Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Ecole Doctorale des Sciences Pour l'Ingénieur ▪ Université de Poitiers

THESE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National — Arrêté du 30 mars 1992)

Spécialité : INFORMATIQUE

Présentée et soutenue publiquement
devant la Commission d'Examen le 7 Décembre 2006 par

Nicolas Guibert

Validation d'une approche basée sur l'exemple pour l'initiation à la programmation.

Directeurs de Thèse : ***Patrick Girard et Laurent Guittet***

JURY

Président :		
Rapporteurs :	Philippe TRIGANO	Professeur, Unité de Technologie de Compiègne
	Jean – Pierre PEYRIN	Professeur, Université Joseph Fourier, Grenoble
Examinateurs :	Charles Duchateau	Professeur, FUNDP, Namur
	Patrick Girard	Professeur, Université de Poitiers
	Laurent GUITTET	Maître de Conférences, ENSMA, Futuroscope
	Guy PIERRA	Professeur, ENSMA, Futuroscope

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Laboratoire d'Informatique Scientifique et Industrielle (E.A. 1232)
Téléport 2 — 1, avenue Clément Ader — BP 40109 — 86961 Futuroscope Chasseneuil cedex — France
(+33/0) 5 49 49 80 63 — (+33/0) 5 49 49 80 64 — www.lisi.ensma.fr

Résumé

Alors qu'ordinateurs et programmes informatiques se sont implantés dans nombre de disciplines scientifiques en tant qu'outils d'analyse ou instruments de mesure, l'acquisition des compétences requises pour la conception de programmes ne se fait pas aisément. De nombreuses études ont caractérisé les erreurs et difficultés rencontrées par les programmeurs novices. Les environnements actuellement utilisés pour l'apprentissage de la programmation se composent d'outils conçus dans une unique optique de développement, et non pas dans un cadre explicitement pédagogique. A cette approche « industrielle » s'oppose une approche explicitement pédagogique, où le but premier est la découverte et la construction de connaissances, et non pas la réalisation de tâches techniques. Cette Thèse étudie l'usage d'un paradigme de programmation alternatif, la programmation graphique sur exemple, comme support à la construction active d'un savoir viable par l'étudiant, en s'appuyant sur des expérimentations en situation réelle avec un environnement adapté, conçu explicitement pour l'apprentissage.

Abstract

Although computers and programs have now become essential in experimental sciences as analysis or measurement tools, many students still find learning Computer Science is extremely difficult. Many studies have characterised the errors and difficulties encountered by novice programmers. Environments in use nowadays for learning programming are tools built in the unique perspective of development, and not in a pedagogical perspective. This "industrial" approach is often opposed to a genuine pedagogical approach, where the goal is discovery and acquisition of knowledge, and not the realisation of technical tasks. This thesis explores the use of an alternative interaction paradigm, "programming by example", to support the student's active construction of viable knowledge, by the use of experimental studies led in a concrete environment, with an adapted programming by example environment, engineered specifically for a pedagogical purpose.

Table des Matières

Table des figures

Liste des tableaux

Je tiens avant tout à remercier :

Patrick Girard, mon directeur de thèse, pour la confiance qu'il m'a accordée dans la réalisation de cette thèse et pour l'enrichissement personnel qu'il m'a procuré par ses connaissances.

Laurent Guittet, pour son dévouement et sa disponibilité pendant cette thèse.

Jean-Pierre Peyrin et **Philippe Trigano** pour avoir accepté de rapporter mon travail de thèse.

Charles Duchâteau, pour avoir accepté de faire parti de mon jury.

Guy Pierra, Directeur du LISI, pour m'avoir accueilli dans son laboratoire et pour sa participation à mon jury, **Claudine Rault,** secrétaire du LISI, pour la réalisation des tâches administratives (j'aurai bien été incapable d'y arriver sans elle), mes « voisins de bureau », **Mickey**, **Loé** et **Francis** de grenoble, et aussi les autres membres du laboratoire, et particulièrement **Boulou, Dago, David, Ricou, faby, Fred & Fred** (qui se reconnaîtront)**, JCP, Jéjé, Ladjel, Manu, Pascal, Yamine, Tex, Micky, Hondjack, Dung, Youcef, Stéphane, Karim, Hieu, Gaëlle** et **Sybille,** et aussi **Antoine** et mes compères doctorants du SIC pour les agréables moments passés en leur compagnie

Et enfin, mes amis, ma famille et plus spécialement mes parents, pour leur soutien sans faille tout du long de cet éprouvant marathon.

A tous, du fond du coeur,

Merci.

Introduction générale

Ces dernières années, la puissance toujours croissante des ordinateurs a entraîné l'adoption de l'informatique comme outil indispensable dans presque tous les corps de métiers. Les disciplines scientifiques ne dérogent pas à la règle, et aujourd'hui des programmes informatiques sont utilisés couramment que ce soit comme outil de simulation, de modélisation ou d'analyse, dans un grand nombre de domaines ; on peut notamment citer la génétique, la physique des particules, l'astrophysique, la météorologie, la géologie, la botanique…

Des spécialistes de l'usage de l'outil informatique pour la recherche en science ont ainsi fait émerger des disciplines telles que la bioinformatique, ou encore l' « informatique physique » - *computational physics.* Cette intégration croissante de l'informatique, et en particulier des bases de données et de la programmation, dans les disciplines scientifiques, pose avec une acuité accrue le problème de l'apprentissage de l'informatique dans une optique réellement pluri-disciplinaire (et non plus seulement pour former des ingénieurs analystes-programmeurs). En effet, l'acquisition des compétences requises pour utiliser l'ordinateur comme outil d'analyse scientifique, et, plus encore, pour la conception de programmes est reconnue comme étant très difficile : entre 25 et 80 % des étudiants échouent ou abandonnent en cours d'initiation à la programmation en université. Cette statistique illustre parfaitement la difficulté d'accès à cette matière pour des débutants complets.

Ces constats ont mené à l'émergence de la « Psychologie de la Programmation », un champ de la psychologie cognitive intéressé par les processus cognitifs mis en œuvre par les programmeurs pendant le développement. De nombreuses recherches ont ainsi été menées concernant le recensement et la catégorisation des erreurs de l'apprenant, la construction de modèles de schémas de connaissances et de compétences en programmation, les facteurs pouvant influer sur le succès en apprentissage académique…

Parallèlement, avec les progrès des capacités graphiques des machines, des recherches en informatique dans le champ de l'Interaction Homme-Machine se donnèrent pour objectif de rendre la programmation accessible à un public plus large : on parle ainsi de « End User Development » (développement par l'utilisateur final). Halbert, le premier, puis Cypher (Cypher 1993) à sa suite, ont formalisé cette démarche :

> *« Le fait qu'un utilisateur soit capable d'exécuter une tâche dans un environnement donné devrait être suffisant pour que le système soit en mesure de créer un programme qui exécute cette tâche. »*

Cette mouvance fut initiée dans les années 70 par Smith (Smith 1993), dont le système Pygmalion introduisait les concepts d'icône, de métaphore graphique, de programmation

« visuelle », et de programmation « basée sur exemple ». Plus tard, Halbert (Halbert 1984) et Myers (Myers 1986) à sa suite ont défini ce dernier paradigme :

> *« Un système est dit « basé sur exemple » si une instance d'exécution se déroule parallèlement à la conception du programme. »*

Depuis les débuts, qui avaient pour cadre la programmation (Pygmalion – (Smith 1993)) ou l'apprentissage du LISP (Tinker – (Lieberman 1993)), de nombreux environnements basés sur ce paradigme (pour la plupart des prototypes de recherche) sont apparus, couvrant un plus large spectre d'applications (personnalisation de logiciels, CAO, agents d'aide, composition, recherche dans un SIG… (Lieberman 2001)), s'écartant ainsi du domaine initial de la programmation. Cette succession de systèmes se caractérise par une fuite en avant technologique toujours plus grande :

- usage d'algorithmes d'intelligence artificielle de plus en plus élaborés dans les systèmes d'aide (Lau, Wolfman et al. 2001),
- programmation à base d'agents dans StageCast Creator (Smith 2000) ou Agentsheets (Repenning and Perrone 2001),
- usage d'un micromonde en 3D dans ToonTalk (Kahn 2001),
- exploration de nouveaux styles de dialogue entre l'environnement et l'utilisateur (McDaniel and Myers 1999), (Wolber 1996)),

…et par une fortune variable. Nombre d'applications en sont restées au stade de prototype, ce qui fait que ce paradigme n'a jamais fait l'objet d'une véritable validation scientifique, en quelque domaine d'application que ce soit. Le rôle de chacune des techniques associées à ce paradigme dans le succès ou l'échec de tel et tel système dans tel ou tel contexte (et donc la validité des concepts sous-jacents) reste donc inconnu.

A la croisée de ces deux mouvances, nos travaux ont pour objectif d'étudier la pertinence des techniques de programmation interactive « basées sur l'exemple » dans la construction d'un environnement d'apprentissage de l'algorithmique, soutenant la thèse de la validité d'un environnement interactif basé sur des exemples concrets pour apprendre à programmer. A l'instar des travaux en programmation interactive, nous partons de l'hypothèse qu'une partie de la difficulté et de la frustration expérimentées par certains étudiants en phase d'initiation provient de l'usage en enseignement d'environnements de programmation pour professionnels, peu adaptés à cet emploi.

Pour dépasser le stade des idées reçues et des évidences, nous nous attachons à étayer cette hypothèse fondatrice en nous appuyant sur la bibliographie de la « Psychologie de la Programmation » (Psychology of Programming), et de façon plus restreinte sur celle de la didactique de la programmation. Puis nous cherchons à définir ce que serait un environnement d'apprentissage adapté pour l'initiation à l'algorithmique et la programmation. Pour cela, nos travaux de recherche ont suivi une démarche de recherche en trois étapes.

Premièrement, nous avons réalisé une analyse préalable des différentes techniques d'interaction mises en œuvre par la programmation « visuelle » et la programmation « basée sur l'exemple », en nous appuyant sur des métriques issues de l'ergonomie cognitive. Cette phase nous ont permis de synthétiser un cahier des charges des composants et fonctionnalités attendues dans un environnement informatisé pour

l'apprentissage humain (EIAH) en initiation à l'algorithmique. Deuxièmement, nous avons implémenté ce cahier des charges dans la réalisation d'un EIAH basé sur l'exemple, qui supporte la construction de Situations d'Apprentissage Actives pour l'algorithmique, et pas la construction de composants logiciels. Nous l'avons appelé MELBA : Metaphor-based Environment to Learn the Basics of Algorithmics. Troisièmement, nous avons mené une évaluation rigoureuse, basée sur des expérimentations de l'outil en milieu réel et en milieu contrôlé.

Ce manuscrit s'appuie sur la structure de notre démarche, et se décompose en cinq étapes :

Le premier chapitre s'attache à étayer l'hypothèse fondatrice selon laquelle un environnement d'apprentissage inadapté peut jouer un rôle (négatif) de catalyseur dans l'apparition des difficultés des apprenants et dans la génération d'une frustration pouvant devenir, en elle-même un obstacle sérieux à la compréhension. Pour cela, nous nous appuyons sur la bibliographie existante en « Psychologie de la Programmation ».

Le second chapitre définit et illustre le paradigme alternatif de « programmation basée sur l'exemple ». Après avoir expliqué les différents concepts associés à ce paradigme, en nous appuyant sur des exemples ad hoc ou issus de la littérature, nous cherchons à dépasser les préjugés intuitifs, en discutant précisément les contraintes et les différentes applications de cette approche, dans le domaine de la programmation.

Dans le troisième chapitre, notre contribution consiste à déterminer le contexte de pertinence des techniques interactives utilisées dans les environnements de programmation et / ou d'apprentissage, d'un point de vue pédagogique. Nous nous appuyons dans notre analyse sur un outil d'évaluation de notations, issu du champ de l'ergonomie cognitive, le « Cognitive Dimensions Framework » (Green 1989). Nous détournons légèrement cet outil de son utilisation « classique », qui est d'évaluer une notation ou un langage spécifique, en l'utilisant pour analyser, d'un point de vue cognitif, et de façon globale, les différents types d'interaction et de visualisation présentés dans la littérature. Cette analyse nous permet de relier les techniques d'interaction employées aux savoirs et aux savoir-faire décrits dans le premier chapitre. A partir de là, nous sommes à même de définir la combinaison de techniques d'interaction la plus pertinente pour réaliser le cahier des charges du chapitre précédent.

Le quatrième chapitre décrit l'environnement MELBA, qui implémente ce cahier des charges, en proposant tout d'abord une vue d'ensemble de sa structure, avant de détailler l'environnement destiné aux apprenants et les interactions entre ceux-ci et le système, ainsi que la conception et l'intégration d'exercices par un enseignant. Nous discutons dans cette présentation des choix de conception en les mettant en relation avec le cahier des charges.

Dans le cinquième chapitre, nous proposons d'évaluer de façon rigoureuse l'efficacité de MELBA pour l'apprentissage de la programmation. Pour cela, dans un premier temps nous passons en revue et classifions les différentes techniques et métriques d'évaluation des EIAH, en les illustrant à l'aide d'exemples de la littérature, ayant pour contexte la programmation. Nous discutons de leur pertinence dans le cadre de l'évaluation d'un EIAH de la programmation tel que MELBA. Puis nous présentons les protocoles des expérimentations menées avec l'outil et en analysons les résultats.

Enfin nous concluons ce mémoire en esquissant une réponse au sujet de cette thèse, et en indiquant les perspectives de recherches qu'ouvrent ces travaux.

Chapitre 1

Initiation à La Programmation: classification des difficultés et analyse des besoins.

Résumé. *Les premiers pas de l'apprentissage de la programmation sont réputés difficiles : jusqu'à 80% des étudiants dans un cours d'initiation à la programmation échouent ou abandonnent.*

Dans cette thèse, nous émettons l'hypothèse qu'une partie de la difficulté et de la frustration expérimentées par certains étudiants en phase d'initiation provient de l'usage en enseignement d'environnements de programmation pour professionnels, peu adaptés à cet emploi. Nous investiguons ce que pourrait être un environnement d'apprentissage adapté, et tentons d'en produire une implémentation concrète, étayée par plusieurs études expérimentales.

Le présent chapitre s'attachera à étayer cette hypothèse fondatrice selon laquelle un environnement d'apprentissage inadapté peut jouer un rôle (négatif) de catalyseur dans l'apparition des difficultés des apprenants et dans la génération d'une frustration pouvant devenir, en elle-même un obstacle sérieux à la compréhension. Pour cela, notre contribution consistera à extraire, à partir de la bibliographie existante en « Psychologie de la Programmation », une réponse synthétique aux questions suivantes :

« Qu'appelle-t-on programmer ?»

« Quels sont les mécanismes cognitifs d'apprentissage de la programmation ? »

« Quelles sont les erreurs communes chez les programmeurs novices ? »

« Quelles sont les sources de leurs difficultés ?»

« En quoi l'environnement d'apprentissage peut-il niveler, vers le haut ou vers le bas, ces difficultés ? »

« Quels seraient les besoins, d'un point de vue didactique, pour supporter plus efficacement l'apprentissage ? »

1 Qu'est ce que programmer ?

Ces dernières années, l'intégration croissante de l'informatique, et en particulier des bases de données et de la programmation, dans les disciplines scientifiques, a fait émerger des disciplines telles que la bioinformatique, ou encore la « informatique physique » - *computational physics.* Cela pose avec acuité le problème de l'apprentissage de l'informatique dans une optique réellement pluri-disciplinaire (et non plus seulement pour former des ingénieurs analyste-programmeur). En effet, l'acquisition des compétences requises pour utiliser l'ordinateur comme outil d'analyse scientifique, et, plus encore, pour la conception de programmes, est reconnue comme étant très difficile ; ainsi Kaäsboll (Kaasboll 2002) rapporte-t-il que, de par le monde, entre 25 et 80 % des étudiants échouent ou abandonnent en cours d'initiation à la programmation en université. Cette statistique illustre parfaitement la difficulté d'accès de cette matière pour des débutants complets.

Dans ce premier chapitre, nous explicitons les sources des difficultés des étudiants. Pour cela, nous dégageons tout d'abord une définition de la programmation en faisant la synthèse de différents points de vue :

- Epistémologique : nous nous intéressons à l'évolution des définitions sur la programmation au cours du temps.
- Structurel : nous décomposons fonctionnellement la programmation en tant que tâche.
- Cognitif : nous nous intéressons aux processus mentaux mis en œuvre par le programmeur.

Cette définition synthétique de la programmation nous permet, par la suite, de dégager un certain nombre de difficultés intrinsèques à cette tâche.

1.1 « Programmer ? » - une approche épistémologique.

Dans les années 50, à l'apparition des premiers langages de programmation (tels FORTRAN - mathematical FORmula TRANslating system- et LISP -LISt Processing language- en 1958, qui sont encore utilisés de nos jours dans des versions plus évoluées), les ordinateurs étaient dédiés au calcul, dans un contexte scientifique. Les praticiens d'alors proposaient des définitions claires de la programmation. Ainsi, selon Hartree en 1950 (Hartree 1950): *« Le processus de préparation d'un calcul pour une machine se divise en deux parties, programmer et coder. La programmation est le processus d'élaboration de la séquence d'opérations requise pour réaliser le calcul »*; ou encore Wilkes en 1956 (Wilkes 1956): *« La séquence d'ordres est connue sous le nom de programme, et la machine l'exécute automatiquement sans intervention de l'utilisateur »*.

Alors que les langages de programmations précités remplaçaient peu à peu l'assembleur voire le langage binaire, le codage cessa d'être envisagé comme une activité à part entière, et l'activité de programmation fut dès lors davantage rattachée au registre de langue de la communication, allant jusqu'à se substituer pratiquement à la notion antérieure de codage chez Booth (Booth 1992): *« Le processus d'organisation d'un calcul peut se subdiviser en deux : la formulation mathématique et la véritable programmation […] traduisant (celle-ci)*

[…] dans le langage de l'ordinateur ». Ainsi, à la fin de la décennie, les notions de « programme » en tant que séquence exécutée automatiquement, de « programmation » en tant que spécification mathématique, et de traduction linguistique entre les deux étaient communément établies, comme dans la définition de Wrubel (Wrubel 1959): *« Cette séquence est appelée programme et son processus de préparation est appelée programmer »*.

Cependant, avec les années, le champ d'application de l'informatique grandit, dépassant les calculs mathématiques auxquels elle était originellement cantonnée. De même, la nature des opérations de base, les symboles des langages, et les formulations des programmes ont évolué, devenant plus génériques. De par cette généricité, l'écart entre ces opérations et leur symbolique et l'univers des tâches automatisées par l'ordinateur s'est globalement accru, augmentant de façon conséquente la taille des programmes, et créant de nouveaux besoins en termes d'organisation et de modularité. Toute une nouvelle génération de langages vit ainsi le jour dans les années 70, tels Pascal (1970), C, Prolog et SmallTalk (1972), ou Basic (1975). Leurs évolutions (dans les années 90 la généralisation du paradigme 'Orienté Objet' introduit par SmallTalk, et des concepts de sous-typage et d'encapsulation apparus avec ADA en 1983 eut pour effet l'apparition de nouveaux langages les prenant en compte : ainsi Pascal engendra t-il Delphi et C, C++, par exemple) demeurent parmi les langages les plus répandus aujourd'hui encore chez les programmeurs professionnels.

Parallèlement, l'apparition ces dernières années d'« applications » génériques de conception (au sens large : traitements de texte, tableurs, logiciels d'architecture, de dessin technique, …) destinées à un public non-informaticien a induit l'émergence de tâches de nature programmatique dans des domaines étrangers aux programmeurs de métiers. Ce qui amena à définir le champ de recherche de la « end-user programming », destiné à fournir à ces utilisateurs des outils à même d'accomplir ces tâches, sans pour autant exiger d'eux de fortes connaissances en informatique. Mais, alors que le terme « end user » est relativement parlant, l'apparition d'outils de programmation destinés a des non-informaticiens, qui diffèrent des langages utilisés par ceux-ci, amène à se reposer la question de la sémantique de la « programmation » dans ce cas précis. D'autant que, parallèlement à l'apparition d'une programmation sans langage, on peut trouver des activités dont le support est un langage informatique, mais que nombre d'informaticiens professionnels hésiteraient à qualifier de « programmation », comme la conception d'un site Internet en HTML, ou d'un rapport en LaTEX. Il apparaît donc que l'usage d'un langage informatique ne saurait constituer le seul critère de définition pour l'activité du programmeur.

Enfin, l'augmentation conséquente de la taille des applications informatiques eut pour effet de faire croître le nombre de programmeurs associés à un projet, mettant en exergue une dimension sociale dans l'activité du programmeur. Cet état de fait, et le développement rapide de l'ingénierie de l'informatique, entraîna l'apparition d'un nouveau champ de recherche en psychologie, la « psychologie de la programmation » (Hoc 1990) (Mayer 1988) (Soloway 1988) (Soloway 1989), s'intéressant aux processus cognitifs mis en œuvre par les programmeurs, et à la dimension sociale de leur travail. Des questions telles que : « *Que pouvons nous présumer de [leurs] compétences, [leurs] pratiques de travail, [leur] background, ou des échanges d'informations entre eux ?*» sont caractéristiques des objectifs de recherche dans cette discipline.

En ce qui concerne la définition de l'activité du programmeur, ces praticiens se distinguent des praticiens de l'informatique, en ce qu'il importe moins pour eux de savoir, quand les usagers disent qu'ils « programment » leur site web, leur chauffage, ou leur magnétoscope… s'il s'agit bien de programmation « véritable », que de comprendre quelles sont les

caractéristiques cognitives de ces activités qui amènent Mr. Tout-Le-Monde à les qualifier de programmation. Cette définition plus large et plus générique de l'activité de programmation, qui s'appuie sur des caractéristiques cognitives communes aux différentes activités de programmation, est conduite jusqu'à son terme par Blackwell dans (Blackwell 2002), où sont distinguées deux caractéristiques essentielles à la programmation :

- **La perte de la manipulation directe.** En effet, toutes les tâches de programmation ont en commun de spécifier un comportement *futur* de la machine (y compris si celle-ci est un magnétoscope, une interface de chauffage, ou de machine à laver…). Cette absence d'écho immédiat se révèle un élément clé et intrinsèque dans la difficulté à programmer.

- **L'usage de notations pour supporter l'abstraction** (d'ailleurs pas obligatoirement textuelles) à travers lesquelles le programmeur exprime le comportement futur de la machine. Cet usage de notations se justifie par la possibilité qu'il offre de décrire de façon abstraite une multitude d'exécutions différentes. Plusieurs niveaux de notation peuvent permettre de gérer la complexité en échelonnant la description du programme (approche « top-down »).

Et est dès lors considérée comme de la « programmation » toute activité possédant ces deux caractéristiques. Remarquons cependant que plusieurs éléments importants dans la programmation « classique » (c'est-à-dire avec les langages évoqués précédemment : C, Pascal, Ada, Delphi, C++ … etc.) sont passés sous silence.

D'une part, le coté « traduction » – qui provient de la nature de l'exécutant : capacités limitées, connues à l'avance et surtout indépendantes de la tâche à programmer ! – n'y est pas manifeste. Du coup, la difficulté « instrumentale » n'y apparaît pas clairement, alors qu'elle est très importante dans l'apprentissage de la programmation. En effet, le « problème » posé dans les exercices d'initiation à la programmation n'est pas tant lié aux tâches à accomplir (calculer la moyenne d'une liste de nombres, trouver le plus petit (ou grand) élément, trier par ordre croissant (décroissant) … etc.) qu'à la relative inadéquation des outils dont l'exécutant dispose. Pour prendre un exemple hors cadre informatique, pour la tâche « allumer un feu », il est de prime abord difficile de croire qu'elle puisse se révéler un « problème » requerrant l'application d'un savoir faire spécifique. Mais tout change en admettant qu'on dispose non pas d'un briquet ou d'une boîte d'allumettes, mais d'une paire de silex…

D'autre part, il n'est pas assez souligné dans cette définition que les « programmes », indépendamment de l'usage qu'ils font de notations abstraites, sont composés d'« instructions ». C'est-à-dire que l'écriture de programmes requiert non pas un « savoir-faire » mais un « savoir faire faire » qui nécessite de la part de l'étudiant une prise de recul par rapport à ses connaissances sur la tâche.

Dans un contexte non-informatique, une personne peut très bien être compétente pour ce qui est de cuisiner un plat, et pas pour ce qui est de composer la recette de ce plat. Ou encore, dans le registre musical, il est reconnu qu'être un bon interprète ne saurait être une condition suffisante pour devenir compositeur. Il existe donc bien une difficulté spécifique à ce « faire faire » qui ne réside pas uniquement dans le fait qu'il soit « différé », et que la caractéristique de « perte de la manipulation directe » n'induit pas totalement.

1.2 « Programmer » : caractéristiques et définitions.

Nous proposons donc une synthèse de ces différents points de vues, en reprenant en les raffinant et en les connectant aux différentes sémantiques précitées, les définitions du didacticien Charles Duchâteau dans (Duchâteau 1992). D'après Duchâteau, l'activité du programmeur consiste à « faire faire une tâche par un exécutant – l'ordinateur - ». Pour cela, il va concevoir une marche à suivre à l'intention de l'exécutant-ordinateur, appelée « algorithme », qui pilotera la réalisation ultérieure de la tâche par ce dernier (figure 1).

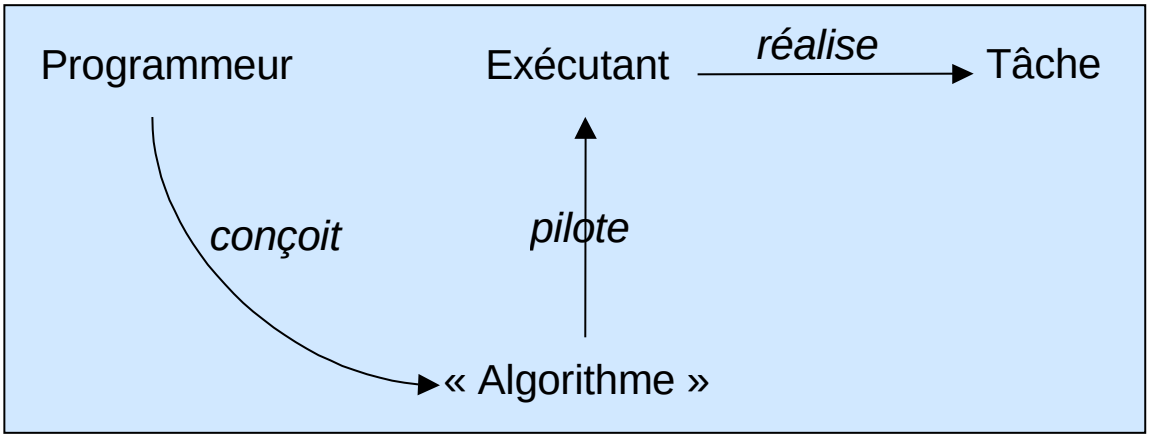


Figure 1 – Qu'est ce que programmer ? (Duchâteau 1992)

Attachons-nous à replacer à présent les caractéristiques de la programmation, que nous avons relevées dans la section précédente, dans ce schéma. Tout d'abord, et contrairement aux premières définitions historiques, l'emploi du terme « tâche » n'est pas rattaché à un domaine particulier, et traduit donc la diversité des applications des programmes informatiques. Ensuite, il met en exergue le problème du « faire faire », de par la réalisation de la tâche par un exécutant, piloté non par le programmeur lui-même mais par l'algorithme conçu par ce dernier. On retrouve donc à ce niveau la perte de la manipulation directe décrite par Blackwell, le programmeur n'ayant pas de prise directe sur la réalisation effective de la tâche. La difficulté de la conception de cet algorithme est bien sûr conditionnée par la nature de l'exécutant, et des instructions qu'il est apte à réaliser. En marge du schéma, nous pouvons ajouter que l'algorithme conçu a vocation en programmation « traditionnelle » à piloter un grand nombre de réalisations de la tâche, en utilisant les notations et outils d'abstraction des langages informatiques pour parvenir à cette généralisation.

Récapitulons les quatre caractéristiques essentielles à la programmation, exhibées par cette définition ; « programmer », c'est :

- **« Faire faire …»** : un programme, à la base, se compose d'*instructions* ; la programmation se caractérise donc par une prise de recul de la part du programmeur, qui doit décrire l'accomplissement de la tâche (que lui-même réalise le plus souvent de façon quasi inconsciente – réfléchira-t-on soi-même profondément avant de s'attaquer au « problème complexe » que constitue le tri d'une main au tarot ?). Il doit passer du domaine du savoir-faire inconscient à une représentation consciente de toutes les actions requises pour réaliser la tâche.

- « **… en différé …** », un programme est une *planification*, qui résultera en une réalisation *ultérieure* de la tâche, aussi le programmeur doit-il recourir à une représentation mentale de l'état de la tâche pour compenser la « perte de la manipulation directe ».

- « **… à un exécutant aux capacités limitées…** ». Lorsqu'on programme, on s'adresse à un exécutant parfaitement spécifié, et capable uniquement d'opérations formelles, et non sémantiques. L'exécutant magnétoscope est ainsi incapable de reconnaître « le début du grand prix de formule 1 » (ou la fin) ; il ne manipule que des données formelles, telles que des heures, des numéros de chaînes, des codes « showview » … La possible (et fréquente…) inadéquation entre les instruments dont dispose l'exécutant et les savoir-faire du programmeur concernant la tâche sera comme nous le verrons dans le chapitre suivant source de nombreuses difficultés…

- « … **en utilisant un formalisme donné** ». Pour généraliser le comportement de l'exécutant sur plusieurs réalisations, le programmeur fait usage de notations (textuelles, graphiques, ou autres) exprimant les capacités de celui-ci.

Dans le cadre, plus restreint, de la programmation structurée traditionnelle, il est possible de raffiner cette définition pour obtenir une décomposition détaillée de l'activité du programmeur (Guibert 2005) (figure 2).

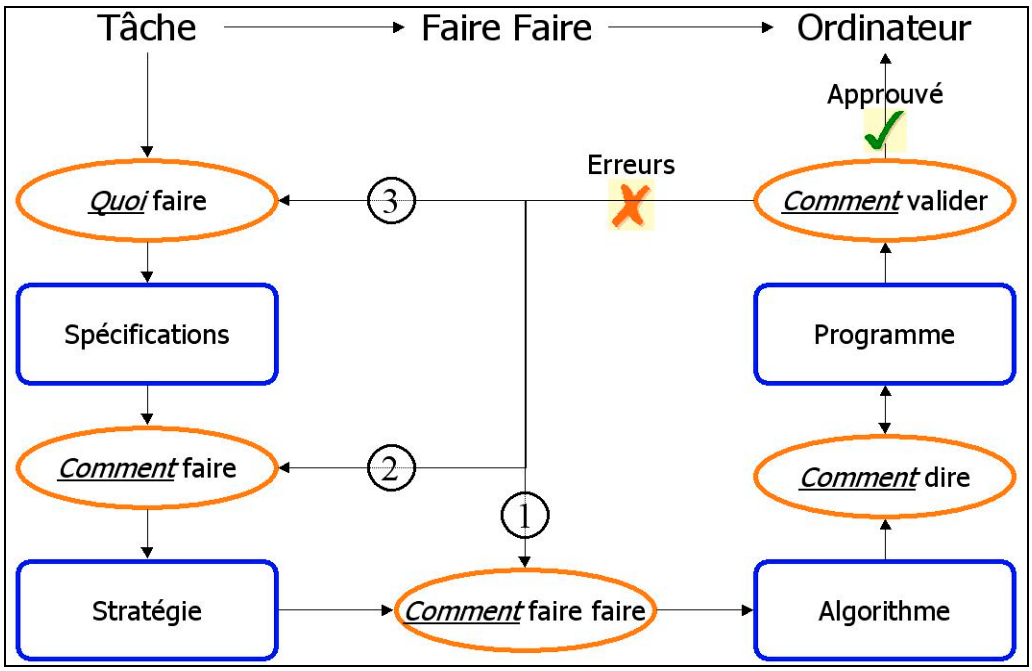


Figure 2 – Décomposition détaillée de l'activité du programmeur (Guibert 2005), inspiré de (Duchâteau 2000).

Dans le but de faire faire une tâche par l'ordinateur, le programmeur passera donc chronologiquement par les étapes suivantes :

- **Quoi Faire ?** La première étape de l'activité du programmeur consistera le plus souvent en l'analyse de la tâche que la machine aura à traiter, activité débouchant sur une spécification détaillée du comportement de celle-ci. Cet aspect prend une importance critique dans un cadre professionnel, où les informaticiens doivent, face à un interlocuteur (le client) qui dispose du savoir-faire (généralement en partie inconscient) sur la tâche considérée, poser les questions pertinentes pour l'automatisation de cette tâche, en fonction de leurs connaissances sur les capacités de la machine. Cette étape, pour intéressante et délicate qu'elle soit n'est généralement pas (ou peu) abordée dans les cours d'initiation, car les tâches considérées sont souvent triviales (mais longues, inintéressantes etc. - d'où l'intérêt de les faire réaliser par un programme) et parfaitement maîtrisées par l'apprenti programmeur. Nous ne nous étendrons donc pas plus avant sur les spécificités et difficultés de cette étape par la suite.

- **Comment Faire ?** L'étape suivante, une fois établies les spécifications précises de la tâche, consiste à prendre du recul par rapport à la réalisation de celle-ci, de façon à dégager une stratégie, permettant d'abstraire tous les comportements de la machine lors de l'exécution. Par exemple, dans la tâche illustrée par la figure 3, qui consiste à réaliser un report de notes, depuis une pile de copies notées, les copies n'étant pas ordonnées, (en bas à gauche Figure 3) vers une liste ordonnée (en haut), deux stratégies viennent naturellement à l'esprit :

  - Parcourir la liste ligne par ligne, et à chaque fois extraire la copie correspondant au nom de la ligne courante, recopier la note puis passer à l'élève suivant.

  - Parcourir la pile de copies, et à chaque fois aller à la ligne correspondant au nom de l'élève sur la liste, et y recopier la note.

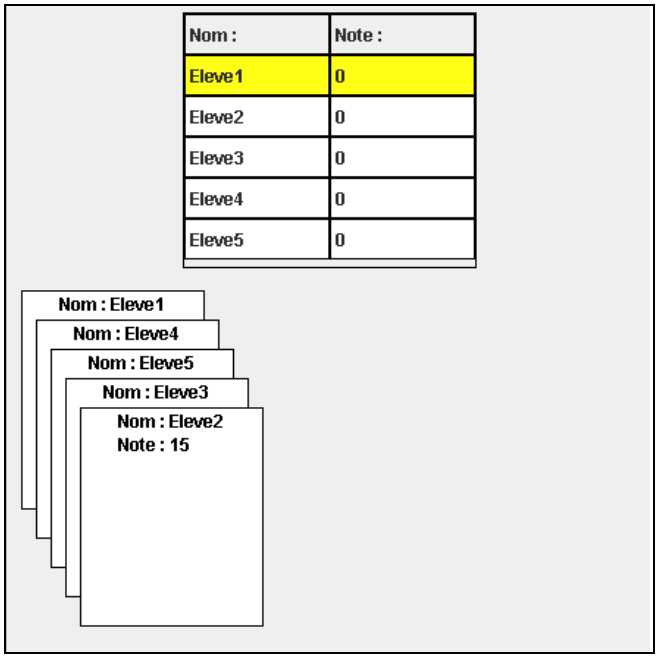


Figure 3 – Tâche de report de notes.

Les mots clés à attacher à cette phase seront donc « exhaustif » – puisqu'il s'agit de couvrir TOUS les comportements que l'on attend du programme – et « explicite » - la difficulté intrinsèque consistant bien à « mettre à plat » son savoir-faire, à passer du domaine de l'inconscient à celui du conscient ... dans les « problèmes » du tri d'une main de tarot, ou de la recherche du plus petit élément d'une série d'entiers, ce qui est en question ce n'est pas du « savoir-faire » - l'enseignant informaticien sait ses élèves capables de réaliser ces tâches – mais du savoir « comment faire », et plus la tâche est anodine et exécutée de façon automatique, plus l'exercice peut être difficile, car la question « comment fais-tu ? » a alors quelque chose d'incongru, voire d'absurde.

- **Comment Faire Faire ?** Une fois qu'il dispose d'une stratégie explicite de réalisation de la tâche, le programmeur doit s'attaquer à la tâche consistant à prendre en compte les capacités de l'exécutant, d'adopter son « point de vue ». Cette étape lui impose de structurer à l'aide d'opérateurs particuliers (itération, appel, alternative) sa stratégie, et de décrire les objets d'intérêt de la tâche sous une forme compréhensible pour l'exécutant. Il lui faut à ce stade franchir la barrière cognitive entre sa représentation de l'univers de la tâche, et celle, complètement formelle et le plus souvent numérique, employée par l'ordinateur (figure 4).

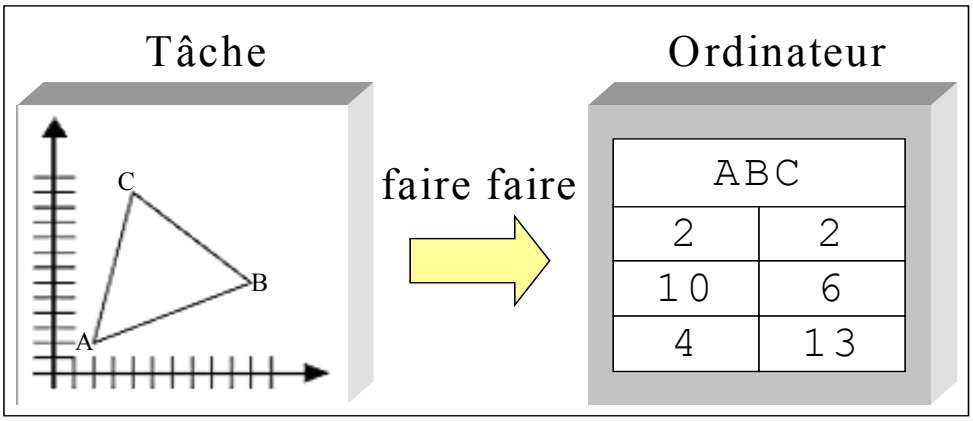


Figure 4 – Le fossé cognitif entre le modèle mental de l'apprenant et le modèle de l'ordinateur.

- **Comment Dire ?** Cette description doit ensuite être traduite dans un langage de programmation pour être compréhensible pour l'ordinateur. Cette étape se caractérise par la prééminence de la syntaxe sur la sémantique, et il convient de s'y exprimer de façon rigoureuse, tant grammaticalement qu'orthographiquement. La difficulté est que le langage de programmation peut comporter de nombreux éléments syntaxiques générateurs d'erreurs ou de confusion (nous détaillerons cet aspect dans le chapitre 3) et peut ne pas être l'expression exacte du modèle logique qu'il doit présenter à l'utilisateur. Ces différences entre le modèle logique du système et le modèle qui transparaît à travers le langage risquent d'être amplifiées par l'interprétation que le programmeur fait de celui-ci, causant des erreurs de conception.

- **Comment Valider ?** Après avoir apporté de (nombreuses) modifications au programme pour le rendre syntaxiquement valide, le programmeur doit alors le faire exécuter pour en vérifier la sémantique. L'expérience montre qu'à la fin de cette séquence, il y a presque toujours des erreurs. Ce retour d'erreur (tardif) de la phase de conception se fait le plus souvent dans l'ordre inverse à la conception, comme décrit sur la figure 2 : ce sont les erreurs algorithmiques qui sont généralement détectées en premier, après quoi l'on s'aperçoit d'erreurs de stratégie (modélisation de la tâche non exhaustive, ou stratégie incorrecte) et finalement on peut avoir la mauvaise surprise de ne pas avoir réalisé l'outil attendu par le client.

Bien sûr, il faut alors reprendre le développement au point où l'erreur a été localisée et il n'est pas rare que cette mise au point génère de nouvelles erreurs en aval. De fait, re-recherche de l'erreur, re-édition, re-compilation, re-tests… nouvelles erreurs. Ce cercle vicieux – Figure 2 – dévore la majorité du temps alloué au développement. Et la plupart du temps, les tests ne permettent pas de certifier qu'il ne reste aucun bogue, pour peu que le programme soit un tant soit peu volumineux. Le nombre de « patchs » et autres « service packs » dans les logiciels commerciaux en apporte une preuve évidente. Et ce qui est difficile pour des programmeurs de métier l'est plus encore pour des débutants.

2 Etude des difficultés de l'apprentissage de la programmation

Attachons-nous à présent à répertorier les difficultés rencontrées par les débutants, à partir d'un état de l'art des travaux en psychologie et en didactique de la programmation. Nous tâchons d'abord d'en établir les causes, en décrivant quels sont les mécanismes de construction du savoir, quels pré-requis ils imposent à l'enseignement de l'informatique, et quels obstacles sociaux ou technologiques fragilisent la base de cet enseignement. Puis, nous décrivons les mécanismes qui, à cause de ce savoir de base fragilisé, conduisent les étudiants à l'erreur. Nous illustrons ces mécanismes par la revue d'erreurs communes dans la littérature. Ces erreurs seront également reliées à la phase qu'elles affectent dans le cycle de conception. Enfin, nous décrirons les comportements que ces mécanismes induisent sur les étudiants ainsi placés en situation d'échec lors de l'apprentissage de la programmation.

2.1 Introduction aux mécanismes de construction du savoir

2.1.1 Théorie des Schémas

De nos jours, la théorie dominante sur la construction du savoir par compréhension[1], connue sous le nom de « *théorie des schémas* » est que ce processus a une nature récursive : les connaissances préexistantes sont fortement sollicitées pour la production d'un nouveau savoir. Les connaissances préexistantes en mémoire sont structurées en « *schémas* » et le processus permettant de connecter le nouveau savoir aux anciens y est appelé « *assimilation* ». Dans cette théorie, le système cognitif humain est composé de :

- **La mémoire à court terme,** à capacité limitée, permettant de stocker de façon temporaire pour la manipuler l'information en provenance du système sensoriel. Elle est aussi connue sous le nom de « **mémoire de travail** ».
- **La mémoire à long terme,** dont la capacité est virtuellement illimitée, qui a pour vocation de stocker le savoir existant de façon permanente, et organisée.

Comme le montre la Figure 5, les nouvelles informations venant du système sensoriel pénètrent dans le système cognitif par la mémoire à court terme, et doivent passer par les étapes suivantes pour parvenir à la compréhension :

- **La Réception** de l'information, qui transite alors dans la mémoire à court terme de l'apprenant.
- **La Recherche** dans la mémoire à long terme de schémas préexistants permettant d'assimiler l'information.
- **L'Activation.** L'apprenant doit utiliser activement ce savoir préalable pour y connecter les nouvelles informations. Ce processus est appelé **assimilation**. Puis, ses nouvelles connaissances lui permettent de créer une nouvelle information qui transite vers la mémoire à court terme.

[1] La *compréhension* se caractérise par une capacité de transfert des connaissances nouvellement acquises (comprendre = prendre avec soi), et s'oppose ainsi à l'apprentissage « par cœur », où le savoir acquis peut être facilement reproduit, mais reste fortement ancré au contexte d'acquisition : il est donc significativement plus difficile à utiliser et à transférer.

- **L'Action :** cette information donne lieu à une réponse qui transite vers le système moteur, pour expérimenter les nouvelles connaissances.

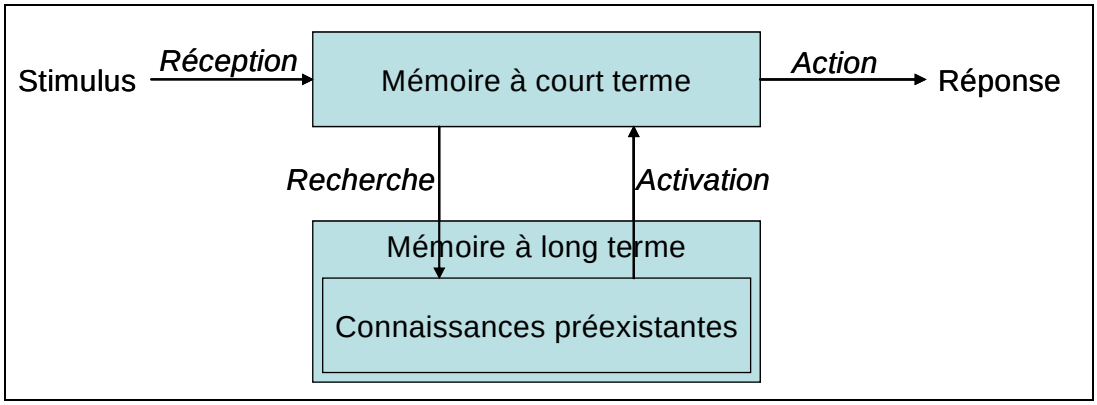


Figure 5 – Le mécanisme de compréhension de nouvelles connaissances.

Chacune de ces étapes doit se produire si l'on veut que le nouveau savoir soit effectivement *compris.* Sans quoi, les liaisons avec les autres concepts normalement connexes ne se mettent pas en place (ce processus est appelé « ideational scaffolding » - échafaudage d'idées). L'information serait simplement stockée dans la mémoire à long terme, et en l'absence de pointeurs vers d'autres schémas, serait difficilement accessible dans un autre contexte.

Les études en psychologie de la programmation ont dégagé plusieurs types de connaissances en mémoire chez un expert en programmation (Détienne 1998). On distingue ainsi :

- Des connaissances **syntaxiques**, qui regroupent les définitions des éléments lexicaux et syntaxiques d'un langage de programmation.
- Des connaissances **sémantiques** se référant aux concepts (ex. la *notion* de « variable », d' « appel », d'« objet », d'« instanciation », d'« héritage » …)
- Des connaissances **schématiques** qui utilisent les précédentes pour définir des structures génériques de solution.

On peut ranger ces dernières selon quatre catégories :

- Des **schémas élémentaires de programmation** qui représentent des connaissances sur la structure de contrôle et les variables (« control plan » - « variable plan ») (cf. *Comment Faire Faire ?*)
- Des **schémas algorithmiques** (ou schémas complexes de programmation) ; ceux-ci représentent des connaissances sur des structures d'algorithmes – ex : schémas de recherche, de tri …, et peuvent se composer de plusieurs schémas élémentaires. (cf. *Comment Faire Faire ?*)
- Des **schémas du problème**, qui structurent les connaissances que l'informaticien possède sur certains types de tâches, ou sur certains domaines d'application (ex. bio-informatique, infographie …) (cf. *Comment Faire ?*)
- Des **schémas d'implémentation** qui représentent l'image des concepts précédents dans la syntaxe d'un langage particulier. (cf. *Comment Dire ?*)

Ces schémas sont reliés entre eux par des connexions qui peuvent être de trois types :

- Des liens de composition ; (par exemple le schéma d'un algorithme spécifique peut se décrire comme la composition de plusieurs structures de contrôles représentées par un schéma élémentaire)
- Des liens de spécification (par exemple un schéma de tri par tas spécialise un schéma de tri)
- Des liens de mise en œuvre.

2.1.2 Théorie de l'apprentissage expérimental

Une vision plus globale du processus d'apprentissage chez l'adulte est proposée par Kolb (Kolb 1986). Ce modèle, le processus d'**apprentissage expérimental,** y est modélisé par un cycle découpé en 4 phases (figure 8) :

- Une phase d'**expérience concrète,**
- qui donne lieu a une phase d'**observation** et de réflexion (étude de traces, brainstorming);
- ce qui amène l'apprenant à une phase de **conceptualisation**, qui a pour but de **comprendre** l'information observée, via le processus d'**assimilation** décrit précédemment.
- Après quoi, la théorie assimilée servira de guide dans la **planification** et la création d'expériences qui pourront confirmer, ou modifier le modèle théorique (phase d'**expérimentation :** simulations, études de cas, travail sur des applications pratiques).

Dans le cadre d'un apprentissage « classique » de la programmation (qu'il soit autodidacte ou académique), les nouveaux concepts sont issus de manuels ou d'un cours magistral, la planification correspond à l'écriture ou au déboguage du programme (placement des sorties « print » ou des points d'arrêts), l'expérience concrète est la compilation ou l'exécution du programme, et l'observation correspond à l'analyse des retours du système (qu'il s'agisse d'un résultat ou de messages d'erreurs).

Ce modèle peut être vu comme un raffinement du précédent, dans la mesure où il précise quand et comment sont acquises les nouvelles connaissances, qu'elles soient de nature déclarative ou procédurale. Ainsi, les connaissances de nature sémantique et syntaxique sont acquises pendant la phase de conceptualisation, tandis que les connaissances de nature plus schématique sont acquises pendant les phases d'observation ou de planification des nouvelles expériences, selon les cas.

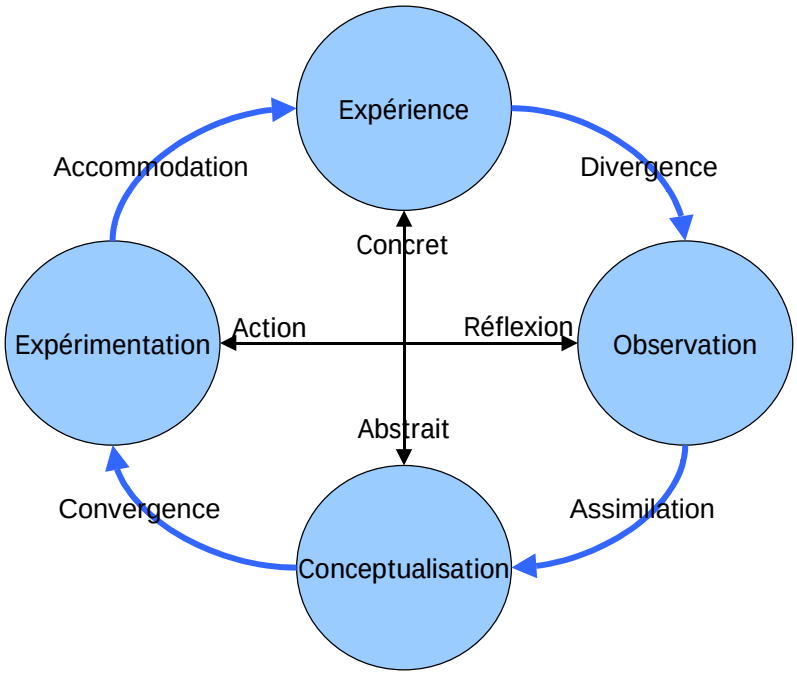


Figure 6 – Le cycle de l'apprentissage expérimental chez l'adulte (Kolb 1986).

L'apprentissage s'appuie donc sur ces quatre modes particuliers. L'apprenant doit être capable de s'investir complètement et sans préjugés dans de nouvelles expériences, de conduire une réflexion sur celles-ci, à travers une phase d'observation, de créer des concepts qui intègrent ces observations dans une théorie consistante, et de se servir de ces concepts pour prendre des décisions et résoudre des problèmes.

Cependant, dans les faits, peu de personnes sont réellement performantes dans tous les modes; la plupart tendent à se focaliser sur une ou deux activités (généralement connexes). Cela amène Kolb à définir quatre styles d'apprentissages, qui dépendent des modes les plus adaptés à l'apprenant :

- La **Convergence.** Ce style d'apprentissage se caractérise par des capacités plus importantes en conceptualisation et en planification d'expériences. La force des apprenants ayant ce style réside dans l'application pratique de concepts théoriques. C'est un style commun chez les ingénieurs, et d'après plusieurs études (dont (Byrne 2001)) le style le plus largement représenté chez les étudiants en informatique (*id est* les étudiants voulant faire de l'informatique leur métier).

- La **Divergence.** En opposition, ce style d'apprentissage correspond à des étudiants dont les qualités premières sont l'imagination, et la capacité à observer une situation concrète depuis plusieurs points de vue. Ce genre d'apprenant excelle dans des situations de brainstorming.

- L'**Assimilation.** Ce style d'apprentissage amène à privilégier une approche logique et concise. Ces étudiants ont besoin d'explications claires, logiques plutôt que d'applications concrètes ; leur apprentissage est focalisé plus sur les idées et la théorie que sur la technique, les applications pratiques, ou la coopération. La lecture d'ouvrage théorique, les cours magistraux ou l'exploration de modèles analytiques leur sont particulièrement adaptés.

- L'**Accommodation.** Ce style d'apprentissage correspond à des personnes qui s'appuient plus sur l'intuition que sur la logique. Le terme « accommodation » a été choisi car ces apprenants excellent souvent dans des situations où il faut s'adapter aux circonstances présentes : dans des situations où le plan ou la théorie viendrait à aller à l'encontre des faits, ils tendront à privilégier les faits et à écarter la théorie. Ils résolvent les problèmes de façon intuitive, par une approche « essai - erreur ».

En pratique, les quatre pôles du cycle d'apprentissage sont antagonistes. D'une part, les apprenants ont un savoir qui est relatif soit à leur expérience concrète (par **appréhension**), soit à des connaissances théoriques (par **compréhension**). D'autre part, le cycle de construction du savoir par voie expérimentale permet de distinguer deux approches opposées permettant la connexion et l'échafaudage des idées :

- L'expérimentation est utilisée comme source de compréhension : l'apprenant transforme ses connaissances théoriques – acquises en premier de façon livresque, ou via un cours magistral – en les soumettant à des tests pratiques : c'est une approche par **dénotation**[1] : on va de la *compréhension* à l'*appréhension*.

- Par **connotation**[2], au contraire, l'étudiant part d'une série d'observations objectives, pour construire des connaissances théoriques cohérentes via la formulation d'hypothèses. Il part donc de l'*appréhension* pour y connecter, à travers sa réflexion sur les observables, la *compréhension*.

On notera que les approches pédagogiques des cours à l'université (plutôt *convergentes*) comme des livres sur un langage particulier (plutôt *accommodatrices*) sont presque exclusivement dénotatives. Elles s'opposent en revanche au niveau du savoir : en *compréhension* pour l'approche académique, et en *appréhension* pour l'approche des manuels des langages.

De façon évidente, les besoins – en termes de retours d'informations du système - de ces différents types d'apprenants ne sont pas identiques. Pour s'adresser au plus grand nombre, un environnement d'apprentissage de la programmation se devra donc de supporter un mode de fonctionnement qui convienne à chacun.

---

[1] Dénoter = « *désigner par une caractéristique* » [] ≅ signifier (être le signe de). Cette approche est dite dénotative car l'étudiant recherche dans son expérience le « signe », la manifestation de la connaissance théorique préalablement acquise.

[2] Connotation = « *valeur que prend une action* (ou une information) *en plus de sa signification initiale* »

2.2 Implications sur les difficultés d'apprentissage de la programmation.

A la lumière de ces définitions, Ben-Ari (Ben-Ari 1998) a mis en exergue ce qui caractérise l'enseignement de l'informatique (tout du moins à des débutants complets) :

- **L'absence de modèle initial de l'ordinateur** chez les étudiants novices. L'ordinateur leur apparaît comme une « boite noire » qui reçoit les informations saisies au clavier et à la souris, et fabrique des retours écrans, le traitement entre les deux demeurant assez mystérieux…

- L'ordinateur, en programmation, forme une **« réalité ontologique[1] accessible ».** Tester la correction d'une conception y est facile, et, surtout, un *modèle correct* du système y est absolument nécessaire : il n'y a, à terme, pas d'« à peu près », et les conséquences de concepts erronés apparaissent rapidement (sous la forme d'erreurs à la compilation, ou encore de bogues à l'exécution).

Ces caractéristiques transparaissent à travers de nombreuses études empiriques. Ainsi, Du Boulay (Du Boulay 1989) remarque-t-il que « *Même si aucun effort n'est fait (de la part de l'enseignant) pour représenter ce qui se passe « à l'intérieur », les apprenants formeront la leur* »; Perkins (Perkins 1986) « *…attribue la fragilité des connaissances des étudiants en programmation en grande partie à l'absence d'un modèle mental de l'ordinateur.* » et Sleeman (Sleeman 1988) affirme-t-il que « *…même après un semestre complet de Pascal, la connaissance des étudiants de la machine conceptuelle derrière Pascal peut être très floue.* ». La création par l'étudiant de modèles incohérents du comportement de l'ordinateur est également mise en exergue par Spohrer (Spohrer 1985; Spohrer 1986; Spohrer 1986).

Elles expliquent de nombreux phénomènes observés par les enseignants en informatique :

- Les concepts et schémas de programmation construits par les élèves sont flous, fragiles ou incohérents car les données sensorielles provenant de l'enseignant ou du retour (feedback) du système doivent être intégrés dans un cadre de connaissances qui est trop superficiel.

- La frustration des étudiants et l'impression que l'informatique est difficile provient du fait que les modèles doivent être construits par l'étudiant lui-même, en partant de zéro.

- L'expérience des étudiants ayant pratiqué au préalable en autodidactes la programmation ne se corrèle pas avec un succès en étude universitaire, car ces étudiants arrivent aux cours avec un modèle de connaissance bâti par accommodation qui se révèle incohérent avec les modèles conceptuels enseignés; or l'évaluation « académique » de la programmation est centrée sur la *compréhension* et non pas sur la capacité à *réaliser* un programme via un cycle d'essais et d'erreurs. Leur *expérience*

[1] Onto – logie = théorie de l'existence, en métaphysique. Une *réalité ontologique* signifie que toutes les lois et tous les mécanismes de l'ordinateur sont connus, qu'il existe un « vrai » modèle de l'ordinateur - dont les concepteurs de programme peuvent avoir une image incomplète. En informatique, une ontologie est l' « *Ensemble d'informations dans lequel sont définis les concepts utilisés dans un langage donné et qui décrit les relations logiques qu'ils entretiennent entre eux.* » - [Office québécois de la langue française, 2002]

concrète est donc inutile, voire conflictuelle avec l'enseignement proposé, qui est focalisé sur les abstractions.

- Les retours d'erreurs du système (qui requiert une syntaxe et une sémantique *exacte*), parfois brutaux, peuvent décourager des étudiants au style d'apprentissage plus réflectif ou social (divergence et assimilation).

Malheureusement ces styles d'apprentissage sont relativement courant chez les étudiants en science fondamentale (physique et chimie) et science de la vie ou science humaine (psychologie) qui ont un besoin croissant de l'ordinateur, voire de la programmation, en tant qu'outil … Cette absence de savoir intuitif sur le fonctionnement interne de la machine est imputable à différentes spécificités techniques de l'ordinateur. De fait, il devient difficile à l'utilisateur soit de réaliser les actions de la tâche qu'il a en tête avec les outils disponibles sur la machine, soit (et c'est le plus fréquent) de comprendre le retour d'information que lui envoie l'ordinateur.

2.3 Conséquences sur les étudiants

Une fois confrontés à une situation d'échec, les apprenants peuvent adopter deux comportements tout à fait opposés : le bricolage – qui se caractérise par une succession de petites modifications de façon presque aléatoire jusqu'à trouver une solution dont on constate qu'elle marche, mais dont on serait bien embêté d'expliquer pourquoi – ou la passivité.

2.3.1 Bricolage

Initialement utilisé – de façon détournée – par l'anthropologiste Claude Levi-Strauss pour désigner le savoir technique « concret » dans les sociétés « primitives », par opposition à l'approche de la science et de l'ingénierie dans la civilisation occidentale – le terme « bricolage » a été transféré au domaine de l'apprentissage de la programmation par Turkle et Papert (Papert 1980). Il se caractérise par la prééminence d'expérimentations *concrètes* sur l'ordinateur pour obtenir une *appréhension* de son comportement (styles d'apprentissages accommodant ou divergent). Malheureusement, l'absence de compréhension du feedback de la machine peut amener les étudiants à modifier encore et encore (par cycles d'« édition-compilation-test ») le programme par petites touches sans comprendre ce qu'ils font, ce qui implique que « résoudre » le même problème ne sera pas plus simple ni plus court si celui-ci se représente. Il peut même leur arriver d'essayer la même solution plusieurs fois dans une même session, comme des personnes perdues qui tournent en rond sans s'en apercevoir, note Perkins (Perkins 1986)

2.3.2 Passivité

De façon diamétralement opposée, des étudiants peuvent en venir à la conclusion que la résolution du problème dépasse leurs capacités, la dissonance cognitive entre le feedback attendu et le feedback obtenu les plaçant dans une situation de blocage. Ils cèdent dès lors au découragement et demeurent passifs en attendant que l'enseignant les aident, voire en espérant qu'il résolve le problème à leur place. Si l'explication fournie se révèle hors de leur compréhension, cela peut développer encore plus leur découragement, et affaisser d'autant leur estime d'eux-mêmes, les amenant à moins demander d'aide car cela supposerait l'aveu de leur ignorance. Il en résulte alors non pas une discussion constructive avec l'encadrant mais

encore plus de passivité. McIver (Mc Iver 2001) appelle ce phénomène « *Learned Helplessness* » (« Impuissance apprise ») : les apprenants cessent d'essayer de nouvelles choses, de tester d'autres hypothèses, car ils sont par avance persuadés d'échouer (Norman 1990). C'est pourquoi ils cessent toute initiative, et se bloquent dès la première difficulté, ou dès que le système répond d'une façon qui ne correspond pas à leurs attentes. Les systèmes d'aides, les messages d'erreurs qui sont rédigés dans un jargon technique qui limite leur accessibilité, ou ne contiennent pas les réponses que cherchent les étudiants, favorisent ce phénomène.

2.3.3 Importance de ces phénomènes

Ces deux phénomènes tendent à avoir un impact considérable dans les cours d'initiation, comme le confirment les études de la littérature : ainsi, selon Garner (Garner 2005), le second[1] problème le plus rencontré par les étudiants est d'être « bloqué sur la conception », ce que Smith (Smith 1993) appelle le « syndrome de la page blanche ». Ce type de problème est également reporté par Carbone (Carbone 1998) et par Kaäsboll : « … *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 Ouille! 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 …* ».

Selon les résultats de Goold (Goold 2000) – voir table 1- et Wilson (Wilson 2000) les facteurs qui corrèlent le plus avec les résultats (respectivement négatifs et positifs) en initiation à la programmation sont le « *dégoût de la programmation* » et le « *niveau de confort* » ressentis pendant le cours. Ce sont les variables qui reflètent la motivation, le découragement ou l'estime d'eux-mêmes des apprenants.

	Concepts de base.	**Structure de données et Algorithmes.**	
	Total	**Examen**	**Total**
Variable	**Coefficients de régression**		
Score dans les autres modules	0,32	0,89	0,71
Expérience préalable de la programmation	-	-	12,32
Dégoût de la programmation	-12,50	-28,38	-22,40
Résolution de problèmes	1,36	-	-
Sexe (h|f)	7,52	-	-

Tableau 1 — Effets de différentes variables sur les résultats des modules d'initiation à la programmation, selon Goold et al. (Goold 2000).

[1] le premier étant l'accumulation d'erreurs de syntaxe « mécaniques » – oublis de '{', de ' ;' et fautes d'orthographe sur les mots clés … – apparemment hors de propos pour la compréhension de la sémantique du programme, mais pouvant faire chuter la motivation des étudiants de façon drastique

3 Conclusion et Perspective

Dans ce chapitre, nous nous sommes appuyés sur une riche bibliographie, dans le champ dit de la « Psychologie de la Programmation » (Psychology of Programming), et dans celui de la didactique de la programmation, pour explorer les mécanismes de construction des connaissances (notamment le système cognitif humain et les styles d'apprentissage), et les relier au contexte de la programmation.

Nous avons donc pu proposer une synthèse des différentes définitions de la programmation qui tient à la fois compte de la structure de cette activité et de l'aspect cognitif. On peut dire de façon générale que programmer c'est :

- **« Faire faire …»** : un programme, à la base, se compose d'*instructions* ; la programmation se caractérise donc par une prise de recul de la part du programmeur. Il doit passer du domaine du savoir faire inconscient à une représentation consciente de toutes les actions requises pour réaliser la tâche.

- « **… en différé …** », un programme est une *planification*, qui résultera en une réalisation *ultérieure* de la tâche, aussi le programmeur doit il recourir à une représentation mentale de l'état de la tâche pour compenser la « perte de la manipulation directe ».

- « **… à un exécutant aux capacités limitées…** ». Lorsqu'on programme, on s'adresse à un exécutant parfaitement spécifié, et capable uniquement d'opérations formelles, et non sémantiques. La possible (et fréquente…) inadéquation entre les instruments dont dispose l'exécutant et les savoir-faire du programmeur concernant la tâche est source de nombreuses difficultés…

- « … **en utilisant un formalisme donné** ». Pour généraliser le comportement de l'exécutant sur plusieurs réalisations, le programmeur fait usage de notations (textuelles, graphiques, ou autres) exprimant les capacités de celui-ci.

Nous avons catalogué et classé les principaux types d'erreurs et de difficultés, que ces dernières soient liées aux concepts manipulés ou à l'environnement au sens large. On constatera, à la lumière de ces difficultés et des recommandations des didacticiens de l'informatique, qu'une grande part des difficultés des novices sont liées à deux caractéristiques qui tiennent plus de l'environnement d'apprentissage que des concepts abordés :

- Dans le but de pouvoir s'appliquer à n'importe quel domaine, les langages de programmation proposent des outils d'un faible niveau sémantique, et imposent une représentation qui est totalement déconnectée du contexte et donc étrangère aux représentations que l'apprenant se fait à l'origine des objets du domaine.

- Les environnements de programmation ne maintiennent pas en permanence une représentation de l'état de la machine, et donc obligent l'apprenant à construire et animer, au prix d'une lourde charge cognitive, leur conception de celui-ci, qui peut être incohérente et non viable.

Par conséquent, pour faciliter la construction d'un modèle mental du système, et donc l'apprentissage de la programmation, le rôle des interactions avec les environnements de

programmation est central. Notons que les difficultés posées par les concepts et la syntaxe se révèlent être de nature différente. Les difficultés rencontrées avec la syntaxe ne posent pas un problème de fond, mais peuvent devenir « l'arbre qui cache la forêt ». A contrario, les difficultés liées à l'acquisition des concepts et des schémas conceptuels, et plus important encore, aux *relations* entre les différents schémas, sont de nature plus *sémantique*. Cette différence de nature explique la relative inadéquation des outils de développement professionnels au problème de l'apprentissage : s'adressant à un public expert, le support de difficultés conceptuelles y est tout simplement nul.

Norman (Norman 1990) exprime par des distances ces différences entre la représentation du langage et le modèle que s'en fait l'utilisateur. La distance sémantique (figure 7) a trait à la connaissance des objets manipulés et à la signification des commandes et des retours d'informations (franchie dans les phases d'intention et d'évaluation). La distance articulatoire, comme son nom l'indique, mesure la difficulté à faire le lien entre les grandeurs et observables du système et le modèle mental de l'utilisateur (**Spécification** et **Interprétation**).

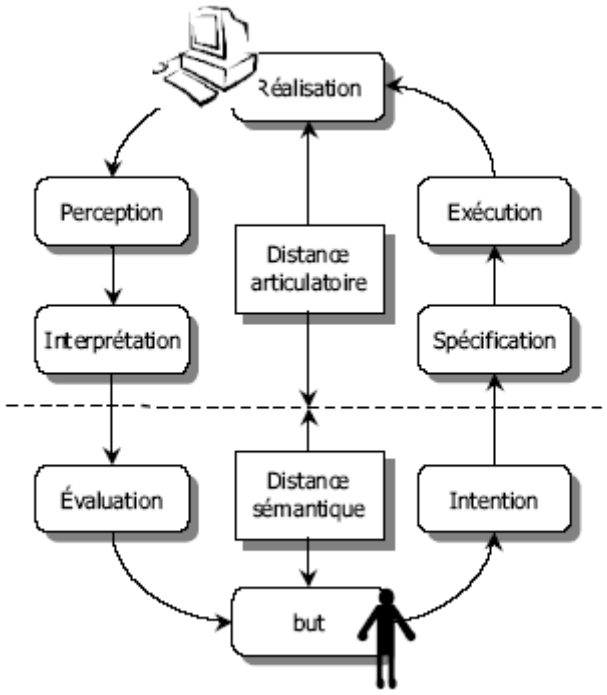


Figure 7 – Distances cognitives d'exécution et d'évaluation séparant l'ordinateur de l'utilisateur et la tâche qu'il souhaite accomplir, selon (Norman 1990).

Dans le cas de la programmation, la transition entre le modèle mental de l'apprenant et le modèle interne du système est hasardeuse, en raison des distances articulatoires qui les séparent :

- La plupart des langages de programmation posent des difficultés aux étudiants, de par leurs notations délibérément abstraites, et cherchant avant tout la concision. Cela rend l'interprétation de l'état du modèle à partir du système difficile. Ces notations peuvent paraître cryptiques aux étudiants, et représenter un obstacle à la ***spécification***.

- La plupart des paradigmes de programmation, pour vérifier la correction des concepts de l'étudiant, demandent d'accomplir un grand nombre d'actions (édition,

compilation, exécution) avant d'avoir le moindre écho du système. Le modèle de l'état du système n'est donc pas constamment présenté à l'étudiant (ce que (Blackwell 2002) appelle la « perte de manipulation directe »). D'autre part, la phase de vérification syntaxique est souvent parsemée d'erreurs qui n'ont rien à voir avoir avec la sémantique du programme. Cela a pour effet d'augmenter considérablement la distance articulatoire correspondant à l'***interprétation*** des retours du système.

Une difficulté majeure de la programmation est particulièrement liée à la « perte de la manipulation directe qui impose à l'apprenant de construire et d'animer, au prix d'une lourde charge cognitive, sa propre conception de celui-ci, qui peut être incohérente et non viable. Il nous paraît donc intéressant de tester la capacité d'une représentation continue de l'état de l'ordinateur à supporter l'acquisition des liens de composition et de mise en œuvre des schémas de programmation traitant des structures de données, et des schémas algorithmiques. Cette approche, visant à rétablir l'écho immédiat des opérations, est connue dans la littérature sous le nom de « programmation basée sur l'exemple ». Dans le chapitre suivant, nous décrivons ce paradigme, les attentes de ses auteurs, et les contraintes qu'il entraîne.

Chapitre 2

Programmation « sur exemple » : définition, illustrations, et pertinence en apprentissage

Résumé. *Au cours du chapitre précédent, nous avons dégagé quatre difficultés essentielles de la programmation, enseignée de façon traditionnelle : la perte de la manipulation directe, l'usage de notations abstraites, le « faire-faire », et le fossé cognitif entre l'univers de l'informatique et le domaine de l'application.*

Ce chapitre définit et illustre un paradigme alternatif de conception de programmes, la « programmation basée sur l'exemple » (Myers 1986). Celle-ci a pour principale caractéristique que l'exécution se déroule parallèlement à la conception du programme.

Il paraît ainsi naturel de penser que l'approche basée sur exemple puisse aplanir les difficultés liées à l'abstraction et à la perte de manipulation directe, et donc puisse faciliter l'apprentissage de la programmation, en supportant une approche incrémentale qui divise la difficulté.

Après avoir expliqué les différents concepts associés à ce paradigme, en nous appuyant sur des exemples ad hoc ou issus de la littérature, nous cherchons à dépasser les préjugés intuitifs, en discutant précisément des contraintes et des différentes applications de cette approche, dans le domaine de la programmation. Cette analyse nous conduit à définir la démarche de nos travaux de recherche.

1 Concepts et Approches : définitions et illustrations

Dans ce chapitre, nous décrivons le paradigme de programmation « basée sur l'exemple », expliquons en quoi celui-ci constitue une rupture du cycle habituel de développement, et cherchons à voir comment il pourrait être utilisé dans le cadre de l'initiation à la programmation. Pour cela, nous commençons dans cette section par définir les différents concepts afférents, et illustrons ces définitions à l'aide de maquettes ou d'exemples tirés de logiciels commerciaux ou issus de la recherche académique.

Le paradigme de programmation « basée sur l'exemple », est formalisé par Halbert (Halbert 1984) puis Myers (Myers 1986) à sa suite, par la définition :

> *Un système est dit « basé sur exemple » si une instance d'exécution se déroule* ***parallèlement*** *à la conception du programme.*

Le terme central ici est « parallèlement ». La caractéristique principale de la programmation « basée sur l'exemple » sera donc que le programme va être débogué au moment même de l'édition. A chaque instruction, une instance d'exécution permettra au programmeur, en quasi temps réel, de recevoir un écho sémantique de la validité de son algorithme. La Figure 8 et la Figure 9 illustrent, à l'aide du formalisme de description de tâches CTT[1] (ref) les différences entre l'approche « standard » et le paradigme « avec exemple ».

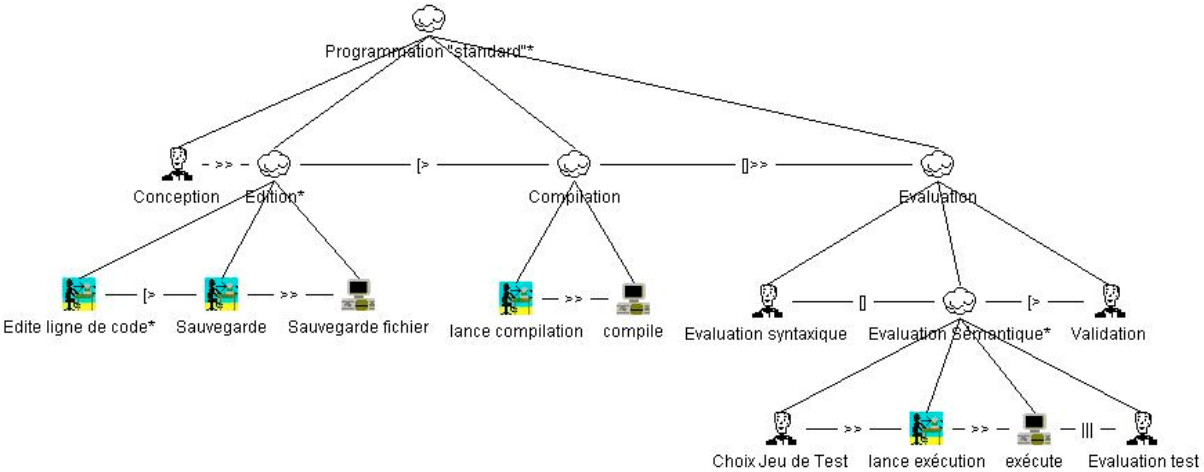


Figure 8 – Décomposition de la tâche du programmeur dans un environnement « standard », sans exemple, dans le formalisme graphique CTT.

