Prolégomènes à une critique du génie logiciel[1]

Partie IV : Exigences

PAR IVAN MAFFEZZINI, LOUIS MARTIN & ALICE PREMIANA

*Ma main tirait avec force le serpent, tirait, tirait - inutilement ! elle ne réussissait pas à arracher le serpent de la bouche [*du jeune berger*]. Alors un cri m'échappa : « Mors ! Mors ! Détache-lui la tête. Mors » [...] Le berger mordit comme le conseillait mon cri ; et il mordit bien ! Il cracha la tête du serpent loin de lui et sauta sur ses pieds.*

F. Nietzsche*, Ainsi parla Zarathoustra.*

Dans un développement scientifique mûr, les préjugés et les résistances semblent être la règle plutôt que l'exception. Non seulement : dans des circonstances normales ils caractérisent la recherche la meilleure et la plus créative, mais aussi la simple routine.

T. S. Kuhn*, La fonction du dogme dans la recherche scientifique.*

Le signe est type, pas occurrence.

U. Eco*, Sémiotique et philosophie du langage.*

Résumé : Cet article est le dernier d'une série de quatre où nous avons essayé de montrer la nécessité d'élaguer le *génie logiciel* (GL) afin qu'il puisse se transformer en une branche à part entière de l'ingénierie. Dans le premier article, nous avons « arbitrairement » fixé les principes qui soutiennent notre démarche. Dans le deuxième, nous avons mis en évidence certaines lacunes dans le domaine de la qualité et des mesures logicielles. Dans le troisième, nous avons employé les interfaces comme lentilles pour grossir certains aspects du génie logiciel. Dans cet article, nous considérerons la problématique des exigences en nous appuyant sur le chapitre de SWEBOK les concernant. Une ébauche d'une approche alternative aux exigences fondée sur la stabilité est proposée. La conclusion résume les points essentiels traités dans les quatre articles. La bibliographie est précédée par quatre encadrés qui abordent des particularités de thèmes traités dans le corps de l'article.

Mots clés : génie logiciel, exigences, description en langage naturel, méthodes agiles

1. PRÉAMBULE

Comme nous l'écrivions dans le premier article de la série [1], les quatre articles avaient comme but de montrer que « *le génie logiciel (GL) ne peut aspirer à devenir l'un des centres - robuste - des activités d'automatisation que si les branches hypertrophiées par les vicissitudes historiques et les conflits économiques sont élaguées* » et nous espérons avoir au moins en partie atteint cet objectif. Ce qui est par contre certain, c'est que nous n'avons pas respecté le plan proposé : nous n'avions pas prévu de parler de mesures logicielles mais nous l'avons fait dans le deuxième article où nous n'avons pas pu isoler la qualité des mesures ; nous avons mis en conclusion l'article sur les exigences qui aurait dû être le troisième de la série.

Que les « exigences » concluent nos considérations n'est nullement anodin, car ce sont surtout les exigences qui, à notre avis, troublent le GL. Et si, dans la version 2004 de SWEBOK [2], un premier pas a été effectué en laissant tomber les interfaces personne-machine, nous croyons que, sans un coup de hache vigoureux qui coupe nettement l'énorme branche des exigences, l'arbre du GL continuera à donner des fruits fort âpres.

À moins d'indication contraire, lorsque nous emploierons « on », nous ferons référence au GL tel qu'il est considéré par la majorité des praticiens et des théoriciens qui s'intéressent à lui et que SWEBOK synthétise de façon si remarquable. Nos critiques des idées présentées dans SWEBOK ne seront donc pas des critiques du texte lui-même mais de la conception des exigences qui domine actuellement dans le GL.

Même si les liens avec les autres trois articles sont assez importants nous avons essayé de rendre celui-ci le plus autonome possible. Pour ce faire nous avons repris dans l'introduction la figure qui est à la base de nos considérations.

2. INTRODUCTION

2.1 CYCLE D'AUTOMATISATION

La figure suivante, tirée de [1], présente le cycle que nous avons mis à la base de l'automatisation définie comme « *emploi de machines pour transformer des données reçues de l'extérieur et exécuter des actions autonomes ayant un impact sur le monde* ». La définition d'automatisation a été introduite pour souligner que le logiciel n'est que l'un des moyens de l'automatisation ; un simple moyen même si, à ce stade-ci de l'évolution de la technique, il est sans doute le moyen le plus important.

Nous employons le terme machine selon la définition donnée en [1] : « *Un artefact humain composé de parties nécessaires à la réalisation des fonctions qui ont présidé à sa construction. Une partie d'une machine peut être une machine* ». Un programme informatique est donc une machine.

Nous ne sommes concernés ici que par les activités et les produits qui permettent le passage du domaine à la description en langue naturelle (DLN). Passage fondamental et difficile, s'il en est. Les difficultés de ce passage sont dues, principalement, au fait que l'on doit produire un document en partant d'intrants flous et qui proviennent de plusieurs acteurs. Parler d'intrants, c'est déjà une simplification artificielle, il faudrait sans doute parler d'une nébuleuse qui enveloppe les acteurs, même ceux qui ont la prétention d'être à l'extérieur et d'analyser le

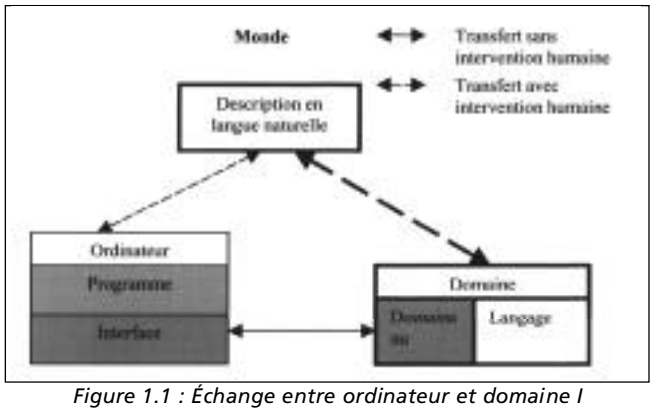

Figure 1.1 : Échange entre ordinateur et domaine I

domaine objectivement. Et si l'on considère que la métaphore du problème est apte à caractériser l'automatisation, il faudra ajouter que, souvent, le problème le plus ardu à résoudre est celui de découvrir le problème.

Dans le syntagme *Description en Langue Naturelle*, ce qui est important, c'est ce « naturelle » qui s'oppose à « formelle » et indique qu'il ne contient pas de propositions (théorèmes) démontrables. Cette opposition est très importante, mais elle n'implique pas qu'une description en langue naturelle et une spécification formelle ne puissent cohabiter dans un projet, avec des fonctions complémentaires. Ce qui est certain, c'est qu'il est impossible de spécifier un programme en langue formelle sans avoir une description en langue naturelle comme intrant[2].

« Naturelle » n'implique surtout pas que la description ne puisse pas contenir des diagrammes (des diagrammes UML, par exemple) ou des prototypes exécutables des interfaces personne-machine.

Ce passage du domaine à la DLN, d'un certain point de vue, ajoute quelque chose par rapport à ce que l'on appelle activités des exigences et, par ailleurs, enlève quelque chose.

- Ajoute, parce qu'il intègre aussi ce que l'on appelle souvent description des besoins (et la partie organisationnelle de l'étude de faisabilité).
- Enlève, parce que nous ne considérons ni les spécifications formelles éventuelles qui, dans notre approche, font partie du passage de la DLN à l'ordinateur ni l'allocation des exigences aux composants.

Parmi toutes les activités nécessaires pour automatiser une partie d'un domaine, celles qui concernent le passage vers la DLN sont celles qui nécessitent le plus d'intervention humaine en tant qu'intervention pilotée par la compréhension, la créativité et la capacité d'adaptation. Et c'est bien la « compréhension », la « créativité » et la capacité d'« adaptation » à pratiquement

n'importe quelle situation qui nous fait dire que ce dont on a besoin pour ces activités, c'est ce qui est propre à l'humain, indépendamment de toute spécialisation : sa capacité langagière. Pour éviter tout malentendu, nous ajoutons que ce n'est pas en introduisant des experts des exigences ou des cognitivistes que le GL fera des progrès dans sa tentative de devenir une vraie branche du génie. À notre avis, la seule spécialisation nécessaire dans cette phase qui est celle liée au domaine.

2.1 Exigences : définitions

Notre point de départ terminologique est la définition de Requirements d'IEEE [4], parce qu'elle est sans doute la définition la plus universellement acceptée : *1) une condition ou une capacité dont un utilisateur a besoin pour résoudre un problème ou pour atteindre un objectif. 2) Une condition ou une capacité que doit posséder un système ou une composante d'un système pour satisfaire un contrat, une norme, une spécification ou un autre document officiellement imposé. 3) La représentation dans un document des conditions ou des capacités comme en 1) et 2).*

Dans la suite, nous écrirons exigences de type 1 pour indiquer les exigences définies par l'acception 1) et exigences de type 2 pour la deuxième acception.

Software Requirements, est traduit en français en considérant logiciel comme épithète d'exigences (*Exigences logicielles*) ou en le considérant comme un complément indirect (*Exigences du logiciel*). L'adjectif dans un cas et le complément indirect dans l'autre ont une fonction déterminative, c'est-à-dire qu'ils restreignent le sens du mot « exigence ». Quand on écrit *exigences logicielles* (ou *exigences du logiciel*), parmi toutes les exigences énoncées, on ne considère que celles qui concernent le logiciel. Mais ce syntagme si simple et si commun recèle une ambiguïté qui se reflète immédiatement dans la difficulté de classification et qui crée un certain fouillis terminologique lorsqu'en GL on parle d'exigences. Ceci est dû au fait que l'on peut considérer la détermination de deux manières fort différentes :

a) *Exigences que l'on impose au logiciel.* Selon cette détermination, il s'agit de considérer parmi l'ensemble des exigences de type 1, celles qui concernent le composant logiciel (exigences de type 2). Donc le déterminatif a la fonction de faire basculer de exigences de type 1 à exigences de type 2. Ce basculement n'est pas sans conséquences, car il demande une définition des interfaces et, éventuellement, une organisation en composants.

b) *Exigences que le logiciel impose.* Cette détermination porte à considérer les exigences que le logiciel qua logiciel impose au matériel ou à d'autres logiciels. Il ne s'agit donc d'aucune des deux acceptions d'IEEE mais plutôt d'un glissement vers la conception système.

Il est intéressant de considérer que [4], tout étant un dictionnaire du logiciel, ne définit pas *Software Requirements* mais seulement *Requirements* ; il en est sans doute ainsi parce que les exigences sont indépendantes du type de la machine qui est censée les satisfaire.

En 1993, Davis, pour montrer les inconsistances terminologiques entourant les exigences, construisit un tableau avec les termes employés par douze auteurs [5]. Aujourd'hui, si l'on refaisait l'exercice il faudrait ajouter bien des auteurs, car les inconsistances n'ont fait qu'augmenter. Ces inconsistances rendent souvent les échanges entre ingénieurs du logiciel complexes et inefficaces. Cela, à notre avis, n'est pas dû aux seules difficultés propres à toute classification ni au désir qu'ont les différents auteurs d'imposer leur marque de commerce. Il y a des difficultés réelles, car on s'efforce de subsumer sous un seul terme des éléments fort hétéroclites. Le passage d'exigences de type 1 à exigences de type 2 n'est pas innocent par rapport à ces difficultés. En fonction du degré de liberté que nous donnons aux besoins par rapport aux machines ou à ses composants, nous avons des documents fort différents et des activités agencées différemment. L'objection « mais, ce qui compte, c'est le résultat final » n'est pas recevable, car, lorsque la maintenabilité est importante - et nous croyons qu'elle l'est toujours quand il s'agit de GL - ce sont les possibles transformations ultérieures qui sont au centre et pas tellement le produit « à sa première sortie ».

