes devront travailler dessus

La grande difficulté consiste à faire ces mesures alors qu'on ne dispose que de très peu d'information. C'est pourquoi, au fur et à mesure de l'avancement du projet, alors que l'on collecte de l'information, on raffine les estimations.

Il existe 5 grandes familles de techniques d'estimation :

1. l'analogie
2. paramétrique
3. oracle
4. PERT
5. bottom-up

Pour estimer par analogie, il faut conserver les données de tous les projets antérieurs qui sont jugées pertinentes pour le dimensionnement d'un projet afin que, par comparaison, on évalue les caractéristiques du prochain projet. C'est la méthode « intuitive ».

L'estimation paramétrique est une méthode par analogie soutenue par des outils d'analyse statistique. Le modèle le plus connu s'appelle COCOMO (Constructive Cost Model) ; il sera détaillé en cours. Son principe établit une relation qui, à partir du nombre de lignes de code livrées (Source Line Of Code - SLOC), déduit la durée du développement et l'effort à y consacrer. Des variantes prennent en compte divers paramètres comme la compétence des programmeurs, le niveau du langage utilisé, ou la criticité du système à construire. Ce modèle, comme on le voit, repose lui-même sur une estimation du nombre de lignes de code du système. Pour estimer ce nombre de ligne de code, une autre technique a été proposée : le point de fonction (*fonction point*). Elle dérive le nombre de lignes de codes d'un comptage du nombre de fonctions à réaliser dans le système.

La méthode de l'oracle consiste à s'adresser à des experts afin d'obtenir par négociation une valeur de l'estimation consensuelle.

---

<sup>1</sup> Attention, ces quelques lignes ne dispensent pas de la lecture plus complète des patterns...

La méthode PERT repose sur l'une des 3 méthodes précédentes appliquées à la recherche d'une valeur minimale, moyenne et maximale des caractéristiques recherchées. La valeur retenue étant la moyenne pondérée (1-4-1) de ces valeurs.

Comme la méthode PERT, la méthode Bottom-up repose sur des estimations de composants élémentaires qui sont ensuite agrégées (consolidées) jusqu'à avoir une estimation globale du projet.

L'expérience montre que ces techniques doivent être utilisées en complément l'une de l'autre et toujours avec beaucoup de précautions. Quoi qu'il en soit, la qualité - et la rentabilité - d'un projet repose essentiellement sur la capacité à estimer. Le métier d'ingénieur estimateur est crucial dans une entreprise.

### **5.3. La planification**

La planification permet de décrire l'enchaînement des différentes activités d'un projet. Elle s'appuie sur des estimations de durée ou d'effort associés à chacune de ces activités, puis décrit les dépendances entre les activités dans le temps. A chacune des activités sont attribuées des ressources humaines, matérielles ou budgétaires. On peut ainsi avoir une vue globale dans le temps de la répartition :

- des efforts individuels - pour les ressources humaine,
- des dépenses et recettes - pour les ressources budgétaires,
- de l'utilisation des machines, instruments de mesure, etc. - pour les ressources matérielles.

Les techniques de planification seront développées dans le cours de gestion de projet.

Les activités typiques du développement d'un système logiciel sont :

- Analyser : comprendre le problème (modéliser) et exprimer les exigences (qualité)
- Concevoir : construire une architecture du logiciel (interface, stockage, communication)
- Réaliser : programmer
- Tester
- Intégrer : assembler les composants existants à ceux développés
- Valider : soumettre au client
- Former : à l'utilisation, la maintenance du système
- Apprendre : à son équipe de développement des techniques nouvelles

Certaines étapes comme le maquettage ou le prototypage d'application peuvent apparaître comme activité spécifiques ; en fait elles regroupent des phases d'analyse, conception, réalisation, validation de manière plus condensée.

Il est important de noter que chacune de ces activités doit pouvoir être déclarée « terminée » par une décision : en général par la validation d'un document clôturant l'activité (rapport d'analyse, rapport de tests, etc.)

## **5.4. La qualité du logiciel**

### **5.4.1. Définition et mesure de la qualité à atteindre**

La qualité est définie de manière générale par « l'aptitude d'un produit ou d'un service à satisfaire les besoins des utilisateurs ». Cette définition s'applique aux systèmes informatiques. La difficulté principale réside dans la capacité à exprimer ces besoins. On constate trop souvent que l'utilisateur n'est pas satisfait a posteriori ! Certes le cycle de vie en spirale, ou des approches par maquettage réduisent ces risques, mais, il est fondamental de permettre à l'utilisateur (et/ou au client) d'exprimer ces exigences ; ce sur quoi il va juger la qualité du système.

La qualité est un processus qui dépasse la notion de logiciel. On peut l'appliquer à tout processus industriel pour peu qu'on souhaite l'améliorer. Améliorer quoi ? le produit, le service, la façon de développer le produit, les délais, les coûts, etc. Peu importe, ce qui compte c'est la volonté de mesurer, observer, contrôler, apprendre pour améliorer quelque chose. Le CMM (§4.2) est une façon de classer les entreprises en fonction de la volonté qu'elles affichent et des techniques qu'elles utilisent pour cette amélioration.

