Apprentissage de la pensée informatique par des étudiants en informatique (au XXIe siècle)

Julien Gossa, Benoît Naegel, Mathieu Zimmermann

Apprentissage de la pensée informatique par des étudiants en informatique (au XXI^e siècle).

Julien Gossa, Benoît Naegel, Mathieu Zimmermann

IUT Robert Schuman, Université de Strasbourg,
72 Route du Rhin, 67411 Illkirch-Graffenstaden, France
gossa@unistra.fr, b.naegel@unistra.fr, ma.zimmermann@unistra.fr

Résumé. Cet article s'intéresse au problème de l'acquisition de la pensée informatique par des étudiants en informatique. Aujourd'hui, les technologies logicielles et matérielles permettent de concevoir rapidement des applications complexes, sans avoir à maitriser l'algorithmique ni rencontrer de problèmes de performances. La pensée informatique n'est donc plus nécessaire à l'exercice du métier d'informaticien. Cet article présente trois expériences pédagogiques visant à accompagner l'acquisition de cette pensée par des étudiants en DUT informatique. L'enseignement principal de cette étude est que l'acquisition de la pensée informatique par des informaticiens est favorisée par des dispositifs rendant cette pensée indispensable à la résolution de problèmes concrets, découverts par l'apprenant lui-même.

Mots-clés pensée informatique, informaticiens, expériences pédagogiques.

1 Introduction

L'apprentissage de la pensée informatique est aujourd'hui un défi enfin reconnu par l'Éducation Nationale [1,2]. Cette pensée a des vertus qui ne sont pas limitées au monde numérique, mais s'ancrent aisément aussi bien dans les tâches quotidiennes que professionnelles. Mais alors que des programmes pédagogiques commencent à voir le jour pour débuter cet apprentissage dès la maternelle, et que les défis s'orientent beaucoup sur l'apprentissage de la programmation à destination des plus jeunes [3], on peut constater un accroissement significatif des difficultés pour les développeurs à s'emparer de cette pensée.

Ce qui peut apparaitre comme paradoxal de prime abord s'explique en grande partie par les évolutions des technologies numériques sur les deux dernières décennies. Au XXième siècle, la pensée informatique était rendue indispensable au développeur pour deux raisons. Premièrement, les technologies disponibles, tant en terme de matériel que de logiciel, étaient plus simples et moins diversifiées. Le développement des fonctionnalités d'une application nécessitait une bonne maitrise de l'algorithmique : pour résoudre les problèmes, le développeur devait souvent développer ses propres versions des structures et algorithmes bien connus. De plus, l'apprentissage des technologies existantes laissait de la place à la pensée informatique dans les formations. Deuxièmement, les ressources des

machines étaient très contraintes, notamment en matière de mémoires et de puissance de calcul : les applications atteignaient rapidement leurs limites, et le développeur devait donc en prendre conscience, ainsi que penser et repenser ses programmes.

Depuis, les technologies de l'informatique ont largement évolué. Les langages de programmation haut niveau (orientés objets, mais aussi python ou perl par exemple) fournissent des abstractions et des implantations natives confortables pour le développeur. Mais ces dernières éloignent le développeur de la réalité de l'exécution de son programme sur la machine. De nombreux *frameworks* (Java EE, .Net, Unity, etc.) mettent à disposition des fonctionnalités préconçues qui permettent au développeur de concevoir rapidement des applications très complexes, mobilisant des technologies très différentes de façon totalement transparente. Les plateformes d'exécution se sont également très largement diversifiées, des plateformes mobiles aux serveurs virtualisés, et n'ont pas forcément besoin d'être connues du développeur. Par exemple, Unity permet de mobiliser GPU et CPU avec un moteur 3D et un moteur physique poussé, dans une application exécutable sur plateformes mobiles, PC et web, sans qu'aucune ligne de code ne soit nécessaire, mais sans empêcher la programmation pour autant. On peut également noter le développement récent d'une myriade de technologies web qui permettent de déployer des applications de plus en plus complexes, faisant intervenir de plus en plus de technologies différentes, logicielles et matérielles, de façon totalement transparente pour le développeur. Ce dernier n'a ainsi plus besoin d'avoir conscience des opérations matérielles que ses programmes mobilisent.

