Chapitre V

LES QUESTIONS CRITIQUES DE L'ECONOMIE DU LOGICIEL

L'économie du logiciel est l'objet depuis ses origines de trois critiques récurrentes : un manque de fiabilité des logiciels produits (résultant soit d'une erreur de comportement du programme dans une situation donnée, soit d'une situation non prévue), une relative inadaptation aux besoins des utilisateurs, et une évolution jugée insuffisante de la productivité dans la production des logiciels avec ses conséquences en terme de coûts et de délais. Les deux premiers aspects expliquent l'importance démesurée prise par la maintenance, que ce soit une maintenance corrective destinée à éliminer les imperfections constatées ou une maintenance adaptative pour améliorer l'adéquation du produit aux attentes des utilisateurs, qui à son tour influe négativement sur la productivité globale du secteur. Vu l'importance prise par les logiciels dans l'économie, "la productivité et la qualité de la programmation (…) sont maintenant des problèmes susceptibles d'affecter les profits et les aptitudes compétitives des entreprises, les délais de livraison des nouveaux produits, la façon dont sont organisées et gérées les entreprises" (Capers Jones, 1989, p. 14).

La réalité de ces problèmes est attestée par la perception des utilisateurs et par l'importance qu'ils occupent dans la littérature spécialisée, qui fourmille d'exemples spectaculaires. Une des explications, fréquemment avancée, est la complexité de l'activité de production de logiciels : "la programmation est ressentie subjectivement comme l'une des plus complexes parmi les activités humaines, et le degré élevé de complexité de la programmation est mentionné comme une raison majeure de sa faible productivité, de ses longs délais et de sa qualité médiocre" (Capers Jones, 1989, p. 91). Les spécificités technico-économiques des logiciels (cf. chapitre III) peuvent également contribuer à expliquer ces problèmes. La résolution de ces problèmes est au cœur du génie logiciel, dont Frederick P. Brooks résume les objectifs : "comment concevoir et construire un ensemble de programmes pour en faire un système ; comment concevoir et construire un programme ou un système pour en faire un produit robuste, testé, documenté et supporté ; comment garder sous son contrôle intellectuel la complexité à larges doses" (1996, p. 249).

Toutefois au-delà du sentiment généralisé d'une production peu efficace, de produits peu fiables et inadaptés[1], l'analyse précise de l'ampleur réelle de ces phénomènes reste à effectuer. En particulier, la persistance de ces problèmes ne signifie pas qu'il n'existe pas des progrès réels dans la production des logiciels, mais plus vraisemblablement que ces progrès restent insuffisants par rapport à l'augmentation de la quantité et à la diversité des logiciels à produire, aux exigences croissantes des utilisateurs, et à la complexité grandissante de ce qui est informatisé. Il importe également de noter que les évolutions à l'œuvre dans l'économie du logiciel sont plurielles, et que, dans certains cas, des améliorations significatives par rapport à une dimension des problèmes (par exemple la fiabilité) peuvent se traduire par une dégradation par rapport à une autre dimension (par exemple la productivité).

Nous examinerons successivement le problème de la productivité dans la production des logiciels (section I), le problème de la fiabilité des logiciels (section II) et le problème de l'adaptation aux besoins des utilisateurs (section III), en analysant les progrès réalisés et les écarts permanents par rapport à l'évolution des besoins, et en soulignant la pluralité des arbitrages réalisés pour privilégier la résolution de certains de ces problèmes et tenter de sortir de "la crise du logiciel". La présence, peut-être excessive, de constats empiriques et d'exemples concrets, se justifie par l'importance des problèmes pratiques, auxquels, à notre avis, l'observateur doit prêter une attention suffisante pour véritablement "comprendre" ce secteur.

[1] "L'avis unanime des gestionnaires et des responsables industriels, des dirigeants des administrations et des chefs militaires est que la programmation est aujourd'hui la moins professionnelle et la plus gênante de toutes les avancées technologiques. La programmation est universellement jugée trop coûteuse, génératrice de trop d'erreurs et beaucoup trop lente. On estime également qu'elle échappe très largement aux mesures, qu'elle est peut-être impossible à mesurer, et que les prévisions y sont inacceptablement imprécises en matière de fiabilité, de consommation de ressources ou de délais" (Capers Jones, 1989, p. 14-15).

Section I - Une amélioration jugée insuffisante de la "productivité" dans la production des logiciels face à l'évolution de la demande

Il s'agit ici de la productivité dans le processus de production des logiciels qu'il faut distinguer de la productivité apportée par l'utilisation des logiciels ou productivité indirecte (cf. chapitre III). Le constat de Jacques Printz, selon lequel "la productivité de la programmation est le problème numéro un du bon usage des ordinateurs" (1998, p. 322), est très largement partagé[2]. En effet, la productivité dans la production de logiciels est réputée se situer à un niveau désespérément bas et croître très lentement (quand elle n'est pas considérée comme étant stagnante). Existe également un accord assez général sur les causes de cette faiblesse de la productivité : part trop faible de produits logiciels fabriqués en série ; faible réemploi de composants déjà disponibles ; automatisation trop faible, parfois même inexistante, de processus de production pourtant généraux et répétitifs ; intégration perpétuelle de nouvelles techniques sur lesquelles l'état des connaissances n'en est parfois qu'aux balbutiements ; complexité grandissante (pas toujours justifiée) des systèmes fabriqués ; utilisation d'un environnement de travail trop artisanal et individualiste ; manque de rigueur de la part de l'ensemble des participants au cycle de développement dans la définition du service et des fonctions des objets à fabriquer ; cycles de décisions trop lents dans les organisations qui définissent les systèmes (Serge Bouchy, 1994, p. 142-143). Cette faiblesse de la productivité du travail est d'autant plus problématique pour le coût des logiciels produits, qu'elle entretient des tensions sur le marché du travail des informaticiens, ce qui est la cause de leurs rémunérations relativement élevées.

Selon William J. Baumol, l'absence d'amélioration de la productivité dans la production des logiciels justifie le classement de l'informatique dans le secteur "à stagnation asymptotique" (A). Pourtant, malgré les difficultés de mesure de la productivité dans une activité comme la production de logiciels, l'examen des différents indicateurs existants montre que la productivité dans la production des logiciels connaît une croissance significative (B),

[2] Par exemple, selon Capers Jones, "la productivité de la programmation est devenue un sujet majeur de préoccupations sur le plan international" (1989, p. 14).

même si elle est plus faible que dans le domaine du matériel informatique et si elle reste insuffisante face à la très forte croissance des besoins (C).

A - LA THESE DE WILLIAM BAUMOL : L'ABSENCE D'AMELIORATION DE LA PRODUCTIVITE ET SES CONSEQUENCES

L'examen critique des modèles de William J. Baumol, conçus pour analyser la dynamique de l'ensemble de l'économie, est nécessaire pour notre étude, en raison de l'impact de ces modèles sur les débats concernant l'évolution de la productivité, et surtout parce qu'une des deux activités, qui est étudiée pour appuyer sa thèse est l'informatique et en son sein les problèmes liés à l'importance prise par la production des logiciels.

Le premier modèle de "croissance déséquilibrée" de William J. Baumol (1967) voulait rendre compte des effets des taux de croissance de la productivité plus faible dans les activités de service. L'économie y était divisée en deux secteurs, un secteur "stagnant" et un secteur "progressif". Le secteur "stagnant" comprenait les activités de service dans lesquelles le travail était l'intrant principal et les gains de productivité faibles. Le secteur "progressif" comprenait des activités de production de biens dans lesquelles l'usage croissant de capital et la mise en œuvre de nouvelles technologies entraînaient des accroissements continus du produit par travailleur et, par conséquent des salaires plus élevés. Comme les salaires plus élevés se propageaient du secteur "progressif" vers le salaire "stagnant", les coûts et les prix dans ce secteur devaient croître de manière continue ("maladie des coûts"). De plus la faiblesse de la croissance de la productivité dans le secteur "stagnant" conduisait à prévoir que ce secteur absorberait progressivement des proportions de plus en plus importantes de l'emploi total, qui, par effet de structure, entrainerait un "déclin du taux de croissance général de la productivité dans l'économie".

Vingt années après, William J. Baumol, Sue Anne Batey Blackman, et Edward N. Wolf (1985) présentent un second modèle dans lequel ils ajoutent un troisième secteur "à stagnation asymptotique". Les activités de ce secteur ont une composante très avancée sur le plan technologique (intrants du secteur progressif) et une composante relativement irréductible fortement intensive en main d'œuvre (intrants du secteur stagnant). De ce fait "le dynamisme de ces activités peut se révéler passager et quelque peu illusoire" (p. 816). En effet, ces activités "démarrent comme des activités de pointe dominées par leur composante technologique à forte productivité, mais à mesure que la main d'œuvre représente une part

croissante des coûts globaux (parce que la composante dynamique, novatrice, entraîne une réduction de ses propres coûts), elles finissent par revêtir les caractéristiques des services stagnants" (idem). L'analyse du secteur asymptotiquement stagnant est basée sur des données relatives à deux activités, l'informatique et la télévision. En ce qui concerne l'informatique, la composante progressive est l'équipement informatique, dont "le coût par unité de puissance de traitement semble avoir chuté d'environ 25 % par an" (p. 813). Le segment ayant les caractéristiques du secteur stagnant est la production de logiciels : "entre temps, le coût des logiciels (à fort coefficient de main d'œuvre) représentait une proportion toujours plus grande du coût global d'un système informatique" ; selon William J. Baumol, Sue Anne Batey Blackman, et Edward N. Wolf, la part du logiciel dans le coût d'un système est passée de 5 % en 1973, à 80 % en 1978 et 90 % en 1980 (p. 813), ceci s'expliquant par le fait que "l'élaboration du logiciel demeure une activité essentiellement artisanale et, jusqu'à présent un service stagnant" (idem).

Ces analyses ont trouvé un écho dans les prévisions alarmistes régulièrement effectuées, mais non confirmées jusqu'à aujourd'hui, d'une pénurie structurelle de programmeurs dépassant les simples tensions conjoncturelles sur le marché du travail effectivement constatées. Par exemple, l'OCDE (1991 A) cite une étude américaine, selon laquelle, si les tendances actuelles concernant le matériel et le logiciel se poursuivaient, en 2040, toute la population des Etats-Unis (hommes, femmes et enfants) devrait écrire des logiciels !

B - UNE CRITIQUE PARTIELLE : UNE CROISSANCE DE LA PRODUCTIVITE…

Les modèles de Baumol ont suscité de nombreux débats, concernant notamment le fait qu'ils sont basés uniquement sur l'offre et qu'ils reposent sur certaines hypothèses discutables : prise en compte d'un seul facteur de production, taux de salaire unique dans l'économie, volume de population active constant…(cf. pour une synthèse récente, Mara C. Harvey, 1998). L'hypothèse de base selon laquelle les services ne connaissent pas de gains de productivité ("stagnants" dans le premier modèle, "stagnants" ou "asymptotiquement stagnants" dans le second modèle) a été critiquée par Jean Gadrey. Cette critique est basée sur la remise en question de l'argument central de l'approche de William Baumol : "dans certains cas, le travail est un instrument indirect d'obtention du produit final, alors que dans d'autres cas, pour diverses raisons pratiques, il est lui-même le produit final" (William J. Baumol, 1967, p. 417) ; en effet, il est évident que si l'on considère que pour certaines activités, le travail constitue le produit final, il ne peut y avoir par définition d'amélioration de la

productivité du travail dans ces activités. Or, il est toujours possible de définir le produit d'une activité en le dissociant du travail fourni, même si la mesure de la quantité de "produits" n'est pas toujours simple à réaliser. En appliquant cette méthode pour de nombreux secteurs de services, ce qui permet d'appréhender l'évolution de la productivité à partir d'indicateurs plus adéquats, Jean Gadrey montre empiriquement que "dans aucun cas on ne pouvait parler de stagnation, même approximative, de la productivité au niveau de l'ensemble du secteur" même pour des secteurs "où l'on pouvait observer la progression du poids relatif des activités ayant les plus fortes dimensions relationnelles et professionnelles interactives" (1996 A, p. 218-219).

De même, il est possible de montrer que contrairement à ce qu'affirme William J. Baumol il existe des gains de productivité dans la production des logiciels. Le raisonnement élémentaire suivant en fournit une confirmation intuitive : certes, les effectifs employés pour la production de logiciels connaissent une croissance importante, mais qui reste sans commune mesure avec l'augmentation du nombre et de la puissance des matériels utilisés, alors même que ces matériels ont des utilisations de plus en plus diverses et requièrent donc de plus en plus de logiciels différents. Il en découle logiquement que la productivité du travail dans la production des logiciels a augmenté ce que nous allons tenter de mettre en évidence à partir de l'examen de certains indicateurs empiriques.

Les difficultés pour mesurer la productivité

L'étude de la productivité du travail pour la production des logiciels présente des difficultés. La productivité du travail se définit comme le rapport entre la quantité produite pendant une certaine période (en général l'année) et la quantité de travail nécessaire pour fournir cette production. La mesure du dénominateur (la quantité de travail) nécessite d'effectuer certains choix : l'unité de mesure du travail, qui peut être les effectifs moyens sur la période considérée (ce qui nécessite de raisonner en équivalent temps plein pour tenir compte du travail à temps partiel) ou le nombre d'heures travaillées (productivité horaire) ; la délimitation des travailleurs qui contribuent à la production effectuée avec notamment les problèmes de prise en compte des travailleurs "indirectement productifs", et des choix d'imputation du travail des personnes qui contribuent également à d'autres productions que celles mesurées au numérateur. La mesure du numérateur (la quantité produite, qui nécessite de pouvoir définir une unité de production) pose des problèmes en général plus difficiles à solutionner : son caractère le plus souvent hétérogène nécessite de trouver des indicateurs

pertinents pour pouvoir agréger la production. Ces indicateurs peuvent être des indicateurs techniques (ou physiques) que l'on estime représentatifs de la production réalisée, ou des indicateurs économiques (chiffre d'affaires, valeur ajoutée) ce qui nécessite d'éliminer les évolutions des prix non liés à des changements qualitatifs de la production réalisée. Pour que la notion de productivité ait une certaine pertinence comme mesure de l'efficience du travail de production dans l'obtention de résultats donnés, il est nécessaire que l'on puisse identifier un output séparable de l'activité économique.

La mesure de l'output en termes réels ne pose pas de problèmes particuliers en cas de "validité de l'hypothèse de nomenclature qui veut que la liste des biens et des services soit plus ou moins une donnée de connaissance commune, c'est à dire relativement appréhendable, en essence et en quantité, par l'ensemble de la communauté concernée" (Pascal Petit, 1998, p. 348). Suivant la plus ou moins grande hétérogénéité de l'output, on pourra soit mesurer la production directement avec des indicateurs techniques (ou physiques), soit mesurer la production par des indicateurs économiques, la nomenclature des produits constitutifs de l'activité permettant de construire un indice des prix pour dissocier évolution "en volume" et en prix. Pour la plupart des produits industriels, il est possible de procéder ainsi de façon à peu près satisfaisante. Une première difficulté existe pour appréhender l'évolution de la qualité, quand les produits sont complexes (le nombre de caractéristiques pertinentes est élevé) et évoluent rapidement, ce qui pose le problème de la saisie comptable des innovations, en particulier de l'innovation dite non proportionnelle "qui améliore la qualité des produits en augmentant la quantité de caractéristiques d'un input relativement à leur coût" (Bernard Guilhon, 1991, p. 530). Les ordinateurs constituent un bon exemple de ces difficultés (cf. les fortes différences de baisse des prix selon les méthodes utilisées pour mesurer l'effet-qualité). Des difficultés supplémentaires apparaissent pour de nombreuses activités de services "quand aucun objet ne vient fractionner le rapport entre les producteurs et les consommateurs, quand les résultats apparaissent comme des produits conjoints des uns et des autres, ou quand la rétroaction et l'adaptation interdisent de repérer des unités standardisées, des séries de biens ou de services comparables dans le temps" (Jean Gadrey, 1996 A, p. 134). Il importe dans ses situations d'analyser avec soin la pertinence des indicateurs existants ou d'en construire de plus adéquats (la productivité dans ce cas pouvant rarement être appréhendée à partir d'un indicateur unique), pour éviter des contresens majeurs sur les tendances à l'œuvre (cf. de multiples exemples dans les travaux de Jean Gadrey).

Le secteur de la production des logiciels combine ces difficultés. En effet, il comprend d'une part des progiciels, produits standardisés et reproductibles, mais qui changent très rapidement et dont la qualité ne peut être appréhendée par la connaissance de quelques caractéristiques (cf. les difficultés pour appréhender la valeur des logiciels, chapitre III). Il comprend, d'autre part, des logiciels sur mesure, qui ont en commun avec certains services d'être "des produits marqués par les aléas de la création, de l'initiative, des relations sociales rétroactives" (Jean Gadrey, 1996 A, p. 76). De plus les produits ou résultats des activités de création de logiciels peuvent être repérés selon des perspectives différentes et selon des horizons temporels plus ou moins longs (cf. la distinction entre caractéristiques d'usages directes et indirectes, analysée dans le chapitre I).

Sans nier ces difficultés et sans prétendre avoir réussi à mesurer "*la*" croissance de la productivité dans ce secteur, il est toutefois possible à partir de l'examen de certains indicateurs techniques (1) et économiques (2) dont on examinera les limites, de dégager des tendances d'évolution significatives.

2 - La croissance de la productivité mesurée par des indicateurs techniques

a - Le nombre de lignes de codes par personne

La première possibilité consiste à mesurer la productivité à partir d'indicateurs techniques (parfois également appelée productivité physique). En effet, techniquement il existe un output séparable de l'activité (le logiciel), même, si de plus en plus, il ne recouvre qu'une partie de celle-ci (services liés). Le problème est que les logiciels sont très différents les uns des autres et qu'il faut leur trouver une caractéristique commune, pour disposer d'une unité de mesure de ce que produit le processus de développement de logiciels. La plus évidente est qu'ils sont tous constitués d'un code source et que l'on peut facilement connaître le nombre de lignes de codes. Il suffit ensuite de calculer le nombre de lignes de codes sources (LS ou KLS pour Kilo Lignes Source) par homme année (KLS/HA) ou par homme mois (KLS/HM)[3] pour disposer d'un indicateur de productivité.

[3] A notre connaissance, il n'existe pas de tentative de mesure de la productivité horaire, notamment parce qu'il est souvent difficile de connaître la durée réelle du travail des programmeurs ; Capers Jones signale que la pratique répandue des heures supplémentaires non payées peut avoir pour conséquence de sous-estimer les gains de productivité réels, "un gain de 20 % ou 25 % [risquant] de se traduire uniquement par une diminution des heures supplémentaires non payées" (Capers Jones, 1989, p. 272).

Cet indicateur de productivité ne posait pas trop de problèmes d'interprétation quand la production de logiciels était une activité essentiellement individuelle, basée sur le même langage de programmation et que l'indicateur était appliqué à des programmes d'une taille et d'une complexité comparables. La facilité et le coût très faible d'obtention de cet indicateur fait qu'il est resté "l'unité de mesure la plus répandue dans l'industrie de la programmation" (Capers Jones, 1989, p. 24). Il correspondait à "l'hypothèse de loin la plus répandue au sujet de la productivité, depuis les débuts de l'industrie de la programmation, [qui] était qu'améliorer la productivité voulait dire augmenter la possibilité de rédiger des lignes de code-source à plus grande vitesse" (idem, p. 21).

Le premier constat mis en évidence par l'utilisation de cet indicateur est l'importance de la différence de productivité à qualité constante selon les programmeurs : Jacques Printz (1998, p. 88) a constaté des variations de productivité dans un rapport de 1 à 10, Capers Jones, 1989, p. 128) cite différentes études mettant en évidence un rapport de 1 à 20. A partir de la construction d'un modèle ("*software productivity, quality and reliability model*") utilisé pour normaliser et synthétiser des projets réels, Capers Jones a estimé l'influence de la seule expérience sur la productivité : pour réaliser un programme de 1000 lignes en Cobol, la productivité passe de 100 lignes par mois en moyenne pour un programmeur inexpérimenté à 250 pour un programmeur expérimenté (idem, p. 130). Parmi les programmeurs expérimentés, les différences de productivité peuvent également être importantes : Frederick P. Brooks, (1996, p. 24) cite une étude qui montrait, à l'intérieur d'un groupe de programmeurs expérimentés, que le rapport entre les meilleures et les plus mauvaises performances étaient en moyenne de 10 pour la productivité, et de 5 pour la vitesse du programme et la mémoire occupée. Toran DeMarco et Timothy Lister organisent des *Coding War Games* depuis 1977, compétition (publique depuis 1984) entre informaticiens portant sur la conception, le codage et la vérification d'un programme de moyenne importance ; de 1984 à 1986 ces compétitions ont opposé plus de 600 programmeurs représentant 92 compagnies ; les principaux résultats sont que les meilleurs font mieux que les moins bons dans un rapport de 10 pour 1, le meilleur exécutant est 2,5 fois supérieur que l'exécutant moyen, la moitié supérieure à la moyenne fait mieux que l'autre moitié dans un rapport dépassant 2 pour 1 (1991, p. 54).

b - L'importance des conventions utilisées

Cependant, le nombre de lignes de code source par personne ne peut être utilisé valablement qu'en précisant clairement les nombreuses conventions adoptées, ce qui est loin

d'être toujours le cas, alors même que ces conventions sont fréquemment différentes selon les études effectuées. Concernant le numérateur, Capers Jones dénombre onze façons différentes, concernant les programmes et les projets, pour compter le nombre de lignes de code source. Les variantes au niveau des programmes concernent principalement la définition de ce qui termine une ligne de code (un retour à la ligne ou un séparateur logique), la prise en compte des définitions de données, des commentaires ou uniquement des instructions exécutables. Elles ont pour conséquence des variations d'au moins cinq à un entre la technique de comptage la plus lâche et la technique de comptage la plus compacte. Les variantes au niveau des projets concernent l'éventuelle prise en compte du code réutilisé, des lignes de codes supprimées (qui peuvent demander beaucoup d'efforts et permettre une amélioration du produit par augmentation de la vitesse de traitement et diminution de l'encombrement en mémoire), du code temporaire de mise au point et du code d'assistance ; selon les choix effectués l'estimation de la productivité peut aller de 333 lignes de code par homme-mois à 7666 lignes de codes par homme-mois (Capers Jones, 1989, p. 31 et 32). Concernant le dénominateur, selon que l'on prend en compte uniquement les personnes qui font du codage et que pendant les périodes où elles effectuent celui-ci, ou qu'au contraire on comptabilise l'ensemble des personnes qui participe au cycle de développement du logiciel depuis l'expression des besoins jusqu'à la maintenance des applications, les écarts de productivité sont encore plus importants : ainsi Capers Jones estime que la productivité apparente contenus dans les lignes de code source par homme-année est de 25 000 pour "le codage seul, mesuré pendant une journée et converti en taux annuel", à 250 pour "les moyens totaux consacrés en un an au logiciel par l'entreprise, y compris projets abandonnés, tous les développements et améliorations et toute la maintenance" (1989, p. 38).

L'importance de ces écarts traduit la place de plus en plus limitée occupée par l'écriture du code inclus dans le logiciel par rapport à l'ensemble des activités de développement d'un logiciel. Par exemple, Jacques Printz (1998, p. 278) précise qu'il existe trois types de code : le code fonctionnel livré qui correspond aux spécifications fonctionnelles de l'application, le code non fonctionnel livré dont seules certaines parties comme les messages d'erreurs ou les aides en ligne sont visibles de l'usager, et le code non fonctionnel non livré qui a été écrit pour atteindre le but final et qui correspond à des constructions auxiliaires : générateurs de programmes, maquettes, simulateurs, programmes de substitution... Le plus souvent les estimations ne portent que sur le code livré, ce qui est d'autant plus dommageable que le code non-fonctionnel n'est pas proportionnel au code fonctionnel. Surtout une amélioration

apparente de la productivité qui consisterait à réduire le code non fonctionnel risque de se traduire par une chute de la productivité de la maintenance, le code non fonctionnel représentant "la seule vraie garantie de pérennité des investissements réalisés : (…) le risque [étant] moins grand de voir disparaître le produit, que de voir disparaître ceux qui l'ont fait naître" (idem, p. 281). La détection des erreurs dans un logiciel nécessite également une écriture importante de code non fonctionnel non livré : Frederick P. Brooks (1996, p. 125 et 214) estime qu'il est rentable d'écrire jusqu'à 50 % de code supplémentaire par rapport au produit en cours de débogage dans des "échaffaudages", c'est à dire "des programmes et des données élaborés pour les besoins du débogage, mais qui ne feront pas partie du produit final".

Un deuxième exemple expliquant les écarts selon les activités prises en compte concerne la documentation. Son importance est tout à fait décisive pour l'utilisation et la maintenance des applications réalisées, et elle peut mobiliser des moyens très importants. Capers Jones (1989, p. 39-40) cite l'exemple d'un important système de télécommunication qui était accompagné de 100 différents modèles de documents représentant 60 000 pages et 30 millions de mots. La moyenne de ce projet était de 120 mots anglais par ligne de code-source ce qui n'est pas considéré comme anormalement élevé, les grands systèmes des administrations civiles et militaires comportant 200 mots anglais en moyenne de documentation pour chaque ligne de code du système livré au client.

Enfin indépendamment de ces problèmes qui peuvent se résoudre par une normalisation des conventions utilisées, il faut également tenir compte du "style" de programmation qui peut aboutir à des programmes de taille sensiblement différente pour résoudre le même problème. Un exemple célèbre est le conflit opposant Microsoft à IBM lors de leur projet d'écrire en commun un système d'exploitation (OS2) : la contribution de chaque entreprise, de culture très différente, au projet commun était mesurée par le nombre de lignes de code source produit, et les programmeurs de Microsoft considéraient que les programmeurs d'IBM avaient besoin d'écrire (inutilement) beaucoup plus de lignes de codes qu'eux pour réaliser les mêmes fonctions (Paul Carroll, 1994).

c - Le problème de l'hétérogénéité des langages de programmation

Une complication supplémentaire dans l'utilisation de l'indicateur du nombre de lignes de codes par personne pour mesurer la productivité provient de l'utilisation de plus de 500 langages de programmation différents, ce qui ne permet pas d'effectuer une comparaison

directe pour des programmes écrits dans des langages différents. La création de langages de plus haut niveau avait notamment pour objectif d'améliorer la productivité des programmeurs. *Or on constate invariablement que lorsque l'on utilise un langage de plus haut niveau, la productivité mesurée par le nombre de lignes de code-source diminue.* C'est ce que Capers Jones (1989, p. 21) appelle le "paradoxe le plus significatif de toute la profession : les langages de haut niveau tendent à réduire, au lieu d'accélérer, la vitesse de production des lignes de codes". L'explication de cet apparent paradoxe est la suivante : plus le niveau du langage est élevé, moins il y a de code à rédiger pour réaliser une même fonction. Certes, le codage d'une ligne dans un langage de plus haut niveau ne prend pas plus de temps (il aurait même plutôt tendance à diminuer légèrement lorsque le niveau du langage s'élève), mais la part relative des activités indépendantes des langages de programmation (spécifications, conception, documentation…) va augmenter, ce qui dégrade apparemment la productivité mesurée par le nombre de lignes de code-source.

L'exemple suivant de trois programmes fonctionnellement identiques rédigés dans trois langages de niveau croissant permet d'illustrer ce raisonnement :

Tableau XXVII
Le paradoxe des lignes de code :
un programme identique dans trois langages différents

	Assembleur	**PL/1**	**APL**
Lignes source	100 000	25 000	10 000
Activité (hommes-mois) :			
Spécifications	10	10	10
Conception	30	30	30
Codage	115	25	10
Documentation	20	20	20
Intégration/tests	25	15	10
Total (hommes-mois)	**200**	**100**	**80**
Lignes de code source par homme-mois	**500**	**250**	**125**

Source : Capers Jones, 1989, p. 22

Le même programme nécessite 100 000 lignes de code source dans le langage de plus bas niveau (l'assembleur) et seulement 10 000 dans un langage de plus haut niveau comme APL. Il est clair que l'utilisation d'un langage de plus haut niveau a permis d'augmenter la productivité "réelle" puisqu'il fallait 200 hommes-mois pour développer la version en assembleur et seulement 80 pour développer la version en APL. Mais mesurée en lignes de code source par mois, la productivité a apparemment chuté (de 500 à 125 !). Il est intéressant de noter que ce n'est pas l'activité de codage qui explique ce phénomène (le programmeur en Assembleur codait en moyenne 870 lignes par mois, et 1000 lignes par mois en PL/1 ou en APL), mais la baisse de la part des efforts consacré au codage, ce qui est justement une des motivations pour utiliser des langages de plus haut niveau.

Formulé autrement, ce paradoxe signifie qu'il n'est pas justifié de comparer directement deux programmes de même taille écrits dans des langages différents : un programme de 1000 lignes Cobol demande beaucoup plus d'efforts pour établir ses spécifications, sa conception et sa documentation qu'un programme de 1000 lignes Assembleur parce que ses fonctionnalités sont beaucoup plus importantes.

Il faut pourtant remarquer que le plus souvent l'utilisation très répandue de la mesure de l'évolution de la productivité par le nombre de lignes de code par personne, ne tient pas compte des différences des niveaux de langages utilisés, ce qui a certainement contribué à sous-estimer l'augmentation réelle de la productivité dans la production de logiciels.

Une première possibilité pour éliminer ce bais consisterait à évaluer le logiciel produit en remplaçant la mesure des lignes de code-source, par des lignes de code-objet ou octets occupés en mémoire, une ligne de code source en langage de haut niveau se traduisant par beaucoup plus de lignes de code-objet qu'une ligne de code-source en langage de bas niveau (un programme de 1000 lignes Cobol génère un programme exécutable de taille beaucoup plus importante qu'un programme de 1000 lignes assembleur). Toutefois cette solution, si elle est très simple à mettre en pratique, risque d'aboutir à un nouveau paradoxe : l'amélioration continuelle des compilateurs permet de générer un code-objet de plus en plus compact, ce qui permet d'améliorer les performances (vitesse d'exécution, place occupée en mémoire) mais ferait apparaître une baisse de la productivité mesurée par la taille du code-objet.

La seconde possibilité, retenue par Capers Jones, consiste à convertir les programmes écrits dans les différents langages en "équivalent assembleur" en se basant sur un nombre moyen d'instructions exécutables générées par une instruction source. En reconnaissant que

l'estimation de ces ratios est approximative, Capers Jones propose les conversions suivantes : une instruction source en assembleur génère une instruction exécutable, une instruction source en Cobol ou en Fortran génère en moyenne trois instructions exécutables (et est donc équivalente à trois instructions assembleur), une instruction source en APL génère en moyenne 10 instructions exécutables, une instruction source en Smalltalk (un langage objet) génère 15 instructions exécutables, une instruction dans un langage de tableur génère 50 instructions exécutables… (1989, p. 69). A partir de ce ratio, on peut calculer l'évolution de la productivité mesurée en unités artificielles d'équivalent assembleur. Pour un système remplissant les mêmes fonctions réalisé à vingt ans d'intervalle, Capers Jones donne les estimations suivantes :

Tableau XXVIII
Productivité de systèmes remplissant les mêmes fonctions à vingt ans d'intervalle

	1964	**1984**
Langage	Assembleur	Cobol
Lignes source	30 000	10 000
Activité (hommes-mois) :		
Spécifications	10	4
Conception	15	6
Documentation interne	14	4
Documentation utilisateurs	29	11
Codage	99	20
Tests d'intégration	40	11
Correction des défauts	48	31
Gestion	25	7
Total (hommes-mois)	**280**	**94**
Lignes de code source par homme-mois	**107**	**106**

Source : Capers Jones, 1989, p. 106

On constate que bien qu'aient été pris en compte des améliorations de productivité dans l'ensemble des activités du cycle du développement du logiciel, le changement de langage de programmation fait que la productivité mesurée par le nombre de lignes de code-source (sans tenir compte du changement de langage) reste stable. Par contre, en considérant qu'une ligne

Cobol équivaut à trois lignes en assembleur, la productivité en équivalent assembleur est de 319 lignes par homme-mois en 1984, soit une hausse de près de 200 % en vingt ans[4], ce qui correspond à un taux de croissance annuel moyen de 5,6 %. Certains auteurs font état de gains de productivité encore plus importants : par exemple, Frederick P. Brooks estime que "la productivité en programmation peut aller jusqu'à quintupler lorsqu'un langage évolué approprié est utilisé" (1996, p. 207).

Toutefois Capers Jones estime que pour les langages dont le niveau dépasse 15 fois la puissance de l'assembleur, tels les langages à base graphique (par exemple Visual Basic où la programmation repose plus sur l'utilisation de menus déroulants ou des commandes par boutons que sur des instructions déclaratives), la mesure par les lignes de codes perd toute signification et ne doit pas être utilisée (1998, p. 20)[5].

d - L'augmentation de la taille et de la complexité des projets à réaliser

Jusqu'à maintenant nous avons uniquement considéré des projets de taille identique. Or les projets de taille importante demande des efforts qui croissent beaucoup plus rapidement que l'augmentation du nombre de lignes de codes produites. Par exemple, Jean-Marc Geib estime que l'écriture d'un compilateur Pascal ou C nécessitait un effort de 10 années-hommes (HA), qu'il était de 150 HA pour un compilateur ADA, et qu'il s'élevait à 1000 HA pour le logiciel de la navette spatiale (1989, p. 3). Frederick P. Brooks, manager du logiciel Operating System/360 d'IBM évalue à plus de 5000 années-hommes la conception, la construction et la documentation de ce système entre 1963 et 1966 (Frederick P. Brooks, 1996, p. 25). De façon plus générale, il a été estimé que l'effort exprimé en homme-mois (HM) croît de façon exponentielle avec la taille du logiciel à développer, exprimée en milliers de lignes source livrées (KLS)[6]. Ceci signifie que la productivité des programmeurs décroît fortement avec le volume de code à réaliser (Jacques Printz, 1998, p. 270).

---

[4] Capers Jones estime qu'elle est de 300 % mais il s'agit vraisemblablement d'une erreur de calcul.

[5] Un des effets des générateurs automatiques de code est que, s'ils facilitent considérablement la programmation, ils produisent toujours beaucoup plus de lignes de codes qu'un codage manuel.

