Le programmeur modeste

Edsger W. Dijkstra

C'est par une série de coïncidences que j'ai embrassé la profession de programmeur, officiellement le premier matin du printemps de 1952, et pour autant que j'aie pu chercher, je fus le premier Hollandais à le faire dans mon pays. Rétrospectivement, la chose la plus étonnante fut la lenteur avec laquelle la profession de programmeur émergea, du moins dans ma région du monde, une lenteur qui est maintenant difficile à croire. Mais j'ai deux souvenirs très nets de cette période qui confirment cette lenteur sans aucun doute.

Après avoir programmé pendant trois ans, j'eus une discussion avec A. van Wijngaarden qui était alors mon patron au Centre de Mathématiques d'Amsterdam, une discussion pour laquelle je lui resterais à jamais reconnaissant. La question était que, étant simultanément supposé étudier la physique théorique à l'université de Leiden, et trouvant les deux activités de plus en plus difficiles à concilier, je devais me décider soit à arrêter de programmer et à devenir un véritable, un respectable théoricien de la physique, soit à achever mes études de physique avec le minimum d'effort et à devenir... oui quoi ? Un programmeur ? Mais était-ce une profession respectable ? Car après tout, qu'était-ce que la programmation ? Où était le solide champ de connaissances qui pouvait la soutenir comme une discipline intellectuellement respectable ? Je me rappelle très vivement comme j'enviai mes collègues travaillant sur le matériel, qui, interrogés sur leurs compétences professionnelles, pouvaient au moins mettre en avant leur pleine connaissance des tubes à vide, des amplificateurs et tout le reste, alors que j'avais l'impression que, confronté à cette question, je n'aurais rien eu à répondre. Empli de doutes, je frappais à la porte du bureau de Wijngaarden et lui demandais si je pouvais "lui parler quelques instants" ; quand je quittais son bureau plusieurs heures plus tard, j'étais une autre personne. Car, après avoir patiemment écouté mes problèmes et dit, d'accord avec moi, qu'il n'y avait pas vraiment de discipline de la programmation, il se mit à expliquer posément que les ordinateurs automatiques étaient là pour durer, que nous n'en étions qu'au début et que peut-être je serais une des personnes appelées à faire de la programmation une discipline respectable dans les années à venir. Ce fut un tournant de ma vie, et je me débarrassais de mes études de physique le plus rapidement possible. Une des morales de cette histoire est, bien sûr, que nous devons faire très attention lorsque nous donnons des conseils à nos cadets : parfois ils les suivent !

Deux ans plus tard, en 1957, je me mariais et, la cérémonie hollandaise du mariage demandant que vous déclariez votre profession, je déclarais que j'étais programmeur. Mais les autorités municipales de la ville d'Amsterdam ne l'acceptèrent pas, considérant qu'une telle profession n'existait pas. Et, croyez-le ou non, sous le titre "profession", mon acte de mariage porte la mention ridicule de "physicien théorique" !

Voilà pour la lenteur avec laquelle j'ai vu la profession de programmeur émerger dans mon propre pays. Depuis lors, j'ai vu une plus grande partie du monde et mon impression générale est que dans les autres pays, avec quelques décalages de dates, le modèle de croissance a été en grande partie le

Référence : exposé d'Edsger Dijkstra en 1972 lors de la remise de sa récompense Turing
https://amturing.acm.org/lectures.cfm, également ici
https://artechnic.wordpress.com/2014/08/27/le-programmeur-modeste/.
Transcription en LaTeX : Denise Vella-Chemla, décembre 2023. Identité du traducteur ?

même.

Laissez-moi essayer de décrire la situation dans ces jours anciens avec un peu plus de détail, dans l'espoir que cela nous donne une meilleure compréhension de la situation aujourd'hui. En poursuivant notre analyse, nous verrons comme tant de malentendus à propos de la véritable nature de l'activité de programmation peuvent remonter à ce passé maintenant lointain.

Les premiers ordinateurs électroniques automatiques étaient tous des machines uniques, en un seul exemplaire, et ils étaient tous situés dans un environnement imprégné de l'ambiance enthousiasmante d'un laboratoire expérimental. Quand l'idée de l'ordinateur automatique apparut, sa réalisation fut un formidable défi pour la technologie électronique disponible à l'époque, et une chose est certaine, nous ne pouvons pas nier le courage des équipes qui décidèrent de se lancer dans la construction d'un équipement aussi fantastique. Car c'était des équipements fantastiques ; rétrospectivement, on peut seulement s'émerveiller que ces premières machines aient seulement fonctionné, du moins parfois. La tâche écrasante était de mettre et de conserver ces machines en état de fonctionnement. La préoccupation envers les aspects matériels du calcul automatique se reflète toujours dans les noms des plus vieilles sociétés scientifiques du secteur, comme l'Association pour l'Outillage Informatique (Association for Computing Machinery - ACM) ou la Société Britannique de l'Ordinateur (British Computer Society), des noms faisant directement référence à l'équipement matériel.