Dans un but de professionnalisation, ces technologies ont pris de plus en plus d'importance dans les programmes de formations des informaticiens, en particulier le DUT informatique. Cela n'a pas seulement mené à diminuer les volumes horaires consacrés aux matières bas-niveau, qui permettent de mieux saisir le fonctionnement de la machine et donc la pensée informatique. Cela a surtout éloigné les étudiants de ces considérations, qui peuvent facilement considérer comme plus gratifiant d'apprendre à concevoir concrètement une application très complexe, que de comprendre intellectuellement les multiples fonctionnements sur lesquels cette application repose.

De plus, la puissance des machines s'est décuplée et, dans le même temps, leur prix a très largement diminué. Le *cloud* permet, en quelques minutes, de disposer de serveurs performants, très bien connectés. Aujourd'hui, les ressources physiques peuvent facilement apparaitre comme infinies : en cas de problème de performance, l'approche la plus rapide et la plus sûre est de surdimensionner la plateforme d'exécution, ce qui se fait à la volée par une interface web. Les limites des machines sont de plus en plus difficiles à atteindre, menant à la disparition des contraintes de performances apparentes, surtout dans le cadre de travaux pratiques. Les développeurs peuvent donc aisément considérer que penser et repenser son programme est devenu inutile, sinon une perte de temps.

En conséquence, les enseignements bas-niveau apparaissent de plus en plus scolaires, déconnectés de la réalité, à des étudiants qui n'ont plus besoin de comprendre les algorithmes pour les utiliser, et qui n'ont plus besoin de comprendre

le fonctionnement de la machine pour obtenir des performances acceptables. On peut aujourd'hui être un développeur efficace sans comprendre la pensée informatique, simplement en maitrisant l'usage des technologies disponibles, sans en comprendre le fonctionnement profond.

Cet article présente trois expériences visant à faire renouer des développeurs avec la pensée informatique menées au sein d'un DUT d'informatique. La première expérience vise à acquérir directement la pensée informatique grâce à des activités débranchées. Les deux suivantes consistent à faire prendre conscience de la pensée informatique au travers des performances applicatives, d'abord sur l'exemple de tris développés par les étudiants, puis sur celui de structures de données natives de langages haut-niveau.

2 Activités débranchées

Trois ans d'affilée, un groupe de cinq étudiants en deuxième semestre a été chargé de monter des ateliers de découverte de l'algorithmique à destination de lycéens visitant l'IUT dans le cadre des journées portes ouvertes. Pour ce faire, ils ont utilisé les activités débranchées proposées par Martin Quinson dans le cadres des Sciences Manuelles du Numérique [4] ; en particulier, ces étudiants ont exploité le jeu de Nim et le plus court chemin.

2.1 Jeu de Nim

Description. Dans ce jeu [5], deux joueurs peuvent à tour de rôle retirer 1, 2 ou 3 allumettes d'un tas de 16 ; le gagnant est celui qui prend la dernière. Ce jeu permet aux joueurs d'appréhender la pensée informatique en découvrant par eux même une stratégie gagnante, à savoir toujours conserver un nombre d'allumettes restantes multiple de 4, suffisamment facile pour la faire mettre en œuvre par un ordinateur.

Les étudiants devaient jouer avec les visiteurs, leur faire découvrir cette stratégie gagnante, puis transcrire cette stratégie en langage informatique.

Constat. Des invariants sur les trois années ont pu être constatés. D'abord, la partie jeu débranché a été systématiquement sous-estimée, considérée comme un prétexte plus qu'une approche. Ensuite, plutôt que de se tourner vers des environnements simples, tels que du C en console affichant des nombres, les étudiants ont instinctivement développé des applications complexes avec interfaces graphiques. Enfin, ils ont fabriqué ces interfaces graphiques à la main, disposant les allumettes une à une, à la souris, plutôt que par du code.

2.2 Plus court chemin

Dans cette activité, le joueur tente de trouver le plus court chemin passant par plusieurs clous plantés dans une planche, à l'aide d'une cordelette. La longueur

restante de la cordelette après être passée par tous les clous représente la qualité de la solution trouvée. L'objectif de cette activité est de montrer qu'il n'existe pas toujours une stratégie simple gagnante : certains problèmes sont si difficiles qu'il faut ruser si on veut les faire résoudre par un ordinateur.

