Prolégomènes à une critique du génie logiciel

Partie 1 : Contextualisation

par Ivan Maffezzini, Alice Premiana et Bernardo Ventimiglia

Avec le temps, le plaisir d'avoir raison tout seul moisit.
E. W. Dijkstra, *On the role of scientific thought*

Ce livre ne sera peut-être compris que par qui aura déjà pensé lui-même les pensées qui s'y trouvent exposées – ou du moins des pensées semblables.
L. Wittgenstein, *Tractatus logico-philosophicus.*

Résumé : Le présent article est le premier d'une série de quatre où il est tenté 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. Après une mise au point terminologique et une description de ce que les auteurs appellent la *chaîne d'automatisation*, sont énoncés les deux principes qui sont selon les auteurs à la base de l'automatisation. Suivent une courte histoire du génie logiciel, des principes additionnels et une synthèse de l'organisation du génie logiciel tirée de SWEBOK et des normes d'IEEE.

1. INTRODUCTION

Dans cette série de quatre articles nous souhaitons 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. Nous n'avons pas la naïveté de vouloir *démontrer* mais nous espérons pouvoir *montrer* que la condition *sine qua non* pour que le génie logiciel soit bâti sur des concepts auxquels on puisse appliquer des méthodes rigoureuses et, éventuellement, des mesures, c'est que le génie logiciel renonce à certaines de ses parties périphériques par rapport à la construction des logiciels mais centrales par rapport aux utilisateurs.

Nous sommes conscients que ce n'est pas parce qu'on décide, de façon volontariste, que certains processus ne font plus partie du génie logiciel ou parce qu'on décide de changer le nom d'un ensemble d'activités qu'on améliore les produits fondés sur les ordinateurs. Mais il est dangereux de sous-évaluer l'importance de nommer et d'ignorer que le changement de contenant peut donner une valeur nouvelle au contenu.

Une critique de notre approche, à laquelle on ne peut pas nier une certaine efficacité psychologique, pourrait s'énoncer ainsi : « *Comment pouvez-vous penser que votre approche est la* bonne *quand la majorité des gens qui travaillent ou font de la recherche dans les domaines connexes aux ordinateurs pensent d'une tout autre manière ?* » Réponse possible : « Si le génie logiciel se situe quelque part du côté de la technique et de la science, le nombre d'individus qui pensent d'une certaine manière n'est pas nécessairement un élément qui rend une position plus digne de confiance. Et encore, l'histoire n'a cesse de montrer qu'en situation de crise – et les tenants du génie logiciel nous parlent de crise depuis trente ans –, c'est exactement l'inverse qui est vrai ». Il s'agit là d'une réponse facile, et nous ne la donnerons pas. Par contre, il semble que des signes d'une autre manière de penser l'automatisation commencent à surgir (pensons surtout aux livres de M. Jackson, de T. Winograd, de M. Sewell et d'A. Cooper). Même si ces signes font espérer en un futur plus rose, ils n'auraient pas suffit pour nous lancer dans cette aventure si les énormes travaux qui ont permis de réaliser SWEBOK n'avaient systématisé et nommé une matière qui, avant ces travaux, était trop informe même pour pouvoir être critiquée de manière efficace.

Nous avons construit cette série d'articles autour de *principes* et de *corollaires*. Les principes sont les notions de base à partir desquelles l'argumentation est construite. Ils sont plus que des axiomes

(ils ne sont pas interchangeables, ni complètement arbitraires) mais comme les axiomes ne sont pas démontrables : ils sont un distillé d'expérience[1]. Deux méta-principes très simples règlent la production des principes :

- un principe ne doit pas en contredire un autre ;

- les principes ne doivent pas être inutilement multipliés (rasoir d'Occam).

Un corollaire est une conséquence directe d'un principe. Un principe peut découler d'un autre principe mais, pour les principes, il existe des éléments contingents qui rendent leur rapport moins direct que celui qui existe entre un principe et ses corollaires.

Le premier des quatre articles propose un lecture du contexte historique pour en dégager les principes qui se trouvent à la base de nos propositions ; le deuxième analyse les interactions entre les domaines concernés par l'automatisation ; le troisième aborde plus en détail le génie des exigences et la qualité ; le quatrième présente quelques propositions constructives pour une nouvelle approche et une bibliographie commentée.

NOTE : Un texte qui veut réfléchir en profondeur sur le génie logiciel doit tout à tous ceux qui ont écrit ou produit dans le domaine et qu'on a plus ou moins virtuellement côtoyé. Mais, pour ne pas alourdir le texte d'un nombre trop élevé de références et de citations, nous assumons personnellement tout ce que nous affirmons et nous ne citerons que les normes qui sont à la base de l'article et quelques livres concernant des éléments très ponctuels. La bibliographie du quatrième article sera, par contre, commentée.

2. CHAÎNE D'AUTOMATISATION ET LANGAGE

Dans cette section, il s'agira de présenter succinctement les quelques réflexions sur l'automatisation et le langage qui sont à la base de nos considérations sur le génie logiciel.

2.1. Mise au point terminologique

Étant donné que notre ambition est de mettre un certain ordre dans la terminologie de la nébuleuse qui entoure l'ordinateur, il faut commencer par mettre de l'ordre dans notre propre terminologie.

Langue et langage. On emploiera ici *langue* et *langage* dans l'acception saussurienne que l'on peut, de façon très concise, exprimer ainsi : le langage englobe la langue (système de signes social) et la parole (élocution individuelle avec ses composantes phonétiques et psychologiques). La traduction en anglais (la langue dans laquelle on écrit le plus sur le génie logiciel) peut aider à éclaircir la différence entre langue et langage que les informaticiens semblent ignorer. Selon les notes de Tullio di Mauro dans l'édition critique du *Cours de linguistique générale* de Saussure : *langue* devrait être traduit par *language*, *langage* par *speech* et *parole* par *speaking*. Le langage naturel, du point de vue de l'automatisation, nous intéresse en tant que couples signifiant/signifié historiquement déterminées dont certains signifiés ont des référents dans la « réalité physique ». C'est-à-dire que, dans l'automatisation, les « jeux de langage » ont une contrepartie hors langage qui fonctionne au moins comme garde-fou : le référent est réel dans le sens qui plie les jeux de langage à des règles qui lui sont externes. En d'autres termes : le réseau conceptuel qui a présidé à la mise en œuvre de l'automatisation est confronté à un réseau physique (de référents) qui a le pouvoir de l'invalider.

Machine. Une *machine* est 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.

Automatisation. Nous employons *automatisation* (et ses dérivés) dans le sens d'« emploi de machines pour transformer des données reçues de l'extérieur et exécuter des actions autonomes ayant un impact sur le monde » et lui redonnons ainsi sa signification que l'emploi du terme dans le contrôle des procédés a fait quelque peu oublier[2]. Même s'il s'agira surtout d'automatisation réalisée avec les ordinateurs, une définition plus générale permet plus facilement de caractériser le logiciel.

À titre de contre-exemple, nous ne considérons pas comme des machines qui automatisent (automates) :

- l'eau qui fait tourner la roue d'un moulin ou l'ouvrier à la chaîne de montage qui fait des actes machiniques, parce qu'ils ne sont *pas* des artefacts ;

- un livre, car, tout en étant un artefact qui a souvent le but de transformer le monde, il manque d'autonomie ;

- une voiture en tant que *voiture,* car, tout en étant une machine, elle ne transforme pas des données mais de l'énergie.

À titre d'exemple nous considérons comme des automates :

- l'équipement de contrôle de vitesse d'une voiture ;

- un programme pour le calcul des taxes.

Monde. Le monde est la réalité physique enveloppée par le langage naturel.

Domaine. Le domaine est une partie du monde, un point de vue langagier sur la réalité. La relation entre monde et domaine est de type homéomère avec une invariance qui n'est jamais totale : c'est-à-dire qu'un domaine a les mêmes caractéristiques qu'un monde et ne peut jamais être complètement séparé du tout. Puisqu'un domaine implique le langage comme élément d'observation, il s'agit toujours d'un domaine de connaissance.

Exemple : un atome n'est pas un domaine, la physique atomique l'est.

NOTE : à cause de leur ambiguïté et des mauvaises compréhensions qu'ils ont engendrées dans la brève histoire de l'automatisation assistée par ordinateur, nous tâcherons d'éviter l'emploi des termes *système*, *symbole et problème*.

2.2. Automatisation

La figure suivante montre le *monde* avec à l'intérieur un schéma simplifié du processus d'automatisation lorsque l'automatisation est réalisée à l'aide d'ordinateurs.

Les flèches pleines indiquent des transferts entre des machines ou entre des machines et un domaine. Les flèches en pointillé indiquent des transferts réalisés avec l'intervention des humains surtout, mais pas seulement, en tant qu'êtres dotés de langage.

La couleur grise a été employée pour caractériser les éléments du monde qui interagissent sans les humains (et donc sans langage).

Dans la figure 2.1 la Description en langue naturelle (DLN) est au centre de deux transferts qui peuvent être réalisés seulement grâce à des échanges entre humains. Même lorsque le DLN et les programmes sont réalisés par une seule personne, il y a des échanges. Une personne, même quand elle parle entre soi et soi, est dans une situation sociale et donc d'échange, car le langage naturel est « social » par définition.

L'échange entre humains est un processus qui est à la base de deux transferts :

- Passage du Domaine à la DLN.