Notre travail de déconstruction des exigences ne s'attaquera pas à la définition mais à ce qui en découle, car ce n'est que dans l'instanciation de la définition dans des documents plus détaillés que l'on peut voir les dangers qui se cachent derrière une définition.

3. Exigences logicielles : une lecture critique de SWEBOK

Dans cette section, nous proposons une lecture critique du chapitre 2 de SWEBOK dont nous garderons les mêmes titres de section. Cette lecture « critique « ne prétend être ni complète, ni objective : il s'agit d'une lecture influencée par l'approche du GL que nous avons présentée dans les trois articles précédents. Ce qui implique qu'une critique de notre critique pourrait s'avérer très utile, car notre objectif principal, à ce stade-ci de la réflexion, est de permettre de

déclencher des discussions en partant d'un fond sinon solide au moins un peu moins nébuleux.

Toutes les citations, à moins d'indication contraire, sont tirées de ce chapitre de SWEBOK. Il n'est sans doute pas inutile d'insister sur le fait que la critique ne concerne pas le chapitre en soi, qui reflète d'une manière très scrupuleuse l'état de l'art, mais la vision sous-jacente des exigences, une vision qui est celle de l'énorme majorité des ingénieurs du logiciel.

3.1 INTRODUCTION

Une exigence logicielle est définie comme l'expression « *des contraintes et des besoins auxquels un produit logiciel doit répondre, logiciel qui contribue à la solution d'un problème du monde réel* ». Ici, il s'agit donc des exigences de type 2 : c'est-à-dire que l'on ne considère les besoins et les contraintes que le logiciel doit satisfaire en tant que composant d'une machine. Quand on considère le logiciel du point de vue des utilisateurs ou des machines externes, il n'est qu'un composant d'un système qui contient minimalement l'ordinateur où le logiciel s'exécute. Parler de composant implique que la conception du système a déjà été réalisée, au moins partiellement. Malheureusement cette définition est incompatible avec le fait que l'on parle ensuite d'élicitation des exigences, car l'élicitation est le propre des exigences de type 1. Dans cette introduction on a une premier exemple de dérapage entre exigences de type 1 et exigences de type 2 ; dérapage que bien peu d'auteurs évitent, et qui est, sinon la cause principale, certainement l'une des causes importantes du manque de clarté actuel.

On privilégie une classification des exigences fondée sur le processus plutôt que sur les produits, car « *Le découpage fondé sur le processus reflète le fait que le processus des exigences, pour réussir, doit être considéré comme un processus comportant des activités complexes et fortement couplées* ». Mais les produits, sortis par la porte rentrent par la fenêtre, car, dans la suite du chapitre, on nous propose de considérer les exigences à travers : 1) Document de définition du système, 2) Spécification des exigences système, 3) Spécification des exigences logicielles.

NOTE. Qu'à l'intérieur d'un chapitre intitulé Exigences logicielles on introduise une section Exigences à l'intérieur de laquelle on parle, entre autres, des Spécifications des exigences système serait très étonnant si le glissement des exigences de type 1 à celles de type 2 n'était pas très normal dans le GL. FIN DE NOTE

3.2 FONDEMENTS

« *Les exigences imposées à un logiciel particulier sont généralement une combinaison complexe d'exigences en provenance de personnes différentes à différents niveaux de l'organisation et de l'environnement dans lequel le logiciel opère.* » Ici, puisque l'on considère les exigences du point de vue de l'origine, il s'agit des exigences de type 1. L'expression « *combinaison complexe* » souligne qu'il ne s'agit pas d'une simple juxtaposition d'exigences et laisse justement entendre qu'il peut y avoir des superpositions, des mélanges et même des contradictions entre les points de vue des différents intervenants sur les exigences.

On peut imaginer un grand nombre de façons très différentes de démêler cette combinaison complexe, toutes à l'intérieur des deux pôles suivants :

1. Par des discussions, des entrevues, des observations des utilisateurs au travail, etc. essayer de purifier les exigences et préparer un document « propre » où les problèmes d'interprétation et les différends sont réglés. Il s'agit de la méthode la plus classique, celle qui a fait naître dans les années 1970 le développement que l'on appelle « fondé sur les exigences ».
2. Conserver les différentes interprétations dans la documentation des exigences, à la limite jusqu'à la fin du projet, et employer les différences comme des verres grossissants pour mieux saisir certaines particularités qui pourraient se révéler fondamentales dans les développements futurs. Cette approche, pour être efficace, nécessite un développement « fondé sur l'architecture « plutôt que sur les exigences. Puisque, quand la machine sera fonctionnelle, elle n'aura pas de contradictions internes[3], il faudra que ces dernières soient résolues par des personnes avant ou pendant la programmation. Cette manière extrême[4] de fonctionner, à l'apparence si étrange quand on parle de GL, n'est pas dépourvue de retombées positives en ce qui concerne la qualité : les programmeurs doivent rendre leurs programmes plus solides et plus facilement modifiables pour être prêts à répondre aux changements d'orientation.

Le fait que, dans les *Fondements*, l'on souligne la différence entre exigences *imposées au produit* et exigences imposées au processus et que, comme exemple d'exigence de ce dernier cas on cite « *le logiciel doit être développé en Ada* », met encore une fois en évidence la variété des signifiés d'« exigence ». Variété qui n'est pas liée au « *problème du monde réel* » mais aux contraintes techniques et organisationnelles[5] du « monde réel » mises en œuvre pour résoudre le « problème ».

Un autre élément classificatoire (discriminateur) introduit dans cette section est *fonctionnel* versus *non-fonctionnel :*

- Exigences fonctionnelles : « *décrivent les fonctions que le logiciel doit exécuter* »
- Exigences non-fonctionnelles : celle qui « *fixent des contraintes sur la solution* ». Les exigences non fonctionnelles étant « *parfois connues comme contraintes ou exigences de qualité* »

Cette séparation n'est valide que pour les exigences de type 1. Quand on parle d'exigences de type 2, les exigences de qualité sont déjà transformées en fonctions qui les mettent en œuvre. Qu'il suffise, à ce propos, de penser aux composants logiciels qu'il faut développer pour obtenir certains niveaux de sécurité, de sûreté, d'interopérabilité, de maintenabilité ou de fiabilité.

« *Certaines exigences représentent des propriétés émergentes du logiciel : c'est-à-dire des exigences qui ne peuvent pas être abordées par un seul composant, mais qui dépendent, pour sa satisfaction, de la manière dont les composants interagissent* ». Ici, on considère la machine logicielle comme constituée des composants - elle est une boîte ouverte et donc au moins partiellement conçue. De cette manière on s'est éloigné des deux définitions d'IEEE et de la définition citée au début de cette section. L'utilisation du mot « composant » pour les exigences de type 2, n'est pas la même dans IEEE et dans la citation ci-dessus. Dans la définition d'IEEE, le composant est considéré un élément isolé, il est un « système » à lui tout seul et doit répondre à des exigences parce que contractuellement on le demande (comme dans le cadre d'une maintenance perfective). Dans la phrase que nous venons de considérer, composant est considéré comme en tant que partie du tout, du système d'où émergent certaines propriétés. Ce qui importe, ce n'est pas le composant en soi, composant qui tout étant partie d'une unité plus grande a son autonomie, mais d'un composant en tant qu'élément d'une structuration où, ce qui compte, c'est la structuration. En ce qui concerne les propriétés émergentes, nous croyons qu'il faut les considérer de façon complètement opposée : la propriété « émergente » est ce qu'on exige, c'est la vraie exigence et la division en composants est une manière de concevoir pour satisfaire l'exigence qui doit être insensible à la structure du logiciel : ce n'est pas la propriété qui émerge mais le composant.

Le concept de propriétés émergentes de SWEBOK est le symptôme d'une approche aux exigences du point de vue des concepteurs de logiciel qui, aux prises avec « leur » composant, doivent considérer la façon dont il interagit avec les autres composants pour partager les responsabilités et faire émerger les propriétés holistiques.

3.3 Processus des exigences

De cette section nous ne considérons que la partie concernant les acteurs, c'est-à-dire « *le rôle des gens qui participent au processus des exigences* ». L'acteur principal est le spécialiste des exigences qui doit être un médiateur entre « *le domaine des parties prenantes*[6] *et le domaine du GL* ». Pourquoi introduire cette nouvelle figure ? La réponse nous est donnée au début du chapitre.

« Le syntagme « ingénierie des exigences « est largement employé dans le domaine [du GL] pour dénoter un maniement systématique des exigences. Pour des raisons de consistance, pourtant, ce syntagme ne sera pas employé [...] puisqu'on a décidé d'éviter l'emploi du terme « ingénierie » pour des activités autres que celles du GL. [...] Pour la même raison, « ingénieur des exigences » [...] ne sera pas employé non plus. À la place, le syntagme « ingénieur du logiciel » ou, dans certain cas particuliers « spécialiste des exigences » sera employé, le deuxième quand le rôle en question est normalement joué par un individu autre qu'un ingénieur du logiciel »

Cette justification d'aller à l'encontre de ce que la majorité des gens font (« *largement employé* ») qui semble être dictée par des choix éditoriaux ou bureaucratiques (« *on a décidé* »), est à notre avis un autre élément qui souligne un malaise évident dans la manière de considérer les exigences dans le GL. Le fait de parler d'« *ingénierie des exigences* », d'une part, peut être considéré un pas en avant dans le sens de libérer les exigences de la tutelle du GL, mais de l'autre il s'agit d'un pas en arrière, car on impose aux exigences la laisse trop courte de l'ingénierie. Le syntagme « spécialiste des exigences », introduit un peu à la bonne franquette, et qui semblerait aller dans le sens de notre réflexion est en effet, à notre avis, un remède qui ne fait que retarder l'explosion du mal. Comme nous l'avons signalé en [1], le nombre pratiquement infini des domaines rend inefficace toute introduction de domaines horizontaux couvrant des besoins communs à tous les domaines verticaux. Le seul « domaine » horizontal valable pour toutes les automatisations est le « domaine » du langage qui... n'est pas un domaine du tout.

Que cette figure ne soit qu'une échappatoire éditoriale, on le voit quand, après l'avoir introduite, on écrit « *Il ne sera pas possible de satisfaire parfaitement les exigences de tous les intervenants, et c'est le travail de l'ingénieur du logiciel de négocier les compromis* ». Si c'est l'in-

génieur du logiciel qui négocie, à quoi sert le spécialiste des exigences ?

À propos des choix terminologiques et de la consistance[7] il nous semble utile d'ajouter que, dans SWEBOK, on souligne que le découpage du domaine des connaissances des exigences est compatible avec IEEE 12207, mais qu'on appelle « processus » des exigences ce que, dans IEEE 12207.0, on appelle « activité ». Activité constituée de trois tâches (*Établir et documenter les exigences logicielles, Évaluer les exigences, Tenir des revues conjointes*) qui ne sont pas nécessairement cohérentes avec le découpage de SWEBOK.

Il est évident que toutes ces difficultés terminologiques et les tentatives, plus ou moins adroites, de les régler ne sont pas le résultat d'une mauvaise qualité de la synthèse de SWEBOK, mais de la volonté du GL de s'approprier des « exigences ».

3.4 Élicitation des exigences

Élicitation est un terme emprunté à la psychologie pour indiquer l'« extraction » des connaissances des personnes. Dans le processus des exigences, c'est « *la première étape de la construction, une compréhension du problème qu'on demande au logiciel de résoudre* ». Il s'agit donc d'extraire les exigences des différentes sources. Et les différentes sources sont les éléments hétérogènes et imbriqués suivants : *Objectifs de haut niveau du logiciel, Connaissance du domaine, Parties prenantes, L'environnement opérationnel, L'environnement organisationnel.*

Si on voit très bien comment on peut éliciter les exigences des parties prenantes, on voit mal comment on peut éliciter les connaissances des documents existants ou en observant les comportements des différents intervenants et on voit encore plus mal comment on peut éliciter les connaissances des connaissances du domaine.