Les premières études systématique de la qualité des logiciels datent de 1977 [McCall]. Depuis des modèles de qualités se sont développés, mais on retrouve toujours 3 niveaux :

1. Les **facteurs** qualité : expression des exigences (point de vue externe, client)
2. Les **critères** qualité : caractéristiques du produit (point de vue interne, technique)
3. Les **métriques** : ce qui permet de mesurer un critère

Les facteurs de qualité du logiciel sont les suivants :

1. Conformité ; satisfaire aux spécifications - il faut qu'elles existent !
2. Robustesse ; ne pas tomber en panne (tolérance aux pannes, recouvrement d'erreurs, etc.)
3. Efficacité ; optimisation des ressources (cpu, E/S, mémoire, etc.)
4. Sécurité ; surveiller, contrôler, interdire les accès
5. Maniabilité ; minimiser l'effort d'apprentissage de l'utilisation du système
6. Maintenabilité ; minimiser l'effort pour localiser et corriger les fautes
7. Testabilité ; minimiser, automatiser l'effort de test
8. Adaptabilité ; minimiser l'effort d'évolution du système
9. Portabilité ; minimiser l'effort pour changer de plate-forme
10. Réutilisabilité ; optimiser la conception pour faciliter la réutilisation de parties du système
11. Interopérabilité ; garantir l'ouverture du système à d'autres systèmes

Ils doivent être appréciés par le client pour définir les points qu'il juge capitaux ou secondaires. Différentes stratégies de notation de facteurs qualités seront présentées en cours.

Il faut remarquer que ces facteurs ne sont pas indépendants les uns des autres. Ainsi, par exemple, obtenir une grande Sécurité et une grande Interopérabilité est difficile. Un système ouvert est rarement sûr ! C'est au client de définir ses priorités et au développeur de mettre en pratique les techniques qu'il connaît pour assurer le niveau de qualité requis.

Les techniques mises en œuvre couvrent les critères qualité. On y retrouve les « bons » principes de programmation comme la modularité, l'auto-descriptivité (les commentaires), l'indépendance matérielle logiciel, la concision, etc. Mais aussi des « bons » principes d'organisation comme l'utilisation de standards ou la **traçabilité**. Cette dernière est la capacité à suivre dans le temps l'évolution d'un produit et de ses plans et de pouvoir associer les éléments de la solutions aux éléments du problème.

Pour mesurer la qualité du logiciel, des métriques sont associés aux critères eux même rattachés aux facteurs. Ces métriques peuvent caractériser les qualités :

- du produit
- du processus de développement
- du service rendu

Elles peuvent se faire par des mesures objectives (comptages) ou par des enquêtes d'opinion (« pensez-vous que les résultats soient présentés clairement ? - de 0 à 5 »). Les métriques concernant le logiciel sont celles du produit. Quelques exemples :

- nombre de lignes de code, de fonctions, d'opérateur par fonction,... (concision)
- nombre de lignes de commentaires, ... (auto-descriptivité)
- nombre des modules, nombre de liens avec d'autres modules - couplage (modularité)
- nombre de chemins possibles dans une fonction (simplicité)
- nombre de données en entrées, en sortie, fréquence...(simplicité)

### 5.4.2. Processus pour obtenir la qualité

La qualité s'obtient par la mise en place d'activités de contrôle : contrôle techniques pour le logiciel et contrôle de processus pour les aspects gestion et qualité. Ces contrôles consistent souvent en des observations par des tiers des produits (documents, logiciels) et des processus mis en œuvre. Le point important est la décorrélation entre celui qui fait et celui qui contrôle. Pour garantir cela dans de très grands projets, on peut faire appel à des organismes de vérification et de validation indépendants (IV&V).<sup>2</sup>

Les contrôles techniques portent donc sur les documents et les codes source produits. Ils sont relus par une ou plusieurs personnes, individuellement ou lors de réunions dites d'inspection pour mettre en évidence des problèmes de fond comme des contradictions, des omissions, des ambiguïtés ou des ajouts fonctionnels mais aussi des non respects de forme comme le non respect de standard, du bruit, des redondances, etc.

---

<sup>2</sup> Les outils GNU sont ainsi de grande qualité car, fourni avec leur code source, ils sont lus, relus, critiqués et améliorés.

Les contrôles de processus évaluent les procédures de gestion et la démarche technique employées. Ils se font par des observations et des audits qui vérifient l'existence des procédures, leur respect, leur pertinence.

Bien évidemment, la mise en place d'une démarche qualité a un coût qui peut être important. Il y a quelques années, un mot d'ordre courait dans les entreprises : « zéro défaut » ou « qualité totale ». Il faut plutôt prendre cela comme une incitation à la qualité, une volonté de prise de conscience du bien fondé d'un processus qualité que comme un objectif à atteindre... Si la qualité totale pouvait être atteinte elle permettrait de réduire considérablement les coûts de maintenance - sans les supprimer toutefois, car, pour perdurer un logiciel doit s'adapter, évoluer - au prix d'un coût de développement faramineux. L'absence totale de qualité - est-ce envisageable ? - aurait à la fois un coût de développement et un coût de maintenance important. Encore une fois, tout l'art de l'ingénieur va consister à trouver le bon équilibre pour satisfaire le client, assurer un développement à moindre coût tout en n'abusant pas de contrôles...

