Réussir les grands logiciels

Laurent Force, Thierry Michelon

▶ **To cite this version:**

Laurent Force, Thierry Michelon. Réussir les grands logiciels. Sciences de l'Homme et Société. 1996. hal-01908531

HAL Id: hal-01908531
https://minesparis-psl.hal.science/hal-01908531

Submitted on 30 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Nationale Supérieure
des Mines de Paris

Corps Techniques de l'État

Bibliothèque
IG 1
[321]

RÉUSSIR LES GRANDS LOGICIELS

Mémoire de troisième année de
Laurent FORCE et Thierry MICHELON

sous la direction de Gérard BERRY (Terrain)
et Alain BAMBERGER (Pilote)

Années 1995-1996

CONSULTATION SUR PLACE

Résumé

Explosion en vol du premier exemplaire d'Ariane 5, abandon du développement du système de contrôle commande de la centrale nucléaire de Chooz, crash du réseau téléphonique d'ATT sur tout le territoire américain,... la liste des catastrophes logicielles ne cesse de s'étendre. Une étude récente d'un cabinet américain montre d'ailleurs que seul un projet informatique sur deux parvient à son terme.

La situation française en matière de production de logiciels nous est apparue très contrastée. La situation est catastrophique dans certaines entreprises, alors que d'autres s'en sortent très bien. Il est même possible par une politique volontariste de passer d'une situation préoccupante à une situation satisfaisante en quelques années. L'échec n'est pas une fatalité.

Sans disposer d'une panacée, nous avons repéré des voies d'amélioration possibles de la production de logiciel. Résumées en deux mots ce serait organisation et modestie. Le fait qu'il faille une organisation adaptée n'a rien de nouveau, mais suppose un minimum d'efforts. La modestie signifie que la meilleure façon des réussir les grands logiciels est de ne pas en faire : essayer chaque fois que possible d'acheter des progiciels à la place, ne pas réinventer la roue et donc acheter le plus de composants possibles à l'extérieur, ne pas hésiter à réutiliser ce qui existe déjà et développer ses propres composants.

Remerciements

Nous n'aurions pas pu réaliser ce mémoire sans aide extérieure. Nous sommes donc particulièrement reconnaissant à Gérard BERRY, directeur du centre de mathématiques appliquées de l'École des mines à Sophia-Antipolis, et "terrain" de ce mémoire, et à Alain BAMBERGER, directeur du centre de mathématiques appliquées de l'institut français du pétrole, et "pilote" de ce mémoire de tout le temps qu'ils ont consacré à nous aider et à nous guider.

Nous avons également bénéficié des conseils avisés du professeur Claude RIVELINE, de l'École des mines de Paris, de l'aide de François BOURDONCLE, de l'École des mines de Paris, et du soutien de Alan BRYDEN, vice-président du mouvement français pour la qualité. Nous les en remercions vivement.

Enfin, nous exprimons notre reconnaissance à toutes les personnes qui ont accepté de nous recevoir, malgré leur emploi du temps souvent bien chargé. C'est à ces personnes, dont la liste se trouve ci-dessous, que nous devons la substance de ce mémoire.

Thierry BERGIER	AFNOR
Bernard SAUTEUR	Cabinet Synthèse Informatique
Jean BISSELICHES	Caisse des Dépôts et des Consignations
Arnaud BOUVET	CÉGÉLEC
Eric CAZEAUX	CÉGÉLEC
Jacques JOUBERT	CÉGÉLEC
Jean-Louis COULLON	CÉGÉLEC
Jean-Pierre HAUET	CÉGÉLEC
Yves BENOIT	CÉGÉLEC
Eddy SOULIER	CIGREF
Serge SOUDOPLATOFF	CP2I
Etienne DROIT	DASSAULT SYSTEMES
Pierre BLANC	EDF
Pierre LECOCQ	EDF
Yves BAMBERGER	EDF
Yves COUSIN	EDF
Pierre BERGER	LE MONDE INFORMATIQUE
François LE BRETON	MATRA DATAVISION
René GASQUET	Ministère de l'Économie-DSI
Patrick KNIGHT	Ministère de l'Industrie - SERICS
Sylvie METZ-LARUE	Ministère de l'Industrie - SERICS
Pierre CAZALA	Ministère de l'Industrie-SQUALPI
Bertrand MERCIER	PSA
Jean-Marc ROMAIN	PSA
François de NAZELLE	QUAL-AS
Philippe BISMUT	SAT-SAGEM
Alain MARBACH	SCHNEIDER
Jean-Claude MOREAU	SCHNEIDER
Jacques PINOTEAU	SEMA GROUP
M. UTTER	SEMA GROUP
Pierre JUGUET	SEMA GROUP
Pierre MESSULAM	SNCF
Daniel PILAUD	VERILOG

Table des matières

1. INTRODUCTION 4

2. LES DIFFICULTES 8

2.1 LES GRANDS LOGICIELS 8
2.2 UN TRAVAIL DE CONCEPTION 9
2.3 UN CONTEXTE EN EVOLUTION TRES RAPIDE 9
2.4 UN OBJET COMPLEXE 10
- *Le nombre d'éléments 10*
- *Des erreurs pénibles 11*
- *Une activité difficile à gérer 11*
- *Difficultés spécifiques 12*

2.5 CONCLUSION 13

3. LES METHODES DU GENIE LOGICIEL 14

3.1 LA RIGUEUR 14
3.2 LE CYCLE EN V 15
3.3 L'ORIGINE DES BUGS 18

4. NE PLUS FAIRE D'INFORMATIQUE 20

4.1 GRANDS LOGICIELS ? ON N'EN VEUT PLUS. 20
4.2 L'EXPLOSION DU MONDE DES PROGICIELS 20
- *Le big bang 20*
- *Le marché du progiciel 21*
- *Une préférence pour le progiciel 22*

4.3 POURQUOI CETTE PREFERENCE POUR LE PROGICIEL ? 22
- *Un logiciel spécifique performant mais à faible durée de vie 23*
- *Un progiciel général mais attractif et suivi 23*

4.4 NE PLUS FAIRE D'INFORMATIQUE OU L'INFORMATIQUE TRANSPARENTE 24
- *L'informatique : une boite noire facile d'utilisation 24*
- *Programmer sans taper une ligne de code 25*

4.5 LIMITES DU MYTHE 26
- *Intégration non aisée du progiciel 26*
- *Poids de l'existant 27*
- *Dépendance de l'éditeur 27*

5. COMMENT REUSSIR UN GRAND PROGICIEL ? 28

5.1 S'ORGANISER EN PROFESSIONNEL 28
- *S'armer de méthode et de rigueur 28*
- *Contrôler ses informaticiens 29*
- *Certifier le procédé de fabrication du logiciel 29*
- *Les outils 30*

5.2 UTILISER DES COMPOSANTS ELEMENTAIRES 32
- *Construire une base de composants maison 32*
- *Faire son marché et assembler 33*
- *À la recherche d'un standard de communication 35*

5.3 FAIRE VIVRE LE PROGICIEL 35
- *Utiliser les millions d'utilisateurs 35*
- *Innover sans cesse 36*
- *... mais innover prudemment 37*

6. CONCLUSION 39

BIBLIOGRAPHIE 41

1. Introduction

Dès que l'on parle de logiciels, un mot vient immédiatement à l'esprit : *Bug*. A l'origine, le mot désignait les insectes qui venaient bloquer le fonctionnement des premiers ordinateurs à lampe. Aujourd'hui, il désigne toutes ces petites erreurs, plus ennuyeuses les unes que les autres, qui surviennent chaque fois que l'on utilise un logiciel. Il suffit d'utiliser un traitement de texte pour être agacé par tous ces dysfonctionnements qui font perdre du temps.

Les bugs ne se contentent d'ailleurs pas de dégrader la productivité de chacun d'entre nous. Aujourd'hui, les ordinateurs, et donc les logiciels sont partout. Les bugs peuvent donc avoir des conséquences critiques. Prenons quelques exemples :

> Le 15 janvier 1990, ATT s'est trouvée dans l'obligation d'arrêter l'ensemble de ses commutateurs téléphoniques à la suite d'un bug. Une modification récente et infime du logiciel avait introduit une erreur, erreur particulièrement insidieuse puisqu'elle propageait les pannes d'un commutateur à un autre. Une grande part des abonnés américains se sont trouvés privés de communications interurbaines à un moment où ATT venait de lancer une grande campagne de publicité sur le thème de la fiabilité de ses services.
>
> La banque de New York (BoNY) a été victime d'une mésaventure extrêmement coûteuse : un bug dans un logiciel a empêché l'encaissement de 32 milliards de dollars provenant de la vente de produits financiers. La banque a eu un découvert vis à vis de la réserve fédérale de 24 milliards de dollars. Le bug a été corrigé et les choses sont rentrées dans l'ordre dès le lendemain, mais les intérêts d'une telle somme sur 24 heures sont de 5 millions de dollars !

Il semble donc qu'à l'égard des grands logiciels, la plus grande méfiance soit de mise. Nous avons enquêté auprès de ceux qui font des logiciels, et nous nous sommes rendu compte que la situation était bien pire encore que ce que nous venons de décrire. Les bugs ne sont que la partie émergée de l'iceberg, la vérité est que l'on ne sait pas vraiment faire de grands logiciels. Quand on commence un grand logiciel, on n'est jamais sûr d'arriver jusqu'au bout. Nous avons pu le constater à plusieurs reprises.

> Voyons les mésaventures survenues à un grand groupe industriel français, honorablement connu et dont nous continuerons à préserver l'anonymat. Le projet que nous avons étudié concernait une base de données, avec environ 1800000 informations, 700 utilisateurs et des problèmes de gestion de confidentialité. Le problème, sans que sa solution soit évidente, est plutôt classique. Le projet semblait plutôt bien parti : La base de données était construite autour de DB2, le progiciel de gestion de base de données d'IBM, il était fait usage d'un atelier de génie logiciel, une méthodologie était utilisée, l'équipe projet contenait non seulement des informaticiens mais aussi des représentants des utilisateurs, ... Sur le papier, ce projet avait tout pour réussir.
>
> Lorsque nous sommes arrivés, le projet était en route depuis huit ans et n'avait toujours pas donné de résultat tangible.

Les carences de la gestion de projet sont alors évidentes. On peut discuter pour savoir s'il s'agit d'une cause ou bien d'un symptôme. On constate par exemple qu'il n'y a aucune maîtrise du planning. Lorsqu'une phase est en retard, aucune action correctrice n'est prise. En fait, les responsabilités de chacun ne sont pas claires. Il n'y a pas de contrat, même implicite, pas de procédure de recette du produit final, les participants ne sont liés que par un engagement moral de bonne collaboration.

Le projet ayant duré huit ans, il a aussi été victime des changements de chef de projet. Trois se sont succédé. La méthodologie de développement utilisée est imposée par la direction informatique du groupe. Là aussi il y a eu trois changements pendant la durée du projet, qui ont bien évidemment retardé ou perturbé sa bonne marche.

Les méthodologies utilisées sont de toute façon mal maîtrisées. Elles sont lourdes et trop contraignantes. L'une d'elles par exemple prévoit que les utilisateurs ont besoin de se former à son langage de spécification pour exprimer leur besoin. La formation s'étale sur neuf mois ! On comprend la réserve des utilisateurs, qui d'une part ne comprennent pas très bien le formalisme abstrait mis en oeuvre, d'autre part ont d'autres activités plus importantes qu'expliciter leurs besoins en détail.

Les techniques mêmes ne sont pas très bien maîtrisées. La modélisation des données par exemple, est une phase importante de la définition d'une base de données. Sur ce projet, il a fallu la recommencer trois fois. A chaque fois elle était validée, et à chaque fois il a fallu la refaire au bout de quelques mois car elle était trop imparfaite.

Enfin la cause première de l'échec du projet est probablement l'impossibilité de déterminer ce qu'il fallait faire. Les discussions entre les informaticiens et les utilisateurs ont rapidement tourné au dialogue de sourds. Langage abscons des uns, indécision des autres, absence de position commune concourent à empêcher la rédaction d'une véritable spécification. La durée du projet n'arrange rien : en huit ans, les besoins des utilisateurs ont le temps de changer. En fait, c'est un outil de dialogue et de visualisation qui fait défaut. Un formalisme de spécification nécessitant une formation de la part des utilisateurs ne fait certainement pas l'affaire. Il faudrait pouvoir discuter autour d'une maquette ou d'une représentation de la base de données.

On pourrait penser que le projet que nous venons de décrire est caricatural. Il n'en est rien, nous avons vu ses frères dans d'autres entreprises.

Un grand spécialiste des automates cherchait à réaliser une carte de contrôle de moteur électrique ainsi que le logiciel associé. Sa taille est estimée aujourd'hui à 100 000 lignes, même si on n'en avait pas conscience au démarrage. Les causes d'échec sont similaires à celles du projet de base de données. Le projet souffre d'une organisation chaotique, alors que les activités autres que l'informatique sont bien disciplinées dans cette entreprise. Des difficultés

techniques se rajoutent : pour faire des économies, on utilise un microprocesseur bon marché, mais qui ne permet pas de bénéficier d'outils d'aide au développement très élaborés. Il s'avère difficile de définir des spécifications claires et surtout de les figer. Au bout du compte, le logiciel attendu sort avec un an et demi de retard et ne réalise que la moitié des fonctionnalités prévues.

Les problèmes ne concernent pas seulement des projets limités, qu'on a pu cacher dans un coin de l'entreprise. Le fait qu'un projet soit crucial pour l'avenir n'assure pas de réussir.

Cégélec et EDF développaient conjointement un nouveau système de contrôle commande pour la centrale de Chooz, dans les Ardennes. Ce système devait être révolutionnaire : il s'agissait de remplacer un système classique, uniquement basé sur des automates, où à chaque bouton poussoir correspondait une action sur un élément de la centrale, par un système entièrement informatisé, où les commandes sont passées de façon fonctionnelle, sans que l'opérateur ait à préciser le détail des éléments qui vont être activé. Les contraintes de sécurité sont extrêmement fortes (c'est ce qui amène à faire un développement spécifique), à la fois pour ce qui est de la résistance aux pannes et pour la garantie de temps de réponse court.

Le système n'a jamais vraiment fonctionné. Son développement a échappé à tout contrôle. EDF a fini par mettre fin à l'aventure en 1990, après huit ans de travaux et plusieurs centaines de millions de francs de dépense.

Les exemples que nous venons de citer sont purement français. On ne fait cependant pas mieux à l'étranger.

La bourse de Londres a cherché à informatiser son système d'échange de titres. Le projet s'appelait Taurus. Lorsqu'il a été abandonné, il avait déjà coûté près de 800 millions de livres (soit environ 3 milliards de francs). Le directeur général de la bourse a dû démissionner.