[6] Barry Boehm (1981, pp. 81-84) donne les estimations suivantes dans le modèle de coût CoCoMo :

$\text{Effort}_{HM} = 2{,}4 * (\text{Volume}_{KLS})^{1{,}05}$ pour les programmes simples, et $\text{Effort}_{HM} = 3{,}6 * (\text{Volume}_{KLS})^{1{,}20}$ pour les programmes complexes. Frederick P. Brooks cite d'autres études selon lesquelles l'exposant serait proche de 1,5 (1996, p. 74 et 206).

En réalité, c'est moins la taille que la complexité de l'application qui affecte la productivité : "il est possible d'envisager de très grandes collections de codes comme par exemple une bibliothèque de macros, un magasin de modules réutilisables ou un ensemble de programmes utilitaires, où la taille totale a relativement peu d'effets sur la productivité d'ensemble parce que les modules individuels sont complètement déconnectés les uns des autres et ne font que se côtoyer dans un même catalogue" (Capers Jones, 1989, p. 122). Mais, le plus souvent, la complexité d'un logiciel augmente de façon non linéaire avec sa taille (Frederick P. Brooks, 1996, p. 159). Gérard Dréan estime que pour les activités de rédaction et de test, la productivité va de 60 instructions (testées) par jour-homme pour un programme simple à moins de 2 instructions par jour-homme pour des sytèmes complexes" (1996 A, p. 199). Cette chute de la productivité pour la production de grandes applications complexes est d'autant plus gênante que la tentation d'augmenter le nombre de personnes qui travaillent sur le projet pour pouvoir l'effectuer dans des délais raisonnables se révèle peu efficace. C'est ce que Frederick P. Brooks, qui coordonnait jusqu'à plus de 1000 personnes participant au développement de l'OS/360 d'IBM, appelle "le mythe du mois-homme" : les mois et les hommes ne sont interchangeables que lorsqu'une tâche peut être divisée entre plusieurs travailleurs sans réclamer de communication entre eux, ce qui est vrai pour la récolte du blé ou la cueillette du coton mais pas pour la programmation (1996, p. 14). Dans ce cas, il faut prendre en compte la formation des travailleurs au but du projet, à sa stratégie globale, à son plan de travail et aux technologies utilisées, ce qui représente un surcroît de travail qui augmente linéairement avec le nombre de travailleurs. Il faut surtout tenir compte de l'effort supplémentaire de communication : si n taches doivent être séparément coordonnées avec chaque autre tâche, l'effort augmente en n(n-1)/2 (idem, p. 15). Dans des situations extrêmes ces activités supplémentaires font plus que compenser l'apport de travailleurs supplémentaires, ce qui est connu sous le nom de "loi de Brooks" : "ajouter des gens à un projet logiciel en retard le retarde encore davantage" (Frederick P. Brooks, 1996, p. 20)[7].

Toutefois, il existe une possibilité de contourner cette "loi" et d'arriver à une augmentation significative de la productivité, malgré l'augmentation de la taille moyenne des applications développées. Elle correspond au fait que c'est moins la taille que la complexité

[7] Sur les effets néfastes des tentatives de raccourcir les délais des projets, on peut également mentionner les travaux de Toran Demarco et Timothy Lister qui montrent que la productivité est meilleure quand il n'y a pas d'estimation initiale du temps nécessaire pour développer le projet (1991, p. 38) !

qui importe. Elle consiste à effectuer un important travail préalable au niveau de l'architecture du système pour le décomposer en petits modules qui doivent avoir une indépendance maximale. Joël Aron à partir de l'étude du développement de neuf grands systèmes chez IBM a montré que la productivité variait de 1500 lignes de code par année-homme quand il y avait beaucoup d'interactions entre les programmeurs et les parties du système à 10 000 quand il y en avait très peu (Frederick P. Brooks, 1996, p. 206). Il apparaît ainsi que "la clé de la productivité est l'architecture", une bonne architecture consistant à mettre en place "les interfaces nécessaires à une croissance indépendante des différentes composantes du logiciel envisagé" (Jacques Printz, 1998, p. 271). C'est du reste le domaine de l'architecture des logiciels qui a connu le plus d'innovations dans la période récente (cf. chapitre III). En même temps, plus les activités de conception se développent au détriment du codage (de plus en plus automatisé), moins il semble pertinent de se baser sur le nombre de lignes de codes pour mesurer la production réalisée.

e - Les autres indicateurs techniques

Malgré "l'importance surprenante historiquement accordée aux lignes de codes" (Capers Jones, 1989, p. 45) alors que presque personne ne se satisfait de cette métrique (idem, p. 61), il existe des tentatives pour construire des indicateurs plus significatifs de l'activité réalisée.

La première tentative est celle de Maurice Halstead (1977) de construire des indicateurs de complexité textuelle à partir d'une démarche scientifique (le titre de son ouvrage, très controversé, est *Elements of software science*). L'idée de base est de comptabiliser séparément les instructions fonctionnelles du programme (appelées "opérateurs") et les définitions de données (les constantes et les variables du programme appelées "opérandes"). Ceci lui permet de mesurer le "vocabulaire" n d'un programme (la somme du nombre d'opérateurs et du nombre d'opérandes différents), et la "longueur" N d'un programme (la somme du nombre d'occurrences des opérateurs et des opérandes). A partir de ces deux grandeurs sont calculées le "volume" du programme (égal à $N \text{Log}_2(n)$). Ces indicateurs de base ont été ensuite utilisés pour calculer de nouveaux indicateurs (la "difficulté" D, "l'effort" E, le "contenu d'intelligence" I). Le bilan de cette tentative est mitigé : si la distinction opérée entre les aspects fonctions et données des programmes a permis de mettre en évidence l'importance des données, les indicateurs de base "partagent plus ou moins [avec le nombre de lignes de codes] les mêmes problèmes et les mêmes sources d'ambiguïté" (Capers Jones, 1989, p. 90) ; quant aux autres indicateurs (D, E, I), ils sont "purement subjectifs et ne proviennent en aucune

façon de données objectives" (idem, p. 90-91). Finalement, selon Jacques Printz, "personne n'a été capable de démontrer l'avantage de ces mesures par rapport aux mesures plus traditionnelles" (1998, p. 215).

Une autre tentative consiste à tenter de mesurer la complexité structurelle d'un programme. La mesure la plus connue est celle de Thomas McCabe basée sur la représentation d'un programme par un graphe orienté composés de nœuds (blocs séquentiels d'instructions) et d'arcs (les transferts possibles de contrôle entre les blocs). A partir du nombre de nœuds et d'arcs d'un programme, Thomas McCabe (1976) définit la "complexité cyclomatique" d'un programme. Cette méthode, qui connaît un certain succès notamment pour prédire le nombre de défauts d'un programme, a suscité deux critiques. D'une part, la complexité mesurée est celle de la solution qui a été trouvée et qui, selon les compétences des concepteurs, peut être plus ou moins différente de la complexité du problème posé initialement. D'autre part, cette mesure ne tient pas compte de la complexité des données ; or, selon certains auteurs, la complexité d'un programme vient principalement de la complexité des données qu'il doit traiter (Capers Jones, 1989, p. 95).

Une troisième solution de nature plus empirique est la technique dite des "points de fonction", expression peu adéquate pour une technique qui ne s'occupe pas explicitement des fonctions. Cette technique a été développée par A.J. Albrecht en 1979, qui tentait de mesurer l'évolution de la productivité chez IBM pour des programmes écrits dans une grande variété de langage et qui se heurtait au paradoxe exposé précédemment. La mesure des points de fonction, déterminés à partir des caractéristiques d'un projet de logiciel et indépendants du langage de programmation utilisé, correspond aux totaux pondérés et ajustés de cinq éléments : les entrées de l'application, les sorties de l'application, les fichiers logiques associés à l'application, les requêtes pouvant être effectuées vis à vis de l'application, les interfaces entre l'application considérée et d'autres applications externes. Cette méthode conserve un certain degré de subjectivité dans la détermination des coefficients de pondération et par l'introduction d'une plage de variation autorisée de plus ou moins 25 %, en fonction de certains facteurs comme une grande complexité ou un traitement temps réel.

Cette technique a été utilisée au départ par A.J. Albrecht sur un ensemble de 22 projets s'étendant sur cinq ans. Sur cette période de cinq ans et avec cette métrique, A.J. Albrecht et J.E. Gaffney (1983) ont estimé que la productivité avait été multipliée par 3, ce qui représente un taux de croissance annuel moyen de près de 25 %. Il faut noter que cette période a connu

beaucoup d'améliorations des technologies de développement et des langages de programmation. Pour expliquer cette forte croissance de la productivité, A.J. Albrecht mentionne comme facteurs principaux, l'utilisation de la programmation structurée, des langages de haut niveau, du développement *on-line*, et d'une bibliothèque de développement de programmes.

En constatant en 1989, qu'aucune métrique communément admise par tout le monde n'avait réussi à se substituer au nombre de lignes de code, Capers Jones plaidait pour la création d'un indicateur unique ("équivalent logiciel du "produit étalon" de la comptabilité analytique") qui aurait intégré les points de fonction de A.J. Albrecht, la mesure de la complexité de T. McCabe et une mesure de la complexité des données fondée sur les méthodes de conception par l'analyse des données (la méthode la plus connue étant celle de Jackson, de Warnier et de Orr). L'examen des écrits postérieurs de Capers Jones semble indiquer que cette tentative n'a pas abouti et que ceux qui ne se satisfont pas de la métrique du nombre de lignes de codes recourent principalement à la méthode des points de fonctions, dont les aspects les plus subjectifs ont été en partie éliminés par son application à de nombreux projets et par l'importance des programmes de recherche qui lui ont été consacré (Capers Jones, 1998).

f - Une productivité manifestement croissante

Il est difficile d'obtenir des informations statistiques globales et fiables sur les indicateurs techniques permettant d'appréhender la productivité. Tout d'abord, beaucoup d'entreprises n'effectuent pas ces mesures : "les mesures relatives à la programmation restent le maillon le plus faible de toute la science du génie logiciel" (Capers Jones, 1989, p. 19) ; de même, Serge Bouchy (1994, p. 194) indique que si les indicateurs de complexité ont été partiellement instrumentés dans des outils de logimétrie, ceux-ci restent peu utilisés. Ensuite, les entreprises, en général de grande taille[8], qui effectuent ces mesures communiquent rarement leurs résultats. Les seules sources sur lesquelles on peut s'appuyer sont les écrits de personnes travaillant (ou le plus souvent ayant travaillé) dans ces sociétés (par exemple Frederick P. Brooks à IBM), et des consultants spécialisés dans le développement de la

[8] Ces mesures peuvent s'avérer coûteuses : Capers Jones estime qu'IBM consacre l'équivalent de 5 % de tous ses coûts de développement à des opérations liées aux mesures (1989, p. 289).

productivité concernant les logiciels (Tom DeMarco et Timothy Lister, Capers Jones, fondateur et président de la société *Software Productivity Research*).

Ce qui frappe à la lecture de ces écrits, c'est que tous mentionnent des estimations statistiques de gains de productivité élevés (voire dans certains cas très élevés, cf. A.J. Abrecht, supra), tout en les considérant comme minimes. On peut l'expliquer par le fait que ces études émanent d'informaticiens dont la référence semble plus être les gains de productivité dans la production de matériel informatique, que dans l'ensemble de l'économie. Par exemple, Jacques Printz cite une étude de ACM SIGSOFT (Vol 8, n° 2 April 1983) qui indiquait que la productivité moyenne d'un programmeur avait été multipliée par 3,6 en 30 ans –ce qui représente un taux de croissance annuel moyen de 4,36 % - et considère qu'il s'agit d'une performance "ridicule comparée à la performance réalisée pour le matériel" (1998, p. 233). Une autre explication possible de cette sous-estimation des progrès constatés, est la volonté de relativiser les annonces, fréquentes dans le domaine des logiciels (et souvent commercialement intéressées), d'une nouvelle solution[9] ou d'un produit miracle, censé à lui seul, multiplier la productivité par un nombre très élevé (la promesse d'une augmentation de la productivité de 1000 % n'est pas rare !).

Les facteurs d'augmentation de la productivité

La thèse centrale du livre de Tom DeMarco et Timothy Lister "Les hommes de l'ordinateur (Les conditions de la productivité des équipes de projets informatiques)"[10] est que "les difficultés majeures de notre travail ne sont pas tant de nature technologique que de nature sociologique" (1991, p. 14). Ayant constaté lors des compétitions qu'ils organisent régulièrement entre des programmeurs de différentes entreprises (cf. supra), que les écarts de productivité très importants sont moins des écarts entre des individus que des écarts entre les différentes entreprises, ils prodiguent de nombreux conseils sur la gestion des ressources humaines pour atteindre les performances des meilleures sociétés. Cette insistance sur les facteurs humains est justifiée par le fait que les concepteurs de logiciels "travaillent dans la

[9] "La communauté du génie logiciel se distingue de tous les autres groupes professionnels par le fait qu'elle généralise immédiatement une idée utile et la prend ensuite comme solution par excellence pour chaque problème", avec une "croyance presque religieuse à cette nouvelle pierre du savoir" (Maarten Boasson, 1998, p. 5-6).

[10] Le titre anglais est "Peopleware : productive projects and teams". La traduction de *peopleware* par "les hommes de l'ordinateur" n'est pas des plus judicieuses ; "humaniciel" aurait peut-être mieux rendu compte de la pensée des auteurs.

communication et sont au service des utilisateurs pour donner forme à l'expression de leurs besoins. Cette fonction restera primordiale, quels que soient les changements qui interviendront par ailleurs, et elle est peu susceptible d'être automatisée" (idem, p. 43). C'est ce qui explique que, "la productivité dans l'industrie du logiciel a progressé au rythme de 3 à 5 % par an, à peine mieux que dans l'industrie de l'acier ou de l'automobile", pourcentage qu'ils reprennent d'une étude de 1979, et qu'ils estiment "malheureusement très faible" (idem, p. 42).

A partir de la construction d'un modèle basé sur des projets réels, Capers Jones identifie vingt facteurs dont il a pu quantifier l'impact sur la productivité et la qualité de la programmation, et vingt-cinq autres facteurs significatifs "pour lesquels les informations ne sont pas suffisantes pour pouvoir porter un vrai jugement quantitatif" (Capers Jones, 1989, p. 111-112). Outre l'influence des langages de programmation, les principaux facteurs quantifiés qui permettent d'améliorer la productivité, sont l'utilisation des méthodes de programmation structurée, d'outils et de conditions d'environnement favorables, de générateurs d'applications et de programmes, du prototypage rapide… En consolidant quelques-unes de ces variables, Capers Jones estime que la productivité est plus de deux fois supérieure pour une entreprise utilisant des méthodes de pointe par rapport à une entreprise utilisant des méthodes d'arrière-garde (idem, p. 251). En prenant en compte l'ensemble des facteurs, Capers Jones juge qu'en 1985, les programmeurs des entreprises de pointe sont capables de fournir presque dix fois plus de fonctionnalité par unité de travail que les moyennes de l'industrie de 1975 (idem, p. 16). Un facteur d'amélioration de la productivité particulièrement prometteur était la réutilisation de composants déjà développés, mais en 1985, Capers Jones jugeait que c'était "une technologie naissante, encore immature et peu formée" (idem, p. 189) et insistait sur l'importance des investissements à effectuer : la mise au point d'une bibliothèque de schémas et modules normalisés peut demander plusieurs années et coûter plus d'un million de dollars. En 1998, Capers Jones souligne que la réutilisabilité offre la plus grande valeur ajoutée et le meilleur retour sur investissement (Capers Jones, 1998, p. 23). En insistant sur le fait que la réutilisabilité ne se limite pas au code, mais peut concerner douze composants réutilisables (architecture, cahier des charges, plans, estimations, conception et spécification, interfaces, données, écrans et éléments d'écrans, code source, documents d'utilisation, plans et tests, jeux de tests), Capers Jones estime qu'une réutilisation réussie – ce qui exige un contrôle de la qualité qui approche le niveau du zéro défaut - améliore de 65 % la productivité, réduit de 50 % les délais et de 85 % les défauts (1998, p. 35).

"L'essence" et la "substance" des logiciels

Une contribution majeure au débat sur l'évolution de la productivité dans la production de logiciels est un article de Frederick P. Brooks, écrit en 1986[11] et intitulé "Pas de balle d'argent : l'essence et la substance en génie logiciel". Cet article suscita de nombreux articles de réfutation, auxquels Frederick P. Brooks répondit en 1996 dans "Pas de balle d'argent. Seconde salve". L'article de 1986 fut souvent interprété comme une démonstration du caractère inéluctable de la faiblesse des gains de productivité dans la production des logiciels, ce que pouvait laisser entendre son titre : les "balles d'argent" sont des armes mythiques capables d'abattre par magie les loups-garous, en l'occurrence les projets logiciels, "monstres crachant des retards, des dépassements de budgets et des produits déficients" (Frederick P. Brooks, 1996, p. 156). En réalité, la thèse défendue par Frederick P. Brooks n'était pas si pessimiste : il affirmait qu'"aucun développement en génie logiciel ne produirait, *à lui seul, un gain d'un facteur 10* en productivité de programmation dans les dix ans à venir" (idem, p. 184) et il précisera dix ans plus tard, que "l'article prédisait que *conjointement* les innovations en cours de développement en 1986 engendreraient effectivement un gain d'un ordre de grandeur en productivité", (idem) ce qu'il juge avoir été une prédiction un peu trop optimiste. Ce qui explique que des gains de productivité aussi importants ne soient pas perçus comme tels, c'est que la référence explicite est le fantastique progrès qu'a connu le matériel informatique : "on ne peut entrevoir aucune invention qui augmente la productivité, la fiabilité et la simplicité du logiciel, comme l'électronique, le transistor et l'intégration à grande échelle l'ont fait pour le matériel informatique (…) [avec] des gains d'un facteur deux tous les deux ans" (idem, p. 157) ; "aucune percée technologique ne nous promet les résultats magiques dont nous sommes si familiers dans le domaine du matériel" (idem, p. 170).

Ceci s'explique par l'existence de difficultés spécifiques à la production de logiciels. A partir d'une distinction entre l'essence et la substance d'un phénomène, reprise d'Aristote, Frederick P. Brooks divise les difficultés de la technologie logicielle en "essence" (les difficultés inhérentes à la nature du logiciel) et en "substance" (les difficultés qui gênent la production de logiciel, mais n'y sont pas inhérentes).

[11] Nous nous référons à la traduction française publiée dans la deuxième édition du "Mythe du mois-homme. Essais sur le génie logiciel" (1996).

L'essence d'une entité logicielle est un édifice fait de concepts étroitement imbriqués. La partie la plus difficile du développement d'un logiciel consiste à bâtir les structures conceptuelles complexes qui forment l'entité logicielle abstraite. En effet, ces tâches "essentielles" seront toujours difficiles en raison de quatre "propriétés intrinsèques de cette essence irréductible des systèmes logiciels modernes" (idem, p. 158) : la *complexité*, la *conformité* ("le logiciel doit se conformer à ce qui existe parce qu'il est le dernier arrivé sur le terrain" et "parce qu'il est perçu comme étant le plus facile à rendre compatible au reste"), la *variabilité* ("le logiciel d'un système lui fournit sa fonctionnalité, et celle-ci est la plus sujette au changement" et "le logiciel peut être modifié plus facilement que les objets matériels : ce n'est que de l'abstraction, infiniment malléable") et *l'invisibilité* ("un logiciel n'a pas de réalité intrinsèque dans l'espace", "les structures d'un logiciel restent intrinsèquement impossibles à visualiser", et les multiples représentations partielles possibles "ne donnent qu'une vision très réduite de l'enchevêtrement touffu du programme" ce qui "freine la conception chez une personne donnée et limite en outre sévèrement la communication entre personnes différentes").

La substance du logiciel est la représentation des entités abstraites (constitutives de l'essence du logiciel) dans un langage de programmation et leur conversion en langage machine. Les tâches substantielles consistent à effectuer cette représentation et à tester sa fidélité. Dans les premiers temps de la programmation, les tâches substantielles dominaient largement les tâches essentielles. De ce fait, les innovations concernant les tâches substantielles ont permis des "améliorations de plusieurs ordres de grandeur" (idem, p. 186) : les trois principales innovations ont été l'adoption progressive des langages de haut niveau, qui a multiplié la productivité par cinq, la substitution du temps partagé au traitement par lots, dont l'impact est jugé moins important, et l'utilisation largement répandue d'environnements de développement unifiés, qui a doublé ou triplé la productivité, voire plus (idem, p. 161-162).

Mais, du fait de ces améliorations, les tâches substantielles ne représentent plus, en 1986, qu'environ la moitié du total, voire moins, des tâches de développement d'un logiciel[12].

[12] D'une certaine façon on retrouve à l'intérieur même de la production des logiciels, la coexistence, qui est à la base des modèles de Baumol, d'activités ayant des potentiels de gains de productivités différents, les activités dont la croissance de la productivité est la plus forte (ici les tâches "substantielles") voyant logiquement leur importance relative diminuer. Par contre la divergence importante avec les hypothèses de Baumol est que les

Or, à partir du moment où les tâches substantielles constituent moins des neuf dixièmes du total, des innovations qui ne concernent que les tâches substantielles ne peuvent permettre de décupler la productivité, même si elles parvenaient à réduire à zéro les tâches substantielles (idem, p. 181). A partir de ce raisonnement, Frederick P. Brooks en déduit que les innovations des années quatre-vingt dont beaucoup espéraient qu'elles permettraient de résoudre les problèmes de productivité concernant la production de logiciels, ne pourront multiplier la productivité par plus de dix. En effet, selon l'auteur ces innovations "permettent d'améliorer uniquement les composantes d'expression des concepts" (idem, p. 169). Ces innovations, qualifiées de "fausses balles d'argent" sont l'utilisation d'Ada comme langage universel, la programmation à objets, l'utilisation de l'intelligence artificielle et des systèmes experts, la programmation "automatique", la programmation graphique (ou visuelle), la vérification de programme, et de nouvelles améliorations concernant l'environnement de programmation, les outils et les postes de travail.

Les facteurs qui concernent "l'essence" des logiciels

Cependant, en conclusion de son article, Frederick P. Brooks pointe quatre facteurs qui "s'attaquent à l'essence conceptuelle", des "approches très prometteuses", qui prennent pour cible l'essence du problème logiciel, c'est à dire l'élaboration de ces structures conceptuelles complexes. Il s'agit tout d'abord de la détection systématique "des grands concepteurs", en observant que "la construction de logiciel est un processus *créatif*" et que "les logiciels qui ont déchaîné la ferveur d'adeptes passionnés sont ceux qui ont été créés par une poignée de grands concepteurs, voire par un seul" (idem, p. 174).

Le deuxième facteur est le développement d'approches et d'outils de prototypage rapide de systèmes, utilisés pour la spécification itérative du cahier des charges. En effet, selon Frederick P. Brooks, "la plus importante fonction que remplissent les constructeurs de logciels vis-à-vis de leurs clients est l'extraction et l'affinage progressif des spécifications du produit" (idem, p. 172). Décider exactement ce qu'il faut construire, établir des spécifications techniques détaillées, y compris toutes les interfaces avec les utilisateurs, les machines et les autres systèmes logiciels, sont les parties les plus difficiles de la construction d'un logiciel. Elles peuvent avoir des conséquences catastrophiques sur le système résultant si elles sont mal faites, et elles sont les plus difficiles à rectifier ultérieurement. C'est pourquoi les possibilités

tâches essentielles ne se caractérisent pas par une absence de gains de productivité, même si ceux-ci sont moins élevés que pour les tâches "essentielles" (cf. infra).

d'effectuer de nombreuses itérations entre le client et le concepteur sont la source d'améliorations essentielles.

Le troisième facteur, qui est complémentaire du précédent, est le développement incrémental. Il s'agit par analogie avec l'évolution naturelle de ces structures si complexes que sont les êtres humains, de "faire grandir les logiciels au lieu de les construire" (idem, p. 173). En se basant sur une conception descendante, on commence par faire "tourner" un système embryonnaire, qui ne fait pas grand chose d'utile, puis progressivement on le dote de ses différentes fonctionnalités. Ce mode de développement a une influence très positive sur le moral des développeurs, permet de vérifier en permanence la conformité du système aux attentes des utilisateurs et de revenir facilement en arrière.

Le quatrième facteur est la "réutilisabilité", vu la faiblesse des coûts de reproduction de ce qui a déjà été développé. Celle-ci peut être pratiquée au niveau des différents composants d'un logiciel ou au niveau du produit global (progiciel). Selon Frederick P. Brooks, la réutilisation des composants se heurte à certains obstacles : elle nécessite que chaque composant puisse être spécifié complètement, sans être dépendant des propriétés d'autres composants (Maarten Boasson, 1998, p. 9) ; les composants doivent bénéficier d'une bonne conception et d'une excellente documentation, ce qui les rend beaucoup plus coûteux à réaliser que les composants à usage unique (nécessite trois fois plus d'efforts selon Frederick P. Brooks) ; surtout, il suffit que le consommateur potentiel *pense* (indépendamment du coût réel) qu'il sera plus coûteux de trouver et de vérifier un composant qui satisfasse ses besoins plutôt que d'en écrire un, pour qu'il écrive un nouveau composant, "réinventant la roue". Dans sa réponse de 1996, Frederick P. Brooks estime que si la réutilisation des composants a suscité beaucoup d'études, elle est "relativement peu mise en pratique aux Etats-Unis" à l'exception de quelques "communautés" (les mathématiciens, les développeurs de codes de calculs des réacteurs nucléaires, de modèles climatiques ou de modèles océaniques). Dans de tels cas, ce qui facilite la réutilisation c'est que les composants nécessitent pour leur mise au point un énorme effort intellectuel, et qu'il existe une riche nomenclature normalisée pour décrire les fonctionnalités de chaque composant.

Par contre la production de progiciels apparaît comme offrant des perspectives radicales d'amélioration de la productivité : "le développement du marché de masse est à mon avis la tendance à long terme la plus marquante du génie logiciel" ; en effet, "le coût du logiciel a toujours été un coût de développement pas un coût de duplication. (…) Partager ce coût entre

plusieurs utilisateurs (…) diminue radicalement le coût par utilisateur (…). La mise en service de n exemplaires d'un système logiciel multiplie par n la productivité de ses développeurs", (Frederick P. Brooks, 1996, p. 171)[13]. L'utilisation des progiciels a suscité au départ certaines réticences, en raison notamment d'exigences trop spécifiques des utilisateurs, mais la baisse du coût du matériel a favorisé leur diffusion[14] à des centaines de milliers (et dans certains cas à des millions) d'utilisateurs.

Cependant dans le cas des progiciels on peut de moins en moins mesurer l'évolution de la productivité en se basant uniquement sur les efforts de développement du logiciel original. Si les coûts de reproduction sont dérisoires, il faut également prendre en compte les activités de marketing, de commercialisation, et les services de support liés dont la part relative augmente fortement. De ce fait, l'utilisation d'indicateurs techniques de productivité, qui dans les hypothèses les plus favorables permettent uniquement d'évaluer l'évolution de l'effort de développement, devient de plus en plus inadaptée, et il est nécessaire de recourir à des indicateurs économiques de productivité. Toutefois il importe de préciser que pour toutes les sociétés qui produisent des logiciels, dont certaines sont appelées – improprement – éditeurs de progiciels, l'essentiel de la production consiste à développer et à mettre au point des logiciels. Même chez Microsoft, éditeur de progiciel qui supporte les dépenses les plus importantes de marketing, de commercialisation et de communication, la majeure partie des salariés est affectée aux activités de développement des logiciels.

3 - La croissance de la productivité mesurée par des indicateurs économiques

La mesure "traditionnelle" de la productivité, notamment dans les comptes nationaux, consiste à mesurer la production par la valeur ajoutée, à pratiquer une double déflation pour éliminer les évolutions de prix et à la diviser par les effectifs (ou les heures de travail). Les études qui traitent de la productivité pour la production des logiciels ne procèdent pas ainsi :

---

[13] D'une certaine façon, dans le cas extrême mais qui se développe rapidement où un logiciel est disponible sur un serveur et où le nombre de fois où il est téléchargé par des utilisateurs n'entraîne aucun coût supplémentaire pour le prestataire, se manifeste une forme "d'effet d'audience" (Jean Gadrey, 1996 A, p. 215). Cet effet vise à montrer comment dans des prestations "en public", dont l'exemple typique est le quintette à vents mis en exergue par William Baumol, on peut observer des gains de productivité, à condition de "se placer du point de vue de la consommation des prestations, et non du point de vue de leur production".

[14] "En 1960, l'acheteur d'une machine de deux millions de dollars pouvait se permettre d'en dépenser 250 000 de plus pour un programme de paie personnalisé (…). De nos jours, les acheteurs de machines de gestion à 50 000 dollars ne peuvent absolument pas se permettre de faire faire leurs propres programmes ; ce sont leurs procédures de paie qui s'adaptent aux progiciels disponibles" (Frederick P. Brooks, 1996, p. 171).

tout d'abord, la plupart mesurent la production par le chiffre d'affaires, ce qui ne pourrait permettre que d'évaluer ce que André Vincent (1968) appelle la productivité brute, moins significative que la productivité nette construite à partir de la valeur ajoutée. Surtout ces études utilisent des indicateurs non déflatés (a), ce qui s'explique par les difficultés pour déflater les données, en l'absence d'indice des prix adéquats (b).

a - L'utilisation d'indicateurs non déflatés

Ces indicateurs sont le chiffre d'affaires par employé, et plus rarement la valeur ajoutée par employé, mesurés en monnaie courante. S'il peut paraître abusif de parler de productivité à propos de ces indicateurs – ce que font le plus souvent les études citées - leur connaissance fournit néanmoins quelques informations intéressantes sur le secteur de la production de logiciels.

Eurostaf fournit des données sur le chiffre d'affaires et les effectifs de la plupart des entreprises produisant des logiciels. Le calcul du chiffre d'affaires par employé pour quelques producteurs importants au niveau mondial montre l'importance des écarts entre les sociétés.

Tableau XXIX
Chiffre d'affaires par employé de quelques producteurs importants de logiciels (en 1993)

Données 1993	**CA (millions de $)**	**Effectifs**	**C.A. / Effectif (en milliers de $)**
Microsoft	4649	14430	322,18
Computer Sciences	2583	28600	90,31
Computer Associates	2148	7200	298,33
Cap Gemini Sogetti	2076	20559	100,98
Novell	1123	4429	253,56
Adobe	313	999	313,31

Source : calculs effectués d'après les données Eurostaf (1995 B, p. 63).

L'examen de ce même indicateur à quatre années d'intervalle (en monnaie courante) pour les principales sociétés européennes montre qu'il existe également des différences importantes dans l'évolution du chiffre d'affaires par employé, particulièrement pour les entreprises produisant principalement des progiciels.

Tableau XXX
Evolution du chiffre d'affaires par employé des principales sociétés européennes
(C.A. par employé en milliers de F. courants)

	1991	1995	Taux de croissance	TCAM
Axime	636,62	739,08	16,09%	3,80%
SG2	509,1	596,31	17,13%	4,03%
Sligos	592,38	635,6	7,30%	1,78%
Baan	540,1	706,48	30,81%	6,94%
CCMX	671,96	634,78	-5,53%	-1,41%
Misys	486,96	523,7	7,54%	1,84%
Multihouse	792,94	704,63	-11,14%	-2,91%
SAP	983,34	1185,62	20,57%	4,79%
Software AG	850,6	834,73	-1,87%	-0,47%
Sopra	553,05	512,95	-7,25%	-1,86%
BSO/Beheer	478,37	482,72	0,91%	0,23%
Cisi	444,27	437,25	-1,58%	-0,40%
Data Sciences	529,25	505,42	-4,50%	-1,15%
debis Systemhaus	* 906,1	1014,3	11,94%	5,80%
Finsiel	527,1	629,59	19,44%	4,54%
Getronics	947,92	1074,23	13,32%	3,18%
Logica	426,43	503,11	17,98%	4,22%
Steria	485,39	494,30	1,84%	0,46%
Syseca	521,43	586,67	12,51%	2,99%
Unilog	472,33	487,25	3,16%	0,78%
Cap Gemini	558,01	544,62	-2,40%	-0,61%
Computer Sciences	378,67	442,94	16,97%	4,00%
EDS	533,01	612,63	14,94%	3,54%
Sema Group	435,79	528,17	21,20%	4,92%
Centre Inf. Catalogne	280,66	303,46	8,12%	1,97%
Centrisa	371,77	334,62	-9,99%	-2,60%
Datev	703,11	726,52	3,33%	0,82%

** 1993*

En italique, les sociétés produisant principalement des progiciels.

Source : Calculs effectués d'après les données Eurostaf (1996 B, p. 19).

Pour les principales sociétés françaises, où l'on dispose également d'indications sur la valeur ajoutée, on constate les mêmes disparités, avec des évolutions qui peuvent être très contrastées d'une année à l'autre.

Tableau XXXI
Evolution de la valeur ajoutée par employé des principales sociétés françaises (en milliers de F. courants)

	1992	1993	93/92 (en %)	1994	94/93 (en %)
Cap Gemini	374	370	-1,1	384	+2,7
CCMX	414	351	-15,2	361	+2,9
Cegid	396	440	+11,1	465	+5,7
Sema (K.E.)	35	36	+2,7	43	+19,5
Sligos	377	368	-2,4	357	-3
Sogeris	1087	475	-56,3	615	+29,5
Sopra	394	351	-10,9	363	+3,4
Unilog	384	372	-3,1	390	+4,8
Axime	313	374	+19,5	384	+2,7

Source : Eurostaf (1996 D, p. 41).

Pour l'ensemble des sociétés adhérentes à Syntec Informatique, l'évolution du chiffre d'affaires par employé (en francs courants) est inégale selon les années, mais est globalement positive avec un taux de croissance annuel moyen de 3,6 % sur la période 1988-1996.

Tableau XXXII
Evolution du chiffre d'affaires par employé des SSII françaises (en milliers de F. courants)

Année	C.A. /Effectif	Taux de croissance
1988	480,8	
1989	513	6,70%
1990	543	5,85%
1991	576,9	6,24%
1992	598,5	3,74%
1993	612,4	2,32%
1994	626,9	2,37%
1995	623,3	-0,57%
1996	637,9	2,34%

Source : calculs effectués d'après les données Syntec Informatique citées in Eurostaf (1997 B, p. 240).

