Une approche pragmatique de la programmation pour des biologistes qui programment presque

Catherine Letondal

Une approche pragmatique de la programmation pour des biologistes qui programment *presque*

Catherine Letondal

Institut Pasteur, Service d'informatique scientifique
28, rue du Docteur Roux
75724 Paris Cedex 15
letondal@pasteur.fr

Laboratoire de Recherche en Informatique
URA 410 du CNRS
LRI - Bâtiment 490 - Université de Paris-Sud
91405 Orsay Cedex, France

RESUME

Cet article s'intéresse à l'accès à la programmation pour des biologistes qui doivent pouvoir adapter des programmes existants ou en écrire de nouveaux pour leur recherche. Nous expliquons pourquoi les outils de programmation pour non-informaticiens existants ne correspondent pas à leur besoins et nous donnons un point de vue critique sur les conceptions de la programmation des informaticiens professionnels. Enfin, nous présentons notre approche basée sur la conception participative et l'exploration technique.

MOTS CLES

Programmation par l'utilisateur final, conception participative, langages de prototype, langages de script, tableurs.

INTRODUCTION

La biologie moléculaire fait maintenant largement appel à l'informatique, notamment pour l'analyse des séquences d'ADN et de protéines. Les chercheurs en biologie ne peuvent pas toujours se contenter d'utiliser les logiciels existants, ils doivent aussi pouvoir les adapter ou en développer de nouveaux. Certains d'entre eux écrivent déjà leurs propres programmes, et parfois les distribuent. Mais pour ceux qui ne peuvent pas passer trop de temps à la programmation et aux concepts informatiques, il faut un environnement qui leur permette de tester quelques algorithmes simples et traiter leurs données par eux-mêmes. Mon propos dans cet article n'est pas de décrire un nouvel environnement de programmation pour ce type d'utilisateurs, mais d'expliquer pourquoi, selon mes observations, cela constitue un problème difficile.

L'Institut Pasteur enseigne régulièrement un cours de programmation de 3 mois, à l'issue duquel les biologistes sont capables de concevoir des algorithmes et de définir des structures de données en Scheme. Cependant, l'application par la suite de ces connaissances dans le contexte de leur travail s'avère souvent difficile. J'observe également depuis deux ans l'utilisation d'interfaces Web que j'ai développées pour faciliter l'accès à des logiciels d'analyses biologiques. Ces interfaces proposent une fonction d'enchaînement de programmes équivalente aux pipelines Unix. L'utilisation régulière de cette fonction suggère que si les biologistes avaient un moyen aussi simple et à leur portée pour faire la même chose sous Unix, ils seraient probablement capables de programmer des scripts.

Que faut-il faire alors pour mettre la programmation *à leur portée* ? Mon hypothèse est qu'une partie du problème est du côté des idées que les informaticiens peuvent avoir *a priori* concernant ce qui est facile ou difficile dans la programmation : à titre d'exemple, ce n'est pas nécessairement l'algorithmique qui bloque les biologistes - qui sont pour la plupart des scientifiques de haut-niveau, mais plutôt les innombrables détails qui entourent l'algorithme et qui n'ont aucun intérêt pour eux : les entrées-sorties, les formats des fichiers ou les problèmes d'interfaces.

LES TECHNIQUES ACTUELLES

Parmi les techniques qui tentent de faciliter la pratique de la programmation pour les non-informaticiens, citons : la programmation visuelle où les concepts de programmation sont manipulés graphiquement, les environnements de composition visuelle et la programmation par démonstration.

Il est utile de distinguer parmi les langages visuels ceux qui sont bi-modaux (modes édition et exécution), comme LabView [1], de ceux qui permettent une programmation directement dans l'interface : Self [10], Forms/3 [2], et les tableurs. L'idée sous-jacente des outils de la première catégorie est que la difficulté de la programmation est d'ordre syntaxique : en permettant la manipulation directe des concepts de programmation sous une forme graphique, comme les boucles, les tests, les classes ou les composants, on pense faciliter l'accès à la programmation. Mais Nardi [8] a montré que cette distinction entre visuel et textuel ne discrimine pas réellement les langages professionnels des langages non-professionnels. Nardi donne ainsi des exemples de langages textuels utilisés sans problèmes par des non-informaticiens, comme un langage d'instructions pour une machine à tricoter ou comme un langage de codage du base-ball. En revanche, l'approche «programmation dans l'interface» semble prometteuse : en supprimant la distinction entre programmer et utiliser, et en rendant les deux activités accessibles

