La "software crisis" et le développement de la production communautaire de logiciel ouvert

Libero MAESANO

libero.maesano@wanadoo.fr

url: http://seminaire.samizdat.net/article.php3?id_article=71

03/05/2005

Résumé

(à pourvoir)

Table des matières

> «… *No, I'd say that of the world's economies, there's more that believe in intellectual property today than ever. There are fewer communists in the world today than there were. There are* ***some new modern-day sort of communists*** *who want to get rid of the incentive for musicians and moviemakers and software makers under various guises. They don't think that those incentives should exist.*
> *And this debate will always be there. I'd be the first to say that the patent system can always be tuned--including the U.S. patent system. There are some goals to cap some reform elements. But the idea that the United States has led in creating companies, creating jobs, because we've had the best intellectual-property system--there's no doubt about that in my mind, and when people say they want to be the most competitive economy, they've got to have the incentive system. Intellectual property is the incentive system for the products of the future.*»
>
> Bill Gates, Interview à CNET news.com, 5 janvier 2005 - http://news.com.com/Gates+taking+a+seat+in+your+den/2008-1041_3-5514121-4.html?tag=st.num.

Avant-propos

La recherche qui est derrière ce papier a plusieurs motivations :

- La tentative de caractérisation plus précise du « capitalisme cognitif », par l'analyse d'un coté des processus de travail concret de la sphère de production de cet immatériel spécifique qui est le logiciel, qui occupe désormais la place centrale dans la production et reproduction de la richesse, et, de l'autre coté, par les rapports de production capitalistique. En d'autres termes, de voir comment le rapport de production capitalistique soumet (ou essaye de soumettre) la production du logiciel. La sensation initiale, nourrie aussi par plus de vingt cinq ans d'expérience professionnelle, est pour ce qui touche la reproduction et la crise, cette dernière, loin de pouvoir être caractérisée de façon classique, comme crise de la reproduction élargie (comme processus qui vient interrompre un cycle long de reproduction « ordonné »), est plutôt immanente, constante et toujours présente : plutôt que de crise dans la reproduction, il s'agit de *reproduction dans la crise*.

- La tentative d'expliquer des phénomènes apparemment inexplicables et contradictoires comme : (i) l'entrée massive du logiciel ouvert et libre, produit par un processus de production non capitalistique, dans le cœur de la production et la reproduction du capital ; (ii) la cession massive, de la part d'entreprises comme IBM, de logiciels avancés, produits par des efforts importants de recherche et développement, et donc en théorie source d'avantage compétitif, vers le monde du logiciel ouvert et libre, avec renonciation des supposés avantages ; (iii) la croissance extraordinaire et le taux de profit élevé d'entreprises comme Microsoft, dont le modèle est basé à l'inverse sur une revendication forte de la propriété intellectuelle du logiciel.

Le parti pris est de conduire l'analyse au cœur du processus de production (et d'éviter les pièges de l'analyse en termes « distributifs »).

La crise du logiciel

Le point de départ de cette contribution est un phénomène qui peut être considéré « technique » et propre à un secteur spécifique de l'industrie : la « *software crisis* ». La *crise du logiciel* est la difficulté, largement constatée dans tous les secteurs, à *estimer* correctement l'*effort* requis et la *fiabilité* du résultat de l'activité de production de logiciel. Les entreprises et les administrations sont sans cesse confrontées aux erreurs graves de sous-estimation des coûts et des délais, ainsi que des risques, des projets de développement logiciel. Il résulte aussi très difficile d'estimer correctement la fiabilité (à savoir l'absence de dysfonctionnements) du résultat des projets qui « réussissent » - et l'erreur va toujours dans le sens d'une sous-estimation. Le manque de fiabilité ajoute aux coûts de développement d'autres coûts, dit de *maintenance*, qui, sont souvent cachés et très difficiles à faire apparaître. Les coûts « apparents » de maintenance (ceux qui apparaissent explicitement dans les comptes de l'entreprise) sont déjà exorbitants et absorbent des pourcentages croissants des budgets informatiques des entreprises et des administrations, mais ne représentent qu'une partie des coûts. Par ailleurs, un nombre élevé de projets importants échoue complètement sans fournir aucun résultat exploitable.

En fait, dire que la crise du logiciel est un phénomène « technique », réservée aux professionnels d'un secteur particulier de l'industrie, revient à en sous-estimer l'impact, qui est de portée beaucoup plus générale. Le « grand public », au moins celui qui est « en deçà » de la fracture numérique, a droit quand même à un aperçu du problème, si l'on pense d'un coté aux dysfonctionnements (et trous de sécurité) de Microsoft Windows et à leurs conséquences en termes de gestion des « fixes » et des « services packs » qu'il faut installer régulièrement pour corriger ces dysfonctionnements, et de l'autre coté aux dates de livraison des « prochaines » version du logiciel qui, après plus de vingt ans d'existence de Microsoft, sont toujours fantaisistes, sans cesse remises en causes, sans parler du contenu de ces fameuses prochaines versions, qui lui aussi change sans cesse. Tout cela frappe l'imaginaire du « grand public » et donne lieu à toute sorte de théories, comme par exemple la théorie du complot : il s'agirait d'une ruse, d'une stratégie commerciale et marketing de Microsoft pour faire fonctionner son *business model*, qui est quand même toujours centré sur la vente en masse de « boites », de produits logiciels unitaires (essentiellement les Windows et les « suites » Office). Ces nouvelles versions sont évidemment « irresistibles » et plus prosaïquement censées résoudre les problèmes des versions précédentes. On n'imagine pas que le plus grand industriel du logiciel du monde (et, de surcroît, un *pure player* de cette industrie) ne soit pas capable de maîtriser ses plans de développements et la fiabilité de ses produits. Donc , de ceux choses l'une : soit il fait exprès ou bien, selon les compétiteurs et ennemis de Microsoft, il est vraiment « mauvais » e fonde sa puissance sur un coup de bol de départ, suivi par une agressivité commerciale à la limite du gangstérisme et, ensuite, par des produits complexes dont les rendements croissants d'adoption verrouillent le « parc » clients. On verra par la suite qu'il n'en est rien et que l'explication de tout cela est beaucoup plus terre terre.

Un des documents les plus connus sur la question est « The Chaos Report », résultat d'une étude conduite en 1995 par le Standish Group, un firme d'analyse et recherche en IT. Les données présentées sont frappantes : nous allons les résumer brièvement.

Aux Etats Unis, en 1995, la dépense en développement des projets IT a été d'environ 250 milliards de dollars, pou environ 175 000 projets. Le coût moyen d'un projet pour une grande entreprise est de 2 322 000 $, pour une moyenne entreprise de 1 331 000 $, et pour une petite entreprise de 434 000 $. De ces projets, en 1995 :

- 16,2% se sont terminé avec succès, ayant implémenté les fonctionnalités initialement spécifiées ;
- 52,7% se sont terminés avec un dépassement, en délai ou en coût ; le dépassement moyen en coût est de 89% ; les entreprises et les administrations ont dépensé 59 milliards de dollars de coût supplémentaire, non initialement prévu ;
- 31,1% ont été arrêtés en cours de route et n'ont produit aucun résultat exploitable ; la dépense globale pour ces projets est de 81 milliards de dollars.

L'étude couvre une partie importante du développement du logiciel, à savoir celui qui fait tourner les entreprises et les administrations, mais il ne couvre pas toute l'industrie du logiciel (l'industrie des jeux sur

ordinateur, par exemple, ou même les activités de Microsoft évoquées plus haut, et en général les éditeurs de logiciel).

Ce constat est dressé à un moment de forte croissance de l'industrie du logiciel, qui devient une industrie de premier plan et, selon certains pronostiques, risque de devenir la première industrie. Le logiciel devient la « matière première » principale de la plupart des produits, même les traditionnels - dans une voiture d'aujourd'hui il y a plus de logiciel que de matériel en terme de recherche et développement - et des services - le « client » interagit de plus en plus avec des ordinateurs pour s'informer, choisir, commander, suivre la prestation du service et même pour la « consommer », s'il s'agit d'un service informationnel. Le logiciel est aussi le « moyen de production » fondamental, présent dans tous les processus industriels ou tertiaires.

En plus de cette croissance en termes industriels et économiques, la production de logiciel envahit et « colonise » un grande partie de l'activité intellectuelle, qui devient quelque chose d'assimilable à la production de logiciel : on pense à la production de modèles par ordinateur, activité entre la recherche théorique et expérimentale, désormais centrale et irremplaçable dans toutes les disciplines : une « théorie » est aujourd'hui très rapidement un modèle implémenté sur ordinateur.

Le matériel (*hardware*), dont le développement connaît les lois de Moore - qui dit que la rapport puissance de calcul / prix des ordinateurs double tous les 2 ans, ce qui veut dire un ordre de grandeur d'un million pour ces 40 dernières années - et de Gilder - la bande passante du réseau triple de capacité tous les 12 mois - support local et distant (« *the network is the computer* » est la devise de Sun Microsystms) totalement banalisé sur lequel faire tourner de plus en plus de logiciel (*software*) !

L'évolution vertigineuse de la puissance du matériel joue un rôle particulier de déstabilisation « positive » de la création de logiciel : un effet immédiat et assez loufoque est que l'activité d'amélioration des performances d'un logiciel (étude, modification du code,test…) est plus lente de l'évolution de la puissance des machines ! Un autre effet, propre à l'expansion d'Internet est qu'il faut penser les logiciels sans limites d'utilisation, ou, plus précisément, éviter d'en introduire sans s'en apercevoir, car ils doivent pouvoir supporter des montées en puissance dans l'utilisation (*scalability*) exponentielles - de 10 à 1 000 000 d'utilisateurs en un ans.

«The Chaos Report » a exposé au grand jour une situation qui était ressentie depuis longtemps dans les milieux professionnels. En fait, la crise du logiciel existe depuis que le logiciel existe, et donc depuis le début de l'utilisation du « calculateur à programme mémorisé », dit improprement machine de Von Neumann. La crise endémique du logiciel a poussé la naissance d'une discipline spécifique, le « génie logiciel » (software engineering), avec son cortège d'experts, professeurs, gurus, chaires, sociétés de conseil spécialisées etc., qui se propose comme objectif de rationaliser sur des bases scientifiques et techniques solides soit le processus que le produit de l'activité de développement (au sens large) du logiciel.