Pour la France, on dispose également des données détaillées de l'INSEE pour le secteur 72 "Activités informatiques", sur le chiffre d'affaires, la valeur ajoutée et les effectifs. Pour les effectifs, on a additionné les effectifs salariés et les effectifs non salariés. Pour les effectifs salariés, nous avons calculé leur "équivalent temps plein" à partir des effectifs à temps partiel, en appliquant un coefficient de travail à temps partiel calculé à partir de l'année 1996, année où nous disposions de données sur le nombre de travailleurs à temps partiel et sur l'effectif salarié moyen. Le calcul du chiffre d'affaires par effectif pour les années 1993[15] à 1997 donne les résultats suivants:

Tableau XXXIII
Evolution du chiffre d'affaires par effectif en France
(en milliers de F. courant)

		1993	1994	1995	1996	1997	TCAM
72	**Activités informatiques**	633,236	623,945	660,389	682,007	712,135	2,98%
721	Conseil en systèmes informatiques	597,040	571,830	603,232	647,448	700,198	4,06%
722	Réalisation de logiciels	606,202	611,917	611,574	661,364	665,311	2,35%
723	Traitement de données	689,039	677,243	746,422	731,217	806,506	4,01%
724	Activités de banques de données	633,638	960,691	763,834	852,618	916,787	9,67%
725	Entretien et réparation de machines de bureau et de matériel informatique	656,771	631,163	784,006	691,900	683,758	1,01%

Source : calculs effectués d'après des données INSEE
in Annuaire statistique de la France (Editions de 1996 à 1999)
et Tableaux Economiques de la France 1999-2000 pour les données 1997.

[15] On ne dispose pas pour les années antérieures à 1993 des données selon la nouvelle nomenclature (NAF).

La croissance de la valeur ajoutée par effectifs apparaît plus rapide :

Tableau XXIV
Evolution de la valeur ajoutée par effectif en France
(en milliers de F. courants)

		1993	1994	1995	1996	1997	TCAM
72	**Activités informatiques**	330,448	336,234	366,451	384,838	16,46%	3,88%
721	Conseil en systèmes informatiques	320,759	327,408	365,150	393,108	22,56%	5,22%
722	Réalisation de logiciels	333,209	351,369	390,434	400,105	20,08%	4,68%
723	Traitement de données	346,294	343,613	356,674	383,492	10,74%	2,58%
724	Activités de banques de données	280,469	326,036	351,450	378,635	35,00%	7,79%
725	Entretien et réparation de machines de bureau et de matériel informatique	295,049	284,856	296,402	276,057	-6,44%	-1,65%

Source : calculs effectués d'après des données INSEE
in Annuaire statistique de la France (Editions de 1996 à 1999)
et Tableaux Economiques de la France 1999-2000 pour les données 1997.

Pour les Etats-Unis la connaissance du chiffre d'affaires[16] et des effectifs du secteur logiciel et services informatiques nous permet de mettre en évidence un doublement (en dollars courants) du chiffre d'affaires par employé entre 1985 et 1995.

Tableau XXXV
Evolution du secteur logiciel et services informatiques aux Etats-Unis
(en dollars courants)

	1985	1995	Taux de croissance	TCAM
C.A. (en millions de $)	45132	152213	237%	12,9%
Emploi	637409	1083977	70%	5,5%
CA / Employé (en milliers de $)	708,054	1404,209	98%	7,1%

Source : calculs effectués d'après les données OCDE (1998 A, p. 18-19).

[16] Le titre du tableau de l'étude de l'OCDE ("Le secteur du logiciel : un profil statistique pour certains pays de l'OCDE") dont sont extraites les données conncernant la production s'intitule "Valeur ajoutée". Toutefois il est précisé en note que pour les Etats-Unis, il s'agit des "estimations des recettes des entreprises imposables" et pour le Japon (cf. infra) des ventes !

Sur la période 1990-1995 qui se caractérise par une croissance moins rapide du chiffre d'affaires par employé, nous disposons de statistiques par sous-secteurs qui permettent de mettre en évidence un niveau plus élevé de cet indicateur pour les progiciels (un taux de croissance annuel moyen de 3,26 % à comparer à 1,53 % pour les "services de programmation informatique"), avec vraisemblablement des disparités inter-entreprises plus importantes à l'intérieur du sous-secteur des progiciels (cf. infra).

Tableau XXXVI
C.A. par employé pour les services informatiques aux Etats-Unis
(en milliers de $ courants)

	1990	1995	Taux de croissance global	Taux de croissance annuel moyen
737 Services de programmation informatique, traitement de données, et autres activités de services informatiques rattachées	**114,392**	**139,658**	**22,09%**	**4,07%**
7371 Services de programmation informatique	141,366	152,658	7,99%	1,55%
7372 Progiciels	**146,480**	**171,941**	**17,38%**	**3,26%**
7373 Configuration de systèmes informatiques intégrés	132,472	158,522	19,66%	3,66%
7374 Préparation et traitement de données, services de traitement	90,595	139,597	54,09%	9,03%
7375 Services de récupération d'information	74,361	96,467	29,73%	5,34%
Total 7376 à 7379 *	97,206	104,191	7,19%	1,40%

* 7376 : Services de sous-traitance informatique, 7377 Location et leasing informatiques, 7378 Maintenance et réparation informatique, 7379 Services informatiques n.c.a.

Source : calculs effectués d'après les données de l'OCDE (1998 A, p. 23-24

Pour le Japon, le chiffre d'affaires par employé (en monnaie courante) croît beaucoup plus rapidement qu'aux Etats-Unis mais reste à un niveau beaucoup plus faible qu'aux Etats-Unis. Ceci peut s'expliquer par la plus faible proportion de progiciels dans les logiciels au Japon et par leur introduction plus tardive.

Tableau XXXVII
Evolution du secteur logiciel et services informatiques au Japon
(en dollars courants)

	1985	1995	Taux de croissance	TCAM
C.A. (en millions de $)	655	6764	933%	26,3%
Emploi	162010	407396	151%	9,7%
C.A/Employé (en milliers de $)	4,042	16,603	311%	15,2%

Source : calculs effectués d'après les données OCDE (1998, p. 18-19).

Les différentes études statistiques font donc apparaître une croissance de la "productivité" (basée sur des grandeurs nominales) importante pour la production des logiciels, indépendamment des fortes variations selon les années et selon les entreprises. Il est vraisemblable que l'évolution réelle de la productivité est encore plus favorable, les logiciels à qualité constante se caractérisant par la baisse de leurs prix ou, plus fréquemment par une amélioration des fonctionnalités fournies pour un prix donné. Pour vérifier cette hypothèse et pouvoir estimer la croissance de la productivité, il faudrait pouvoir déflater les données.

b - Les difficultés pour déflater les données

Pour pouvoir appréhender l'évolution en volume d'une production à partir de la connaissance de son évolution en valeur, on déflate cette production par un indice des prix mesurant l'évolution des prix pour le secteur considéré, en tenant compte des effets qualités. Malheureusement de tels indices de prix n'existent pas pour le secteur du logiciel, du moins actuellement[17]. En effet, "définir, mesurer et enregistrer au cours du temps des prix, et

[17] L'INSEE mène actuellement des études pour construire un indice des prix des logiciels.

construire ensuite des indices de prix, exige le respect d'une condition essentielle : l'existence d'unités d'output suffisamment standardisées, dont la nature qualitative reste à peu près stable au cours de la période d'étude, ou à la rigueur dont les variations de qualité puissent faire l'objet d'une mesure acceptable lorsqu'on modifie la base statistique des biens pris en compte" (Jean Gadrey, 1996 A, p. 194). Il est clair que cette condition est très difficile à remplir pour les logiciels sur-mesure, et que pour les progiciels la mesure de l'effet-qualité est délicate à réaliser. Dans de telles circonstances, qui sont fréquentes dans les activités de services où il est souvent difficile de définir des unités produites et donc des prix unitaires, différentes méthodes, plus ou moins satisfaisantes sont utilisées (Jean Gadrey, 1996 A, p. 66 et suivantes).

Une première méthode consiste à déflater la valeur produite par le secteur en utilisant l'indice général des prix de l'économie[18]. Cette méthode ne peut être utilisée que si l'on suppose que l'évolution des prix du secteur n'est pas très différente de l'évolution du niveau général des prix, ce qui n'est manifestement pas le cas pour les logiciels.

Une deuxième méthode consiste à effectuer la déflation à partir d'un indice des prix dont on dispose pour un secteur complémentaire, correspondant en général à des biens. Elle est par exemple utilisée pour déflater la production des architectes-conseils, qui est effectuée par l'indice du coût de la construction (idem, p. 67). Appliquée au logiciel, elle consisterait à déflater la production de logiciels par l'indice des prix du matériel. Les estimations de l'évolution des prix du matériel divergent selon ce que l'on estime le plus représentatif (ordinateur ou composant électronique) et selon les méthodes utilisées pour tenir compte de l'amélioration de la qualité (traitement par chaînage ou méthode hédonique), mais dans tous les cas, l'importance des baisses de prix du matériel (cf. chapitre IV) ferait apparaître une très forte croissance de la productivité pour la production des logiciels. Même si on peut justifier cette façon de procéder, par les complémentarités entre les deux secteurs (des ordinateurs plus puissants permettent d'utiliser des logiciels aux fonctionnalités plus étendues et donc plus complexes à réaliser), il ne nous semble pas que cette méthode puisse être retenue valablement.

[18] Cette méthode est fréquemment utilisée pour les services d'assurance où on déflate les primes par l'indice général des prix à la consommation (Jean Gadrey, 1996 A, p. 199).

Une troisième méthode consiste à déflater par l'évolution des rémunérations des salariés du secteur[19]. Cette méthode est très contestable, notamment pour les logiciels sur mesure, où la tarification de fait en régie est le plus fréquemment pratiquée, conséquence d'une situation où il n'existe pas d'output séparable économiquement de l'activité de production. Dans ce cas l'évaluation de la production réalisée repose principalement sur les salaires versés, et cette méthode peut être inadéquate pour appréhender l'évolution de la productivité. En effet, une augmentation des salaires moyens peut très bien ne pas correspondre à une augmentation des prix à qualité constante, mais traduire une augmentation de la complexité moyenne des prestations réalisées et donc être représentative d'une amélioration d'une qualité des prestations réalisées. Le calcul et l'interprétation des indicateurs pour deux entreprises dont l'activité principale consiste à réaliser des prestations sur mesure peuvent permettre d'illustrer ce raisonnement.

[19] C'est la méthode qui est utilisée pour mesurer la production de l'ingénierie "en volume" qui est déflatée par l'évolution des honoraires des ingénieurs-conseils (Jean Gadrey, 1996 A, p. 82).

Tableau XXXVIII
Evolution de quelques indicateurs
de Cap Gemini Sogeti

	1989	1990	1991	1992	1993	TCAM
C.A. consolidé (en millions de F. courants)	7055	9172	10028	11884	11027	
Variation (en %)		30,00	9,33	18,51	-7,21	11,81
Effectif moyen	12974	16490	17971	21675	20898	
Variation (en %)		27,10	8,98	20,61	-3,58	12,66
C.A consolidé/Effectif moyen (en milliers de F. courants)	543,8	556,2	558,0	548,3	527,7	
Variation (en %)	7,1	2,2	0,3	-1,8	-3,8	-0,75
Frais de personnel/Effectif moyen (en milliers de F. courants)	315,5	316,2	329,0	340,8	332,6	
Variation (en %)	7,9	0,3	4,1	3,6	-2,4	1,33
C.A consolidé/Effectif moyen (déflaté)	543,8	554,97	535,10	507,60	500,57	
Variation (en %)		2,05	-3,58	-5,14	-1,38	-2,05

Source : calculs effectués d'après des données Eurostaf (1994 B, p. 44).

Tableau XXXIX
Evolution de quelques indicateurs
de Sema Group

	1991	1992	1993	1994	1995	TCAM
C.A. consolidé (en millions de £ courantes)	412	417	502	596	678	
Variation (en %)		1,21	20,38	18,73	13,76	13,26
Effectif moyen	7441	6919	7254	8154	8737	
Variation (en %)		-7,02	4,85	12,41	7,15	4,10
C.A consolidé/Effectif moyen (en milliers de £. courantes)	55,37	60,27	69,2	73,09	77,6	
Variation (en %)		8,85	14,82	5,62	6,17	8,80
Frais de personnel/Effectif moyen (en milliers de £. courantes)	28,91	32,28	34,45	34,46	37,26	
Variation (en %)		11,66	6,72	0,03	8,13	6,55
C.A consolidé/Effectif moyen (déflaté)	55,37	53,98	58,07	61,32	60,21	
Variation (en %)		-2,51	7,58	5,59	-1,81	2,12

Source : calculs effectués d'après des données Eurostaf (1996 E, p. 62).

Sur des périodes de cinq ans ces deux entreprises connaissaient des progressions voisines de leur chiffre d'affaires (un taux de croissance annuel moyen de 11,8 % pour Cap Gemini Sogeti et de 13,26 % pour Sema Group). Mais dans la mesure où les effectifs augmentaient beaucoup plus rapidement pour Cap Gemini Sogeti que pour Sema Group, l'évolution du chiffre d'affaires par employé était logiquement très différente. Sur cinq ans, cet indicateur baissait en moyenne de 0,75 % par an pour Cap Gemini Sogeti alors qu'il augmentait fortement pour Sema Group (taux de croissance annuel moyen de 8,8 %). Il peut sembler contestable que cet indicateur basé sur des grandeurs nominales rende correctement compte de l'évolution de la productivité. Mais la déflation par l'évolution des frais de

personnel unitaires de l'entreprise risque d'induire également des conclusions erronées. En appliquant cette technique, l'évolution de la "productivité" apparaît négative pour Cap Gemini Sogeti (-2,05 % en moyenne annuelle) et faible pour Sema Group (2,12 %) dans la mesure où on considère que l'activité "réelle" a connu une croissance plus faible que ce que montrait l'évolution nominale du chiffre d'affaires, qui recouvrait pour partie une augmentation apparente des prix estimée par l'augmentation des salaires versés[20]. Or il est vraisemblable qu'au début des années quatre-vingt-dix, période de récession passagère de l'activité informatique, l'essentiel de l'augmentation des rémunérations provenait d'un effet de structure d'augmentation de la qualification moyenne des personnels, qui s'expliquait par la complexité croissante des prestations à réaliser. Si cette hypothèse est vérifiée, la croissance de l'activité réelle (et de la productivité) serait plus forte que ce que montrent les données déflatées par les rémunérations. Ce qui donne du crédit à cette hypothèse, c'est la comparaison entre les données pour les deux entreprises. Alors que la différence entre les taux de croissance annuels moyens de la "productivité" en monnaie courante était de près de 10 points, elle n'est plus que de 4 points pour les données déflatées. C'est la conséquence de l'évolution beaucoup plus rapide des rémunérations unitaires à Sema Group qu'à Cap Gemini Sogeti. Mais l'explication probable de cette différence est une amélioration plus importante des compétences utilisées, correspondant à un prompt positionnement de Sema Group sur des prestations plus sophistiquées[21] (intégration de système, infogérance) (Eurostaf, 1996 E, p. 7). En l'absence d'indications complémentaires, la déflation par les salaires apporte peu de connaissances sur l'évolution de la productivité et peut induire des contresens d'interprétation.

Une tentative de déflation pour le sous-secteur des progiciels

Par contre, si on se limite au sous-secteur des progiciels, on peut tenter d'effectuer une déflation par un indice des prix spécifique relativement représentatif. Une étude effectuée par

[20] En théorie si l'ensemble des prestations était uniquement effectué en régie et était constitué uniquement de travail direct, la déflation par l'évolution des rémunérations, conduirait à une "productivité" stationnaire, à l'exception d'écarts pouvant résulter des difficultés d'adaptation des effectifs à l'évolution de l'activité, ce que semble confirmer l'évolution des indicateurs de Cap Gemini Sogeti sur la période. Le risque, souligné par Jean Gadrey dans le cas de nombreuses activités de service, est de mesurer "un pseudo-produit, qui reflète mal l'output qui importe vraiment, économiquement et socialement, à savoir les services rendus" et de l'utiliser pour calculer une "pseudo-productivité qui peut fort bien être stagnante ou décliner, alors que l'importance des services rendus (par heure de travail) progresse" (1996 A, p. 169-170).

[21] Même, si par définition dans le cas d'une prestation sur mesure, il est difficile de définir des catégories fines de produits indépendamment du prestataire, les services informatiques sont répartis (imparfaitement, avec des nomenclatures différentes dans le temps et selon les sources) dans des sous-secteurs où la complexité de l'activité est en moyenne différente (cf. chapitre IV).

Eurostaf sur un échantillon de 18 entreprises de progiciels, le plus souvent leaders dans leur secteur, évaluait la croissance de la "productivité" du travail (mesurée par le chiffre d'affaires par employé en monnaie courante) à 20,31 % entre 1990 et 1994, soit un taux de croissance annuel moyen de 4,73 %, avec de fortes disparités entre les sociétés : SAP avait sa "productivité" qui augmentait de 53,1 % (TCAM de 11,24 %), Microsoft de 65,6 % (TCAM de 13,44 %), Informix de 75 % (TCAM de 15,02 %), le record étant Nat Systems, spécialisé dans les outils de développement clients-serveurs avec une croissance de 107,5 % (TCAM de 20,02 %). Cette étude précisait que les prix des progiciels étaient en baisse, d'où une augmentation réelle de la productivité plus importante, mais ne fournissait pas d'indications sur l'importance de la baisse des prix (Eurostaf, 1996 C, p. 164).

Pour les Etats-Unis, nous disposons de données sur le chiffre d'affaires et les effectifs pour le sous-secteur 7372 "Progiciels", que nous pouvons essayer de déflater. L'étude des prix des progiciels pose des problèmes : évolution spectaculaire des prix en fonction des stratégies commerciales, tarification différenciée selon les utilisateurs (cf. chapitre III). Des indications partielles semblent témoigner de fortes baisses de prix alors même que les fonctionnalités des progiciels augmentent : par exemple, Eurostaf estime que les prix des progiciels bureautiques ont été divisés par 3 en 1993 avec l'arrivée sur le marché des suites bureautiques (Eurostaf, 1995 B, p. 26) ; selon la Software Publishers Association, le nombre d'unités de logiciels d'application pour ordinateur personnel vendues, pour l'Europe de l'Ouest, avaient augmenté en 1994 par rapport à 1993 de 69 %, mais leur valeur correspondante ne s'était accrue que de 4 %, ce qui correspondait à une baisse du prix moyen de 38,4 % (OCDE 1995 B p. 149). Mais à notre connaissance, le seul indice existant aux Etats-Unis est un indice portant sur les logiciels d'application pour ordinateur personnel basé sur la méthode des modèles correspondants, et qui fait état d'une baisse plus modeste. Si nous l'utilisons pour déflater le chiffre d'affaires des progiciels, en faisant l'hypothèse – discutable - qu'il reflète l'évolution des prix de l'ensemble des progiciels, nous obtenons les résultats suivants :

Tableau XXXX
Productivité brute pour les progiciels aux Etats-Unis

	1990	1991	1992	1993	TCAM
C.A nominal (en millions de $ courants)	16523	18306	21236	24648	
Variation (en %)		10,8%	16,0%	16,1%	14,26%
Evolution annuelle des prix (en %)		-1,6	-6,5	-0,8	
C.A réel (en millions de $, aux prix de 1990)	16523	18604	23082	27006	
Variation (en %)		12,59%	24,07%	17,00%	17,79%
Effectifs	112800	124400	130800	144800	
Variation (en %)		10,28%	5,14%	10,70%	8,68%
C.A./Effectif (en milliers de $ courants)	146,480	147,154	162,354	170,220	
Variation (en %)		0,46%	10,33%	4,85%	5,13%
Productivité (C.A. /Effectif en milliers de $ aux prix de 1990)	146,480	149,547	176,464	186,506	
Variation (en %)		2,09%	18,00%	5,69%	8,39%

Source : calculs effectués d'après des données OCDE (1998, p. 24,25 et 41).

Indépendamment de l'amplitude des fluctuations (qui reflète en partie celle des prix), la productivité brute a connu une croissance importante (un taux de croissance annuel moyen de plus de 8 %). Il faut toutefois noter que ce qui est mesuré c'est l'évolution de la productivité au sein du sous-secteur des progiciels. La forte hausse constatée peut s'expliquer par la diminution de la quantité de travail pour produire un original (cf. infra la productivité mesurée par des indicateurs techniques) mais également (et peut-être surtout) par le fait qu'un même progiciel est vendu à un nombre croissant d'utilisateurs. Par contre, ce que nous ne pouvons

mesurer (en raison de l'absence d'indice des prix adéquat) et qui représente vraisemblablement la plus grande partie des gains de productivité dans la production des logiciels considérée globalement, *ce sont les conséquences de la substitution de progiciels à des logiciels sur mesure*. La méthode, assez proche de certaines techniques parfois utilisées[22], qui consisterait à déflater l'ensemble de la production de la branche des logiciels par un indice des prix des progiciels nous semble trop contestable pour pouvoir être utilisée, dans la mesure où il est très peu vraisemblable que l'évolution des prix des progiciels soit représentative de l'évolution des prix de l'ensemble des logiciels. Toutefois une indication des gains de productivité résultant de la substitution des progiciels aux logiciels sur mesure, est constitué par la formidable baisse des prix qui résulte, pour la résolution d'un problème donné (exploitation d'un ordinateur, paye, comptabilité…), du passage d'une solution basée sur des développements sur-mesure à une solution basée sur des progiciels pour remplir les mêmes fonctionnalités[23] : "un progiciel entier aux riches fonctionnalités coûte moins cher qu'une journée de programmeur avec ses coûts indirects" (Frederick P. Brooks, 1996, p. 244).

C - …MAIS INSUFFISANTE FACE A L'AUGMENTATION DE LA DEMANDE DE LOGICIELS

Que conclure de l'examen de ces différents indicateurs techniques et économiques ? Même si nous ne prétendons pas avoir réussi à mesurer globalement de façon précise et rigoureuse l'évolution de la productivité dans la production de logiciels, nous pensons avoir rassemblé suffisamment d'indices (au sens policier du terme) partiels et convergents pour pouvoir affirmer qu'il y a eu une incontestable augmentation de la productivité dans la production des logiciels. Il est toutefois vraisemblable que ces gains de productivité sont bien inférieurs aux gains exceptionnels constatés pour le matériel informatique. Cependant

[22] Par exemple, pour les services bancaires et financiers, le produit en valeur de cette branche est déflaté par un indice des prix de divers services annexes de gestion (location de coffres, gestion de portefeuilles, placement de titres) plus faciles à repérer mais qui ne représentent qu'une faible partie de l'activité (Jean Gadrey, 1996 A, p. 67).

[23] En réalité les fonctionnalités ne sont jamais rigoureusement identiques : en général le passage à un progiciel permet de fournir une gamme plus large de fonctionnalités ; par contre, étant conçu pour répondre à des besoins standards, un progiciel peut être moins adapté aux spécificités de chaque problème concret qu'un logiciel sur mesure, sauf à entreprendre des développements complémentaires.

"l'anomalie n'est pas la lenteur des progrès du logiciel, mas la rapidité de ceux du matériel"[24] (Frederick P. Brooks, 1996, p. 157).

En conséquence de l'importance des progrès de la productivité dans le matériel informatique, la poursuite de la croissance du nombre des ordinateurs, de l'augmentation de leur puissance et de la baisse de leurs prix, fait que l'amélioration de la productivité dans la production des logiciels continuera à constituer un défi permanent, face à "l'explosion de la demande de logiciels" (OCDE, 1991 A). Certes la vision "industrialiste" qui, en mettant au premier plan le matériel informatique, considère que le secteur du logiciel est seulement "tiré" par la croissance du matériel[25], doit être relativisée. A l'heure actuelle, les relations entre les deux secteurs sont certainement beaucoup plus équilibrées, et dans de nombreuses situations, c'est la volonté de pouvoir utiliser les possibilités nouvelles offertes par les logiciels les plus récents, qui entraîne un renouvellement accéléré du matériel.

L'écart persistant entre les gains de productivité dans le matériel et dans les logiciels confirme la thèse de William Baumol d'une part grandissante des dépenses en logiciels dans les dépenses informatiques, même si les statistiques disponibles (cf. chapitre IV) font état d'une évolution beaucoup moins rapide que celle qu'il mentionne. Par contre l'augmentation incontestable de la productivité dans la production des logiciels nous conduit à rejeter la classification effectuée par William Baumol du secteur informatique (intégrant les logiciels et les matériels) dans les secteurs asymptotiquement stagnants. Il semblerait plus juste de caractériser le secteur informatique comme étant un *secteur asymptotiquement croissant*[26]**,**

---

[24] Frederick P. Brooks estime que " l'explosion de la technologie des ordinateurs qui a connu des progrès sans équivalent dans l'histoire humaine" a permis de "multiplier par au moins mille durant ces vingt dernières années" la productivité dans la fabrication du matériel" (1996, p. 222).

[25] Cf. par exemple le raisonnement tenu par Frédéric Georges Roux (1991) : "Les progrès de la technologie alliés aux effets bénéfiques de la concurrence, qui fait pression sur les prix, font que la puissance moyenne de chaque unité centrale installée double environ tous les trois ans. A ce rythme, dans dix ans, l'ordinateur moyen sera dix fois plus puissant qu'aujourd'hui. Comme il y en aura dix fois plus, c'est cent fois plus de MIPS qui seront là. (...) Ces MIPS, cent fois plus, les utilisateurs les consommeront, sinon les constructeurs les garderont en stock ou ne les produiront pas. Or, il faut prendre conscience que si cela arrivait, IBM, DEC, les deux à la fois, feraient faillite. C'est une hypothèse que je ne saurais retenir comme probable. Donc les MIPS seront là. (...) Donc, il y aura de nouvelles applications qui utiliseront toute cette puissance nouvelle disponible".

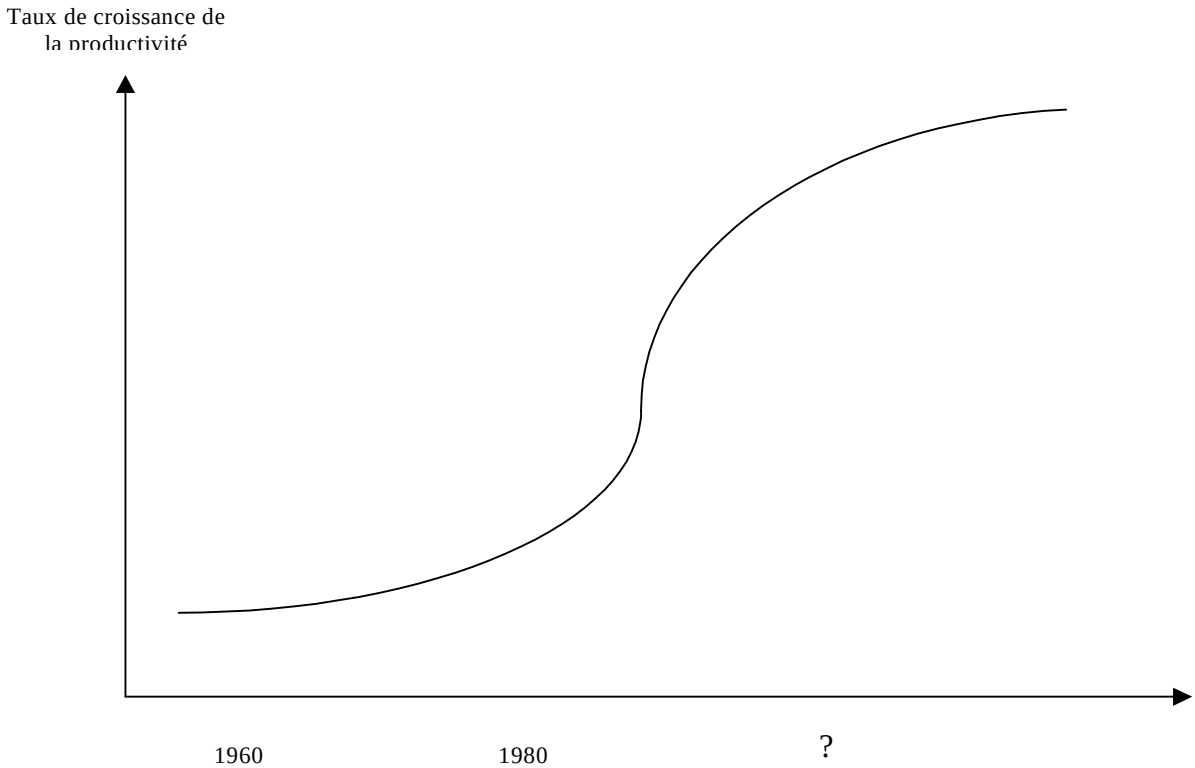
[26] William Baumol, au vu d'une version provisoire de cette partie sous la forme d'un document de travail, tout en soulignant son accord avec l'analyse principale et sa conclusion (l'informatique comme un secteur asymptotiquement croissant), estime qu'à long terme "la question de la stagnation peut finalement surgir, la pensée étant un des inputs du développement des logiciels (…) et la pensée moderne n'étant pas plus productive que celle des grands mathématiciens du XVIIème siècle".

dont la pente de l'asymptote est le rythme de la progression de la productivité dans la production des logiciels, tant que la productivité croît plus rapidement dans le domaine du matériel que dans le domaine du logiciel.

La productivité continuera-t-elle à croître plus rapidement pour le matériel que pour les logiciels ? La réponse à cette question est délicate. Du côté du matériel, il semble que la poursuite de la "loi de Moore" (à la base des progrès de productivité) puisse se heurter à des limites physiques d'ici une dizaine d'année, et il est à l'heure actuelle difficile de savoir si de nouvelles technologies (encore au stade expérimental) permettront les mêmes gains de productivité. Du côté de la production des logiciels, où il n'existe pas de limites physiques à la poursuite de la croissance de la productivité, les prévisions sont également très incertaines. Les tendances observées sur les évolutions (passées et actuelles) dans la production des logiciels peuvent toutefois laisser supposer que la croissance de la productivité suit une "courbe en S".

La "courbe en S" de la productivité a été mise en évidence par Jean Gadrey (1999 C, p. 20) pour certains services (commerce, transports, télécommunications, banques et assurances).

Graphique IX



La première phase, correspondant dans la représentation pour les services de Jean Gadrey à un "stade de modernisation lente d'un service traditionnel" (1999 C, p. 20), couvrirait pour les logiciels la période 1960-1980. Dans cette phase, les logiciels sont principalement produits sur-mesure et l'augmentation de la productivité résulte essentiellement des améliorations introduites dans les méthodes de développement, les langages et les outils utilisés.

La deuxième phase qui commence dans les années quatre-vingt, voit la croissance de la productivité s'accélérer, principalement en raison de la part croissante des progiciels dans les logiciels. En effet, la production des progiciels recouvre des activités où les gains de productivité sont moyens (conception de l'original, marketing, commercialisation…) et des activités de reproduction, où l'amélioration de la productivité est très importante avec la croissance du nombre d'exemplaires fournis. Elle comprend également la fourniture de services de support, où les gains de productivité sont plus faibles, mais qui représentent une part modeste de l'activité, au moins dans un premier temps. Elle s'apparente à la phase "de production de masse d'un service standardisé permettant de réaliser des économies d'échelle" (Jean Gadrey, 1999 C, p. 20) observée pour certains services.

Jean Gadrey dans le cas des services identifie une troisième phase dite de "service à valeur ajoutée, où l'on réalise encore des gains de productivité mais à un rythme inférieur" (idem) en soulignant que, dans cette phase, la productivité perd de son sens comme mesure de la performance. Cette évolution correspond à la montée en puissance d'un "sur mesure de masse" dans le cas de la production des logiciels, dont nous avons identifié deux composantes principales : la fourniture avec un progiciel d'une proportion croissante de services divers et variés ; le développement de logiciels sur mesure en réutilisant des modules déjà développés et testés. Dans les deux cas, il s'agit de fournir une solution plus adaptée aux besoins spécifiques de l'utilisateur que ne peut le faire par définition un produit standard (cf. chapitre VII).

Si ces formes de production sont appelées à se développer et si elles sont porteuses d'une moindre croissance de la productivité au sens classique, cela n'implique pas que pour l'ensemble de la production des logiciels, le point d'inflexion de la courbe de croissance de la productivité soit atteint rapidement. En effet, plusieurs facteurs peuvent permettre à la productivité de continuer à croître à un rythme soutenu dans la période actuelle. Tout d'abord les solutions dites "sur mesure de masse" peuvent se substituer à des développements de

logiciels sur mesure (et non à des progiciels) et dans ce cas contribuer à améliorer la productivité. Ensuite et surtout, la production de progiciels pour répondre à des besoins standards a encore des perspectives de croissance très importantes : poursuite de la substitution des progiciels aux logiciels sur mesure dans des domaines encore peu couverts par des progiciels, domaines nouveaux couverts d'emblée par des progiciels, et extension à de nouveaux utilisateurs qui utiliseront des progiciels déjà existants, d'où des gains de productivité très importants.

Par contre, le développement de la fourniture d'une solution adaptée à un problème particulier met au premier plan la nécessité *de prendre en compte d'autres dimensions que la productivité du travail pour apprécier l'efficacité dans la production des logiciels.* Elle repose sur le fait que ce qui importe le plus, est moins la mesure du produit immédiat de l'activité que l'évaluation des résultats indirects de cette activité pour les utilisateurs. Cette évaluation nécessairement multidimensionnelle est encore plus complexe à effectuer dans la mesure où interviennent des facteurs extérieurs à l'activité du prestataire (caractéristiques de l'environnement, compétences des utilisateurs…), où l'horizon temporel à prendre en compte est plus important, et où elle nécessite d'analyser de multiples aspects qualitatifs difficiles à appréhender.

En 1989, Capers Jones notait que "la productivité de la programmation est devenue un sujet majeur de préoccupations sur le plan international, et la qualité des programmes a de fortes chances de prendre bientôt la même importance" (1989, p. 14). La qualité d'une solution logicielle comporte deux dimensions principales : l'adaptation aux besoins des utilisateurs et la fiabilité. Le fait de traiter à part la question de la fiabilité s'explique par son importance particulière pour un produit destiné à réaliser des traitements automatiques[27]. Elle présente toutefois le risque d'introduire des redondances dans l'analyse, certaines explications étant communes à ces deux caractéristqiues fondamentales de la qualité des logiciels.