dans le même contexte, elle fait de la programmation une activité cohérente et continue avec le travail en cours.
L'idée de la programmation par démonstration ou par l'exemple [3] est d'utiliser des séquences d'actions de l'utilisateur pour générer des programmes. Plus l'interface est interactive, plus cette approche est utile, bien qu'elle comporte certaines limites : la programmation par l'exemple est conçue pour ré-utiliser les actions, mais encore faut-il que ces actions aient eu lieu, et que le vocabulaire interactif soit suffisamment expressif. De plus il est parfois plus simple de décrire une action que de la démontrer au logiciel. C'est pourquoi, même si une fonction d'historique et d'enregistrement de macros avec, pourquoi pas, un mécanisme d'inférence serait extrêmement utile, cela ne répondrait pas à tous les besoins.

Le problème qui se pose s'explique bien en termes de degrés de flexibilité logicielle [9] :

- la paramétrisation fonctionnelle comporte une limite inhérente, puisque tous les paramètres doivent avoir été prévus ;
- la composition logicielle, où la structure est un paramètre, apporte plus de souplesse : mais ce niveau d'accès est soit trop difficile, s'il s'agit d'utiliser un framework, soit bien trop limité, s'il s'agit de faire de l'accrochage de boîtes graphiques (Java Beans) ; il faut pouvoir, si c'est nécessaire, mener l'utilisateur au niveau du composant lui-même pour le modifier ou pour en programmer un nouveau ;
- la flexibilité maximum est atteinte avec les langages de programmation généraux : il nous faut donc un degré de programmabilité de ce niveau, c'est-à-dire un langage général qui soit accessible directement depuis l'interface utilisateur de l'application. Le fait qu'il soit général n'exclut d'ailleurs pas une orientation vers le domaine par la présence de composants spécialisés.

Et Programmer, au fait, c'est quoi ?

Compte tenu de cette analyse, notre problème semble donc résolu : il suffit de créer un langage/environnement de programmation ayant les propriétés énoncées ci-dessus. Mais le fait que les biologistes, dans leur ensemble, n'utilisent pas les nombreux langages et environnements à leur disposition amène à penser que cette analyse est trop courte. En réalité le problème des biologistes n'est pas tant les fonctionnalités du langage qu'on leur propose que l'intégration de l'activité de programmation dans leur contexte de travail. Or, si l'on sait bien ce qu'est un langage de programmation, il n'est pas si facile de définir la programmation elle-même. En guise d'illustration, le tableau 1 propose un certain nombre de définitions, dont chacune est aisément contredite.

UNE APPROCHE EMPIRIQUE

En s'intéressant à l'activité logicielle réelle des biologistes plutôt qu'à la programmation et comment la définir, il s'agit en définitive d'aborder le problème du point de vue des utilisateurs. La conception centrée sur l'utilisateur a montré son utilité, et je pense qu'une telle démarche est d'autant plus importante dans un domaine où l'informaticien professionnel pense qu'il *connaît d'avance* la nature du problème et de sa solution.

Conception Participative

Du point de vue du biologiste, la question la plus importante à poser est : « *Qu'est-ce que les biologistes programmeraient, et comment ?* » Pour répondre à cette question, j'ai choisi de m'inspirer des méthodes et des techniques de conception participative [12].
Un certain nombre d'actions ont été menées jusqu'à présent :

- plusieurs scénarios d'utilisation de logiciels existants ont été observés, et enregistrés (audio et video) ;
- de nombreux entretiens informels ont eu lieu (une vingtaine environ) : ces entretiens portent en général sur la problématique spécifique du chercheur, mais peuvent aussi porter sur un logiciel particulier ;
- une enquête à base d'un questionnaire ayant obtenu environ 600 réponses a essayé d'évaluer l'usage des logiciels scientifiques et l'impact de la formation sur cet usage ;
- trois ateliers de conception participative ont eu lieu depuis début 1998 :
  - un atelier de prototypage papier avec un petit groupe de personnes (5 biologistes et 2 informaticiennes) pour démarrer la conception d'une nouvelle interface pour un algorithme de découverte de motifs[11] (figure 1) ;

Figure 1 : Prototype papier pour Alice.

