Initiation à la Programmation " par l'exemple " : concepts, environnement, et étude d'utilité

Nicolas Guibert, Laurent Guittet, Patrick Girard

Initiation à la Programmation « par l'exemple » : concepts, environnement, et étude d'utilité

Nicolas Guibert*, Laurent Guittet*, Patrick Girard*

** LISI-ENSMA*
Téléport 2 - 1 avenue Clément Ader
BP 40109
86961 Futuroscope Chasseneuil cedex
guibert@ensma.fr, guittet@ensma.fr, girard@ensma.fr

RÉSUMÉ. *La programmation, depuis ses débuts, a toujours été vue comme une discipline difficile à apprendre et à enseigner. Même de nos jours, j. Kaasböll rapporte que les taux d'échec ou d'abandon aux cours d'initiation à la programmation en premier cycle universitaire varient de 25 à 80% de part le monde [KAASBOLL 02]. Dans le même temps, l'importance qu'a acquis l' informatique comme outil d'analyse ou de simulation dans les sciences expérimentales confère au problème de l'initiation à la programmation un grand enjeu. Dans cet article, nous proposons une synthèse de différents travaux de psychologie et de didactique de la programmation, et classons les problèmes liés à l'apprentissage de la programmation. Puis nous explorons les utilisations pédagogiques de la « Programmation sur Exemple » et présentons un environnement d'apprentissage, MELBA (Metaphor-based Environment to Learn the Basics of Algorithmics). Enfin nous analysons l'efficacité du concept et du système à partir d'une expérience menée sur 65 étudiants.*

MOTS-CLÉS : *psychologie et didactique de la programmation, interactions homme-machine, programmation sur exemples, environnement d'apprentissage.*

1. Introduction

Bien que l'écriture de programmes informatiques ait pris ces dernières années une place importante en sciences expérimentales, l'apprentissage de l'informatique demeure difficile : J.Kaasböll rapporte que les taux d'échec en premier cycle universitaire s'échelonnent entre 25 et 80 %. Qu'est ce qui rend l'apprentissage de la programmation si difficile ? Nous présentons ci-après une classification synthétique des difficultés rencontrées par les étudiants, en faisant le lien avec une description structurée de l'activité d'un programmeur.

2. Modélisation de la conception d'un programme, classification des difficultés

Charles Duchâteau propose une définition de la programmation : « faire faire une tâche à un exécutant ordinateur » [DUCHATEAU 00]. Cette définition peut être raffinée en un modèle de la conception d'un programme (figure 1).

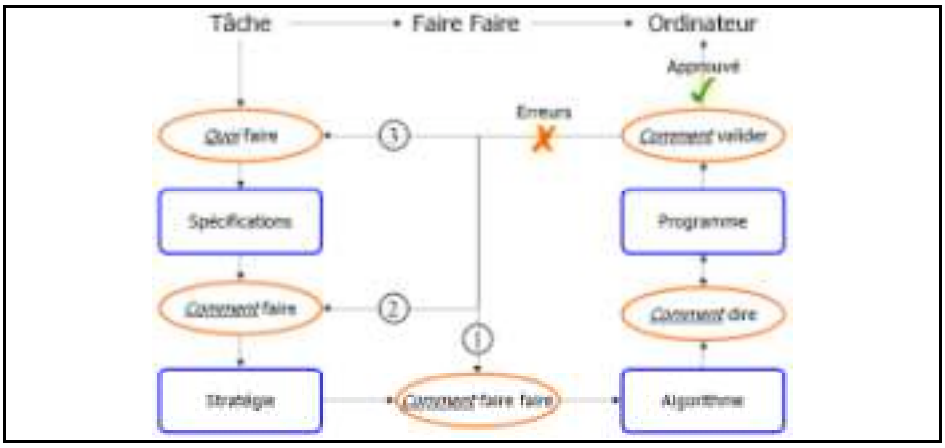


Figure 1. *Conception d'un programme, d'après [DUCHATEAU 00]*

La première étape, le "quoi faire", consiste à définir précisément la tâche à automatiser pour parvenir aux spécifications du programme. Cependant, cette étape est généralement court-circuitée : les tâches concernées sont très (ou trop) simples. Les premières difficultés surviennent donc à l'étape suivante, lorsque l'on s'efforce d'abstraire une stratégie permettant l'automatisation de celles-ci. Prenons l'exemple d'une tâche consistant à remplir n verres alignés avec une pipette. On peut envisager deux stratégies : remplir complètement chaque verre, ou déposer une goutte dans chaque verre jusqu'à ce qu'ils soient tous remplis.

