Apprendre la programmation par l'exemple : méthode et système

Nicolas GUIBERT, Laurent GUITTET, Patrick GIRARD

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

Résumé

Alors que micro-ordinateurs et programmes informatiques se sont implantés dans de nombreuses disciplines scientifiques en tant qu'outils d'analyse ou qu'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 rapporte que, de par le monde, entre 25 et 80 % des étudiants dans un cursus d'initiation à la programmation sont en situation d'échec ou d'abandon. Nous présentons de manière synthétique une typologie des erreurs et des difficultés des programmeurs débutants. Nous explorons les utilisations pédagogiques d'un paradigme de programmation alternatif, la « Programmation sur Exemple ». Enfin, nous présentons un environnement d'apprentissage de la programmation et de l'algorithmique, MELBA (*Metaphor-based Environment to Learn the Basics of Algorithmic*), ainsi que la démarche didactique qu'il supporte.

Mots-clés : Interactions Homme-Machine, Didactique de la programmation, Programmation sur Exemple, Métaphores, Programmation graphique.

Abstract

Although programs take now a greater importance of programs as analysis or measurement tools in science (physics, chemistry, biology … in this case bio-informatics has even become a topic in itself), introductory programming courses are still known to be difficult for students. Kaasboll recently reports that their rate of drop out and failure vary from 25% to 80% worldwide. We present a typology of the difficulties in learning programming, and explore the pedagogical issues of an alternative programming paradigm, Programming by Example. Finally we present an innovating Learning Environment for programmers, MELBA (Metaphor-based Environment to Learn the Basics of Algorithmic), and the didactic approach it supports.

Keywords: Human Computer Interaction, Didactics and Psychology of Programming, Programming by Example, Metaphors, Visual Programming.

Introduction

La généralisation de l'ordinateur comme outil indispensable dans les sciences fondamentales (astrophysique, génétique, chimie moléculaire, …) a entraîné l'expansion de l'apprentissage de la programmation à une nouvelle audience universitaire. Cependant, la difficulté avérée des cours d'initiation (Kaasboll [1] relate que 25 à 80% des étudiants dans le monde sont en situation d'échec dans de tels modules – ce qui se traduit par une absence ou un échec à l'examen) nous amène à reconsidérer la pédagogie employée et les facteurs d'échec qui conduisent à ce constat alarmant. Pourquoi la programmation est-elle donc si difficile d'accès ? La synthèse de différents travaux de didactique nous permet d'apporter des éléments de réponse à cette question.

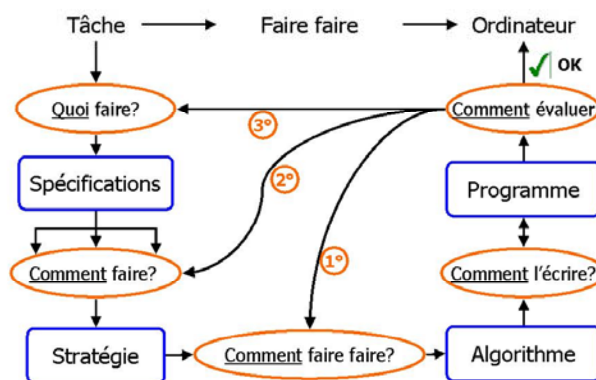


Figure 1 : Le cycle de conception d'un programme.

Classification des difficultés des étudiants.

Nous allons nous attacher à définir et à décomposer l'activité de programmation (figure 1), et de répertorier les difficultés essentielles à chaque étape. Programmer, qu'est ce ? C'est « faire faire une tâche à un ordinateur » [2].

D'un point de vue chronologique, l'activité de conception d'un programme débute donc par l'analyse et la modélisation précise de la tâche, où l'on se pose la question « quoi faire ? ». Cette étape n'est cependant généralement pas abordée en initiation à la programmation, mais reportée au cours de génie logiciel, ceci parce que les tâches que l'on se propose de faire automatiser par l'étudiant sont soit triviales, soit familières.

Les premières difficultés ne surgissent donc qu'un peu plus tard, lorsque le programmeur se pose la question « comment le faire ? ». Il faut pour répondre à

cette question construire un modèle viable de la *structuration temporelle* de la tâche à effectuer.

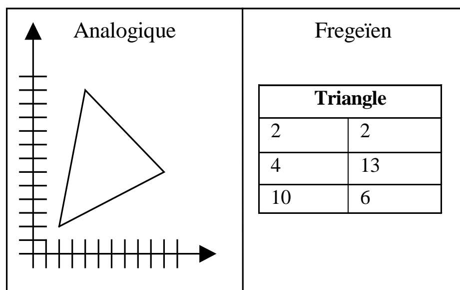
L'étudiant n'ayant pas réussi à construire un tel modèle ne parvient pas à abstraire correctement les différents comportements de la tâche, ou ne parvient pas à prévoir le comportement dynamique du «programme» qu'il a écrit.