Vraisemblablement le terme élicitation est employé dans un sens très vaste et qui couvre donc l'élicitation « classique », l'extraction des documents, la synthèse en partant du comportement, etc. Cet élargissement de sens, qui serait acceptable dans un domaine de connaissance avec une bonne solidité, est très dangereux en GL, où on a encore besoin de l'opération intellectuelle inverse, celle de la différenciation, pour mieux définir les concepts - comme on demande de le faire dans le processus des exigences.

Nous sommes d'accord avec l'affirmation qu'une attention particulière doit être réservée « *à l'évaluation de la priorité relative et des coûts des objectifs* » mais, dans la même ligne de pensée, nous aimerions ajouter qu'il est aussi très important de parler le degré de stabilité des objectifs, ce qui devrait permettre non seulement de les classer en tant qu'objectifs stratégiques ou tactiques, ou en tant qu'objectifs à long, moyen ou court terme mais de leur donner une valeur de stabilité qui n'est pas nécessairement liée à leur portée temporelle ou à leur importance.

En ce qui concerne la *connaissance du domaine*, nous avons des difficultés à comprendre comment elle peut permettre à l'ingénieur du logiciel d'« *inférer les connaissances tacites que les parties prenantes n'articulent pas* » quand ce sont souvent les connaissances tacites qui sont l'essence des connaissances du domaine. Il semble qu'on soit dans un cercle vicieux qu'il n'est facile de rompre que si on limite la signification d'élicitation à son sens primitif.

3.5 Analyse des exigences

L'analyse des exigences est présentée comme étant constituée de quatre sujets :

- *Classification des exigences.*
- *Modélisation conceptuelle.*
- *Conception de l'architecture et répartition des exigences.*
- *Négociation des exigences.*

Si classification, modélisation et n*égociation* ont, entre elles, assez de cohésion pour pouvoir être considérées comme des constituantes de l'analyse, la conception de l'architecture et donc la répartition des exigences nous semblent nécessiter des connaissances, des outils et des méthodes très différentes, ce qui les rend étrangères à l'analyse des exigences. Voir 3.5.3 pour une analyse plus approfondie de ce point.

3.5.1 Classification des exigences

Six dimensions pour la classification sont présentées tout en soulignant que « *d'autres classifications sont valables* ». Analysons les dimensions proposées :

- *Exigences fonctionnelles ou non-fonctionnelles.* Voir nos commentaires en 3.2.
- « *Si l'exigence est dérivée d'une ou plusieurs exigences de haut niveau ou si elle est une propriété émergente* ». Pour les propriétés émergentes, voir nos considérations en 3.2. Pour ce qui est des exigences dérivées, il s'agit clairement d'exigences de type 2 considérées dans un état avancé de la conception système.
- *Exigences du produit ou du processus.* Il nous semble dangereux de subsumer deux éléments ayant des impacts si différents dans la vie d'un produit en un seul concept.
- *Priorité de l'exigence.* Importante pour le projet, mais qui acquiert une grande utilité sur-

tout quand on la met en relation avec la stabilité. Il s'agit clairement d'une exigence de type 1.

- *Portée de l'exigence*. Valable pour les exigences de type 2, si la conception système existe.
- *Stabilité de l'exigence*. C'est à notre avis un des éléments les plus importants et que nous considérerons en 4.

3.5.2 Modélisation conceptuelle

« *Le développement de modèles d'un problème du monde réel est la clef pour l'analyse des exigences logicielles* ». Impossible de ne pas être d'accord avec cette affirmation, même si dès que l'on essaie de bien la comprendre on est envahi par bon nombre de questions. Ici encore, le manque de précision terminologique obscurcit le « monde réel ».

À quel type de modèle fait-on référence ? Analytique ou analogique ? Est-ce que l'on modélise un problème ? Ne modélisons-nous pas plutôt une partie du monde réel et c'est cette modélisation qui permet de comprendre le problème ? Le modèle ne sert-il pas avant tout à décrire les propriétés du domaine plutôt que les exigences ? À moins, bien sûr, d'appeler *exigences* les propriétés du domaine. Questions simples aux réponses complexes et variées que nous essayerons d'effleurer.

« *Dans presque tous les cas, il est utile de commencer* [la modélisation] *en construisant un modèle du contexte du logiciel.* Le contexte du logiciel fournit une connexion entre le logiciel prévu et l'environnement externe. « *Ceci est crucial pour comprendre le contexte du logiciel dans son environnement opérationnel et pour identifier ses interfaces avec l'environnement* ». En dehors du fait que le contexte d'un logiciel est toujours le matériel d'ordinateur ou un autre logiciel, le fait qu'il s'agisse d'un contexte de type fonctionnel (comme les cas d'utilisation) ou d'un contexte physique n'est pas sans conséquences.

3.5.3 Conception de l'architecture et répartition des exigences

« *À un certain moment l'architecture de la solution doit être dérivée. La conception de l'architecture est le point où le processus des exigences chevauche la conception du logiciel ou du système et montre qu'il est impossible de nettement découper les deux tâches*[8] ». Nous pourrions à la limite accepter que la séparation entre exigences et conception système est moins nette que l'on ne pourrait le penser de prime abord, mais nous trouvons qu'il est faux d'affirmer que l'on ne peut pas séparer la conception du logiciel de l'analyse des exigences de type 1. Même en considérant qu'il est impossible de séparer nettement les différentes activités dans le cadre de la réalisation d'un produit[9], il est évident que s'il est une place où la séparation est nette, c'est bien celle entre analyse des exigences et conception du logiciel. Souvent, lorsqu'on entend parler de la difficulté de séparation, on confond la séparation temporelle avec une séparation d'ordre fonctionnel : mais ce n'est pas parce que deux activités sont effectuées en même temps que les deux activités ne sont pas distinctes. Pendant que maman prépare le dîner, sa fille met le couvert. Les deux tâches sont clairement séparées, même si elles sont effectuées en même temps et avec le but commun de faire manger l'ogre qui rentre après une dure journée de bureau, n'est-ce pas ? Pour voir la clarté de la séparation, il suffit de considérer le type différent de connaissances nécessaires pour les deux activités.

Cette confusion ne serait justifiée que si l'on donnait à conception une signification différente de celle d'IEEE ; et pourtant non, car on parle bien de composants logiciels : « *l'ingénieur du logiciel agit comme un architecte du logiciel parce que le processus d'analyser et d'élaborer les exigences demande que le composant qui sera responsable de satisfaire l'exigence soit identifié* ». L'ingénieur du logiciel agit comme un architecte du logiciel, car sa première fonction est celle de définir la structure et non de modéliser conceptuellement le domaine ou d'éliciter les exigences. Le problème, encore une fois, naît du fait que lorsque l'on parle d'exigences de type 2, on se fait facilement emporter par l'organisation en composants. Toutefois, cette confusion n'est pas attribuable à l'incapacité ou au manque de préparation des ingénieurs du logiciel mais au fait que l'angle par lequel on considère les exigences est celui des ingénieurs du logiciel qui agissent comme de vrais ingénieurs du logiciel, c'est-à-dire surtout comme des concepteurs.

Considérons l'exemple rapporté dans SWEBOK pour montrer le chevauchement.

« *Les exigences pour les performances de freinage d'une auto (distance de freinage, sûreté dans des conditions de conduite limitées, douceur de l'application, pression de la pédale requise, et ainsi de suite) peuvent être assignées au matériel de freinage (parties mécaniques et hydrauliques) et à un système de freinage antibloquant (SFA). Seulement quand une exigence pour un système SFA a été identifiée, et l'exigence lui a été allouée, les capacités du SFA, le matériel de freinage, et les propriétés émergentes (comme le poids de la voiture) peuvent être employés pour identifier les exigences logicielles détaillées du SFA.* »

Ici, il s'agit d'allouer des exigences déjà élicitées, analysées, et négociées à l'un des deux composants qui permettent à un conducteur de freiner. L'opération d'allocation peut bien sûr causer des retours sur l'élicitation, l'analyse et la négociation, mais il s'agit d'un simple retour et, à moins d'une impossibilité technique, il n'y aura pas de grands changements. En ce qui concerne le poids de l'auto, nous avons des difficultés à l'imaginer comme une propriété émergente : à moins que l'on établisse le poids de l'auto en fonction du SFA, ce qui est sans doute possible mais ne doit pas être souvent le cas.

NOTE. Même si un système a besoin de tous ses composants pour fonctionner, les différents composants n'ont pas tous le même poids fonctionnel, économique ou structurel. L'arrière-bec d'un pont n'a pas la même importance que l'abloc. Les ongles d'une femme, même s'ils sont importants, n'ont pas, normalement, le même poids pour le « système « femme que ses yeux ou son cerveau. FIN DE LA NOTE.

3.5.4 Négociation des exigences

Quand on négocie, on ne prend pas de décisions unilatérales. Pas d'étonnement donc quand, dans une section sur la négociation des exigences, on lit : « *Il n'est pas sage que l'ingénieur du logiciel prenne des décisions unilatérales* ». Dans ce contexte, « *sage* » est, sans l'ombre d'un doute, un euphémisme. Si l'on considère l'acception de type 1 d'exigence, l'ingénieur du logiciel n'a pratiquement rien à dire sinon comme conseiller expert en logiciel même s'il est un cas où il peut prendre des décision unilatérales : lorsqu'une exigences de type 1 a de tels impacts sur le logiciel qu'il est impossible de réaliser la machine dans le cadre des contraintes fixées. Dans ce cas, l'ingénieur du logiciel veille à ce que les autres parties prenantes ne se laissent pas entraîner trop loin par l'optimisme du « *tout domaine contient un partie automatisable* ».

La négociation des exigences, activité très souvent sous-évaluée[10], est celle où l'on voit le plus clairement que la qualité la plus importante des parties prenantes, c'est une bonne capacité de conviction et donc une grande maîtrise de la langue, le tout bien appuyé sur le socle de la connaissance du domaine. C'est aussi dans la négociation des exigences qu'il apparaît clairement que les métaphores « problème », « ingénierie » et « besoins » sont trop restrictives..

3.6 SPÉCIFICATION DES EXIGENCES

Les trois types de documents « classiques » sont présentés.

- Document de définition du système[11]. « *Ce document présente les exigences système avec les informations de base concernant les objectifs globaux du système, son environnement cible et la formulation des contraintes, les hypothèses et les exigences non-fonctionnelles. Il peut contenir un modèle conceptuel conçu pour monter le contexte du système, les scénarios d'emploi, et les entités principales du domaine [...]* »
- Spécification des exigences du système. Spécification à écrire pour des « *systèmes avec des composants matériels et logiciels importants. [...] Les exigences logicielles sont dérivées des exigences du système* ».
- Spécification des exigences logicielles. « *La spécification des exigences logicielles établit les bases pour un accord [...] sur ce que le produit logiciel doit faire et aussi ce qu'il n'est pas censé faire. [...] Pour des lectures non techniques [elle] est souvent accompagnée d'un document de définition des exigences* ».

La première question que les néophytes posent aux experts quand on leur parle de ces trois types de documents est souvent du genre : « Mais, pratiquement, que dois-je écrire dans chacun d'entre eux ? » Et si l'on ne se contente pas de les renvoyer aux tables des matières proposées par les normes, on s'aperçoit que la réponse n'est jamais facile. Seuls les gens d'expérience qui dorment sur les clichés ou répètent des formules livresques ont l'impression que le partage est clair.