Le graphe représente une décomposition de la tâche, et se parcourt « en profondeur d'abord ». L'opérateur « * » placé à la racine figure un cycle, similaire à celui de la Figure 2. La phase de conception se réfère aux trois problématiques du « Quoi Faire ? », « Comment Faire ? », et

[1] Les deux diagrammes représentent une décomposition arborescente de l'activité du programmeur. Il existe quatre types de tâches : « utilisateur », « système », « interaction » et « abstraite » (tâche se décomposant en sous tâches de types hétérogènes). Les tâche d'un même niveau de décomposition sont reliées entre elles deux à deux par des opérateurs temporels, tels que le choix « [] » (seule une des deux tâches est exécutée, suivant le contexte), la séquence « >> », l'exécution parallèle « ||| », et l'interruption « [> » (le système sort du sous arbre correspondant à la tâche de gauche et démarre la tâche à droite). De plus, chaque tâche peut être facultative ([T]) ou itérative (T*).

« Comment Faire Faire ? » (elle n'a pas été éclatée sur la Figure 8 pour des raisons de lisibilité de la figure). De même, le « Comment Dire ? » y est figuré par le triptyque « Edition – Compilation – Evaluation syntaxique », et la phase d'évaluation sémantique correspond au « Comment Valider ? ». Elle peut entraîner une remise en question des algorithmes, de la stratégie utilisée, ou des spécifications de la tâche du processeur.

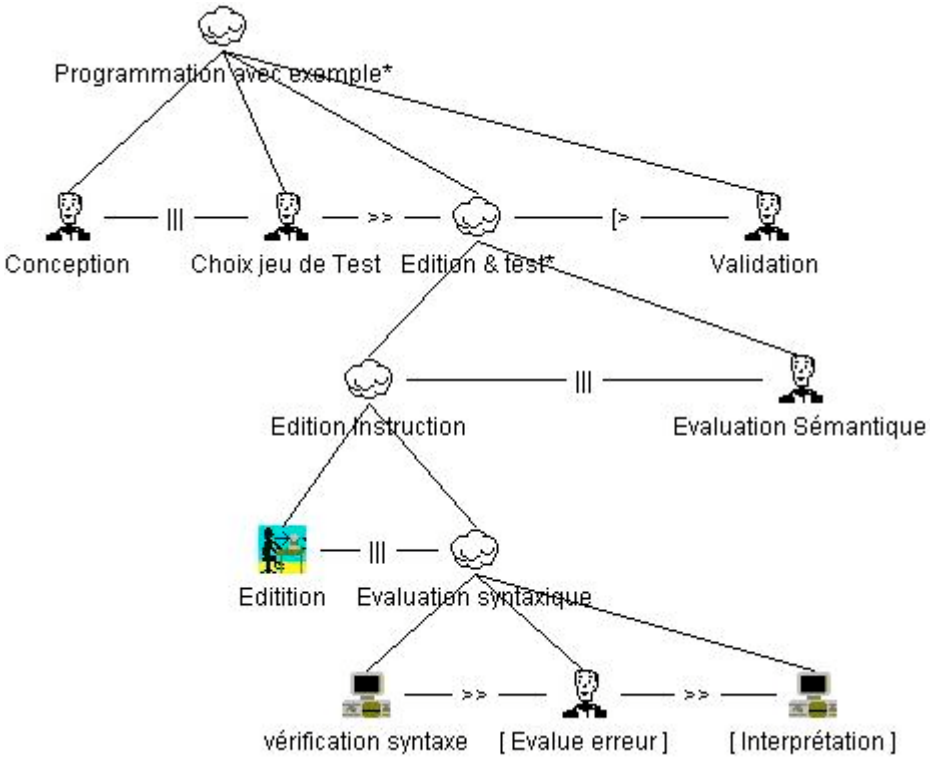


Figure 9 – Modélisation de la tâche de programmation « avec exemple » dans le formalisme graphique CTT.

Les systèmes de programmation « basée sur l'exemple » peuvent être ensuite classés en deux catégories : « avec » ou « sur » l'exemple. Dans le premier cas, l'utilisateur édite explicitement le programme (abstrait), et les valeurs concrètes de l'exemple sont utilisées pour faire de la « visualisation de programme », et du débogage en temps réel. Dans le second cas, le programmeur manipule les données directement, et le programme qui est construit en parallèle par le système lui-même.

1.1 Programmation « avec » Exemple : Illustration.

Pour illustrer le paradigme de programmation « avec » exemple, nous nous proposons de prendre un exemple simple, et graphique. Nous présentons dans cette section une maquette d'un hypothétique environnement « avec exemple » pour le langage LOGO. Elle va nous permettre de décrire, sur un exemple concret, le fonctionnement du paradigme « avec exemple ». L'étude de cas (très simple) consiste à programmer une fonction qui pilote la tortue graphique afin de dessiner un carré dont la taille des côtés est fournie en paramètre.

Cet environnement se compose d'une zone de dessin, d'une ligne de commande et d'une zone de texte où s'enregistre au fur et à mesure le programme défini « avec l'exemple ». Pour commencer, le programmeur saisit en « commande courante » la déclaration de la fonction :

« Pour Carre : Cote » et tape sur return. A ce moment là, le système fait automatiquement apparaître une boite de dialogue ayant pour vocation de récupérer une valeur concrète pour le paramètre de la fonction, afin de définir l'exemple courant (Figure 10).

Une fois cette information saisie, la déclaration est ajoutée à la zone du programme, et une table présentant la valeur courante du contexte du programme « Carre » apparaît. « Cote » y figure avec la valeur 50 (saisie dans la boite de dialogue précédente).

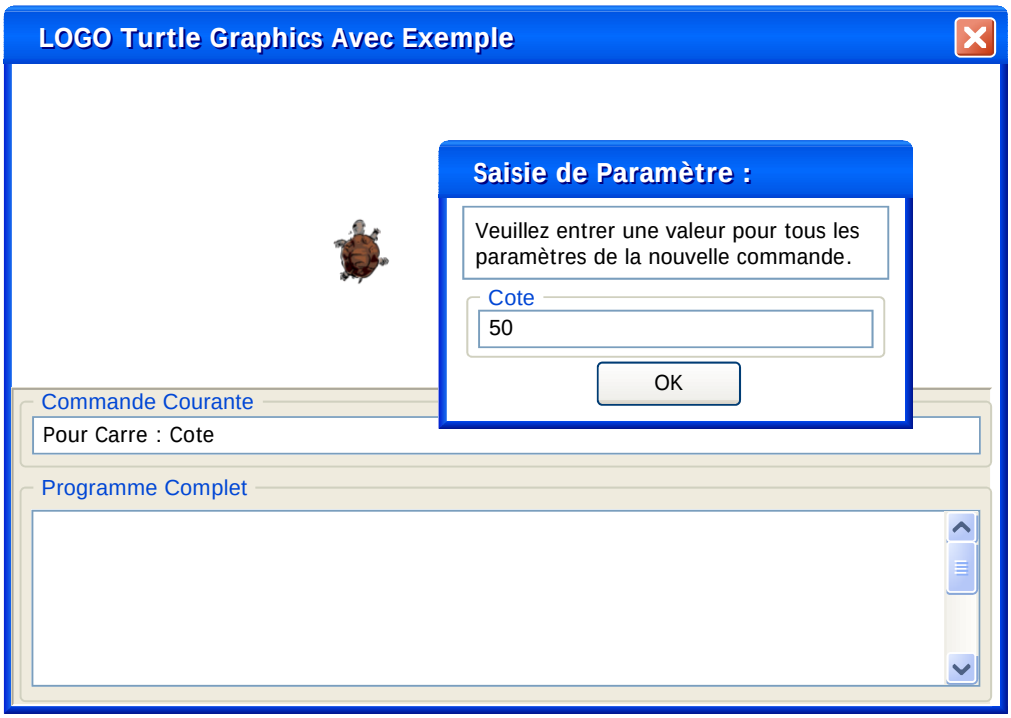


Figure 10 – Edition « avec exemple » d'un programme graphique en LOGO : définition des valeurs des paramètres de l'exemple.

Après quoi, le programmeur peut écrire dans la zone de commande courante l'instruction qui compose le corps de cette procédure. Il écrit « Répète 4 ». A ce moment là, le contexte est mis à jour pour passer dans un contexte d'itération, et la variable système « loop » qui stocke le tour courant de l'exécution apparaît dans la liste des variables. L'édition de l'instruction de répétition se poursuit, et lorsque la première instruction composant la boucle est écrite, le système l'exécute (Figure 11).

Ce comportement met en exergue la différence entre l'approche « avec exemple » et un simple interpréteur. Le contexte dynamique en mémoire permet l'exécution d'instructions manipulant des variables préalablement définies. De la sorte, il est possible de construire un programme complet, avec variables et paramètres, en l'interprétant au fur et à mesure de son édition.

De la même façon, la seconde instruction élémentaire est exécutée dès que son paramètre est renseigné (Figure 12). Enfin, lorsque le programmeur met fin à la séquence composant l'itération («] »), l'environnement en réponse exécute la boucle en dépilant le contexte itératif. Le système peut donc fournir un écho pour les 2ème (Figure 13), 3ème, et 4ème tours de boucle. Enfin, en frappant sur la touche « return » du clavier, le programmeur confirme l'insertion de

l'instruction au programme. Il peut ensuite terminer le programme en saisissant la commande « fin ».

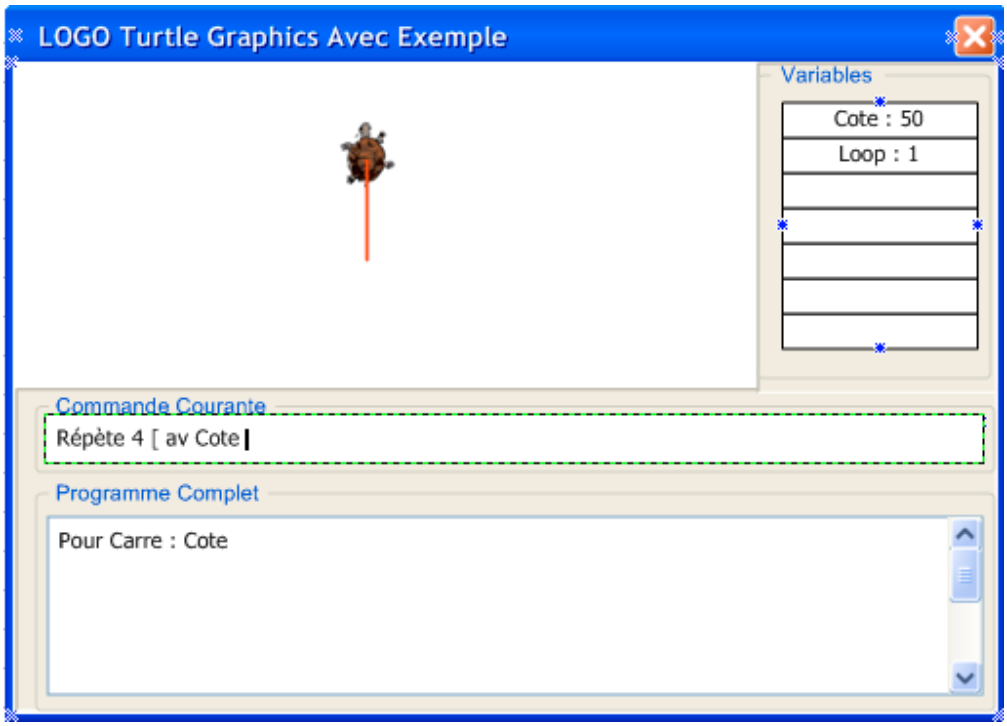


Figure 11 – Exécution progressive en temps réel de la commande d'itération.

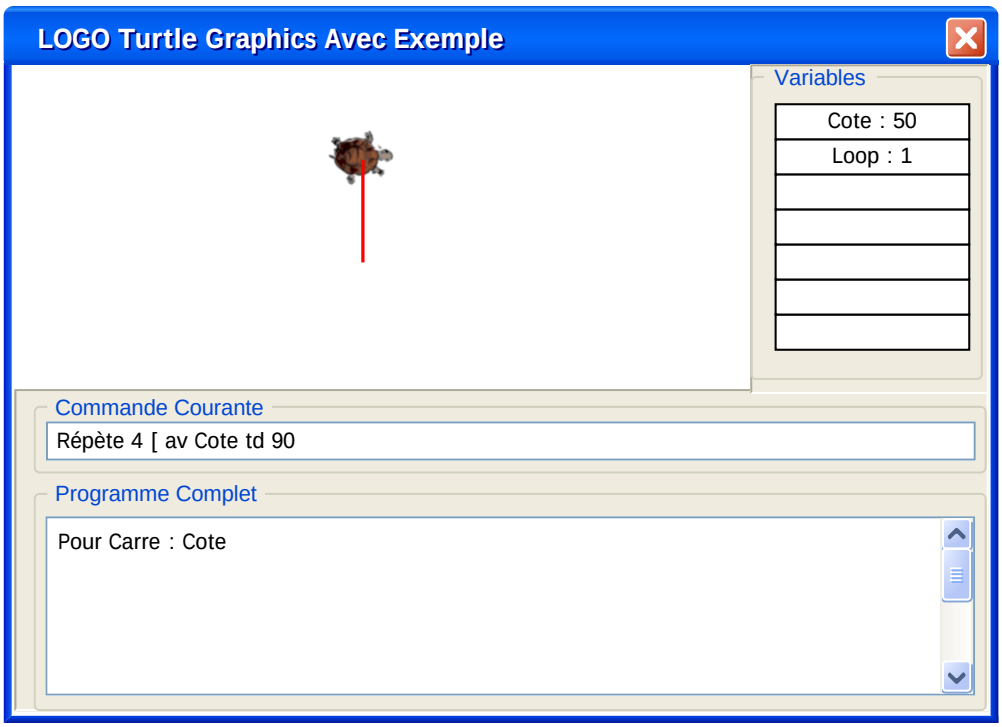


Figure 12 – Exécution de la deuxième instruction de la boucle.

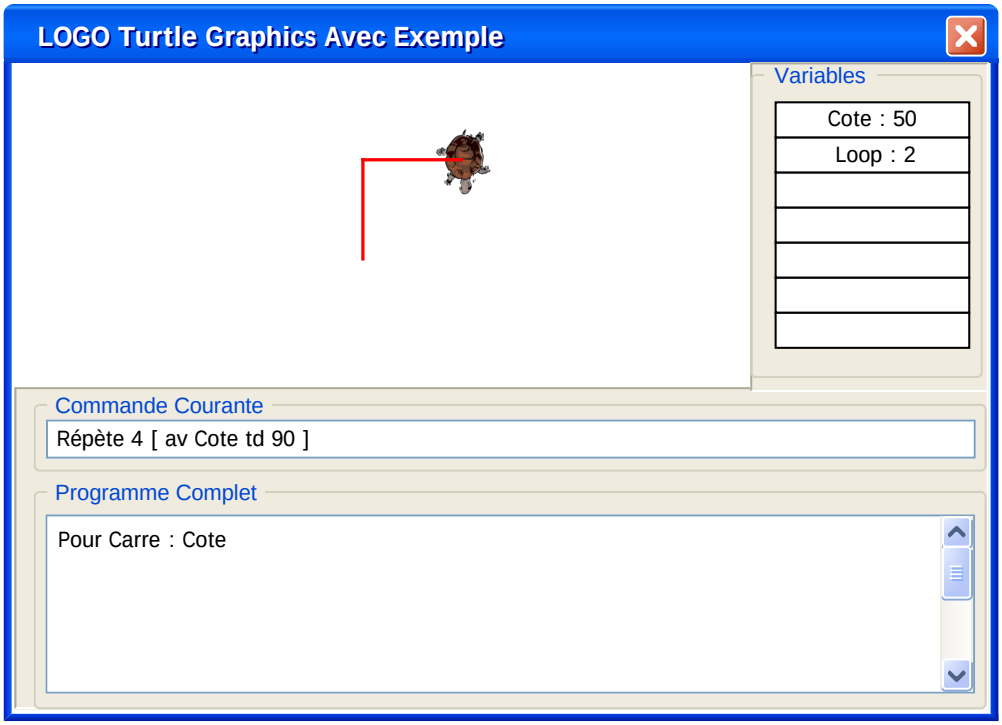


Figure 13 – Exécution de la boucle de dessin du carré : fin du deuxième tour.

Ce style de programmation « avec exemple » se caractérise donc par l'usage d'un contexte dynamique, mis à jour au fur et à mesure de l'édition du programme, qui permet au programmeur de disposer d'un écho sémantique sur l'exemple au moment même où il édite son programme (Figure 14).

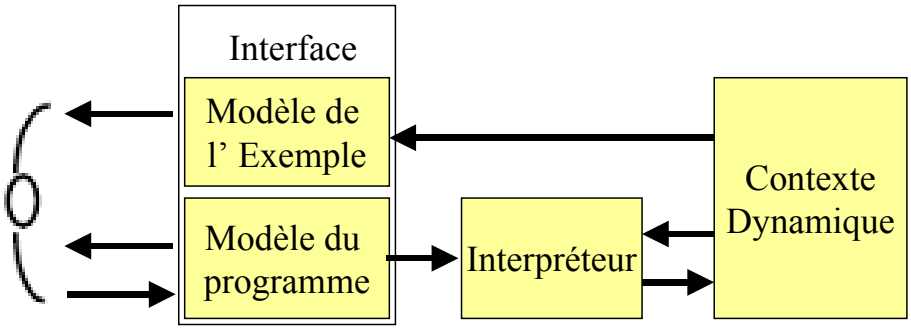


Figure 14 – Modèle de fonctionnement d'un système de programmation « avec » exemple.

1.2 Programmation « sur » Exemple et illustrations.

Alors que dans le cas précédent le programmeur manipulait une représentation du programme, et utilisait l'exemple graphique (ou pas) pour superviser en temps réel le comportement de son programme, le paradigme « sur exemple » (Figure 15) prône l'usage de l'exemple lui-même comme outil de construction. Il se base sur l'utilisation de notations graphiques ou semi-graphiques pour représenter l'espace des données concrètes.

Le système enregistre la séquence d'actions de l'utilisateur sur les données et en génère un programme pouvant être réutilisé sur d'autres cas « analogues ».

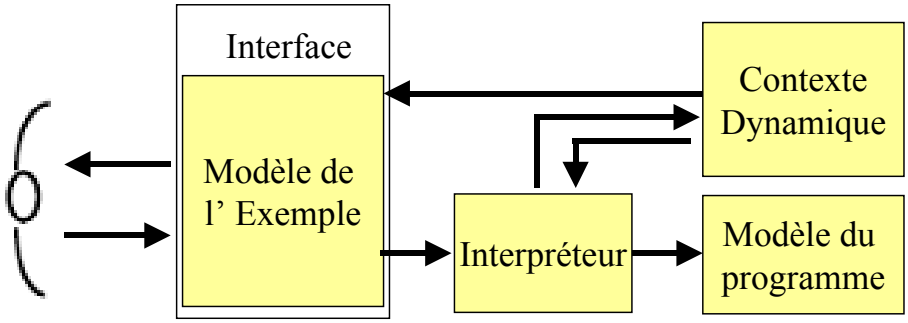


Figure 15 – Modèle de fonctionnement d'un système de programmation « sur» exemple.

Nous présentons pour illustration une maquette d'un hypothétique environnement « sur exemple » pour le langage LOGO (Figure 16). Elle va nous permettre de décrire, sur un exemple concret, le fonctionnement du paradigme « sur exemple ». L'étude de cas est la même que pour la programmation « avec exemple » : la programmation d'un carré de côté n.

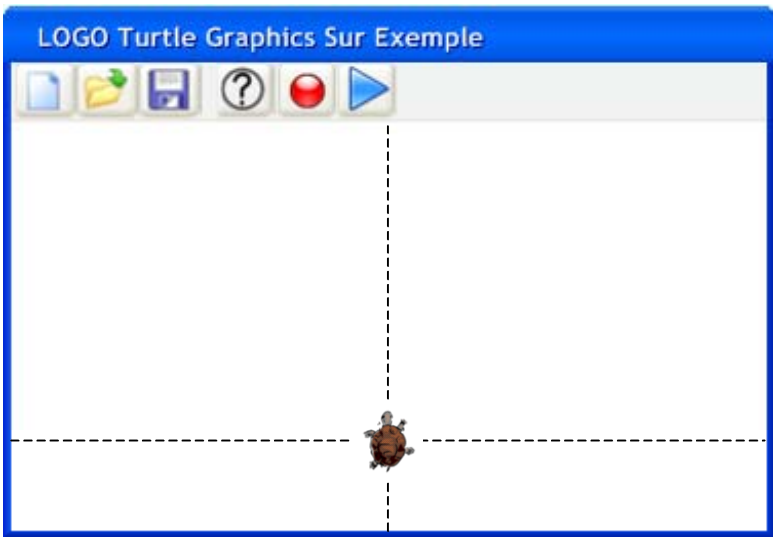


Figure 16 – Environnement « sur » exemple pour la tortue de LOGO : état initial.

Le paradigme « sur exemple » se base sur la manipulation directe de l'exemple, et donc des objets de la tâche ; on remarque immédiatement la disparition de la ligne de commande, et la représentation graphique des caractéristiques des actions élémentaires (les distances et les angles – Figure 17), qui supplante la représentation textuelle structurée du programme.

Les actions sur la tortue graphique sont spécifiées à la souris ; en l'occurrence un glisser – déposer avec le bouton gauche de la souris effectue une translation, et un glisser – déposer en enfonçant le bouton droit exprime une rotation. Dans le premier cas, le mouvement est décomposé en une rotation et une translation selon la direction, pour correspondre aux actions du processeur de la tortue.

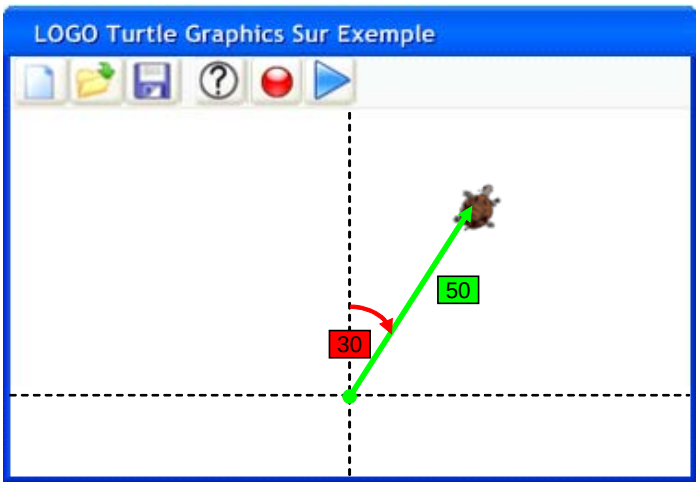


Figure 17 – Décomposition d'une en actions de base de la tortue (td 30 av 50). Les paramètres des actions sont représentés directement sur le dessin.

Cependant, pour écrire des programmes plus complexes qu'une simple séquence d'actions, il s'avère nécessaire de mettre en œuvre des techniques plus complexes d'interaction ou d'intelligence artificielle, afin de permettre au programmeur d'exprimer des structures de contrôle conditionnelles ou itératives et de distinguer les paramètres ou les variables des simples constantes. Ici, pour programmer la répétition, on enfonce le bouton « enregistrement » puis on réalise les deux actions « av 50 » et « td 90 » par manipulation directe (Figure 18). Enfin, pour marquer la fin du corps de la répétition, on relâche le bouton « enregistrement ».

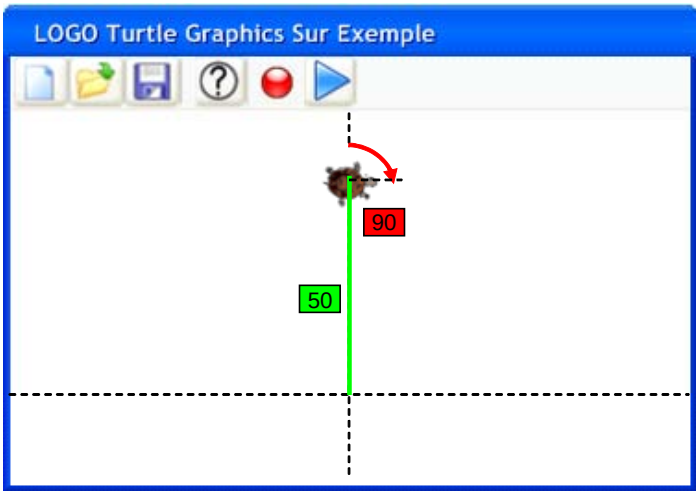


Figure 18 – Réalisation « sur exemple » du corps de la répétition du programme du carré.

A partir de ce moment, pour exécuter les trois autres itérations de la répétition, le programmeur clique trois fois sur la touche « lecture ». On arrive ainsi à l'état de la Figure 19, qui correspondrait au programme LOGO :

```
Répète 4 [av 50 td 90]
```

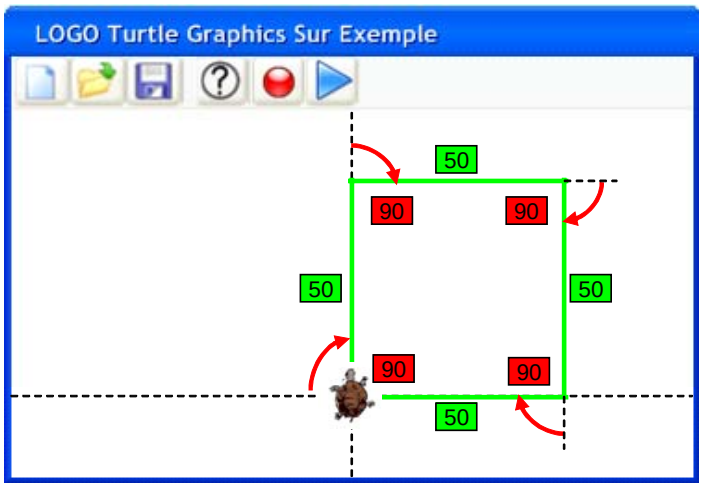


Figure 19 – Après avoir rejoué trois fois la séquence enregistrée : on a programmé une commande qui dessine un carré de côté 50.

Une dernière difficulté consiste à paramétrer le programme généré par le côté du carré. Pour cela, plusieurs approches ont été testées dans la littérature. Ici, on clique avec le bouton droit sur la longueur des tracés (étiquette « 50 »), et un menu déroulant nous permet de « définir un paramètre », auquel on donne le nom « côté » (Figure 20). Les quatre étiquettes sont alors mises à jour simultanément (ayant été définies par rejeu, elles sont liées), et on a fini de définir :

```
Pour Carré : côté
Répète 4 [av côté td 90]
Fin Carré
```

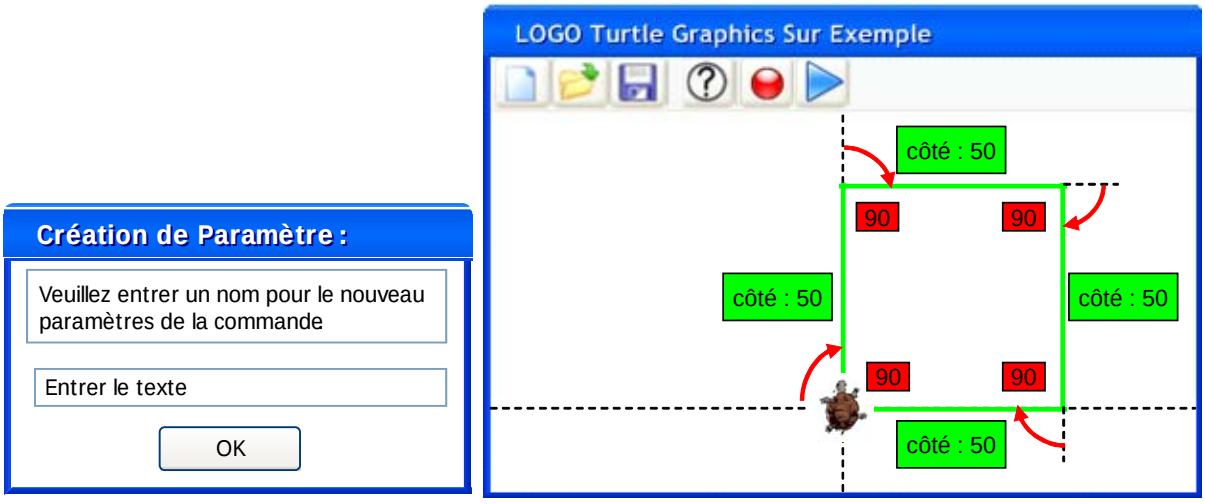


Figure 20 – Création d'un paramètre et écho sur la figure.

2 Utilisation dans des systèmes réels

Dans la pratique, peu de systèmes se limitent à l'utilisation de la programmation avec exemple. Certains interpréteurs[1] en sont une illustration, ou encore l'évaluation des formules dans un tableur. En revanche, la programmation sur exemple a été déclinée dans de nombreuses applications. Nous nous contenterons ici de détailler quelques-unes des plus marquantes, afin de souligner certains points importants, concernant le mode de représentation des programmes ou la manière de construire ces derniers.

En effet, alors que la programmation textuelle traditionnelle utilise une représentation symbolique textuelle au travers d'une explicitation complète de ses mécanismes (les seules techniques implicites, comme par exemple la déclaration automatique des variables, sont rarement prisées des programmeurs professionnels), la programmation basée sur exemple utilise abondamment icônes, métaphores et règles implicites ou même inférence.

Dans la littérature deux approches ont ainsi été étudiées pour permettre la définition des variables et des structures de contrôle:

- d'une part ceux qui adoptent une approche *impérative*, où le programmeur décrit la suite d'opérations exécutées par le système (comme dans l'exemple précédent) ;
- d'autre part ceux qui adoptent une approche *déclarative*, où le programmeur interagit avec l'exemple par description des relations qui existent entre les objets.

2.1 Approche impérative, « par démonstration »

Ces derniers ont souvent recours à une technique appelée « programmation par démonstration » : l'ordinateur enregistre la séquence des différentes opérations effectuées par le programmeur sur les objets de l'exemple par « manipulation directe », il y a donc également programmation visuelle. Notons que le terme « démonstration » renvoie non pas à un aspect déductif (acceptation mathématique du terme), mais plus à une action d'exhibition (montrer, à l'aide d'un exemple).

Les concepteurs peuvent alors se placer soit dans un espace « sémantique », qui décrit les objets du point de vue du système, avec un fort niveau d'abstraction, soit dans un espace « pragmatique », centré sur la tâche de l'utilisateur, et qui adopte le point de vue de ce dernier. Ce dernier cas est formalisé par Halbert (Halbert 1984) à travers la définition suivante :

> *L'utilisateur écrit un programme qui effectue une tâche particulière en utilisant les mêmes commandes que celles qu'il utiliserait pour effectuer cette tâche de façon interactive. L'utilisateur programme dans l'interface du système.*

[1] Pour être qualifiés de programmation « avec exemple », un interpréteur doit posséder un contexte dynamique, qui permet d'interpréter pas à pas une suite d'instructions. C'est le cas de nombreux interpréteurs LISP, mais ce n'est pas le cas pour la plupart des interpréteurs des langages impératifs.

Cette décomposition correspond à un critère que Chang (Chang 1986) emploie pour la classification des « Visual Languages » :

- *Les objets du domaine sont naturellement visualisables ou ne le sont pas (leur visualisation passe alors par des conventions plus ou moins naturelles pour l'utilisateur, c'est la notion d' « object icon ») et possèdent alors une représentation schématique.*

Dans le but d'illustrer cette décomposition, nous présenterons deux exemples issus de l'état de l'art, s'appuyant sur une représentation schématique du domaine sémantique de l'informatique, et deux exemples s'appuyant sur la représentation graphique pragmatique des objets du domaine cible.

2.1.1 Pygmalion : une représentation purement schématique

Pygmalion (Smith 1993) représente une première tentative de programmation visuelle par démonstration, basée sur la métaphore du « tableau noir » : la zone de travail est vue comme un tableau où le programmeur peut « dessiner » ses idées. Cette métaphore a été inspirée par l'importance de ce média dans la communication au sein de la communauté scientifique. Il a été à l'origine du concept d'icône disposant à la fois d'une image, d'un contenu (du texte pour une icône de document), et d'un comportement (déplacement d'un fichier par drag and drop).

A travers l'exemple qui suit (la programmation dans Pygmalion de la fonction « n ! » de façon récursive, dont l'algorithme est présenté en C dans le paragraphe suivant), nous tâchons d'expliquer les différents concepts relevant de la programmation visuelle par démonstration, dans le cas d'une notation schématique.

```
int factorial (int n) {
   if (n==1)  then return 1 ;
      else return n*factorial(n-1) ;
}
```

La programmation de la fonction commence en lui associant une icône, c'est-à-dire une boîte à laquelle on associe :

- Deux « sous-boîtes » correspondant respectivement au paramètre d'entrée et à la valeur de retour de la fonction.
- Un symbole, « ! ».

L'approche de Pygmalion étant « par l'exemple », il convient ensuite d'instancier le paramètre d'entrée, auquel nous associons la valeur « 6 » (Figure 21). La fonction est alors déclarée, et peut être appelée par le système.

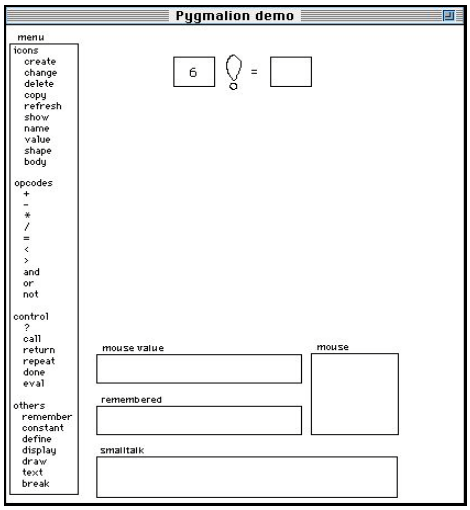


Figure 21 – L'environnement Pygmalion lors de l'écriture de n ! : définition et instanciation des entrées.

La définition de toute fonction récursive nécessite de spécifier le comportement du processeur dans deux cas :

- le cas de récurrence (n > 1), qui sera rejoué par le système jusqu'à arriver au :
- cas de base (n=1).

Comme l'exemple que nous avons choisi est n=6, le programmeur doit décrire en premier le cas de récurrence (Figure 22), car en programmation « basée sur exemple », l'ordre d'édition des instructions est calqué sur l'ordre de leur exécution.

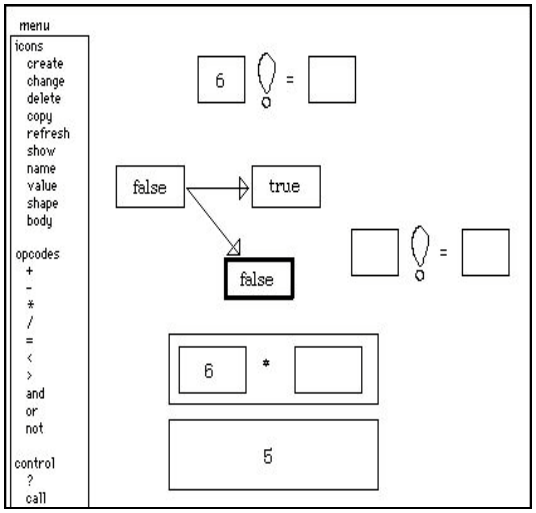


Figure 22 – Cas de récurrence pour la définition de n ! avec Pygmalion.

Le test apparaît sous la forme de trois boîtes. Celle en haut à gauche contient la formule, et les deux autres les deux résultats possibles. Le programmeur manipule les formules en ajoutant les opérateurs élémentaires au tableau. Ainsi la formule « n=1 » requiert d'ajouter grâce au menu à gauche l'opérateur « = ». Une convention de dialogue permet ensuite de faire la distinction entre valeurs et références. Ainsi, l'utilisation d'une référence au paramètre d'entrée se traduit par un glisser-déposer de l'icône, alors que pour manipuler une constante

numérique (« 1 ») on utilise le case de paramètre comme un champ texte, en tapant la valeur à l'intérieur. On distingue ainsi par convention de dialogue l'utilisation du paramètre de factorielle, dont la valeur dans le contexte dynamique est 6, de la constante numérique 6. L'évaluation est ensuite réalisée dès que possible par le système, et l'écho en est immédiat (Figure 23).

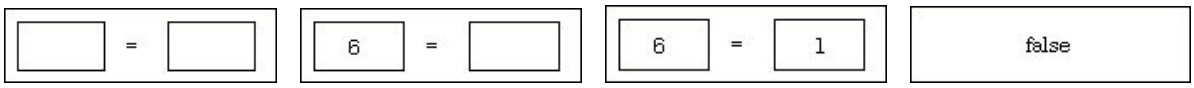


Figure 23 – Programmation « par démonstration » de la formule « n==1 » ; création, entrée d'une référence, entrée d'une constante numérique, écho immédiat.

On peut remarquer que, même si la représentation du modèle de l'exemple (Figure 15) est schématique dans Pygmalion, c'est bien « sur exemple » que le programmeur agit, et il n'y a aucune représentation « globale » du programme. C'est l'état courant du contexte du programme qui est schématisé, et non le flot de contrôle, ou le flot de données.

Ensuite, le résultat de ce test est déposé dans l'icône de formule du test, ce qui le fait basculer directement dans la branche « sinon ». On construit de même les expressions « n-1 », « n x ? » et l'appel à factorielle pour arriver dans l'état illustré par la Figure 22. On fait glisser l'expression « n-1 » dans l'icône de paramètre d'entrée de factorielle, et le programme s'exécute de façon automatique, jusqu'à arriver dans l'état correspondant au cas de base de la récurrence (à droite). A ce moment là, on remplit l'icône de résultat, et le système dépile automatiquement le contexte pour revenir au dernier état correspondant au comportement récurrent (Figure 24).

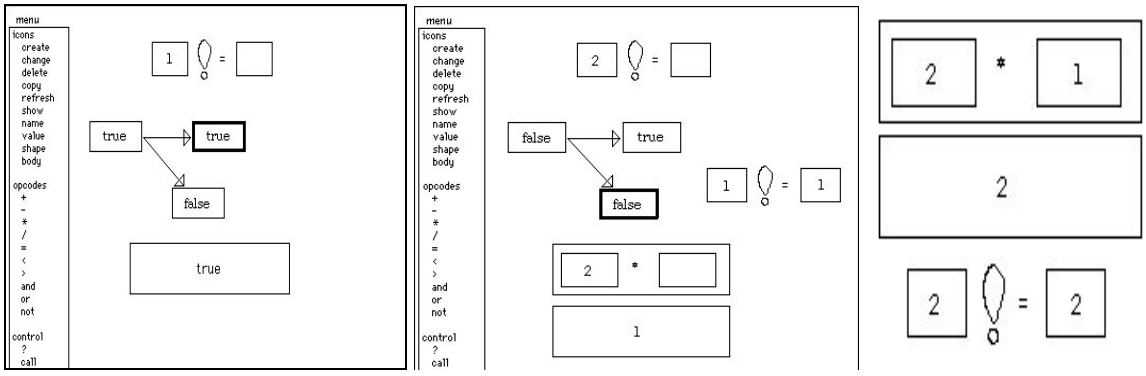


Figure 24 – Définition du cas de base de l'appel récursif, puis dépilement de l'appel récursif.

Pour compléter le programme, le programmeur fait glisser l'icône de résultat de « 1 ! » dans « 2 * ? », le résultat est automatiquement interprété, et la programmation de « n ! » sur l'exemple « n=6 » peut être conclue en déposant cette icône dans le résultat de « 2 ! ». L'environnement termine l'exécution automatiquement en dépilant les contextes des appels récursifs pour arriver finalement au résultat (Figure 25).

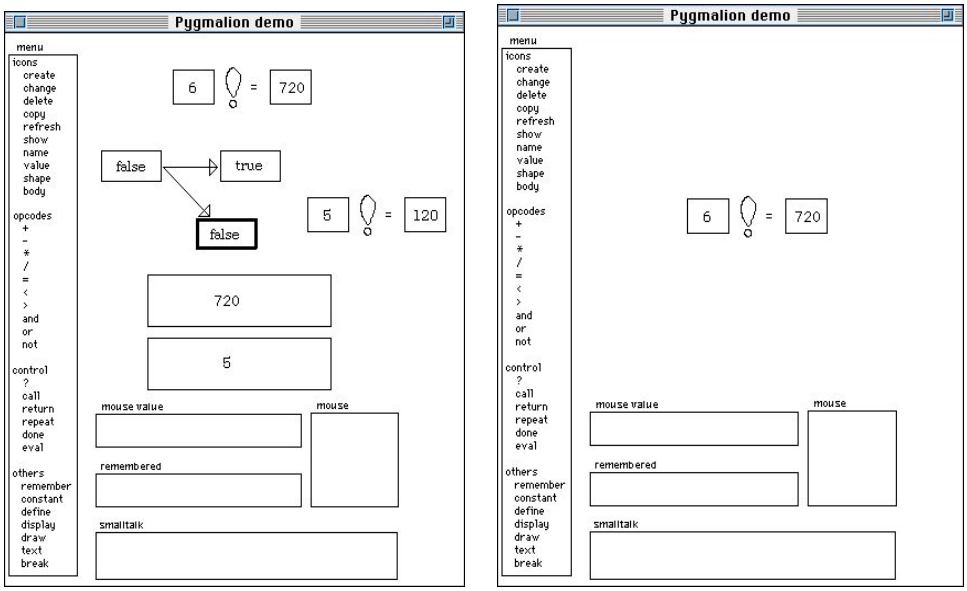


Figure 25 – Résultat du programme « n ! » sous Pygmalion, avec l'exemple « 6 ! »

Cette approche a été reprise depuis dans ToonTalk (Kahn 2001), où le programmeur est un acteur dans un monde virtuel, et où chaque objet – abstrait – des programmes est représenté grâce à une métaphore. L'enjeu n'est pas de faire oublier que l'on programme, mais d'abaisser la barrière d'abstraction d'un activité qui est assumée comme étant de la programmation.

2.1.2 ToonTalk, une représentation schématique par métaphore

La Figure 26 résume la métaphore de jeu de construction utilisée par ToonTalk : chaque objet du monde de la programmation y est associé à son équivalent dans le monde de ToonTalk. L'application s'adresse essentiellement à des enfants, et cette métaphore fait l'analogie entre la programmation et un jeu de construction. Un programme complet en ToonTalk se représente sous la forme d'une ville, les sous-programmes, les processus sont représentés par des maisons, que l'on peut faire communiquer à l'aide de pigeons voyageurs. Ceux-ci peuvent transporter des objets depuis leur maison jusqu'à des nids (ce qui figure la synchronisation de processus par sémaphores). Chaque maison peut contenir plusieurs robots (=méthodes).

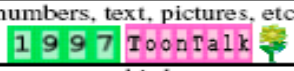
Computational Abstraction	ToonTalk Concretization
computation	city 
actor process concurrent object	house 
method clause	robot 
guard method preconditions	contents of thought bubble 
method actions body	actions taught to a robot
message array vector	box 
comparison test	set of scales 
process spawning	loaded truck 
process termination	bomb 
constants	numbers, text, pictures, etc. 
channel transmit capability message sending	bird 
channel receive capability message receiving	nest 
persistent storage file	notebook 

Figure 26 – L'usage de la métaphore du jeu de construction dans ToonTalk ; à gauche les concepts cibles, et à droite ces concepts vu à travers la métaphore.

ToonTalk possède cependant tout le pouvoir d'expression d'un langage concurrent à objets ; il contient dans sa métaphore presque tous les concepts utilisés par ses équivalents « textuels ». Aussi rien n'interdit de l'appliquer à des exercices classiques de tri de tableaux, ou à la programmation de programmes ayant une interface graphique.

Ainsi, pour programmer « n ! » de façon récursive (Figure 27), le concepteur va créer une équipe de deux robots (un correspondant à n=0, et l'autre à n>=1), et chaque appel récursif va créer un nouveau contexte (maison), le retour de valeur pendant le dépilage des contextes des appels récursifs se réalisant par l'envoi de pigeon voyageurs.

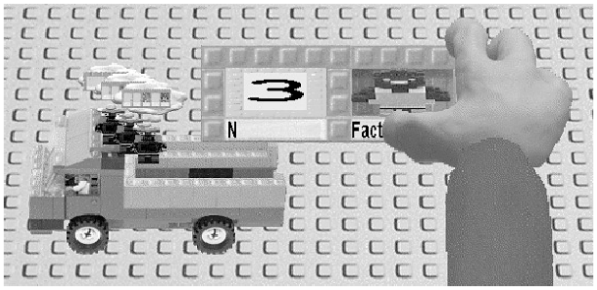


Figure 27 – Programmation de « n ! » dans la métaphore de ToonTalk.

C'est d'ailleurs bien là le principal atout des systèmes utilisant des représentations schématiques, avec ou sans métaphore. Etant donné qu'ils s'appuient sur une sémantique abstraite du domaine informatique, il n'y a pas de limite à leur domaine d'application (jeux, traitements de texte, bases de données). C'est pourquoi nous les qualifions de **sémantiques** en ce qu'ils manipulent un modèle de l'état interne de l'ordinateur. A contrario, dans la maquette d'application pour la tortue graphique LOGO, les conventions de dialogue ne peuvent pas être transférées à un autre domaine que celui du dessin avec la tortue. Nous qualifions une telle approche de **pragmatique**, en ce qu'elle est fortement liée au domaine d'application. Les systèmes basés sur cette approche trouvent leur justification dans le fait que la perte de pouvoir d'expression y est généralement compensée par une utilisabilité accrue du système.

2.1.3 StageCast Creator, une représentation pragmatique

StageCast Creator (Smith 2000), est un système dont le domaine d'application est la conception de simulations ou de jeux, dans un cadre ludique ou éducatif. Le style de programmation y est événementiel et s'inspire du paradigme des feuilles de calcul. La Figure 28 illustre les différents composants de l'environnement, dans le cas de la programmation d'un jeu d'aventure. Le « micromonde » (1) est une grille, qui est associée à une image de fond, et où chaque cellule peut contenir un agent « intelligent ». Lorsqu'une garde particulière est vérifiée, l'agent exécute une séquence d'actions. La barre d'outils (2) permet de créer de nouveaux agents, et de définir leur apparence et leur comportement. Garde et actions leur sont spécifiées par démonstration par le programmeur (redessiner, agrandir, déplacer par drag and drop, … les agents concernés).

Les agents sont ensuite activés par le système en suivant une structure de type :

```
REPETER TOUJOURS
   SI <garde 1> FAIRE <séquence 1>
   …
   SI <garde N> FAIRE <séquence N>
FIN REPETER
```

Cela permet de masquer les concepts de flots de contrôle aux utilisateurs, et assure en conjonction avec la représentation graphique une barrière d'abstraction très basse. Le panneau latéral (3) offre au programmeur une représentation de cette structure, et donc une vue statique générale du comportement du programme en cours de développement.

Figure 28 – L'environnement de conception de simulations de StageCast Creator

La Figure 29 illustre un exemple de programmation de comportement. Ici, la règle que le programmeur veut ajouter porte sur l'agent « fermier » qui va devoir pousser l'agent « cochon » pour le faire avancer. Dans un premier temps, le programmeur sélectionne l'agent sur lequel va porter la règle. Puis il étire (étape 2) le cadre, pour définir l'état de départ. Ici l'état de départ est : « le fermier sur la case de gauche, le cochon sur le case du milieu et rien sur la case à droite. Enfin il déplace le cochon sur la case de droite et confirme. La règle est alors ajoutée au système et le programmeur peut jouer la simulation afin de la tester.

Figure 29 – Programmation par démonstration du comportement d'un agent sur StageCast.

2.1.4 Le domaine de la CAO paramétrique, la pragmatique orientée métier

Hormis le cadre des micromondes et des simulateurs, la « programmation par démonstration » a également été appliquée sans nécessiter de notation schématique, dans le domaine de la CAO paramétrique, dans LIKE (Girard 1992) et EbP (Potier 1995) notamment. L'utilisateur crée la pièce en mode graphique, en spécifiant à l'aide de conventions de dialogue quelles cotes sont les paramètres caractéristiques de la famille de pièces. Ces conventions permettent de créer un filtre de sélection pour connecter l'interface utilisateur au moteur de programmation par démonstration (Figure 30). De plus, une calculatrice grapho-numérique permet l'accès aux structures de contrôles itératives ou conditionnelles (Figure 31).

La « PpD » s'est avérée particulièrement efficace dans le contexte de la CAO paramétrique car deux conditions y sont réunies : (1) les objets du domaine sont intrinsèquement graphiques, et (2) l'interface du système permet d'exprimer naturellement les relations entre objets. Ces deux conditions sont nécessaires et suffisantes pour l'usage d'une approche pragmatique, caractérisée par la manipulation directe de représentations naturelles des objets du domaine.

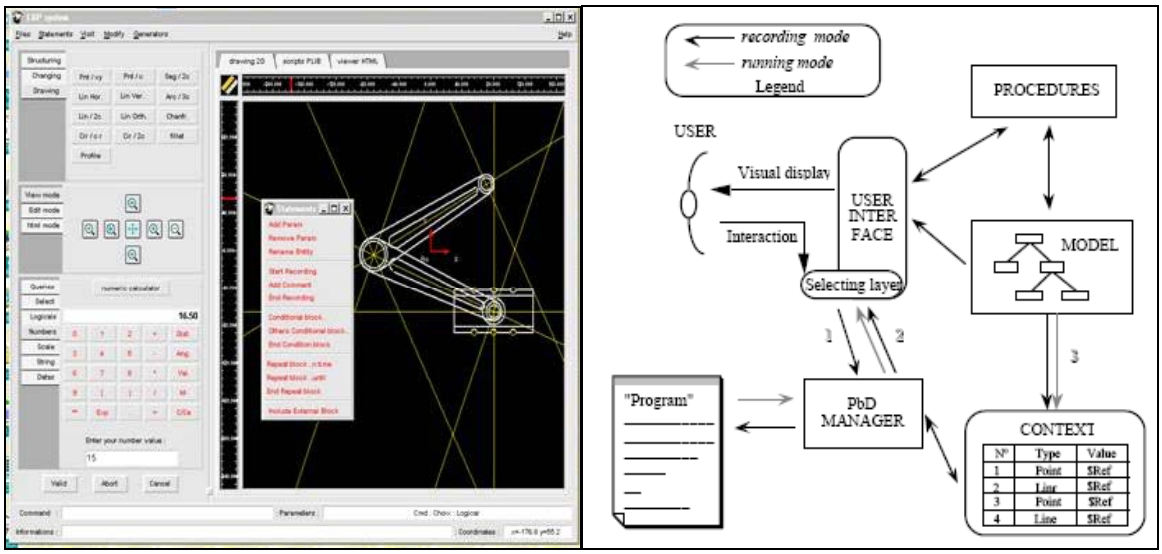
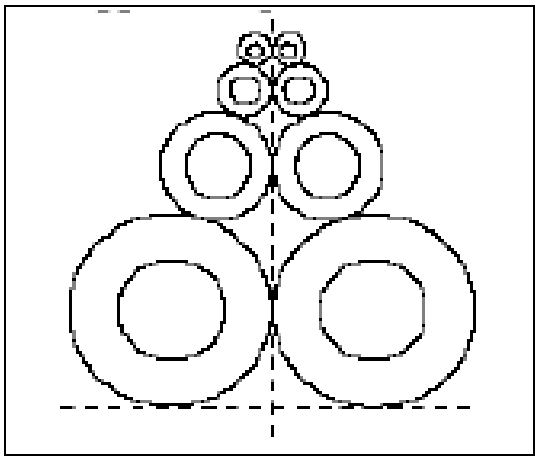


Figure 30 – Programmation par démonstration en CAO : le modèle du système EBP.



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

Figure 31 – Programme générique construit à partir d'une figure de géométrie, utilisant une itération.

2.2 Approche déclarative, par « inférence probable »

A cette approche « explicite », impérative, où le programmeur désigne à l'aide de conventions de dialogue les paramètres et distingue les variables des constantes, et où l'usage de structures alternatives ou répétitives passe par l'utilisation consciente d'opérateurs par le programmeur, s'oppose une approche « par inférence probable », où le système déduit automatiquement l'intention du programmeur, et sépare de façon plus ou moins autonome les données statiques de celles qui dépendent de l'instance d'exécution sans intervention de l'utilisateur. Cette fonctionnalité d' « inférence » connaît un engouement important parmi les différents prototypes issus de la recherche académique dans les années 90. Pour autant, leur définition de ce concept y est assez floue : Myers (Myers 1986) parle en effet de « l'aptitude du système à générer de nouveau faits à partir d'autres informations ». Les systèmes de programmation qui font appel à l'inférence ont en commun de chercher à décharger le concepteur de la question du « comment ». Ces derniers se contentent d'exprimer à l'aide d'un exemple le type de donnée cible et les résultats attendus. L'avantage majeur se situe donc au niveau de la barrière d'abstraction : de tels systèmes ont pour avantage d'être accessibles à des utilisateurs n'ayant aucun savoir faire algorithmique. Le style de la programmation y est par conséquent le plus souvent déclaratif.

2.2.1 Illustration

Pour illustrer le fonctionnement de ce paradigme, nous nous appuyons sur un exemple tiré de la recherche académique, qui nous semble représenter particulièrement fidèlement cette approche. SmartEdit est un système permettant de construire des macro-commandes à partir d'expressions régulières inférées des sélections de l'utilisateur. Dans l'exemple de la Figure 32, il s'agit de supprimer dans un texte tous les commentaires HTML. Pour cela, la tâche du programmeur se décompose en deux activités. D'abord, il enregistre la séquence d'actions qu'il souhaite voir généralisée : à savoir, déplacer le curseur avec la souris (1) sélectionner l'ensemble du commentaire avec la souris, puis supprimer (2). Puis, il supervise l'exécution en pas à pas du programme inféré (3 et 4). Ainsi, le système généralise le déplacement au 20^{e} caractère comme étant un déplacement au début du motif « < !-- ». Le « programmeur » peut ne pas être satisfait du résultat de l'inférence, et le système permet de remplacer celui-ci par un autre résultat qu'il avait initialement jugé moins plausible. Une autre technique classique permettant de résoudre ce type de problème est de passer par un nouvel exemple. L'avantage de l'approche inférentielle est de ne pas requérir que l'utilisateur acquière quelque concept que ce soit. Ici, elle lui permet de faire un filtre, et d'y associer une macro, sans avoir aucune connaissance en expressions régulières, ni en algèbre booléenne.

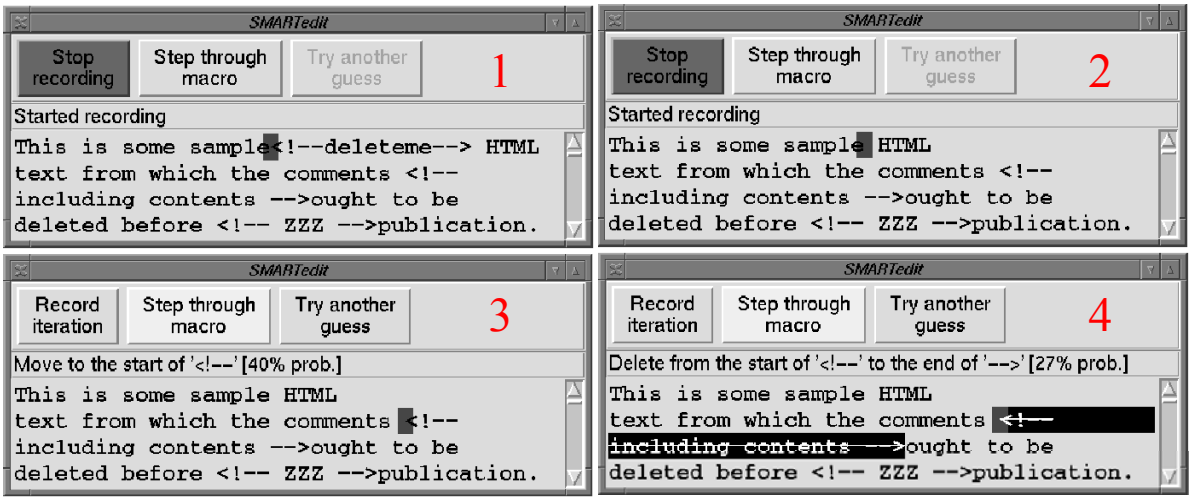


Figure 32 – Smart Edit (Lau, Wolfman et al. 2001)

Pour autant, malgré la pertinence de cette technique dans le domaine de la « end-user programming », l'approche inférentielle n'a jamais confirmé dans des applications « à taille réelle » l'engouement incontestable qu'elle a soulevé en tant que sujet de recherche académique. L'explication se trouve certainement dans les contradictions intrinsèques soulevées par l'usage d'algorithmes d'intelligence artificielle (IA) pour analyser les entrées de l'utilisateur. Dans le cadre d'applications de conception, il est nécessaire que le concepteur soit toujours capable de prévoir le résultat des commandes qu'il utilise. Cela implique de fortes contraintes sur la robustesse, et sur la prévisibilité de l'état du système. Hors, les techniques d'IA adaptables à l'inférence sont soit efficientes (elles nécessitent peu de données) mais peu fiables, soit sont robustes et assez fiables et efficaces, mais nécessitent alors beaucoup de données pour fonctionner (c'est le cas des algorithmes issus de la fouille de donnée – *data mining*).

Le compromis entre la fiabilité et l'efficacité nécessaires pour rendre l'approche *utile* et la réactivité nécessaire pour la rendre *utilisable* est donc très difficile à trouver, et c'est sans doute pourquoi cette approche d'inférence de l'intention du concepteur ne s'est pas répandue dans l'industrie. A contrario, dans un contexte où fiabilité et prédictibilité ne sont pas aussi cruciales, l'inférence peut se révéler pertinente : c'est notamment le cas pour le domaine des agents d'assistance.

2.3 Approche hybride

L'application commerciale la plus connue de l'approche « sur exemple » est le tableur (Figure 33). On remarque que la notation semi-graphique (la feuille de calcul – *spreadsheet* en anglais) y fait la part belle aux valeurs concrètes des données, au détriment de la représentation du programme sous-jacent (c'est pourquoi, bien que les formules soient saisies en ligne de commande, nous classons les tableurs comme systèmes « sur » exemple – et non « avec »). Dans le cas des tableurs, ce modèle sous jacent correspond aux formules et aux liens entre cellules, qui sont masqués dans la feuille de calcul elle-même, et qui n'apparaissent qu'au niveau local.

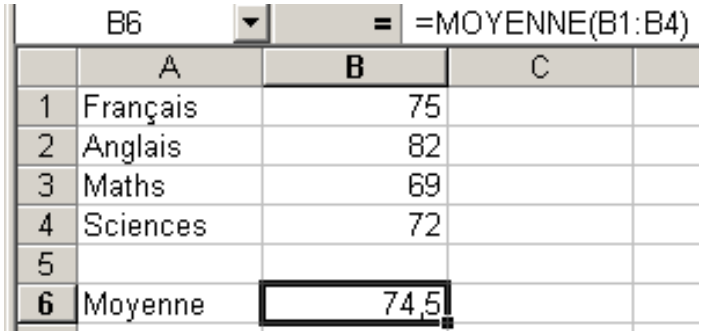


Figure 33 – Un cas (répandu) de programmation « par l'exemple » : les tableurs. La représentation graphique est centrée sur les données et ne propose aucune visualisation globale du modèle sémantique sous-jacent (les formules et les relations entre les cellules ne sont pas représentées dans la notation graphique)

Dans le cas des tableurs, le « programme » correspond bien sûr à l'ensemble des formules et des scripts de la feuille de calcul, et la « généralisation » correspond à un changement de référentiel. Alors que l'utilisateur saisit les références aux cellules dans un référentiel « absolu » (exemple : E9), l'interpréteur passe dans un référentiel « relatif » (exemple : ColonneCourante – 4, LigneCourante). Ce qui permet la réutilisation dans d'autres cas « analogues » - mécanisme de « rejeu » (Figure 34).

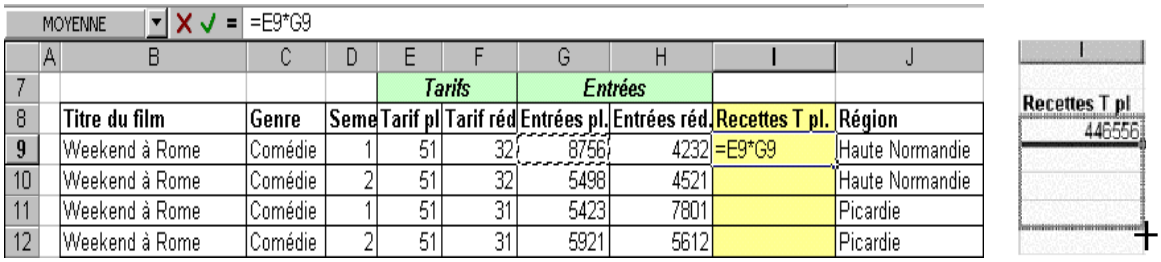


Figure 34 – Processus d'enregistrement (à gauche) et de rejeu (à droite) - cas des tableurs.

Nous qualifions l'approche utilisée ici d'« hybride », dans le sens où, si le programmeur définit les formules de façon impérative, le système repose sur un algorithme de propagation défini de façon déclarative et caché à l'utilisateur (principe de l'inférence). De même, de nombreux systèmes définis dans le champ de la recherche en IHM combinent ces différentes approches (avec exemple, sur exemple déclaratif, sur exemple impératif), tels que Pavlov (Wolber 1996) ou Gamut (McDaniel and Myers 1999).

3 Pertinence de l'approche « basée sur l'exemple » dans un cadre pédagogique

3.1 Les promesses de la programmation sur exemple

Comme on a pu le voir, l'originalité de l'approche « basée sur l'exemple » est de tenter de restaurer la manipulation directe en programmation. Elle seule adresse la dimension temporelle (le « quand ») du feedback sémantique, en promulguant un écho graphique et immédiat, qui casse le cycle de construction du programme. Pour défendre l'intérêt de la programmation « sur exemple », ses pionniers s'appuient sur trois hypothèses fondatrices :

- **Première Hypothèse :** l'évaluation en temps réel des opérations (évaluation progressive) est supposée permettre une meilleure compréhension et limiter les erreurs.

- **Deuxième Hypothèse :** l'usage d'exemples concrets est supposée faciliter la réflexion du programmeur et donc la résolution des problèmes (Comment faire ?). Le passage par des exemples concrets est vu comme :

  « …le meilleur moyen d'expliquer des concepts à étudiant (pour un professeur) et la meilleure approche (pour l'étudiant) pour apprendre. Pourquoi les exemples ne pourraient-ils pas jouer un rôle dans la programmation ? » (Lieberman 1993)

  Il devrait ainsi permettre à n'importe qui de programmer (plus) facilement.

- **Troisième Hypothèse :** la manipulation directe des objets concrets, dans l'approche « par démonstration », doit permettre de franchir le fossé cognitif séparant les représentations pragmatiques et informatiques, en se « mettant à la place de la machine ». La programmation passerait ainsi dans le domaine du « comment faire ». L'approche « par inférence » est encore plus extrême, car elle court-circuite également l'étape du « comment faire », et ne demande à l'utilisateur que des tâches de spécification (« quoi faire ») et d'évaluation.

L'arrivée à maturité technique de systèmes de programmation sur exemple dans le domaine éducatif (ToonTalk, StageCast) ou dans celui de la conception paramétrique, tout comme son succès à travers le paradigme des feuilles de calcul semble attester de l'intérêt de rétablir l'écho immédiat en programmation, et de la pertinence d'une évaluation sémantique progressive pour soutenir l'apprenabilité de l'environnement, et la création ou la mise à jour de modèles mentaux viables.

Ces recherches ont eu également l'incontestable mérite d'initier la réflexion sur ce que peut être un dialogue « naturel » en programmation. Quels rôles dans la communication doivent adopter l'utilisateur et le système ? Quels sont les modalités d'interaction et les supports les plus adaptés ? Et pour finir, quelles informations doivent être spécifiées par l'utilisateur ? Les systèmes décrits précédemment proposent une large palette de réponses à ces différentes questions, ce qui bien sûr provient en partie des différences de domaines d'application entre eux.

3.2 « Programmer sans apprentissage », « Programmer pour apprendre », ou « Apprendre à programmer » : Convergences et divergences

En effet, on peut dénombrer trois objectifs radicalement différents, parmi les systèmes présentés plus avant, en ce qui concerne l'objectif de l'utilisateur et la nature de sa tâche. On distinguera donc les cas où il s'agit :

- D'accomplir une tâche spécifique, qui nécessite de programmer telle ou telle fonctionnalité, qui n'est pas directement disponible dans les logiciels. La littérature en programmation sur exemple fourmille de tels exemples, qui sont souvent connus à travers des termes tels que *« end-user programming »* ou *« natural programming »*. Pour plus de précisions sur les contraintes liées à cette approche, que nous ne survolerons qu'assez superficiellement, le lecteur pourra notamment consulter (Girard 2000). C'est cette première approche que nous désignerons sous l'appellation *« programmer sans apprentissage »*.

- D'apprendre, à travers la programmation, certains types de raisonnements, tels que des stratégies de résolution de problèmes. Cette idée a émergé dans les années 70, avec les travaux de Seymour Papert (Papert 1980). Nous l'associons à l'appellation *« programmer pour apprendre »*.

- D'*« apprendre à programmer »,* dans le but explicite de pouvoir ensuite utiliser de façon avertie les langages et environnements de programmation existants.

Contrairement à ce que l'on pourrait imaginer de prime abord, les caractéristiques recherchées dans ces différents systèmes, qui furent souvent rassemblés dans la littérature, diffèrent notablement, car les contraintes et les prérequis de ces différents objectifs divergent souvent. Ainsi, une notation graphique comme celle de ToonTalk, qui ne correspond pas à l'idée préconçue que l'on se fait de la programmation, est sujette à soutenir la motivation d'un utilisateur final mais à affecter négativement celle d'un programmeur novice qui lui, désire programmer.

Lorsque nous parlons de « programmation sans apprentissage », le but est de réaliser une tâche, qui nécessite de programmer, sans pour autant exiger nombre des savoirs et savoir-faire associés à cette discipline. En quelque sorte, il faut faire du concepteur un « Mr Jourdain de la programmation », qui programme sans le savoir. Le pinacle de cette approche est certainement dans la programmation « par l'exemple » où le programme généré est caché, et tout particulièrement dans l'approche par inférence, où le système génère des filtres et des contraintes complexes à partir d'interactions simples de l'utilisateur.

Dans l'approche « programmer pour apprendre », en revanche, la visibilité du programme devient importante, dans la mesure où il doit être au cœur de l'activité de réflexion de

l'apprenant et la représentation de sa stratégie de « résolution de problème », et doit nécessairement être explicitement représenté, comme c'est le cas dans StageCast. L'approche par inférence probable n'a donc ici plus de raison d'être, car l'utilisateur veut avoir la main sur le processus de conception, et car la prédictibilité et la visibilité de l'état du système sont cruciaux pour l'apprentissage. En revanche, l'objectif étant de faire réfléchir, d'« apprendre à comprendre », il n'est pas plus nécessaire que dans la catégorie précédente que les formalismes et modèles informatiques sur lesquels se base l'activité de programmation soient ceux utilisés par l'industrie du développement logiciel. Pas besoin que la programmation soit « impérative », « modulaire », « orientée-objet », car l'objectif n'est **pas** d'être à terme capable d'utiliser de tels formalismes et outils. Il y a donc là deux points de convergence entre ces deux approches, qui ont donc comme point commun de promouvoir une programmation « naturelle » et « centrée sur l'utilisateur ». Le lecteur trouvera plus d'informations sur les contraintes que cela entraîne sur le « style » de programmation en consultant l'étude expérimentale de Pane (Pane 2001). Pour notre part, nous nous concentrerons sur la troisième approche, qui est l'objet de notre étude.

Dans cette dernière approche, l'objectif visé est de permettre à terme aux apprenants de devenir des utilisateurs avertis des outils de programmation de l'industrie du logiciel, en les initiant aux concepts sur lesquels ceux-ci s'appuient. Il devient dès lors impossible de se baser sur des modèles conceptuels plus « user friendly », et il est nécessaire d'apprendre aux étudiants à manipuler le programme. Celui-ci n'est donc plus seulement nécessaire en « sortie » mais en « entrée ». Il est également indispensable de répondre au problème de la réécriture et de la correction des programmes, afin de permettre à l'apprenant de s'appuyer sur un cycle de test et de validation. Pour rendre compte de cette différence fondamentale, nous réserverons l'appellation « novice programming » aux systèmes ayant pour but de faire de l'utilisateur un initié, voire un expert.

4 Contributions et conclusion.

A l'exclusion de Tinker (Lieberman 1993), un prototype d'environnement « sur exemple » conçu par Lieberman au début des années 80 et destiné à l'apprentissage du LISP, aucun système « basé sur exemple » n'a été conçu explicitement dans cette dernière optique. Et encore ce dernier système ne s'attaque pas à la programmation impérative, qui est le style de programmation le plus répandu dans l'industrie, et n'a jamais atteint, à notre connaissance, un stade fonctionnel permettant une utilisation en « conditions réelles.

Pourtant, les hypothèses de base de l'approche « sur exemple » paraissent, de prime abord, compatibles avec la problématique de l'initiation à la programmation, telle que nous l'avons résumée au chapitre 1. Casser le cycle de développement pour rétablir la « manipulation directe » semble donc une méthode pertinente pour aider à comprendre l'exécution d'un programme, tout comme l'usage d'exemples concrets tirés de domaines connus pour supporter la construction d'un modèle de base de la sémantique du processeur, même si elle a été peu explorée. De même, il nous semble que l'usage de commandes et de représentations issues du domaine de la tâche permettraient de se focaliser sur l'abstraction du comportement de la tâche, reportant la modélisation des objets du domaines par des structures informatiques à une étape ultérieure. Une telle division des difficultés serait, de notre point de vue, éminemment favorable à l'étudiant.

Notre première contribution consistera donc à proposer une synthèse des caractéristiques que devrait posséder, au vu des difficultés de l'apprentissage de la programmation que nous avons répertoriées et synthétisées au chapitre précédent, un environnement « basé sur exemple » construit dans l'optique d' « apprendre à programmer ».

4.1 Caractéristiques attendues d'un environnement d'initiation à la programmation

4.1.1 Support de l'apprentissage expérimental

Nombreux sont les travaux empiriques en psychologie cognitive (tels que (Rogalsky J. 1988), (Ben-Ari 1998), (Sleeman 1988), (Du Boulay 1989), (Pea 1986), (Perkins 1986)…) qui mettent en exergue la difficulté pour les étudiants de construire une représentation mentale viable du fonctionnement de l'exécutant-ordinateur. Cette difficulté est généralement attribuée à deux facteurs : l'absence d'un modèle « naïf » utilisable, d'une part, et l'absence de retour d'information sémantique pendant la conception du programme, d'autre part (ce que Blackwell (Blackwell 2002) appelle la « perte de la manipulation directe »). Cette absence d'évaluation progressive dans les retours d'informations du système se révèle naturellement très handicapante dans le cadre d'une exploration expérimentale, par cycle d'essais-erreurs. Il s'ensuit que le processus d'apprentissage est régulièrement interrompu en phase d'expérimentation comme d'observation, ce qui se révèle déroutant et frustrant pour des apprenants inexpérimentés dans l'usage de l'environnement lui-même.

Par conséquent, pour faciliter la construction d'un modèle mental du système, et donc l'apprentissage de la programmation, un environnement d'apprentissage devra permettre une évaluation progressive du flot de contrôle et de données, qui ne sera possible qu'en cassant le

cycle classique de conception du programme (Comment Faire → Comment Faire Faire → Comment dire → Comment Evaluer).

Deux besoins distincts peuvent être identifiés dans cette optique. D'une part un retour sémantique concis et contextualisé pendant la phase même de conception. Le but est alors de permettre à l'apprenant de vérifier en temps réel si les assertions qu'il fait mentalement sur l'état du système sont correctes : on est ici dans une tâche de supervision, dans une optique de validation. Le système doit permettre à l'apprenant d'infirmer ou de consolider une construction mentale préexistante. D'autre part, il existe un besoin pour un retour plus différé mais plus analytique, lorsque l'apprenant ne comprend pas et n'est pas en attente d'une correction de ses hypothèses mais d'une *explication* détaillée du fonctionnement du programme.

4.1.2 Une nécessaire répartition des rôles

L'étude des systèmes de programmation « basés sur l'exemple » met en évidence une certaine contradiction entre l'aide réelle à la réflexion que peuvent apporter les valeurs concrètes de l'exemple, et les compétences qu'insinuent la capacité à choisir un jeu de test pertinent : *« L'usage d'exemples concrets facilitera la réflexion du programmeur et donc la résolution des problèmes.»*

Cette hypothèse, considérée comme de bon sens par ses auteurs, ne s'est pas révélée dans la pratique une évidence absolue. En effet, dans le domaine éducatif, source de la métaphore utilisée par Lieberman, si l'utilisation systématique d'exemples peut incontestablement prétendre être une des (si ce n'est « la ») pédagogies les plus efficaces, du point de vue de l'enseignant (donc du programmeur, après transposition de l'analogie « programmeur = enseignant, ordinateur = élève »), trouver des exemples simples, signifiants et édifiants à la fois n'est pas une sinécure. Cela suppose une connaissance approfondie du comportement que l'on souhaite démontrer, afin de pouvoir mettre en valeur par l'exemple les points importants. De même, il est sans doute significatif qu'en programmation traditionnelle, il soit beaucoup plus facile pour le programmeur d'extraire une stratégie pour l'accomplissement de la tâche, puis de la traduire sous forme de programme, que de déterminer un ensemble exhaustif de jeux de test permettant d'assurer la complétude et la correction du programme.