Ensuite se pose le problème du « faire faire », c'est-à-dire la prise en compte de l'exécutant-ordinateur. Il faut pour répondre à cette question intégrer à sa stratégie un modèle viable de l'exécutant-ordinateur .

Une difficulté spécifique à la programmation est que, contrairement à d'autres sciences comme la physique, l'étudiant débutant en programmation n'a pas de modèle « naïf » viable de l'ordinateur, qui pourrait lui servir comme base pour construire des modèles plus sophistiqués [3]. Il en résulte deux types d'erreurs :

- Des erreurs anthropomorphiques [4 ; 5 ; 6], qui proviennent du fait que, en l'absence de modèle de fonctionnement de l'ordinateur, l'étudiant utilise par défaut le modèle d'un exécutant humain (ex. compréhension contextuelle des instructions… - donner exemple : nom de variable).
- Des erreurs liées à la numérisation [7] de la tâche, causées par la distance cognitive qui sépare les objets de l'univers de la tâche de leur formalisation dans le programme [8]. Généralement les objets du domaine de la tâche sont analogiques dans le sens où leur forme traduit leur «contenu» ; par opposition les représentations numériques utilisées dans les ordinateurs sont fregeïennes car elles ne supportent pas cette analogie entre la forme et le contenant (figure 2).

La dernière étape consiste à interpréter les résultats de l'exécution du programme. Cette interprétation présente la même difficulté.



Triangle	
2	2
4	13
10	6

Figure 2 : Représentations analogique et fregeïenne d'un triangle.

Cette classification démontre clairement qu'apprendre la programmation ne peut être réduit à apprendre la syntaxe d'un langage spécifique, tout comme il faut plus que la connaissance du solfège pour devenir compositeur de musique. Mais, surtout, elle met en exergue la pertinence de mettre en place des activités pédagogiques correspondant à chaque étape de ce processus, et de disposer pour cela d'un environnement support adapté.

En effet, l'utilisation des langages / environnements de programmation traditionnels, semble être la cause directe d'un ensemble de difficultés d'apprentissages que l'on pourrait qualifier d' « articulatoires », en référence à la théorie de l'action de Norman [9]. Celle ci associe la réalisation d'une tâche au parcours d'une distance – figure 3. Les distances articulatoires traduisent les difficultés à adapter l'intention de l'utilisateur aux commandes disponibles, et à interpréter l'état du système à partir de l'état de l'interface.

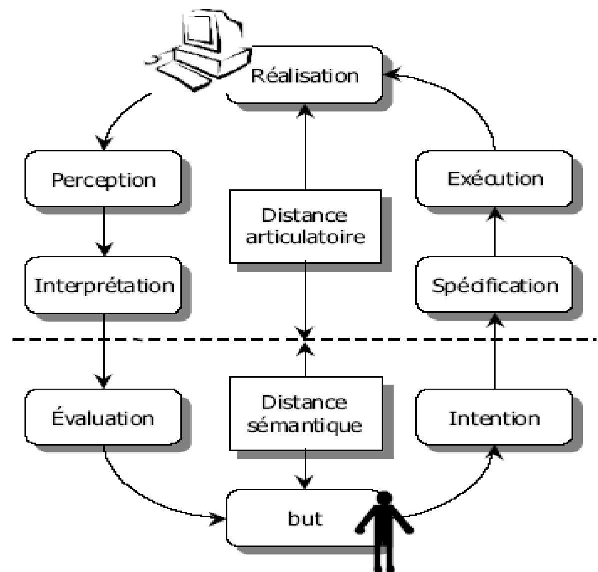


Figure 3 : Distances sémantiques et articulatoires.

La première difficulté tient à ce que ces environnements, étant destinés à un public d'experts, favorisent l'évaluation mais pas la conception de programmes.

Ainsi, coloration lexicale et analyse syntaxique, «debuggers», «assertions» fournissent ils un support à l'évaluation du «comment écrire», du «comment faire faire», du «quoi faire». Mais aucun outil n'est explicitement destiné à supporter le processus de la conception du programme.

On comprend dès lors mieux les difficultés exprimées par les étudiants dans Carbone, Hagan, et Sheard [10] :

> « … 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 … »

Deuxièmement, lorsque les étudiants connaissent leur première expérience concrète de la programmation, ils ont en premier lieu un retour sur les erreurs syntaxiques (comment écrire ?), qui leur sont signalées par le compilateur.

Plus insidieuses sont les erreurs sémantiques, où le programmeur débutant exprime avec des mots du

langage des instructions qui n'ont pas de sens pour l'exécutant, car il se fait une idée fausse du fonctionnement de celui-ci (comment faire faire ?).

Il doit alors détecter à partir de l'exécution du programme l'origine et la cause de l'erreur [8], c'est-à-dire reconstruire le lien correct entre les objets analogiques de la tâche et les données numériques de l'exécutant-ordinateur d'une part, et entre le comportement des objets analogiques et l'algorithme d'autre part.