Pourquoi ?

Nous hasardons une réponse timide : les différences entre les trois documents, fondée sur un discriminateur « technique » et sur le détail et la précision de la description, empêchent de fixer les piliers de l'édifice des exigences. La discrimination devrait, plutôt, être fondée sur des critères internes aux exigences comme leur stabilité. Si le discriminateur est la stabilité, le premier document pourrait être très détaillé quand il décrit des exigences sûres et stables et plus superficiel quand il traite d'exigences « en discussion ». C'est-à-dire que l'on ne demanderait pas à un document d'être uniforme par rapport aux détails seulement parce qu'il s'agit d'une description système et non d'une spécification logicielle. De la même manière que l'on établit une architecture du logiciel, on peut établir une architecture des exigences sans, bien sûr, penser qu'il y a une quelconque transformation simple de l'une à l'autre, comme pourraient le penser les adeptes de l'un des mythes les plus dangereux du génie logiciel, celui du développement « sans couture » (*seamless development*).

3.7 VALIDATION DES EXIGENCES

La validation est un processus très délicat dont les responsabilités sont énormes, car c'est le c'est lui qui doit établir si le logiciel exécute les fonc-

tions voulues avec la qualité exigée. La validation exige des points d'ancrage solides par rapport auxquels on s'assure que le produit demandé a bien été réalisé. Un produit final, par exemple, est validé par rapport à un document contenant les tests d'acceptation. Mais, lorsqu'on parle des exigences, par rapport à quoi peut-on les valider, étant donné qu'elles sont, en quelque sorte, à l'origine du cycle ? SWEBOK présente quatre moyens :

- *Revues.* Les revues sont réalisées par les parties prenantes qui analysent le document, souvent avec l'aide d'une liste des éléments à vérifier. Fondées sur l'échange, elles peuvent être considérées comme un condensé, dans le temps et dans l'espace, des discussions qui ont eu lieu lors de l'élicitation et de l'analyse. L'analyse détaillée en solitaire des documents avant l'échange en réunion est ce qui rend les revues si utiles.
- *Prototypes.* Pour la validation des exigences, ce sont surtout les prototypes des interfaces personne-machine qui sont considérés. Ils ont souvent des fonctions multiples, car ils peuvent aussi être employés pour éliciter, analyser ou décrire les exigences.
- *Validation de modèles.* Il s'agit d'une partie de l'analyse du document des exigences qui, dans cette classification de SWEBOK, a été soustraite aux revues sans doute à cause de son importance. Mais il va de soi que la validation du modèle doit être discutée dans une revue.
- *Tests d'acceptation.* Il s'agit surtout de la conception des tests d'acceptation. Dans ce type de test, il est important de ne pas se faire influencer par la structure modulaire du logiciel - comme dans les tests système ou d'intégration. Les tests d'acceptation sont, à leur tour, validés par rapport aux exigences, ce qui crée une dynamique d'interaction fort utile pour mieux décrire les éléments flous.

« *La validation des exigences consiste en l'examen du document*[12] *des exigences pour assurer qu'il définit le bon logiciel, c'est-à-dire le logiciel auquel s'attendent les utilisateurs »,* si l'on place cette phrase à côté de cette autre *« Les exigences peuvent être validées pour assurer que l'ingénieur du logiciel ait compris les exigences* », on peut facilement constater que l'on considère que l'ingénieur du logiciel a participé à l'écriture du document des exigences et, éventuellement, qu'il en était le responsable. Mais il suffit de jeter un coup d'œil aux quatre moyens présentés par SWEBOK pour s'apercevoir qu'aucun ne demande la présence d'un ingénieur du logiciel.

1) *Revue.* Lors des revues des exigences, non seulement les connaissances en génie lgiciel sont inutiles, mais elles sont contre-productives, car elles risquent de donner aux exigences une tonalité trop orientée conception ou mise en œuvre.
2) *Prototypage.* Dans le prototypage, la facilité qu'ont les ingénieurs du logiciel à manipuler les ordinateurs devient parfois un élément qui pousse les utilisateurs, en fonction de leur caractère, à exprimer leurs idées avec trop de timidité ou avec trop d'arrogance. Et ce sont ces « trop » qui augmentent la difficulté d'atteindre les objectifs. Il va de soi que, compte tenu des langues actuellement disponibles pour les interfaces personne-machine, il est préférable que le prototype ne soit pas préparé par l'ingénieur du logiciel.
3) *Modèle*. Pour la validation du modèle, en raison de ses connaissances des langages de modélisation, l'ingénieur du logiciel augmente le bruit et les dissonances, et risque de donner une place centrale au formalisme et non au contenu, surtout si c'est lui qui a choisi le langage de modélisation.
4) *Acceptation.* La présence de l'ingénieur du logiciel risque de favoriser une approche par composants physiques plutôt que par éléments fonctionnels.

Quoiqu'on en dise dans la littérature, ces quatre moyens ont des faiblesses propres au fait que les exigences sont pratiquement à l'origine du développement[13] et donc il ne peut y avoir d'ancrage solide, au moins au niveau des documents. Même un prototype, qui peut être considéré comme un document spécial, n'est pas assez solide en raison de ses fonctions contradictoires : élicitation, analyse et validation. Les véritables points d'ancrage des exigences, ce sont les échanges entre les humains fondés sur leurs connaissances du domaine et des objectifs visés. Il est donc primordial que les personnes qui valident possèdent aussi (surtout ?) des connaissances du domaine qui n'ont pas pu être inscrites dans les documents - les connaissances tacites.

Dans la définition précédente, « logiciel « est employé pour indiquer le produit final exécutable et non l'ensemble des artefacts produits tout au long du cycle de vie. Ce qui est cohérent avec ce que l'on écrit, par exemple, dans la norme pour la mesure de la qualité des produits logiciels [7] : « *La validation est normalement effectuée sur le produit final dans des conditions d'opération définies* ». Ce « normalement » est une conséquence du fait qu'il faut avant tout s'assurer qu'on a bien réalisé, comme il est indiqué dans SWEBOK, « *le logiciel auquel s'attendent les utilisateurs* » et on peut aussi le considérer comme l'indice d'une conscience, ou d'un fatalisme, par rapport à la difficulté de la validation continuelle si importante pour obtenir un produit final inacceptable. Le manque de validation

continuelle est justement considéré comme un symptôme de la pauvreté des méthodes d'ingénierie chez les développeurs. Mais nous y voyons aussi, encore une fois, le symptôme d'une difficulté liée à un certain manque de rigueur dans l'emploi de la terminologie du GL. Si, par exemple, dans la phrase « *le logiciel auquel s'attendent les utilisateurs* » on considère le logiciel comme l'ensemble de tous les artefacts produits, alors la validation devient une validation de tous les artefacts[14]. Ce qui implique que, si l'on veut être cohérent, le terme « utilisateurs » doit lui aussi être étendu aux « utilisateurs» de tous les artefacts et donc, éventuellement, il faut le substituer à celui de « parties prenantes » et considérer que les parties prenantes peuvent changer en fonction du produit considéré. Dans le cas du produit « document des exigences », l'ingénieur du logiciel est un utilisateur final comme l'utilisateur de la machine où le logiciel s'exécute : il peut dire s'il considère que le produit qu'on lui a livré est facile à lire, cohérent, etc. Mais, moins il a participé à la préparation des exigences et plus ses critiques sur la forme pourront être approfondies et utiles.

Voir à ce propos le premier encadré.

3.7.1 Considérations pratiques

« *Le processus des exigences embrasse tout le cycle de vie du logiciel.* » Voilà une affirmation qui, au début des années 1970, aurait fait sursauter tout bon informaticien (c'était l'âge d'or des développement fondés sur les exigences, au moins en théorie) et qui actuellement fait partie des clichés de tout bon ingénieur du logiciel.

Comme tout cliché, il s'agit d'un durcissement et d'une dogmatisation d'évidences et de vérités contextuelles, mais la peur des clichés ne doit pas nous faire oublier que derrière le cliché se trouvent des vérités importantes. Qu'y avait-il d'important dans le vieux cliché sur les exigences qui devaient être « parfaites » à la fin de l'analyse et dans le nouveau cliché sur les exigences « toujours changeantes » ?

Dans le premier, il y a sans doute la nécessité de pousser les humains à approfondir leurs connaissances du domaine, car, trop souvent, une mince couche de savoir donne l'illusion d'une connaissance approfondie. Dans le deuxième, il y a la nécessité de permettre aux parties prenantes de changer d'avis sans que chaque fois les ingénieurs du logiciel aient à refaire leurs applications. En effet, dans la vie - dans la réalité physique comme dans la réalité intellectuelle - il est bien rare qu'il n'existe pas d'éléments fixes, éléments qui, si on les considère comme changeants, peuvent créer des situations angoissantes, impossibles à maîtriser et qui feront dépenser inutilement des quantités énormes de ressources.

Le processus embrasse donc tout le cycle de vie, mais de manière très différente. Une approche que nous avons déjà employée avec une bonne dose de succès dans certains projets, c'est de considérer même le premier développement comme de la maintenance perfective[15].

« *Il est presque toujours irréaliste de mettre en œuvre le processus des exigences comme un processus linéaire et déterministe [...] c'est à coup sûr un mythe que les exigences, dans le cadre des grands projets logiciels, puissent être parfaitement comprises et parfaitement spécifiées.* » On pourrait certes se demander pourquoi ce mythe a eu une telle emprise, mais il nous semble encore plus intéressant de se demander pourquoi tant de mythes circulent dans le GL et surtout pourquoi l'ensemble de ces mythes a une si forte cohérence[16]. Questions qui mériteraient de faire l'objet d'un livre à elles seules.

À l'essentielle *gestion des changements*, on ajoute la nécessité de pouvoir suivre à la trace les exigences (tracing) : « *une exigence devrait pouvoir être suivie en amont jusqu'aux exigences et aux parties prenantes qui l'ont introduite (de l'exigence logicielle à l'exigence système qu'elle aide à satisfaire, par exemple)* ». L'exemple signalé est important pour un concepteur système qui a un point d'ancrage solide dans les exigences systèmes et qui veut vérifier l'origine de l'exigence que doit satisfaire un composant. Mais, pour une exigence de type 1, de celles qui concernent les parties prenantes, il s'agit d'aller chercher l'origine de l'exigence système qui peut bien sûr être renvoyée en amont vers un « besoin « décrit dans un *ConOps*[17] ou dans un autre document qui serait à l'origine du processus formel d'automatisation. Malheureusement, la difficulté principale est inhérente à la gestion des changements des « besoins » initiaux (c'est pour cette raison que l'on parle de négociation des exigences). Où ancrer les besoins ? Qu'y a-t-il avant les besoins ? Rien. Souvent rien, du point de vue contractuel, mais hors contrat, avant, il y a le « chaos » des échanges, les luttes de pouvoir, les compromis... tout ce qui nie toute possibilité d'ingénierie. Mais impossibilité d'ingénierie n'implique pas impossibilité totale de maîtrise. Loin de là.

