Quels savoir-faire faut-il cultiver pour des candidats aux métiers de l'informatique ?

Bernard Morand

Quels savoir-faire faut-il cultiver pour des candidats aux métiers de l'informatique ?

Bernard Morand

Introduction

En matière de formation de lycéens ou d'étudiants en informatique, il est habituel de dérouler des programmes d'enseignement qui vont des « fondamentaux » aux « applications », si possible déclinés à l'intérieur d'une progression pédagogique marquée par des bornes temporelles. Cette façon d'aborder la question de l'apprentissage de l'informatique a certainement ses avantages sous nombre d'aspects, notamment administratifs ou budgétaires. Cependant du fait que la démarche se focalise sur des *contenus* de formation, elle a tendance à rater une autre exigence, au moins aussi importante, celle des objectifs individuels et sociétaux d'un tel apprentissage. C'est du moins le bilan que nous tirons après deux dizaines d'années d'enseignement de la discipline à des étudiants d'IUT (instituts universitaires de technologie) aussi bien qu'auprès d'élèves – ingénieurs du CNAM (Conservatoire national des arts et métiers).

En effet, l'enseignement de contenus hors contexte prépare assez mal aux métiers qui seront rencontrés dans les entreprises. Il ne permet pas non plus de développer les qualités et les compétences qui permettront ensuite aux informaticiens, lorsqu'ils auront quitté le cursus scolaire, d'améliorer leur savoir-faire par auto-apprentissage. Nous ne sommes pas en train d'ouvrir un plaidoyer en faveur d'un enseignement « pratique » : en informatique plus qu'ailleurs la pratique doit s'appuyer sur une solide formation théorique et, au surplus, la dichotomie théorie/pratique y constitue une mauvaise façon de poser le problème. Pourtant, elle sévit dans les enseignements de l'informatique depuis les débuts de la discipline, celle-ci étant née, par les hasards des besoins du calcul automatique, dans le giron des mathématiques. L'informatique telle qu'elle s'est développée dans la société en est cependant sortie depuis au moins trois décennies sans que son statut dans le système éducatif en ait été véritablement influencé. L'informatique est devenue bien moins une science des algorithmes qu'une science de la conception de logiciels au service d'une demande sociale. En termes d'apprentissages, cette évolution n'a pas été sans conséquences : à côté d'un raisonnement purement *déductif* dans lequel il s'agissait de prouver la conformité des résultats du programme à ses spécifications, s'est développée une démarche *inductive* dans laquelle il faut établir les spécifications elles-mêmes et s'assurer de leur correspondance aux besoins des utilisateurs. On peut même estimer que cette évolution a conduit à procéder de plus en plus fréquemment à des raisonnements d'un troisième type : *l'abduction* [1]. En effet, un projet d'informatisation est le plus souvent un problème singulier, un cas, qui doit inventer ses propres solutions. En plus d'une science des objets abstraits, l'informatique est largement devenue une *science expérimentale*.

C'est dans cet esprit que nous développons ci-dessous sept points qui semblent importants dans toute formation aux métiers de l'informatique. Pas toujours indépendants les uns des autres, ils visent à mettre en évidence des compétences, des aptitudes, des qualités qu'il est nécessaire de développer chez les apprenants. Au passage, ces sept points peuvent constituer un essai de définition d'objectifs parallèles pour d'autres enseignements que ceux de l'informatique proprement dite dans un cursus scolaire ou universitaire préparant aux métiers de l'informatique.

1. Savoir dialoguer

Un programme ou un ensemble de programmes constitue une réponse, partiellement ou totalement automatisée, aux besoins d'un groupe social identifiable au sein d'une collectivité plus ou moins vaste. Il se peut même dans certains cas extrêmes que des groupes antagonistes se trouvent en concurrence dans la définition des besoins d'informatisation. Citons à titre d'exemple le cas du