Certes, les langages de programmation, les méthodes et les outils de développement, ainsi que les connaissances et le « niveau »de la force de travail intellectuelle ont énormément évolué depuis cinquante ans. Mais des situations comparables à celles décrites dans « The Chaos report » se reproduisent sans cesse : l'auteur a connaissance directe d'un grand projet d'une grande banque française qui, peu après le passage de millénaire, a été arrêté définitivement sans aucun résultat et après une dépense d'environ deux milliards d'anciens francs – à noter que ce projet utilisait des technologies de développement *up to date*. La simple affirmation que le programmeur d'aujourd'hui est en moyenne plus « productif » de celui d'il y a trente ans, qui semble relever du bon sens, n'est pas justifiable rigoureusement de façon simple. Même si l'on « sent » qu'il est plus productif, la mesure de la productivité, la prévisibilité des projets et la fiabilité des résultats sont autant incertaines aujourd'hui qu'hier. La seule différence est qu'aujourd'hui l'on essaye de contourner le problème de l'estimation en réduisant les « tailles » des projets dans le but de limiter les dégâts en cas d'échec ou de dépassement.

La question est : cette situation est provisoire ou définitive ? Les tenants du « génie logiciel » répètent inlassablement le leitmotiv : l'ingénierie du logiciel est une discipline jeune (par rapport à l'ingénierie, que sais-je, des ponts et chaussées qui est vieille de cinq mille ans) et, si l'on persévère dans la recherche de méthodes scientifiques de production du logiciel, qui réduisent l'incertitude des projets et augmentent la fiabilité des produits, on va certainement les trouver, c'est seulement une question de temps.

Les outils d’estimation du logiciel

Le génie logiciel propose une pléthore de méthodes de conception, de processus de développement, de méthodologies de programmation, de stratégies de vérification et validation, dont un nombre important intègre des méthodes et techniques « quantitative » d’estimation et du logiciel.

Un fameux « Manifesto » proclame :

« … il existe une base quantitative objective pour juger de la qualité du produit et pour analyser les problèmes du produit (logiciel) et du processus (de développement). Les plans et les budgets sont basés sur les données historiques et sont réalistes… »

Seulement il y a quelques années Watts Humphrey, inventeur du « Capability Maturity Model » (CMM) [Paulk] un modèle d’organisation des processus de production de logiciel (pour faire vite), écrit que les concepts et les méthodes propres au contrôle et à la planification des processus industriels : « ...sont autant applicables à la production du logiciel qu’elles les sont à la production de voitures, de cameras videos, de montres et de l’acier. » [Humprey 1988].

En fait une « théorie quantitative » de l’estimation du logiciel doit pouvoir répondre, entre autres, aux questions suivantes :

1. Peut-on estimer à priori, c’est à dire avant de l’écrire, la « taille » d’un logiciel ? Autrement dit, peut-on estimer la « complexité » de la tâche que le logiciel est censé accomplir ? Peut on formaliser la tâche que le logiciel doit accomplir (sa *spécification*) avant le production du logiciel lui-même, de façon précise et gérable ?

2. Peut-on estimer a priori, de façon fiable, l’effort nécessaire au développement d’un logiciel ? Peut-on estimer cet effort en unités de temps employées par un « programmeur moyen » comme par exemple l’homme*mois ?

3. Peut-on estimer a priori et objectivement la productivité des programmeurs affectés au projet ? Quelle est la relation entre la productivité des programmeurs, la complexité de la tâche et la taille du programme à ecrire ?

4. Peut-on estimer a priori le délai de développement du logiciel ? Autrement dit, est l’effort (la quantité totale de temps nécessaire pour produire un logiciel) « élastique » ? (cela signifie : si l’effort estimé est de 100 mois*programmeur et le développement peut être effectué en 10 mois par 10 programmeurs, peut-il également être effectué en 1 mois par 100 programmeurs ?).

5. Peut-on formaliser efficacement la tâche que le programme doit accomplir, avant le développement du programme ? En d’autres termes, peut-on formaliser des *spécifications* précises et exploitables avec un effort quand même moindre que le développement du programme ?

6. Peut-on poser a priori un objectif de fiabilité du logiciel (une mesure du « taux de conformité » du logiciel à ses spécifications) ? Peut-on estimer a priori l’effort en vérification et validation par rapport à un objectif de fiabilité ? Peut-on évaluer la conformité d’un logiciel à ses spécifications sans être obligé de le tester exhaustivement ?

7. Peut-on estimer l’effort de maintenance du logiciel, c’est à dire de la correction des erreurs résiduelles constatées lors du fonctionnement « en production » du logiciel (pour simplifier, l’on considère seulement la maintenance corrective) ?

Toutes ces questions (et il y en a d’autres) ne sont que des déclinaisons d’une question plus générale : existe-t-il un **processus formel**, donc spécifié et reproductible, qui peut être suivi par des agents indépendants (ordinateurs ou humains - la thèse de Church-Turing dit que ce processus est essentiellement un algorithme

qui peut être exécuté soit par un ordinateur que par un humain) pour parvenir à la même conclusion à partir des mêmes prémisses, capable de fournir a priori une estimation objective du logiciel (de l'effort, du délai, de la fiabilité) ? Ce processus doit être non seulement *calculable*, mais en plus son calcul doit être *traitable* (*faisable*), c'est-à-dire qu'il doit pouvoir être effectué en un temps raisonnable avec des ressources raisonnables (comme contre-exemple, les algorithmes dont le temps d'exécution croit de façon *exponentiellle* par rapport à la taille des données auxquelles ils s'appliquent ne sont pas traitables).

Différentes méthodes de conception du logiciel prétendent de pouvoir estimer l'effort de développement d'un logiciel en termes d'unité de temps pour un « programmeur moyen » (le fameux « homme*mois »). Les modèles les plus communs considèrent l'effort (en temps) de développement en fonction d'une mesure de « taille » (du problème ou du logiciel), et de la « productivité » du « programmeur moyen ». Les « mesures » de taille les plus utilisées sont :

- le *nombre de lignes de code source* : cette mesure implique d'un coté qu'il soit possible d'estimer a priori le nombre de ligne de code source d'un programme à partir d'une caractérisation de la tâche que le programme doit accomplir, e de l'autre coté que le nombre de ligne de code source soit une mesure homogène de la complexité de l'activité de programmation et donc de la productivité effective du programmeur ;

- le *nombre de points fonctions* : les points-fonctions sont des « régularités » décelées dans la caractérisation de la tâche que le programme doit accomplir ; cette méthode implique que le point fonction soit une mesure homogène de la complexité de l'activité de programmation et donc de la productivité effective du programmeur ;

L'approche du nombre de lignes de code source [COCOMO], est aujourd'hui pratiquement abandonnée, à la lumière aussi de ses échecs. Son problème principal, mis à part celui de définir précisément l'unité de mesure (qu'est-ce qu'une ligne de code source ?), ce qui ne va pas de soi, est l'évaluation a priori du nombre de ligne d'un programme que l'on doit encore écrire. Cette évaluation se base sur des données historiques issues de projets déjà accomplis dont l'objet rappelle celui du projet à accomplir et donc à estimer.

La métrique des *points fonctions* semble plus intéressante. En fait, elle s'attaque à une caractérisation de la « tâche » que le programme exécute, et non du programme lui-même. Elle est utilisée, par exemple dans les évaluations qui comparent des propositions concurrentes en réponse aux appels d'offre pour le développement de logiciels.

La complexité algorithmique

Introduction

Plus récemment, certains auteurs [Lewis2001], professionnels du domaine, prétendent que nous sommes en présence d'une impasse : la crise du logiciel ne serait pas due à la jeunesse de l'industrie et donc au manque d'expérience et d'approfondissement, voir de compétence de ses protagonistes, mais à quelque chose de plus fondamentale qui rend l'activité de production de logiciel « inestimable » et, par conséquent, non planifiable scientifiquement. La preuve fait appel à la théorie de la *complexité algorithmique de l'information*.

Au milieu des années '60, lorsque la *science de la computation* était en quelque sorte à ses débuts, mais la théorie générale des machines de Turing était déjà fort bien connue, certains ont posé la question de la mesure objective de la quantité d'information contenue dans les objets. La théorie de la complexité algorithmique de l'information, dite aussi complexité de Kolmogorov, fut inventée part R.J. Solomonoff, A.N. Kolmogorov e G.J. Chaitin de façon indépendante et en cet ordre. Cette théorie, qui a connu depuis un développement scientifique important dans ses fondements et ses applications, est aujourd'hui généralement acceptée comme approche standard de l'étude de la *quantité d'information* et donc de la *complexité* (les termes sont considérés synonymes) des objets ayant une représentation numérique, et donc des *suites binaires*, ou suite de chiffres binaires (*bits* – avec un code qui convient on peut toujours représenter un objet par une suite de bits, ce que font systématiquement les ordinateurs dont nous disposons aujourd'hui).

Intuitivement, la suite :

x = 11111111111111111111111111111111111111111111111111111111111111111111111111111…

composée de 1 000 000 chiffres 1 contient très peu d'information et donc est très peu complexe. En fait, je peux la décrire en français avec un énoncé de huit mots (e): « La suite composée d'un million de 1 ». Cette description est sans aucun doute plus courte que la suite elle-même, pourvu que l'on ait un code raisonnable pour représenter e en chiffres binaires. Intuitivement, un objet contient peu d'information s'il peut être décrit par un énoncé sensiblement plus court que l'objet en question. Si un agent A doit communiquer à un agent B x, il fait des économies en lui communiquant e à la place. Evidemment, B doit être capable d'*interpréter* p, et plus précisément d'effectuer l'opération de *dénotation* de e à x.