Les États-Unis ne sont pas en reste. Une étude récente du Standish group (1995), portant sur 8380 projets aux États-Unis montre que seulement la moitié d'entre eux est arrivé à terme. Un tiers a été abandonné en cours de route et le sixième restant n'a survécu qu'au prix du sacrifice d'une partie des fonctionnalités.

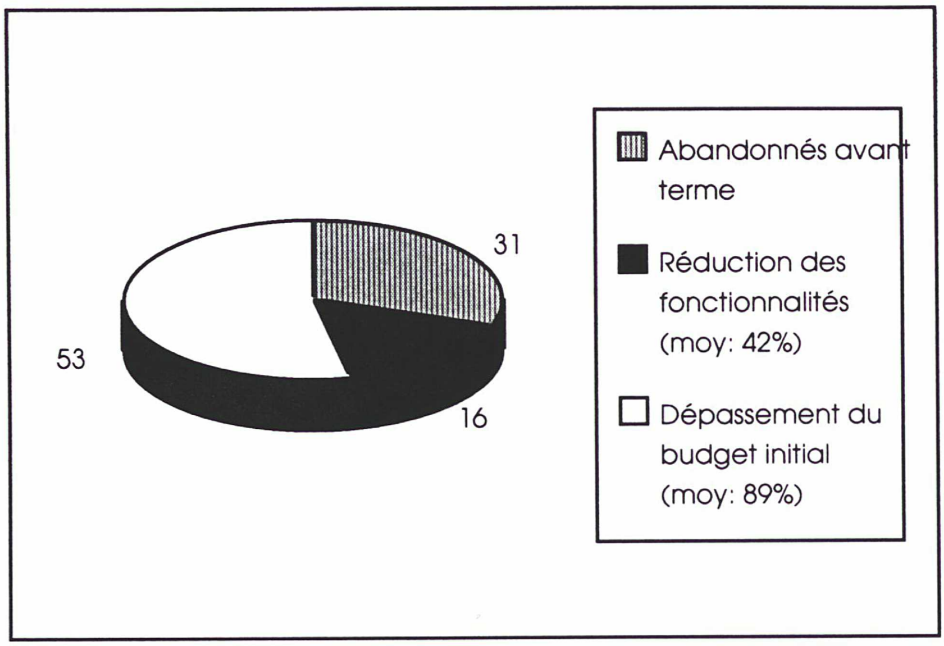

Figure 1 : sort de 8380 projets informatiques. Source : Standish group (1995), publié dans "Le Monde Informatique", dossier : faut-il supprimer les méthodes ? du 14 Juin 1996.

Une conclusion s'impose : se lancer dans la réalisation d'un logiciel est une entreprise risquée. Le problème n'est pas uniquement un problème de qualité, comme on peut en voir dans d'autres secteurs économiques. Les bugs sont certes très ennuyeux, mais le principal risque reste de ne pas arriver au bout de son développement.

2. Les difficultés

Réaliser un grand logiciel, c'est partir pour l'aventure. Tout le monde n'en a pas conscience, mais le résultat est net : seul un projet sur deux arrive à terme. Le domaine du logiciel est intrinsèquement difficile, ce que nous allons essayer d'expliciter, en commençant tout d'abord par quelques réflexions sur les particularités du *grand* logiciel.

2.1 Les grands logiciels

La distinction la plus marquante entre un grand logiciel et les logiciels que chacun (enfin surtout les plus jeunes...) a pu faire au cours de ses études d'ingénieurs, ou ceux qui ont pu être développés en interne par un employé talentueux est que dans le cas d'un grand logiciel, le fabriquant est différent de l'utilisateur. Cette différence est fondamentale : il va falloir que l'un explique ce dont il a besoin, ce qui suppose qu'il le sache. Il va falloir que l'autre écoute le premier, le comprenne et réalise ce dont il a besoin. Il apparaît un terrible problème de communication, aggravé par le fait que le logiciel est un objet extrêmement difficile à décrire.

Nous avons donné en introduction l'exemple d'un projet de base de données. C'est bien l'impossibilité de communiquer entre informaticiens et utilisateurs qui est à notre avis la cause première de l'échec du projet. La difficulté de trouver un langage commun pour décrire cet objet étrange qu'est le logiciel a atteint son paroxysme le jour où l'on a expliqué aux utilisateurs qu'ils devaient suivre une formation étalée sur neuf mois avant de savoir exprimer leur besoin.

Le grand logiciel est bien sûr aussi caractérisé par sa taille, mais les avis divergent quant à l'unité dans laquelle il faut l'exprimer :

- budget : un grand logiciel coûte plus de 200 MF nous a dit un de nos interlocuteurs chez EDF ;
- délais : un grand logiciel dure de une à plusieurs années ;
- nombre de personnes : un grand logiciel emploie plusieurs personnes. Certains ne commencent vraiment à parler de grands logiciels que lorsqu'il y a plusieurs équipes à coordonner.
- Nombre de lignes. Pour le Department of Defense aux USA, un grand logiciel fait plus de 100 000 lignes.

Il n'y a pas vraiment d'avis unanime, ni même d'accord quant à la façon de mesurer la taille d'un logiciel, ce qui traduit bien la difficulté de décrire le logiciel. Une chose est sûre : un grand logiciel emploie beaucoup de monde pendant plusieurs années. C'est un point qu'il a en commun avec beaucoup d'autres activités, comme les chantiers de travaux public. Pourtant, on n'entend jamais dire que de tels chantiers ne réussissent qu'une fois sur deux. La réalisation de logiciels souffre donc de difficultés ou tout au moins de déterminants particuliers, que nous allons essayer d'expliquer.

2.2 Un travail de conception

Une première remarque à faire concernant le logiciel est qu'il s'agit d'un travail de conception. Le logiciel est d'ailleurs protégé en France par un droit d'auteur, ce qui l'apparente à un livre, et non pas par un brevet.

La phase de fabrication, c'est à dire de reproduction est minime dans le logiciel : il suffit de le copier sur une disquette, sur un CD-ROM, voire de l'envoyer par l'intermédiaire d'un réseau. La copie est fidèle, le taux d'erreur minime. De ce fait, beaucoup de méthodes utilisées dans l'industrie sont inopérantes dans le domaine du logiciel : elles visent à assurer la conformité de la production au modèle qui a été défini lors de la conception, ce qui n'est pas un problème pour le logiciel.

Une activité de conception est une activité dans laquelle le facteur humain est prépondérant. Or utiliser des hommes a quelques inconvénients : d'une part la main d'oeuvre réellement qualifiée est rare, même si la situation actuelle est moins tendue. D'autres part, les hommes sont sujets à l'erreur. Et l'erreur est l'ennemie du logiciel.

Ces deux facteurs ont poussé au développement de ce que l'on pourrait appeler un fantasme de taylorisation. Certains responsables rêvent d'une usine à logiciels. Il suffirait de remplir un formulaire à l'entrée, d'appuyer sur un bouton et de récupérer le logiciel à la sortie. C'est un peu ce dont rêvent certains avec les méthodologies : en suivant pas à pas le mode d'emploi, être assuré d'arriver au résultat souhaité. L'autre versant du rêve, ce sont les ateliers de génie logiciel. Ils devaient être la machine outil de l'industrie du logiciel, qui le suit tout au long de sa chaîne de production et réduit l'intervention humaine au minimum.

Les espoirs placés dans l'usine à logiciel ont été déçus. "Faut-il supprimer les méthodes ?" titrait récemment le monde informatique[1]. Les ateliers de génie logiciel sont passés de mode. Bien souvent, leurs fabricants n'osent même plus utiliser le mot. Ils parlent de plate-forme de développement , d'outils, etc., ce qui signifie que même si l'on est parvenu à des résultats positifs, on a renoncé l'atelier complètement intégré permettant de réaliser un logiciel de A à Z.

C'est là une conséquence pratique du fait que le logiciel est un travail de conception : il est possible de faire des progrès dans l'automatisation de certaines tâches, mais le coeur du métier reste un travail de réflexion[2].

2.3 Un contexte en évolution très rapide

Le contexte dans lequel évoluent les logiciels est marqué par une évolution très rapide. La croissance des entreprises du secteur est exponentielle. Pour une entreprise, cela signifie qu'un concurrent aujourd'hui négligeable peut devenir le leader du secteur en quelques années. Hier Microsoft était un petit sous-traitant d'IBM, à qui on confiait

---

[1] = Le Monde informatique, n°682, 14 juin 1996

[2] = On peut lire à ce sujet l'article de F.P. Brooks, "No silver bullet ; essence and accident of software engineering", *IEEE Computer*, vol. 20, No 4, avril 1987, p 10-19.

quelques tâches faciles. Aujourd'hui, sa capitalisation boursière a dépassé celle d'IBM. Il faut réagir vite dans ce secteur.

La course en avant qui caractérise le secteur est le résultat de la concurrence effrénée que se livrent ceux qui font les logiciels, mais aussi de l'évolution tout aussi rapide des ordinateurs qui servent de support aux logiciels. Un projet qui dure un peu trop longtemps et l'ordinateur sur lequel on devait le faire tourner n'est plus commercialisé. Peu de gens peuvent se permettre de faire comme la NASA : jusqu'en 1987, la navette spatiale a fonctionné avec des ordinateurs à la pointe du progrès... ... de 1972 !

Les changements obligent à des révisions déchirantes : IBM a bâti sa renommée et sa fortune sur les gros ordinateurs centraux, les mainframes, avec des systèmes d'exploitations fermés. Aussi performants qu'ils soient, ces systèmes sont aujourd'hui passés de mode et IBM a été obligé de se convertir au PC et à Unix.

Il faut de plus faire des choix sans savoir. Par exemple, Verilog, éditeur de logiciel français, avait bien anticipé la richesse et les potentialités des langages orientés objets. En 1988, ils ont fait le choix du langage Eiffel, dont les spécialistes s'accordent à penser que c'est techniquement l'un des meilleurs choix possibles. Par manque de chance, c'est C++ qui s'est imposé, et en 1994 Verilog a décidé de basculer de l'un sur l'autre.

Pour réussir, il faut être capable d'anticiper les besoins. Lancer un logiciel de navigation sur l'internet lorsque toute la presse en parle, c'est déjà trop tard. Il fallait faire comme Netscape et se lancer alors que le World Wide Web n'était qu'un frémissement limité à la communauté scientifique.

Le flair, la réactivité et la créativité sont les qualités les plus utiles pour survivre dans un contexte aussi mouvant.

2.4 Un objet complexe

Le logiciel est un objet extrêmement complexe. Sous ce mot se cachent plusieurs aspects que nous allons essayer de détailler.

• Le nombre d'éléments

Un grand logiciel, c'est tout d'abord un très grand nombre de lignes de code. Il peut y en avoir plusieurs millions. Peu d'objets atteignent une telle complexité. Une fusée Saturne V, par exemple, contient environ un million de pièces, le module de commande Apollo, plusieurs millions. Voilà qui nous place très loin du mythe américain de l'inventeur écrivant son logiciel dans son garage. On peut faire un petit logiciel chez soi, avec peu de moyens, comme on a pu faire les premières automobiles dans un garage. Faire un grand logiciel, c'est passer à la complexité de la fusée, qu'il ne viendrait à l'idée de personne de tenter de construire dans son garage.

Face à un tel nombre d'instructions, aucun doute n'est permis : un certain nombre d'entre elles sont fausses. Qui plus est, celles qui sont justes prises isolément, peuvent

devenir fausses une fois mises dans leur contexte : les lignes de programme interagissent les unes avec les autres. Le nombre d'état que peut prendre un programme croît d'une façon extraordinaire avec la taille du programme. La combinatoire des états explose et une vie entière ne suffirait pas à les tester tous.

Il y a donc nécessairement des erreurs dans tout logiciel.

• Des erreurs pénibles

Les erreurs que l'on rencontre dans un logiciel sont particulièrement ennuyeuses. Tout d'abord, il n'y a aucune proportionnalité entre l'erreur et ses conséquence. Il suffit d'un détail infime, d'une virgule mal placée, pour faire tout échouer. Les physiciens qualifient de tels systèmes de chaotiques : un changement infime des conditions initiales suffit à provoquer un changement catastrophique du devenir du système. Il n'est pas possible, comme dans d'autre domaines de l'ingénierie, de se protéger en prenant des coefficients de sécurité. Lorsque l'on bâtit un pont, on le fait deux fois plus résistant que ce que les calculs disent nécessaire. Cette marge de sécurité permet de compenser les aléas, les imprévus et les petites erreurs. Il n'existe rien de tel pour le logiciel : toutes les erreurs sont potentiellement mortelles.

De plus, l'ordinateur est profondément stupide. Si l'on ne veut pas faire d'anthropomorphisme, on peut dire qu'il n'est pas doté d'intelligence. Il ne fait que ce qu'on lui dit, tout ce qu'on lui dit, rien que ce qu'on lui dit. Il n'y a donc aucune autocorrection des erreurs. Lorsqu'un service comptable travaille, ses employés corrigent toutes les petites erreurs qui ne manquent pas de se produire. Lorsqu'un logiciel fait le même travail, il le fait sans aucun bon sens, de sorte que toutes les erreurs se payent, mêmes les plus petites.

Enfin, le plus angoissant avec les erreurs d'un programme informatique, c'est le fait qu'à chaque fois qu'on en corrige une, on prend le risque d'en introduire une ou plusieurs autres. Réparer, c'est souvent aggraver les choses, faire que ce qui marchait avant ne marche plus. Or, la régression est une chose extrêmement mal tolérée par les utilisateurs : ils sont près à accepter que les nouvelles fonctionnalités ne fonctionnent pas parfaitement bien du premier coup, mais ils ne veulent pas de retour en arrière.

Ce risque est suffisamment présent pour que la politique d'IBM, lorsque l'on découvre une erreur dans un de ses logiciels soit de corriger l'erreur et de limiter la diffusion de la correction aux seuls clients qui sont gênés par cette erreur. Il est tout à fait inutile de faire courir le risque de nouveaux bugs à des clients qui ne sont pas gênés par le bug qui vient d'être découvert.

• Une activité difficile à gérer

La gestion de projet d'un grand logiciel n'est pas une activité simple. Il y a déjà un grand nombre de gens à coordonner. Matra Datavision, par exemple, qui produit le logiciel de CAO Euclide, emploie 200 développeurs. Son concurrent Dassault système, qui développe Catia, en emploie environ 600. Le logiciel est un objet complexe qui comporte beaucoup d'éléments. Ces difficultés ne sont cependant pas

propres au domaine du logiciel : gérer un gros chantier pose le même genre de problèmes. On y parvient généralement, même si les dérives de coût sont parfois préoccupantes.

