Performances et usages d'un environnement d'apprentisage de la programmation « basé sur exemple ».

Nicolas Guibert, *Patrick Girard,* *Laurent Guittet*

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

RESUME
Malgré la place grandissante occupée par l'activité de programmation, en tant qu'outils d'analyse ou instruments de mesure, dans les sciences expérimentales, son apprentissage demeure toujours très difficile. De nombreuses études ont caractérisé les erreurs et difficultés rencontrées par les programmeurs novices. Depuis quelques années, nous explorons l'utilisation d'un paradigme particulier, la programmation sur exemple, dans le but de réduire ces difficultés.

Le travail présenté ici se veut une évaluation de cette démarche, et sera articulé autour de deux axes, celui de l'analyse des usages et celui de l'efficacité de l'apprentissage. Trois expérimentations de MELBA, l'outil développé dans le cadre du projet, utilisant diverses méthodes d'évaluation, sont ainsi présentées et analysées.

MOTS CLES : Psychologie et didactique de la programmation, Programmation sur exemples, Environnement Informatique pour l'Apprentissage Humain, Expérimentations, Occulométrie.

ABSTRACT
Although computers and programs have now become essential in experimental sciences such as analysis or measurement tools, many students still find learning Computer Science is extremely difficult. Many studies have characterized the errors and difficulties encountered by novice programmers. For some years, we explore the use of a particular paradigm, programming by example, to lower these difficulties.

The work being presented here intends to be an evaluation of this approach, in the perserpective of analysing its efficiency in learning, and how learners appropriate it. Three experiments of MELBA, the tool developed in the context of the project, using different methods and metrics, are thus described and discussed.

KEYWORDS : Psychology and didactics of programming, Programming by demonstration, Computer-aided learning and teaching, Experimentations.

INTRODUCTION

Alors que micro-ordinateurs et programmes informatiques se sont implantés dans de nombreuses disciplines scientifiques en tant qu'outils d'analyse ou instruments de mesure (physique, chimie, sciences de la vie… on parle même dans ce dernier cas de bio-informatique), l'acquisition des compétences requises pour la conception de programmes ne se fait pas aisément. Käasboll [10] rapporte que, de par le monde, entre 25 et 80 % des étudiants sont en situation d'échec. Pourquoi la programmation est-elle si difficile d'accès ? Nous nous attachons, en préambule, à ébaucher une réponse à cette question. Pour cela, à partir de la littérature en psychologie ou en didactique de la programmation, nous pratiquons une synthèse des différents types de difficultés auxquelles sont confrontés les programmeurs débutants.

Par la suite, nous définissons l'approche suivie pour réduire ces difficultés, et décrivons l'outil réalisé dans la perspective de la mettre en œuvre. Puis, nous présentons une évaluation de l'apprentissage réalisée à partir de deux expérimentations en situation réelle, et discutons du choix et de la pertinence des métriques et du protocole, en comparaison avec d'autres études en psychologie de la programmation. Enfin, nous complétons cette évaluation et analysons les usages de l'environnement à partir des résultats d'une expérience avec un oculomètre.

DIFFICULTES DANS L'APPRENTISSAGE DE LA PROGRAMMATION.

Duchâteau propose la définition suivante de la programmation : « faire faire une tâche à un ordinateur » [5]. Cette définition peut être raffinée en un modèle de la conception d'un programme (figure 1), qui nous permet de regrouper les difficultés et les erreurs des programmeurs novices, rencontrées dans la littérature [4; 12; 13].

La première étape, le "quoi faire", consiste à définir précisément la tâche à automatiser pour parvenir aux spéci-

fications du programme. Cependant, cette étape est généralement court-circuitée : les tâches concernées sont très (ou trop) simples (pour un humain). Les premières difficultés surviennent donc à l'étape suivante, lorsque l'on s'efforce d'abstraire une stratégie en permettant l'automatisation.

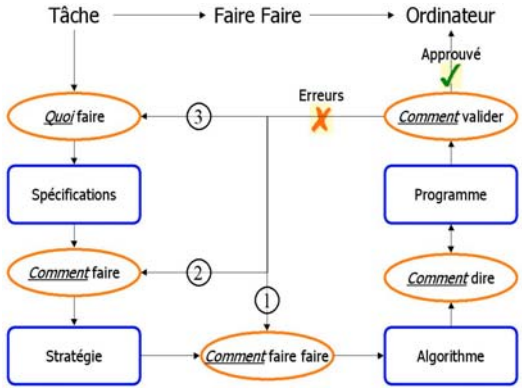

Figure 1: Modèle de conception d'un programme, d'après [5].

Cette activité met en exergue la première difficulté majeure de la programmation, à savoir la difficulté d'abstraction : le programmeur doit factoriser dans le programme l'ensemble des comportements de la tâche. Il en résulte un « syndrome de la page blanche », mis en évidence notamment par Käasboll. Selon les étudiants :

> « … lorsque le problème est présenté … on le décompose comme ça, comme ça, comme ça. Tout a l'air simple et très logique, et puis c'est à toi et Ouch! Par quoi je commence ? Peut-être que c'est facile, mais le problème c'est que tu ne sais pas par quel bout commencer quand il faut résoudre le problème … »

L'étape suivante, « comment faire faire », décrit cette stratégie sous une forme compréhensible par l'exécutant-ordinateur. Le programmeur débutant est alors confronté à une difficulté importante venant de la distance cognitive séparant sa sémantique de la tâche de la sémantique utilisée par l'ordinateur.