Constat. Si la conclusion de cette activité a été assez bien comprise par les étudiants, certains ont tout de même passé plusieurs heures à chercher effectivement la meilleure solution, sans établir aucune stratégie, dans ce qui s'apparente à de l'optimisation stochastique. D'autres ont décidé de mobiliser les connaissances du domaine en faisant des recherches sur les algorithmes de plus court chemin en vue de les implanter. Encore une fois, leur objectif n'était pas de percevoir la pensée informatique derrière ces approches, mais plutôt d'arriver à une application fonctionnelle, sinon une solution.

2.3 Conclusion

Les activités débranchées n'ont pas permis à des étudiants de second semestre d'acquérir la pensée informatique. Étant dans le cadre de journées portes ouvertes, les étudiants avaient la volonté de faire une démonstration de ce qu'ils savaient faire. L'exercice a permis de montrer que s'ils maitrisent des environnements de développement complexes, ou sont prêts à passer des heures à faire de l'optimisation stochastique à la main, la pensées informatique, pourtant au cœur des activités proposées, leur échappe complètement. Un étudiant, tout à fait représentatif, l'a même exprimé explicitement : « *L'algorithmique ne sert à rien.* ». Après discussion, il fallait comprendre que, d'après lui, l'apprentissage de l'algorithmique est inutile dans la résolution des problèmes que rencontre aujourd'hui un informaticien.

Cela illustre bien la problématique présentée en introduction, mais ne semble pas une approche efficace pour atteindre notre objectif : **Si une solution technologique existe, que la pensée informatique n'est pas nécessaire pour résoudre un problème, alors il sera difficile d'encourager des informaticiens à s'y intéresser.**

Sur la base de cette constatation, il s'agissait de chercher des problèmes dont la résolution ne peut se passer de la pensée informatique. C'est ce que nous avons fait au travers des deux expériences suivantes.

3 Performance des tris

3.1 Dispositif pédagogique

Cette expérience a pour cadre un module de quatrième semestre, comportant 4h de cours magistraux, 8h de travaux dirigés et 16h de travaux pratiques, pour les étudiants ayant choisi l'option Ouverture Scientifique. Ces étudiants sont généralement ceux ayant les meilleures capacités d'abstraction, compte tenu du volume élevé de mathématiques.

Description. Dans ce module, il était proposé aux étudiants d'étudier les performances de différents algorithmes de tri, aidés par un enseignant de mathématiques et un enseignant d'informatique. Pour ce faire, l'enseignant de mathématiques a expliqué le fonctionnement des principaux tris, ainsi que les bases de la complexité algorithmique. Celui d'informatique a montré comment évaluer les performances d'un programme en boîte-noire, à l'aide d'un script shell parcourant ses arguments et stockant les mesures des temps d'exécution dans un fichier plat, ensuite analysé à l'aide d'un script R [6]. Les étudiants devaient coder les tris en C, analyser les résultats préliminaires, puis formuler des hypothèses et recommencer le processus pour vérifier ces hypothèses. L'exemple de ce processus, présenté aux étudiants en *live-coding*, est disponible en ligne [7].

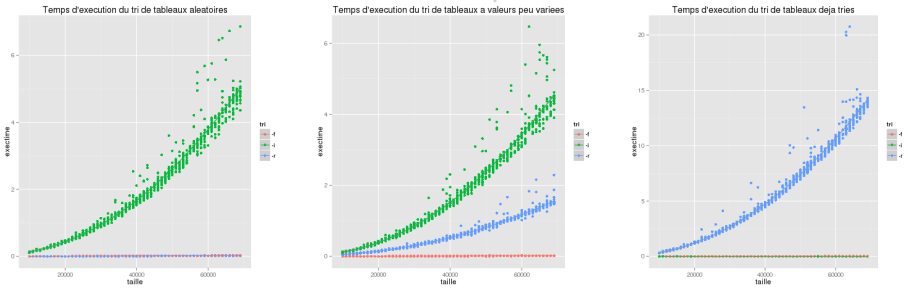

Figure 1. Exemple produit par un étudiant de comparaison des performances de tris en fonction des données à trier

Constat. Ce dispositif a favorisé la mobilisation de la pensée informatique grâce aux aspects suivants :

- l'étude théorique de la complexité des tris a permis de favoriser la compréhension abstraite du problème ;
- l'étude pratique de la complexité des tris, qui a consisté à compter les opérations atomiques en lien avec l'étude théorique, a permis de favoriser la compréhension concrète du problème en mimant le fonctionnement de la machine ;
- l'utilisation d'un langage bas-niveau, très proche de la machine et ne fournissant aucune fonctionnalité élaborée, a forcé les étudiants à connaitre intégralement les opérations menées par la machine, et ainsi favorisé le lien avec l'étude pratique ;
- obtenir des résultats réels a permis d'ancrer ce travail dans la réalité, d'effacer le sentiment d'activité purement scolaire de la part des étudiants.