Pour ce faire, il faut s'appuyer sur les techniques du génie logiciel : modélisation d'une démarche de développement, choix d'une méthode, d'outils, définitions de documents de références, de métriques etc.

L'ensemble des procédures et techniques employés dans une entreprise devraient être regroupées dans un document unique le « **manuel qualité** » ; c'est le rôle des normes ISO 9000 de s'assurer de l'existence et du contenu de ce manuel - hélas, pas de son application !

Le rôle d'un chef de projet, ou d'un ingénieur qualité associé à un projet de développement logiciel consiste à extraire de cet ensemble de procédures et techniques de l'entreprise celles qui sont adaptées au projet et de définir ainsi le « **plan qualité** » du projet. Une fois encore, l'art de l'ingénieur est de trouver le bon équilibre, entre pas assez et trop de procédures.

### **5.5. L'analyse de risque**

L'analyse du risque est encore une pratique réservée aux très gros projets (construction d'usine) et encore assez rarement en informatique. Pourtant, au vu de l'état des lieux (cf. 2.) et 30 ans après la naissance du génie logiciel, les risques sont encore grands ; dépassement de budget plus d'une fois sur deux, et retard près de deux fois sur trois.

L'analyse de risque consiste à améliorer les capacités d'estimation, d'anticipation et de réaction à des « risques » lors du développement. Le cycle de vie de ce processus peut se modéliser par les phases suivantes :

1. Identification du risque
2. Évaluation du risque
3. Réaction au risque
4. Apprentissage

Sur ce schéma plusieurs stratégies peuvent être appliquées. La plus courante est dite Just In Case (JIC) ; elle consiste à stocker (surestimer les coûts, les délais) pour éviter les ennuis éventuels. Son principal défaut est la surévaluation conséquente qui peut repousser le client. Une autre

stratégie émerge, Just In Time. Elle repose sur une grande réactivité et une analyse très fréquente des plannings.

Voici encore un cas où l'ingénieur devra trouver un équilibre entre faire assumer tout le risque par le client (JIC) au risque de le perdre, et assumer lui-même tout le risque (JIT) au prix d'un effort de suivi de projet plus grand.

Les risques sont classés en risques stratégiques (valeur des actions, retour sur investissement, part de marché, etc.) et opérationnels (budget, délais, personnel, technologie, etc.) et peuvent avoir des répercussions évaluées en trois classes :

- I. pertes qui ne perturbent pas l'entreprise (i.e. retard de mise sur le marché de Windows NT)
- II. pertes qui forceront à faire des transactions (emprunts, ventes, etc.)
- III. risques de banqueroute (i.e. the wrong product at the right time or the right product at the wrong time)

Nous nous intéresserons ici qu'aux risques opérationnels, c'est-à-dire, ceux, quotidiens, relatifs au développement d'un projet. Ils peuvent être budgétaires, liés au planning ou aux aspects techniques :

- Budgétaires
  - ◇ compétition avec d'autres projets
  - ◇ sous-estimation du produit par manque de spécification
  - ◇ pas ou peu de références historiques pour l'estimation
- Planning
  - ◇ disponibilité des personnes et des matériels
  - ◇ temps d'apprentissages des langages, techniques et outils nouveaux
- Techniques
  - ◇ introduction d'une nouvelle technologie (essayée par d'autres)
  - ◇ pionnier dans une nouvelle technologie
  - ◇ manque de spécification du produit à faire

Une classification des risques selon trois déterminants permet d'aider à choisir une réponse au risque considéré :

*Manque de contrôle* : événement imprévisible comme une panne du système en phase finale des tests -> attendre ou trouver une alternative

*Manque d'information* : comme une erreur dans un outils utilisé ou la connaissance de la durée nécessaire de test -> collecter de l'information, changer d'outils...

*Manque de temps* : comme la couverture de test ou la connaissance du temps passé à la conception -> gagner du temps ailleurs, retarder, avancer en mode dégradé...

Les facteurs de risques spécifiques au logiciel sont résumés dans le tableau suivant. Chacun des facteurs de risque peut être décomposé en facteurs plus élémentaires comme pour les outils (la pratique des outils, le degré d'automatisation pour la conception, pour le test, la stabilité de l'ensemble compilateur/éditeur de lien/débogueur, la disponibilité des outils, etc.).

Le facteur à plus haut risque est le Personnel, ce qui était connu depuis longtemps. Boehm disait « The biggest influence on most projects is the quality of people who work on it » ; il concluait en incitant à acheter, louer les meilleurs...ou à faire « avec » !