Il est en revanche une difficulté qui touche plus particulièrement le logiciel : le logiciel est très difficile à visualiser. Certains vont même jusqu'à dire qu'il est invisible. Nous avons expliqué dans notre introduction les malheurs d'un projet de base de données : une part des ennuis était due à la difficulté de représenter le logiciel, à la fois pour dialoguer entre concepteurs et utilisateurs, et aussi pour juger si une phase était correcte ou non.

L'état d'avancement d'un logiciel est une chose extrêmement difficile à évaluer. Tant que l'on n'a pas réussi l'intégration finale, un logiciel n'est qu'un ensemble de bouts de code sans la moindre valeur marchande. Un logiciel en cours de réalisation est comme un film en cour de tournage. Seul le metteur en scène est capable de dire où l'on en est, et bien malin qui peu dire à partir de quelques plans si le film sera bon.

La difficulté de se représenter le logiciel complique la vie des développeurs et des chefs de projets, mais elle touche encore davantage les dirigeants qui pour beaucoup d'entre eux n'ont pas de culture informatique. Bien souvent, le seul moyen de pallier cette insuffisance est de faire appel à des experts, avec tous les inconvénients que cela suppose, en particulier le problème d'en trouver.

• Difficultés spécifiques

Toutes les difficultés que nous avons citées jusqu'à présent sont génériques, c'est à dire qu'elles se posent de façon plus ou moins aiguës pour tous les logiciels. Il y a en plus des difficultés propres à chaque type de logiciel, qui font que l'on ne sait pas tout réaliser.

Dans le cas de la base de données que nous avons vu en introduction, et de façon plus générale pour beaucoup de logiciel de gestion, la contrainte est de parvenir à satisfaire un grand nombre d'utilisateurs ayant des besoins extrêmement différents.

Dans le cas de la carte de contrôle moteur, et de façon plus générale pour les logiciels "embarqués", c'est à dire servant à piloter des systèmes physiques, la contrainte la plus forte est une contrainte de temps réel. Il faut obtenir de façon certaine une réponse aux stimuli dans un temps donné.

Le logiciel de contrôle commande de la centrale de Chooz obéit également à des contraintes de temps réel, mais subit de plus des contraintes de sécurité. Il fait partie d'un système redondant pour résister aux pannes.

Enfin, une des difficultés les plus fréquentes est le problème de l'héritage : il faut rester compatible ou pouvoir s'interfacer avec les systèmes précédents. Ce n'est pas une mince affaire, et le système d'information d'une grande entreprise ressemble rapidement davantage à une usine à gaz qu'à un modèle extrait d'un cours d'informatique. Il devient impossible de modifier une partie sans apporter des changements à toutes les autres.

2.5 Conclusion

Après avoir dressé la liste de toutes les difficultés qui attendent les développeurs de logiciel, on réalise qu'il ne faut pas s'étonner du taux d'échec très élevé, mais plutôt s'étonner du fait que parfois ça marche.

"Le logiciel, ça tombe en marche" nous a-t-on déclaré.

En matière de logiciel, le cas général est que ça ne marche pas. De toute façon, ça n'a aucune raison de marcher. Il existe des cas particuliers de logiciels qui fonctionnent plus ou moins bien, avec quelques problèmes, mais qui finalement donnent satisfaction à leurs utilisateurs. Naturellement, ils s'en plaignent, mais force est de constater qu'ils continuent à acheter.

Enfin, il existe un mythe : celui d'un logiciel exempt de bugs. Un tel logiciel n'existe pas.

3. Les méthodes du génie logiciel

3.1 la rigueur

L'écriture d'un logiciel est une tâche difficile. Il faut concilier d'une part le fait que l'écriture de logiciels est une activité créative et dynamique, donc qui s'accommode mal des contraintes et d'autre part le caractère complexe du logiciel, qui ne peut être maîtrisé qu'avec la plus grande rigueur. Les nombreuses méthodes qui existent se distinguent, entre autres choses, par le poids qu'elles accordent à l'un plutôt qu'à l'autre de ces deux facteurs.

La méthode RAD (*Rapid application development*), dont on parle beaucoup en ce moment, met l'accent sur la rapidité et la dynamique de la programmation. Un logiciel destiné à aider la force de vente dans le cadre d'une promotion ponctuelle n'a aucune valeur s'il faut plus de six mois pour le développer. Peu importe que le résultat obtenu soit peu fiable, l'essentiel est de produire très rapidement une ébauche du logiciel que l'on va affiner par itérations successives avec les utilisateurs. Une telle méthode est privilégiée lorsque la rapidité de développement est le critère déterminant du succès de l'application. Les logiciels ainsi développés sont souvent qualifiés de jetables, car leur faible coût fait qu'on n'hésitera pas à s'en débarrasser pour les remplacer par d'autres lorsque les besoins auront changé. De toute façon, un processus de développement de type RAD est plutôt anarchique et ne favorise donc pas la maintenance ultérieure de l'application.

Les grands logiciels ne sont pas en général dans une situation permettant d'envisager un développement suivant une méthode de type RAD. Tout d'abord ils sont très complexes, ensuite ils représentent un investissement important que l'on va vouloir conserver pendant plusieurs années, ce qui oblige donc à prévoir leur maintenance, enfin, ils remplissent souvent des fonctions essentielles (sinon on n'investirait pas des sommes importantes dans leur réalisation), qui s'accommodent mal des erreurs. Les logiciels de pilotage d'un avion ou d'une centrale nucléaire n'ont rien de simple, vont devoir être maintenus en parfait état pendant au moins vingt ans et ne supportent pas les bugs. En revanche, ils ne subissent pas des variations trop rapides de leur environnement.

Pour les grands logiciels, la balance penche donc plus du côté de la rigueur que du côté de la spontanéité.

Les grands logiciels qui marchent sont développés dans un cadre en général très organisé. Il reste peu de place pour les fantaisies individuelles. De nombreuses méthodologies sont utilisées, mais dans ses grandes lignes, la manière de faire est proche de ce qui se fait dans d'autres domaine d'ingénierie complexe comme l'industrie de l'armement ou l'industrie spatiale. Elles reposent sur un découpage du problème en tâches simples, une gestion saine du planning, des moyens et des hommes et surtout sur un découpage en phases délimitées par des jalons dont on vérifie qu'ils sont bien atteints à l'aide de revues avant de passer à la phase suivante.

Un point a particulièrement retenu notre attention : les rapports qu'entretiennent la maîtrise d'ouvrage et la maîtrise d'oeuvre. Nous avons cité trois exemples de projets

en grande difficulté dans notre introduction : une base de données, une carte de contrôle de moteur et un système de contrôle commande de centrale nucléaire. Ces trois programmes ont en commun le fait que le rôle de la maîtrise d'oeuvre et de la maîtrise d'ouvrage est extrêmement mal clarifié.

Dans le cas de la base de données, il n'y a tout simplement pas de maîtrise d'ouvrage. On constate que l'écriture du logiciel butte sur la définition des besoins, ce qui est normalement du ressort de la maîtrise d'ouvrage. Les diverses tentatives d'exprimer un cahier des charges sont en fait un mélange des besoins fonctionnels et de leur solution technique.

Dans le cas de la carte de contrôle, c'est le dialogue entre la maîtrise d'oeuvre et la maîtrise d'ouvrage qui se passe mal. Il est impossible de stabiliser la définition des besoins, les modifications incessantes désorganisent l'équipe de développement.

Dans le cas du système de contrôle commande de la centrale nucléaire de Chooz, il y a bien une équipe de maîtrise d'oeuvre et une de maîtrise d'ouvrage bien définies, mais il y a en quelque sorte collusion entre ces deux équipes. La maîtrise d'ouvrage ne se contente pas de définir les besoins, ce qu'il faut faire, mais elle définit aussi le comment qui devrait être du ressort de la maîtrise d'oeuvre. Elle se retrouve alors prisonnière de ses choix et n'a plus aucun recul. EDF a bien identifié le problème, et a décidé de mieux séparer les rôles à l'avenir et surtout a institué un "régulateur", complètement indépendant, dont le rôle est d'intervenir chaque fois qu'il juge qu'il y a risque de dérapage.

En fait, comme nous l'a expliqué l'un de nos interlocuteurs, *on a l'informatique que l'on mérite* et la maîtrise d'ouvrage joue un rôle prépondérant pour éviter les grosses déconvenues.

3.2 Le cycle en V

L'organisation en projet n'est pas particulière à l'industrie informatique. Le découpage en phases est en revanche plus spécifique du logiciel. Il en existe de nombreuses variantes, mais la plus connue est celle du cycle de vie en V. Ce n'est pas la seule façon de voir les choses, mais c'est un modèle que tout le monde a en tête. Nous allons donc en exposer les grandes lignes.

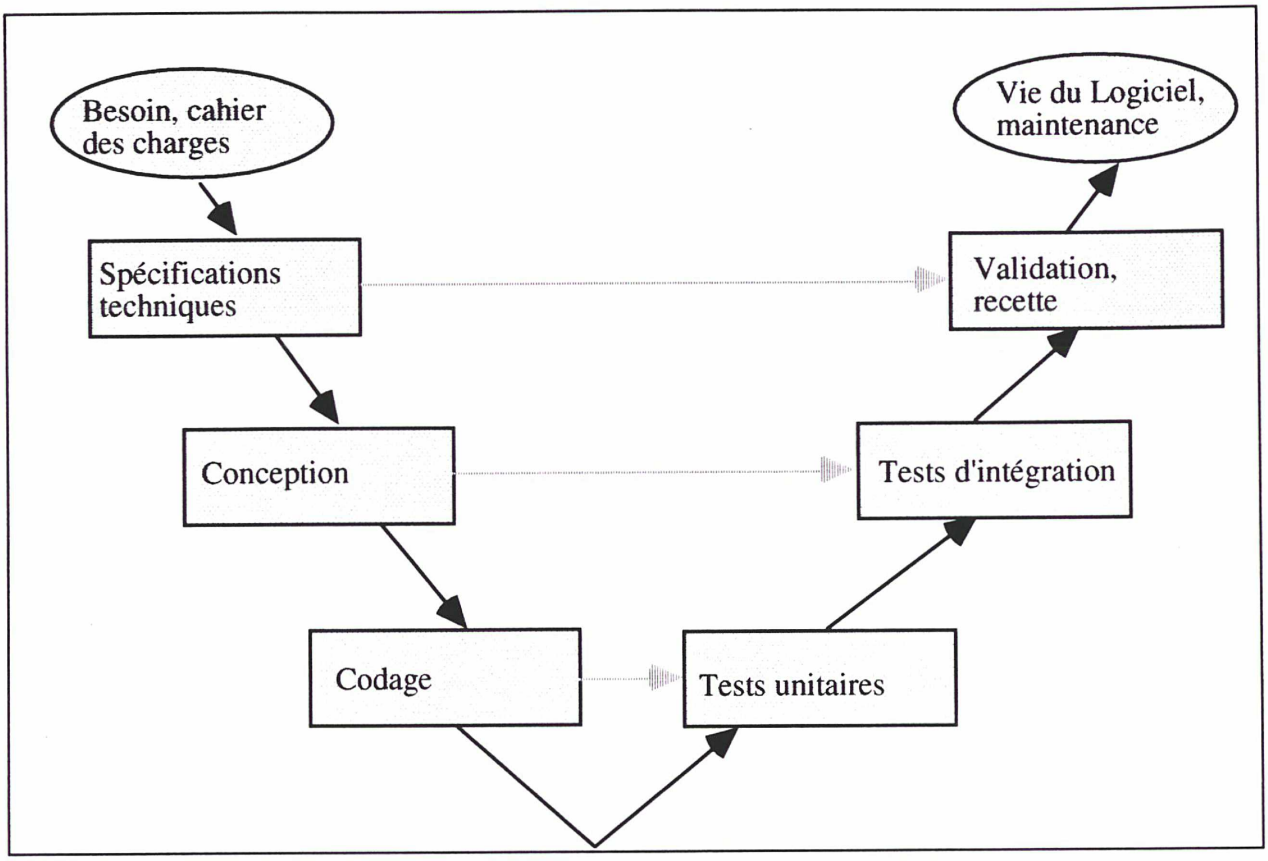

Figure 2 : les 8 phases du cycle en V.

Le cycle de vie du logiciel commence avec l'expression des besoins du client. Ceux-ci peuvent être formalisés sous forme d'un cahier des charges ou être plus flous. Lors de la première phase on rédige les spécifications techniques du logiciel, c'est à dire qu'on le décrit aussi complètement que possible, dans ses fonctions et ses comportements, ses interfaces, etc.. Il ne doit pas rester de zones d'ombre ou de définitions implicites qui risqueraient d'amener à réaliser un produit différent de celui attendu.

La phase suivante est la phase de conception, où l'on définit l'architecture du logiciel. On le découpe en morceaux plus simples ou même en fonctions élémentaires qui seront assemblés pour obtenir le résultat voulu. Suivant le talent de l'architecte ce découpage permet d'obtenir une situation plus ou moins simple. Le principe de base est d'obtenir des morceaux aussi indépendants les uns des autres que possible, de façon à pouvoir les tester séparément.

On passe enfin à la phase de codage proprement dite. Contrairement à ce l'on pourrait penser a priori, l'écriture du code n'est pas la phase la plus longue. Elle peut ne représenter que le sixième du travail nécessaire pour aboutir à un logiciel en état de marche. C'est sans doute la phase qui a connu le plus de progrès depuis les débuts de l'informatique. La programmation en assembleur ou langage machine, qui demandait une véritable gymnastique intellectuelle, a presque disparu au profit de langages évolués comme Fortran, Cobol, et plus récemment C, C++, ADA,... qui permettent d'écrire des programmes beaucoup plus long. De plus, il existe de plus en plus d'outils d'aides, qui vont jusqu'à la génération automatique de code dans le cas des ateliers de génie logiciel.

Ces phases constituent la partie descendante du cycle en V, en ce sens que l'on part du plus général, le besoin du client, que l'on analyse et décompose en éléments les plus petits possibles. Le cycle en V se nomme ainsi parce qu'il comporte aussi une branche ascendante, où l'on va assembler les éléments de bases pour constituer le logiciel complet. la branche ascendante est placée sous le signe du test. Lors de chaque phase descendante, on a prévu des tests qui permettront de vérifier que la phase a bien été exécutée correctement. Les deux branches du V sont en étroite correspondance.

Associer des tests de vérification à chaque phase est une pratique courante de la gestion par projet. Le logiciel se différencie cependant des autres activités en ce que le test peut tenir une place prépondérante : on peut très bien consacrer plus de la moitié d'un projet à dans les tests pour vérifier que le logiciel se comporte bien comme prévu. Le logiciel est sans doute l'une des rares activités où l'on peut passer plus de temps à vérifier que les choses sont faites correctement qu'à les faire.