Ainsi, il est manifeste dans les exemples de Pygmalion et de ToonTalk que le programmeur a déjà en tête, avant de commencer sa démonstration, une abstraction englobant plusieurs scénarios d'exécution de la tâche. Il y balise en effet explicitement le début et la fin des conditionnelles. Donc, si, avant de commencer la démonstration des instructions composant la conditionnelle, le programmeur spécifie que « ceci est un SI », il avait nécessairement dans la tête la structure, le plan du programme, avant de commencer sa démonstration, et même avant de choisir son exemple. De plus, la stratégie sous-jacente relève du raisonnement par récurrence, technique imparfaitement maîtrisée par de nombreux étudiants en sciences (sans parler des autres types d'utilisateurs !). Or le choix des exemples pris est induit par l'usage conscient de cette stratégie. Cette contradiction provient de ce que l'apprenti-programmeur joue un double rôle : il « enseigne » à l'ordinateur comment réaliser des procédures, mais est lui-même novice dans cette activité d'« apprendre à faire faire » et les compétences de tests de programmes sont pour lui enjeu d'apprentissage.

En conclusion, l'exemple peut certes avoir un rôle d'aide à la conception de l'algorithme, cependant, dans le cas d'apprenants complètement novices, le choix de ce jeu de test est un

engagement prématuré, dans la mesure où certaines connaissances sur le problème, sur les schémas de conception et d'implémentation des algorithmes, et sur les erreurs qui peuvent survenir, sont requises pour choisir judicieusement un jeu de test à la fois de petite taille (pour éviter des problèmes de verbosité et donc de surcharge cognitive lors de l'évaluation) et significatif.

Il apparaît donc nécessaire, dans ce cas précis, de dépasser le cadre des « environnements de développement » pour se placer dans celui des « Objets Pédagogiques Interactifs » (OPIs) (Guéraud 2005), centrés sur les activités de l'apprenant. Dans cette perspective, deux intervenants sont à prendre en compte ; d'une part l'apprenant – concepteur de programmes, qui réalise des activités expérimentales en programmation, et d'autre part, l'enseignant - concepteur d'activités pédagogiques, qui a pour rôle de mettre en place des Situations d'Apprentissage Actif (SAA) pour l'apprenant. Le concepteur pédagogique, connaissant d'avance les difficultés et les erreurs potentielles des tâches qu'il soumet à l'étudiant, est idéalement placé pour définir, à l'avance, un ou plusieurs exemples qui confronteront l'étudiant à ces difficultés.

4.1.3 Une approche « incrémentale »

La majorité des environnements actuellement utilisés pour l'apprentissage de la programmation intègrent nombre d'outils conçus dans une optique de développement, et non pas dans un cadre explicitement pédagogique. Ils ont pour unique but de fournir un ensemble d'outils d'analyse devant servir de support à la conception de programmes (le plus souvent par des experts). Une critique récurrente à l'utilisation de ce type de système dans un cadre d'apprentissage est qu'ils induiraient implicitement un type d'apprentissage se concentrant davantage sur les ressources et les outils proposés que sur les activités de l'apprenant.

Or, comme souligné dans le chapitre 1 (section 2.2), l'initiation à la programmation est rendue difficile par l'absence chez l'étudiant d'un modèle naïf viable du fonctionnement interne de l'ordinateur. Pour éviter une surcharge cognitive, il nous paraît donc important d'adopter une approche incrémentale, où les différents concepts et compétences à acquérir seraient abordés le plus progressivement possible. Pour cela, l'environnement devra permettre de simuler le comportement de l'ordinateur à plusieurs niveaux. Il est admis que proposer des représentations concrètes des concepts et manipuler ceux-ci à travers leur représentation permet aux apprenants d'appréhender ces abstractions de façon plus concrète et de confronter la conception qu'ils en ont à la simulation qui leur en est proposée, ce qui paraît pertinent dans l'objectif de permettre aux étudiants de se construire un modèle mental de l'exécution des programmes, modèle que leur expérience quotidienne avec les ordinateurs ne permet pas d'appréhender.

Contrairement à ce qui se passerait avec un environnement de développement complet, la mise en œuvre d'un simulateur sur plusieurs couches de modèles devrait permettre de focaliser l'attention de l'apprenant débutant et cibler son apprentissage sur des concepts fondamentaux, sans lui demander à cette étape une maîtrise complète de tous les détails du système réel. De la sorte, nous espérons permettre de bâtir ce socle de connaissances de base, qui manque aux apprenants débutants pour appréhender la programmation.

4.2 Problématique de Recherche

L'exploration de cette approche, et la validation de sa pertinence constituent le cœur de notre travail de thèse. Pour cela, nous procédons en trois étapes.

Premièrement, nous proposons un évaluation synthétique *a priori* des différentes techniques d'interaction utilisables en programmation basée sur l'exemple, en adaptant une grille d'évaluation cognitive des notations. Celles-ci sont en effet nombreuses ; ne serait-ce que dans le présent chapitre, nous avons ainsi passé en revue des implémentations de la programmation basée sur l'exemple qui s'appuient sur l'édition textuelle, graphique, ou l'espionnage des actions de l'utilisateur, et qui peuvent utiliser des représentations symboliques, métaphoriques, ou pragmatiques. Le but de cette démarche est donc de comprendre le contexte de pertinence de chaque technique, d'un point de vue pédagogique, afin de déterminer la combinaison de techniques d'interactions la plus pertinente pour un environnement d'apprentissage basé sur une approche expérimentale centrée sur l'exemple, tel que nous l'avons défini dans la section précédente.

Deuxièmement, nous présentons l'environnement MELBA (Metaphor-based Environment to Learn the Basics of Algorithmics). Nous justifions la pertinence pédagogique de ses différents composants, en relation avec le cahier des charges précédemment défini. Nous mettons en exergue ce qui différencie cet environnement d'apprentissage, dont le but est de supporter la compréhension des concepts algorithmiques et des relations entre eux, d'un environnement destiné à supporter la programmation de composants logiciels. Nous expliquons comment concevoir pour cet environnement un exercice mettant en place une SAA.

Enfin, alors que cette approche de conception « basée sur l'exemple » n'a jusqu'ici été validée que par des études de faisabilité, nous proposons d'évaluer de façon rigoureuse sa pertinence dans le cadre de l'apprentissage de la programmation, à travers une série d'expérimentations en milieu réel et en milieu contrôlé.

Chapitre 3

Notations et techniques d'interaction : analyse cognitive pour des apprenants novices

Résumé. *Dans le chapitre précédent, nous avons passé en revue de multiples implémentations du paradigme de programmation « basée sur exemple » qui s'appuient sur l'édition textuelle, l'édition graphique, ou l'espionnage des actions de l'utilisateur, et qui peuvent utiliser des représentations symboliques, métaphoriques, ou pragmatiques. Nous avons également défini « un cahier des charges » répertoriant les caractéristiques d'un environnement d'apprentissage basé sur une approche expérimentale centrée sur l'exemple.*

Dans ce chapitre, notre contribution consistera à déterminer le contexte de pertinence de chaque technique d'interaction, d'un point de vue pédagogique. Nous nous appuyons dans notre analyse sur un outil d'évaluation de notations, issu du champ de l'ergonomie cognitive, le « Cognitive Dimensions Framework » (Green 1989). Nous détournons légèrement cet outil de son utilisation « classique », qui est d'évaluer une notation ou un langage spécifique, en l'utilisant pour analyser d'un point de vue cognitif, et de façon globale, les différents types d'interactions et de visualisations de la littérature.

Cette analyse nous permet de relier les techniques d'interaction employées aux savoirs et aux savoir-faire décrits dans le premier chapitre. A partir de là, nous sommes à même de définir la combinaison de techniques d'interaction la plus pertinente pour réaliser le cahier des charges du chapitre précédent, ce qui ouvre la voie à l'implémentation de ces techniques et concepts dans un EIAH innovant.

1 Caractérisation cognitive des techniques d'interaction

Dans le premier chapitre, nous avons mené une revue et une analyse des difficultés de l'apprentissage de la programmation. Il en ressort que la nature et la qualité du feedback fourni par l'environnement utilisé pour l'apprentissage ont une importance centrale, que ce soit pour l'acquisition des concepts ou de la construction des liens entre ces concepts (schémas de connaissance).

Dans ce chapitre, nous nous proposons de caractériser le support à la compréhension que peuvent apporter les techniques d'interaction connues, en commençant par le niveau syntaxique pour aller vers le niveau pragmatique. Pour cela, nous utiliserons la métrique des « Dimensions Cognitives » (Green 1989) présentée dans la section suivante. Cette métrique nous permettra de caractériser chacune des techniques considérées, et de décrire les contreparties qu'induisent l'usage de ces techniques.

1.1 Les « Dimensions Cognitives » : un cadre d'évaluation

Les dimensions cognitives des notations (DC) forment un ensemble de 14 mesures de l'utilisabilité et de l'utilité de notations. Les dimensions de Green sont dites « cognitives » car elles se focalisent sur les aspects de l'utilisabilité qui rendent l'action et l'apprentissage compliquées et difficiles pour des raisons qui ne sont pas d'ordre *physique* (la taille des boutons, la distance entre des composants… leur approche est donc perpendiculaire à celle de méthodes telles que Keystroke/KLM) (Dix, Finlay et al. 1993), ni *esthétique*.

Ce cadre d'évaluation fut introduit à partir de la fin des années quatre-vingts par Green (Green 1989), qui mettait en exergue la généralité de cette approche. Elle fut appliquée de façon détaillée par Green et Petre (Green 1996) pour analyser les avantages et les contraintes des notations de la « programmation visuelle iconique » par rapport aux langages de programmation plus conventionnels. Cette évaluation reste la présentation la plus détaillée de ces métriques publiée jusqu'ici.

Ces « dimensions » permettent de définir un profil de la représentation évaluée. Chaque caractéristique n'est ni « bonne » ni « mauvaise » intrinsèquement, car le profil idéal dans chacune d'elles peut varier selon le type de tâche et le type d'utilisateur, ou être contradictoire avec le profil idéal dans une autre dimension. Elles ont pour but d'aider des non-spécialistes de l'ergonomie des logiciels à évaluer l'utilisabilité et l'utilité d'un système d'informations, en préférant une vue d'ensemble à une analyse plus longue et approfondie.

La pertinence de cet outil pour notre analyse tiens à deux caractéristiques. D'une part, le caractère purement cognitif de l'approche permet de faire facilement un parallèle avec les caractéristiques de l'apprenant, et avec les difficultés du domaine, qui sont elles-même de nature cognitive. D'autre part, le but de notre démarche est d'évaluer les *interactions*, et non pas une *interface* spécifique (contrairement à la plupart des évaluations ergonomiques qui nécessitent donc un outil de précision, prenant en compte des caractéristiques sensorielles et/ou motrices). Il s'ensuit que le grand degré d'abstraction de ces métriques les rend plus adaptées pour une analyse de plus haut niveau. Enfin, si les dimensions cognitives des notations ont été au départ pensées pour évaluer des langages, les différentes métriques

peuvent facilement s'appliquer à des représentations graphiques. Elles ont qui plus est l'avantage de pouvoir prendre fortement en compte l'aspect temporel de l'interaction.

Dans les sections suivantes, nous définirons dans un premier temps chaque dimension cognitive. Puis, nous procéderons à l'analyse des différentes techniques d'interaction en programmation à travers ces métriques, en mettant en relation leurs caractéristiques avec les schémas de connaissance en programmation et les styles d'apprentissages des étudiants.

1.2 Gradient d'Abstraction

La barrière d'abstraction définit le nombre de nouvelles abstractions qui doivent être comprises avant de pouvoir maîtriser le système. Un avantage classique de l'apprentissage assisté par l'ordinateur (EAO) est de posséder un gradient d'abstraction fort : les activités qui y sont simulées peuvent faire appel à un niveau de détail croissant, passant progressivement de représentations synthétiques issues du modèle conceptuel naïf de l'utilisateur (qui dans le cas qui nous occupe sont en fait des modèles de la tâche, permettant de raisonner au niveau du *« Comment Faire »*) à des représentations beaucoup plus précises, mettant en œuvre un corpus de concepts bien plus large (il s'agit dans notre cas de modèles *informatiques* complets, qui permettent le raisonnement au niveau algorithmique – *« Comment Faire Faire »*). Un fort gradient d'abstraction implique la volonté de proposer un degré de modélisation variable, et de s'adapter ainsi à la progression de l'apprenant.

D'autre part, certains systèmes interactifs permettent à l'utilisateur de définir ses propres abstractions ; on peut ainsi les qualifier de :

- « Abstractophile », lorsqu'ils imposent, pour pouvoir être utilisés, la définition par l'utilisateur de nouvelles abstractions. Les langages informatiques de *spécification formelle* tels que B, ou Z, sont contenus dans cette classe.
- « Tolérant à l'abstraction », lorsqu'ils permettent à l'utilisateur de définir de nouvelles abstractions, mais qu'ils peuvent être déjà utilisés « tels quels ». Les traitements de textes, qui permettent à l'utilisateur de définir des *styles* qui lui sont propre, mais ne le nécessitent pas absolument, sont une instance de cette classe.
- « Abstractophobe », lorsqu'ils ne permettent pas à l'utilisateur de définir de nouvelles abstractions (et en contiennent le minimum possible). Les tableurs sont une instance de cette classe de systèmes.

Les abstractions rendent les notations plus ***concises*** et peuvent parfois améliorer leur ***correspondance au domaine***. Elle représentent un investissement à long terme : en changeant les notations maintenant, soit elles seront plus faciles à utiliser plus tard, soit elles seront plus génériques et réutilisables, soit elles réduiront la viscosité et donc rendront les productions plus faciles à modifier. Parmi les abstractions définies par l'utilisateur lui-même, on peut distinguer deux classes :

- Persistantes, comme par exemple les macros, les feuilles de styles, hiérarchies de classes.
- Transitoires. Elles ont alors une durée de vie prédéfinies, en général une seule utilisation, comme par exemple : une sélection groupée de fichiers dans l'explorateur, ou les expressions à rechercher et/ou remplacer dans tout un document.

L'usage d'abstractions se révèle en contrepartie souvent coûteux. Penser en terme d'abstractions est difficile et ne survient que tardivement dans l'apprentissage d'un nouveau domaine de savoir, ou d'une nouvelle discipline. Des systèmes qui renforcent l'abstraction créent une barrière à l'apprentissage, qui peut se révéler insurmontable pour certains étudiants. D'autre part, l'introduction de hiérarchies d'abstractions induit un coût en maintenance (qui peut augmenter la viscosité du système) : par exemple, si on modifie la signature d'une méthode *m* d'une classe, cette modification doit être reportée dans toutes les classes filles qui la redéfinissent, et dans toutes les méthodes qui l'utilisent. Cela coûte donc du temps et de la main d'œuvre, et est susceptible de générer des erreurs.

1.3 Correspondance au domaine

La métrique de « correspondance au domaine » désigne la capacité de l'environnement à minimiser le gouffre cognitif entre la structure de connaissances sur la tâche, que possède l'étudiant, et le modèle du système. Elle indique que l'environnement permet à l'apprenti programmeur de raisonner à partir de modèles de la tâche, et correspondent donc également à une réponse à la question « quoi ». Un système ayant une forte correspondance au domaine peut également supporter des modélisations plus abstraites (à condition d'avoir un fort gradient d'abstraction).

Par exemple, les notations du langage graphique iconique LabVIEW entretiennent une correspondance étroite au domaine des ingénieurs électroniciens auxquels il est destiné (Figure 35), de même pour l'environnement de la « tortue » du langage LOGO avec le domaine du dessin. Le gradient d'abstraction de ces deux langages étant par ailleurs assez élevé, on peut faire des programmes LabVIEW qui ne font pas appels à des instruments physiques, et des programmes LOGO où on ne se sert pas de la tortue

```
pour poly.spirale :taille :angle
 si :taille>205 [stop]
 avance :taille
 droite :angle
 poly.spirale somme :taille 5 somme :angle 0.12
end
```

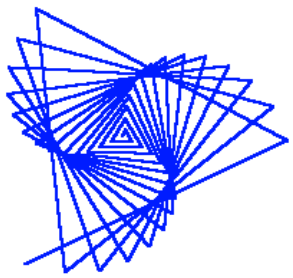


Figure 35 – Exemple de programmation avec la tortue LOGO. Les capacités du processeur touchent à un domaine connu : le dessin.

1.4 Cohérence

Un langage ou un environnement est « cohérent » quand la similarité sémantique implique une similarité syntaxique dans la notation. Il existe en particulier, deux type de cohérence :

- La cohérence « interne » touche les différentes fonctions du langage. Par exemple, si celui-ci possède deux fonctions manipulant les chaînes de caractères (copie et concaténation), qui ont pour paramètre la chaîne (ou sous-chaîne) que l'on souhaite copier et la chaîne destination, l'ordre des paramètres devra être identique pour assurer la cohérence (Mc Iver 2001):

```
copyString(source,destination)
catString(source,destination)
```

Dans le cas de systèmes de programmation graphique, la cohérence « interne » de la notation fait fortement écho au critère d'homogénéité de Scapin et Bastien (Scapin 1986).

- Par ailleurs, on distinguera un deuxième niveau de cohérence, externe, entre le langage et ses pairs, ou entre le langage et les connaissances préalables de l'apprenant. Par exemple, les notations de langages comme Perl, Python ou Java sont cohérentes avec celles de C, tout comme il y a cohérence de notation entre Ada et Delphi ou Pascal. Concernant la cohérence avec les notations de l'utilisateur, un (mauvais) exemple célèbre est l'opérateur « == » du langage C, notoirement incohérent avec la représentation naturelle (« = »). En l'occurrence, cette incohérence a de plus un impact sur ***l'incitation aux erreurs***, car « = » a aussi un sens en C….

1.5 Prolixité

La prolixité, ou verbosité, correspond à la longueur des commandes dans la notation. Ainsi, COBOL est un langage verbeux :

```
MULTIPLY A BY B GIVING C
```

Par opposition, Forth est un langage extrêmement concis ; ainsi la commande permettant d'aller à la ligne prend la forme:

```
.
```

La prolixité est une autre dimension dont le profil peut être aussi bien bon que mauvais, selon l'activité. Différents niveaux de prolixité peuvent être appropriées selon la situation : par exemple, une commande souvent tapée (et donc susceptible d'être facilement enregistrée, par répétition) se devra d'être concise – pour éviter de surcharger la mémoire de travail[1] – alors qu'une commande moins souvent tapée devra être plus longue, pour augmenter son expressivité et la lisibilité générale.

Par extension, dans un contexte d'interaction de type « WIMP[2] », la prolixité pourra désigner le nombre d'interactions associées à une action (un système très prolixe proposera donc beaucoup de commandes de bas niveau). La prolixité est naturellement liée à la *viscosité* du système.

[1] Des études sur la mémoire à court terme ont en effet montré que les performances en calcul mental sont supérieures chez les enfants dont le langage associe des noms courts aux nombres. Cela impliquerait que garder des descriptions longues en mémoire handicaperait l'utilisateur dans des activités gourmandes en mémoire de travail, telle que la conception exploratoire.

[2] Windows, Icons, Menus and Pointing device

1.6 Incitation à l'erreur

L' incitation à l'erreur signifie que la notation tend à générer des mécanismes de fausse consonance ou de dissonance cognitive de par le choix de ses symboles (par exemple, « = » pour l'affectation et « == » pour l'égalité dans C, ou encore les identifiants I et O dans le langage Forth). Cela regroupe aussi les problèmes de boîtes de dialogue (par exemple associer « Entrée » au bouton « OK » dans toutes les boîtes sauf une où il se réfère au bouton « Annuler »). Autre exemple, la déclaration automatique de nouvelles variables lorsqu'un nouvel identifiant valide est rencontré par l'interpréteur – Prolog, BASIC, PERL – peut causer des erreurs difficile à comprendre au moment de l'exécution…

1.7 Dépendances cachées

Une dépendance cachée est une relation entre deux composants, telle que l'un d'entre eux dépende de l'autre, mais que cette dépendance ne soit pas totalement visible. Les références peuvent être :

- *Unilatérales* ou *symétriques*
- *Locales* ou *distantes*. Un pointeur local ne donne d'information que sur l'extrémité du lien, alors qu'un pointeur distant donne des informations sur toute la hiérarchie.
- *Explicites* ou *cachées*.

Par exemple, les liens entre cellules d'un tableur sont problématiques de par leur nature *locale* et *asymétrique* : il est impossible d'y centraliser une information telle que « quelles cellules utilisent la valeur d'une cellule donnée ? » car seule l'*extrémité* du lien y est apparente) – voir Tableau 2.

	A	B	C	D	E	F
1	Marchandise	Prix Unitaire	nombre	Prix HT	TVA	Prix TTC
2	CD	12,99 €	2	=B2*C2	= 0,055*D2	=D2+E2

Tableau 2 – Dépendances cachées locales et unilatérales dans une feuille de calcul.

Cela entraîne un coût de recherche (pour suivre « manuellement » les liens) qui a un impact direct sur l'incitation aux erreurs, la difficulté des opérations mentales, la charge cognitive et la viscosité. Un exemple de dépendances cachées en programmation réside dans la difficulté, pour l'utilisateur, d'appréhender la combinaison des règles de déduction (dans un paradigme déclaratif comme pour Prolog) ou des règles d'action (dans un paradigme plus événementiel comme pour HANDS, StageCast Creator, AgentSheets). Si chaque règle indépendamment est facile à comprendre, le fonctionnement global du système devient difficile à appréhender, de par sa forte modularité, déclenchant l'apparition de nombreuses erreurs dues à ce que l'on appelle en informatique les « effets de bord »…

1.8 Charge Cognitive

Comme son nom l'indique, cette dimension permet de mesurer la difficulté des opérations mentales que l'utilisateur doit réaliser, et la charge de travail que celles-ci induisent. La

densité informationnelle de la notation a également une importance centrale pour cette mesure. Typiquement, la gestion « dynamique » de la mémoire en C est réputée pour solliciter fortement les ressources cognitives du programmeur au moment de décider où allouer et désallouer (à cause de la ***prévisualisation forcée***, voir section suivante…).

1.9 Engagement Prématuré et Prévisualisation Forcée

L'engagement prématuré se produit lorsque l'ordre des actions imposé par le système force l'utilisateur à prendre des décisions sans avoir toutes les informations requises. Cela peut résulter de contraintes spatiales ou temporelles, et est très fréquent avec les langages utilisant des notations à base de graphes – voir la Figure 36 .

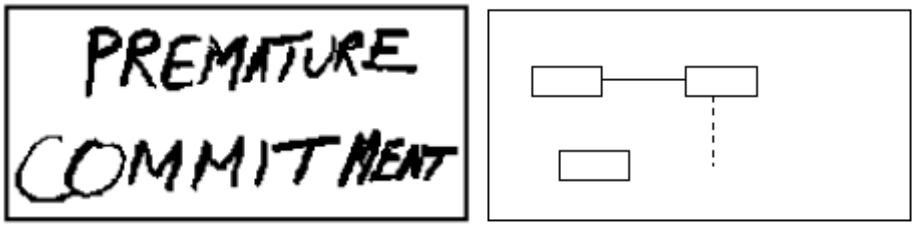


Figure 36 – Deux exemples d'engagement prématuré : à gauche, les contraintes spatiales forcent l'utilisateur à diminuer la taille des lettres à la fin des deux mots ; à droite, dans un éditeur de schémas Entités – Association, le connecteur en pointillé ne pourra être rajouté que postérieurement à l'Entité à laquelle il se réfère (contrainte temporelle).

Dans cette même dimension, on inclut la prévisualisation forcée (« enforced lookahead »), qui arrive lorsque les contraintes du système obligent l'utilisateur à préparer mentalement toute une série d'actions qui sollicitent fortement sa mémoire de travail – voir Figure 37. Par conséquent, la prévisualisation forcée a un fort impact sur la charge cognitive.

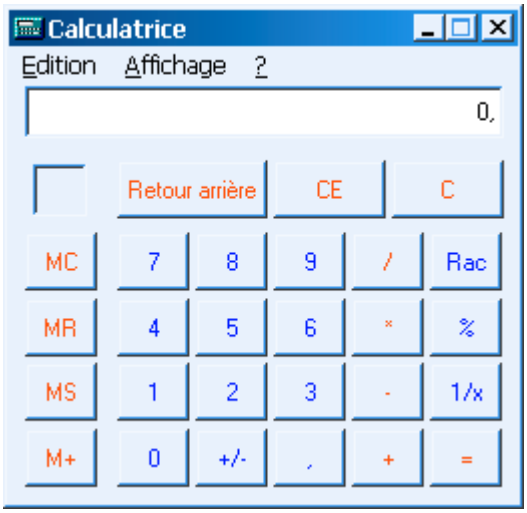


Figure 37 – Un cas de « prévisualisation forcée » : le calcul sur la calculatrice ci-dessus de l'expression `(1.2 + 3.4 – 5.6)/((9.7 – 6.5) * 4.3)`

1.10 Evaluation progressive

Cette dimension correspond à la latence et la granularité du feedback ; une évaluation progressive favorise la compréhension de l'état du système tout en ménageant la charge de travail de l'utilisateur. En l'occurrence, dans la programmation avec exemple, l'évaluation progressive est maximisée (le programme est évalué « à la volée »…), alors que le cycle « édition – compilation – test » retarde le retour sémantique. Les interpréteurs LISP ou TCL représentent un niveau d'interactivité intermédiaire. En général, une évaluation plus progressive va de pair avec une manipulation plus « directe ».

1.11 Degré d'engagement

Le degré d'engagement exprime l'importance des actions de l'utilisateur, en fonction des conséquences que celles-ci peuvent avoir. Ainsi, lorsque les actions de l'utilisateur sont réversibles, le degré d'engagement est bas, alors que quand elles pilotent un système critique (train, avion, centrale nucléaire…) il est au plus haut. Dans le cadre de l'apprentissage de la programmation, un environnement de TP serveur / terminaux impose un degré d'engagement fort : une erreur critique dans un programme d'un étudiant étant susceptible d'avoir une conséquence sur le travail de tous les étudiants travaillant sur le serveur à ce moment là (si le serveur doit être redémarré toutes les modifications non sauvegardées de tous les étudiants de toutes les classes travaillant dessus à l'instant t seront perdues !).

1.12 Expressivité

Une forte expressivité implique que le rôle des opérateurs soit explicite. Cette métrique exprime la facilité avec laquelle un utilisateur novice peut inférer le rôle d'une commande à partir de son nom ou de sa forme. Par exemple, l'opérateur de pointeur « * » de C est peu expressif… A contrario, les qualificateurs de paramètres « in » et « out » en Ada ont été conçus dans une logique d'expressivité (du point de vue des auteurs, et pour un utilisateur anglophone…).

1.13 Notation Secondaire

Comme son nom l'indique, cette dimension permet d'évaluer la capacité du système à permettre à l'utilisateur d'annoter ses schémas, ou programmes, dans un formalisme autre que la notation principale. Un avantage reconnu des annotations est de faciliter la relecture et la réutilisation ultérieure par un tiers ou par l'auteur lui-même. Par exemple, l'indentation, ou les commentaires dans les programmes textuels participent de cette dimension.

1.14 Viscosité

La viscosité d'un système exprime la résistance qu'il oppose au changement ; sur un système visqueux le coût de petites modifications se révèlera élevé. Green distingue deux types de viscosité :

- Viscosité par répétition : lorsqu'une tâche particulière sur la structure d'informations requiert un grand nombre d'opérations de bas niveau. Par exemple, passer de

l'orthographe britannique à l'orthographe américaine sans l'aide d'un correcteur automatique.

- Viscosité par effets de bord : lorsqu'un changement de structure impose de faire de nombreuses autres modifications pour maintenir la cohérence. Par exemple, insérer une nouvelle figure dans un document impose la renumérotation de toutes celles qui suivent, et de toutes les références dans le texte, ce qui est extrêmement fastidieux en l'absence d'outil automatique.

Le Tableau 3 illustre un exemple de viscosité par effets de bord : si on souhaite déplacer le cours de Mr Adams aux élèves de première année dans le créneau 10h-11h, la totalité de la structure doit être modifiée, pour maintenir la cohérence.

Cours	9h – 10h	10h – 11h	11h – 12h
1° Année	Mr Adams	Mme Burke	Mme Cooke
2° Année	Mme Cooke	Mr Davis	Mr Adams
3° Année	Mme Burke	Mr Adams	Mr Davis

Tableau 3 – Viscosité par effets de bords dans un gestionnaire d'emplois du temps.

La viscosité casse le déroulement de la pensée. Elle dissuade du bricolage, de la prise des risques, ou même de la conception exploratoire (par cycles essai-erreur), car le coût en temps des modifications serait trop important. Par là-même, elle encourage la planification et la réflexion, et réduit ainsi potentiellement le risque d'erreurs catastrophiques. La viscosité par répétition est un outil couramment utilisé pour rendre difficile une action dangereuse – ce qui a pour effet secondaire de diminuer le degré d'engagement. Par exemple, le typage « fort » en Ada participe de cette logique (en opposition avec les conversions implicites en C).

Le nombre de dépendances (cachées ou pas) est proportionnel à la viscosité par effet de bord, et dans toutes les notations basées sur des graphes, elle prendra une importance cruciale. Ainsi, si on supprime une cellule dans un tableur, toutes les cellules se référant à celle-ci peuvent prendre des valeurs incohérentes… de même, la modification d'un graphe de programme LabView est rendue pénible à cause des liens entre les icônes qui doivent être fréquemment reconnectées.

1.15 Visibilité et Juxtaposition

La visibilité détermine la capacité du système d'information à fournir un écho fidèle et complet de son état interne, ou de sa structure d'information. La juxtaposition est la capacité à présenter plusieurs informations (en général complémentaires) simultanément côte à côte. Un défaut de visibilité peut impliquer soit qu'une information n'est pas présente, soit qu'elle est difficile d'accès.

Par exemple, dans le cas d'un annuaire téléphonique classique, il est difficile d'accéder à la personne correspondant au numéro XX XX XX XX XX, d'où la floraison de multiples services d'annuaires dits « inversés » : l'information est bien présente dans le système de notations, mais la structuration des données en interdit (en pratique, car trop malaisé) l'accès. De nombreuses recherches en Interaction Homme-Machine portent, encore actuellement, sur

la visualisation (de grandes quantités) d'information, on peut citer par exemple (Plaisant 2004).

L'importance de la visibilité est évidente lorsqu'il s'agit de rechercher un élément spécifique d'information. Cependant, bien que ce soit moins apparent, elle est tout aussi importante pour permettre à l'utilisateur de prendre du recul et de considérer la structure de l'information (par exemple, les programmeurs experts n'examinent pas les programmes dans le détail ligne à ligne, mais les parcourent par bloc). Une faible visibilité empêche ce parcours structuré.

L'importance de la juxtaposition est souvent sous-estimée. Cependant, un traitement cohérent d'informations similaires n'est possible qu'en comparant les instances des données, ce qui nécessite la juxtaposition de celles-ci. De façon plus subtile, la solution à un problème requiert souvent l'étude et l'adaptation d'une solution à un problème voisin. L'absence de vues juxtaposées sous-entend que chaque problème est résolu indépendamment de tous les autres.

1.16 Synthèse

Ces différentes définitions sont résumées dans le Tableau 4 ci-dessous. La description de la colonne de droite fait référence à une forte valeur dans la dimension concernée.

Dimension Cognitive	Description
Gradient d'Abstraction (« Abstraction Gradient »)	Les différents types d'abstraction disponibles.
Correspondance au Domaine (« Closeness of Mapping »)	Le degré de dépendance de la notation du système d'information par rapport au domaine d'application.
Cohérence (« consistency »)	La similarité sémantique implique une similarité syntaxique.
Prolixité (« diffuseness »)	La notation est « verbeuse ».
Incitation à l'erreur (« Error Proneness »)	La notation choisie est susceptible de provoquer des erreurs.
Charge cognitive (« Hard mental operations »)	Le système d'information impose à l'utilisateur une forte utilisation de ses ressources cognitives.
Dépendances Cachées (« Hidden dependencies »)	Certains liens entre entités ne sont pas visibles.
Engagement Prématuré (« Premature Commitment »)	De fortes contraintes sur l'ordre des actions de l'utilisateur l'obligent à prendre une décision sans avoir accès à l'information
Evaluation Progressive (« Progressive evaluation »)	Le résultat des actions est accessible à tout moment (le système est « WYSIWYG »)
Degré d'engagement (« Provisionnality »)	Le degré d'engagement de l'utilisateur dans ses actions. Celles-ci sont elles réversibles ? Définitives ? Dangereuses ?
Expressivité (« Role-Expressiveness »)	Le rôle d'un composant est facile à identifier.
Notation Secondaire (« secondary notation »)	Le système permet de rajouter des informations contextuelles sous forme d'annotations (ex. commentaires en programmation)
Viscosité (« Viscosity »)	Une grande résistance du système aux changements.
Visibilité (« Visibility »)	La capacité à voir les composants facilement.

Tableau 4 – Les quatorze dimensions cognitives des systèmes d'informations : une synthèse.

2 Evaluation et Classification des Techniques d'Interaction

Après avoir présenté les métriques qui permettent d'évaluer l'acceptabilité d'un environnement d'apprentissage de la programmation, nous présentons dans cette section l'évaluation de plusieurs environnements pour programmeurs, et tâchons d'en déterminer les caractéristiques générales qui conditionnent leur qualité en tant qu'outil éducatif.

Il est difficile de préconiser un profil « idéal », dans la mesure où certaines caractéristiques peuvent vite devenir incompatibles. Par exemple, si on souhaite diminuer les dépendances cachées, on tendra à modéliser les relations qui existe entre les objets de façon explicite (comme en adoptant une notation « flot de données » - exemple : LabVIEW). Mais la représentation explicite de ces relations aura alors un coût aussi bien en viscosité (maintenir la cohérence de la notation sera plus délicat) qu'en prolixité et en charge cognitive (plus grand nombre de détails affichés), en visibilité (encore une fois à cause des détails : l'arbre cache la forêt), ou encore en engagement prématuré (ces relations devant être saisies obligatoirement après les objets qu'elles connectent). Il s'ensuit que la conception devient avant tout une affaire de compromis.

Nous émettons de plus l'hypothèse que l'impact des préférences du concepteur en terme de dimensions cognitives peut varier en fonction du style d'apprentissage de l'apprenant. Ainsi la **visibilité**, l'**expressivité** et les **dépendances cachées** s'associent-elles plus à une démarche par *connotation* ; l'étudiant, s'appuyant sur l'observation des échos du système pour construire sa compréhension des concepts, a un plus grand besoin d'une interface représentant en haute fidélité l'état du système. Des observables de piètre qualité l'handicapent dès lors dans son apprentissage. A contrario, un fort **degré d'engagement**, une **prévisualisation forcée** importante, une notation **incitant à l'erreur** ou une **viscosité** trop grande seront de nature à troubler des étudiants au style *dénotatif*, très dépendants de leur expérimentation, et à favoriser la *passivité* au détriment du *bricolage*. Comme le résume le Tableau 5, les dimensions cognitives les plus critiques diffèrent grandement selon le profil concerné.

Style d'apprentissage	Dimensions cognitives
Dénotatif	Degré d'engagement, Viscosité, Engagement prématuré, Juxtaposition, Evaluation progressive.
Connotatif	Visibilité, Juxtaposition, Evaluation progressive, Gradient d'abstraction, Charge Cognitive.

Tableau 5 – Résumé des dimensions cognitives des outils et notations les plus cruciales, selon le style d'apprentissage.

En revanche, l'**évaluation progressive** ou la **juxtaposition** sont des dimensions qui sont de nature à profiter grandement à tous lorsqu'elles sont renforcées. En effet, l'aspect lacunaire et la fragilité des modèles naïfs des apprenants commande une réponse rapide, pour faciliter l'évaluation de l'état du système et empêcher la génération de modèles mentaux erronés. C'est pourquoi il nous paraît préférable de proposer un feedback automatique, et non pas « à la demande », et de rendre l'évaluation la plus « progressive » possible, en tendant vers l'écho actif (réponse immédiate). Pour le niveau « articulatoire » de la syntaxe, un écho pro-actif nous paraît même préférable (pour éviter que le retour du système sur les erreurs sémantiques liées à des conceptions erronées, et donc sujettes à provoquer un apprentissage par

compréhension, soient bruitées par des messages d'erreurs d'un niveau articulatoire qui n'est pas l'objet premier de l'apprentissage).

Ne pouvant bien sûr être exhaustif, nous décomposons la palette (trop large pour le cadre de ce manuscrit !) des outils existants en grandes classes de systèmes et de techniques. Nous évaluons ensuite chacune en fonction de ses caractéristiques intrinsèques, et pour chaque classe, nous illustrons cette évaluation à travers les caractéristiques particulières de langages ou de systèmes connus.

2.1 Techniques d'édition du programme

2.1.1 Langages à syntaxe textuelle

La majorité des langages de programmation se basent sur une syntaxe textuelle, et un cycle d'interaction de type : « édition – compilation – test ». L'interaction avec le système se fait via un éditeur de texte. Elle se caractérise par :

- Un **engagement prématuré** très faible ; en effet, l'ordre d'édition du programme n'est pas contrainte, le programmeur peut accéder facilement à toute partie du code.

- Une **prévisualisation forcée** variant d'assez importante à très importante, liée à une **évaluation progressive** variant de nulle à médiocre (selon le degré d'interactivité de l'environnement). Ainsi certains environnements de programmation comme JBuilder effectuent une vérification syntaxique interactive, mais obligent toujours l'utilisateur à une prévisualisation importante du point de vue sémantique. Dans la majorité, ni l'évaluation syntaxique ni l'évaluation sémantique ne sont interactives.

- Une **visibilité** globalement bonne à très bonne. Trois raisons expliquent ce phénomène : d'une part le texte structuré prend peu de place à l'écran par rapport aux notations graphiques (le nombre d'éléments visibles à chaque instant est donc plus important) ; d'autre part, la forme graphique permet d'avoir facilement une vue d'ensemble, en favorisant la structure sur le contenu. Enfin, les utilisateurs (même novices) sont entraînés très tôt à extraire le sens général d'un texte structuré, et la syntaxe des langages de programmation tire profit de cet entraînement à la lecture, par des analogies aux opérateurs de structure du discours du langage naturel (au risque d'une fausse consonance). Il est donc, globalement facile de comprendre le sens général d'un programme textuel, impératif et structuré[1].

- Cette lisibilité accrue est également due à un fort **gradient d'abstraction** (tous ces langages permettent la définition d'actions d'un plus haut niveau sémantique appelées « procédures ») qui permet de factoriser les actions élémentaires en un nouvel élément syntaxique plus **expressif**.

- Parallèlement, le support accru des **notations secondaires** via un mécanisme de commentaires qui permet d'une part d'exprimer le « comment faire » quand la stratégie

[1] Cet avantage a d'ailleurs un important effet pervers, car rendre les corrigés de l'enseignant globalement compréhensibles, voire évidents, peut être dévastateur sur la confiance en soi d'un étudiant expérimentant de grandes difficultés dans la conception des programmes.

sous jacente n'est pas explicite, et d'autre part de lier le faire (univers de la tâche) au faire faire (univers de l'informatique), permet d'encore accroître la lisibilité.

- En revanche, cette facilité à présenter l'idée et le plan général qui sous-tendent le programme se paye, en l'occurrence, par un grand nombre de **dépendances cachées** unilatérales.

- Celles-ci, couplées à une forte prévisualisation forcée sollicitent fortement les ressources cognitives de l'utilisateur (critère de **charge cognitive**).

2.1.2 Langages graphiques à base d'icônes

Dès l'apparition des systèmes graphiques (Pygmalion, 1975), certains concepteurs ont commencé à proposer des langages totalement graphiques, basés sur l'utilisation d'icônes pour représenter les opérateurs et les données, et de liens graphiques figurant le flot de donnée ou de contrôle (Figure 38).

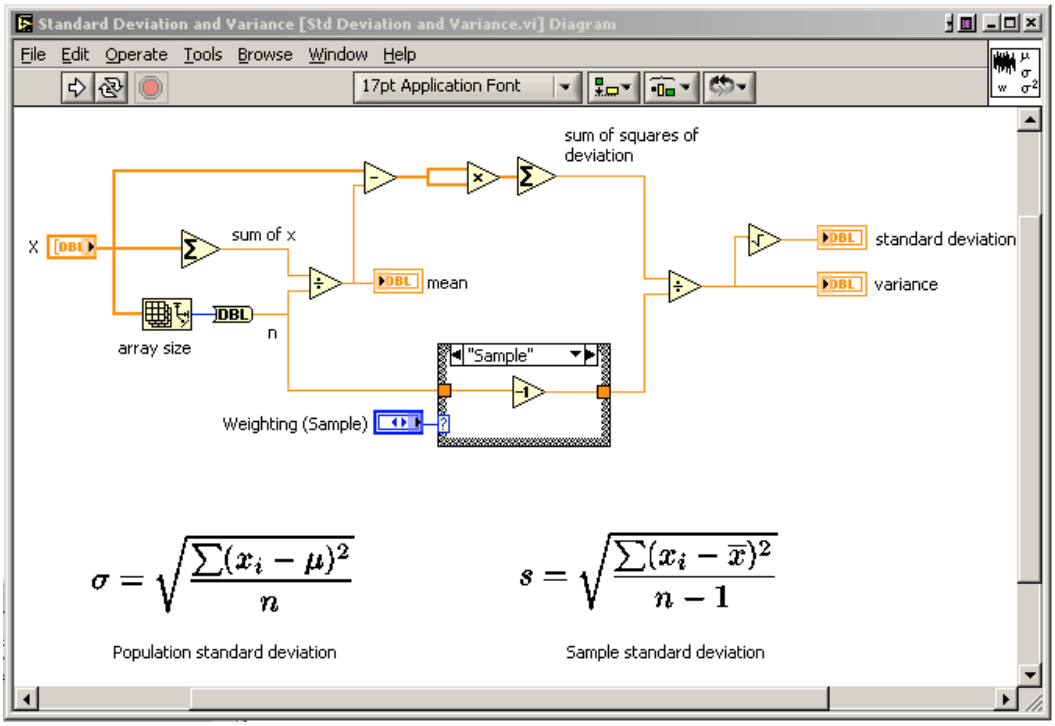


Figure 38 – Un exemple de langage visuel iconique : « G » de LabVIEW et son IDE.

Pour ses pionniers, l'utilisation d'images dans la programmation devait se généraliser et certains, comme Shu (Shu 1986), en analysent les raisons ainsi :

- En règle générale, et pas seulement en informatique, les gens préfèrent les illustrations au texte ;

- Une image est capable de véhiculer davantage de sens qu'un mot, voire qu'une phrase. Cela lui confère un caractère de concision que ne possèdent pas les signes littéraires ;

- Les images sont universelles, elles peuvent être appréhendées de la même manière par des personnes de nationalité, de culture ou d'âge différents.

Cependant, les langages visuels, même si ils ont depuis effectué des percées dans certains domaines, ou démontré leur efficacité dans des contextes spécifiques, sont restés minoritaires par rapport aux langages textuels. Les premiers arguments de Shu apparaissent maintenant comme un jugement rapide, parfois naïf, et c'est ailleurs qu'il faut aller chercher les raisons des succès et des échecs des systèmes de programmation visuelle.

Rappelons le, un langage iconique utilise un éditeur graphique, la plupart du temps basé sur un graphe bipartite avec pour sommet des icônes représentant les données d'une part, et les opérateurs du système d'autre part. Les arcs quand à eux représentent le flux de contrôle ou de données.

Le profil cognitif des langages visuels iconiques tend ainsi à se caractériser par une **barrière d'abstraction** plus faible, grâce à l'usage des notations iconiques pour représenter certains concepts qui restent dans les notations textuelles, très abstraits. De plus, spécialement dans les notations « à flot de données », les **dépendances cachées** sont plus rares (Figure 39). D'autre part, de telles notations sont à même de représenter des liens **sémantiques** : les connexions existant entre les objets.

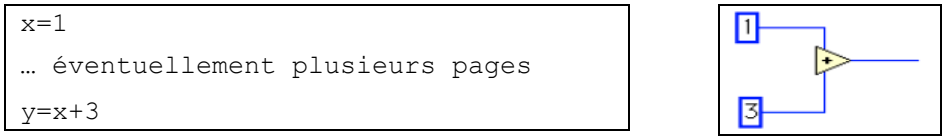


Figure 39 – Représentations implicite (BASIC, à gauche) et explicite (LabVIEW, à droite) des liens entre données.

Pour autant, une telle approche implique certaines contreparties parfois non négligeables. Ainsi, de nombreux langages souffrent de problèmes de **visibilité** (de trop nombreuses connexions peuvent rendre la notation illisible, on parle de « plat de spaghettis ») et de **juxtaposition** (la notation prenant plus de place, il est difficile de lui associer une représentation complémentaire).

Figure 40 – Représentation d'une conditionnelle dans le langage G : il est impossible de visualiser les deux branches simultanément.

De plus, de tels systèmes auront tendance à être **visqueux**, à la fois par répétition (de nombreuses opérations élémentaires à la souris) et par effet de bord (pour maintenir la cohérence syntaxique du graphe). Pour illustration, la Figure 41 décrit les temps d'édition comparés, à algorithme égal, en LabView et en BASIC, la modification à effectuer ayant été décrite explicitement, et des experts chronométrés pendant qu'ils la réalisaient (Green 1996). Cette différence s'explique par une viscosité par effet de bord extrêmement marquée : pour chaque icône ajoutée, déplacée, ou supprimée, il s'est en effet avéré nécessaire de changer tous les connecteurs qui y étaient liés. Green pronostique de plus que l'écart constaté croît

proportionnellement à la taille du programme à modifier (ce qui s'est vérifié dans la pratique avec LabVIEW).

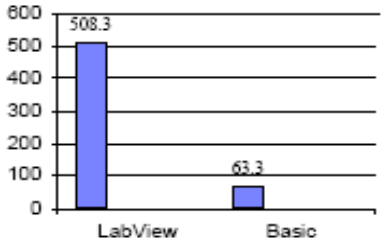
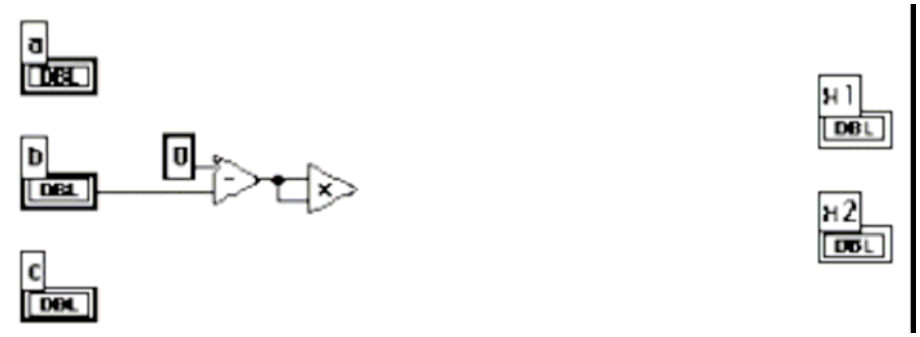


Figure 41 – Temps d'édition (en s) avec LabVIEW et BASIC

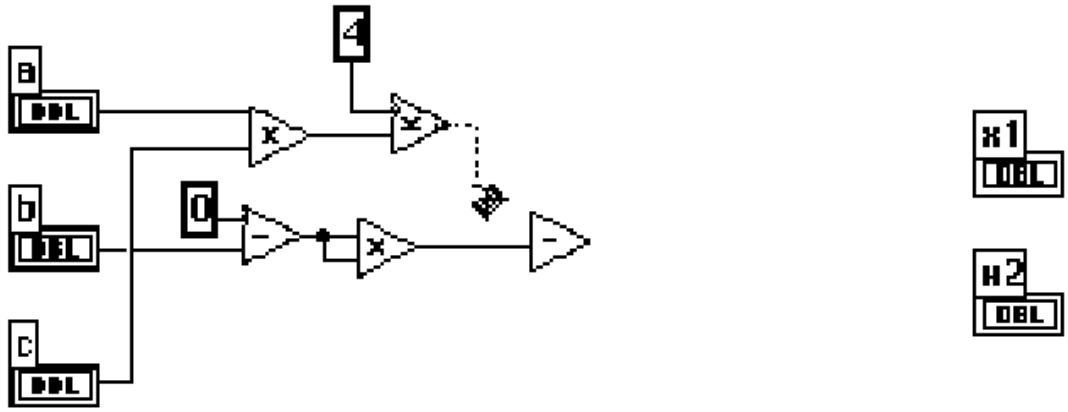
Enfin, pour contrôler de façon active la correction syntaxique voire sémantique du programme, l'ordre de construction tendra à être imposé, ce qui pourra représenter ce que Green appelle un **engagement prématuré**. L**'engagement prématuré** apparaît souvent dans les langages iconiques, à la fois à cause des contraintes spatiales (liée à la forte consommation de l'espace à l'écran par la notation graphique) et d'ordre de construction, comme le montre la Figure 42, tirée de (Green 1996) et qui reprend le log d'un utilisateur de LabVIEW programmant la formule :

$x = (-b \pm \sqrt{(b^2 - 4ac)})/2a$

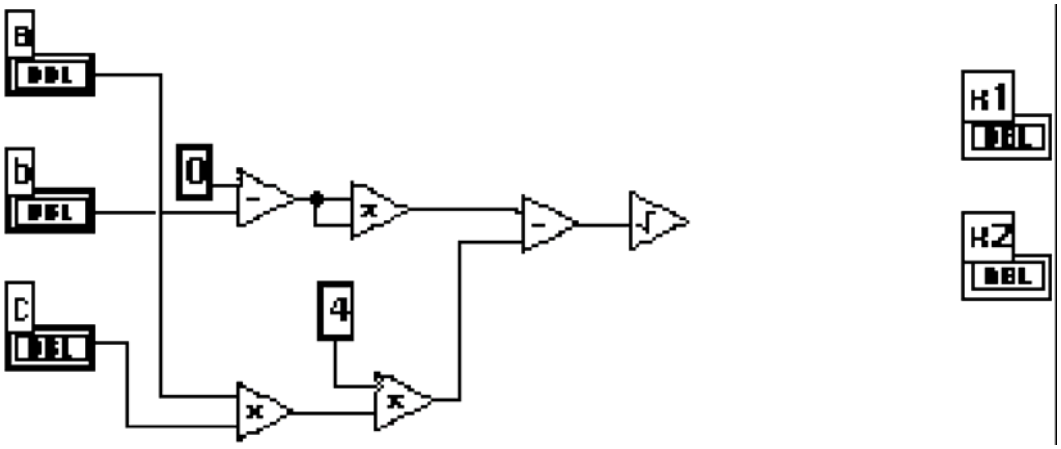
… Alors, d'abord « -b »…



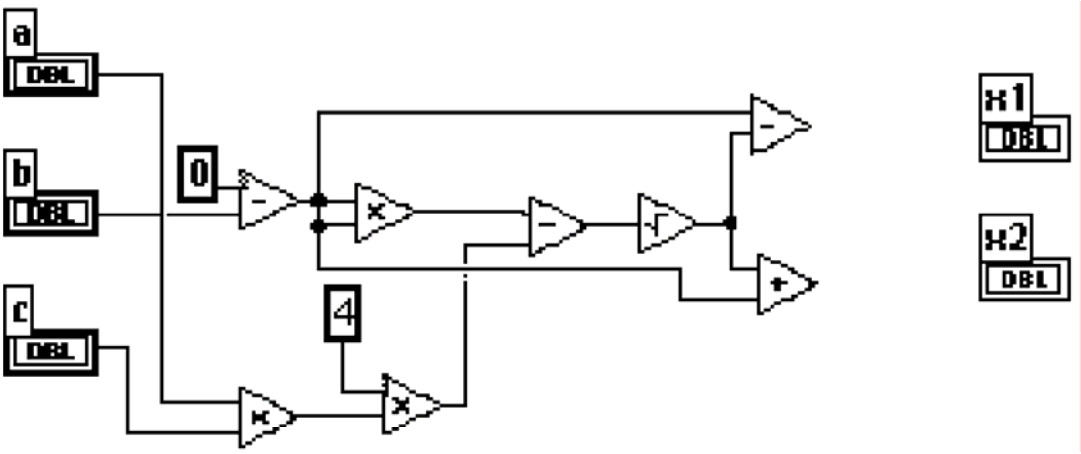
… maintenant j'aurai besoin de b^2 …



… - 4ac …



… ah zut, il faut que je mette 4ac en bas sinon l'expression sera dans le mauvais sens … bon je reconnecte …



… et maintenant –b ± tout çà … ça se croise partout, quel #$%&* !! Et dire qu'il reste encore /2a …

Figure 42 – Langage iconique et engagement prématuré : Log d'un utilisateur LabVIEW.

2.1.3 Approche hybride

Une approche intermédiaire consiste à proposer plusieurs « niveaux » d'éditions, en combinant l'approche « visuelle » pour représenter les modules et les objets de haut niveau à l'approche « textuelle » pour éditer un programme particulier (Figure 43).

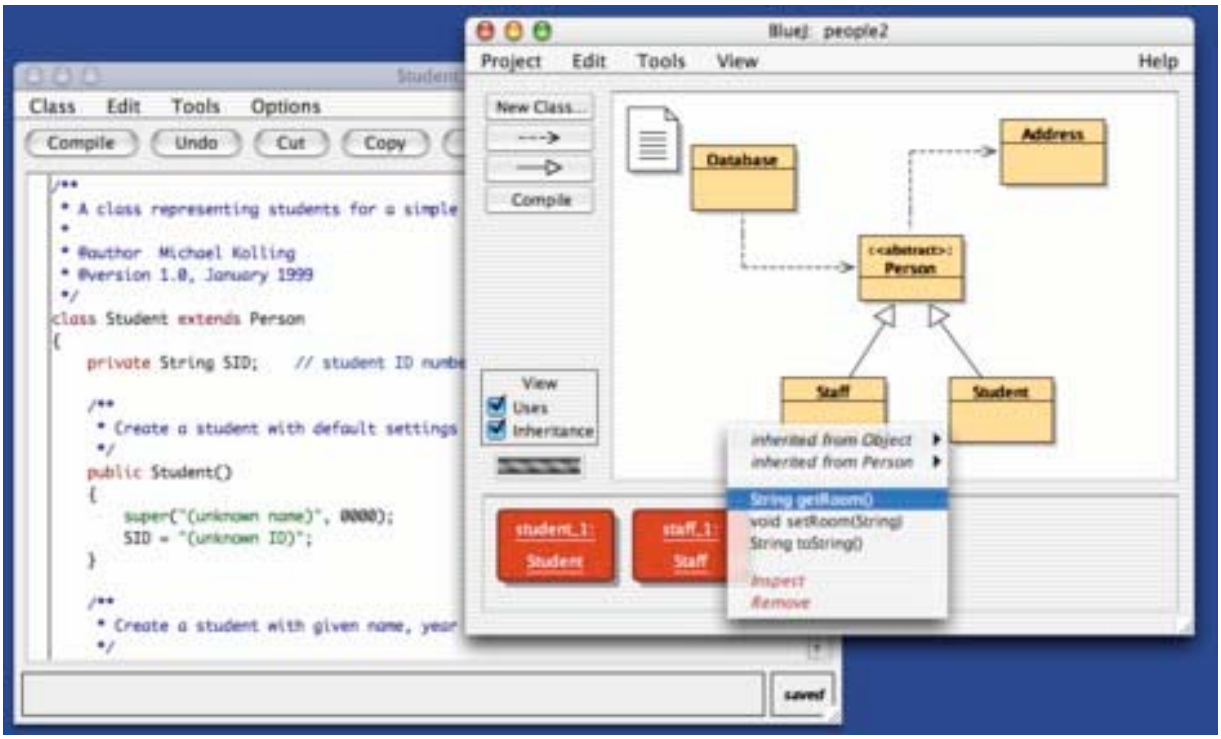


Figure 43 – Un environnement de programmation objet combinant notation graphique (pour représenter les relations inter-classes et les instances courantes) et textuelles (pour représenter les méthodes) : BlueJ

La logique de cette approche « hybride » tient en la complémentarité des deux styles de notation. En effet, les caractéristiques cognitives des notations textuelles sont parfaitement symétriques des caractéristiques des notations graphiques. Cette démarche peut donc donner de bons résultats, étant donné qu'elle permet de résoudre la difficulté de l'explosion du nombre de connexions (qui a un fort impact négatif sur la visibilité et la viscosité) dans le cas de l'approche visuelle, en juxtaposant deux styles de notation complémentaires.

Dans ce cas, la programmation visuelle a pour objet d'exprimer les liens de haut niveau invisibles dans la notation textuelle, alors que cette dernière fait profiter de ses qualités en terme de visibilité et de viscosité à plus bas niveau. Le nombre de **dépendances cachées** est ainsi réduit. L'**engagement prématuré** reste faible ; en effet, l'ordre d'édition du programme est peu contraint, l'accès au code aisé.

La **prévisualisation forcée** demeure cependant aussi importante que pour les environnements textuels « classique ». De plus, le risque d'une telle approche est de surcharger les capacités cognitives de l'utilisateur en raison du plus grand nombre d'informations explicites à gérer simultanément. Nous ne nous étendrons pas sur cette approche, car elle ne s'entend réellement qu'à partir du moment où le programmeur manipule plusieurs sous-programmes, classes ou modules. Elle est donc efficace pour soutenir l'élaboration de liens de **composition** entre des schémas de programmation existants (conception modulaire et orientée objet) et de schémas de **problème** de haut niveau (des décompositions – types dépendant du problème : des

modèles d'architecture, par exemple). Comme nous souhaitons adresser en priorité le cas de débutants, les problématiques de génie logiciel ne nous concernent pas directement.

2.1.4 Programmation « sur » exemple

Dans cette approche (programmation par démonstration) l'état du programme est directement manipulé à la souris (exemple : Pygmalion, ToonTalk…). Cela n'implique pas pour autant qu'il n'y ait aucune syntaxe : en effet l'enregistrement seul est uniquement capable de générer une séquence de commandes, et par conséquent, il faut donc utiliser des conventions de dialogue et widgets particuliers pour casser la linéarité du programme.

C'est avec cette approche que la recherche d'une **barrière d'abstraction** est la plus faible, et l'**évaluation** la plus **progressive**. Cela se fait souvent au détriment du **gradient d'abstraction** des notations utilisées (dont le pouvoir n'est pas critique : il est de loin préférable de faire les tâches du domaine facilement que de faire beaucoup de choses), et cela peut entraîner un grand nombre de **relations cachées** dans la représentation.

Par exemple, une vue d'ensemble d'une feuille de calcul ne permet pas de distinguer les relations entre les cellules, et une capture d'écran de Pygmalion ou de ToonTalk ne permet pas de distinguer les variables des constantes, ni d'identifier le flot de contrôle ayant amené à l'état courant. L'absence de **visibilité** du modèle abstrait a potentiellement peu d'importance en elle-même car le programme est souvent « jetable » dans ce contexte, comme le remarque (Letondal 2001)). Elle pose cependant d'autres problèmes, en sollicitant parfois fortement la **charge cognitive** de l'utilisateur, simplement pour savoir « où il en est ». De plus, l'approche « par démonstration » obligeant l'utilisateur à simuler à l'avance mentalement le comportement de la machine, dans le but d'en faire la démonstration à l'enregistreur, accroît parfois dangereusement ce phénomène, en raison de la **prévisualisation forcée**, et du faible niveau sémantique des opérations qu'il « joue » au regard de la tâche à accomplir (**viscosité par répétition**).

Ce phénomène est beaucoup plus limité lorsque le système est capable d'inférer des contraintes exprimant l'intention de l'utilisateur à partir d'une masse plus restreinte de données, mais la contrepartie est que le système devient plus chaotique et imprévisible (une petite différence au niveau des données d'entrées pouvant entraîner une grande différence sur les contraintes générées) et dès lors assez inadapté à une tâche de programmation.

Une difficulté supplémentaire apparaît dans la réédition et la modification de programmes existants, car en l'absence d'accès direct au programme, le programmeur doit rejouer l'animation du programme jusqu'au point qu'il compte modifier (accès séquentiel), avant de basculer en mode enregistrement. Combinée à l'absence du « undo » dans pratiquement tous les systèmes « par démonstration », cela entraîne un **degré d'engagement** fort dans chaque démonstration, et comme la démonstration des actions doit être absolument faite dans l'ordre d'exécution, cela amène également une **prévisualisation forcée** importante (le concepteur doit préparer mentalement la « démo » avant de la réaliser).

Ces derniers points semblent peu compatibles avec une exploration expérimentale, par cycle d'essais-erreurs, même si il est possible de diminuer cet effet en distribuant le code, et en augmentant le niveau d'abstraction (pour réduire la taille des séquences à enregistrer). Une telle approche ne saurait donc être pertinente dans notre contexte qu'en l'associant à une représentation manipulable du programme, qu'elle soit textuelle, graphique, ou hybride.

2.2 Techniques de retour d'information

2.2.1 Niveau syntaxique

Les interactions et les retours d'information les plus courants dans de nombreux environnements portent sur le niveau syntaxique (*« Comment Dire »*). Cela provient naturellement de ce que, d'une part, ce niveau d'information est le plus facile à traiter, et d'autre part du public éclairé voire professionnel que ces environnements visent. Il leur est donc facile de sous-estimer l'importance de l'écho sémantique, du fait que le programmeur est supposé plus sujet à des « étourderies » - *« slips »* en anglais – syntaxiques qu'à des erreurs sémantiques.

2.2.1.1 Coloration syntaxique

Historiquement, la « coloration syntaxique » dans un éditeur de texte fut la première technique d'aide syntaxique à se généraliser. Elle consiste à colorier les mots-clés du langage, les littéraux, et les commentaires de couleur différente selon leur nature (Figure 44). Cette technique représente une aide au développeur à deux niveaux : d'une part elle permet de repérer les erreurs de frappe dans les mots clés. On peut donc la considérer comme la première avancée vraiment significative du point de vue de **l'évaluation progressive** de la syntaxe, de même qu'un outil efficace permettant au débutant de vérifier sa rétention des mots clés du langage et à lutter contre certains problèmes d' « **Incitation à l'erreur**» dans le choix de ces mots clés. D'autre part cette coloration tend naturellement à améliorer la **visibilité** globale du programme.

On pourra en revanche lui reprocher de ne fournir qu'un feedback de niveau élémentaire : elle n'est d'aucun secours pour vérifier la correction de la composition des différents éléments syntaxiques, et ne permet pas de tester la correction des « **schémas d'implémentation** » de l'étudiant. Cela entraîne de surcroît qu'un grand nombre de messages d'erreurs syntaxiques ne seront levés qu'à la compilation, risquant de surcharger les **ressources cognitives** de l'apprenant. En résulte beaucoup de temps perdu pour le novice, qui pourrait être ainsi amené à accorder une place trop importante à la maîtrise de la syntaxe, allant jusqu'à, dans un cas extrême, considérer la compilation du programme comme principal (seul ?) objectif. Cela entraîne également le comportement consistant à ignorer systématiquement les « warnings ».

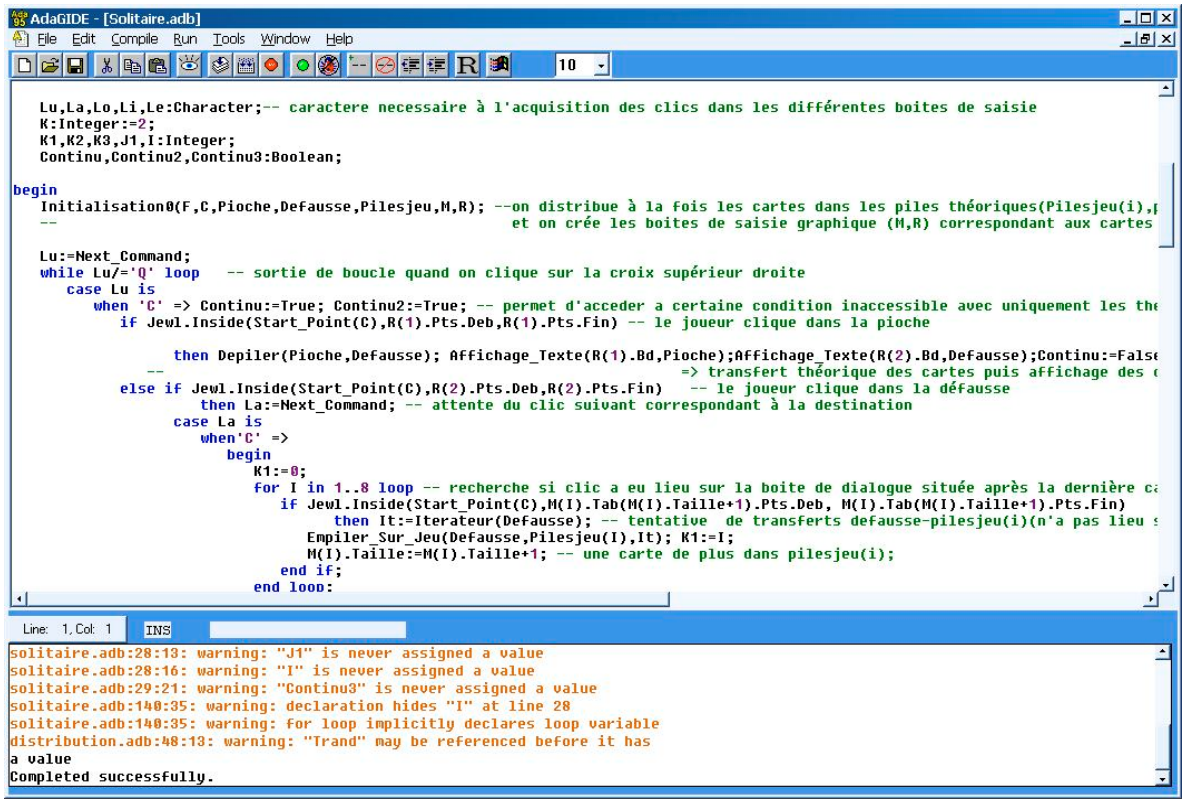


Figure 44 – Un exemple d'environnement de programmation (IDE) pour langage impératif avec coloration syntaxique : AdaGIDE sous windows.

2.2.1.2 Vérification syntaxique « à la volée »

C'est pourquoi, à partir du début des années 90, avec l'augmentation continue de la puissance des machines, et également de leurs capacités graphiques, une fonctionnalité de vérification syntaxique « à la volée » (c'est-à-dire en cours d'édition) a fait son apparition, d'abord en tant que sujet de recherche, puis en finissant par s'imposer dans la plupart des éditeurs commerciaux (Borland, Sun, Microsoft) à la fin de la décennie ; de plus en plus d'IDE gratuits proposent désormais cette fonction.

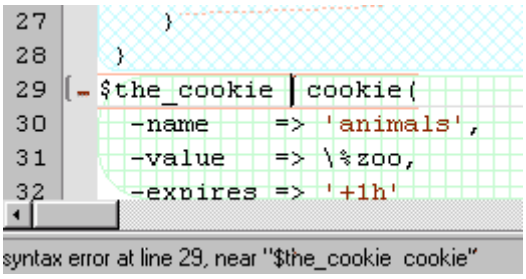


Figure 45 – Vérification syntaxique en cours d'édition, sur un éditeur PERL.

Bien qu'entraînant un surcoût d'utilisation des ressources du système, une telle approche procure en effet de nombreux avantages. D'une part, **l'évaluation progressive** de la syntaxe permet une meilleure compréhension des erreurs, dans le sens où cela évite à l'étudiant d'être submergé de message d'erreurs à la compilation. Du coup, il est possible de rendre le vérificateur syntaxique plus **prolixe** et donc de fournir des retours plus longs, plus **expressifs**, tout en adressant plus d'erreurs potentielles, et ce sans que cela se traduise par une **charge**

cognitive trop élevée. Cela ajoute une plus value directe à l'enseignement en donnant à l'étudiant un retour immédiat sur la correction de certains *schémas d'implémentation.*

2.2.1.3 Edition contrainte

Enfin, une approche plus radicale, mise en place dans les éditeurs de programmes graphiques, consiste à contrôler la construction de l'arbre ou du graphe par l'utilisateur, en lui interdisant de saisir des relations incohérentes (par exemple en grisant certains widgets, ou en refusant la connexion de certaines icônes). Ces environnements induisent une **incitation à l'erreur** plus faible, dans la mesure où l'environnement peut interdire des opérations d'éditions vides de sens (du point de vue du système). Du point du vue syntaxique, cela porte donc l'évaluation progressive à son paroxysme, car l'écho y devient **proactif.** Cela permet à l'apprenant d'acquérir plus rapidement les **schémas d'implémentation** corrects, et permet de porter son attention plus sur la sémantique que sur la seule syntaxe. Ce style d'édition tendra par ailleurs à augmenter la **viscosité** du système.

2.2.2 Niveau sémantique

Alors que les aides au développeur au niveau syntaxique sont nombreuses et pour certaines implantées depuis longtemps dans les environnements de programmation, ceux-ci font généralement moins de cas du retour *sémantique,* alors que paradoxalement, les bogues qui infestent nombre de logiciels commerciaux sont tous dus à des erreurs sémantiques. Cette (relative) désaffection est préjudiciable pour l'apprentissage, dans la mesure où la difficulté principale est justement la mise en place de modèles conceptuels viables des programmes, à travers les schémas corrects de programmation et d'algorithmique, et les liens de composition qui les relient. Cette section présente les techniques de la littérature dédiées au retour sémantique, de l'antique fonction « print » et ses avatars aux techniques spécifiques à l'enseignement.

2.2.2.1 Fonctions de sortie texte de type « terminal »

La plus ancienne et la plus répandue des techniques permettant de modéliser dynamiquement l'état de l'ordinateur consiste à afficher sur un émulateur de terminal le contenu d'une ou plusieurs variables, à des endroits choisis du programme, à travers une fonction de sortie texte. A partir de ces sorties, l'utilisateur cherchera à inférer l'état du système à un moment donné.

Cet écho peut être qualifié d' « à la demande » puisque, hormis les erreurs syntaxiques ou les erreurs fatales d'exécution, l'environnement n'affiche généralement de l'état du système que ce qui lui est explicitement demandé par le programmeur. Cette façon de procéder n'est pas sans inconvénients :

- Dans le cas d'affichage dans une boucle, par exemple, de nombreuses sorties non significatives peuvent bruiter le feedback : la **visibilité** est donc en cause, par excès de **prolixité**. La somme d'informations affichée pouvant être importante, cela risque d'entraîner un problème de **charge cognitive**.

- Le choix de la (des) variable(s) à afficher est fait par l'utilisateur, et à partir de peu d'informations (**engagement prématuré**) ; le **degré d'engagement** dans ce choix est fort car il n'y a pas moyen de changer d'avis en cours d'exécution (le programme

s'exécute toujours jusqu'au bout, et peut difficilement être interrompu). Cela donne une grande importance à la phase de **planification** et donc un désavantage évident pour les étudiants n'ayant pas un style d'apprentissage **dénotatif** (**convergence** ou **accommodation**)

A cause de tous ces problèmes, il est très difficile à un apprenant, particulièrement de style d'apprentissage **connotatif**, de se forger à partir d'un tel feedback un modèle cohérent du fonctionnement du programme.

2.2.2.2 Débogueurs

Les débogueurs, présents aujourd'hui dans tout environnement de programmation digne de ce nom, sont une évolution (majeure) de cette technique primitive. Leur principe de fonctionnement consiste à positionner dans le programme édité et syntaxiquement valide des « points d'arrêt » (Figure 46).

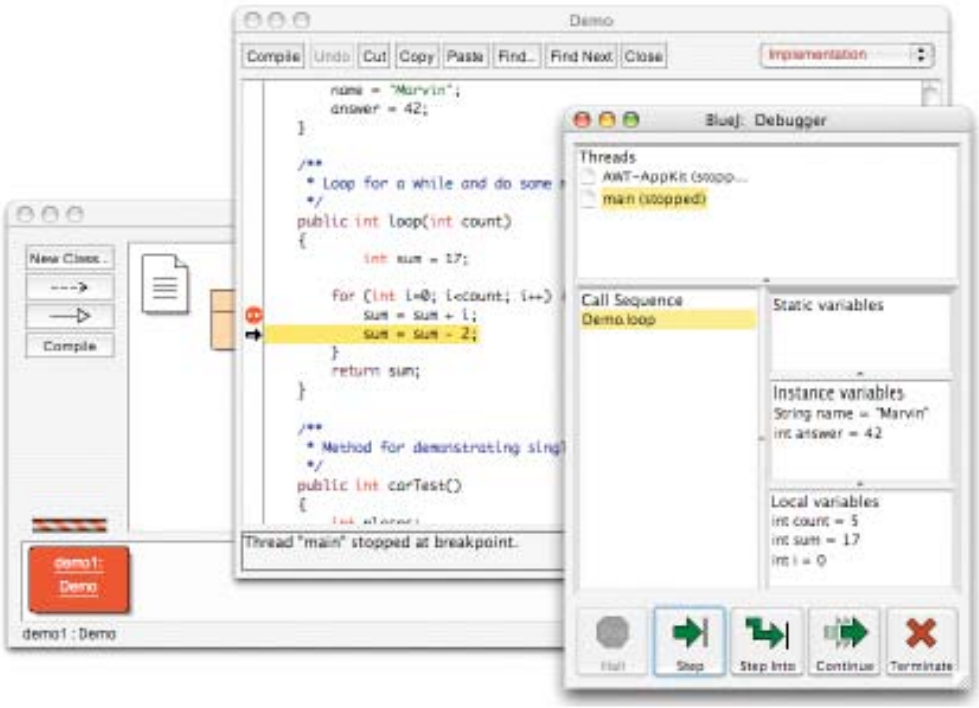


Figure 46 – Un exemple de débogueur (BlueJ) : la petite icône « stop » figure le point d'arrêt (fenêtre centrale), le processus, la méthode et la ligne courante sont surlignés, et les variables du contexte sont présentées avec leur valeur.