La difficulté consiste à construire un modèle de la structure temporelle de la tâche, c'est-à-dire visualiser mentalement le déroulement du programme.

L'étape suivante, « comment faire faire », décrit cette stratégie sous une forme compréhensible de l'exécutant-ordinateur par des opérateurs de flots de contrôle

(conditionnelle, itération) et des structures de données informatiques, avec pour conséquence des erreurs « temporelles » [DU BOULAY 88].

Une difficulté propre à l'informatique, est que l'étudiant ne possède aucun modèle « naïf » de l' « intérieur » de l'ordinateur [BEN-ARI 98]. Une personne étudiant la physique cherche à comprendre des phénomènes qui lui sont familiers. En informatique, l'expérience pratique des ordinateurs est complètement inutile pour comprendre la programmation. Ceci conduit à deux grandes classes d'erreurs :

- des erreurs « anthropomorphiques » [SPOHRER 86] : les étudiants agissent comme si l'ordinateur possédait une « intelligence cachée » [PEA 86];
- des erreurs liées à la distance cognitive séparant les objets de la tâche de leur représentation informatique dite « frégeïenne » [ARSAC 91]. Considérons la différence des descriptions analogique et informatique d'un triangle (figure 2) par « int [][] ABC = {{2,2}, {4,13}, {10,6}} ! [SMITH 93]

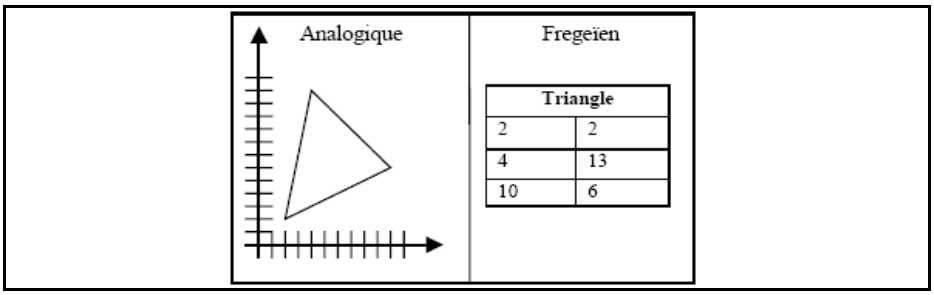


Figure 2. *Représentations « analogique » vs. « frégeïenne » d'un triangle ABC*

La dernière étape consiste à valider le programme par des tests dont l'interprétation soulève aussi les difficultés décrites ci-dessus.

L'étude de ces difficultés nous amène à définir trois objectifs pédagogiques indépendants :

- la modélisation de la tâche (la prise en compte de tous ses comportements) ;
- le modèle d'exécution du programme (l'ordre d'exécution des instructions) ;
- le modèle des données (le fossé cognitif qui sépare la représentation analogique des objets de la tâche, et leur représentation informatisée de type fregeïenne).

De cette classification, nous avons acquis la conviction qu'il faut enseigner un modèle de base du fonctionnement de l'ordinateur, en affichant une modélisation de son contenu, et en visualisant l'effet des instructions sur ce modèle de façon interactive.

3. Programmation « sur Exemple »

Tentant de réduire ces difficultés de conception, Smith introduisit avec Pygmalion [SMITH 93] la notion de « Programmation sur Exemple »

([CYPHER 93], [LIEBERMAN 01]). Le programme édité est **associé** à un exemple concret, qui fournit **en temps réel** un retour sur le comportement du programme. Deux techniques sont associées : la programmation « avec » exemple (le programmeur édite un programme dans un langage immédiatement interprété) et la programmation « par démonstration » (le programmeur manipule les objets de l'exemple, et l'environnement infère de cette démonstration un programme).