4. Quelques pistes

4.1 Introduction

Après avoir écrit en [1] que « *Toute analogie fondée sur les génies traditionnels crée des modèles erronés et de faux espoirs* » dans le GL, n'est-il

pas contradictoire, voire illogique, de vouloir sauver le côté « génie » dans le logiciel ? N'est-il pas préférable de favoriser une approche plus radicale qui consiste à éliminer toutes ces normes excessivement rigides ? Ne faut-il pas s'épargner la rédaction de tous ces documents qui rendent le travail des concepteurs et des programmeurs lourd et sans souplesse quand le propre de la « matière logicielle » c'est justement la légèreté et la souplesse ? Ne faut-il pas se libérer de l'écrit et mettre au centre l'oral et le produit lui-même ? On pourrait sans doute suivre les voies plus radicales vers lesquelles pointent ces questions mais nous croyons que, à l'étape actuelle du développement de la technique logicielle, ceux qui ont un regard critique sur le génie logiciel devraient essayer de trouver un équilibre entre deux approches enchanteresses qui fascinent par leur apparente simplicité :

1. Laisser le GL évoluer selon les grandes lignes qui ont été tracées dans les quarante dernières années en déplaçant les mêmes éléments de gauche à droite et de droite à gauche, en dépoussiérant de vieilles babioles et en introduisant des distinguos baroques. C'est-à-dire en ne changeant que ce qui ne change rien.
2. Mettre une croix sur le GL et passer à d'autres métaphores moins rigides et moins formelles. Ce qui veut dire, à notre avis, jeter le bébé avec l'eau du bain.

Les pistes qui suivent tentent de proposer une démarche possible qui évite le Charybde du GL actuel sans tomber dans le Scylla de la programmation naïve.

4.2 Un principe et ses corollaires

Des commentaires à propos du chapitre de SWEBOK sur les exigences nous allons extraire le principe suivant que nous allons ajouter aux neuf principes fixés dans les articles précédents (voir l'encadré 2) :

Dans l'automatisation il n'existe pas de domaine des exigences. (P10)

Ce qui existe, ce sont les exigences des (ou à l'intérieur des) domaines. C'est-à-dire que, dans le cadre de l'automatisation, les exigences sont liées à un domaine qui, lui, est l'objet d'une informatisation éventuelle. On peut bien sûr analyser les exigences indépendamment des domaines, mais il s'agit alors de la sociologie ou de la philosophie des exigences et non pas d'automatisation. De ce principe découle le corollaire suivant :

Le rôle d'expert ou d'ingénieur des exigences est un rôle vide. (C10-01)

Les exigences sont du ressort des experts du domaine, des utilisateurs des machines et des gestionnaires des entreprises qui développent ou achètent les machines.

4.3 Exigences et contraintes

Comme nous l'avons déjà souligné, le passage du domaine au DLN est fondé sur les capacités langagières des humains qui ont un fond de créativité non mécanisable. Cette créativité, unie au nombre pratiquement infini de domaines automatisables, implique qu'il n'est pas réaliste de proposer des méthodes, des langues et des outils universels. Nous proposons donc une simple classification générale qu'il faut adapter non seulement à chaque domaine mais aussi à l'histoire de l'automatisation particulière de l'« instance » du domaine concernée par l'automatisation.

Nous commençons par proposer une légère modification de la première définition d'exigence d'IEEE [4] qui n'en change pas le sens mais le précise selon la direction que nous privilégions. À l'original : « *Une condition ou une capacité dont un utilisateur a besoin pour résoudre un problème ou pour atteindre un objectif* » nous ajoutons une précision tirée de [9] « *L'exigence est ce que votre client aimerait considérer comme vrai dans le domaine du problème* ». Les exigences doivent donc se transformer en algorithmes qui ont la responsabilité de limiter le nombre de combinaisons possibles des données du domaine.

Nous appellerons *cadenassage sémantique* la fermeture que la machine opère sur la signification des termes de la langue du domaine. Le cadenassage sémantique ne doit pas être confondu avec l'appauvrissement sémantique dont nous avons parlé en [1] : l'appauvrissement opéré par la DLN laisse des ouvertures ambiguës et éventuellement contradictoires dans la langue du domaine tandis que le cadenassage de la machine ferme toutes les voies différentes de celles qui sont intégrées dans ses algorithmes. Ce qui ne veut pas dire qu'il n'y aura plus d'ambiguïtés mais que les ambiguïtés sont liées aux interprétations que les utilisateurs humains font des sorties de la machine - si les sorties deviennent les entrées d'autres machines, il n'y aura pas d'ambiguïtés.

Nous appellerons *contraintes* les propriétés du domaine qui ne peuvent pas être modifiées par le fonctionnement de la machine en construction. À titre d'exemple, les machines qui existent dans le domaine et qui interagissent avec la machine à construire imposent des contraintes. Du point de vue de la gestion de projet, les contraintes ne sont pas contestables, on en prend tout simplement acte pour, éventuellement, les approfondir.

Du point de vue du développement, les contraintes sont plus contraignantes que les exi-

gences et, tandis qu'une exigence peut facilement se transformer en contrainte, l'inverse est moins probable. La normalisation internationale est un moyen de transformer une exigence en contrainte. Par exemple, la normalisation d'IEC de l'interopérabilité des postes de transport de l'électricité a transformé l'exigence non-fonctionnelle interopérabilité en contrainte du domaine.

NOTE. Le terme contrainte n'est sans doute pas idéal, mais nous l'aimons à cause de l'idée d'entrave à la liberté d'action de la machine en construction opposée aux exigences qui sont les entraves que notre machine impose à la liberté du domaine. FIN DE LA NOTE.

4.4 CLASSIFICATIONS

Avant de présenter notre classification, nous allons discuter un paragraphe tiré de [10] qui nous semble donner un bon exemple de la nouvelle vision du développement du logiciel qui apporte une critique justifiée de la rigidité des approches classiques du GL.

« *Le développement du logiciel est, dans l'ensemble, un domaine de grands changements et d'instabilité, c'est-à-dire qu'il s'agit d'un domaine de développement de nouveaux produits. Le logiciel n'est pas normalement un domaine de production de masse ou prévisible : des zones de petits changements où il est possible et efficace de définir toutes les spécifications stables et les plans fiables dès l'origine.* » Nul doute que, normalement, le logiciel n'est pas « *un domaine de production de masse et prévisible* », mais ce *normalement* implique qu'il y ait aussi une « *production de masse et prévisible* » qu'une approche du logiciel ne peut l'oublier, quitte à le retrouver sur son chemin dans un avenir pas forcément très éloigné.

Par contre, il n'est pas évident que « dans l'ensemble » il existe un « *domaine de grands changements et d'instabilité* ». L'instabilité et les changements ne dépendent pas du logiciel mais des domaines que l'on veut automatiser. Ce que Larman appelle grands changements et instabilité sont souvent de petits changements fonctionnels que la mauvaise structure du logiciel ne peut supporter. Mais, même si l'on suppose qu'il y ait de grands changements, la tendance est orientée vers une plus grande stabilité. Pourquoi ? Parce que plus on avance et moins on a de domaines *software free*, ce qui implique que les machines logicielles qui existent déjà imposent des contraintes qui empêchent les « grands changements » parce que le domaine à été au moins partiellement cadenassé d'un point de vue sémantique.

Pour la classification, nous nous appuierons sur deux concepts de l'épistémologie de Imre Lakatos[18] : celui de noyau dur et de ceinture de protection. La figure suivante présente, à l'intérieur du domaine, le noyau dur entouré par la ceinture de protection et par les éléments flottants.

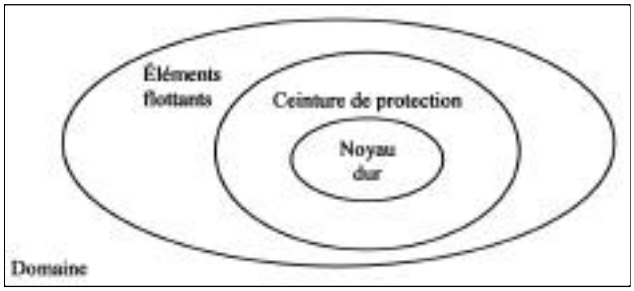

Figure 2 : Stabilité des exigences, des contraintes...

Cette classification peut aussi être considérée comme une généralisation et une mise au centre du degré de stabilité proposé en [8].

4.4.1 Noyau dur

- *Nom :* Noyau dur (ND).
- *Définition :* un ensemble d'exigences, de contraintes, d'objets ou d'objectifs que les parties prenantes considèrent comme stables pendant la durée de vie de la machine.
- *Commentaire :* à noter que l'on ne considère pas juste les contraintes.
- *Mobile :* avoir des points de repère solides pendant toute la vie du produit. Ce qui permet, par exemple, aux ingénieurs du logiciel de travailler sans avoir sans cesse à mettre en doute la nécessité de certains éléments.
- *Objection :* l'insertion d'un élément dans le ND complique les changements radicaux lorsque la conservation d'un élément crée des difficultés.
- *Analogie avec la physique :* les lois de la dynamique de Newton font partie du ND que la perturbation de l'orbite de Pluton ne change pas. Par contre, il faut l'approche radicale et géniale des pionniers de la mécanique quantique pour casser le ND de la physique atomique.
- *Note 1 :* le choix des éléments du ND peut être très difficile à faire et peut être influencé par les paradigmes cachés des parties prenantes. C'est pour cela que les éléments du ND doivent être documentés avec leurs justifications.
- *Note 2 :* la situation extrême présentant un ND vide est très rare. Elle peut s'expliquer par la nouveauté du domaine mais aussi par la pauvreté de l'analyse initiale.
- *Note 3 :* la situation extrême présentant tous les éléments dans le ND est assez rare. Elle peut s'expliquer par une définition formelle des interfaces préexistante, par la rigidité des méthodes de développement ou par une approche pilotée par les gestionnaires.

- *Note 4 :* la lecture du document présentant le ND permet aux ingénieurs du logiciel d'entrer dans le cœur du domaine et de saisir l'essence du problème.

4.4.2 Ceinture de protection
- *Nom :* Ceinture de protection.
- *Définition :* un ensemble d'exigences, de contraintes, d'objets ou d'objectifs ayant une stabilité satisfaisante dans la partie initiale du développement mais qui ont une grande probabilité de changer avant le retrait du produit.
- *Commentaire :* ce qu'est une « stabilité satisfaisante » dépend des parties prenantes et de l'histoire de l'automatisation du domaine.
- *Mobile :* protéger le ND des changements abrupts et de l'instabilité.
- *Objection :* définition trop vague qui dépend d'éléments très subjectifs.

4.4.3 Éléments flottants
- *Nom :* Éléments flottants.
- *Définition :* un ensemble d'exigences, d'objets ou d'objectifs qui varient, naissent et meurent tout au long du cycle de vie du produit.
- *Commentaire :* parmi les éléments flottants, il n'y a pas de contraintes.
- *Mobile :* tenir compte des changements et de l'instabilité.
- *Objection :* les éléments flottants ont une vie brève, car ils doivent cesser d'être flottants si l'on veut réaliser une machine.

4.5 Processus des exigences

4.5.1 Approche

Non seulement, en raison des grandes différences internes, il n'existe pas de processus[19] applicable de manière efficace à toutes les domaines mais il n'existe pas non plus de processus applicable de manière efficace à toutes les instances d'un domaine, car le processus est influencé par les dimensions relatives du noyau dur, de la ceinture de protection et des éléments flottants.

Cette difficulté n'implique pas pour autant que tout développement soit un développement *ad hoc* et qu'il n'y ait pas de discipline. La classification que nous proposons n'a d'autre prétention que de montrer une possible mise en ordre des activités sans que cela devienne une infrastructure qui limite les mouvements vitaux.

Pour que le processus des exigences soit un processus itératif et incrémentiel qui s'achève avec le retrait de la machine, il faut que les efforts, les ressources et les outils soient modulés dans le temps. Dans notre proposition, la modulation est donnée par l'importance relative du ND, de la ceinture et des éléments flottants. Mais cette importance relative non seulement module les efforts dans le temps mais aussi elle fait en sorte que le processus soit plus ou moins discipliné : de l'extrême d'une approche disciplinée de genre GL classique (ceinture et éléments flottants vides) à l'autre extrême d'une approche « jongleur » (ND et ceinture de protection vides) en passant par toutes les nuances possibles d'ordre et de discipline.