[27] Capers Jones considère même que la fiabilité constitue l'aspect central de la qualité pour un logiciel : "Il existe de nombreuses définitions du mot "qualité" associées à d'aussi nombreuses ambiguïtés. Pour les objectifs pratiques de tous les jours, on peut considérer qu'un système de programmation de qualité supérieure est un système sans un seul défaut susceptible de l'arrêter complètement ou de fournir des résultats inacceptables" (1989, p. 47).

Section II - La fiabilité

La présence de défauts dans les logiciels est certainement l'attribut de la programmation qui affecte le plus les perceptions qu'ont les utilisateurs finals des programmes (Capers Jones, 1989, p. 207)[28]. En raison de l'importance acquise par les logiciels dans de multiples aspects de la vie économique et sociale, où la sécurité et la fiabilité reposent fréquemment sur des dispositifs logiciels, leurs conséquences peuvent atteindre potentiellement l'ensemble de la population, utilisatrice indirecte et le plus souvent non consciente de multiples composants logiciels, comme l'a révélé la mobilisation impressionnante autour du bogue de l'an 2000. Et pourtant malgré les efforts déployés pour corriger les erreurs, les logiciels contiennent toujours des défauts (A). Ceux-ci étant nécessairement le produit d'erreurs humaines, ils sont de moins en moins acceptés vu le rôle croissant des logiciels (B), alors même que les défauts dans les logiciels sont de plus en plus difficiles à éliminer (C).

A - CONSTAT ET PREMIERES EXPLICATIONS

1 - Les défauts des logiciels

Le constat de la présence de défauts dans les logiciels est très largement partagé par les utilisateurs, la presse spécialisée, qui rapporte de nombreux exemples souvent spectaculaires, et même par les producteurs des logiciels : "les systèmes sont toujours remplis de bogues"[29] reconnaissent les dirigeants de diverses grandes sociétés dans des entretiens avec Capers Jones (1989, p. 15). Un défaut dans un logiciel peut, dans certaines circonstances, provoquer une défaillance informatique. Cette défaillance informatique est parfois qualifiée de simple (*fault*) quand on peut la récupérer, ou de grave (*failure*) quand elle provoque l'arrêt définitif du système. Ces défauts concernent tous les types de logiciels, depuis les progiciels destinés au grand public[30] jusqu'à des logiciels critiques.

---

[28] Cet auteur souligne que "le taux de défaut est l'élément de coût le plus important de la programmation" (Capers Jones, 1989, p. 16).

[29] Un (ou une) bogue est "un défaut de conception ou de réalisation d'un programme se manifestant par des anomalies de fonctionnement" (Journal Officiel du 19 février 1984).

[30] Daniel Icbiah (1995), biographe pourtant admiratif de Bill Gates, considère qu'un des plus grands exploits du président de Microsoft, avait été une présentation effectuée pour la commercialisation d'une des premières versions du traitement de texte Word ; alors que ce logiciel contenait encore de multiples défauts, Bill

Le logiciel est à l'origine d'un grand nombre d'erreurs et de pannes des systèmes (OCDE, 1991 A, p. 9). On peut citer, entre autres exemples : le cas d'une personne de 106 ans devant se présenter au CP de l'école communale sous peine de retrait des allocations familiales pour ses parents ; les 22 000 ayants droit de la Caisse d'Allocations Familiales de Vesoul qui ont touché deux fois leurs prestations ; les 486 750 gagnants d'un million de pesos à un jeu promotionnel organisé par Pepsi-Cola à Manille, par suite d'une erreur dans le progiciel devant imprimer le numéro millionnaire ; l'existence de rames de métro "fantômes" à San Francisco ; le cas d'un système d'informations sur la circulation routière affichant que la circulation est nulle quand elle est totalement bloquée ; les premières versions du logiciel de navigation écrit pour le chasseur F16 qui provoquaient un retournement de l'avion lorsqu'il passait l'équateur ; l'inondation en 1983 de la vallée du Colorado River, due à une mauvaise modélisation du temps d'ouverture des barrages. Dans certains cas les explications a posteriori apparaissent très simples (oubli qu'une personne peut dépasser l'âge de 99 ans, fluidité du trafic mesuré par des capteurs se basant sur la vitesse de circulation des véhicules qui n'existaient plus pour le système quand ils ne pouvaient plus avancer du tout), mais certaines défaillances peuvent rester inexpliquées très longtemps (le cas des rames de métro "fantômes" n'a jamais été expliqué). Surtout des erreurs minimes peuvent avoir des conséquences majeures : une erreur dans une équation d'un programme (l'omission d'une "barre" sur un symbole indiquant qu'il fallait utiliser des valeurs moyennes) fut la cause de la destruction par erreur d'une fusée Atlas-Agena d'une valeur de 18,5 millions de dollars le 22 juillet 1962 (Lauren Ruth Wiener, 1994, p. 27) ; une virgule à la place d'un point fit passer la sonde de la mission Venus à 500 000 km de la planète au lieu des 5 000 km prévus ; une erreur de signe fit pointer une fusée russe sur Hambourg au lieu du pôle Nord (Patrick Jaulent, 1992, p. 17) ; une variable codée sur 16 bits, alors qu'une autre partie du programme pensait qu'elle était codée sur 32 bits fit perdre cinq millions de dollars à la Bank of New York en 1985[31]. Plus récemment, c'est une erreur de programmation des plus simples qui entraîna la perte d'Arianne V lors de son premier vol en juillet 1996.

Gates avait réussi à faire une démonstration d'utilisation du traitement de texte sans que le système ne se bloque en mémorisant de longues suites d'instructions qui évitaient soigneusement les défauts résiduels.

[31] Cette variable servait de compteur dans un logiciel pour traiter les transactions sur les titres d'Etat émis par la banque centrale américaine. Ce logiciel, qui fonctionnait normalement depuis un certain temps, devait traiter un nombre de transactions particulièrement élevé le mercredi 20 novembre 1985, et par suite de l'erreur sur la capacité du compteur, se mit à transcrire les nouvelles informations en recouvrant les transactions précédentes ; au bout de 90 minutes quand on constata l'incident et qu'on interrompit les transactions, la Bank of New York devait 32 milliards de dollars à la Réserve Fédérale qu'elle ne pouvait pas récupérer auprès des autres banques (Lauren Ruth Wiener, 1994, p. 33 et 66).

Le nombre réel de défauts que contient un logiciel au moment de sa commercialisation est évidemment inconnu. Une mesure habituelle consiste à recenser le nombre de défauts apparaissant ultérieurement lors de l'exploitation du logiciel, ce qui conduit à une estimation inférieure à la réalité. En effet, certains défauts peuvent très bien ne jamais se manifester ou se manifester après la période d'observation[32], ou on ne peut attribuer à un logiciel précis les dysfonctionnements constatés d'un système. Un taux d'erreur est calculé en divisant le nombre d'erreurs découvertes en exploitation divisé par le volume du logiciel en milliers de lignes sources (exprimé en KLS, Kilo Instructions Sources livrées).

L'ordre de grandeur de ce taux d'erreur fait l'objet d'estimations divergentes. Toran Demarco et Timothy Lister considèrent que l'industrie du logiciel a habitué ses clients à accepter des programmes qui présentent en moyenne de dix à trente défauts par milliers de lignes de code (1991, p. 31). Une étude conduite par Capers Jones en 1983 sur de gros systèmes rédigés en assembleur considérait que les logiciels contenaient initialement plus de 50 défauts pour 1000 instructions de code-source ; comme les différentes opérations pour supprimer ces défauts avant la livraison du logiciel (cf. 3) ne permettaient pas d'en éliminer plus de 85 %[33], il restait de l'ordre de 7,5 défauts pour 1000 lignes source dans les programmes livrés aux utilisateurs (Capers Jones, 1989, p. 48). Lors d'une étude ultérieure sur des logiciels commercialisés dont les taux de défaut après livraison ont été quantifiés, Capers Jones estimait que les logiciels contenaient au moment de leur remise au client de l'ordre de 4 ou 5 défauts au moins pour 1000 instructions. Jacques Printz estime que le taux d'erreur se situe entre un et cinq (1998, p. 54). Il s'agit d'estimations moyennes avec des écarts importants. En effet, le nombre de défauts est le "résultat final d'interactions entre la nouveauté du programme ou du système à réaliser, les compétences du personnel chargé de l'étude, les méthodes adoptées pour la conception et le codage, et encore beaucoup d'autres facteurs" (Capers Jones, 1989, p. 211-212). Capers Jones cite le cas d'un programme commercial dont on a découvert 31 erreurs pour 1000 instructions de code source durant les trois premières années d'exploitation (idem, p. 209). Plus récemment le nombre de défauts de Windows 2000 a été estimé à 65 000, à rapporter aux 35 millions de lignes de codes de ce

[32] Un défaut fut découvert pour un système de contrôle aérien dans un morceau de programme écrit en Cobol qui n'avait pas été modifié depuis vingt ans (Lauren Ruth Wiener, 1994, p. 137).

[33] Un taux d'élimination des défauts pour l'ensemble des opérations avant livraison de 85 % est selon l'auteur considéré comme maximal. Dans les cas les moins favorables observés par Capers Jones, le rendement d'élimination des défauts n'était que de 44 % (1989, p. 209).

logiciel. Par contre la NASA arrive à des taux beaucoup plus faibles : 0,11 erreur/KLS pour le logiciel embarqué de la navette spatiale (qui avait un volume de 500 KLS ce qui fait 55 erreurs)[34], et 0,4 erreur/KLS pour le logiciel au sol d'un volume de 1700 KLS, soit 680 erreurs (Jacques Printz, 1998, p. 53).

Il faut toutefois signaler que rapporter le nombre d'erreurs aux lignes de code source présente les mêmes biais statistiques que ceux mis en évidence dans l'évaluation de la productivité, en raison de l'hétérogénéité des langages de programmation. En effet, surtout pour les programmes d'une certaine taille, la majeure partie des erreurs ne sont pas des erreurs de codage, mais des erreurs imputables aux spécifications et à la conception (Capers Jones, 1989, p. 212). C'est pourquoi il est plus pertinent, mais plus coûteux à mesurer, de diviser le nombre de défauts résiduels par le nombre de points de fonction. En 1998, Capers Jones estime qu'en moyenne un logiciel livré contient environ 0,75 défaut résiduel par point de fonction, correspondant à cinq défauts latents par point de fonction, avec une norme d'élimination des défauts de 85 % (Capers Jones, 1998, p. 25).

Les conséquences de ces défauts comprennent les coûts de réparation de ce système, mais aussi les coûts liés à l'arrêt du système qui ne rend pas les services attendus. Ces coûts peuvent être évidemment très variables selon le caractère plus ou moins critique du logiciel[35], depuis la simple impossibilité d'effectuer des opérations prévues mais non cruciales, jusqu'à des effets parfois dramatiques[36], en passant par des opérations absurdes, des pertes de données ou des dégâts matériels parfois très importants (par exemple dans le cas d'Ariane V outre la destruction de la fusée, la perte de quatre satellites scientifiques de très grandes valeurs).

---

[34] Un logiciel de 500 KLS correspond à 10 volumes de 1000 pages aux normes de l'édition format Pléiade, ce qui fait en moyenne un peu plus de cinq erreurs par volume.

[35] On peut distinguer les logiciels critiques "de mission" dont la défaillance est désastreuse sur un plan financier, des logiciels critiques de "sécurité" dont la défaillance peut provoquer des victimes humaines. Ce caractère critique est évidemment relatif : une défaillance du logiciel de gestion d'une entreprise n'est pas critique pour la collectivité, mais peut l'être pour l'entreprise considérée. De même certaines utilisations d'un logiciel peuvent lui conférer un caractère critique inattendu : par exemple, l'emploi (effectivement réalisé) d'un banal tableur par un chirurgien pour traiter des données concernant un patient sur lequel il était en train d'effectuer une opération à cœur ouvert (Lauren Ruth Wiener, 1994, p. 100).

[36] En raison d'une séquence d'utilisation non prévue au cours de son développement, une machine à rayons X pilotée par ordinateurs il y environ une dizaine d'années a été la cause de plusieurs accidents mortels dans des hôpitaux (Maarten Boasson, 1998, p. 2).

Dans le cas des progiciels, l'absence de toute garantie de bon fonctionnement de la part des éditeurs de progiciels est significative[37]. Cette absence totale de garantie n'existe pour aucun autre produit commercialisé. Certes en Europe de telles clauses sont illégales (cf. chapitre III), du moins pour la vente à des particuliers ou à des professionnels non informaticiens, mais il demeure des problèmes d'interprétation notamment autour de la notion de vices cachés, qui ne s'applique que quand il y a vente d'un produit, alors qu'un progiciel n'est jamais vendu : seul est concédé à l'acquéreur un droit d'utilisation.

2 - L'imperfection de procédures qui sont exécutées automatiquement

Comment expliquer cette présence persistante de défauts dans les logiciels ? Pourquoi sont-ils si difficiles à éliminer ? Pourquoi des défauts, en apparence anodins, peuvent avoir des conséquences si dommageables ? La réponse à ces questions nécessite de repartir du rôle d'un logiciel qui pilote un traitement automatique d'informations, et de la contradiction, intrinsèque au logiciel, entre les possibilités de création diversifiée illimitée d'un texte (caractéristique de l'activité d'écriture) et les contraintes résultant du fait que ce texte est un texte actif, un ensemble d'instructions qui seront exécutées automatiquement par des machines (cf. chapitre I). "Un logiciel est une structure composée de symboles logiques arrangés en fonction du modèle que quelqu'un se fait d'un aspect du réel" mais "pour servir à quelque chose, le logiciel doit être intégré à un système physique [et humain] qui affecte une partie du monde" (Lauren Ruth Wiener, 1994, p. 63)[38]. C'est ce qui explique qu'un système logiciel doit avoir "un niveau de fiabilité très supérieur à celui du système humain, ou de la pratique humaine, que l'informatique remplace ou complète, faute de quoi le système informatique est incompatible avec le rôle social qui lui est dévolu" (Jacques Printz, 1998, p. 18).

---

[37] Par exemple, pour le dernier langage de programmation Delphi proposé par l'éditeur Borland, la "garantie" stipule : "La responsabilité maximum encourue par Borland au titre de la présente garantie est limitée au remplacement du support défectueux et aucune responsabilité n'est encourue pour tout autre dommage sur quelque fondement que ce soit et, notamment, tout dommage indirect et accessoire, même si Borland a été expressément informé de l'éventualité de tels dommages." ; et plus loin en majuscules "AUCUN AUTRE ENGAGEMENT OU GARANTIE N'EST CONSENTI PAR BORLAND ET NOTAMMENT AUCUNE GARANTIE EXPRESSE OU TACITE DE CONFORMITE OU D'ADEQUATION A UN USAGE SPECIFIQUE".

[38] Ceci peut permettre de comprendre l'affirmation assez surprenante de Lauren Ruth Wiener : "les logiciels *sont bogués parce qu'ils sont construits selon des règles logiques*. Ce sont des structures abstraites, complexes et étrangères à notre vécu. Les logiciels sont bogués parce que nous nous en servons pour effectuer une tâche. Ils font donc partie d'un système physique, lui-même vulnérable aux aléas de l'environnement" (1994, p.63).

Dans un système informatique, les composants matériels[39] ont atteint une grande fiabilité, permise par l'omniprésence de codes correcteurs d'erreurs (produit des avancées de la théorie du codage), qui permettent d'éliminer les aléas nombreux auxquels sont soumis les circuits[40]. Dans le cas d'équipements critiques, il existe des systèmes à tolérance de panne basés sur la présence de composants redondants, qui permettent de compenser automatiquement la défaillance d'un composant. Cette stratégie est efficace pour des composants matériels qui s'usent selon des processus physiques et que l'on peut tester de manière complète[41]. En tant que machine traitant des impulsions électroniques, "la fiabilité [d'un ordinateur] est immense et il est correct de dire que l'ordinateur ne fait quasiment jamais d'erreur" (Philippe Breton, 1990, p. 58).

Mais bien évidemment ces impulsions électroniques, qui représentent des données et des instructions, n'ont aucune signification pour l'ordinateur[42]. Celui-ci exécute mécaniquement les actions prévues par le programme sans en connaître la finalité ou le sens. L'ordinateur ne fait que calculer en suivant machinalement les règles qui lui ont été transmises, toute interprétation lui est impossible, aucune action n'est "aberrante", aucune connaissance n'est implicite. "L'ordinateur est le domaine de la pure raison où tout doit être explicable rationnellement" (Jacques Printz, 1998, p. 317). "L'ordinateur, et par voie de conséquence le raisonnement informatique, ne supporte pas, constitutionnellement, l'ambivalence, l'ambiguïté, c'est à dire la possibilité d'incarner simultanément deux choix

---

[39] Les architectures des éléments matériels sont plus simples et comprennent notamment de nombreuses structures répétitives.

[40] Déjà en 1967, 90 % des incidents informatiques étaient dus à des défauts de programmation (Lauren Ruth Wiener, 1994, p. 98).

[41] Cette stratégie est par contre peu efficace dans le cas des logiciels. Faire tourner en parallèle plusieurs exemplaires du même programme n'aurait strictement aucun intérêt (pour ce qui concerne les défauts des logiciels), les programmes étant rigoureusement identiques, défauts compris. Faire réaliser plusieurs logiciels par des équipes différentes de programmeurs pour répondre au même problème, et les exécuter simultanément peut sembler judicieux (c'est le cas pour la navette spatiale où le logiciel d'un des cinq ordinateurs, qui effectuent les mêmes traitements, a été réalisé par une autre société). Malheureusement des études ont mis en évidence que si les solutions apportées par des programmeurs différents peuvent être très dissemblables, elles ont tendance à provoquer les mêmes défaillances correspondant à des erreurs identiques concernant les mêmes points délicats (Lauren Ruth Wiener, 1994, p. 166).

[42] C'est ce qui explique la force et la faiblesse de la vérification automatique d'orthographe comparée à une vérification humaine. Lors d'une vérification automatique, toutes les erreurs dans des mots compliqués dont l'orthographe n'est pas évidente sont toujours corrigées si le mot est présent dans le dictionnaire intégré ; par contre, une erreur de frappe, qui retire toute signification à une phrase, ne sera pas détectée si elle a eu pour conséquence un mot présent dans le dictionnaire.

différents à propos d'une même situation" (Philippe Breton, 1990, p. 82). Tout doit pouvoir être tranché par oui ou par non, toute situation qui peut survenir dans le déroulement du programme doit avoir été explicitement prévue. "Par définition l'ordinateur ne peut réagir qu'aux situations pour lesquelles il a été programmé. Dans le cas le plus favorable, face à une situation imprévue (…), il devra s'arrêter en avertissant l'opérateur humain qui le pilote (…). Dans le cas défavorable, il appliquera une procédure prévue pour une autre situation et n'importe quoi pourra se produire" (Jacques Printz, 1998, p. 35). Il est donc nécessaire *qu'ex-ante* (lors de l'élaboration du logiciel) aient été spécifiées complètement toutes les entrées d'informations possibles (données et instructions) qui pourront se produire *ex-post* (lors de l'utilisation du logiciel), et qu'un traitement adéquat ait été élaboré, pour que les sorties d'information résultant de l'exécution du programme soient celles escomptées par les utilisateurs. Toute la difficulté est de réaliser un "pavage" du domaine d'emploi du logiciel, qui ne laisse pas de "vide" (problème *d'incomplétude* du logiciel, c'est à dire de situations non prévues qui peuvent survenir) et qui ne comporte pas de "chevauchement" (problème *d'ambiguïté ou d'indéterminisme* avec l'existence de deux traitements différents possibles face à une même situation).

Un système informatique n'est pas doté d'un comportement intelligent

Il importe de bien saisir le fossé existant entre le comportement d'un ordinateur, même doté des logiciels les plus perfectionnés et le comportement humain. L'insistance sur les différences de comportement entre l'homme et l'ordinateur est motivée par la place particulière qu'occupe le logiciel à la *frontière* entre le monde humain et le monde machinique. En effet, il est le produit d'une création humaine mais est exécuté automatiquement par une machine. Dans beaucoup de situations, il correspond à une activité antérieurement effectuée par l'homme et qui a été modélisée pour être réalisée automatiquement. Or le fait que *rétrospectivement*, on puisse décrire l'action humaine comme étant réalisée conformément à des règles et en suivant des plans bien établis, peut donner l'illusion que tel est le cas dans la réalité, alors que l'action humaine se caractérise par la manière dont elle traite en souplesse chaque situation nouvelle et imprévue (Harry M. Collins, 1992, p. 66).

Tout d'abord, la puissance du cerveau humain est sans commune mesure avec celle d'un ordinateur ; en se basant sur les estimations (imprécises) de la neurobiologie sur le nombre de neurones et sur le nombre de connexions qui les relient, Jacques Printz calcule que pour

simplement représenter ce réseau il faudrait un rayonnage de disques de 10 gigaoctets (de 2 mètres de hauteur) d'une longueur de 3125 km (1998, p. 41-42). Ensuite l'homme a une vision spatio-temporelle globale de son environnement grâce aux associations autorisées par ces cinq sens, alors que les possibilités de traitement de l'ordinateur ne le rendent apte qu'à une vision parcellaire et linéaire de la réalité[43]. Par exemple, la reconnaissance d'une photographie parmi 10 000 photographies (ce qui correspond à la capacité estimée de mémorisation du cerveau) prend moins d'une seconde à un être humain, alors que l'exécution d'un programme pour effectuer la même opération par un ordinateur dure plus de dix minutes (idem, p. 20-21). Le principe de fonctionnement du cerveau humain n'a rien de comparable avec celui d'un ordinateur muni de sa programmation. Le cerveau n'est jamais enfermé dans un système de signes. Nous pouvons créer librement les signes dont nous avons besoin, sous la seule contrainte de pouvoir communiquer avec nos semblables. *A la différence des langages informatiques qui sont des systèmes fermés (ou complets), le langage humain fonctionne comme un système ouvert sur le monde extérieur, avec la possibilité de créer de nouveaux concepts*. Nous pouvons effectuer des raisonnements inductifs, ce dont les ordinateurs sont absolument incapables, malgré les discours sur l'existence de prétendues machines à induction dans le cas de certains systèmes experts (Harry M. Collins, 1992, p. 178). Le cerveau entretient en permanence une image spatio-temporelle de la réalité qui nous permet de garder le souvenir du passé (en enregistrant la succession des expériences réussies et des échecs qui jalonnent notre existence), de nous projeter dans le futur, de faire des projets, d'assigner des objectifs à atteindre et de définir des critères de jugement. Notre capacité d'auto-référence (l'image de la réalité que nous construisons fait partie de la réalité) est une source de sens nouveau et de création, qui nous permet en permanence d'ajuster notre comportement à la situation nouvelle qui se crée à chaque instant, par le jeu des essais/erreurs/corrections et en intégrant de manière pro-active de nouvelles informations. Notre univers mental est ouvert et réinterprétable en permanence ce qui explique le non-déterminisme foncier des situations et de l'histoire humaine, à l'opposé du comportement d'un ordinateur qui est et doit rester fondamentalement déterministe (Jacques Printz, 1998, p. 45-46 et 321).

[43] Jacques Printz précise pourquoi notre perception immédiate de l'espace rend la programmation si difficile et si contre-intuitive. En effet, les résultats de la programmation sont des textes qui ont pour caractéristique principale d'être unidimensionnels, l'ordinateur ne pouvant faire qu'une lecture séquentielle et linéaire de l'information et des ordres qui lui sont communiqués. Il est donc nécessaire d'opérer "une "déconstruction" de la perception globale que nous pouvons avoir du logiciel pour la faire rentrer dans la logique unidimensionnelle de l'ordinateur" (p. 74-75).

Un système informatique n'est pas doté de capacités d'interprétation autonome

Certes il a existé de nombreuses tentatives pour rapprocher dans des situations de travail, le comportement humain du comportement d'un automate : Organisation Scientifique du Travail et taylorisation des tâches, édiction bureaucratique de règles et de procédures devant couvrir tous les cas possibles, mise en place de routines qui "consistent à spécifier complètement *ex-ante* l'ensemble des comportements qui seront requis *ex-post*" (Eric Brousseau, 1997, p. 45). Ces tentatives ont en commun de reposer sur des processus de codification et de formalisation accentuées des différents savoirs humains. Mais ces procédés ont également montré leurs limites : rôle décisif (même si fréquemment sous-estimé) des connaissances tacites non codifiables, notamment en présence d'interactions multiples, importance de l'indéterminisme et de l'ambiguïté comme ressources de l'organisation (Geoffrey Bowker, Susan Leigh Star, 1997, p. 300), "mérites méconnus du flou, et ceux mieux connus du tacite ou de l'implicite" (Olivier Favereau, 1998, p.198), constat que "la simple application d'une règle n'est jamais une application simple" (idem, p. 224).

Contrairement à ce que suppose certaines représentations la rationalité humaine ne se limite pas à un calcul pour trouver la solution optimale, qui dépasse fréquemment les capacités humaines (cf. chapitre VII). L'esprit humain, qui à la différence de l'ordinateur "a plutôt des difficultés à mettre en place un pur raisonnement séquentiel" (Philippe Breton, 1990, p. 82), a par contre des possibilités singulières, plus riches et plus complexes[44] : les hommes connaissent les finalités des actions qu'ils entreprennent, ils sont impliqués dans l'issue des opérations, ce qui leur permet d'autoréguler leurs comportements (Toran Demarco, Timothy Lister, 1991, p. 123).

En particulier, *ils ont la possibilité d'interpréter les règles* ; or, il existe une "part inéliminable, et parfois capitale, d'interprétation pour exécuter une règle, sans même parler de choisir la règle à exécuter", dans la mesure où "les règles sont toujours incomplètes, en ce qu'elles ne fixent pas exhaustivement leurs conditions d'application" (Olivier Favereau, 1998, p.218). De même, Harry M. Collins souligne que les règles ne peuvent pas contenir les règles de leur propre application, et "qu'il est impossible de prévoir toutes les circonstances d'application d'une règle, (…) cette application [étant] soumise, dans chaque cas particulier,

[44] A la différence des ordinateurs, "les êtres humains sont très doués pour reconnaître des modèles, pour comprendre la langue naturelle, pour prendre en considération les circonstances extérieures au système, pour faire preuve d'initiative" (Lauren Ruth Wiener, 1994, p. 242).

aux précédents que les hommes sont en droit et en mesure de créer" (1992, p. 130)[45]. Pour avoir une réponse appropriée dans des circonstances imprévues, il est indispensable qu'existe une compréhension des règles et une "sensibilité socialisée au contexte grâce à laquelle nous savons quand et avec quelle fréquence il est possible de transgresser les règles" (idem, p. 273)[46]. Olivier Favereau énonce trois caractérisations de ce que signifie "*interpréter*" (en tenant compte de l'existence de capacités d'apprentissage) et qu'il oppose à "*calculer*". Ces caractérisations nous semblent particulièrement fécondes pour comprendre la différence entre le comportement d'une machine programmée par un logiciel, qui reste "prisonnière" d'un modèle fermé constitutif (indépendamment de l'éventuelle complexité de la modélisation réalisée), et le comportement humain. Par sa capacité à interpréter, l'homme peut "élaborer des modèles décidant simultanément de ce qui est général (traitable par le modèle) et de ce qui est particulier (extérieur au modèle)" (idem, p. 199). Il peut "rentrer à l'intérieur d'un système de règles (au sens le plus large) pour lui prêter sens et pertinence, tout en restant à l'extérieur de celui-ci, afin de l'appréhender dans un jugement critique" (idem, p. 205-206). Enfin, nous pouvons "nous mettre à la place d'autrui, non pour anticiper, de l'extérieur ses réactions que pour nous donner toutes les chances de communiquer nos intentions, sans équivoque, et ainsi comprendre de l'intérieur, le sens profond de ses réactions" (idem, p. 220).

Un système informatique ne peut réagir correctement à l'imprévu

Une autre différence importante est la réaction à l'imprévu. L'ordinateur ne pouvant réagir qu'à ce pourquoi il a été programmé, l'imprévisible ne peut lui être que fatal (Jacques Printz, 1998, p. 102), alors que l'homme dispose de capacités d'adaptation à l'imprévu, en ayant notamment la possibilité de réagir en "temps réel" au vu des conséquences des décisions qu'il prend, et des actions qu'il entreprend. C'est ce qui explique les conséquences différentes d'une défaillance psychocognitive et d'une défaillance d'un programme informatique. "Notre

[45] Harry M. Collins précise qu'il est toujours nécessaire d'établir de nouvelles interprétations, qui ne peuvent pas s'inspirer de l'expérience passée. Il montre que même "dans le cas de sports bien établis, avec des règles organiques hautement codifiées, appliquées à un très haut niveau avec l'organisation rigoureuse convenant aux compétitions de niveau mondial, les règles [sont] incapables par elles-mêmes de faire face à des circonstances imprévues" (p. 130), ce qu'il illustre par le cas, lors de la série de tirs au but pour départager le Brésil et la France en quarts de finale de la Coupe du Monde de football de 1986, d'un tir qui rebondit sur un poteau puis sur le gardien avant de rentrer dans le but (p. 118).

[46] Harry M. Collins cite le contre-exemple d'un conducteur d'autobus suspendu parce qu'il avait suivi trop strictement une règle (ne pas s'écarter de son itinéraire sans autorisation), alors qu'une personne avait une crise cardiaque dans le bus qu'il conduisait. Il considère qu'il s'agit d'un cas manifeste de "socialisation déficiente en matière de priorités morales" (1992, p. 115-116).

système psychocognitif est une fantastique "machine" tolérante à un nombre considérable de pannes" (idem). En particulier, en cas d'erreur, nous tentons de restaurer l'environnement initial, ce qui est impossible pour l'ordinateur face à une situation imprévue, et qui peut troubler profondément l'usager, compte tenu de son modèle psychocognitif (idem, p. 127).

Enfin, l'ordinateur est évidemment dépourvu de deux qualités importantes de l'être humain, le bon sens et l'intuition. "Le bon sens (…) est constitué du stock de situations dont nous disposons à l'instant t pour juger du bien-fondé de ce que nous sommes en train d'effectuer à 'instant $t+\Delta t$. Le bon sens fonctionne comme une procédure d'exception, un démon de Maxwell, qui observe en permanence nos actions et qui est en mesure d'en déclencher l'arrêt si celles-ci, eu égard au passé, sont en mesure de nuire à notre intégrité présente et future. Sans le bon sens nous aurions probablement disparu de la planète depuis bien longtemps" (Jacques Printz, 1998, p. 34). Quant à l'intuition c'est une " qualité spontanée dont on ne contrôle ni la présence, ni l'absence, mais qui semble jouer un rôle permanent – parfois diffus, parfois aigu – dans toute activité humaine" et "aucun raisonnement produit par le cerveau humain ne peut d'ailleurs éviter de mettre en œuvre l'intuition" (Philippe Breton, 1990, p. 61)[47].

L'utilisation d'un système informatique requiert fréquemment des "dialogues" entre les utilisateurs et l'ordinateur, qui sont pilotés par les logiciels. Aux incertitudes sur le comportement des utilisateurs s'ajoutent les incertitudes sur l'environnement sur lequel l'ordinateur capte des informations et sur lequel éventuellement il agit, d'autant plus que ces environnements se modifient par rapport au moment où a été conçu le logiciel. Le problème principal est celui de la prédictibilité : tous les possibles doivent être prédéterminés, précodés par le programmeur, alors qu'il "reste presque toujours une marge de virtualités indéterminées, un espace libre pour la création de possibles que nul n'avait prévus" (Pierre Lévy, 1992, p. 27)[48]. Comme toute interprétation des règles est impossible, les règles doivent être plus complètes, plus cohérentes, plus détaillées, pour éviter que leur application aveugle ne conduise à des absurdités : relance de clients pour zéro franc, poursuite d'un contribuable

[47] Philippe Breton ajoute que même dans le cas des informaticiens qui affirment n'utiliser qu'une démarche logique, l'observation de leurs comportements montrent que, en réalité, les informaticiens recourent fréquemment à une démarche intuitive (1990, p. 67).

[48] Comme le note Harry M. Collins, si "on peut avoir une science sociale causale tournée vers le passé, on ne peut faire une science sociale prédictive et déterministe" (1992, p. 174).

pour trois centimes (Jean-louis Peaucelle, 1997, p. 27). Il faut en particulier se méfier des situations qui n'apparaissent pas ambiguës à un être humain, parce qu'elles renvoient pour lui à des connaissances élémentaires[49], quasi naturelles (comme le fait que l'année qui suit l'année 1999 n'est pas l'année 1900 !), mais qui le seront pour l'ordinateur, si ces informations élémentaires n'ont pas été explicitement introduites. De plus, le logiciel de par sa nature immatérielle apparaissant facilement modifiable, la tentation peut être forte de modifier plus fréquemment les règles, ou de les complexifier pour traiter des situations particulières (qui, traitées "manuellement" n'auraient pas nécessité l'édiction de nouvelles règles). Cependant, la croyance, qu'en rajoutant règle après règle à des programmes pour répondre aux exceptions et aux cas particuliers découverts on finira par reproduire le comportement humain et qu'il ne sera plus nécessaire d'effectuer des modifications ultérieures, est vaine : "réussir à intégrer quelque chose rétrospectivement dans une règle ne veut pas dire que la règle pourra rendre compte d'exemples futurs imprévus" (Harry M. Collins, 1992, p. 67)[50].