3.1. Illustration du concept

L'approche sur exemple remplace le cycle de conception par une démarche interactive et permet d'appréhender un modèle du fonctionnement de l'ordinateur. Nous avons construit l'environnement MELBA qui implémente les programmations « avec exemple » et « par démonstration » La figure 3 représente l'environnement MELBA.

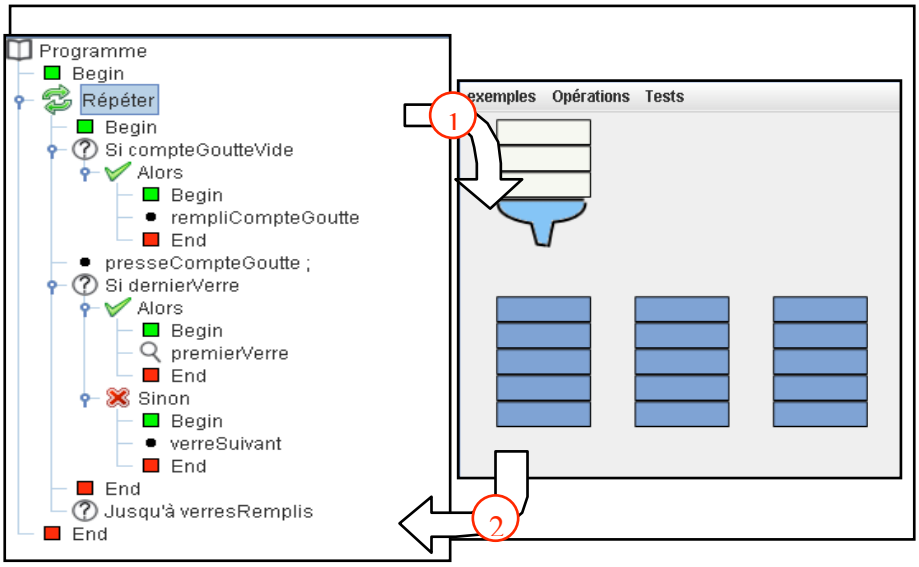


Figure 3. *Exemple de tâche à automatiser : le remplissage des verres par la pipette*

Le panneau de droite représente l'exemple, celui de gauche représente le programme. Lorsque l'étudiant clique sur une instruction, le programme est alors exécuté jusqu'à cette instruction. Cette exécution est animée sur les deux panneaux. Lorsqu'il ajoute une instruction, le système se re-synchronise pour fournir un écho de l'action exécutée (programmation « avec exemple », flèche 1de la figure).

La « programmation par démonstration » (flèche 2) se fait en manipulant les objets de la fenêtre de droite. Ces actions sont interprétées par le système qui ajoute au programme l'instruction. Le composant suivant (figure 4) représente l'état des données du programme. Il s'appuie sur la **métaphore du bureau** pour expliciter les concepts de variable, de type, de paramètre et de référence. L'interactivité avec l'environnement minimise les risques d'interprétations erronées (comme dans le cas

variable = boîte étudié par [DU BOULAY 89]) car les étudiants peuvent vérifier à chaque instant la correction de leur interprétation.