Les premiers tests, dits unitaires, se font sur les morceaux élémentaires de code qui ont été définis lors de la phase de conception. Si celle-ci a été bien faite, on peut les tester de façon très complète dès cette phase. Il est très intéressant de pouvoir ainsi tester le programme petits bouts par petits bouts, car on sait que la complexité d'un programme croît exponentiellement avec sa taille. Il serait illusoire d'espérer tester directement le programme dans son ensemble.

On passe ensuite aux tests d'intégration qui permettent de vérifier que les différents morceaux coopèrent bien entre eux.

La dernière phase est la validation ou la recette du programme. Cette phase peut prendre des formes très variées, comme un essai sur un site pilote ou la distribution d'une pré version à des utilisateurs bien choisis. On peut ainsi éliminer les derniers bugs (enfin, les presque derniers...), en testant des cas auxquels les concepteurs n'avaient pas pensé.

Il faut bien avoir conscience que ce modèle n'est justement qu'un modèle. Jamais un vrai logiciel n'a cette vie idyllique où chaque phase se conclut par une revue favorable qui permet de passer à la suivante. Il est très difficile de juger si une phase s'est terminée de façon satisfaisante ou non. En fait il reste toujours des problèmes, que l'on ne découvrira que dans les phases ultérieures et qui obligeront à revenir en arrière. Les "bons" informaticiens parviennent à limiter le nombre de retour en arrière et leur amplitude : il est beaucoup plus ennuyeux de découvrir une erreur de conception lors de la phase d'intégration que de se rendre compte d'une erreur de codage lors d'un test unitaire.

La durée d'un cycle en V peut de plus atteindre plusieurs années. C'est un inconvénient majeur, car c'est aussi le temps qu'il faudra attendre pour avoir les premières réactions des utilisateurs. Pour éviter cet inconvénient, certains développeurs utilisent un cycle dit "en spirale", où l'on préfère construire progressivement le logiciel. Les phases sont similaires à celle du cycle en V, mais elle n'ont pas vocation à être achevées à la première tentative, ce qui permet d'aller beaucoup plus vite. On commence par définir des spécifications simplifiées, on les

code, on les vérifie et l'on recommence en étoffant les spécifications, codant,... jusqu'à obtenir après quelques cycles le résultat souhaité.

3.3 L'origine des bugs

Le processus de fabrication du logiciel est dominé par les tests. Le logiciel passe par une série de cribles qui éliminent leur paquet d'erreurs à chaque étape. Malgré toute l'attention que l'on porte aux tests, il reste toujours quelques erreurs à la fin. Ce sont les bugs.

On ne peut totalement supprimer les bugs, mais il est possible d'en avoir plus ou moins en fonction de l'intensité de test que subit le programme et aussi en fonction de la qualité de la fabrication du programme. La qualité de l'écriture et de la conception ont un effet sur le fait de pouvoir corriger les erreurs que l'on trouve sans en introduire de nouvelles en plus grand nombre. Un programme bien écrit produira de plus moins d'erreurs à corriger, mais au bout du compte, la quantité d'erreurs résiduelle dépend beaucoup des tests.

Tous les programmes n'ont pas de plus le même besoin de qualité : les utilisateurs sont beaucoup plus tolérants sur les bugs d'un traitement de texte, qui tout au plus font perdre un peu de temps, que sur les bugs du système de pilotage d'un Airbus. Lorsque des vies humaines sont en jeu on ne peut plus se permettre d'avoir des bugs graves.

Puisque les logiciels sont testés de façons multiples, il est possible de connaître l'origine des bugs.

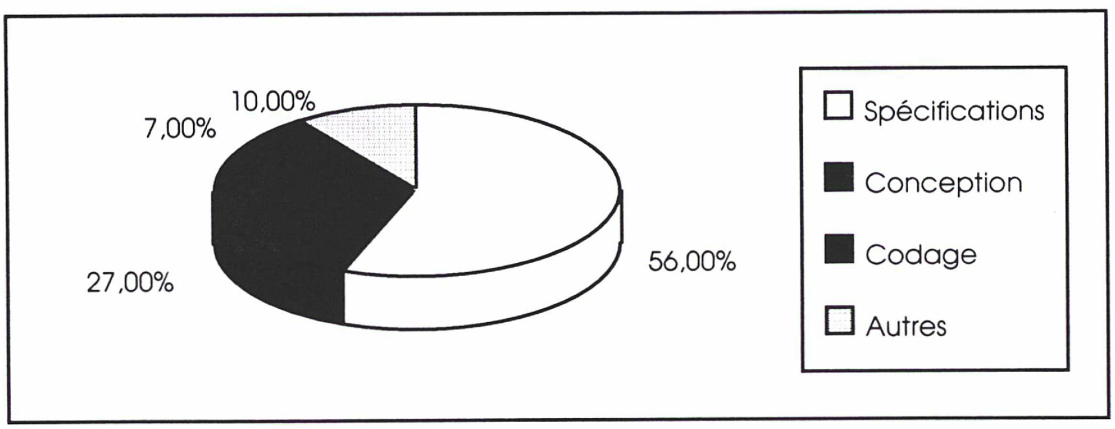

Figure 3 : contribution en bugs des différentes étapes de la réalisation d'un logiciel. Source : Dirk Bender (1983) in "Writing testable requirements", cité par Edward Kit in "Software testing in the real word", Addison-Wesley, 1995.

La majorité des erreurs se produisent en amont du cycle de vie du logiciel.

Contrairement à une opinion répandue, la phase de codage n'est responsable que de très peu de bugs. Un bug de codage, c'est par exemple cette mésaventure survenue au programme Mercury dans les années soixante. A la place de l'instruction Fortran "`DO I=1,10`" il y avait "`DO I=1.10`". Cette toute petite erreur (un point à la place d'une virgule) suffit à faire que la boucle n'est exécutée qu'une seule fois au lieu de dix.

Ce genre de bug n'est pas le plus fréquent aujourd'hui. En fait, l'erreur du logiciel de la capsule Mercury est quasiment impossible aujourd'hui : le compilateur effectue des tests qui permettent de détecter ce genre de petites erreurs. Les outils modernes dont on dispose aujourd'hui permettent de se prémunir contre beaucoup de fautes d'inattention.

Il n'y a pas de méthode automatique pour éviter les erreurs plus en amont. Les bugs de conceptions sont relativement fréquents. La série d'accident des TERAC 25 était due à une erreur de conception : le TERAC 25 était un appareil destiné au traitement du cancer par irradiation. La conception du logiciel de commande était défectueuses, de sorte qu'il était possible, dans certaines circonstances rares de délivrer au patient une dose bien plus élevée que celle prévue. Plusieurs ont été gravement brûlés, certains en sont morts. Les logiciels peuvent être extrêmement dangereux.

Enfin, le cas le plus fréquent, c'est le bug de spécification. Le logiciel ne fonctionne pas parce que l'on ne lui a pas demandé de faire ce qu'il fallait faire. Cela n'a rien de surprenant, nous avons déjà expliqué dans les chapitres précédants combien il était difficile d'exprimer ses besoins avant de se lancer dans l'écriture d'un logiciel.

4. Ne plus faire d'informatique

4.1 Grands logiciels ? On n'en veut plus.

Les utilisateurs ne veulent plus de grands logiciels spécifiques, de grands projets pharaoniques. Socrate, la centrale nucléaire de Chooz, dernièrement Ariane 5... la liste des catastrophes est longue et coûteuse. Le logiciel spécifique s'apparente à un pari technique, à de la recherche, à du poker. Or on ne veut plus être des cobayes, on n'a plus l'argent et on n'a plus le temps.

Quand EDF rompt en 1990 son contrat avec Cégélec concernant la fourniture du système de contrôle-commande de la centrale nucléaire de Chooz, EDF met à la poubelle 6 ans d'études et 600 millions de francs. Plus grave, EDF n'a alors plus que trois ans pour trouver une alternative : la centrale démarre en août 1993.

Mais où trouver un système de contrôle-commande de centrale nucléaire nouvelle génération ? La réponse fut trouvée en faisant appel au marché. EDF s'est rendu compte qu'un système de contrôle standard utilisé par des dizaines de centrales thermiques allemandes pouvait remplir 90% de ses besoins. EDF l'a acheté et n'a développé spécifiquement qu'une petite application satisfaisant aux exigences de sécurité propres au nucléaire.

Ce repli d'EDF sur une solution existante n'est pas une attitude singulière. Récemment, plutôt que de développer son propre logiciel, Air France a acheté un logiciel de gestion des clients commercialisé par Air Canada. Plus massivement la plupart des PME françaises ont abandonné leurs logiciels de comptabilité, faits maison dans les années 70, au profit de progiciels du commerce vendus dans des bureaux de tabac.

4.2 L'explosion du monde des progiciels

• Le big bang

Progiciel, le mot est lâché. Contraction de produit et de logiciel, le progiciel ou logiciel standard a en 15 ans envahi le marché en même temps que le PC. Le big bang remonte à 1981, quand IBM lance le premier PC. On comptait l'année dernière, en 1995, 200 millions de PC à travers le monde[3]. En 1992, Windows donnait accès à plus de 6000 progiciels et MS-DOS à plus de 35000[4].

Progiciels et PCs ont inondé simultanément le marché pour deux raisons. D'abord le PC doit son succès à un progiciel : le tableur, qui apporte à toute personne qui manie des tableaux de chiffres des gains de productivité immédiats. Ensuite le PC a rendu le développement de logiciels accessible à des petits éditeurs indépendants, en même temps qu'elle a créé un marché pour les progiciels ainsi développés. Ainsi en France, le bureau d'études CXP, grand spécialiste français du progiciel, recensait dans un

---

[3] = Journal Valeurs Actuelles du 28/05/95 in "L'ordinateur est mort, vive le logiciel !".

[4] = Journal 01 informatique du 09/10/92 in "OS/2 : les développeurs restent prudents".

rapport[5] paru en 1996 plus de 2832 éditeurs de progiciels, dont 40% ont moins de 10 salariés.

• Le marché du progiciel

Les progiciels sont réalisés par des éditeurs de progiciel comme Microsoft, dont c'est le coeur du métier, des constructeurs de matériel informatique comme IBM ou des SSII, qui se diversifient ou exploitent un savoir faire comme une base de donnée temps réel dans le cas de SEMA. Les entreprises américaines dominent le marché des progiciels pour la micro-informatique et la bureautique en particulier. Seules quelques bastions comme la CFAO avec Dassault Système et Matra Datavision, ou quelques niches comme les logiciels de comptabilité générale - propre à la France - sont farouchement défendus par des éditeurs français.

Du côté des acheteurs, les entreprises françaises ont investi environ 20 milliards de francs en 1993 dans l'achat de progiciels[6]. Les entreprises industrielles sont le premier consommateur de progiciels (33% des 20 milliards), suivies par les entreprises commerciales (22%), les banques et assurances (18%) et l'administration (15%).

Le parc de progiciels installés dans les entreprises françaises est évalué à 63 Milliards de francs[7]. Ce parc est constitué à 70% de progiciels de gestion (bureautique, comptabilité, paye, gestion du personnel), à 10% de progiciels techniques (CAO, scientifique, communication, réseau, télématique) et à 20% de progiciels dits métier qui sont spécifiques à un secteur économique (enseignement, transport, banque, agriculture).

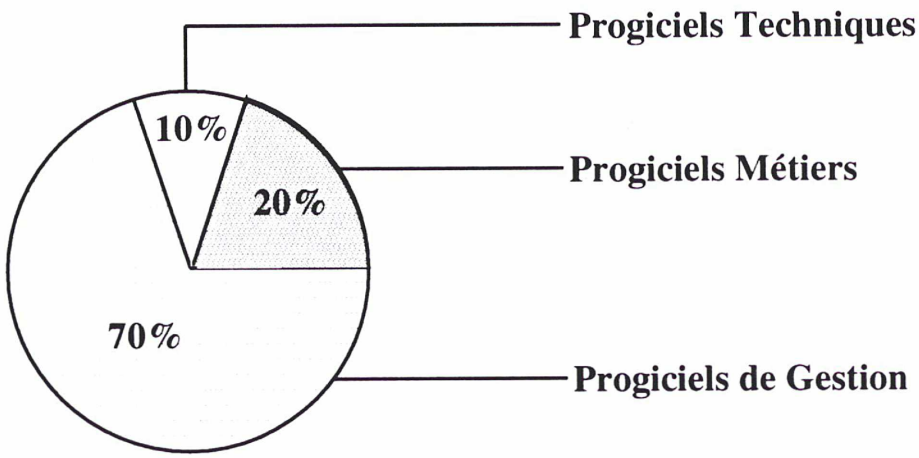

Figure 4 : proportions des trois sortes de progiciels. Source : Rapport du CXP, GAMIX les acteurs du marché informatique et des progiciels, édition 1996.

Le progiciel, ce n'est pas que Word ou Excel de Microsoft. C'est aussi le système de contrôle de la centrale nucléaire de Chooz ; c'est l'ensemble de la gestion d'une grande entreprise. Certains progiciels sont mêmes adaptés à des besoins spécifiques et de plus en plus de logiciels spécifiques utilisent des composants progiciels : là une interface

---

[5] = Rapport du CXP, GAMIX les acteurs du marché informatique et des progiciels, édition 1996.

[6] = Rapport de Pierre Audoin Conseil, "Analyse générale du marché français" - SSII 1998, Juin 1994.

[7] = Rapport du CXP, GAMIX les acteurs du marché informatique et des progiciels, édition 1996.

standard, ici un système de gestion de bases de données du marché. Bref, le progiciel est partout et prend la place du logiciel spécifique.

• Une préférence pour le progiciel

L'histogramme qui suit présente la part en progiciel du parc applicatif japonais, européen et américain. En 1991, en Europe les progiciels représentaient 40 % du patrimoine logiciel, contre 10% au Japon et 60% aux USA. Si on ajoute que, dans les années 70, l'Europe était dans la même situation que le Japon aujourd'hui et que les USA représentent la référence en matière d'informatique, la tentation est grande de transformer cette vision statique en une vision dynamique, cet histogramme spatial en un histogramme temporel et d'en déduire que la progicialisation est en marche.