Et le pauvre programmeur ? Eh bien, à la vérité on le remarquait à peine. D'abord, les premières machines étaient si massives qu'elles étaient pratiquement impossibles à déplacer, et de plus elles demandaient une maintenance si considérable qu'il est naturel que l'endroit où les gens essayaient de les utiliser était le laboratoire même où elles avaient été mises au point. Deuxièmement, son travail à peu près invisible était sans prestige, vous pouviez montrer la machine aux visiteurs et c'était plus spectaculaire, de plusieurs ordres de grandeur, que quelques feuilles de listing. Mais le plus important de tout était que le programmeur lui-même avait une opinion très modeste de son travail, toute la signification de son travail venait de l'existence de cette prodigieuse machine. Parce que cette machine était unique, il ne savait que trop bien que ses programmes n'avaient qu'une portée locale et aussi, puisqu'il était bien évident que cette machine aurait une durée de vie limitée, il savait que très peu de son travail aurait une valeur durable. Finalement, il y a un autre aspect des circonstances qui avait une influence profonde sur l'attitude du programmeur envers son travail ; d'un côté, en plus d'être peu fiable, sa machine était habituellement trop lente et sa mémoire trop petite, il était à l'étroit, alors que de l'autre son code d'instruction en général plutôt singulier laissait le champ libre aux constructions les plus inattendues. Et à cette époque beaucoup de programmeurs astucieux retiraient une immense satisfaction intellectuelle des astuces ingénieuses grâce auxquelles ils parvenaient à faire tenir l'impossible dans le cadre contraignant de leur équipement.

Deux opinions concernant la programmation datent de cette époque. Je les mentionne simplement, j'y reviendrais tout à l'heure. L'une était qu'un programmeur vraiment compétent devait être amateur d'énigmes et d'astuces ; l'autre était que la programmation n'était rien de plus que l'optimisation du processus de calcul, dans un sens ou dans l'autre.

Cette dernière opinion était le résultat des circonstances fréquentes où l'équipement disponible était véritablement très contraignant, et l'on rencontrait souvent l'espérance naïve qu'une fois que des machines plus puissantes seraient disponibles, la programmation ne serait plus un problème, puisqu'à ce moment-là l'effort pour pousser la machine à la limite de ses capacités ne serait plus nécessaire, et c'était bien là tout ce en quoi la programmation consistait, non ? Mais au cours des décennies suivantes, quelque chose de complètement différent arriva, des machines plus puissantes apparurent, pas seulement plus puissantes d'un ordre de magnitude, mais plus puissantes de plusieurs ordres de magnitude. Mais au lieu de nous retrouver dans un état de félicité éternelle où tous les problèmes de programmation seraient résolus, nous nous sommes retrouvés dans la crise du logiciel jusqu'au cou ! Pourquoi ?

Il y a une raison mineure : par certains aspects, les appareils modernes sont tout simplement plus difficiles à maîtriser que les anciens. Premièrement, nous avons les interruptions d'entrée/sortie qui se produisent à des moments imprévisibles et non reproductibles, à comparer avec les anciennes machines séquentielles qui prétendaient être des automates entièrement déterministes, cela a été un changement très important et les cheveux gris de nombreux programmeurs système sont là pour témoigner que nous ne devons pas prendre à la légère les problèmes logiques créés par cette caractéristique. Deuxièmement, nous avons des machines équipées de plusieurs niveaux de stockage, ce qui nous pose des problèmes de stratégie de gestion qui restent encore insaisissables, malgré une vaste documentation sur le sujet. Voilà pour les complications supplémentaires dues aux changements structurels des machines mêmes.

Mais j'ai appelé cela une raison mineure, la raison majeure est... que les machines sont devenues plus puissantes de plusieurs ordres de magnitude ! Pour le dire de manière abrupte : tant qu'il n'y avait pas de machines, la programmation n'était pas du tout un problème, quand nous avions quelques ordinateurs de faible puissance, la programmation devint un léger problème, et maintenant que nous avons des ordinateurs colossaux, la programmation est devenue un problème tout aussi colossal. En ce sens, l'industrie de l'électronique n'a pas résolu un seul problème, elle en a seulement créé, elle a créé le problème consistant à utiliser ses produits. Dit d'une autre manière, alors que la puissance des machines disponibles croissait d'un facteur de plus de mille, les usages que voulait en faire la société croissaient en proportion, et ce fut le pauvre programmeur qui trouva son travail pris dans cette tension entre les moyens et les fins. La puissance accrue du matériel, combinée à l'augmentation peut-être encore plus importante de sa fiabilité, rendait réalisables des solutions que le programmeur n'aurait même pas osé rêver quelques années auparavant. Et maintenant, quelques années plus tard, il devait les rêver et, bien pire, il devait transformer ces rêves en réalité ! Est-ce si étonnant que nous nous soyons retrouvés dans une crise du logiciel ? Non, certainement pas, et comme vous pouvez le deviner, cela avait même été prédit bien à l'avance, mais le problème avec les prophètes mineurs, bien sûr, c'est qu'on ne sait vraiment qu'ils avaient raison que cinq ans après.