Considérons une description informatique d'un triangle par : « int [][] ABC = {{2,2}, {4,13}, {10,6}} ». Cette présentation « formelle » [5], issue de la sémantique de l'ordinateur, diffère grandement de la représentation que l'humain peut se faire des données de la tâche[1] (en l'occurrence le triangle). Cette problématique est compliquée par le fait que, comme le fait remarquer Ben-Ari [1] le débutant ne possède aucun modèle « naïf » de l' « intérieur » de l'ordinateur. Une personne, étudiant la physique, cherche à comprendre des phénomènes qui (pour la physique élémentaire !) lui sont déjà familiers. En revanche, en informatique, l'expérience pratique des ordinateurs (suite bureautique, Internet, jeux vidéos …) est pratiquement inutile pour comprendre la programmation.

Après avoir traduit l'algorithme ainsi obtenu dans un langage particulier, et après l'inévitable cortège d'erreurs syntaxiques peu utiles pour modéliser le fonctionnement de l'ordinateur, la dernière étape consiste à valider le programme, par une série de tests, dont l'interprétation des résultats soulève les mêmes difficultés. Il faut attendre cette phase (qui se caractérise par un retour d'information « inversé » dans le temps – 1, 2, 3, figure 1) pour pouvoir juger de la validité de toutes les précédentes.

Cette absence d'écho immédiat est la troisième difficulté majeure de la programmation. Blackwell [2] qui, lui, parle de « perte de manipulation directe », note que cette caractéristique est celle qui conditionne l'appellation de « programmation » dans le sens commun. Les gens disent qu'ils « programment » leur magnétoscope, leur site Web en HTML, etc. Cette absence représente bien sûr un important facteur aggravant pour les deux difficultés précédentes…

Ces différentes difficultés nous ont conduits à définir trois objectifs pédagogiques spécifiques :

1. Apprendre à modéliser la tâche (abstraire de façon exhaustive son déroulement)
2. Apprendre le modèle d'exécution du programme (être capable de faire le lien entre la position dans le programme et l'état du système).
3. Apprendre le modèle des données (pouvoir manipuler les structures de données informatiques et pouvoir modéliser les objets de la tâche par des structures adaptées).

Pour ce faire, il nous paraît essentiel de disposer d'un environnement d'apprentissage permettant de séparer le plus possible les activités et les difficultés liées à chaque objectif, afin de supporter un modèle incrémental de la programmation[2].

[1] Notons bien que ce « fossé cognitif » touche à des représentations « conceptuelles », et qu'il est donc quasiment indépendant du choix du langage de programmation que fera l'enseignant. Par exemple, il sera identique avec Ada, Pascal et C, car tous trois sont des langages procéduraux qui manipulent les même concepts.

[2] En effet, contrairement à la physique (où on étudie d'abord la cinématique du point, puis la mécanique des solides, puis la mécanique des fluides), en programmation, l'apprentissage des différents concepts se fait quasisimultanément. Ceci est en partie du au fait que l'enseignement s'appuie sur des langages et des environnements « professionnels » (dans le sens où ils s'adressent tous à des programmeurs confirmés).

UNE APPROCHE « BASEE SUR L'EXEMPLE » DE L'APPRENTISSAGE DE LA PROGRAMMATION.

Tentant de réduire ces difficultés de conception, Smith introduit avec Pygmalion le concept de « Programmation sur Exemple » ([3] et [9]). Le programme édité est associé à un exemple concret, qui fournit en temps réel un retour sur le comportement du programme.

Illustrations de l'approche.

Nous nous proposons à présent d'illustrer cette approche à partir de deux exemples tirés de l'état de l'art, à savoir Pygmalion [3], le premier environnement de programmation basé sur l'exemple, et StageCast Creator [9], un environnement permettant de concevoir « sur exemple » des simulations ou des jeux vidéo en 2D.

Pygmalion, tout d'abord est un environnement de programmation visuelle « sur exemple » basé sur la métaphore du tableau blanc (ou noir), et sur le langage SmallTalk. Il ne s'agit donc pas d'un *langage*[3] visuel à proprement parler, mais d'un espace de travail graphique. Comme on peut le constater figure 2, les différentes variables et paramètres apparaissent sous la forme de cases, lesquelles contiennent la valeur courante de la variable. L'état du programme peut ainsi être affiché de façon exhaustive et continue, ce qui rétablit le principe de manipulation directe.