- Passage de la DLN à l'ordinateur.

NOTE : même quand une description dans une langue formelle est présente, on ne peut pas se débarrasser de la DLN, car les humains ont toujours besoin du langage naturel pour comprendre une langue formelle.

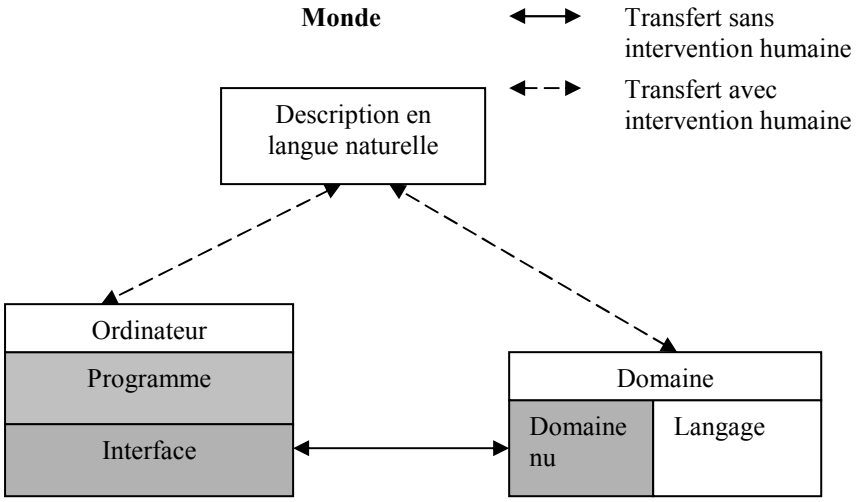

Figure 2.1 : Échange entre ordinateur et domaine I

2.3. Du domaine à la DLN

Généralement, en génie logiciel, on parle des « problèmes » qui déclenchent le processus d'automatisation. Nous préférons penser qu'il n'existe pas de problèmes déclencheurs mais que, du domaine, surgissent des exigences d'automatisation qui non seulement peuvent ne pas répondre à un problème mais qui, éventuellement, en créent. Le mot « problème » est un bon exemple de l'influence des mots sur l'approche de l'automatisation : parler de problème et éventuellement de résolution de problème nous met d'emblée dans une position où le problème existe quelque part à l'extérieur et les humains ont la tâche, l'obligation ou l'envie de le résoudre. Pour nous, la position de l'humain à l'extérieur du problème est une simplification excessive et inutile au sens où l'automatisation ne se résume pas à répondre à un problème. Souvent c'est l'automatisation elle-même (une première phase[3]) qui crée les conditions pour de nouvelles automatisations, conditions qui créent un besoin et donc un problème.

Une DLN contient deux types d'informations principales :

- *Les caractéristiques du domaine*. Elles sont constituées des concepts et des associations entre les concepts que la machine ne pourra pas modifier et auxquelles elle doit s'adapter. Par exemple : F = ma, dans la mécanique classique est une relation entre trois concepts que, lors de l'automatisation, on ne peut pas modifier.

- *Les contrôles sur le domaine*. Il s'agit de ce qu'on veut imposer au domaine ; des exigences et des contraintes que l'on veut fixer pour le domaine. C'est ce pour quoi la machine est créée. Dans l'exemple précédent une exigence pourrait être d'empêcher que F soit supérieure à une limite fixée. Que cela soit possible ou non ne dépend pas de la loi de Newton mais d'autres caractéristiques du domaine.

Même si différencier les *caractéristiques* du domaine des *contrôles* sur le domaine est très important dans la pratique, les deux types de description ne sont pas toujours facilement différenciables, comme dans l'exemple précédent. Ceci est dû au fait qu'un domaine un tant soit peu complexe peut changer par le simple fait qu'on l'étudie pour le décrire, ce qui est fort naturel puisqu'un domaine est par définition un domaine de connaissance. Donc, dans la figure, la flèche qui relie le domaine à la DLN n'est pas bidirectionnelle seulement à cause des contrôles qu'on veut imposer au domaine mais aussi parce que le domaine (ses caractéristiques conceptuelles) peut changer au cours de l'étude et, éventuellement, à cause d'elle.

Puisque les humains (surtout les humains !) sont dans le monde, ils interagissent entre eux à l'aide du langage, mais eux aussi ont une « réalité nue » qui s'offre à la perception. Les méthodes, les enjeux, les contraintes, les caractéristiques individuelles, les désirs, les paradigmes cachés, les intérêts, etc. ont donc un impact sur la vision du domaine et sur la description qui en résulte. Les

humains en tant qu'humains sont dans l'impossibilité de n'échanger que sur le domaine et de cette impossibilité naissent les paroles qui modifient la perception de celui-ci[4].

2.3.1. Le domaine

Même si, pour les humains, un domaine est une unité inextricable[5], de langage et de « matière muette », dans la représentation de la figure 2.1 les deux éléments sont nettement séparés. Cette séparation rend difficile l'opération de nommer ce qui reste dans le domaine quand on lui ôte la langue : la *nature*, le *monde physique*, la *réalité physique,* la *matière…* tous ces noms sont chargés de trop de signification pour ne pas causer de faciles incompréhensions. Pour souligner le fait que le langage permet l'entrée du domaine dans le social, nous appelons le domaine sans l'« enveloppe » de la langue *Domaine nu*. Cette difficulté de trouver un nom est un indice de l'unité « forte » entre langage et *domaine nu*. Pourquoi donc séparer ce qui ne semble pas être séparable ? Parce que cette séparation, simpliste d'un point de vue théorique, permet de représenter les interactions de l'ordinateur avec le monde comme des interactions d'une tout autre nature que celles des humains. L'ordinateur, dans notre schéma, interagit avec le domaine nu via un dispositif qui, lui non plus, n'est pas dans le langage.

La langue du langage du domaine est une spécialisation de la langue naturelle et son étendue dépend des caractéristiques du domaine. Le fait d'être une spécialisation n'implique pas que la langue naturelle dans son entièreté ne soit pas présente lorsque que les humains échangent à propos du domaine.

Pour simplifier la figure, nous n'avons pas représenté les liens entre les domaines, mais il est évident que les domaines se conditionnent réciproquement créant ainsi d'énormes zones grises, qu'on pourrait appeler des *no domain lands*, et que les frontières tracées pour faciliter la compréhension sont souvent très arbitraires et rendent artificiellement claires les zones de superposition des domaines. Créer des frontières nettes et appauvrir ainsi les domaines est une exigence de la science/technique pour rendre le langage opératoire.

Même si, théoriquement, un domaine est influencé par tous les autres (ne fût-ce qu'à cause du partage du langage quotidien), en pratique les relations inter-domaines importantes dépassent rarement le nombre de 7 ou 8[6]. Voir à ce propos les relations entre le génie logiciel et les domaines connexes à la section 4.

Le nombre de domaines, même d'une petite partie du monde, est théoriquement illimité parce que les domaines sont continuellement créés, modifiés et détruits. Le rythme de création, modification et destruction dépend des caractéristiques du domaine lui-même et de l'évolution de la technique et des connaissances. Il y a aussi des phénomènes culturels et historiques qui influencent l'organisation du monde en domaines. Si on prend à titre d'exemple la physique, il suffit de penser à un sous-domaine (ou domaine ?) comme la mécanique quantique qui, inexistant il y a un siècle, a aujourd'hui une importance énorme, et cela pas seulement en physique.

2.3.2. Description en langue naturelle

La DLN, même avant de considérer les exigences, contraint les caractéristiques du domaine dans un réseau conceptuel qui, en simplifiant les liens entre les concepts et en diminuant leur nombre, crée un appauvrissement sémantique. Cet appauvrissement sémantique est la condition *sine qua non* de l'automatisation. La description apporte, comme contrepartie de l'appauvrissement sémantique, une augmentation de la précision. Une description, même mal faite, est un moyen de diminuer l'ambiguïté en donnant, au minimum, un angle d'approche. Dire que plus la description est non ambiguë et plus l'appauvrissement est grand ne peut sembler paradoxal que si l'on oublie que la richesse sémantique du langue naturelle va main dans la main avec l'ambiguïté[7].

La simplification du domaine effectuée par la description est parfois ce qui fait que le domaine est un domaine : ceci est particulièrement évident dans le cas des domaines « normalisés ».

La description contient souvent des graphiques qui aident à synthétiser certaines relations conceptuelles. Lorsque la description est accompagnée par des prototypes de l'interface personne-machine, le prototype devrait être considéré comme une exemple qui aide à comprendre et non comme une description et ceci même quand le prototype sera intégré sans modification dans le produit final.

2.4. Du DLN à l'ordinateur

Si la description du domaine en langue naturelle réduit le domaine à un nombre fini de phrases et à des graphiques, le passage à une langue[8] compréhensible par l'ordinateur à cause de la transformation de la langue naturelle en une suite de 010101 interprétable par une unité centrale, implique le franchissement d'un gouffre et c'est sur ce gouffre, et sur ce gouffre seulement, que le génie logiciel peut aider à jeter un pont. La largeur du gouffre est déterminée, d'une part, par le niveau de précision de la DLN et, d'autre part, par la richesse sémantique de la langue[9] que la machine ordinateur peut interpréter (langue cible). Les langues formelles constituent un moyen afin d'augmenter la richesse sémantique de la langue côté machine.