Les suites :

y = 10101010101010101010101010101010101010101010101010101010101010101010101010101…

z = 10100100010000100000100000010000000100000000000100000000000100000000000…

composées toujours d'un million de chiffres binaires sont toujours très peu complexes (un peu plus de x, peut être), car elles peuvent être décrites par des énoncés sensiblement plus courts qu'elles mêmes. Cela correspond à notre intuition de la complexité : complexe est ce qui n'a pas de *régularité*, de *motif* reconnaissables, ce qui est *aléatoire*, car ce sont les motifs, les régularités que nous décelons, qui nous permettent intuitivement de donner d'un objet une description plus concise de l'objet lui-même. Mais attention (nous passons à une représentation en chiffres décimaux) :

r = 41421356237309504880168872420 9…

semble aléatoire, mais en réalité la ligne ci dessus ne représente la suite des premiers 30 chiffres décimaux après la virgule du nombre réel √2. Cette suite, poursuivie jusqu'à un million de chiffres, n'a pas de régularités (nous savons qu'il n'en a pas car c'est un nombre irrationnel !), mais « La suite du premier million de chiffres décimaux après la virgule de la racine carrée de 2 » est un énoncé qui décrit de façon économe r.

Nous pouvons donner une première définition intuitive de complexité ou quantité d'information : la quantité d'information contenue dans un objet est la longueur de la plus petite description de cet objet.

En effet, la notion de description que nous avons donnée se perd dans les délices de la *dénotation*. Pour éviter le piège des *paradoxes sémantiques* (du type « Cet énoncé est faux » ou « Le plus petit nombre entier qui ne peut être décrit par un énoncé de moins de dix-neuf mots ») il faut formaliser la notion de description. C'est ce que fait la théorie de la complexité algorithmique, en utilisant la notion de *computation* (un anglicisme pour un calcul effectué par un ordinateur). L'on considère un ordinateur T, auquel on donne en entrée une suite binaire p (un programme). Si l'ordinateur T s'arrête après avoir rendu une suite binaire x en sortie, alors on peut écrire : T(p) = x et l'on peut dire que le programme p est une description de x (pour la machine T).

La quantité d'information (complexité algorithmique) contenue dans une suite binaire est la longueur - toujours en nombre de bits - du plus petit programme qui, a partir d'une mémoire vide et sans aucune autre donnée en entrée, produit en sortie la suite en question et s'arrête.

$$C_T(x) = \min |p| \quad \text{(pour tout } x \text{ et tout } p \text{ tels que } T(p) = x)$$

La même définition peut être donnée pour les objets infinis : dans ce cas le programme produit les éléments successifs de la suite sans s'arrêter. Une définition utile est aussi celle de *complexité conditionnelle* : C(x|y) est la longueur du plus petit programme qui prend la suite y en entrée et produit la suite x en sortie (c'est la complexité de x « en connaissant » y). Chaitin appelle ce plus petit programme le *programme élégant* sut T pour x, que nous noterons $p_T^e(x)$, définit donc formellement par les propriétés $T(p_T^e(x)) = x$ et $|p_T^e(x)| = C_T(x)$.

Cette définition de la complexité algorithmique (la quantité d'information) d'un objet quelconque semble dépendre de l'ordinateur utilisé, alors que nous voudrions que la complexité soit une propriété intrinsèque des objets. Il n'en est rien, Solomonoff a démontré que de toute façon les programmes sur deux ordinateurs différents (en termes d'architecture interne) ne diffèrent que d'une constante (qui est grosso modo la longueur du programme qui simule un ordinateur sur l'autre) et que cette constante est négligeable par rapport aux suites « suffisamment » longues. Plus précisément, il existe une machine « optimale » U telle que, pour toute autre machine T il existe une constante c telle que $C_U(x) < C_T(x) + c$. Nous pouvons donc choisir un ordinateur de référence U et poursuivre et parler de complexité C « dans l'absolu ».

Une suite est *incompressible* si $C(x) \geq |x|$. Non seulement il y a des suites incompressibles de toute longueur, mais, en plus, on peut montrer, au moyen d'un argument purement comptable, que si certaines suites sont énormément compressibles, (la suite d'un milliard de `1`, par exemple est produite par un tout petit programme !), la plupart des suites sont incompressibles. Il y a 2^n suites binaires de longueur n et $\Sigma_0^{n-k} 2^i = 2^{n-k} - 1$ suites de longueur inférieure à $n-k$, donc au plus $2^{n-k} - 1$ programmes de longueur inférieure à $n-k$, et cela est vrai pour chaque n. Donc, il y $2^n - 2^{n-k} + 1$ suites binaires x, telle que $|x| = n$ et $C(x) \geq n - k$, et ce nombre croit exponentiellement avec n et $n-k$.

La théorie de la complexité algorithmique a eu un développement très important, depuis les années 60, dans ses fondements et ses applications, mais elle est restée largement inconnue au « grand public », ou même au public qui, par exemple, a entendu parler du théorème d'incomplétude de Gödel, par exemple. Justement, une application remarquable de la théorie de la complexité algorithmique est une nouvelle preuve « élémentaire » et très courte du fameux résultat de Gödel sur l'incomplétude de l'arithmétique formelle, qui en plus nous épargne des affres de l'auto-référence de la preuve de Gödel (Barzdin, Chaitin). La théorie de la complexité algorithmique de l'information fonde en fait une véritable *physique de l'information*, qui n'a pas encore eu un effet épistémologique comparable à celui de la physique de l'énergie et de la matière, classique, relativiste et quantique (pour citer un exemple bien connu, l'on peut penser à l'influence de la mécanique et thermodynamique classique sur la théorie de la valeur/travail).

Résultats fondamentaux

Le problème, avec la théorie de la complexité algorithmique, est que son premier résultat fondamental n'est pas du tout rassurant : il dit que celle-ci n'est pas *calculable*, à savoir *qu'il n'existe pas une méthode générale, un algorithme, qui, à partir de tout x, est capable de calculer $C(x)$* ! La preuve est obtenue par l'absurde : elle postule qu'il existe un tel algorithme et en dérive une contradiction. Elle utilise une traduction formelle du vieux paradoxe sémantique de Berry (Le nombre dénoté par « Le plus petit nombre entier qui ne peut être défini par un énoncé de moins que dix-neuf mots » est bien défini en dix-huit mots !).

Supposons qu'il existe un algorithme A qui, à partir de toute suite binaire x, calcule $C(x)$ telle que nous l'avons définie. Il existe aussi un algorithme B qui, à partir d'un nombre n, produit la plus courte suite binaire x_c telle que $C(x_c) > n$. Nous savons qu'une telle suite existe, car nous avons vu qu'il existe des suites de complexité arbitrairement grande. L'algorithme B génère en boucle les suites binaires en ordre de longueur croissante (`0`, `1`, `00`, `01`, `10`, `11`, `100`…), applique à chaque suite générée x l'algorithme A pour produire $C(x)$, et, à la première suite générée x_n avec $C(x_n) > n$, s'arrête.

Soit m la longueur en bits du programme qui implémente B sur notre machine de référence. Nous pouvons donc écrire, pour notre ordinateur de référence, un programme r_n qui implémente l'algorithme B et produit en sortie la suite x_n. La longueur de ce programme r_n est le résultat de la somme de m (longueur en bits de l'implémentation de B), plus $\log_2 n$ qui est le nombre de bits qui sert a représenter n en base 2, plus k qui est la longueur en bits du programme « principale » qui fait « fonctionner » le tout et produit la x_n en sortie :

$$|r_n| = m + k + \log_2 n$$

Donc pour tout n tel que $n > m + k + \log_2 n$, le programme r_n de longueur en bits inférieure à n produit une suite x_n de complexité supérieure à n, ce qui est parfaitement contradictoire !

Evidemment, si la complexité algorithmique n'est pas calculable, il n'est pas questions non plus de pouvoir générer le programme élégant, car si c'était possible, il suffirait de le générer et de calculer sa longueur ! Nous pourrions nous accommoder avec ce résultat s'il était possible au moins de délimiter le champs d'incertitude de la complexité, de lui donner des bornes exploitables. Mais en fait deux résultats ultérieurs nous nient aussi cette possibilité.

D'abord, *la borne supérieure « banale » de la complexité algorithmique d'une suite arbitraire peut être définie mais sa production n'est pas traitable.* La borne supérieure de la complexité algorithmique est facilement définissable : $C(x) \leq |x| + c$ (c est une constante). Le programme p dont la longueur est $|p| = |x| + c$, contient la suite x plus un code de longueur c qui produit en sortie la suite en question. Nous appellerons ce programme le *programme tabulaire* pour x (ou *description tabulaire* de x) et nous le noterons $p^t(x)$.

Le programme $p^t(x)$ est facilement définissable, mais il n'est pas « faisable », à savoir qu'il devient vite ingérable avec la « croissance » de x. Pour s'en rendre compte, il suffit de considérer l'implémentation d'un fonction f:x,y→z, qui accepte comme arguments deux entiers représentés avec 32 bits et produit un entier (toujours sur 32 bits) en résultat (par exemple, l'addition). Soit v la suite binaire qui représente la table des tous les couples (x,y) d'arguments et z la suite binaire qui représente la table de tous les valeurs z (on peut toujours *coder* une table comme une suite binaire !). Le n-ième élément de la table z (z_n) représentant la valeur de la fonction f:x,y→z calculée à partir du couple d'arguments représenté par la n-ième entrée de la table v (v_n). Le programme tabulaire $p^t(z|v)$ reçoit en entrée un objet w de 64 bits, contient le couple (v,z) et en plus le code pour parcourir la table v, comparer chaque élément v_i avec w, et, si v_i est syntaxiquement égale à w, s'arrêter et produire z_i en sortie, sinon continuer jusqu'à la fin de la table t et s'arrêter.

La table v contient 2^{64} éléments de 64 bits, la table z contient 2^{64} éléments de 32 bits, pour un total de 2^{64} éléments de 98 bits (environ 10^{22} bits). La longueur du programme $p^t(z|v)$ ainsi conçu est sans aucun doute la borne supérieure de la complexité algorithmique conditionnelle $C(z|t)$, mais cette borne supérieure n'a aucun intérêt pratique car est intraitable.