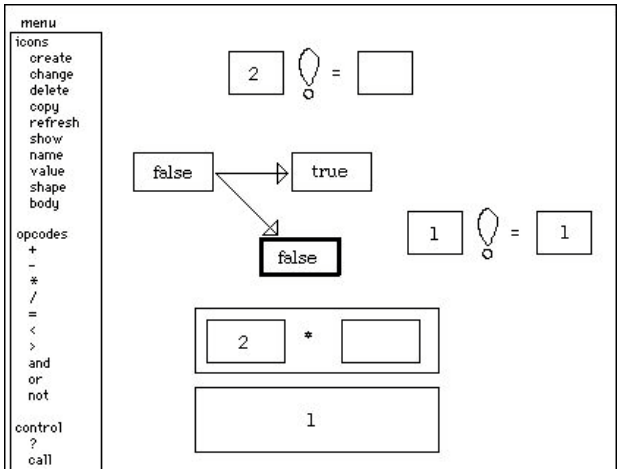

Figure 2: l'environnement de programmation Pygmalion ; la figure représente la conception d'un programme calculant n ! .

Stagecast Creator, pour sa part, reprend l'approche sur exemple en y ajoutant un aspect « pragmatique », dans le sens où toutes les actions sont exprimées dans le référentiel du domaine de la tâche, et non dans un référentiel informatique (par exemple, avec des trains et des rails, et non pas avec des tableaux, des entiers, ou des «booléens» -figure 3).

[3] Pour plus de détail sur la distinction entre langage visuel (comme G pour Labview) et environnement de visualisation de programmes, le lecteur pourra consulter les taxonomies de Myers[10].

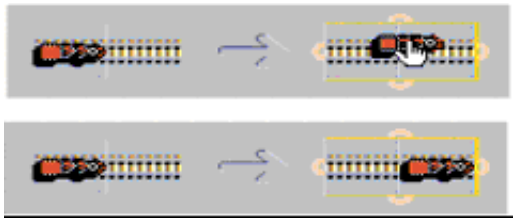

Figure 3: l'environnement Stagecast Creator, pendant la construction d'une simulation de traffic ferroviaire

Pertinence et limitations de l'approche « sur exemple ».

Dans le cadre de l'apprentissage de la programmation, l'écho immédiat pourrait permettre de construire un modèle mental viable du comportement du programme, et une représentation graphique « pragmatique » de l'état du système devrait faciliter l'évaluation de celui-ci en comblant le gouffre d'évaluation lié à la différence de référentiels. L'apprentissage du modèle informatique de représentation de la tâche serait alors l'étape suivante, et pourrait ainsi s'appuyer sur le socle de connaissances déjà existant.

On peut cependant reprocher aux systèmes « sur exemple » existants deux défauts majeurs. D'une part, ayant pour cible un public d'« utilisateurs finaux », ils s'attachent majoritairement à cacher le programme informatique qu'ils construisent, et donc ne permettent pas de construire la relation programme-état au cœur de nos objectifs pédagogiques. D'autre part, l'évaluation progressive de l'état du système est généralement obtenue en contraignant les possibilités d'interactions avec le programme.

En effet, l'écho proposé par le système impose obligatoirement que le programme soit toujours dans un état cohérent. La plupart de ces environnements vont plus loin encore en obligeant le programmeur à entrer les commandes dans l'ordre chronologique. Les facilités apportées par l'évaluation progressive de l'état du système et l'expressivité des commandes sont alors gommées par ces contraintes, que Green [7] désigne sous le terme d' « engagement prématuré ». Il leur manque le plus souvent la capacité de « revenir en arrière », ce qui génère un grand « degré d'engagement », dans le sens où chaque erreur oblige l'utilisateur à reprendre tout depuis le début. Il est donc conduit à planifier ses actions à l'avance (« prévisualisation forcée ») pour éviter la moindre erreur.

