N° d'ordre: 6625

THÈSE

présentée

devant l'Université de Paris XI

pour obtenir

le grade de : DOCTEUR EN SCIENCES DE L'UNIVERSITÉ DE PARIS XI ORSAY
Mention INFORMATIQUE

par

Catherine LETONDAL

Équipe d'accueil : LRI
École Doctorale : Informatique
Composante universitaire : UNIVERSITÉ DE PARIS SUD

Sujet :

Interaction et Programmation

soutenue le 27 Septembre 2001 devant la Commission d'examen

M. :	Christine	FROIDEVAUX	Présidente
MM. :	Henry	LIEBERMAN	Rapporteurs
	Patrick	GIRARD	
MM. :	Magali	ROUX-ROUQUIÉ	Examinateurs
	Manolo	GOUY	
	Michel	BEAUDOUIN-LAFON	

à François

The hardest single part of building a software system is deciding precisely what to build.

Frederick P. Brooks, Jr., *Computer Magazine; April 1987*

Remerciements

Je remercie Christine Froidevaux, Professeur, qui me fait l'honneur de présider ce jury.

Je remercie Henry Lieberman, Professeur, et Patrick Girard, Professeur, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Magali Roux-Rouquié, Directrice de Recherche, et Manolo Gouy, Directeur de Recherche, d'avoir bien voulu juger ce travail.

Je remercie Michel Beaudouin-Lafon, qui a dirigé ma thèse, d'être un directeur de thèse motivant, exigeant, pertinent et disponible malgré ses très nombreuses responsabilités.

Je remercie Laurent Bloch, ancien chef du Service d'Informatique Scientifique, de m'avoir encouragée à entreprendre cette thèse et de m'avoir sans cesse fait confiance sur le fond, bien que les formes n'y soient pas.

Je remercie Stewart Cole, Directeur des Equipements et Technologies Stratégiques, d'avoir autorisé la poursuite de cette thèse jusqu'à son terme.

Je remercie Wendy Mackay grâce à qui j'ai découvert d'autres "interfaces" de l'informatique.

Je remercie Wendy Mackay et Michel Beaudouin-Lafon de leur passion pour le progrès, de leur dynamisme et de leur enthousiasme.

Je remercie Stéphane Chatty de m'avoir fait découvrir le domaine de l'IHM lorsque je m'ennuyais comme ingénieur système au Centre d'Études de la Navigation Aérienne, et sans qui tout cela n'aurait tout simplement jamais eu lieu.

Je remercie Roland Moreau, grâce à qui cette thèse aura été heureuse (et sans qui il n'y aurait pas eu de vidéo à la soutenance !).

Je remercie Katja Schuerer, pour sa belle intelligence, qui me profite souvent, et pour son amicale compagnie.

Je remercie Mary O'Connell, qui m'a aidée ô combien de fois pour l'anglais, et pour son amitié.

Je remercie Stéphane Bortzmeyer, qui n'est jamais d'accord avec mes idées, mais qui au moins en discute.

Je remercie Éric Deveaud, qui n'est jamais d'accord avec mes idées, et qui en discute parfois.

Je remercie Bernard Caudron pour son soutien et son sens de la psychologie.

Je remercie Marie-France Sagot d'avoir toujours été solidaire et présente dans les moments difficiles. Je la remercie en particulier d'avoir assuré une sorte d'encadrement scientifique informel mais sensible, car il est difficile d'apprendre la recherche au quotidien en dehors d'un laboratoire de recherche.

Je remercie les équipes système et logiciels biologiques du service d'informatique : Daniel Azuelos, Marc Baudouin, Bernard Caudron, Éric Deveaud, Louis Jones, Michel Keller, Gérard Masson, Olivier Perret, Katja Schuerer, Annick Thébault, ainsi que Stéphane Bortzmeyer, Frédéric Chauveau, Irène Wang et Christophe Wolfhugel pour leur efficacité et leur aide technique. Je remercie en particulier Nicolas Joly d'avoir bien voulu prendre en charge l'installation des logiciels scientifiques, ce qui m'a permis de travailler sur cette thèse.

Je remercie Marie-Christine Boussin dont l'aide m'a été bien utile ces derniers mois.

Je remercie Alan Bleasby, du MRC, de m'avoir beaucoup encouragée au déploiement du logiciel Pise.

Je remercie Albane Leroch, Cédric Tétard et Alexis Gambis, mes "petits stagiaires" adorables qui ont fait de la fin de la thèse un moment riche et intéressant.

Je remercie Uwe Zdun et Gustaf Neumann d'avoir créé un si bon langage, XOtcl, avec de si bonnes idées. Sans leur travail, je ne sais pas où j'en serais aujourd'hui.

Je remercie Marina Zelwer et Raquel Tavares pour leur vivacité d'esprit, leur bonne humeur et leur souci des choses de la cité.

Je remercie Laurent Marsan pour sa gentillesse, son esprit critique et son souci des choses de la cité.

Je remercie Victoria Dominguez Del Angel, pour son amicale participation à mon travail et pour ses encouragements.

Je remercie Lionel Frangeul pour sa participation très professionnelle et assidue aux ateliers et à notre travail technique en général.