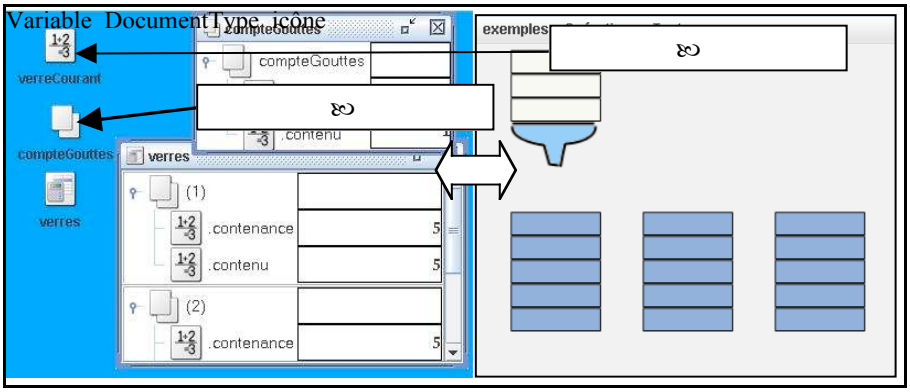
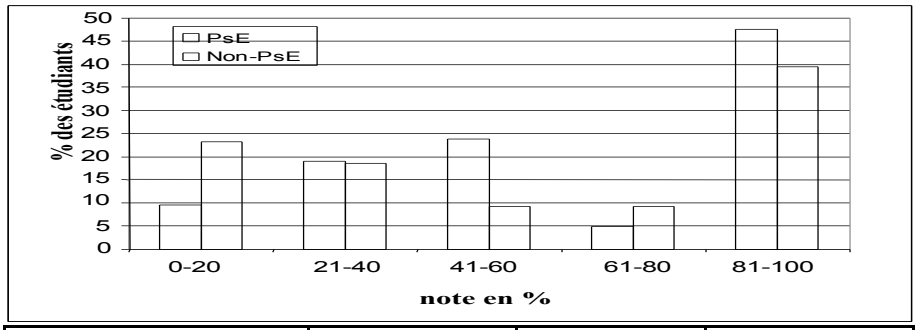


Figure 4. *Usage de la métaphore du bureau pour la modélisation des données*

3.2. Expérimentations avec MELBA

Nous avons mené une expérimentation sur 65 étudiants en bio-informatique séparés en groupes (chacun conservant un taux égal d'étudiants ayant déjà programmé). Ils ont été ensuite soumis à une évaluation portant sur la capacité à comprendre et à rédiger un programme. Les résultats avec l'outil sont encourageants, avec une augmentation de 18% des notes au dessus de la moyenne (table 2).



	Classe	Groupes PsE	Autres Groupes
% de résultats >=moyenne	63,07 %	76,19 %	58,13 %

Table 2. *Evaluation comparée sur les modèles de la tâche et du programme*

4. Conclusion

Dans cet article nous avons proposé une synthèse des différents types d'erreurs auxquels sont confrontés les étudiants en initiation à la programmation, à partir de

la littérature existante en didactique et psychologie de la programmation. Nous en avons tiré une classification des difficultés auxquels ils sont confrontés.

Nous avons étudié la pertinence d'un paradigme alternatif de conception de programme : « la programmation sur exemple » pour atténuer ces difficultés. Nous avons présenté un environnement d'apprentissage de la programmation, MELBA, basé sur ces concepts, et rapporté les résultats d'expérimentations menées avec cet environnement. Nous prévoyons à court terme de mener de nouvelles expérimentations avec un environnement plus évolué dans le but de confirmer les résultats obtenus et de tester l'utilité de ses extensions.

5. Bibliographie

[ARSAC 91] Arsac J., *Préceptes pour programmer*, Paris : Dunod, 1991.

[BEN- ARI 98] Ben-Ari M. « Constructivism in Computer Science Education », in *29th ACM SIGCSE Technical Symposium on Computer Science Education*, Atlanta Georgia: ACM press. 1998.

[CYPHER 93] Cypher A. *Watch What I Do: Programming by Demonstration,* The MIT Press: Cambridge, Massachusetts. 1993.

[DU BOULAY 89] Du Boulay B., « Some Difficulties of Learning to Program », in *Studying the Novice Programmer,* Lawrence Erlbaum Asssocites. p. 283-299. 1989.

[DUCHATEAU 00] Duchâteau C., *Images pour programmer*, Vol. 1, Namur: Facultés Universitaires Notre Dame de la Paix, 2000.

[KAASBOLL 02] Kaasboll J., Learning Programming, University of Oslo, 2002.

[LIEBERMAN 01] Lieberman H., *Your Wish is my command*, Morgan Kaufmann (Ed.), 2001.

[SMITH 93] Smith D.C., « Pygmalion, An Executable Electronic Blackboard », in *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor. 1993, The MIT Press: Cambridge, Massachusetts.

[SPOHRER 86] Spohrer J. « Analysing the high frequency bugs in novice programs », in *First Workshop on Empirical Studies of Programmers,* 1986.