Une manière de rendre le gouffre encore plus étroit, c'est d'employer des langues formelles non universelles, c'est-à-dire des langues qui n'aspirent pas à exprimer des concepts valables pour tous les domaines, mais qui se lient fortement à un domaine donné. C'est-à-dire des langues qui insèrent des caractéristiques du domaine dans des briques (ou des composantes) fonctionnellement riches par rapport au domaine à automatiser. Mais quelle que soit la richesse de la langue cible, il y aura toujours une partie du chemin qui a besoin de l'humain.

2.4.1. Ordinateur

L'ordinateur, à ce stade-ci, nous intéresse surtout comme machine qui contient des programmes (qui sont eux-mêmes des machines) et des interfaces : les interfaces étant la *réalité nue* de l'ordinateur qui agit sur la *réalité nue* du domaine. Que l'interface soit une carte programmable ne change rien à nos considérations : pour nous, l'interface est la partie physique qui échange des données, d'une part, avec les programmes et, de l'autre, avec l'extérieur de l'ordinateur. Par exemple : pour une ligne de communication branchée à l'ordinateur, l'interface est l'ensemble des circuits et des composants passifs (mécaniques et électriques) qui permettent de générer des tensions et des courant sur les broches où l'on attache le connecteur. Que l'interface ait aussi des caractéristiques fonctionnelles (par exemple une valeur de tension qui passe de +12 Volts à 0 Volt indique que l'ordinateur est dans l'impossibilité de recevoir) est secondaire par rapport à nos considérations sur les interfaces.

L'écran d'un PC n'est pas une interface dans l'acception que nous avons choisi d'employer : l'écran est un dispositif qui relie l'interface (la carte interne qui gère l'écran) au domaine externe constitué d'un milieu qui laisse passer la lumière – et pas les humains ! Même si l'écran, avec ses sorties, est conçu pour des humains, l'ordinateur peut continuer à afficher des données sans se soucier de la présence[10] d'êtres qui sont dans le langage.

Il est sans doute plus réaliste d'introduire, comme à la figure 2.2, un dispositif entre les interfaces de l'ordinateur et le domaine nu pour mieux caractériser notre définition des interfaces. Il serait possible de considérer le dispositif comme faisant partie du domaine mais, étant donné que, souvent, les dispositifs couvrent plusieurs domaines, nous avons préféré le sortir. Cette dernière observation met en évidence ce qui était implicite dans toute notre présentation, c'est-à-dire que les domaines sont des tranches verticales qui s'élèvent du langage naturel pour atteindre un détail fonctionnel. La figure 2.3 donne une vision schématique de ce que nous venons de dire.

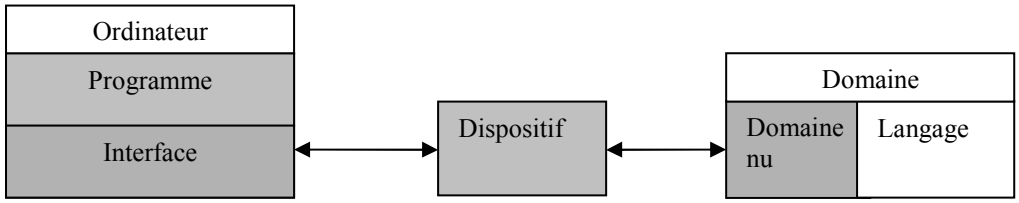

Figure 2.2 : Échange entre ordinateur et domaine II

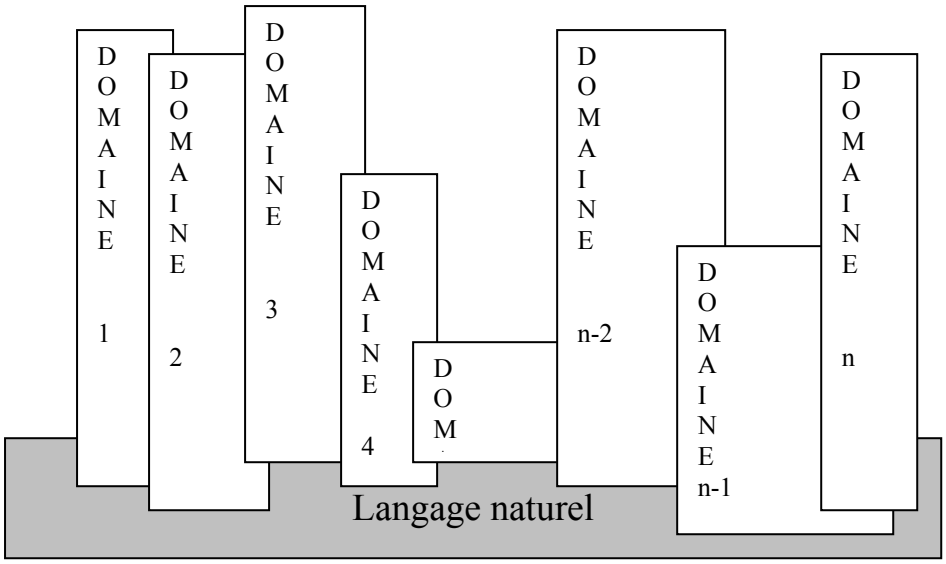

Figure 2.3 : Domaines verticaux

Dans la figure nous avons essayé de montrer que les domaines sont plus ou moins « enfoncés » dans le langage et plus ou moins superposés. Nous n'avons pas estimé nécessaire de montrer des domaines horizontaux formant une couche entre le langage et les domaines verticaux, car nous croyons qu'aucun domaine ne peut prétendre avoir l'étendue du langage naturel.

Nous sommes loin de croire, comme les naïfs[11] des sciences cognitives, que le cerveau soit un système[12] traitant des éléments physiques dotés d'une signification et que dans l'ordinateur c'est le programmeur qui crée le lien entre la suite de 0 et de 1, et la signification – dans le sens que la signification est obtenue à partir du sous-ensemble des manipulations possibles de 2^n états où n est le nombre de bits. Ce qui, même en ignorant les nouveaux états qui peuvent être acquis sur un disque ou sur tout autre support externe, pour un ordinateur avec 128 M octets de mémoire donne $2^{1\,024\,000\,000}$ états possibles – valeur qui donne le vertige. La possibilité de créer des sous-états significatifs et surtout d'y parvenir de manière contrôlée dépend du fait que l'unité centrale (la partie de l'ordinateur qui en lisant des états et en les interprétant décide quels sont les nouveaux états à lire) a intégré des règles assez simples. Ce qu'en informatique on appelle langues de haut niveau (les langues qui permettent une représentation avec des chaînes de caractères des sous-états possibles et des séquences de travail de l'unité centrale) n'est important que du point de vue pratique[13], dans le sens que les humains seraient incapables de modifier et de comprendre des programmes un tant soit peu complexes sans leur aide.

Avant de conclure ce chapitre, nous tâcherons d'éclaircir les concepts que nous venons de présenter plus ou moins maladroitement avec un exemple tiré de notre travail quotidien.

2.5. Exemple : aube

Le signifiant « aube » (c'est-à-dire la chaîne de caractères a u b e) est associé, dans la tête de pratiquement la totalité des francophones adultes, au signifié (ou concept) *la première lueur du soleil*. Pour une grande majorité, « aube » est aussi associé à *palette d'une roue hydraulique*, et une petite minorité, quand elle entend « aube », peut penser à *un vêtement ecclésiastique*. Pour un francophone qui connaît bien le français il y a donc trois concepts auxquels renvoie le signifiant « aube ». Lorsque un auditeur/lecteur entend/lit « aube », il peut espérer associer à « aube » la bonne aube en fonction du contexte. Il s'agit d'un simple espoir parce que même dans un cas où les signifiés sont très éloignés l'un de l'autre, comme dans l'exemple d'aube, les possibilités d'ambiguïté sont facilement envisageables[14].

Si l'on doit automatiser une partie du fonctionnement d'une centrale hydroélectrique et, en particulier, si l'on doit contrôler le débit d'eau en fonction de la puissance électrique demandée, il est clair que lorsqu'on trouve « aube » dans un document de projet on pense aux palettes d'une turbine et non au vêtement en lin blanc que les prêtres portent sous la chasuble. Dans un tel cas, la désambiguïsation par le contexte (on pourrait parler de macro contexte) est très facile. Le référent du signifié associé à « aube » est un élément métallique qui, en fonction de la surface et de l'angle, change la relation entre l'énergie cinétique de l'eau et l'énergie mécanique que la turbine transmet au rotor de l'alternateur. Si l'alternateur est branché au réseau, pour une turbine donnée (turbine étant un référent complexe qui englobe les aubes), en ouvrant plus ou moins les vannes, on contrôle la puissance injectée dans le réseau. La figure suivante présente les interactions simplifiées entre les objets du monde réel (les référents) pour les aubes d'une centrale hydro-électrique.

Dans le dessin, les boîtes ne représentent pas de concepts mais des objets ou référents et les noms des boîtes ne sont pas des noms communs, comme il se doit pour des concepts, mais des noms propres : *Barrage Manic* et non *Barrage.* Sur la terre, à notre connaissance, il existe un seul *Barrage Manic*. L'eau[15]en sortant du barrage, transite par la *Vanne no 03*[16]et tombe sur les *aubes 01, 02…* de la *turbine no7* qui, via l'*axe 7-5,* fait tourner l'alternateur *A05* qui alimente le transformateur *T01*. L'automate *Auto-01* lit les mesures de puissance du *wattmètre 01* et agit sur le *moteur no 1* qui agit sur la *vanne no 3* et… la boucle est bouclée.