Je remercie François Huetz des intéressantes discussions que nous avons eues et pour l'intérêt qu'il porte à ma démarche.

Je remercie Michaela Muller-Trutwin, Céline Renoux, Pierre Roques, Pierre Dehoux, Rémi Cheynier, Sophie Bachellier et Elie Dassa pour leur participation enthousiaste à la démarche "participative"

et pour le soutien qu'ils m'apportent.

Je remercie également Charlie Roth, Valérie Caro, Jean-François Bureau, Pierre Daubersies, Christophe Guilhot, Yves Goguet, Nicolas Le Novere et de nombreux autres biologistes pour leur participation aux entretiens, aux séances de test d'interface et aux ateliers.

Je remercie Patrick Grimont, Gordon Langsley et Gilles Marchal de m'avoir consacré un peu de temps pour me parler de leur travail ou pour discuter de bio-informatique.

Je remercie Frédéric Guyon d'avoir bien voulu relire une partie du manuscrit et des conseils qu'il m'a donnés.

Je remercie Stéphane Conversy et Nicolas Roussel, anciens thésards au LRI, de m'avoir souvent bien fait rire.

Je remercie Denise Ogilvie pour son ouverture d'esprit, sa gaieté, son dynamisme et sa rapidité de compréhension en toutes choses.

Je remercie Martine Croissant, du LRI, pour son aide sympathique pour s'orienter dans les méandres administratives.

Je remercie le personnel de la bibliothèque de l'Institut Pasteur qui s'occupe des commandes de documents à l'extérieur. Ce service est une aide précieuse pour la poursuite d'un travail de recherche.

Je remercie encore l'équipe CEDRIC du CNAM, Gérard Florin, Éric Gressier, Pierre Cubaud et Laurence Duchien, pour leur dévouement pédagogique et pour m'avoir fait connaître il y a 10 ans un environnement de travail et de recherche plein de respect et d'estime mutuelle. Je n'ai pas oublié !

Je remercie enfin la bande du café chantant : Béatrice, Maryline, Christian, Isabelle, Gérard, Mylène, Kunio, Pascale, Étienne, Michel, Marie-Claire, Marie-Annick et tous les autres pour les bons et nombreux moments passés ensemble ces dernières années.

Introduction générale

La thèse qui est présentée dans ce manuscrit s'intéresse aux diverses possibilités pour les biologistes de mieux contrôler les systèmes informatiques qu'ils sont amenés à utiliser. L'informatique est en effet devenue un élément fondamental de la biologie, qu'il s'agisse de la recherche de nouveaux algorithmes d'analyse des données biologiques apportées entre autres par le séquençage des génomes, ou qu'il s'agisse de manipulations quotidiennes d'outils informatiques. Mais ce développement ne va pas de pair avec la maîtrise de ces objets ne seraient-ce que par ceux-là même qui en auraient besoin.

L'approche que nous avons choisie pour traiter cette question est une approche de recherche en informatique, sous la forme d'une thèse, et cela surprend plus d'un collègue biologiste. Ne suffit-il pas de développer de nouvelles interfaces graphiques - des cliquodromes comme les appellent parfois ceux qui n'en programment jamais - comportant toutes les fonctions nécessaires pour réaliser les tâches d'analyse ou de manipulation de données ? Ne suffit-il pas de construire des machines-outils, pour reprendre une expression entendue dans une discussion à ce sujet, qui automatisent ces tâches pénibles afin d'en libérer le biologiste une bonne fois pour toutes ? La réponse n'est pas simple : pour de nombreux cas, en effet, ces solutions conviennent très bien, et nous y avons d'ailleurs contribué. Mais tout aussi nombreux sont les cas où cela ne suffit pas, et dans ces cas-là, la seule solution qui reste est de fabriquer les outils par soi-même, c'est-à-dire de programmer. C'est d'ailleurs ce que font un nombre grandissant de biologistes : la majorité des outils d'analyse sont écrits par des biologistes qui ont pris goût à l'informatique et à l'écriture d'algorithmes. D'autres parviennent parfois à s'associer avec un informaticien, mais les centaines de milliers de biologistes dans le monde, ou même seulement ceux qui en auraient besoin, ne peuvent pas disposer chacun d'un informaticien personnel.

Mais programmer, c'est difficile, et cette thèse n'a ni pour argument ni pour but de dire le contraire. Plusieurs domaines de recherche en informatique se sont attaqués à cette question : la psychologie de la programmation, la pédagogie informatique, les langages de programmation visuels, la programmation par démonstration, ... autant d'approches dont nous essayons de donner un panorama réparti dans plusieurs chapitres de ce manuscrit, mais dont aucune n'a trouvé de solution miracle. Et surtout, programmer n'est pas une chose nécessairement intéressante en soi. Que diraient les informaticiens si on les obligeait à faire des PCR ou des *northern* ?