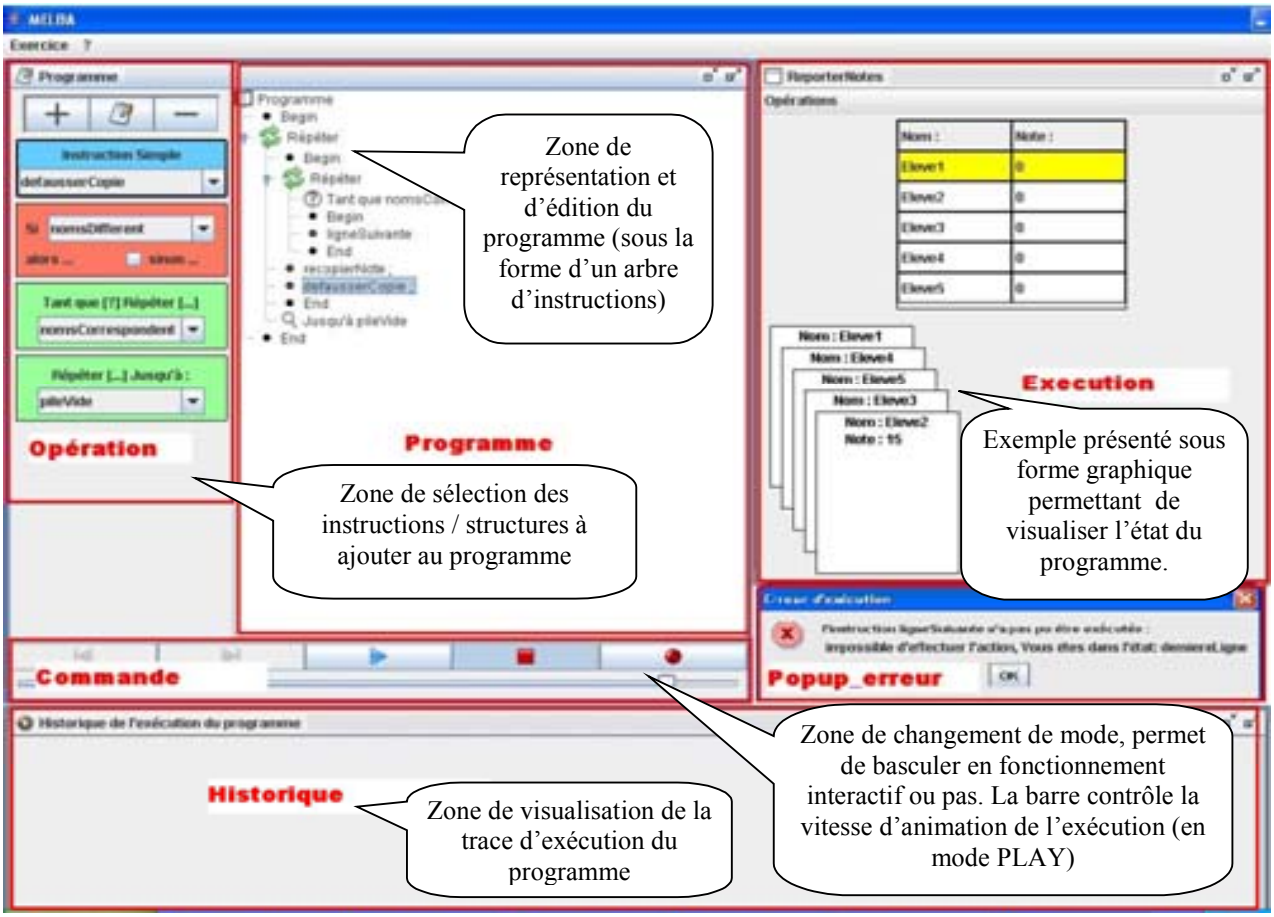

Figure 4 : Vue d'ensemble des composants de l'environnement MELBA.

Pour mesurer les apports que pourrait apporter à l'apprentissage l'usage d'un exemple concret pendant la conception du programme, nous avons conçu l'environnement MELBA (Metaphor-based Environment to Learn the Basics of Algorithmic) décrit par la figure 4.

MELBA : UN ENVIRONNEMENT D'APPRENTISSAGE DE LA PROGRAMMATION « BASE SUR L'EXEMPLE »

L'étudiant interagit avec le système en construisant par insertions et suppressions le programme, qui est présenté sous la forme d'un arbre. A chaque exercice est associé un ensemble spécifique de commandes et de tests, accessibles par menus déroulants dans la zone opération, qui agissent sur un modèle formel du système représenté sous forme graphique dans la fenêtre d'exécution. Alternativement, il est possible de faire des exercices « classiques » manipulant des données informatiques par l'opération d'affectation. Une fenêtre d'historique supporte une représentation de la trace d'exécution du programme.

Le système peut être piloté suivant 3 modes. En mode « Record », les fenêtres d'exécution et d'historique sont synchronisées avec l'instruction en surbrillance dans le programme. A chaque fois que l'utilisateur clique sur une instruction et à chaque fois qu'une instruction est ajoutée, l'environnement affiche l'état après exécution de cette instruction, et la trace associée à l'exécution du programme jusqu'à ce point. « Lecture » fonctionne de même, mais propose une animation de l'exécution, et non pas seulement le résultat. Enfin, « stop » désactive l'exécution immédiate, et le programme peut alors être édité de façon « classique ».

OBJETS DES ETUDES EXPERIMENTALES

Nous présentons dans les sections suivantes un ensemble d'expérimentations sur l'environnement MELBA pour éclairer la pertinence de l'approche « à base d'exemples » pour l'apprentissage de la programmation. Pour ce faire nous avons testé les hypothèses suivantes :

1. L'évaluation progressive permise par une exécution interactive peut-elle jouer un rôle positif dans l'apprentissage de la programmation ?
2. La programmation « sur exemple », vu qu'elle ajoute des contraintes à l'édition du système, est elle utilisable par un apprenant débutant, ou bien le fait de devoir intégrer simultanément le fonctionnement d'un programme et le fonctionnement de l'environnement crée-t-il chez lui une surcharge cognitive ?
3. Quelles compétences sont alors concernées, du diagnostic et de la compréhension de programmes à leur composition ?

4. Est-il pertinent d'utiliser la visualisation de programme en apprentissage de l'algorithmique et de la programmation ?
5. Faut-il privilégier en ce cas une approche superficielle (qui permet d'avoir rapidement une vue d'ensemble de l'état du système) ou bien en profondeur (qui fournit une information précise et détaillée, mais nécessite d'être explorée par l'apprenant) ?
6. Ces deux approches peuvent-elles être combinées ? (par exemple en proposant deux représentations simultanées et en partie redondantes entre lesquelles l'utilisateur pourrait créer des liens), ou cela crée-t-il une surcharge cognitive ?
7. L'animation a-t-elle une importance critique dans la visualisation de programme pour l'apprentissage de la programmation ? Ou bien est-il plus pertinent de représenter uniquement l'état du programme à certaines positions importantes (sur le modèle des « points d'arrêts » des debuggers) ?