Puis, au milieu des années soixante, quelque chose de terrible arriva : les ordinateurs de la prétendue troisième génération firent leur apparition. La littérature officielle nous dit qu'un des objectifs majeurs de leur conception était leur ratio prix/performance. Mais si vous prenez comme critère de "performance" le cycle d'activité des divers composants de la machine, vous pouvez très bien vous retrouver avec un modèle de conception dans lequel la part la plus grande de votre objectif de performance est atteinte par des activités de nettoyage interne d'une nécessité douteuse. Et si votre

définition du prix est le prix à payer pour le matériel, vous pouvez très bien vous retrouver avec un modèle terriblement difficile à programmer : par exemple, le code d'instruction peut être tel qu'il force le programmeur ou le système à prendre des décisions de lien anticipé présentant des conflits qui ne peuvent réellement pas être résolus. Et dans une large mesure, ces possibilités déplaisantes semblent s'être réalisées.

Quand ces machines furent annoncées et que leurs caractéristiques fonctionnelles furent connues, cela chagrina un certain nombre d'entre nous, en tout cas moi. On pouvait raisonnablement s'attendre à ce que des machines de ce type inondent la communauté informatique, il était donc d'autant plus important que leur conception soit aussi correcte que possible. Mais le modèle de conception renfermait des défauts si importants que j'eus l'impression que le progrès de l'informatique avait été retardé d'un seul coup d'au moins dix ans : ce fut à ce moment que je connus la semaine la plus sombre de toute ma vie professionnelle. Le plus affligeant peut-être est que, même après toutes ces années d'expériences frustrantes, il y a toujours tant de gens qui croient honnêtement que quelque loi de la nature nous dit que les machines doivent être ainsi. Ils font taire leurs doutes en considérant la quantité de ces machines qui s'est vendue, et en retirent la fausse impression de sécurité qu'après tout, la conception n'a pu en être si mauvaise. Mais en y regardant de plus près, cette ligne de défense à la même force de conviction que l'argument selon lequel fumer des cigarettes serait bon pour la santé puisque tant de gens le font.

C'est à cet égard que je regrette qu'il ne soit pas dans l'habitude des journaux scientifiques dans le domaine de l'informatique de publier des tests des ordinateurs nouvellement annoncés de la même manière que nous examinons les publications scientifiques, un examen des machines serait au moins aussi important. Je dois d'ailleurs faire un aveu : au début des années soixante, j'écrivis un tel examen dans l'intention de le soumettre aux CACM, mais malgré le fait que les rares collègues à qui le texte fut envoyé pour avis me poussaient tous à le faire, je ne l'osais pas, craignant que les difficultés auxquelles j'aurais, ou auxquelles la commission éditoriale aurait, à faire face ne s'avèrent trop grandes. Cette mise à la trappe fut de ma part un acte de lâcheté que je me reproche de plus en plus. Les difficultés que j'anticipais découlaient de l'absence de critères généralement acceptés et, bien que je fusse convaincu de la validité des critères que j'avais choisi d'appliquer, je craignais que mon article ne soit refusé ou écarté comme "une question de goût personnel". Je crois toujours que de tels articles seraient extrêmement utiles et j'espère les voir apparaître, car leur acceptation serait un signe sûr de maturité de la communauté informatique.

La raison pour laquelle je me suis attardé sur le domaine du matériel est que j'ai le sentiment qu'un des aspects les plus importants de tout outil informatique est son influence sur les habitudes de pensée de ceux qui essaient de l'utiliser, et que j'ai des raisons de croire que cette influence est bien plus importante qu'on ne le pense généralement. Portons maintenant notre attention sur le domaine du logiciel.

Ici, la diversité est si grande que je dois m'en tenir à quelques grandes avancées. Je suis tout à fait conscient du caractère arbitraire de mon choix et je vous prie de ne pas en tirer de conclusion en ce qui concerne l'appréciation que je fais des nombreux travaux qui ne seront pas mentionnés.

À l'origine, il y eut l'`EDSAC` à Cambridge, en Angleterre, et je pense qu'il est assez impressionnant

que dès le début la notion de bibliothèque de sous-programmes jouât un rôle central dans la conception de cette machine et de la manière dont elle devait être utilisée. Près de 25 ans ont passé et l'informatique a énormément changé, mais la notion de logiciel de base est toujours là, et la notion de sous-programme fermé est toujours un des concepts clés de la programmation. Nous devrions reconnaître le sous-programme fermé comme une des plus grandes inventions du logiciel ; il a survécu à trois générations d'ordinateurs et il survivra à quelques autres, car il permet l'implémentation d'un de nos motifs d'abstraction de base, malheureusement, son importance a été sous-estimée dans la conception des ordinateurs de troisième génération, dans lesquels le grand nombre de registres explicitement nommés de l'unité arithmétique entraîne un coût important d'utilisation du mécanisme de sous-programme. Mais même cela n'a pas tué le concept de sous-programme, nous ne pouvons qu'espérer que la mutation ne s'avèrera pas héréditaire.

La deuxième grande étape dans le domaine du logiciel que j'aimerais mentionner est la naissance du `FORTRAN`. À ce moment, c'était un projet d'une grande témérité et les personnes qui en sont responsables méritent la plus grande admiration de notre part. Il serait tout à fait injuste de leur en reprocher des limites qui n'apparurent qu'après une décennie d'usage intensif : les équipes faisant avec succès des prévisions pour dix ans sont plutôt rares ! Rétrospectivement, nous devons juger le `FORTRAN` comme une technique de programmation réussie, mais avec très peu d'aide efficace à la conception, des aides pour lesquelles il y a maintenant un besoin si urgent qu'il est temps de le considérer comme obsolète. Plus vite nous pourrons oublier que le `FORTRAN` a jamais existé, mieux ce sera, car il n'est plus adapté comme véhicule pour la pensée : il gâche notre intelligence, il est trop risqué et donc trop cher à utiliser. Le destin tragique du `FORTRAN` a été sa large diffusion, enchaînant mentalement des milliers et des milliers de programmeurs à nos erreurs passées. Je prie tous les jours qu'un plus grand nombre de mes collègues programmeurs trouve des moyens de se libérer de la malédiction de la compatibilité.