Enfin les erreurs pragmatiques, résultant d'un défaut dans les spécifications du programme, ou d'une modélisation non exhaustive des comportements de la tâche sont classiquement les dernières (car les plus difficiles) à être détectées. Le retour d'erreur sur les différents processus qu'englobe la programmation s'effectue donc en sens inverse de leur exécution.

Le cycle de conception « classique » de la programmation, qui s'appuie sur l'écriture complète d'un programme, ne permet donc pas de séparer les différents problèmes : pour pouvoir évaluer le « comment faire » il faut avoir déjà répondu au problème « comment faire faire » et avoir évalué la solution ; et pour ce faire, il faut avoir écrit et corrigé syntaxiquement le programme. Cet état de fait permet de comprendre l'échec relatif des tentatives de séparer ces différentes étapes, comme le rapporte Rogalsky [11] :

> « L'observation d'élèves débutant en informatique dans un cadre scolaire semble indiquer que les méthodes d'analyse enseignées ne sont pas vues par les élèves comme un outil pour résoudre leurs problèmes de programmation, mais comme un contrat (avec l'enseignant) qu'il faut respecter. L'expression de la phase de travail qui correspond à l'analyse apparaît assez souvent comme une paraphrase du texte du programme écrit, qui peut précéder, accompagner, voire succéder à l'écriture directe dans un langage de programmation ».

Tout ceci démontre la piètre adéquation des langages et environnements de programmation actuels à un but d'apprentissage. Mais quel est l'impact de cette difficulté articulatoire sur les résultats d'un étudiant, comparativement à la difficulté intrinsèque d'une activité de conception ?

Un élément de réponse peut être extrait des résultats de deux études menées par Goold et Rimmer [12] et Wilson et Schrok [13] sur des étudiants en cours d'initiation à l'informatique. Leurs résultats corrèlent parfaitement le fait que le facteur prioritaire dans la réussite est ce qu'ils nomment respectivement « dégoût de la programmation » et « degré de confort pendant l'apprentissage » (table 1).

Ces termes englobent le niveau d'anxiété lors du travail sur la machine, la difficulté perçue de la compréhension des concepts – en soi et comparativement aux autres étudiants, et la difficulté perçue à réussir les exercices sur machine.

	Concepts de base.	Structure de données et Algorithmes.	
	Total	Examen	Total
Variable	Coefficients de régression		
Score moyen dans les autres modules	0,32 (2,07)*	0,89 (5,19)**	0,71 (3,35)**
A fait de la programmation avant l'université	-	-	12,32 (2,00)
Dégoût de la programmation	-12,50 (-2,52)*	-28,38 (-4,51)**	-22,40 (-2,94)**
Résolution de problèmes	1,36 (1,84)	-	-
Sexe (h|f)	7,52 (2,11)*	-	-

Table 1 : Effets de différentes variables sur les résultats des modules d'initiation à la programmation.

Un autre résultat qui nous semble démontrer l'impact primordial de cette difficulté « articulatoire » a été exhibé par Mancy et Reid [14] dans une étude récente portant sur 150 étudiants de l'université de Glasgow, étude qui cherchait les corrélations entre les caractéristiques cognitives des étudiants et leurs résultats en programmation.

Les résultats y montrent que la dépendance aux champs, caractéristique qui mesure la capacité (ou pas) à « extraire un élément particulier dans un champ de perception organisé » [15], c'est-à-dire à isoler des informations élémentaires indépendantes d'un message complexe, est critique dans l'apprentissage de la programmation (table 2)

	Field dependent	Field intermediate	Field independent
Score range on test	0-11	12-16	17-20
Number of students	55	54	45
Average final exam mark	44.0%	53.5%	63.5%

Table 2 : Influence de la dépendance au champ sur les résultats du module d'initiation à la programmation.

Cette caractéristique s'y révèle même être un indicateur statistiquement plus significatif de la capacité à programmer que les tests spécialisés tel le PAT (IBM Programming Aptitude Test). Ces résultats nous paraissent pouvoir être corrélés avec l'utilisation d'environnements d'apprentissage qui mélangent dans une seule représentation (celle du programme final) tous les modèles sous-jacents de l'activité de programmation.

Si sauter 6 mètres en triple saut est pratiquement à la portée de tous, faire de même en saut en longueur n'est possible que pour une minorité d'athlètes très entraînés; nous avons la conviction que, de même, fournir un environnement d'apprentissage qui permette de s'attaquer à chaque modèle séparément, en autorisant pour chacun une expérimentation sur machine, faciliterait l'apprentissage.

Encore faut il pour cela disposer d'un autre

paradigme de programmation, qui supporte l'élaboration d'une stratégie dans le domaine de la tâche (comment faire ?), et qui puisse servir de support à un modèle de l'exécutant-ordinateur, dans le but d'illustrer ses capacités et son fonctionnement (comment faire faire ?).