Nous pouvons essayer de chercher une borne supérieure de la complexité algorithmique qui soit un peut plus utile que la borne supérieure banale donnée par la longueur du programme tabulaire. Malheureusement, *une borne supérieure asymptotique peut être calculée mais le calcul n'est pas traitable (faisable).*

On peut écrire un programme de recherche r qui recherche des programmes pour x plus courts que $|p^t(x)|$. Ce programme a nécessairement la structure d'un *déployeur universel*. Le programme produit les suites binaires en ordre croissant et les exécute comme programmes avec une stratégie bien définie d'allocation de temps d'exécution. Au cycle 1, il génère la première suite (programme) p_1, alloue un intervalle temporel fixe d'exécution t à cette suite et l'exécute pendant cet intervalle. Si l'exécution de p_1 se termine avant la fin de l'intervalle, il teste si le produit de l'exécution est syntaxiquement égale à x : dans ce cas, r s'arrête et produit p_1 en sortie, sinon il passe au cycle 2, tout en gardant en mémoire le contexte d'exécution de p_1. Au cycle n, il génère la suite p_n, alloue en séquence un intervalle temporel d'exécution t à chacun des programmes $p_1,\ldots, p_n$ qui ne se sont pas terminées dans les cycles précédents. Pour chaque intervalle d'exécution t alloué au programme p_i ($1 \leq i \leq n$), il adopte la stratégie décrite précédemment (pour le premier intervalle alloué à la suite p_1 dans le cycle 1). S'il ne s'arrête pas en produisant un programme p_k ($1 \leq k \leq n$) en sortie, il passe au cycle n+1.

Le déployeur universel r doit implémenter la stratégie d'allocation d'intervalles temporels fixes t aux programmes qu'il génère (que nous venons de décrire) car il doit faire face au *théorème de l'arrêt* de Turing, un des résultats des plus importants de la logique du siècle dernier, qui dicte qu'il n'existe pas de programme capable de recevoir n'importe quel programme en entrée et de détecter s'il s'arrête ou non. Le programme r, lorsqu'il génère le programme p_n, non seulement ne « sait » pas a priori s'il va produire x, mais il ne « sait » pas non plus s'il va s'arrêter ou non. S'il ne s'arrête pas avant, le programme r, va certainement s'arrêter lorsqu'il produit et exécute jusqu'à terminaison le programme tabulaire $p^t(x)$. Supposons qu'il s'arrête avant en produisant le programme p_k. Par construction, $C(x) \leq |p_k| < |p^t(x)|$. Nous avons trouvé une borne supérieure de la complexité de x, inférieure à la borne banale $|p^t(x)|$. Evidemment, nous ne savons pas si nous sommes tombés par hasard sur le programme élégant ($p_k = p^e(x)$) ou si, parmi les p_i ($1 \leq i < k$) qui sont encore en exécution il y en a un ou plusieurs qui vont se terminer un jour en produisant x. Nous savons avec

certitude que p_k est le programme élégant $p^e(x)$ si et seulement si lorsqu'il termine tous les p_i $(1 \leq i < k)$ ont terminé sans produire x.

Naturellement, nous pouvons modifier le programme r et continuer à lui faire chercher une borne « meilleure » que p_k : lorsqu'il a trouvé p_k, le programme, au lieu de s'arrêter, ne génère plus de nouvelles suites, mais continue à allouer des intervalles temporels d'exécution aux p_i $(1 \leq i < k)$ qui n'ont pas encore terminé leur exécution. Il s'agit d'une situation *semi-décidable* : si tous les p_i $(1 \leq i < k)$ se terminent, avant avoir pris la décision d'arrêter r, nous serons fixés. En revanche, si nous prenons la décision d'arrêter r, nous ne saurions jamais si, au prochain cycle d'allocation d'intervalles, un des programmes ne va s'arrêter après avoir produit x, donnant une « meilleure » borne supérieure de complexité.

Le programme r est non traitable tout simplement parce que le nombre de suites à tester croit de façon exponentielle avec leur longueur (il y a $2^{n+1} - 1$ suites de longueur inférieure ou égale à n). Donc, nous n'avons pas de méthode générale traitable pour déterminer une borne supérieure de la complexité de x qui soit « meilleure » que la borne banale $|p^t(x)|$.

Systèmes formels

Un exemple remarquable de la « mécanique » de la théorie de la complexité algorithmique est dans le *théorème d'incomplétude* de Chaitin, qui statue que *une théorie formelle avec n bits d'axiomes ne peut pas prouver des formules de type $C(x) > c$ si c est « suffisamment » plus grand que n.*

La preuve se fait par l'absurde : on peut raisonnablement assumer que, si $C(x) > c$ peut être prouvé, alors il est possible extraire de la preuve l'objet particulier x. Cet objet peut être accolé à la preuve. Cela veut dire que l'on peut générer x en utilisant approximativement n bits. Mais la preuve a montré que $CA(x) > c > n$, ce qui est contradictoire.

La preuve s'éclaircie si l'on représente le système formel en une forme calculatoire. Un système formel est :

- Un ensemble de *symboles*
- Une *grammaire* qui permet la combinaison des symboles en *formules bien formées* (*fbf*)
- Un ensemble de fbf qui sont accepté sans preuve (*axiomes*)
- Un ensemble de *règles d'inférence*, qui permettent de dériver des fbf (*théorèmes*) des axiomes et d'autres théorèmes.

Une *preuve* est la séquence d'inférences (applications des règles d'inférence) qui permettent de dériver un théorème. La preuve doit être formellement (à savoir « mécaniquement ») vérifiable, donc il existe une correspondance entre le système formel et un système de computation :

Système formel	*Système de computation*
axiomes	entrées ou état initial
règles d'inférence	interprète
théorèmes	sorties
dérivation	computation

Considérons un programme s qui contient une représentation binaire des axiomes et règles d'inférence d'un système formel quelconque S, génère successivement toutes les preuves en ordre de longueur et, lorsqu'il

trouve une preuve de la formule $C(x) > c$, extrait x de la formule, le produit en sortie et s'arrête. La taille en nombre de bits de la représentation binaire des axiomes et des règles d'inférence de S est n. Le programme nécessite aussi le codage en base 2 de la constante c, dont la longueur est $\log_2 c$. La taille du programme est $\log_2 c + n$ plus la taille d'un code qui gère la recherche, extrait et imprime la suite x. Soit k la longueur de ce code. La longueur finale est $|s| = \log c + n + k$. Pour n'importe quel n et k il est toujours possible de choisir un c tel que $c > \log c + n + k$. Nous avons donc un programme de longueur $|s| < c$ qui génère une suite x en passant par le preuve que $C(x) > c$, ce qui est contradictoire.

En conclusion, soit la preuve dérive un énoncé faux et donc le système n'est pas *sain* (un système formel est *sain* si les théorèmes qu'il produit sont tous vrais), soit le programme ne prouvera jamais la formule $C(x) > c$. En tout cas, un système formel ne pourra jamais construire de suite x telle que $C(x) > c$. Etant donné que le système formel n'est pas spécifié, on peut conclure que aucun système formel ne pourra jamais prouver que des suites sont de complexité supérieure à la longueur de son système d'axiomes et de règles d'inférence (plus une certaine constante). Une conséquence immédiate de ce résultat est que si l'on veut faire croître c, il faut faire croître n et donc ajouter des axiomes au système formel ! Le lecteur averti aura reconnu une certaine ressemblance entre le théorème d'incomplétude de Chaitin et le théorème d'incomplétude de Gödel : en effet la nouvelle preuve de Chaitin du théorème de Gödel se base sur des arguments semblables. Par ailleurs la preuve de Chaitin du théorème de Gödel est beaucoup plus simple et immédiate (sans les artifices de l'autoréférence de la preuve fournie par Gödel), qui pourrait être exprimée « naïvement » dans les termes suivants : en fait les axiomes de l'arithmétique de Peano ne contiennent pas suffisamment d'information pour déduire toutes les vérités arithmétiques, et il y en a qui restent en dehors. Pour repousser les limites, qui existeront qund même toujours, il faut ajouter d'autres axiomes (de l'information) à la théorie !

Conclusions sur la complexité algorithmique

Nous avons introduit sommairement et « naïvement » la théorie de la complexité algorithmique de l'information. Le lecteur doit commencer à entrevoir l'application de ces résultats à l'estimation du logiciel, qui sera l'objet du paragraphe suivant.

Pour conclure sur le sujet, et si l'on prend un peu de recul, on peut situer la rupture toujours en 1931, lorsque Gödel publie son fameux résultat d'incomplétude de l'arithmétique de Peano (après avoir prouvé la complétude et la cohérence de la logique de Frege) : il y a des propositions vraies de l'arithmétique qui ne peuvent pas être déduites à partir des axiomes de l'arithmetique. Von Neumann est le seul à comprendre tout de suite l'effet destructeur du résultat de Gödel sur le programme de Hilbert de formalisation axiomatique/déductive des mathématiques (et arrête sec toute activité de recherche sur les fondements des mathématiques). Par ailleurs, Von Neumann considère le résultat de Gödel le troisième des grands bouleversements intellectuels du vingtième siècle, les autres étant la théorie de la relativité et la mécanique quantique.

En 1936, Turing enterre définitivement le programme de Hilbert, par la démonstration de l'indécidabilité du « problème de la décision » (*Entscheidungsproblem*): aucun algorithme ne peut établir si une proposition quelconque est un théoreme dans la logique des prédicats, à savoir si c'est une conclusion dérivable de premises données utilisant les règles de calcul logique de Frege (autre théoreme d'indecidabilité).

Comme sous-produit de cette démonstration, Turing invente le calculateur universel (la machine de Turing) et démontre aussi un résultat qui est en rélation directe avec la théorie de la complexité algorithmique (on peut même dire à son origine): il est impossible de concevoir une machine de Turing qui prend en input la description de n'importe quelle autre machine et est capable de dire en temps fini si cette dernière s'arrête ou tourne à l'infini (autre théoreme d'incalculabilité).

Presque en même temps (années 60) et de façon indépendante, Solomonoff, Kolmogorov et Chaitin fondent la théorie de la complexité algorithmique de l'information en démontrant son résultat fondamental d'inexistence d'une méthode effective de calcul de la complexité d'une suite quelconque. Chaitin revient sur le résultat de Gödel avec une nouvelle démonstration basée sur la complexité algorithmique. Cela supporte formellement un mouvement qui repense la mathématique comme une science expérimentale, dans laquelle

le problème est moins de trouver des preuves kilométriques à partir d'ensemble d'axiomes intouchables, mais plutôt, comme dans les sciences expérimentales, de trouver des nouveaux axiomes et des nouveaux systèmes d'axiomes plus informants et plus productifs. Cette nouvelle « physique » des mathématiques fait de paire avec cette véritable « physique » de l'information qui est la théorie algorithmique.