La démarche que nous avons suivie est partie d'une double interrogation, la première sur les rapports entre deux activités dont l'une est considérée comme conviviale : *interagir* , et l'autre rébarbative : *programmer* ; la seconde sur les rapports entre deux faces de l'informatique, dont l'une correspond au monde des non-professionnels : *utiliser* , et l'autre au monde des professionnels : *programmer* . Ce que nous avons essayé de mettre en évidence à travers cette thèse, c'est que, alors que dans le monde réel les dichotomies ne sont pas aussi tranchées [Nar95], loin s'en faut, les approches techniques en revanche s'appuient fortement sur ces distinctions [Eis97]. En effet, nous verrons que les interactions avec un logiciel sont parfois d'une complexité équivalente à de la programmation et, inversement, que l'interaction et l'utilisation peuvent servir à programmer, ou encore qu'il existe des formes d'utilisation qui servent à programmer, à modifier le logiciel, comme les techniques de personnalisation. Notre hypothèse est que le problème est là : à force de supposer que *nous savons bien ce qu'est programmer* - nous le savons effectivement sur le plan théorique depuis les travaux de Turing, mais ici ce sont les aspects pratiques qui sont en jeu ; à force de constater qu'un logiciel est quelque chose qui s'*utilise* , mais qu'il est quasiment interdit d'en faire autre chose, il y a peut-être des solutions au problème que nous posons ici qui n'apparaissent pas aisément. On peut alors soit faire en sorte que l'interaction serve directement à programmer, soit rendre la programmation plus accessible en la situant dans le cadre d'une utilisation. C'est la deuxième option que nous avons choisie de développer, sous la forme d'un environnement de travail qui est à la fois un environnement de programmation et d'analyse

de séquences. Mais chacune de ces approches tient compte du fait que la programmation n'est pas nécessairement un but en soi, mais une forme d'utilisation un peu spéciale qui peut être utile le cas échéant.

Par rapport aux approches de psychologie cognitive pour comprendre et applanir les difficultés inhérentes à la programmation, mettre l'accent sur l'accessibilité de la programmation et le contexte dans lequel elle peut se produire est plutôt une approche de conception d'environnement. Mais s'agit-il de construire un environnement de programmation de plus ? La réponse est non, tout d'abord parce que cet environnement de programmation doit être *en même temps* un environnement de travail. Mais c'est aussi parce que la conception de cet environnement a bénéficié d'une participation importante des biologistes afin de déterminer les concepts, aussi bien de l'application que des outils de programmation, qui ont de l'importance, et de travailler *effectivement* avec eux à la forme que ces éléments devaient prendre.

Ce manuscrit est composé de la manière suivante : un premier chapitre expose plus précisément le problème que nous avons évoqué ici. Le second chapitre développe les approches centrées sur l'utilisateur qui sont une tentative pour trouver des éléments confirmant notre hypothèse. Le chapitre suivant a pour objet de poser la question de la flexibilité sur le plan technique et sur le plan de la conception. Un quatrième chapitre décrit les éléments dont nous disposons dans l'état actuel de la bio-informatique. Enfin le dernier chapitre présente les développements que nous avons effectués pour étayer nos hypothèses.

Chapitre 1

Introduction

1.1 Ouverture et interactivité des systèmes informatiques.

D'après [Weg98] ou [WG99b] les systèmes informatiques sont de plus en plus interactifs, non pas dans le sens où ils utilisent la manipulation graphique, mais dans le sens où ils correspondent de moins en moins au modèle classique du programme algorithmique transformateur d'une entrée en une sortie déterministe. Un système *interactif* se définit comme un système à entrées échelonnées dans le temps et dont l'entrée Ik+1 est déterminée par la sortie Ok et éventuellement par des éléments extérieurs au système [Weg98]. C'est à ce type de comportement que correspondent ces logiciels de l'économie symbolique [CIS92], dits aussi outils d'intelligence [NZ93] - non plus dans le sens d'intelligence artificielle sous forme algorithmique mais dans celui de l'intelligence humaine, celle des chercheurs scientifiques [Bro87], des *knowledge workers* [CIS92] ou des *visual thinkers* [CKM93a] se servant des formalismes visuels et interactifs pour modéliser et résoudre des problèmes. Une graduation des tâches informatiques selon le degré plus ou moins important d'intervention humaine est proposée dans [SOM87] : les plus automatisables sont les tâches de *transport* et de *transformation* , suivies par les *traitements déterministes d'information* . Viennent ensuite en position intermédiaire - et c'est là que nous nous situons - les *traitements complexes multi-dimensionnels* , faisant éventuellement intervenir la recherche de patterns ou la négociation (enseigner, persuader, apprendre), mais nécessitant un apport informatique. Puis viennent les tâches de *création d'information* dans lesquels il y n'y a plus d'informatique.

Mais c'est, curieusement, l'idée de *programmation* , ou le terme *programmable* , qui sont parfois utilisés pour décrire la très grande *interactivité* de ces outils. Ainsi, la classification des systèmes de visualisation scientifique de [BHP+94], illustrée dans la figure 1, place ces systèmes sur plusieurs axes, dont un mesure la possibilité pour l'utilisateur de contrôler le déroulement de l'exécution, allant des systèmes à post-traitement sans aucune interaction durant le processus de calcul, jusqu'aux systèmes complètement programmables comme Khoros [RASW90] ou ThingLab [Bor81].