La Programmation sur Exemple.

Dès la fin des années 70, la difficulté d'accès de la programmation a conduit à l'émergence d'un axe de recherche en Interface Homme Machine nommé « End User Programming ».

L'utilisateur « final » dont il est question est défini comme un utilisateur confirmé de l'outil informatique, mais qui n'a pas de connaissance en programmation [16].

L'idée de base est que savoir comment accomplir une tâche devrait être suffisant pour la programmer. C'est dans ce cadre que Smith a introduit avec Pygmalion [7] le paradigme de « Programmation sur Exemple » (PsE).

Illustrons ce paradigme alternatif avec un exemple tiré de Pygmalion, système «pionnier» de la programmation sur exemple, qui permet de se focaliser sur les concepts de la PsE sans risquer d'interférence avec d'autres concepts.

La tâche concernée est la recherche du minimum de deux nombres.

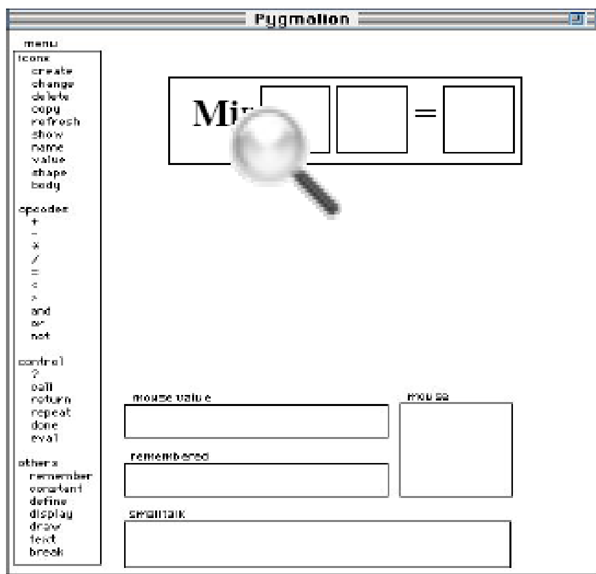


Figure 4 : Vue d'ensemble de Pygmalion

Premièrement, nous définissons la fonction « Min », en créant une icône de fonction et en la nommant ; les paramètres et la valeur de sortie ont la forme de boîtes permettant aussi bien de saisir du texte que de déposer les valeurs d'autres boîtes du même type. La valeur qu'elles contiennent peut être soit un littéral, soit le résultat d'un calcul d'expression.

Il nous reste alors à définir un exemple concret, pour faire à l'ordinateur la démonstration du comportement attendu du programme ; pour cela nous remplissons nos deux champs de saisie des paramètres. Puis nous créons une icône si … alors … sinon, et nous déposons dans la boîte de la condition l'icône correspondant à la fonction « < ».

Enfin, par glisser-déposer, nous remplissons le premier argument de « < » avec le premier argument de la fonction Min (figure 5).

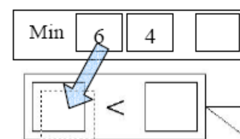


Figure 5 : passage de paramètres dans Pygmalion.

De la même façon, nous déposons le deuxième argument de Min dans l'autre case. Le système de programmation, étant interactif, calcule automatiquement le résultat de cette expression (false) et nous aiguille dans le cas « sinon ». Pour compléter ce cas, il nous suffit de déposer le second argument de « Min » dans la case de la valeur de retour et le tour est joué. Nous avons programmé :

```
« function Min (integer a,
                integer b)
                return integer
 begin
      if (a<b) then ????
      else return b
 end »
```

Lorsque cette fonction est appelée dans le cas (a<b), le système repasse automatiquement en mode édition et attend les instructions du programmeur.

Ultérieurement, ces systèmes se sont développés en deux grandes familles, des systèmes que nous qualifierons de « sémantiques » et d'autres que nous qualifierons de « pragmatiques ».

Les premiers représentent explicitement l'exécutant au travers d'une métaphore, et choisissent de se positionner au niveau «comment faire faire». De tels environnements prennent souvent la forme d'un micromonde dédié à la programmation.

Ils diminuent les difficultés habituelles du niveau «faire faire» de par leur représentation explicite de l'état et des capacités de l'exécutant, et par la technique de programmation par démonstration, qui a pour but de faciliter le processus de conception, et non d'évaluation, du programme. L'utilisation d'une métaphore permet de décrire ceux-ci en des termes connus du programmeur.

Un exemple classique de système de PsE sémantique est l'environnement de programmation ToonTalk [17], destiné à des enfants. La métaphore utilisée pour figurer l'exécutant-ordinateur est celle d'un jeu de construction de style LEGO, et la figure 6 illustre la correspondance entre concepts de la programmation et objets du micromonde.