Application à l'estimation du logiciel

Introduction

Il s'agit d'abord de fournir les passages qui permettent l'application de la théorie de la complexité algorithmique à l'estimation du logiciel.

Prenons un *logiciel fonctionnel*. Un logiciel fonctionnel est l'implémentation sur un ordinateur d'une ou plusieurs fonctions au sens mathématique (ou au sens des fonctions récursives) $f:x\rightarrow y$ qui, à partir d'arguments calcule des valeurs. La spécification de cette tâche, en termes d'un objet binaire, est la représentation binaire d'une table qui contient tous les couples arguments/valeurs. On peut représenter donc cette table par le couple d'objets binaire (x,y), ou x est la représentation binaire de la table des arguments et y est la représentation binaire de la table des valeurs correspondants : si l'argument de la fonction est représenté par x_n, nième élément de la table des arguments x, la valeur correspondante est représenté par y_n, nième argument de la table des valeurs y.

Nous avons déjà introduit $p^t(y|x)$, *programme tabulaire* implémentation de la fonction $f:x\rightarrow y$. La longueur d'un tel programme $|p^t(y|x)|$ est supérieure à la longueur en bits de l'objet qu'il doit produire. Nous avons vu aussi que le couple d'objets binaires (x,y) et donc la représentation tabulaire des fonctions ne sont pas traitables, car ils croient de façon exponentielle, ce qui est une évidence dans le cas des logiciels, même les plus simplistes.

La plupart des logiciels que nous côtoyions ne sont pas des programmes purement fonctionnels, mais des programmes *impératifs*, qui gèrent des *états* (par exemple la réservation d 'une place sur un avion) et dont le comportement est dépendent de ces *états*. Ces programmes peuvent être reconduits à des programmes fonctionnels : il suffit de considérer l'état comme un argument en plus(et un résultat en plus).

Les programmes *interactifs*, à savoir ceux qui interagissent avec les usagers, peuvent aussi être reconduits à des programmes *impératifs* et donc, par le truchement présenté dans le paragraphe précédent, à des programmes fonctionnels. Les commandes de l'usager peuvent être considérées des arguments en plus. Typiquement, le programme est un ensemble de sous-programmes (routines) et une représentation partagée de l'état global. La longueur du programme est donc la longueur des sous-programmes combinés plus la longueur du « programme principal » qui donne le contrôle aux sous-programmes sur la base de la commande de l'usager.

Nous pouvons donc affirmer que, dans l'abstrait, toute spécification de logiciel peut être réduite au couple de suites binaires (x,y) (dans l'abstrait veut dire sans tenir compte du fait que cette représentation est *intraitable*) et que la borne inférieure (non computable) du programme qui implémente ces spécifications est $C(y|x)$.

Détermination de la complexité de la tâche

Différents méthodes de conception et modèles de processus de développement de logiciel adressent la problématique de l'estimation de *l'effort de développement du logiciel*. Nous avons vu que ces modèles d'effort estiment le temps de développement en fonction d'une mesure de *taille*, comme le *nombre de lignes de code sour*ce ou le *nombre de points fonctions*. Ces tailles peuvent être reconduites respectivement à la taille du programme $|p|$ et à la taille de la tâche $|x| + |y|$. Ces méthodes posent donc une relation fonctionnelle (généralement linéaire) entre l'effort de développement du logiciel, mesuré en mois*homme par exemple, et ces tailles. La relation fonctionnelle est issue de l'extrapolation de données historiques, ce qui postule, une

« similitude » entre la tâche que le logiciel à développer doit accomplir et les tâches accomplies par les logiciels qui ont fournit les données historiques. Ces méthodes suggèrent que les modèles d'effort, issu des données historiques sur les temps de développement, peuvent fournir l'estimation des temps de développement des projets futurs.

Le problème est que, même en supposons que la relation fonctionnelle entre *taille* et *effort*, issue des données historiques, soit fiable, précise et reproductible (certains chercheurs ont trouvé sur le terrain des écarts d'un ordre de grandeur entre estimation et réalité constatée a posteriori), la difficulté réside dans la caractérisation de la tâche que le logiciel à développer doit accomplir et dans son appariement avec celles accomplies par les logiciels dont les données historiques et donc la relation fonctionnelle entre taille et effort sont issues.

A ce propos, [Lewis] présente un scénario paradoxal. Une entreprise de production de logiciel décide d'améliorer ses capacités d'estimation des processus de développement du logiciel, et donc décide de collecter des données sur les temps de développement pour lui appliquer des modèles statistiques. Il effectue une opération « taylorienne » d'étalonnage de la productivité. On assigne à chaque programmeur une série d'exercices. On constate que le « programmeur moyen » effectue ces exercices en 4 heures. L'entreprise est sollicitée dans un appel d'offres pour le développement d'un nouveau système d'exploitation. Sur la base des données fraîchement collectées et des statistiques fraîchement élaborées, la compagnie réponde qu'elle est capable de produire le système d'exploitation en 4 heures en y affectant un « programmeur moyen ».

L'histoire est paradoxale, mais met bien en évidence que toutes les méthodes d'estimations font l'impasse sur la caractérisation de la tâche que le programme doit accomplir. Sans cette caractérisation, on ne peut pas appliquer la relation fonctionnelle entre taille et effort issue des données historiques. Le problème est double. Si l'on avait bien identifié une classe de tâches et une relation fonctionnelle entre taille et effort pour cette classe (issue des données historiques), et si l'on pouvait disposer d'une méthode effective pour décider si une nouvelle tâche fait ou non partie de la classe, on pourrait procéder à des estimations plus ou moins précises pour les nouvelles tâches de la classes (mais non pour les autres). Le problème est que l'on ne sait pas caractériser la classe de tâches, et que donc l'appariement d'une nouvelle tâche avec les tâches anciennes qui sont à l'origine des données historiques est pour le moins imprécis, et en fait totalement subjectif et basé sur l'intuition. Comme souvent arrive, ces méthodes déroulent des calculs savants à partir d'assomptions non fondées.

D'un point de vue pratique, au vu du rythme effréné d'innovation qui caractérise l'industrie du logiciel, cette méthode ne serait applicable que de façon marginale. Tous les jours sortent des logiciels nouveaux qui font des choses nouvelles, qu'aucun autre logiciel avait fait auparavant. Et même dans le cas de *refontes* (des logiciels qui sont développé à nouveau, quoique jamais à l'identique mais par exemple avec des fonctionnalités nouvelles, en utilisant des nouvelles technologies - nouveaux langages de programmation, par exemple), qui sont justifiées par l'obsolescence des technologies logicielles sur lesquelles étaient basés les anciens logiciels, ce sont justement ces différences recherchées qui rendent improbable l'application des méthodes d'estimation basées sur la taille et issues des données historiques.

Sur la base des résultats de la théorie de la complexité algorithmique et de toutes les considérations précédentes, on peut établir un premier résultat : (i) *la tâche qu'un logiciel doit accomplir ne peut pas être caractérisée a priori de façon traitable*. Le fameux couple (x,y) ne peut pas être établi ni maîtrisé. Ceci a d'ailleurs des conséquences intéressantes sur les activités initiales du cycle de développement classique du logiciel, dites activités d' *analyse*, qui ont comme but d'établir des *spécifications* (nous avons défini (x,y) comme la *spécification* du programme à développer). Nous ne développerons pas cette problématique. Des mêmes prémisses, suit immédiatement un deuxième résultat : (ii) *la complexité de la tâche (au sens de la théorie de la complexité algorithmique) que un logiciel doit accomplir ne peut pas être estimée de façon traitable à priori* ; en d'autres termes *la taille d'un logiciel ne peut être pas estimée a priori*. Nous rappelons que la borne supérieure banale de la complexité de la tâche est la taille du programme tabulaire, qui est du même ordre que la taille du problème $|x| + |y|$ et donc est inexploitable : non seulement n'est pas traitable mais est en plus est inutile (nous espérons bien que le logiciel que nous allons développer sera de taille inférieure de plusieurs ordres de grandeur de la taille « énorme » $|x| + |y|$).

Si l'estimation exacte de la complexité de la tâche que le programme doit résoudre n'est pas computable, on peut se poser la question d'une estimation rapprochée. Est-il possible d'avoir une estimation $E(y|x)$ de la

complexité telle que $C(y|x) \leq E(y|x) \leq C(y|x) + k$. En fait, le théorème d’incomplétude de Chaitin s’applique: sa reformulation dit qu’un programme qui calcule une estimation de complexité rapprochée des programme ne peut pas produire une limite inférieure de la complexité plus grande que sa propre taille ! La conclusion est que (on saute la démonstration) : (iii) *il n’y a pas d’estimation rapprochée qui produit une limite correcte et exploitable pour tous les problèmes*.

Ces résultats sonnent le glas de toute velléité de trouver une méthode générale effective d’estimation du logiciel basée sur la complexité de la tâche et la taille du logiciel. Même si nous supposons d’avoir une mesure fiable de la productivité en termes de taille du logiciel (un programmeur moyen produit N lignes de code sources en un mois !), nous ne pouvons pas calculer l’effort d’un logiciel dont on ne connaît pas la taille. Si nous ne pouvons pas calculer l’effort, même si nous supposons l’élasticité totale de cet effort (l’effort de 10 programmeurs pendant 10 mois calendaire est le même de celui de 100 programmeurs pendant un mois calendaire) alors : (iv) *le délai de développement d’un logiciel ne peut pas être estimé a priori*.