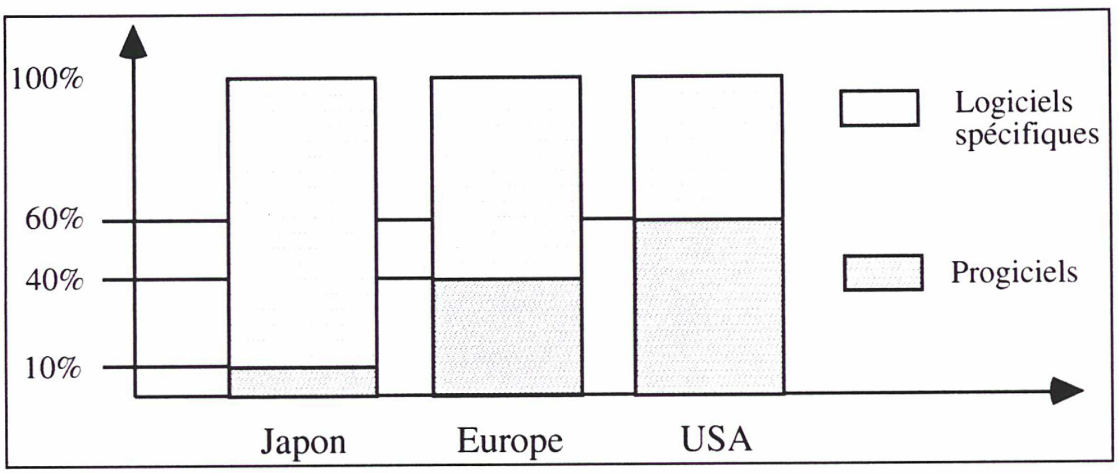

Figure 5 : proportion du parc applicatif en progiciels en 1991. Source : William Toten in "Gérer et Comprendre" de Mars 1991.

En ce qui concerne la France, un cabinet de conseils, Pierre Audoin Conseil, annonçait en 1994 dans une étude[8], qu'aujourd'hui les progiciels représentent 30 % du patrimoine logiciel. Mais dans 10 ans, plus de 50 % du parc applicatif sera composé de progiciels. La France est donc en dessous de la moyenne européenne en proportion de progiciels, mais la progicialisation y est en marche également.

4.3 Pourquoi cette préférence pour le progiciel ?

Pour répondre à cette question, il s'agit de clarifier les notions de progiciel et de logiciel spécifique et d'en présenter les caractéristiques. Étymologiquement le logiciel spécifique se distingue du progiciel par sa faible diffusion, mais la frontière entre ces deux catégories de logiciel est floue.

En effet si on représente les différents types de logiciel en fonction de leur diffusion, on obtient une pyramide. Son socle est composé des progiciels de type Word ou un compilateur C, vendus à des millions d'exemplaires. Plus haut, on trouve les progiciels paramétrables comme un logiciel de gestion de production. Ils sont constitués d'un noyau central fixe qui est adapté aux besoins spécifiques des clients. Encore plus haut règnent les logiciels spécifiques qui font usage de progiciels. Et enfin, du sommet

[8] = Rapport de Pierre Audoin Conseil, "Analyse générale du marché français" - SSII 1998, Juin 1994.

dominent les logiciels spécifiques écrits de la première ligne de code à la dernière, de type nouveau langage de programmation ou programme de recherche.

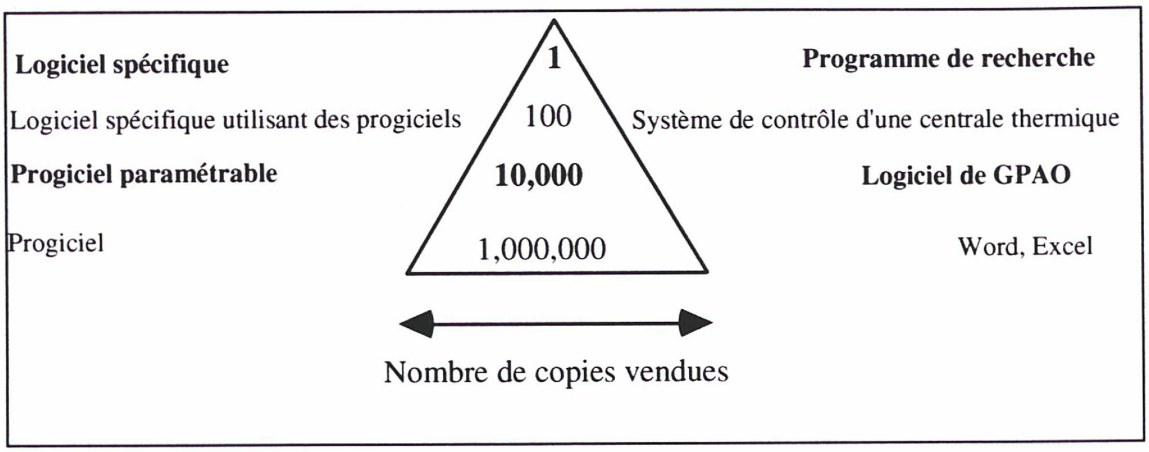

Figure 6 : classification des logiciels en fonction de leur diffusion.

Est-on un progiciel au delà de un exemplaire ? de 10 ? De 1000 ? Il s'agit donc de caractériser plus finement ce qu'on entend par logiciel spécifique et progiciel.

• Un logiciel spécifique performant mais à faible durée de vie

Un logiciel spécifique, appelé aussi logiciel sur mesure, est un logiciel réalisé en un exemplaire par une SSII ou une direction informatique pour un client particulier avec lequel elle passe un contrat.

L'énorme avantage de ce genre de logiciel est de répondre le mieux possible aux besoins des utilisateurs, car les concepteurs du logiciel et les utilisateurs sont en contact, face à face et peuvent communiquer. Le sur mesure permet également de remplir des exigences particulières en matière de performance ou de sécurité.

Mais son inconvénient principal est d'être inutilisable à moyen terme, car personne ne le fait régulièrement évoluer en profondeur. C'est trop dangereux, ça coûte cher, le logiciel marche, donc on le maintient juste, mais progressivement l'entreprise, qui a commandé ce logiciel spécifique, perd la connaissance du contenu du logiciel. Exemple parmi tant d'autres, La Poste a un logiciel de gestion des ressources humaines, qui parfois refuse la nomination de telle personne à tel poste et on ne sait pas pourquoi.

• Un progiciel général mais attractif et suivi

Les progiciels ou logiciels standards sont réalisés à des milliers, voire des millions d'exemplaires par un éditeur de logiciels et pour une masse d'utilisateurs non identifiés. Le progiciel est un produit qui vit. À partir du moment où on vend beaucoup d'exemplaires du même logiciel, il devient rentable pour l'éditeur de logiciel, de consacrer une équipe pour réaliser, maintenir et faire évoluer ce logiciel.

Au cours de ses évolutions le progiciel capitalise de l'expérience. Comment un logiciel de CAO fait maison peut-il être compétitif face au progiciel Euclide développé par

Matra Datavision et qui représente un investissement de plus de 1400 années hommes ?

La capitalisation d'expérience et la maintenance associée au progiciel sont les deux forces du progiciel. Il reste que le progiciel ne répond qu'aux besoins standard des utilisateurs. Mais d'une part les progiciels couvrent des domaines toujours plus vastes et d'autre part les utilisateurs préfèrent être moins spécifiques et comme EDF adapter leurs besoins à ce qui existe sur le marché.

4.4 Ne plus faire d'informatique ou l'informatique transparente

Cette préférence quasi unanime du progiciel reflète moins la peur d'une catastrophe logicielle, que l'envie de ne plus faire d'informatique et de se concentrer sur son métier.

• L'informatique : une boite noire facile d'utilisation

Commençons par une image. En 1920, pour conduire une voiture il fallait être un virtuose en mécanique, car les voitures tombaient en panne à cette époque tous les cent kilomètres. Aujourd'hui le CAP en mécanique n'est pas requis pour obtenir son permis Véhicules Légers. Nous exprimons par cette image le fait qu'aujourd'hui on n'a plus besoin de taper des lignes de code en fortran ou C pour tirer parti de son informatique. L'informatique, au sens de la ligne de code, devient transparente.

Ilog, éditeur de logiciels, propose à l'utilisateur une boite à outils. Celui-ci avec sa souris choisit des boutons, des voyants, des graphes dans cette boite à outils et se constitue un tableau de bord qu'il relie à ses données. Ce tableau de bord peut l'aider ainsi à superviser le réseau d'automates qui fournit ces données et accomplir des tâches complexes comme contrôler une centrale nucléaire. Et ce sans jamais taper une seule ligne de code.

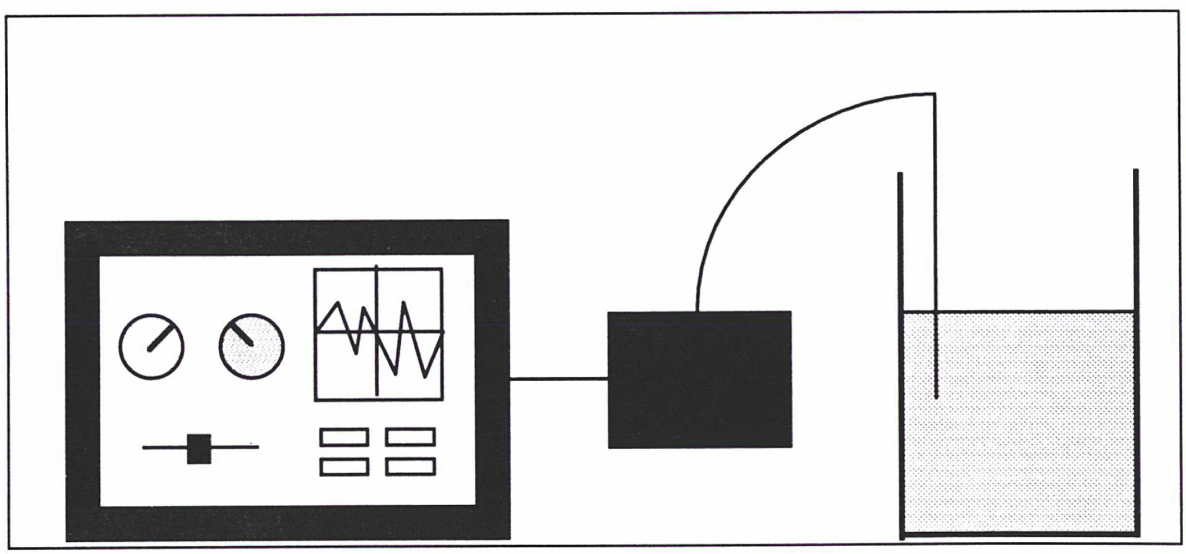

Figure 7 : supervision d'une cuve à partir d'un tableau de bord et à travers une boite noire constituée d'automatismes et d'informatique.

C'est un peu ce que tout un chacun réalise avec les progiciels Power Point, Excel et Word: des objets informatiquement évolués sans taper une seule ligne de code. "L'interface graphique cache à l'utilisateur tous les détails liés au fonctionnement de l'ordinateur"[9]. L'informatique devient une boite noire, elle fournit à l'utilisateur des données qu'il met en forme et elle reçoit des commandes de l'utilisateur, qu'elle transmet au système sous surveillance.

Bien sûr, on pourrait nous objecter que toutes ces applications sont surfaciques, que ce n'est que du dessin, des frises autour du coeur, la boite noire, qui reste de l'informatique pure.

• Programmer sans taper une ligne de code

Mais même la boite noire peut être programmée sans faire de l'informatique. Verilog, autre éditeur de logiciels, a vendu à l'Aérospatiale un outil de génie logiciel qui génère automatiquement du code à partir de spécifications exprimées à l'aide de schémas logiques.

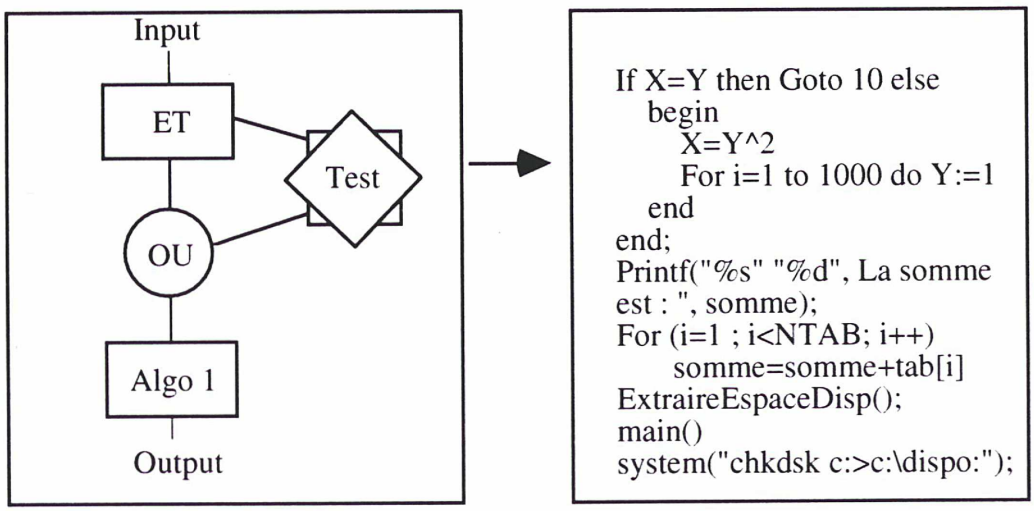

Figure 8 : l'informatique transparente. Génération automatique de code à partir de spécifications écrites dans du langage d'automaticien.

L'ingénieur de l'Aérospatiale remplit les diagrammes avec des Inputs, des Outputs, des opérateurs binaires, des fonctions plus évoluées... C'est un langage d'automaticien. Et le logiciel de Verilog traduit automatiquement ce schéma en code, en langage d'informaticien.

Cet outil a ainsi permis de réaliser le système de pilotage automatique de l'Airbus A320. Le volume de logiciel embarqué par l'A320 est 3 fois plus important que celui de l'A310 - avion non massivement informatisé -, mais le nombre de bugs a été divisé par 3. Les ingénieurs de l'Aérospatiale ont fabriqué 150 fonctions élémentaires, chacune commandant un automate ou un dispositif élémentaire de l'Airbus. Puis ils ont réalisé plus de 3000 schémas combinant ces fonctions élémentaires et décrivant

[9] = "Les nouvelles clés de l'informatique" par Bernard Sauteur et Jean-Marie Desaintquentin. Avril 1996. Les Éditions Synthèse Informatique.

comment ces différents automates sont reliés. Le logiciel de Verilog s'est ensuite chargé de générer automatiquement le code à partir de ces schémas.

Cet outil est triplement révolutionnaire. Premièrement les ingénieurs de l'Aérospatiale programment le réseau d'automates pilotant l'Airbus à l'aide de leur langage, un langage d'automaticien. Ils peuvent ainsi se concentrer sur ce qu'ils veulent obtenir, sur les spécifications, qu'ils visualisent en un seul coup d'oeil et pas sur le code, la traduction. Deuxièmement, ce langage de schémas et de diagrammes obéit à une sémantique rigoureuse, ce qui permet de détecter automatiquement certains oublis et certaines incohérences des spécifications. Troisièmement, on peut modifier les spécifications et regénérer le code automatiquement sans effort.