Cette mise en cohérence, de la théorie abstraite aux résultats concrets, est un terreau fertile pour la pensée informatique. Cependant, elle ne rend pas sa mobilisation indispensable. Cette mobilisation a été favorisée par le dispositif d'hypothèse/vérification. En effet, les étudiants ont pu constater des comportements

étonnants de prime abord. Par exemple, l'étudiant ayant produit les résultats montrés en Figure 1 a pu constater que les performances des tris dépendaient fortement des données à trier : le tri rapide s'avère très efficace pour des valeurs aléatoires, beaucoup moins avec peu de valeurs différentes, et très lent avec des données déjà triées. Pour arriver à comprendre ce type de phénomène, l'étudiant a dû se « mettre à la place de la machine », comprendre finement le comportement des tris, et mettre à l'épreuve ses hypothèses en élaborant de nouveaux tests, qui ne peuvent être issus que d'une pensée informatique.

3.2 Conclusion

Cette expérience a montré qu'**un dispositif pédagogique tissant un lien cohérent de la théorie à la pratique autour d'un problème très précis nécessitant la mobilisation de la pensée informatique pour résoudre des problèmes découverts directement par des étudiants ayant une bonne capacité d'abstraction est efficace pour l'acquisition de la pensée informatique par des informaticiens.**

Cependant, la nécessité d'une bonne capacité d'abstraction peut être un frein pour certains étudiants. C'est pourquoi nous avons mené l'expérience suivante.

4 Performance des structures par défaut

4.1 Dispositif pédagogique

Cette expérience a pour cadre un module de quatrième semestre, comportant 2h de cours magistraux, 2h de travaux dirigés et 6h de travaux pratiques, pour les étudiants ayant choisi l'option Approfondissement Technologique. Le choix de ces étudiants est en partie motivé par l'absence de mathématiques dans ce parcours.

Description. Dans ce module, il était proposé aux étudiants d'étudier les performances de différentes structures de données natives des langages C# ou JAVA, aidés par deux enseignants d'informatique. Pour ce faire, un enseignant a expliqué le fonctionnement théorique des principales structures de données [8] (tableaux, listes chainées et tas binaire de recherche). L'autre enseignant a présenté le même dispositif expérimental que précédemment. Les étudiants devaient élaborer un programme de test des performances des différentes structures, analyser les résultats préliminaires, puis formuler des hypothèses et recommencer le processus pour vérifier ces hypothèses.

Constat. Ce dispositif a favorisé la mobilisation de la pensée informatique grâce aux aspect suivants :

— l'explication théorique, sous l'angle des performances, a permis aux étudiants de revisiter les structures de données sous un angle qu'ils jugent plus utile que l'implantation, puisque ces structures sont déjà disponibles dans les langages qu'ils utilisent ;

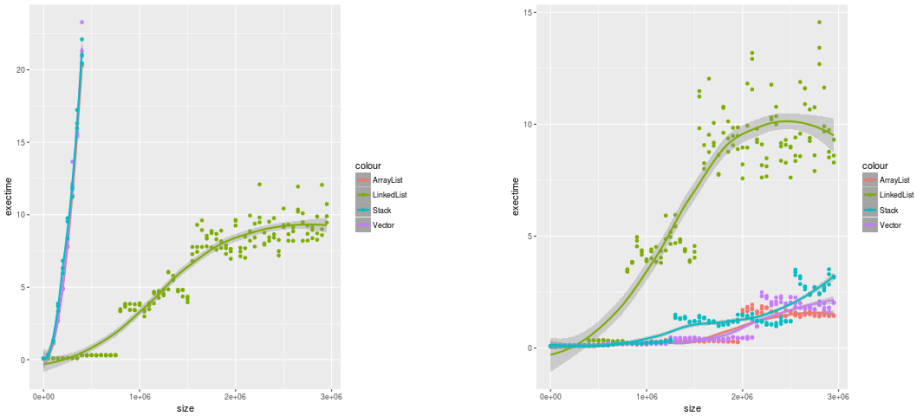

Figure 2. Exemple produit par un étudiant de comparaison des performances de structures de données

— la conception d'un banc de test pour des structures de données n'est pas triviale, et nécessite de mobiliser, au moins superficiellement, une pensée informatique.