Les conséquences sur les difficultés à développer des programmes "corrects"

Une des principales difficultés dans l'écriture d'un programme correct, consiste à réaliser un traitement adéquat pour tous les cas qui pourront se présenter, pour toutes les combinaisons de toutes les entrées d'informations possibles. Il faut, en général, plus d'effort pour traiter et vérifier la validité du traitement des quelques cas qui se présentent exceptionnellement que pour la grande majorité des situations habituelles. Cela nécessite de la part de l'informaticien de "pousser l'utilisateur dans ses derniers retranchements pour obtenir de lui toutes les réactions à prévoir face à toutes les situations envisageables", alors que l'utilisateur a tendance, "fort pragmatiquement, à laisser dans l'oubli ces cas de figures irréalistes voire invraisemblables, relevant à l'évidence d'un traitement pour période de crise, sur lequel il serait toujours temps de se pencher le cas échéant" (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 11). De plus la pleine collaboration des utilisateurs n'est pas nécessairement acquise : méfiance face à une "obsession inquisitoriale, révélatrice de cette

[49] "Un problème, bien connu des anthropologues et des chercheurs en sciences humaines qui ont un cadre de pensée interprétatif, est l'invisibilité de ce qui nous est trop familier" (Harry M. Collins, 1992, p. 202-203).

[50] Ce même auteur souligne que "l'incapacité à faire des progrès est souvent imputée, non pas à la nature fondamentale du savoir, mais à des défauts que l'on n'a pas vus dans l'établissement des règles. Les programmeurs et les cogniticiens tentent de faire en sorte que les règles reflètent la réalité de plus en plus précisément, mais la précision ne désigne pas la qualité la plus essentielle d'une bonne performance. C'est une erreur que d'essayer de contrefaire le savoir-faire sur la base d'un surcroît de précision" (Harry M. Collins, 1992, p. 282-283).

volonté de pouvoir qui, selon [l'utilisateur], anime les informaticiens" (idem, p. 12), crainte d'une critique des procédures suivies dans la réalité et parfois éloignées des normes théoriques en vigueur, inquiétude par rapport à une dévalorisation éventuelle de ses compétences résultant de leur transfert dans un logiciel[51].

Ce problème concerne également l'introduction des données par les utilisateurs. Il faut distinguer l'intégrité formelle des données (assurée par un contrôle automatique et systématique des données entrées par rapport à des fourchettes de valeurs autorisées[52]) et l'intégrité réelle (la validité du contenu même de l'information), qui ne peut être garantie que par l'utilisateur. Or, la saisie des données n'est jamais purement technique mais toujours de nature sociotechnique (Philip E. Agre, 1997, p. 258), la détention d'informations étant un des fondements du pouvoir des acteurs. En particulier, la pertinence des données entrées peut être problématique, lorsque les personnes qui les saisissent ne sont pas celles qui les utiliseront après traitement (Geoffrey Bowker, Susan Leigh Star, 1997, p. 292). Peuvent s'enclencher des cercles vicieux de données erronées entrées, de sorties d'informations résultantes non fiables, et de désintérêt pour saisir des données correctes.

En conséquence, "les systèmes informatiques, dans la mesure où il cohabite avec des humains, ou fonctionnent dans des environnements hostiles, doivent être dotés de mécanismes compensateurs et/ou réparateurs permettant de répondre aux fluctuations de l'environnement externe et aux aléas internes, tous imprévisibles par définition" (Jacques Printz, 1998, p. 18)[53].

Cette analyse peut permettre de comprendre *pourquoi les logiciels comprennent des erreurs et pourquoi celles-ci sont si difficiles à éliminer* : elles peuvent se manifester

---

[51] Nous reviendrons plus en détail dans la section suivante, consacrée aux difficultés d'adaptation du logiciel aux besoins, sur l'importante question de la collaboration, pas nécessairement spontanée des utilisateurs.

[52] Il faut se montrer très prudent dans la restriction des données acceptées en entrée d'un programme, surtout quand la validité des données ne dépend pas de conventions humaines (une note ou un prix) mais de l'observation de phénomènes physiques. Par exemple, dans les années soixante-dix et quatre-vingt, le logiciel qui traitait les relevés des taux d'ozone recueillis par les satellites de la Nasa, les rejetait en les considérant comme des mesures erronées parce qu'ils étaient trop bas, avant qu'une équipe de scientifiques britanniques en 1986 ne signale la chute du niveau d'ozone (Lauren Ruth Wiener, 1994, p. 36).

[53] Dans le cadre d'une analyse plus générale des systèmes artificiels, Herbert Simon souligne que "nous aurons souvent à nous satisfaire d'approximation pour accorder la conception aux objectifs. Dans de tels cas, les propriétés du système interne "transparaîtront" à travers sa frontière - autrement dit, le comportement du système répondra seulement de façon partielle aux sollicitations de l'environnement, et il s'ajustera partiellement aussi aux limitations du système interne" (1974, p. 28).

seulement après des années d'utilisation des logiciels[54], il est souvent difficile d'isoler au sein de systèmes de plus en plus interconnectés le logiciel défectueux[55] surtout lorsque les défaillances se produisent de façon fugace et intermittente et qu'il est difficile de reproduire le contexte d'apparition de l'erreur[56].

3 - La détection et la correction des erreurs

De nombreuses étapes dont l'objectif est d'éliminer les défauts jalonnent les processus de développement des logiciels. Capers Jones distingue jusqu'à treize niveaux d'élimination des défauts : revue de spécifications, revue de conception générale, revues de conception détaillée, revues d'assurance-qualité, validations et vérifications faites par des indépendants, preuves de correction, tests sur table, tests unitaires, tests fonctionnels, tests de composants, tests du système, tests sur le terrain, tests de réception par le client (Capers Jones, 1989, p. 207-208)[57].

Les différentes opérations d'élimination des défauts ont un rendement très inégal, mais qui est difficile à évaluer, dans la mesure où on ne connaît les défauts non trouvés qu'a posteriori. En plus des efforts pour trouver les défauts, il faut prendre en compte le coût des réparations. Ce coût peut être très élevé quand la vérification du système logiciel à l'aide de tests met en évidence des erreurs de spécification et de conception. Or les principales sources d'erreurs des logiciels, surtout pour les logiciels de taille importante, sont beaucoup plus fréquemment des problèmes de spécification et de conception, que des problèmes de codage. C'est pourquoi dans les années soixante-dix et quatre-vingt se sont développées des

---

[54] Il a été constaté pour certains logiciels que le nombre d'erreurs découvertes, qui décroît logiquement durant la phase d'utilisation de ce logiciel, pouvait se remettre à croître en conséquence d'une plus grande expertise des utilisateurs qui l'exploitaient pour des opérations plus complexes dans des environnements de plus en plus diversifiés.

[55] Les défaillances fréquentes lors d'une connexion sur un serveur distant peuvent provenir d'une erreur dans un des nombreux logiciels concernés (systèmes d'exploitation de l'ordinateur client et du serveur, logiciel de navigation sur le réseau, système de gestion des données sur le serveur, logiciels assurant la transmission des données…) sans qu'il soit aisé de déterminer lequel.

[56] Une enquête sur un système d'exploitation très répandu a démontré que 30 % des bogues signalées présentaient un risque d'apparition d'une fois tous les cinq mille ans de fonctionnement du système (Lauren Ruth Wiener, 1994, p. 129).

[57] En intégrant la documentation et les autres fournitures qui accompagnent un logiciel (matériel d'accompagnement et de formation), Capers Jones liste plus de quarante variétés de méthodes d'élimination des défauts qui peuvent être utilisées (Capers Jones, 1989, p. 212-223).

techniques de validation d'un logiciel qui "s'intéressent à la spécification externe du logiciel que l'on considère alors comme une boîte noire" (Jacques Printz, 1998, p. 351). Ces méthodes jugées extrêmement rentables sont des techniques d'inspection et de revue des documents issus du cycle de développement.

Mais même en appliquant ces méthodes, la phase la plus importante reste la phase des tests, qui peut représenter jusqu'à la moitié du temps total de développement d'un logiciel (Frederick P. Brooks, 1996, p. 16)[58]. Paradoxalement, "le passé de l'informatique n'a pas mis en avant le caractère essentiel de cette tâche" (Serge Bouchy, 1994, p. 177), pour laquelle il existe peu d'enseignements théoriques. Cette activité longue, fastidieuse et coûteuse reste "la bête noire" de l'informatique (idem), voire la "honte de la communauté informatique, tant au niveau industriel, qu'au niveau universitaire" (Jacques Printz, 1998, p. 249)[59].

Le test est ce qui valide *in fine* la bonne correspondance entre le modèle informatique que l'ordinateur exécute et la réalité. Il consiste à effectuer une exécution sur machine de scénarios d'essais, conçus comme des protocoles expérimentaux parfaitement reproductibles, ce qui exige un certain déterminisme du système observé, l'élimination progressive des erreurs pouvant être assimilée à la falsification expérimentale d'une théorie selon Popper (Jacques Printz, 1998, p. 104).

On distingue les tests "boîte blanche" où on vérifie pas à pas les différents "chemins" possibles dans le programme, des tests "boîte noire" où on ignore volontairement le détail du programme pour ne s'intéresser qu'à la correspondance entre état initial et état final. Cette dernière stratégie peut être extrêmement coûteuse le nombre de correspondances possibles connaissant rapidement une "combinatoire explosive" et le coût des tests augmentant probablement de façon exponentielle (Jacques Printz, 1998, p. 258).

C'est une des raisons qui a justifié de recourir à une conception modulaire des logiciels, avec la constitution de modules les plus indépendants possibles qui pourront être testés

[58] Frederick P. Brooks précise qu'en général cette importance n'est pas celle qui a été prévue initialement, mais celle que l'on constate *a posteriori*. Par exemple, les coûts des tests pour la navette spatiale américaine ont constitué plus de 60 % du coût total des logiciels (Jacques Printz, 1998, p. 110).

[59] Les réponses à une enquête de 1985 indiquait que moins de 14 % des entreprises faisaient état de mesures de la qualité du logiciel ou des éliminations de défauts (Capers Jones, 1989, p. 40).

séparément (tests unitaires). Il faut éviter le plus possible le "couplage" entre modules[60], car dans ce cas la fiabilité de la somme fonctionnelle de deux modules est égale au produit des fiabilités (Jacques Printz, 1998, p. 287)[61]. Néanmoins il faudra ensuite réaliser des tests d'intégration avec les autres composants du système, dans toutes les combinaisons possibles, dont le nombre croit très vite. Ce test prend énormément de temps, car "des bogues subtils peuvent résulter d'interactions inattendues entre des composants déjà débogués" (Frederick P. Brooks, 1996, p. 5)[62]. "Face à de telles combinatoires, seuls la puissance du cerveau et le savoir-faire humain sont à même de donner des résultats sémantiquement pertinents. Les tests sont donc une activité à haute valeur sémantique ajoutée *où, très rapidement, le cerveau humain est irremplaçable*" (Jacques Printz, 1998, p. 259)[63].

Aux difficultés pour détecter les erreurs, s'ajoutent les problèmes occasionnés par la réparation des erreurs constatées. En effet, "en corrigeant un bogue on a une chance substantielle (20 à 50 %) d'en introduire un autre" (Frederick P. Brooks, 1996, p. 103), une des explications étant que fréquemment ceux qui corrigent les logiciels sont moins compétents que ceux qui ont écrit le code initial. De plus, "toute correction tend à détruire la structure du système, à augmenter son entropie et son désordre" (idem), ce qui est d'autant plus vrai que le logiciel a été conçu longtemps auparavant, par des personnes qui ne sont plus nécessairement dans l'entreprise et que sa documentation n'est pas forcément de qualité. Dans ces situations, les personnes chargées de la maintenance résolvent le problème par des "colmatages, c'est-à-dire des corrections locales inadéquates et des rafistolages de fortune" (Lauren Ruth Wiener, 1994, p. 138). C'est notamment ce qui explique que les derniers défauts découverts sont toujours les plus coûteux à éliminer (Capers Jones, 1989, p. 25). Dans la mesure où les effets de la correction d'une erreur peuvent être lointains et indirects, il est donc

[60] Des modules sont faiblement couplés quand les échanges entre modules sont peu nombreux.

[61] Quand la fiabilité globale est le produit des fiabilités élémentaires, elle tend inéluctablement vers zéro dès que le nombre d'éléments devient grand (Jacques Printz, 1998, p. 132).

[62] C'est, par exemple, une des difficultés pour corriger le bogue de l'an 2000 : l'existence de deux logiciels, qui testés séparément passent le cap de l'an 2000, ne garantit pas que leur utilisation simultanée ne posera pas de problème.

[63] Philippe Breton constate qu'il est souvent nécessaire de recourir à l'intuition pour retrouver une erreur dans un logiciel (1990, p. 61).

nécessaire de refaire de nombreux tests (tests de régression)[64]. De ce fait, le coût du passage de n tests croit en n^2, chaque nouveau test accompagné d'une correction nécessitant une vérification de conformité des tests déjà acquis, soit au total n(n+1) / 2 passages (Jacques Printz, 1998, p. 259).

En conclusion, il faut prendre conscience qu' "un test n'est pas destiné à montrer qu'un programme fonctionne correctement, mais à trouver des erreurs" (Jean-Marc Geib, 1989, p. 99). "Certaines études ont permis d'estimer que le nombre d'erreurs de programmation présentes au début des tests est couramment de l'ordre de 2 pour 10 lignes de code. Pour un programme de 100 000 lignes, c'est donc 20 000 erreurs, de gravité heureusement très variable, qui devront être décelées et corrigées" (Gérard Dréan, 1996, p.216). En général, on estime qu'après les tests, seulement 85 % des défauts existants ont été éliminés.

B - DES ERREURS DE MOINS EN MOINS ACCEPTEES…

La présence de défauts dans les logiciels est difficile à accepter car elle résulte d'erreurs purement humaines (1), alors même que leurs conséquences sont de plus en plus sensibles avec l'augmentation de l'importance des logiciels (2).

1 - Des erreurs humaines difficiles à accepter…

Un logiciel est un artefact. Tout objet artificiel créé par l'homme peut bien évidemment comporter des défauts. Mais la singularité des logiciels est que ce sont des artefacts complètement immatériels, qu'ils sont constitués exclusivement des produits de l'activité humaine seule[65]. De ce fait, l'origine des défauts est toujours une erreur humaine, cette erreur ayant pu se produire en phase de conception/programmation, en phase

[64] Un logiciel de routage d'appels comportant plusieurs millions de lignes de codes et dont les tests avaient duré treize semaines, subit une très légère modification (elle ne concernait que trois lignes). Les programmeurs, jugeant qu'ils connaissaient parfaitement les conséquences de cette modification, estimèrent qu'il était inutile de refaire les tests. Cette modification minime comprenait une erreur (un "6" à la place d'un "D"), qui entraîna une série de coupures pour les abonnés au téléphone de plusieurs grandes villes américaines fin juin et début juillet 1991 (Lauren Ruth Wiener, 1994, p. 27 et 67).

[65] "Les bugs sont une panne d'un genre inédit dans l'histoire des techniques. Ils ne sont pas le fruit de conditions extérieures, comme l'oxydation, qui grippe un mécanisme. Ils ne sont pas non plus le résultat d'une pièce qui se brise. C'est une sorte d'oubli de la part des concepteurs de logiciels, incapables d'envisager tous les cas de figure de l'utilisation de leurs produits. Le bug est le résultat d'une limite des pouvoirs de l'homme, de sa capacité d'abstraction et de sa faculté d'anticiper des scénarios d'utilisation et des combinaisons de technologies sans cesse plus complexes" (Thibault Honnet, interview à Libération-Multimédia, 31-12-1999).

d'installation/maintenance ou en phase d'exploitation. "Il est donc impossible d'attribuer à une quelconque fatalité ce qui est totalement imputable aux concepteurs et aux programmeurs du système" (Jacques Printz, 1998, p. 57). Par contre, dans le cas d'un artefact matériel, un défaut peut résulter d'une "imperfection de la nature", avec la présence de matériaux imparfaits dans l'artefact. C'est ce qui explique qu'un manque de fiabilité au niveau du matériel soit mieux accepté qu'au niveau du logiciel, alors que la grande majorité des problèmes informatiques sont des problèmes logiciels, et que les rares défaillances des composants matériels sont actuellement le plus souvent la conséquence d'erreurs dans les logiciels (microcode) qu'ils intègrent[66].

Il semble que la société tolère moins bien les imperfections humaines que les imperfections naturelles. "Les limites de la science" et les "limitations d'ordre social" sont des évidences récentes et mal intégrées dans le corps social (Jacques Printz, 1998, p. 13). Accepter l'existence d'erreurs humaines – tout en cherchant à en limiter les conséquences – semble difficile et constitue une rupture radicale avec un "système éducatif qui nie radicalement l'erreur et qui surtout la culpabilise, rendant ainsi sa prise en compte de façon sereine extrêmement problématique" (idem, p. 19-20). Il est pourtant inévitable que dans une activité aussi exigeante que la programmation se manifestent des "défaillances du processus psychocognitif"[67] dont la conséquence sera une erreur humaine. Un problème supplémentaire provient du fait que quand une défaillance logicielle se manifeste, si celle-ci est inévitablement le produit d'une erreur humaine, il peut être très difficile (et dans certains cas impossible) de l'imputer à un individu ou à un collectif d'individus précis : la responsabilité est distribuée, au sens fort, entre l'ensemble des agents impliqués (Laurent Thévenot, 1997, p. 236).

Cette difficulté à reconnaître que ce sont des erreurs humaines qui sont à l'origine des défauts des logiciels, se retrouve dans le terme choisi pour les désigner : les *bugs* (que l'on a tenté de franciser sans grand succès en bogues). C'est une mathématicienne, Grace Murray Hopper, qui travaillait sur le Mark 2 de l'Université de Harvard le 9 septembre 1947, qui décida d'appeler "*bug* tout ce qui empêche un programme de marcher", après que l'on eut

[66] Un exemple récent est le cas de la première génération de microprocesseurs "Pentium" commercialisés, dont le programme intégré pour effectuer les divisions sur des nombres à virgule flottante, comportait une erreur.

[67] Jacques Printz a établi une nomenclature impressionnante de la variété des défaillances psychocognitives qui peuvent se manifester dans l'activité de conception de logiciels (p. 101).

constaté qu'une défaillance sur un de ces premiers ordinateurs s'expliquait par la présence d'un papillon de nuit dans les circuits de l'appareil[68]. Il est significatif, qu'alors que la présence intempestive de ces créatures naturelles ne peut plus expliquer des défaillances informatiques depuis plusieurs dizaines d'années, ce soit une expression qui fait disparaître toute référence humaine au problème, que l'on continue à utiliser.

2 - …avec l'augmentation de l'importance des logiciels

Ces erreurs sont de plus en plus difficiles à accepter dans la mesure où leurs conséquences concernent directement et indirectement de plus en plus de monde. L'environnement socioéconomique est de plus en plus dépendant du bon fonctionnement de nombreux systèmes informatisés (téléphone, contrôle aérien, TGV, billetteries, futur système informatisé pour la santé). Quand ces systèmes rendent des services en ligne au grand public, les pannes sont jugées insupportables (Jacques Printz, 1998, p. 260). Les erreurs en informatique ont toujours eu "la particularité d'arriver à tout bloquer rapidement" (Philippe Breton, 1990, p. 37), mais les risques sont plus graves avec l'interconnexion grandissante de l'ensemble des systèmes informatiques. En outre, la complexité croissante de ces systèmes augmente considérablement les coûts des réparations : Jacques Printz estime que le coût de réparation d'un système suite à une défaillance peut être très supérieur à 10 hommes/année d'effort pour une seule réparation (1998, p. 52). Les dépenses engagées par les entreprises et les administrations pour que leurs systèmes passent l'an 2000 (en raison du codage des années sur deux chiffres), sont délicates à évaluer dans la mesure où une part importante de ces dépenses consiste à moderniser des systèmes, une modernisation qui aurait été effectuée ultérieurement en l'absence du bogue de l'an 2000. Au niveau mondial, les estimations se situent entre 300 et 600 milliards de dollars selon le Gartner Group (250 milliards de dollars selon IDC) ; les 330 plus grandes entreprises américaines ont dépensé 22,8 milliards de dollars selon une enquête du Wall Street Journal (3/1/2000), le record étant atteint par le groupe bancaire Citigroup qui, à lui seul, a dépensé 950 millions de dollars.

En même temps, le fait que le marché informatique, longtemps dominé par l'offre, soit de plus en plus déterminé par la demande, fait que les utilisateurs sont de moins en moins disposés à supporter les conséquences des défauts logiciels. Par exemple, le groupe Toshiba a

[68] Elle ajouta que c'était "le premier cas réel de bug à avoir été trouvé" dans la mesure où cette expression était déjà utilisée pour désigner des pannes du télégraphe électrique depuis son origine.

provisionné un milliard de dollar pour éviter un procès visant un disfonctionnement du logiciel pilotant les lecteurs de disquettes de ses ordinateurs portables susceptible d'entraîner des pertes de données, ce problème concernant plus de 5 millions d'ordinateurs vendus aux Etats-Unis depuis 1985. D'autres procédures judiciaires visant des grands noms de l'informatique (Packard Bell, filiale de NEC, Hewlett-Packard, Compaq…) sont en cours (18h.com le quotidien de l'Expansion, 2-11-99).

Cependant, le problème des défauts logiciels ne va pas s'estomper car ils sont de plus en plus difficiles à éliminer.

C - …MAIS DE PLUS EN PLUS DIFFICILES A ELIMINER

1 - Des logiciels de plus en plus complexes dans des environnements de plus en plus diversifiés

Certes des progrès substantiels ont été effectués par rapport à l'époque des débuts de l'informatique, où Jean-Marie Desaintquentin et Bernard Sauteur parlaient de l'état à peine avouable des logiciels de base quand ils arrivaient chez le client et où la mise au point était achevée sur le site (1991, p. 10). Deux facteurs importants de l'existence de nombreux défauts durant cette période ont disparu. Il s'agit d'une part du manque de ressources des premières générations d'ordinateurs qui avait créé une culture de la programmation fondée sur la recherche du codage des données et des algorithmes le plus compact possible (Jacques Printz, 1998, p. 136), et d'autre part, de l'utilisation de langages de programmation de bas niveau. Mais en même temps, les progrès réalisés dans ces domaines ont entraîné l'apparition de nouveaux problèmes : les langages de haut niveau permettent de masquer la complexité de l'architecture matérielle sous-jacente et leur utilisation permet d'effectuer moins d'erreur lors de la conception des logiciels, mais lorsqu'il se produit une défaillance difficilement explicable du système, il est nécessaire de bien comprendre ce que fait le système jusqu'à des niveaux de détails architecturaux masqués par le langage utilisé. De plus la rapidité des changements dans le secteur informatique empêche fréquemment d'atteindre la phase de maturité, où "le taux de défaillance par unité de temps est stable, ce qui veut dire qu'il y a autoadaptation entre le système, les usagers et les utilisateurs" (Jacques Printz, 1998, p. 346). Surtout, la formidable augmentation des ressources matérielles mises à disposition a entraîné

la création de logiciels d'une taille[69] et d'une complexité de plus en plus importante (cf. chapitre III).

Or, "le nombre d'erreurs croît de façon non linéaire avec la taille des systèmes [et] leurs fréquences d'apparition augmentent avec la puissance des installations" (Jacques Printz, 1998, p. 11)[70]. De plus, un aspect critique pour la fiabilité des logiciels est qu'ils constituent un des composants d'un système composé d'autres logiciels, d'unités centrales et de périphérique. Avec l'explosion de la microinformatique et l'existence de périphériques et de logiciels de plus en plus divers, il existe une "infinie variété de configurations de matériels et de logiciels, assemblés de façon personnalisée en systèmes adaptés à chaque utilisation particulière" *(*Gérard Dréan, 1996 A, p. 311). Des incertitudes supplémentaires viennent de la mise en réseau progressive de l'ensemble de ces équipements et de leur ouverture sur le monde extérieur, ce qui les rend particulièrement sensible aux "agressions" extérieures[71]. De plus en plus de systèmes, fonctionnant en temps quasi réel, tendent à s'interconnecter (systèmes boursiers et bancaires, systèmes de transports et de réservations de plus en plus divers). Les interactions se multiplient entre de plus en plus d'acteurs (hommes, organisations, ordinateurs, réseaux, capteurs, actionneurs) avec des procédures et des règles à respecter toujours plus nombreuses. S'appliquent les conclusions des travaux d'Ashby : plus on augmente le nombre de connexions dans un système, plus celui-ci tend à devenir instable (cité par Pascal Petit, 1998, p. 405)[72].

---

[69] Un logiciel d'un million de lignes de code source est devenu courant.

[70] "Quand vous augmentez de 50 % la taille d'un programme, il faut 100 ou 150 % de temps en plus pour le déboguer" (Jon Shirley de Microsoft, cité par Frédéric Dromby, 1999, p. 640).

[71] Ces "agressions" peuvent dans certains cas être intentionnelles. Il s'agit alors de "virus" informatiques, métaphore partiellement adéquate pour désigner la propagation de petits programmes malveillants : avec l'interopérabilité des systèmes on peut parler de disparition des "barrières immunitaires" existantes. Toutefois, un virus informatique est toujours une création humaine. Selon le cabinet de recherche Computer Economics, pour l'année 1999 c'est plus de 12 milliards de dollars qui, à l'échelle mondiale, sont allés à la prévention contre les virus, à la remise en état des systèmes et à la compensation des pertes de données occasionnées par ces attaques (Les Chroniques de Cybérie, 18-1-2000).

[72] Dans un livre empreint d'un certain catastrophisme ("La bombe informatique", 1998), Paul Virilio fait une analogie entre l'interactivité qui est à l'information ce que la radioactivité est à l'énergie, en précisant que les risques de la première sont plus importants ("l'émergence redoutable de "l'Accident des accidents", p. 148) vu son caractère global, alors que la radioactivité n'est que locale.

Enfin, l'amélioration de la "convivialité" des applications développées s'accompagne également d'un fort accroissement de complexité[73]. Dans le même temps, le passage des architectures centralisées, où les traitements s'effectuent sur un mainframe avec de simples moniteurs transactionnels pour les usagers, à des architectures clients-serveurs où les traitements sont répartis sur l'ensemble des machines connectées et où les interactions sont multidirectionnelles, crée de nouvelles difficultés pour localiser les défauts responsables d'une défaillance. L'ensemble de ces évolutions contribue à la multiplication des défauts "diffus" à la différence des défauts localisés physiquement dans un programme bien précis, qui en général déclenchent une panne franche facilement détectable et reproductible. A l'inverse les défauts diffus engendrent fréquemment des pannes intermittentes et fugaces, résultant d'une combinaison d'actions très diverses, et sont donc très difficiles à reproduire, ce qui rend la correction de ces erreurs particulièrement laborieuse et coûteuse (Jacques Printz, 1998, p. 58).

On peut ajouter qu'une cause de complexité supplémentaire, qui fut la source de nombreux problèmes, provient paradoxalement des mesures de sécurité qui conduisent pour des missions critiques à ajouter au logiciel de multiples fonctions de sécurité, ou à faire exécuter les mêmes traitements par plusieurs ordinateurs différents (jusqu'à cinq pour la navette spatiale) pour le cas où un ordinateur tomberait en panne (Lauren Ruth Wiener, 1994, p. 56 et 170).

"Jamais dans son histoire, l'homme n'a fait face à un tel niveau de complexité. Outre les erreurs de conception, les erreurs de manipulation sont inévitables. L'environnement du système est plus ou moins bien contrôlé, voire même incontrôlable. Tout le problème est donc, non pas d'éviter les erreurs - c'est désormais définitivement impossible - mais de s'organiser pour vivre avec, et de limiter les dégats qu'elles peuvent occasionner" (Jacques Printz, 1998, p. 19).

2 - L'impossibilité des logiciels sans défaut et ses conséquences

"Les informaticiens, à la différence des non initiés savent que le logiciel sans défaut est impossible" (OCDE, 1991 A, p. 9). "Toute personne bien informée des questions de génie logiciel et de formes plus classiques d'ingénierie pourra vous confirmer que l'état de l'art en

[73] Jacques Printz estime que la demande irrépressible en facilité d'emploi et en convivialité s'accompagne d'une croissance au minimum exponentielle de la complexité du logiciel (1998, p. 200).

génie logiciel est loin d'atteindre l'état de développement d'autres secteurs d'ingénierie. Lorsqu'un produit d'ingénierie classique a été réalisé, testé et mis sur le marché, on peut raisonnablement penser qu'il a été correctement conçu et qu'il fonctionnera de manière fiable. Il est normal, en revanche, qu'un produit logiciel contienne des bogues importantes et ne fonctionne pas de manière satisfaisante. (…) Les professionnels du logiciel (…) savent que les programmeurs les plus compétents ne peuvent éviter ce genre d'incidents" (David L. Parnas, préface à Lauren Ruth Wiener, 1994, p. XV). Cette impossibilité, tout du moins pour des programmes d'une certaine importance, peut être montrée sur un plan théorique (a), mais elle résulte surtout de contraintes économiques (b).

a - Une impossibilité théorique

Cette impossibilité s'explique par la double nature d'un logiciel comme étant un texte numérique actif (cf. chapitre I). Un logiciel est un texte numérique et sa création s'effectue sur un support d'une souplesse inégalée : "peu de supports de création se montrent si flexibles, si faciles à effacer et à retravailler, si aptes à accueillir de grandes structures conceptuelles" (Frederick P. Brooks, 1996, p. 6). Il permet au pouvoir de l'imagination de s'exercer pleinement, "le programmeur, comme le poète, [maniant] des abstractions voisines de la pensée pure" (idem). Ceci se retrouve dans l'extraordinaire diversité des logiciels qui sont écrits, tant au niveau de la variété des problèmes qu'ils tentent de résoudre, que de la diversité des solutions proposées pour un même problème[74]. Mais un logiciel est également un "texte qui agit" dans la mesure où il se compose d'un ensemble d'instructions qui seront exécutées automatiquement par une machine. "La construction du programmeur, au contraire des vers du poète, est réelle : elle agit, produisant des résultats distincts de la création elle-même" (idem). On peut distinguer dans toute activité créatrice trois étapes : l'idée, l'implémentation et l'interaction. Quand une création s'effectue sur un support matériel, la puissance de l'imagination est bridée par les contraintes de l'implémentation. Ce n'est pas le cas du logiciel où c'est seulement au moment de son interaction que se manifesteront les imperfections inévitables d'une activité humaine[75]. Le logiciel, au stade de sa conception est "libre des

[74] "Pour résoudre un même problème, vous ne trouverez jamais deux programmeurs proposer la même solution. Un programme est un miroir qui reflète l'esprit qui l'a produit, et les esprits humains sont très dissemblables" (Lauren Ruth Wiener, 1994, p. 120).

[75] Frederick P. Brooks ajoute deux arguments : le fait que le programmeur dépend tant pour la conception que pour l'exécution de son programme, des programmes d'autrui (qui également ne sont pas parfaits), et le fait

contraintes du sens commun et des lois de la physique" (Lauren Ruth Wiener, 1994, p. 64), mais dans son utilisation, comme tout artefact, il n'est pas "hors de la nature" et n'a "nulle dispense pour ignorer ou pour violer les lois de la nature" (Herbert Simon, 1974, p. 17). Par contre, on peut considérer que "la programmation, et plus encore l'analyse, reste malgré tout un art, un art rigoureux et logique, certes, mais un art quand même, qui résiste en tant que tel à tout algorithme" (Philippe Breton, 1987, p. 216), et il est malheureusement impossible de démontrer que le produit de cet art se comportera "correctement"[76].

Les limites des vérifications formelles des programmes

Certes, se sont développées des techniques de vérification formelle des programmes, parfois également appelées "preuves de correction". Il existe deux grandes familles de méthodes formelles qui ont en commun un substrat mathématique fort. Les méthodes constructives ou logiques sont inspirées des mathématiques axiomatisées qui garantissent qu'en appliquant des axiomes et des règles logiques de raisonnement, on engendre nécessairement des théorèmes vrais. Appliquées au logiciel, ces méthodes consistent, à partir des spécifications initiales du logiciel, à démontrer rigoureusement tous les raisonnements, qui par des procédés de raffinements successifs, conduisent au programme final. La plus représentative de ces méthodes est la méthode B, qui a succédé à la méthode Z, appliquée notamment à certains modules logiciels de sûreté dans les transports. Les autres méthodes sont les méthodes par vérification de modèles, qui reposent sur le fait que les programmes peuvent se modéliser à partir d'objets mathématiques aux propriétés bien définies : les automates. La vérification du programme consiste à explorer à l'aide d'algorithmes l'espace des états de l'automate résultant de toutes les interactions possibles. Leur mise en œuvre nécessite impérativement d'utiliser des langages de programmation spécifiques (Lustre, Esterel[77]…) qui ne permettent de construire que des automates, ce qui n'est pas le cas des langages de programmation classiques comme ADA, C, ou C++ (Jacques Printz, 1998, p. 111-112).

---

qu'il "est amusant d'élaborer de savants concepts, alors qu'extirper de petits bogues de son code n'est qu'un labeur ingrat" (1996, p. 7).

[76] Philippe Breton rappelle opportunément que Turing a "inventé" l'ordinateur, non pour calculer mais pour prouver les limites du calcul ; cela lui a servi à démontrer, entre autres, qu'il n'existait pas d'algorithme capable de démontrer qu'un autre algorithme pouvait résoudre un problème donné (1987, p. 216-217).

[77] Ces langages sont utilisés pour programmer des systèmes temps réels synchrones, fréquents en avionique ou dans les protocoles de communication.

Les limites de ces méthodes sont connues. La première de ces limites tient aux limites de la logique mathématique. Kurt Gödel a démontré en 1931, à partir de travaux sur les systèmes formels de l'arithmétique, que tout système complexe contient des énoncés indécidables, c'est à dire indémontrables si l'on demeure dans le formalisme du système considéré. Un système formel est nécessairement soit incomplet (une théorie est complète si toutes les propositions logiquement valides que l'on peut former avec les termes de cette théorie sont ou vraies ou fausses), soit incohérent (une théorie est cohérente ou consistante lorsque l'on ne peut pas démontrer un théorème et son contraire). Il est certes nécessaire de s'assurer que l'emploi qui est fait d'un modèle est "à la fois consistant et complet, mais ce sont malheureusement des propriétés indécidables par des algorithmes" et c'est "le cerveau humain, et lui seul, qui *in fine* pourra prononcer la validité de la construction logique, car lui seul est maître de la correspondance avec le réel" (Jacques Printz, 1998, p. 169).