Facteur de risque	Risques liés à	Implication		
		Technique	Budget	Planning
Organisation	la maturité de l'organisation (CMM), communication, management	Faible	Haute	Haute
Estimation	la précision des estimations (budget, planning, ressources)	Faible	Haute	Haute
Monitoring	la capacité à identifier les problèmes	Moyenne	Haute	Haute
Méthode	la méthode de développement utilisée	Moyenne	Haute	Haute
Outils	aux outils utilisés	Moyenne	Moyenne	Moyenne
Culture risque	la prise en compte des risques dans le processus de décision	Haute	Moyenne	Moyenne
Usage	l'usage du produit logiciel une fois livré	Haute	Faible	Faible
Correction	l'adéquation du logiciel une fois livré	Haute	Faible	Faible
Fiabilité	la fiabilité du logiciel livré	Haute	Faible	Faible
Personnel	l'utilisation par le personnel des méthodes, outils	Haute	Haute	Haute

L'analyse des risques est aujourd'hui l'une des activités les plus en amont du développement logiciel. De ce fait elle s'appuie sur toutes les activités précédemment décrites (planification, estimation, qualité, développement). C'est sans doute l'une des activités managériales essentielles des années à venir.

### 5.6. La gestion de configuration

Ce processus, souvent négligé - à tort - lors des projets de courtes de durée comme ceux réalisés dans une école, prend une dimension d'autant plus critique que le projet est de grande taille et de longue durée. Il consiste à :

- identifier ce qui compose le développement
  - les modules matériels et logiciels
  - tous les documents : spécifications, conception (toutes les variantes), contrats, schémas, définitions d'interface, procédures de tests, manuels qualité, rapports d'anomalie, lettres échangées avec le client, etc.
- suivre l'évolution de ces composants
  - versions
  - états (ex : demande d'anomalie : enregistrée, transmise, trouvée, corrigée, intégrée à la nouvelle version)
- assurer la cohérence entre les composants ; une nouvelle version d'un module logiciel doit savoir à quelle version des documents techniques (spécification, schémas, interfaces, etc.) et des documents utilisateur il fait référence...

La plupart de ces informations existent sur un support électronique, et donc, des outils informatiques de gestion de configuration ont été développés pour faciliter ce processus. Les plus rudimentaires se contentent de gérer les versions d'un fichier, de revenir à une ancienne version, de fusionner deux versions, etc. Sur Unix, nous vous invitons à consulter le **manuel** de rcs ou de sccs...

## 6. Les techniques

### 6.1. La modélisation du produit

L'un des points techniques délicat de la production de logiciel est la description du produit que l'on souhaite construire. Pour ce faire, on s'est longtemps appuyé sur des descriptions spécifiques à des méthodes (cf. §6.2) qui décrivaient en plus la démarche de construction de cette description. En 1997, une petite révolution a eu lieu, car un organisme de standardisation constitué du plus gros groupe d'utilisateurs d'informatique du monde (l'Object Management Group - OMG) a choisi un langage de description de systèmes informatiques qui a de bonnes chances de s'imposer comme LE standard. Il s'agit d'UML - Unify Method Language. 3 des plus grands noms en matière de méthode y ont collaboré : Rumbaugh, l'auteur de OMT, Grady Booch, auteur d'une des premières méthodes orientées objet, et Ivar Jacobson, qui a popularisé les « use cases ».

La description d'une application repose sur 3 axes :

1. **Structurel**
2. **Fonctionnel**
3. **Temporel**

L'axe structurel permet de décrire ce qui compose le système. La tendance actuelle est de décrire un système par un ensemble **d'objets**. Chaque objet peut être héritier d'un autre objet ou composé d'autres objets et réaliser des opérations qui lui sont spécifiques, *indépendamment des fonctions du système auquel il participe*. Pour décrire cette dimension du système, UML s'appuie sur des extensions des schémas Entité-Association exploités depuis longtemps dans les systèmes d'informations (base de données) où la dimension structurelle est prépondérante.

Les éléments constitutifs de ces diagrammes sont des « objets » reliés par des **relations** d'héritage ou de composition. <Exemple en cours>.

L'axe fonctionnel permet de décrire ce que fait le système. Les fonctions sont des entités qui font passer le système d'un état dans un autre suite à une sollicitation venant du monde réel ou du système lui-même. Pour décrire cette dimension UML s'appuie sur des diagrammes de **scénarios** (« use cases ») qui représentent l'enchaînement (les flux) des messages (données) entre les acteurs du système. D'autres notations comme les diagrammes de flux de la méthode SADT ou des workflows sont aussi utilisés.

Les éléments constitutifs de ces diagrammes sont des « objets » reliés par **flux** d'informations. <Exemple en cours>.

L'axe temporel permet de décrire quand les opérations sont effectuées. Les fonctions mises en évidence précédemment sont décrites par un enchaînement d'opérations. Pour décrire cet axe, UML s'appuie sur des diagrammes de State Charts qui sont une extension des Automates à Etats Finis. Les state charts permettent de décomposer des automates, de mettre en parallèle des automates et de synchroniser des transitions.