dossier médical informatisé qui doit concilier les intérêts du corps médical, des organismes de financement, des pouvoirs publics et ceux des patients. Le développeur de logiciels n'a pas en général les compétences associées au métier de médecin, de personnel soignant, de financier ou de responsable d'une politique de santé. Il agit donc nécessairement comme intermédiaire entre des demandes exprimées par les utilisateurs du futur programme (supposées cohérentes) et les fonctionnalités que ce logiciel offrira. Il devra expliquer aux demandeurs ce qu'il est possible d'attendre du produit dans des conditions techniques et de coût qu'il doit spécifier. Non spécialiste du ou des métiers concernés, il devra nécessairement entrer dans un dialogue avec ceux-ci pour comprendre leurs besoins d'une part et pour leur retourner de manière compréhensible les contraintes techniques informatiques associées. Dans les débuts de l'informatique, on a souvent pensé que la partition métier demandeur/spécialiste informatique pouvait s'institutionnaliser en deux entités distinctes, parties prenantes d'un même contrat : « le cahier des charges ». On a ainsi pensé pouvoir fixer une répartition des rôles dans laquelle le demandeur est responsable du contenu de la demande et l'informaticien est en charge de sa réalisation technique. On sait aujourd'hui que cette politique est la plupart du temps vouée à l'échec : le demandeur ne sait pas ce qu'il est en droit d'attendre de l'informaticien et ce dernier se trouve bien en peine de comprendre la nature des problèmes pour lesquels il a été sollicité. Une autre idée fausse de cette politique était de penser qu'il soit possible de déterminer une borne temporelle à partir de laquelle le cahier des charges serait définitif. On s'est rendu compte au contraire que le cahier des charges évolue tout au long du développement du projet informatique, voire même après que le logiciel ait été mis en service. La satisfaction de certains besoins, une fois réalisée, en fait émerger d'autres. Par conséquent, le dialogue entre les responsables de l'expression des besoins et les réalisateurs informatiques est nécessairement permanent, phases de réalisation techniques comprises dans lesquelles les résultats intermédiaires doivent pouvoir être appréciés par les futurs utilisateurs du logiciel.

2. Savoir écouter

Cette compétence est tout simplement une extension de la précédente, savoir dialoguer. Dans l'échange avec les demandeurs, l'informaticien doit être à l'écoute, ne serait-ce que pour comprendre un minimum leurs habitudes de travail, leurs demandes, les concepts et le jargon associés à leur métier. Il est même bon qu'il puisse aller au-delà en faisant émerger dans le dialogue les besoins qui ne viennent pas spontanément à l'esprit des demandeurs. En effet, ceux-ci évoquent volontiers leurs insatisfactions quant au système de travail existant ainsi que les aspects saillants des besoins ; ils pensent moins spontanément aux aspects routiniers ou qui ne leur posent pas de problèmes particuliers. En tant qu'il est une partie externe au fonctionnement habituel du métier concerné, l'informaticien agit – filons la métaphore ! – comme un traducteur ou un psychanalyste. On voit bien que l'informaticien qui se contenterait de développer un logiciel d'après les modes de la technologie informatique du moment se trouverait en porte à faux : il doit garder en permanence une oreille attentive aux indications fournies par les demandeurs.

3. Savoir observer et expérimenter

Aucun logiciel n'est correct du premier coup. Les applications informatiques d'aujourd'hui tiennent de plus en plus de la conception d'une architecture d'éléments à assembler. Elles relèvent de moins en moins de la traditionnelle résolution de problèmes dans laquelle, depuis un point de départ donné (l'énoncé du problème ou cahier des charges), une suite correctement ordonnée de décisions conduirait inévitablement à une solution unique. Au contraire l'accent est mis sur la flexibilité des logiciels, leur adaptabilité et leur capacité à permettre des modifications fonctionnelles. L'informaticien doit donc apprendre à *observer* le comportement de ce qu'il a produit. Il doit en *évaluer* la pertinence par rapport à la demande et en évaluer la qualité notamment en termes d'extensibilité. Nous avons pu remarquer que cette aptitude à observer de manière critique les résultats de son propre travail est une conduite qui ne va pas du tout de soi pour les jeunes étudiants. En effet le système scolaire et universitaire, excessivement focalisé sur la transmission de contenus, renforce deux comportements qui ne sont pas de mise ici : la mesure objective d'un résultat par une

note et l'effectuation de cette mesure par une tierce partie (l'enseignant). D'une part, ce qui est produit ne s'apprécie pas en termes booléens (vrai / faux ; sait / ne sait pas) mais en termes de degré de satisfaction (plus ou moins conforme à une demande). D'autre part, l'évaluation doit être faite par l'apprenant lui-même de manière à pouvoir a) détecter ce qui est améliorable et b) procéder à ces améliorations. Nous sommes donc typiquement dans une démarche d'*expérimentation*. À tel point que dans l'univers professionnel il n'est pas rare de commencer la réalisation par un *prototype* qui permettra d'éclairer la manière d'architecturer le logiciel définitif.