  - un atelier de brainstorming pour un projet d'environnement d'analyse de séquences ;
  - un atelier de prototypage papier sur deux des thèmes qui émergeaient du brainstorming (figure 2).

Figure 2 : Prototypage papier de l'environnement d'analyse de séquences.

Le premier bénéfice attendu de cette approche est l'intérêt évident qu'il y a à intéresser les futurs utilisateurs d'un logiciel à sa conception, et la source potentielle d'innovation technologique qu'ils représentent [6]. Mais c'est aussi de répondre à la question posée ci-dessus sans préjuger de la réponse : on ne cherche pas à définir un environnement ni un langage de programmation mais *un environnement d'analyse de séquences*. Si la programmation y trouve sa place, ce sera dans la continuité de la problématique et non comme un ensemble d'idées *a priori*.

programmer c'est...	***mais...***
La programmation c'est l'**abstraction**, la **généralisation** : écrire un programme c'est modéliser des objets d'une manière suffisamment générale.	Programmer amène parfois aussi à spécialiser, comme par exemple aller directement dans le code pour personnaliser une partie. Écrire un pilote de périphérique, c'est l'opposé d'une généralisation. Prendre une algorithme général (programmation dynamique), et régler ses équations pour l'appliquer à un domaine spécifique (la comparaison de séquences d'ADN), c'est aussi de la programmation.
L'essence de la programmation c'est la conception d'**algorithmes**.	Une partie infime des programmes écrits comporte un nouvel algorithme. C'est même un lieu commun de dire que la plus grande partie des programmes n'est pas de nature algorithmique, mais est plutôt composée de «colle», d'entrées-sorties, d'interface utilisateur. Ou bien doit-on suivre une certaine idée selon laquelle la simple présence d'une boucle dénote la conception d'un algorithme ?
La programmation c'est l'**automatisation** : sauvegarder des instructions dans un fichier pour leur exécution ultérieure, c'est programmer.	Cela signifie-t-il que taper des commandes complexes au niveau d'un interpréteur n'est pas de la programmation ?
Programmer, en sauvegardant des instructions pour leur exécution à une date donnée, c'est aussi **planifier**.	Cela veut-il dire que l'utilisation des commandes `at` ou `cron` sous Unix est de la programmation ?
La programmation c'est la traduction vers un niveau **meta** ou **symbolique** , comme par exemple lorsqu'on nomme un objet (c'est une référence à la dichotomie usage-mention [7]).	Est-ce que définir un alias c'est programmer ?
Programmer c'est **écrire du code**.	Est-ce que l'édition d'un fichier de configuration c'est de la programmation ?
Programmer, c'est utiliser un **compilateur** ou un interpréteur.	Est-ce que soumettre un fichier à l'interpréteur LaTeX ou même utiliser Netscape c'est programmer ?
La programmation c'est la **construction d'un logiciel** ou de composants logiciels.	Les programmeurs professionnels construisent-ils si fréquemment des logiciels ou des composants ?
Un programme c'est la représentation statique de quelque chose de dynamique [5].	La conception et l'implémentation d'une base de données n'est-elle pas de la programmation ? Et la réalisation des pages d'un serveur Web ?

Tableau 1: définitions de la programmation.

SUR QUELS PRINCIPES TECHNIQUES PEUT-ON S'APPUYER ?

Mon objectif n'est pas (ici) de proposer une boîte à outils ou une interface utilisateur, mais plutôt d'évaluer, parmi les techniques actuelles, celles qui correspondent le mieux au contexte du problème. J'ai commencé à implémenter certains des principes qui suivent dans un premier prototype dont le but essentiel est de fournir une base concrète à des discussions avec les biologistes, afin de susciter des idées de leur part et d'évaluer la pertinence de ces hypothèses techniques.

• Un **langage de script** semble un choix raisonnable dans un contexte d'outils traitant essentiellement des chaînes de caractères et du texte. Pour le prototype, c'est Tcl/Tk qui a été choisi. Il existe d'ailleurs déjà une base de composants développés dans ce langage par la communauté des bio-informaticiens.