Mais que signifie que la boucle soit bouclée ? Que tout fonctionne sans besoin des humains parce qu'un lien fonctionnel a été créé entre la quantité d'eau et la puissance électrique injectée dans les lignes de transport. Ce lien a été possible parce que l'eau dans le barrage, lors de la conceptualisation, n'a été considérée que comme énergie potentielle et l'eau qui frappe les aubes comme énergie cinétique. La boucle a pu être bouclée parce qu'il y a eu l'intervention de l'homme qui, entre autres, a été capable d'abstraire à partir de toutes les caractéristiques de l'eau, celles qui pouvaient avoir une influence sur le but ultime[17]qui est celui de produire de l'énergie électrique.

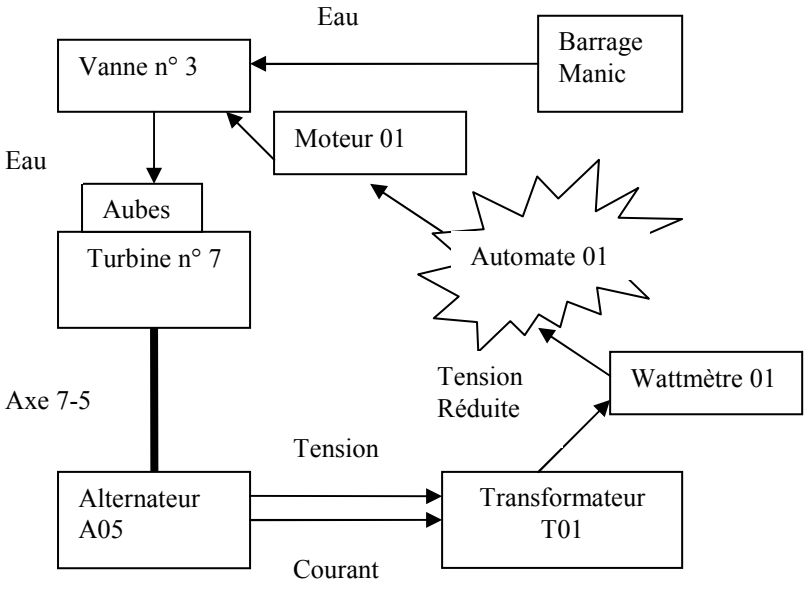

Figure 2.4 La lumière des aubes.

Dans la boucle toute référence au langage a disparu parce que l'automate a intégré la partie nécessaire au fonctionnement établi dans le DLN.

L'automate 01 a une interface vers le moteur pour commander l'ouverture ou la fermeture de vanne et une vers le wattmètre pour lire la puissance générée : le moteur et le wattmètre sont les parties du domaine nu des centrales sur lesquelles l'automate agit. À moins d'entendre information dans un sens si vaste qu'il ne veuille plus rien dire, l'automate n'envoie pas des informations au moteur mais des signaux physiques qui, pour nous, qui avons décrit le domaine, et pour nous seulement, ont une

signification. Que l'alternateur génère de l'énergie en fonction des besoins implique bien sûr que la boucle s'ouvre pour permettre aux humains de changer les consignes mais, même dans ce cas, les consignes font partie du *domaine nu*[18] **:** elles ont été réduites à des signaux provenant d'un clavier, par exemple.

2.6. Les principes fondamentaux

Nous appelons *principes fondamentaux de l'automatisation* les deux principes qui suivent parce que ce sont eux qui ont les impacts les plus importants sur notre démarche et parce que leur non acceptation implique l'affaissement du petit édifice que nous sommes en train de construire.

> *Tout domaine contient au moins une partie automatisable.* (**P01**)

Ce principe dit que quelle que soit la façon de diviser le monde, quelle que soit la dimension du domaine, il contiendra toujours des éléments qui peuvent être mis en relation avec des éléments d'une machine, et cela de manière telle que le « tout » fonctionne avec un certain degré d'autonomie.

Si un domaine est constitué de sous-domaines, le principe P01 s'applique aux sous-domaines.

P01 n'implique pas qu'il ne soit pas possible de trouver un domaine qui n'ait pas encore été automatisé. Ce que, par contre, il implique c'est que si l'on démontre qu'un « domaine » n'a aucune partie automatisable, alors il n'en est pas un. Par exemple : un atome d'hydrogène « en soi[19] » n'est pas automatisable (seulement réalité physique) mais le domaine hyper-spécialisé que nous créons en même temps que nous le nommons « étude des atomes d'hydrogène dans la molécule d'eau à 4 degrés Celsius dans les lacs américains » contient des parties informatisables.

Nous appelons P01 principe d'ouverture.

Ce principe nous permet, par exemple, de comprendre pourquoi les responsables du marketing de toutes les compagnies d'informatique proposent des « applications » là où jusqu'à il n'y a pas si longtemps seuls les illuminés en auraient imaginé.

Le principe suivant limite l'ouverture de P01 en indiquant qu'on ne peut pas tout automatiser. Nous appelons P02 principe de fermeture ou de réalité.

> *Une partie d'un domaine est complètement automatisable si, et seulement si, ses frontières sont formalisables.* (**P02**)

C'est-à-dire que pour la partie concernée la langue du domaine doit pouvoir être réduite à une langue formelle. Un domaine complètement automatisable est un domaine dégénéré, au sens où la langue du domaine a été réduite à une langue formelle.

Le deuxième principe, en tant que principe de réalité, est un garde-fou pour les experts du domaine qui nourrissent de trop grands espoirs par rapport à l'automatisation.

P01 et P02 permettent d'aborder le problème de la formalisation des spécifications et du passage d'une description en langue naturelle à une description en un langage formel ou en un langage de programmation.

À ces deux principes, nous ajoutons un principe que l'on doit à A. Tannenbaum et qui est trop souvent oublié par ceux qui s'intéressent au logiciel : *matériel et logiciel sont fonctionnellement interchangeables*. Ce principe, avec le théorème de Nyquist sur l'échantillonnage des signaux, nous permet de considérer (à moins d'indication contraire) les ordinateurs programmés comme des machines qui peuvent (théoriquement) automatiser tout ce qui est automatisable.

Ces deux principes ont de grandes conséquences dans la façon de voir, de décrire et éventuellement de contrôler l'évolution de l'automatisation et donc l'informatisation. Poser le principe P01 (*Tout domaine…*) nous ouvre la porte à l'automatisation de n'importe quelle partie du monde, mais en ajoutant que l'automatisation ne peut être complète que si « ses frontières sont formalisables » (P02)», on souligne le fait que seulement si une partie du monde a été réduite à des expressions formelles on peut l'automatiser complètement.

3. GÉNIE LOGICIEL : HISTORIQUE

Ce bref aperçu historique nous permet d'introduire trois autres principes avant de faire une présentation concise du génie logiciel tel que perçu par les théoriciens et les praticiens aujourd'hui.

3.1. Années cinquante

Dans les années cinquante, l'excitation pour la nouveauté et la flexibilité qui permet de faire faire n'importe quoi, ou presque, à un ordinateur donne aux mathématiciens et aux ingénieurs l'illusion que l'automatisation du monde sera un jeu d'enfant. Mais mathématiciens et ingénieurs sont très mal placés pour comprendre les problèmes qu'engendrera cette machine qui semble se contenter d'ingurgiter des 0 et des 1. Ils sont mal placés parce que les mathématiciens se sentent très peu concernés par les applications pratiques et les ingénieurs sont intéressés surtout par le fonctionnement de la machine. Ils sont surtout mal placés parce qu'en étant dedans ils voient l'arbre et non la forêt. Par contre, les militaires, les banquiers et les gouvernements voient très bien la forêt et comprennent assez vite les potentialités de la nouvelle machine. Ils s'engagent donc, eux aussi, avec grand enthousiasme. Mais, hélas ! Pour le logiciel, la réalité est plus dure que prévu. Dès qu'on veut bâtir des produits pour des usages autres que la recherche en laboratoire, les douleurs commencent à se faire sentir : fonctions mal exécutées, coûts excessifs, retards, faible disponibilité, etc., toutes les maladies qu'on n'a pas encore réussi à éliminer, qu'on n'éliminera jamais totalement et, surtout, que les approches actuelles risquent d'aggraver.

C'est dans les années cinquante que naissent les premières langues de programmation de haut niveau : plus ou moins orientées vers les sciences et la technique (Fortran) ou les affaires (Cobol), elles auraient dû augmenter énormément la productivité. Et, effectivement, elles ont permis de coder plus rapidement parce qu'elles ont permis de créer des états de la machine plus facilement compréhensibles par les humains. Mais coder rapidement n'a pas donné l'augmentation de productivité escomptée. Devant ce clivage entre espoirs et résultats, et en raison de l'expérience de programmation très courte, les mathématiciens se laissent attirer par les formalismes et les rêves de l'écriture automatique du code et les ingénieurs par la possibilité d'écrire/tester rapidement un grand nombre d'instructions que l'unité centrale refusait très souvent d'exécuter tel que désiré.

Quelque chose de pourri commence à poindre au royaume de l'ordinateur.