4. Savoir abstraire

Cette compétence est certainement celle qui est la mieux développée dans le système éducatif actuel. Encore faut-il selon nous distinguer entre au moins deux sortes d'abstraction. La plus connue est celle du mathématicien qui se *donne* de pures fictions, par exemple l'objet « Ensemble » ou bien l'objet Nombre « Réel » (sic), sans avoir à rendre compte de leur existence dans le monde dit « réel » justement. De ce donné axiomatique, il déduit des propriétés, des théorèmes ou encore des structures dérivables. La consistance de son travail réside dans la cohérence interne du système, elle-même vérifiable au moyen de preuves. Ses résultats peuvent être employés, appliqués, dans le monde réel, en physique, en statistiques ou en informatique par exemple, mais ceci est une autre affaire. Dans le développement de logiciels, le procédé d'abstraction concerne au contraire le monde réel. Il est semblable au travail du cartographe lorsqu'il cherche à représenter un objet géographique. Le maître mot de cette activité est celui de *modélisation* [2] qui vise à produire une représentation des flux, des données et des processus informationnels réels au moyen d'un système de description (notation) convenu. Dans ce cas, hormis la notation, rien n'est donné : tout est à trouver et c'est pourquoi il y faut savoir observer et dialoguer (cf. 1.). Contrairement à la démarche de résolution de problèmes, il n'y a pas d'énoncé ; une bonne part du travail consiste justement dans l'élaboration de cet énoncé. Certes le développeur informatique fait aussi usage d'abstractions techniques, lesquelles sont des *concepts* propres au métier de la programmation et le plus souvent incorporées dans les structures des langages. Ces abstractions prêtes à l'emploi relèvent d'un savoir plus que d'un savoir-faire. Par contre l'abstraction par modélisation présente des caractères spécifiques du point de vue de son apprentissage : nouvelle expression de phénomènes réels, elle suppose la prise de connaissance d'un terrain à chaque fois différent. Les élèves et étudiants de formation initiale n'ont par définition aucune familiarité préétablie avec ces terrains. C'est pourquoi on peut penser que, à côté des stages de terrain, le système éducatif gagnerait à disposer de laboratoires aptes à la simulation de tels environnements. Avec le bond en avant opéré par les technologies numériques et multimédia, ce devrait être un objectif atteignable.

5. Savoir prendre des risques

Chaque développement de logiciel est singulier même s'il est possible de trouver des règles méthodologiques générales pour les construire et même si l'on emploie des langages de programmation plus ou moins standards. Si tel n'était pas le cas la très ancienne prophétie selon laquelle *« l'informatique n'aura plus besoin de programmeurs »* (parce qu'il existerait des programmes pour faire leur travail à leur place) se serait réalisée. Tel n'est pas le cas. Développer un logiciel revient le plus souvent à fabriquer un produit, une seule fois, dans un contexte donné, sur la commande et la prescription d'un client particulier. Le maximum que l'on puisse faire consiste à réutiliser des éléments qui ont déjà servi pour des clients précédents – quand c'est possible. Il n'existe donc pas de recettes toutes prêtes et il n'y a jamais de garantie que l'on obtiendra le résultat exactement comme il serait souhaité. Il faut donc accepter au cours de l'activité de fabrication de logiciels de (a) prendre des risques et (b) faire des erreurs. Prendre des risques peut consister à explorer une voie nouvelle, à faire d'une manière différente de celle à laquelle on est habitué, etc. Il est clair que la prise de risque et l'acceptation des erreurs ont des conséquences non triviales sur la manière d'enseigner l'informatique. Outre l'indispensable apprentissage de l'autonomie par l'étudiant, il lui faut admettre qu'il peut commettre des erreurs. Mais surtout il faut qu'il apprenne à les reconnaître, les détecter et ensuite les corriger par lui-même. C'est en ce sens que nous faisions

allusion en introduction au caractère expérimental de la discipline : confronter sa propre production à ce qui est attendu et se soumettre au diagnostic de l'utilisateur ; ensuite corriger autant que de besoin.