Le principal avantage des débogueurs par rapport à l'approche précédente, provient bien sûr du fait que l'exécution du programme s'arrête lorsqu'un point d'arrêt est atteint. Les problèmes de **visibilité** et de **charge cognitive** levés par sorties textes traditionnelles ne se posent donc plus, car l'état du programme auquel se réfèrent les informations du contexte est explicite (le programmeur n'a pas à faire de recherche difficile et fastidieuse à travers l'ensemble des traces des fonctions de sorties). De plus, l'ensemble des valeurs des variables est accessible à chaque « pause » de l'exécution.

Le problème de choisir à l'avance quelles informations seraient pertinentes pour appréhender l'état du programme (**engagement prématuré**) ne se pose donc plus. Cette **prolixité** accrue (le feedback concerne l'ensemble des variables, et non un sous-ensemble choisi par le

programmeur) ne pose pas beaucoup de problèmes, car d'une part, ces informations sont présentées dans leur contexte, et d'autre part, le programmeur a tout son temps pour analyser ces informations.

Enfin, la plupart des débogueurs permettent à chaque arrêt de changer les points d'arrêts (ou tout au moins de passer simplement à l'instruction suivante et d'arrêter l'exécution), la **planification** de l'observation a un **degré d'engagement** moins élevé (vu que le programmeur peut dynamiquement, à la lumière de ses **observations** modifier les observables ou arrêter l'expérience).

Ces deux techniques de feedback ont en commun de fournir une aide à la représentation de l'**état du programme.** Elles ne cherchent pas à visualiser explicitement les flots de contrôle ou de données. Cela implique que pour les concepteurs, ce qui est difficile (et doit être objet d'une aide du système) c'est la relation entre le programme (qui est une représentation statique) et son état (dynamique). Le domaine d'application de ces représentations couvre donc essentiellement les **liens de composition** (entre **schémas de programmation**, et entre schémas de programmation et **schémas algorithmiques**) et les **liens de mise en œuvre**. Les concepts concernés par ces schémas sont essentiellement ceux des structures de contrôle, même si incidemment les feedbacks du contenu dynamique des variables peuvent servir de support à une assimilation des concepts de typage et de valeurs.

2.2.2.3 Programmation « avec exemple »

La programmation « avec exemple » a pour principe de fournir un écho immédiat à l'utilisateur, en cours d'édition, grâce à un (des) exemple(s) défini(s) au moment de la déclaration du programme, voire encore avant. Un résultat direct de cette approche est une **évaluation** complètement **progressive**. La **visibilité accrue** par la présentation graphique de l'abstraction de la stratégie suivie par l'élève pour programmer la tâche, proposée en **juxtaposition** avec un modèle graphique de l'état courant du système, (comme dans StageCast) a pour effet de diminuer la **charge cognitive** liée à la représentation mentale de l'état du programme. Par ailleurs, la juxtaposition des vues du programme et de l'état du contexte permet de réduire les **dépendances cachées,** qui peuvent s'avérer être un obstacle à l'évaluation du programme.

Néanmoins, ce style de retour d'informations n'est possible qu'en éditant le programme dans l'ordre de son exécution, sans quoi les retours seront au mieux incohérents. Cette prévisualisation forcée n'est peut-être pas au goût ou à la portée de tous…

2.2.2.4 Animation de programmes

Une autre approche, utilisée dans des environnements spécifiquement dédiés à l'apprentissage de la programmation, consiste à animer l'ensemble du programme, une fois celui-ci édité et compilé avec succès. On présente donc à l'apprenant un « film » de l'exécution du programme, et plus seulement quelques « instantanés ». Il y a donc contrairement à précédemment, une volonté explicite (à travers le déroulement du film) de représenter le flot de contrôle et le flot de données. Cette approche se différencie aussi des précédentes dans la réponse à la question « qui ». La responsabilité du choix des points d'arrêts de l'exécution est dans cette approche laissée au système (qui s'arrête puis redémarre après chaque commande). La phase de planification y est donc réduite au choix du jeu de test.

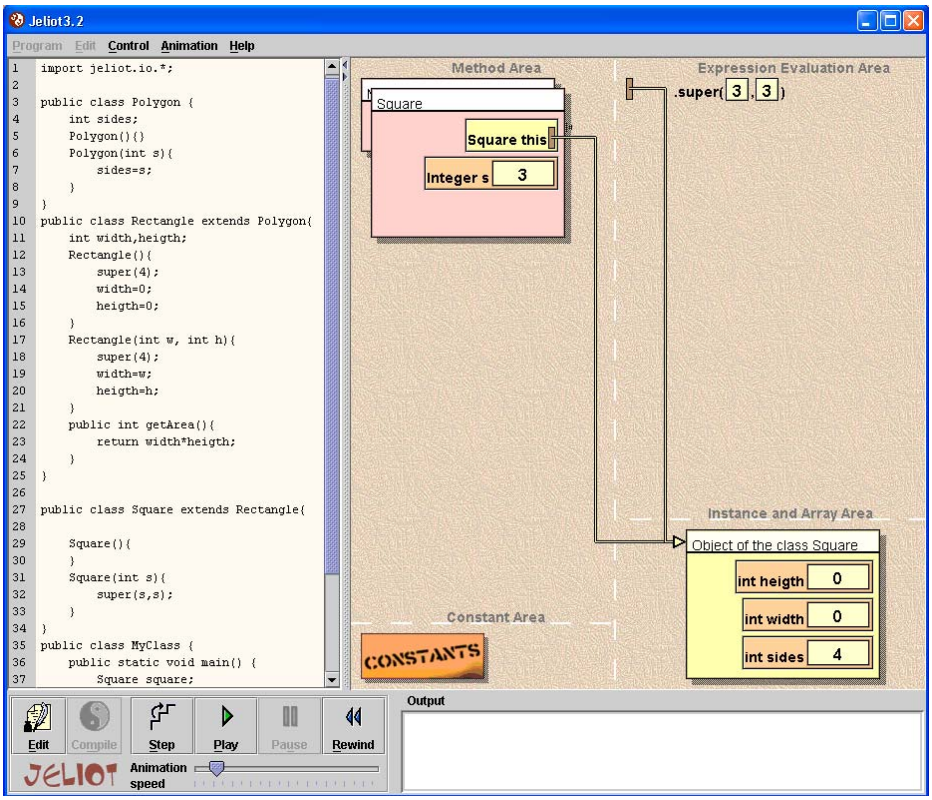


Figure 47 – Animation d'un programme Java (Jeliot)

Cette différence peut expliquer que l'animation ait un taux de satisfaction élevé chez les débutants, contrairement à nombre de débogueurs. En effet, elle implique un **engagement prématuré** plus réduit. Certaines connaissances préalables sur le fonctionnement des structures de contrôle et les erreurs types sont en effet nécessaires pour savoir à l'avance où placer les points d'arrêts (et l'animation, au contraire, ne souffre d'aucune **barrière d'abstraction**). Pour la même raison, on peut s'attendre à ce que les étudiants ayant un profil plus **connotatif** préfèrent cette technique aux précédentes.

Pour autant, une telle approche n'est pas la panacée ; elle supporte presque exclusivement l'acquisition de **schémas élémentaires de programmation** sur les structures de contrôle et les variables. Son importante **prolixité** conduit en effet les étudiants à traiter un très grand nombre d'informations (pour chaque commande, cette approche affiche en effet le contexte en son totalité, alors qu'il n'y aura en fait qu'un faible nombre de variables et d'états significatifs). Il y a donc un risque non négligeable de problème de **charge cognitive**, dès lors que le nombre de variables augmente et que la structure du programme se complexifie. Cela entrave fortement la capacité de cette technique à supporter la construction de **liens de composition et de mise en œuvre** (ou même de **schémas algorithmiques**, dès lors que l'exemple choisi n'est pas trivial). Il y a donc également un risque de désaffection au fur à mesure de la progression des étudiants.

2.2.2.5 Programmation visuelle iconique

Une notation « graphique », telle que celle présentée dans les sections précédentes, est également à mesure d'offrir un support sémantique. Comme la plupart de ces notations manipulent des graphes décrivant les relations (sémantique) entre objets, ou entre opérateurs et opérandes, la structure du graphe représente, en soi, un début de support sémantique. Par

ailleurs, il est possible de contraindre la construction du graphe, en empêchant l'utilisateur de saisir des relations incohérentes. Ainsi, alors que les techniques précédentes gèrent le feedback **« en aval »**, une fois le programme écrit puis validé par un compilateur – et ont donc l'inconvénient de requérir une forte **prévisualisation forcée** pour compenser la perte de manipulation directe – ici la notation supporte directement la *conception*. L'**évaluation** devient plus donc **progressive,** certes au détriment d'une **viscosité** plus grande, et du risque d'**engagement prématuré**. Cette aide sémantique apporté par le guidage lors de la création du graphe concerne essentiellement la sémantique de plus bas niveau, c'est-à-dire les **schémas élémentaires de programmation**.

2.3 Niveaux de représentations de l'état du système

2.3.1 Représentations symboliques

Les représentations du contexte que ce soit dans les IDE commerciaux, ou dans de nombreux environnements académiques, font majoritairement appel à des notations *symboliques* (Figure 48). Celles-ci ont pour principal avantage, dans une optique de développement, d'être indépendantes du contexte d'usage. Cette généralité rend l'environnement **cohérent** quel que soit le domaine d'application. Un développeur pourra donc y conserver ses repères. Cela peut desservir les novices, car une telle généricité va souvent de pair avec une **barrière d'abstraction** accrue. Par ailleurs, une telle approche a pour conséquence de diminuer la **correspondance au domaine**. Elle traite donc exclusivement des difficultés qu'il y a à relier le programme à son comportement dynamique en utilisant les concepts de la programmation. Les difficultés à passer du domaine de la tâche au domaine informatique ne sont pas abordées.

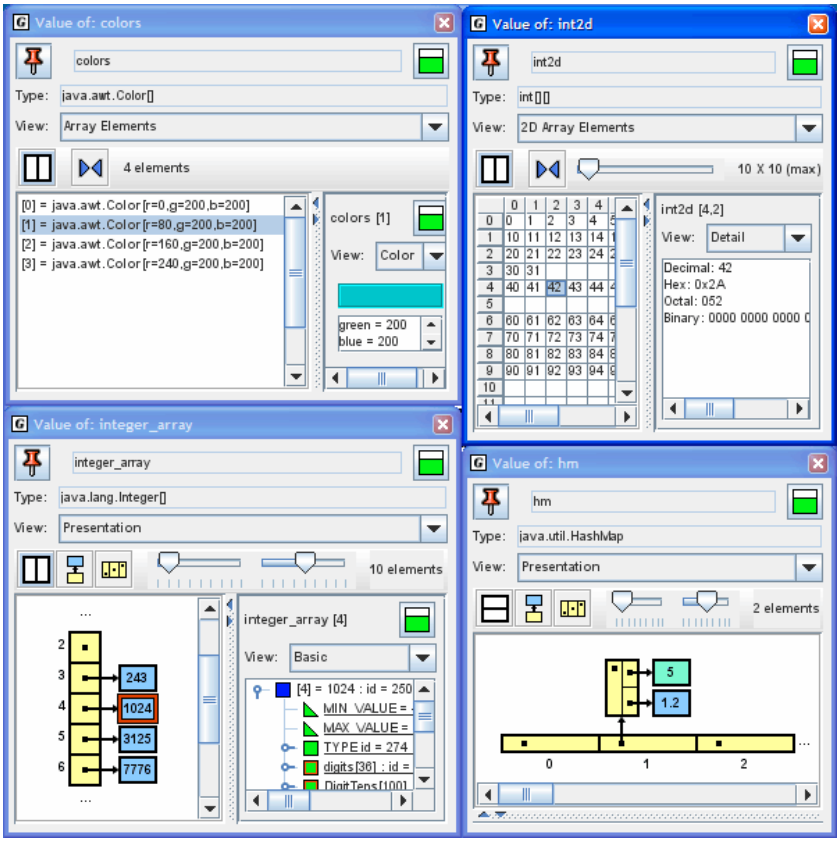


Figure 48 – Différentes présentations de certains types de données (JGrasp).

2.3.2 Représentations par métaphores

Une approche maintenant assez répandue en informatique pour pallier à l'absence de modèle naïf du fonctionnement de l'ordinateur consiste à se servir de métaphores. Cette approche a l'avantage de s'appuyer sur des connaissances préexistantes de l'apprenant dans d'autres domaines pour expliquer des modèles de fonctionnement informatiques. L'introduction avec succès de la métaphore du bureau à l'apparition du paradigme WIMP, à la fin des années 1970, en est l'exemple le plus marquant, mais il y en a bien d'autres (telles que l'emploi assez généralisé de métaphores de réseaux ferroviaires ou postaux pour expliquer le fonctionnement de réseaux informatiques). Cette approche a été employée intensivement dans le domaine de la « end-user programming », dans les cas où le public visé était des enfants, par exemple dans le cas de HANDS – ***H**uman-centered **A**dvances for the **N**ovice **D**evelopment of **S**oftware* (Pane 2002).

Le modèle-cible y est celui d'un langage de programmation événementielle, tirée d'études (Pane 2001) sur la façon dont des non-programmeurs abordent l'expression de tâches de résolution de problème. Ainsi, le programme (dans cette exemple une simulation où des abeilles récoltent le nectar de fleurs en floraison) est un ensemble de « règles » d'actions, exprimées sous la forme : « lorsque <condition> faire <actions> » - voir Figure 49. De la sorte, le système facilite à l'utilisateur la description du faire faire. Celle-ci se fait à partir de structures de contrôle issues des habitudes de l'utilisateur, et non pas des capacités de la machine[1].

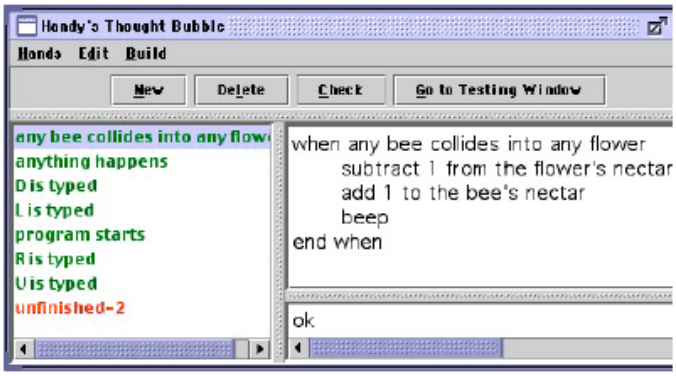


Figure 49 – Règles d'actions composant un programme sous HANDS.

La métaphore utilisée pour le représenter est illustrée par la Figure 50. L'interpréteur y est explicitement représenté sous la forme d'un agent (« Handy ») qui a comme attributs une bulle de pensée (cliquer sur celle-ci ouvre la fenêtre des règles de la Figure 49) et une main de cartes. Chacune de ces cartes correspond, dans la métaphore, à une variable, à laquelle est associée un nom, une représentation graphique dépendant de son type et de son état (le coté « pile » de la carte). Une fois retournée, chaque carte affiche les valeurs de l'ensemble de ses attributs. L'environnement d'exécution est une table ronde sur laquelle sont posées toutes les cartes en jeu.

[1] Mais incidemment, cela interdit (ou tout au moins restreint fortement) l'utilisation de HANDS pour « apprendre à programmer » : il est dès lors plutôt question de « programmer pour apprendre » (la résolution de problèmes).

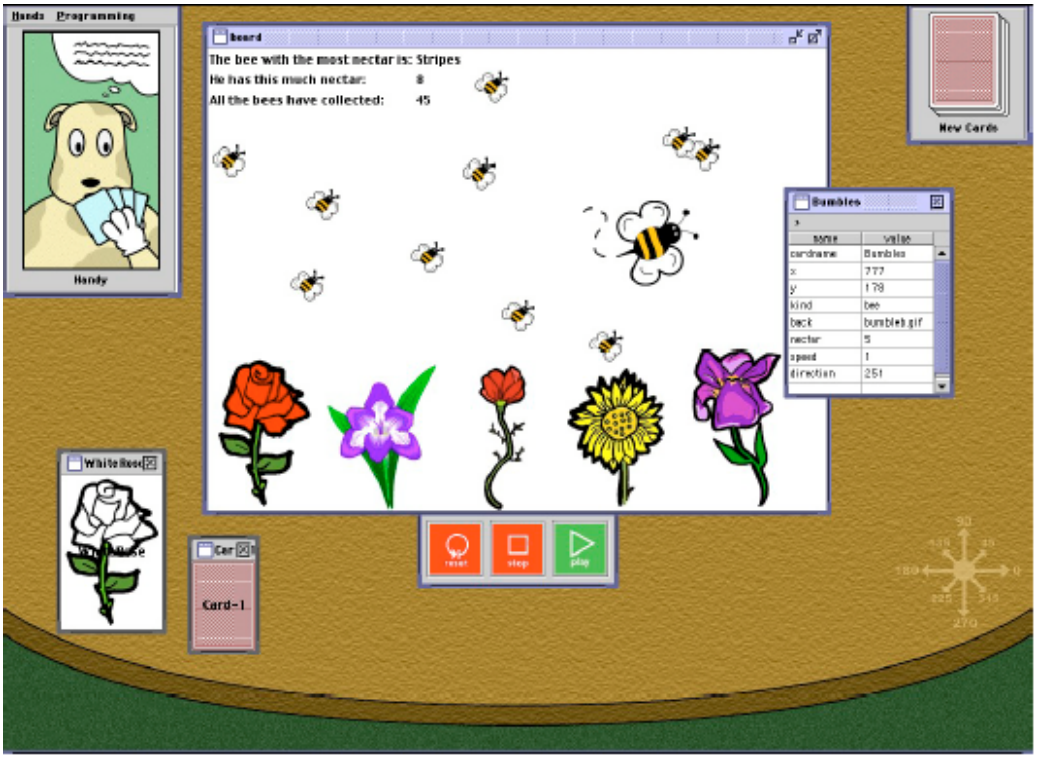


Figure 50 – Environnement d'exécution de HANDS.

D'un point de vue cognitif, l'usage de la métaphore a comme avantage immédiat de permettre d'**appréhender** les concepts informatiques par référence à des expériences concrètes préalables, ce qui est directement bénéfique à des étudiants ayant un style d'apprentissage par **connotation**. Cependant, le domaine de la métaphore ne recouvre jamais totalement celui de l'informatique. Du coup, les métaphores comportent souvent un effet de bord gênant : elles tendent à générer de la « fausse consonance », et donc **inciter à l'erreur**. La fausse consonance se produit lorsqu'une nouvelle information apparaît familière, alors qu'elle porte en fait un sens différent. Cet aspect familier est susceptible d'induire l'apprenant en erreur, entraînant des interprétations faussées des réactions du système. En induisant des explications qui paraissent raisonnables à l'apprenant, avant de les réfuter brutalement, ce phénomène amène l'apprenant à penser qu'il ne peut pas se fier à son savoir, à ses explications, à ses hypothèses pour expliquer le comportement du système.

Par exemple, la métaphore de la boîte appliquée au concept de variable, étudiée par Du Boulay (Du Boulay 1989), est propice à la fausse consonance en ce que, comme une boîte peut couramment contenir plusieurs objets, certains étudiants se figurent qu'une variable peut contenir l'historique de ses différentes valeurs au cours de l'exécution. Une autre erreur observée provenant de cette analogie est que les variables sont toujours assumées initialisées automatiquement à 0, car : *« une boîte est vide jusqu'à ce qu'on mette quelque chose à l'intérieur »*. De même, d'autres élèves, confrontés à l'affectation d'une variable par une expression, s'imaginent que la variable contient l'expression elle-même, et non la valeur de celle-ci ; en effet le terme « reçoit » et l'analogie à « mettre dans la boîte », utilisés par l'enseignant pour décrire l'opération ne sous-entendent nullement à leurs yeux une quelconque évaluation. Un autre exemple rencontré plusieurs fois dans la littérature est celui des opérateurs d'itération conditionnelle « Tant que [condition] répéter [séquence] » et « Répéter [séquence] jusqu'à [condition] ». Aiguillés par la signification de ce texte dans la

langue naturelle, certains étudiants pensent à tort que la séquence s'interrompt dès que la condition devient vraie (Tant Que …) ou fausse (pour Jusqu'à …). Ainsi le code :

```
a := 0 ;
…
While (a<4)
   Repeat
         a :=  a + 1;
         println (« coucou ») ;
   End Repeat ;
```

… leur semblerait devoir n'afficher « coucou » que trois fois, l'itération étant interrompue dès que la variable « a » atteint la valeur 4, c'est-à-dire juste après l'instruction d'incrémentation « a :=a+1 ».

Le choix du domaine d'origine d'une métaphore et l'examen des différences de comportements entre le concept cible et la comparaison ont donc une importance cruciale. Enfin, l'usage de métaphores ne porte que sur les connaissances, en particulier sur les concepts liés aux données, et pas sur les schémas de programme, d'algorithmes ou de problèmes, ni sur les liens entre eux. Son domaine d'application est donc assez restreint (mais potentiellement crucial). Ajoutons pour conclure que l'usage de métaphores s'appuie sur une hypothèse implicite qui veut que la transition entre le domaine de la tâche et le domaine de la métaphore soit plus facile que celle entre le domaine de la tâche et le domaine informatique.

2.3.3 Représentations pragmatiques

Alternativement, les concepteurs de certains langages et/ou environnements ont choisi de se centrer sur le domaine de la tâche et d'adopter le point de vue du concepteur humain. Ainsi, avec StageCast Creator (Smith 2000), Smith et Cypher ont-ils adoptés un point de vue centré sur le domaine de la tâche (Figure 51) : *« … la programmation [y] est maintenue dans des termes du domaine, tel que des trains et des voies ferrées, plutôt que dans des termes informatiques, tel que des tableaux ou des vecteurs… »*

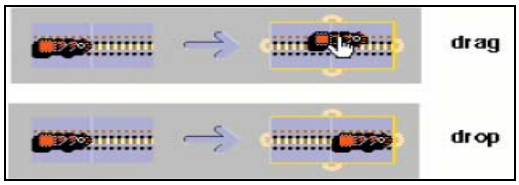


Figure 51 – Programmation dans un micromonde modélisant le domaine de la tâche : une simulation de réseau ferroviaire dans StageCast Creator.

Pour cela, le programmeur manipule non pas des commandes et des objets abstraits et génériques, mais un modèle du domaine de la tâche, appelé « micromonde ». Le plus ancien et le plus célèbre exemple de cette approche est le langage Logo, et son micromonde de la tortue graphique, dans le domaine du dessin de figures géométriques (Figure 52).

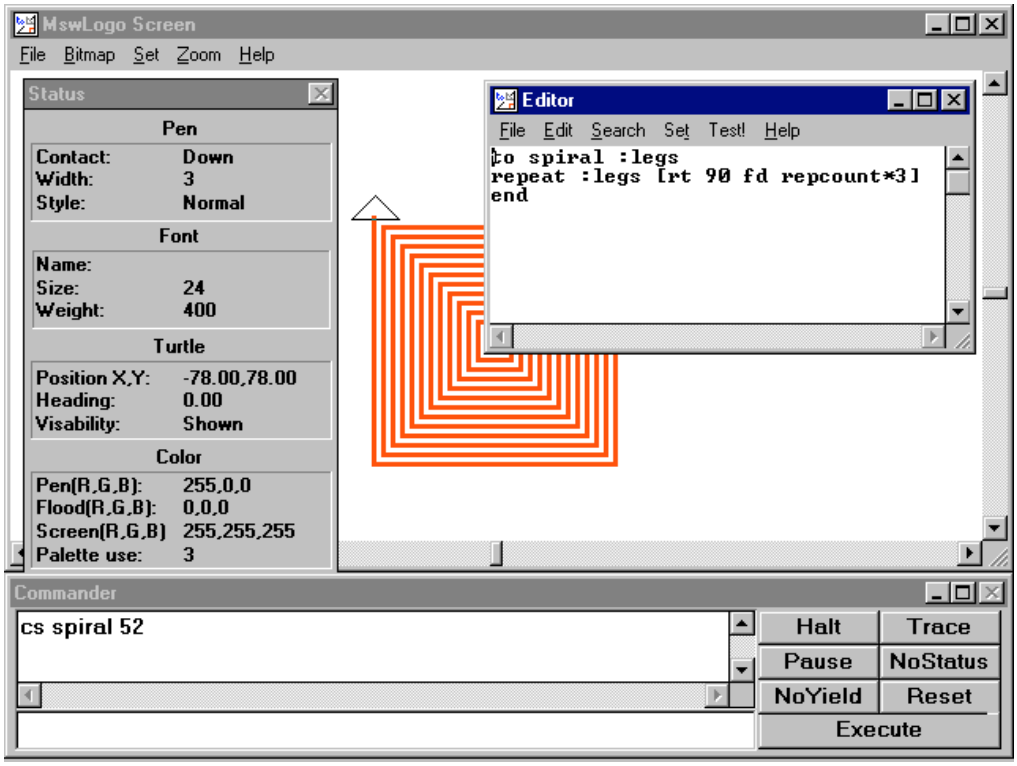


Figure 52 – LOGO, le plus célèbre système d'apprentissage de la programmation par micromonde, et sa fameuse tortue graphique.

Le but de cette approche est de supprimer les difficultés liées à l'apprentissage d'un modèle informatique, abstrait et générique, d'une part, et aux transitions entre ce modèle et le modèle de la tâche. Pour cela, les capacités de l'exécutant sont formalisées dans le domaine de la tâche. Dès lors, « faire faire » ne pose plus autant de difficultés à l'utilisateur, car l'exécutant possède un cadre de référence proche du sien et car ses capacités sont explicites et claires. Il peut donc se concentrer sur les problèmes d'abstraction de la tâche et d'élaboration de stratégies.

Cette approche n'en est pas moins riche en applications, dans le but d'apprendre la programmation. Le fait de masquer (temporairement) un niveau de détail particulier est une façon d'appréhender progressivement une réalité préalablement simplifiée, de diviser la difficulté de l'apprentissage en abordant les barrières les unes après les autres. Mettre en œuvre une réalité réduite peut être utile pour focaliser l'attention de l'apprenant débutant et cibler son apprentissage sur des concepts fondamentaux, sans lui demander à cette étape une maîtrise complète de tous les détails du modèle.

Du point de vue des dimensions cognitives, une telle approche possède donc pour principaux avantages de minimiser la **barrière d'abstraction** et d'augmenter l'**expressivité** des opérateurs, en jouant sur la **correspondance au domaine** (de la tâche). De plus, le fait de permettre à l'apprenant de raisonner sur un modèle graphique simplifié permet d'optimiser la **visibilité** de l'état du système, sans que cela ait d'influence négative sur la **charge cognitive**. Cependant, l'acquisition des connaissances et des compétences supplémentaires nécessitera de repasser à un modèle plus complet et abstrait, car le principal inconvénient des micromondes graphiques dédiés à la programmation est d'être **abstractophobes**.

3 Conclusion

Le Tableau 6 ci-dessous récapitule les compétences dont l'acquisition est supportée par les différentes techniques passées en revue dans ce chapitre. La pertinence des différentes techniques à supporter l'acquisition des connaissances ou schémas de connaissance y est notée de « -- » (aucun support) à « ++ » (très pertinent). On y remarque que les outils communément utilisés (coloration syntaxique, vérification syntaxique à la volée, sorties terminal, et débuggeurs) privilégient essentiellement les compétences liées à la syntaxe, et aux schémas de programmation de « bas niveau ». Cela n'a rien de surprenant : étant issues d'environnement d'inspiration « professionnelle », ces techniques tendent naturellement à se focaliser plus sur les erreurs de manipulation, de nature articulatoire. En effet, les développeurs supposés confirmés, auxquels ces environnements sont prioritairement destinés, n'ont pas besoin de support à l'assimilation des concepts et des schémas plus sémantiques. Ceux-ci sont, en revanche, pris en charge, quoi qu'imparfaitement, par les outils issus d'un contexte plus académique.

Technique / Savoir ou Savoir-faire	**Coloration syntaxique / Vérification à la volée**	**Programmation visuelle**	**Métaphore**	**Débuggeur**	**Animation de programme**	**Programmation basée sur l'exemple**	**Approche centrée domaine (pragmatique)**
Connaissances syntaxiques	++	++	−−	−−	−−	−−	−−
Connaissances sémantiques	−−	+	++	−	+	+	−−
Schémas Elémentaires de programmation	+−	+	−−	+	++	++	−−
Schémas algorithmiques	−−	−	−−	+	+	++	++
Schémas d'implémentation	−−	++	+	−	−	−	−−
Schémas du problème	−−	−−	+	−−	−−	−	++

Tableau 6 – Adéquation des différentes techniques d'interaction et de retour d'informations en programmation avec l'acquisition des connaissances du domaine.

Cette analyse des caractéristiques cognitives des différentes techniques d'interactions et de leur domaine d'application à l'apprentissage de la programmation nous permet de définir la

combinaison de techniques la plus à même d'implémenter deux aspects définis dans le cahier des charges fonctionnel du chapitre 2 :

- **Support d'un apprentissage expérimental :** pour cela, l'approche de la **programmation « avec exemple »** paraît le meilleur candidat, car elle permet une ***évaluation progressive*** qui soutient le bricolage et les cycles essai-erreur dans la conception du programme ; enfin elle s'adapte parfaitement à des tâches de mise au point et induit peu de ***prévisualisation forcée*** (comparativement à l'approche « sur » exemple). Afin de ne pas perturber l'étudiant par des retours d'erreurs syntaxiques intempestifs (faible ***incitation à l'erreur***), il nous semble qu'un **style d'édition graphique et contraint** est le plus adapté. Pour autant le choix d'une **grammaire textuelle** nous paraît plus judicieux que celui d'un langage graphique iconique, car la ***viscosité par effet de bords*** que ces formalismes génèrent peut se révéler un frein à l'expérimentation, sans compter qu'ils induisent une ***visibilité*** moindre que celle du texte indenté. De façon complémentaire, des fonctionnalités d'**animation de programme** devraient permettre de soutenir la phase d'observation plus efficacement en cas d'évaluation globale du programme.

- **Approche « incrémentale » :** pour faciliter l'accrétion, il est recommandé de limiter le nombre de concepts à appréhender simultanément, c'est-à-dire de rechercher un fort ***gradient d'abstraction*** et une faible ***barrière d'abstraction***. Pour cela, il nous paraît préférable de combiner **représentations pragmatiques** et **symboliques**. En effet, en retardant l'introduction des concepts afférents aux structures de données, les représentations pragmatiques permettent d'utiliser la ***correspondance au domaine*** pour augmenter l'***expressivité*** des actions reconnues par le processeur (tout comme la ***visibilité*** de son état). La ***juxtaposition*** de ces deux styles de représentations pourrait être utilisée pour faciliter la création de schémas d'implémentations portant sur la modélisation des objets sous forme de structure de données informatiques.

Chapitre 4 — *MELBA, un environnement « basé sur l'exemple » pour apprendre à programmer*

***Résumé**. Dans l'optique d'initier les étudiants aux concepts fondateurs de la programmation telle qu'elle est pratiquée dans le contexte industriel, nous avons expliqué que l'usage d'outils issus de ce contexte et conçus dans une optique de développement, posait de nombreux problèmes, et ne permettait pas aux novices d'acquérir efficacement le socle de connaissances et de savoir-faire indispensable à l'appréhension de l'activité du programmeur.*

A cette approche « industrielle » s'oppose une approche explicitement pédagogique, visant à mettre en place des Situations d'Apprentissage Actif ou Situations Actives d'Apprentissage (SAA), dans lesquelles les interactions de l'utilisateur avec l'environnement ont pour but premier la découverte et la construction de connaissances, et non pas la réalisation d'une tâche technique. Guéraud & al. (Guéraud 2004) ont introduit le terme d' « Objet Pédagogique Interactif » (OPI) pour désigner les environnement supports de cette approche centrée sur l'apprenant et ses activités (généralement des simulateurs).

C'est dans cette seconde optique que nous avons conçu l'environnement d'apprentissage « MELBA » (Metaphor-based Environment to Learn the Basics of Algorithmics). Il a pour but explicite l'apprentissage, et pas la conception de programmes. Ce chapitre décrit l'environnement MELBA, en proposant tout d'abord une vue d'ensemble de sa structure, avant de détailler l'environnement destiné aux apprenants et les interactions entre ceux-ci et le système, puis celui destiné à la conception et l'intégration d'exercices par un enseignant. Nous discutons dans cette présentation des choix de conception en les mettant en relation avec le cahier des charges établi à la fin du chapitre précédent.

1 Introduction

Dans l'optique de mettre en place des Situations d'Apprentissage Actif (SAA), où les interactions de l'utilisateur avec l'environnement ont pour but premier la découverte et la construction de connaissances, nous avons conçu l'environnement d'initiation à l'algorithmique MELBA (Metaphor-based Environment to Learn the Basics of Algorithmics). MELBA est destiné à supporter l'apprentissage des concepts fondamentaux de la programmation. Ses principales caractéristiques sont d'être :

- un outil autonome qui propose à l'élève des exercices de programmation à résoudre interactivement : l'édition, l'interprétation, et l'exécution du programme écrit par l'élève y sont intégralement gérées, sans passer par un compilateur ou un interpréteur externe ;
- un environnement modulable au gré des souhaits de l'enseignant. Pour faciliter l'apprentissage, l'environnement élève de MELBA peut être personnalisé par l'enseignant selon l'exercice et les concepts ou compétences cibles de l'apprentissage. Il s'appuie pour cela sur une structure modulaire, basée sur des agents logiciels qui peuvent être ou non activés selon l'exercice.

Par ailleurs, nous avons fait le choix de nous concentrer sur l'acquisition de compétences précises, ayant pour cadre l'initiation à la programmation et l'algorithmique, dans un cadre académique. De ce fait, MELBA n'est pas :

- Un outil d'auto-apprentissage. En effet, si le e-learning et l'auto-apprentissage connaissent actuellement un engouement certain dans la recherche sur les EIAH, ces pratiques d'enseignement induisent certaines contraintes et caractéristiques qui leur sont propres. Ces caractéristiques risquant d'interférer avec l'évaluation de l'utilité pédagogique du paradigme « sur exemple », nous avons choisi de ne pas supporter ces pratiques d'enseignement dans la version courante (1.5) de l'outil. Leur support pourra être l'objet de développements ultérieurs.
- Un outil de suivi de l'évolution de l'élève. En l'occurrence, la didactique de l'informatique est une science récente et encore peu développée (en particulier en France), et par conséquent, il n'existe pas, à notre connaissance, de modèle précis et véritablement consensuel de l'apprenant (ce qui rend difficile le suivi précis de son évolution). Par ailleurs, cette technologie est plus liée au support de l'activité de l'enseignant qu'au support de l'acquisition de compétences et de connaissances par l'élève, ce qui place son implémentation en dehors de l'objectif proprement dit de la thèse.
- Un environnement de programmation « pleine échelle ». Contrairement à la plupart des environnements décrits dans les deux chapitres précédents, qui s'appuient sur des langages du commerce, ou bien ont pour ambition d'être équivalents à ceux-ci, MELBA s'appuie un langage algorithmique en français. Le but n'est absolument pas de remplacer un environnement professionnel pour un langage professionnel, et le système n'est pas adapté à un tel usage.

Avec MELBA, un cursus d'initiation à l'algorithmique se gère en deux étapes. Premièrement, l'enseignant prépare une série d'exercice, en s'appuyant éventuellement sur un prototype

d'environnement auteur, qui sera décrit ultérieurement dans ce chapitre. Pour chaque exercice, il spécifie les composants utilisés, et plus généralement configure l'environnement de façon à ce qu'il supporte les concepts qui vont être travaillés et eux seuls (afin de ne pas surcharger l'interface de l'environnement de commandes inutiles). Dans un second temps, l'élève résout les exercices dans l'ordre, dans une configuration « travaux dirigés », avec le support éventuel de l'enseignant.

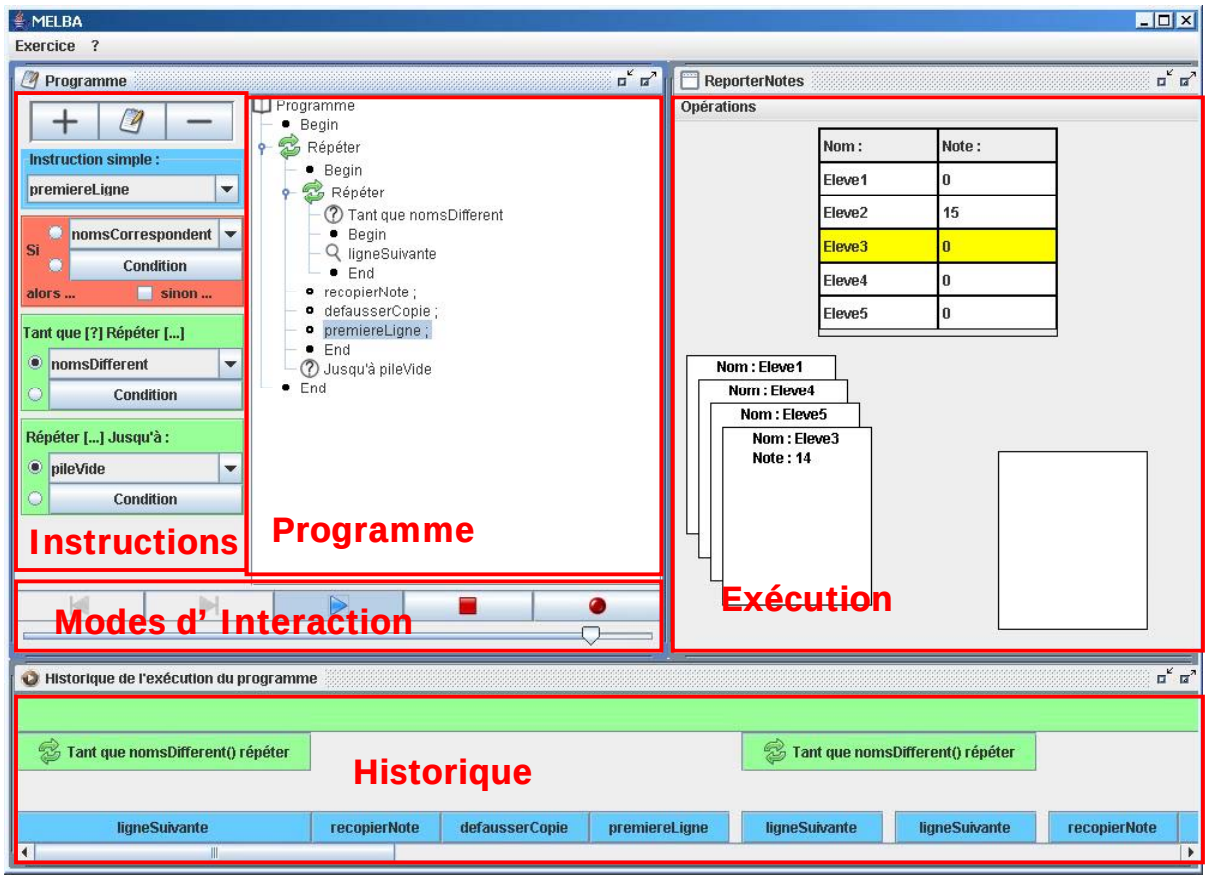


Figure 53 – Vue d'ensemble des différents composants de l'environnement élève de MELBA.

La Figure 53 représente les différents composants de l'environnement MELBA. On y distingue un agent de présentation du programme, organisé de façon arborescente (« Programme »), un agent permettant de créer des instructions, ou d'en retirer du programme (« Instructions »), un agent présentant un contexte graphique d'exécution de l'exemple (« Exécution »), un agent permettant de spécifier le mode de fonctionnement du système (« Modes d'Interaction »), et un agent proposant une représentation arborescente de l'historique d'exécution du programme (« Historique »). En plus de ces cinq composants interactifs, le système affiche des messages d'erreurs contextuels. Dans la suite de ce chapitre, nous présentons séquentiellement les différents composants de MELBA et les concepts sous-jacents. Dans chaque section, et pour chaque sous-section, le plan suivi sera :

- présentation du concept de Melba,
- illustration,
- buts recherchés par rapport aux objectifs du chapitre 3, et justification des choix,
- comparaison avec les outils existants.

2 Un éditeur de programmes contraint

L'étudiant interagit avec le système en construisant un programme par insertions et suppressions d'instructions sur un arbre (Figure 54) : cette programmation, bien que « visuelle » dans le sens où les interactions se font à la souris, sur des éléments d'un niveau sémantique, s'appuie sur un langage algorithmique structuré « classique » (annoté par un mécanisme d'icônes, un peu à la manière de ce qui se fait dans Grasp). En effet, contrairement à la plupart des environnements de l'état de l'art (chapitre 2), MELBA a pour objectif de supporter non pas « la programmation sans apprentissage », mais l'apprentissage de la programmation structurée, c'est pourquoi le choix d'un langage réellement visuel aurait compliqué inutilement le transfert de connaissances et de compétences vers les langages commerciaux existants.

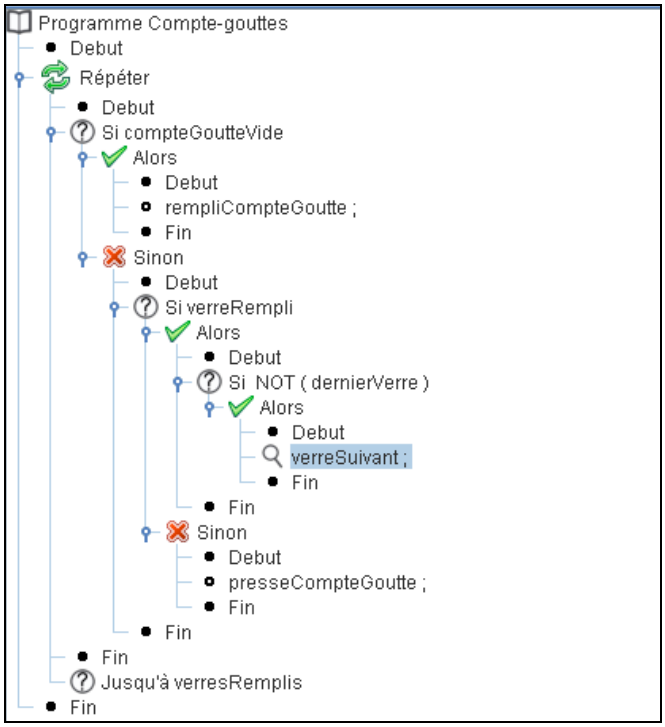


Figure 54 – Exemple de programme : algorithme de remplissage des verres (exercice 2).

2.1 Le langage manipulé

Le choix de délaisser la programmation fonctionnelle, qui conserve une certaine place dans le cadre académique, s'est fait sur la base de la population d'étudiants ciblée pour la mise en œuvre de MELBA et donc les cursus informatiques les concernant. Le choix d'adopter un style impératif et non déclaratif a été fait sur ces mêmes bases. Dans la version 1.5 de l'outil, la syntaxe s'approche d'un Pascal francisé. Elle est décrite précisément dans la section suivante, au format BNF.

2.1.1 Définitions

```
<Instruction> ::= <Simple>|<Répétitive>|<Alternative>|<Bloc>

<Simple> ::=      <Affectation>|<Appel>

<Appel> ::=       <TkPgm>[« ( »<Expression>[{« , »<Expression>}]« ) »]

<Affectation> ::= <TkVar>« := »<Expression>

<Répétitive> ::=  <RepeteJusqua>|<TantQueRepete>

<TantQueRepete>::=« Répéter »
                     « Tant Que » <Expression>
                     <Bloc>

<RepeteJusqua> ::=« Répéter »
                     <Bloc>
                     « jusqu'à » <Expression>

<Bloc> ::=        [« Variables locales : »
                     {<Déclaration>}]
                  « Début »
                  {<Instruction>}
                  « Fin »

<Déclaration> ::= <TkVar> « est un » <TkType>

<Programme> ::=   « Programme » <Nom>
                     [« Entrées : »
                        {<Déclaration>}]
                     [« Sorties : »
                        {<Déclaration>}]
                     <Bloc>

<Expression> ::= <ElmtExpression>[<Nom><Expression>]

<ElmtExpression>::= « ( » <Expression> « ) » |
                  <Nom> « ( » [<Expression> [{« , »<Expression>}]]« ) »|
                  <TkVar> |
                  <TkLitteral>

<TkType> ::=      « Booléen » | « Entier » | « Flottant » | « Caractère » |
                  « Texte » | <Type_composé> | <Type_Enuméré> | <Nom>

<Type_composé>::= « Tableau » « de » <Expression> <TkType> |
                  « Articles » « : »
                     {<Nom> « est un » <TkType>}|
                  « Pointeur sur » <TkType>

< Type_Enuméré>::= « Enumération : » {<Nom>}

<TkVar> ::=       <Nom> « ( » <Expression> « ) » |
                  « Acces » « ( » <Nom> « ) » |
                  <Nom> « . » <TkChamp> |
                  <Nom>

<Nom> ::=   <lettre>[{<alphanum>}]

<TkLitteral> ::=  {<alphanum>}

<lettre> ::=   a..z|A..Z

<alphanum> ::= <lettre>|0..9|","|"-"
```

2.1.2 Justifications

La grammaire au format BNF ci-dessus exprime la syntaxe du langage algorithmique. Le lecteur s'étonnera probablement de la décision d'y ignorer les paradigmes événementiel et orienté-objet, compte tenu de leur succès dans l'industrie. La programmation événementielle, bien que présente dans de nombreux langages commerciaux, y est presque exclusivement cantonnée à la programmation des interfaces homme-machine. De plus, elle n'y est utilisée que comme surcouche à un paradigme structuré ou objet. Enfin, l'approche « orienté-objet », bien que s'étant imposée aussi bien en université que dans l'industrie, a elle pour principal inconvénient d'élever la ***barrière d'abstraction***, dans le sens où elle requiert la compréhension de concepts tels que l'instanciation ou l'héritage… en plus de ceux de la programmation structurée (comme dans les langages C#, Java, Delphi) ou de ceux de la programmation fonctionnelle (c'est le cas pour Ocaml, par exemple). L'absence d'opérateurs d'entrée-sortie s'explique, quant à elle, par le fait que ceux-ci sont gérés en externe par un mécanisme rappelant les « bibliothèques » de la plupart des langages[1].

Le choix d'un langage algorithmique créé pour l'occasion répond à l'objectif de focaliser l'attention et les efforts de l'étudiant sur la sémantique de ce paradigme, et non pas sur les idiosyncrasies spécifiques d'un langage particulier. De la sorte, nous espérons favoriser le transfert de connaissances et de compétences vers un langage impératif « réel ».

Ce langage est ***prolixe*** et fortement structuré – de façon à faciliter la lisibilité du programme (dimension d'***expressivité***). Il s'appuie sur le français et non l'anglais, toujours dans le but de faciliter la lisibilité et la compréhension des programmes (par des étudiants francophones). Ces dimensions sont spécialement importantes pour des étudiants au style d'apprentissage connotatif.

L'interaction « graphique », avec la souris, a rendu possible ce choix. En effet, la structuration syntaxique, gérée par le système (par exemple, les nœuds Début et Fin sont ajoutés automatiquement, de même que l'indentation) et la sélection des instructions via une barre d'outils graphique permettent d'éviter les écueils majeurs liés aux notations verbeuses, à savoir lenteur de saisie (***viscosité*** par répétition), difficulté d'apprentissage des commandes, et par conséquent multiplication des erreurs lexicales (***incitation à l'erreur***).

2.2 Une édition contrainte

L'éditeur de programme de la Figure 54 n'est pas un éditeur en texte libre, à la manière d'un bloc-notes, ou de ce qui se fait dans la majorité des environnements « professionnels ». On ne peut saisir le programme qu'à travers une interface graphique très contrainte, fixant les choix possibles, dans le but d'éliminer les erreurs purement syntaxiques et de guider l'apprenant.

2.2.1 Description

La construction du programme, comme nous l'avons évoqué brièvement dans la section précédente, se fait en mode graphique. Dans une barre d'outils graphique (Figure 55 – à

[1] C'est d'ailleurs aussi de cette façon que fonctionnent la plupart des langages du commerce, tels qu'Ada, C, C++, Java…

gauche), le programmeur paramètre via des menus déroulants et des cases à cocher l'instruction qu'il souhaite ajouter au programme. Dans cet exemple, seules sont disponibles les structures « Si … Alors », « Tant que », « Répéter Jusqu'à », ainsi que les appels aux actions pragmatiques correspondant à l'exercice (PremierVerre, VerreSuivant, PresseCompteGouttes …). Puis, par un glisser-déposer, il l'insère à l'emplacement de son choix dans le programme[1] (en l'occurrence, le nœud « Début » du bloc de la boucle principale). Celui-ci est alors mis à jour : l'instruction nouvellement construite apparaît à la suite de celle sur laquelle elle a été déposée – Figure 55, à droite.

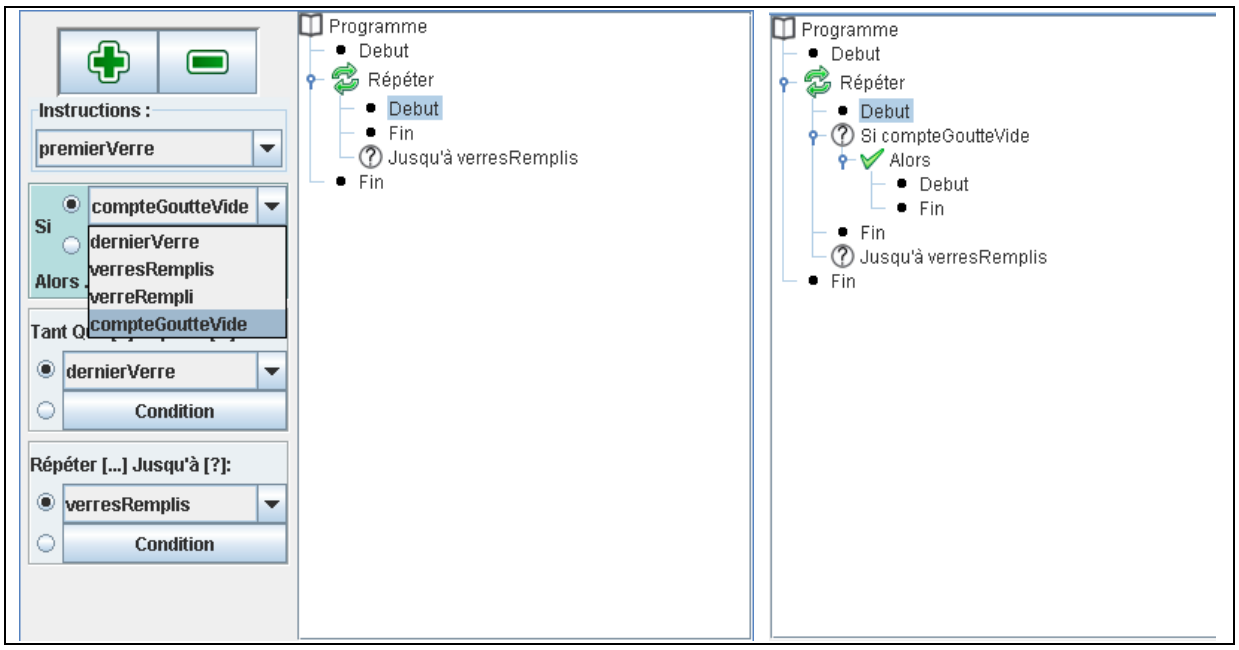


Figure 55 – Construction du programme par sélection graphique et glisser-déposer : ajout d'un « Si ... Alors », dans l'exercice 2.

Selon le paramétrage préalable fait par l'enseignant, au moment de la création de l'exercice, cette barre d'outils graphique se compose d'un nombre variable d'éléments correspondant aux concepts travaillés dans l'exercice ; ainsi les différentes structures de contrôles, les appels, les déclarations de type ou de variable seront ou non accessibles. Par exemple, pour l'exercice 8, où la tâche est un tri de tableau, la barre d'outils propose les composants de déclaration et d'affectation de variables (Figure 56). L'opération d'appel n'y figure pas, car l'exercice n'utilise pas de bibliothèque de commandes « pragmatiques[2] », et ne fait pas appel aux concepts de sous-programme ou de récursivité. De même, il n'y est pas possible de créer des variables de type article ou accès, ni de déclarer des types composés ou énumérés : seule la déclaration de variables de type tableau anonyme y est autorisée.

Les sélections de variables, de types ou d'appels de procédures et de fonctions externes ou internes sont toujours contraintes par la sélection dans un menu déroulant. Ce dernier est mis à jour dynamiquement selon le contexte du programme ou de l'expression : ainsi, on peut

[1] Alternativement, un clic sur le bouton « + » au sommet de la barre d'outils ajoute l'instruction à la suite de l'instruction courante.

[2] Ce concept sera explicité dans la section qui suit.

éviter des erreurs de nom, mais aussi certaines erreurs sémantiques (cohérence de type notamment). Par ailleurs, ce mode de sélection permet de manipuler des noms longs rapidement et sans risque d'erreurs, évitant les effets pervers de la *prolixité* – voir section 2.3.4.

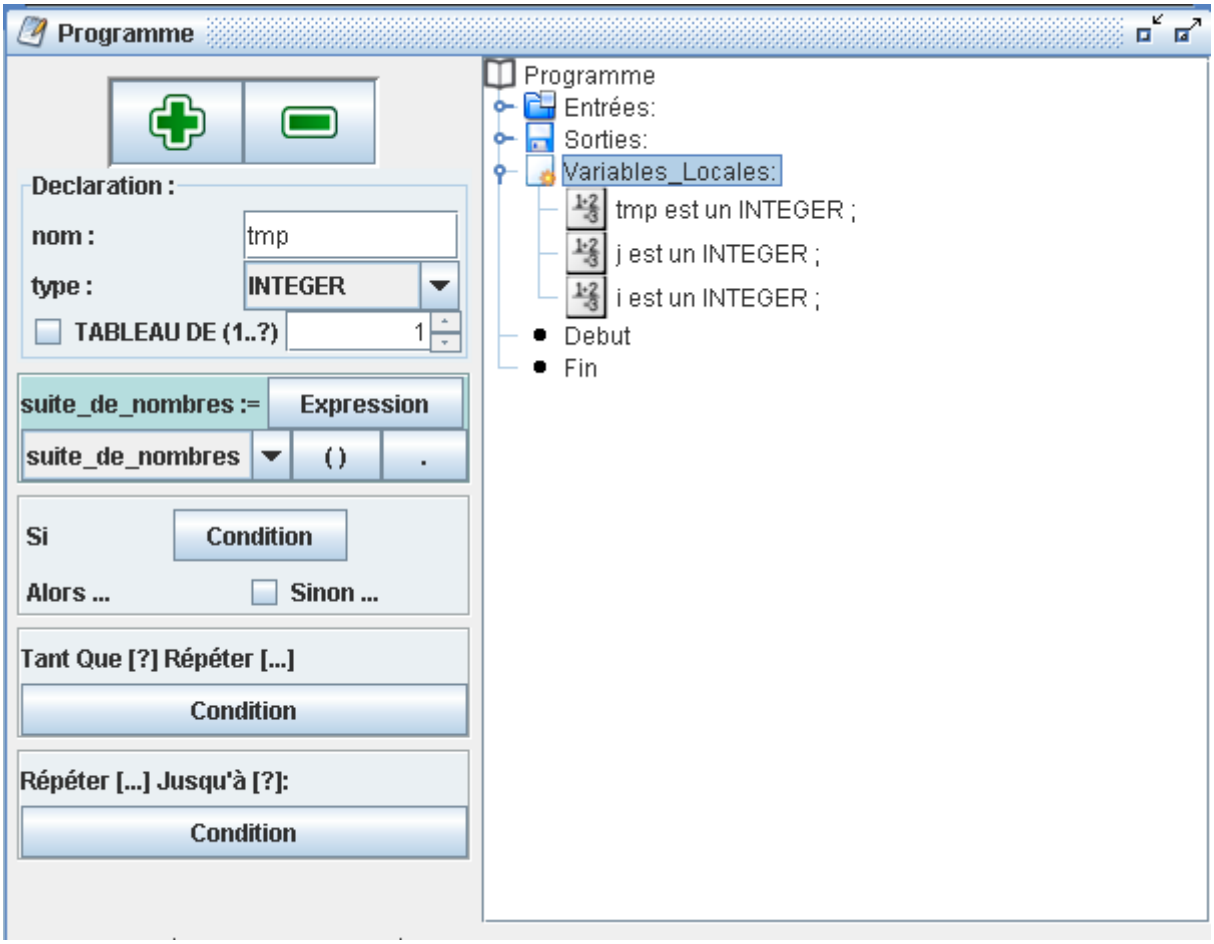


Figure 56 – Les opérateurs de la barre d'outils dans l'exercice 8 : Tri de tableau.

De même, l'insertion dans le programme est elle aussi contrainte ; on ne peut par exemple insérer d'instruction que sur un nœud « Début » ou sur une instruction existante (dans le cas d'instruction composée, sur la racine (« Si » – « Répète »)), et on ne peut déclarer de variable que sur un nœud « Variables Locales : », et de type qu'à la racine de l'arbre.

Suivant la même logique, l'édition d'expression n'est pas réalisée de façon libre dans un champ texte, mais à travers une calculatrice symbolique graphique (représentée Figure 57). Celle-ci permet d'éviter les erreurs de noms, les erreurs de parenthèses (l'expression ne peut être validée que lorsque que toutes les parenthèses ouvertes ont été fermées). Certaines contraintes d'ordre sémantique y sont également prises en compte (tel que la visibilité des variables).

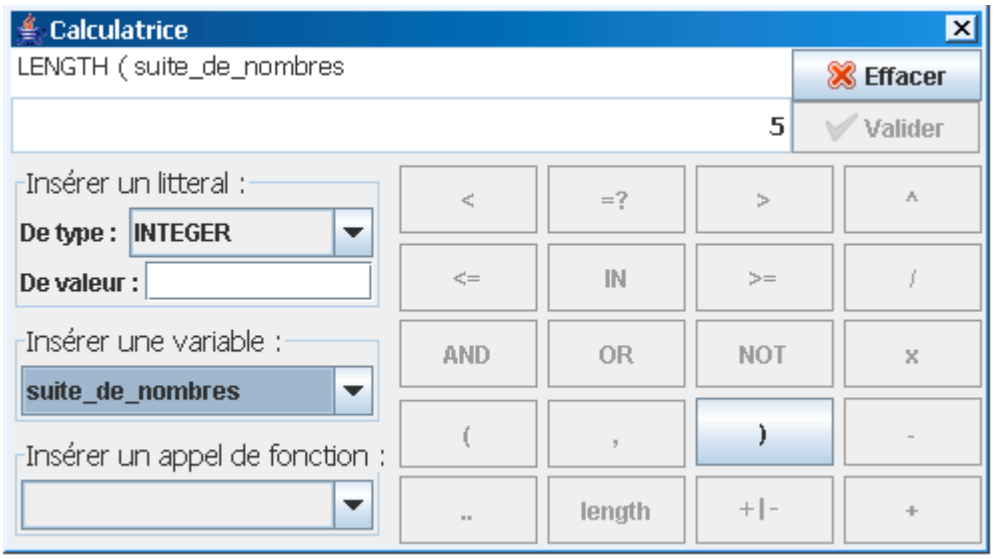


Figure 57 – La calculatrice symbolique permet la gestion des expressions en évitant des erreurs syntaxique de bas niveau.

2.2.2 Justifications et comparaison avec les outils existants

Ce mode d'édition graphique et contraint distingue l'éditeur de MELBA de celui de Grasp, qui permet de manipuler et visualiser les structures, en s'appuyant aussi sur la structure et des icônes. Bien qu'étant connue pour augmenter la *viscosité*, cette approche a le grand avantage de diminuer la distance articulatoire de sortie entre l'apprenant et le programme (en proposant un écho proactif et en limitant le nombre d'erreurs syntaxiques). C'est d'ailleurs pourquoi elle est couramment utilisée en programmation visuelle, y compris en programmation sur exemple (Pygmalion, EBP…). A contrario, Grasp s'appuie sur un éditeur de texte, ce qui lui donne les mêmes propriétés cognitives que les éditeurs de la programmation textuelle traditionnelle (voir la section 2.1.1 du chapitre précédent).

Du point de vue de l'apprentissage, l'approche sémantique par contraintes tend par conséquent à faciliter la phase d'assimilation, cruciale pour des étudiants de style d'apprentissage connotatif. Elle sert à éviter le phénomène de l'« arbre qui cache la forêt ». L'objectif de l'activité pédagogique devient alors de faire des programmes qui fonctionnent (correctement), et non pas simplement des programmes qui compilent …

De plus, comme nous l'avons évoqué dans la section précédente, en relevant le niveau sémantique d'interaction par rapport à un éditeur de texte libre (les éléments manipulés ne sont plus des caractères, mais des mots-clés – dans la barre d'outils – ou des instructions, y compris structurées – dans l'arbre du programme), ce style d'édition permet d'utiliser une syntaxe plus prolixe et proche du langage naturel, ce qui a tendance à abaisser la barrière syntaxique, ainsi que renforcer les concepts. Par ailleurs, l'écho proactif qui informe l'apprenant, pendant l'action, si celle-ci est licite ou non, peut permettre à l'étudiant de vérifier et de renforcer son assimilation des concepts. Enfin, cette approche facilite un retour sémantique immédiat sur l'exemple, dans le sens où les interactions de l'utilisateur font passer le système d'un état syntaxiquement correct à un autre état syntaxiquement correct, et donc évaluable sémantiquement. Les avantages de ce retour sémantique « actif » sont détaillés plus avant en section 3.1 de ce chapitre.

2.3 La notion d'exercice

Une difficulté que rencontrent d'une façon récurrente les programmeurs débutants est le trop grand nombre de concepts à appréhender au même moment (par exemple, Ada est réputé difficile car il introduit trop de concepts abstraits, dont certains doivent être nécessairement appréhendés dès le début, comme la transformation des entrées-sorties…).

Penser en terme d'abstractions est difficile et ne survient que tardivement dans l'apprentissage d'un nouveau domaine de savoir, ou d'une nouvelle discipline. Des systèmes qui renforcent l'abstraction créent une barrière à l'apprentissage, qui peut se révéler insurmontable pour certains étudiants. C'est pourquoi un environnement dédié à l'apprentissage doit permettre d'appréhender les concepts de façon incrémentale, en abaissant la barrière d'abstraction. Pour cela, MELBA permet à l'enseignant de construire des exercices pouvant faire appel à plusieurs niveaux d'abstraction.

Ainsi, on peut distinguer trois niveaux d'exercices dans MELBA. Premièrement, ceux qui portent exclusivement sur les structures de contrôle et le « faire-faire » : ces exercices s'appuient sur une bibliothèque d'actions et de tests *pragmatiques*, spécifiée par l'enseignant ayant conçu l'exercice, qui définissent un processeur de haut niveau. Deuxièmement, il est possible de gérer avec l'outil des exercices d'algorithmique « classiques », c'est-à-dire manipulant les concepts de structures de données, tels que les variables, les types… Enfin, on peut recourir à une approche hybride.

2.3.1 Une approche « pragmatique »

Dans cette première approche, le concepteur définit un processeur de haut niveau, en spécifiant les actions qu'il peut accomplir, et l'environnement sur lequel porte sa tâche. Pour cela, il s'appuie sur une structure conceptuelle par un patron. Une pragmatique se compose ainsi d'un ensemble de méthodes, qui spécifient quelles actions le processeur peut accomplir (procédures), et ce qu'il peut mesurer (fonctions, généralement booléennes). Par ailleurs, on définit le contexte d'exécution de ce processeur, à travers l'ensemble de variables d'état. Ce contexte est généralement associé à un environnement d'exécution graphique, comme l'exercice du compte-gouttes (Figure 58 et Figure 59), extrait des travaux pédagogiques de Charles Duchâteau (Duchâteau 2000).

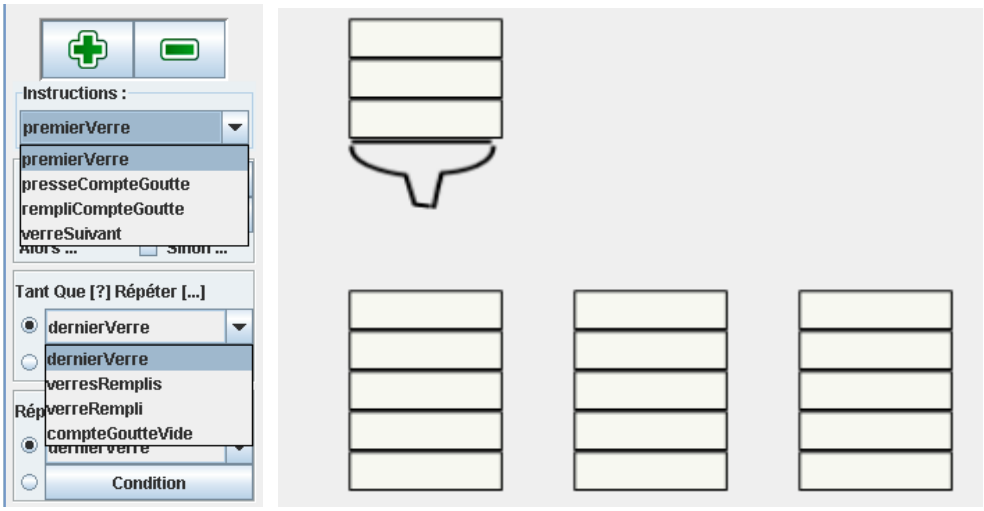


Figure 58 – La « pragmatique » d'un robot manipulant une pipette, réifiée dans l'interface de MELBA.

Ce premier niveau permet de masquer la représentation informatique interne du contexte, et donc d'abaisser la barrière d'abstraction en retardant l'introduction des concepts qui y sont associés. De plus, il est possible de paramétrer l'environnement de façon à ce que seules soient disponibles les structures de contrôle faisant l'objet d'apprentissage.

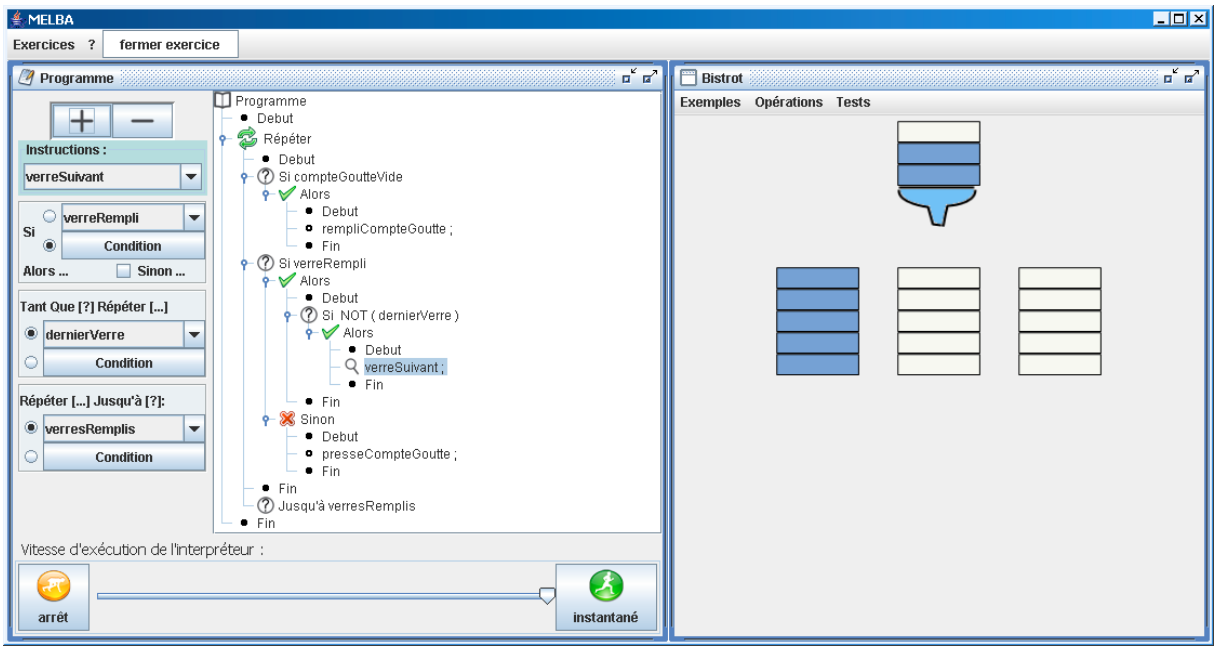


Figure 59 – L'exercice « pragmatique » d'algorithmique basé sur le compte-gouttes. Il a pour objectif pédagogique de comprendre l'imbrication de structures de contrôles, et d'apprendre à « faire faire ».

2.3.2 Un contexte « symbolique »

Alternativement, une représentation symbolique des structures de données informatiques permet à l'enseignant de construire des exercices portant sur cet autre domaine conceptuel. Un exemple d'exercice de ce type est exhibé par la capture d'écran de la Figure 60, qui est tirée d'un exercice de tri de tableau (en l'occurrence, l'algorithme implémenté est un tri à

bulle). Dans ce cas, le contexte est associé à une représentation graphique abstraite. Il est alors possible de paramétrer l'interface de l'environnement selon les concepts de structure de données que l'on souhaite aborder (types simples, tableaux, pointeurs, articles, déclaration des types, portée des variables…), en masquant les outils correspondant aux concepts n'ayant pas encore été introduits.

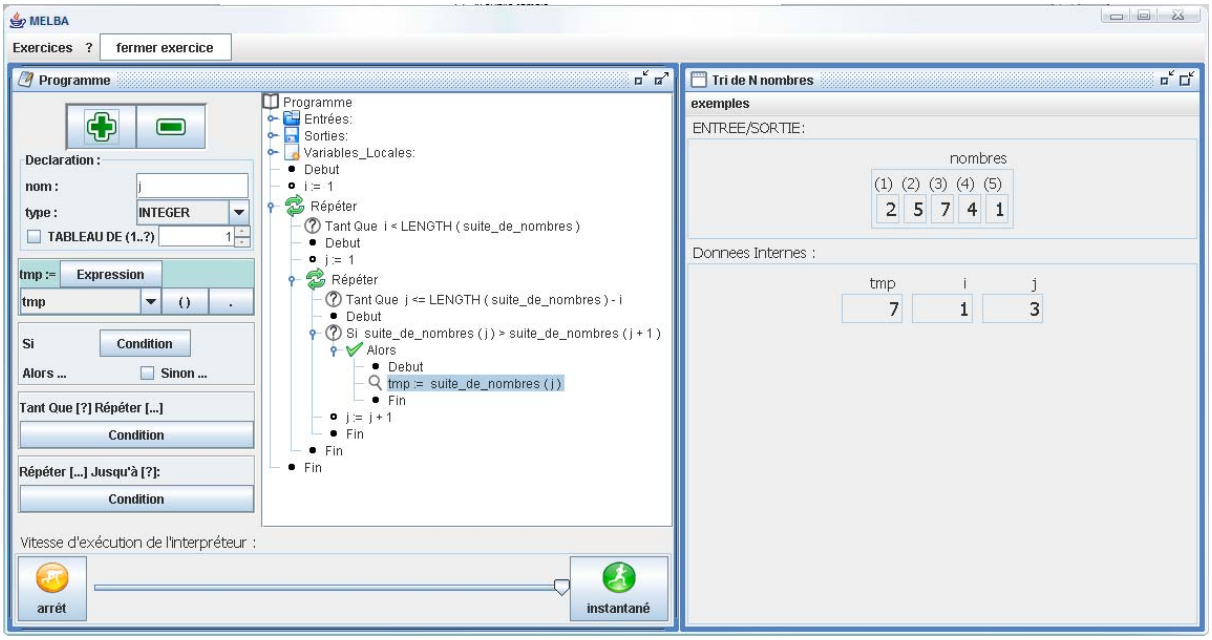


Figure 60 – L'environnement MELBA (version 1.5), avec un exercice sur les structures de données informatique (tri de tableau de taille variable)

2.3.3 Une approche hybride

Enfin, les deux approches précédentes peuvent être combinées, avec un processeur pragmatique s'appuyant sur des actions paramétrées (tel que la tortue graphique de Logo). Un tel exercice peut servir de liaison entre les deux approches précédentes, et son contexte sera représenté par à la fois une vue concrète pragmatique (la table traçante de la tortue), et une vue abstraite permettant de visualiser les variables internes abstraites : par exemple, un entier servant d'indice de boucle.

2.3.4 Buts recherchés et comparaison avec les outils existants

La mise en œuvre d'un apprentissage incrémental, s'appuyant sur un modèle décomposé en plusieurs niveaux d'abstraction nous paraît cruciale pour éviter à l'apprenant une surcharge cognitive, et devrait permettre de focaliser son attention sur des concepts fondamentaux, puis de progresser par paliers, en s'appuyant sur le socle de connaissances acquis à l'étape précédente.

Par ailleurs, il est admis que proposer des représentations concrètes des concepts et manipuler ceux-ci à travers leur représentation permet aux apprenants d'appréhender ces abstractions de façon plus concrète et de confronter la conception qu'ils en ont à la simulation qui leur en est proposée. Cette approche concrète nous paraît pertinente pour supporter la construction par

les étudiants d'un modèle de l'exécution des programmes, modèle que leur expérience quotidienne avec les ordinateurs ne permet pas d'appréhender.