3.2. Années soixante

Dans les années soixante, le plus grand consommateur de logiciel (le ministère de la défense américain, DoD) décide qu'il faut aborder plus systématiquement la production du logiciel employé dans ses machines. Il faut être systématique dans la production pour obtenir un produit de qualité à des coûts raisonnables, il faut du… génie. Le génie logiciel naît parce qu'on a besoin d'une certaine assurance que, quand on achète un produit – si on a la chance de le recevoir ! –, il s'agit d'un produit de qualité qui exécute les fonctions demandées comme dans l'ingénierie « normale ». Le génie logiciel naît donc comme un mécanisme pour rassurer les acheteurs. Ce qui n'est pas un mal en soi mais a eu parfois des effets pervers dus à un excès de bureaucratisation dans la production : c'est-à-dire que certains producteurs se sont mis à générer des quantités effarantes de papier pour justifier une qualité qui, parfois, n'en était pas une, sauf sur papier[20]. Entre « pas de papier » et « trop de papier », il n'était pas facile de choisir surtout que les raisons justifiant ou non l'usage du papier étaient elles aussi sur papier !

Dès qu'il naît, étant donnés les enjeux économiques pour les producteurs, le génie logiciel occupe, de la manière la plus naturelle, tout l'espace qu'il trouve devant lui. Veut-on un programme pour contrôler un radar ? Il faut définir ce dont on a besoin, il faut que les développeurs comprennent ce que le client veut, il faut définir une structure de manière à ce que le logiciel se tienne, il faut bien sûr écrire le code, etc. Ce qu'on appellera le cycle de vie classique du logiciel est né. Y a-t-il d'autres possibilités ? Pratiquement pas. On construit un logiciel par analogie avec la construction d'une maison, d'un avion, d'une voiture, selon les approches des industries qui ont au moins quelques dizaines d'années d'avance.

On considère le logiciel comme n'importe quel produit doté d'un cycle de vie allant de l'idée initiale au retrait. Le cycle est divisé en phases pour mieux contrôler la progression des travaux et les phases sont définies avec beaucoup de bon sens en s'inspirant des autres branches du génie :

1- Analyse : compréhension et description du problème.

2- Conception : création d'une structure pour le logiciel.

3- Codage : écriture du code.

4- Test : vérification des résultats par rapport aux attentes.

5- Maintenance : modifications du logiciel pour l'adapter aux changements.

« L'invention » de ce domaine et sa division représentent ce qu'il y a de plus normal pour un acheteur de logiciel. Et semble constituer la solution idéale du point de vue économique pour le producteur aussi. Mais, selon nous, cette origine, pilotée par les acheteurs et fondée sur l'analogie avec les autres branches du génie, est une des causes des difficultés du génie logiciel. Sans doute même la principale. À ce propos, nous énonçons le troisième principe qui découle de P01 :

> *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.* (**P03**)[21]

Non seulement le génie logiciel n'est pas un domaine comme les autres, mais il est un domaine plus complexe, car il n'a pas de limites sinon celles « illimitées » du langage naturel. De P03 et de cette considération découle le corollaire suivant :

> *Toute analogie fondée sur les génies traditionnels crée des modèles erronés et de faux espoirs (***C-P03***)*.

Depuis quelques années, on commence à rencontrer des auteurs qui soulignent la différence entre le génie logiciel et les autres branches du génie. Mais ceux qui, comme M. T. Sewell, introduisent l'analogie avec l'architecture comme la carte gagnante pour la crise du logiciel ne font que répéter l'erreur des années soixante dans le nouveau contexte. L'architecture, comme le génie, peuvent être employés comme des analogies utiles pour des processus internes à l'automatisation, mais une analogie qui couvre le tout dépasse toutes les professions actuelles et peut éventuellement trouver son appui seulement sur quelque chose qui englobe toutes les professions et qu'on appelait autrefois amour du savoir.

À essayer de différencier le génie logiciel des autres génies, certains arrivent même à proférer des considérations à notre avis inacceptables, comme celle qui suit que nous commentons en détail dans le quatrième article : « Dans le génie logiciel, l'approche est inversée [par rapport au génie traditionnel]. On va du concret vers l'abstrait. Le produit logiciel final est la virtualisation (le code) et la transposition invisible d'une conception originelle qui exprime un problème du monde réel[22] »

3.3. Années soixante-dix

Les années soixante-dix sont les « grandes années » de la programmation structurée qui a ses fondations théoriques dans le théorème de Jacopini, théorème qui établit qu'il suffit de trois types d'énoncés (séquence, sélection, itération) pour construire n'importe quel programme. C'est à partir de là que prend origine la célèbre « querelle » entre Dijkstra et Knuth à propos des GOTO : querelle qui oppose deux approches qu'on pourrait taxer de « purisme » (Dijkstra) et de « lisibilisme » (Knuth). Une trentaine d'années après cette controverse et après tant d'articles pour ou contre les formalismes, pour ou contre la « pureté », on peut retenir ce qui est commun aux deux approches : un grand souci de « maintenabilité »[23]. Mais, la maintenabilité – puisqu'on peut faire faire n'importe quoi au logiciel et puisqu'il ne peut pas vieillir[24] – n'est-ce pas la qualité la plus importante si on veut systématiser les interventions sur les produits ? Au fond, Dijkstra et Knuth, ces deux « théoriciens » de l'informatique, étaient en train de montrer qu'il fallait transformer l'informatique en génie. Quel génie ? C'est ça qui est difficile à définir[25].

En partant de ces éléments historiques, on peut donc énoncer un quatrième principe :

> *La* maintenabilité *est la qualité intrinsèque principale pour toute approche d'ingénierie au développement du logiciel.* (**P04**)

qui a comme corollaire :

Là où la maintenabilité *n'est pas importante, on a affaire à de la recherche ou du prototypage*[26]**(C-P04)**

Le principe P04, au lieu de le faire dériver de considérations historiques, peut être déduit d'une des caractéristiques principales du logiciel :

Le logiciel, à cause de la facilité relative de modification, peut être adapté à des situations très différentes.

Cette affirmation ne doit pas être considérée comme équivalente à l'affirmation que l'ordinateur est une machine universelle, car on pourrait imaginer une machine universelle dont les « programmes » sont difficilement modifiables[27].

Il faut souligner que, dans les autres branches du génie, la maintenabilité est moins importante, car ces dernières ne sont concernées pratiquement que par la maintenance *corrective*. Dans le génie logiciel, la maintenance corrective, dans des produits avec un degré acceptable de qualité, est souvent bien moins importante (du point de vue des coûts) que la maintenance *perfective* ou *adaptative*[28]. Élargir ainsi la maintenance rend réaliste le fait de considérer même le premier développement comme une maintenance caractérisée par moins de contraintes fixées par des développements antérieurs[29].

Dans les années soixante-dix, les interactions entre l'informatique et le génie logiciel sont très intenses[30] et leurs frontières sont floues et mouvantes, surtout parce qu'on est en train de fixer leurs frontières à une époque où la « technique liée aux ordinateurs » est encore dans les langes. Et, comme dans toute fixation de frontières, c'est rarement la raison qui l'emporte !

3.4. Années quatre-vingt

Avec la programmation structurée, on a commencé à aborder la programmation comme une activité qui n'est pas la simple écriture d'une suite d'instructions : on demande au programmeur de penser une structure, de… concevoir. Mais on s'aperçoit que le passage de la programmation « anarchique » à la programmation structurée cache un très grave danger pour des systèmes un tant soit peu complexes : le danger de se faire « guider » par l'algorithme et de favoriser ainsi la création d'une cohésion de type procédurale[31]. On met alors au centre les données en introduisant leur flux dans l'analyse. La chanson « les données sont plus stables que les fonctions » a un succès énorme et mérité.

Le passage des « données » aux « concepts » [32] est presque automatique surtout que, dans la programmation, on a introduit des langages qui traitent des « objets » en tant qu'instances d'un concept (ou classe). Il y a de quoi se réjouir : on a trouvé comment décrire le problème et la solution avec le même langage. En ayant des objets pour l'analyste (dans le domaine), des objets pour le concepteur et des objets pour le programmeur (dans le système), il est imaginable qu'au moins la tâche de communication entre les différents intervenants soit simplifiée et puisqu'un des problèmes centraux que le génie logiciel a détecté est la difficulté de communication entre les personnes, il est normal de penser qu'on est proche de la solution idéale.

À un certain moment, tout doit donc être « objet » (comme, dans les années soixante-dix, tout était modulaire) et l'héritage devient la clef pour augmenter la productivité, la fiabilité, pour faciliter les tests, etc. Mais on ne tarde pas à s'apercevoir que ce n'est pas parce qu'on parle d'objets du début à la fin du cycle qu'on règle les problèmes. Souvent les problèmes empirent, car les « objets » de l'analyse rendent le système excessivement lourd et coûteux. On s'aperçoit que la difficulté principale, après l'analyse, est de savoir comment choisir les objets et quelles transformations opérer lors de la conception et de la programmation : les objets du domaine et ceux de la solution ne doivent pas être nécessairement les mêmes. Ce constat, d'une part, montre pour une énième fois qu'il n'y pas de *Silver bullets* et, de l'autre, que même si l'on emploie la même langue dans l'analyse et dans la conception, les « objets » de la conception sont « déformés[33] » pour les adapter à la technologie informatique.