La productivité et la fiabilité

Mais le problème est plus profond : ce sont les relations même entre taille de la tâche ($|x| + |y|$ - de toute façon ingérable), complexité de la tâche ($C(y|x)$ – non computable), taille du logiciel effectivement développé ($|p| \geq C(y|x)$) et temps de développement qui sont en question ! On peut effectivement considérer l’activité de développement de logiciel comme une activité de résolution de problèmes (*problem solving*) : ce qui intéresse en termes de productivité est le temps de production d’un programme qui accomplit une tâche spécifiée (en supposant qu’il soit possible de spécifier la tâche que le programme doit accomplir de façon effective) et cela indépendamment de sa taille. La concision est considérée une qualité d’un logiciel, surtout pour sa maintenance, mais il est empiriquement prouvé qu’un logiciel vraiment concis est produit dans un délai plus long qu’un logiciel partiellement bavard, car cela demande une activité de mise en facteur commun que le génie logiciel moderne appelle *refactoring*, qui est inévitablement effectué en grande partie a posteriori, après un premier développement. De toutes façon, les études empiriques font état de différences importantes en taille (d’un ordre de grandeur) pour des programmes écrits dans le même langage à partir des mêmes « spécifications ». La productivité s’établit donc en relation à une « complexité » du problème à résoudre pour le programmeur pour écrire un programme qui accomplit une tâche. Cette « complexité » n’a pas de relation immédiate et décelable avec la complexité algorithmique de la tâche et qu’il est encore plus difficile à caractériser à priori : (v) *la productivité absolue ne peut pas être déterminée de façon objective*.

La notion de *productivité* est aussi en relation forte avec celle de *fiabilité* du logiciel. Intuitivement, et cela arrive tout le temps dans les situations réelles, la pression exercées sur les programmeurs pour raccourcir les délais de développement risque de provoquer une baisse de la fiabilité du logiciel, donc une augmentation des dysfonctionnements constatés après mise en production et une augmentation donc considérable de l’effort de maintenance. Par ailleurs, c’est vraiment dans l’activité de maintenance que l’on mesure l’écart entre taille des modifications produites et les efforts pour les identifier : l’on constate couramment que des dysfonctionnements très difficiles à *diagnostiquer* (et parfois à *détecter*, car ils se produisent dans des conditions particulières dont la configuration n’a pas été tracée lorsqu’ils sont apparus pour la première fois) sont, après des journées d’investigation, corrigés par le changement d’une ligne de code !

Par ailleurs, parfois les erreurs « minimes » en termes de « taille » de la correction ont de conséquences catastrophiques : le 4 juin 1996 la fusée Ariane 5 lancée de Kourou un des programmes embarqués a introduit une donnée représentée en 64 bits en un registre de 16 bits. La perte d’information qui en a suivi a provoqué une chaîne d’événements qui a conduit au crash de la fusée pour une perte de 500 millions de dollars.

En termes formels, un logiciel est *correct* si son comportement s’apparie avec le comportement défini par sa spécification. La situation idéale serait, pour tout (x,y), de disposer du programme tabulaire et donc de pouvoir comparer le chaque résultat du programme développé p avec le résultat du programme tabulaire. Naturellement nous savons que le programme tabulaire et toute la procédure est intraitable. Au lieu de tenter l’examen exhaustif du comportement du programme, qui est intraitable, nous pouvons espérer que l’on pourrait vérifier le comportement du programme par l’examen de sa structure (sa logique interne). Supposons d’avoir à notre disposition une spécification s, objet qui pour l’instant n’est pas précisément

caractérisé, et le programme p qui est censé implémenter une telle spécification. Nous voulons prouver M(s,p), un énoncé qui dit que le comportement du programme p s'apparie avec le comportement défini par sa spécification s. La spécification s doit apparaître dans la preuve et doit être formelle pour y « participer ». Si la spécification s spécifie une fonction f:x→y, dont la représentation tabulaire est (x,y), de façon formelle, il doit y avoir un programme r de longueur fixe qui est capable d'interroger s avec un argument x_0 et obtenir la valeur correspondante y_0. L'ensemble (s,r) constitue un programme qui se comporte comme p et dont la longueur en bits ne peut pas être inférieure a $|s| + |r| \geq C(y|x)$. La spécification est d'une taille comparable au programme p et par définition plus longue que le *programme élégant*. *Même si l'on est capable de prouver M(s,p), qui nous prouve que la spécification s est correcte ?*

Dans le cycle de développement classique (mais encore largement utilisé) du logiciel qui prévoit un travail d'analyse dont le résultat escompté est la « spécification » du logiciel à développer, cette dernière n'est rien d'autre qu'une exposition informelle et totalement incomplète de la tâche que le programme à développer est censé accomplir. Cette exposition sert simplement à rapprocher le programmeur du problème à résoudre et sert aussi à préciser autant que l'on peut certains aspects de la tâche. En réalité, en l'absence d'une spécification formelle, la notion même de *correction* du programme est dénouée de sens formel : un dysfonctionnement n'est pas un écart entre une spécification formelle (qui n'existe pas) et un programme, mais est plutôt défini « socialement » comme un comportement du programme qui n'est pas acceptable par la communauté de ses utilisateurs. C'est évident que cela ajoute encore de l'incertitude à l'estimation du logiciel.

Conclusion sur le génie logiciel

Le but de Lewis est de prouver que les revendications d'*estimation objective* du logiciel issues de la communauté du génie logiciel, sont erronées. En fait, la communauté du génie logiciel peut être partagée en deux champs:

- Le champ « processus » : ceux qui croient que un logiciel de qualité peut être produit dans les délais et les coûts si un processus particulier et/ou une technologie particulière sont appliqués au cycle de développement. Ces derniers proposent les techniques d'estimation objective du logiciel que nous avons évoquées.

- Le champ « *problem solving* » : ceux qui croient que l'activité de programmation soit essentiellement une activité de résolution de problèmes et que donc elle résiste à toute formalisation hâtive. Ces derniers proposent des méthodes et outils qui sont censés améliorer la productivité (relative) du développement et la fiabilité (relative) du résultat, sans pour autant fournir méthodes et outils d'estimation absolue.

Dans le contexte de ce papier, nous ne nous intéressons pas à une discussion technique entre professionnels du domaine, mais plutôt à examiner comment se configure, à la lumière de ces résultats sur l'estimation objective du logiciel, une organisation capitalistique de sa production.

L'organisation capitalistique de la production du logiciel

Soumission formelle et soumission réelle

Les méthodes et les techniques d'estimation objective du logiciel proposées par le génie logiciel ne sont pas fondées scientifiquement: nous n'avons donc aucune méthode objective qui nous permet d'estimer l'effort et la durée d'un projet de développement de logiciel. Nous sommes très loin de l'organisation scientifique du travail de Taylor. Dans cette situation, comment l'organisation capitalistique de la production du logiciel est possible ?

Nous allons examiner la question sous l'angle logique et nous nous réservons de revenir après sur la phénoménologie de l'industrie du logiciel. Nous savons avec Marx que le mode capitalistique de production

nécessite toujours la soumission du travail au capital, qui marche sur deux figures possibles: la *soumission formelle* et la *soumission réelle*.

La soumission du travail au capital est essentielle pour que l'argent fonctionne comme capital et donc le cycle

$$A - M \ldots P \ldots M' - A'$$

puisse s'accomplir. Dans la figure de la soumission formelle, le *capitaliste-marchand* « anticipe » l'achat des matières premières, des outils de production et de la force de travail (A – M). Il laisse ensuite « le miracle se produire » (M … P … M'), et récupère des marchandises M' qui, valorisées sur le marché (M' – A')donnent une somme d'argent A' > A comprenant la plus-value. Le capitaliste-marchand ne rentre pas dans l'organisation de la production : il se limite à gérer des transactions marchandes d'achat de « facteurs » (D – M) et de revente de produits finis (M' – A'). La soumission de la force de travail est *formelle* car le capitaliste n'organise pas de l'intérieur ma se limite à contrôler de l'extérieur un élément essentiel du processus de valorisation du capital (A - - A'): le processus de production M ... P ... M'.

La soumission du travail au capital dévient *réelle* lorsque le *capitaliste-organisateur* entre dans le processus de production et l'organise, ou, mieux, consacre une partie de la force de travail à concevoir une organisation de la production (la recherche et développement, le bureau des méthodes) et une autre partie de la force de travail a diriger la production en temps réel (le *management*), ou mieux, à contrôler que l'organisation de la production établie soit effectivement suivie et a prendre les décisions qui s'imposent.

Le *capitaliste-organisateur* fait travailler son argent comme capital selon le même cycle A – M ... P ... M' – A', mais cette fois-ci ne se limite pas à acheter les ingrédients (les matières premières, les outils de production et la force de travail) et à laisser prendre la mayonnaise. Donc il ne se limite pas à faire fonctionner l'échange inégal à la source de la valeur (l'achat de la force de travail et l'appropriation du travail). D'un coté il achète aussi la disponibilité du salarié à obéir pour une partie de sa journée à des consignes et des ordres, ou, pour être plus précis, à suivre des *cours d'action*. Le capitaliste-organisateur obtient, en échange de la rémunération de la force de travail, le *travail* et le *commandement sur le travail* de surcroît.

De l'autre coté, cela a du sens si le capitaliste est en mesure de concevoir l'organisation de la production, et donc des cours d'action pour ses salariés, et, ensuite, de mettre en place, cette organisation et la conduire. Pour cela, il donne mandat à d'autres salariés, d'un coté de concevoir l'organisation du travail et la technologie qui la supporte (bureau de méthodes, bureau d'études, recherche et développement) et de l'autre coté de mettre en place la structure technique et organisationnelle de la production, de surveiller son fonctionnement dynamique (hiérarchie directe) et, en cas de dysfonctionnement ou d'événement perturbateur, de prendre les décisions qui s'imposent et de les implémenter (management réactif). Bien entendu, tout cela marche si le salarié accepte une fois pour toutes et à tout instant, l'hétéro-direction de son cours d'action.

Tout cela est bien connu, véhiculé par de pages de Marx qui on encore une grande force et pertinence. Les phénomènes comme la captation des savoirs artisanaux, leur réification, les processus d'abstraction appliqué à l'organisation vivante du travail concret, l'aliénation, la condition quand même saugrenue de se faire dicter sa conduite par quelqu'un d'autre et par une machine, en échange de pouvoir se reproduire, le sentiment de faire quelque chose dont on ne connaît ni les tenants ni les aboutissants ont été longuement décrit et analysé. Certains sont allés plus loin ont et ont identifié d'abord l'*indifférence* du salarié à l'organisation de la production, issue d'une condition objective d'aliénation, la capacité de réflexion individuelle et collective sur sa propre condition que cette indifférence permet et même encourage et la puissance de développement et de transformation de l'indifférence en *autonomie* de décision et en comportement antagoniste.