Dans la littérature, il n'existe à notre connaissance aucun environnement qui soit ainsi centré sur la notion d'exercice (la plupart sont des environnements de programmation, et non d'apprentissage), ni qui permette d'aborder la programmation de façon aussi incrémentale. Il existe certes des systèmes basés sur la juxtaposition de représentation pour gérer plusieurs niveaux de concepts, comme BlueJ (Kölling 2003). Mais dans BlueJ, si les concepts liées aux modèle objet et ceux liés à la programmation structurée sont effectivement gérés dans deux représentations séparées, ces deux niveaux d'abstractions doivent toujours être appréhendés en bloc (ou presque). Cette contrainte, issue du langage – Java – se retrouve dans l'environnement, où les deux interfaces, UML et éditeur de texte, sont interdépendantes. Pour réaliser un programme, il faut toujours maîtriser ces deux niveaux d'abstraction, même si ils sont représentés à part. Il existe aussi des systèmes s'appuyant sur un modèle graphique du contexte, comme Eliot (Myller 2004) (Lattu 2000), mais tous sont des systèmes « pleine échelle » qui ne permettent pas de décomposer l'appréhension des différents concepts. Il existe enfin des systèmes basés sur un modèle limité (issus des approches basées sur un mini-langage ou un sous-langage), tels que Logo (par rapport à Lisp) mais cependant leur modèle n'est pas incrémentiel.

Le système le plus proche de MELBA dans l'esprit est donc sans doute Logo, dans le sens où la tortue graphique permet, à travers ses opérations concrètes et son domaine d'application accessible à tous, d'aborder la programmation de façon plus incrémentale (Logo est issu de LISP, et il est donc possible de faire de la programmation « abstraite » après avoir découvert un sous-ensemble de concepts dans l'univers de la tortue). Mais Logo est avant tout un *langage*, qui supporte la programmation, alors que MELBA est conçu d'entrée dans un but d'*apprentissage* de la programmation. La différence est donc que MELBA n'a pas l'ambition de tendre vers la pleine échelle, mais propose en contrepartie une décomposition plus fine du domaine conceptuel de la programmation, et donc un nombre bien plus important de paliers. Par ailleurs, il prend en compte l'enseignant en tant qu'acteur, pour choisir le niveau d'abstraction adapté à la tâche de l'apprenant à travers le paramétrage des exercices.

3 Un système fondamentalement basé sur exemple

Une autre difficulté majeure dans l'apprentissage et la maîtrise de la programmation est la perte de la manipulation directe (chapitre1, section 1.2). Celle-ci a un double effet négatif. D'une part l'absence de retour d'informations et de cas concret pour supporter la réflexion est un frein à une expérimentation active et une des sources de ce que l'on appelle le « syndrome de la page blanche » : l'étudiant est bloqué car il ne sait pas par où commencer. D'autre part, il existe un risque, en retardant ainsi l'évaluation des conceptions de l'apprenant, de le laisser développer un savoir non viable, et de créer une dissonance cognitive au moment de la validation, entre ce à quoi l'étudiant s'attendait et ce qu'il observe du comportement du programme.

Pour favoriser un apprentissage expérimental, impliquer le plus possible l'étudiant, et éviter qu'il ne construise des représentations mentales non viables, sources d'incompréhensions et de blocages ultérieurs, nous avons opté avec MELBA pour une programmation très interactive, basée sur exemple. Ainsi, lorsque l'élève lance un exercice, un exemple est systématiquement chargé au démarrage. Cet exemple est défini par l'enseignant au moment

de la création de l'exercice : il peut permettre ou interdire la définition de nouveaux exemples par l'élève, selon que la conception de jeux de tests pour valider un programme soit ou non un de ses objectifs pédagogiques. Cet exemple doit servir de base concrète à la réflexion de l'élève.

3.1 Programmation « avec exemple »

Un premier mode d'interaction entre l'élève et l'exemple dans MELBA a été la mise en œuvre d'un mécanisme de programmation « avec » exemple au niveau des composants « programme » et « exécution ». Ainsi, lorsque l'apprenant ajoute une nouvelle instruction au programme, celui-ci notifie immédiatement le composant « exécution ». Ce composant (Figure 61, à droite) se met alors automatiquement à jour, de façon à refléter l'état du programme au niveau de l'instruction courante. De la sorte la manipulation directe est rétablie ; la programmation devient interactive.

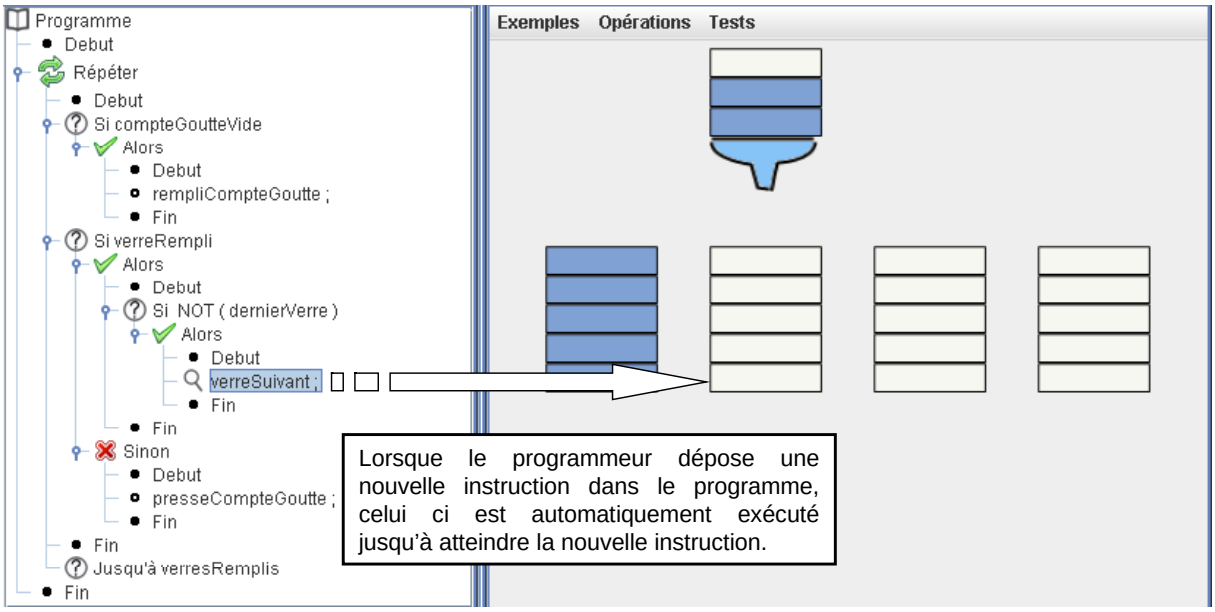


Figure 61 – Programmation « avec exemple » dans MELBA : synchronisation entre le programme et le contexte de l'exemple au moment de l'ajout d'une nouvelle instruction.

Par ailleurs, ce mécanisme de notification et de mise à jour en parallèle à l'édition s'applique aussi à l'exploration du programme par l'apprenant. Lorsque celui-ci clique sur une instruction donnée du programme, la fenêtre d'exécution est automatiquement mise à jour pour conserver la correspondance entre l'instruction courante et l'état du contexte.

Pour autant, cette approche impose des contreparties. Le fait que l'état du contexte soit automatiquement mis à jour en parallèle de l'exploration et de l'édition du programme implique qu'une erreur est déclenchée lorsque l'apprenant clique sur un endroit inatteignable avec les valeurs de l'exemple (Figure 62).

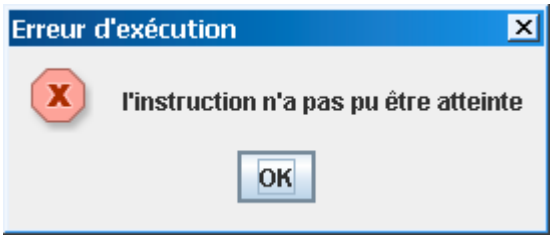


Figure 62 – Message d'erreur à l'exécution ; il survient notamment lorsque l'apprenant tente d'éditer une partie du programme inatteignable dans le cadre de l'exemple choisi.

Cette impossibilité provient soit de l'exemple, qui ne permet pas de couvrir tous les chemins d'exécution du programme, soit de l'apprenant qui édite une partie du programme alors qu'une autre partie incomplète doit impérativement être exécutée pour atteindre l'instruction sélectionnée. Ainsi, dans la Figure 63, si le programmeur cherche à éditer en premier la branche « sinon » de l'alternative « Si verreRempli » (alors que le verre en question est vide), une erreur sémantique est levée, car la branche « Alors » est atteinte en premier, et comme elle est vide, le programme boucle indéfiniment – ce qui nous a amené à mettre en place une sécurité pour stopper son exécution au bout d'un nombre fini de tours de boucle.

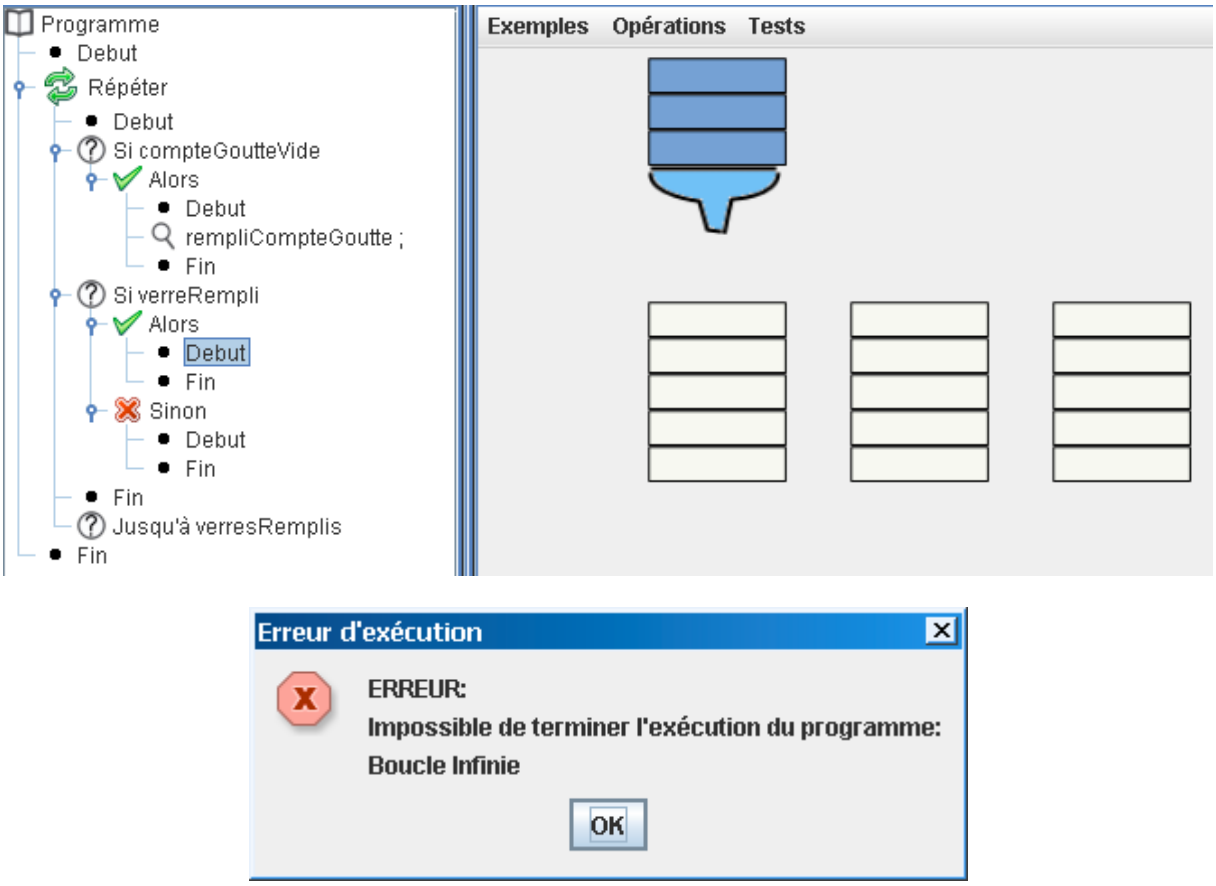


Figure 63 – Illustration des contraintes d'ordre d'édition imposées par la programmation avec exemple : déclenchement d'une erreur de type : boucle infinie.

Il existe donc des contraintes dans l'ordre d'édition lorsque l'on programme « avec » exemple : l'ordre d'édition doit correspondre à l'ordre d'exécution pour que les retours de celle-ci soit cohérent, et cela entraîne de la *prévisualisation forcée* chez l'apprenant. Il est donc possible que cette contrainte soit pénalisante pour certains styles d'apprentissage, ou

qu'un modèle mental minimal de l'exécution soit nécessaire pour programmer « avec » exemple. Notons que le système fonctionne avec exemple y compris lorsque les exercices portent sur un contexte symbolique, et non pragmatique.

3.2 Programmation « sur » exemple

Alors que la programmation « avec » exemple correspond à une approche « dénotative » de la programmation (de l'abstrait au concret), l'environnement permet également à l'élève de programmer « sur » l'exemple (approche connotative) dans les exercices « pragmatiques ».

Dans cette approche, les actions du processeur sont déclenchées par manipulation directe des objets du contexte pragmatique, et insérées dans le programme à la suite de l'instruction courante – à la manière d'un enregistreur de macro. Par exemple, dans l'exercice du compte-gouttes, un glisser-déposer du conteneur (JPanel) du compte-gouttes déclenche l'action « premierVerre ou verreSuivant », le clic souris sur une partie du compte-gouttes lance « remplitCompteGouttes » ou « presseCompteGouttes » selon que celle-ci est vide ou pleine (Figure 64). L'implémentation de cette technique est bien évidemment différente selon la pragmatique et la représentation graphique.

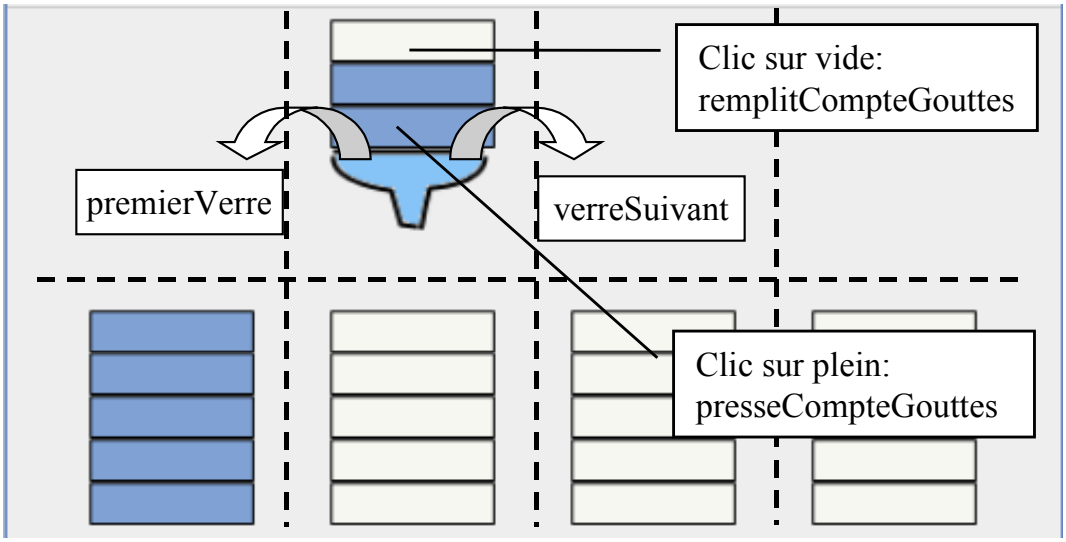


Figure 64 – Implémentation de la programmation sur exemple dans l'exercice du compte-gouttes.

Il devient alors possible de réaliser un programme « à rebours » : l'étudiant réalise la tâche dans l'interface à manipulation directe. Il obtient ainsi un programme séquentiel, qui fonctionne pour cet exemple (mais lui seul). Il construit alors un programme « générique » en factorisant la séquence d'action obtenue avec les opérateurs de structures de contrôle. Cette tâche de factorisation peut se faire « avec exemple », en utilisant les valeurs du contexte graphique comme un test de non-régression, réalisé automatiquement en parallèle. Puis, pour vérifier que son nouveau programme s'applique bien dans tous les cas, l'apprenant peut le tester sur d'autres exemples.

Une possibilité que nous n'avons pas explorée dans le développement de la version courante de l'outil (1.5), mais que nous avons envisagée dans le cadre d'un éventuel support de l'auto-apprentissage, consiste à utiliser un agent logiciel comme tuteur pendant cette phase de factorisation. Cet agent s'appuierait sur des algorithmes d'inférence connus dans la littérature

(comme ceux développés dans (Nevill-Manning 1997), (Nevill-Manning 1997), ou (Lau, Wolfman et al. 2001)) pour détecter des répétitions (lesquelles peuvent contenir des éléments facultatifs qui seraient modélisées par des structures alternatives) et ainsi guider l'étudiant qui en éprouverait le besoin lors de cette tâche de généralisation.

3.3 Justifications

La programmation « basée sur exemple » est un paradigme particulièrement séduisant dans l'optique de l'initiation à la programmation pour deux raisons essentielles : il permet à l'étudiant de raisonner sur des exemples concrets, et il rétablit la manipulation directe, permettant de ce fait une *évaluation progressive* du comportement du programme. Cette dimension est cruciale pour permettre la construction et la consolidation des modèles mentaux par l'apprenant, et incite à l'expérimentation active, minimisant le syndrome de la page blanche.

Toutefois, les différentes techniques qu'elles met habituellement en œuvre ne vont pas sans contreparties, notamment en *prévisualisation forcée*, voire en *degré d'engagement*. C'est en particulier vrai pour les systèmes « par démonstration », qui s'appuient sur un fonctionnement du type enregistreur de macro, et où l'absence de représentation abstraite globale rend difficile la correction des erreurs ou la réutilisation de programmes existants. Cela nous a amené à réfléchir à une implémentation qui minimiserait ces travers, assez incompatibles avec notre idée directrice de permettre une initiation à la programmation par une approche véritablement expérimentale. Dans cette optique, minimiser le *degré d'engagement* est crucial, car celui-ci a tendance à induire un comportement passif.

C'est pourquoi nous avons opté pour une implémentation s'appuyant sur une approche essentiellement « avec exemple », centrée sur les concepts, et basée sur plusieurs niveaux de modélisation. Le programme peut être directement manipulé, un éditeur contraint sert à minimiser les erreurs syntaxiques (peu significatives et rarement formatrices), et de multiples garde-fous, au niveau sémantique ont été implantés. L'erreur devait être rattrapable facilement, et source d'apprentissage. Cela nous a amené à concevoir le premier éditeur de programmes « avec exemple » réellement fonctionnel, à notre connaissance.

Cet environnement est également original en ce qu'il combine les deux approches « avec » et « sur » exemple, pour s'adapter à un public le plus large possible, ayant des styles d'apprentissages différents. Pour supporter tous les stades du cycle expérimental, l'approche sur exemple, *connotative*, et l'approche avec exemple, *dénotative*, sont complémentaires.

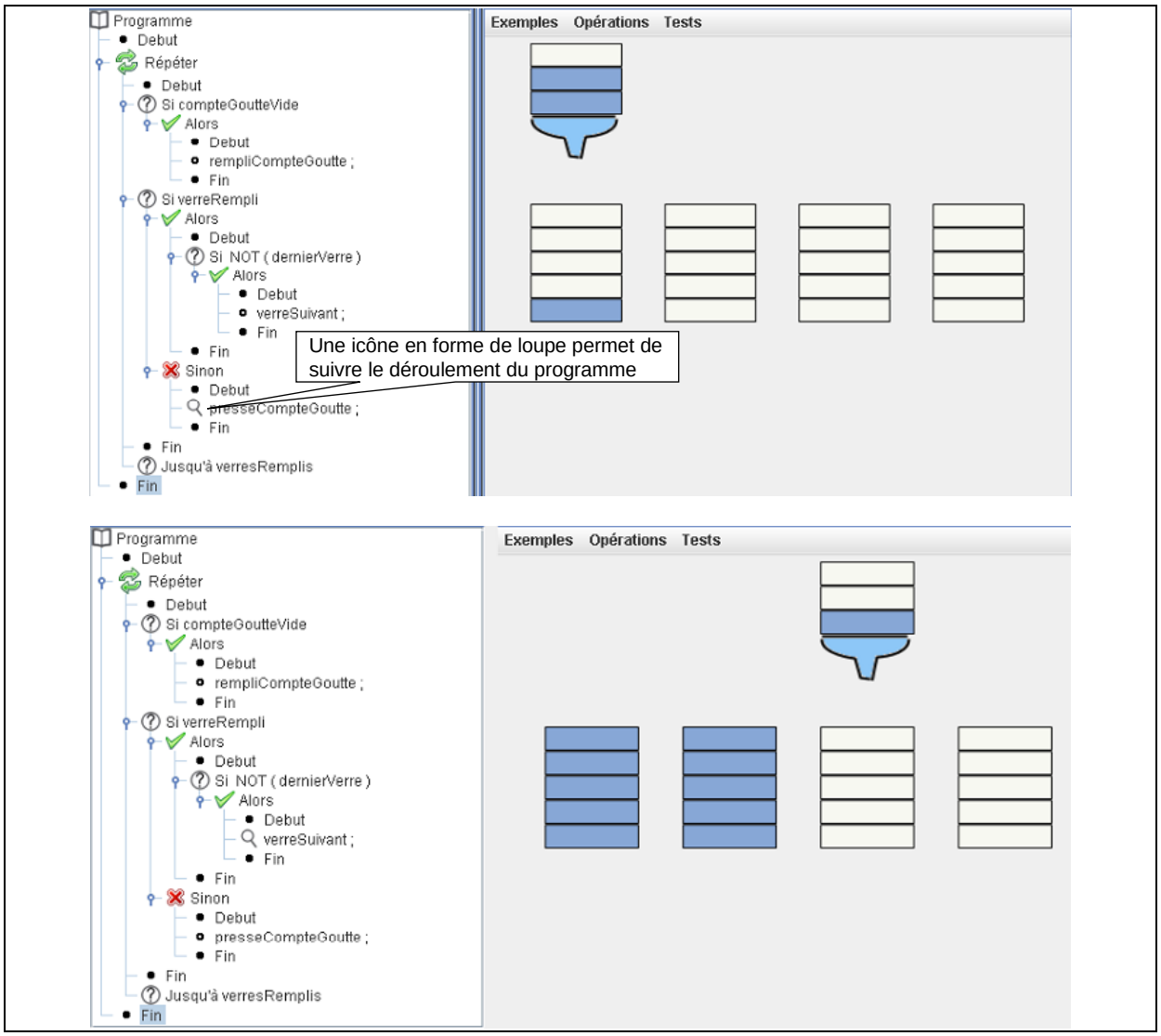
3.4 Des fonctions de visualisation de programme

Le retour immédiat fourni par l'exemple doit permettre à l'apprenant de construire ses modèles mentaux de façon expérimentale, en les testant et en les complétant au fur à mesure. Cependant, en raison des contraintes sur l'ordre d'édition imposées par l'approche sur exemple représentant, malgré nos efforts pour la minimiser, un risque de blocage à l'utilisation pour certains étudiants, il nous a paru nécessaire de proposer d'autre outils, en complément ou en concurrence. Ainsi, à travers l'analyse de l'activité des apprenants, nous espérons déterminer un profil d'utilisateur de la PsE (niveau de connaissance, style cognitif …), et voir dans quelle mesure ce retour « actif » et les retours différés « classiques » peuvent se compléter selon la tâche de l'utilisateur (par exemple, l'utilisateur utilise l'exemple en

conception, mais l'animation de programmes ou la trace pour étudier un programme préexistant).

3.4.1 L'animation de programmes

L'animation de programmes ne correspond pas à une démarche active (comme l'approche sur exemple) mais à une démarche d'observation plus passive. D'un coté, l'apprenant teste ses assertions sur le comportement du programme, de l'autre il n'a pas d'idée préconçue et recherche une explication. Nous pensons donc que ces deux techniques peuvent se compléter. La Figure 65 illustre le processus tel qu'il a été implanté dans MELBA : l'interpréteur exécute pas à pas la totalité de l'algorithme et l'exemple est mis à jour à chaque pas. Une icône de loupe permet de repérer l'instruction courante, et la vitesse de l'animation est paramétrable.



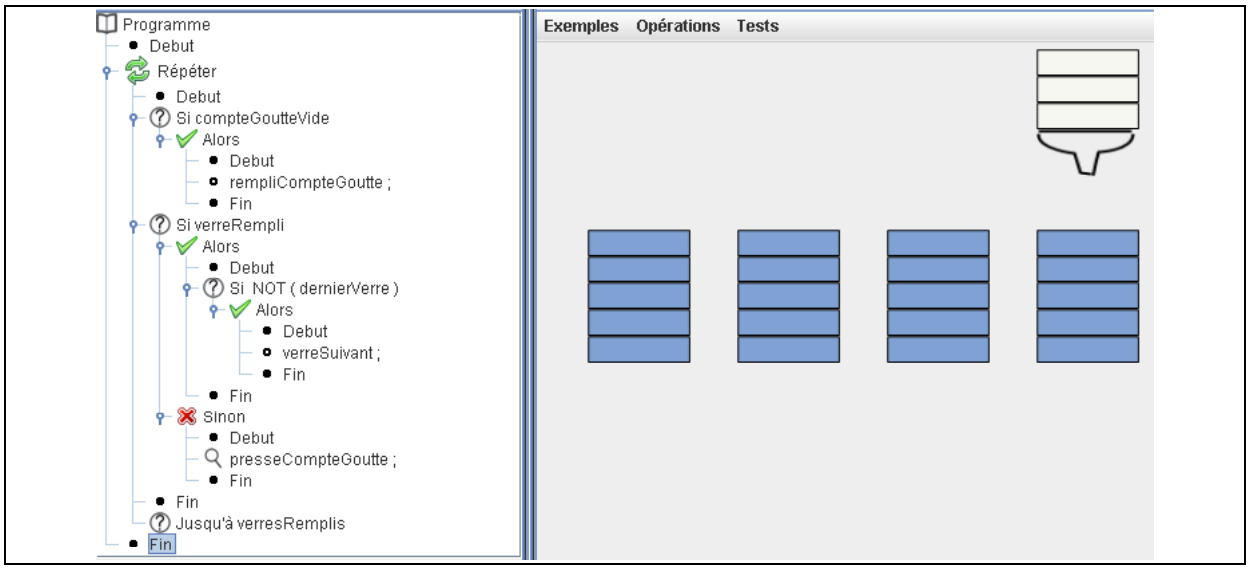


Figure 65 – L'animation du programme du compte-gouttes, à travers plusieurs étapes.

3.4.2 Le contrôle du mode

Dans cette optique, il est important de donner le choix à l'utilisateur entre une approche interactive, et une approche classique. Le système propose intrinsèquement trois fonctionnalités : l'édition « simple », l'animation du programme pas à pas, et la programmation basée sur l'exemple. Nous avons donc développé dans le cadre de l'outil un composant interactif permettant de basculer d'un mode de fonctionnement à un autre.

Initialement, et par analogie aux systèmes physiques (magnétoscopes) ou aux enregistreurs de macros, ce panneau de commande avait la forme décrite Figure 66.

Figure 66 – Les différents modes d'interaction avec l'environnement MELBA.

Le système peut y être piloté suivant 3 modes : avec exemple (RECORD), avec exemple animé (PLAY) et sans exemple (STOP). En mode « Record », le système a un mode de fonctionnement « basé sur exemple ». Le mode « play » correspond à l'animation, et le mode « stop » à un style de programmation classique, dont l'exécution n'est pas immédiate. Les deux boutons de gauche servent à revenir ou avancer d'un pas.

Malheureusement, cette métaphore est apparue à l'usage peut claire pour les étudiants. D'une part parce que le symbole « record » et l'approche sur exemple n'ont pas de lien sémantique évident d'autre part parce que le symbole stop était associé à un mode où le programme pouvait être modifié : c'est l'exécution immédiate qui y est « stoppée ». Cela nous a amené à réorganiser ce panneau, comme le montre la Figure 67. Il n'y a plus de référence aux magnétoscopes ou enregistreurs de macros. Le composant agit sur la vitesse d'animation du code, lorsque le programme est exploré ou mis à jour. Lorsque la barre de vitesse est totalement à gauche, le système se comporte de façon classique, sans exemple : il n'y a alors

pas d'interprétation en réponse aux actions sur le programme. Sinon, le système exécute automatiquement le programme de sorte que l'état dans la fenêtre de l'exemple corresponde à la position du curseur dans le programme. Cette exécution peut être animée, ou instantanée (seul le résultat est affiché, conformément au fonctionnement avec exemple standard).

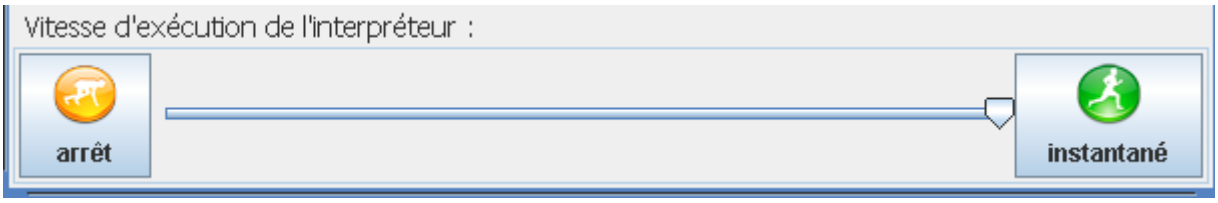


Figure 67 – Le basculement de modes d'utilisation dans l'interface des versions 1.4 et postérieures.

3.4.3 La visualisation de l'historique

En complément de ces différentes représentations dynamiques (elle se concentrent sur la description d'un état particulier lors du déroulement du programme), nous avons étudié pour MELBA une représentation statique de l'exécution du programme, basée sur le trace d'exécution. Ce composant a reçu pour nom l'historique de l'exécution. La Figure 68 illustre sa présentation.

Le principe de fonctionnement de l'historique est à la base celui d'une « time line ». La trace des différentes instructions simples exécutées (1) forme la ligne du dessous, où le temps s'écoule de gauche à droite. La dernière instruction exécutée se trouve ainsi à droite, grisée (2). A cette dimension temporelle se rajoute une dimension structurelle, dans l'axe vertical. Chaque instruction simple est recouverte par la hiérarchie de ses structures de contrôle. On remarque donc que l'instruction « remplitCompteGouttes » est à l'intérieur de la branche alors de la conditionnelle « Si compteGouttesVide » (3), elle même à l'intérieur de la boucle principale « Répéter … jusqu'à verresRemplis ». Les fins d'itérations sont marqués par des blancs (5), qui premettent de repérer les différents tours de boucle. Cette structuration a pour objectif de faire le lien avec le composant programme, en permettant à l'utilisateur de repérer plus facilement la position de chaque élément de la trace (en effet, plusieurs instructions peuvent avoir le même texte, et que les structures de répétitions créent plusieurs instances d'une instruction donnée). Un code de couleur a également été mis en place pour augmenter la lisibilité de cette structure : les instructions simples y sont ainsi figurées en bleu, (gris pour l'instruction courante), les alternatives en rouge, et les itérations en vert.

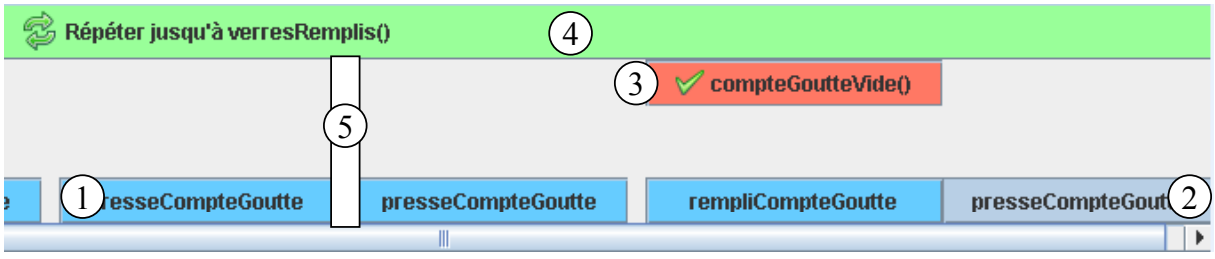


Figure 68 – Trace augmentée de l'exécution du programme, dans MELBA.

Dans cette représentation, les instructions simples (en bas en bleu) sont des composants dynamiques. Le programmeur peut cliquer sur l'une d'elles, et à ce moment là, la fenêtre

d'exécution est mise à jour pour afficher l'état correspondant à l'instruction sélectionnée. De même le composant programme est notifié et positionne l'icône de loupe sur cette instruction. Il est donc possible grâce à ce composant d'explorer véritablement la trace, en revenant à un état précis de l'exécution, comme par exemple à la fin de la énième itération d'une structure répétitive.

Par ailleurs, l'historique est notifié à chaque interaction sur le programme, ce qui permet sa mise à jour dans le cadre du paradigme de programmation « basée sur l'exemple ». Comme le montre la Figure 69, l'insertion de l'instruction « verreSuivant » dans le programme de l'exercice du compte-gouttes a pour effet la mise à jour de l'historique : on note à droite l'apparition de « verreSuivant » sous les deux structures alternatives « Si verreRempli Alors » et « Si NOT dernierVerre Alors ».

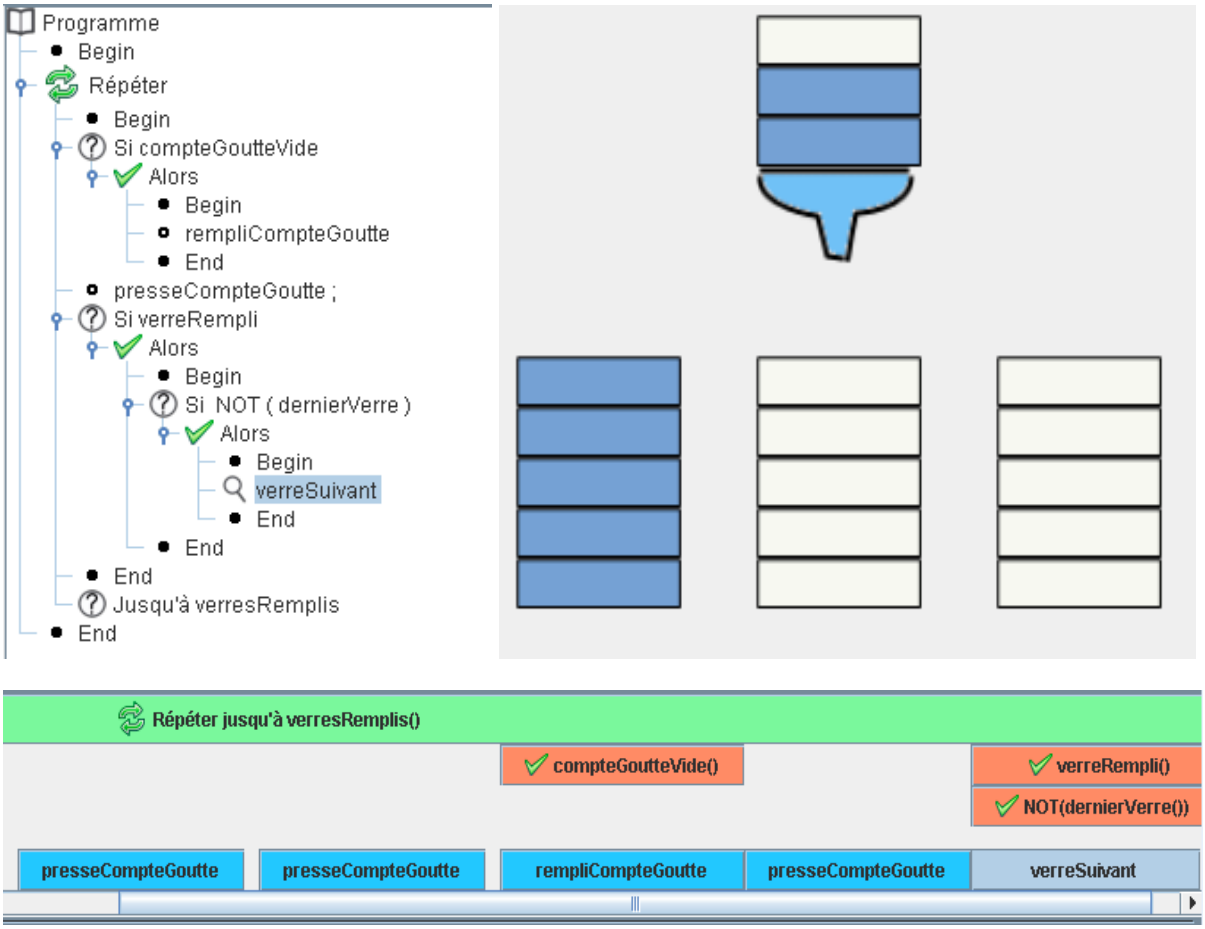


Figure 69 – Mise à jour des différents composants, dont l'historique, à l'insertion de l'instruction « verreSuivant » dans le programme.

3.4.4 Justifications

D'un point de vue pédagogique, ces différents outils ont pour objectif de faciliter la compréhension du fonctionnement du programme, en fournissant des services complémentaires, dans le but d'être utiles à des étudiants aux styles d'apprentissage variés. En effet, la programmation avec exemple, l'animation de programmes, et la trace d'exécution se complètent dans le sens où elles s'adaptent à différentes tâches cognitives, à différents stades du processus expérimental.

Alors que l'attrait principal de la programmation avec exemple est de permettre à l'apprenant, grâce au rétablissement de la manipulation directe, de *vérifier* en temps réel sa compréhension, l'animation de programme a au contraire des vertus *explicatives*, alors que la trace peut servir de base *d'analyse*.

Selon le degré de compréhension de l'étudiant, ou selon son style d'apprentissage, l'apprenant tendrait donc à préférer telle ou telle technique, et proposer un ensemble d'interactions qui supportent des tâches différentes permettrait de couvrir un large éventail d'apprenants. On peut ainsi émettre l'hypothèse que la programmation avec exemple toucherait la phase d'expérimentation, et plus spécifiquement les « *accommodateurs* » qui s'appuient beaucoup sur le *bricolage* (Papert 1980) (Ben-Ari 1998) et ont des cycles essai-erreur très courts, alors que l'animation et la trace, supportant davantage une phase d'observation plus à posteriori, devraient davantage convenir à des étudiants « *divergents* » ou « *assimilateurs* ».

Pour autant, ces dernières conjectures restent à étudier et à valider ou invalider expérimentalement, tant il est vrai que les études de psychologie cognitives sur l'impact des outils interactifs dans l'apprentissage de la programmation sont rares.

4 Un environnement modulable et évolutif en fonction des objectifs pédagogiques

Après avoir présenté la structure de l'environnement MELBA, ainsi que le choix des outils d'interactions entre l'élève et l'apprenant, nous décrivons les possibilités d'appropriation de l'environnement par l'enseignant, à savoir la conception et l'intégration de nouveaux exercices, « à la main » ou par un prototype d'environnement auteur.

4.1 Conception d'un exercice : programmation en java

Un exercice, dans l'environnement MELBA, est un agent que l'on connecte à la structure de l'environnement. Des mécanismes d'introspection assurent alors automatiquement le lien avec le reste de l'environnement, en personnalisant la barre d'outils au lancement de l'exercice, et en mettant à jour menu « exercices ». Cela implique qu'un exercice possède une structure logique fixe. Il se compose :

- D'un processeur « pragmatique », lui-même composé :
  - De la liste des actions reconnues par le processeur (telles que, « verreSuivant() » ou « remplitCompteGouttes() »).
  - D'une liste de fonctions permettant de renseigner sur l'état du contexte, et en particulier de tests logiques permettant la construction de conditionnelles (comme « verresRemplis() ➔ boolean »)
  - A chaque action élémentaire peut être associée une garde, sous la forme d'une expression booléenne (par exemple, pour exécuter « presseCompteGouttes() », on doit vérifier la garde « NON compteGouttesVide() ET NON verreRempli() »)
- D'un programme qui sera support de la tâche pédagogique, lequel peut être éventuellement prérempli au moment du chargement de l'exercice (selon la nature de la tâche : analyse, correction d'erreurs, composition …).
- D'un ou plusieurs exemples, caractérisés par la valeur de leurs attributs (dans l'exercice du compte-gouttes, ces attributs sont le nombre de verres, la contenance des verres, et la contenance du compte-gouttes).
- Des attributs de l'exercice lui-même, qui renseignent sur la personnalisation de l'interface globale de MELBA : peut-on ou non créer de nouveaux exemples, déclarer des variables, et si oui de quels types (simples, tableaux, articles, définis, accès ….)

4.1.1 Modèle d'architecture du composant

MELBA a été développé en Java (langage de programmation objet) et s'appuie sur un modèle de données EXPRESS (cf. Annexes). MELBA est construit sur l'architecture PAC-Amodeus (Coutaz, Nigay et al. 1995), (Nigay 1994). Celle-ci est un modèle basé sur la notion d'agents composés de trois classes maximum à chaque fois : Présentation, Abstraction et Contrôle (Figure 70 –a).

- La **présentation** (P) définit le comportement de l'agent par rapport à l'utilisateur (entrées et sorties).
- L'**abstraction** (A) définit les aspects conceptuels de l'agent et fait le lien avec le modèle conceptuel EXPRESS.
- Le **contrôle** (C) relie entre eux les deux autres facettes. Il gère l'état de l'agent et la communication avec d'autres agents PAC. Il s'agit donc de la partie publique, accessible de l'extérieur.

Les agents sont organisés en hiérarchie arborescente. Un agent parent a la responsabilité des agents fils (Figure 70 –b).

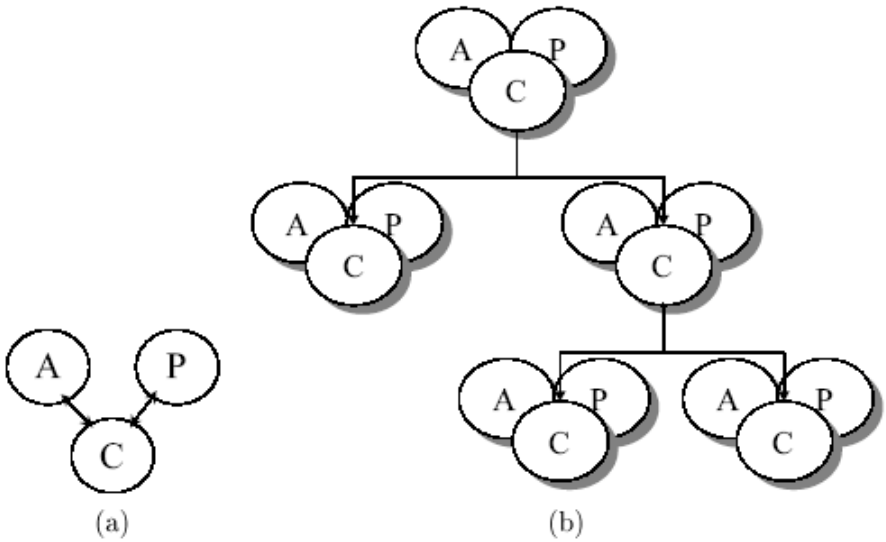


Figure 70 – Structure d'un agent PAC (a) et organisation hiérarchique des composants (b).

Dans notre implémentation, chaque facette (A, C, P) correspond à une classe Java, et nous avons conçu deux classes abstraites PragmatiqueControle et PragmatiqueAbstraction, ainsi que la classe utilitaire ReflectAbstraction pour structurer et encadrer la conception de l'agent. Dans la suite de cette section, nous décrivons le contenu et la structure de ces différentes classes Java.

4.1.2 La facette Abstraction

La facette Abstraction a pour rôle principal d'implémenter les capacités du processeur, et stocke les caractéristiques de l'exercice. Elle fait également le lien avec le composant de NoyauFonctionnel, permettant à l'interpréteur d'appeler les actions et les tests de la

pragmatique comme une bibliothèque. Pour cela, elle implémente un certain nombre de méthodes permettant d'instancier le noyau fonctionnel et de gérer le déroulement des actions.

Dans le but de lier automatiquement le processeur pragmatique défini par l'enseignant à l'interpréteur, nous avons écrit une classe qui fournit, à travers les mécanismes d'introspection de java (réflexivité) un accès à cette couche « modèle ». Cette classe possède une méthode « agit » qui peut être accédée par les autres composants de Melba pour déclencher l'exécution des actions du processeur. De même, cette classe permet, toujours par introspection de la facette « A » de l'exercice, de renseigner l'environnement Melba sur les noms des actions exécutables par le processeur.

```
public class ReflectAbstraction {
  private Method[] actions;
  private Method[] tests;
  ...
  public ReflectAbstraction (PragmatiqueAbstraction a) {
    Method[] methodes = a.getClass().getMethods();
    ...
  }
  public void agit(String action) throws GardeInvalide {
    for (int i = 0; i < actions.length; i++) {
      if (action.equals(actions[i].getName())) {
        try {
          actions[i].invoke(obj, null);
        }
      ...
      }
    }
   ...
}
```

Pour qu'elle puisse être ainsi introspectée, l'implémentation de la facette abstraite doit suivre certaines règles, à savoir :

- les actions du processeur sont des méthodes publiques de type de retour void,
- les tests du processeur sont des méthodes publiques de type de retour boolean,
- il n'y a pas dans la classe d'autres méthodes publiques « void » que les actions, ni d'autres méthodes publiques « boolean » que les tests.

D'autre part, l'Abstraction de l'exercice doit hériter de la classe abstraite « PragmatiqueAbstraction », qui possède les attributs et méthodes « protected » qui permettent de gérer les paramètres de l'exercice et la construction ou le chargement de nouveaux exemples. Il est donc possible d'implémenter facilement cette facette sans connaissances préalables sur le modèle des exercices et des programmes, à condition de s'accommoder de ces limitations.

4.1.3 La facette Contrôle

Dans l'architecture PAC, la facette Contrôle a un rôle privilégié : celui de connecteur avec les autres composants. Elle permet ici le pilotage de la pragmatique de façon externe par l'environnement élève. Pour ce faire, elle hérite d'une classe abstraite « PragmatiqueControle », qui définit une interface de callback des différentes actions du processeur par leur nom :

```
public abstract void operation(String nom_op, Stack inputs ,Stack outputs)
throws GardeInvalide;

public abstract boolean test(String nom_te, Stack inputs)  ;

public abstract String fonction(String nom_fct, Stack inputs);
```

Les actions du processeur possèdent souvent une garde (c'est-à-dire une précondition – par exemple l'action « verreSuivant » de la pragmatique du compte-gouttes nécessite que le compte-gouttes ne soit pas sur le dernier verre). Cette garde est gérée par un mécanisme d'exception au moment du callback et permet de bloquer l'exécution en déclenchant une erreur sémantique, comme par exemple :

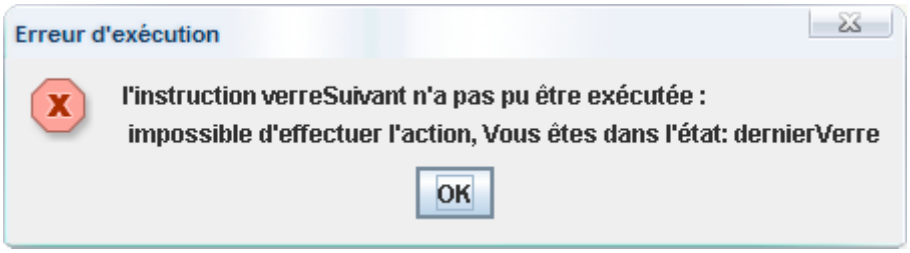


Figure 71 – génération d'une erreur sémantique à partir des préconditions des actions.

Par ailleurs, de façon à permettre le mécanisme de programmation « avec exemple », le contrôle doit fournir des fonctionnalités de mise à jour (pour actualiser la présentation) et de chargement des exemples (pour pouvoir, dans l'exemple du compte-gouttes réexecuter le programme avec un nombre de verres différent). Une méthode de réinitialisation permet de revenir et l'état de base de l'environnement (en l'occurrence, cela consiste à vider tous les verres et le compte-gouttes, et à positionner celui-ci sur le premier verre)

```
public abstract void maj()

public abstract void initialise()

public abstract void nouvelExemple(Vector attributs)

public abstract void charger(int num_ex)
```

Les « attributs » de la pragmatique, dont il est question dans la méthode nouvelExemple, sont les paramètres d'entrée qui définissent l'état initial de l'environnement : ici, le nombre de verres, leur taille, et la taille du compte-gouttes.

Pour permettre la programmation « sur exemple », la facette contrôle possède également une méthode d'insertion d'instruction au point courant du programme, définie dans la classe abstraite, qui notifie environnement, et actualise la présentation :

```
public void insertionDansProgramme (String nom_action,
                                    Stack entrees,
                                    Stack sorties ){
...
}
```

Enfin, comme le composant Pragmatique référence l'environnement d'exécution, il propose des méthodes d'introspection et d'exploration de la trace, qui permettront l'actualisation du composant d'historique :

```
public abstract String [] getFilm() ;
public abstract void goto_inst(int pos_instruction);
```

4.1.4 La facette Présentation

La facette présentation est une fenêtre interne Swing (JinternalFrame). C'est la facette qui varie le plus d'un exercice à l'autre. Elle suit un squelette, avec des constructeurs, les menus génériques et les méthodes d'initialisation et de mise à jour permettant le pilotage de la pragmatique, telles que :

```
...
public void initialise (Vector attributs /*variables d'initialisation*/) {
...
public void maj( Vector etat /* variables modélisant l'état du contexte*/)
{
// Par exemple, dans l'exercice du compte-gouttes, la position du compte
gouttes, le contenu du compte gouttes et le contenu de chaque verre.
...
}
```

La partie graphique de la facette présentation n'est pas spécifiée par le squelette, car elle varie du tout au tout d'un exercice à l'autre. L'enseignant est donc totalement libre dans ses choix de développement, et peut éventuellement s'appuyer sur un concepteur d'interface (GUI-builder).

De plus, un ensemble d'écouteurs de bas niveau de type « MouseListener » doit être développé pour supporter la programmation « sur exemple ». Les conventions de dialogue variant d'une pragmatique à l'autre, cette partie n'est pas non plus prise en compte par le squelette.

4.2 Un prototype d'environnement auteur

De nombreux morceaux de code dans ces agents sont donc soit toujours identiques, soit suivent un modèle unique qui est paramétré par les variables de l'exercice, ce qui permet de générer automatiquement les facettes A et C à partir d'un environnement auteur. La présentation graphique, quant à elle, n'est pas automatisable mais le squelette peut être généré par l'environnement. Ces structures communes nous ont permis la réalisation d'un prototype d'environnement auteur, à partir de ces patrons d'exercices (Figure 72).

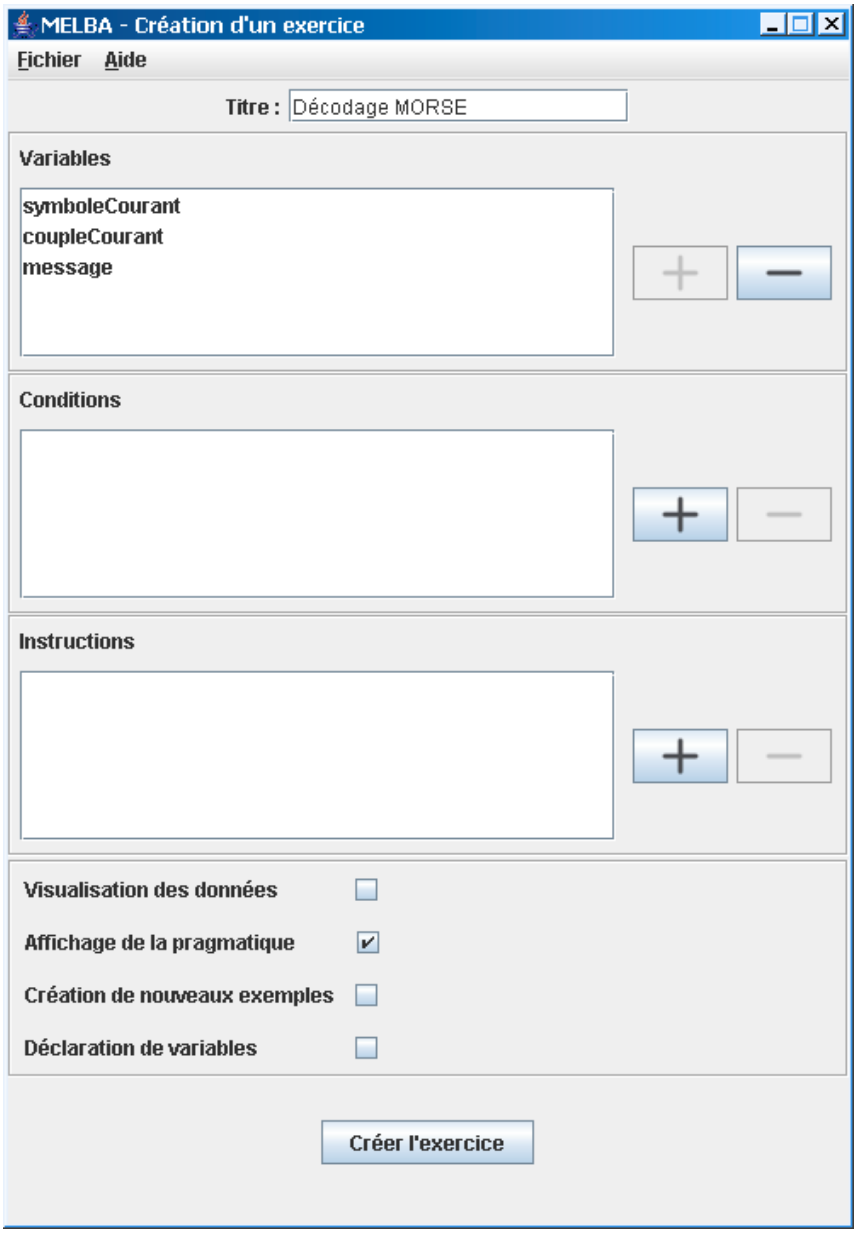


Figure 72 – Ecran principal du prototype d'environnement auteur d'exercices pour MELBA.

Ce prototype permet de créer deux types d'exercices :

- des exercices portant sur uniquement l'algorithmique et les flots de contrôle (auquel cas, l'auteur aura créé des instructions et des conditions, coché la case « affichage de la

pragmatique » et décoché les cases « visualisation des données » et « déclaration de variable »),

- des exercices portant sur les variables et les structures de contrôles (auquel cas, il n'est pas nécessaire de fournir des actions ou des conditions élémentaires, la visualisation générique de données serait activée, ainsi que la possibilité pour l'étudiant de déclarer ses propres variables).

Ce prototype permet ainsi de paramétrer quels composants utiliser, et quels parties de la barre d'outils sont utilisées dans la résolution de l'exercice. Ainsi les cases à cocher permettent d'activer ou de désactiver la visualisation pragmatique, la visualisation symbolique, les appels, les affectations et les déclarations (et ce, évidemment, sans éditer le code des agents concernés).

L'autre avantage de cet environnement auteur est qu'il ne requiert que la connaissance de la structure d'un exercice (variables + instructions + tests) et la connaissance du langage algorithmique utilisé par l'élève. Il n'est donc pas nécessaire de connaître l'architecture PAC correspondant à l'organisation des composants (dont le composant d'exercice), ni *a fortiori* les API du composant NoyauFonctionnel. Comme le montre la Figure 73, le corps des instructions et des conditions élémentaires d'une pragmatique y sont spécifiés avec le même langage algorithmique utilisé par l'étudiant. Parallèlement, il permet l'implémentation des trois types d'exercices (approches pragmatique, symbolique, hybride), contrairement à l'utilisation de la classe réflexive.

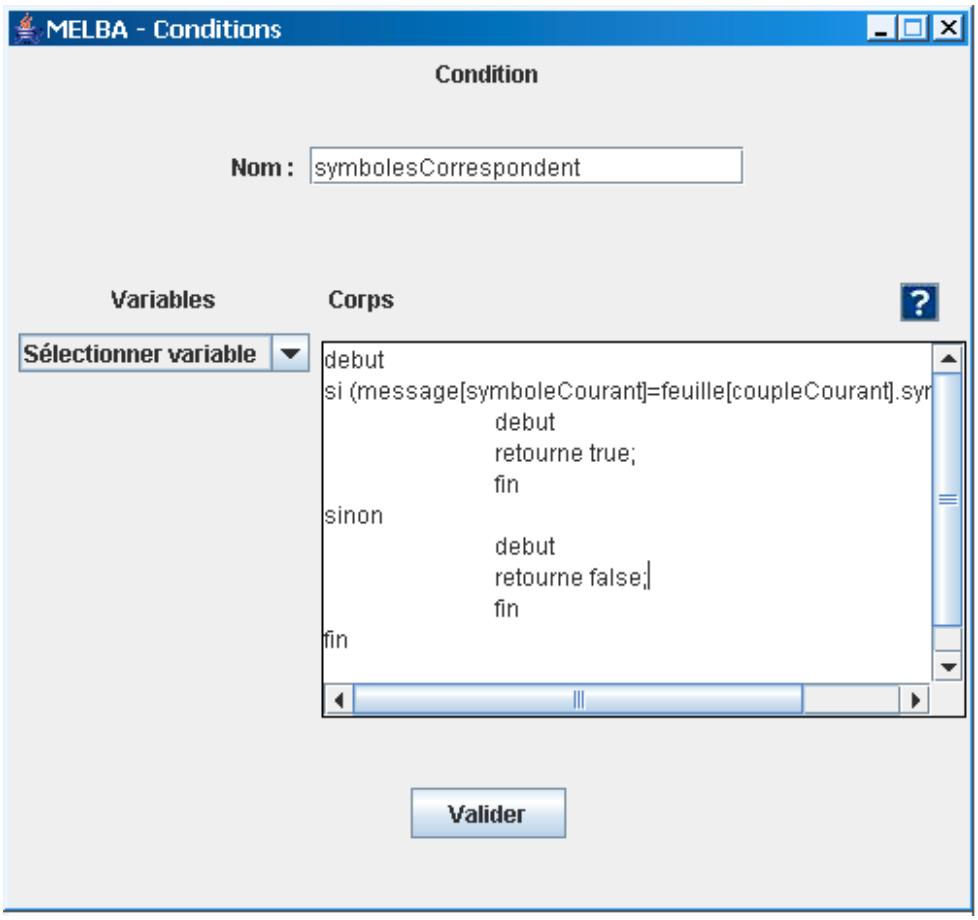


Figure 73 – Edition du corps d'une condition élémentaire avec l'environnement auteur.

5 Conclusion

Dans ce chapitre, nous avons présenté l'environnement de support à l'apprentissage de la programmation MELBA (Metaphor-based Environment to Learn the Basics of Algorithmics). Il a pour but explicite l'apprentissage de la programmation, à la différence des environnements de programmation (IDE) dont l'objet est la conception de programmes. Pour nous guider dans nos choix de conception, nous nous sommes appuyés sur le cahier des charges dégagé par le chapitre précédent. Nous avons donc conçu MELBA avec trois priorités :

- **supporter un apprentissage expérimental**. Pour cela, nous avons cherché à rendre l'*évaluation* des actions de l'apprenant la plus *progressive* (chapitre 3, section 1.10) possible, en concevant un système fondamentalement basé sur l'exemple. Nous avons également cherché à supporter toutes les phases du cycle d'apprentissage par l'expérimentation (section 2.1.2, chapitre 1) et tous les styles d'apprentissage, en proposant des outils d'édition et d'analyse du programme qui peuvent être utilisés de façon complémentaire en s'adaptant à une large gamme de tâches cognitives : programmation « avec » et « sur » exemple, animation de programmes, multiples modes d'interaction, représentations du contexte pragmatique et symbolique, trace d'exécution…

- **adapter l'environnement à l'activité pédagogique** (et non l'inverse). Dans ce but, nous avons rendu l'ensemble des composants paramétrables, afin de permettre au concepteur des exercices de définir quel outil était pertinent au vu de l'activité pédagogique. Nous n'avons pas cherché à faire un environnement de programmation « pleine échelle », basé sur un langage commercial, mais conçu un moteur d'exercices de programmation, basé sur un langage algorithmique conçu dans une optique d'utilisabilité et d'utilité plutôt que de performance. Nous avons opté pour une interaction « graphique » et très contrainte, permettant d'éviter les erreurs syntaxiques (dans le but de concentrer l'attention de l'apprenant sur les structures sémantiques) et de guider le novice, au lieu de nous baser sur un éditeur de texte « libre » plus efficace entre les mains d'un expert.

- **supporter une démarche d'apprentissage incrémentale.** Pour permettre l'introduction progressive des différents concepts, nous avons rendu les différents outils syntaxiques (structures de contrôle, types, déclarations, affectations, appels) activables ou désactivables en fonction de l'activité d'apprentissage. Nous avons conçu les exercices de l'outil selon trois niveaux, ou étapes, (approches pragmatique, hybride, symbolique de la représentation du contexte).

Cet environnement comporte ainsi plusieurs aspects réellement originaux ou innovants :

- Sa principale caractéristique provient de l'éditeur de programmes « avec » exemple, qui est le premier réellement fonctionnel. A notre connaissance, MELBA est le seul environnement « finalisé » à proposer cette fonctionnalité (ce type d'interaction était déjà possible avec les interpréteurs de certains langages, mais seulement avec un terminal, le programme interprété n'était pas sauvegardé, il était impossible de revenir en arrière en cas d'erreur, etc.).

- MELBA est fait pour « apprendre à programmer » et non pour « programmer sans apprentissage ». Il est de ce fait atypique dans la « programmation basée sur exemple ». Il est également le seul système à combiner l'approche sur exemple, basée sur des

conventions de dialogue en manipulation directe, et la manipulation du programme généré « avec » l'exemple.

- De même, il est original pour un environnement basé sur le paradigme impératif structuré d'être « dirigé par les modèles », et de se positionner explicitement au niveau sémantique ; la totalité des outils décrits dans le chapitre trois se positionnent au niveau langage.

- Enfin, il est le seul à proposer d'aborder de façon incrémentale, sur autant de niveaux sémantiques, l'apprentissage de la programmation. Le concept de « mini-langage » ou « sous-langage » est ancien, mais ses implémentations se limitent généralement à un seul niveau sémantique (voire deux).

Chapitre 5

Utilité et usages de l'environnement MELBA : trois expérimentations

Résumé. *Alors qu'en tant qu'outil de conception, la programmation « basée sur l'exemple » n'a jusqu'ici été validée que par des études de faisabilité, nous proposons d'évaluer de façon rigoureuse sa pertinence dans le cadre de l'apprentissage de la programmation, à travers une série d'expérimentations en milieu réel et en milieu contrôlé.*

Pour cela, dans un premier temps nous passons en revue et classifions les différentes techniques et métriques d'évaluation des EIAH, en les illustrant à l'aide d'exemples de la littérature, ayant pour contexte la programmation. Nous discutons de leur pertinence dans le cadre de l'évaluation d'un EIAH de la programmation tel que MELBA.

Puis, nous détaillons deux évaluations empiriques de l'efficacité du logiciel en milieu réel. Ces expériences s'appliquent à un cours d'initiation à la programmation et comparent les performances d'un groupe avec l'environnement MELBA à celles de groupes témoins ayant suivi un cursus de TD classique (papier + crayon). Les résultats de ces différentes expérimentations sont interprétés en relation avec des observations qualitatives et des résultats d'études existantes en Psychologie de la programmation.

Enfin, nous complétons ces évaluations de l'efficacité par une étude des usages et de l'appropriation du logiciel par les étudiants, réalisée en milieu contrôlé à l'aide de logs d'interaction et d'indicateurs oculométriques.

Ces deux études nous permettent d'ébaucher une réponse argumentée aux questions suivantes, au cœur de notre problématique de recherche :

La programmation basée sur exemple est-elle efficace pédagogiquement, et si oui quels schémas, savoirs, compétences recouvrent son domaine d'application ?

Est-elle accessible, voire naturelle pour tous ? Ou convient-elle mieux à certains types d'apprenants, en fonction de leur niveau de maîtrise, de leur profil cognitif ?

1 Modes d'évaluations des EIAH

Concernant un environnement destiné à l'apprentissage, la mesure de l'utilité couvre deux aspects distincts et cependant connectés. D'une part, elle requiert d'évaluer l'apprentissage (de la discipline enseignée et non pas de la manipulation du système), et d'autre part elle recouvre la capacité à réaliser des tâches – ici, des exercices de programmation – avec l'EIAH (Nogry 2004). Pour ceci, nous nous appuierons sur des méthodologies de tests issues de différentes disciplines dont l'objet d'étude est l'apprentissage.

On peut distinguer plusieurs approches majeures, qui nous permettront ci-après de catégoriser les techniques les plus communément rencontrées, et de déterminer quel est le type de technique le plus pertinent au regard de notre objectif. Certains critères de distinction sont assez communément reconnus, et ont été exposés notamment par (Nogry 2004), et (Hû 1998). Ils pratiquent une première distinction entre les méthodes dites « quantitatives », et, à l'inverse, « qualitatives ».

1.1 Les méthodes quantitatives

Les méthodes quantitatives visent à mesurer de manière objective l'impact d'un dispositif sur l'apprentissage. La méthode comparative, développée par la psychologie cognitive, est souvent utilisée pour évaluer les EIAH. Elle consiste généralement à séparer les apprenants en deux (et parfois plus) groupes, incluant un groupe témoin. Les participants font « exactement la même chose », sauf ce qui est évalué.

Pour que cette évaluation soit valide, on prend la précaution de présenter dans la condition contrôle et dans la (les) condition(s) testée(s) les mêmes contenus, la même démarche, pendant le même temps, avec le même environnement, la même consigne. De plus amples informations sur cette méthode sont présentées par (Shute 1993).

Une première difficulté de cette méthode réside dans le choix de la condition contrôle. Des options communément rencontrées consistent à comparer l'EIAH à un enseignement non informatisé, à un autre système, ou à une version tronquée de l'environnement testé, selon ce que l'on souhaite mesurer. La seconde difficulté majeure se situe dans le choix de la métrique, qui doit être à même d'évaluer en priorité l'objectif de plus haut niveau qu'est l'apprentissage. Malgré ces difficultés, cette méthode permet d'observer le résultat d'un changement dû au système et d'inférer les connaissances acquises par l'apprenant avec un certain degré de généralité.

Presque indépendamment de la métrique employée, il est possible de classifier les analyses comparatives en deux groupes. Une première approche consiste à tester l'apprentissage *a posteriori* (post-test), éventuellement en comparaison avec un test initial (pré-test). Cette évaluation peut être le résultat de la phase d'apprentissage (par exemple, des compte-rendus de Travaux Dirigés ou Pratiques rédigés par les apprenants), avoir lieu juste après, ou en être plus décalée dans le temps… On parle alors d'approche hors-ligne (*offline*).

L'approche inverse consiste à faire des mesures pendant la tâche objet de l'apprentissage, et d'analyser la trace, ou les erreurs, par exemple, de l'apprenant… Cette approche permet

d'analyser ce que fait l'apprenant au cours de l'activité proposée (l'utilisation du logiciel). On parle alors, *a contrario*, d'approche « en-ligne » (*online*).

1.1.1 Approche hors-ligne *en programmation : exemples d'évaluations et de métriques*

L'analyse de résultats d'examens ou de tests ayant valeur de standard est la voie la plus classiquement utilisée dans les méthodes hors ligne. Selon ce que l'on souhaite vérifier, la métrique peut différer : moyenne des résultats, taux de réussite… De même, la granularité peut varier selon les études ; parfois les analyses portent sur des résultats globaux, parfois, différentes parties du test (correspondant à des thématiques différenciées) sont étudiées séparément.

De telles évaluations hors-ligne sont couramment utilisées en Psychologie de la Programmation, où l'on cherche à déterminer l'existence (ou non) de corrélations entre des caractéristiques cognitives données et la compréhension de concepts ou plus généralement le succès à l'examen. Par exemple, Mancy (Mancy 2004) utilise cette approche pour mesurer la corrélation[1] entre la taille de la mémoire de travail, l'indépendance au champ, et les résultats moyens aux tests et examens d'informatiques (CS1) à l'université de Glasgow ; les résultats de son étude sont présentés ci-après (Tableau 7 et Tableau 8).

	TP 1	**Test**	**TP 2**	**Examen**
Mémoire de travail	0,15	0,13	**0,26**	0,16
Dépendance au champ	**0,27**	**0,24**	**0,36**	**0,40**
Nombre d'étudiants	158	159	145	154

Tableau 7 – Corrélations entre mémoire de travail et résultats aux tests, et entre dépendance au champ et résultats aux tests, selon (Mancy 2004). L'auteur a mis en gras les coefficients significatifs.

	Dépendant au champ	**Intermédiaire**	**Indépendant**
Score au test de dépendance au champ	0-11	12-16	17-20
Nombre d'étudiants	55	54	45
Moyenne à l'examen	44,0/100	53,5/100	63,5/100

Tableau 8 – Score au test de dépendance au champ et score à l'examen, selon (Mancy 2004).

Le principal avantage de ces méthodes hors-ligne, couplées à des métriques telles que le taux de réussite ou la moyenne à l'examen, est de pouvoir évaluer le transfert de connaissance et la

[1] Le coefficient de corrélation de Pearson est gradué de –1 à 1. Un résultat de –1 implique que lorsque l'une deux variables observées augmente, l'autre baisse proportionnellement (la courbe de leur évolution est une droite de pente négative), un résultat de 1 implique le contraire (droite de pente positive). Donc plus on est proche de 1, plus le résultat est positivement significatif.

compréhension. Cependant, elles ne peuvent fournir aucune indication sur le cheminement de l'apprenant, ni sur la façon dont il pourrait utiliser l'environnement. Pour cela, on a recours à des méthodes *en ligne*.

1.1.2 Approche en ligne *en programmation : exemples d'évaluations et de métriques*

Plusieurs mesures ont été proposées par la littérature, dans le cadre d'études comparatives : ainsi Pane (Pane 2002) a-t-il adopté deux mesures pour l'utilité de son environnement, HANDS. D'abord, la comparaison, sur une série de tâches, des performances de groupes d'utilisateurs novices en programmation (des lycéens, en l'occurrence) avec une version complète et une version limitée de son environnement. La métrique a été une mesure d'efficacité classique : « combien d'utilisateurs ont accompli (avec succès) les tâches ».

	Version Complète	Version limitée
Nombre de participants	12	11
Ont terminé le tutorial	9	9
Ont résolu au moins une tâche	7	1
Nombre de tâches (cumulées) résolues	19	1

Tableau 9 – Comparaison de performances : l'environnement HANDS comparé à une version limitée (sur deux groupes de lycéens).

Ensuite, il se livre à une comparaison entre son système et un environnement de l'état de l'art (StageCast Creator) pour l'accomplissement d'une tâche (en l'occurrence la programmation du jeu Pacman par un lycéen).

	HANDS	StageCast Creator
Temps d'apprentissage avec le tutorial (en mn)	40	60
Temps de réalisation (en h)	15	9
Types d'objets	9	6
Nombre de gestionnaires d'événements	15	64
Nombre de « lignes » de code	183	253

Tableau 10 – Comparaison de performances : l'environnement HANDS comparé à StageCast Creator (sur un seul lycéen)

Concernant ces métriques de comparaison (temps de réalisation, « taille » et « complexité » du programme obtenu, représentativité de la tâche choisie…), au-delà de leur représentativité de l'« efficacité » (qui peut éventuellement être discutée), elles ne nous semblent pas pertinentes dans notre cas, car elles se focalisent sur une seule dimension : la capacité (et facilité) à réaliser des tâches de programmation. Si la validité de cette approche est difficilement contestable dans l'approche « programmer sans apprendre », elle est en revanche, de notre point de vue, moins adaptée à notre objectif. En effet, il n'y a pas de lien démontré entre la réalisation de la tâche et l'apprentissage effectif, un échec dans la réalisation de la tâche pouvant même, dans certaines conditions, être bénéfique pour l'apprentissage.

Une autre approche « en ligne » comparative est proposée par McIver dans (Mc Iver 2001), où sont utilisées comme métriques le nombre, le type et la fréquence des erreurs syntaxiques et sémantiques qu'à commis l'apprenant au cours de la réalisation d'une tâche. Les résultats

d'un groupe utilisant le langage GRAIL conçu dans le cadre de sa thèse sont confrontés à ceux d'un groupe utilisant le langage LOGO. Le but était, dans son cas, de tester la corrélation entre erreurs syntaxiques et erreurs sémantiques. Les graphes de la Figure 74 exprime cette corrélation : chaque point indique le nombre d'étudiants à avoir commis un certain nombre (N) d'erreurs de type syntaxique ou sémantique dans chacun des deux groupes.