• Le modèle de programmabilité offert par les **tableurs** semble adéquat : il combine la puissance de la manipulation directe avec celle de la manipulation programmatique. Le prototype développé reprend cette idée de définir automatiquement des variables associées aux objets graphiques : par exemple, des tags peuvent être définis interactivement, et utilisés comme variables.

• Le **code source** d'une partie *significative* de l'application - et c'est une des choses à déterminer lors des ateliers - doit pouvoir être accessible depuis l'interface, comme le code HTML d'une page Web, ou comme l'*outliner* d'un objet de l'environnement du langage de prototype Self [10]. Dans le même ordre d'idées les **protocoles méta-objets** [4] ou plus généralement les idées d'**open implementation**, ainsi que les développements autour de la **réflexivité** des langages peuvent s'avérer de bons modèles techniques pour la manipulation du code des différents niveaux de complexité de l'application. Le prototype étant développé dans un langage comportant une assez bonne réflexivité, il a été possible d'y implémenter les principes d'accès à travers l'interface : accès à l'arborescence des widgets ainsi qu'à toutes leurs caractériques, accès au code des commandes, clonage des objets graphiques, copier-coller direct de code source, ...

• L'idée de ne pas avoir nécessairement à définir systématiquement des types et des classes semble raisonnable, puisqu'il s'agit plus d'une approche d'utilisation avancée que de construction logicielle et de modélisation. Toutefois cela doit pouvoir être possible.

• Il est indispensable de disposer d'un répertoire de composants (commandes, fonctions, structures de donnés, widgets, ...) réalistes qui fonctionnent d'emblée sans programmation (un framework, mais déjà instancié) et qui peuvent être ré-utilisés non seulement par composition, mais aussi tout simplement comme exemples.

CONCLUSION

L'idée principale de ces travaux est d'abaisser le niveau de la marche à franchir pour le biologiste qui a identifié le besoin de programmer mais qui n'arrive pas à le faire : il faut ici accepter un autre point de vue sur la programmation, non plus comme un outil pour les informaticiens professionnels mais comme une forme avancée d'interaction, obéissant à des critères et à des objectifs différents.

BIBLIOGRAPHIE

1. Baroth, E and Hartsough, C. Visual Programming in the Real Wold. In *Visual Object-Oriented Programming, Concepts and Environments,* 1995, Prentice Hall, pp 21-42.
2. Burnett, M. M, Goldberg, A and Lewis, T.G. *Visual Object-Oriented Programming, Concepts and Environments,* 1995, Prentice Hall, pp 3-20
3. Cypher, A. *Watch What I Do. Programming by Demonstration*. 1993, MIT Press.
4. Kiczales, G, des Rivieres, J and Bobrow, D. G. *The Art of the Meta-Object Protocol,* 1991, MIT Press.
5. Lieberman, H and Fry, C. Bridging the Gulf Between Code and Behavior in Programming. In *ACM conference on Human Factors in Computing Systems (Summary, Demonstrations)* (CHI '95), 1995, ACM Press, p.480-486.
6. Mackay, W.E. Beyond iterative design: User innovation in co-adaptive systems. Rank Xerox EuroPARC, Cambridge, England, 1992.
7. Smith, Randall B and Ungar, David and Chang, Bay-Wei. The Use-Mention Perspective on Programming for the Interface. In *Languages for Developping User Interfaces*, Jones and Bartlett, pp 79-89.
8. Nardi, B.A. *A small matter of programming: perspectives on end user computing*. 1995, MIT Press.
9. Nierstrasz, Oscar and Dami, Laurent. Component-Oriented Software Technology. In *Object-Oriented Software Composition*, 1995, Prentice Hall, pp 3-28.
10. Smith, Randall B. and Ungar, David. Programming as an Experience: The inspiration for Self. In *Proceedings ECOOP'95*, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 303-330.
11. Sagot, M-F, Viari, A and Soldano H Multiple sequence comparison - A peptide matching approach. In *Theoretical Computer Science,* 1997, pp 115-137
12. Schuler, D and Namioka, A. *Participatory Design: Principles and Practices*. 1993, Hillsdale, NJ: LEA