CHOIX DES METRIQUES ET DES MODALITES D'EVALUATION.

Concernant un environnement destiné à l'apprentissage, la mesure de l'utilité couvre deux aspects distincts et cependant connectés [11]. D'une part, on souhaite évaluer l'apprentissage (de la discipline enseignée et non pas de la manipulation du système), et d'autre part on souhaite tester la capacité à réaliser des tâches (exercices de programmation).

Pour ceci, nous allons utiliser des méthodologies de tests issues de différentes disciplines dont l'objet d'étude est l'apprentissage. On peut pratiquer une première disctinction entre méthodes « quantitatives », qui visent à quantifier l'impact d'un environnement sur la réalisation de tâches et/ou l'apprentissage, ou méthodes « qualitatives » qui permettent de comprendre le cheminement interne de l'apprenant ou de donner des indications sur son degré de conscience face à son apprentissage. D'autre part, on peut distinguer deux types de mesures : « en-ligne », où la mesure se fait pendant la tâche (nombres d'erreurs, temps pour accomplir la tâche, fixations oculaires …) ou « hors-ligne ». Cette seconde approche ne permet pas d'évaluer le comportement de l'étudiant pendant son apprentissage, mais peut donner des informations importantes sur sa compréhension.

PREMIERE EXPERIMENTATION.

Dans un premier temps, nous avons cherché à obtenir des indicateurs sur la première hypothèse. Pour ce faire, nous avons choisi de pratiquer une évaluation en conditions réelles dans un cours d'initiation à la programmation en L3 pour des bio-informaticiens, comportant 4h de cours et 12 h de Travaux Dirigés (TD). Le choix de la méthode s'est portée sur une évaluation quantitative hors-ligne, sous la forme d'un test (classique dans ce contexte), avec documents, deux semaines après les séances de TD, car notre objectif était de quantifier l'impact à long terme sur la compréhension. Sur les 65 étudiants suivant le cours, 24 ont utilisé l'environnement MELBA. Le reste de la promotion a suivi un cursus de TD classique (papier-crayon).

L'environnement Melba, dans cette expérimentation, était dégradé de telle sorte que seul le mode « sur exemple » (RECORD) était disponible, car cette fonctionnalité constitue le socle de l'approche – et devait donc être évaluée en priorité (pas de programmation « classique » ni d'animation). L'historique n'était pas non plus disponible. Les études sur les facteurs influant sur les résultats aux premiers modules de programmation – telles que [6] - ayant mis en évidence une corrélation importante avec une expérience préalable en programmation, nous avons fait en sorte de « distribuer » uniformément les élèves ayant déjà ces compétences. Les étudiants du groupe machine étaient répartis à un par machine. Les résultats du partiel montrent que le taux d'élève ayant la moyenne aux exercices portant sur les concepts travaillés avec l'outil est supérieure de 18 points dans le groupe avec MELBA (table 1). Cela représente une significativité de 74% (0,26) selon le test khi2. Si, dans l'absolu, un tel résultat est assez faible, au vu de la taille des échantillons, il s'avère suffisant pour ouvrir la voie à des expérimentations plus approfondies.

	Groupes PsE (22)	**Groupes témoins (43)**
% **de résultats >= moyenne**	76 % (16)	58 % (25)

Table 1 : Résultats comparés des groupes avec et sans l'outil.

La répartition des notes (table 2) montre que les différences se concentrent essentiellement sur les notes les plus faibles et sur les notes moyennes, même si le pourcentage de scores supérieurs à 75% est également supérieur avec l'outil. Ces résultats tendent à confirmer l'hypothèse 1.

Répartion des notes (par quart)				
	Q1	**Q2**	**Q3**	**Q4**
Melba	2 (9%)	4 (18%)	6 (27%)	10 (45%)
Témoin	10 (23%)	8 (19%)	8 (19%)	17 (40%)

Table 2 : Répartitions des notes des groupes avec et sans l'outil.

DEUXIEME EXPERIMENTATION

Nous avons ensuite complété l'outil en ajoutant les autres modes, et en permettant deux types de visualisation. Pour le programme, le composant d'historique complète le panneau du programme, en permettant d'explorer très finement la trace de l'exécution. Pour l'exemple, une vue « système » montrant les structures de données fut rajoutée, pouvant remplacer ou compléter la visualisation graphique de la tâche. Ces nouveaux composants nous permettent dès lors de tester directement les hypo-

thèses 4, 5, 6 et 7, par les mêmes modalités que précédemment.

Nous avons donc mené une deuxième expérience sur un cursus d' « Initiation à la programmation pour les biologistes » en L1/L2. Celui-ci se décomposait en deux parties : une partie introductive, d'initiation à l'algorithmique (4h de cours et 6h de TD) validée par un partiel, sans document, et une partie consacrée au langage Perl. L'expérimentation avait pour cadre la première partie.