En réalité, on voit bien ici que les systèmes les plus programmables sont les plus interactifs, ceux qui permettent le plus de contrôle et d'interaction, par opposition à ceux qui ne redonnent le contrôle à l'utilisateur qu'une fois le calcul fini.

C'est aussi en terme de *contrôle* que [Rep93a] décrit l'utilité de la programmation pour des non professionnels : selon lui (et selon [Nar95]), la programmation, du moins si elle est réalisée avec un outil suffisamment souple, tend à aider dans le processus de résolution de problèmes. Cela différencie complètement la programmation non-professionnelle, plus exploratoire, de la programmation professionnelle dont le but explicite est de construire des systèmes sur la base de solutions connues. Dans [NZ93] les sujets d'une enquête sur l'utilisation des tableurs pour la résolution de problèmes racontent que ces outils leur permettent de définir par eux-mêmes, de manière itérative, les données du problème à résoudre, et notamment les paramètres qui ne sont pas nécessairement clairs dans leur esprit. Ainsi la programmation constitue un outil d'aide à la modélisation pour le scientifique, et moins il y a d'intermédiaires informaticiens, plus précis est le modèle. C'est pourquoi nous sommes à mi-chemin entre utilisation et programmation : même si les moyens utilisés pour la résolution de problème sont

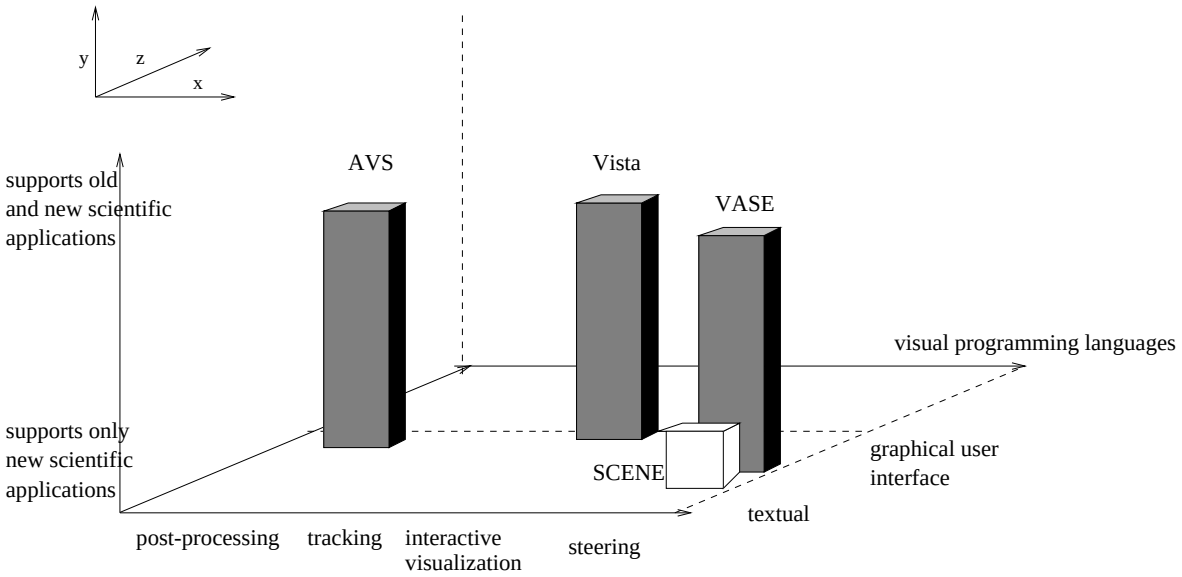


FIG. 1.1 – Classification des systèmes de visualisation scientifique (emprunté à [BHP+94]).

l'écriture d'algorithmes, il ne s'agit peut-être pas toujours de programmation au sens où on l'entend habituellement (fabrication d'un logiciel). Cet aspect bricolage, d'*instant software* [CIS92] est tout à fait légitime dans notre contexte.

Il est clair que, dans ces deux exemples, programmer veut dire *paramétrer* un appareil (programmer une machine à laver ou un réveil - une des forme les plus courantes de programmation selon [Shn00]), lui donner le plus d'informations possibles pour qu'il fasse ce qu'on veut. Comme le dit également [BD95] à propos de la customisation, on est ici dans le cadre de la programmation comme une interaction : la *programmabilité* correspond à une meilleure *interactivité* . [Rep93a] le formule assez bien : tout l'art de l'interaction homme-machine, c'est de savoir quel est le moment approprié, selon le niveau de contrôle/effort voulu, pour invoquer les fragments de programmes.