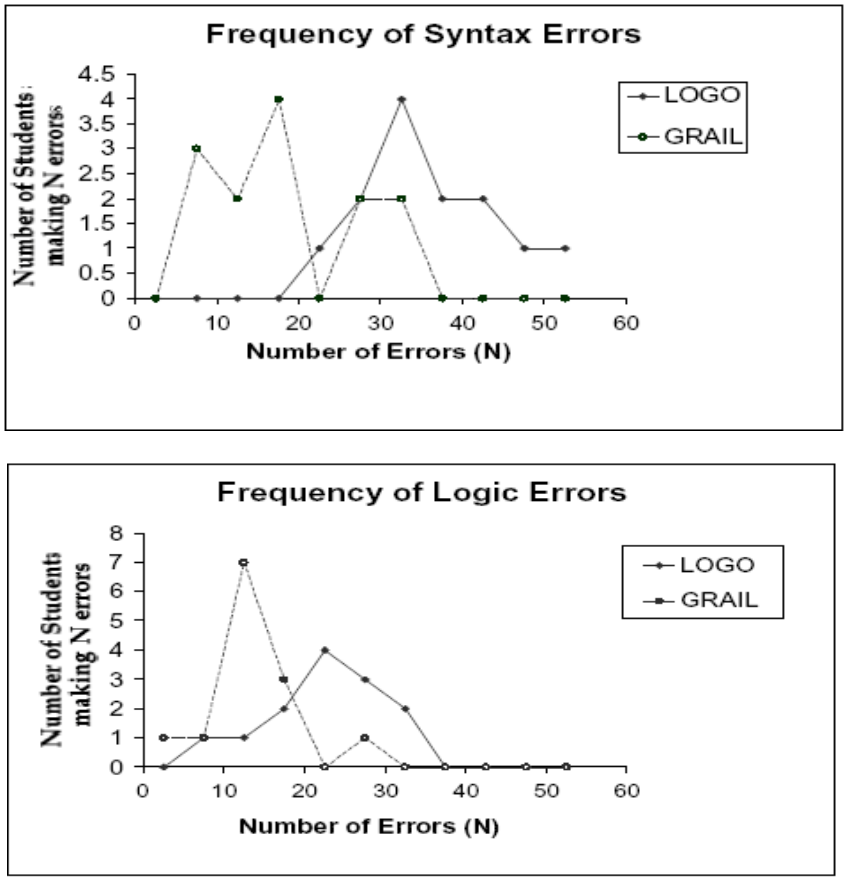


Figure 74 – Etude en-ligne d'un langage de programmation conçu dans un objectif d'utilisatibilité (GRAIL) : fréquence des erreurs syntaxiques et sémantiques, issue de (Mc Iver 2001).

Encore une fois cependant, l'adéquation de la métrique choisie avec l'évaluation de l'apprentissage peut être contestée. L'élève qui aura réalisé la tâche avec le moins d'erreurs pendant son apprentissage ne sera pas forcément celui qui aura le mieux compris, certaines erreurs étant utiles pour le processus d'apprentissage. De plus, l'approche en-ligne, si elle permet d'apporter une précision intéressante sur les processus de fonctionnement de l'apprenant, souffre d'une lacune flagrante ; il est presque impossible d'y évaluer l'inférence de savoir ou de savoir-faire utilisable hors de l'environnement. Dès lors, cette méthodologie ne parait pas optimale pour valider la compréhension de l'apprenant.

Dans la littérature, de nombreuses méthodes, issues de recherches en compréhension de texte, permettent d'accéder à une précision encore supérieure, et d'identifier sur quels éléments l'apprenant a focalisé son attention. Ces méthodes sont donc tout à fait intéressantes pour l'étude des processus attentionnels de l'apprenant au cours de la tâche (particulièrement dans les hypermédias), mais offrent encore une fois peu d'informations sur les processus d'apprentissage mis en œuvre.

1.2 Les méthodes qualitatives

En complément des méthodes précédentes qui visent à quantifier l'impact d'un environnement sur la réalisation de tâches et/ou l'apprentissage, les méthodes qualitatives offrent des outils pour évaluer les dimensions sociales et affectives en jeu dans le processus d'apprentissage. Elles sont particulièrement utilisées pour l'évaluation de l'apprentissage « en situation », et permettent également de comprendre le cheminement interne de l'apprenant ou de donner des indications sur le degré de conscience de l'apprenant face à son apprentissage. Comme pour les approches quantitatives, de telles évaluations peuvent être basées sur une méthodologie « hors ligne » ou bien « en ligne ».

1.2.1 Approches hors-ligne

Une méthodologie communément employée dans le cadre d'évaluations qualitatives des EIAH consiste à conduire des entrevues, qui peuvent être collectives ou individuelles, avec les apprenants. Ces entretiens ont lieu à la fin de la période d'apprentissage (ce qui vaut à cette approche d'être qualifiée de hors-ligne).

Il existe plusieurs types d'entretiens, selon la liberté qui est offerte aux participants de choisir ou pas les thèmes qui seront débattus. On parlera ainsi respectivement d'entretiens « ouverts » (les thèmes abordés varient en fonction du participant), « structurés » (les questions sont prédéfinies) ou « semi structurés » (les questions à poser sont prédéfinies, mais une place est laissée pour des réponses plus individualisées). Dans le cadre d'une évaluation en classe, il peut être utile de faire en complément des entretiens avec les enseignants pour confronter les différents points de vue sur le déroulement de l'évaluation.

Par exemple, Van Haaster et Hagan, dans leur évaluation de BlueJ (Van Haaster 2003), s'appuient notamment sur une étude de satisfaction, menée sur 115 étudiants. On demande à ceux-ci d'évaluer BlueJ selon certaines heuristiques d'ergonomie sur une échelle de 1 à 5. L'étude consistait également à noter l'utilité des fonctionnalités spécifiques de BlueJ, et leur aide à la compréhension des concepts de la programmation orientée objet, ou à la réussite à l'examen. Pour compléter cette étude, les auteurs proposent un calcul de corrélation entre les résultats aux différentes questions. Le Tableau 11 reproduit le résultat de cette étude. Le contenu des cellules mesure la corrélation entre l'utilité des fonctionnalités (deux à deux).

Fonction	**Mesure**	Inspection des Objets	Accès à un Objet	Passage d'Objet en paramètre	Instancier un Objet
Inspection des Objets	Corrélation	1	0,78	0,56	0,62
Accès à un Objet	Corrélation	0,78	1	0,55	0,56
Passage d'Objet en paramètre	Corrélation	0,56	0,55	1	0,67
Instancier un Objet	Corrélation	0,62	0,56	0,67	1

Tableau 11 – Corrélation de Pearson entre les évaluations de l'utilité des composant de BlueJ.

1.2.2 Approches en-ligne

Une autre technique qualitative très répandue, surtout dans les pays anglo-saxons, est le recueil des verbalisations, qui consiste à demander à l'apprenant de penser à haute voix durant la tâche proposée afin d'identifier les raisonnements qu'il met en œuvre pour réaliser la tâche demandée.

Enfin, on peut observer la situation d'apprentissage « de l'intérieur » ; on parle alors de méthode « ethnographique ». L'observateur se doit alors de prendre une position réflexive vis-à-vis de ses observations, de par la subjectivité de sa position. En complément de ces observations, il peut être pertinent de tenir un journal de bord afin d'augmenter la fiabilité des observations en intégrant le point de vue subjectif et réflexif du chercheur. Ce journal de bord peut documenter l'évaluation mise en place en décrivant le déroulement du projet ou en relevant les éventuels changements d'attitude intervenus au cours du projet. Il peut également contenir des données réflexives (perception de l'évolution des processus d'apprentissage, questions émergeant de l'observation).

L'observation peut être facilitée par le recours à des instruments d'enregistrement (enregistrements vidéos, verbalisations) et à des instruments d'observation (grilles d'observation, oculomètres) qui permettent une analyse plus fine de la situation.

Cette dernière technique est employée par (Bednarik 2005), qui s'appuient sur cet instrument pour proposer une analyse qualitative de l'usage de l'environnement JEliot et de ses fonctionnalités d'animation de code, en comparant programmeurs experts et débutants en Java. Pour cela, l'environnement est tout d'abord découpé en quatre zones d'intérêt (ou AOI – pour Area Of Interest) – voir Figure 75.

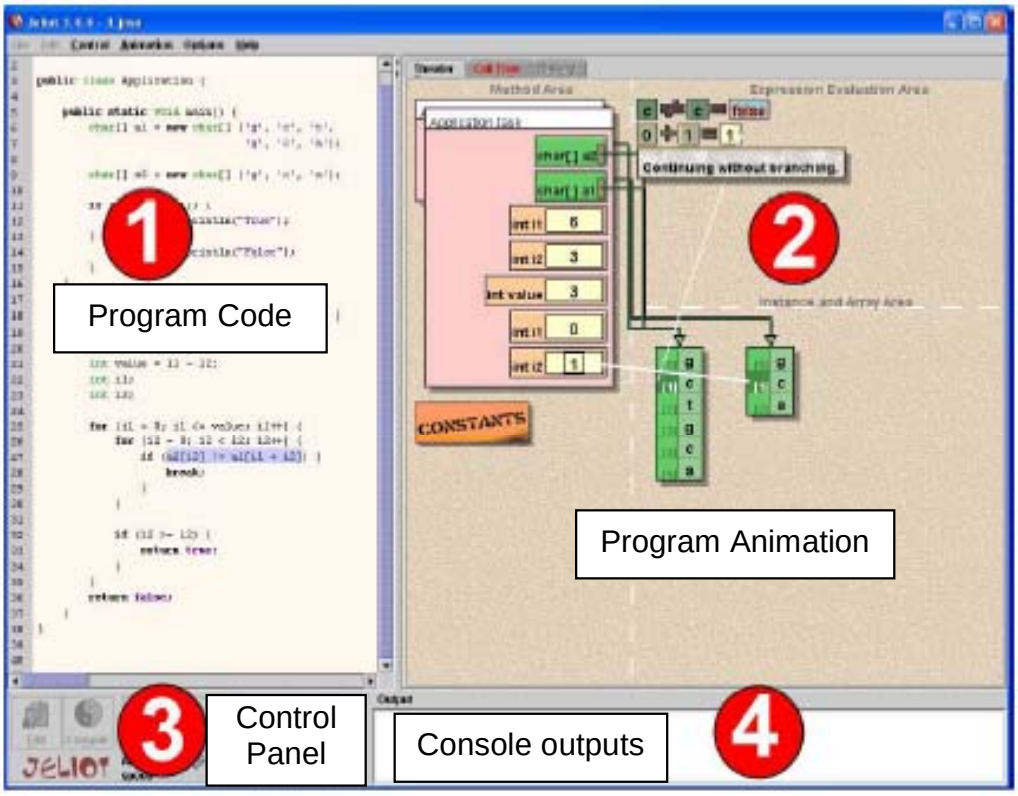


Figure 75 – L'environnement JEliot, décomposé en quatre zones d'intérêt.

Il est alors possible de mesurer grâce à l'oculomètre des données telles que le nombre de fixations oculaires dans chaque zone, ou le nombre de « transitions » d'une zone vers une autre. La Figure 76 illustre ces mesures dans l'expérience de (Bednarik 2005). Les données ne concernent en l'occurrence que les phases d'animation du programme (et pas les phases d'édition de celui-ci).

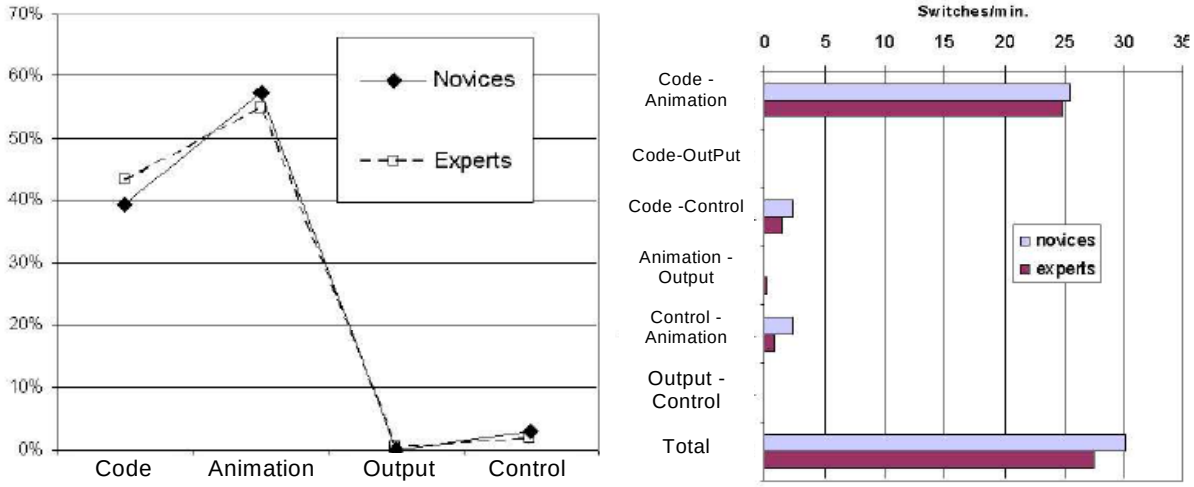
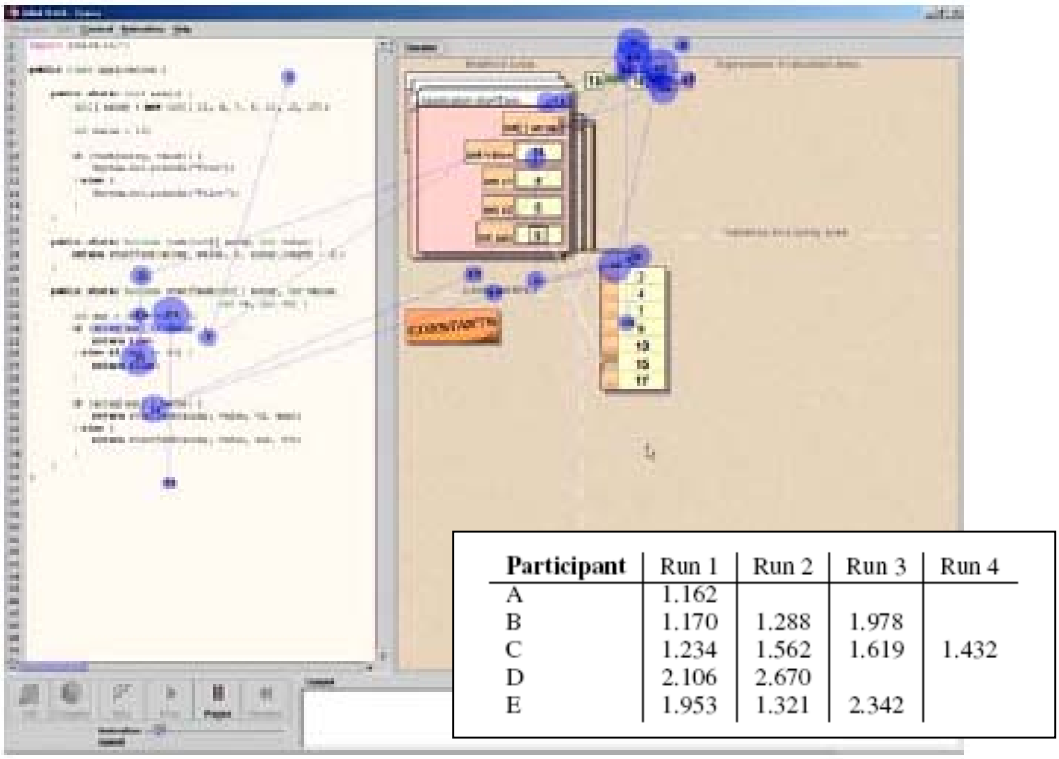


Figure 76 – Pourcentages de fixations oculaires dans chaque zone, et fréquence de transition d'une zone à une autre (nombre de transitions par minute).

Ce type de métrique globale se révèle cependant insuffisant pour définir avec précision le comportement de l'usager. C'est pourquoi les études les plus récentes cherchent à compléter ces données par la définition et la reconnaissance de « patterns » de trajectoires oculaires, à l'image de l'étude ultérieure de Bednarik et Tukiainen (Bednarik 2006), Figure 77.



Participant	Run 1	Run 2	Run 3	Run 4
A	1.162			
B	1.170	1.288	1.978	
C	1.234	1.562	1.619	1.432
D	2.106	2.670		
E	1.953	1.321	2.342	

Figure 77 – Reconnaissance de circuits de trajectoire oculaires dans JEliot.

Cette approche souffre cependant de deux limitations majeures :

- D'une part, il est généralement impossible de faire une estimation statistique (coefficient de corrélation de Pearson, Khi2, Anova …) des résultats, à cause du faible nombre de participants. Ce faible nombre de participants a pour origine trois facteurs principaux. Premièrement un facteur financier : le coût de l'oculomètre induit que peu de laboratoires en possèdent plusieurs exemplaires. Deuxièmement, le coût temporel de l'étude : de par la nature de l'oculomètre, un seul étudiant peut l'utiliser à la fois. Et comme les laboratoires en possèdent en général un seul (voire deux maximum) le temps d'expérimentation (et donc incidemment, d'occupation des locaux) est important y compris avec peu d'expérimentateurs. Troisièmement un coût d'exploitation des données : l'oculomètre enregistre les fixations oculaires ; il va donc sans dire que pour chaque sujet, un grand nombre de données de très bas niveau sémantique sera généré, rendant l'exploitation à grande échelle difficile.

- D'autre part, l'oculométrie ne permet pas non plus de suivre le processus de pensée de l'apprenant. On peut suivre le processus de résolution de tâche très finement (beaucoup plus qu'avec des traces d'interaction) mais légender ce processus se révèle en contrepartie ardu de par le faible niveau sémantique des données produites (et aussi incidemment le grand nombre de données), d'autant que les données produites n'ont pas de « qualité » intrinsèque. On ne peut, généralement, pas dire que « plus c'est mieux » (ou le contraire) : un grand nombre de fixations oculaires peut en effet être causé soit par la richesse du niveau sémantique (il y a beaucoup d'informations intéressantes) soit par la pauvreté du niveau articulatoire (il faut un (trop) grand nombre de « petites » interactions pour atteindre le but, ou l'information est trop fragmentaire ou mal organisée et nécessite un gros travail cognitif de l'utilisateur pour être exploitée).

Pour autant, malgré ces limitations, cette technique est la seule qui permette de suivre finement le processus de travail de l'usager de façon totalement non intrusive. Elle connaît à ce titre un succès croissant auprès des expérimentateurs.

2 Evaluation d'une approche basée sur exemples pour l'initiation à la programmation.

Pour évaluer la validité de l'approche « basée sur exemple » dans le cadre de l'initiation à la programmation, nous nous proposons d'évaluer l'environnement MELBA selon deux axes :

- d'une part, nous chercherons à qualifier et quantifier l'aide qu'il procure à l'apprenant dans l'apprentissage ;
- d'autre part, nous chercherons à qualifier l'accessibilité, l'utilisabilité du paradigme « avec exemple » dans MELBA.

A travers cette double approche, nous testerons sept hypothèses :

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 des 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) ?

2.1 Evaluations de l'apprentissage

Notre premier objectif sera donc d'évaluer sur l'environnement MELBA non pas la possibilité et la facilité de réalisation des tâches de programmation, mais avant tout l'acquisition de connaissances aussi bien déclaratives que procédurales sur la programmation, qui pourraient être aisément réutilisables avec un autre outil.

Pour cela, nous avons choisi de nous écarter des évaluations « en ligne » sur l'outil lui-même, au profit d'une approche hors-ligne comparative. Elle porte sur un cours d'initiation à la

programmation en L3 pour des bio-informaticiens, composé de 4h de cours et 12 h de TD. On analysera dans le cadre de cette étude un examen écrit (donc papier-crayon) situé dans le temps une dizaine de jours après la fin de l'apprentissage de l'outil. De cette façon, nous pouvons mesurer en priorité, et de façon quantitative, l'existence et la qualité du transfert de connaissances.

2.1.1 Protocole expérimental

La question essentielle est, dès lors, celle du choix de la condition de contrôle. Nous avons choisi pour le groupe témoin un support papier-crayon, pour plusieurs raisons. D'une part, ce support était utilisé précédemment dans ce contexte. D'autre part, les exercices choisis en travaux dirigés dans ce cursus étaient difficilement adaptables sur un environnement de programmation classique. Dès lors, les conditions dans le groupe témoin ne seraient en rien généralisables à un autre cursus de TD, alors que les cursus d'algorithmique sans machine sont nombreux en cadre universitaire.

Sur les 65 étudiants suivant le cours, 22 ont utilisé l'environnement MELBA. Le reste de la promotion a donc suivi un cursus de TD classique (papier-crayon), car pour des raisons logistiques (disponibilité des salles PC Windows) il était impossible d'affecter le même nombre d'étudiants à chaque groupe. D'autre part, une approche ethnographique a été appliquée aux groupes sur machine, afin de disposer d'observations qualitatives complémentaires.

Enfin, la dernière difficulté est liée à la composition des groupes qui doit prévenir d'éventuels biais dans l'expérimentation. Il était impossible, pour des réalisations logistiques, de faire subir aux étudiants un pré-test de dépendance au champ. Cependant, la composition aléatoire des groupes minimise l'effet de ce paramètre. Les études sur les facteurs influant sur les résultats aux premiers modules de programmation – telles que (Goold 2000) et (Wilson 2000)– ayant d'autre part mis en évidence une corrélation des résultats avec une expérience préalable en programmation, nous avons fait en sorte de « distribuer » uniformément les élèves ayant déjà cette expérience au sein des différents groupes. De plus, nous avons affecté un étudiant à chaque machine, pour que tous utilisent l'environnement. En effet, lorsque les étudiants sont organisés par binôme, il peut arriver qu'un des deux soit toujours opérateur de l'environnement et que l'autre prenne un rôle de supervision pendant tout le cursus.

2.1.2 Observations qualitatives

D'un point de vue qualitatif, pendant les séances de TD, les groupes avec MELBA ont montré une vitesse moindre, jusqu'à 30 % inférieure. Cette observation semble commune dans les études qualitatives sur les EIAH, même si ces vitesses ne peuvent que difficilement être comparées directement. En effet, dans un TD classique, la validation vient de l'enseignant, et il arrive souvent qu'au moment où celui-ci propose une correction, il reste des apprenants qui ne sont pas parvenus à une solution correcte. A contrario, avec l'outil, tous peuvent constater la correction de leur solution, et les cycles d'essai / erreurs induisent naturellement un temps plus long. De fait, de par le retour immédiat d'informations, les étudiants peuvent voir d'eux-mêmes si leur algorithme est ou pas correct, et les questions qu'ils posent à l'enseignant ne sont pas de même nature (il y a plus de « pourquoi ? » que de « est-ce que ? »). Concernant l'utilisation du logiciel, plusieurs observations ont semblé apparaître de façon récurrente.

Premièrement, le syndrome de la page blanche n'est pour ainsi dire jamais survenu, et le comportement en cas d'échec répété semble plus pencher vers le bricolage que vers l'abandon. Cela est dû d'après nous à l'expressivité des opérations de base, et à la facilité d'expérimentation, qui favoriseraient les activités de conception exploratoire.

Deuxièmement, une typologie des usages a semblé se dessiner. Ainsi, certains étudiants (pour la plupart, ceux qui avaient déjà une expérience de la programmation) semblent ne jamais utiliser le mode « animé », préférant soit le fonctionnement classique de la programmation (asynchrone) soit un fonctionnement synchrone (avec exemple) mais sans animation. Ceux qui semblaient avoir le niveau le plus faible semblaient au contraire beaucoup utiliser l'animation. Une interprétation de ces usages différents peut être que les élèves les plus faibles ont des problèmes avec le flot de contrôle en lui-même (d'où l'intérêt pour eux de l'animation), alors que pour les autres, la difficulté consisterait davantage à être capable de lier le flot de contrôle à l'état du système, et à pouvoir inférer à partir du programme les principales modifications de celui-ci.

Troisièmement, il est apparu que le composant « d'historique » était très peu utilisé, du moins par les élèves, et utilisé marginalement par l'enseignant pour fournir des explications. Notre opinion sur cette observation est liée à l'observation précédente. Comme l'historique est à la base une représentation « à plat » du flot de contrôle (la suite exhaustive des commandes exécutées), elle aurait peu d'intérêt pour relier programme et états, et ne serait utile que pour combattre de fausses conceptions sur le flot de contrôle (donc à l'usage des étudiants les plus faibles). Son utilisation par l'encadrant plus que par les élèves, qui lui préfèrent l'animation, serait liée à une utilisabilité inférieure. De plus, la trace pose un problème de charge cognitive, car personne ne peut gérer mentalement le dépliage exhaustif d'un programme au-delà d'un certain degré de complexité (assez faible dans l'absolu). C'est pourquoi nous envisageons de travailler préférentiellement sur le *film* (la succession d'états) plutôt que sur la trace (succession des commandes), même si les contraintes techniques sont plus grandes.

Dernièrement, l'hypothèse selon laquelle la juxtaposition des deux représentations (analogique et système) de l'état du programme pouvait procurer une aide pédagogique pour franchir le fossé cognitif entre la représentation mentale qu'a l'étudiant des objets du contexte et la représentation informatique du modèle du programme ne semble pas se vérifier. Les deux sources d'informations semblent, au contraire, être perçues comme redondantes, et les apprenants tendraient à utiliser exclusivement la représentation analogique.

2.1.3 Résultats quantitatifs

2.1.3.1 Première expérimentation

Pour analyser quantitativement l'efficacité de l'approche, les étudiants ont été soumis à un test devant évaluer leur apprentissage, à travers des exercices d'analyse et de correction de programmes, et à travers des exercices de production d'un programme « from scratch ». Les résultats du partiel (Tableau 12) montrent un écart de 0,7 points sur 10 au niveau de la moyenne, assorti d'un écart type plus faible. Cela est à mettre en corrélation avec un taux de réussite qui est supérieur de 15 points dans le groupe avec MELBA.

	Groupes PsE	Groupes témoins
Moyenne des notes	6,5 /10	5,8/10
Ecart-Type	34	39,5
% de résultats >= moyenne	73 %	58 %

Tableau 12 - Résultats comparés des groupes avec et sans l'outil.

Nous avons alors souhaité analyser de plus près les différences entre les groupes. Pour cela, nous avons étudié la répartition des notes (Tableau 13) pour les différents groupes. Plusieurs différences apparaissent de façon sensible : d'une part, une plus faible proportion de très faibles (<=2,5/10) résultats dans le groupe MELBA (-14 points), d'autre part le prorata plus important de notes en Q3 (>=5 et <7,5) dans le groupe MELBA (+15), et le plus grand pourcentage de notes au dessus de 7,5 (+5).

	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%)

Tableau 13 – Répartition des résultats dans les exercices du partiel portant sur les concepts travaillés avec l'outil.

Les répartitions des notes semblent indiquer un « glissement » de 0-25 vers 50-75, ce qui voudrait dire que l'usage de l'environnement aurait été le plus profitable pour les étudiants faibles (passages de Q1 à Q2 et de Q2 vers Q3), tout en apportant un bénéfice moins significatif aux « bons » étudiants[1]. Ce phénomène nous parut nécessiter plus amples études, dans le but d'identifier quelles sont les compétences mises en jeu dont l'acquisition avait été facilitée par l'usage de l'outil.

2.1.3.2 Seconde expérimentation

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

[1] C'est là un phénomène assez classique. Très souvent ce sont les étudiants les plus faibles qui tirent profit de l'usage d'un EIAH. Cependant ce phénomène s'accompagne souvent d'une désaffection pour l'outil de la part des meilleurs étudiants, qui n'a ici pas été constatée, au contraire.

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. 24 autres ont suivi un cursus de TD classique, parmi lesquels 18 novices complets. Pendant les TD, la tendance des groupes sur l'outil à travailler plus lentement a été confirmée, même si la différence était moindre que lors de la précédente expérimentation. Parmi les exercices du partiel, trois portaient sur un objectif spécifique traité avec l'outil en TD. Il s'agissait :

1. de décrire de façon précise (à travers sa trace) le comportement d'un programme dont la tâche est connue, mais qui présente des erreurs, et d'en donner le résultat (5pts).
2. de rédiger un programme devant accomplir correctement cette tâche (5pts).
3. de déduire la tâche qu'un programme donné doit accomplir (et la stratégie appliquée) (2pts).

En complément de cette dernière question, il était demandé de détecter les erreurs que contenait ce programme et de les corriger, cependant, tous les étudiants ne sont pas allés au bout de ces dernières questions. Comparativement, ces tâches ont donné les résultats suivants (Tableau 14) :

% de notes supérieures à la moyenne	Melba (15)	Témoin (18)
1) Description de la trace :	6 (40%)	4 (22%)
2) Rédaction de programme :	6 (40%)	6 (33%)
3) Compréhension de la tâche :	14 (93%)	14 (78%)
Total :	7 (47%)	5 (28%)

Tableau 14 – Taux de réussite 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). Ce dernier écart est significatif à 86% selon le test du khi2[1] ce qui est assez important compte tenu de la faible taille de l'échantillon. Pour ce qui concerne l'écriture de programmes, l'écart est plus faible, et peu significatif.

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. A l'inverse, leur rôle d'observateur leur

[1] Les exercices ne peuvent pas être évalués individuellement à cause de la présence de valeurs strictement inférieures à 5.

aurait 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%

Répartition des Notes (Globales) :				
Q1	Q2	Q3	Q4	>=50%
1 (7%)	7 (47%)	4 (27%)	3 (20%)	47%
1 (6%)	12 (67%)	3 (17%)	2 (11%)	28%

Table 4 : *Répartition des notes dans les deux groupes (Le groupe MELBA est dans la ligne supérieure, le groupe témoin dans la ligne du dessous).*

Même en ignorant les étudiants qui ont manifestement été gênés par sa formulation, l'exercice d'analyse apparaît plus difficile que celui de rédaction. Comparé aux très bons résultats au dernier exercice, une interprétation possible serait ce que Pea (Pea 1986) appelle un « bogue d'intentionnalité ». Etant entraînés depuis l'enfance à pratiquer une lecture de texte

« globale » dont l'objectif est d'extraire le sens du texte, les étudiants ou même des programmeurs plus expérimentés seraient handicapés en recherche de bogue par ce mode de lecture préférentiel. La difficulté serait que le code « lu » par les programmeurs ne serait pas celui écrit dans le programme ou l'énoncé, mais une version « bruitée » par leur connaissance de ce qu'il « devrait faire » (un cas de dissonance cognitive). Il semble que les étudiants du groupe machine y soient globalement moins sujets.

Un autre élément qualitatif qui pourrait plaider pour cette hypothèse, serait le « transfert » du résultat 3 à l'exercice d'analyse vers le résultat 5 en composition : en effet les étudiants concernés apportent dans leur programme, une correction à des erreurs qu'ils n'évoquent pas à l'exercice précédent.

Au niveau de cet exercice, la différence de +18 points d'étudiants ayant la moyenne en faveur du groupe machine, tendrait à montrer un effet positif de la programmation avec exemples et de l'animation du code pour acquérir les connaissances procédurales nécessaires à l'usage des structures de contrôles et à la construction d'un modèle mental de l'état du système. Ces résultats confirment ceux du premier test dans le sens où on retrouve le pic à la moyenne dans le premier exercice, et une différence de +9 à 5 points en composition. Ces résultats sous-entendent également que le composant d'historique et la fonctionnalité d'animation auraient un impact limité, car les écarts de performances avec et sans ne sont pas significatifs.

2.1.4 Interprétation et analyses croisées

L'interprétation de ces effets se révèle toutefois difficile, et nous sommes essentiellement réduits à faire des conjectures. Une première hypothèse, qui irait dans le sens des résultats de Goold (Goold 2000) ou Wilson (Wilson 2000), serait que la vertu essentielle de l'environnement MELBA tiendrait dans le feedback sémantique et analogique, dans l'expressivité des actions de la pragmatique. Les facteurs principaux amenant à l'échec selon ces études seraient le degré de « confort », et la « frustration » de la programmation ressentie par les étudiants. Ce sentiment est d'après Ben-Ari (Ben-Ari 1998) lié à la pauvreté et à la brutalité des feedbacks des environnements de programmation. Ceux-ci ne permettraient pas de fournir aux novices une représentation accessible de l'état du système, mais seraient suffisants pour invalider la correction du modèle mental que l'étudiant se construit. *A contrario*, dans MELBA, l'état du système est rendu immédiatement accessible par une représentation graphique analogique, et est mis à jour automatiquement (et non à la demande du programmeur) par le mécanisme de programmation « avec exemple ». Il serait donc plus facile au novice de construire les liens qui existent entre le programme et l'état du système, et d'interpréter la cause de ses erreurs. Une autre observation qui va dans le sens de cette théorie est la quasi-disparition du syndrome de la « page blanche ».

Une autre hypothèse serait en relation avec les styles d'apprentissage des étudiants. On pourrait imaginer que l'environnement étant doté d'un feedback graphique immédiat conçu pour favoriser la construction d'un modèle mental de l'état du programme (et des liens avec les structures de contrôle de celui-ci), serait particulièrement bénéfique pour des étudiants ayant un style d'apprentissage « divergent » ou « assimilateur ». Ceux-ci, s'appuyant prioritairement sur l'observation pour apprendre, tireraient grand avantage du paradigme « avec exemple » alors qu'ils seraient bridés par la brutalité et le caractère peu accessible des retours du système avec un environnement classique (et *a fortiori* dans un TD papier). Des études telles que celle de Byrne et Lyon (Byrne 2001) défendent l'importance du style d'apprentissage parmi les facteurs pouvant causer l'échec dans l'apprentissage de la

programmation. Néanmoins d'autres études ne trouvent pas de corrélation, telle que (Goold 2000). La difficulté de généraliser les résultats de telles études est cependant importante, car elles sont généralement menées par des psychologues qui ne sont pas directement impliqués dans le cours. L'absence d'informations sur la nature du cursus ou les médias employés explique sans doute en bonne partie de telles contradictions, on peut par exemple imaginer que des étudiants ayant un style d'apprentissage basé sur l'observation seraient plus réceptifs à des exercices de recherche / correction d'erreurs sur des programmes déjà écrits qu'à des exercices de conception « from scratch ».

La caractéristique cognitive de dépendance au champ peut également être un facteur qui expliquerait les effets de l'environnement. En effet, les retours du système pourraient permettre à des apprenants « dépendants » ou « intermédiaires » de construire un lien entre programme et état qu'ils auraient sinon plus de mal à inférer à partir du seul programme, et que les pics observés (notamment le médian) seraient dû à de meilleurs résultats de ces types d'apprenants. Enfin, une hypothèse simple serait que, grâce au guidage fourni par l'environnement (impossibilité de faire des erreurs purement syntaxiques), les apprenants pourraient se concentrer sur les difficultés d'ordre sémantique et pragmatique, ce qui leur permettrait de mieux contextualiser leurs savoir et savoir-faire, et donc de mieux comprendre.

2.2 Evaluation de l'usage de l'environnement

Ces multiples hypothèses nous amènent naturellement à planifier de nouvelles expériences pour mieux cerner les processus de l'apprenant en interaction avec l'environnement. Une première piste serait de procéder à une évaluation par oculométrie et analyse des traces, dans le but de confirmer les observations qualitatives. En effet, cette approche paraît la plus à même de quantifier les usages et les difficultés d'utilisation éprouvées par les apprenants. Les analyses proposées permettront de répondre aux questions suivantes :

- Est-ce que certains composants du logiciel sont plus utilisés que d'autres ?[1]
- Est-ce que la zone historique est utilisée par les étudiants ?
- Quel est le mode d'utilisation le plus utilisé ? [2]
- Les étudiants changent-il de mode d'utilisation en cours d'exercice ? à quelle fréquence ?

[1] Par exemple : Est-ce que, lorsque les représentations analogique et informatique du contexte de la tâche sont disponibles simultanément, une des deux est utilisée de façon quasi-exclusive au détriment de l'autre ? Ou est-ce que l'étudiant utilise les deux de façon conjointe ?

[2] Entre « synchrone avec animation », « synchrone sans animation » - programmation sur exemple ou exploration active du programme - et « asynchrone ».

Pour cela, une expérimentation a été conduite sur un oculomètre Tobii 1750 (voir Figure 78) par l'équipe Multicom du laboratoire CLIPS-IMAG, à Grenoble, auprès de 6 sujets, 3 étudiants scientifiques de première année et 3 lycéens de première S.

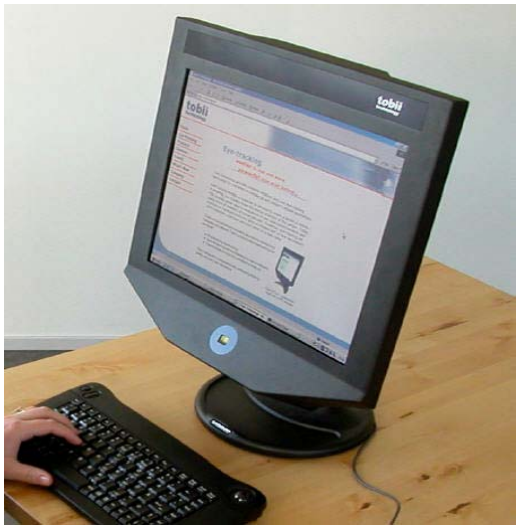


Figure 78 – Oculomètre non intrusif Tobii 1750

Les premiers avaient une expérience de la programmation, les seconds n'en avaient pas. Deux tâches spécifiques (compréhension et correction, rédaction complète) y ont été étudiées. Les apprenants ont tout d'abord suivi un tutoriel, ayant pour but de les familiariser avec les différents composants. La construction d'un programme avec MELBA y est spécifiée pas à pas. Après quoi, le protocole de cette expérimentation comporte deux étapes de 20 minutes chacune, séparées entre elles par 10 minutes de pause :

- une tâche d'analyse de programme et de correction d'erreur ;
- une tâche de conception de programme, « from scratch ».

Cette expérience a délivré des traces de trois types : premièrement, des indicateurs oculaires, deuxièment, un log des clics souris, pour les changements de mode d'utilisation, et enfin un questionnaire permettant d'exprimer la satisfaction des utilisateurs. Ce questionnaire fournit des données sur l'utilisabilité de MELBA (questions fermées) ainsi que sur l'appréciation personnelle des utilisateurs (questions ouvertes).

L'adéquation entre les traces et les hypothèses est résumée ci-après dans le Tableau 15.

Hypothèses / questions	Indicateurs oculaires	Clics souris
Est-ce que certains composants du logiciel sont plus utilisés que d'autres ?	Nb, durée et pourcentage sur chaque zone d'intérêt par étudiant, par exercice. Matrice de transition entre chaque zone par étudiants, par exercice.	
Quel est le mode d'utilisation le plus utilisé ?		Log des clics souris sur les boutons « lecture/enregistrer » et « stop ».
Les étudiants changent-ils de mode d'utilisation en cours d'exercice ? A quelle fréquence ?		IDEM Ecart (temps) entre 2 clics sur chaque bouton.
Est-ce que la zone historique est utilisée par les étudiants ?	Nb, durée et pourcentage de fixation sur la zone d'intérêt « historique » par étudiants, par exercice [+ questionnaire de satisfaction].	

Tableau 15 – Hypothèses et traces récupérées pendant les sessions à l'oculomètre.

2.2.1 Indicateurs Oculaires

Afin d'étudier les différents composants du logiciel Melba, nous avons défini sur l'interface du logiciel six zones d'intérêt ou AOI (Areas Of Interest) – Figure 79. On y retrouve les cinq zones décrites au chapitre 4 (Opération pour la barre d'outils, Commande pour les modes du logiciel, Programme, Exécution pour l'exemple, Historique). L'interface a été retouchée pour laisser un espace fixe pour les boites de dialogue affichant les message d'erreurs (Popup_erreur).

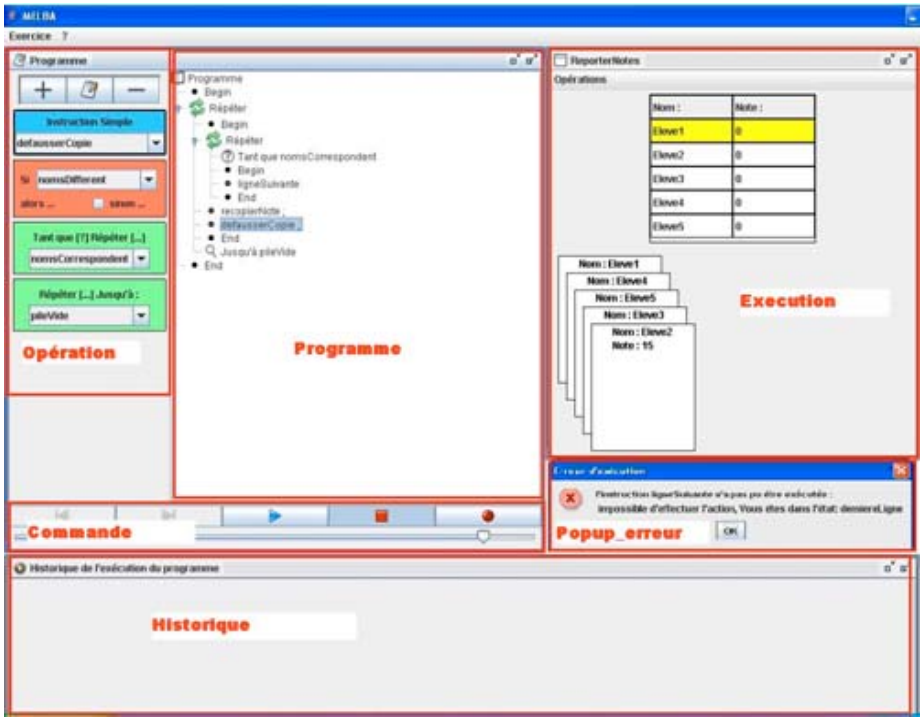


Figure 79 – Découpage de l'interface de MELBA en zones d'intérêt.

Les indicateurs de l'oculomètre (Tableau 16) nous permettent ainsi de répondre rapidement aux deux premières questions (« Est-ce que certains composants du logiciel sont plus utilisés que d'autres ? » ; « Est-ce que la zone historique est utilisée par les étudiants ? »). On constate ainsi 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.

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%
Commande	3,3%	1,9%
Popups_erreur	5,7%	6,8%

Tableau 16 – Répartition des fixations oculaires entre les AOI selon l'exercice.

Par ailleurs, une 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. Plusieurs hypothèses peuvent être avancées pour expliquer ce changement. Nous tendrions à mettre en avant deux facteurs orthogonaux :

1. La première hypothèse que l'on peut avancer serait un effet d'apprentissage. Les sujets sauraient mieux ce qui leur est utile ou non pour construire le programme. Cela expliquerait le désintérêt pour l'historique, notamment. Cette hypothèse est supportée par les « hotspots » représentant la distribution des durées de fixation ; on constate ainsi que lors de l'exercice 1 (Figure 80 et Figure 82), les fixations restent très diffuses. Les sujets cherchent des informations dans tous les composants. A contrario, lorsqu'on regarde les durées des distributions de l'exercice 2 (Figure 81 et Figure 83), on remarque que la concentration des fixations est moins diffuse.

2. Deuxième hypothèse, les deux tâches sont de nature différente, et cette différence peut être considérée comme la principale raison de ces changements. En effet, la première tâche consiste à corriger un programme existant, donc l'observation et l'exploration du programme sont prépondérantes sur l'ajout ou la suppression d'instructions. Cela explique que la zone d'exécution soit bien plus regardée que la barre d'outils. *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. Par contre le nombre d'ajouts et de suppressions dans le programme est bien plus important (10 instructions en moyenne contre 2 dans la première tâche), ce qui explique la progression de l'utilisation de la barre d'outils.

En outre, un grand nombre de fixations internes dans une zone n'est pas nécessairement une bonne chose : cela peut témoigner d'un trop grand nombre d'interactions articulatoire de bas niveau, ou d'une mauvaise organisation de l'information. C'est d'ailleurs à la suite de cette étude que le glisser / déposer d'instructions dans le programme a été implémenté, dans le but de diminuer la distance articulatoire à la modification de celui-ci.

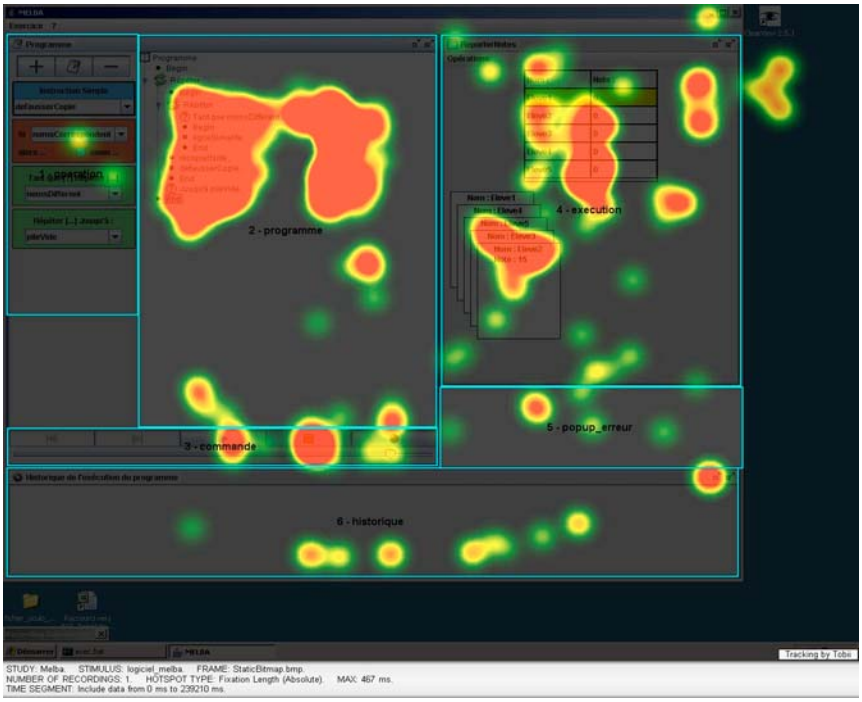


Figure 80 – Concentration des durées de fixation → Sujet 5 Exercice 1

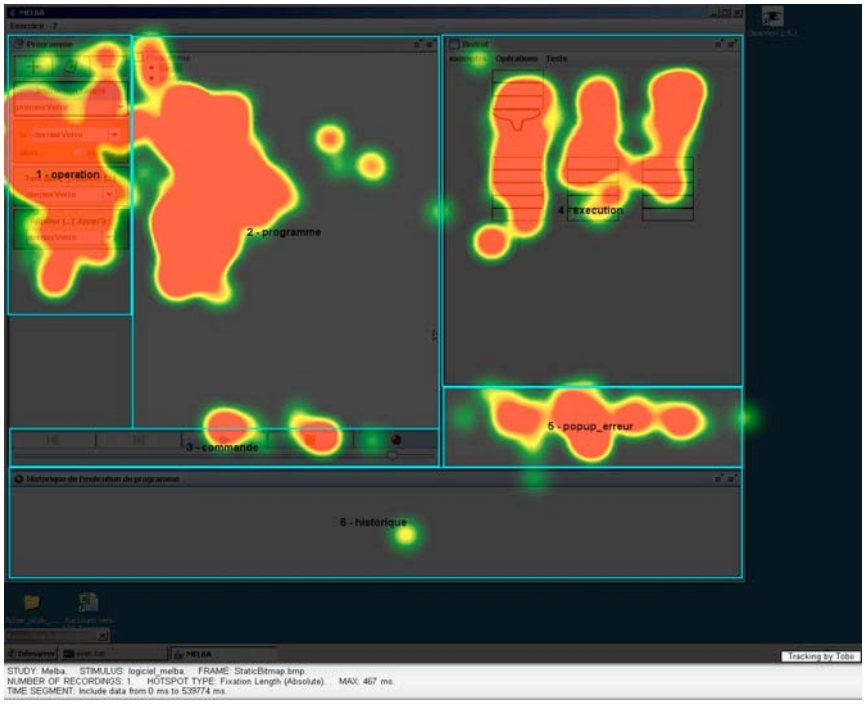


Figure 81 – Concentration des durées de fixation → Sujet 5 exercice 2

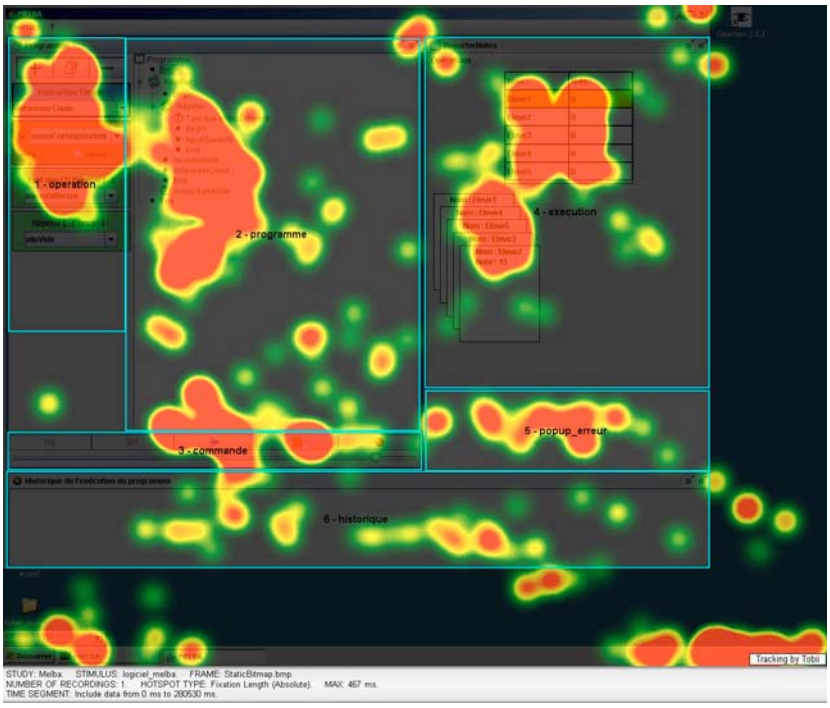


Figure 82 : Concentration des durées de fixation → Sujet 6 Exercice 1.

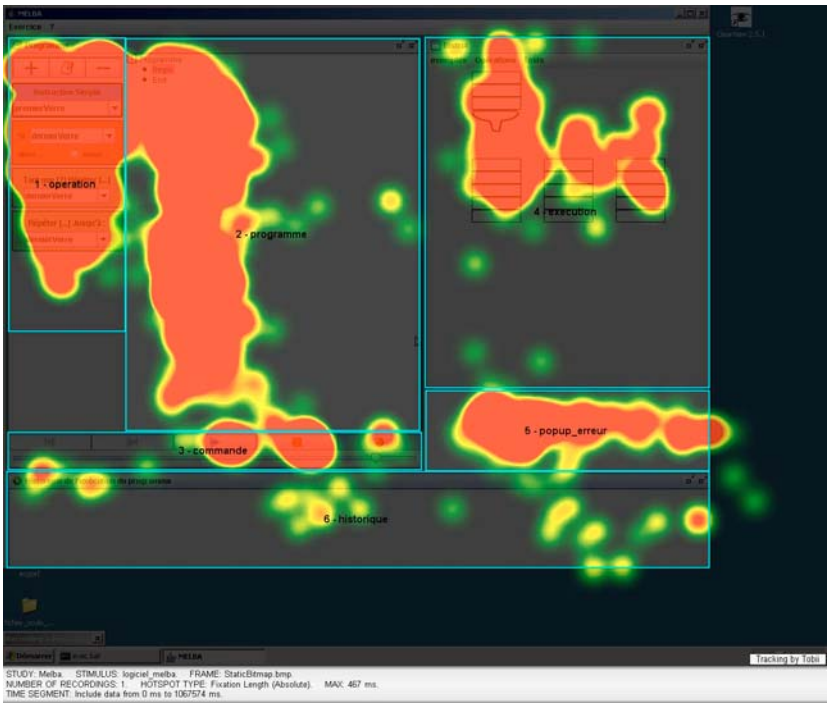


Figure 83 : Concentration des durées de fixation → Sujet 6 Exercice 2.

Les Figure 84 et Figure 85 répertorient les transitions entre chaque AOI (en pourcentages). On constate que les transitions les plus communes dans la première tâche relient les AOI Programme et Exécution (33%), avant les transitions Programme et Opération (26,5%). Cela s'inverse lors de la seconde tâche (25% et 41,5%). Les transitions les plus rares (<2%) sont ignorées (ce qui explique l'absence des zones de Commande et d'Historique, trop peu visualisées).

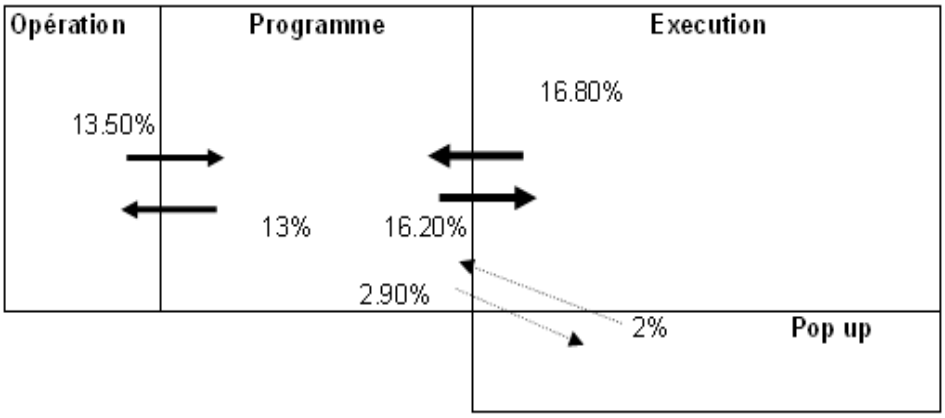


Figure 84 – Transitions entre les différentes zones de l'interface, dans l'exercice de détection et correction d'erreur.

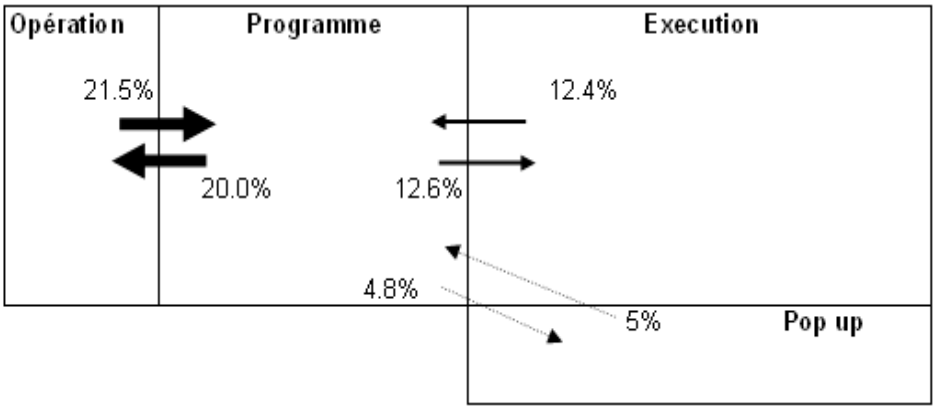


Figure 85 – Transitions entre les différentes zones de l'interface, dans l'exercice de rédaction de programme.

Ces modifications des transitions entre zones sont cohérentes avec la deuxième hypothèse : la faible diminution des transitions Programme - Exécution (8,5%) atteste que c'est l'exploration interne de la zone d'exemple qui diminue dans la seconde tâche (l'utilisateur sait mieux ce qu'il cherche), plus que les consultations de celle-ci. Par ailleurs l'augmentation des transitions avec la zone « Opération » (la barre d'outils) est très importante, attestant du plus grand nombre de modifications de l'arbre du programme.

Enfin, ces indicateurs laissent à penser que la recherche d'un écho du système à chaque modification n'est pas systématique. Il existerait donc différents types d'interactions, influant sur la façon d'utiliser l'écho de l'exemple. Pour pouvoir confirmer ou infirmer cette hypothèse, nous analysons l'utilisation des différents modes d'interaction du système.

2.2.2 Traces des clics souris

Intéressons-nous en premier lieu à la répartition de l'usage des modes (synchrone sans animation, – RECORD – synchrone avec animation – PLAY – et asynchrone – STOP – cf. chapitre 4, section 3.4.2) sur l'ensemble des deux tâches, en mesurant le prorata de temps passé dans chaque mode (Tableau 17).

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%

Tableau 17 – Usage de chacun des modes de MELBA selon la tâche.

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)…

Ces informations demeurent cependant encore trop générales pour pouvoir tirer des conclusions sur les usages de l'environnement. Les Tableau 18 et Tableau 19 répertorient l'utilisation de chaque mode par chaque sujet, et le temps qu'a mis celui-ci pour compléter la tâche.

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

Tableau 18 – Utilisation des modes de MELBA pour chaque sujet, en compréhension et correction de programmes

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

Tableau 19 – Utilisation des modes de MELBA pour chaque sujet, en conception de programme.

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). 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).

Les Figure 86, Figure 87 et Figure 88 illustrent l'utilisation des modes au cours du temps, selon le profil.

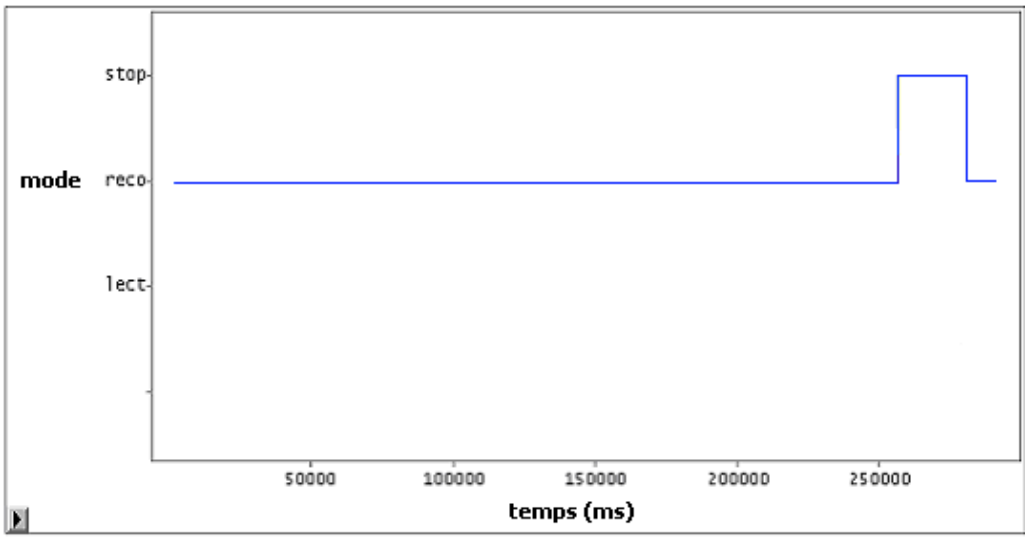


Figure 86 – Utilisation des modes dans l'exercice 2 : Sujet 3 (profil programmation avec exemple)

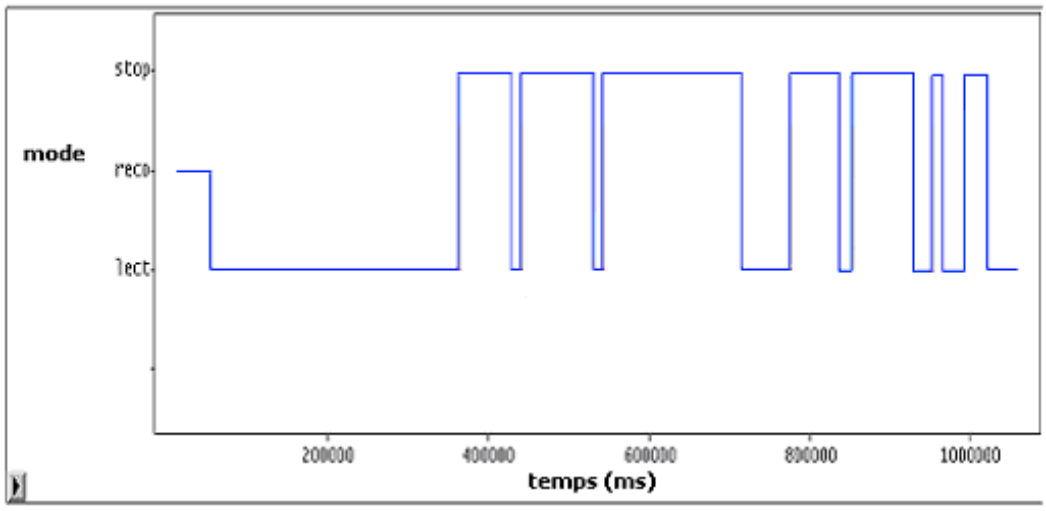


Figure 87 – Utilisation des modes dans l'exercice 2 : Sujet 6 (profil classique)

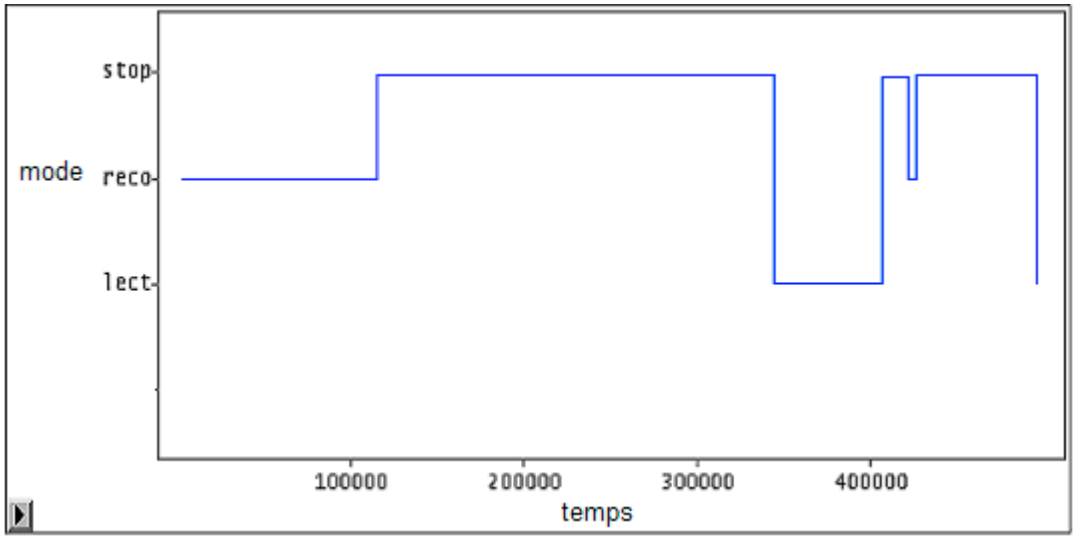


Figure 88 – Utilisation des modes dans l'exercice 2 : Sujet 1 (profil hybride).

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.

2.2.3 Observations qualitatives

Dans le but d'identifier des motifs communs dans l'utilisation du logiciel, nous avons visionné les vidéos de rejeu des fixations oculaires. Cela nous a permis d'identifier quatre types d'utilisation de l'exemple :

1. (STOP + PLAY) L'utilisateur est passif dans son utilisation de l'exemple. Il n'a pas ou peu de modèle préalable sur le comportement du programme. L'animation du programme joue un rôle d'explication. Le temps passé en animation de programme est équivalent au temps d'édition.

2. (STOP + PLAY) L'utilisateur est actif dans son utilisation de l'exemple. Il possède un modèle du comportement du programme, et utilise l'animation afin de superviser périodiquement l'adéquation de son modèle mental avec le comportement réel du programme.

3. (RECORD) l'utilisateur est actif ; il a en tête un modèle de comportement du programme, mais celui-ci est fragile, aussi supervise-t-il fréquemment le comportement du programme sur la zone d'exemple : les consultations du programme et de l'exemple sont donc symétriques.

4. (RECORD) l'utilisateur est actif comme précédemment, mais son modèle de comportement est plus sûr, et de fait les consultations de l'exemple sont plus espacées. Ce mode d'interaction est d'une certaine façon l'opposé du précédent, dans le sens où les rôles de l'utilisateur humain et du système sont inversés. En effet, l'avantage pour l'utilisateur de laisser le programme s'exécuter parallèlement « en tâche de fond » est d'être immédiatement prévenu d'une erreur (débordement, dépassement de capacité, boucle infinie …). On peut alors dire que c'est le système qui supervise l'utilisateur.

L'identification de ces quatre types d'interaction confirme l'assertion que l'usage ou pas de la fonction de programmation avec exemple n'est pas liée au niveau en algorithmique de l'apprenant (1 et 3 correspondent à des novices, 2 et 4 à des programmeurs plus avertis…).

2.2.4 Questionnaire de satisfaction

En complément des indicateurs oculométriques et des traces d'interaction, qui nous permettent d'analyser le comportement et les stratégies des utilisateurs, nous avons souhaité de façon plus informelle connaître leur opinion sur le logiciel, afin d'analyser l'utilité et l'utilisabilité *perçues*.

Pour cela, nous nous sommes appuyés sur un questionnaire de satisfaction, composé de 15 questions. La plupart des questions sont des questions fermées où l'utilisateur devait indiquer son degré d'adhésion aux affirmations proposées. Ils avaient le choix entre 5 niveaux de réponse partant de 1 (pas du tout d'accord) et allant jusqu'à 5 (tout à fait d'accord). Ces affirmations sont les suivantes :

1. L'interface du logiciel MELBA me plaît.
2. La prise en main du logiciel MELBA me semble facile.
3. Je me repère facilement dans l'interface du logiciel MELBA.
4. Je n'ai éprouvé aucune difficulté à me servir du logiciel MELBA.
5. J'ai rapidement repéré les différentes parties de l'interface de MELBA.
    a. J'ai clairement identifié leurs rôles respectifs.
6. L'agencement des différentes parties de l'interface de MELBA me semble logique.
7. Les différentes parties de l'interface de MELBA me semblent faciles à comprendre.
8. L'agencement des informations (ex : liste déroulante…) dans la partie « instruction » de l'interface de MELBA me semble logique.

9. Les informations (ex : liste déroulante…) dans la partie « instruction » de l'interface de MELBA me semblent faciles à comprendre.
10. La partie « historique » du logiciel est bien mise en valeur et visible.
11. La hiérarchisation des informations dans la zone « historique » est facile à voir.

Les autres questions appelaient une réponse booléenne, que l'utilisateur pouvait commenter :

1. Trouvez-vous que l'objectif pédagogique du logiciel MELBA est intéressant ?
2. Utiliseriez-vous ce logiciel pour vous aider dans votre cursus universitaire ?
3. Avez-vous utilisé les trois boutons d'édition du programme (ajouter, supprimer, remplacement d'une instruction) ?
4. Avez-vous utilisé la zone « historique » ?

Enfin, les sujets étaient invités à commenter plus librement ce qui pourrait, de leur point de vue, être amélioré.

Dans l'ensemble, l'aspect esthétique de l'interface était apprécié des sujets. La prise en main de l'application a globalement été jugée facile par 2/3 des sujets, bien que le dernier tiers n'ait trouvé ni facile ni difficile cet aspect. Cinq sujets sur six n'ont éprouvé aucune difficulté pour se repérer dans l'interface. En revanche, deux d'entre eux ont éprouvé des difficultés dans l'utilisation de MELBA. Tous les sujets ont trouvé que l'objectif pédagogique du logiciel Melba était intéressant et ils ont déclaré être prêts à l'utiliser pour les aider dans leur cursus, à l'exception d'un sujet qui considère avoir un niveau supérieur en algorithmique.

Les sujets ont visiblement rapidement repéré et identifié les différentes parties de l'interface ainsi que leurs rôles respectifs. L'agencement de ces parties a été jugé plutôt « logique » et relativement facile à comprendre, mis a part une personne qui ne s'est pas prononcée sur ce dernier point. Les sujets ont tous utilisé les boutons d'édition du programme « ajouter », « supprimer ». En revanche, la moitié d'entre eux n'ont pas utilisé le bouton « modification » présent dans cette partie. Ils ont déclaré ne pas en avoir eu besoin ou n'en avoir pas vu l'utilité. Mais peut être ne l'ont-ils pas utilisé car ils n'en ont pas compris le fonctionnement.

Visiblement, l'agencement des différentes listes déroulantes présentes dans la zone « opération », a été jugé logique et l'utilisation des listes, facile à comprendre par tous les sujets. Concernant la zone « historique », deux des sujets ont trouvé qu'elle n'était pas assez mise en valeur et quatre sujets ne l'ont pas utilisée car il « n'en ont pas vu l'utilité » ou « ne l'ont pas trouvé assez mise en valeur ou lisible ». Visiblement cette partie n'est pas assez mise en valeur et l'information qui est proposée n'est pas comprise. Ce résultat confirme ce que nous avons pu observer (au niveau des « hot spot ») : l'intensité des fixations dans cette zone est faible.

Concernant les commentaires, ils concernent plus des problèmes d'utilisabilité (qui ont pour certains été corrigés depuis) et ne remettent pas en question les concepts et les principes de fonctionnement du logiciel :

- Sujet 1 : *« L'avis d'erreur avec pointeur sur la partie de l'arbre comportant l'erreur. Arbre en mode historique au lieu d'historique et arbre dissociés (ou historique vertical accessible par onglet). Parade contre les boucles infinies. »*

- Sujet 2 : *« Aucun commentaire. »*
- Sujet 3 : *« Pouvoir déplacer les instructions dans l'arbre de construction en tirant, et mettre en valeur les erreurs dans la partie historique afin de repérer et décoder la séquence d'instructions défaillantes. »*
- Sujet 4 : *« une action(un bouton) permettant de glisser une fonction déja écrite dans une autre, Multitâche, éviter d'avoir une succession de messages d'erreur »*
- Sujet 5 : *« Il faudrait plus mettre en évidence les instructions « begin » et « end » pour pouvoir faire plus facilement la différence avec le reste des instructions. »*
- Sujet 6 : *« Il faudrait que l'on puisse effacer une seule chose à la fois. Modification parfois difficile, mais sinon logiciel assez clair dans l'ensemble ».*

Les utilisateurs de MELBA ont signalé avoir rencontré des difficultés dans la modification du programme. Notamment sur le fait, « où doit-on se situer pour ajouter des instructions ou les modifier ». Cette application a été bien perçue par l'ensemble des utilisateurs, ils ont tous émis un avis positif et ont trouvé MELBA intéressant et utile.

3 Conclusion

Dans ce chapitre, nous cherchons à évaluer de façon rigoureuse la pertinence de l'approche « basée sur exemple » dans le cadre de l'apprentissage de la programmation.

Pour ce faire, nous menons sur l'environnement d'apprentissage MELBA une étude expérimentale en conditions réelles. Nous expliquons le choix des méthodes, et des métriques, que nous avons appliqué à deux expérimentations avec des étudiants, par rapport aux différentes méthodes connues dans la littérature (comparative, ethnographique, entrevues, occulométrie, verbalisation …) que nous avons présentées selon des critères reconnus (hors ligne ou en ligne, quantitatives ou qualitatives).

Nous interprétons les résultats de ces expériences. Les résultats quantitatifs montrent une différence significative concernant une mesure particulière (et importante) : le taux de réussite, et un avantage moins significatif au niveau de la moyenne. Cet avantage de l'utilisation du système semble profiter le plus aux plus faibles des apprenants, et plus marginalement aux bons élèves. Les résultats qualitatifs et la comparaison avec des études préalables en psychologie de la programmation nous permettent de compléter ces résultats, et de leur fournir un cadre d'interprétation.

Nous complétons ensuite ces études comparatives en conditions réelles par une expérimentation en milieu contrôlé, ayant pour but d'analyser les usages que font les étudiants des outils d'interaction avec le programme. Ces mesures des usages de l'environnement nous ont permis de découvrir plusieurs profils d'utilisation, indépendants du niveau en programmation. L'association de ces 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.

La combinaison de ces deux approches complémentaires nous permet d'arriver à un certain nombre de conclusions, concernant nos sept hypothèses de travail :

- 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). Les indicateurs oculométriques confirment une importance centrale de l'exemple dans les processus mentaux des apprenants, quel que soit le type de la tâche, tout en mettant en exergue de fortes variations intra et inter-individuelles sur son usage.
- 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 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).

- La programmation « sur l'exemple » peut être utilisée indifféremment par de parfaits débutants ou par des étudiants plus qualifiés (H2). Cependant, ce n'est pas le mode d'interaction le plus naturel pour nombre d'apprenants. Les usages que les étudiants font de ce mode d'interaction diffèrent significativement d'un individu à l'autre.

Conclusion générale et Perspectives

Ces dernières années, l'informatique a connu un essor formidable qui l'a amenée à devenir un outil indispensable dans de nombreuses activités professionnelles. Le contexte scientifique n'échappe pas à cette observation, et les 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 ainsi de bio-informatique, ou encore d' « informatique physique » - *computational physics*).

Pour autant, l'acquisition des compétences requises pour utiliser l'ordinateur comme outil de mesure scientifique, et, plus encore, pour la conception de programmes est réputée difficile. Ces constats ont conduit à l'émergence de la « Psychologie de la Programmation », un champ de la psychologie cognitive s'intéressant aux processus cognitifs mis en œuvre par les programmeurs pendant le développement. De nombreuses recherches ont ainsi été menées concernant le recensement et la catégorisation des erreurs de l'apprenant, la construction de modèles des schémas de connaissances et de compétences en programmation, les facteurs pouvant influer sur le succès en apprentissage académique…

Parallèlement, avec les progrès des capacités graphiques des machines, des recherches en informatique dans le champ de l'Interaction Homme-Machine se donnèrent pour objectif de rendre la programmation accessible à un public plus large : on parle ainsi de « Programmation basée sur l'exemple », et plus généralement de « End User Development » (développement par l'utilisateur final). Deux grands paradoxes peuvent cependant être observés à ce propos. D'une part, et alors que l'une des grandes idées de la « End User Programming » est la possibilité d'aider l'utilisateur à construire des programmes, le champ concret de l'apprentissage de la programmation fait figure de parent pauvre. Très peu de travaux ont été réalisés avec cet objectif pédagogique. D'autre part, malgré le grand nombre de systèmes développés selon les préceptes de la programmation basée sur l'exemple, il est aujourd'hui difficile de conclure quant à la validité de cette approche. En effet, on ne trouve pratiquement aucune étude permettant d'évaluer les techniques mises en œuvre. Tout au plus peut-on citer les applications qui ont franchi a barre de l'utilisation par une tierce personne, dans les domaines spécialisés que sont la conception technique (CAO) et la simulation.

L'objectif de notre travail de thèse a consisté à valider la pertinence de la mise en œuvre des concepts de la programmation basée sur l'exemple pour supporter la difficile phase d'initiation à la programmation. Pour cela, nous avons conçu, sur la base des travaux en psychologie et en didactique de la programmation d'une part, et de la riche expérience du domaine de la programmation basée sur l'exemple, un environnement original exploitant le

principe des Situations d'Apprentissage Actif, et l'avons soumis à une évaluation expérimentale pour valider ses hypothèses de conception.