6. Se confronter au résultat

Contrairement au fonctionnement habituel de l'institution éducative, le résultat attendu de la part de l'apprenant n'est pas un produit qui témoigne de son aptitude à maîtriser un savoir, à résoudre un problème ou à trouver la bonne réponse à des questions. Ce qu'il faudrait obtenir est davantage un comportement autonome, une aptitude à juger soi-même de son produit et à prendre les dispositions nécessaires pour que cette évaluation soit la plus fiable possible. Nous avons constaté que ce genre de comportement n'est absolument pas évident pour la plupart des étudiants. Admettre qu'un travail a pu aboutir à un échec tout en étant évalué positivement parce que la nature et les raisons de cet échec ont été identifiées constitue un réflexe qui ne va pas de soi. Mais admettre aussi que ce n'est pas parce qu'on a « travaillé » que pour autant le compte y est : le compte y est seulement lorsque le résultat a été correctement évalué et si possible valorisé. À cet égard la maxime de Coubertin est particulièrement mal venue. Non, il n'est pas seulement nécessaire de participer, il faut en plus savoir « aller au résultat ».

7. Travailler en équipe

Il est bien loin le temps où l'informaticien travaillait seul devant son écran, seul face à son « système ». Le développeur de logiciels participe à des projets dans lesquels plusieurs personnes coopèrent. Que la répartition des tâches soit plus ou moins imposée ou plus ou moins libre au sein d'une équipe, chacun a en permanence besoin de connaître le travail des autres et son avancement. Le produit lui-même doit pouvoir être accessible à tout autre développeur pour pouvoir en assurer la maintenance. Ceci nécessite un apprentissage de l'organisation du travail à plusieurs auquel la plupart des étudiants sont aussi mal préparés. Tous les enseignants savent que lorsqu'un travail est assigné à un binôme, la répartition consiste le plus souvent à affecter toute la tâche à un seul des deux participants, à charge de revanche pour la fois suivante ! L'organisation du travail collectif doit donc être aussi le sujet des enseignements d'informatique.

Conclusion

Enseigner l'informatique n'est donc pas seulement une affaire de contenus et de programmes. C'est aussi une question de manières de travailler. Il ressort clairement des points précédents qu'une orientation pédagogique favorisant l'apprentissage de ces savoir-faire consiste à plonger les apprenants en situations – simulées ou réelles – de *projets*. Ce type de pédagogie pose des problèmes eu égard au fonctionnement habituel de l'institution scolaire et universitaire : emplois du temps, services des enseignants, méthodes pédagogiques, suivi et évaluation des étudiants, etc. Il présuppose certainement un minimum d'autonomie de la part des apprenants, notamment au lycée. Mais on peut penser qu'il permettrait une meilleure efficacité dans les apprentissages, ceux des contenus théoriques compris.

Bernard Morand
Groupe de recherche en informatique, image,
automatique et instrumentation de Caen (GREYC),
Université de Caen

Paru dans Informatique et progiciels en éducation et en formation, *sous la direction de Georges-Louis Baron, Éric Bruillard et Luc-Olivier Pochon, 2009, coédition ENS de Cachan, IRDP et INRP.*
http://www.inrp.fr/publications/catalogue/web/Notice.php?not_id=BT+069
Ce livre est issu du colloque Didapro 3 organisé à l'université Paris Descartes en 2008.

NOTES

[1] L'abduction a été mise en évidence par le philosophe et logicien américain Charles S. Peirce et se trouve au coeur de sa méthode « pragmatiste ». Elle rend compte de la manière dont nous pouvons formuler des hypothèses. À ce titre il s'agit d'un raisonnement qui n'a qu'une très faible force probante, contrairement à la déduction (qui est certaine) et à l'induction qui mesure sa force par un degré de probabilité. Néanmoins et selon son auteur, c'est le seul mode de raisonnement par lequel une connaissance nouvelle peut se produire.

[2] Les mathématiques font aussi un large usage du terme de modélisation, mais il est remarquable que leurs modèles fonctionnent de manière inversée. Ce ne sont pas des modèles du « réel », mais des modèles de théories qu'ils ont pour fonction de vérifier.

Association EPI
Mai 2010