En effet, ces méthodes ne peuvent au mieux que vérifier la conformité du programme aux spécifications initiales. Or celles-ci sont difficiles à établir[78] et peuvent omettre certains phénomènes extérieurs au système perçus comme des évidences. La spécification initiale est un modèle de la réalité[79] dont la validité ne peut être établie qu'expérimentalement. "Le processus d'abstraction, par la simplification qu'il impose, permet de passer du monde réel à des abstractions mathématiques idéalisées, mais ce processus n'est pas lui-même mathématique, et sa validité n'est pas du domaine de la preuve" (Jacques Printz, 1998, p. 113). Ces méthodes concernent le domaine de la validité (ou adéquation ou correction) du logiciel, c'est à dire l'aptitude du système à assurer exactement ses fonctions définies dans le cahier des charges, qu'il importe de différencier de la fiabilité (ou robustesse ou tolérance) qui désigne l'aptitude du système à fonctionner même dans des conditions anormales (Jean-Marc Geib, 1989, p. 20). Jacques Printz ajoute : "les figures du monde réel, comme les programmes

[78] "Il est presque toujours difficile pour l'expert humain de décrire le savoir dans des termes précis, complets et suffisamment cohérents pour pouvoir être utilisés dans un programme d'ordinateur. Cette difficulté vient de la nature inhérente du savoir qui constitue l'expertise humaine : il est souvent subconscient et peut-être approximatif, incomplet et incohérent". Cette citation de Buchanan et al. (1983) effectuée par Harry M. Collins, (1992, p. 125) à propos du problème de l'élucidation du savoir pour construire des systèmes experts, nous semble pouvoir être généralisée à l'élaboration des spécifications d'un logiciel.

[79] "Les modèles informatiques ne peuvent être que limités. Comme l'information numérisée, leur résolution est finie. On ne peut représenter qu'un certain nombre de propriétés, prendre en compte qu'un nombre limité de paramètres et ne calculer les résultats qu'à un nombre précis de décimales. Tous les modèles sont plus ou moins subtils, mais ils sont toujours moins subtils que l'infinie complexité de la réalité" (Lauren Ruth Wiener, 1994, p. 197).

concrets ne sont pas des objets mathématiques, mais ce qui permet de les dessiner et de les écrire peut l'être ; cela donnera à l'objet réel et concret un certain nombre de caractéristiques logiques minimales éventuellement démontrables. Quant à la réalité proprement dite, autant vouloir "démontrer" qu'un moteur n'explosera pas, ou qu'un pont ne s'effondrera pas, à la seule vue des équations qui gouvernent le phénomène physique caractéristique de l'artefact" (1998, p. 68).

Une autre limite de ces méthodes est l'importance du travail qu'elles requièrent pour être appliquées, ce qui les rend très coûteuses et fait qu'elles ne peuvent être utilisées que pour des composants logiciels de taille limitée[80]. Leur efficacité est controversée : Frederick P. Brooks, même s'il reconnaît que la vérification de programmes ne veut pas dire qu'on obtient des programmes sans erreur[81], estime que ces méthodes peuvent réduire (mais non supprimer) les tests à effectuer (1996, p. 169). Par contre, selon Capers Jones, aucune étude n'a montré que la fiabilité opérationnelle des programmes ayant subi des preuves de correction, était supérieure aux autres programmes (1989, p. 214 et 215).

L'influence de l'environnement

Une deuxième raison de l'impossibilité théorique des logiciels sans défauts résulte du fait qu'une défaillance d'un programme peut provenir soit de l'algorithme du programme uniquement, soit de la conjonction de l'algorithme et de son comportement dans un environnement donné. Or si un algorithme est généralement déterministe (sauf s'il est faux !), ce n'est pas le cas du comportement d'un programme qui est fondamentalement non déterministe. On oublie fréquemment qu'un ordinateur n'est pas rigoureusement déterministe,

[80] Le coût est encore plus élevé si l'on tente de combiner la vérification formelle du logiciel avec des méthodes pour élaborer des spécifications proches des utilisateurs. David Gianazza et alii (1997) ont ainsi développé pour un système de contrôle de trafic aérien, un premier système à partir des spécifications des utilisateurs, puis ils ont procédé à une rétro-conception pour aboutir à une spécification formelle en Z permettant une validation du modèle. Celle-ci a permis de détecter des incomplétudes, des ambiguïtés et quelques contradictions, invalidant les spécifications initiales, ce qui nécessitait de reprendre tout le processus de développement à son point de départ. Les auteurs en concluent que l'application conjointe des deux méthodes semi-formelle et formelle, si elle est particulièrement intéressante, ne peut être utilisée que pour certains éléments et pas pour un système dans son ensemble.

[81] Frederick P. Brooks ajoute aux arguments déjà mentionnés le fait que même les démonstrations mathématiques peuvent être fausses.

est une machine à états discontinus[82] - alors que la plupart des phénomènes physiques sont continus - et qu'il ne respecte pas toujours les règles logiques de l'arithmétique élémentaire, la logique ignorant le temps et la durée ce qui n'est pas le cas des ordinateurs[83]. Le non-déterminisme des ordinateurs est aggravé par deux caractéristiques importantes des systèmes informatiques : la recherche d'un partage maximal des opérations et des ressources ; l'asynchronisme et la non-instantanéité des opérations effectuées par la machine[84]. L'importance de ces facteurs augmente avec l'existence de programmes de plus en plus dynamiques et interactifs[85]. En fonction de la durée de la session, du niveau d'interactivité (nombre d'usagers connectés), de la capacité du programme à créer dynamiquement les entités nécessaires à son exécution et de la vitesse relative des différentes unités qui interagissent au sein d'une même machine, il se produit une dégradation inévitable de l'environnement et l'apparition de "fantômes" (Jacques Printz, 1998, p. 117). L'existence de ces "fantômes", qui ne peuvent être détruits que par l'arrêt physique de la machine et la réinitialisation de toutes ses structures de travail et du système d'exploitation, n'a par définition pas été prévue lors de l'écriture d'un programme. Elle rend la détermination des causes de certaines défaillances, et donc la correction des erreurs, particulièrement difficiles, puisque l'on ne peut reproduire exactement le contexte d'apparition de la défaillance. Une solution consiste à rajouter des programmes observateurs, appelés "démons" dont le rôle consiste à noter tous les états intermédiaires du processus de traitement de façon à faciliter le diagnostic en cas de défaillance. Cette solution n'est toutefois pas parfaite car les programmes "démons"

[82] Un ordinateur "est une machine à états discrets, ne pouvant traiter que des informations de type digital, à l'exclusion par exemple des informations continues, qui sont pourtant dominantes dans la nature et forment sans doute l'une des bases du raisonnement humain" (Philippe Breton, 1987, p. 163).

[83] Même si les performances des ordinateurs sont extrêmement spectaculaires, il faut intégrer le fait qu'avec le temps les logiciels peuvent être utilisés dans des conditions qui dépassent les marges de sécurité prévues dans les spécifications initiales. Par exemple, le logiciel installé sur le système de défense antimissile Patriot générait une erreur d'un millionième de seconde toutes les dix secondes pour tracer la fenêtre de poursuite des missiles. Cette erreur restait sans conséquence tant que le système était utilisé pendant moins d'une journée sans interruption ce qui correspondait au cahier des charges de l'armée américaine. Par contre pendant la guerre du golfe ce système fut utilisé pendant cinq jours sans interruption. De ce fait, l'erreur n'était plus négligeable et fut responsable de la non-interception d'un missile irakien le 25 février 1991 qui tua vingt-huit soldats américains et en blessa quatre-vingt-dix-huit autres (Lauren Ruth Wiener, 1994, p. 31-32).

[84] Si le système était synchrone, il devrait fonctionner au rythme de son organe le plus lent. Or il existe de très grandes différences de vitesse d'exécution entre les différents organes d'un ordinateur.

[85] Ces facteurs sont particulièrement critiques dans le cas de traitement en parallèle où le programme exécute des instructions sur plusieurs processeurs simultanément.

consomment eux-mêmes des ressources, en proportion de l'importance des observations que l'on veut effectuer, et perturbent de ce fait l'observation de l'état réel du système[86].

En conséquence, "dans l'état actuel des technologies de programmation, le seul moyen de vérifier qu'un programme est correct est de l'exécuter" (Gérard Dréan, 1996 A, p. 216)[87]. C'est ce que réalisent les tests. Mais l'exécution des tests, même très nombreux, ne permet pas de certifier qu'un programme ne contient pas d'erreur. D'une part, il faudrait énumérer tous les cas possibles de défaillance des modules et de leurs combinaisons, puis procéder aux vérifications correspondantes à l'aide de test. Les combinaisons induites étant en n puissance n, les coûts et les délais de réalisation deviennent rapidement impraticables (Jacques Printz, 1998, p. 110). D'autre part, les programmes de tests et les jeux d'essais peuvent eux-mêmes contenir des erreurs[88] : Capers Jones cite une étude informelle menée à IBM en 1975 qui montrait que la densité des erreurs des données de test d'un programme était supérieure à celle du programme lui-même (1989, p. 216).

On ne peut donc jamais certifier qu'un programme ne contient plus d'erreur, mais plus modestement dire "que la probabilité de détecter une erreur existante croît avec la durée des tests, et que la probabilité qu'il existe des erreurs résiduelles est d'autant plus faible que le système a fonctionné sans incidents pendant plus longtemps dans une plus grande variété de configurations" (Gérard Dréan, 1996 A, p. 218).

Cette impossibilité d'une fiabilité totale des logiciels dans toutes les circonstances a des implications sur la conception des logiciels, qui va à rebours de l'utopie technologique dominante, pour laquelle "tout changement, toute modification d'une situation, toute résolution de problème, passe désormais par une solution technologique, notamment en

---

[86] La présence du "démon" fausse l'observation ; ce qui est en fait observé c'est le système plus son "démon". On peut établir une analogie avec l'observation des phénomènes microphysiques, où le comportement externe observé est la somme des comportements du phénomène proprement dit, et de celui de l'observateur.

[87] Cela rejoint le constat que l'analyse théorique ne peut se passer d'expériences (Herbert Simon, 1974, p. 36-37).

[88] De façon plus générale une défaillance, même en phase d'utilisation n'est pas nécessairement perçue comme telle, si le programme fournit des données en sortie erronées mais plausibles. Lauren Ruth Wiener cite le cas d'un logiciel qui fournissait des prévisions incorrectes sur les marges brutes de ventes et donc sur les bénéfices. Ce n'est qu'à la fin de l'année que l'entreprise se rendit compte que son bénéfice estimé à 500 000 dollars était en fait une perte de 4 millions de dollars, ce qui contraignit la société à déposer son bilan (1994, p.36).

termes d'information et de communication" (Philippe Breton[89]). Par exemple, dans le domaine des systèmes experts, il est illusoire et il peut être dangereux de vouloir remplacer des experts par des machines ; il est préférable de limiter les ambitions des systèmes experts au rôle d'aide des experts humains (Harry M. Collins, 1992, p. 236). De façon plus générale, il faut, quand c'est possible, prévoir des possibilités d'intervention humaine qui contournent les automatismes, pour traiter les cas particuliers qui n'auront pas tous été prévus par les concepteurs[90]. Pour les systèmes critiques, il faut envisager systématiquement les conséquences d'une défaillance toujours possible de ces systèmes, ce qui peut conduire à limiter les rôles de tels systèmes[91]. Notamment, il importe de se poser la question de la comparaison entre la situation résultant d'une défaillance (toujours possible) d'un système et la situation caractérisée par l'absence de système. L'impossibilité pour pouvoir tester dans des conditions réalistes les logiciels de l'initiative de défense stratégique (IDS), ce qui aurait nécessité de procéder à des explosions nucléaires à l'air libre, a motivé l'opposition à ce projet du professeur David L. Parnas, membre du comité de réflexion sur les aspects informatiques de l'IDS, pour qui il est impossible qu'un système logiciel de grande taille fonctionne correctement dès son premier usage[92].

---

[89] in "L'utopie de la guerre technologique", *Le Monde*, 30 avril 1999.

[90] Ces possibilités reposent sur la conception que "la machine va plus vite que l'homme pour traiter de la même façon les mêmes informations, tandis que l'homme, dont le raisonnement est plus que de la "simple" logique, recèle bien d'autres possibilités que celles de la machine" (Philippe Breton, 1990, p. 92). Elles supposent l'entretien des compétences humaines pour effectuer des opérations, qui faute de pratique risquent de s'atrophier (Abbe Moshowitz, 1997, p. 35-36). Dans de nombreux cas, elles nécessitent l'existence d'une "traçabilité" des traitements automatiques qui ont échoué, pour comprendre les raisons de ces échecs. Dans le domaine de l'aviation, Lauren Ruth Wiener cite le cas de la protection d'enveloppe installée sur la plupart des systèmes numériques de pilotage et qui garantit que les commandes de l'avion ne franchiront pas des limites mettant en cause l'intégrité de la structure de l'avion. Dans certaines situations quasi désespérées, il peut être préférable que le pilote puisse dépasser ces limites (1994, p. 117 et 167).

[91] Par exemple, dans le cas du traitement par radiations où un défaut logiciel fut responsable de plusieurs décès, le simple maintien des systèmes de sécurité mécaniques qui équipaient les générations antérieures d'appareils non automatisés, aurait suffi à éviter les conséquences dramatiques de la défaillance du logiciel. De même, un logiciel destiné à économiser le carburant en ralentissant au maximum les moteurs sur les Boeing 767 fut responsable d'un vol plané de 14 minutes lors de la descente vers un aéroport. La température extérieure était exceptionnellement basse de sorte que le ralentissement des moteurs entraîna la formation de glace qui provoqua l'arrêt des moteurs. Le paramètre de la température extérieure, critique dans ce cas précis, n'avait pas été pris en compte par les concepteurs du logiciel (Lauren Ruth Wiener, 1994, p. 71).

[92] Il est amusant de noter qu'un autre expert, Solomon J. Buchsbaum, directeur aux laboratoires ATT Bell, consulté par le congrès des Etats-Unis sur les logiciels destinés au projet d'IDS, estima que l'on pouvait concevoir un tel système avec une fiabilité garantie, la preuve en ayant été apportée par système logiciel du réseau public américain de télécommunications… quatre années avant que ce système soit responsable d'une impossibilité de communiquer pour la moitié de la clientèle d'ATT (Lauren Ruth Wiener, 1994, p. 141).

b - Les contraintes économiques

Cependant, affirmer qu'une fiabilité parfaite est impossible ne signifie nullement que tous les logiciels aient des niveaux de fiabilité équivalents. Ce qui va jouer un rôle déterminant ce sont les arbitrages économiques différents qui vont être effectués entre la fiabilité et les autres caractéristiques du logiciel et de son processus de production. Nous avons signalé à de nombreuses reprises le temps et le coût très élevé, pour détecter et corriger les défauts, facteurs qui augmentent fortement lorsque le nombre de défauts restants diminue.

Les impératifs d'une production marchande de logiciels font prédominer "la contrainte qu'un système doit être développé avec un coût convenu à l'avance et dans un délai donné" (Maarten Boasson, 1998, p. 4). Dans la pratique, une entreprise fixe un calendrier pour des raisons de marché ou de concurrence et laisse les erreurs qui sont découvertes trop tard en les signalant et en les corrigeant dans une prochaine version (Gérard Dréan, 1996 A, p. 218). De même, Pierre Bonnaure estime qu'une des raisons importantes de la présence de bogues dans les progiciels est leur commercialisation prématurée (1996 B, p. 66). Ce phénomène est particulièrement présent pour les développeurs de logiciels destinés au "grand public", qui "travaillent le plus souvent sous une telle pression que les produits qu'ils livrent sont criblés d'imperfections" (Lauren Ruth Wiener, 1994, p. 98). Dans la mesure où les tests constituent le plus souvent la dernière étape du processus de développement, il est fréquent que la phase de test soit raccourcie en raison des dépassements de calendrier des étapes précédentes. Outre les problèmes de délais, qui, en raison de l'importance des rendements croissants d'adoption dans ce secteur, peuvent être décisifs, il faut prendre en compte les coûts de développement : Barry W. Boehm indique que la productivité s'effondre si l'on recherche une fiabilité extrême comme dans le cas du logiciel de commandes de la navette spatiale développé par IBM (Frederick P. Brooks, 1996, p. 188)[93]. Katia Messika, dirigeante d'une entreprise spécialisée dans le test des logiciels (*International Testing*), précise qu'elle teste les produits sur cinquante machines, qui correspondent à 80 % du parc informatique[94] et ajoute que, si "tous

[93] Ce logiciel a coûté environ mille dollars la ligne, alors que la moyenne se situe entre vingt-cinq et cent dollars. Malgré cela, en se basant sur le nombre de bogues découvertes dans des versions antérieures, le producteur lui-même, IBM, estime qu'il contient encore une cinquantaine de bogues (Lauren Ruth Wiener, 1994, p.155).

[94] Ceci implique qu'un utilisateur sur cinq risque de se retrouver confronté à des bogues dus à un conflit entre le programme et le matériel. Il faut de plus ajouter que ces tests sont faits sur des machines "vierges" et ne peuvent donc intégrer les bogues résultant d'interactions avec d'autres programmes installés par l'utilisateur.

les bugs que nous signalons sont corrigeables, tous ne sont pas corrigés" (*Libération-Multimédia*, 31 décembre 1999)[95]. En effet, en intégrant le fait que ces défauts ne sont pas visibles au moment de l'achat du logiciel et que les préjudices éventuels ne sont généralement pas à la charge de l'entreprise productrice, la présence de défauts peut ne pas avoir que des conséquences négatives pour certains producteurs placés en situation de quasi-monopole : revenus provenant du service après vente[96], incitation à acheter les nouvelles versions qui corrigeront notamment certains des bogues constatés[97].

Selon Capers Jones, "la seule façon de se rapprocher d'un rendement de 100 % dans l'élimination des défauts est de tourner en production sur une longue période avec l'ensemble des clients, bien que ce soit là une méthode coûteuse, qui cause souvent beaucoup de soucis à l'équipe de développement et une certaine exaspération chez les utilisateurs de programmes" (1989, p. 217)[98]. De même, Lauren Ruth Wiener, estime que "rien n'est plus efficace pour mettre au jour des bogues qu'une utilisation massive pendant des années dans des conditions réelles et impitoyables" et que "l'usage réel par des utilisateurs réels permet de découvrir plus de bogues que n'importe quel programme de tests" (1994, p. 132)[99]. Ce type de pratique, qui suppose une coopération en partie désintéressée des utilisateurs, est difficilement praticable dans un cadre marchand avant la commercialisation du produit. Si l'on souhaite de plus que certains utilisateurs puissent non seulement signaler les défauts rencontrés mais également proposer des solutions à ces défauts, cela suppose de diffuser le code source du logiciel ce qui rend difficile la vente du logiciel (cf. chapitre VII).

---

[95] Selon Mark Minasi (2000), "90 % des bugs signalés par l'utilisateur final étaient déjà connus de l'éditeur avant la mise en vente".

[96] Ces revenus peuvent être très importants, notamment quand l'entreprise qui commercialise le logiciel ne diffuse pas son code source (cas le plus fréquent) ce qui la met en situation de monopole pour le service après vente du logiciel.

[97] Précisons également que les nouvelles versions si elles corrigent des anciens bogues identifiés peuvent en introduire de nouveaux, notamment quand l'introduction de nouvelles fonctionnalités aux effets secondaires mal identifiés, est ajouté rapidement au produit sans reconcevoir et retester l'ensemble du logiciel.

[98] Une pratique proche pour les progiciels consiste à proposer à certains utilisateurs une version dite *bêta* (déjà testée par les développeurs mais non encore commercialisée), en espérant que ces utilisateurs transmettront les bogues qu'ils constateront.

[99] Pour les systèmes techniques en général, Jacques Girin note que "les réseaux de relations qui aboutissent à des systèmes techniques n'ayant pas d'utilisateurs réels et contemporains – mais seulement des utilisateurs potentiels et futurs – ont toutes les chances d'être défaillants" (1994, p. 28).

En fait, il faut lier la question de la fiabilité du logiciel aux autres caractéristiques du logiciel, avec lesquelles les relations peuvent être contradictoires : la rapidité d'exécution du logiciel (une plus grande fiabilité consommant des ressources qui peuvent être importantes[100]), la portabilité du logiciel (avec les contraintes supplémentaires provenant de sa possibilité d'exécution sur des machines différentes), la "convivialité" (source d'une complexité additionnelle). Les différents producteurs de logiciels effectuent des compromis qui peuvent être très variables entre ces différentes caractéristiques, et qui ne sont pas toujours en faveur de la fiabilité. Pour certains utilisateurs et certains types d'utilisation la fiabilité n'est pas nécessairement la caractéristique principale, d'autant plus qu'en cas de disfonctionnement (pas toujours perceptible[101]) il peut être difficile de savoir quel est le produit incriminé.

Section III - Des logiciels inadaptés aux besoins des utilisateurs ?

Il s'agit ici d'examiner, au-delà de leur fiabilité, dans quelle mesure les logiciels sont conformes aux attentes des utilisateurs, indépendamment du fait que ces attentes aient été formellement exprimées. C'est le problème plus général de la qualité, au sens classique du terme, qui est posé. En effet, l'AFNOR qui définissait la qualité d'un produit comme "son aptitude à satisfaire les besoins des utilisateurs" (définition AFNOR NF X 50-109), a modifié cette définition en juillet 1982, pour préciser qu'il s'agissait "des besoins exprimés ou implicites" (AFNOR NF X 50-120)[102]. Cette précision est importante pour les logiciels où "la qualité des applications informatisées s'apprécie autant, voire même plus, par la capacité permanente de compréhension et d'adaptation des informaticiens à la problématique des utilisateurs, que par la conformité des spécifications ou des produits résultant simplement de la bonne application d'une démarche professionnelle maîtrisée" (Serge Bouchy, 1994, p. 171). En ce domaine, la principale difficulté consiste à "passer de l'expression imprécise d'un besoin

[100] C'est pour cette raison que les logiciels de commande de processus (comme les systèmes de contrôle aérien, ou de pilotage d'une centrale nucléaire), qui sont des logiciels critiques de sécurité, sont particulièrement difficiles à développer dans la mesure où ils exigent simultanément un haut niveau de fiabilité et une grande vitesse d'exécution pour réagir en temps quasi réel.

[101] Un des reproches adressé au système d'exploitation de Microsoft est qu'il laisse des morceaux de mémoire non récupérés après fermeture des applications, ce qui à terme dégrade les performances. Ceci n'est pas directement perceptible pour un utilisateur non expert.

[102] De même la norme internationale ISO 8042 définit la qualité comme "l'ensemble des caractéristiques d'une entité qui lui confère l'aptitude à satisfaire des besoins exprimés et implicites".

utilisateur, à la mise en place d'un produit sans faille, activable par des utilisateurs divers dans des conditions multiples et qui correspond aux attentes opérationnelles des usagers" (idem, p. 29).

De ce point de vue, l'insatisfaction des utilisateurs à l'égard des logiciels apparaît comme importante, même si nous verrons qu'il est nécessaire de relativiser le constat effectué, la qualité étant plus une construction sociale qu'une réalité objective (A). En première analyse cette qualité perçue comme insuffisante peut s'expliquer par une relation initialement très déséquilibrée entre les informaticiens et les utilisateurs (B). L'amélioration du rapport de forces en faveur des utilisateurs a contribué, entre autres, à des progrès significatifs dans la qualité des logiciels, mais la diversité croissante des logiciels et de leurs utilisateurs, et les exigences de la concurrence – qui peut ne porter qu'accessoirement sur la qualité des produits – font qu'il existe une insatisfaction permanente des utilisateurs (C). Celle-ci peut également s'expliquer par les spécificités du logiciel (D).

A - LE CONSTAT D'UNE INADAPTATION APPAREMMENT PERMANENTE AUX BESOINS DES UTILISATEURS

1 - Un constat…

Selon Serge Bouchy, "le bon logiciel n'est pas seulement celui qui, *in fine*, répond exactement à l'énoncé d'origine, mais c'est surtout celui qui satisfait l'utilisateur final, au moment de son emploi, dans des conditions de coûts et de délais qui correspondent aux ressources de l'entreprise concernée" (1994 p. 175). Selon cette conception, le bilan n'apparaît pas comme étant particulièrement positif : "les concepteurs de logiciels sont souvent accusés d'être en retard sur les projets et de ne pas apporter les fonctions promises. (…) [il existe] une impression persistante que l'utilité et la facilité d'utilisation (…) sont souvent surestimés par les fournisseurs, les utilisateurs devant s'accomoder de systèmes inappropriés" (OCDE, 1993, p. 114).

Même s'il faut considérer avec prudence des estimations statistiques toujours partielles, la proportion des applications qui ne sont jamais utilisées ou qui nécessitent d'importantes modifications est impressionnante. Une étude célèbre des services de la comptabilité gouvernementale américaine effectuée en 1979 sur neuf grands projets de l'administration américaine en tirait un bilan désastreux :

Tableau XXXXI
Bilan des logiciels de neuf grands projets de l'administration américaine
(en milliers de dollars et en pourcentage de l'ensemble)

	Montant	Pourcentage
Logiciels utilisés tel que livrés	118	2%
Logiciels utilisés après modifications	196	3%
Logiciels utilisés mais refondus ou abandonnés plus tard	1300	19%
Logiciels payés mais non livrés	1950	29%
Logiciels livrés mais jamais utilisés avec succès	3200	47%
Total	6764	100%

Source : Etude des services de la comptabilité gouvernementale américaine, citée par Patrick Jaulent (1992, p. 16)

Frédéric Georges Roux à partir d'une enquête de satisfaction auprès des entreprises françaises estime que 53 % des applications sont "inadéquates et à refaire" (1991, p. 39). Concernant les systèmes d'information, Marie-Christine Monnoyer-Longe note que les échecs sont de 20 à 50 % selon les sources, et estime que l'insatisfaction des utilisateurs dépasse 30 % (1997, p. 118). Les fonctionnalités disponibles ne sont que très partiellement utilisées (Pierre Bonnaure, 1996 B, p. 67) ; dans le cas des progiciels c'est seulement 20 % en moyenne de ces fonctionnalités dont se servent les utilisateurs (Eurostaf, 1996 C, p. 79). L'étude des services des grandes organisations françaises montre que dans de nombreux cas, l'informatique est moins vécue comme une aide que comme un "casse-tête supplémentaire" (Serge Bouchy, 1994, p. 182). Cette qualité insuffisante ne concerne pas que le fonctionnement du logiciel, mais également sa documentation : celle-ci est jugée difficile à comprendre dans 75 % des cas et contient des erreurs dans la moitié des cas ; elle manque le plus souvent d'"exemples qui marchent vraiment" (Capers Jones, 1989, p. 217-218).

De façon plus générale, Philippe Breton (1990) estime que le développement de l'informatique a toujours été fondé sur la promesse de services à venir et non sur les services réellement rendus.

Une autre façon d'apprécier la qualité plus faible des logiciels que des autres produits, est la mesure de la maturité des processus de développement de logiciels. Celle-ci peut être

effectuée à partir d'un modèle, *Software Process Maturity Model* (SMM)[103] élaboré en 1991 par le *Software Enginneering Institute* (SEI) de l'Université de Carnegie Mellon. Ce modèle se base sur les mêmes principes que la série des normes ISO 9000, c'est à dire que la qualité du produit est appréhendée à partir de la qualité du processus de fabrication. Il analyse une organisation selon cinq axes (organisation de projet, ingénierie du produit, gestion quantitative, amélioration continue et technologie utilisée) et dix-huit domaines d'activité principaux (planification de projet, gestion des spécifications, assurance qualité…). Le modèle distingue cinq niveaux de maturité (initial, répétitif, défini, géré, optimisé) ; seul un nombre infime d'organisations atteignent le niveau 3, les niveaux 4 et 5 étant "des territoires quasiment inexplorés en ce qui concerne le logiciel" alors que "les caractéristiques de ces niveaux ont été définies par analogie avec ce qui s'observe pour d'autres types d'industrie" (Luc Rubiello, 1997, p. 232).

2 - …qu'il convient de relativiser

Toutefois il importe de relativiser ce constat négatif et le fait qu'il serait de la responsabilité exclusive des producteurs de logiciels.

La responsabilité des utilisateurs

Tout d'abord, à côté des reproches des clients-utilisateurs aux prestataires, il est nécessaire de prendre en compte les reproches des prestataires à leurs clients, notamment dans la formulation des besoins. A partir de l'observation de projets ayant dû subir des modifications, Capers Jones remarque que "les utilisateurs eux-mêmes ne savaient pas toujours très bien ce qu'ils voulaient réellement que fasse le système" (1989, p. 273). Frederick P. Brooks estime "qu'il est en fait impossible aux clients, même assistés d'ingénieurs logiciels, de spécifier correctement, précisément et complètement les détails d'un système logiciel moderne avant d'avoir construit et essayé au moins une version du produit qu'ils spécifient" (1996, p. 172). Le constat de cet auteur est tranché : "il faut bien le dire, les clients ne savent pas ce qu'ils veulent. Ils ne savent pas en général quelles questions il faut poser, et ils n'ont souvent jamais pensé au problème avec le niveau de détail qu'il faut pour écrire les spécifications. Même les réponses simples ("faites marcher le nouveau logiciel comme notre ancien système de traitement manuel") sont en fait trop simples. Les clients ne

[103] Parfois également appelé *Capacity Maturity Model for Software* (CMM).

veulent jamais dupliquer exactement l'existant. En outre, les systèmes logiciels complexes sont des choses qui agissent, qui bougent, qui travaillent. La dynamique de ces actions est difficile à imaginer" (idem).

Le problème, analysé par Faïz Gallouj (1994, p. 194 et suivantes) dans le cas plus général des activités de services, est celui de l'écart existant entre le besoin "réel" de l'utilisateur à partir duquel il va évaluer la qualité de la prestation fournie, et le besoin "exprimé" par l'utilisateur et "perçu" ou "ressenti" par le prestataire qui va déterminer la prestation réalisée. Comme le note Serge Bouchy, dans le cas des logiciels il est "toujours difficile de faire formaliser clairement par les utilisateurs finals (…) l'expression de la qualité explicite" (1994, p. 179). Pour étudier les différents agencements organisationnels, Jacques Girin souligne l'asymétrie fondamentale dans une relation de mandat, entre le mandant qui "dit" et le mandataire qui "fait" (1994, p. 18). Jacques Girin distingue quatre situations possibles selon que le "dire du mandant" (ou "mandement") est clair ou confus, et selon que l'activité du mandataire est simple (facile à décrire) ou complexe (difficile à décrire). On peut considérer que dans l'activité de productions de logiciels, on se trouve le plus souvent dans la situation la plus problématique, d'un mandat confus et d'une activité complexe, où "le mandataire ne maîtrise pas tous les éléments conditionnant la réussite de la prestation, laquelle dépend aussi de la coopération et des capacités du mandant" (idem, p. 20).

La complexité de l'activité fait qu'il est souvent difficile pour le mandant de comprendre que "de petites variations de la description du système peuvent engendrer de grandes variations de l'effort de développement et/ou des ressources nécessaires à l'exécution du système" (Jacques Printz, 1998, p. 190). Les utilisateurs ne soupçonnent pas notamment les conséquences que peuvent avoir des exigences accrues en termes de "convivialité" des applications (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 16)[104]. Maarten Boasson note que "les maîtres d'ouvrage, faute d'une bonne compréhension, peuvent avoir des attentes des plus étranges" (1998, p. 7) et "qu'expliquer à un client qu'il rédige des exigences fondamentalement impossibles est embarrassant et n'est pas l'usage dans ces métiers" (idem, p. 6). Pierre Lévy souligne que "non seulement le but à atteindre n'est pas parfaitement défini

[104] Symétriquement, mais moins fréquemment, des demandes peuvent ne pas être formulées par des utilisateurs soucieux des coûts et des délais de réalisation de l'application, alors que leur satisfaction demanderait peu d'efforts de développement.

au départ, mais [qu'] il subit en cours de route nombre de remaniements essentiels" (1992, p. 44) [105].

Indépendamment de l'évolution des besoins pendant le processus de développement du logiciel, c'est surtout pendant sa phase d'utilisation que les besoins des utilisateurs se modifient. Ces besoins évoluent en fonction des nouvelles possibilités qu'apportent les évolutions rapides du matériel. Dans le domaine des logiciels, un produit paraît obsolète dès son achèvement, voire même avant (Frederick P. Brooks, 1996, p. 8). Certes, cette obsolescence relative concerne plus souvent des innovations qui sont plus au stade de l'idée nouvelle que de la réalisation effective, mais les demandes de modifications déstabilisent des applications qui avaient été implantées avec difficulté (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 17). Evidemment, les demandes de modifications existent également pour des produits matériels, "mais l'existence même d'un objet tangible sert à canaliser et à quantiser les demandes de changement de l'utilisateur, tandis que la souplesse et l'invisibilité d'un produit logiciel exposent ses constructeurs à d'incessants changements d'exigences" (Frederick P. Brooks, 1996, p. 99). Enfin au fur et à mesure de l'utilisation d'un logiciel, les caractéristiques auxquelles l'utilisateur accorde le plus d'importance évoluent : au départ l'utilisateur peut être particulièrement sensible à la simplicité d'utilisation d'un nouveau logiciel, alors qu'une fois la période d'apprentissage réalisée ce sont l'efficacité du produit et la richesse de ces fonctionnalités qui peuvent lui sembler décisives. Le logiciel ne peut s'adapter aux évolutions des compétences de ses utilisateurs : un logiciel qui explique dans le détail les opérations à effectuer peut-être plus facile à appréhender mais devenir rapidement fastidieux à utiliser.

Des caractéristiques contradictoires

Le problème est que les différentes caractéristiques d'un logiciel sont relativement contradictoires : par exemple, un logiciel écrit en langage Java est doté d'une très grande portabilité[106] mais est estimé deux à vingt fois moins rapide à l'exécution qu'un logiciel écrit

[105] Capers Jones, à partir de l'étude des devis de 64 projets effectués par IBM au début des années soixante-dix, constate que 35 % des fonctionnalités réalisées ne figuraient pas dans les spécifications initiales mais avaient été ajoutées après coup sur demande des clients (1989, p. 261-262).