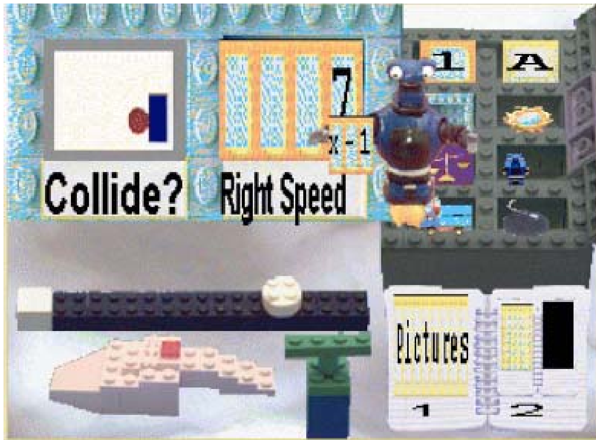


Figure 6 : La métaphore du jeu de constructions dans ToonTalk.

A contrario, les systèmes dits « pragmatiques » de ne représentent pas explicitement l'état de l'exécutant-ordinateur, mais se positionnent au niveau «comment faire».

C'est pourquoi l'état de l'exécutant est mis dans une «boite noire» et ses capacités sont exprimées en termes de la tâche à accomplir : les instructions élémentaires du système sont des tâches élémentaires du domaine.

Ces systèmes se basent sur une métamodélisation des tâches du domaine pour paramétrer et structurer la séquence d'interactions de l'utilisateur. De la sorte, ils parviennent à maintenir complètement la programmation dans le domaine de la tâche, en contrepartie d'une certaine perte d'expressivité et de généricité.

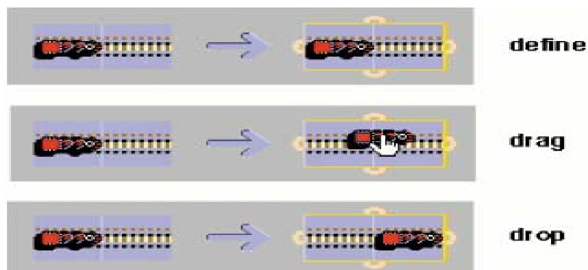


Figure 7 : Etape de construction d'un simulateur de réseau ferroviaire avec StageCast Creator.

On peut trouver des exemples de systèmes de PsE «pragmatiques » dans le domaine de la conception de simulations, avec StageCast Creator (figure 7), ou encore dans les domaines de la géométrie dynamique et de la CAO paramétrique, tel qu'EbP (figure 8) [18].

Dans EbP - Example-based Programming –, le mécanisme de programmation par exemple permet de générer de façon semi-automatique le programme structuré ci dessous à partir du dessin en mode interactif de la figure 8, puis de reconstruire ce schéma dans un autre contexte, défini par la lecture graphique des deux extrémités du segment support.

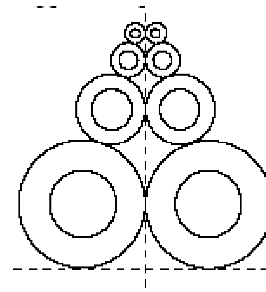


Figure 8 : Un dessin interactif sous EbP.

```
 « S1 : segment ;
   D1 : droite ;
   C1, C2, C: cercle ;
 S1←segment (lire_pt (),
             lire_pt ()) ;
 D1←mediatrice (S1) ;
 C1←cercle (D1, S1) ;
 C ←symetrique (C1, D1) ;
 C2←cercle (C1.centre,
           (C1.rayon)/2) ;
 C←symetrique (C2, D1) ;
 TANT QUE (C2.rayon >= 1.0)
 FAIRE
      C1←cercle (D1, C1) ;
      C ←symetrique (C1, D1) ;
      C2←cercle (C1.centre,
                (C1.rayon)/2) ;
      C←symetrique (C2, D1) ;
 FIN TANT QUE »
```

Utiliser l'exemple dans l'apprentissage de la programmation : MELBA.

Nous avons conçu l'environnement d'apprentissage MELBA (Metaphors-based Environment to Learn the Basics of Algorithmic), dans le but de mesurer l'impact sur l'apprentissage :

- De la représentation graphique explicite des différents modèles sous-jacents à la conception de programmes.
- Du support explicite par un paradigme d'interaction adapté du processus de conception et pas seulement d'évaluation.

La Programmation sur Exemples, pragmatique et sémantique, y a été adaptée pour fournir un support à une initiation incrémentale et expérimentale à la programmation.

MELBA est composé de trois panels : l'espace du programme, qui permet de représenter et d'éditer celui ci, un micromonde sémantique (représentant l'exécutant) et un micromonde pragmatique. Chaque zone y est interactive, et la cohérence de l'ensemble est

assurée par des mécanismes de programmation sur ou avec exemples.

Dans les sections suivantes, nous explorerons les différentes parties de MELBA, en reliant leur utilisation au modèle de conception d'un programme (figure 1).

Modélisation de la tâche : PsE pragmatique et apprentissage.