4.5 Limites du mythe

Cette tendance nouvelle - l'emploi de progiciels - n'est pas exempte de dangers pour les utilisateurs de progiciels. Le club informatique des grandes entreprises françaises (CIGREF) a publié en 1995 un intéressant rapport concernant l'utilisation d'un progiciel en remplacement d'une application spécifique[10].

• Intégration non aisée du progiciel

Il en ressort que le progiciel pose des difficultés d'intégration dans le système d'information de l'entreprise utilisatrice. "Le progiciel dispose d'un vocabulaire qui peut être différent de celui du dictionnaire de données de l'entreprise"[11]. Aussi pour rendre le progiciel conforme ou du moins interfaçable avec le système d'information, les informaticiens de l'entreprise utilisatrice doivent pouvoir maîtriser à la fois le paramétrage du progiciel mais aussi son fonctionnement technique.

Ainsi pour pouvoir faire communiquer son progiciel allemand de contrôle commande avec la partie hardware du système de contrôle commande de la centrale nucléaire de Chooz, EDF a relu 10% des 200 000 instructions assembleurs contenues dans ce progiciel. Le recours à un progiciel n'exclut donc pas des développements spécifiques associés qui font que le projet de refonte ne se résume pas à l'achat d'un progiciel et que "le coût d'appropriation du progiciel est largement supérieur au prix d'achat proprement dit"[12].

Néanmoins, le progiciel, en tant que noyau de la nouvelle application, donne un cadre structurant au projet de refonte de l'ancienne application spécifique. Par ailleurs, même si à l'achat, le progiciel ne couvre pas tout les besoins des utilisateurs ou ne tourne pas encore sur le système d'information de l'entreprise utilisatrice, les utilisateurs peuvent d'une part le visualiser et savoir si c'est ce qu'ils veulent, et d'autre part conduire à vue son paramétrage.

---

[10] = Rapport final "Conduite de Projet" - Septembre 1995 - réalisé par le CIGREF, 21 avenue de Messine Paris VIII.

[11] = Ibidem.

[12] = Ibidem.

• **Poids de l'existant**

L'utilisateur peut se plaindre de son progiciel qui s'intègre difficilement dans le système d'information de son entreprise, mais il pourrait tout aussi bien pester contre ce système d'information qui n'a pas évolué ou est fait de couches superposées tenues avec du scotch. De nombreuses grandes entreprises en France et ailleurs reposent encore sur de vieux systèmes d'informations maintenus par des bataillons d'informaticiens à coups de Cobol ou d'assembleur. Citons le Crédit Lyonnais et ses 800 informaticiens chargés de maintenir à bout de bras son système informatique.

Avec le temps, ces dinosaures ont constitué de gigantesques et complexes réseaux d'applications imbriquées et dépassées. Pour l'instant, les progiciels ne peuvent concerner que les écailles de leur carapace, l'intérieur étant alimenté en développements spécifiques. Mais ces grands systèmes, tels des châteaux de cartes s'écrouleront. Les pièces détachées des ordinateurs les supportant commencent déjà à se faire rares.

• **Dépendance de l'éditeur**

Mais le principal reproche que le CIGREF dans ce rapport adresse aux progiciels, c'est qu'il les rend trop dépendants de leurs fournisseurs. Bien souvent pour ne pas se faire pirater son progiciel ou son savoir faire, l'éditeur refuse de communiquer le code source à ses clients. Or l'éditeur peut mourir et ne plus assurer l'évolution et la maintenance de ses produits.

Cette éventualité a conduit l'Aérospatiale à exiger de Verilog, un de ses fournisseurs de progiciel, de déposer dans un coffre à la banque en présence d'un huissier le code source. De plus l'éditeur fait évoluer son progiciel, sort de nouvelles versions qui peuvent être source de nouvelles erreurs ou n'être plus compatibles avec l'environnement du progiciel propre à l'utilisateur.

Le progiciel n'est pas une recette miracle. Il ne suffit pas, comme le téléphone, de le brancher à une prise pour qu'il fonctionne. D'abord ni la prise - le système d'information de l'entreprise - ni le téléphone - le progiciel - ne sont encore standardisés. Et puis ces inconvénients sont à relativiser. La dépendance vis à vis d'un éditeur de progiciel est beaucoup plus faible que celle vis à vis d'une SSII qui aurait développé le logiciel spécifique correspondant.

Ainsi le progiciel, ce n'est pas la panacée, il faut l'intégrer à un environnement informatique souvent hostile, mais il soulage les utilisateurs. Il leur permet de se concentrer sur leur métier et ils développent de moins en moins de logiciels spécifiques, de moins en moins de lignes de code. Mais il faut bien des informaticiens pour réaliser les progiciels, comment font-ils pour les réussir, comment font-ils pour les rendre fiables ?

5. Comment réussir un grand progiciel ?

5.1 S'organiser en professionnel

• S'armer de méthode et de rigueur

"En physique, tout travail n'est pas utile : le mouvement désordonné des molécules de vapeur d'eau ne produit de travail utile que s'il est dirigé par le cylindre du piston"[13]. Ce que nous voulons exprimer par là, c'est que bien sûr la difficulté du logiciel c'est sa complexité et toutes les procédures et autres manuels d'assurance qualité ne la diminueront pas. Mais sans discipline, ni organisation du travail, le logiciel est voué à l'échec.

Cet échec est d'autant plus assuré que le projet informatique est important. Comment en effet coordonner et faire converger la centaine d'informaticiens travaillant en parallèle sur le projet, dont les milliers de bouts de code doivent pouvoir communiquer et former un tout harmonieux ? Cette question, les militaires se la sont longuement posée quand ils ont voulu construire leur force de frappe nucléaire et les millions de lignes de code associés.

Ces grands programmes militaires vont enfanter des méthodes, des normes qui vont se propager dans le domaine civil. Ainsi, en France, les normes RG-AÉRO, conçues par les militaires du secteur de l'aéronautique ont inspiré de nombreuses méthodes des années 80 (Merise, SDM, MCP-MGP...). Ces méthodes sont des méthodes dites en cascade et sont basées sur le cycle de vie en V. Le processus de production du logiciel est découpé en phases successives et pour chaque phase sont définis les activités à réaliser, les produits d'entrées et de sorties, les responsabilités et les vérifications associées aux activités et produits.

Ces méthodes introduisent beaucoup de rigueur dans la réalisation du logiciel, mais sont très lourdes. L'ensemble des documents issus de la méthode MODAL, qu'a conçue Cégélec pour tous ses services informatiques, remplit un CD-ROM ! Avec ces méthodes, le projet dure longtemps. Il faut passer beaucoup de temps lors de la phase de spécifications, pour être sûr que les utilisateurs aient bien définis correctement tous leurs besoins au départ, car ceux-ci ne verront le logiciel qu'à la fin du projet.

Monsieur Hauet, Directeur Général de Cégélec, ne parle de la méthode MODAL qu'en termes élogieux. Elle a apporté de la rigueur aux informaticiens de Cégélec et surtout elle a permis d'uniformiser leurs pratiques, leurs façons de développer, ce qui est capital, car les informaticiens d'un même projet sont sans cesse en train d'interagir. Ceci expliquerait, selon M. Hauet, pourquoi, depuis l'affaire du système de contrôle commande de Chooz, Cégélec n'a plus connu d'échec dans les projets d'envergure similaire, qui ont suivi Chooz.

---

[13] = "Le développement de grands logiciels ou la quête de l'entropie zéro" par J.P. Bourguignon ; Revue Réalités industrielles de Mai 1991. Une série des annales des Mines.

• **Contrôler ses informaticiens**

En fait, plus que la méthode, c'est la prise de conscience par la direction des fournisseurs de logiciels du risque représenté par le développement de logiciel, qui explique pourquoi des sociétés comme Cégélec ont redressé la tête. Cette prise de conscience s'est traduite par un renforcement du contrôle de ce que fait l'informaticien.

Chez Cégélec, tout projet démarre par une analyse de faisabilité. Chaque phase du processus de fabrication de logiciel se termine par une revue devant des membres de la direction et des experts mandatés par cette dernière. EDF lance des audits sur ses propres projets et l'auditeur est responsable des erreurs et des dérives qu'il n'a pas décelés. Chez Matra Datavision, on va jusqu'à lancer des programmes d'analyse automatique de code pour vérifier que les règles de programmation ont été suivies.

Chez Cégélec, à la fin du codage on demande à un autre informaticien d'effectuer une revue de code, qu'il signe. Une autre pratique courante est de faire tester le logiciel par l'équipe qui en assurera la maintenance ou de le faire tester par une société spécialisée dans le test. L'important étant de séparer les activités de tests et de développement pour pouvoir augmenter le taux de couverture de test et par là le nombre de bugs trouvés.

• **Certifier le procédé de fabrication du logiciel**

Ce qui rend le développement d'un logiciel si difficilement maîtrisable, c'est la caractéristique du logiciel d'être invisible, immatériel. On sait mesurer la qualité d'une soudure rien qu'en la regardant, en revanche le logiciel ne pèse rien, n'a pas de dimensions.

Le logiciel est invisible, mais on peut rendre visible le procédé de fabrication du logiciel. Les méthodes comme Merise et toutes les bonnes pratiques, que nous avons vues plus haut visent à vaincre l'obscurité ou à rendre tangible, visible le logiciel. Les méthodes permettent d'ordonner ces bonnes pratiques et apportent de la rigueur à l'équipe de développement. Mais il existe aussi un nouvel outil, un modèle qui permet d'évaluer la mise en pratique de ses méthodes et d'améliorer la qualité du processus de fabrication du logiciel.

Ce modèle s'appelle le Capability Maturity Model ou CMM. Il a été conçu en 1987 par les informaticiens du Department of Defense américain. Ce dernier est le plus gros consommateur de logiciels au monde et il est sans cesse à la recherche de langages, méthodes, modèles pour améliorer la qualité des logiciels qu'il achète.

Avez-vous un outil de gestion de configuration ? Avez-vous défini une procédure de modification des spécifications ? Qui contrôle la tenue du planning ? Avez-vous planifié les activités de l'assurance qualité ? ... Des auditeurs CMM posent ainsi plus d'une centaine de questions à une quarantaine d'employés du fournisseur de logiciels. Ces employés sont développeurs, chefs de projet ou même directeurs généraux.

Suivant les réponses détaillées des interviewés, le processus de fabrication du logiciel se voit attribuer un niveau qualifiant son niveau de maturité. La note ou le niveau 1

correspond au chaos : tout le monde travaille dans son coin. Pour atteindre le niveau 2, il faut savoir gérer un projet et avoir mis en place des procédures d'assurance qualité. Au niveau 3, on maîtrise les techniques et méthodes du génie logiciel. Pour passer au niveau 4, il faut être capable de mesurer et quantifier à la fois le logiciel et son processus de fabrication. Enfin au niveau 5, on arrive à optimiser les mesures précédentes.

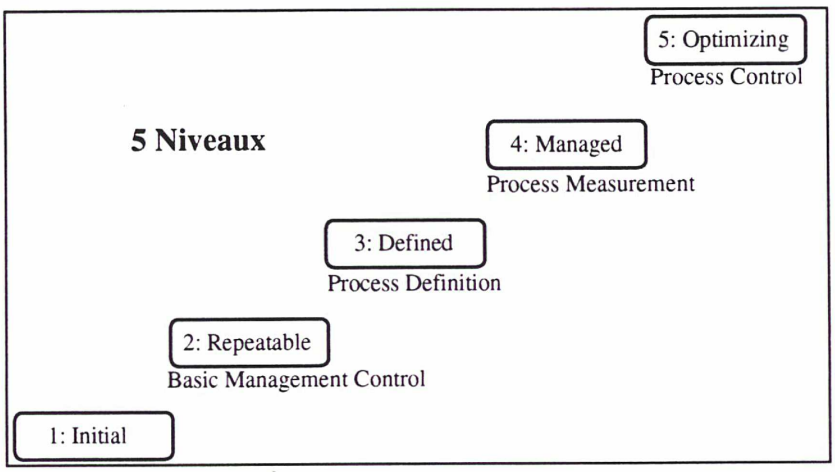

Figure 9 : Les 5 niveaux de maturité du Capability Maturity Model.

D'après des entreprises comme Matra Datavision et Verilog, qui ont expérimenté l'évaluation CMM, ce modèle a trois avantages sur les innombrables méthodes qui peuplent l'univers informatique.

D'abord ce modèle parle le langage des informaticiens, il pose les bonnes questions sans vouloir, comme c'est bien souvent le cas avec les normes ISO 9000, imposer une vérité. Ensuite il synthétise toutes les bonnes pratiques en vigueur chez les as du logiciel et surtout il les classe. Il est illusoire pour un joueur de tennis de vouloir s'améliorer dans tous les compartiments de jeu à la fois. Avec CMM, c'est pareil. Pour passer du niveau n au niveau n+1, il faut appliquer tel lot de pratiques. Inutile de vouloir mesurer une couverture de test du logiciel, quand on ne sait pas gérer un projet. Enfin dernier avantage, le modèle CMM est en train de devenir un standard de fait mondial.

Le modèle CMM est supporté par le Department of Defense, qui exige de ses fournisseurs d'être au moins CMM niveau 2. Boeing, Motorola ont des exigences similaires. En France, la contagion est plus lente. EDF et Dassault Systèmes n'en ont jamais entendu parler. Bull, Thomson, Verilog et SNECMA se sont déjà faits évaluer. Cégélec et Matra Datavision sont au niveau 2. Une vague de fonds se prépare et elle est susceptible d'avoir la même portée dans l'Industrie du logiciel que l'ISO 9000 dans les usines. Sans la certification CMM point de salut.

- **Les outils**

Il existe tout de même des outils qui facilitent la réalisation de logiciels. Citons des langages de programmation très rigoureux comme ADA. Même avec des langages structurés comme Pascal, on ne peut éviter dans de grands logiciels l'interférence entre les variables globales utilisées qui engendre de nombreux bugs dans un programme.

Or avec ADA, "les variables globales ne sont visibles que par un sous-ensemble parfaitement identifié de sous-programmes qui sont les seuls à les manipuler"[14].

Ainsi avant de modifier une variable globale, on voit, dans son sous-ensemble associé, quelle est la signification de cette variable pour chacun des sous-programmes utilisant cette variable. On peut aussi se rendre compte facilement si la façon dont un sous-programme utilise une variable peut avoir une incidence sur les autres sous-programmes du sous-ensemble associé[15].

Citons aussi le logiciel Logiscope qui permet de visualiser un programme.