[106] Il peut s'exécuter sur n'importe quelle machine disposant d'un petit programme d'interprétation de ce langage (machine virtuelle Java), ce qui explique son utilisation pour des applications stockées sur des serveurs et exécutées sur l'ordinateur de l'utilisateur lors d'une connexion Internet.

en C++ (Eurostaf, 1997 A, p. 26). Les compromis nécessaires entre les différents facteurs de qualité sont d'autant plus difficiles à réaliser que le logiciel est destiné à des utilisateurs hétérogènes. Il est en effet extrêmement rare qu'un logiciel soit développé pour un seul utilisateur. Les applications sur mesure sont destinées le plus souvent à des utilisateurs multiples et les progiciels sont par définition des produits standards destinés à des utilisateurs anonymes. Dans ce cas, le producteur de logiciel répond à un besoin "abstrait", c'est à dire le "besoin du marché dans son ensemble" (Faïz Gallouj, 1994, p. 195). L'utilisateur est un être abstrait qui recouvre une grande diversité d'utilisateurs concrets, alors même que le programmeur doit "inscrire dans un scénario fixé à l'avance les acteurs qui vont assumer le rôle de l'utilisateur, car l'usage d'un logiciel ne peut se dissocier d'une série de prescriptions plus ou moins naturelles, conviviales ou ergonomiques, mais toujours contraignantes" (Pierre Lévy, 1992, p. 23). "Un produit logiciel propre et élégant doit présenter *à chaque utilisateur* un modèle cohérent de l'application, des stratégies employées pour effectuer la tâche applicative, et des tactiques à appliquer pour spécifier les actions et leurs paramètres à l'aide de l'interface utilisateur" (Frederick P. Brooks, 1996, p. 223). Le problème est que "pour un produit logiciel donné, chaque attribut de l'ensemble des utilisateurs suit en fait une distribution, ayant de nombreuses valeurs différentes, chaque valeur ayant sa propre fréquence" (idem, p. 226). Dans ces conditions, il est tentant de surcharger le logiciel de fonctionnalités diverses (ce qui explique que ces fonctionnalités soient en moyenne faiblement utilisées), mais ce qui risque de nuire à l'efficacité du logiciel et à sa simplicité d'utilisation[107].

Une autre explication de l'insatisfaction des utilisateurs est que dans de nombreux cas ce ne sont pas eux qui sont les clients (acheteurs des produits ou donneurs d'ordres) des producteurs de logiciels. Cette situation d'une relation triangulaire entre producteurs, clients et utilisateurs n'est évidemment pas spécifique aux logiciels, mais elle est plus fréquente pour les produits informatiques, les services informatiques des organisations ayant souvent acquis un pouvoir de décision particulièrement important. On peut ajouter que les intérêts et les critères

[107] Roberto Di Cosmo et Dominique Nora (1998, p. 54) parlent "d'obésiciels" pour désigner les logiciels produits par Microsoft, qui au fil des versions successives se sont alourdis de multiples fonctionnalités peu utilisées. Microsoft fait toutefois remarquer que ces progiciels étant utilisés par des millions de personnes différentes, si les utilisateurs n'utilisent effectivement qu'une faible partie des possibilités, celle-ci est différente selon les utilisateurs. Ajoutons que l'importance des économies d'échelle fait qu'il est beaucoup plus rentable (du moins pour le moment) de produire un seul progiciel nécessairement sous-utilisé par grand type de besoin, que de le décliner en des versions différentes adaptées à des segments d'utilisateurs.

de choix de ces services sont certainement plus divergents de ceux des utilisateurs pour les produits informatiques, que pour les autres produits et services consommés par une organisation.

Les logiciels, boucs émissaires des mécontentements ?

Enfin, il faut prendre en compte le fait qu'une partie des reproches formulés par les utilisateurs aux logiciels le sont à tort. Il s'agit tout d'abord des cas où les disfonctionnements observés proviennent en fait des compétences et de la formation insuffisantes des utilisateurs. Nous avons déjà signalé l'importance du rôle des futurs utilisateurs dans le développement des logiciels sur mesure, où on peut considérer qu'existe un processus de co-production, mais le savoir-faire et le savoir-utiliser des utilisateurs déterminent également la part plus ou moins importante des potentialités du logiciel qu'ils pourront exploiter. Il faut notamment prendre en compte les exigences particulières à un traitement automatique, d'une extrême formalisation de l'ensemble des données introduites lors du processus d'utilisation. Le processus d'apprentissage, qui dépasse la seule dimension technologique, est plus délicat à mettre en œuvre quand l'utilisation du logiciel est imposée aux utilisateurs, que cette imposition soit le fait de la direction d'une entreprise vis à vis de ses salariés, ou d'une entreprise à ses sous-traitants[108]. Jacques Printz estime qu'il faudrait pouvoir calculer "un indice de satisfaction qui fasse la part des choses entre ce qui est la responsabilité de l'usager et celle du fournisseur de système", ce qui n'est pas simple à réaliser car il faut pouvoir "observer le système en des points qui ont à la fois un sens pour l'usager (où tout est sémantique et pragmatique) et pour le concepteur de système (où tout est structure et syntaxe)" (1998, p. 288-289).

Dans un certain nombre de cas, l'insatisfaction concerne en réalité les réorganisations de l'activité opérées à l'occasion de l'introduction d'un logiciel, le logiciel ne faisant que "traduire" cette réorganisation, qui est en quelque sorte "neutralisée"[109]. En effet,

---

[108] Ramon Salvador Valles, Lucas Van Wunnik, et Félix Pineda, montrent à propos des effets de l'échange de données informatisées (EDI) dans le secteur automobile, que quand cette introduction est imposée par de grandes entreprises aux PME sous-traitantes, l'apprentissage se limite à la maîtrise technologique des outils et n'intègre pas la composante organisationnelle de l'apprentissage, ce qui limite l'obtention des bénéfices potentiels de cette technologie (1997, p. 140).

[109] Ce phénomène s'apparente aux actions de résistance par rapport à l'introduction des machines, quand les salariés les jugent responsables de leur déqualification. Comme le fait observer, Harry M. Collins (1992), "ce ne sont pas les machines qui causent la déqualification, mais l'organisation du travail" (p. 156). Pour que la tâche effectuée par un opérateur humain puisse être réalisée par une machine, il faut "d'abord redéfinir le travail de telle sorte que celui-ci puisse être effectué de façon machinique" (en imitant les machines dans nos actes), et "c'est là qu'intervient le processus de déqualification" (idem, p. 289).

l'informatisation peut permettre de faire accepter des bouleversements organisationnels, qui n'auraient pas été acceptés en son absence (François Pichault, 1990, p. 99). Il faut également prendre en compte le fait que la modélisation systématique des organisations existantes effectuée lors de la conception d'un logiciel conduit à repérer les nombreuses incohérences existantes, qui résultent de la construction incrémentielle, par strates des procédures organisationnelles (Eric Brousseau, 1997, p. 54). La résorption de certaines failles organisationnelles peut conduire à en révéler de nouvelles. Des inefficacités organisationnelles, comme les redondances, peuvent être mises en évidence lors de la création d'un logiciel (Philip E. Agre, 1997, p. 251). La systématisation de la production, de la collecte et du stockage de l'information permet de procéder à des analyses de performances qui peuvent révéler des dysfonctionnements (Eric Brousseau, 1997, p. 56). "Les problèmes d'organisation et de management ainsi révélés sont réels et ne sont pas provoqués par l'informatisation", mais "l'informatique contribue cependant à les mettre en pleine lumière" (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 15). Un problème particulier est le dévoilement de la non-observation en pratique de certaines règles qui devaient être théoriquement appliquées[110], les contraintes du traitement automatique nécessitant l'exposition systématique des pratiques effectives[111]. La conception du logiciel nécessite l'élaboration d'un nouveau système de règles "plus compliqué (…) correspondant à la fois aux anciennes règles, aux pratiques des personnes, aux cas traités à part par la hiérarchie, aux spécificités des ordinateurs, à des souhaits d'évolution du travail, au traitement des erreurs" (Jean-Louis Peaucelle, 1997, p. 29). Ce nouveau système de règles élimine au maximum les "zones d'ombre", dont la sociologie des organisations de Michel Crozier et Ehrard Friedberg (1987) a pourtant mis en évidence l'importance pour que l'organisation puisse fonctionner. Cette théorie, qui analyse le pouvoir des acteurs comme la résultante de l'existence de "zones d'incertitudes", permet de comprendre les oppositions des utilisateurs à une véritable transparence organisationnelle, qui renforce les pouvoirs de la direction générale et/ou des

[110] L'écart entre les pratiques effectives et les règles théoriques peut s'expliquer par une volonté des acteurs d'améliorer leur situation au détriment de l'organisation, mais également par le fait que le fonctionnement effectif de toute organisation est tributaire d'une interprétation des règles avec une certaine souplesse, ce que démontre a contrario l'efficacité des "grèves du zèle" qui consistent à appliquer scrupuleusement les règles existantes.

[111] Un rapport de l'Inspection Générale de l'Administration de l'Education Nationale consacré notamment à l'introduction des logiciels Nabucco (pour la comptabilité) et Apogée (pour la gestion des étudiants) souligne les très grandes libertés que les Universités prenaient avec les textes légaux et qui ont été révélées lors de l'implantation de ces logiciels. Le rapport précise en conclusion que "l'informatisation devrait aider les universités à s'inscrire dans des cadres plus respectueux des règles" (IGAEN, 1998, p. 54).

services informatiques à leur détriment. A partir de ce cadre d'analyse appliqué à l'étude de projets d'informatisation, Francis Pavé (1989) conclut que les projets réussis sont en fait les projets "dévoyés" dans leur utilisation (utilisés dans des perspectives non projetées au départ), car ce sont ceux qu'ont pu se réapproprier les utilisateurs, "les pratique sociales [réussissant] à contourner la transparence organisationnelle" (p. 13).

En appliquant l'analyse de Jacques Girin, on peut dire que la qualité d'un logiciel définie comme étant "sa capacité à réaliser les performances que l'on attend de lui", est la "compétence d'un agencement" entre des "ressources humaines, matérielles et symboliques" (on peut ajouter informationnelles). Cette compétence "réside dans les relations entre ses divers éléments, bien plus que dans les propriétés de chaque élément pris séparément" (Jacques Girin, 1994, p. 27).

3 - La qualité des logiciels, une construction sociale conventionnelle

La qualité d'un logiciel est donc plus une construction sociale faisant intervenir des attentes réciproques entre les producteurs et les utilisateurs, qu'un ensemble de caractéristiques objectives universellement et atemporellement acceptées. Ce constat s'appuie sur l'analyse effectuée par Pierre-Yves Gomez (1994) de l'économie de la qualité. Cette analyse critique la vision de la qualité de la micro-économie standard, selon laquelle la qualité est un simple problème de différenciation des produits : dans un premier temps chaque objet à échanger possède un ensemble de caractéristiques qualitatives qui définissent le champ de leur utilité ; dans un second temps, lorsque la qualité est définie, le jeu classique offre/demande sur les prix (Pierre-Yves Gomez, 1994, p. 57-58). Il reste en effet à expliquer pourquoi et comment telle norme de qualité émerge de préférence à une autre. Le libre jeu du marché, basé sur l'axiome du choix autonome d'individus calculateurs, ne peut suffire à révéler le choix des acteurs, comme l'ont montré différents travaux : dans des situations d'asymétrie d'informations (imperfection de l'information sur le marché avec possibilité d'une connaissance cachée ou *sélection adverse*[112]), la qualité la plus basse peut l'emporter sur la qualité la plus élevée, ce que démontre George A. Akerlof (1970) à partir du marché des voitures d'occasions ; le prix n'est pas systématiquement un bon indicateur de l'information

[112] La sélection adverse (ou anti-sélection) est une situation où un des contractants dispose d'informations pertinentes pour la relation et ignorées de l'autre contactant, ce qui lui permet de bénéficier de rentes informationnelles.

sur les objets échangés (Joseph Stiglitz, 1985), il est plutôt "un signal qui suppose un accord préalable sur la qualité des objets" (Pierre-Yves Gomez, 1994, p. 62), d'où la nécessité d'une répétition des relations entre les acteurs, pour créer la confiance indispensable à l'émergence d'évaluations communes ; l'existence d'une garantie ne suffit pas à résoudre le problème de l'incertitude initiale sur la qualité, car la garantie constitue "un coût et devient donc elle-même un objet de calcul" source de nouvelles incertitudes avec des risques d'action cachée ou *moral hazard* (idem, p. 63-65) ; en raison de comportements opportunistes[113] (par exemple incursions courtes sur des marchés), la libre concurrence peut être source de mauvaise qualité ; même en l'absence de comportements opportunistes, le théorème d'Arrow, qui s'inscrit dans la logique du paradoxe de Condorcet, montre l'impossibilité d'une émergence spontanée de la qualité, fondée sur la totale subjectivité des choix et le libre jeu du marché (idem, p. 67).

La nécessité d'intégrer la confiance et la convergence des représentations sociales implique de considérer la qualité comme une construction sociale, qui n'est pas la conséquence des rapports marchands, mais qui préexiste à l'échange et à la production (Pierre-Yves Gomez, 1994, p. 137). Pour l'étudier, il faut utiliser des "outils d'analyse alternative à l'axiomatique standard en puisant dans le corpus conceptuel produit par la théorie des conventions" (idem, p. 75)[114]. Les conventions ne portent pas directement sur la qualité des objets échangés mais sur le rôle des acteurs de l'échange, le type de comportement attendu de l'offreur et du demandeur. La qualité est définie par cet auteur comme une conjonction entre une convention d'effort et une convention de qualification. La convention de qualification "établit la compétence d'un professionnel", "elle offre une procédure de résolution récurrente de problèmes de détermination de la qualité lors de l'échange, en émettant une information sur les pouvoirs de qualifier qu'il s'agit d'attendre des professionnels d'une part, des clients de l'autre" (idem, p. 145) ; la convention de qualification établit un consensus entre le client et son fournisseur tel que l'un et l'autre savent ce qu'il faut attendre d'une relation marchande, elle opère dans le corps social le partage des compétences entre clients et professionnels. La convention d'effort supporte la réalisation de la qualité et se situe au niveau interne de la firme, la qualité résultant d'un effort (conventionnel) de production. De cette analyse de la

[113] L'opportunisme est une forme de rationalité qui pousse des agents économiques à ne pas respecter un contrat s'ils n'y sont pas contraints et si cela leur permet d'améliorer leur situation personnelle. La sélection adverse et le risque moral sont deux types d'opportunisme.

[114] Nous reviendrons sur le cadre théorique de l'économie des conventions dans le chapitre VII.

qualité, comme doublement contrainte par une convention d'effort et par une convention de qualification, découle logiquement qu'une diminution du niveau de qualité perçue provient des absences de cohérence, soit dans l'une des deux conventions, soit entre les conventions (idem, p. 217).

Cette analyse peut permettre de comprendre à la fois les tentatives de résoudre le problème de la qualité des logiciels par la certification des entreprises (notamment la norme ISO 9000-3 spécifique au logiciel) et l'échec relatif de ces tentatives. En effet, le présupposé de cette norme est qu'il suffit d'un bon processus de développement pour obtenir un "bon" produit et que la qualité du processus de développement est obtenue par une extrême formalisation des procédures mises en œuvre. Comme le souligne, Maarten Boasson, "le développement d'un système complexe a davantage de ressemblance avec la création d'une œuvre d'art qu'avec la simple production, et (…) il n'existe aucune relation démontrable entre le processus de développement et la qualité du produit résultant. L'essence du développement est qu'il y a des compromis à faire et des décisions à prendre qui exigent connaissances, compréhension et, ce qui est malheureusement trop souvent contesté, intuition. L'idée que ces facteurs essentiels pour la qualité du produit final, puisse être éliminés en décrivant minutieusement dans le détail comment le processus de développement doit se dérouler est une erreur de raisonnement fondamentale" (1998, p. 4-5)[115]. Une démarche de type ISO 9000 ne peut répondre que très partiellement aux exigences de qualité des utilisateurs. C'est ce que permet de mettre en évidence l'analyse de Pierre-Yves Gomez qui montre non seulement pourquoi et comment la qualité évolue dans le temps (passage de la "qualité inspection" et de la "qualité contrôle" caractéristiques du taylorisme, à la "qualité assurance" du fordisme et émergence actuellement d'une "qualité totale"), mais également pourquoi des formes de qualités distinctes peuvent coexister, en conséquence de la pluralité des cohérences possibles entre des conventions différentes d'effort et de qualification.

Il reste à expliquer dans le cadre de l'économie des logiciels, pourquoi ces cohérences ont tant de difficultés à émerger. Une première réponse possible a trait aux relations initialement très déséquilibrées entre les informaticiens et les utilisateurs.

[115] Pour un bilan très critique des normes ISO 9000 concernant les logiciels, cf. également Jacques Printz (1998, p. 289-290).

B - UNE RELATION INITIALE DESEQUILIBREE ENTRE INFORMATICIENS ET UTILISATEURS

A l'origine les producteurs de logiciels ont été dans une situation de domination forte des utilisateurs, ce qui a rendu difficile la stabilisation de conventions de qualité.

Cette domination se situait tout d'abord sur un plan technique. Elle était particulièrement présente au début de l'informatique, dans la mesure où les possibilités relativement limitées des premiers ordinateurs et l'absence de programmes utilitaires bien conçus pour gérer ces ressources, faisaient que l'essentiel des efforts des informaticiens portaient sur les aspects les plus proches du matériel au détriment des aspects les plus proches des utilisateurs (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 6). L'intercompréhension était particulièrement difficile, jusqu'au niveau du langage utilisé, entre des informaticiens connaissant mal les caractéristiques spécifiques des besoins des utilisateurs, et des utilisateurs peu au fait des contraintes particulières d'un traitement automatique : "les clients pensent en termes de flux d'air ou de sommes à facturer, alors que les développeurs réfléchissent en termes de structures de données et d'interfaces entre modules. Il peut arriver que le même terme - fichier par exemple - ait un sens différent pour les uns et pour les autres" (Lauren Ruth Wiener, 1994, p. 115). La relation qui s'instaurait entre eux était moins une relation de coproduction, qu'une relation maître-élève. Au nom d'exigences d'ordre technique, les utilisateurs étaient contraints d'accepter les logiciels tels qu'ils avaient été conçus par les informaticiens et de s'y adapter. Cette situation s'est pérennisée en raison de l'évolution accélérée des techniques, sans qu'il y ait en même temps de "véritable progression de la maîtrise technologique" (Serge Bouchy, 1994, p. 172).

Ces impératifs techniques sont renforcés par la culture spécifique des informaticiens : "beaucoup d'informaticiens de métier aiment suffisamment l'informatique pour construire d'abord des programmes qui sont un prétexte à une relation de qualité avec l'ordinateur plutôt que des outils véritablement utilisables par l'environnement. Quelle entreprise ne résonne pas des plaintes de ceux à qui l'on propose des programmes parfaitement "esthétiques" d'un point de vue informatique, mais assez peu conforme au simple « outil » qu'attendent les utilisateurs" (Philippe Breton, 1990, p. 46). Les informaticiens accordent souvent plus d'importance aux caractéristiques techniques des logiciels produits qu'à leurs caractéristiques d'usage. Jacques Printz estime que "les programmeurs ont souvent tendance à fabriquer de la complexité sans véritable justification économique, car c'est une façon de montrer leur maîtrise logique"

(1998, p. 83). Philippe Breton considère même "qu'aucun développeur de logiciels qui se respecte, ne peut construire un programme qui ne fasse appel à un minimum "d'esprit informatique" de la part de celui qui l'utilise" (1990, p. 85)[116]. "Jusqu'à une période récente, la pénurie apparente en matière de compétences informatiques d'une part, et le côté pionnier de cette activité d'autre part" (Serge Bouchy, 1994, p. 174) faisaient accepter cette situation.

Sur le plan interne aux organisations, les informaticiens ont un pouvoir important. Dans leur capacité à "déboguer" les programmes, ce pouvoir s'apparente à celui des ouvriers d'entretien analysé par Michel Crozier dans le monopole industriel. Le pouvoir des informaticiens est à son apogée lors de la première période de l'informatique, caractérisée par de grands systèmes centralisés, selon le modèle qu'IBM a développé chez ces clients (Claire Charbit, Jean-Benoît Zimmermann, 1997, p. 6). L'introduction de ces systèmes implique fréquemment une refonte des structures organisationnelles, dans laquelle l'avis des informaticiens est déterminant[117]. Initialement techniques, les compétences des informaticiens deviennent sociales, les utilisateurs devant négocier avec eux la forme et souvent le contenu de chaque information qui circule dans l'entreprise (Philippe Breton, 1987, p. 208). Dans le même temps, les utilisateurs voient leur propre pouvoir diminuer, en raison de la dépossession des informations qui résulte de leur centralisation, ce que Jean-Marie Desaintquentin et Bernard Sauteur appelle un "hold-up sur les fichiers" (1991, p. 10). Les nouveaux services informatiques deviennent de plus en plus proche de la direction générale de l'entreprise, qui se contente le plus souvent d'entériner leurs choix, en étant incapable de les apprécier par elle-même (idem, p. 17). Ce règne des informaticiens correspond toutefois moins à une volonté de prise de pouvoir de salariés experts, qu'aux conséquences du désir des directions - très empreint de rationalité technicienne - de transférer les mécanismes décisionnels aux machines (Philippe Breton, 1987, p. 207)[118].

---

[116] Cette conception élitiste de l'utilisation de l'informatique est illustrée par le véritable mépris de la communauté informatique dans les années soixante et soixante-dix, pour les informaticiens qui travaillaient sur les interfaces hommes-machines, selon Nicholas Negroponte (1995, p. 116).

[117] Jean-Marie Desaintquentin et Bernard Sauteur soulignent le rôle de ces ingénieurs technico-commerciaux, délégués par les constructeurs de matériel informatique, qui ne connaissaient rien aux métiers de la banque, de l'assurance ou de l'industrie, et qui se retrouvaient installés aux commandes du système d'information de ces entreprises (1991, p. 6).

[118] De même, Harry M. Collins constate "l'aspect séduisant qu'ont en commun la bureaucratie et la tyrannie des machines : (…) tous les doutes, et la responsabilité qui y est liée de devoir prendre des décisions dans des conditions d'incertitude, ont été levés" (1992, p. 202).

L'externalisation éventuelle du développement des applications ne change pas significativement la situation, du moins durant les premières périodes de l'informatique. En effet, globalement le marché est dominé par l'offre. L'existence de systèmes "propriétaires" limite singulièrement la concurrence et place les clients dans une situation de dépendance vis à vis des fournisseurs (*customer lock-in*), avec des coûts de transfert très élevés pour une activité qui acquiert un caractère de plus en plus stratégique. L'asymétrie d'informations entre le mandant et le mandataire, liée à la faible substituabilité entre fournisseurs, augmentent les risques de comportements opportunistes tirant partie des failles de contrats nécessairement incomplets (Jacques Girin, 1994, p. 10). De plus les relations avec les fournisseurs externes restent le plus souvent l'apanage des services informatiques internes, qui peuvent se sentir plus solidaires de ces fournisseurs, dont ils partagent la culture, que de leur propre organisation (idem, p. 24).

Toutefois les relations entre les informaticiens et les utilisateurs vont progressivement se rééquilibrer sous l'influence de plusieurs facteurs. Tout d'abord la substitution d'architectures ouvertes aux systèmes "propriétaires" introduit une plus grande concurrence, sur des marchés moins dominés par l'offre et qui ont déjà connu une période de récession. Ensuite et surtout, l'irruption de la micro-informatique introduit de profonds bouleversements qui vont avoir des conséquences sur l'ensemble de l'informatique. Une part croissante des dépenses informatiques est effectuée directement par les services utilisateurs[119]. La diffusion des micro-ordinateurs permet une démystification et une plus grande appropriation sociale de l'informatique par les utilisateurs, qui se montrent beaucoup plus exigeants, notamment en termes de convivialité des applications utilisées, ce que facilite la formidable augmentation des capacités de traitement et de mémorisation du matériel. En même temps, l'augmentation des compétences des utilisateurs les rend plus sensibles aux exigences particulières d'un traitement automatique, et plus réalistes dans la formulation de leurs besoins. Certes, plus récemment la mise en réseau progressive de l'ensemble des ressources informatiques des organisations permet une certaine reprise en main par les services informatiques, pour garantir la cohérence globale du système d'information, mais celle-ci doit bien d'avantage tenir compte des souhaits d'utilisateurs qui sont devenus plus autonomes. "L'époque où l'esthétique

[119] En Europe, la part des dépenses informatiques effectuées par les directions informatiques passe de 70 % en 1993 à 58,3 % en 1995, suivant la tendance qu'avaient connu les Etats-Unis (Eurostaf, 1996 A, p. 210).

technique d'une solution suffisait à la justifier est définitivement révolue" (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 56).

C - DES PROGRES REELS MAIS UNE INSATISFACTION MAINTENUE DES UTILISATEURS

Le rééquilibrage des relations entre informaticiens et utilisateurs est une des causes des améliorations qualitatives incontestables des logiciels (1). Toutefois, la diversité croissante des types de logiciels, de leurs utilisateurs et les possibilités de comportements opportunistes rendent difficiles la stabilisation de conventions de qualité (2).

1 - Amélioration de la "convivialité" et nouvelles méthodes de développement

Ces améliorations qualitatives concernent principalement l'augmentation de la "convivialité" des applications et une meilleure perception des besoins des utilisateurs grâce à l'apparition de nouvelles méthodes de développement.

Un outil peut être défini comme convivial, lorsqu'il est mis "au service de la personne intégrée à la collectivité et non au service d'un corps de spécialistes" (Ivan Illich, 1973, p. 13). Une première amélioration importante résulte du passage du modèle "batch" au modèle "conversationnel", devenu le modèle dominant avec l'apparition des terminaux "intelligents", puis des micro-ordinateurs, et le développement des architectures clients-serveurs. Dans le modèle "batch" (ou traitements par lots), l'utilisateur soumet à l'ordinateur un ensemble de travaux à effectuer, les données résultant des traitements effectués par l'ordinateur lui parvenant globalement quelque temps après, sans que l'utilisateur ait la possibilité d'intervenir pendant les traitements. Par contre dans le modèle "conversationnel" (temps réel et transactionnel), les interactions sont permanentes entre l'ordinateur et l'utilisateur, celui-ci pouvant ajuster continuellement les demandes effectuées à l'ordinateur au vu des résultats délivrés quasi-instantanément par la machine. Il faut souligner que la plus grande souplesse qu'apporte ce mode de traitement à l'utilisateur, a nécessité une importante augmentation de la complexité des logiciels pour maintenir la fiabilité et la performance des traitements, et l'intégrité des données (Jacques Printz, 1998, p. 198-200).

La deuxième amélioration, qui complète la précédente, est l'amélioration des interfaces entre l'utilisateur et l'ordinateur, avec l'apparition de l'interface graphique. Les interfaces graphiques résultent des travaux effectués par Xerox au PARC, et ont été intégrées dans un

produit commercialisé par Apple avec le MacIntosh dans les années quatre-vingt, avant de se généraliser à l'ensemble des ordinateurs. Elles reposent principalement sur la substitution d'un écran graphique (écran *bit-map*) au traditionnel écran alphanumérique et l'utilisation d'un dispositif de pointage (principalement la souris), qui facilitent considérablement les communications avec le système informatique par l'utilisation de symboles graphiques (icônes), des menus déroulants, et par le réalisme des représentations dans des "fenêtres" présentes sur un "bureau". Il est significatif, de la conception des informaticiens des compétences que devait posséder un utilisateur, que la souris et les fenêtres furent originellement conçues pour les handicapés mentaux incapables de verbaliser leurs requêtes (Emmanuel Saint-James, 1993, p. 8). Une illustration des possibilités que pouvaient apporter des logiciels de ce type, est le succès du tableur, qui fut un des éléments déterminants dans l'avènement de la micro-informatique. Le tableur n'oblige pas l'utilisateur "à s'engager dans des processus incompréhensibles mais va droit au but et leur donne un accès immédiat à la puissance des machines à partir d'une formulation de leurs problèmes en termes simples : les termes mêmes qu'ils employaient avant que l'outil informatique leur soit proposé" (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 25). Cependant les améliorations ergonomiques des logiciels, qui nécessitent la consommation de ressources matérielles plus abondantes et moins coûteuses, entraînent également une augmentation de la complexité des logiciels (programmation "événementielle").

Un autre domaine qui a permis des améliorations qualitatives des logiciels est l'apparition de nouvelles méthodes de développement. Les méthodes traditionnelles basées sur une première phase d'écriture de spécifications détaillées devant rester intangibles durant tout le processus de développement du logiciel, processus qui peut être long, se sont révélées relativement inadaptées pour appréhender les besoins réels des utilisateurs, tels qu'ils s'expriment lors de l'emploi du logiciel. Effectuer des revues et relectures de documents volumineux et rébarbatifs pour considérer que les spécifications d'un logiciel sont acceptées par les utilisateurs, est insuffisant. En effet, d'une part les besoins évoluent, et d'autre part, il est souvent difficile pour des futurs utilisateurs, dont la motivation peut être variable, d'exprimer abstraitement leurs besoins[120]. Le prototypage rapide, les méthodes de

[120] Par exemple, en étudiant le développement d'un grand système d'informations (4000 hommes*mois), le projet SIMAT (Système d'Informations de la Maintenance de l'Armée de Terre), Jean-Paul Hamon constate que les besoins exprimés en phase de "définition des besoins" sont entachés de 30 % d'incertitude (1996, p. 18).

développement incrémentales et la conception participative[121], qui donnent la possibilité aux utilisateurs d'apprécier concrètement les futures caractéristiques du logiciel et de réagir durant le processus de développement, peuvent permettre de mieux répondre aux besoins réels des utilisateurs. Les technologies objet, notamment dans les possibilités de constituer des "objets-métiers" caractéristiques des activités des utilisateurs, peuvent faciliter le dialogue et l'intercompréhension entre concepteurs et utilisateurs des applications[122].

2 - Les difficultés pour stabiliser des conventions de qualité

Malgré ces progrès qualitatifs indéniables, il existe une insatisfaction maintenue des utilisateurs. Une des explications de ce paradoxe nous semble résider dans les difficultés pour stabiliser des conventions de qualité. On peut ajouter que dans certains cas, les améliorations observées rendent même encore plus difficile l'émergence de telles conventions.

Tout d'abord il faut prendre en compte l'élargissement continu et la diversité croissante des utilisateurs de logiciels. En même temps que les compétences des anciens utilisateurs augmentent en raison des phénomènes d'apprentissage par l'usage (*learning by using*), apparaissent de nouveaux utilisateurs inexpérimentés. Quand c'est le même logiciel qui est destiné à être employé par des utilisateurs très hétérogènes (grand système impliquant des milliers d'utilisateurs ou progiciel vendu à d'innombrables clients), il devient difficile de réaliser un produit capable de répondre simultanément à des exigences différentes. Paradoxalement, l'amélioration des interfaces entre l'ordinateur et l'utilisateur accentue ce problème, dans la mesure où la conception de telles interfaces nécessite des présuppositions fortes sur le comportement de l'usager, en raison notamment de l'importance de ce qui est considéré comme implicite par le programme lors d'une action explicite de l'utilisateur. En effet, demander à l'utilisateur d'expliciter l'ensemble des paramètres de sa requête rend celle-ci rapidement très laborieuse à construire. Une autre conséquence de l'amélioration des

[121] Ces méthodes ont été présentées dans le chapitre III.

[122] Par exemple la technique des cas d'utilisation, introduits par Ivar Jacobson (1993). Cette technique détermine les besoins fonctionnels selon le point de vue d'une catégorie d'utilisateurs à la fois. Facilement compréhensibles par les utilisateurs potentiels du système, les cas d'utilisation permettent de valider les fonctionnalités à l'aide de scénarios. La réalisation des cas d'utilisations est effectuée par une société d'objets collaborant, à partir des objets du domaine et de leurs relations.

interfaces est qu'elles peuvent contribuer à faire prêter inconsciemment à l'ordinateur des fonctions de bon sens dont il est parfaitement incapable (Jacques Printz, 1998, p. 39)[123].

La deuxième difficulté pour stabiliser des conventions de qualité réside dans l'apparition continue de logiciels répondant (au départ souvent imparfaitement) à de nouveaux besoins. Ces logiciels correspondent le plus souvent à des projets de plus en plus importants, qui ne peuvent être réalisés qu'en répartissant le travail entre des sous-équipes, avec le risque que chaque sous-équipe optimise mal sa partie pour atteindre ses objectifs, au détriment des qualités globales du produit (Frederick P. Brooks, 1996, p. 207). Enfin, pour un logiciel d'un type donné, de nouvelles versions sont mises sur le marché à un rythme extrêmement rapide, en raison des facilités de modification d'un produit existant sous forme numérique et des intérêts des producteurs. Ceci a pour conséquence une "obsolescence rapide des connaissances" des utilisateurs, qui "rend épuisante la poursuite d'un niveau élevé de maîtrise" (Jean-Pierre Faguer, Michel Gollac, 1997, p.115), l'apprentissage de l'utilisation d'une nouvelle version nécessitant des désapprentissages, source de confusions ou d'erreurs.

Ceci nous conduit à la troisième explication qui est la facilité pour les producteurs d'adopter des comportements opportunistes, concernant certaines dimensions qualitatives qui ne peuvent être perçues par les consommateurs, notamment *ex ante* lors de l'achat du produit. Il importe de distinguer de ce point de vue le cas des logiciels développés sur-mesure du cas des progiciels. Pour un logiciel sur mesure, le produit (et donc *a fortiori* ses caractéristiques qualitatives) n'existe pas au moment de la signature du contrat. Le contrat ne peut porter que sur des exigences, mais dont la "quantification est, dans la plupart des cas, soit difficile, soit invérifiable en pratique" (Philippe Robert, 1997, p. 169). De plus, "on ne peut pas démontrer, le plus souvent, que la satisfaction des exigences implique celle des besoins" (idem). Cette situation d'incomplétude des contrats est encore plus importante avec l'utilisation des nouvelles méthodes de développement (cf. supra) où les spécifications détaillées du logiciel sont élaborées progressivement lors du processus de développement du logiciel par des interactions continues entre les concepteurs et les utilisateurs. Si ces méthodes permettent de réduire l'écart entre les besoins latents des utilisateurs et les besoins perçus et exprimés par les

[123] Une étude fine des conséquences de la diversité des utilisateurs et de leurs compétences par rapport à l'utilisation d'un logiciel, un automate point de vente de billets SNCF, a été réalisée par Marc Breviglieri (1997). Sur l'absence de "bon sens" de tels programmes, on peut mentionner la recherche des trains en partance de Lille, depuis l'existence d'une seconde gare TGV distante d'une cinquantaine de mètres de la précédente. Dans certains cas, le programme ne propose que les trains en partance d'une des deux gares !

concepteurs, elles empêchent par définition l'inscription des spécifications précises dans le contrat initial. De ce fait, *une part importante des relations entre les producteurs de logiciels et leurs clients ne peut être basée que sur une confiance réciproque*. Celle-ci peut être particulièrement difficile à établir, et des incompréhensions successives peuvent rapidement détériorer les relations entre les partenaires comme l'atteste la fréquence des recours juridiques dans le domaine des logiciels sur mesure. La tentation peut être grande pour les producteurs de privilégier la productivité (permettant de maîtriser les délais et les coûts) au détriment de la qualité. Certes, certaines études estiment que des exigences accrues de qualité peuvent influencer positivement la productivité des développeurs en augmentant leur satisfaction personnelle et leur motivation[124], mais il ne peut s'agir que de la qualité telle qu'elle est perçue par les développeurs, dont on a vu qu'elle pouvait reposer sur des critères différents de ceux des utilisateurs.