Cependant, face à cette évolution dans l'utilisation des systèmes informatiques, non seulement l'idée courante de ce qu'est et de ce que doit être la programmation n'a pas beaucoup évolué, mais les modèles théoriques de l'informatique n'ont pas beaucoup changé. [BL93a] constate en effet que, comparés aux formalismes tels que le lambda-calcul ou la machine de Turing, les systèmes interactifs ne possèdent pas de modèle formel suffisant pour les décrire, les spécifier ou les valider. Par exemple [CD97] décrit le Web comme une machine informatique qui justifierait un langage mais qui ne correspond pas aux modèles classiques des machines informatiques : ni à la machine de von Neumann, car il n'y a pas de programme enregistré en mémoire ou de compteur d'instructions, ni à une collection de machines de von Neumann car chaque machine est protégée des accès extérieurs, ni à un système de fichiers car il n'y a pas d'instruction d'écriture, ni à une base de données distribuée car il n'y a ni schéma des données, ni langage de requêtes, ni mécanismes de cohérence ; il ne correspond pas non plus à un système objet distribué à cause du problème de l'intégrité des références et de la sémantique RPC. Il est difficile de modéliser un système interactif comme un calcul, avec en entrée les actions de l'utilisateur et en sortie l'affichage à l'écran : cette description serait beaucoup trop complexe, et, d'après [GW98] les systèmes interactifs, décrits par des entrées successives tenant compte des sorties intermédiaires, sont peu adaptés à la modélisation fonctionnelle, complètement décrite par un calcul sur une entrée. D'après [Weg98] un tel modèle serait de toute façon différent de celui de Turing : il ne pourrait pas être *complet* , car cela supposerait que le système interactif, censé englober l'utilisateur et son environnement, soit un système fermé et réductible à une syntaxe.

Dans la logique de cette approche, qui considère presque qu'un système informatique se doit d'être incomplet, l'idée d'un système ouvert paraît assez naturelle. Il faut comprendre le terme "système ouvert" de la manière suivante :

1. un système capable de rendre compte de lui-même comme d'un produit rationnel (boîte blanche)

[Dou97][DE95a][dS00][KdRB91][HH99], voire une théorie empirique et réfutable [Mør95],

2. un système évolutif et modifiable [FG90][Mør97c][MCLM90].

Le premier sens pose la question de la construction des systèmes interactifs. Le deuxième aspect est la possibilité de programmation de ce système par son utilisateur.

1.2 Objectifs.

Le rôle de cette section est, en partant des problèmes tels qu'ils sont cités dans la littérature, de dégager progressivement les objectifs réels que nous pensons devoir suivre. En effet, les formulations, en particulier celle de "programmation par l'utilisateur final", nous semblent trop préjuger de certains aspects de la problématique à laquelle nous nous attachons. Qu'entend-on par programmation, par utilisateur final, quelles sont les buts et les principes de ces approches ? Que voulons-nous exactement, et comment nous situons-nous par rapport à ces techniques ? Pour trouver une réponse à ces questions, une première sous-section analyse les termes de l'expression "programmation par l'utilisateur final" ; une seconde sous-section recentre la problématique sur l'idée plus générale de répartition du travail ; puis une dernière sous-section précise les aspects que nous considérons comme importants pour le contexte de la programmation par les biologistes.

1.2.1 Programmation par l'utilisateur final.

L'expression "programmation par l'utilisateur final" (end-user programming) recouvre plusieurs aspects :

- l'*activité de programmer* par l'utilisateur final ;
- un domaine de *recherche* ;
- un ensemble assez riche de *techniques* (programmation graphique, programmation par démonstration, programmation dans un langage spécialisé, écriture de scripts ou de macros, tableurs,...), La programmation par l'utilisateur correspond à de très nombreuses activités et semble d'ailleurs difficile à circonscrire autrement que par les techniques employées.

Ces techniques comprennent essentiellement :

- la *définition de préférences* , consistant à choisir des valeurs dans une liste prédéfinies dans un formulaire ;
- la *programmation avec un langage adapté au domaine* qui consiste à proposer par exemple un mécanisme de calcul de haut niveau dans lequel l'utilisateur intègre ses spécifications, sous une forme proche de la programmation déclarative, par exemple sous forme de contraintes, de descriptifs (bases de données, composants graphiques), de formules (tableur) ou de règles logiques (systèmes experts) ;
- la *composition* , le scripting, c'est-à-dire la spécification d'un enchaînement d'actions, assistée ou non, visuelle ou non ; l'assistance peut résulter d'un enchaînement effectif d'actions que le système propose à l'utilisateur de réifier dans une commande ; figure également dans cette rubrique la spécification visuelle non assistée : tous les environnements dataflow à gros grain, comme Khoros/Cantata [RASW90] qui compose des modules de traitement d'images ;
- la *programmation par l'interaction* ou programmation par démonstration (autre que la composition d'actions) : l'idée directrice est que c'est l'interaction qui est le langage de description ; cela peut consister à construire un modèle (d'une expression régulière comme avec SWYN [Bla00]), des styles de texte comme avec Tourmaline [Wer92] ou [Mye91], une grammaire comme avec Grammex [LNW98], un format [Mas00], une interface utilisateur comme pour Amulet [MMM+97], ou bien des objets divers, par exemple graphiques comme avec Metamouse [MWK89], ou encore des jeux comme StagecastCreator [SCT00] ;
- la *programmation visuelle* : l'objet à construire est explicitement un programme, le mode d'expression pouvant être proche de ceux utilisés dans la catégorie précédente [Smi75] ; la limite entre les deux n'est d'ailleurs pas simple à déterminer : par exemple, un langage de démonstration d'interface utilisateur a bien pour objet de construire une partie d'un programme.