Si nous devions donner un nom à ce que nous proposons, par analogie avec le vieux MCEF (*Most Critical Element First*), nous l'appellerons MSEF(*Most Stable Elements First*), c'est-à-dire que le début du processus de développement consiste dans la recherche, l'élicitation, l'analyse et la spécification des éléments les plus stables, pour les intégrer au noyau dur.

4.5.2 Documentation

On suppose que trois documents différents sont générés, parmi lesquels au moins un ne doit pas être vide :
- Spécification du ND (SND).
- Spécification de la ceinture protectrice (SCP).
- Description des éléments flottants (DEF).

Les trois documents sont indépendants de la machine et sont le résultat du travail des parties prenantes mais, sans l'intervention des ingénieurs du logiciel, si ce n'est, éventuellement, que pour les vérifications formelles des documents. Par définition, la SND est la plus stable, la SCP est moyennement stable et la DEF est très volatile. Même si, dans la préparation des trois documents, on peut observer une certaine superposition, l'ordre d'écriture normal va du document le plus stable au document le moins stable.

Le tableau suivant donne un aperçu des différences entre les trois documents en fonction du moment du début de l'activité qui les concerne, des changements après la livraison du document, des parties prenantes principales et des mécanismes principaux de validation.

	Début	Changements après livraison	Parties prenantes principales	Validation
SND	Démarrage du projet	Très rares	Experts du domaine	Revues ; Historiques du domaine
SCPs	Spéc. ND avancée	Rares	Experts du domaine ; Cerdonnateurs ; Utilisateurs	Revues ; Historiques des projets
DEF	Spéc. ceinture avancée	Très fréquents	Utilisateurs	Produit final

Tableau 1 : Caractérisation des documents

Contrairement à SWEBOK où le premier document n'est pas une spécification tandis que les deux suivants le sont, dans notre approche c'est le dernier qui n'est pas une spécification. Ceci implique que ce n'est pas l'ordre de production des documents qui cause un formalisme plus ou moins grand, un détail plus ou moins grand, mais son contenu et en particulier l'indépendance du contenu de la machine à réaliser. Dans

le cas extrême que nous avons appelé « jongleur «, le document qui décrit les éléments flottants pourrait être un simple ensemble de notes de travail informelles[20].

- *SND.* La SND est une spécification sans aucune connotation logicielle et est complètement indépendante de la structure de la machine (des machines) qui mettra (mettront) en œuvre les fonctions requises pour satisfaire les exigences. Même si cette spécification a la même position dans le processus de développement que le *Document de définition du système* de SWEBOK, elle a un rôle et un contenu très différents. Le rôle de la SND n'est pas de « *définir les exigences système de haut niveau* » mais de « spécifier les éléments stables avec le maximum de détails». C'est-à-dire que la SND n'a pas besoin d'être ultérieurement détaillée, car les détails des éléments sont recherchés et documentés dès le début. Quant au contenu, tandis que le *Document de définition du système* définit *toutes* les exigences de *haut niveau* la SND définit *quelques* éléments (surtout les contraintes) avec tous les détails possibles.
  Dans un développement itératif et incrémentiel fondé sur l'architecture, la SND donne un minimum de stabilité pour qu'il n'y ait pas de « tsunami » fonctionnel ou de qualité qui balaie la machine.
  Dans un domaine où les seuls éléments stables sont les machines préexistantes, la SND ne contiendra que des références aux spécifications des machines.
  NOTE. La SND n'est pas l'équivalent de l'architecture fonctionnelle. Le ND est un ensemble d'éléments stables. FIN DE NOTE

- *SCP.* Il s'agit d'un document plus agile que la spécification du ND mais avec la même structure de la spécification du ND. Elle contient rarement des contraintes et des objectifs. Elle est surtout concernée par les exigences, c'est-à-dire par les limites que la machine doit imposer au domaine.
  Lorsqu'on demande de réaliser une machine dans un domaine très cadenassé par d'autres machines où il existe une ouverture vers un utilisateur spécifique, la SCP peut être vide.

- *DEF.* Il s'agit du document qui contient les éléments les plus volatiles et qu'il serait excessivement coûteux de consigner dans une spécification.

4.5.3 Quelques caractéristiques

Voici quelques caractéristiques de notre ébauche d'approche, qu'il faudra un jour préciser et justifier.

- Absence de raffinement d'un document à l'autre.
- Description de certains détails dès le début du processus.
- Mise en place des exigences comme trois mouvements à spirale sur la spirale principale du développement.
- Développement fondé sur l'architecture.
- Développement à approche variable en fonction de la stabilité relative des éléments.
- Importance des documents écrits en langue naturelle pour permettre un type de réflexion que la parole ou la parole et la machine ne permettent pas.
- Fondement sur l'hypothèse que les nouveaux développements seront toujours plus cadenassés.
- Adaptabilité à n'importe quel type de domaine.
- Application possible de la production à la chaîne à la recherche pure.

4.5.4 Exemples

Voici des exemples de classification des exigences à l'intérieur d'applications typiques :

1. *Protocole HDLC* pour l'environnement Unix. Il s'agit d'un cas où toutes les exigences fonctionnelles font partie du noyau dur (inscrites dans la norme ISO qui décrit le protocole). Par contre, certaines exigences non-fonctionnelles pourraient être placées dans la ceinture de protection.
2. *Système de contrôle/commande* pour des postes d'EDF où l'on impose la norme IEC 61850 pour l'interopérabilité. L'exigences non-fonctionnelle interopérabilité est transformée en contrainte par la norme. L'exigence fonctionnelle commande en deux phases est dans la ceinture.
3. *Gestion de la clientèle* pour un cabinet de dentistes. Si le cabinet n'a jamais été informatisé, il est probable que seuls certains objectifs se trouvent dans le ND.
4. *Contrôle de vitesse* d'une auto. Les exigences non-fonctionnelles de sûreté sont certainement dans le ND.
5. Une application pour gérer la petite (31 bouteilles) cave à vin d'un copain. ND et ceinture vide.

5. CONCLUSION

Dans les prolégomènes, et en particulier dans ce dernier article, nous avons essayé de montrer qu'il est préférable de soustraire du génie logiciel l'ensemble des activités concernant les exigences. Nous allons synthétiser les résultats de notre démarche sous forme de principe :

> *L'élicitation, l'observation, l'analyse, la spécification, la synthèse, la validation et la vérification des exigences concernent les experts du domaine, les utilisateurs, les gestionnaires et ne concernent pas les ingénieurs du logiciel.* (P11)

Même si nous avons proposé une nouvelle approche des exigences fondée sur la stabilité, il est clair que le fond de notre réflexion se limite à l'aspect critique.

Nous avons essayé, autant que possible, d'éviter l'emploi du terme « système », d'une part parce qu'il est devenu un terme fourre-tout et, de l'autre, parce que nous croyons qu'il est très important dans l'automatisation d'insister sur la différence entre l'humain et la machine, même si tous deux sont des systèmes, même si, ensemble, ils créent un système. Cette différence fondée sur la capacité langagière, source de créativité et de signification, permet de considérer l'automatisation selon deux points de vue opposés :

1. rigidité algorithmique et manipulation syntaxique avec ce que nous avons appelé un cadenassage sémantique du domaine pour les machines ;
2. adaptabilité et capacité d'interprétation avec la création de nouveaux espaces de signification pour les humains.

Ces deux points de vue opposés permettent de voir le processus des exigences, processus centré sur le langage, comme un processus qui nécessite une mise à distance dans l'écrit des concepts pour limiter les clashes interprétatifs des parties prenantes.

Voici pêle-mêle les points principaux, que nous avons parfois appelés principes, et qui nous ont guidés ou que nous avons eu l'impression d'atteindre :

- SWEBOK a été un énorme pas en avant, car il a permis de donner une image assez précise de ce que l'on appelle génie logiciel.
- Le génie logiciel a perdu trop de temps à imiter la grenouille d'Ésope.
- Le génie logiciel n'a pratiquement pas d'analogie avec les autres génies.

* La branche des exigences devrait être coupée du génie logiciel et laissée aux experts du domaine, aux gestionnaires et aux utilisateurs.

- Pour sortir de la cacophonie de la métrologie de la qualité, il faudrait avant tout employer les mesures comme un simple moyen de clarification conceptuelle.
- Les interfaces machine-machine devraient être au centre des intérêts des ingénieurs du logiciel.
- La stabilité pourrait être un discriminateur efficace dans le processus des exigences.
- Il est impossible d'établir des méthodes universelles valides (dans le sens de l'efficacité et de la qualité des produits livrés) pour tous les types de développement.
- La maintenabilité perfective est au centre de l'automatisation fondée sur le logiciel et un développement qui dépasse une durée de quelques semaines devrait être considéré comme de la maintenance perfective.
- Dans l'automatisation, il n'est d'aucune utilité de parler de domaine des exigences en général et donc d'introduire la figure de l'expert des exigences, de l'ingénieur des exigences ou du cognitiviste.
- La validation des exigences s'ancre au dialogue des parties prenantes.

Même si, sur le fond, les trois auteurs sont en accord, sur certains détails, les visions de chacun demeurent assez différentes, à propos de l'importance de la DLN, par exemple. C'est pourquoi nous avons introduit un encadré titré *La DLN court-circuitée* ! où une approche sans description en langue naturelle est proposée.

ENCADRÉ 1

DU BON ET DU BIEN AVEC CODA DIALOGIQUE

La définition « vulgaire » de la *validation* en tant que processus permettant de s'assurer que l'on a réalisé le *bon* produit et celle de la *vérification* en tant que processus permettant de s'assurer que l'on réalise *bien* le produit font désormais partie des acquis de base de tout étudiant en génie logiciel. Cette distinction est analogue à celle qui existe entre exigences et conception que l'on discriminait jadis à l'aide de deux autres termes magiques : *quoi* et *comment*. Mais la différence entre le « quoi » et le « comment » s'estompe dès que l'on change de point d'observation, il en va de même pour le « bon » et le « bien ». Ce n'est donc pas un hasard si la vérification et la validation sont si souvent solidement couplées dans l'acronyme V&V.

Après cette mise en garde, voici la définition de [3] de validation :

« *Confirmation par examen en fournissant une preuve objective que les exigences particulières concernant un emploi spécifique visé sont satisfaites.* »

Après cette définition, il n'est pas étonnant de retrouver la note suivante : « *la validation est normalement effectuée sur le produit final dans des conditions d'opération définies* » ce qui n'est qu'une définition plus formelle de ce que l'on entend par l'expression « bon produit «. Mais la note continue ainsi : « *elle peut être nécessaire dans les phases précédentes* », ce qui nous ramène à nos considérations sur les exigences. Pour nous aider à éclaircir les implications de la validation des exigences, on peut considérer une autre note associée à la définition précédente : « *Dans la conception et dans le développement, la validation concerne le processus qui consiste à examiner un produit pour en déterminer la conformité avec les besoins de l'utilisateur* ». Même si nous ne considérons pas la désinvolture avec laquelle, dans une norme qui définit la conception comme l'une des phases du développement, on met sur le même plan conception et développement, nous ne pouvons pas ne pas nous arrêter sur les « besoins de l'utilisateur ».

CODA DIALOGIQUE.