Ce que nous venons de dire ne devrait pas être considéré comme une prise de position contre l'approche objet mais comme une simple réflexion sur le fait que la complexité de l'informatisation ne peut être réduite au-delà d'un certain seuil, quoiqu'on fasse. Ni les approches, ni les paradigmes,

ni les processus ne peuvent rendre la tâche de décrire un domaine « banal » quand on veut aller au-delà de ce qui est déjà informatisé (ou mécanisé). Ceci nous permet d'introduire notre cinquième principe :

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

Ce principe permet de contrer certains enthousiasmes faciles qui sont souvent la cause de mauvaises évaluations. S'il est vrai que l'introduction d'activités qui précèdent la conception et le codage et d'activités de contrôle est une exigence pour la qualité d'un produit, les retombées économiques sont rarement immédiates.

Comme corollaire de ce principe, nous avons que :

> *Une solution simple à quelque chose de complexe n'implique pas nécessairement un coût global faible, car la solution est souvent élaborée à partir d'un travail d'analyse énorme.* (**C-P05**)

Ce principe et ce corollaire considèrent des projets réalisés avec des individus ayant à peu près les mêmes capacités et les mêmes connaissances[34] dans un environnement de même niveau de maturité.

Voici un mauvais mythe que l'approche objet a aidé à solidifier : *l'analyse est un processus qui permet de passer des objets réels du monde aux objets du modèle informatique*. Cette affirmation est fausse, car elle ne tient pas compte du fait que le « monde » est, pour les humains, déjà « modélisé » par le langage. Par le langage qui est notre plus grande richesse sans laquelle aucune informatisation n'est possible[35], mais qui, en même temps, est notre plus grande source d'« erreurs », car non seulement le nombre de modèles langagiers pour un monde donné est infini mais il est aussi clairement impossible de trouver un algorithme pour choisir le meilleur modèle[36]

3.5. Années quatre-vingt-dix

Si les années soixante et les années soixante-dix ont été des années importantes pour la réflexion sur le génie logiciel d'un point de vue mathématique, les années quatre-vingt-dix sont caractérisées par une réflexion plus pratique. Il est pratiquement impossible de donner une description linéaire de l'évolution du génie logiciel pendant cette période. Nous préférons présenter une liste de mots clefs et de lieux communs qui nous semblent très importants même si parfois ils sont un simple support à une pensée magique.

Thèmes

1- Les processus[37] ont une importance centrale et l'amélioration de la qualité passe surtout par leur définition et par leur amélioration.

2- Les phases « classiques » du cycle de vie (analyse, conception, codage et test) restent les phases centrales du processus de développement même si elles se situent dans une spirale plutôt qu'en cascade.

3- Les activités concernant les exigences et les interfaces personne-machine ont droit au statut spécial de « génie » dans le génie logiciel (ingénierie des exigences et de la facilité d'utilisation). Les domaines d'application sont absorbés de manière plus ou moins explicite dans l'ingénierie des exigences tandis que le cycle de vie de la facilité d'utilisation est introduit pour systématiser les interactions entre le développeur et les clients/utilisateurs.

4- La sûreté de fonctionnement regroupe des éléments de qualité auparavant séparés (comme sécurité et disponibilité, par exemple).

5- Les exigences non fonctionnelles sont toujours plus au centre du développement, ce qui a comme corollaire la mise en valeur des développements fondés sur l'architecture.

6- L'importance accrue de la maintenabilité rend nécessaires des environnements avec possibilité de « traçage ».

Éléments technologiques

1- Patrons de conception et cadres deviennent deux piliers de la conception.

2- La réutilisation force à repenser la programmation comme composition.

3- L'accès aux données à distance via Internet a des impacts énormes sur l'architecture.

4- La ré-ingénierie devient toujours plus répandue.

5- Des bases de données objet commerciales sont disponibles.

6- La compatibilité au niveau du « binaire » devient toujours plus importante (DCOM et CORBA).

4. LE GÉNIE LOGICIEL AUJOURD'HUI

4.1. Définitions

Considérons la définition de génie logiciel donnée en ÎEEE Std 610.12-1990 : *L'application d'une approche systématique, maîtrisée, quantifiable au développement, à l'exploitation et à la maintenance du logiciel : c'est-à-dire l'application de l'ingénierie au logiciel*. Dans cette définition le « c'est-à-dire » nous donne implicitement une définition de l'ingénierie (comme une approche systématique, fondée sur des mesures et qui permet de maîtriser les processus) et caractérise le logiciel du point de vue de ses phases (développement, exploitation et maintenance). Pour que cette définition soit d'une utilité quelconque, il faut définir le logiciel. Dans la norme que l'on vient de citer, le logiciel est défini comme : les *programmes, les procédures et la documentation associée et les données qui concernent le fonctionnement d'un système informatique*. Donc, si on unit les deux définitions, on obtient que le génie logiciel est *l'application de l'ingénierie aux programmes, aux procédures, à la documentation et aux données qui permettent de réaliser un système informatique et de l'entretenir tout au long de sa vie utile*. Il serait très naïf de notre part de penser que l'on puisse avoir une définition qui définisse parfaitement le génie logiciel (il suffit de considérer le travail énorme de SWEBOK pour essayer de définir un guide au corpus de connaissances) mais cette définition nous semble une amorce suffisante.

4.2. Contexte

Puisque le génie logiciel, depuis une trentaine d'années, essaye de trouver sa place dans le cadre de l'automatisation à l'aide des ordinateurs, il est important d'en définir le contexte : c'est-à-dire de tracer ses frontières.

Pour caractériser l'état du génie logiciel au début des années 2000, nous avons choisi, à cause de leur généralité et de leur assez grande acceptation, trois normes (ISO12207, IEEE 12207.1, ISO 9126) et le guide pour la définition du corpus des connaissances publié par IEEE (SWEBOK) :

ISO 12027 : cette norme a pour but d'« établir un cadre pour les processus du cycle de vie du logiciel avec une terminologie bien définie à laquelle l'industrie du logiciel peut faire référence ».

IEEE 12207.1 cette norme décrit les données du cycle de vie (CV) (la documentation et le code) dont on a besoin dans une entreprise qui respecte la norme ISO 12207.

ISO 9126 : cette norme « définit six caractéristiques qui décrivent, avec le minimum de recouvrement, la qualité du logiciel ». Les six caractéristiques sont : la capacité fonctionnelle, la fiabilité, la facilité d'utilisation, la rendement, la maintenabilité et la portabilité.

SWEBOK[41] SWEBOK est un projet du *Software Engineering Coordinating Committee* (IEEE) et est « un guide pour le corpus de connaissances qui existe dans la littérature et qui s'est formé dans les derniers trente ans » et il a pour but de « fournir une caractérisation validée par consensus des limites[42] de la discipline de l'ingénierie du logiciel ».

4.2.1. Selon les normes IEEE

Pour être cohérents avec les définitions, nous reprenons la définition du contexte qu'IEEE présente dans la préface à ses normes. Nous avons légèrement modifié la figure originale en supprimant la boîte *Safety* et en la considérant comme une des composantes de la *Dependability*[43] (Sûreté de fonctionnement).

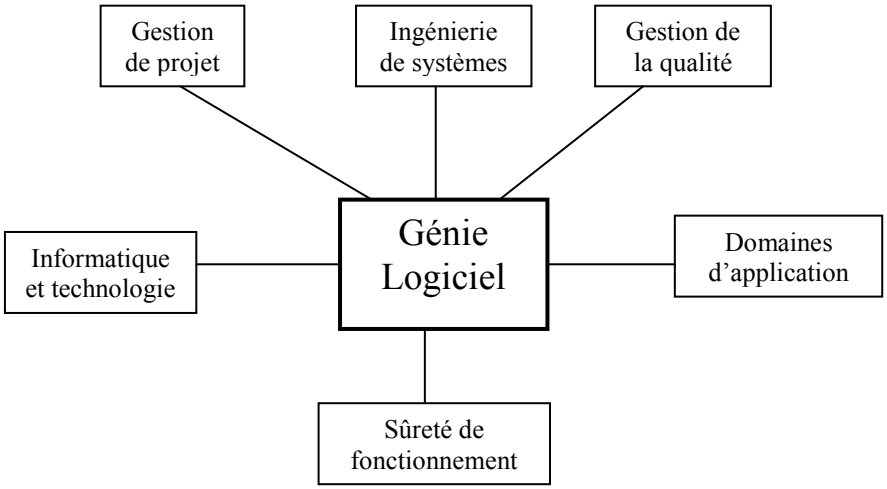

Fig 3.1 : Contexte I : IEEE

Le génie logiciel est délimité par cinq disciplines et les domaines d'application (chaque domaine d'application étant plus ou moins caractérisé par une discipline) :

- la gestion de projet[44] ;
- l'ingénierie de systèmes ;
- la gestion de la qualité ;
- l'informatique et la technologie ;
- la sûreté de fonctionnement (*dependability*) ;
- les domaines d'application.