Sur les 41 étudiants qui suivaient ce cours, 17 ont formé un groupe de TD travaillant sur MELBA. Pour 15 d'entre eux, ce cours était une première initiation. Pour des raisons logistiques, les étudiants étaient 2 par machine, et non pas seuls face à l'environnement. 24 autres ont suivi un cursus de TD classique, parmi lesquels 18 novices complets. Parmi les exercices du partiel, trois portaient sur un objectif spécifique traité avec l'outil en TD (voir table 3). Ce découpage plus fin doit permettre de tester la troisième question. Ces tâches ont donné les résultats suivants:

Notes supérieures à la moyenne	**Melba (15)**	**Témoin (18)**
Description d'exécution:	6 (40%)	4 (22%)
Rédaction de programme:	6 (40%)	6 (33%)
Compréhension de la tâche:	14 (93%)	14 (78%)
Total :	7 (47%)	5 (28%)

Table 3 : pourcentages de notes supérieures ou égales à la moyenne dans les deux groupes, selon la tâche demandée.

On retrouve le même type d'écart observé dans l'expérimentation précédente, pour tout ce qui concerne la compréhension (au sens large) de programmes (i.e. +18, +16 et +19 points). Ces écarts sont significatif à respectivement 89, 68 et 86% selon le test du khi2. Le premier et le dernier écart sont donc très significatifs de par la faible taille de l'échantillon. Pour ce qui concerne l'écriture de programmes, l'écart est plus faible (« significatif » à 56% !).

Cela est étonnant, car le test de la première expérimentation portait essentiellement sur cette activité. Plusieurs hypothèses peuvent être faites à ce sujet, qui prennent en compte deux changements dans le protocole. D'une part, il est possible que placer les étudiants à deux par machine ait changé la donne. Les étudiants ayant eu un rôle moins actif n'auraient pas eu le même bénéfice dans une situation de composition, mais que leur rôle d'observateur leur ait permis de développer la compréhension de programmes. La deuxième hypothèse serait que dans le premier test, les étudiants ayant eu droit aux documents, leur meilleure compréhension des corrigés leur aurait permis de s'en inspirer efficacement.

Un des enseignements que l'on peut en tirer est que, alors que nous avons observé pendant les TD (évaluation en-ligne qualitative) que de nombreux étudiants faisaient un usage intensif de la fonctionnalité d'animation, les résultats quantitatifs ne sont pas meilleurs pour autant. En parallèle, nous avons noté une désaffection du composant d'historique, et une utilisation exclusive de la représentation la plus synthétique de l'état du système, quand deux vues parallèles étaient proposées. Il semblerait donc que si la visualisation se soit avérée pertinente dans notre cas (4) pour exprimer les relations état-programme, elle doive y être focalisée sur une vue superficielle permettant de donner du premier coup d'œil l'état général du système (5). L'hypothèse 6 sur l'usage en combinaison des deux approches de la visualisation s'est avérée non fondée. Le tableau 4 ci-dessous illustre la répartition des résultats dans les différentes tâches. Celle-ci s'avère plus « uniforme » que lors du premier test, même si le pic au troisième quartant y réapparait lors de la tâche de compréhension, la différence dans Q4 se retrouvant, elle, dans la tâche de rédaction.

Répartition des Notes (Description de la trace) :				
Q1	Q2	Q3	Q4	>=50%
9 (60%)	0%	4 (27%)	2 (13%)	40%
12 (67%)	2 (11%)	2 (11%)	2 (11%)	22%

Répartition des Notes (Rédaction de programme) :				
Q1	Q2	Q3	Q4	>=50%
5 (33%)	4 (27%)	2 (13%)	4 (27%)	40%
5 (28%)	7 (39%)	2 (11%)	4 (22%)	33%

Table 4 : Répartition des notes dans les deux groupes.

USAGES DE L'ENVIRONNEMENT

Par ailleurs, dans le but de répondre aux questions tenant plus à l'usage et à l'accomplissement de la tâche (2, par exemple), une évaluation en-ligne s'imposait, car seule cette approche est à même de quantifier les usages et les difficultés d'utilisations éprouvées par les apprenants. Pour cela, une expérimentation a été conduite sur un oculomètre Tobii 1750 par Multicom, à Grenoble, auprès de 6 sujets, 3 étudiants de première année et 3 lycéens de première S. Deux tâches spécifiques (compréhension et correction, rédaction complète) y ont été étudiées. Cette expérience a délivré des traces de deux types, d'une part des indicateurs oculaires, d'autre part le log des clics souris, notamment sur les changements de mode.

Indicateurs Oculaires

Les indicateurs de l'oculomètre (table 5) nous permettent ainsi de répondre rapidement à la dernière question. On constate que la zone d'historique est très peu regardée (respectivement 2,5% et 1,1% des fixations oculaires), les zones les plus pertinentes pour les apprenants étant (sans surprise) le programme, suivi des zones opération et exécution. La deuxième information importante est la

forte augmentation du pourcentage de fixations dans les zones programme et opération au détriment de la zone exécution, lors de la tâche 2.