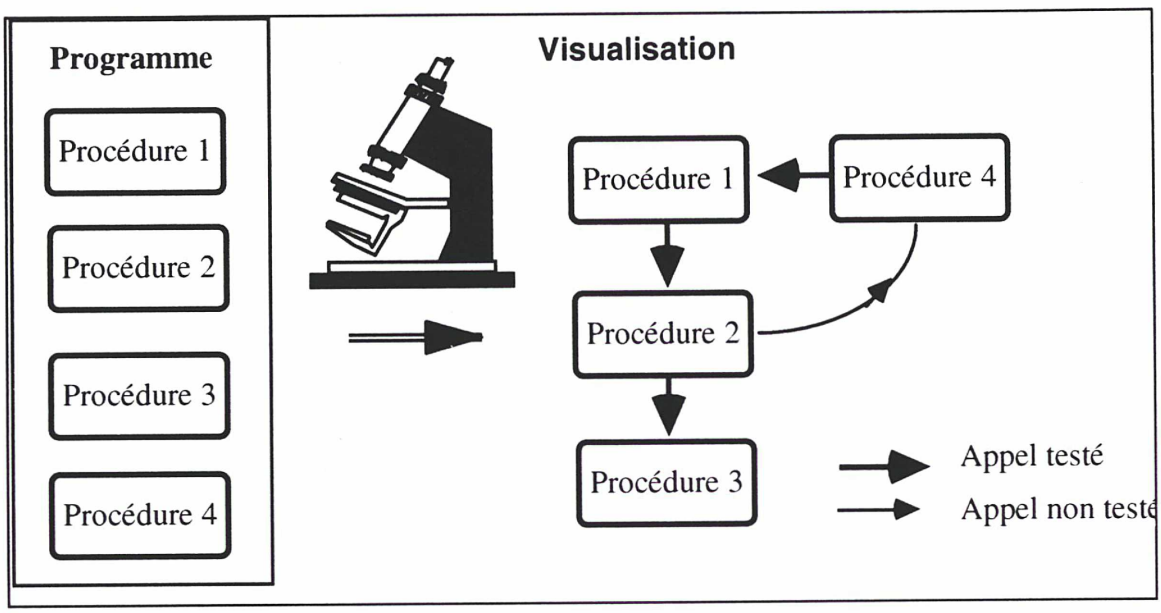

Figure 10: le logiciel Logiscope permet de visualiser les appels entre les différentes procédures d'un programme. Logiscope est réalisé par la société Verilog.

Un programme est fait de procédures écrites séquentiellement et qui interagissent entre elles. Logiscope permet de visualiser ces interactions et ainsi pointe du doigt les procédures trop dangereuses qui interagissent trop et qui doivent être simplifiées. Il pointe aussi du doigt les parties du code, les chemins à l'intérieur du programme, qui n'ont pas été testés. Le logiciel est invisible, mais on peut le rendre visible à l'aide d'invisible, à l'aide d'autres logiciels.

L'informaticien a à sa disposition de plus en plus d'outils lui facilitant la tâche. Les méthodes lourdes ci-avant mentionnées sont allégées par l'existence de logiciels associés qui automatisent toutes les tâches répétitives et rendent plus clair l'usage de la méthode.

Ainsi à la méthode IEM vendue par Texas Instrument est associé un outil : IEF. Grâce à celui-ci, le chef de projet peut générer des diagrammes représentant les différents flux de données de son programme, qui sont décrits par la méthode IEM. Il peut à l'aide de sa souris et en manipulant ces diagrammes créer des fichiers et des bases de

---

[14] = Issu du livre "Réutilisation du logiciel" écrit par Bernard Coulange, édition Masson, 1996.

[15] = Ibidem.

données qui sont automatiquement reliés entre eux et qui vont constituer l'ossature du programme.

5.2 Utiliser des composants élémentaires

Mais une autre façon de rendre plus fiable son logiciel, de se simplifier la tâche, c'est d'éviter de chaque fois réinventer la roue et de répéter les mêmes erreurs.

• Construire une base de composants maison

Dans une société d'édition de logiciels, des dizaines de logiciels sont en chantier. Mais tous ces logiciels ont besoin de définir une interface, de faire des calculs. Et donc tous ces logiciels font appels à des bouts de code identiques. Dans le cas de deux versions successives d'un même logiciel cette partie commune peut atteindre 90% du code.

On pourrait dupliquer les fonctions ou parties communes entre les logiciels en faisant du copier-coller. Mais non seulement on risque d'introduire des erreurs car les lignes de code qu'on a dupliquées dépendent d'un contexte et font appel à d'autres parties du code. Mais aussi on ne bénéficie pas automatiquement des améliorations futures du bout de code qu'on a dupliqué. Donc on rajoute des bugs, on gagne un peu de temps, mais on pourrait en gagner plus.

D'où l'idée de repérer à l'intérieur de tous les logiciels que fabrique l'éditeur, les bouts de code qui sont appelés de nombreuses fois. Il suffit ensuite de les transformer en composant, c'est à dire de les rendre autonomes, sans interaction avec d'autres parties du code. On réalise ainsi une base de composants maison à usage interne. Il ne manque plus qu'une équipe chargée de maintenir ces composants, de les faire évoluer et de les faire connaître aux autres équipes de l'éditeur de logiciel.

Les composants, tels que je les ai décrits, existent depuis longtemps, en fait depuis l'invention des langages structurés de type Pascal. Mais ils sont devenus vraiment attractifs quand, grâce aux langages orientés objet, on a pu les rendre ajustables très facilement. L'inconvénient d'une procédure de langage Pascal, c'est que bien que paramètrable, on ne peut simplement lui rajouter ou enlever des fonctionnalités sinon en la récrivant complètement.

Mais avec les langages orientés objet, une nouvelle technique dite de l'héritage simplifie la donne. Prenons un exemple. Si vous avez dans votre bibliothèque un composant animal qui mange, boit et dort et que vous voulez un kangourou qui mange, boit, dort et saute. Eh bien vous pouvez très facilement créer votre composant kangourou en déclarant hériter du composant animal et en définissant seulement votre fonction "sauter".

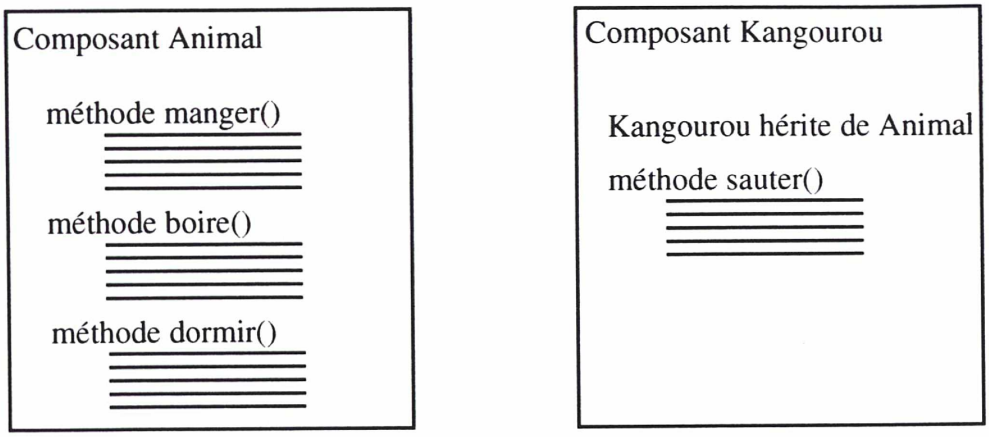

Figure 11: lignes de code nécessaires pour définir les trois composants Animal, Chien et Kangourou. Exemple tiré du livre "Réutilisation du logiciel" écrit par Bernard Coulange, édition Masson, 1996.

La société Verilog a mis en pratique ces idées, a constitué une bibliothèque de composants maison et elle peut se vanter d'avoir eu un taux d'utilisation de ses composants maison compris entre 50 et 70% dans plusieurs de ses projets[16]. Cela signifie que plus de la moitié de ses logiciels est faite de composants fiables qu'elle n'a pas eu à réécrire ou à retester. C'est tout autant de bugs et de temps de développement et de maintenance gagné.

• Faire son marché et assembler

Mais des composants logiciels, bien souvent ce n'est pas la peine de les fabriquer soi-même : on en trouve sur le marché. On trouve des bibliothèques de fonctions mathématiques, comme Mathlab, on trouve des interfaces toutes faites, des composants élémentaires qui calculent la date du jour, ou aident à concevoir l'architecture de son logiciel.

Les composants logiciels qui sont en vente se répartissent suivant trois catégories : des composants techniques appelés "widget" dits de bas niveaux qui s'intègrent dans le code d'une application, des objets génériques qui sont indépendants et visualisables graphiquement et des objets métiers qui sont eux-aussi visualisables mais qui sont destinés à un secteur d'activités particulier.

D'après Daniel Levy, directeur de SOS Développeurs, on trouve sur le marché plus de 600 widgets. Ils effectuent des fonctionnalités des couches de bas niveaux et facilitent le développement d'applications. Ils permettent de faire des "conversions de format, des communications de type fax ou TCP/IP, des compressions de documents, de la gestion d'images, du multimédia, des fonctions hypertextes, du dessin, de l'accès aux données, pour ne donner que quelques exemples"[17].

Les objets génériques sont destinés à tous les secteurs d'activités et comprennent les tableurs, les éditeurs de graphes ou les accès aux bases de données. Quant aux objets métiers, Lyon Consultants, filiale du Crédit Lyonnais, propose un catalogue de plus de

---

[16] = Ibidem.

[17] = Dossier Génie Logiciel : "Composants réutilisables : l'offre se structure", Le Monde Informatique du ? 1995.

1500 composants dans le domaine de la finance. SOCS commercialise des bibliothèques d'objets du métier de l'assurance[18].

Les objets génériques et métiers sont assemblés grâce à un logiciel atelier fourni par l'éditeur. Matra Datavision propose ainsi le logiciel Euclide, qui est un "framework" ou une plate-forme de travail - espèce d'établi - avec une boite à outils permettant à l'utilisateur de construire son application.

Le marché des composants n'en est qu'à ses débuts. Bertrand Meyer, inventeur du langage Eiffel, affirme que "l'internet rend possible une industrie des composants à grande échelle"[19]. Internet, c'est bien sûr pour l'éditeur une vitrine, un outil marketing fabuleux et peu onéreux pour les éditeurs de logiciel permettant une livraison ultra rapide par téléchargement et une maintenance en temps réel. Mais Internet, c'est aussi une gigantesque base de données et bientôt une gigantesque base de composants, que l'utilisateur va pouvoir trier, tester et acheter. Un hypermarché des composants logiciels est en cours de construction.

Finalement les informaticiens des éditeurs de logiciel, tout comme ceux des utilisateurs, écrivent de moins en moins de lignes et leur travail consiste de plus en plus à assembler des composants tout faits. Ces composants, parce qu'ils sont suivis et utilisés par une multitude de clients, sont fiables. Ainsi, on change d'échelle, on change de granularité, on réalise son grand progiciel non plus à l'aide de lignes de codes sémantiquement fiables, mais à l'aide de composants fiables.

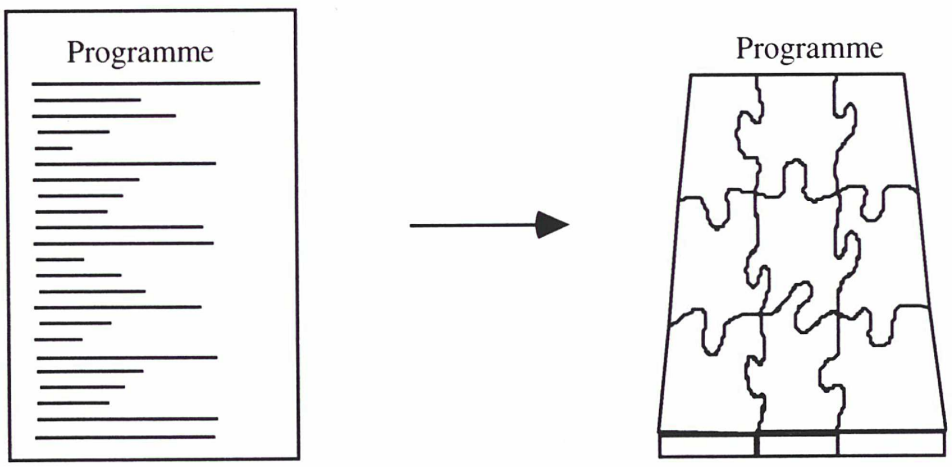

Figure 12: Changement de granularité. Un programme n'est plus fait de lignes de code mais devient un assemblage de composants

La brique de base est de plus en plus fiable, mais le problème s'est décalé d'un cran. Il s'agit maintenant de mettre ensemble ces briques, qui ont des provenances diverses, de les faire communiquer. Je gère mes données avec Oracle, je fais des calculs avec Mathlab, je visualise les résultats avec Ilogviews. Il faut pour cela qu'Oracle, Mathlab et Ilog adoptent le même format de données. C'est un problème de normalisation, qui est en cours, mais n'est pour le moment que partiellement résolu.

---

[18] = Dossier Composants Logiciels: "La vente sur catalogue fait son nid... petit à petit", Le Monde Informatique du 29 Mars 1996.

[19] = "Internet et les objets vont révolutionner le logiciel", Le Monde Informatique du 22 Mars 1996.

• **À la recherche d'un standard de communication**

L'Object Management Group réunit des constructeurs de matériel informatique et des éditeurs de logiciels comme SSA. Cette association est sur le point de définir une norme appelée CORBA contenant les spécifications des technologies standard pour la gestion des objets. Tous les composants logiciel construits en suivant ces spécifications pourraient alors communiquer et ce quelque soit le constructeur.

Cette approche générale, multi acteurs et multi-directionnelle est très ambitieuse, mais elle se heurte à des intérêts économiques de taille et on ne voit que peu de réalisations concrètes en résulter. En fait, il y a plus à attendre du côté des gros acteurs comme Microsoft ou SAP.

Ceux-ci, en raison de leur poids économique et du succès de leurs logiciels - Windows pour Microsoft et R/3 pour SAP -, attirent d'autres éditeurs qui développent des logiciels compatibles avec les premiers. Ainsi on peut enrichir le logiciel R/3 de gestion d'entreprise de SAP par le module d'inventaire de Marcam ou le logiciel de comptabilité générale proposé par Coda. Cette compatibilité est obtenue grâce au respect de spécifications d'échanges entre objets.

Mais le petit monde de SAP ne pèse pas lourd face à la galaxie d'éditeurs qui tournent autour de Microsoft. On dénombrait en 1992, plus de 6000 progiciels tournant sur Windows. Microsoft a imposé à ces éditeurs son standard d'échange entre objets sous Windows. Il s'appelle OLE, pour Object Linking and Embedding et permet ainsi de réaliser des documents composites comprenant du texte issu de Word, des tableaux issus d'Excel ou des images issues de Powerpoint. Et ce n'est pas du simple copier-coller puisque un double-clic de la souris sur ces objets nous les renvoie dans leur logiciel d'origine, où ils vont pouvoir être modifiés et mis à jour[20].