La situation est différente pour les progiciels où le produit existe au moment de son achat par un client, même s'il ne lui est pas toujours possible de connaître ses qualités réelles avant de l'avoir utilisé, notamment en raison du renouvellement très rapide des produits. Surtout l'importance des rendements croissants d'adoption, et notamment des externalités de réseaux, permet d'expliquer comment des progiciels, indépendamment de leurs qualités peuvent s'imposer sur un marché[125]. De ce fait les stratégies des producteurs de progiciels peuvent ne pas accorder à la qualité du produit l'importance souhaitée par les utilisateurs.

Indépendamment des comportements des producteurs, les difficultés pour répondre aux besoins des utilisateurs s'expliquent également par des caractéristiques spécifiques des logiciels, tant au niveau de leur conception que de leur utilisation.

---

[124] Toran Demarco, Timothy Lister estiment même que "des produits d'une qualité bien supérieure à celle qu'accepte l'utilisateur final sont synonymes de meilleure productivité" (1991, p. 31). En effet, pour un programmeur "la nécessité où il se trouve d'avoir à livrer un produit nettement inférieur à ce qu'il sait pouvoir réaliser est une atteinte à son amour-propre et compromet toute satisfaction personnelle" (idem, p. 146). Réaliser des produits de piètre qualité, pour répondre aux contraintes de délais et aux exigences immédiates des clients a des effets négatifs en augmentant l'insatisfaction des développeurs sur leur comportement et leur efficacité. Ces auteurs citent les exemples au Japon de certaines firmes (Hitachi Software, certains secteurs de Fujitsu) où "l'équipe des réalisateurs a le pouvoir effectif de s'opposer à la livraison d'un produit qu'elle estime ne pas être vraiment prêt" même si "le client est d'accord pour l'accepter en l'état" (idem, p. 33).

[125] Ces mécanismes seront décrits précisément dans le chapitre VII.

D - LES DIFFICULTES DUES AUX SPECIFICITES DU LOGICIEL

Au niveau de la conception, le problème provient de l'écart inévitable entre une modélisation (point de passage obligé pour concevoir un logiciel) et la partie de la réalité à modéliser (1). Au niveau de l'utilisation, il s'agit des problèmes de "communication" entre un logiciel et ses utilisateurs (2). Certes, ces problèmes existent pour beaucoup d'autres artefacts, dont la conception repose sur une modélisation et dont l'utilisation nécessite des interactions avec des êtres humains, mais les particularités du logiciel font qu'ils prennent dans ce cas une importance particulière.

1 - L'écart entre la modélisation et la réalité

La conception d'un logiciel passe nécessairement par une phase de modélisation[126] de la réalité concernée, qu'elle soit physique, sociale, psychologique ou institutionnelle. C'est à partir de cette modélisation que le problème à résoudre sera traduit en instructions exécutables par un ordinateur au cours du développement du logiciel. Une modélisation constitue une abstraction basée sur une rationalité de type logico-mathématique ; par définition, elle réalise une simplification de la réalité concrète, elle ne peut être qu'un décalque imparfait et incomplet de celle-ci.

Une modélisation destinée à être traduite en un programme informatique a des exigences particulières, quasiment contradictoires avec cette définition, dans la mesure où elle n'est pas simplement destinée à représenter un phénomène pour l'expliquer, mais à réaliser des traitements automatiques. Elle devrait donc être exhaustive (en n'omettant aucun élément pouvant être pertinent), totalement déterministe et ne tolérer aucune ambiguïté pouvant induire plus d'une interprétation possible. Or la réalité, notamment sociale, ne peut être connue parfaitement et complètement, et recèle toujours une part importante d'ambiguïté et d'interprétation pour que les organisations puissent fonctionner réellement[127]. La modélisation

[126] Le terme de modélisation "recouvre un vaste champ de significations, depuis la mathématisation d'un processus jusqu'au choix des données susceptibles de constituer a priori toute l'information nécessaire et suffisante pour représenter l'activité d'une partie de l'entreprise" (Jean-Louis Peaucelle, 1997, p. 22).

[127] "L'indéterminisme, ou l'ambiguïté, est utile dans de nombreuses situations humaines et se prête très bien au flou de certaines situations sémantiques" (Jacques Printz, 1998, p. 345).

est donc nécessairement "imparfaite"[128] parce qu'il ne peut exister "un ensemble parfait de règles [qui] aurait à couvrir toutes les situations possibles (…) [et qui] constituerait une description complète du passé et de tous les futurs possibles" (Harry M. Collins, 1992, p. 130). Il faudrait, pour les règles gouvernant le comportement humain, pouvoir connaître "des règles prescrivant comment agir en toute circonstance concevable, sinon encore conçue, et de règles dans les règles pour leur dire ce qu'il faut rechercher pour identifier ces circonstances, et de règles dans les règles pour les aider à identifier les choses qui doivent les aider à identifier les circonstances, et ainsi de suite *ad infinitum*" (idem, p. 131).

Pour que le logiciel résultant de cette modélisation puisse néanmoins être opérationnel, il importe que les utilisateurs assimilent les inévitables simplifications qui ont été réalisées par les concepteurs, et que ces simplifications soient pertinentes par rapport à la réalité de l'activité informatisée. "Tout acte de programmation projette, en fait, à travers le programme dont il est le résultat, une certaine vision logique de la réalité telle qu'elle est perçue par le programmeur ou le concepteur" (Jacques Printz, 1998, p. 100). La qualité de la réponse apportée par le logiciel aux besoins des utilisateurs dépend donc de manière importante de la qualité de l'intercompréhension entre concepteurs et utilisateurs.

Des difficultés communicationnelles

Une première difficulté est d'ordre communicationnel. Nous avons déjà mentionné l'absence de solution satisfaisante pour énoncer les spécifications à la base de la modélisation : rédigées en langage naturel elles contiennent une foule d'ambiguïtés ; formulées dans un langage formel de spécifications elles sont incompréhensibles pour les utilisateurs et les experts du domaine[129]. Plus généralement, Lauren Ruth Wiener note que "la communication qui se déroule entre clients et développeurs est profondément viciée", car "ces deux groupes de personnes vivent dans des mondes qui n'ont rien à voir " (1994, p. 115). Toran Demarco et Timothy Lister soulignent à juste titre que la "sphère d'activité [des concepteurs de logiciels] met en jeu la capacité à la communication inter-humaine plus que la

[128] "Enormément de projets échouent précisément à cause d'aspects qui n'avaient jamais été définis" (V.A. Vyssotsky, 1972).

[129] Lauren Ruth Wiener cite l'exemple de l'entreprise Telectronics Pacing Systems qui développe des logiciels pour des défibrillateurs cardiaques. Vu l'importance des problèmes de fiabilité, cette société entreprit de faire écrire les spécifications en langage formel, mais de ce fait les experts cardiologues ne furent plus en mesure d'affirmer si telle ou telle spécification devenue pour eux incompréhensible, était correcte ou totalement aberrante (1994, p. 144).

capacité à communiquer avec des machines" (1991, p. 112). Cependant, Jacques Printz remarque que "des personnalités à profil très technique comme les programmeurs ont souvent peu de talent, et peu de doigté en matière de communication" (1998, p. 83). La communication est d'autant plus difficile que "l'environnement dans lequel "fonctionne" un programmeur, et *a fortiori* un concepteur de systèmes informatiques est probablement parmi les plus compliqués que l'on puisse rencontrer dans les métiers d'ingénierie, complication aggravée par le côté non matériel et purement abstrait des entités manipulées qui fait que le non expert n'y comprend rien" (idem, p. 86). Dans toute communication, l'implicite occupe une place importante, notamment dans l'utilisation des métaphores et des analogies, rendant plus difficile la compréhension entre des membres de groupes ayant des références culturelles (techniques et professionnelles) différentes[130].

Des difficultés cognitives

A ces difficultés communicationnelles, s'ajoute l'écart entre ce que les individus disent ("*espoused theory*", les énoncés des acteurs sur leurs comportements) et ce que les individus font réellement ("*theory in use*", qui gouverne effectivement l'action des gens) (Chris Argyris et Donald A. Schön, 1978). Cet écart s'explique tout d'abord par l'importance, la diversité et la complexité des connaissances tacites (cf. chapitre I)[131] : "il se peut que le savoir livresque soit plus aisément exprimable, précisément parce que sa nature abstraite lui confère une constante et inévitable absence de familiarité. Il nous est en permanence extérieur, ce qui en fait un élément visible en permanence de notre vie mentale" (Harry M. Collins, p. 202-203)[132]. Cet écart s'explique également par la distance existant entre les règles et les procédures formelles, et les routines réellement existantes *(*Jean-Claude Tarondeau, 1998, p. 43). Certes, "toute activité humaine est codifiée par des règles", mais "ces règles ne sont qu'en partie explicites, leur efficacité tient au fait qu'elles ne se dévoilent qu'en situation et qu'elles ne sont jamais

---

[130] Robert Michon et Lin Gingras (1988) parlent du "fossé sémantique" qui empêche une véritable compréhension mutuelle entre concepteur et utilisateurs des systèmes d'information.

[131] "La règle générale est que nous en savons plus que ce que nous pouvons en dire et, si nous réussissons à en savoir plus que ce que nous pouvons en dire, c'est parce que nous apprenons par le bais de notre socialisation et non grâce à l'instruction" (Harry M. Collins, 1992, p. 22).

[132] Capers Jones note que, y compris "dans les activités les plus simples du genre des programmes de saisie de commande ou de paie, il a été maintes fois remarqué que même des praticiens d'expérience, par exemple des personnes employées depuis plusieurs années à l'administration des ventes, ne savaient pas toujours expliquer ce qu'ils faisaient, en termes utilisables à la reprise par la machine des tâches humaines concernées" (1989, p. 133).

appliquées mais montrées, interprétées et éprouvées dans l'interaction et la négociation" (Michel Callon et Bruno Latour, 1991, p. 15). Pour gérer les événements et les aléas de façon efficiente, les opérateurs doivent mobiliser des capacités d'initiative et des savoirs diffus, souvent ignorés et occultés par les directions des organisations (Anne Mayère, 1997, p. 205-208). La modélisation ne peut s'appuyer sur aucune convention implicite (au sens d'écran informationnel, d'économie d'information) entre un système informatique et des êtres humains, et ces conventions sont limitées entre les concepteurs et les utilisateurs, alors même que les conventions les plus implicites - qui semblent "naturelles" - sont les plus susceptibles d'être omises dans la modélisation : les routines sont fréquemment d'autant plus efficaces que nous sommes moins conscients des connaissances sur lesquelles elles reposent et qu'elles s'apparentent à des réflexes. En effet, un apprentissage peut être analysé comme s'effectuant en trois étapes : lors de la première étape (étape cognitive), un individu apprend à partir d'instructions ou d'observations sur le type d'actes appropriés dans telles ou telles circonstances ; dans un second temps, ces apprentissages de la première phase sont mis en pratique jusqu'à ce qu'ils deviennent aisés et précis (étape associative) ; enfin, dans la phase autonome, les actions intériorisées par la pratique sont effectuées de moins en moins consciemment, avec une rapidité et une dextérité de plus en plus importantes, mais le savoir déclaratif s'étant transformé en une forme procédurale, il ne nous est plus accessible et nous ne pouvons plus l'exprimer verbalement (Harry M. Collins, 1992, p. 110). En conséquence, la connaissance des savoirs utiles, notamment pour transformer les heuristiques pratiquées en des algorithmes exécutables par un ordinateur[133], requiert fréquemment l'observation directe, voire l'apprentissage, par les concepteurs du comportement concret des utilisateurs. Par exemple, Ikujiro Nonaka et Hirotaka Takeuchi citent l'exemple de la conception d'un logiciel pour piloter une machine à fabriquer le pain ; les échecs pour formaliser correctement le pétrissage de la pâte ne purent être surmontés que par l'apprentissage par la responsable du développement du logiciel et plusieurs ingénieurs, de la fabrication du pain, auprès d'un boulanger célèbre pour la qualité de ses pains ; cet apprentissage direct fut indispensable pour réussir à "socialiser la connaissance tacite du maître boulanger par l'observation, l'imitation et la pratique" (1997, p. 85).

[133] Une heuristique donne des règles générales pour atteindre des objectifs généraux, alors qu'un algorithme indique la solution optimale pour atteindre un but bien défini.

Des difficultés socio-politiques

Les difficultés ne sont pas seulement d'ordre communicationnel ou cognitif, notamment lorsque l'introduction du logiciel a des conséquences organisationnelles. Le développement des logiciels est fréquemment basé sur une vision de l'organisation comme étant apolitique, neutre, où tous les acteurs poursuivent un objectif commun, et où l'exhortation et la pression hiérarchique suffisent pour s'assurer une collaboration active des utilisateurs. Si à l'inverse on envisage une organisation comme un lieu de confit entre différents pouvoirs, dont la source réside notamment dans la détention de connaissances et le contrôle de l'information[134], on comprend que cette participation ne va pas nécessairement de soi. Une attitude de non-collaboration ne doit pas être analysée comme une peur irrationnelle du changement ou un simple "refus du changement", motivé par la nécessité d'effectuer de nouveaux apprentissages. Elle renvoie à un comportement rationnel basé sur la compréhension qu'au-delà de la représentation de l'activité actuelle, ce qui est en jeu est la réorganisation de l'activité, avec ses conséquences en termes de répartition des pouvoirs, de nouvelles tâches à effectuer, et, dans certains cas d'expropriation de connaissances détenues par les opérateurs.

Tout d'abord l'introduction d'un logiciel est l'occasion d'une restructuration des activités humaines, présentée comme n'étant pas inventée, mais simplement découverte, alors qu'elle résulte d'une construction qui nécessite une phase d'imposition importante (Philip E Agre, 1997, p. 251-258). Déjà en 1970, Michel Crozier soulignait que "l'application du système rationnel des informaticiens au système social que constitue une entreprise se heurte à des obstacles beaucoup plus profonds que ceux auxquels on pense généralement" dans la mesure où "il ne s'agit pas, seulement d'habitudes qu'il faudrait changer ni même d'intérêts qui seraient menacés" mais "de tout un ensemble de pratiques et d'arrangements qui constituent en fait le mode de gouvernement réel de l'entreprise, ou si l'on veut, les règles du jeu implicites des rapports entre les hommes, les groupes et les catégories". En particulier, la perte de la maîtrise de ressources informationnelles dont disposaient les utilisateurs peut susciter une opposition résolue d'agents économiques qui pouvaient valoriser certaines asymétries informationnelles (Eric Brousseau, 1993, p. 226). Les logiciels peuvent souvent permettre l'instauration de nouvelles méthodes de contrôle, beaucoup plus efficaces, de l'activité réalisée (Robert Michon et Lin Gingras, 1988). Ils nécessitent l'accomplissement de nouvelles tâches,

[134] Cf. l'exemple des ouvriers d'entretien qui avaient fait disparaître les notices des machines pour préserver leur pouvoir, analysé par Michel Crozier (1963).

imposant de nouvelles contraintes dans l'activité (exactitude, vigilance, réponse en temps réel), qui ne sont pas toujours reconnues au niveau de pénibilité qu'elles impliquent (*stress*). En même temps, ils peuvent représenter le risque, en codifiant dans les logiciels des connaissances tacites détenues par les opérateurs, de permettre de ne plus avoir besoin de certains d'entre eux ou de rendre obsolètes certaines de leurs compétences. Analysant de façon plus générale, la capitalisation des connaissances, Anne Mayère se demande "s'il est raisonnable et réaliste" de supposer que les acteurs et les collectifs dans l'entreprise vont contribuer à un "modèle organisationnel qui vise à les instrumentaliser toujours plus et qui, en leur demandant de formaliser leurs savoirs, se donne des moyens supplémentaires de se passer de leurs services" (1997, p. 206). Cet "effort de rationalisation et de formalisation des savoirs mobilisés dans l'entreprise, (…) projet taylorien poussé à son extrême, en étendant le concept de "tâches" à toutes les activités de l'entreprise" (idem, p. 205) a été analysé précisément par Yvette Lucas (1989) dans le cas de l'aéronautique, comme représentant un "vol de savoir" ; en effet, la codification des connaissances dans un logiciel entraîne un transfert de "propriété" des connaissances codifiées vers ceux qui détiennent les droits de propriété sur le logiciel.

Enfin, même "dans l'hypothèse particulièrement favorable de consensus sur les bénéfices à retirer du changement", Olivier Favereau montre, en s'appuyant sur différents exemples et analyses, la nécessité, pour enclencher des dynamiques de transformation, "d'établir une confiance réciproque cumulative entre dirigeants et exécutants" ; cette confiance est difficile à construire car "les caractéristiques mêmes du pouvoir, qui se révèlent si efficaces pour produire la conformité, sont contradictoires avec les mécanismes de production de la confiance" (1998, p. 225-228).

Il apparaît donc qu'il est particulièrement difficile pour des raisons simultanément communicationnelles, cognitives et sociales, de construire l'intercompréhension entre les concepteurs et les utilisateurs, dont dépendra pourtant ultérieurement la qualité des "communications" entre les logiciels et leurs utilisateurs.

2 - Les problèmes de "communication" entre les logiciels et leurs utilisateurs

L'utilisation de la plupart des logiciels implique l'existence de "dialogues" avec les utilisateurs. Un programme d'ordinateur est "un texte écrit, destiné à résoudre "interactivement" (l'ordinateur répond et pose des questions), un problème ou un ensemble de problèmes donnés. Pour un utilisateur, dialoguer avec un programme, c'est comme lire un livre qui répondrait aux questions qu'on lui pose" (Philippe Breton, 1990, p. 58). Derrière, ce

"dialogue" existe en réalité une forme de communication avec les auteurs du logiciel, qui ont "leur manière propre de résoudre le problème en question" (idem) avec leur style, leurs qualités et leurs défauts. Toute la difficulté est que cette communication n'est pas directe mais médiée par le logiciel. Dans l'hypothèse la plus favorable, il a pu exister une communication directe avec une partie des utilisateurs du logiciel lors de sa phase de conception. Lors de l'utilisation du logiciel les dialogues de l'utilisateur ne s'effectuent plus qu'avec le logiciel, si l'on excepte les échanges avec les concepteurs pour des améliorations ultérieures. A la différence d'un dialogue direct entre des êtres humains où des incompréhensions peuvent être surmontées par la reformulation des questions, l'introduction de nouvelles questions élaborées au cours du dialogue pour préciser les réponses, le dialogue avec le logiciel ne peut reposer que sur les questions et les réponses qui ont été enregistrées par les concepteurs *ex-ante* lors du développement du logiciel. Bernard Conein (1997) souligne l'asymétrie entre une interaction objet/humain, caractérisée par la rigidité et la fixité des instructions, et une interaction humain/humain où les divergences d'interprétation peuvent être surmontées au cours de la conversation. La perfection du dialogue avec le logiciel supposerait que les questions et les réponses soient complètement dépourvues d'ambiguïté pour les utilisateurs, ce qui est évidemment impossible. Intervient notamment de façon importante le contexte concret dans lequel est plongé l'utilisateur[135].

Philippe Breton compare la communication entre concepteurs et utilisateurs d'un logiciel à la communication qui s'établit entre l'auteur et le lecteur d'un livre (1990, p. 66). En suivant cette comparaison, il est frappant de remarquer comment cette citation de Socrate dans le Phèdre de Platon à propos des écrits semble s'appliquer parfaitement aux logiciels : "on croirait que de la pensée anime ce qu'ils disent ; mais qu'on leur adresse la parole avec l'intention de s'éclairer sur un de leurs dires, c'est une chose unique qu'ils se contentent de signifier, la même toujours ! Autre chose : quand une fois pour toutes il a été écrit, chaque discours s'en va rouler de droite et de gauche, indifféremment auprès de ceux qui s'y connaissent et, pareillement, auprès de ceux dont ce n'est point l'affaire, il ne sait pas quels sont ceux à qui justement il doit ou non s'adresser. Que d'autre part il s'élève à son sujet des voix discordantes et qu'il soit injustement dédaigné, il a toujours besoin de l'assistance de son

[135] Le rôle du contexte est décisif dans le cas de l'utilisation de logiciels "systèmes experts" qui ne peuvent être appliqués qu'à des domaines étroitement limités sans dénaturer la logique de l'expertise (Robin Cowan, Dominique Foray, 1998, p. 315).

père : à lui seul, en effet, il n'est capable, ni de se défendre, ni de s'assister lui-même" (cité par Harry M. Collins, 1992, p. 27).

Toutefois, les conséquences des imperfections de la communication entre auteur et "lecteur" sont en général plus importantes dans le cas des logiciels : dans le cas d'un livre, une mauvaise compréhension de ce qu'a voulu dire l'auteur ne peut avoir éventuellement que des conséquences indirectes, quand la lecture du livre influence les actions des lecteurs. Par contre, le logiciel étant un texte directement "actif", il comprend en quelque sorte deux types de "lecteurs" : les utilisateurs qui entrent des données et choisissent les traitements à effectuer en fonction de leur compréhension du programme, et le matériel informatique qui va exécuter automatiquement les parties du programme correspondant aux choix des utilisateurs.

C'est à ce stade de l'utilisation du logiciel que vont se manifester les difficultés d'intercompréhension concepteurs/utilisateurs analysées précédemment. "Un programme au début surprend toujours son utilisateur par son imprévisibilité apparente : les réponses de la machine sont parfois surprenantes quand on ne connaît pas le style du programme en question" (Philippe Breton, 1990, p. 69). Un exemple bien connu est le désarroi de l'utilisateur néophyte de Windows qui, pour arrêter correctement l'ordinateur, doit cliquer sur le bouton … "Démarrer". Ce qui est considéré comme absurde par l'utilisateur apparaissait en fait logique pour les concepteurs de Windows, pour lesquels arrêter un ordinateur consiste à démarrer l'exécution d'un programme spécifique[136].

Certes ces problèmes ne sont pas spécifiques aux logiciels et existent pour tout artefact dont les possibilités d'une utilisation satisfaisante dépendent de la qualité de l'intercompréhension entre concepteurs et utilisateurs. Mais ce problème prend une importance spécifique dans le cas des logiciels, en raison du nombre très élevé des choix proposés par l'artefact à l'utilisateur. La plupart des objets n'ont qu'un nombre de fonctionnalités limitées, ce qui restreint l'ampleur des communications avec l'utilisateur. Plus précisément, la variété des utilisations de multiples objets a été étendue en les rendant

[136] De même, la fonction "aperçu avant impression" qui permet de visualiser à l'écran un document tel qu'il sera imprimé, se trouve curieusement dans le menu "Fichier" et non dans le menu "Affichage". Ce fait déconcertant pour l'utilisateur, qui veut choisir un "affichage" différent de son travail, s'explique par l'historique de cette fonction. La fonction "aperçu avant impression", qui se trouve maintenant sur la plupart des logiciels, a été à l'origine conçue par un programmeur qui mettait au point un module d'impression des fichiers, et avait pour seul objectif de s'épargner les fréquents déplacements qu'il devait effectuer jusqu'à une imprimante distante pour juger de la qualité de son module.

programmables, c'est à dire en intégrant des logiciels (le plus souvent gravés dans des composants) dans ces objets. Dans le cas d'objets "grand public", cette programmabilité a occasionné des difficultés pour l'utilisateur, dont le meilleur exemple est le magnétoscope, ce qui peut conduire à limiter la surabondance des fonctions. Par contre, dans le cas de l'ordinateur, machine universelle dont la multiplicité des usages ne cesse de s'étendre, la variété des communications va continuer à se développer, d'autant plus que les interfaces entre l'homme et l'ordinateur se diversifient de plus en plus. En même temps, la richesse de ces nouveaux canaux de communication (par exemple la parole) contribue à prêter à un système informatique des formes d'intelligence dont il est dépourvu, ce qui ne peut qu'augmenter le sentiment d'insatisfaction de l'utilisateur devant des comportements du système différents de ceux qu'il escompte[137].

Conclusion : l'économie du logiciel en crise permanente ?

En 1968, au cours d'une conférence sur le thème du génie logiciel organisée par l'Otan, des professionnels du logiciel créèrent l'expression "crise du logiciel", qui selon la plupart des études perdurerait depuis cette époque (cf. par exemple, OCDE, 1991 A, p. 41). Il est vrai que de nombreux exemples que nous avons mentionnés lors de l'étude de la productivité, de la fiabilité et de la qualité semblent confirmer la réalité de cette crise. Un aspect significatif est la proportion des projets qui doivent être abandonnés en cours de développement. Capers Jones estime que 24 % des projets sont arrêtés sans avoir été achevés (1998, p. 21)[138]. Cette proportion est encore plus importante pour les grands projets : selon la même source, le taux d'annulation dépasse 50 % pour les systèmes de taille importante comportant plus de 10000 points de fonction. Frederick P. Brooks compare les développements des gros programmes

[137] Harry M. Collins souligne le danger des interprétations anthropomorphiques des systèmes informatiques, qui mettent en place des attentes de la part des usagers, qui seront forcément déçus (1992, p. 108). Certains auteurs estiment même qu'il est négatif que les ordinateurs acquièrent des compétences dans le domaine de la conversation naturelle, car cela aboutirait à les faire passer pour doués d'une expertise et d'une infaillibilité dont ils ne disposent pas en réalité.

[138] Toran Demarco, et Timothy Lister titre le chapitre I de leur livre "quelque part, en ce moment un projet est en train d'échouer" (1991, p. 12), Lauren Ruth Wiener affirme connaître "des programmeurs qui ont vingt ans de métier derrière eux et qui n'ont jamais contribué à un produit commercialisé" (1994, p. 110).

aux énormes animaux préhistoriques qui s'engluaient dans les lacs de bitume naturel : parmi ces programmes, "peu ont atteint leurs buts et respecté leurs calendriers et leurs budgets (…), la difficulté ne [semblant] pas provenir d'un point particulier (…) mais de l'accumulation de facteurs qui interagissent simultanément et ralentissent les mouvements de plus en plus" (1996, p. 4). De nombreux exemples peuvent être cités à titre d'illustration : le nouveau système de contrôle aérien aux Etats-Unis dont les travaux ont été suspendus après quelques milliards de dollars de dépenses (Jacques Printz, 1998, p. 40) ; le système financier développé pour la Bank of America d'un coût de 20 millions de dollars sur quatre ans et qui ne fonctionna que très épisodiquement pendant moins d'un an (Lauren Ruth Wiener, 1994, p. 110) ; un système de manutention automatique de bagages qui a retardé de plus d'une année l'ouverture du nouvel aéroport de Denver, système qui ne fonctionne toujours pas correctement, une partie de la manutention des bagages devant être effectuée manuellement (Maarten Boasson, 1998, p. 2).

Une deuxième manifestation (et conséquence) de cette crise est l'importance prise par la maintenance. Les estimations sont variables selon les auteurs mais toujours très élevées : une enquête menée en 1986 par *EDP Software Maintenance* auprès de 55 entreprises concluait que 53 % du budget total du logiciel est affecté à la maintenance (Patrick Jaulent, 1992, p. 17) ; Jean-Marc Geib estime que la maintenance représentait 65 % des coûts de développement des logiciels (1989, p.5), évaluation proche de celle de Michael A. Cusumano qui est de 67 % (1991, p. 65) ; Frédéric Georges Roux considérait que 75 % de l'activité des informaticiens, en 1988, était consacrée à la maintenance au détriment du développement de nouvelles applications (1991, p. 39). Comme la maintenance croît avec le stock d'applications utilisées, et que ce stock augmente en raison de la différence positive entre le l'introduction de nouveaux programmes et le retrait des anciens programmes, le Gartner Group estimait que les dépenses de maintenance pourraient atteindre dans les années à venir jusqu'à 95 % des budgets de programmation (Jean-Marie Desaintquentin, Bernard Sauteur, 1991, p. 60). Il est vrai que la maintenance peut recouvrir dans le domaine des logiciels des activités plus variées, que celles habituellement désignées par ce terme. Lauren Ruth Wiener estime même que le terme de maintenance est utilisé d'une "manière totalement abusive pour désigner ce qui arrive à un logiciel une fois que le client l'a acheté. A la différence des voitures et des autres systèmes physiques, un logiciel ne s'use pas quand l'on s'en sert. Il n'a pas à surmonter frottement ou inertie. Il n'a besoin ni de graissage, ni de nettoyage. (…). Ce dont le logiciel a besoin une fois commercialisé, ce n'est pas de la maintenance, mais des réparations" (1994,

p. 132). Toutefois la maintenance corrective (la correction des défauts résiduels d'un logiciel après sa mise en service) ne représente qu'une partie de la maintenance. La "maintenance" remplit également d'autres fonctions : amélioration des performances (maintenance perfective), adaptation à des modifications d'environnement comme le changement du matériel ou du logiciel système (maintenance adaptative), et surtout l'ajout de nouvelles fonctionnalités représentant des améliorations pour répondre aux besoins des utilisateurs (maintenance évolutive)[139]. L'importance prise par la maintenance s'explique également par une productivité plus faible que dans le développement de nouveaux logiciels, en raison de la nécessité d'intégrer des changements à un cadre déjà existant (Capers Jones, 1989, p. 57) et, en général, les moins grandes qualifications et expériences des personnes affectées à la maintenance des logiciels, opération considérée comme peu valorisante. En conséquence la maintenance des logiciels fait croître leur entropie (Frederick P. Brooks, 1996, p. 104), rendant encore plus difficiles les opérations de maintenance ultérieures. Au bout d'un certain temps, il se produit un phénomène de vieillissement du logiciel, rendant nécessaire des "soins gériatriques" (Capers Jones, 1998, p. 26). La durée de vie des applications beaucoup plus longues que prévue, dont certaines encore en service ont été écrites dans les années soixante-dix (Luc Rubiello, 1997, p. 134), peut rendre nécessaire de reprendre complètement la conception du logiciel pour le restructurer et l'adapter (réingénierie), ce que certains auteurs considèrent comme faisant également partie de la maintenance.

La crise du logiciel semble correspondre à un cercle vicieux, dont il est difficile de sortir. Les qualités insuffisantes (fiabilité, inadaptation) des logiciels génèrent une maintenance importante, qui influe négativement sur la productivité globale du secteur. De ce fait, il existe un nombre important de développements en attente[140] auquel il faut ajouter, pour les grands projets, la longueur du cycle de développement, dont nous avons vu qu'elle ne pouvait être réduite en augmentant les effectifs sur un projet ("loi de Brooks")[141]. Dès lors, la

[139] Sur 15 000 programmeurs en 1983 à IBM, 10 000 étaient occupés à ajouter des fonctionnalités aux programmes existants (Capers Jones, 1989, p. 55).

[140] Jean-Marie Desaintquentin et Bernard Sauteur soulignent que l'arriéré de demandes insatisfaites (ou *backlog*) entre les utilisateurs et le service informatique a pris aujourd'hui dans certaines entreprises "des dimensions réellement colossales" (1991, p. 61). En moyenne, Jean Brès estime qu'il existe deux ou trois ans de développements en attente (1994, p.5).

[141] Les systèmes informatisés de réservations aériennes, qui ont mis 30 ans pour être développés, constituent un cas extrême (Godefroy Dang Nguyen, 1995, p. 346).

pression est forte de produire le plus rapidement possible un programme utilisable avec un minimum de travail préalable à la programmation (analyse des besoins, spécification, planification et conception), ce qui a évidemment des conséquences en termes de fiabilité et de qualité des logiciels produits. Elle a également pour conséquence de freiner l'introduction de méthodes susceptibles d'améliorer la productivité à moyen terme, mais dont le temps d'apprentissage et d'assimilation risque de dégrader les performances à court terme.

Pourtant, nous avons également mis en évidence l'importance des améliorations existantes, améliorations partiellement masquées par la progression concomitante des besoins en logiciels et des exigences des utilisateurs. Certains auteurs estiment que les questions critiques de l'économie du logiciel s'expliquent avant tout par la jeunesse de cette activité et que les progrès scientifiques suffiront pour résoudre la "crise du logiciel" de façon identique à ce qui s'est produit dans d'autres domaines. Par exemple, Capers Jones conclut son ouvrage par le constat que "programmation et logiciel ont encore moins de quarante ans d'âge. Le rythme du progrès qui nous entraîne de cette forme d'art intuitif, pratiqué isolément, vers une discipline d'ingénierie est en fait aussi rapide que pour n'importe quelle autre science. Un panorama du génie chimique d'il y a cinquante ans, de la géologie d'il y a cent ans et de la physique d'avant 1910 feraient ressortir des problèmes analogues à ceux qui sont aujourd'hui les nôtres" (1989, p. 296). Il nous semble au contraire qu'indépendamment des progrès bien réels existants, la production des logiciels présente des spécificités qui rend particulièrement difficile la réalisation de logiciels qui seraient conjointement très fiables, bien adaptés aux besoins des utilisateurs et produits avec de hauts niveaux de productivité. En particulier la résolution d'une des questions critiques de l'économie du logiciel rend souvent impossible des progrès identiques sur les autres dimensions. Il en résulte que dans l'économie du logiciel se développe simultanément des processus différents de rationalisation de l'activité, distincts selon les nécessaires compromis opérés entre les différentes caractéristiques de l'activité, ce qui confère à l'économie du logiciel une grande diversité.