Le premier panel de l'environnement est un micromonde qui présente les capacités de l'exécutant dans le domaine de la tâche (figure 9).

Il permet ainsi de séparer l'apprentissage du « faire faire » de celui des types & structures de données. Il supporte la Programmation sur exemples.

L'objectif dans ce module est de permettre à l'étudiant d'appréhender la structuration temporelle des algorithmes. Le fonctionnement interne de l'exécutant n'y est donc pas explicitement représenté. Le but des exercices manipulant cet agent est de construire des programmes composés d'instructions élémentaires du domaine de la tâche.

Dans l'exemple figure 9, la tâche consiste à remplir un alignement de verres avec une pipette. Les opérations possibles sont : remplir la pipette, presser une goutte, aller au premier verre, aller au verre suivant.

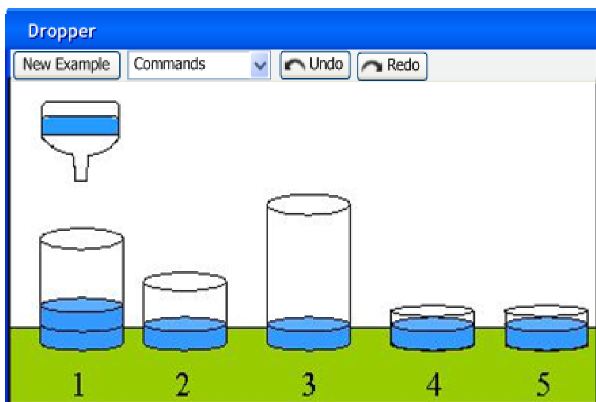


Figure 9 : Micromonde *pragmatique* de MELBA

L'utilisation de la programmation sur exemples permet de franchir la barrière de la « page blanche », car le micromonde de la tâche est interactif, et génère l'écriture d'une séquence d'actions dans l'éditeur du programme.

Celui ci fournit ensuite une aide pour restructurer la trace à partir des opérateurs temporels de la programmation impérative (conditionnelle, itération).

Inversement, lorsque l'étudiant édite son programme, chaque instruction y est exécutée interactivement, de sorte qu'il voie à tout instant l'état courant de la tâche.

Dans l'objectif pédagogique d'apprendre à modéliser le déroulement d'une tâche par un programme, celui-ci est la connaissance cible, le modèle que l'étudiant doit apprendre à générer et animer mentalement.

Par opposition, l'état de la tâche permet de modéliser la connaissance « concrète » de l'étudiant.

De la sorte, les mécanismes de programmation « sur » et « avec » exemple se complètent de façon à offrir un support à la compréhension aussi bien par connotation que par dénotation [19], figure 10 : l'étudiant peut partir son savoir-faire sur la tâche (« Concrete Experience ») et induire un programme (« Abstract Conceptualisation ») avec l'aide du mécanisme de programmation sur exemples (approche par connotation), ou au contraire chercher à comprendre un programme grâce à l'animation du micromonde de la tâche (approche par dénotation).

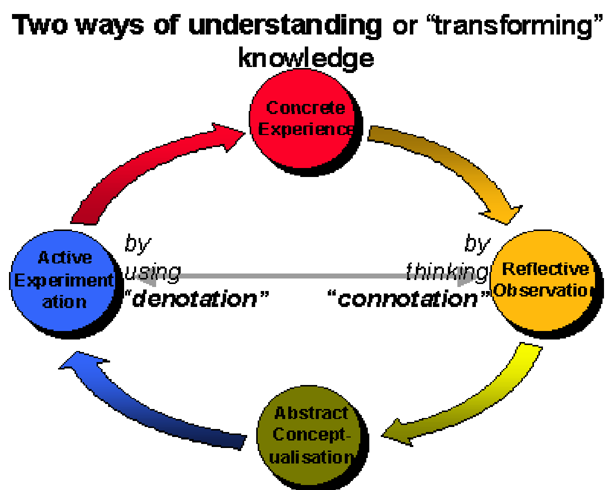


Figure 10 : cycle d'apprentissage expérimental selon Kolb.

L'étape suivante, après que l'étudiant ait appris à construire et analyser un algorithme, est de comprendre le fonctionnement interne de l'exécutant-ordinateur. Pour ce faire, nous introduisons un nouveau modèle graphique directement manipulable.

Modélisation de l'ordinateur : PsE sémantique et apprentissage.

Pour cela, un deuxième micromonde, qui représente le fonctionnement interne de l'exécutant ordinateur, est introduit. Celui-ci se base sur la métaphore du bureau et est composé (figure 11) :

- D'un gestionnaire de « fichiers » pour représenter le contexte du programme : c'est la « mémoire à long terme » de l'exécutant-ordinateur.
- D'une calculatrice symbolique, permettant d'appréhender des expressions de différents types.
- D'un panel représentant le contenu de la « mémoire de travail » de l'ordinateur. Celui-ci est extrêmement éphémère, et ne peut contenir qu'un seul élément. L'évaluation des expressions de la calculatrice est stockée provisoirement dans ce tampon.