Nous avons proposé au premier chapitre une synthèse des différentes définitions de la programmation qui tient à la fois compte de la structure de cette activité et de l'aspect cognitif. On peut dire de façon générale que programmer c'est :

- **« Faire faire …»** : un programme, à la base, se compose d'*instructions* ; la programmation se caractérise donc par une prise de recul de la part du programmeur. Il doit passer du domaine du savoir-faire inconscient à une représentation consciente de toutes les actions requises pour réaliser la tâche.

- « **… en différé …** », un programme est une *planification*, qui résultera en une réalisation *ultérieure* de la tâche, aussi le programmeur doit-il recourir à une représentation mentale de l'état de la tâche pour compenser la « perte de la manipulation directe ».

- « **… à un exécutant aux capacités limitées…** ». Lorsqu'on programme, on s'adresse à un exécutant parfaitement spécifié, et capable uniquement d'opérations formelles, et non sémantiques. La possible (et fréquente…) inadéquation entre les instruments dont dispose l'exécutant et les savoir-faire du programmeur concernant la tâche est source de nombreuses difficultés…

- « … **en utilisant un formalisme donné** ». Pour généraliser le comportement de l'exécutant sur plusieurs réalisations, le programmeur fait usage de notations (textuelles, graphiques, ou autres) exprimant les capacités de celui-ci.

Nous avons catalogué et classé les principaux types d'erreurs et de difficultés observés dans la littérature de la Psychologie de la programmation, que ces dernières soient liées aux concepts manipulés ou à l'environnement au sens large. Notre synthèse met en lumière le rôle négatif de l'environnement d'apprentissage :

- Dans le but de pouvoir s'appliquer à n'importe quel domaine, les langages de programmation proposent des outils d'un faible niveau sémantique, et imposent une représentation qui est totalement déconnectée du contexte et donc étrangère aux représentations que l'apprenant se fait à l'origine des objets du domaine.

- Les environnements de programmation ne maintiennent pas en permanence une représentation de l'état de la machine, et donc obligent l'apprenant à construire et animer, au prix d'une lourde charge cognitive, leur conception de celui-ci, qui peut être incohérente et non viable.

Elle étaye ainsi l'hypothèse fondatrice selon laquelle un environnement d'apprentissage inadapté peut jouer un rôle (négatif) de catalyseur dans l'apparition des difficultés des apprenants et dans la génération d'une frustration pouvant devenir, en elle-même un obstacle sérieux à la compréhension.

Notre deuxième chapitre définit et illustre un paradigme alternatif de conception de programmes, la « programmation basée sur l'exemple » (Myers 1986). Celle-ci a pour principale caractéristique que l'exécution se déroule parallèlement à la conception du programme. Après avoir expliqué les différents concepts associés à ce paradigme, nous analysons précisément les contraintes et les différentes applications de cette approche, dans le domaine de l'apprentissage de la programmation.

Nous aboutissons ainsi à un cahier des charges des fonctionnalités d'un environnement d'apprentissage de la programmation « basé sur l'exemple » :

- support d'un apprentissage expérimental
- intégration du rôle de l'enseignant dans la construction d'exemples supportant des Situations d'Apprentissage Actif (SAA)
- support d'un apprentissage incrémental.

Dans le troisième chapitre, nous déterminons le contexte de pertinence de chaque technique d'interaction, d'un point de vue pédagogique. Nous nous appuyons dans notre analyse sur un outil d'évaluation de notations, issu du champ de l'ergonomie cognitive, le « Cognitive Dimensions Framework » (Green 1989). Nous détournons légèrement cet outil de son utilisation « classique », qui est d'évaluer une notation ou un langage spécifique, en l'utilisant pour analyser d'un point de vue cognitif, et de façon globale, les différents types d'interactions et de visualisations de la littérature.

Cette analyse nous permet de définir la combinaison de techniques d'interaction la plus pertinente pour réaliser les fonctionnalités précédentes :

- Pour supporter un apprentissage expérimental, qui repose sur des SAA, nous nous appuierons sur **la programmation « avec exemple »,** qui permet une ***évaluation progressive*** qui soutient le bricolage et les cycles essai-erreur dans la conception du programme. Cette technique s'adapte parfaitement à des tâches de mise au point et induit peu de ***prévisualisation forcée*** (comparativement à l'approche « sur » exemple). Afin de ne pas perturber l'étudiant par des retours d'erreurs syntaxiques intempestifs (faible ***incitation à l'erreur***), il nous semble qu'un **style d'édition graphique et contraint** est le plus adapté. Pour autant le choix d'une **grammaire textuelle** nous paraît plus judicieux que celui d'un langage graphique iconique, car la ***viscosité par effet de bords*** que ces formalismes génèrent peut se révéler un frein à l'expérimentation, sans compter qu'ils induisent une ***visibilité*** moindre que celle du texte indenté. De façon complémentaire, des fonctionnalités d'**animation de programme** devraient permettre de soutenir la phase d'observation plus efficacement en cas d'évaluation globale du programme.
- Pour faciliter l'accrétion, il est recommandé de limiter le nombre de concepts à appréhender simultanément, c'est-à-dire de rechercher un fort ***gradient d'abstraction*** et une faible ***barrière d'abstraction***. Pour cela, il nous paraît préférable de combiner **représentations pragmatiques** et **symboliques**. En effet, en retardant l'introduction des concepts afférents aux structures de données, les représentations pragmatiques permettent d'utiliser la ***correspondance au domaine*** pour augmenter l'***expressivité*** des actions reconnues par le processeur (tout comme la ***visibilité*** de son état). La ***juxtaposition*** de ces deux styles de représentations pourrait être utilisée pour faciliter la création de schémas d'implémentations portant sur la modélisation des objets sous forme de structures de données informatiques.

Nous avons mis en œuvre ces différentes techniques, en développant MELBA (Metaphor-based Environment to Learn the Basics of Algorithmics), un EIAH que nous décrivons précisément dans le chapitre 4. Cette phase de développement a débouché sur un outil utilisable, qui s'est révélé assez robuste pour une utilisation en milieu réel.

Cet environnement comporte plusieurs aspects réellement originaux ou innovants :

- Sa principale caractéristique provient de l'éditeur de programmes « avec » exemple, qui est le premier réellement fonctionnel. A notre connaissance, MELBA est le seul environnement « finalisé » à proposer cette fonctionnalité (ce type d'interaction était déjà possible avec les interpréteurs de certains langages, mais seulement avec un terminal, le programme interprété n'étant pas sauvegardé, le retour en arrière en cas d'erreur impossible, etc.).

- MELBA est fait pour « apprendre à programmer » et non pour « programmer sans apprentissage ». Il est de ce fait atypique dans la « programmation basée sur exemple ». Il est également le seul système à combiner l'approche sur exemple, basée sur des conventions de dialogue en manipulation directe, et la manipulation du programme généré « avec » l'exemple.

- De même, il est original pour un environnement basé sur le paradigme impératif structuré d'être « dirigé par les modèles », et de se positionner explicitement au niveau sémantique ; la totalité des outils décrits dans le chapitre trois se positionnent au niveau langage.

- Enfin, il est le seul à proposer d'aborder de façon incrémentale, sur autant de niveaux sémantiques, l'apprentissage de la programmation. Le concept de « mini-langage » ou « sous-langage » est ancien, mais ses implémentations se limitent généralement à un seul niveau sémantique (voire deux).

Enfin, nous avons, dans le chapitre cinq, développé un protocole de validation expérimentale de l'outil, en nous basant sur une analyse des différentes approches et métriques de la littérature. Nous nous sommes appuyé sur deux séries d'expérimentations aux approches complémentaires, pour répondre aux questions de recherche suivantes :

- La programmation basée sur exemple est-elle efficace pédagogiquement, et si oui quels schémas, savoirs, compétences recouvrent son domaine d'application ?

- Est-elle accessible, voire naturelle pour tous ? Ou convient-elle mieux à certains types d'apprenants, en fonction de leur niveau de maîtrise, de leur profil cognitif ?

Nos analyses des résultats, aussi bien quantitatifs que qualitatifs nous permettent d'ébaucher *in fine* une réponse à ces questions :

- L'évaluation progressive permise par une exécution interactive joue un rôle positif dans l'apprentissage d'un modèle d'exécution du programme, en particulier en compréhension de programme, et en recherche / correction d'erreurs. Les indicateurs oculométriques confirment une importance centrale de l'exemple dans les processus mentaux des apprenants, quel que soit le type de la tâche, tout en mettant en exergue de fortes variations intra et inter-individuelles sur son usage.

- Pour cela, des techniques de visualisation de programme proposant un modèle graphique de l'état du système sont nécessaires ; pour être efficaces, elles doivent fournir une vue d'ensemble où les informations importantes peuvent être extraites rapidement. Combiner plusieurs modèles graphiques de la même information semble dans ce cadre inutile et superflu.

- L'animation de l'exécution du programme, quoique naturellement utilisée par beaucoup d'étudiants, ne semble pas apporter de gain quantifiable.

- La programmation « sur l'exemple » peut être utilisée indifféremment par de parfaits débutants ou par des étudiants plus qualifiés. Cependant, ce n'est pas le mode d'interaction le plus naturel pour nombre d'apprenants. Les usages que les étudiants font de ce mode d'interaction diffèrent significativement d'un individu à l'autre.

Ces recherches nous ouvrent ainsi un certain nombre de perspectives, pour la poursuite de recherches sur l'usage de la programmation sur exemple dans le domaine éducatif :

- Premièrement, si nos résultats étayent la thèse de la pertinence d'un environnement interactif et adapté pour apprendre à programmer, certaines questions restent en suspens. Par exemple, nous avons constaté des différences interindividuelles très fortes sur les préférences en termes de style d'interaction, qui ne sont pas corrélées avec le niveau de l'utilisateur. Une piste sérieuse serait d'étudier la possible corrélation du style d'apprentissage du sujet ou d'une autre caractéristique cognitive avec ces préférences. Peut-être ces caractéristiques cognitives représentent-elles également un facteur influant sur l'efficacité de la programmation « avec exemple ». Ces travaux pourraient donner lieu à la définition d'un modèle de l'apprenant au sein de l'outil, permettant d'adapter les techniques d'apprentissage proposés à l'élève.

- Deuxièmement, ces résultats plus qu'encourageants nous incitent à développer davantage l'environnement MELBA pour supporter un plus grand nombre de concepts et de domaines de compétences. Par exemple, la version actuelle de l'environnement ne supporte pas la construction de programmes récursifs, alors que la récursivité tient une place majeure dans le domaine conceptuel de la programmation, et que l'empilement / dépilement du contexte pourrait très certainement être visualisé graphiquement sous MELBA, et fournir ainsi un support à la compréhension du phénomène pour des penseurs « visuels » ou « actifs ».

- Troisièmement**,** en permettant à l'étudiant d'auto-évaluer à chaque instant la correction de ses conceptions, celui-ci acquiert une certaine autonomie par rapport à l'enseignant. Il nous parait donc intéressant de tester l'approche « basée sur l'exemple » dans le contexte de formation « ouverte et à distance », où l'étudiant a, par définition, des contacts moins directs avec les formateurs. Il semble qu'il tirerait un grand avantage du feedback sémantique en temps réel que lui offre l'exemple. Dans ce contexte, il pourrait également être intéressant d'utiliser MELBA comme instrument d'expérimentation en vue de la définition d'un modèle des connaissances en programmation. Ce modèle, pourrait, en retour, être réinjecté dans l'environnement pour modéliser en temps réel les connaissances et les difficultés de l'apprenant, afin de lui offrir une aide personnalisée.

Bibliographie

Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M. (2005). *Effects of Experience on Gaze Behavior during Program Animation*. **PPIG Workshop**, University of Sussex, Brighton, UK.

Bednarik, R., Tukiainen, M. (2006). *An Eye-tracking Methodology for Characterizing Program Comprehension*. **Symposium on Eye Tracking Research and Applications**, San Diego, CA, USA, ACM Press.

Ben-Ari, M. (1998). *Constructivism in Computer Science Education*. **29th ACM SIGCSE Technical Symposium on Computer Science Education**, Atlanta Georgia, ACM press.

Blackwell, A. (2002). *What is Programming?* PPIG, Brunel University, London, UK.

Booth, S. (1992). **Learning to program: a phenomenographic perspective.**

Byrne, P. L., G. (2001). *the Effects of Student Attributes on Success in Programming.* **ITICSE**, Canterbury, United Kingdom.

Carbone, A., Hagan, D. L. & Sheard, J. (1998). *Consolidate, preserve, and build: a tutor training program for a new school.* Australasian Conference on Computer Science Education, Brisbane, Queensland, Australia.

Chang, S.-K. (1986). *Visual languages and iconic languages.* **Visual languages.** S.-K. Chang, T. Ichikawa and P. Ligomenides. New-York, Plenum Press: 1-7.

Coutaz, J., L. Nigay, et al. (1995). *Agent-Based Architecture Modelling for Interactive Systems.* **Critical Issues in User Interface Engineering**. P. Palanque and D. Benyon. London, Springer-Verlag: 191-209.

Détienne, F. (1998). **Génie logiciel et psychologie de la programmation.**

Dix, A., J. Finlay, et al. (1993). **Human-Computer Interaction**, Prentice Hall.

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

Duchâteau, C. (1992). *From "DOING IT ..." to "HAVING IT DONE BY ...": The Heart of Programming. Some Didactical Thoughts.* NATO Advanced Research Workshop "Cognitive Models and Intelligent Environments for Learning Programming", S Margherita, Italy.

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

Garner (2005). *My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems*. **Australian Computing Education Conference**, Newcastle, Australia.

Girard, P. (1992). **Environnement de Programmation pour Non-Programmeur et Paramétrage en Conception Assistée par Ordinateur : le système LIKE**. LISI/ENSMA, Université de Poitiers: 195.

Girard, P. (2000). **Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur**. LISI/ENSMA. Poitiers, Université de Poitiers: 92.

Goold, A. a. R., R. (2000). *"Undergraduates in business computing and computer science (poster session)."* **SIGCSE Bulletin 32(3): 188.**

Goold, A. R., R. (2000). *"Factors affecting Performance in Firs-Year Computing."* **SIGCSE Bulletin 32(2): 39-43.**

Green, T. K. G. (1989). *Cognitive dimensions of notations*. **People and Computers.**

Green, T. R. G. P., M. (1996). *"Usability analysis of visual programming environments: a " cognitive dimensions" framework."* **Visual languages and computing 7: 131-174.**

Guéraud, V. (2005). **Approche auteur pour les Situations Actives d'Apprentissage : Scénarios, Suivi et Ingénierie**. Informatique, Grenoble.

Guibert, N. Guittet, L. and Girard, P. (2005). *Validation d'une approche « basée sur exemples » pour l'apprentissage de la programmation* **17e Conférence francophone sur l'Interaction Homme-machine (IHM 2005)**, Toulouse.

Halbert, D. (1984). **Programming by Example**. Berkeley, University of California: 121.

Hartree, D. R. (1950). **Calculating instruments and machines**, Cambridge University Press.

Hoc, J., Green, T., Samurc¸ay, R., & Gilmore, D. (1990). **Psychology of programming.**

Hû, O. T., P. (1998). *Proposition de critères d'évaluation de l'interface homme-machine des logiciels multimédias pédagogiques*. **IHM**, Nantes.

Kaasboll, J. (2002). **Learning Programming**, University of Oslo. 2003.

Kahn, K. (2001). *How Any Program Can Be Created by Working with Examples*. **Your Wish is My Command**. H. Lieberman: 21-44.

Kolb, D. A. (1986). **Learning Style Inventory : Technical Manual.**

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003). *"The BlueJ system and its pedagogy."* **Journal of Computer Science Education 13(4).**

Lattu, M., Tarhio, J. and Meisalo, V. (2000). *How a Visualization Tool Can Be Used - Evaluating a Tool in a Research & Development Project*. **PPIG Workshop**, Cozenza, Italy.

Lau, T., S. Wolfman, et al. (2001). *Learning Repetitive Text-editing Procedures with SMARTedit*. **Your Wish is My Command**. H. Lieberman: 209-226.

Letondal, C. (2001). **Interaction et Programmation.**

Lieberman, H. (1993). *Tinker: A Programming by Demonstration System for Beginning Programmers*. **Watch What I Do: Programming by Demonstration**. A. Cypher. Cambridge, Massachusetts, The MIT Press: 49-66.

Mancy, R. R., N. (2004). *Aspect of Cognitive Style and Programming*. **PPIG Workshop**, Institute of Technology, Carlow, Ireland.

Mayer, R. (1988). **Teaching and Learning Computer Programming**. Hillsdale, Erlbaum.

Mc Iver, L. K. (2001). **Syntactic and Semantic Issues in Introductory Programming Education**. Computer Science, Monash, Australia: 200.

McDaniel, R. G. and B. A. Myers (1999). *Gamut : Creating Complete Applications Using Only Programming by Demonstration*. **Human Factors in Computing Systems (CHI'99)**

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.

Myller, N. (2004). *The Fundamental Design Issues of Jeliot 3*. Computer Science, Joensu.

Nevill-Manning, C. G. and Witten, Ian H. (1997). *"Compression and Explanation Using Hierarchical Grammars."* **The Computer Journal 40.**

Nevill-Manning, C. G. and Witten, Ian W. (1997). *"Identifying Hierarchical Structure in Sequences: A linear-time algorithm."* **Journal of Artificial Intelligence Research 7: 67-82.**

Nigay, L. (1994). **Conception et Modélisation Logicielle des Systèmes Interactifs : Application aux Interfaces Multimodales**. CLIPS/IMAG. Grenoble, Université Joseph Fourier.

Nogry, S. Jean-Daubias, S. Ollagnier-Beldame, M. (2004). *Evaluation des EIAH: une nécessaire diverstité des méthodes* **TICE**, Compiègnes.

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

Pane (2001). *Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems*. Ratanamahatan: 26**.**

Pane (2002). *A Programming System for Children that is Designed for Usability***. Computer Science.** Pittsburgh, PA, Carnegie Mellon University: 190.

Papert, S. (1980). **Mindstorms: Children, Computers and Powerful Ideas** New York, Basic Books.

Pea, R. D. (1986). *"Language-Independent Conceptual "Bugs" in Novice Programming."* **Journal of Educational Computing Research 2(1): 25-36.**

Perkins, D. N. a. H., C. and Hobbs,R. and Martin, F. and Simmons, R. (1986). *Conditions of Learning in Novice Programmers.* **studying the Novice Programmer,** Lawrence Erlbaum Associates: 261-279.

Plaisant, C. (2004). *The Challenge of Information Visualization Evaluation.* **Advanced Visual Interfaces,** Gallipoli, Italy.

Potier, J.-C. (1995). **Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP.** LISI/ENSMA, Université de Poitiers: 140.

Rogalsky J., Samurçay. R., Hoc, J-M. (1988). *"L'apprentissage des méthodes de programmation comme méthodes de résolution de problème."* **Le travail humain(51): 309-320.**

Scapin, D. (1986). **Guide ergonomique de conception des interfaces homme-machine,** INRIA Rocquencourt.

Shu, N. (1986). *Visual Programming Languages: a perspective and a dimensional analysis*. **Visual Languages.** S.-K. Chang, T. Ichikawa and P. Ligomenides. New-York, Plenum Press: 10-34.

Shute, V. J. R., J. W. (1993). *"Principles for Evaluating intelligent Tutoring Systems."* **Journal of Artificial Intelligence and Education 4: 245-271.**

Sleeman, D. (1988). *An introductory Pascal class: A study of student errors.* **Teaching and learning Computer Programming.** R. E. Mayer, Lawrence Erlbaum Associates.

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

Smith, D. C. (2000). ***"Novice Programming comes of Age."*** **Communications of the ACM 43 (3): 75-81.**

Soloway, E. a. I., S. (1988). **Empirical Studies of Programmers.** Norwood, Ablex.

Soloway, E. and Spohrer, J. C. (1989). **Studying the novice programmer,** Lawrence Erlbaum Associates.

Spohrer, J. C. (1986). Novice Mistakes: Are the Folk Wisdoms Correct?

Spohrer, J. G. and Soloway, E. and Pope, E. (1985). *Where the bugs are.* **CHI'85 Conference on Human Factors in Computing Systems.**

Spohrer, J. G. and Soloway, E. (1986). *Analysing the high frequency bugs in novice programs.* **First Workshop on Empirical Studies of Programmers.**

Van Haaster, K. a. H., D. (2003). *"Teaching and learning with BlueJ: an Evaluation of a Pedagogical Tool."* **Issues in Informing Science and Information Technology 13(4).**

Wilkes, M. V. (1956). **Automatic digital computers.,** Cambridge University Press.

Wilson, B. C. (2000). *Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors.* **SIGCSE,** ACM.

Wolber, D. (1996). *Pavlov : Programming by Stimulus-Response Demonstration*. **Human Factors in Computing Systems (CHI'96)**, Vancouver, Canada, ACM/SIGCHI.

Wrubel, M. H. (1959). **A primer of programming for digital computers,** Mc Graw Hill**.**

Liste des publications liées à ce mémoire de thèse

1 Conférences internationales anglophones avec comité de programme.

1.1 Articles longs :

2005 **A STUDY OF THE EFFICIENCY OF AN ALTERNATIVE PROGRAMMING PARADIGM TO TEACH THE BASICS OF PROGRAMMING.**

Nicolas Guibert, Laurent Guittet and Patrick Girard., WCCE (World Conference on Computers in Education), IFIP, 2005, Cape town, South Africa .

PROGRAMMING BY EXAMPLE: A POWERFUL PARADIGM TO SUPPORT THE EXPERIMENTAL ACQUISITION OF PROGRAMMING SKILLS.

Nicolas Guibert, Laurent Guittet and Patrick Girard., HCII 2005, Las Vegas, USA.

1.2 Article court :

2004 **"EXAMPLE-BASED PROGRAMMING: A PERTINENT VISUAL APPROACH FOR LEARNING TO PROGRAM"** (article + poster).

Nicolas Guibert, Laurent Guittet and Patrick Girard., Advanced Visual Interfaces (AVI), Gallipoli, Italy, acm press, 2004, pp. 358-361

1.3 Symposium :

2003 **TEACHING AND LEARNING PROGRAMMING WITH A PROGRAMMING BY EXAMPLE SYSTEM.**

Nicolas Guibert and Patrick Girard., International Symposium on End User Development, Sankt Augustin (Bonn), Germany, EUD - net, 2003

2 Conférences internationales francophones avec comité de programme.

2.1 Articles longs :

2006 **PERFORMANCES ET USAGES D'UN ENVIRONNEMENT D'APPRENTISSAGE DE LA PROGRAMMATION « BASE SUR EXEMPLE ».**

Nicolas Guibert, Patrick Girard and Laurent Guittet., ERGO'IA , 2006, pp. 103-110

2005 **VALIDATION D'UNE APPROCHE « BASEE SUR EXEMPLES » POUR L'APPRENTISSAGE DE LA PROGRAMMATION.**

Nicolas Guibert, Laurent Guittet and Patrick Girard., IHM 2005, Toulouse, France.

2004 **APPRENDRE LA PROGRAMMATION PAR L'EXEMPLE : METHODE ET SYSTEME.**

Nicolas Guibert, Laurent Guittet and Patrick Girard., Technologies de l'Information et de la Connaissance dans l'Enseignement Supérieur et l'Industrie (TICE), UTC Compiègnes- France, 2004

2.2 Articles courts :

2005 **INITIATION A LA PROGRAMMATION « PAR L'EXEMPLE » : CONCEPTS, ENVIRONNEMENT, ET ETUDE D'UTILITE.**

Nicolas Guibert, Laurent Guittet and Patrick Girard., EIAH 2005, Montpellier, France .

2003 **PROGRAMMATION SUR EXEMPLE ET ENSEIGNEMENT ASSISTE PAR ORDINATEUR DE L'ALGORITHMIQUE : LE PROJET MELBA**

Nicolas Guibert and Patrick Girard., 15° Conférence Francophone sur l'Interaction Homme-Machine (IHM'2003), vol. 1, Caen, ACM Press, 2003, pp. 248-251

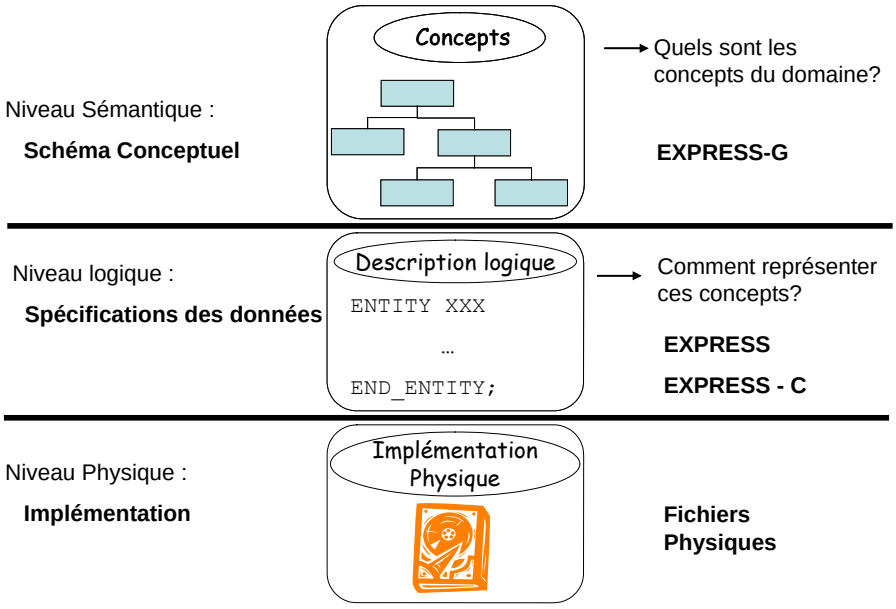
Annexe A : Modèles conceptuels EXPRESS

1 Présentation d' EXPRESS.

L'environnement EXPRESS comprend un ensemble d'outils, qui s'appuient sur différentes normes ISO, dans l'objectif de supporter l'ingénierie des modèles et des connaissances à trois niveaux (Conceptuel, Sémantique, Physique) – voir **Erreur! Source du renvoi introuvable.**:

- un langage de modélisation de l'information : EXPRESS (ISO 10303-11 :1994) ;
- un format d'instances qui permet l'échange de données entre systèmes (ISO 10303-21 :1994) ;
- une infrastructure de méta-modélisation associée à une interface d'accès normalisée, appelée SDAI, pour accéder et manipuler simultanément les données et le modèle de n'importe quel modèle EXPRESS ;
- un langage déclaratif de transformation de modèles (ISO 10303-14 :2002).

Enfin, le langage EXPRESS possède un langage procédural complet (analogue à PASCAL) pour l'expression de contraintes. Au prix de quelques extensions mineures, ce langage peut également être utilisé comme langage impératif de transformation de modèles.



EXPRESS en lui-même est un langage formel qui permet de spécifier les concepts d'un domaine les structures de données permettant de représenter ces concepts sous forme de données traitables par machine. C'est un langage orienté objet : il supporte l'héritage et le polymorphisme. Il est également ensembliste : il permet les manipulations de collections. Nous présentons succinctement ses composantes structurelles, descriptives et procédurales.

En EXPRESS, l'univers du discours est décomposé en SCHEMA. Chaque schéma « USE » ou « REFERENCE » d'autres schémas existants. Le découpage de la modélisation d'un domaine donné peut ainsi se faire selon deux approches qui peuvent être combinées :

- *horizontal* : chaque schéma modélise un sous domaine du domaine considéré ;
- *vertical* : chaque schéma représente une modélisation du domaine à un différent niveau d'abstraction.

Ces schémas se décomposent en entités (classes) qui permettent de pour réaliser l'abstraction et la catégorisation des objets du domaine de discours. Les entités sont hiérarchisées par des relations d'héritage qui peuvent relever de l'héritage simple, multiple ou répété. Une entité est décrite par des attributs. Les attributs permettent de caractériser les instances d'une entité par des valeurs. Ces valeurs peuvent être définies indépendamment de tout autre attribut (attributs libres) ou calculées à partir de valeurs d'autres attributs (attributs dérivés). Elles appartiennent au type de données associé à chaque attribut. Le langage EXPRESS définit quatre familles de types :

- Les types simples : ce sont essentiellement les types chaînes de caractères (STRING), numériques (REAL, BINARY, INTEGER) ou logiques (LOGICAL, BOOLEAN) ;
- Les types nommés : ce sont des types construits à partir de types existant auxquels un nom est associé. Un type nommé peut être défini par restriction du domaine d'un type existant. Cette restriction peut être faite par la définition d'un prédicat qui doit être respecter par les valeurs du sous domaine créé. Il peut également être défini par énumération (ENUMERATION) ou par l'union de types (SELECT) qui, dans un contexte particulier, sont alternatifs.
- Les types agrégats : ce sont des types qui permettent de modéliser les domaines dont les valeurs sont des collections. Les types de collections disponibles sont les ensembles (SET), les ensembles multi-valués (BAG), les listes (LIST) et les tableaux (ARRAY). Un type collection n'a pas a être nommé : il apparaît directement dans la définition de l'attribut qu'il type.
- Les types entités : un attribut d'un tel type représente une association.

Par ailleurs, le langage EXPRESS permet de spécifier un grand nombre de contraintes portant sur les instances de ces entités. On distingue :

- Les contraintes locales, qui s'appliquent individuellement sur chacune des instances de l'entité ou du type sur lequel elles sont définies.
- Les contraintes globales, qui nécessitent une vérification globale sur l'ensemble des instances d'une entité donnée.

Les contraintes locales (WHERE) sont définies au travers de prédicats auxquels chaque instance de l'entité (ou chaque valeur du type), sur lequel elles sont déclarées, doit obéir. Ces

prédicats permettent par exemple de limiter la valeur d'un attribut en définissant un domaine de valeurs, ou encore de rendre obligatoire l'évaluation d'un attribut optionnel selon certains critères. Concernant les contraintes globales :

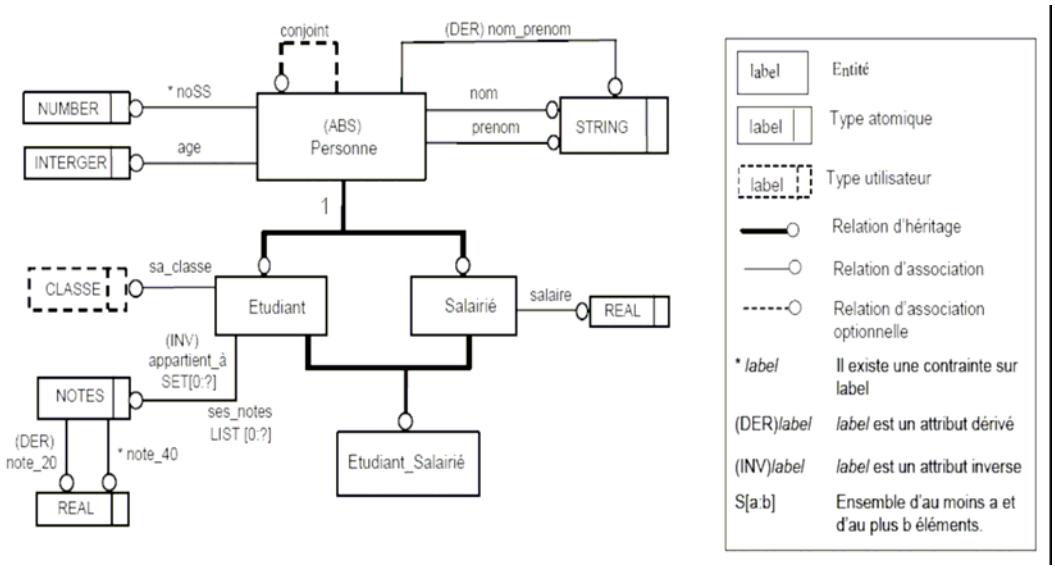
- La contrainte d'unicité (UNIQUE) contrôle l'ensemble de la population d'instances d'une même entité pour s'assurer que les attributs auquel s'applique la contrainte possèdent une valeur unique sur toute cette population.
- La contrainte de cardinalité inverse (INVERSE), permet de spécifier la cardinalité de la collection d'entités d'un certain type qui référencent une entité donnée dans un certain rôle. L'attribut inverse exprime l'association entre une entité sujette et des entités référençant l'entité sujette par un attribut particulier.
- Les règles globales (RULE) ne sont pas déclarées au sein des entités mais sont définies séparément. Elles permettent d'itérer sur une (ou plusieurs) population(s) d'entités pour vérifier des fonctions logiques qui doivent s'appliquer à l'ensemble des instances de ces populations.

Des fonctions (FUNCTION) et des procédures (PROCEDURE) peuvent être écrites. De plus, le langage EXPRESS propose un ensemble de fonctions prédéfinies : QUERY (requête sur les instances), SIZEOF (taille d'une collection), TYPEOF (introspection : donne le type d'un objet), USED_IN (calcul dynamique des associations inverses).

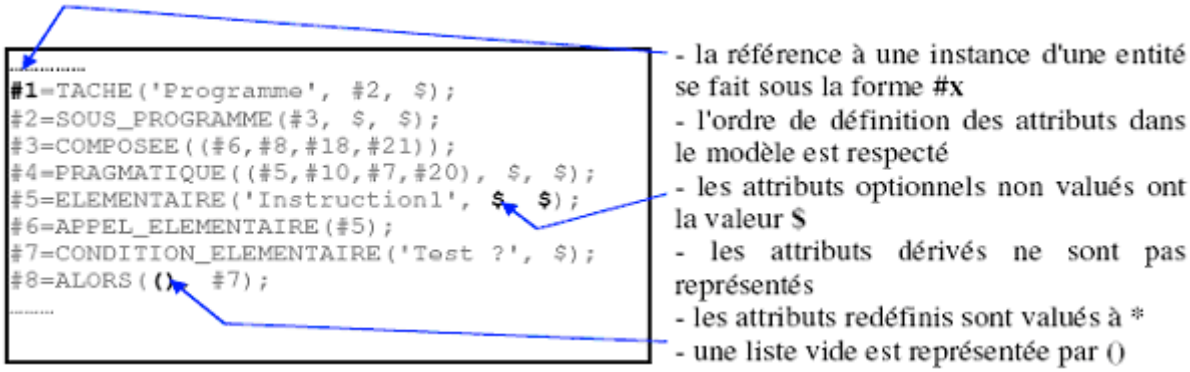
Pour illustration, considérons la **Erreur! Source du renvoi introuvable.**. Dans cet exemple, nous définissons un schéma de nom universitaire. Ce schéma est constitué de cinq entités. Les entités « étudiant » et « salarié » qui héritent de l'entité Personne. L'entité « étudiant_salarié » qui hérite des entités (héritage multiple) étudiant et salarié. Enfin l'entité notes qui est co-domaine de l'attribut « ses_notes » de étudiant. L'entité Personne définit un attribut dérivé (« nom_prenom ») qui est associé à la fonction EXPRESS (« nom_complet ») retournant la concaténation de l'attribut nom et prénom d'une instance de l'entité Personne. On peut remarquer la contrainte locale dans l'entité notes qui permet de vérifier que les valeurs de l'attribut note_40 sont comprises entre 0 et 40.

```
SCHEMA Universitaire ;
TYPE CLASSE = ENUMERATION OF (A1, A2, A3);
END_TYPE;
ENTITY Personne ;
SUPERTYPE OF ONEOF (Etudiant, Salairié);
  noSS : NUMBER;
  nom : STRING ;
  prénom : STRING;
  age : INTEGER;
  conjoint : OPTIONAL Personne;
DERIVE
 nom_prenom : STRING := Nom_complet (SELF);
UNIQUE ur1 : NoSS;
END_ENTITY ;
ENTITY Notes ;
  module_ref : STRING;
  note_40 : REAL ;
INVERSE
 appartient_à : SET[0 :?] OF Etudiant FOR ses_notes ;
WHERE
  wr1: {0 <=  note_40 <= 40};
DERIVE
  note_20 : REAL :=  note_40/2;
END_ENTITY ;

ENTITY Etudiant ;
SUBTYPE OF (Personne);
 sa_classe : CLASSE;
 ses_notes : LIST[0 : ?] OF NOTES;
END_ENTITY ;
ENTITY Salarié;
SUBTYPE OF (Personne)
  salaire : REAL;
END_ENTITY ;
ENTITY Etudiant_Salarié;
SUBTYPE OF( Salarié, Etudiant);
END_ENTITY ;

FUNCTION Nom_complet(per : Personne): STRING;
RETURN (per.nom + ' ' + per.prenom);
END_FUNCTION;

END_SCHEMA ;
```

EXPRESS possède également une représentation graphique (sauf pour les contraintes), proche d'un schéma de classes UML, appelée EXPRESS-G. Cette représentation a pour objectif de donner une vue synthétique d'un schéma EXPRESS et est adaptée aux phases préliminaires d'une conception .En EXPRESS-G, les entités sont représentées par des rectangles avec à l'intérieur leur nom. Les notions d'attribut et d'héritage sont décrites par la figure ci-après. Les attributs peuvent être précédés des diminutifs suivants : (DER) pour signifier que l'attribut est dérivé, (INV) pour inverse, (RT) pour redéfini.



Au niveau physique, les instances de ces différentes entités sont représentées conformément à la norme d'échange de données ISO 10303-21 :1994, et référencées par des « OID » (Object IDentifier) qui permettent d'appeler des instances par le biais des attributs des entités.



Ce fichier physique peut ensuite être manipulé à travers un langage de requêtes telles que :

- **create**

Syntaxe : `create <SCHEMA_NAME> <TYPE_NAME>`

Nom du type

Cette primitive sert à créer une instance (ou objet) de l'entité TYPE_NAME, où SCHEMA_NAME est le schéma dans lequel figure l'entité. Il est impératif que le nom du schéma et le nom de l'entité soient en lettres majuscules. La valeur de retour de cette primitive est l'OID.

- **Population**

Syntaxe : `population <SCHEMA_NAME> <ENTITY_NAME> [-value]`

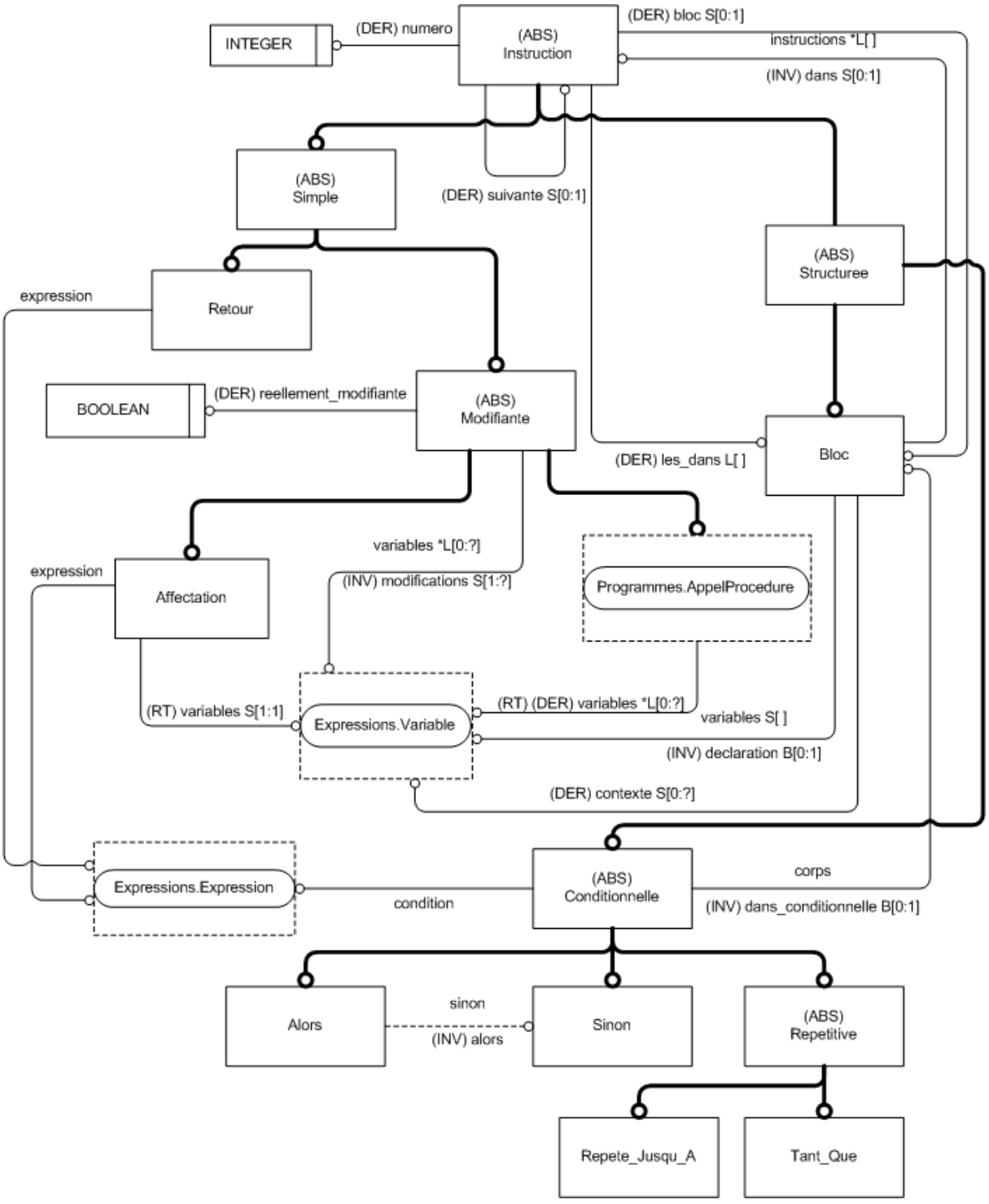
Nom du modèle de données — Nom de l'entité — Option de la requête

La commande population permet de lister toutes les entités d'un type et d'un schéma donné en paramètre.

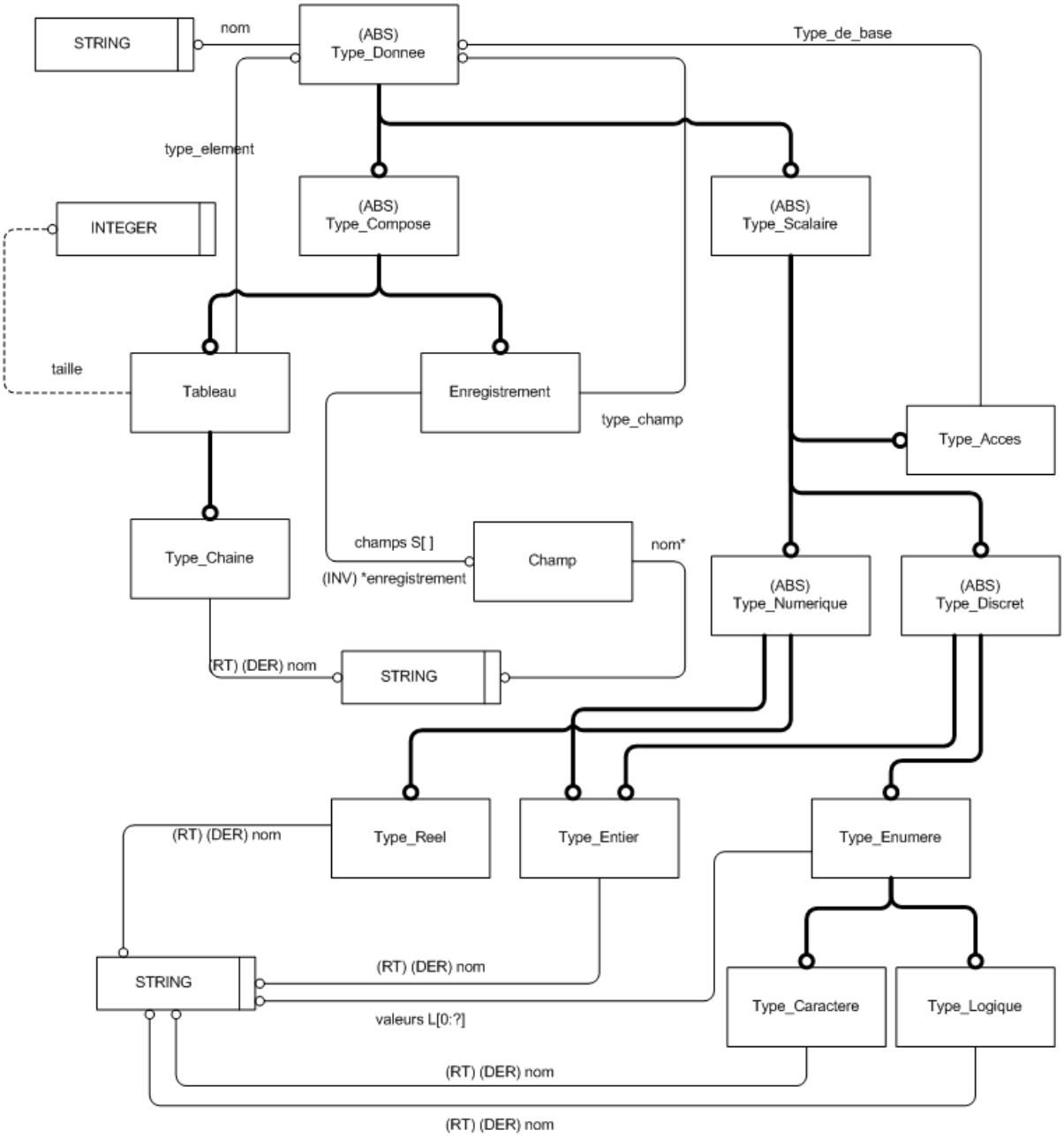
Exemple : `population SEMANTIQUE PRAGMATIQUE -value`

2 Modèles conceptuels de la programmation structurée en EXPRESS-G

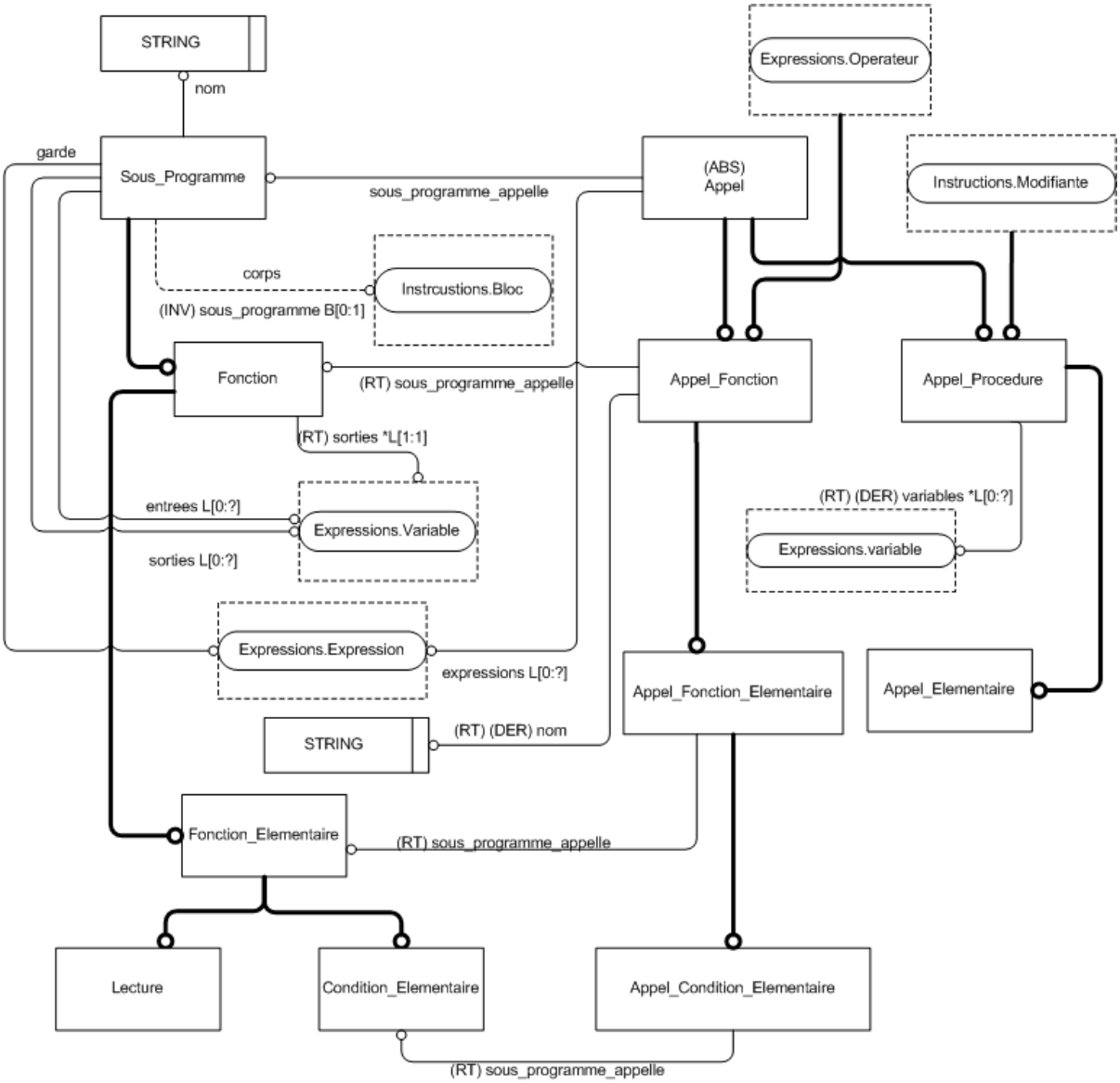
2.1 Modèle des Instructions.



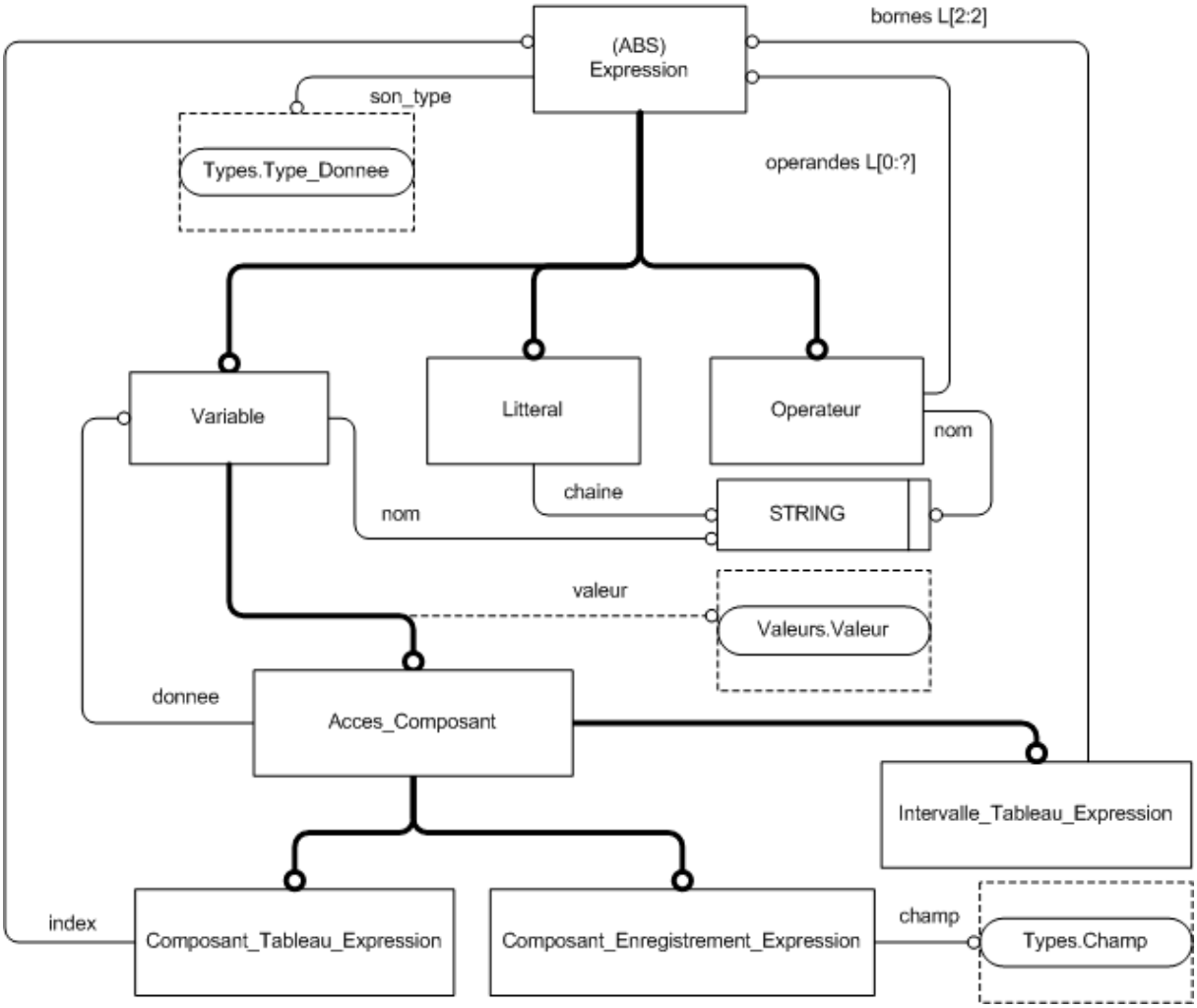
2.2 Modèle des types de données



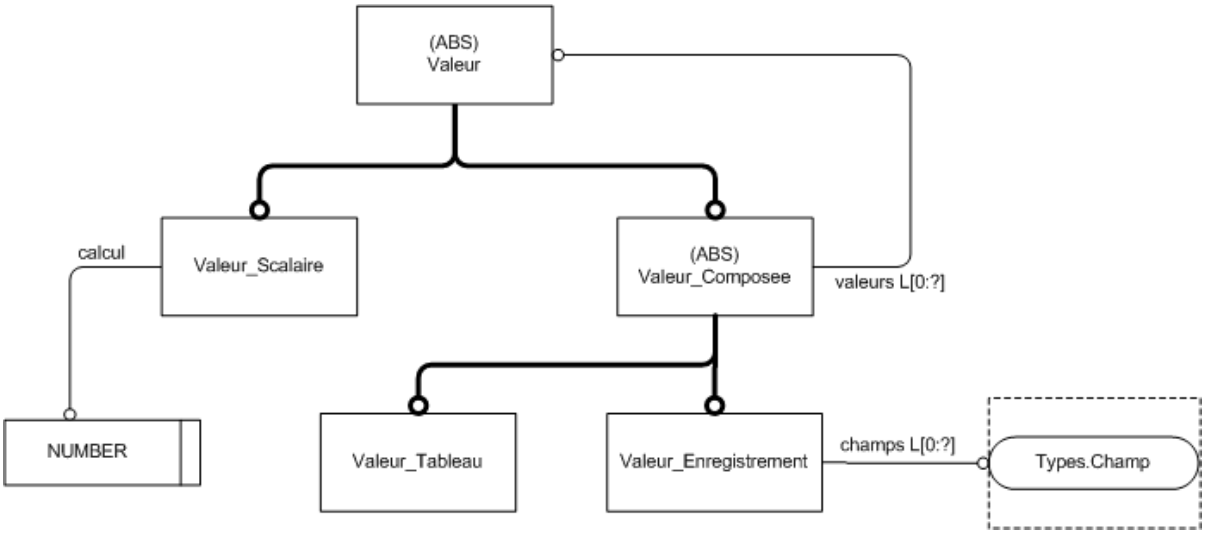
2.3 Modèle des programmes



2.4 Modèle des expressions



2.5 Modèle des valeurs



3 Modèles conceptuels de la programmation structurée en EXPRESS

```
(* modele EXPRESS_G : EUP_Express.VSD en visio
la génération est faite dans EUP_Express.exp mais sans WHERE RULE ni les opérations


*)


SCHEMA ISO13584_Expressions_Schema;


REFERENCE FROM Semantique;
REFERENCE FROM Memo_LIB;--pour get_oid
REFERENCE FROM String_LIB;
REFERENCE FROM Io_LIB;--pour print et print_nl


ENTITY Expression
ABSTRACT SUPERTYPE OF (ONEOF(variable, Litteral, Operateur));
son_type : Type_Donnee;
DERIVE
representation : STRING := '' ; -- calculé dans les sous-types
OPERATIONS
evalue: GENERIC ;
END_ENTITY;


OPERATION Expression.evalue: GENERIC ;
END_OPERATION;


ENTITY Litteral
SUBTYPE OF (Expression);
chaine : STRING;
DERIVE
SELF\Expression.representation : STRING := chaine;
OPERATIONS
SELF\Expression.evalue: GENERIC ;
END_ENTITY;


OPERATION Litteral.evalue: GENERIC ;
-- Désolé c'est pas élégant, faudrait le revoir ultérieurement !
LOCAL
-- String_val : STRING;
Int_val : INTEGER;
Float_val : REAL ;
vt : Valeur_Tableau ;
lvs : LIST [0:?] OF valeur_scalaire;
vs : valeur_scalaire;
END_LOCAL;