Il existe des définitions de la programmation par l'utilisateur n'utilisant pas le critère de la technique de spécification, mais plutôt celui de la tâche à réaliser ou du contexte dans lequel elle se produit. Pour [CIS92], on parle de programmation par l'utilisateur dès lors que l'utilisateur et le programmeur d'un logiciel sont la même personne, et celle-ci se caractérise par son rôle exploratoire et temporaire. Il s'agit essentiellement d'automatiser l'utilisation, de transformer et de manipuler ses données (cette description convient assez bien au domaine de la bio-informatique). [dS00] définit les langages pour l'utilisateur comme devant permettre deux types de construction : des fonctions du domaine, ainsi que des composants graphiques pour les utiliser dans l'interface.

1.2.1.1 La programmation sans programmer.

Les approches dites de programmation par démonstration ou de programmation sur exemples représentent l'un des axes de recherche les plus riches dans le domaine de la programmation par l'utilisateur. L'idée centrale de ces approches est, non pas de ne pas programmer du tout, mais de pouvoir le faire sans écrire de code, ou plus exactement, dans un langage de spécification familier à l'utilisateur. Et un langage a priori familier à l'utilisateur est celui de l'application elle-même.

Usages

Ce type de programmation peut servir selon [Cyp93a] :

1. à effectuer de petits changements dans un logiciel, comme enregistrer définitivement la réponse à une boîte de dialogue afin de supprimer une étape dans une action effectuée fréquemment ;
2. à contruire de petites applications, comme le permet HyperCard ;
3. à l'automatisation d'actions répétitives, qui est sans doute l'application la plus courante et la plus importante.

Spécifications familières

Les types de spécification "familiers" à l'utilisateur, c'est-à-dire les données à partir desquelles le système peut fabriquer un programme, peuvent être par exemple :

- les *interactions habituelles de l'interface utilisateur* , comme dans les systèmes d'enregistrement de macros, ou ceux qui, comme Chimera [CKM93b], fonctionnent sur un historique (graphique) pour la sélection interactive des commandes qui construisent le programme ; Pygmalion [Smi75], un système iconique d'édition directe des états successifs du programme (un des premiers systèmes de programmation par démonstration) ; SmallStar [Hal93], un outil de programmation dans l'interface (l'invention du terme vient de là) appliqué au bureau de Xerox Star, Rehearsal World [FG84], fonctionnent également sur le principe de l'enregistrement d'actions ;
- les *états de l'exécution* , comme pour les systèmes inférant un programme à partir d'un état initial et un état final [SCT00], ou bien comme Mondrian [Lie92], qui construit une procédure de dessin à partir d'un storyboard des états successifs ; on peut inclure dans cette catégorie les systèmes de construction d'interfaces ou de dessins géométriques qui définissent des contraintes et des relations entre les objets d'après l'état courant de l'interface ou des figures géométriques ;
- des *données correspondant à un modèle* : dans le cas de [Gir00], le modèle est un composant industriel, dans celui de SWYN [Bla00] ou de Grammex [LNW98] les données sont un ensemble de mots dont peut être déduite une expression régulière ou une grammaire, dans celui de Tourmaline [Wer92] les données sont le document dont on extrait le format.

Inférences

Quelle que soit la technique utilisée, les spécifications données par l'utilisateur ont un caractère assez littéral, et il faut, pour construire un programme qui soit réutilisable dans d'autres contextes, opérer quelques transformations pour le généraliser, c'est-à-dire spécifier des variables, des paramètres, du contrôle et des conditions (en somme tout ce que "programmer du code" signifie le plus souvent). Les techniques utilisées pour construire le programme, construction qui est donc réalisée par un programme et non directement par l'utilisateur, se répartissent d'abord entre celles qui utilisent l'inférence et celles qui ne l'utilisent pas. Parmi celles qui utilisent l'inférence, on distingue ensuite [MMW00] :

- les *systèmes à base de règles simples* :
  - Peridot [CKM93c], un système pour créer des widgets, utilise par exemple 50 règles pour inférer

un layout : chaque règle contient 3 parties : le test, c'est-à-dire comment reconnaître qu'il faut appliquer la règle à un objet, le feedback, et l'action ;
- Tourmaline [Wer92], qui permet de définir des formats de documents, essaie de déterminer les différentes parties du document, comme l'en-tête et le corps : titre, section, ... ; le résultat inféré est éditable par une boîte de dialogue ;
- Pavlov [Wol96] sert à programmer des logiciels interactifs par l'exemple : le modèle utilisé est le modèle stimulus/réponse ou évènement/action avec proportionalité de la réponse ; les conditions sont des relations graphiques entre le stimulus et les objets que l'utilisateur peut éditer ; la démonstration est en plusieurs phases : situation initale, description du stimulus, description de la réponse ;
- Mondrian (procédures de dessin)

- les *systèmes qui utilisent des algorithmes plus complexes* , comme Gamut [MM97] qui infère le programme à partir d'exemples multiples, ou Eager [Cyp91b] qui déduit des motifs dans les actions de l'utilisateur, s'appuie sur une base de connaissances et s'en sert pour détecter des itérations.