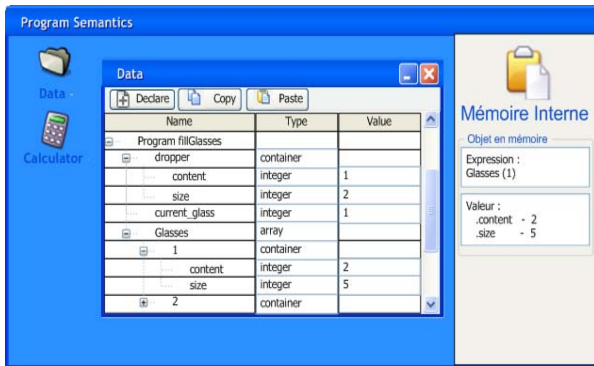


Figure 11 : environnement *sémantique* de MELBA.

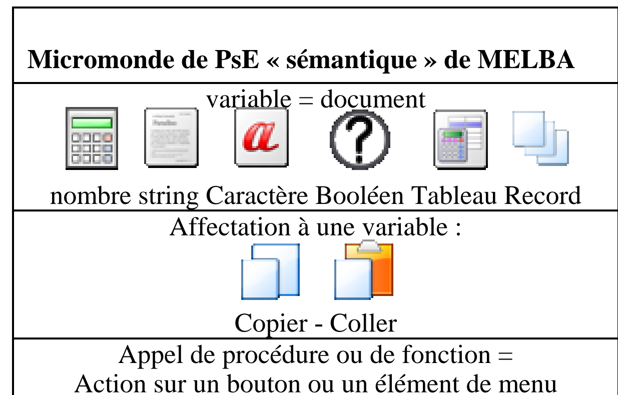
De même que pour la partie pragmatique, il est synchronisé avec le programme par des mécanismes de programmation sur/avec exemples, qui permettent d'appréhender par construction ou animation les concepts de variables, types, et expressions. De plus, il est associé à une pragmatique pour relier explicitement la représentation « fregiënne » des objets dans l'exécutant-ordinateur à une représentation analogique.

Notons cependant qu'à la différence de l'étape précédente, c'est micromonde qui représente le « modèle » (abstract conceptualisation) à acquérir, alors que le micromonde pragmatique représente l'expérience concrète. La programmation sur l'exemple depuis le micromonde sémantique a donc un aspect « dénotationel » pour ce qui est de la compréhension du modèle de l'exécutant en lui-même, mais connotationel pour ce qui de la compréhension du fonctionnement de l'affectation.

Le choix de la forme de ce micromonde a été motivé par la volonté de partir de la conception « anthropomorphique » de l'exécutant-ordinateur pour construire un modèle cohérent.

Nous avons choisi de représenter l'état de l'exécutant (le contexte du programme) en utilisant la métaphore du bureau (tableau 3). Celle ci a en effet deux avantages :

- Elle est déjà familière à notre public, et est déjà associée à la modélisation de « l'intérieur » de l'ordinateur.
- Elle permet une correspondance très forte entre les objets de la métaphore et les concepts de la programmation. Par exemple, un document est une « boîte » avec un nom qui contient une (et une seule) donnée typée (= variable).



Micromonde de PsE « sémantique » de MELBA
variable = document nombre string Caractère Booléen Tableau Record
Affectation à une variable : Copier - Coller
Appel de procédure ou de fonction = Action sur un bouton ou un élément de menu

Tableau 3 : La métaphore de l'environnement sémantique.

Modalités d'évaluation.

L'étape suivante consiste donc à mener une étude sur l'usage de MELBA en conditions réelles. Celle-ci analysera le comportement et les performances de 60 étudiants d'IUP de bioinformatique dans un module d'initiation, dans le but de mesurer l'influence d'un environnement dédié à l'apprentissage et l'apport de l'approche sur exemples dans l'acquisition des concepts de la programmation.

Nous nous proposons de conduire cette analyse selon deux axes :

a) Validation de l'utilité d'un environnement basé sur deux niveaux de modélisation du processeur (points de vue *externe* – pragmatique – et *interne* –sémantique-).
b) Validation d'une approche « sur exemple ».

Ceci nous a entraîné à définir quatre groupes pour l'étude :

a) groupe travaillant sans outil, avec une méthode d'enseignement « classique ».
b) groupe travaillant sans outil, avec une méthode d'enseignement à partir d'exemples (çad que le programmeur exécute virtuellement la tâche à la place du processeur, puis structure son programme à partir de la trace).
c) groupe travaillant avec un outil de programmation de type Visual Programming (un sous-ensemble de Melba sans sa partie PsE ; l'exemple n'est pas directement manipulable, mais seulement visualisé pour évaluer interactivement le programme).
d) groupe travaillant avec MELBA, en suivant une méthode d'enseignement à partir d'exemples.