Le troisième projet que je ne voudrais pas oublier est `LISP`, une entreprise fascinante d'une nature complètement différente. Avec quelques principes de bases simples comme fondations, il a montré une stabilité remarquable. De plus, `LISP` a été le support d'un nombre considérable de certaines de nos applications en un sens les plus sophistiquées. `LISP` a été décrit en plaisantant comme "la manière la plus intelligente de mal employer un ordinateur". Je considère cette description comme un grand compliment, car elle exprime tout un parfum de libération, il a permis à un grand nombre des humains les plus doués de concevoir des pensées auparavant impossibles.

Le quatrième projet à mentionner est `ALGOL 60`. Alors que jusqu'à aujourd'hui les programmeurs en `FORTRAN` ont encore tendance à concevoir leur langage de programmation en fonction de l'implémentation spécifique qu'ils utilisent - d'où la prévalence des dumps en octal ou en hexadécimal - alors que la définition de `LISP` est toujours un mélange étrange de ce que signifie le langage et de la manière dont le mécanisme fonctionne, le fameux Rapport sur le Langage Algorithmique `ALGOL 60` est le fruit d'un authentique effort pour amener l'abstraction un pas vital plus loin et de définir un langage de programmation d'une manière indépendante de l'implémentation. On pourrait d'ailleurs penser que ses auteurs ont si bien réussi leur tâche qu'ils ont fait naître de sérieux doutes quant à la possibilité de simplement l'implémenter ! Le rapport démontrait magistralement la puissance de la méthode formelle `BNF`, maintenant bien connue sous le nom de Backus-Naur-Form, et le pouvoir de l'anglais attentivement exprimé, du moins entre les mains de quelqu'un d'aussi brillant que Peter

Naur. Je crois qu'il est juste de dire que seuls quelques rares documents aussi courts que celui-ci ont eu une influence aussi profonde sur la communauté informatique. La facilité avec laquelle les termes d'ALGOL et ALGOL-like, la marque n'étant pas déposée, ont été utilisés dans les années qui suivirent pour donner un peu de son prestige à des projets plus jeunes qui n'avaient parfois pas grand-chose à voir, est un hommage quelque peu déplacé à sa grandeur. La force de la notation BNF comme outil de définition est la raison de ce que je considère comme une des faiblesses du langage : une syntaxe trop élaborée et pas très systématique pouvait désormais être concentrée dans les limites de quelques pages à peine. Avec un outil aussi puissant que la notation BNF, le Rapport sur le Langage Algorithmique ALGOL 60 aurait dû être bien plus court. De plus, je suis de plus en plus sceptique quant au mécanisme de paramètres d'ALGOL 60, il permet au programmeur une telle liberté combinatoire que son usage requiert une grande discipline de la part du programmeur. Coûteux à implémenter, il semble également dangereux à utiliser.

Finalement, bien que le sujet ne soit pas agréable, je dois mentionner PL/1, un langage de programmation pour lequel la documentation de définition est d'une taille et d'une complexité effrayantes. Utiliser PL/1 doit être comme piloter un avion avec 7000 boutons, interrupteurs et manettes à manipuler dans le cockpit. Je ne vois absolument pas comment nous pourrions garder la maîtrise intellectuelle de nos programmes en construction quand, par son seul caractère baroque, le langage de programmation - notre outil de base ! - échappe déjà à notre intellect. Et si je dois décrire l'influence de PL/1 sur ses utilisateurs, la métaphore la plus proche qui me vient à l'esprit est celle d'une drogue. Je me souviens d'une allocution en défense de PL/1, prononcée par un homme se présentant comme un de ses utilisateurs assidus lors d'une conférence sur les langages de programmation de haut niveau. Lors de son allocution d'une heure en célébration de PL/1, il trouva le moyen de réclamer l'ajout d'environ cinquante nouvelles fonctionnalités, ignorant le fait que la source principale de ses problèmes pouvait bien être que le langage contenait déjà bien trop de "fonctionnalités". L'orateur présentait tous les signes affligeants de l'accoutumance, réduit à un état de stagnation mentale dans lequel il ne pouvait que demander plus, plus, plus... Alors que le FORTRAN a été appelé une maladie infantile, PL/1 au complet, avec les caractéristiques de croissance d'une dangereuse tumeur, pourrait bien s'avérer une maladie fatale.

Voilà pour le passé. Mais il ne sert à rien de faire des erreurs, à moins que nous soyons capables d'en tirer des enseignements. En fait, je crois que nous avons appris tellement qu'en quelques années, la programmation peut devenir une activité largement différente de ce qu'elle a été jusqu'à maintenant, si différente que nous ferions mieux de nous préparer au choc. Laissez-moi vous présenter une esquisse d'un des futurs possibles. Au premier abord, cette vision de la programmation dans ce qui est peut-être déjà le futur proche vous semblera peut-être complètement fantaisiste. J'y ajouterais donc les considérations pouvant laisser croire que cette vision pourrait être une possibilité bien réelle.