Les éléments constitutifs de ces diagrammes sont des états reliés par des transitions qui sont les opérations des objets. <Exemple en cours>.

En fait, la tendance actuelle est de réduire l'importance de l'axe fonctionnel au profit de l'axe temporel. Les fonctions restent toutefois des abstractions d'organisation des événements tout à fait naturelles.

On complète cette description par des outils d'organisation et de compréhension. La taille des systèmes étant généralement très grande, la description doit pouvoir être décomposée/organisée en modules et sous-modules qui restreignent la description Structurelle-Temporelle-Fonctionnelle à un sous-ensemble du système.

La délimitation du système est aussi un problème délicat qui fait intervenir son interface avec le monde réel (l'utilisateur, des capteurs, des actionneurs, etc.). Pour décrire et comprendre cette interface (cf. §6.3) on utilise des « use cases » qui sont des schémas concrétisant des scénarios d'utilisation du système. Ils constituent un début de la description fonctionnelle du système.

La démarche utilisée pour construire cette description, la méthode, est liée à l'axe qui est privilégié dans la compréhension du système.

## **6.2. Méthodes**

Le terme de méthode recouvre beaucoup de choses. Ici, nous l'entendrons comme un ensemble de procédures (processus) qui mène d'une idée que l'on se fait d'un système logiciel à sa modélisation comme présenté en §6.1 même si le modèle résultant diffère parfois (en particulier avec l'approche formelle).

Historiquement il a 4 grandes familles de méthodes :

1. Orientée donnée
2. Orientée traitement
3. Orienté dynamique
4. Formelles

L'approche orientée donnée est issue des systèmes d'informations (comptabilité, gestion de personnels, de stocks, etc.) où la quantité d'information à traiter est considérable et les traitements à faire relativement simples. Elle met l'accent sur la construction du modèle de données : l'axe structurel. En France, le prototype de ces méthode est Merise. Les premières étapes consistent donc à chercher les données pertinentes et à les relier entre elles dans un schéma Entité-Association. Les traitements sont définis et élaborés dans une seconde phase d'analyse. La cible de ces applications est un système construit sur une (ou des) bases de données qui assurent le stockage des informations et des programmes qui assurent les fonctions et la conservation de la

cohérence des données. Le passage du schéma abstrait Entité-Association à l'implantation dans une base de données est du domaine de la conception. De nombreux choix et compromis sont à faire pour satisfaire aux exigences qualités (efficacité, sécurité, etc.) ; cet aspect sera étudié dans la partie « base de données » de ce bloc.

L'approche orientée traitement est issue des systèmes dits temps-réel (systèmes de contrôle, de commande, automatismes, etc.) dans lesquels la multiplicité des fonctions et leur inter-relation sont telles que cette approche met l'accent, dès le début de l'analyse, sur cette dimension. Le prototype de ces approches est SADT. Les modules, les fonctions sont mises en évidence ainsi que les flux d'information (et de contrôle) entre eux. Ces modules sont ensuite transformés en entités (programmes) exécutables. La cible de ces applications est souvent un système composé de nombreux éléments de calculs indépendant réalisant une petite partie de l'ensemble des calculs et des fonctions de contrôle. Le parallélisme résultant de l'exécution de l'ensemble de ces fonctions pose des problèmes tout à fait nouveaux (par rapport à de la programmation séquentielle). L'étude des difficultés rencontrées par la mise en parallèle de fonctions et la présentations de quelques unes des solutions constitue la partie « parallélisme » de ce bloc.

L'approche orientée dynamique est, en fait, une première synthèse des deux approches précédentes. La difficulté principale d'une méthode c'est de démarrer - le syndrome de la feuille blanche. Comment trouver les « bonnes » fonctions ou entités sans avoir une idée a priori de la solution (que souvent les experts croient avoir). L'idée, ici, est de se concentrer sur les événements à l'interface du systèmes. Ils sont simples car instantanés et bien localisés, à l'interface. Ordonnés, les événements décrivent l'axe temporel du système. De ces événements on déduit les fonctions et les objets du système. Le prototype de cette approche est Jackson System Development (JSD).

L'approche formelle est plus confidentielle et réservée aux institutions académiques, même si, IBM a développé il y a plusieurs années déjà une partie de son système d'exploitation avec une méthode formelle (VDM, Vienna Development Method). La démarche se rapproche d'une analyse mathématique du système, où chaque fonction est décrite par une spécification formelle (mathématique) puis, pas à pas, transformée en un objet informatique : un programme. La méthode la plus en vogue actuellement s'appelle Z. L'intérêt principal de cette approche est la rigueur mathématique, qui permet de démontrer la correction des programmes ainsi spécifiés. Cependant compte tenu de la difficulté des démonstrations, ces approches ne sont utilisées dans l'industrie, car elles sont tout de même employées, que pour des sous-ensemble critiques de systèmes.