Zône d'intérêt	Tâche 1	Tâche 2
Programme	45,5%	51,9%
Exécution	24,2%	9,8%
Opérations	18,8%	28,5%
Historique	2,5%	1,1%
Modes	3,3%	1,9%
Popups_erreur	5,7%	6,8%

Table 5 : Distribution des fixations en pourcentage, pour les tâches de correction (1) et de rédaction de programme (2).

Les différences de nature entre les deux tâches peuvent être considérées comme la première raison de ces changements. En effet, la première tâche consiste à corriger un programme existant, donc l'exploration du programme pour le comprendre est essentielle et explique la place prépondérante des zones de programme et d'édition. A contrario, dans la deuxième tâche, le programme manipulé est celui de l'utilisateur, de plus construit progressivement, et ne nécessite donc pas une exploration approfondie pour en inférer le sens. Cette hypothèse est cohérente avec les transitions entre zones (Figures 5a et 5b). Les transitions Programme-Exécution ne diminuent que de 4 points environ, ce qui atteste que c'est l'exploration interne de la zone d'exemple qui diminue dans la seconde tâche, bien plus que les consultations de celle-ci.

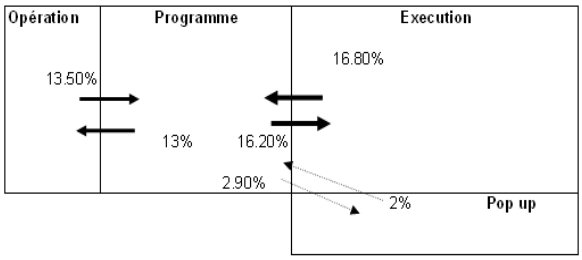

Figure 5a : Transitions entre les différentes zones de l'interface, dans l'exercice de détection/correction d'erreur.

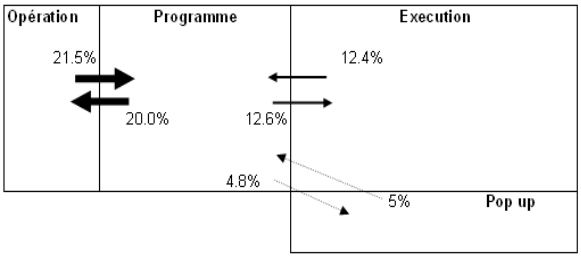

Figure 5b : Transitions entre les différentes zones de l'interface, dans l'exercice de rédaction de programme.

Par ailleurs, ces indicateurs laissent à penser que la recherche d'un écho du système à chaque modification n'est pas systématique, ce qui contredit un peu le paradigme « sur exemple ». Pour pouvoir confirmer ou infirmer cette hypothèse, analysons l'utilisation des différents modes du système.

Traces des clics souris

Intéressons-nous en premier lieu à la répartition de l'usage des modes sur l'ensemble des deux tâches, en mesurant le prorata de temps passé dans chaque mode (table 6). Le temps passé en mode interactif (qui fournit un écho actif : RECORD et PLAY) est plus important, ce qui confirme le besoin d'un retour rapide et facile à décrypter. Néanmoins les valeurs importantes de PLAY et STOP semblent également confirmer la difficulté d'intégrer le paradigme d'édition « sur exemple ». Par ailleurs RECORD progresse au détriment de PLAY. Cela n'est qu'à première vue incohérent avec les indicateurs oculaires. En effet, si le nombre de commandes exécutées est de 10, par exemple, le programmeur consulte l'exemple 10 fois en mode PLAY, contre 1 en mode RECORD (l'état final)…

Mode :	Debug :	Composition :	Total :
RECORD	26,2%	37,6%	31,9%
PLAY	29,2%	19,3%	24,2%
STOP	44,3%	43,1%	43,7%

Table 6 : Répartition de l'usage des différents modes du logiciel pour chaque tâche.

Ces informations demeurent cependant encore trop générales pour pouvoir tirer des conclusions sur les usages de l'environnement. Les tables 7a et 7b répertorient l'utilisation de chaque mode par chaque sujet. On constate une grande variabilité interindividuelle. On peut cependant extraire trois catégories, une proche de la programmation « sur exemple » (80%+ en mode interactif, édition en RECORD), une correspondant à ce que permettent les environnements de programmation classique (≈35% PLAY, 65% STOP) et un profil hybride (≈50% interactif).

Exercice 1 – Correction d'erreurs				
Sujet	RECORD	PLAY	STOP	Temps
S1	73,9%	13,3%	12,2%	17 mn 07 s
S2	10,4%	35,5%	53,2%	17 mn 24 s
S3	70,4%	11,8%	17,9%	4 mn 52 s
S4	0%	34,6%	65,4%	10 mn 24 s
S5	2,4%	38,4%	58,4%	2 mn 58 s
S6	0%	41,8%	58,2%	4 mn 25 s

Table 7a : Utilisation de chaque mode par chaque sujet, et temps de réalisation (1° tâche)

On peut constater que les trois profils sont équitablement répartis et ne sont pas en corrélation avec le temps d'accomplissement de la tâche. De plus, un individu peut changer de profil d'une tâche à l'autre (tels que S1, S2, et S4). Il se peut que le choix du style d'interaction selon la tâche dépende du style d'apprentissage du sujet ou

d'une autre caractéristique cognitive. Le confirmer ou pas demandera des investigations ultérieures.