Cette vision est que, bien avant la fin des années soixante-dix, nous serons capables de concevoir et d'implémenter le type de systèmes qui est maintenant à la limite de nos capacités de programmation pour un faible pourcentage en années-hommes de ce qu'ils nous coûtent actuellement, et que de plus, ces systèmes n'auront virtuellement aucun bogue. Ces deux améliorations sont liées. Dans cet aspect, le logiciel semble être différent de bien d'autres produits pour lesquels la règle est qu'une plus grande qualité entraîne un plus haut coût. Ceux qui veulent vraiment des logiciels fiables découvriront qu'ils doivent trouver des moyens d'éviter la majorité des bogues d'abord, et

qu'en conséquence le processus de développement sera moins coûteux. Si vous voulez des programmeurs plus efficaces, vous découvrirez qu'ils ne doivent pas perdre leur temps à déboguer, ils doivent commencer par éviter d'introduire des bogues. En d'autres termes, les deux objectifs mènent au même changement.

Un changement aussi radical sur une si courte période serait une révolution, et les chances qu'il se produise doivent sembler négligeables à tous ceux qui fondent leurs attentes du futur sur une légère approximation du passé récent - faisant appel à quelque loi non écrite de l'inertie culturelle et sociale. Mais nous savons tous que parfois, des révolutions se produisent bel et bien ! Quelles sont les chances que celle-ci se produise ?

Il semble qu'il y ait trois conditions principales devant être satisfaites. Le besoin de changement doit être reconnu largement dans le monde ; deuxièmement, le besoin économique de changement doit être suffisamment fort ; troisièmement, le changement doit être réalisable techniquement. Laissez-moi discuter de ces trois conditions dans cet ordre.

En ce qui concerne la reconnaissance du besoin d'une plus grande fiabilité du logiciel, je n'escompte désormais plus de désaccord. C'était différent il y a seulement quelques années, parler de crise du logiciel était un blasphème. Le tournant fut la Conférence sur le Génie Logiciel à Garmisch, en octobre 1968, une conférence qui fit sensation, car elle vit la première reconnaissance de la crise du logiciel. Il est généralement reconnu aujourd'hui que la conception d'un système sophistiqué de grande taille est une tâche très difficile et, lorsqu'on rencontre des personnes responsables de tels projets, on les voit très préoccupées, et avec raison, de la question de la fiabilité. En résumé, notre première condition semble satisfaite.

Maintenant, voyons le besoin économique. De nos jours, on entend souvent l'opinion selon laquelle la profession de programmeur avait été surpayée dans les années soixante, et que dans les années à venir, on devait s'attendre à ce que les salaires des programmeurs baissent. En général, cette opinion est exprimée en parlant de la récession, mais elle pourrait être le symptôme d'une prise de conscience différente et plutôt saine, celle que les programmeurs de la décennie précédente n'ont pas fait un aussi bon travail qu'ils auraient dû faire. La société devient insatisfaite des performances des programmeurs et de leurs produits. Mais il y a un autre facteur, d'un poids bien plus important. Dans la situation actuelle, il est plutôt fréquent que pour un système particulier, le prix du développement du logiciel soit du même ordre que le prix du matériel nécessaire, et la société accepte plus ou moins cela. Mais les fabricants de matériel nous disent que dans la décennie à venir, les prix du matériel devraient être divisés par dix. Si le développement logiciel devait continuer d'être le même processus lourd et coûteux qu'il est aujourd'hui, cela entraînerait un déséquilibre total. On ne peut pas attendre de la société qu'elle accepte cela, et en conséquence nous devons apprendre à programmer d'une manière d'un ordre de magnitude plus efficace. Dit d'une autre manière, aussi longtemps que les machines constituaient le plus gros poste du budget, la profession de programmeur pouvait s'en tirer avec ses techniques grossières, mais ce parapluie va se replier très rapidement. En résumé, notre seconde condition semble également satisfaite.

Et maintenant, la troisième condition : est-ce techniquement réalisable ? Je crois que cela pourrait l'être et je vais vous donner six arguments en soutien de cette opinion.

Une étude des structures de programmes a révélé que les programmes - y compris des programmes différents remplissant une tâche semblable et ayant le même contenu mathématique - varient énormément dans leur maniabilité intellectuelle. Un certain nombre de règles ont été découvertes, dont la violation handicape sérieusement ou détruit complètement la maniabilité intellectuelle d'un programme. Ces règles sont de deux sortes. Celles de la première sorte sont faciles à imposer mécaniquement, c'est-à-dire par le choix d'un langage de programmation approprié. Citons comme exemple l'exclusion de l'instruction goto et des procédures renvoyant plus d'une valeur. En ce qui concerne celles de la deuxième sorte, je ne vois quant à moi - mais ce pourrait être dû à un manque de compétence de ma part - aucun moyen de les imposer mécaniquement, car cela semblerait nécessiter une sorte de mécanisme automatique pouvant prouver des théorèmes, mécanisme dont je n'ai pas la preuve de l'existence. En conséquence, pour la période actuelle et peut-être pour toujours, les règles de la seconde sorte se présentent comme des éléments d'une discipline requise de la part du programmeur. Certaines des règles que j'ai en tête sont si claires qu'elles peuvent être enseignées et qu'il n'y a pas de discussion nécessaire pour savoir si un programme donné les respecte ou non. Un exemple est la prescription qu'aucune boucle soit écrite sans fournir de preuve de sa terminaison ni sans préciser la relation dont l'invariance ne sera pas détruite par l'exécution de l'instruction à répéter.