Auparavant, les étudiants auront rempli un questionnaire sur leurs connaissances préalables, pour filtrer ceux qui ont déjà suivi un cours d'initiation.

Ceux-ci suivront des séances de révision qui s'accompagneront d'une évaluation de l'outil (Melba).

D'autre part, les étudiants passeront un test pour déterminer leur style d'apprentissage et leur caractéristique de dépendance au champ.

De façon à garder une trace du cheminement suivi par les étudiants, ceux suivant un cursus TD sans outils verront leur brouillons photocopiés pour analyse des traces. Les outils permettront d'enregistrer les traces des interactions.

De plus, les étudiants seront soumis à un test « scolaire » par semaine, dans le but d'évaluer leur progression, aussi bien en connaissance des concepts, qu'en conception et évaluation de programmes. Nous retiendrons comme critères pour l'évaluation en cours de TD le temps de résolution de l'exercice et le degré de correction de la solution.

L'évaluation englobera les concepts suivants : structures de contrôle, variable et affectation, données & résultats (E/S), qui seront abordés dans cet ordre. La notion de constructeur de types, même si elle sera intégrée à MELBA, ne sera pas incluse dans l'évaluation. Cependant, des exercices pourront faire appel à la manipulation de listes à accès séquentiel et direct.

Conclusion

Dans cet article, nous avons introduit les différentes difficultés liées à l'initiation à la programmation. Nous émettons l'hypothèse que ces difficultés sont en fait renforcées par les moyens d'interaction avec l'exécutant-ordinateur employés classiquement (ils sont non interactifs, abstraits, et requièrent l'apprentissage d'une syntaxe difficile. Ils nécessitent la construction l'animation d'une représentation mentale complexe de l'état de la machine …).

Nous suggérons d'adopter une nouvelle méthode d'apprentissage non axée sur le langage, et qui permette de mieux séparer l'apprentissage des concepts de la programmation.

Pour supporter celle-ci, nous étudions l'utilité d'un paradigme de programmation alternatif, la programmation sur exemples.

Nous présentons un nouvel environnement d'apprentissage de la programmation, MELBA, construit à partir de ces concepts pour supporter cette méthode d'apprentissage, et les modalités d'évaluation de ce système et de cette méthode.

Références

Livres

[2] Duchâteau, C. (2000). Images pour programmer. Namur, Facultés Universitaires Notre Dame de la Paix.

[8] Arsac, J. (1991). Préceptes pour programmer. Paris, Dunod.

[9] Norman, D. A. (1990). The design of every day things. NewYork NY, USA, Doubleday Currency.

[10] Carbone, Hagan, et Sheard 1998

[11] Rogalsky et Hoc 1988

[15] Witkin, H. A., and Goodenough, D. R. (1981) Cognitive Styles: Essence and Origins Field Dependence and Field Independence. International University Press, New York

[16] Cypher, A., Ed. (1993). Watch What I Do: Programming by Demonstration. Cambridge, Massachusetts, The MIT Press.

[18] Lieberman, H. (2001). Your Wish is my command, Morgan Kaufmann.

[19] Kolb, D. A. (1984). Experiential Learning: Experience as the Source of Learning and Development. Prentice-Hall, Inc., Englewood Cliffs, N.J.

Sections de livres.

[6] Du Boulay, B. (1989). Some Difficulties of Learning to Program. Dans : Studying the Novice Programmer, Lawrence Erlbaum Asssocites: 283-299.

[7] Smith, D. C. (1993). Pygmalion, An Executable Electronic Blackboard. Dans : Watch What I Do : Programming by Demonstration. A. Cypher. Cambridge, Massachusetts, The MIT Press.

[17] Kahn, K. (2001). How Any Program Can Be Created by Working with Examples. Dans : Your Wish is My Command. H. Lieberman, Morgan Kaufmann ed. , 21-44.

Articles de Revue

[3] Ben-Ari, M. (1998) Constructivism in Computer Science Education, ACM SIGCSE Bulletin, 30(1): pp. 257-261.

[4] Pea, R. D. (1986). “Language-Independent Conceptual “Bugs” in Novice Programming.” Journal of Educational Computing Research 2(1): pp. 25-36.

[12] Goold, A & Rimmer, R. (2000) Factors affecting performance in First Year Programming, ACM SIGCSE Bulletin, Vol. 32, pp. 39 - 43.

[13] Wilson, B. C. & Schrock, S. (2001) ACM SIGCSE Bulletin, Vol. 33, pp. 184-188.

Actes de Conférences

[1] Kaasboll J. (1998) Exploring didactic models for programming.

Norsk Informatikk-konferanse, Høgskolen i Agder.

[5] Spohrer, J. (1986) Analysing the high frequency bugs in novice programs.First Workshop on Empirical Stufies of Programmers.

[14] Mancy R., Reid N. (2004) Aspects of cognitive style and programming. PPIG 2004, 16th Annual Workshop.