Ces systèmes à base d'inférence se distinguent des autres selon [Gir00] en ce qu'ils construisent un nouveau fait : l'algorithme, ce qui n'est pas le cas des systèmes sans inférence, dont le résultat, l'objet déduit, est parfaitement connu de l'utilisateur. Ce qui caractérise également ces systèmes est le caractère *implicite* de l'intention de l'utilisateur, très opposé à la forme explicite d'un programme classique.

Interactions

Les systèmes n'utilisant pas l'inférence, comme Pygmalion, Tinker [Lie93], Rehearsal World, Small-Star, Triggers [Pot93b], demandent les informations qui leur manquent à l'utilisateur, et s'appuient sur ce que [Gir00] appelle des conventions de dialogue. Plusieurs systèmes à base d'inférence interagissent d'ailleurs aussi avec l'utilisateur pour lui demander confirmation et pour le laisser éventuellement modifier les définitions inférées, comme Tinker [Lie93] qui propose une liste objets et de descriptions, ainsi que Predictive Calculator [Wit93], Peridot, Metamouse, Eager, ou Turvy [Mau93] qui demandent une confirmation à l'utilisateur (Garnet [MGD+90] permet de désactiver cette demande de confirmation). On remarque que dans les deux cas, il s'agit d'interactions supplémentaires par rapport à l'application elle-même : elles sont donc moins "familières" que les spécifications dans le langage de l'interface, et ne doivent par conséquent ni être trop intrusives ni incompréhensibles. A titre de mauvais exemple, la boîte de dialogue de Netscape postée lors de la soumission d'un formulaire, demandant confirmation de la soumission à l'utilisateur, comporte également un bouton de "programmation par l'utilisateur", ou du moins de personnalisation, permettant à ce dernier de demander à ne plus être dérangé par la boîte de dialogue : de notre expérience durant des cours d'utilisation de l'informatique destinés aux biologistes, rares sont les personnes qui lisent ou qui comprennent ce message.

Utilisation des exemples

[Gir00] précise la terminologie associée à l'utilisation d'exemples : parmi les systèmes qui sont "example-based", ceux qui sont "with examples" utilisent les exécutions précédentes comme exemples d'instructions (do what I did), alors que ceux qui sont "by example" font appel à l'inférence (do what I mean). Les techniques sans inférence sont donc les techniques "with examples" et constituent essentiellement une aide pour l'utilisateur pour programmer les parties où il n'est pas expert.

Limites

Ces approches très utiles comportent des limites. La critique de [Nar95] porte essentiellement sur la construction d'exemples, qui seraient selon elle difficiles à trouver, ou sur l'aspect irréaliste de la déduction d'itérations à partir d'actions de l'utilisateur, qui normalement ne s'applique pas à répéter des actions consciencieusement, sans erreur ni variation, pour que le système les comprenne.

La principale limitation pour l'utilisation de ce type d'approche comme *solution générale* pour la programmation par l'utilisateur dans le domaine de la bio-informatique est leur caractère implicite, du moins pour les systèmes avec inférence. Les approches qui permettent l'édition du programme déduit, qu'elles soient sur exemples ou par exemples, conviennent mieux à cette nécessité de connaître de manière rationnelle et prédictible l'objet construit par le système. La meilleure illustration est celle des

systèmes déduisant des modèles (expressions régulières et grammaires) : ce qui est utile aux biologistes est de pouvoir réutiliser directement le modèle déduit, sans compter l'intérêt pédagogique qu'il y a à parvenir à maîtriser ces formalismes, qui sont d'ailleurs directement utilisés dans des banques de données biologiques, comme la banque Prosite [Bai91], une banque de motifs représentant des domaines protéiques, en réalité une banque d'expressions régulières. S'il est utile d'aider l'utilisateur à la formulation de ces objets, l'empêcher d'en acquérir les concepts est à notre avis plutôt désavantageux.

Situation dans la problématique

Enfin, dans le cadre du raisonnement que nous poursuivons ici, concernant les rapports entre programmation et interaction, ces approches apportent plusieurs éléments intéressants :

- elles donnent des exemples de spécification *interactive et incrémentale* de programmes ;
- elles donnent des exemples de spécification de programmes dans un langage qui n'est pas un langage de programmation, mais *un langage d'utilisation* ;
- elles sont une illustration limite, peut-être involontaire, d'une caractéristique de l'activité de programmation (ou de conception) qui consiste en un certain recul dans l'attitude de l'utilisateur par rapport au système qu'il est en train d'utiliser et qui soudain lui pose un problème suffisant pour qu'il décide de s'y intéresser ; cette démarche souvent évitée [Mac91a], et inhabituelle sinon "anormale" [HJ93] semble facilitée par des systèmes qui tentent d'immiscer la programmation - nous entendons précisément par là la modification du logiciel - dans une continuité avec l'utilisation de l'outil ;
- elles montrent que la programmation que l'on est obligé de faire est souvent *inutile* puisqu'elle pourrait pour une grande part être évitée par ces techniques ; mais à part les techniques d'enregistrement de macros, peu de ces techniques sont effectivement utilisées dans les systèmes courants (et à vrai dire, quand elles le sont, elles sont inutilisables, il suffit de songer au mécanisme d'enregistrement de macro *nommées* d'Emacs, ou à d'autres fonctions de personnalisation de cet éditeur de texte, fonctions dont l'inutilisabilité est décrite avec humour par [Mac91b]) ; et pourtant, l'ouvrage collectif [Cyp93b] recense, dans le chapitre sur l'histoire de ces approches, pas moins de 80 systèmes entre le début des années 1970 et 1993.