Je suggère maintenant que nous nous restreignions à la conception et à l'implémentation de programmes intellectuellement maîtrisables. Si quelqu'un craint que cette restriction soit si sévère que nous ne puissions la supporter, je tiens à le rassurer : la catégorie de programmes maîtrisables intellectuellement est suffisamment riche pour contenir un grand nombre de programmes tout à fait réalistes pour n'importe quel problème ayant une solution algorithmique. Nous ne devons pas oublier que notre métier n'est pas de réaliser des programmes, notre métier est de concevoir des classes de calculs qui présentent un comportement désiré.

La suggestion de nous restreindre à des programmes maîtrisables intellectuellement est la base des deux premiers de mes six arguments annoncés.

Le premier argument est que, le programmeur n'ayant à considérer que les programmes maîtrisables intellectuellement, il est plus facile de faire face aux alternatives entre lesquelles il doit choisir.

Le deuxième argument est que, aussitôt que nous avons décidé de nous restreindre au sous-ensemble des programmes maîtrisables intellectuellement, nous avons accompli, une fois pour toutes, une réduction radicale du domaine des solutions à prendre en considération. Et cet argument est distinct du premier argument.

Le troisième argument est basé sur l'approche constructive du problème de la justesse d'un programme. Aujourd'hui, la technique habituelle est de réaliser un programme puis de le tester. Mais si tester un programme peut être une technique très efficace pour montrer la présence de bogues, elle est désespérément incapable d'en montrer l'absence. La seule manière efficace de relever significativement le niveau de confiance en un programme est de produire une preuve convaincante de sa justesse. Mais on ne doit pas d'abord réaliser le programme et ensuite prouver sa justesse, car l'exigence de fourniture de preuve ne ferait qu'alourdir le fardeau du pauvre programmeur. Au contraire,

le programmeur doit faire croître la preuve de justesse et le programme de concert. Le troisième argument est essentiellement fondé sur l'observation suivante. Si l'on se demande d'abord quelle serait la structure d'une preuve convaincante et que, l'ayant trouvé, l'on construit un programme satisfaisant les exigences de cette preuve, alors ces préoccupations de justesse s'avèrent constituer une gouverne heuristique très efficace. Par définition, cette approche n'est applicable que si nous nous restreignons aux programmes maîtrisables intellectuellement, mais elle nous fournit un moyen très efficace d'en trouver un satisfaisant parmi ceux-ci.

Le quatrième argument est lié au rapport entre l'ampleur de l'effort intellectuel nécessaire à la conception d'un programme et la taille du programme. Il a été suggéré qu'il y a quelque loi de la nature selon laquelle l'ampleur de l'effort intellectuel nécessaire croît selon le carré de la taille du programme. Mais, Dieu merci, personne n'a été capable de prouver cette loi, et cela parce qu'elle n'est pas nécessairement vraie. Nous savons tous que le seul outil mental par l'usage duquel une capacité de raisonnement finie est capable d'aborder une myriade de cas est appelé "abstraction" ; il en découle que l'exploitation efficace de ses capacités d'abstraction doit être considéré comme une des activités les plus vitales d'un programmeur compétent. À ce propos il est peut-être bon de préciser que l'objet de l'abstraction n'est pas d'être flou, mais de créer un nouveau niveau sémantique auquel on peut être absolument précis. Bien sûr, j'ai tenté de découvrir une cause fondamentale qui empêcherait nos mécanismes d'abstraction d'être suffisamment efficaces. Mais malgré tous mes efforts, je n'ai pas trouvé une telle cause. En conséquence, je m'en tiens au postulat - jusqu'à maintenant non contredit par l'expérience - que par une application appropriée de nos capacités d'abstraction, l'effort intellectuel nécessaire à la conception ou à la compréhension de programmes n'a pas besoin de croître plus que proportionnellement à la taille du programme. Mais un sous-produit de ces recherches pourrait être d'une bien plus grande importance pratique, il est en fait à la base de mon quatrième argument. Ce sous-produit consistait en l'identification d'un certain nombre de modèles d'abstraction qui jouent un rôle vital dans le processus d'ensemble de la composition de programmes. On en sait maintenant assez sur ces modèles d'abstraction pour consacrer une conférence à pratiquement chacun d'entre eux. Ce qui découle de la familiarité et la connaissance consciente de ces modèles d'abstraction, je l'ai compris lorsque j'ai réalisé que, s'ils avaient été largement connus il y a quinze ans, l'étape de la notation `BNF` aux compilateurs dirigés par la syntaxe, par exemple, aurait pu ne prendre que quelques minutes au lieu de plusieurs années. C'est pourquoi je présente notre connaissance récente des modèles d'abstraction essentiels comme le quatrième argument.