La complexité des systèmes grandissant, la dichotomie système d'information - système temps réel disparaît ; les systèmes d'informations deviennent distribués grâce aux réseaux et les systèmes temps-réel font face à tant d'information qu'il devient nécessaire de penser à leur persistance dans des bases de données. De plus, la généralisation du paradigme de programmation objet a fortement influencé les méthodes de sorte qu'aujourd'hui une synthèse est en cours entre les aspects donnée-traitement-dynamique et formel (comme dans la méthode Syntropy[M-Syn]).

### 6.3. **Analyse des besoins et analyse de domaine**

L'analyse des besoins est une phase extrêmement importante. Elle doit permettre au concepteur de comprendre ce que souhaite son client, tant d'un point de vue qualité, en faisant exprimer au client ses exigences, que d'un point de vue fonctionnel. C'est ce dernier point que ce chapitre aborde.

Plusieurs stratégies existent, en fonction du cycle de vie retenu :

1. tenter une fois pour toute de spécifier les besoins (cycle en chute d'eau et en V), avec le risque de mal se comprendre. Cette stratégie est toutefois bien adaptée lorsque le client a lui-même bien analysé ce qu'il voulait et que le contrat est clair.
2. élaborer au fur et à mesure de la construction du système en découvrant les besoins à chaque itération du cycle (en spirale) ; avec le risque d'avoir un client en veut toujours plus<sup>3</sup>.

L'identification des fonctionnalités et la délimitation du système (choisir ce qui va être automatisé de ce qui va rester hors du domaine du système) se réalise de plus en plus à l'aide de scénarios d'utilisation (« use cases »). Il faut noter que pour comprendre un système complexe on est toujours amené à avoir une vision (compréhension) plus vaste que le système lui-même. Et donc, pour analyser un système, on cherche maintenant aussi à comprendre son environnement. On effectue ce qu'on appelle une **analyse de domaine** (« domaine analysis »).

Cette analyse de domaine utilise les mêmes techniques que l'analyse des besoins, sans se restreindre, dans un premier temps aux besoins. La restriction aux besoins se fait en délimitant les frontières du système ; en ne gardant que les fonctions dont on a effectivement besoin.

La construction de scénarios se fait par l'interview des clients et futurs utilisateurs. On élabore, guidé par leurs informations, les différentes façons dont ils envisagent que le système sera utilisé. Ces scénarios décrivent des acteurs qui interagissent par des messages. Toute ressemblance avec les concepts de la programmation par objets n'est pas du tout fortuite !

Ces scénarios servent ensuite de base à la conception du système, ainsi qu'au procédures de validation du système ; le système construit devra, au moins, assurer les fonctions mises en évidence par l'analyse des besoins.

### 6.4. **Formes de conceptions (design patterns).**

Après quelques dizaines d'années de construction de systèmes informatiques, on s'est aperçu récemment que certaines formes (architecturales) se reproduisaient fréquemment. Depuis quelques années, une communauté d'informaticiens cherche à *identifier* et à *nommer* ces formes connues sous le nom de « design pattern ». Ce simple fait permet de mieux les reconnaître dans un système et surtout de caractériser leurs propriétés (qualités, type de problème résolu, et défaut) afin de les employer pour la construction des nouveaux systèmes.

---

<sup>3</sup> On voit ici, que la nature financière du contrat intervient aussi dans le choix du cycle de vie. Un contrat à prix fixe est plus adapté à un cycle en V et un contrat en régie (prix en fonction du temps passé) plus adapté à la flexibilité du cycle en spirale (les risques sont partagés).

Une forme de conception est définie par :

- son nom
- le problème que la forme essaie de résoudre
- le contexte dans lequel on le rencontre
- la solution proposée
- les exemples d'utilisation
- le contexte résultant (s'il reste d'autres problèmes à résoudre)
- les autres formes de conceptions liées
- des justifications (origine, pourquoi ça marche et on l'utilise)

Gamma[Gamma] a classifié les formes de conception et décrit complètement 15 formes ; les principales sont :

- Les formes de création ; comment créer des objets
  - ◇ Fabrique abstraite ; liée aux classes abstraites dans un langage objet, elles permettent la définition et la création d'objets indépendamment du système, du support d'exécution, de l'implantation réelle.
  - ◇ Prototype ; un objet est créé par copie d'un objet prototype
  - ◇ Singleton ; lorsque le système doit garantir l'unicité d'un objet
- Les formes structurelles
  - ◇ Adapteur ; un objet sert d'interface (traducteur) entre un objet et un autre
  - ◇ Composite ; structure d'objet arborescente ou en graphe
  - ◇ Décorateur ; encapsulation d'un objet dans un autre pour étendre ses services
  - ◇ Façade ; un objet qui masque la complexité d'autres objets en n'en montrant que certains services
- Les formes comportementales
  - ◇ Chaîne de responsabilité ; faire passer une requête d'objet en objet. S'utilise facilement avec la forme Composite
  - ◇ Commande ; regroupe en un objet un ensemble de requêtes pour simplifier leur paramétrage, leur annulation, leur réexécution, etc.
  - ◇ Itérateur ; permet un accès séquentiel aux éléments d'un agrégat d'objet sans avoir connaissance de la représentation interne de l'agrégat.
  - ◇ Observateur ; un objet se dit intéressé par les changements d'état d'un autre objet

La connaissance des formes de conception ne résout pas les problèmes, mais elle donne de bons (car déjà utilisés) éléments de solution. Ce sont des heuristiques de conception.