Exercice 2 – Composition de programme				
Sujet	RECORD	PLAY	STOP	Temps
S1	23,2%	12,9%	63,8%	8 mn 10 s
S2	78,8%	4,5%	16,7%	16 mn 45 s
S3	91,6%	0%	8,4%	4 mn 45 s
S4	28,9%	21,4%	49,8%	8 mn 35 s
S5	0%	28,7%	71,3%	8 mn 42 s
S6	3,1%	48,1%	48,8%	17 mn 07 s

Table 7b : Utilisation de chaque mode par chaque sujet, et temps de réalisation (2° tâches)

CONCLUSION ET PERSPECTIVES

Dans cet article, nous avons étudié la pertinence d'un paradigme alternatif de conception : « la programmation sur exemple » pour supporter l'apprentissage de la programmation. Pour cela, nous avons proposé une synthèse des différents types de difficultés auxquels sont confrontés les étudiants en initiation à la programmation, qui semble attester de l'adéquation de cette approche aux objectifs pédagogiques.

Nous avons présenté un environnement d'apprentissage de la programmation, MELBA, basé sur ces concepts, et rapporté les résultats de trois expérimentations menées sur cet outil pour évaluer l'approche. En travaillant sur la base de sept hypothèses, nous sommes arrivés à un certain nombre de conclusions :

- L'évaluation progressive permise par une exécution interactive joue un rôle positif dans l'apprentissage d'un modèle d'exécution du programme (H1), en particulier en compréhension de programme, et en recherche / correction d'erreurs (H3).
- Pour cela, des techniques de visualisation de programme proposant un modèle graphique de l'état du système sont nécessaires (H4) ; pour être efficaces, elles doivent fournir une vue d'ensemble d'où les informations importantes peuvent être extraites rapidement (H5).
- Combiner plusieurs modèles graphiques de la même information semble inutile et superflu (H6).
- L'animation de l'exécution du programme, quoique naturellement utilisée par beaucoup d'étudiants, ne semble pas apporter de gain quantifiable (H7).
- Les mesures des usages nous indiquent que la programmation « sur l'exemple » est utilisée indifféremment par des néophytes ou par des étudiants plus qualifiés (H2). Cependant, les contraintes qu'elle impose sur l'ordre d'édition n'en font probablement pas le mode d'interaction le plus naturel pour nombre d'apprenants.

Ces mesures nous ont amenés à définir deux autres profils d'utilisation, eux aussi indépendants du niveau en programmation. L'association de ces trois profils à des caractéristiques cognitives de l'apprenant est une piste d'approfondissement, tout comme étudier la pertinence de l'approche en enseignement à distance.

REMERCIEMENTS

Nous souhaitons remercier les membres de l'équipe Multicom du laboratoire CLIPS-IMAG pour leur conduite de l'expérimentation oculométrique, et leur aide à l'exploitation des résultats.

BIBLIOGRAPHIE

1. Ben-Ari, M. *Constructivism in Computer Science Education*. in 29th ACM SIGCSE Technical Symposium on Computer Science Education. 1998. Atlanta Georgia: ACM press.
2. Blackwell, A. (2002). *What is Programming?* PPIG Workshop, Brunel University, London, UK.
3. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. . 1993, The MIT Press: Cambridge, Massachusetts. 604.
4. Du Boulay, B., *Some Difficulties of Learning to Program*, in Studying the Novice Programmer. 1989, Lawrence Erlbaum Asssocites. p. 283-299.
5. Duchâteau, C., *Images pour programmer*. Vol. 1. 2000.
6. Goold, A. and Rimmer, R. (2000). "Factors affecting Performance in Firs-Year Computing." SIGCSE Bulletin 32(2): 39-43.
7. Green, T. R. G. (1989). *Cognitive dimensions of notations*. People and Computers.
8. Käasboll, J., *Exploring didactic models for programming*. 1998 : Norsk Informatikk-konferanse, Høgskolen i Agder.
9. Lieberman, H., *Your Wish is my command*. 2001: Morgan Kaufmann. 416.
10. Myers, B. A. (1986). *Visual Programming, Programming by Example, and Program Visualization : A Taxonomy*. Human Factors in Computing Systems (CHI'86), New-York, ACM/SIGCHI.
11. Nogry, S. Jean-Daubias, S. Ollagnier-Beldame, M. (2004). *Evaluation des EIAH: une nécessaire diversité des méthodes*. TICE 2004, Compiègnes.
12. Pea, R.D., *Language-Independent Conceptual "Bugs" in Novice Programming*. Journal of Educational Computing Research, 1986. 2(1): p. 25-36.
13. *Teaching and Learning Computer Programming*, R.E. Mayer, Editor. 1988, Lawrence Erlbaum Associates. p.153-178.