Maintenant, le cinquième argument. Il a à voir avec l'influence des outils que nous essayons d'utiliser sur nos habitudes de penser. Je constate l'existence d'une tradition culturelle, dont les racines remontent sans doute à la Renaissance, d'ignorer cette influence, de considérer l'esprit humain comme le maître suprême et autonome de ses artefacts. Mais si je commence à analyser mon mode de penser et celui de compagnons humains, j'arrive, que cela me plaise ou non, à une conclusion totalement différente, à savoir que les outils que nous tentons d'utiliser et le langage ou la notation que nous utilisons pour exprimer ou enregistrer nos pensées sont les facteurs majeurs déterminant tout simplement ce que nous pouvons penser ou exprimer ! L'analyse de l'influence qu'ont les langages de programmation sur les modes de penser de leurs utilisateurs et la reconnaissance qu'aujourd'hui, la matière grise est de loin notre ressource la plus rare, ensemble, nous donne une nouvelle série de repères pour comparer les mérites relatifs de différents langages de programmation. Le program-

meur compétent est pleinement conscient de la taille strictement limitée de son crâne ; il approche donc la tâche de la programmation en toute modestie, et, entre autres choses, il évite les astuces ingénieuses comme la peste. Dans le cas d'un langage de programmation interactif bien connu, j'ai appris de plusieurs sources qu'aussitôt qu'une équipe de programmeurs est équipée d'un terminal y donnant accès, un phénomène spécifique apparaît qui a même un nom bien établi : on l'appelle "les one-liners". Il prend l'une des formes suivantes : un programmeur pose un programme d'une ligne sur le bureau d'un autre et soit lui explique fièrement ce qu'il fait et ajoute la question "Peux-tu faire plus court ?" - comme si cela avait la moindre valeur conceptuelle ! - soit il demande simplement : "devine ce que ça fait". De cette observation, nous devons conclure que ce langage en tant qu'outil est une invitation ouverte aux astuces ingénieuses ; et bien que cet aspect précisément puisse être l'explication d'une partie de son attrait, c'est-à-dire envers ceux qui aiment montrer à quel point ils sont malins, j'en suis désolé, mais je suis forcé de considérer cela comme une des pires choses que l'on peut dire d'un langage de programmation. Une autre leçon que nous aurions dû apprendre du passé récent est que le développement de langages de programmation "plus riches" ou "plus puissants" fut une erreur dans le sens que ces monstruosités baroques, ces conglomérats d'idiosyncrasies, sont véritablement impossibles à gérer aussi bien mécaniquement que mentalement. Je prédis un grand avenir aux langages de programmation très systématiques et très modestes. Quand je dis "modeste", je veux dire que, par exemple, non seulement l'instruction "for" d'`ALGOL 60`, mais aussi la boucle "`DO`" de `FORTRAN` pourrait être rejetées comme trop baroques. J'ai conduit une petite expérience de programmation avec des volontaires très expérimentés, et elle a montré quelque chose de vraiment imprévu et de plutôt surprenant. Aucun de mes volontaires n'a trouvé la solution évidente et la plus élégante. Après analyse approfondie cela s'avéra avoir une origine commune : leur notion de la répétition était si étroitement liée à l'idée d'une variable associée à incrémenter qu'ils étaient mentalement bloqués de la vision de la solution. Leurs solutions étaient moins efficaces, inutilement difficile à comprendre et leurs avaient pris très longtemps à trouver. Ce fut pour moi une expérience révélatrice, mais aussi choquante. En somme, il y a un aspect pour lequel on peut espérer que les langages de programmation de demain différeront de ce à quoi nous sommes habitués aujourd'hui : ils devraient nous inviter à refléter dans la structure de ce que nous écrivons toutes les abstractions nécessaires pour affronter conceptuellement la complexité de ce que nous concevons de manière bien plus étendue que jusqu'à maintenant. Voilà pour la meilleure adéquation de nos futurs outils, base de notre cinquième argument.

Je voudrais en passant faire une mise en garde à ceux qui identifient les difficultés de la programmation avec la lutte contre les insuffisances de nos outils actuels, parce qu'ils pourraient en conclure qu'une fois que nos outils seront adéquats, la programmation ne sera plus un problème. La programmation restera très difficile, car une fois libérés de ces contraintes circonstancielles, nous serons prêts à nous attaquer à des problèmes qui sont aujourd'hui bien au-delà de nos capacités de programmation.

Vous pouvez contester mon sixième argument parce qu'il n'est pas si facile de recueillir des preuves expérimentales en sa faveur, ce qui ne m'empêche pas de croire en sa validité. Jusqu'à présent, je n'ai pas mentionné le mot "hiérarchie", mais je crois juste de dire que c'est un concept clé pour tout système concrétisant une solution correctement décomposée. Je pourrais même aller plus loin et en faire un article de foi, à savoir que les seuls problèmes que nous pouvons résoudre de manière satisfaisante sont ceux qui admettent une solution correctement décomposée. Au premier abord,

cette opinion des limites humaines peut vous sembler être une vue plutôt décourageante du sort qui serait le nôtre, mais je ne le vois pas ainsi, bien au contraire ! La meilleure façon d'apprendre à vivre avec nos limites est de les connaître. D'ici à ce que nous soyons devenus suffisamment modestes pour n'essayer que des solutions décomposées, parce que tous les autres efforts échappent à notre maîtrise intellectuelle, nous devons faire de notre mieux pour éviter toutes les interfaces qui limitent notre capacité à décomposer le système de manière utile. Et je ne peux que m'attendre à ce que cela mène régulièrement à la découverte qu'un problème au départ irréductible peut en fait être décomposé. Quiconque a constaté que l'origine de la majorité des problèmes de la phase de compilation appelée "génération de code" peut être trouvée dans des bizarreries du code source sera capable de penser à un exemple simple de ce que j'ai à l'esprit. L'application plus large de solutions correctement décomposées est mon sixième et dernier argument en faveur de la possibilité de la révolution qui pourrait se produire dans la décennie en cours.