### **6.5. Composants réutilisables**

Ne pas réinventer la roue à chaque fois, ne pas tester des programmes que d'autres ont déjà réalisés et testés, en bref améliorer la réutilisation du code produit est aussi un des objectifs du génie logiciel. Il y a déjà longtemps que des composants réutilisables existent, mais hélas, souvent avec de fortes restrictions (système, langage, fonctionnalité).

Voici quelques exemples composants réutilisables :

- ◇ Les bibliothèques (library) système (unix, windows) ; on parle aussi d'API (Application Programming Interface)
- ◇ Les bibliothèques spécialisées, comme pour les mathématiques en Fortran
- ◇ Les bibliothèques de composants d'interface homme-machine (IHM)
- ◇ Des protocoles de communications
- ◇ La persistance de donnée avec des bases de données (SGBD)
- ◇ Les noyaux de système temps-réel pour gérer de multiples activités en parallèle

L'introduction des langages à objets incite les développeurs à concevoir et à réaliser pour la réutilisabilité, mais on oublie parfois le surcoût d'investissement qui y est lié. Cette technologie objet pourtant permet de produire des composants réutilisables de taille variée : de la fonction au système complet (comme dans Smalltalk). En particulier, de nombreux efforts sont faits dans la réalisation de « frameworks » (squelette d'application ?) qui permettent de définir des sous-systèmes très extensibles et adaptables.

Un framework est un ensemble cohérent de classes qui, sans constituer un réel système, met en place l'architecture globale et les abstractions utiles de telle sorte que, pour finir le système, il suffise de remplir les trous - en concrétisant les classes par exemple.

## **6.6. Le test du logiciel**

Ce chapitre traite exclusivement de la vérification du logiciel. L'aspect validation, i.e. l'adéquation au besoin, la satisfaction du client/utilisateur sont abordés lors des chapitres cycle de vie (§4.3) et qualité (§5.4). Le test du logiciel concerne donc la vérification (interne) du point de vue du développeur du système.

L'objectif du test est de mettre en évidence des erreurs, mais la réussite des tests n'est en aucun cas un gage d'absence d'erreurs ! (Dijkstra « Les tests montrent la présence d'erreurs et non l'absence d'erreurs ».) Pour compléter une stratégie de test, ou pour des applications critiques, on pourra formaliser, pour prouver, un programme. Mais, quoi qu'il en soit, comme le fait remarquer R. Lipton « La preuve est un processus social de confiance », ce qui importe est la confiance accordée au système, résultante de toutes ces démarches.

Dans ce cadre, on distingue trois grandes techniques de test :

1. Boîte Noire
2. Boîte Blanche
3. Statistique

Le test boîte noire (TBN) consiste à tester un composant en ne connaissant de lui que son interface. Le jeu de test utilisé est déduit des spécifications du composant. La recherche du jeu de test le plus efficace (au sens où l'on a le plus de chance possible de détecter des erreurs) est un problème difficile. Des heuristiques existent comme le test par classe d'équivalence ou le test au limite. Le test par classe d'équivalence consiste à regrouper les valeurs d'entrée qui devraient produire le même type de résultat ; le test se fait alors avec un représentant de la classe

d'équivalence pris au hasard. Le test au limite complète cette démarche en préférant choisir un représentant de la classe à ses limites (quand cela à un sens).

Ces tests sont adaptés pour vérifier la robustesse, les interfaces, les performances et la non-régression. La non-régression est la vérification que les correction d'erreurs précédentes n'ont pas affectées les fonctions qui avaient été validées.

Le test boîte blanche (TBB) consiste à tester un composant en prenant en compte sa mise en œuvre. Il permet de vérifier si toutes les branches d'exécutions possibles ont été vérifiées. On parle de couverture de test pour mesurer le rapport entre les chemins d'exécutions testés et les chemins d'exécution possible. Il s'utilise pour la vérification (interne) des composants.

Le test statistique (TS) consiste, comme le TBN à tester un composant au travers de son interface mais avec des jeux de tests aléatoires. Ces tests ne sont possibles que si une loi de répartition des données en entrée existe et est connue. Il permet grâce à des modèles de fiabilité de déterminer le temps moyen entre deux pannes, le nombre de pannes observées pour une durée donnée, etc.

Le tableau suivant résume, en fonction de la phase du cycle de vie, ce qui est testé, le but du test et une stratégie possible de test.