Les figures de la soumission formelle et réelle sont parfois présentées uniquement comme figures historiques d'un passage de l'accumulation originelle (soumission purement formelle d'activité artisanales) au capitalisme fordiste (l'apothéose de la soumission réelle). Le sens de l'histoire serait : la captation (et réification) des savoirs artisanaux d'abord et l'application du développement de la science et de la technologie par la suite. Il est plus intéressant d'appliquer ces figures à des formations et à des phénomènes historiquement plus déterminés, comme par exemple, au phénomène de l'*outsourcing*, et aux phénomènes de

précarisation, tendance qui aujourd'hui prend des dimensions énormes sur échelle planétaire et qui inverse la tendance (de la soumission réelle on revient à la soumission formelle).

Marché contre hiérarchie ou contrat de vente contre contrat de travail

La théorie moderne de la firme et des coûts de transactions (Coase, Williamson, Desmetz) reprend en termes moins « critiques » mais néanmoins intéressants la même problématique de la soumission : les figures deviennent le « *marché* » et la « *hiérarchie* », ou, en alternative, le contrat de vente et le contrat de travail. Coase part d'une constatation simple, si le marché est un mécanisme parfait d'allocation de ressources pour la production, pourquoi les firmes (des lieux dans lesquels les ressources sont allouées sur la base de décisions d'une autorité et la force de travail ne négocie pas tout le temps les conditions de sa dépense) existent ? La théorie moderne de la firme conclut que la recherche de l'équilibre entre marché et hiérarchie, et donc des dimensions de la firme (périmètre extérieur de la hiérarchie) est basé sur la minimisation du coût de transaction (le choix make or buy). La recherche de l'équilibre se fait toujours selon des critères *marginalistes* et l'équilibre est atteint toujours comme *équilibre partiel*. (il n'y a pas, à ma connaissance une théorie de l'équilibre général qui inclut les dimensions des firmes).

Ce que nous intéresse ici est la relation entre le jeu de marché (soumission formelle) et hiérarchie (soumission réelle) et *connaissance*. Par exemple, le passage historique de la première forme de soumission formelle à l'usine (première forme de soumission réelle) a demandé la captation des savoirs artisanaux qui ont été formalisés et à nouveau appliqués par la suite à l'organisation du travail. Le capitaliste-marchand n'avait pas le choix : il n'avait pas le maîtrise des savoirs techniques et organisationnels de production, il devait organiser donc leur transfert pour les incorporer dans l'organisation du travail s'il voulait par la suite remplacer le travail savant par le travail « sans phrase ». A l'opposé, aujourd'hui, lorsqu'une entreprise externalise une fonction, une activité, elle le fait toujours parce que le savoir qui préside à cette fonction ou activité est largement diffus, et donc il est possible par ce biais de sortir des rigidités du contrat de travail interne.

Herbert Simon, dans un papier de 1953 (Simon 1953) affronte le problème de la relation de travail salarié et de son contrat de travail incomplet par rapport à la connaissance du cours d'action: le rapport salarial est un rapport contractuel par lequel le salarié cède à la firme qui l'emploie le pouvoir de lui dicter le cours de ses action, pour une portion de sa journée et dans un contexte borné, en échange d'un salaire. Le contrat de travail salarié moderne encadre la soumission réelle du travail au capital.

Simon met en contraste le contrat de travail salarié avec le « contrat de vente », par lequel l'individu, ou un ensemble d'individus échangent le résultat d'un travail avec une somme d'argent, vendent le résultat de leur travail : le contrat de vente est celui qui encadre la forme moderne de soumission formelle.

Simon se pose la question de comment choisir rationnellement entre contrat de vente et contrat de travail, du coté de l'employeur comme du coté de l'employé. Il fait référence explicitement à la théorie des jeux. Lorsque le cours d'action n'est pas précisément déterminé (dans ce contexte cela signifie que, parmi un ensemble finis de cours d'actions possibles, le cours d'action qui sera adopté n'a pas encore été choisi.) le contrat de travail est la meilleure solution pour les deux parties (le jeu n'est pas à somme nulle), car cela permet aux deux parties de commencer à travailler tout en reportant la décision, et donc d'initier le processus de tâtonnement qui conduit à la détermination du cours d'action optimal. Le capitaliste accepte de payer quelqu'un sans savoir établir le cours d'action qu'il achète et le salarié accepte l'incomplétude du contrat de travail, le fait que son cours d'action n'est pas complètement établi par contrat, mais le contrat confère au capitaliste le droit de le lui dicter postérieurement et lui confère l'obligation d'obéir, même si le cours d'action dicté ne le satisfait pas. Dans ce jeu à somme non nulle, l'adoption du contrat de vente dans une situation d'incertitude sur le cours d'action est impossible (toujours dans l'abstrait), car il est impossible de spécifier le cours d'action dans le contrat. A l'inverse, en cas de connaissance parfaite du cours d'actions, le contrat de vente est préférable pour les deux parties. On pourrait rappeler à ce propos la théorie des cours de transactions pour dire que le coût de transaction pour acheter une activité largement connue (dont la connaissance est banalisée) sur le marché est virtuellement très bas, et donc sans doute inférieur au coût d'organisation de cette activité dans la firme.

En fait, le modèle de Simon est un modèle de *planification en situation d'incertitude*. Le modèle décrit une situation dans laquelle pourrait être avantageux de différer la décision (la sélection d'un cours d'action parmi un ensemble de cours d'actions possibles) pour pouvoir profiter d'informations qui seront disponibles seulement postérieurement. Ce retard dans le choix du cours d'action pourrait induire une « préférence pour la liquidité » où la ressource liquide est le temps de travail de l'employé. Donc l'avantage du report de la décision (de choix d'un cours d'action) doit être comparé avec le coût de maintenir des actifs dans une forme liquide.

Le temps de latence, couplé avec la « liquidité » du temps de travail de l'employé sont utilisés pour apprendre, mettent en place une organisation « apprenante », dans laquelle l'acquisition d'informations ultérieures et le retour d'expérience servent à définir et consolider le cours d'action optimal. Si l'apprentissage de l'expérience est nécessaire pour optimiser les choix, le contrat de travail est le meilleur (le seul) cadre pour organiser cet apprentissage.

Dans la discussion précédente sur l'estimation du logiciel, postuler qu'un modèle de relation fonctionnelle entre complexité de la tâche que le logiciel doit implémenter et effort à produire ledit logiciel peut être issu des données historiques et donc de l'expérience, veut dire privilégier le contrat de travail au contrat de vente. En fait l'expérience acquise permet non seulement décider par la suite le meilleur cours d'action, mais aussi de le planifier de façon réaliste.

Le modèle de Simon est finalement très simple et son auteur reconnaît même explicitement la tendance simplificatrice qui réduit un peu abusivement la complexité du réel. Il reconnaît aussi sa limite dans l'utilisation de la rationalité « illimitée » (les agent disposent des informations pour décider et peuvent décider de façon instantanée le choix optimal pour eux, alors que l'adoption d'une approche computationnelle du calcul d'optimum fait rentrer dans le jeu un temps de calcul non nul et les ressources qui lui sont nécessaires. Malgré ses limites, il semble bien s'appliquer aux situations d'apprentissage organisationnel.

Le problème de la production du logiciel est que nous ne sommes dans aucune de ces configurations : la théorie de la complexité algorithmique nous dit que non seulement le cours d'action est inestimable a priori, mais il va aussi le rester, car il n'y a pas d'apprentissage exploitable pour l'estimation. Nous sommes dans une impasse : une situation de planification en situation d'incertitude dans lequel l'incertitude ne pourra jamais être levée !

Production de logiciel comme résolution de problèmes

(à pourvoir)

[La production du logiciel est une activité créatrice, qui se compare à la production d'une théorie et donc à une activité de problem solving, dans laquelle la définition de la tâche que le programme doit implémenter fait partie du problème plus général à résoudre. C'est une activité sociale, dans laquelle est toujours impliquée une communauté beaucoup plus vaste de celle des programmeurs. C'est une activité dans laquelle les supports cognitifs – machines, environnement et outils de développement, réseau – jouent aussi un rôle primordial. Elle ne peut être caractérisée, avec Simon, que par la rationalité limitée, la recherche du *satisfecing*, la rationalité optimisante étant même interdite par le fait qu'il n'existe pas de méthode pour atteindre le *programme élégant*, qui n'est donc pas un objectif. Les principes des méthodes modernes de production du logiciel (*extreme programming*) sont en cohérence parfaite avec les résultats d' « inestimabilité » du logiciel énoncés ci-dessus : l'impossibilité logique des spécifications, un processus incrémental qui mélange en continu définition de la tâche, implémentation du programme et vérification de conformité jusqu'à arriver à des points d'arrêt, toujours provisoires, de *satisfecing* (une version) par rapport à toute la communauté (développeurs et utilisateurs), l'activité de *refactoring* comme activité « de première classe » dans le développement et, pour finir, comme conséquence logique, la renonciation a toute estimation a priori. Par ailleurs la méthode moderne s'inspirent fortement de la pratique des développements communautaires de logiciel ouvert et libre.**]**

Logiciel comme marchandise

(à pourvoir)