— Besoins de quel utilisateur ?
— De l'utilisateur du produit final, bien sûr ! De celui qui va employer la machine.
— Mais le document qui présente les exigences n'est-il pas, lui aussi, un produit ? « Final «, lui aussi, en quelque sorte.
— Certes.
— Qui, lui aussi, a des utilisateurs ?
— Comment le nier.
— Mais, qui sont ces utilisateurs ?
— Les ingénieurs du logiciel, et tous ceux qui participent aux revues.
— La note peut donc vouloir dire qu'il faut valider par rapport aux besoins des ingénieurs du logiciel et de l'utilisateur final, n'est-ce pas ?
— Sans l'ombre d'un doute.
— Est-ce que ces deux genres de validation ont quelque chose en commun ?
— De prime abord, je ne vois pas. Les produits sont tellement différents.
— Nous qui travaillons en GL et qui croyons que la maintenance perfective n'est pas une simple annexe du développement mais son vrai centre, nous tous, nous devrions y réfléchir. Mais aujourd'hui limitons-nous à dire que les besoins de l'ingénieur du logiciel sont beaucoup plus liés à la forme des documents qu'à leur contenu et contentons-nous de considérer l'utilisateur final.
— Oui, c'est déjà assez compliqué comme ça.
— Il faut « une preuve objective » que les besoins sont satisfaits. Il faut donc que... disons que l'exigence E001 soit reconductible à un ou plusieurs besoins et que le besoin soit clairement exprimé et interprétable sans ambiguïté.
— Ce qui est impossible avec des documents en langue naturelle.
— Comment ne pas être d'accord ? Alors doit-on laisser tomber la validation des exigences ?
— Je ne sais pas. Il semble que vous vouliez me faire dire que oui.
— Pas du tout.
— Mais comment faire, alors ?
— Il faut commencer par donner à cet « objectivement » une autre signification que celle qu'on lui donne d'habitude, du genre « indépendant de toute considération subjective ».
— N'est-ce pas plus simple de, simplement, effacer « objectivement » ?
— Plus simple mais moins utile si on veut approfondir ce qu'implique vouloir une « preuve objective » pour pouvoir valider les exigences. Considérons les revues, par exemple. Comment avoir une preuve objective en partant de deux documents en langue naturelle et qui sont donc assujettis à de possibles interprétations différentes ?
— On est revenu à la case de départ.
— Après un tour exploratoire qui permet de mieux cibler les concepts. Imaginons que la preuve objective soit le résultat des échanges entre les participants à la revue. On discute de l'exigence Ex001, on la compare avec le besoin B004, on trouve qu'il faut changer quelque chose en Ex001 mais en même temps en B004...
— Mais si on change le besoin, on ne valide plus rien.

ENCADRÉ 1 (SUITE)

— Ou on valide encore plus, parce qu'on revient en arrière, on approfondit, on fait naître de nouvelles idées...
— Et comme ça le développement ne finira jamais...
— Il finira parce que les temps alloués sont limités et les méthodes de travail doivent être appliquées. La revue prendra fin et finalement on aura des exigences et des besoins mieux précisés.
— Mais l'« objectivité » que l'on recherche dépendra surtout des qualités des intervenants.
— Aussi de la qualité des intervenants, mais surtout de leurs connaissances du domaine et de leur connaissance de ce que veulent les utilisateurs, les clients, les gestionnaires, etc.
— Mais qui connaît tous cela ?
— Les experts du domaine, les utilisateurs, les clients, les gestionnaires, bien sûr.
— Et les ingénieurs du logiciel ?

ENCADRÉ 2

PRINCIPES

P01 : *Tout domaine contient au moins une partie automatisable.* Principe d'ouverture, car il indique que dès que les humains créent un domaine, on peut y introduire une machine qui en automatise une partie

P02 : *Une partie d'un domaine est complètement automatisable si, et seulement si, ses frontières sont formalisables.* Principe de fermeture, car il indique que seul un sous-ensemble des domaines est complètement automatisable.

P03 : *Le génie logiciel n'est pas un génie comme les autres en raison du nombre illimité de domaines pour lesquels on peut produire des logiciels et de l'extrême facilité à générer des copies des exécutables.*
Avec son corollaire : *Toute analogie fondée sur les génies traditionnels crée des modèles erronés et de faux espoirs*

P04 : *La* maintenabilité *est la qualité intrinsèque principale de toute approche d'ingénierie au développement du logiciel.*

P05 : *Les coûts d'informatisation de quelque chose de complexe sont toujours élevés, quelle que soit la simplicité de la solution.*

P06 : *Pour qu'il y ait échange d'information, il faut un substrat.*

P07 : *Pour spécifier une interface, il faut des connaissances dans le domaine où la machine opère.*

P08 : *Seule la spécification des interfaces internes d'une machine logicielle appartient au domaine du GL.*

P09 : *Une personne ne peut pas échanger des données avec un logiciel.*

P10 : *Dans l'automatisation il n'existe pas de domaine des exigences.*
Avec comme corollaire : *Le rôle d'expert ou d'ingénieur des exigences est un rôle vide.*

P11 : *L'élicitation, l'observation, l'analyse, la spécification, la synthèse, la validation et la vérification des exigences concernent les experts du domaine, les utilisateurs, les gestionnaires et ne concernent pas les ingénieurs du logiciel.*

ENCADRÉ 3

TAXONOMIE DES APPLICATIONS VERSUS TAXONOMIE DANS LES APPLICATIONS

Dans l'introduction de [5], l'un des livres les plus cités dans SWEBOK, et, à notre avis, l'un des livres « classiques » les plus intéressants, après une partie sur l'importance des exigences, on en trouve une autre qui est intitulée *Taxonomie des applications*, qui permet d'approfondir notre critique des traitements des exigences dans le génie logiciel.

La taxonomie des applications est proposée dans le but de « Diviser le monde des applications logicielles en classes montrant des propriétés communes qui permettent d'appliquer uniformément un ou des outils et des techniques pour les exigences».

Les cinq classes proposées sont :

«

1. Difficultés du problème.
2. Relations temporelles entre les données et le traitement.
3. Nombre de tâches simultanées à exécuter.
4. Difficultés relatives des aspects données, contrôle et algorithmes du problème.
5. Déterministes versus non-déterministes.

»

Des cinq classes proposées, les quatre dernières sont d'une très grande utilité dans la conception. Il suffit d'avoir participé au développement de quelques applications pour savoir que ces quatre classes ont un grand impact sur la manière de décrire la conception. En effet, le titre du chapitre dit bien « taxonomie des applications », mais la taxonomie des applications, lorsqu'elle est appliquée aux exigences, devient une taxonomie qui brouille les pistes parce qu'elle introduit des exigences *qu'on impose à l'application* (structure de l'application) comme des exigences *qu'on demande de satisfaire* par l'exécution (la dynamique) de l'application.

En ce qui concerne la première classe, *Difficulté du problème*, il faudrait voir de quel problème il s'agit : du problème lié au domaine ou du problème de construction de l'application ? S'il s'agit du problème de construction de l'application, ce n'est pas dans le cadre des exigences qu'on le résout ; s'il s'agit d'un problème lié au domaine, de la difficulté de qui parle-t-on ? De celle des experts pour décrire le domaine et fixer les objectifs ou de celle des ingénieurs du logiciel qui doivent le comprendre ? Dans le premier cas, c'est le domaine (et donc ses experts) qui détermine les outils et les techniques ; dans le deuxième, il s'agit des problèmes de lisibilité, de cohérence, etc. c'est-à-dire de problèmes psychologiques propres aux humains et qui dépassent le cadre du génie logiciel.

Cet exemple de taxonomie est un très bel exemple de la confusion qui règne dans le monde du génie logiciel lorsque le « domaine des exigences « y est annexé. Cette annexion crée des frontières fluides entre la conception et les exigences qui peuvent facilement faire penser que le « développement *seamless* » est autre chose qu'une expression de marketing pour désigner des outils et des méthodes orientés objet. Ce n'est pas parce que l'on emploie UML dans les diagrammes conceptuels et dans la conception qu'entre les deux il y a une confusion possible. Que l'un des livres canoniques sur les exigences puisse se laisser aller à de telles considérations n'est pas un indice de la faiblesse de l'auteur mais celui de la maladie chronique (et non de la crise !) du génie logiciel.

Cette classification est une classification horizontale, c'est-à-dire qu'elle couvre tous les domaines pour trouver des outils et des techniques valables en fonction des attributs des applications et non des caractéristiques du domaine. Nous croyons que ce qui est important, c'est d'adopter des classifications verticales : c'est-à-dire en fonction des domaines d'application. Et c'est à l'intérieur de chaque domaine que l'on peut introduire des classifications des exigences comme nous l'avons fait en 4.

ENCADRÉ 4

LA DLN COURT-CIRCUITÉE !

Certaines habitudes de travail favorisent une approche de développement plutôt qu'une autre. La description en langue naturelle tend à favoriser un cycle de développement en cascade. Il est naturel de vouloir obtenir une DLN la plus complète possible avant d'aborder les phases suivantes. Pour certains systèmes, cela est nécessaire. Dans une grande majorité de cas, le cycle en cascade est contre-productif.

Le livre de Boehm et Turner, *Balancing Agility and Discipline*, Addison Wesley, 2004 propose une grille d'analyse pour déterminer quand une approche « disciplinée » est préférable à une approche « agile ». Force est de constater que, sauf dans les cas où la vie humaine est en jeu ou qu'un projet implique plusieurs centaines de développeurs, l'approche « agile » tend à être favorisée.

Les aléas du cycle de développement en cascade ont fait le sujet de nombreuses études. Larman, dans *Agile & Iterative Development*, Addison Wesley, 2004, consacre un chapitre entier à résumer les études les plus percutantes à ce sujet. Dans une étude citée, sur les projets analysés, 45% des fonctionnalités décrites dans les DLN n'ont pas été utilisées et 19%, l'ont été rarement, une fois le système développé.

Quelle est la solution de remplacement ?
Pour certains, la DLN n'étant ni requise ni souhaitable dans une majorité de cas, plusieurs solutions de remplacement ont été proposées ces dernières années. Parmi les plus connues, citons UP (Unified Process), XP (Extreme Programming), SCRUM et EVO (Evolutionary Project Management). Certaines de ces approches ne sont pas récentes. EVO fut utilisée dès les années 60 et a été publiée à partir de 1976.

Ces approches ont en commun les caractéristiques suivantes :

1. Ces approches sont à géométrie variable. Elles cherchent à couvrir un large espace allant d'un degré de « cérémonie » faible à moyennement fort. Ce degré de « cérémonie » se caractérise par l'exigence de produire ou non certains documents formels. Il appartient à l'équipe de projet d'établir très tôt le degré de cérémonie optimal et de ne pas hésiter, le cas échéant, à le modifier en cours de route.
2. Ces approches sont toutes itératives et incrémentielles. La durée d'un cycle peut varier de deux à huit semaines. Une durée ne dépassant pas quatre semaines est fortement suggérée. À chaque cycle, l'équipe du projet doit livrer un système fonctionnel, même si incomplet. Un système fonctionnel exclut des prototypes et inclut un design et du code de qualité « production ».
3. Plusieurs favorisent un échange directe et continu entre un représentant du client (le client étant le décideur et le payeur pour le système en opposition avec les utilisateurs du système) et l'équipe de développement. Ce représentant partage physiquement l'espace de travail de l'équipe. L'objectif est de diminuer le plus possible les délais et les problèmes d'interprétation entre ce que le client désire et ce que l'équipe en comprend.
4. Au début de chaque itération, l'équipe établit, avec le représentant du client, les fonctionnalités à implémenter dans l'itération. Avec l'approche XP, la description des fonctionnalités est écrite à la main sur une petite carte (story cards), une fonctionnalité par carte. Le détail sera établi au fur et à mesure par des discussions entre tous (réunions quotidiennes où les participants restent debout).
5. L'équipe, après quelques itérations, est capable d'estimer sa vitesse de réalisation. Avec l'approche XP, l'équipe dira qu'elle implémente, par exemple, en moyenne 15 story cards en deux semaines.
6. Le représentant du client, grâce à une rétroaction rapide, peut voir si le système en développement répond à ses besoins. Il est maître de modifier ses besoins et leur séquence d'implémentation à chaque cycle.
7. L'architecture du système est réalisée très tôt. Car on veut pouvoir démontrer le fonctionnement du système même incomplet en mode « production ». Les éléments risqués sont résolus le plus tôt possible.
8. Un minimum de documentation est produite, juste le nécessaire pour aider à la prochaine itération.