Opération	Objets	Buts	Entrées	Technique
test unitaire	algorithmes	correction	spécification des modules	TBN
		variables	code source programme	TBB
test d'intégration	lien entre modules et interfaces	correction	décomposition structurelle	TBN
			code source résultat des tests unitaires	TBB
test du système	fonctions externes	fonctionnalités	définition du système	TBN
		fiabilité données externes	système testé performances	TS
test de recette	fonctions externes	fonctionnalités	CdC	TBN
	données externes	fiabilité performance	documentation utilisateur système	TS

Les techniques de test demandent que des environnement de test soient créés, donc de nouveaux programmes qui vont alimenter automatiquement les composants testés en données ou, au contraire, qui vont simuler des composants non encore existants mais nécessaires pour le composant testé. Ces éléments s'appellent respectivement des pilotes (driver) et des talons (stub).

Compte tenu de la difficulté des procédures de tests, de leur répétitivité et de la lourdeur des environnements nécessaires, des outils sont développés en association avec des environnements de développement pour automatiser cette phase.

## 7. Outils

Gestion de projet	Base de composants réutilisables
Estimation	Compilateur
Outils de communication (workflow)	Générateur de tests
Analyse de risque	Générateur de documentation
Modélisation	Prouveur
(Environnement de développement)	Simulateur
Générateur d'interface	Qualimétrie
Gestion de configuration (outils de gestion de version)	Suivi de rapport d'erreur

## 8. Conclusion

L'art de la production de logiciel est difficile. La nature même du produit rend son élaboration délicate. La complexité et la taille des systèmes développés mais aussi celles des outils utilisés pour développer ces systèmes sont telles qu'il est indispensable de travailler en équipe, de prévoir, d'anticiper. Tout cela dans un contexte où la technologie évolue extrêmement rapidement.

Dans [MJ-I], Mickaël Jackson analyse très finement l'industrie du logiciel, et fait l'hypothèse que le génie logiciel ne progresse pas aussi vite que prévu parce qu'on recherche, naïvement, des techniques applicables à tous les problèmes. Il prône donc une spécialisation des technologies afin, comme ce fut le cas par exemple pour le traitement des grammaires et des langages avec les théories de la compilation, de trouver les théories, les outils, les concepts adaptés à des classes restreintes de problème.

J'espère vous avoir fait prendre conscience de l'ampleur de la tâche et de l'impérieuse nécessité de comprendre que l'informatique ne se réduit pas à la programmation. Comme ce cours tente de le montrer, les activités en passe de devenir dominantes dans la production de logiciel sont liées à l'organisation (le management), l'architecture et la réalisation de composants réutilisables comme les bases de données ou les noyaux temps-réel...

Rappelez-vous que « Les problèmes difficiles aujourd'hui ne sont pas techniques mais organisationnels »...

## Bibliographie

- [MJ-I] Michael Jackson, *Problems, Methods and Specialisation*  
<ftp://st.cs.uiuc.edu/pub/patterns/papers/problem-frames.ps>  
Un point de vue très pertinent sur l'histoire du génie logiciel, ses aspirations, ses erreurs, et quelques pistes de solution.
- [Ency] John J. Marciniak, *Encyclopædia of Software Engineering*, 2 volumes, John Wiley & sons, 1994  
Une encyclopédie...
- [Projet] Cyrille Chartier-Kastler, *Précis de conduite de projet informatique*, Les éditions d'organisation, 1995
- [CMM] James Herbsled et al., *Software Quality and the Capability Maturity Model*, CACM, Juin 1997, 40(6), pp-30-45.  
L'état du CMM est maintenue par Software Engineering Measurement and Analysis (SEMA) à l'adresse <http://www.sei.cmu.edu/activities/sema/profile.html>  
Critiques [http://www.stlabs.com/testnet/docs/CMM\\_API.htm](http://www.stlabs.com/testnet/docs/CMM_API.htm)
- [Estim] JB Dreger, *Function Point Analysis*, Prentice Hall, Inc, Englewood Cliffs, NJ, 1989
- [Qual] Thomas Forse, *Qualimétrie des systèmes complexes, mesures de la qualité du logiciel*, Les éditions d'organisation  
Un livre très complet sur la qualimétrie des logiciels. Plusieurs dizaines de métriques sont détaillées et intégré à un système de mesure globale de la qualité.
- [Risk] Dale W. Karolak, *Software Engineering Risk Management*, IEEE Computer Society Press, 1995
- [Test1] R. Hamlet, *Test du logiciel et confiance*, Génie logiciel et systèmes experts, 30, mars 1993
- [Test2] G. Myers, *The Art of Program Testing*, John Wiley & sons, 1979
- [Bugs] Des erreurs remarquables <http://www.cnet.com/Content/Features/Dlife/Bugs/ss05.html>
- [Pub] JM. Jezequel, *Object-Oriented Software Engineering with Eiffel*, Addison-Wesley, 1996
- [Gam.] Gamma et all, *Design Patterns*, 1996  
Une bible pour les architectes de logiciels. On trouve aussi un site dédié avec cours, catalogue, etc. : <http://hillside.net/patterns/patterns.html> ne pas oublier les patterns d'organisation : <http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns>
- [M-Syn] S. Cook et J. Daniels, *Designing Object Systems, object-oriented modelling with Syntropy*, Prentice Hall, 1994
- [M-JSD] M. Jackson, *System Development*, Prentice Hall, 1983