Cette figure peut être lue comme une description du génie logiciel comme la discipline qui, dans les domaines d'application, en s'appuyant sur des disciplines déjà éprouvées dans des milieux industriels permet de réaliser de manière contrôlée des produits logiciels. Ces disciplines, tout en étant non homogènes, ont bien des éléments en commun. Par exemple, les frontières entre informatique et génie logiciel sont difficiles à tracer[45] ou, encore, pourquoi la sûreté de fonctionnement ne fait-elle pas partie de la qualité ? Nous proposons une réorganisation de la figure qui nous semble, d'une part, mieux représenter le contexte et, de l'autre, nous permettre de mieux appuyer nos critiques :

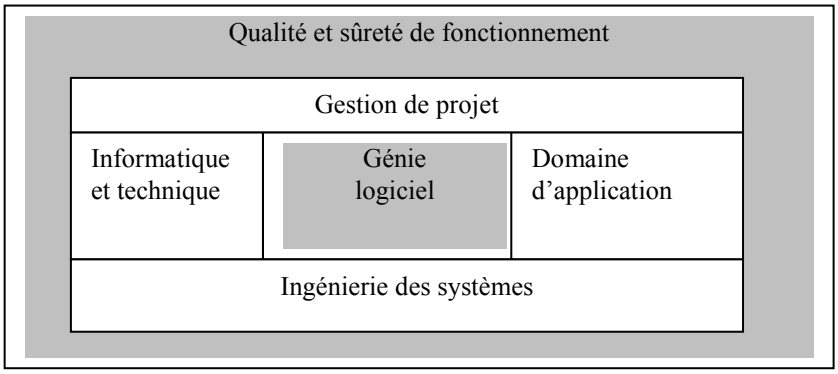

Fig. 3.2 Contexte II : IEEE

La qualité et la sûreté de fonctionnement[46] sont l'arrière plan où le génie logiciel et les autres disciplines sont insérées. Cela nous permet de souligner une fois pour toutes que la qualité et la sûreté de fonctionnement sont des conditions *sine qua non* de toute production de logiciel et, plus généralement, de toute production technique.

Le génie logiciel s'appuie sur l'ingénierie de systèmes et est confiné entre deux colonnes (l'informatique et le domaine d'application). La gestion de projet, comme il se doit, couvre le tout.

4.2.2. Selon SWEBOK

Les disciplines qui entourent le génie logiciel et qui, de façon plus ou moins approfondie, doivent faire partie des connaissances de l'ingénieur du logiciel sont :

1- l'informatique (*computer science*) ;

2- les mathématiques ;

3- la gestion de projet ;

4- l'ingénierie des ordinateurs ;

5- l'ingénierie de systèmes ;

6- les sciences de la gestion ;

7- les sciences cognitives et les facteurs humains.

Dans la figure suivante, nous représentons le contexte de manière graphique comme nous l'avons fait pour IEEE.

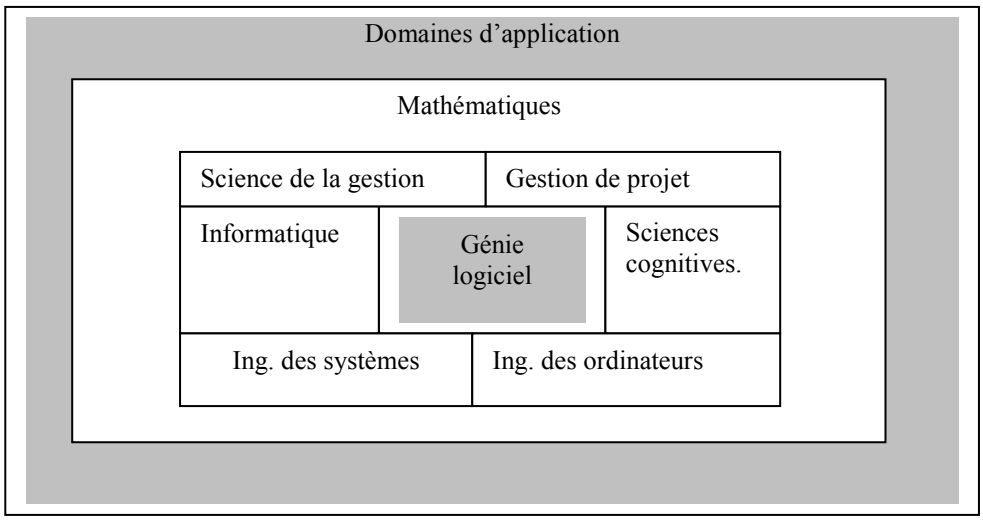

Figure 3.3 contexte III : SWEBOK

Puisque, dans SWEBOK, les domaines d'application ne sont pas considérés explicitement, nous les considérons comme un premier arrière-plan. Nous mettons les mathématiques comme deuxième arrière-plan pour indiquer que seulement dans le cas où les domaines sont mathématisés (dotés de règles plus ou moins formelles), on peut y appliquer les techniques du génie logiciel avec l'aide des disciplines connexes.

4.3. Corpus

Pour l'organisation du corpus de connaissances du génie logiciel nous considérons seulement SWEBOK, car les connaissances implicitement considérées par les normes IEEE sont explicitement indiquées dans SWEBOK.

Voici la liste des secteurs tels que définis dans la version 1.0 :

1- Exigences

2- Conception

3- Construction

4- Test

5- Maintenance

6- Gestion de la configuration

7- Gestion

8- Méthodes et outils du génie logiciel

9- Processus du génie logiciel

10- Qualité

Comme il fallait y s'attendre, à cause de la méthode employée, il n'y a pratiquement pas de surprises dans la classification, même si les secteurs ne sont pas homogènes. Certains secteurs sont des processus[47] (de 1 à 7), un secteur est une description de processus (9), un autre concerne l'environnement de production (8) et un dernier la qualité. La seule surprise est, selon nous, la description du secteur *Construction du logiciel*.

4.4. Comparaison des contextes

Le contexte IEEE et celui de SWEBOK ont en commun les disciplines suivantes : *gestion de projet*, *ingénierie des systèmes*, *informatiqu*e et *ingénierie des ordinateurs*. Cette dernière discipline est mise par IEEE dans la même « boîte » que l'informatique.

1. Disciplines présentes dans le contexte IEEE et absentes dans celui de SWEBOK
    a. *Domaines d'application*. Dans SWEBOK, les domaines d'application sont « absorbés » par le secteur du *génie des exigences*. La discussion sur la place du *génie des exigences* est sans doute l'une des plus importantes pour l'avenir du génie logiciel même si les paradigmes cachés des différents intervenants risquent de la rendre extrêmement difficile.
    b. *Gestion de la qualité*. Pour SWEBOK, elle fait partie du secteur de la qualité.
    c. *Sûreté de fonctionnement* (dependability). SWEBOK traite de la sûreté de fonctionnement dans le secteur qualité et donc à l'intérieur du génie logiciel.
    d. *Sûreté (*safety*)*. La sûreté est pratiquement absente de SWEBOK comme caractéristique séparée. Sans doute parce que certains auteurs considèrent la sûreté (*safety*) comme ne faisant pas partie de la sûreté de fonctionnement (*dependability*), ce que la langue française rend difficile à comprendre.
2. Disciplines présentes dans le contexte SWEBOK et absentes dans celui d'IEEE
    a. *Mathématiques*. L'absence des mathématiques dans IEEE est sans doute due au fait que, pour des ingénieurs, il va de soi qu'elles soient présentes. La mise en évidence des mathématiques dans SWEBOK est assez importante parce que beaucoup d'ingénieurs du logiciel ont tendance à minimiser l'importance des mathématiques.
    b. *Sciences de la gestion*. Le fait d'inclure les sciences de la gestion nous semble indiquer une vision du génie logiciel de SWEBOK plus large que celle d'IEEE.
    c. *Sciences cognitives*. La présence des sciences cognitives dans SWEBOK nous semble être liée au choix qui a été fait par SWEBOK à propos des domaines d'applications.

Même s'il est fort probable que, dans un futur pas très éloigné, on assistera à une uniformisation dans la définition des contextes que nous venons de présenter, il nous semble important de ne pas perdre l'occasion de creuser les implications des divergences actuelles. Divergences qui soulignent à quel point il est complexe de trouver la place du génie logiciel et qui, loin d'être fruits du hasard, mettent en évidence les points les plus délicats de la définition des frontières entre les disciplines qui participent au processus d'automatisation.

Nous considérons donc ces « points délicats » comme des éléments privilégiés pour notre critique du fondement du génie logiciel.

5. CONCLUSIONS

Nous avons introduit cinq principes qui nous semblent importants pour guider la réflexion sur le génie logiciel. Ces principes devraient nous faciliter l'interprétation du passé du génie logiciel et devraient nous permettre de proposer une approche plus scientifique à l'automatisation. Ces principes peuvent aussi être vus comme une manière de rendre explicites nos paradigmes « cachés » concernant l'automatisation.

Nous espérons que le fait d'avoir fixé des principes généraux comme P01 et P03 par exemple, nous permettra de garder une direction stable et de ne pas voleter au gré des « nouveautés » ou des souffles des vents de la mode.

Notre prochain pas consistera à analyser la qualité des processus et des produits logiciels pour essayer de faire ressortir des principes qui, ajoutés à ceux que nous venons de présenter, faciliterons les étapes d'élagage qui suivront.

NOTES

[1] Ce qui devrait ne pas les rendre complètement arbitraires.

[2] Automatisation dérive d'automate qui à son tour dérive du grec *automatos* « qui se meut de soi-même ». Dans notre cas, mouvement doit être interprété au sens plus général de « changement ».

[3] Plus on avance et plus cette première phase est complétée dans beaucoup de domaines.