Les grands utilisateurs font également pression pour faire communiquer leurs différentes applications informatiques. Les grands groupes pétroliers, les constructeurs d'automobiles imposent des normes, comme STEP qui normalise la forme des données utilisées par leurs différents logiciels. Ainsi du design à la production en passant par l'épreuve des calculs de résistance, la voiture virtuelle est conçue, optimisée, renseignée et ce par échange de fichiers informatiques manipulés par des logiciels différents.

5.3 Faire vivre le progiciel

• **Utiliser les millions d'utilisateurs**

Des méthodes, des outils et des composants prêts à l'usage et qui communiquent : on est de mieux en mieux armé pour affronter la complexité du logiciel. Mais pour rendre fiable son progiciel, l'informaticien a un autre allié : les utilisateurs de ce progiciel.

Revenons sur la complexité du logiciel. On nous a dit qu'il était impossible de tester de façon exhaustive un grand logiciel. La combinatoire des états possibles de

---

[20] = Exemple tiré de "Les nouvelles clés de l'informatique" par Bernard Sauteur et Jean-Marie Desaintquentin. Avril 1996. Les éditions Synthèse Informatique.

l'ordinateur est proportionnelle à l'age de l'univers. Certes la puissance de calcul des processeurs double tous les deux ans, mais cette même puissance de calcul permet de réaliser des programmes toujours plus gros et complexes. Donc elle ne nous aidera pas à mieux couvrir de tests nos programmes.

Mais avec le progiciel, l'éditeur de progiciels peut opposer à cette combinatoire une autre combinatoire : celle des états atteints par les ordinateurs des utilisateurs du progiciel. Quelques millions de personnes qui utilisent pendant quelques années le même progiciel, parcourent et donc testent involontairement une bonne partie du progiciel. L'éditeur du progiciel a donc intérêt à profiter de ces tests gratuits et nombreux. Et tout éditeur qui veut durer a un service de maintenance, une hotline, qui est à l'écoute des utilisateurs 24 heures sur 24, qui les aide à se servir du logiciel et en même temps enregistre et traite les problèmes qu'ils ont rencontrés.

Ainsi l'utilisateur peut être l'allié de l'éditeur, il sert à debugger et à rendre fiable le progiciel. Bien sûr si le progiciel est trop buggé, l'utilisateur l'abandonnera au profit du produit concurrent. Bien sûr certains grands logiciels comme le système de contrôle d'une centrale nucléaire ont trop peu d'utilisateurs, ou sont trop sensibles, pour que l'informaticien se repose sur des essais grandeur nature. Mais mêmes ces grands logiciels s'appuient sur des briques, sur des composants plus généraux, sur des progiciels qui, eux, profitent de ce déverminage.

• Innover sans cesse ...

Le progiciel vit, évolue grâce à un équipe dédiée de l'éditeur de logiciels, qui en corrige les bugs et rajoute de nouvelles fonctionnalités répondant aux nouveaux besoins des utilisateurs que le service marketing a perçu. Pour séduire le client, l'éditeur doit sans cesse innover. Chaque nouvelle version doit apporter un plus et ce plus est obtenu en investissant massivement en Recherche et Développement. Des éditeurs comme Ilog ou Informix consacrent plus de 20% de leur chiffre d'affaires en R&D.

Cependant l'enrichissement successif d'un produit, cette gestion par version a un inconvénient : l'équipe qui est en charge du produit est prisonnière des versions précédentes. Elle doit garantir à ses clients une compatibilité ascendante : tous les fichiers réalisés avec l'ancienne version du produit doivent pouvoir être lus par la nouvelle version. Non seulement ces contraintes conduisent à bâtir une usine à gaz et des logiciels lourds qui pèsent plusieurs millions de lignes de code, mais aussi elles empêchent l'éditeur d'utiliser les nouvelles technologies.

À court terme cette continuité dans le changement est rassurant pour la direction générale, elle limite notamment les velléités des informaticiens, mais à moyen terme les performances du logiciel baissent et périodiquement l'éditeur doit réagir en profondeur. On voit ainsi Matra Datavision, Dassault Systèmes ou Oracle s'ébrouer tous les 5 ans, remettre à plat l'architecture de leur logiciel et lui faire un véritable lifting.

• **... mais innover prudemment**

Une des grandes caractéristiques du logiciel et de l'informatique en général, c'est d'évoluer très vite. Tous les deux ans, la puissance des ordinateurs double à prix constant. IBM était hier le numéro 1 en informatique, aujourd'hui c'est Microsoft, demain ce sera Netscape. Il y a 10 ans, sur les dix premiers éditeurs de logiciels CAO, 5 sont morts. Aujourd'hui sur les dix premiers éditeurs de logiciels CAO, 5 sont à vendre. On meurt de ne pas arriver à se renouveler et à coller aux goûts du public.

Il faut donc à chaque coup frapper juste, bien et rapidement. C'est pourquoi l'équipe chargée de réaliser un produit n'aura pas pour objectif de concevoir un produit complet et parfait, mais de concevoir un produit simple mais innovant et de l'enrichir successivement. Cette gestion par versions, cette stratégie par petits sauts a l'immense avantage de suivre l'évolution des besoins de l'utilisateur et de rectifier le tir au coup suivant.

Plus le saut est petit et plus on a de chances d'arriver à en venir en bout. J.P. Hauet, Directeur Général de Cégélec le martèle haut et fort : "Dans le domaine du logiciel, il est très difficile d'apprécier l'ampleur du travail à réaliser et pour s'en sortir, il faut être modeste dans ses ambitions."

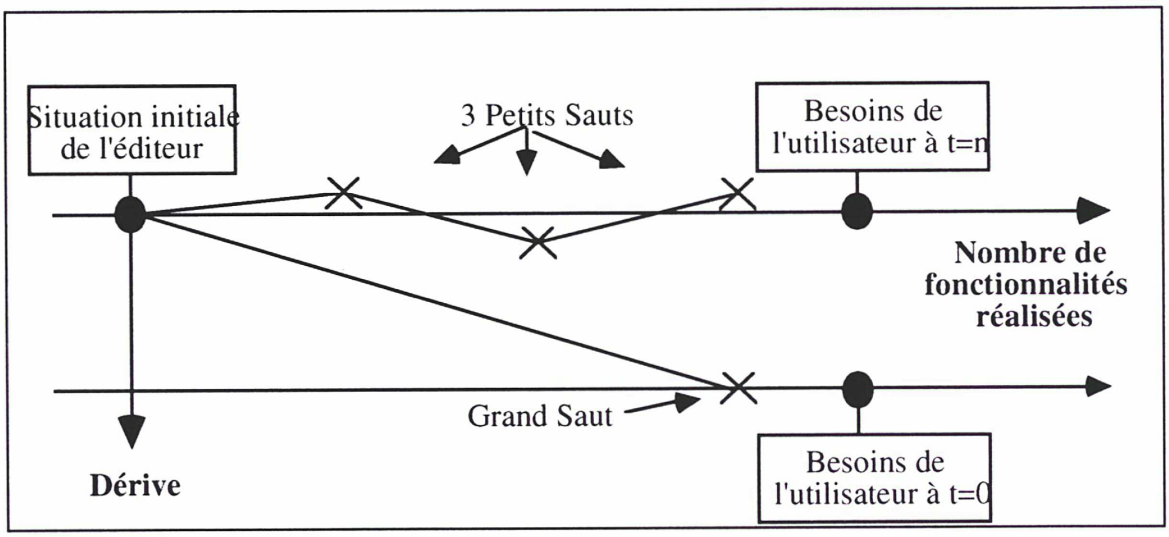

Figure 13: avantage de la gestion par versions et de l'enrichissement successif d'un logiciel par rapport à la fabrication d'un coup d'un seul d'un gros logiciel.

En fait la longueur du saut et donc le nombre de fonctionnalités nouvelles réalisées sont déterminés par la durée du cycle de fabrication du produit. Comme dans une usine, la conception de progiciels est cadencée. Chez Dassault Systèmes, une nouvelle version de leur progiciel de CAO, Catia, est le fruit de plusieurs études et conceptions de sous-produits lancés en parallèle. Tous les six mois, le train part et emmène les nouvelles fonctionnalités.

L'équipe de Cégélec chargée du logiciel de pilotage de réseau électrique a un temps de cycle de fabrication d'environ deux ans et aura mis sur le marché 10 versions de son logiciel entre 1988 à 1998. Ils ont toujours 2 versions en chantier en même temps, l'une en cours de définition et l'autre en cours de développement. Le temps de cycle de Matra Datavision est compris entre 6 mois et 2 ans.

Le "Time to Market" dicte sa loi, il faut frapper avant le concurrent. En fait le cycle de fabrication d'une version n'est pas indéfiniment compressible. D'abord il faut du temps pour innover, ensuite il faut laisser les utilisateurs souffler et s'adapter aux nouvelles fonctionnalités et enfin plus on fait de versions et plus cela coûte cher à l'éditeur pour en assurer la maintenance.

6. Conclusion

Au cours de l'année écoulée, nous nous sommes penchés sur les développements informatiques d'une dizaine d'entreprises françaises. Nous sommes effarés de ce que nous avons pu découvrir dans certaines d'entre elles. Les histoires de bugs et les catastrophes dont parle régulièrement la presse ne sont que la pointe émergée de l'iceberg : les statistiques montrent que seul un projet sur deux parvient à son terme, souvent au prix de dépassements de budget.

Nous sommes cependant convaincus que l'échec n'est pas une fatalité en matière de logiciel.

Nous avons rencontré des entreprises qui réussissent régulièrement leurs projets informatiques, même si tout n'est pas parfait. Mieux encore, nous avons vu une entreprise qui, après un échec retentissant s'est lancée dans une politique volontariste d'amélioration de ses processus de production de logiciel, efforts qui aujourd'hui portent leurs fruits.

Il est donc possible de faire quelque chose.

Améliorer la situation n'a cependant rien de facile ou d'évident. Nous n'avons pas de solution miracle ou de potion magique à proposer pour aplanir toutes les difficultés. Nos entretiens nous ont permis d'identifier deux pistes d'action possibles.

Premièrement, les projets qui réussissent s'insèrent généralement dans un contexte rigoureux. Les premières automobiles ont été construites dans un garage par des bricoleurs de génie, il ne viendrait à personne l'idée de continuer de la même façon aujourd'hui. Il en va de même pour les logiciels : les débuts ont pu se faire avec peu de moyens, beaucoup de créativité et de souplesse, mais, lorsque l'on passe à l'étape industrielle, il ne faut pas hésiter à adopter des méthodes de travail plus rigoureuses pour faire face à la complexité des grands logiciels. Une bonne gestion de projet évite d'échouer sur des problèmes simples et permet de se concentrer sur les vraies difficultés. Nous avons remarqué l'importance particulière des rapports qu'entretiennent la maîtrise d'ouvrage et la maîtrise d'oeuvre (NB : ce qui suppose que ces deux entités soient clairement définies) et les bienfaits de la réflexion au tout début du projet (réunion de lancement de projet, étude de faisabilité préalable,...).

Le développement informatique suppose de plus l'utilisation de méthodologies adaptées. Celles ci ne garantissent pas le succès à coup sûr, mais fournissent un cadre et une aide précieuse qui évitent de partir à l'aventure. Nous n'avons pas cherché à les comparer entre elles, ni à déterminer laquelle était la plus adaptée en fonction du type de travail à effectuer ; certaines méthodologies semblent être en crise du fait de l'évolution de l'informatique. En revanche, nous avons pu remarquer le succès croissant d'un modèle d'évaluation des processus de fabrication des logiciels, le CMM (*Capability Maturity Model*), développé par le *software engineering institute (Carnegie Mellon University)* à la demande du ministère de la défense américain.

La deuxième piste pour améliorer la réalisation des grands logiciels est plus radicale : elle consiste à ne pas en faire. Les grands logiciels ne sont pas un mal nécessaire. La

solution la plus simple consiste à acheter un progiciel. C'est une tendance très forte depuis le début des années 80. L'achat de progiciels ne résout cependant pas tout : ils ne traitent que les problèmes standard. De plus, pour beaucoup, le logiciel est plus que la réponse à un besoin, il est la matérialisation du savoir faire, du métier de l'entreprise, et fait partie intégrante du produit vendu. Dans ces conditions, il est nécessaire de recourir à des développements propres. Cependant, il est bien rare que toutes les fonctions du logiciel à développer soient propres à l'entreprise. Il est normalement possible d'acheter à l'extérieur des composants logiciels pour réaliser simplement toutes sortes de fonctions standards. Il est possible de grandement simplifier un logiciel en évitant de réinventer la roue. Enfin, les fonctions qui ne peuvent être achetées à l'extérieur ne doivent pas être toutes redéveloppées lors de chaque nouveau projet : dans une même entreprise, il existe souvent des points communs entre les différents programmes, qui permettent de développer des composants internes à l'entreprise.

Progiciels, composants achetés ou composants produits en interne permettent de simplifier les grands logiciels et apportent des gains très importants en terme de coût, de délais et de fiabilité.

Construire les logiciels non plus à partir de rien, mais en assemblant des briques et des blocs préexistant nous paraît être la voie qui sera poursuivie pour faire les grands logiciels à l'avenir. Cela change le métier d'informaticien, qui devient un intégrateur de système. Cela change aussi la nature des problèmes à traiter : la principale difficulté est de faire communiquer les différentes parties de l'application entre elles, ce qui passe par des efforts de normalisation et de définition d'interface et de structure de données communes.

Bibliographie

- "No silver bullet ; essence and accident of software engineering", article de F.P. Brooks dans la revue IEEE Computer, vol. 20, No 4, avril 1987, p 10-19.

- "Le développement de grands logiciels ou la quête de l'entropie zéro" par J.P. Bourguignon ; Revue Réalités industrielles de Mai 1991. Une série des annales des Mines.

- "Réutilisation du logiciel" par Bernard Coulange, édition Masson, 1996.

- "Les nouvelles clés de l'informatique" par Bernard Sauteur et Jean-Marie Desaintquentin. Avril 1996. Les Éditions Synthèse Informatique.

- "Analyse générale du marché français - SSII 1998", Juin 1994. Rapport de Pierre Audoin Conseil.

- "Conduite de Projet", Septembre 1995. Rapport réalisé par le CIGREF, 21 avenue de Messine Paris VIII.

- "GAMIX les acteurs du marché informatique et des progiciels", édition 1996. Rapport du CXP.