Je vous laisse le choix de décider quelle importance vous allez donner à mes réflexions, je ne sais que trop bien que je ne peux forcer personne à partager mes convictions. Comme chaque révolution, celle-ci provoquera une opposition violente, et l'on peut se demander où sont les forces conservatrices qui essaieront de contrer de tels développements. Je ne m'attends pas à les trouver principalement dans les grandes entreprises, ni même dans l'industrie informatique ; je m'attends plutôt à les trouver dans les institutions d'éducation qui fournissent les formations actuelles et dans ces groupes conservateurs d'utilisateurs d'ordinateurs qui pensent que leurs anciens programmes sont si importants qu'il n'est pas utile de les récrire et de les améliorer. À cet égard, il est triste de constater que dans de nombreuses universités, le choix du centre informatique est trop souvent déterminé par les exigences de quelques applications bien établies mais coûteuses sans considération des difficultés que cela engendre pour des milliers de "petits utilisateurs" voulant créer leurs propres programmes. Trop souvent, par exemple, la physique des hautes énergies semble extorquer à la communauté scientifique le prix de ses équipements expérimentaux restants. La réponse la plus simple, bien sur, est la dénégation pure et simple de la faisabilité technique, mais je crains qu'il ne faille des arguments plutôt solides pour ça. La remarque selon laquelle le plafond intellectuel du programmeur moyen d'aujourd'hui empêchera la révolution d'arriver n'apporte aucun réconfort : les autres programmant de manière bien plus efficace, il est exposé à être écarté de toute manière.

Il pourrait aussi y avoir des entraves d'ordre politique. Même si nous savons comment former les programmeurs professionnels de demain, il n'est pas certain que la société dans laquelle nous vivons nous laissera le faire. Le premier effet de l'enseignement d'une méthodologie - plutôt que la propagation de la connaissance - est d'améliorer les capacités des plus capables, accentuant les différences d'intelligence. Pour une société dans laquelle le système éducatif est utilisé comme un instrument visant à établir une culture homogène, dans laquelle la crème est empêchée d'atteindre le sommet, l'éducation de programmeurs compétents pourrait être politiquement désagréable.

Laissez-moi conclure. Les ordinateurs automatiques sont avec nous depuis un quart de siècle. Ils ont eu un grand impact sur notre société en qualité d'outils, mais dans ce rôle, leur influence ne sera qu'une ride à la surface de notre culture en comparaison de l'influence bien plus profonde qu'ils auront en qualité de défi intellectuel sans précédent dans l'histoire culturelle de l'humanité. Les systèmes hiérarchiques semblent avoir cette propriété, qu'une chose considérée comme entité indivisible à un niveau est considérée comme un objet composite au niveau inférieur, plus

détaillé ; en conséquence, la granularité spatiale et temporelle décroît d'un ordre de magnitude lorsque nous portons notre attention d'un niveau à celui immédiatement inférieur. Nous pensons les murs en terme de briques, les briques en terme de cristaux, les cristaux en terme de molécules, etc. Il en résulte que le nombre de niveaux pouvant être distingués pertinemment dans un système hiérarchique est à peu près proportionnel au logarithme du ratio entre le plus gros et le plus fin niveau de granularité, et donc, à moins que le ratio ne soit très grand, nous ne pouvons pas prévoir de nombreux niveaux. Dans la programmation informatique, notre élément de construction de base a une granularité temporelle de moins d'une microseconde, mais nos programmes peuvent prendre des heures de temps de calcul. Je n'ai pas connaissance d'une autre technologie couvrant un ratio de 10^{10} ou plus : l'ordinateur, en raison de son extraordinaire rapidité, semble pour la première fois nous fournir un environnement où des artefacts hautement hiérarchisés sont à la fois possibles et nécessaires. Ce défi, à savoir la confrontation avec le travail de la programmation, est si unique que cette nouvelle expérience peut nous apprendre beaucoup sur nous-mêmes. Elle devrait approfondir notre compréhension du processus de conception et de création, elle devrait nous donner un meilleur contrôle sur l'organisation de nos pensées. Si ça n'est pas le cas, à mon avis nous ne méritons absolument pas l'ordinateur !

Elle nous a déjà enseigné quelques leçons, et celle que j'ai choisi de souligner dans cette allocution est la suivante. Nous pouvons faire un bien meilleur travail de programmation, à condition que nous approchions cette tâche en toute appréciation de sa formidable difficulté, à condition que nous nous en tenions à des langages de programmation modestes et élégants, à condition que nous respections les limitations intrinsèques de l'esprit humain et approchions cette tâche comme de très modestes programmeurs.