```

```
IF ( 'SEMANTIQUE.TYPE_LOGIQUE' IN TYPEOF (son_type) ) THEN
print_nl('Litteral Booleen de valeur :'+chaine);
IF ( chaine ='TRUE') THEN
RETURN(TRUE);
ELSE
RETURN(FALSE);
END_IF ;
ELSE
-- traiter le cas de Char avec BEGIN_C++ ... END_C++;
IF ('SEMANTIQUE.TYPE_ENUMERE' IN TYPEOF (son_type) ) THEN
-- renvoyer l'indice correspondant dans la liste de valeurs !!!
RETURN (son_type\Type_Enumere.trouver_valeur(chaine));
END_IF;
END_IF ;
IF ( 'SEMANTIQUE.TYPE_REEL' IN TYPEOF (son_type) ) THEN
from_string(chaine, Float_val);
RETURN(Float_val);
END_IF;
IF ( 'SEMANTIQUE.TYPE_ENTIER' IN TYPEOF (son_type) ) THEN
from_string(chaine,Int_val);
RETURN(Int_val);
END_IF;
IF ( 'SEMANTIQUE.TYPE_CHAINE' IN TYPEOF (son_type) ) THEN
-- creer une instance de valeur_tableau dont l'attribut valeurs contient
les oid
-- des valeurs de caracteres des chaine[i]
REPEAT i:= 1 TO LENGTH(chaine);

vs:=valeur_scalaire(POPULATION('SEMANTIQUE.TYPE_CARACTERE')[1].trouver_vale
ur(chaine[i]));
vs.create;
INSERT(lvs,vs,SIZEOF(lvs));
END_REPEAT;
vt:=valeur_composee(lvs)||valeur_tableau();
vt.create;
RETURN(get_oid(vt));
END_IF;

END_OPERATION;


ENTITY Operateur
SUBTYPE OF (Expression);
nom : STRING;
operandes : LIST [0 : ?] OF Expression;
DERIVE
SELF\Expression.representation : STRING := representation(SELF);
infixe : BOOLEAN := ((nom='+') OR (nom='-') OR (nom='x') OR (nom='/')
OR (nom='mod') OR (nom='rem')OR (nom='puiss')
OR (nom='=') OR (nom='<>') OR (nom='<=') OR (nom='>=') OR (nom='<') OR
(nom='>')
OR (nom='OR') OR (nom='AND'));
WHERE
infixe_que_si_binaire : (NOT infixe OR (SIZEOF(operandes)=2));
```

```
103 -- infixe => 2 operandes
104 coherence_types : coherence_type ( SELF ) ;
105 OPERATIONS
106 SELF\Expression.evalue: GENERIC ;
107 END_ENTITY;
108
109
110 OPERATION Operateur.evalue: GENERIC ;
111 -- opérateurs logiques
112 IF (nom='NOT') THEN
113 RETURN(NOT operandes[1].evalue);
114 END_IF ;
115 IF (nom='OR') THEN
116 RETURN(operandes[1].evalue OR operandes[2].evalue);
117 END_IF ;
118 IF (nom='AND') THEN
119 RETURN(operandes[1].evalue AND operandes[2].evalue);
120 END_IF ;
121 -- opérateurs numériques
122 IF (nom='+|-') THEN
123 RETURN (- operandes[1].evalue);
124 END_IF ;
125 IF (nom='+') THEN
126 RETURN (operandes[1].evalue + operandes[2].evalue);
127 END_IF ;
128 IF (nom='-') THEN
129 RETURN (operandes[1].evalue - operandes[2].evalue);
130 END_IF ;
131 IF (nom='x') THEN
132 RETURN (operandes[1].evalue * operandes[2].evalue);
133 END_IF;
134 IF (nom='/') THEN
135 RETURN (operandes[1].evalue / operandes[2].evalue);
136 END_IF ;
137 IF (nom='mod') THEN
138 RETURN (operandes[1].evalue MOD operandes[2].evalue);
139 END_IF ;
140 IF (nom='rem') THEN
141 RETURN (operandes[1].evalue DIV operandes[2].evalue);
142 END_IF ;
143 IF (nom='puiss') THEN
144 RETURN (operandes[1].evalue ** operandes[2].evalue);
145 END_IF ;
146 -- opérateurs d'égalité, d'inclusion, de comparaison
147 IF (nom='=') THEN
148 RETURN (operandes[1].evalue = operandes[2].evalue);
149 END_IF ;
150 IF (nom='<>') THEN
151 RETURN (operandes[1].evalue <> operandes[2].evalue);
152 END_IF ;
153 IF (nom='<=') THEN
154 RETURN (operandes[1].evalue <= operandes[2].evalue);
155 END_IF ;
```

```
156 IF (nom='>=') THEN
157 RETURN (operandes[1].evalue >= operandes[2].evalue);
158 END_IF ;
159 IF (nom='<') THEN
160 RETURN (operandes[1].evalue < operandes[2].evalue);
161 END_IF ;
162 IF (nom='>') THEN
163 RETURN (operandes[1].evalue > operandes[2].evalue);
164 END_IF ;
165 IF (nom='LENGTH') THEN
166 IF (chaine(operandes[1])) THEN
167 RETURN (LENGTH(operandes[1].evalue));
168 END_IF ;
169 IF (tableau(operandes[1])) THEN
170 RETURN (SIZEOF (operandes[1].valeur.valeurs));
171 END_IF ;
172 END_IF ;
173 END_OPERATION;
174
175
176 FUNCTION representation_operandes ( op :LIST [0:?] OF expression ):
STRING ;
177 LOCAL s : STRING := '' ;
178 END_LOCAL;
179 REPEAT i:=LOINDEX(op) TO HIINDEX(op); --FOREACH o IN op;
180 s := s + ',' + op[i].representation;
181 END_REPEAT;
182 IF (LENGTH(s)>1) THEN
183 RETURN (s[2:LENGTH(s)]);
184 ELSE
185 RETURN ('');
186 END_IF ;
187 END_FUNCTION;
188
189
190 FUNCTION representation( op :operateur ): STRING ;
191 print_nl('representation de'+get_oid(op)+'
infixe='+to_string(op.infixe));
192 IF (op.infixe) THEN
193 RETURN
('('+op.operandes[1].representation+')'+op.nom+'('+op.operandes[2].represen
tation+')') ;
194 ELSE
195 RETURN (op.nom+'('+representation_operandes(op.operandes)+')');
196 END_IF;
197 END_FUNCTION;
198
199
200 FUNCTION logique ( e : expression ) : BOOLEAN ;
201 RETURN( 'SEMANTIQUE.TYPE_LOGIQUE' IN TYPEOF (e.son_type) );
202 END_FUNCTION ;
203
204
205 FUNCTION numerique ( e : expression ) : BOOLEAN ;
```

```
206 RETURN( 'SEMANTIQUE.TYPE_NUMERIQUE' IN TYPEOF (e.son_type) );
207 END_FUNCTION ;
208
209
210 FUNCTION tableau ( e : expression ) : BOOLEAN ;
211 RETURN( 'SEMANTIQUE.TABLEAU' IN TYPEOF (e.son_type) );
212 END_FUNCTION ;
213
214
215 FUNCTION chaine ( e : expression ) : BOOLEAN ;
216 RETURN( 'SEMANTIQUE.TYPE_CHAINE' IN TYPEOF (e.son_type) );
217 END_FUNCTION ;
218
219
220 FUNCTION coherence_type ( o : operateur ) : BOOLEAN ;
221 IF (o.nom='NOT') THEN
222 RETURN( logique(o.operandes[1]) );
223 END_IF ;
224 IF (( o.nom='OR') OR (o.nom='AND') ) THEN
225 RETURN(logique(o.operandes[1]) AND logique(o.operandes[2]));
226 END_IF ;
227 -- opérateurs numériques
228 IF (o.nom='+|-') THEN
229 RETURN( numerique(o.operandes[1]) );
230 END_IF ;
231 IF ((o.nom='+') OR (o.nom='-') OR (o.nom='x') OR (o.nom='/') OR
(o.nom='mod') OR
(o.nom='rem')) THEN
232 RETURN( numerique(o.operandes[1]) AND numerique(o.operandes[2]));
233 END_IF ;
234 -- opérateurs d'égalité, de comparaison
235 IF ((o.nom='=') OR (o.nom='<>') OR (o.nom='<=') OR (o.nom='>=') OR
(o.nom='<')) THEN
236 RETURN (o.operandes[1].son_type = o.operandes[2].son_type);
237 END_IF ;
238 IF (o.nom='LENGTH') THEN
239 RETURN (tableau(o.operandes[1]));
240 END_IF ;
241 RETURN (QUERY (c <* POPULATION('SEMANTIQUE.CONDITION_ELEMENTAIRE') |
c.nom=o.nom ) <> [] ) ;
242 END_FUNCTION ;
243
244
245 END_SCHEMA;
246
247
248 SCHEMA Semantique;
249
250 REFERENCE FROM ISO13584_Expressions_Schema;
251 REFERENCE FROM Com_LIB;--pour send_message
252 REFERENCE FROM Memo_LIB;--pour get_oid
253 REFERENCE FROM Io_LIB;--pour print et print_nl
254 REFERENCE FROM String_LIB;
255
256
```

```
CONSTANT
mot_cle_operation : STRING := 'operation';
mot_cle_test: STRING := 'test';
mot_cle_oid: STRING := 'oid';
mot_cle_garde: STRING := 'garde';
mot_cle_parametre: STRING := 'parametre'; -- d'entrée
mot_cle_variable: STRING := 'variable'; -- paramètre de sortie
mot_cle_infini: STRING := 'infini'; -- pour détecter les boucles infinies
infini : INTEGER := 100 ;
END_CONSTANT ;


RULE itc_unique FOR ( type_caractere ) ;
WHERE
SIZEOF( POPULATION ( 'SEMANTIQUE.TYPE_CARACTERE' ) ) = 1 ;
END_RULE ;


RULE ite_unique FOR ( type_entier ) ;
WHERE
SIZEOF( POPULATION ( 'SEMANTIQUE.TYPE_ENTIER' ) ) = 1 ;
END_RULE ;


RULE itl_unique FOR ( type_logique ) ;
WHERE
SIZEOF( POPULATION ( 'SEMANTIQUE.TYPE_LOGIQUE' ) ) = 1 ;
END_RULE ;


RULE itr_unique FOR ( type_reel ) ;
WHERE
SIZEOF( POPULATION ( 'SEMANTIQUE.TYPE_REEL' ) ) = 1 ;
END_RULE ;


RULE itt_unique FOR ( type_chaine ) ;
WHERE
SIZEOF( POPULATION ( 'SEMANTIQUE.TYPE_CHAINE' ) ) = 1 ;
END_RULE ;


TYPE Simple_Ou_Condition = SELECT (Simple , Expression ) ; END_TYPE;



(* expressions d'accès à des composants de types structurés
Hérite de variable car peut être affecté. Effectivement, un composant EST UNE variable*)
ENTITY Acces_Composant
ABSTRACT SUPERTYPE OF (ONEOF(Composant_Enregistrement_Expression, Composant_Tableau_Expression, Intervalle_Tableau_Expression))
SUBTYPE OF (Variable);
```

```
variable : variable ; -- la mère directe du composant. Ex tab(I).toto à pour mère tab(I)
WHERE
variable_structuree: 'SEMANTIQUE.TYPE_COMPOSE' IN TYPEOF(variable.son_type);
END_ENTITY;


ENTITY Affectation
SUBTYPE OF (Simple);
expression : Expression;
(* La variable affectée *)
SELF\simple.variables : LIST [1 : 1] OF Variable;
DERIVE
SELF\Simple.representation : STRING := variables[1].representation + ' := ' +
expression.representation ;
WHERE
coherence_types: type_coherent (variables [1],expression) ;
OPERATIONS
SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai continue=faux
SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai continue=faux
END_ENTITY;
OPERATION Affectation.run(jusqu_a:instruction):BOOLEAN ;
print_nl('start run AFFECTATION ' + to_string(SELF));
-- print_nl(to_string(variables[1]) + ' (' + variables[1].nom +' de type ' + TYPEOF
(variables[1].son_type) +' ) <-- ' + to_string(expression));
-- dans un premier tps, occupons nous des types "simples" -- manque char
IF ('SEMANTIQUE.TYPE_LOGIQUE' IN TYPEOF (variables[1].son_type)) THEN
IF (expression.evalue) THEN
variables[1].valeur\Valeur_Scalaire.calcul:=1;
ELSE
variables[1].valeur\Valeur_Scalaire.calcul:=2;
END_IF;
ELSE
IF (('SEMANTIQUE.TYPE_REEL' IN TYPEOF (variables[1].son_type)) OR
('SEMANTIQUE.TYPE_ENTIER' IN TYPEOF (variables[1].son_type)) OR
('SEMANTIQUE.TYPE_ENUMERE' IN TYPEOF (variables[1].son_type)))
THEN
print_nl('Adresse Valeur: ' + to_string(variables[1].valeur));
print_nl('Valeur avant: ' + to_string(variables[1].valeur.calcul));
variables[1].valeur.calcul := expression.evalue;
print_nl('Valeur apres: ' + to_string(variables[1].valeur.calcul));
END_IF ;
END_IF;
IF ('SEMANTIQUE.TYPE_COMPOSE' IN TYPEOF (variables[1].son_type)) THEN
variables[1].valeur\Valeur_Composee.valeurs := expression.evalue;
END_IF;
-- nécessite que pour les types composés expression.evalue --> VALEUR_COMPOSEE
RETURN ( test_arret(jusqu_a) );
```

```
353 END_OPERATION;
354
355
356 OPERATION Affectation.run_debut(jusqu_a:Bloc):BOOLEAN;
357 RETURN ( run ( jusqu_a ) );
358 END_OPERATION;
359
360
361
362
363 ENTITY Alors
364 SUBTYPE OF (Alternative);
365 sinon : OPTIONAL Sinon;
366 OPERATIONS
367 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
368 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
369 END_ENTITY;
370
371
372 OPERATION Alors.run_debut(jusqu_a:Bloc):BOOLEAN;
373 IF (condition.evalue) THEN
374 -- print_nl('evalue condition du si : '+to_string(condition.evalue));
375 RETURN ( corps.run_debut(jusqu_a) ) ;
376 ELSE
377 -- print_nl('evalue condition du si : '+to_string(condition.evalue));
378 IF ( test_arret(jusqu_a) ) THEN -- pour rendre l'oid à l'appli
379 RETURN ( TRUE ) ;
380 ELSE
381 IF (EXISTS(sinon)) THEN
382 RETURN ( sinon.corps.run_debut(jusqu_a) ) ;
383 ELSE
384 RETURN ( FALSE ) ;
385 END_IF;
386 END_IF;
387 END_IF ;
388 END_OPERATION;
389
390
391 OPERATION Alors.run(jusqu_a:instruction):BOOLEAN;
392 IF (condition.evalue) THEN
393 RETURN ( corps.run(jusqu_a) ) ;
394 ELSE
395 IF ( test_arret(jusqu_a) ) THEN -- pour rendre l'oid à l'appli
396 RETURN ( TRUE ) ;
397 END_IF;
398
399 IF (EXISTS(sinon)) THEN
400 RETURN ( sinon.corps.run(jusqu_a) ) ;
401 ELSE
402 RETURN ( FALSE ) ;
403 END_IF;
404
```

```
405 END_IF ;
406 END_OPERATION;
407
408
409 ENTITY Alternative
410 ABSTRACT SUPERTYPE OF (ONEOF(Alors, Sinon))
411 -- Choix de modélisation un peu lourd : un attribut booléen aurait
suffit à distinguer alors
de sinon
412 SUBTYPE OF (Conditionnelle);
413 END_ENTITY;
414
415
416 (* Lors de l'appel d'un sous-programme, la liste des paramètres
d'entrée est
417 alimentée par la liste des expressions. Pour les sorties voir
Appel_Procedure *)
418 ENTITY Appel
419 ABSTRACT SUPERTYPE OF (ONEOF(Appel_Fonction, Appel_Procedure));
420 expressions : LIST [0 : ?] OF Expression;
421 sous_programme_appelle : Sous_Programme;
422 WHERE
423 entrees: SIZEOF(sous_programme_appelle.entrees)=SIZEOF(expressions);
424 coherence_types: types_coherents
(sous_programme_appelle.entrees,expressions) ;
425 END_ENTITY;
426
427
428 ENTITY Appel_Condition_Elementaire
429 SUBTYPE OF (Appel_Fonction_Elementaire);
430 SELF\Appel_Fonction_Elementaire.sous_programme_appelle :
Condition_Elementaire;
431 OPERATIONS
432 SELF\Appel_Fonction_Elementaire.evalue: BOOLEAN ;
433 END_ENTITY;
434
435
436 OPERATION Appel_Condition_Elementaire.evalue: BOOLEAN ;
437 LOCAL
438 bidon : STRING := send_message(mot_cle_test);
439 retour : STRING := send_message(nom);
440 END_LOCAL;
441 print_nl('sent message :'+mot_cle_test);
442 print_nl(' had for response : '+bidon);
443 print_nl('sent message :'+nom);
444 print_nl(' had for response : '+retour);
445 RETURN(retour='ok');
446 END_OPERATION;
447
448
449 ENTITY Appel_Elementaire
450 SUBTYPE OF (Appel_Procedure);
451 DERIVE
452 SELF\simple.variables : LIST [0:?] OF variable := [] ;
453 WHERE
```

```
454 pas_fonction: NOT ( 'SEMANTIQUE.FONCTION_ELEMENTAIRE' IN TYPEOF( SELF
));
455 OPERATIONS
456 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
457 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
458 SELF\Instruction.iss (VAR s : simple ; VAR trouvee : BOOLEAN ; VAR
arret : BOOLEAN ) ; --
instruction simple suivante
459 END_ENTITY;
460
461
462 OPERATION Appel_Elementaire.iss (VAR s : simple ; VAR trouvee : BOOLEAN
; VAR arret : BOOLEAN
) ;
463 IF (sous_programme_appelle.garde.evalue) THEN
464 SELF\Instruction.iss(s,trouvee,arret);
465 ELSE
466 arret := TRUE ; trouvee := FALSE ; s := SELF ;
467 END_IF;
468 END_OPERATION;
469
470
471 OPERATION Appel_Elementaire.run(jusqu_a:instruction):BOOLEAN;
472 LOCAL
473 bidon : STRING ;
474 valeur : STRING ;
475 END_LOCAL;
476 -- transmission des paramètres d'entrée
477 REPEAT FOREACH e IN expressions ;
478 valeur := to_string ( e.evalue ) ;
479 print_nl('dans run valeur de parametre ='+ valeur);
480 bidon := send_message(mot_cle_parametre);
481 bidon := send_message(valeur);
482 END_REPEAT ;
483 print_nl('dans run avant evalue garde
'+get_oid(SELF)+'='+sous_programme_appelle.nom);
484 IF (sous_programme_appelle.garde.evalue) THEN -- évaluation avec
paramètres
485 -- attention, l'évaluation de la garde ne doit pas dépiler les
paramètres
486 bidon := send_message(mot_cle_operation);
487 bidon := send_message(sous_programme_appelle.nom);
488 -- récupération des valeurs de retour
489 REPEAT FOREACH v IN variables ;
490 valeur := send_message(mot_cle_variable);
491 v.valeur := valeur ; -- l'instance est crée dans JAVA
492 END_REPEAT ;
493 RETURN ( test_arret(jusqu_a) );
494 ELSE
495 bidon := send_message(mot_cle_garde);
496 bidon := send_message(get_oid(SELF));
497 RETURN (TRUE);
498 END_IF ;
```

```
END_OPERATION;


OPERATION Appel_Elementaire.run_debut(jusqu_a:Bloc):BOOLEAN;
RETURN ( run ( jusqu_a ) );
END_OPERATION;


ENTITY Appel_Fonction
SUBTYPE OF (Appel, Operateur);
SELF\Appel.sous_programme_appelle : Fonction;
DERIVE
SELF\Operateur.nom : STRING := sous_programme_appelle.nom;
OPERATIONS
SELF\Operateur.evalue: GENERIC ;
END_ENTITY;


OPERATION Appel_Fonction.evalue: GENERIC ;
-- post_condition : retour de TYPE simple Express (boolean, integer,
real) ou instance de
VALEUR
-- compatible avec sous_progmme_appele.sorties[1].son_type
END_OPERATION;


ENTITY Appel_Fonction_Elementaire
SUBTYPE OF (Appel_Fonction);
SELF\Appel_Fonction.sous_programme_appelle : Fonction_Elementaire;
OPERATIONS
SELF\Appel_Fonction.evalue: GENERIC ;
END_ENTITY;


OPERATION Appel_Fonction_Elementaire.evalue: GENERIC ;
-- post_condition : retour de TYPE VALEUR d'attribut
-- dépend de la fonction elementaire ...
END_OPERATION;


ENTITY Appel_Procedure
SUBTYPE OF (Appel, simple);
DERIVE
SELF\Simple.representation : STRING := sous_programme_appelle.nom +
representation_parametres ( SELF ) ;
WHERE
sorties : SIZEOF ( variables ) = SIZEOF (
sous_programme_appelle.sorties ) ;
coherence_types: types_coherents (sous_programme_appelle.sorties,
variables) ;
END_ENTITY;


FUNCTION representation_parametres ( a : Appel_Procedure ) : STRING ;
LOCAL
```

```
549 ch : STRING :='';
550 END_LOCAL;
551 IF a.variables = [] THEN
552 RETURN ( '' ) ;
553 ELSE
554 REPEAT i:=LOINDEX(a.variables) TO HIINDEX(a.variables) ;
555 ch := ch
556 -- la ligne suivante ne sert qu'en notation associative
557 + a.sous_programme_appelle.sorties[i].representation + '=>'
558 + a.expressions[i].representation + ',' ;
559 END_REPEAT ;
560 RETURN ( '(' + ch[1:LENGTH (ch)-1] + ')' ) ; -- le '-1' pour virer la
dernière ,
561 END_IF ;
562 END_FUNCTION ;
563
564
565 ENTITY Attribut;
566 donnee : variable;
567 DERIVE
568 visible : BOOLEAN := 'SEMANTIQUE.VARIABLE' IN TYPEOF(donnee);
569 END_ENTITY;
570
571
572 ENTITY Bloc
573 SUBTYPE OF (Structuree);
574 variables : SET [0 : ?] OF Variable;
575 instructions : LIST [0 : ?] OF UNIQUE Instruction;
576 DERIVE
577 (* L'ensemble de toutes les variables vues = l'ensemble des variables
declarees dans les
blocs englobants.
578 Impossible à faire en :
579 QUERY v <* la population de Variable / v.declaration IN les_dans *)
580 contexte : SET [0 : ?] OF Variable := calcule_contexte ( SELF ); --
récursif sur
bloc[1].variables
581 racine : Bloc := racine ( SELF ) ; --
582 INVERSE
583 dans_conditionnelle : BAG [0 : 1] OF Conditionnelle FOR corps;
584 OPERATIONS
585 ajoute_debut(i:instruction);
586 ajoute(i:instruction;apres:instruction);
587 supprime(i:instruction):instruction;
588 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
589 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
590 ajoute_trace(i:instruction); -- ajoute i dans la trace du film si i de
type simple
591 -- nécessite qu'il n'y a qu'un test par tache...
592 SELF\Instruction.numero_de_ligne : INTEGER ;
593 END_ENTITY;
594
595 OPERATION Bloc.numero_de_ligne : INTEGER ;
```

```
596 -- assert numero <> 0
597 IF (numero = 1 ) THEN
598 RETURN ( dans[1].numero_de_ligne + 1 ) ;
599 ELSE
600 RETURN ( dans[1].instructions[numero-1].numero_de_ligne + 1 ) ;
601 END_IF ;
602 END_OPERATION;
603
604
605 FUNCTION racine ( s : Bloc ) : Bloc ;
606 IF ( s.les_dans=[] ) THEN RETURN ( s ) ;
607 ELSE RETURN ( s.les_dans[1] ) ;
608 END_IF ;
609 END_FUNCTION ;
610
611
612 OPERATION Bloc.ajoute_trace(i:instruction); -- ajoute i dans la trace
du film si i de type
simple
613 LOCAL
614 r : Bloc := racine( SELF ) ;
615 s : sous_programme := r.sous_programme[1] ;
616 END_LOCAL;
617 IF ( 'SEMANTIQUE.SIMPLE' IN TYPEOF(i) ) THEN
618 s.tests[s.test].film.ajoute_trace(i);
619 END_IF ;
620 END_OPERATION;
621
622
623 OPERATION Bloc.run_debut(jusqu_a:Bloc):BOOLEAN;
624 -- on s'arrete si on est sur l'instruction jusqu_a
625 IF (test_arret(jusqu_a)) THEN RETURN ( TRUE ) ; END_IF ;
626 REPEAT FOREACH i IN instructions ;
627 ajoute_trace(i);
628 IF (NOT ('SEMANTIQUE.SINON' IN TYPEOF(i))) THEN
629 -- il ne faut pas faire de run si i est un sinon, car le fait d'avoir
fait le alors
630 -- peut avoir changé la condition (on ferai ainsi le alors et le sinon
...)
631 -- et le run du alors fait le run du sinon si la condition est fausse
632 IF i.run_debut(jusqu_a) THEN
633 RETURN (TRUE) ;
634 END_IF;
635 END_IF ;
636 END_REPEAT ;
637 RETURN (FALSE) ;
638 END_OPERATION;
639
640
641 OPERATION Bloc.run(jusqu_a:instruction):BOOLEAN;
642 print_nl('run Bloc :'+ get_oid(SELF));
643 -- on s'arrete si on a exécuté l'instruction jusqu_a
644 REPEAT FOREACH i IN instructions ;
```

```
print_nl('dans la Bloc '+get_oid(SELF)+', instruction courante
='+get_oid(i));
ajoute_trace(i);
IF (NOT ('SEMANTIQUE.SINON' IN TYPEOF(i))) THEN
-- il ne faut pas faire de run si i est un sinon, car le fait d'avoir
fait le alors
-- peut avoir changé la condition (on ferai ainsi le alors et le sinon
...)
-- et le run du alors fait le run du sinon si la condition est fausse
IF i.run(jusqu_a) THEN
RETURN (TRUE) ;
END_IF;
END_IF ;
END_REPEAT ;
print_nl('fin de la Bloc '+get_oid(SELF));
-- ou si on est l'instruction jusqu_a
RETURN (test_arret(jusqu_a)) ;
END_OPERATION;


OPERATION Bloc.ajoute_debut(i:instruction);
POSTCONDITION (i IN instructions);
INSERT(instructions , i , 0);
END_OPERATION;


OPERATION Bloc.ajoute(i:instruction;apres:instruction);
LOCAL
ii : INTEGER := calcule_numero(apres);
END_LOCAL;
PRECONDITION (apres IN instructions);
POSTCONDITION (i IN instructions);
INSERT(instructions , i , ii);
print_nl('insertion de : '+get_oid(i)+' en position ='+to_string(ii)+'
apres :
'+get_oid(apres));
END_OPERATION;


OPERATION Bloc.supprime(i:instruction):instruction;
LOCAL
ii : INTEGER := calcule_numero(i);
derniere : BOOLEAN := (ii=HIINDEX(instructions));
END_LOCAL;
PRECONDITION (i IN instructions);
POSTCONDITION (NOT (i IN instructions));
REMOVE (instructions , ii );
IF ( (instructions=[]) ) THEN
IF (dans<>[]) THEN
RETURN (supprime(dans[1])); -- on supprime le père
ELSE
RETURN (SELF) ;
END_IF ;
ELSE
```

```
694 IF (derniere) THEN
695 RETURN (instructions[ii-1]);
696 ELSE
697 RETURN (instructions[ii]);
698 END_IF ;
699 END_IF ;
700 END_OPERATION;
701
702 ENTITY Champ;
703 nom : STRING;
704 type_champ : Type_Donnee;
705 INVERSE
706 enregistrement : Enregistrement FOR champs;
707 UNIQUE
708 enregistrement, nom;
709 END_ENTITY;
710
711
712 ENTITY Composant_Enregistrement_Expression
713 SUBTYPE OF (Acces_Composant);
714 champ : Champ;
715 DERIVE
716 SELF\variable.nom : STRING := variable.nom+'.'+champ.nom;
717 -- SELF\Donnee.valeur : Valeur := variable.valeur.valeurs[index(SELF)];
718 SELF\variable.valeur : Valeur :=
variable.valeur.valeurs[index(champ,variable.valeur.champs)];
719 SELF\Expression.son_type : Type_Donnee := champ.type_champ;
720 WHERE
721 type_enregistrement: 'SEMANTIQUE.ENREGISTREMENT' IN
TYPEOF(variable.son_type);
722 champ_ok: champ IN variable.son_type.champs;
723 END_ENTITY;
724
725
726 FUNCTION index ( valeur_cherchee : GENERIC ; dans : LIST [0:?] OF
GENERIC ) : INTEGER ;
727 REPEAT indice:=LOINDEX(dans) TO HIINDEX(dans) ;
728 IF dans[indice] = valeur_cherchee THEN
729 RETURN ( indice ) ;
730 END_IF ;
731 END_REPEAT ;
732 print_nl('dans index pas de return');
733 END_FUNCTION;
734
735
736 ENTITY Composant_Tableau_Expression
737 SUBTYPE OF (Acces_Composant);
738 index : Expression;
739 DERIVE
740 SELF\variable.nom : STRING :=
variable.nom+'('+index.representation+')';
741 SELF\variable.valeur : Valeur := variable.valeur.valeurs[index.evalue];
742 SELF\Expression.son_type : Type_Donnee :=
variable.son_type.type_element;
```

```
743 WHERE
744 type_tableau: 'SEMANTIQUE.TABLEAU' IN TYPEOF(variable.son_type);
745 END_ENTITY;
746
747
748
749
750 ENTITY Condition_Elementaire
751 SUBTYPE OF (Fonction_Elementaire);
752 WHERE
753 booleen:'SEMANTIQUE.TYPE_LOGIQUE' IN TYPEOF(sorties[1].son_type);
754 END_ENTITY;
755
756
757 ENTITY Conditionnelle
758 -- fait pour factoriser la condition et un corps au niveau de la
Conditionnelle
759 -- pas très naturel comme choix de modélisation ! (par rapport aux
présentations syntaxiques
usuels)
760 ABSTRACT SUPERTYPE OF (ONEOF(Alternative, Repetitive))
761 SUBTYPE OF (Structuree);
762 condition : Expression;
763 corps : Bloc;
764 END_ENTITY;
765
766
767 ENTITY Ecriture
768 SUBTYPE OF (Elementaire);
769 END_ENTITY;
770
771
772 (* Action pragmatique = tache élémentaire du domaine *)
773 ENTITY Elementaire
774 SUBTYPE OF (Tache);
775 END_ENTITY;
776
777
778 ENTITY Enregistrement
779 SUBTYPE OF (Type_Compose);
780 champs : SET [0 : ?] OF Champ;
781 END_ENTITY;
782
783
784 ENTITY Film;
785 test : Test;
786 trace : LIST [0 : ?] OF Simple_Ou_Condition;
787 etat : INTEGER ; -- trace[1..etat] a été exécuté
788 OPERATIONS
789 ajoute_trace ( s : Simple_Ou_Condition ) ; -- à la fin de trace et maj
etat
790 initialise_trace ; -- trace'=[] et etat'=0
791 pas_a_pas : BOOLEAN ; -- vrai=éxécution réalisée
792 -- exécute la prochaine instruction simple possible après la dernière
793 -- de la trace, et l'ajoute à la trace (si rend vrai)
```

```
794 -- et maj etat
795 avance_un ;
796 -- d'un élément de la trace
797 -- nécessite : etat < trace'last
798 avance ( e : INTEGER ) ;
799 -- va jusqu'à l'état e de la trace
800 -- nécessite : e dans trace'range
801 modifiee ( v : Variable ) : BOOLEAN ; -- teste si v est modifiee
802 END_ENTITY;
803
804
805 OPERATION Film.avance_un ;
806 IF ( etat <> SIZEOF(trace) ) THEN
807 etat := etat + 1 ;
808 trace[etat].run(trace[etat]);
809 END_IF ;
810 END_OPERATION;
811
812
813 OPERATION Film.avance ( e : INTEGER ) ;
814 PRECONDITION ( e <= SIZEOF(trace) );
815 etat:=0;
816 REPEAT i:=1 TO e ;
817 avance_un ;
818 END_REPEAT ;
819 END_OPERATION;
820
821
822 OPERATION Film.ajoute_trace ( s : Simple_Ou_Condition ) ;
823 INSERT(trace,s,SIZEOF(trace));
824 etat := SIZEOF(trace) ;
825 END_OPERATION;
826
827
828 OPERATION Film.modifiee ( v : Variable ) : BOOLEAN ;
829 LOCAL
830 e_i_m_v : BAG OF simple := USEDIN ( v ,
831 'SEMANTIQUE.SIMPLE.VARIABLES' ) ; -- ensemble
832 -- des instructions modifiant v
833 END_LOCAL ;
834 RETURN ( QUERY ( m <* e_i_m_v | m IN trace ) <> [] ) ;
835 -- v est valuee si il existe au moins une instruction modifiant v
836 -- dans la trace du film
837 END_OPERATION;
838
839
840 OPERATION Film.initialise_trace ;
841 trace := [] ;
842 etat := SIZEOF(trace) ;
843 END_OPERATION;
844
845
846 OPERATION Film.pas_a_pas : BOOLEAN ;
847 LOCAL
```

```
848 s : Simple ; -- l'instruction qu'il faut excécuter
849 trouvee : BOOLEAN := FALSE ; -- si elle existe
850 tmp : LIST [0 : 1] OF Instruction;
851 END_LOCAL;
852 IF (trace=[]) THEN
853 -- On se place à la première instruction simple du programme.
854 tmp:=test.tache.corps.instructions;
855 IF tmp = [] THEN
856 -- tentative de faire le "pas à pas" d'un programme vide... ;-p
857 RETURN (FALSE);
858 END_IF;
859 REPEAT WHILE ( NOT ('SEMANTIQUE.SIMPLE' IN TYPEOF (tmp[1]))) UNTIL (
tmp = [] );
860 IF ('SEMANTIQUE.Bloc' IN TYPEOF(tmp[1])) THEN
861 tmp := tmp[1]\Bloc.instructions;
862 END_IF;
863 -- Ici il faut prendre en compte, pour calculer l'iss,
864 -- que la condition peut être fausse...
865 IF 'SEMANTIQUE.REPETE_JUSQU_A' IN TYPEOF (tmp[1]) THEN
866 tmp := tmp[1]\Conditionnelle.corps.instructions;
867 -- dans le cas "REPEAT UNTIL", on fait obligatoirement le premier tour
de
boucle!!
868 ELSE
869 -- sinon, on ne rentre que si la condititon est vérifiée
870 IF (('SEMANTIQUE.TANT_QUE' IN TYPEOF(tmp[1])) OR
871 ('SEMANTIQUE.ALORS' IN TYPEOF(tmp[1]))) THEN
872 IF tmp[1]\Conditionnelle.condition.evalue() = TRUE THEN
873 tmp := tmp[1]\Conditionnelle.corps.instructions;
874 ELSE
875 tmp := tmp[1].suivante;
876 END_IF;
877
878 ELSE
879 IF 'SEMANTIQUE.SINON' IN TYPEOF(tmp[1]) THEN
880 IF ( NOT (tmp[1]\Conditionnelle.condition.evalue() =
TRUE)) THEN
881 tmp :=
tmp[1]\Conditionnelle.corps.instructions;
882 ELSE
883 tmp := tmp[1].suivante;
884 END_IF;
885 END_IF;
886 END_IF;
887 END_IF;
888 END_REPEAT;
889 IF tmp = [] THEN
890 -- tentative de runner un programme sans simple...
891 RETURN (FALSE);
892 ELSE
893 s := tmp[1];
894 END_IF;
895 ELSE
896 print_nl('dans Film.pas_a_pas s non initialise');
```

```
897 -- trace[SIZEOF(trace)].iss ( s , trouvee , arret ) ;
898 END_IF;
899 IF ( trouvee ) THEN
900 ajoute_trace ( s ) ;
901 RETURN ( s.run ( s ) ) ;
902 END_IF ;
903 etat := SIZEOF(trace) ;
904 END_OPERATION;
905
906
907 ENTITY Fonction
908 SUBTYPE OF (Sous_Programme);
909 SELF\Sous_Programme.sorties : LIST [1 : 1] OF UNIQUE variable;
910 WHERE
911 nom_sortie: sorties[1].nom=nom;
912 END_ENTITY;
913
914
915 ENTITY Fonction_Elementaire
916 SUBTYPE OF (Elementaire, Fonction);
917 END_ENTITY;
918
919
920 ENTITY Instruction
921 ABSTRACT SUPERTYPE OF (ONEOF(Simple, Structuree));
922 DERIVE
923 (* bloc immediatement englobant de l'instruction *)
924 bloc : SET [0 : 1] OF Bloc := calcule_bloc(SELF);
925 (* Calcul récursif des Blocs parentes dans l'ordre (jusqu'au sous-
programme)
926 de SIZEOF(les_dans) jusqu'à 1 *)
927 les_dans : LIST [0 : ?] OF Bloc := calcule_les_dans(SELF);
928 (* Calcul récursif des conditionnelles parentes dans l'ordre
929 de SIZEOF(les_dans_conditionnelle) jusqu'à 1 *)
930 les_dans_conditionnelle : LIST [0 : ?] OF conditionnelle :=
calcule_les_dans_conditionnelle(SELF);
931 (* numero de l'instrution dans la liste des instructions. 0 pour la
racine
932 et pour les Blocs dans les conditionnelles *)
933 numero : INTEGER := calcule_numero(SELF);
934 (* rend l'instruction suivante dans l'instruction composée *)
935 suivante : LIST [0 : 1] OF Instruction := calcule_suivante(SELF);
936 INVERSE
937 dans : SET [0 : 1] OF Bloc FOR instructions;
938 WHERE
939 (* dans vide => soit Structuree et sous-programme existe
940 soit Bloc et dans_conditionnelle existe*)
941 dans_quelquechose: (dans<>[]) OR ((('SEMANTIQUE.STRUCTUREE' IN
TYPEOF(SELF)) AND
(sous_programme<>[]))
942 OR
943 (('SEMANTIQUE.Bloc' IN TYPEOF(SELF)) AND
(dans_conditionnelle<>[])));
944 OPERATIONS
```

```
945 test_arret(jusqu_a:instruction):BOOLEAN; -- arret=vrai continue=faux
946 -- factorisation du test de l'instruction finale (et envoi de l'oid si
arrêt)
947
948
949 -- si je pouvais définir les run en ABSTRACT ce serait le pied ...
950 run(jusqu_a:instruction):BOOLEAN; -- arret=vrai continue=faux
951 -- réalise le run récursif de SELF jusqu'à:
952 -- atteindre l'instruction jusqu_a (comprise) ou
953 -- s'arrêter sur une conditionnelle fausse contenant jusqu_a
954 run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai continue=faux
955 -- s'arrête sur l'instruction composée
956 iss (VAR s : simple ; VAR trouvee : BOOLEAN ; VAR arret : BOOLEAN );
957 -- calcule l'instruction simple suivante de l'instruction courante.
958 numero_de_ligne : INTEGER ;
959 -- calcule le numéro de ligne de l'instruction "début" par rapport au
début du programme
960 -- simple : instruction (supposée sur une ligne)
961 -- Bloc : début
962 -- {instructions, 1 par ligne}
963 -- fin
964 -- bloc : declare
965 -- {variables, 1 par ligne}
966 -- Bloc
967 -- alternative : si condition
968 -- Bloc
969 -- [ Bloc ]
970 -- repete_jusau_a : repete
971 -- Bloc
972 -- jusqu'à condition
973 -- fin
974 -- tant_que : repete
975 -- tant que condition
976 -- Bloc
977 -- fin
978
979 END_ENTITY;
980
981
982 OPERATION Instruction.run(jusqu_a:instruction):BOOLEAN;
983 -- IS ABSTRACT
984 END_OPERATION;
985
986
987 OPERATION Instruction.run_debut(jusqu_a:Bloc):BOOLEAN;
988 -- IS ABSTRACT
989 END_OPERATION;
990
991
992 OPERATION Instruction.test_arret(jusqu_a:instruction):BOOLEAN;
993 LOCAL
994 bidon : STRING ;
995 END_LOCAL;
996 IF ( SELF:=:jusqu_a ) THEN
```

```
997 bidon := send_message(mot_cle_oid);
998 bidon := send_message(get_oid(SELF)) ; -- on envoie la chaine oid de
l'instruction
999 END_IF ;
1000 RETURN ( SELF:=:jusqu_a );
1001 END_OPERATION;
1002
1003
1004 OPERATION Instruction.iss(VAR s : simple ; VAR trouvee : BOOLEAN ; VAR
arret : BOOLEAN ) ;
1005 LOCAL
1006 tmp : LIST [0 : 1] OF Instruction;
1007 END_LOCAL;
1008 tmp:=calcule_suivante(SELF);
1009
1010
1011 IF tmp <> [] THEN
1012 -- à partir de là, tmp[1] peut être Simple, Conditionnelle, ou
Séquence.
1013 REPEAT WHILE ( NOT ('SEMANTIQUE.SIMPLE' IN TYPEOF(tmp[1]))) UNTIL (
tmp = [] );
1014 IF ('SEMANTIQUE.Bloc' IN TYPEOF(tmp[1])) THEN
1015 tmp := tmp[1]\Bloc.instructions;
1016 END_IF;
1017 -- Ici il faut prendre en compte, pour calculer l'iss,
1018 -- que la condition peut être fausse...
1019 IF 'SEMANTIQUE.REPETE_JUSQU_A' IN TYPEOF(tmp[1]) THEN
1020 tmp := tmp[1]\Conditionnelle.corps.instructions;
1021 -- dans le cas "REPEAT UNTIL", on fait obligatoirement le premier tour
de
boucle!!
1022 ELSE
1023 -- sinon, on ne rentre que si la conditition est vérifiée
1024 IF (('SEMANTIQUE.TANT_QUE' IN TYPEOF(tmp[1])) OR
1025 ('SEMANTIQUE.ALORS' IN TYPEOF(tmp[1]))) THEN
1026 IF tmp[1]\Conditionnelle.condition.evalue() = TRUE THEN
1027 tmp := tmp[1]\Conditionnelle.corps.instructions;
1028 ELSE
1029 tmp := tmp[1].suivante;
1030 END_IF;
1031
1032 ELSE
1033 IF 'SEMANTIQUE.SINON' IN TYPEOF(tmp[1]) THEN
1034 IF ( NOT (tmp[1]\Conditionnelle.condition.evalue() =
TRUE)) THEN
1035 tmp :=
tmp[1]\Conditionnelle.corps.instructions;
1036 ELSE
1037 tmp := tmp[1].suivante;
1038 END_IF;
1039 END_IF;
1040 END_IF;
1041 END_IF;
1042 END_REPEAT;
1043 -- ici, tmp c'est soit [] soit [Simple].
```

```
1044 IF (tmp =[]) THEN
1045 -- si on est en fin de séquence, on cherche l'iss de père...
1046 IF SELF.dans[1].dans_conditionnelle <> [] THEN
1047 SELF.dans[1].dans_conditionnelle[1].iss(s,trouvee,arret);
1048 ELSE
1049 IF (SELF.dans[1].dans <> []) THEN
1050 SELF.dans[1].iss(s,trouvee,arret);
1051 ELSE
1052 -- on est arrivé à la fin du prog, sans trouver ...
1053 arret := TRUE ; trouvee := FALSE ; s:=SELF;
1054 END_IF;
1055 END_IF;
1056 ELSE
1057 -- on a trouvé !!
1058 arret := TRUE ; trouvee := TRUE ; s := tmp[1];
1059 END_IF;
1060 ELSE
1061 -- si on est en fin de séquence, on cherche la suivante de père...
1062 IF SELF.dans[1].dans_conditionnelle <> [] THEN
1063 SELF.dans[1].dans_conditionnelle[1].iss(s,trouvee,arret);
1064 ELSE
1065 IF (SELF.dans[1].dans <> []) THEN
1066 SELF.dans[1].iss(s,trouvee,arret);
1067 ELSE
1068 arret := TRUE ; trouvee := FALSE ; s:=SELF;
1069 END_IF;
1070 END_IF;
1071 END_IF;
1072 END_OPERATION;
1073
1074
1075 OPERATION Instruction.numero_de_ligne : INTEGER ;
1076 -- is abstract
1077 END_OPERATION;
1078
1079
1080 ENTITY Intervalle_Tableau_Expression
1081 SUBTYPE OF (Acces_Composant);
1082 bornes : LIST [2 : 2] OF Expression;
1083 DERIVE
1084 SELF\variable.nom : STRING :=
variable.nom+'('+bornes[1].representation+'..'+bornes[2].representation+')'
;
1085 END_ENTITY;
1086
1087
1088 ENTITY Lecture
1089 SUBTYPE OF (Elementaire);
1090 END_ENTITY;
1091
1092
1093 (* Définition du domaine d'application *)
1094 ENTITY Pragmatique;
1095 actions : SET [0 : ?] OF Elementaire;
```

```
attributs : SET [0 : ?] OF Attribut;
nom : STRING;
OPERATIONS
elementaires:SET [0 : ?] OF elementaire;--qui ne sont pas
fonction_elementaire
fonctions_elementaires:SET [0 : ?] OF fonction_elementaire;
conditions_elementaires:SET [0 : ?] OF condition_elementaire;
elementaire(nom:STRING):elementaire;--qui peut être une
fonction_elementaire
END_ENTITY;


OPERATION Pragmatique.elementaire(nom:STRING):elementaire;
print_nl('recherche elementaire : '+nom);
RETURN (QUERY(a<*actions|a.nom = nom )[1]);
END_OPERATION;


OPERATION Pragmatique.elementaires:SET [0 : ?] OF elementaire;
RETURN (actions-SELF.fonctions_elementaires);
END_OPERATION;


OPERATION Pragmatique.fonctions_elementaires:SET [0 : ?] OF
fonction_elementaire;
RETURN (QUERY(a<*actions|'SEMANTIQUE.FONCTION_ELEMENTAIRE' IN
TYPEOF(a)));
END_OPERATION;


OPERATION Pragmatique.conditions_elementaires:SET [0 : ?] OF
condition_elementaire;
RETURN (QUERY(a<*actions|'SEMANTIQUE.CONDITION_ELEMENTAIRE' IN
TYPEOF(a)));
END_OPERATION;


ENTITY Programme
SUBTYPE OF (Sous_Programme);
WHERE
pas_d_entrees_sorties: (entrees=[]) AND (sorties=[]);


END_ENTITY;


ENTITY Repete_Jusqu_A
SUBTYPE OF (Repetitive);
OPERATIONS
SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
END_ENTITY;

```

```
1143
1144 OPERATION Repete_Jusqu_A.run_debut(jusqu_a:Bloc):BOOLEAN;
1145 LOCAL j : INTEGER; END_LOCAL;
1146 IF (test_arret(jusqu_a)) THEN
1147 RETURN (TRUE);
1148 END_IF;
1149 REPEAT i:=1 TO infini UNTIL (condition.evalue) ;
1150 j := i ;
1151 IF (corps.run_debut(jusqu_a)) THEN
1152 RETURN ( TRUE ) ;
1153 END_IF ;
1154 print_nl('run_debut du Jusqu a, condition vaut :
'+to_string(condition.evalue));
1155 END_REPEAT ;
1156 RETURN (boucle_infini(j) OR test_arret(jusqu_a)); -- j'espère
l'évaluation paresseuse
1157 END_OPERATION;
1158
1159
1160 OPERATION Repete_Jusqu_A.run(jusqu_a:instruction):BOOLEAN;
1161 LOCAL j : INTEGER; END_LOCAL;
1162 REPEAT i:=1 TO infini UNTIL (condition.evalue) ;
1163 j := i ;
1164 IF (corps.run(jusqu_a)) THEN
1165 RETURN ( TRUE ) ;
1166 END_IF ;
1167 print_nl('run du Jusqu a, condition vaut :
'+to_string(condition.evalue));
1168 END_REPEAT ;
1169 RETURN (boucle_infini(j) OR test_arret(jusqu_a));
1170 END_OPERATION;
1171
1172
1173
1174 ENTITY Repetitive
1175 ABSTRACT SUPERTYPE OF (ONEOF(Repete_Jusqu_A, Tant_Que))
1176 SUBTYPE OF (Conditionnelle);
1177 OPERATIONS
1178 boucle_infini ( i : INTEGER ) : BOOLEAN ;
1179 SELF\Instruction.iss(VAR s : simple ; VAR trouvee : BOOLEAN ; VAR
arret : BOOLEAN);
1180 -- calcule l'instruction simple suivante,
1181 -- soit la première instruction de la boucle
1182 -- soit la suivante de la boucle dans le bloc pere
1183 -- selon l'évaluation de la condition.
1184 END_ENTITY;
1185
1186
1187 OPERATION Repetitive.boucle_infini ( i : INTEGER ):BOOLEAN;
1188 LOCAL
1189 bidon : STRING ;
1190 END_LOCAL;
1191 IF ( i>=infini ) THEN
1192 bidon := send_message(mot_cle_infini);
```

```
1193 RETURN (TRUE);
1194 END_IF ;
1195 RETURN ( FALSE );
1196 END_OPERATION;
1197
1198
1199 OPERATION Repetitive.iss(VAR s : simple ; VAR trouvee : BOOLEAN ; VAR
arret : BOOLEAN);
1200 LOCAL
1201 tmp : LIST [0 : 1] OF Instruction;
1202 END_LOCAL;
1203 IF condition.evalue = TRUE THEN
1204 IF corps.instructions <> [] THEN
1205 tmp:=[corps.instructions[1]];
1206 ELSE
1207 tmp:=[];
1208 END_IF;
1209 ELSE
1210 tmp:=calcule_suivante(SELF);
1211 END_IF;
1212
1213
1214 IF tmp <> [] THEN
1215 -- à partir de là, tmp[1] peut être Simple, Conditionnelle, ou
Séquence.
1216 REPEAT WHILE ( NOT ('SEMANTIQUE.SIMPLE' IN TYPEOF (tmp[1]))) UNTIL (
tmp = [] );
1217 IF ('SEMANTIQUE.Bloc' IN TYPEOF(tmp[1])) THEN
1218 tmp := tmp[1]\Bloc.instructions;
1219 END_IF;
1220 -- Ici il faut prendre en compte, pour calculer l'iss,
1221 -- que la condition peut être fausse...
1222 IF 'SEMANTIQUE.REPETE_JUSQU_A' IN TYPEOF (tmp[1]) THEN
1223 tmp := tmp[1]\Conditionnelle.corps.instructions;
1224 -- dans le cas "REPEAT UNTIL", on fait obligatoirement le premier tour
de
boucle!!
1225 ELSE
1226 -- sinon, on ne rentre que si la conditition est vérifiée
1227 IF (('SEMANTIQUE.TANT_QUE' IN TYPEOF(tmp[1])) OR
1228 ('SEMANTIQUE.ALORS' IN TYPEOF(tmp[1]))) THEN
1229 IF tmp[1]\Conditionnelle.condition.evalue() = TRUE THEN
1230 tmp := tmp[1]\Conditionnelle.corps.instructions;
1231 ELSE
1232 tmp := tmp[1].suivante;
1233 END_IF;
1234
1235 ELSE
1236 IF 'SEMANTIQUE.SINON' IN TYPEOF(tmp[1]) THEN
1237 IF ( NOT (tmp[1]\Conditionnelle.condition.evalue() =
TRUE)) THEN
1238 tmp :=
tmp[1]\Conditionnelle.corps.instructions;
1239 ELSE
```

```
1240 tmp := tmp[1].suivante;
1241 END_IF;
1242 END_IF;
1243 END_IF;
1244 END_IF;
1245 END_REPEAT;
1246 -- ici, tmp c'est soit [] soit [Simple].
1247 IF (tmp =[]) THEN
1248 -- si on est en fin de séquence, on cherche l'iss de père...
1249 IF SELF.dans[1].dans_conditionnelle <> [] THEN
1250 SELF.dans[1].dans_conditionnelle[1].iss(s,trouvee,arret);
1251 ELSE
1252 IF (SELF.dans[1].dans <> []) THEN
1253 SELF.dans[1].iss(s,trouvee,arret);
1254 ELSE
1255 -- on est arrivé à la fin du prog, sans trouver ...
1256 arret := TRUE ; trouvee := FALSE ; s:=SELF;
1257 END_IF;
1258 END_IF;
1259 ELSE
1260 -- on a trouvé !!
1261 arret := TRUE ; trouvee := TRUE ; s := tmp[1];
1262 END_IF;
1263 ELSE
1264 -- si on est en fin de séquence, on cherche la suivante de père...
1265 IF SELF.dans[1].dans_conditionnelle <> [] THEN
1266 SELF.dans[1].dans_conditionnelle[1].iss(s,trouvee,arret);
1267 ELSE
1268 IF (SELF.dans[1].dans <> []) THEN
1269 SELF.dans[1].iss(s,trouvee,arret);
1270 ELSE
1271 arret := TRUE ; trouvee := FALSE ; s:=SELF;
1272 END_IF;
1273 END_IF;
1274 END_IF;
1275
1276
1277 END_OPERATION;
1278
1279
1280 ENTITY Retour
1281 SUBTYPE OF (Simple);
1282 expression : OPTIONAL Expression;
1283 DERIVE
1284 SELF\Simple.representation : STRING := 'RETURN (' +
expression.representation + ')' ;
1285 WHERE
1286 Fonction_avec_expression : ( ( 'SEMANTIQUE.FONCTION' IN TYPEOF
(Les_Dans[1].Sous_Programme)
) =
1287 EXISTS ( expression ) ) ;
1288 OPERATIONS
1289 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
```

```
1290 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
1291 END_ENTITY;
1292
1293
1294 OPERATION Retour.run_debut(jusqu_a:Bloc):BOOLEAN;
1295 IF (test_arret(jusqu_a)) THEN
1296 RETURN (TRUE);
1297 END_IF;
1298 RETURN (TRUE);
1299 END_OPERATION;
1300
1301
1302 OPERATION Retour.run(jusqu_a:instruction):BOOLEAN;
1303 IF (test_arret(jusqu_a)) THEN
1304 RETURN (TRUE);
1305 END_IF;
1306 RETURN ( TRUE ) ;
1307 END_OPERATION;
1308
1309 ENTITY Simple
1310 ABSTRACT SUPERTYPE OF (ONEOF(Affectation, Appel_Procedure, Retour))
1311 SUBTYPE OF (Instruction);
1312 variables : LIST [0 : ?] OF UNIQUE Variable;
1313 DERIVE
1314 modifiante : BOOLEAN := variables<>[];
1315 representation : STRING := '' ; -- calculé dans les sous-types
1316 WHERE
1317 dans_quelque_chose : dans <> [] ;
1318 OPERATIONS
1319 SELF\Instruction.numero_de_ligne : INTEGER ;
1320 END_ENTITY;
1321
1322
1323 OPERATION Simple.numero_de_ligne : INTEGER ;
1324 -- assert numero <> 0
1325 IF (numero = 1 ) THEN
1326 RETURN ( dans[1].numero_de_ligne + 1 ) ;
1327 ELSE
1328 RETURN ( dans[1].instructions[numero-1].numero_de_ligne + 1 ) ;
1329 END_IF ;
1330 END_OPERATION;
1331
1332
1333 ENTITY Sinon
1334 SUBTYPE OF (Alternative);
1335 INVERSE
1336 alors : Alors FOR sinon;
1337 WHERE
1338 (* instruction immediatement suivant le alors*)
1339 suit_le_alors: (calcule_numero(SELF)=1+calcule_numero(alors));
1340 (* meme condition*)
1341 est_alors: (condition=alors.condition);
1342 OPERATIONS
```

```
1343 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
1344 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
1345 END_ENTITY;
1346
1347
1348 OPERATION Sinon.run_debut(jusqu_a:Bloc):BOOLEAN;
1349 IF (NOT condition.evalue) THEN
1350 RETURN ( corps.run_debut(jusqu_a) ) ;
1351 ELSE
1352 RETURN ( test_arret(jusqu_a) ) ;
1353 (* IF ( test_arret(jusqu_a) ) THEN -- pour rendre l'oid à l'appli
1354 RETURN ( TRUE ) ;
1355 END_IF;
1356 IF ( corps IN jusqu_a.les_dans ) THEN -- si jusqu_a est dans le corps
du sinon
1357 RETURN ( test_arret(SELF) ) ; -- on s'arrête ici
1358 ELSE
1359 RETURN ( FALSE ) ;
1360 END_IF; *)
1361 END_IF ;
1362 END_OPERATION;
1363
1364
1365 OPERATION Sinon.run(jusqu_a:instruction):BOOLEAN;
1366 IF (NOT condition.evalue) THEN
1367 RETURN ( corps.run(jusqu_a) ) ;
1368 ELSE
1369 IF ( test_arret(jusqu_a) ) THEN -- pour rendre l'oid à l'appli
1370 RETURN ( TRUE ) ;
1371 ELSE
1372 RETURN ( FALSE ) ;
1373 END_IF;
1374 END_IF ;
1375 END_OPERATION;
1376
1377
1378 ENTITY Sous_Programme;
1379 corps : OPTIONAL Structuree; -- peut être aussi bien une
conditionnelle qu'une séquence ou
un bloc
1380 entrees : LIST [0 : ?] OF variable;
1381 garde : Expression;
1382 nom : STRING;
1383 sorties : LIST [0 : ?] OF variable;
1384 WHERE
1385 (* nom de parametre unique*)
1386 nom_unique: QUERY(p1<* entrees+sorties|QUERY(p2<*
entrees+sorties|(p1<>p2) AND
(p1.nom=p2.nom))<>[])=[];
1387 END_ENTITY;
1388
1389
1390 ENTITY Structuree
```

```
1391 ABSTRACT SUPERTYPE OF (ONEOF(Conditionnelle, Bloc))
1392 SUBTYPE OF (Instruction);
1393 INVERSE
1394 sous_programme : BAG [0 : 1] OF Sous_Programme FOR corps;
1395 END_ENTITY;
1396
1397
1398 ENTITY Tableau
1399 SUBTYPE OF (Type_Compose);
1400 (* not(exists(taille)) <=> taille dynamique (de 0 à n) *)
1401 taille : OPTIONAL INTEGER;
1402 type_element : Type_Donnee;
1403 END_ENTITY;
1404
1405
1406 (* Modélise la tache en cours et les taches élémentaires pragmatiques.
1407
1408
1409 Si c'est une tache élémentaire, il n'y a pas de corps associé.
1410
1411
1412 sinon :
1413 t.tests[i].film.etats[j].pointeur_instruction donne l'instruction en
cours
1414 du ième test sur le jème état *)
1415 ENTITY Tache
1416 SUBTYPE OF (Sous_Programme);
1417 tests : LIST [0:?] OF Test;
1418 test : OPTIONAL INTEGER ; -- le numéro du test en cours
1419 WHERE -- rappel de logique : a=>b <=> non a ou b
1420 -- pas de corps => tests=[] et pas de test
1421 elementaire: EXISTS(corps) OR ((tests=[]) AND NOT EXISTS(test));
1422 -- corps => tests#[] et test dans bounds(tests)
1423 test_coherent: NOT EXISTS(corps) OR ( test <= SIZEOF(tests) ) ;
1424 END_ENTITY;
1425
1426
1427 ENTITY Tant_Que
1428 SUBTYPE OF (Repetitive);
1429 OPERATIONS
1430 SELF\Instruction.run_debut(jusqu_a:Bloc):BOOLEAN; -- arret=vrai
continue=faux
1431 SELF\Instruction.run(jusqu_a:instruction):BOOLEAN; -- arret=vrai
continue=faux
1432 END_ENTITY;
1433
1434
1435 OPERATION Tant_Que.run_debut(jusqu_a:Bloc):BOOLEAN;
1436 LOCAL j : INTEGER; END_LOCAL;
1437 IF ( test_arret(jusqu_a) ) THEN
1438 RETURN ( TRUE ) ;
1439 END_IF ;
1440 REPEAT i:=1 TO infini WHILE (condition.evalue) ;
1441 j := i ;
```

```
1442 IF (corps.run_debut(jusqu_a)) THEN
1443 RETURN ( TRUE ) ;
1444 END_IF ;
1445 END_REPEAT ;
1446 RETURN (boucle_infini(j) OR test_arret(jusqu_a));
1447 END_OPERATION;
1448
1449
1450 OPERATION Tant_Que.run(jusqu_a:instruction):BOOLEAN;
1451 LOCAL j : INTEGER; END_LOCAL;
1452 REPEAT i:=1 TO infini WHILE (condition.evalue) ; -- attention, boucle
infinie possible
1453 j := i ;
1454 IF (corps.run(jusqu_a)) THEN
1455 RETURN ( TRUE ) ;
1456 END_IF ;
1457 END_REPEAT ;
1458 RETURN (boucle_infini(j) OR test_arret(jusqu_a));
1459 END_OPERATION;
1460
1461
1462
1463 ENTITY Test;
1464 etat_initial : SET [0 : ?] OF Attribut;
1465 lectures : LIST [0 : ?] OF Valeur;
1466 INVERSE
1467 film : Film FOR test;
1468 tache : Tache FOR tests;
1469 END_ENTITY;
1470
1471
1472 ENTITY Type_Caractere
1473 SUBTYPE OF (Type_Enumere);
1474 DERIVE
1475 SELF\Type_Donnee.nom : STRING := 'CHARACTER';
1476 END_ENTITY;
1477
1478
1479 ENTITY Type_Chaine
1480 SUBTYPE OF (Tableau);
1481 DERIVE
1482 SELF\Type_Donnee.nom : STRING := 'STRING';
1483 WHERE
1484 nom_chaine: nom='STRING';
1485tableau_de_caracteres:'SEMANTIQUE.TYPE_CARACTERE' IN
TYPEOF(type_element);
1486 END_ENTITY;
1487
1488
1489 ENTITY Type_Compose
1490 ABSTRACT SUPERTYPE OF (ONEOF(Enregistrement, Tableau))
1491 SUBTYPE OF (Type_Donnee);
1492 END_ENTITY;
1493
```

```
1494
1495 ENTITY Type_Discret ABSTRACT SUPERTYPE
1496 SUBTYPE OF (Type_Scalaire);
1497 END_ENTITY;
1498
1499
1500 ENTITY Type_Donnee
1501 ABSTRACT SUPERTYPE OF (ONEOF(Type_Compose, Type_Scalaire));
1502 nom : STRING;
1503 END_ENTITY;
1504
1505
1506 ENTITY Type_Entier
1507 SUBTYPE OF (Type_Discret, Type_Numerique);
1508 DERIVE
1509 SELF\Type_Donnee.nom : STRING := 'INTEGER';
1510 END_ENTITY;
1511
1512
1513 ENTITY Type_Enumere
1514 SUBTYPE OF (Type_Discret);
1515 valeurs : LIST [0 : ?] OF STRING;
1516 OPERATIONS
1517 trouver_valeur ( val : STRING ) : INTEGER;
1518 -- renseigne l'indice de la valeur val, 0 = pas trouvé
1519 END_ENTITY;
1520
1521
1522 OPERATION Type_Enumere.trouver_valeur( val : STRING ) : INTEGER;
1523 IF valeurs <> [] THEN
1524 REPEAT i := LOINDEX(valeurs) TO HIINDEX(valeurs) ;
1525 IF val=valeurs[i] THEN -- attention : = de deux String ?
1526 --print_nl('Type_Enumere.trouver_valeur i='+to_string(i));
1527 RETURN (i);
1528 END_IF ;
1529 END_REPEAT ;
1530 END_IF ;
1531 --print_nl('Type_Enumere.trouver_valeur val='+val+'
valeurs[low]='+valeurs[LOINDEX(valeurs)]);
1532 RETURN (0);
1533 END_OPERATION;
1534
1535
1536
1537
1538 ENTITY Type_Logique
1539 SUBTYPE OF (Type_Enumere);
1540 DERIVE
1541 SELF\Type_Donnee.nom : STRING := 'BOOLEAN';
1542 END_ENTITY;
1543
1544
1545 ENTITY Type_Numerique ABSTRACT SUPERTYPE
1546 SUBTYPE OF (Type_Scalaire);
```

```
1547 END_ENTITY;
1548
1549
1550 ENTITY Type_Reel
1551 SUBTYPE OF (Type_Numerique);
1552 DERIVE
1553 SELF\Type_Donnee.nom : STRING := 'FLOAT';
1554 END_ENTITY;
1555
1556
1557 ENTITY Type_Scalaire ABSTRACT SUPERTYPE OF (ONEOF(Type_Numerique,
Type_Discret))
1558 SUBTYPE OF (Type_Donnee);
1559 END_ENTITY;
1560
1561
1562 ENTITY Valeur
1563 ABSTRACT SUPERTYPE OF (ONEOF(Valeur_Composee, Valeur_Scalaire));
1564 INVERSE
1565 test : BAG [0 : 1] OF Test FOR lectures;
1566 END_ENTITY;
1567
1568
1569 ENTITY Valeur_Composee
1570 ABSTRACT SUPERTYPE OF (ONEOF(Valeur_Enregistrement, Valeur_Tableau))
1571 SUBTYPE OF (Valeur);
1572 valeurs : LIST [0 : ?] OF Valeur;
1573 END_ENTITY;
1574
1575
1576 ENTITY Valeur_Enregistrement
1577 SUBTYPE OF (Valeur_Composee);
1578 champs : LIST [0 : ?] OF Champ;
1579 END_ENTITY;
1580
1581
1582 (* soit réel (number=real) soit autre scalaire (number=integer) *)
1583 ENTITY Valeur_Scalaire
1584 SUBTYPE OF (Valeur);
1585 calcul : NUMBER;
1586 END_ENTITY;
1587
1588
1589 ENTITY Valeur_Tableau
1590 SUBTYPE OF (Valeur_Composee);
1591 END_ENTITY;
1592
1593 (*Une Variable est considérée comme un cas particulier d'expression
comme dans
1594 les grammaires des langages de programmation.*)
1595 ENTITY Variable
1596 SUBTYPE OF (Expression);
1597 nom : STRING;
1598 valeur : OPTIONAL Valeur; -- existe si elle est valuée
```

```
1599 DERIVE
1600 SELF\Expression.representation : STRING := nom;
1601 -- where : la valeur est du bon type. Mais ce n'est pas vérifiable
d'après notre modèle :
1602 -- une valeur ne connait pas son type. Ex valeur.calcul=1 possible
pour type_logique ou
type_entier ou ...
1603 INVERSE
1604 declaration : BAG [0 : 1] OF Bloc FOR variables;
1605 modifications : SET [0:?] OF simple FOR variables; -- instructions
simples dans lesquelles
elle est modifiée
1606 OPERATIONS
1607 SELF\Expression.evalue: GENERIC ;
1608 END_ENTITY;
1609
1610 OPERATION Variable.evalue: GENERIC ;
1611
1612
1613 IF ( 'SEMANTIQUE.TYPE_LOGIQUE' IN TYPEOF (son_type) ) THEN
1614 RETURN (valeur.calcul =1);
1615 ELSE
1616 IF ( ('SEMANTIQUE.TYPE_REEL' IN TYPEOF (son_type)) OR
('SEMANTIQUE.TYPE_ENTIER' IN TYPEOF (son_type))) THEN
1617 RETURN(valeur.calcul);
1618 ELSE
1619 IF ('SEMANTIQUE.TYPE_ENUMERE' IN TYPEOF (son_type) ) THEN
1620 RETURN (son_type\Type_Enumere.valeurs[valeur.calcul]);
1621 END_IF;
1622 END_IF;
1623 END_IF ;
1624 -- si autre type, erreur car pas de return ...
1625
1626 END_OPERATION;
1627
1628
1629 FUNCTION calcule_les_dans ( i : instruction ) : LIST [0:?] OF Bloc ;
1630 IF 'SEMANTIQUE.Bloc' IN TYPEOF(i) THEN
1631 RETURN ( calcule_les_dans_Bloc ( i ) ) ;
1632 END_IF;
1633 IF i.dans=[] THEN
1634 RETURN ([]); -- assert(sous_programme<>[])
1635 ELSE
1636 RETURN ( calcule_les_dans_Bloc ( i.dans[1] ) + [i.dans[1]]);
1637 END_IF;
1638 END_FUNCTION ;
1639
1640
1641 FUNCTION calcule_les_dans_Bloc ( i : Bloc ) : LIST [0:?] OF Bloc ;
1642 IF i.dans=[] THEN
1643 IF i.dans_conditionnelle=[] THEN
1644 RETURN ([]); -- assert(sous_programme<>[])
1645 ELSE
1646 RETURN ( calcule_les_dans ( i.dans_conditionnelle[1] ) );
```

```
1647 END_IF ;
1648 ELSE
1649 RETURN ( calcule_les_dans_Bloc ( i.dans[1] ) + [i.dans[1]]);
1650 END_IF;
1651 END_FUNCTION ;
1652
1653
1654 FUNCTION calcule_les_dans_conditionnelle ( ins : instruction ) : LIST
[0:?] OF conditionnelle;
1655 LOCAL
1656 lc : LIST [0:?] OF conditionnelle := [] ;
1657 END_LOCAL;
1658 REPEAT i := 1 TO SIZEOF(ins.les_dans) ;
1659 IF ( ins.les_dans[i].dans_conditionnelle <> [] ) THEN
1660 INSERT ( lc , ins.les_dans[i].dans_conditionnelle[1] , SIZEOF(lc) ) ;
1661 END_IF ;
1662 END_REPEAT ;
1663 RETURN ( lc ) ;
1664 END_FUNCTION ;
1665
1666
1667 FUNCTION calcule_bloc ( i : instruction ) : SET [0:1] OF bloc ;
1668 -- Rend le plus proche bloc pere de i
1669 IF (i.dans=[]) THEN
1670 RETURN ( [] );
1671 ELSE IF ('SEMANTIQUE.BLOC' IN TYPEOF ( i.dans[1] )) THEN
1672 RETURN ( [i] );
1673 ELSE
1674 RETURN ( calcule_bloc (i.dans[1]) );
1675 END_IF;
1676 END_IF;
1677 END_FUNCTION ;
1678
1679
1680 FUNCTION calcule_contexte ( b : bloc ) : SET [0:?] OF Variable ;
1681 -- Rend toutes les variables vues par le bloc
1682 IF (b.bloc=[]) THEN
1683 RETURN ( b.variables );
1684 ELSE
1685 RETURN ( b.variables + calcule_contexte (b.bloc[1]) );
1686 END_IF;
1687 END_FUNCTION ;
1688
1689
1690 FUNCTION calcule_numero ( s : instruction ) : INTEGER ;
1691 -- rend le numero d'ordre de s dans la liste des instructions de son
pere
1692 -- nécessite que le père existe => dans<>[]
1693 IF s.dans<>[] THEN
1694 REPEAT i := 1 TO SIZEOF(s.dans[1].instructions) ;
1695 IF get_oid(s)=get_oid(s.dans[1].instructions[i]) THEN
1696 RETURN (i);
1697 END_IF ;
1698 END_REPEAT ;
```

```
ELSE
RETURN (0);
END_IF ;
print_nl('dans calcule_numero pas de return');
END_FUNCTION ;


FUNCTION calcule_suivante ( s : instruction ) : LIST [0:1] OF
instruction;
-- rend l'instruction suivante de s dans la liste des instructions de
son pere
IF s.numero=0 THEN
 RETURN ([]);
END_IF ;
IF s.numero=SIZEOF(s.dans[1].instructions) THEN
 RETURN ([]);
END_IF ;
 RETURN ([s.dans[1].instructions[s.numero+1]]) ;
END_FUNCTION ;


(*FUNCTION oid_type(nom:STRING):type_donnee;
 RETURN(QUERY(t<*POPULATION('SEMANTIQUE.TYPE_DONNEE')|t.nom=nom)[1]);
END_FUNCTION;*)


FUNCTION type_coherent ( v : variable ; e : expression ) : BOOLEAN ;
 RETURN (v.son_type = e.son_type) ;
END_FUNCTION ;


FUNCTION types_coherents ( vs : LIST [0:?] OF variable ; es : LIST
[0:?] OF expression ) : BOOLEAN ;
REPEAT i:=LOINDEX(vs) TO HIINDEX(vs);
IF (NOT type_coherent (vs[i],es[i]))THEN
RETURN (FALSE);
END_IF ;
END_REPEAT;
RETURN (TRUE);
END_FUNCTION ;

END_SCHEMA;
```

Annexe B : Exercices avec MELBA

1 Exercice 1 : Reporter des notes

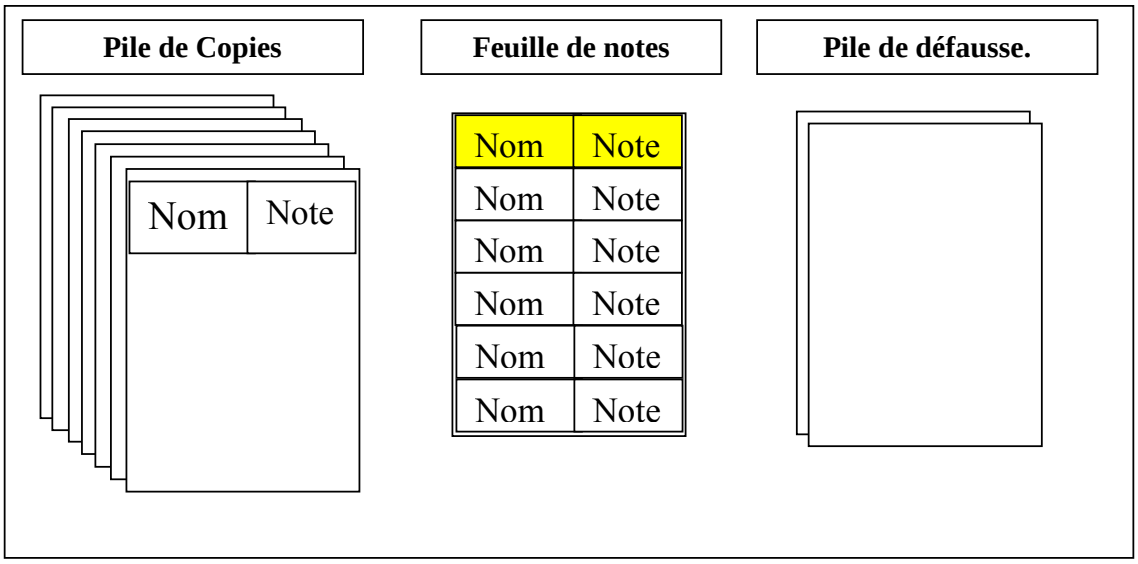
Le but de cet exercice est d'évaluer (et éventuellement de modifier) un programme qui automatise la tâche de report de notes par une secrétaire. Celle ci a devant elle :

- Une pile de copie
- Une feuille de notes comportant une liste de noms associés à la note de l'élève
- Une pile de défausse des copies.

On suppose que chaque étudiant de la liste a rendu sa copie. La « secrétaire automatique » est capable d'exécuter les opérations suivantes :

- Se placer au sommet de la liste de la feuille de notes : premiere_ligne ;
- Passer à la ligne suivante de la feuille : ligne_suivante ;
- Recopier la note de la copie au sommet de la pile (de gauche) dans la case « note » de la ligne courante dans la feuille de note : recopier_note ;
- Mettre la copie du dessus sur la défausse : defausser_copie ;
- Remettre toutes les copies de la défausse dans la pile : rempiler ;
- Tester si le nom de la copie en haut de la pile et celui de la ligne courante correspondent : noms_correspondent , noms_different
- Tester si la pile de copie est vide : pile_vide
- Tester la ligne courante est la dernière de la liste : derniere_ligne

Au début de la tâche, la « ligne courante » est celle du haut de la feuille.



```
Programme reporter_notes
  Begin
  Repeat
    Begin
    While noms_different
      Repeat
        Begin
        ligne_suivante ;
        End
    recopier_note ;
    defausser_copie ;
    End
    Until pile_copies_vide
  End
```

Question 1 :
Exécutez le programme de gauche avec la pile de copies suivantes :
((Eleve1,12), (Eleve4,13),(Eleve5,16), (Eleve3,14), (Eleve2,15))
– la feuille de note est bien sûr ordonnée alphabétiquement (Eleve1 → Eleve2 → Eleve3 … etc) – que constatez vous ? Expliquez !

Question 2 :
Proposez une solution au problème soulevé par la question 1 en modifiant le programme. Re-exécutez pour valider le fonctionnement du nouveau programme sur l'exemple précédent.

```
Programme reporter_notes_bis
  Begin
  Repeat
    Begin
    While noms_different
      Repeat
        Begin
        defausser_copie ;
        End
    rempiler ;
    recopier_note ;
    ligne_suivante ;
    End
    Until derniere_ligne
  End
```

Question 3 :
Le programme « bis » est il correct ?
Si non, expliquez pourquoi à partir du plus simple exemple possible, et corrigez le.

Question 4:
Admettons que tous les élèves n'aient pas rendu de copie. Exemple :
((Eleve1,12),(Eleve4,13), (Eleve5,16), (Eleve2,15)) avec la même feuille de notes.
Cela invalide t-il les deux programmes ? Si oui dites en quoi, et proposez une correction…

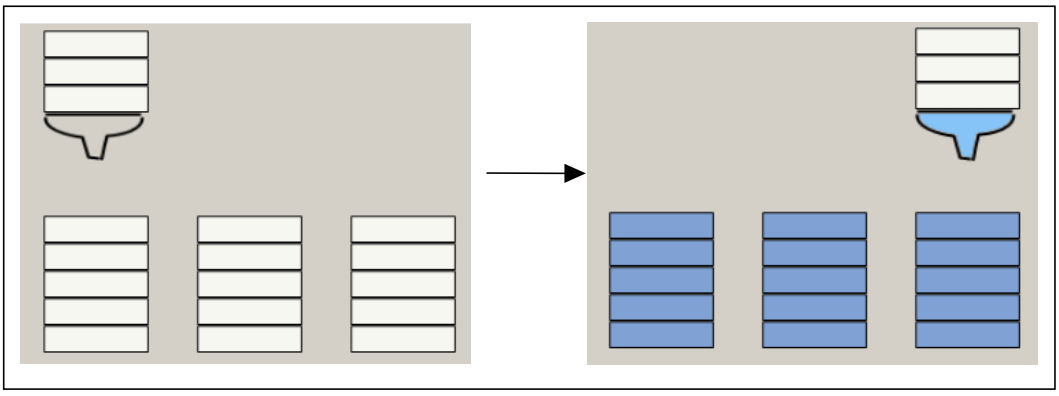
2 Exercice 2 : Le compte-goutte

Le but de cet exercice est d'écrire un programme qui automatise la tâche de remplissage de verres avec un compte-goutte. L'exécutant a devant lui :

- Un alignement de verres, tous initialement vides et de capacité égale.
- Un compte-goutte, initialement vide et au dessus du premier verre.
- Le nombre de verres, la capacité de la pipette et des verres peuvent varier.

L'exécutant est capable des opérations suivantes :

- Placer le compte-goutte sur le premier verre → `premier_verre ;`
- Passer au verre suivant → `verre_suivant ;`
- Presser une goutte → `presse_compte_goutte ;`
- Remplir le compte gouttes → `remplir_compte_goutte ;`
- Constater qu'il se trouve sur le dernier verre → `dernier_verre`
- Constater que le verre courant est plein → `verre_rempli`
- Constater que tous les verres sont pleins → `verres_remplis`
- Constater que le compte goutte est vide → `compte_goutte_vide`



Question 1
Ecrivez un programme permettant de répondre au problème dans le cas de l'exemple 1 :
- 3 verres de taille 5 , compte-gouttes de taille 4.

Question 2
Votre programme est il validé par l'exemple 2 : 2 verres de taille 4, compte-goutte de taille 5. Expliquez ce qui se passe, et modifiez le si nécessaire. Est il nécessaire de tester un cas où compte goutte et verres sont de même taille ?

3 Exercice 3 : Assemblage de polycopies

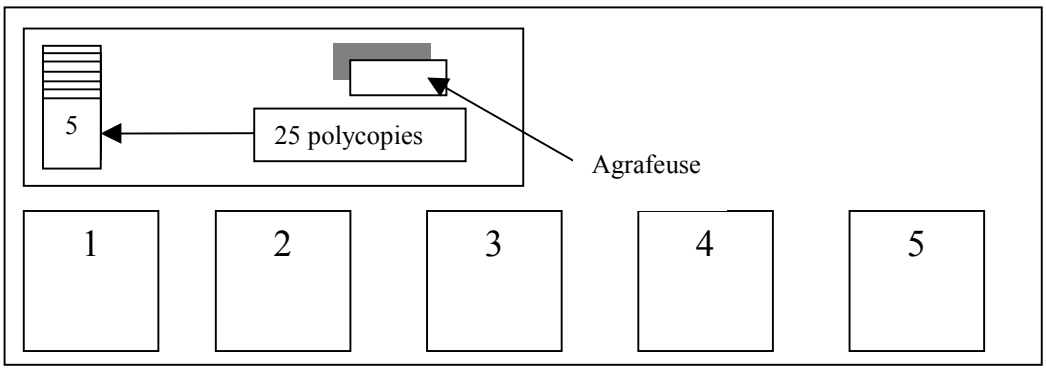
Le but de cet exercice est d'écrire un programme qui automatise la tâche d'assemblage de polycopies. L'exécutant a devant lui :

- Une table où sont empilées les polycopies (5 * 5 feuilles numérotées : 5 feuilles « 1 », en dessous de 5 feuilles « 2 »… etc jusqu'à 5 feuilles « 10 »).
- Une agrafeuse.
- cinq petites tables (numérotées de 1 à 5).

L'exécutant est capable des opérations suivantes :

- Se placer devant la première table → `aller_premiere_table ;`
- Passer à la table suivante → `aller_table_suivante ;`
- Se placer devant le bureau → `aller_bureau;`
- Prendre une feuille sur le bureau / la table courante → `prendre_feuille ;`
- poser une feuille sur le bureau / la table courante → `poser_feuille ;`
- Prendre toute la pile de feuilles sur le bureau / la table courante→ `prendre_pile ;`
- poser toute la pile de feuilles sur le bureau / la table courante → `poser_pile ;`
- agrafer toute la pile de feuilles sur le bureau / la table courante → `agrafer_pile ;`
- Constater qu'il se trouve sur la dernière table → `derniere_table`
- Constater que la table courante à le même numero que la feuille en main→ `numeros_correspondent`
- Constater qu'on a rien en main → `main_vide`
- Constater que la table courante / le bureau est vide → `table_vide`

*Si on a « en main » une pile de feuilles, les opérations / tests portant sur une feuille se réfèrent à la feuille du dessus. (çàd que l'on peut faire `prendre_pile ;` suivi de `poser_feuille ;`)



Question 1 :
Ecrivez un programme permettant de résoudre le problème.

Question 2 :
Plaçons nous dans un cas ou chaque feuille est en 10 exemplaires au lieu de 5 (il y a donc 10 feuilles « 1 », 10 feuilles « 2 »…). Cela invaliderait t-il votre programme ? Expliquez ! Si oui, proposez un programme qui marche dans les deux cas… sinon, cherchez un programme correct dans le premier exemple, pas dans le second.

4 Exercice 4 : Décodage Morse.

Le but de cet exercice est de concevoir un programme exécutant automatiquement une tache de décodage de message en morse. On dispose pour cela :

- Du message à traduire, qui se compose d'une suite de symboles :
- « `-----  --...  ..---` »
- Sur ce message, on a un symbole courant en train d'être traduit (surligné ci-dessus)
- D'une feuille faisant correspondre chaque symbole à une lettre / un chiffre (pour simplifier le problème, on ne prendra en compte que les chiffres)

Symbole	Traduction
`-----`	0
`.----`	1
`..---`	2
`...--`	3
`....-`	4
`.....`	5
`-....`	6
`--...`	7
`---..`	8
`----.`	9

- Sur cette feuille, il y a un couple courant (surligné aussi)
- Une feuille où écrire la traduction.

L'exécutant connaît les opérations suivantes :

- Se placer en haut de la liste symbole/chiffre → `premier_couple ;`
- Ici, le couple courant deviendrait : `(----- / 0)`
- Passer au couple suivant du tableau → `couple_suivant ;`

  Ici, le couple courant deviendrait : `(-.... / 7)`

- Recopier le chiffre du couple courant du tableau sur la feuille → `recopier_lettre ;` ici, on écrirait 6 sur la feuille de la traduction
- Passer au symbole suivant dans le message → `symbole_suivant ;`
  Ici le résultat serait : « `-----  --...  ..---` »
- Tester si le symbole courant dans le message correspond à celui du couple courante du tableau → `symboles_correspondent`
- Tester si on est *sorti* du message → `fin_message`
- Ce test renvoie vrai après l'exécution de `symbole_suivant` dans l'état : « `--  ---  --...  ..---` » (quand il n'y a plus de symbole surligné) .
- Tester si on est hors du tableau → `fin_tableau` (idem, il renvoie vrai après l'exécution de `couple_suivant` lorsque le couple courant était `(----. / 9),` lorsqu'il n'y a plus de couple surligné).

5 Exercice 5 : La caisse du distributeur de boissons.

Le but de cet exercice est de concevoir un programme permettant à un distributeur de sodas – ou de ce que vous voudrez - (qui accepte les pièces de 0,05 à 2€) de rendre la monnaie. Pour cela, il dispose d'une rangée de tiroirs contenant les différentes pièces, rangées de 2€ à 5 cents. Il peut répondre aux ordres suivants :

- aller_premier_tiroir → se met en face du tiroir contenant les pièces de 2€
- aller_tiroir_suivant → passe du tiroir de 2€ au tiroir de 1€, ou du tiroir de 1€ au tiroir 0,50€, etc…
- rendre_pièce → fait tomber une seule pièce du tiroir courant ; cette commande met à jour la somme qu'il reste à rendre…

De plus, on dispose des commandes de contrôle suivantes :

- pièce_>_somme → est vrai si la pièce du tiroir courant vaut plus que la somme à rendre au client.
- pièce_<_somme
- pièce_=_somme
- dernier_tiroir → est vrai si on est sur le tiroir de 0,05€. Si on tente de passer « au tiroir suivant » dans cet état, cela déclenche une erreur.

6 Minimum de trois nombres.

Question:
Ecrivez le programme qui à partir de trois INTEGER fournis, détermine le minimum de ces trois nombres, à partir du patron ci-dessous.

```
PROGRAMME calcule_minimum
    Entrées :
        nb1 est un INTEGER ;
        nb2
        nb3
    Sorties :
        resultat est un INTEGER ;
    Début
    […]
    Fin
```

7 Minimum de N nombres

Question:
Ecrivez le programme qui à partir d'un tableau d'INTEGER fourni (de taille non fixée), détermine le minimum de ces nombres (il sera stocké dans le paramètre de sortie « resultat »).

8 Tri d'un tableau de N nombres.

Question:
Vous êtes désormais capable de parcourir un tableau pour en retirer le plus petit élément. On souhaite maintenant se servir de cela pour écrire le programme qui réalise le tri, par ordre croissant (du plus petit au plus grand) d'un tableau d'INTEGER fourni (de taille non fixée).

Annexe C : Questionnaire de satisfaction MELBA

Renseignements

Sexe : F M

Age : ……..ans

Formation et niveau universitaire actuel : ……..

……..

Satisfaction de l'utilisateur

Indiquez votre degré d'adhésion aux affirmations suivantes : 1= pas du tout d'accord, 2= plutôt pas d'accord, 3= ni d'accord ni pas d'accord, 4= plutôt d'accord, 5= tout à fait d'accord. Cochez une seule réponse à chaque fois.

- **Utilisation de MELBA en général :**

1) L'interface du logiciel MELBA me plaît.

   1 2 3 4 5

2) La prise en main du logiciel MELBA me semble facile.

   1 2 3 4 5

3) Je me repère facilement dans l'interface du logiciel MELBA.

   1 2 3 4 5

4) Je n'ai éprouvé aucune difficulté à me servir du logiciel MELBA.

   1 2 3 4 5

5) Trouvez-vous que l'objectif pédagogique du logiciel MELBA est intéressant ?

Oui Non

Si non, pourquoi ?

……
……

6) Utiliseriez-vous ce logiciel pour vous aider dans votre cursus universitaire ?

Oui Non

Si non, pourquoi ?

……
……

- **Les différentes parties de MELBA :**

7) J'ai rapidement repéré les différentes parties de l'interface de MELBA.

1 2 3 4 5

a. J'ai clairement identifié leurs rôles respectifs.

1 2 3 4 5

8) L'agencement des différentes parties de l'interface de MELBA me semble logique.

1 2 3 4 5

9) Les différentes parties de l'interface de MELBA me semblent faciles à comprendre.

1 2 3 4 5

- **Utilisation de la partie instruction :**

10) Avez-vous utilisé les trois boutons d'édition du programme (ajouter, supprimer, remplacement d'une instruction) ?

Oui Non

Si non, pourquoi ?

……
……
……

11) L'agencement des informations (ex : liste déroulante…) dans la partie « instruction » de l'interface de MELBA me semble logique.

1 2 3 4 5

12) Les informations (ex : liste déroulante…) dans la partie « instruction » de l'interface de MELBA me semblent faciles à comprendre.

1 2 3 4 5

- **Utilisation de la partie historique :**

13) La partie « historique » du logiciel est bien mise en valeur et visible.

1 2 3 4 5

14) Avez-vous utilisé la zone « historique » ?

Oui Non

Si non, pourquoi ?

……
……
……

15) La hiérarchisation des informations dans la zone « historique » est facile à voir.

1 2 3 4 5

Opinion personnelle de l'utilisateur

D'après vous, que faudrait-il ajouter et/ou modifier au logiciel MELBA pour l'améliorer ?

……
……
……
……
……
……

……
……

Validation d'une approche basée sur exemples pour l'initiation à la programmation.

Présentée par :

Nicolas Guibert

Sous la direction de :

Patrick Girard et Laurent Guittet

Résumé Alors qu'ordinateurs et programmes informatiques se sont implantés dans nombre de disciplines scientifiques en tant qu'outils d'analyse ou instruments de mesure, l'acquisition des compétences requises pour la conception de programmes ne se fait pas aisément. De nombreuses études ont caractérisé les erreurs et difficultés rencontrées par les programmeurs novices. Les environnements actuellement utilisés pour l'apprentissage de la programmation se composent d'outils conçus dans une unique optique de développement, et non pas dans un cadre explicitement pédagogique. A cette approche « industrielle » s'oppose une approche explicitement pédagogique, où le but premier est la découverte et la construction de connaissances, et non pas la réalisation de tâches techniques. Cette Thèse étudie l'usage d'un paradigme de programmation alternatif, la programmation graphique sur exemple, comme support à la construction active d'un savoir viable par l'étudiant, en s'appuyant sur des expérimentations en situation réelle avec un environnement adapté, conçu explicitement pour l'apprentissage.

Mots-Clés Programmation sur Exemple, Interaction Homme-Machine (IHM), Environnements Informatisés pour l'Apprentissage Humain (EIAH), Psychologie de la Programmation, Etudes Expérimentales.