[4] Modifier la perception du domaine implique que la partie langagière du domaine est modifiée mais étant donné l'inséparabilité de langage et domaine nu…

[5] Cette impossibilité de séparer provient du fait que ce qui n'est pas langage doit être là pour lui permettre de se référer à quelque chose d'autre que lui-même et qu'il doit exister pour permettre aux humains de construire un monde où l'interaction est possible. On pourrait aussi ajouter que la séparation doit être réalisée par le langage, qui est donc en même temps arbitre et joueur.

[6] Ces chiffres sont arbitraires mais pas dénués de bon sens. Bon sens qui pourrait aussi trouver ses justifications théoriques dans la règle du 7 plus ou moins 2 de Miller. Si le nombre de domaines connexes dépasse dix, par exemple, il est sans doute important de se demander s'il ne s'agit pas d'un sous-domaine, qui pourrait être fusionné avec d'autres sous-domaines dans une espèce de super-domaine.

[7] Ce qui ne veut pas dire que des incapables qui disent n'importe quoi donnent une description plus riche que les gens qui maîtrisent le langage et le domaine, mais seulement que, *ceteris paribus,* l'action de préciser implique une perte de sens et une augmentation de l'opérabililité et donc de la facilité « de faire » avec des machines.

[8] Il faudrait bien sûr arrêter de parler de langage de programmation. Comme on l'a vu dans la note à propos de la différence entre langue et langage, il faudrait parler de langue de programmation.

[9] Langue qui ne se réduit pas à la langue de programmation mais englobe toutes les primitives, les commandes et les outils de l'environnement bâti dans l'ordinateur.

[10] On peut bien sûr penser à une application d'intelligence synthétique où l'ordinateur détecte la présence humaine et arrête d'afficher… mais il s'agit d'un cas particulier qui n'invalide pas ce que nous venons de dire.

[11] Les naïfs des sciences cognitives, ce qui n'implique pas qu'ils soient tous naïfs même si le pourcentage est digne du taux de participation aux élections tunisiennes**.**

[12] *Système* : mot plus neutre que *machine*, mot ramasse tout.

[13] Ce point de vue pratique a en fait une facette théorique très importante : que l'humain ait besoin de granules d'une dimension bien plus grande que les 0 et les 1 pour comprendre (comprendre dans le sens pauvre de possibilité de manipulation) quelque chose et qu'à partir de ces granules on doive pouvoir construire de plus gros granules qui à leur tour deviennent les granules de base, etc. a un grand impact sur la théorie de la cognition et sur la signification de l'abstraction.

[14] C'est dans de tels cas que les traductions automatiques, même très contextualisées, donnent souvent le fou rire.

[15] Ici eau est un nom commun mais, en même temps, elle est « instanciée » par la présence du barrage Manic et de la Vanne no 03.

[16] La vanne no 03, comme tous les autres objets, est identifiée de façon univoque si on considère qu'on est à l'intérieur d'une centrale et que celle-ci a un identificateur unique.

[17] But ultime dans un contexte technique même si le but ultime était sans doute celui de gagner de l'argent (ou d'éclairer nos maisons ?). Cette note, politique seulement en apparence, soulève un problème très important et de solution difficile dans l'automatisation : celui de la hiérarchisation des buts et de l'autonomie des solutions par rapport aux buts premiers.

18 Domaine plus vaste que celui de la simple centrale.

[19] Nous n'aborderons pas ici le problème de savoir si « atome d'hydrogène en soi » signifie quelque chose pour la physique mais notre tendance, suivant ainsi l'école de Copenhague, serait de dire que l'atome est un ensemble de mesures obtenues avec des instruments macroscopiques.

[20] Le logiciel aussi peut être « sur papier ».

[21] Il ne faut pas confondre ce principe avec un lieu commun dangereux qui affirme que la spécificité du logiciel c'est qu'il est « immatériel » ! Contrairement à ce qui se passe dans le génie mécanique ou civil, par exemple, le processus de production de copies des exécutables et de la documentation est simple et complètement automatisable (produire des « copies » de voitures ou de maisons n'est pas complètement automatisable).

[22] Y. Wang et G. King : *Software Engineering Processes*, CRC, 2000.

[23] Même si c'était formulé surtout en termes de lisibilité et facilité de modification.

[24] Vieillissement ni en termes de fonctionnalités ni par rapport à l'évolution des logiciels de base, bien sûr. Mais c'est sans doute aussi à cause du fait que le logiciel, à la différence du matériel, ne

vieillit pas, qu'on le fait vieillir artificiellement en sortant de nouvelles versions à tout bout de champ pour renflouer les caisses des sociétés informatiques.

[25] Vu la grande difficulté de la définition, on pourrait penser, comme certains le font, qu'il est préférable d'abandonner la métaphore du génie. Nous croyons que ce n'est jamais une bonne idée que de jeter le bébé avec l'eau du bain.

[26] Voilà un autre des « gros » problèmes du logiciel : souvent les informaticiens sont en train de faire un prototype et ils ne le savent pas.

[27] Il suffit de penser à ce propos au théorème d'équivalence d'A. Tannenbaum cité en 2.6.

[28] On voit ici, comme dans le cas du syntagme génie logiciel, l'importance des noms. La maintenance a été introduite par analogie avec les autres génies comme maintenance corrective et ensuite, à cause aussi de l'organisation des départements d'informatique, elle s'est élargie pour englober tout genre de changement. Le fait qu'on a maintenu le terme *maintenance*, rend pratiquement inutilisable l'analogie avec les autres branches du génie. Notre principe P04 vaut bien sûr seulement dans l'acception large du mot maintenance.

[29] Penser en termes de réutilisation va dans cette même direction.

[30] Il suffit de penser avec quelle verve un praticien comme Yourdon a repris la querelle des GOTO !

[31] Il existe aussi une « difficulté » dans la conception des algorithmes que nous ne considérons pas ici, car nous la considérons comme faisant partie de l'informatique (*computer science*). Les mésententes entre informaticiens et ingénieurs du logiciel naissent de la manière différente de considérer les algorithmes.

[32] On devint souvent aristotélicien sans le savoir en comprenant l'importance de la classification pour maîtriser le monde. À ce propos, il faut citer un des livres les plus courageux de cette période : *Conceptual Structures* de J. F. Sowa.

[33] Pour anticiper sur nos considérations, c'est cette capacité de « déformation » qui nous aidera à situer les ingénieurs du logiciel par rapport aux experts du domaine.

[34] Il est clair que les capacités des individus sont si variées que, surtout dans le cas de systèmes complexes, en changeant une personne, on peut facilement passer d'une situation de « infaisable à des coûts finis » à « faisable à des coûts raisonnables ».

[35] Si certains logiciels, un jour, sont capables de produire de nouvelles informatisations, ce sera parce qu'à l'aide du langage les humains auront bâti un méta-programme. Les animaux, par contre, n'ayant pas de langage « logique », n'informatiseront jamais quoi que ce soit.

[36] De plus, très souvent, « meilleur » n'a aucun sens.

[37] Dans la littérature autour du génie logiciel, *Software Engineering Process* de Y. Wang et G. King (CRC 2000) est un des exemples les plus clairs non seulement de l'importance des processus mais de la nécessité de fonder la discipline du génie logiciel autour d'une formalisation des processus.

[38] Il est malheureux que les noms des deux documents se différencient seulement par un « - », surtout que leur contenu est assez différent.

[39] Technical Report CMU/SEI -99-TR-004

[40] CMU/SEI -99-TR-004, p. 2.

[41] SWEBOK, *A Stone Man Version* (IEEE – Trial Version 1.0 May 2001). http://www.swebok.org

[42] Ou bornes. Ce que nous appelons frontières.

[43] Le fait que la *dependability* englobe la *safety* est plus cohérent avec les définitions courantes de *dependability* comme étant une caractéristique d'un système constitué de *availability*, *reliability*, *safety* et *security*.

[44] Pour laquelle IEEE a défini un guide au corpus de connaissances : *A Guide to the Project Management Body of Knowledge*.

[45] C'est aussi à cause de la difficulté de définir ces frontières que le génie logiciel a, dans certains milieux, des difficultés à être accepté.

[46] Pour ne pas trop nous éloigner de la figure d'IEEE, nous avons gardé séparées les disciplines de la qualité et de la sûreté de fonctionnement, même si, par cohérence avec la définition même de qualité des normes ISO et IEEE, la sûreté de fonctionnement devrait faire partie de la qualité.

[47] On emploie ici le terme processus comme dans la norme IEEE 1074. Dans ISO 12207, par exemple, on appelle activités ce que nous, en suivant la norme IEEE 1074, appelons processus. Mais, pour notre réflexion, cela n'a aucune importance.

[48] À ne pas confondre avec les domaines que le logiciel est censé automatiser, tel que représentés dans la figure 2.3.

[49] Même si M. Jackson depuis des années non seulement insiste sur l'importance de la séparation mais propose des méthodes qui en tiennent compte.

Ivan Maffezzini
Institut Trempet
Université du Québec à Montréal
Canada
maffezzini.ivan@uqam.ca

Alice Premiana
Institut Trempet
Université du Québec à Montréal
Canada
premiana.alice@uqam.ca

Bernardo Ventimiglia
Institut Trempet
Université du Québec à Montréal
Canada
ventimiglia.bernardo@uqam.ca