[Toujours par rapport au cycle de valorisation A – M … M' – A', comment se caractérise le logiciel comme marchandise (en l'abstrait et en tant qu'objet concret sur support numérique) ? Pour confirmer les hypothèses et les résultats précédents sur la production, il y a en permanence « crise de valorisation », à savoir de l'échange M' – A' (une manifestation retentissante de la crise de valorisation est l'extension colossale de la « piraterie » - en Chine 95% des licences de Windows sont « piratées » !). Le *logiciel sur support numérique* est un bien *non divisible* (à savoir, que son coût de production est indépendant du nombre de « consommateurs », donc il est reproductible de façon illimitée à coût pratiquement nul) *non rival* (à savoir que qu'il n'est pas détruit par l'usage ou consommation, il est donc consommable un nombre illimité de fois) et *non exclusif*, (à savoir qu'il est difficile, voir très difficile d'en exclure l'usage ou la consommation, en générale et des *free riders*, ceux qui n'ont pas « acheté » le droit de le consommer – sur ce dernier point, des dispositif techniques peuvent à un coût élevé et avec beaucoup de difficulté, empêcher la reproduction non autorisée d'objets sur support numérique – c'est le DRM : Digital Right Management). En accord avec la théorie de l'équilibre général, le prix d'équilibre du logiciel – donc d'un bien indivisible et non exclusif - est nul, car il a un coût de production nul – non divisible - et un prix à la consommation nul – il est non rival et non exclusif - donc peut être obtenu gratuitement, donc le consommateur ne paye pas le prix. La divisibilité et la rivalité des biens sont les fondements matériels de la propriété. L'exclusivité est le fondement matériel de l'obligation au respect de la propriété. La soi disant propriété intellectuelle n'a pas de fondement matériel et l'obligation au respect non plus. La propriété intellectuelle est deux fois non fondée – une fois comme propriété et une fois comme intellectuelle. Par ailleurs, sa justification est pratiquement circulaire : la propriété intellectuelle garantirait l'innovation, en garantissant sa rémunération, face à l'incapacité de produire innovation de la part du fonctionnement parfait du marché de l'équilibre général. Il faut moduler le jugement sur l'exclusivité en tenant compte de la « fracture numérique » : en Chine les gens achètent plus les copies « piratées » de Windows que les copies « légales » de Linux car le prix de la copie est pratiquement celui du support (le CD) et la distribution « piratée » de Windows arrive sur un CD alors que la distribution « légale » et « officielle » de Linux est sur 3 CD, et donc coûte trois fois le prix.**]**

La montée en puissance de la production communautaire de logiciel ouvert

Le logiciel ouvert et le logiciel libre

(à pourvoir)

[En parallèle avec la crise endémique de la production capitalistique du logiciel, on assiste depuis désormais vingt ans, à la montée en puissance de la production de logiciel libre (free software) et ouvert (*open source* - code source accessible). Le phénomène part à la marge de la production capitalistique (Stallman – The GNU Project, GNU Emacs) mais s'installe aujourd'hui même au cœur du fonctionnement de l'économie. Un *logiciel ouvert* (open source) est un logiciel dont le code source est accessible sans restriction. Un *logiciel libre* est un logiciel ouvert, dont l'accès et l'utilisation sont réglementés par une licence GPL (General Public License), conçue par la Free Software Foundation, qui, paradoxalement, en restreint l'utilisation de sorte que les œuvres dérivées restent libres et donc réglementées par la même GPL – par exemple un logiciel libre ne peut pas être intégré dans un logiciel propriétaire, dans le sens que l'intégrer le rendrait libre par sa seule présence. En résumé, la licence GPL est une licence « virale », qui prévient la *défection*. La montée en puissance s'exprime à travers trois phénomènes différents qui se nourrissent réciproquement :

- Des logiciels libres ou ouverts comme Linux (système d'exploitation), Apache (serveur Internet), MySql (base de données), PHP (langage de programmation), sendMail (logiciel de courrier électronique) et d'autres sont largement utilisés aujourd'hui dans les entreprises et les administrations (dans le système nerveux de l'économie capitalistique) et leur usage croit ;

- Des sociétés de développement de logiciel, de conseil et d’intégration de système basent une partie grandissante de leur activité professionnelle sur les logiciels libres et ouverts. En fait la typologie est très diversifiée et va des grandes sociétés comme IBM ou Cap Gemini qui basent leurs grands projets d’intégration de système sur les logiciels libres, à des sociétés comme Red Hat, Jboss, MySql qui commercialisent de services à valeur ajoutée au tour du logiciel libre. Dans l’industrie du logiciel on parle aujourd’hui d’un business model qui se construit au tour du logiciel libre et ouvert, et qui est centré sur la vente de services à valeur ajoutée à la place des licenses.

- Des industriels comme IBM et Sun font « cadeau » au monde du logiciel libre et ouvert (par exemple la Fondation Apache) des logiciels sortis d’efforts importants (chifrées en centaines de million de dollars) de leurs laboratoires de recherche et développement. Ce phénomène est particulièrement intéressant : d’un coté il met à mal l’idée que la propriété et « possession » d’objets immatériels à haut contenu de connaissance est la source de l’avantage compétitif, et de l’autre coté est une confirmation empirique de la crise du logiciel : ces acteurs réagissent à cette crise en sortant la production du logiciel et la conséquente nécessaire valorisation du produit de leur business model, qui se construit plutôt sur les services à valeur ajoutée au tour des logiciels devenus libres ou ouverts.

Ces trois phénomènes, qui se présentent de façon plus détaillée dans une grande variété et hybridation des situations, indiquent sans équivoques une tendance lourde : la sortie de la production du logiciel du processus capitalistique de production.**]**

La production communautaire de logiciel ouvert

(à pourvoir)

[Avec Benkler, la production communautaire de logiciel ouvert est caractérisée par le fait qu’elle n’est tenue ni par le *marché* ni par la *hiérarchie*. En paraphrasant, on peut parler d’absence de contrat de travail *et* de contrat de vente, et en fait de contrat tout court. Les participants à un projet de développement de logiciel ouvert ne sont pas liés par un contrat de travail avec une autorité sur le projet, ni par des contrats de vente passés réciproquement. L’absence de contrat élimine la nécessité de *spécifier*, statiquement (contrat de vente) ou dynamiquement (contrat de travail qui encadre les instructions données par la hiérarchie), le processus de production et ses composants (les agents, les efforts, les ressources) pour un objet (le logiciel) dont les spécifications ne sont pas faisables, car nous avons vu que soit sont intraitables soit ont la même taille du programme à réaliser (cela revient à écrire le programme deux fois). Le nécessité de spécification du processus de travail crée selon Benkler deux sources majeures d’inefficience : d’un coté la spécification parfaite et exhaustive implique des coûts de transaction très élevés et est pratiquement inatteignable et, de l’autre coté, les relations contractuelles introduites par la spécification rendent le processus d’identification et d’allocation des ressources pertinentes extrêmement rigide, car justement quadrillé par les contrats (l’employé de la firme va devoir coopérer avec ses collègues, indépendamment de l’adéquation à la tâche, et l’acquisition des ressources externes se fait seulement lorsque le coût de transaction est considéré comme inférieur au gain escompté). La production communautaire de logiciel ouvert permet d’optimiser l’identification et l’allocation des ressources. Pour l’identification des ressources : étant donné la variabilité des individus en termes de talent, expérience, motivation, focalisation, disponibilité, la créativité est une ressource difficile à identifier pour le marché comme pour la hiérarchie. La spécification complète des contrats de vente est intraitable, tandis que la hiérarchie produit inévitablement des pertes d’information. La production paritaire et communautaire fournit un cadre d’auto-identification pour la tâche : les individus ont la meilleure information disponible sur leur adaptation à l’exécution d’une tâche donnée et donc peuvent s’auto-identifier pour la tâche. Naturellement, pour un système d’identification et d’allocation de ressources supérieur à ceux du marché et de la hiérarchie, il est nécessaire d’avoir un système d’autocorrection qui est fondamentalement basé sur la revue paritaire. En résumé, les caractéristiques primaires du processus de production du logiciel ouvert sont :

1. Le résultat de la production, qui est aussi input de sa reproduction (activité de maintenance) est non rival et non exclusif. L’input du processus est lui aussi non rival et non exclusif (information

librement accessible) – on peut penser à la détection, au diagnostic et à la correction des dysfonctionnements. Par opposition, les mécanismes de propriété et de contrat propres aux marchés et hiérarchies introduisent des inefficiences. Loi de Linus (Torvald) : « avec un nombre suffisant d'yeux qui regardent le code, toutes les erreurs remontent en surface ». Loi de Arrow : « *Precisely to the extent that property rights in information are successful, there is an underutilization of information* ».

2. Le processus n'est régi par aucun contrat (ni de travail ni de vente) et donc par aucune des inefficiences en problem solving qui résultent de la rigidité des contrats et de hiérarchies.

3. L'identification et l'allocation de la force de travail (ressource centrale du processus), basée sur l'auto-identification et auto-allocation, est la source primaire d'efficience par rapport aux marchés et hiérarchies. La production paritaire et communautaire de logiciel ouvert crée une meilleure information en temps réel sur la disponibilité de force de travail adaptée (y compris de sa motivation et focalisation) et tend à allouer de façon optimale l'effort créatif. Cela permet aussi une coordination et coopération optimale.

4. Le coût du support physique (ordinateurs, réseaux) de la production d'information a diminue à une grande vitesse (les lois de Moore et de Gilder). On peut ajouter aussi la loi de Metcalfe (inventeur d'Ethernet) qui dit que la valeur ajoutée d'un réseau croit au carré du nombre de personnes connectées. Ceci est une condition qui permet la libre adhésion, la modularité poussée des projets, la possibilité d'intégrer des contributions réparties sur la planète même minimales. Ce sont des conditions qui permettent lé développement de la production paritaire et communautaire du logiciel ouvert

Faut-il, dans ce cadre, affronter les questions de la motivation et de l'organisation, qui tombent sous la critique traditionnelle de la « tragédie du commun » comprenant deux objections « canoniques » : (i) manque de motivation : personne n'« investit » dans un projet si l'on ne peut pas s'*approprier* du résultat ; (ii) manque d'organisation : personne n'a le pouvoir d'imposer la coordination et d'organiser la coopération. Pour la motivation, le manque de rémunération monétaire directe n'empêche pas de business model des entreprises de services basées sur l'open source basé sur l'appropriation indirecte du résultat qui n'est pas basée sur l'exclusivité et donc l'exclusion du bien (en termes de compétence). Par ailleurs, si l'on veut caractériser la participation à un projet de logiciel ouvert en termes d'échange, l'échange est *totalement inégal* dans le sens que le participant reçoit (en termes de quantité d'information) énormément plus de ce qu'il donne (sa contribution) et il apprend beaucoup plus de ce qu'il enseigne (Barbrook).]

La production monopolistique de logiciel

(à pourvoir)

[Le monopole (Microsoft) comme unique alternative au logiciel ouvert et libre. Le monopole tient exclusivement avec le « rendements croissants d'adoption », la création d'un monde qui enferme l'utilisateur.**]**

Le brevets

(à pourvoir)

[Les brevets logiciels (algorithmes et business methods) comme nécessaire évolution du monopole. Le passage du profit à la rente. Brevet de « formes de vie » (business methods).**]**

Conclusions

(à pourvoir)