Dans cette expérience encore, la mobilisation de la pensée informatique a été essentiellement favorisée par le dispositif d'hypothèse/vérification. Par exemple, l'étudiant ayant produit les résultats montrés en Figure 2 a pu constater que les performances des insertions dans les listes chainées dépendent fortement de l'emplacement de cette insertion (sur la figure : insertion en tête à gauche et en queue à droite), ce qui est cohérent avec la présentation théorique de cette structure. Mais il a pu également constater que l'implantation de cette structure fonctionne en réalité par morceaux, alors que la théorie indique plutôt un fonctionnement continu. La pensée informatique est alors indispensable pour émettre des hypothèses. Si la plupart des étudiants ont observé et noté des phénomènes nécessitant la pensé informatique pour être compris, ils n'ont globalement pas réussi à émettre et vérifier des hypothèses sans l'explication des enseignants.

4.2 Conclusion

Cette expérience montre que sans s'appuyer sur des capacités d'abstraction, il est difficile d'établir une cohérence de la théorie abstraite aux résultats concrets. Il faut donc s'appuyer sur des besoins concrets pour des développeurs, tels que connaitre les performances d'implantations natives. Cependant, **ne s'appuyer que sur des besoins concrets, sans solide étude théorique, permet essentiellement de prendre conscience de l'existence de la pensée informatique, sans réellement acquérir cette pensée.**

5 Conclusion

Cet article s'est intéressé au problème de l'acquisition de la pensée informatique par des étudiants en informatique. Aujourd'hui, les technologies logicielles et matérielles permettent de concevoir rapidement des applications complexes, sans maitriser l'algorithmique ni rencontrer de problèmes de performances. La pensée informatique n'est donc plus nécessaire à l'exercice du métier d'informaticien. Trois expériences ont été présentées. La première était basée sur des activités débranchées centrées sur la pensée informatique. Elle n'a pas permis l'acquisition de cette pensée, mais a permis de confirmer la réalité de la problématique. La seconde était basée sur une étude cohérente, de la théorie à la pratique, d'un problème informatique très précis. Elle a permis à des étudiants ayant de bonnes capacités d'abstraction d'acquérir la pensée informatique. La dernière expérience était basée sur une étude plus concrète. Elle a permis à des étudiants ayant de moins bonnes capacités d'abstraction de prendre conscience de l'existence de la pensée informatique, sans réellement acquérir cette pensée. L'enseignement principal de cette étude est que l'acquisition de la pensée informatique par des étudiants en informatique est favorisée par des dispositifs rendant cette pensée indispensable à la résolution de problèmes concrets.

Références

1. B. Hamon, « Encourager l'initiation au code informatique », http ://www.education.gouv.fr/cid81402/-ecole-numerique-encourager-l-initiation-au-code-informatique.html, 2014
2. N. Vallaud-Belkacem, « La semaine du code en Europe pour promouvoir les actions de découverte et d'apprentissage de la programmation informatique », http ://www.education.gouv.fr/cid83050/-la-semaine-du-code-pour-promouvoir-les-actions-de-decouverte-et-d-apprentissage-de-la-programmation-informatique.html, 2014
3. A. Mcgettrick, R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove et K. Mander, « Grand challenges in computing : Education - a summary », The Computer Journal, vol. 14, n°11, pp. 42-48, 2005.
4. M. Quinson, « Sciences Manuelles du Numérique », http ://people.irisa.fr/Martin.Quinson/Mediation/SMN/, 2016.
5. Jeux de Nim, https ://fr.wikipedia.org/wiki/Jeux_de_Nim.
6. R, https ://www.r-project.org/.
7. J. Gossa, « P4z : Complexité et performances applicatives / 101 », https ://git.unistra.fr/gossa/P4z, 2017.
8. Wikipedia, « Structure de données », https ://fr.wikipedia.org/wiki/Structure_de_données.