En conclusion
Les difficultés de la DLN sont inhérentes au fait qu'on tente de décrire un système en devenir et que chaque partie prenante en a une vision propre dans un vocabulaire lui étant propre. Ce problème est similaire à ce qui est rapporté dans la Bible concernant la Tour de Babel. Le génie logiciel cherche encore à résoudre ce problème. Les approches « agiles » contournent le problème en réalisant très tôt un système fonctionnel et en le modifiant à chaque itération avec l'apport nécessaire d'un représentant du client ayant les pouvoirs décisionnels requis. Il serait ainsi possible et préférable d'exclure la DLN du domaine du génie logiciel.

6. Bliographie

Sur le génie logiciel en général :

- I. Sommerville : *Software engineering, septième édition* : Addison-Wesley, 2005
  Avec Pressman, un des deux piliers au niveau de l'introduction au génie logiciel. Ce livre est une référence omniprésente dans le domaine.
- R. S. Pressman : *Software engineering: A practitioner's approach*, 6e édition ; McGraw-Hill, 2005
  L'autre classique.
- S. McConnell : *Rapid development* ; Microsoft Press, 1996
  Ce livre est beaucoup plus que son titre pourrait le laisser croire. Une excellente couverture du domaine avec à l'appui, les résultats de nombreuses recherches présentés de façon claire. Un livre que tout gestionnaire dans le domaine doit lire.
- P. DeGrace & L. Stahl : *The Olduvai imperative* ; Prentice-Hall, 1993
  La vision romaine et la vision grecque du génie logiciel. Pourquoi le GLAO n'a jamais eu le succès escompté.
- R. Glass : *Facts and fallacies of software engineering* ; Addison-Wesley, 2003
  Remet quelques pendules à l'heure...
- C. Larman : *Agile & iterative development: A manager's guide* ; Addison-Wesley, 2004
  Une introduction au développement agile avec des arguments frappants pour les gestionnaires.
- G. Weinberg : *Quality software management*, vol.1, 2, 3, & 4 ; Dorset, 1992, 1993, 1994, 1997
  Une vision parallèle et inspirante des modèles CMM et CMMI. Pour lecteur sérieux avec en prime de belles découvertes.
- F. P. Brooks : *The mythical man-month ;* Addison-Wesley, 1995
  Édition anniversaire de ce classique délicieux.
- T. DeMarco & T. Lister : *Peopleware: Productive projects and teams*, 2e édition ; Dorset, 1999
  La sociologie du développement du logiciel.

Sur les exigences :

- A. M. Davis : *Software requirements: Objects, functions and states* ; Prentice Hall, 1993
  Un des classiques sur les exigences ; Davis s'est joint à Rational avec son outil RequisitePro.
- S. Robertson et J. Robertson : *Mastering the requirements process* ; Addison-Wesley,1999
  Le couple britannique a produit cet ouvrage assez connu. Ils font maintenant partie du groupe The Atlantic Systems Guild Inc. avec, entre autres, Tom DeMarco et Tim Lister. Les Robertson ont un site contenant des ressources associées au domaine des exigences : http://www.volere.co.uk/
- I. Sommerville et P. Sawyer : *Requirements engineering: A good practice guide* ; John Wiley & Sons, 1997
  Un petit livre qui aborde le sujet d'un point de vue pratique et pragmatique... à l'anglaise.
- M. Jackson : *Software requirements and specifications* ; Addison-Wesley, 1995
  Un de nos préférés mais d'une lecture ardue.
- B.L. Kovitz : *Practical software requirements* ; Manning, 1998
  S'inspire de M. Jackson, avec de nombreux exemples facilitant la compréhension du sujet.
- D. Gause & G. Weinberg : *Exploring requirements quality before design* ; Dorset, 1989
  Aborde certains thèmes touchés par le présent article, en particulier la difficulté de la DLN. Son exemple de « Mary had a little lamb » est devenu un classique. Il se méfie des ingénieurs des exigences. Amusant, provoquant et ... à lire.
- K. E. Wiegers : *Software requirements* ; 2e édition ; Microsoft Press, 2003
  Vision contemporaine du domaine des exigences, dans le style génie logiciel. Assez populaire aux États-Unis.

Sur la conception :

- B. Meyer : *Object-oriented software construction*, 2e édition ; Prentice-Hall, 1997
  Une approche cohérente et rigoureuse.
- M. Fowler : *Patterns of enterprise application architecture* ; Addison-Wesley, 2003
  Une vision à un niveau macro avec un point de vue contemporain. À lire.
- M. Page-Jones : *Fundamentals of object-oriented design in UML* ; Addison-Wesley, 2000
  Un version améliorée et à la UML de son livre What every programmer should know about object-oriented design. Ce dernier livre a reçu le Jolt/Software Development 1995 Productivity Award.
- E. Gamma et Coll. : *Design patterns: elements of reusable object-oriented software* ; Addison-Wesley, 1995
  Le classique sur le sujet mais d'une lecture assez ardue. Ne pas commencer par ce livre.
- C. Larman, *Applying UML and patterns: An introduction to object-oriented analysis and design*, 3e édition, Prentice-Hall, 2005
  Représente la pensée dominante dans le domaine. Facile à lire avec deux études de cas réalisées avec l'UP. La dernière édition explique bien l'approche « agile ».

Sur la construction :

- S. McConnell : *Code complete: A practical handbook of software construction*, 2e édition ; Microsoft Press, 2004
  Un des rares ouvrages contemporains à aborder en détail cette thématique. Le passage sur le fameux GOTO vaut le détour.
- K. Beck, *Extreme programming explained: Embrace change* ; Addison-Wesley, 1999
  Un des livres fondateurs de ce mouvement assez mal connu malgré la couverture médiatique à son sujet.

• J. Bently : *Programming pearls*, 2e édition ; Addison-Wesley, 2000
Un style similaire à celui de Jackson. Un pensée profonde.
• A. Hunt & D. Thomas : *The pragmatic programmer* ; Addison-Wesley, 2000
Une vision grecque du travail du programmeur.

7. RÉFÉRENCES

[1] Ivan Maffezzini, Alice Premiana et Bernardo Ventimiglia : *Prolégomènes à une critique du génie logiciel*, Partie I : contextualisation ; Génie Logiciel, n° 66, septembre 2003, pages 2 à 16.
[2] IEEE, SWEBOK, 2004 version, http://www.swebok.org/
[3] Ivan Maffezzini, Alice Premiana et Bernardo Ventimiglia : *Prolégomènes à une critique du génie logiciel*, Partie II : Qualité et mesures ; Génie Logiciel, n° 68, mars 2004, pages 7 à 24.
[4] IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*.
[5] Alan M. Davis : *Software requirements ;* Prentice Hall, 1993.
[6] Ivan Maffezzini, Alice Premiana et Bernardo Ventimiglia : *Prolégomènes à une critique du génie logiciel*, Partie III : interfaces ; Génie Logiciel, septembre 2004, n° 70, pages 2 à 16.
[7] ISO/IEC, *International Standard 9126-1, Software engineering - Product quality*, Part 1: Quality mode, 2001.
[8] IEEE Std 830.12-1998, *IEEE Recommended Practice for Software Requirement Specification.*
[9] Michael Jackson : *Problem frames* ; Addison-Wesley, 2001
[10] C. Larman : *Applying UML and patterns: An introduction to object-oriented analysis and design*, 3e édition ; Prentice-Hall, 2005

NOTES

1 Un remerciement particulier à Véronique Dassas pour le travail ingrat de révision du texte.
2 La critique qui consiste à dire qu'il n'est pas nécessaire d'avoir un document contenant la description en langue naturelle et qu'il suffit d'avoir « la description » en tête est pour nous contraire à toute saine approche technique et donc irrecevable dans ce contexte. À moins bien sûr qu'il s'agisse de variations minimales sur des automatisations déjà réalisées plusieurs fois (mais, là aussi, il faudrait en discuter), des applications jouets ou de la recherche « pure ».
3 À moins de considérer les défauts comme des contradictions et non comme une « réponse correcte » aux erreurs humaines.
4 Cet extrême est un clin d'œil à la « programmation extrême » qui, expurgée d'un grand nombre de naïvetés, pourrait être l'une des méthodes du génie logiciel dans des domaines bien spécifiques.
5 Souvent de simples préférences non motivées (et non justifiables) de certains intervenants.
6 *Stake holders*.
7 Les ingénieurs du logiciel, dans les difficultés de classification de leur propre domaine, pourraient tirer profit de la connaissance des difficultés que, dans la longue histoire des classifications animales, on a eues à classer l'ornithorynque, cet animal qui a bien des ressemblances, cachées bien sûr, avec le génie logiciel.
8 Ce « tâche » est sans doute l'« activité » de ISO 12207.
9 Il est tellement clair que toute séparation qui se veut nette entre des activités humaines est artificielle qu'il est inutile de le souligner. Pourquoi donc, dans le GL, souligne-t-on si souvent quelque chose de si évident ? Sans doute parce que dans les deux premières décennies du GL, on a créé des séparations étanches très artificielles pour contrer une approche naïve qui voyait le développement du logiciel comme une nébuleuse où il était inutile de différencier les activités et les documents.
10 Il est courant dans un projet que des leaders imposent leurs vues en écrasant, du haut de leurs compétence et de leur charisme, des idées qui n'ont pas de champions assez entraînés. On dira : « C'est normal, c'est ça la vie » ; on est d'accord. Normal mais avec des conséquences très négatives pour l'élicitation des exigences qui ont besoin de « démocratie » au moins au niveau du discours. Ce n'est donc pas un hasard si, souvent, les personnalités très fortes, si utiles pour bien réaliser une première version d'un produit, deviennent un frein et un poids quand les temps longs et fondamentaux de la maintenance perfective prennent le dessus.
11 Ce que dans les normes IEEE on appelle *Concept of Operation*.
12 À noter que l'on ne parle pas des spécifications mais, de façon plus générique, de document.
13 L'observation que l'on peut valider les exigences par rapport aux besoins, si les besoins sont consignés dans un document, ne fait que déplacer le problème en amont.
14 Ce qui crée éventuellement de possibles conflits sémantiques avec vérification.
15 Ce qui est exactement le contraire de ce que l'on dit normalement : c'est-à-dire que la maintenance est un développement.
16 Si les mythes ont mauvaise presse dans un monde étroitement scientiste, c'est parce que

l'on ignore la fonction fondatrice des mythes dans notre civilisation (dans toutes les civilisations). Les mythes actuels du GL sont eux aussi une nécessité pour pouvoir un jour passer à... d'autres mythes.

17 Concept of Operation : principes d'opération.

18 Le choix de l'épistémologie de Lakatos avec son « rationalisme » parfois excessif nous semble idéal en ces temps anarchiques où l'on croit régler tous les problèmes à l'aide d'une automatisation sauvage. Cette anarchie devrait faire craindre aux personnes très peu sensibles aux problèmes ouverts et résolus par l'automatisation un retour de balancier où l'on cherchera à diaboliser l'automatisation comme source de tous les maux. Il vaut donc la peine, au moins quand on se pique de réaliser des machines et d'écrire des articles, de limiter une liberté qui pousse trop souvent à la construction de machines de piètre qualité et à la rédaction d'articles exsangues.

19 Nous ne parlons pas nécessairement de processus formel, mais d'un ensemble d'activités nommées et reconnues par les parties prenantes.

20 Sans se réduire à des analyses écrites sur des serviettes de restaurant comme c'était souvent le cas dans les optimistes et inconscientes années 1970.