1.2.1.2 Qui sont les utilisateurs finaux ?

L'expression "programmation par l'utilisateur final" peut cependant laisser un peu perplexe. En effet, dans cette expression, le terme "utilisateur final" est à la fois très vague et très précis. Il est très précis car il désigne sans ambiguité la personne qui *utilise* , par opposition à celle qui programme. Il est aussi très vague, pour deux raisons au moins :

1. Il y a d'énormes différences entres les "utilisateurs finaux".
2. La programmation est quelque chose de difficile à définir.

L'expression "programmation par l'utilisateur final" est aussi un peu paradoxale puisqu'elle associe les deux extrémités opposées de la chaîne de fabrication du logiciel [dS00], et qu'elle propose en plus ce qui est réputé le plus difficile en informatique à quelqu'un pour qui on essaye habituellement d'applanir toute difficulté technique.

Dès le début de sa présentation du logiciel ThingLab [Bor77] spécifie quels en sont les destinaires : des programmeurs capables de décrire les contraintes relevant d'un domaine spécifique (loi d'Ohm,...) au sein d'un ensemble de classes, et des utilisateurs construisant une simulation à l'aide de ces composants. De même, le logiciel Cocoa de [SCT00] s'adresse sans ambiguité à des enfants et Elody, un environnement d'écriture musicale inspiré du lambda-calcul [OFL97] à des musiciens, voire à des compositeurs. Le système de logiciel adaptable de [Mør97b], s'il propose des principes généraux de personnalisation allant jusqu'à la programmation d'extension dans le langage d'implémentation du logiciel, destine cependant très clairement *ces* outils de programmation aux utilisateurs de *ce* même logiciel, c'est-à-dire à la fois des utilisateurs spécifiques du domaine du logiciel, et en connaissant bien l'usage.

Qui est donc cet utilisateur final ? Y a -t-il quelque chose de commun entre le comptable devant sa feuille de calcul, le compositeur spécifiant un circuit acoustique à l'aide d'un langage visuel comme OpenMusic [AADR98], un enfant créant un jeu avec Agentsheets [Rep93a], un biologiste devant sans cesse modifier ses procédures d'analyse pour faire avancer ses idées ou un scientifique programmant

un environnement de visualisation à l'aide de Vista [TJC91] ? Ce sont pourtant tous ces types d'utilisateurs qui sont regroupés sous le terme d'utilisateur final. Bien que tous ces exemples d'"utilisateurs finaux" soient surtout pris dans la catégorie des utilisateurs assez avancés, susceptibles de programmer, il y a d'énormes différences entre eux en tous cas beaucoup trop pour déterminer facilement des caractéristiques communes à toutes ces formes de programmation.

1.2.1.3 La programmation, c'est quoi ?

Aurions-nous plus de chance du côté de l'autre partie de l'expression ? Le terme programmation recouvre-t-il un objet plus précis ? Les articles de la littérature prennent souvent comme point de départ une définition de la programmation qui sert soit :

1. à définir le *but* de la programmation : contrôler la ressource ordinateur [Rep93a], formaliser un modèle ou un problème, adapter une solution [Nar95],...) ;
2. à définir les fonctions d'un environnement d'aide à la programmation : programmer ces définir des variables et des constructions conditionnelles et itératives [Gir00] ;
3. à décrire le processus mental correspondant à l'activité de programmation : programmer c'est transformer un plan mental en un programme [Gre91][PRM00] ;
4. à explorer l'utilisabilité d'un problème lié à la programmation, comme le déboguage : programmer c'est la représentation statique d'un processus dynamique [LF95] ;
5. à faire une taxonomie [Mye90], ou à classer ou délimiter un thème de recherche : tel thème relève-t-il de la programmation, pour le savoir définissons la programmation (cela permet par exemple de définir quels systèmes relèvent de la programmation par démonstration) ;
6. à définir la nature linguistique des programmes : un programme est un objet conceptuel représenté par plusieurs niveaux de code ; il s'agit de savoir lequel des codes le représente le mieux [Lit99].

Ces définitions servent souvent de spécifications : nous savons, grâce à elles, ce qu'est programmer. Il suffit alors de trouver une solution, ou une famille de solutions, et de construire un système correspondant que nous pourrons tester et valider. Le problème, comme on le voit dans le tableau 1, c'est que ces définitions sont assez nombreuses et qu'elles sont assez différentes, voire parfois contradictoires.

Le terme de programmation est donc polysémique. Cela est-il étonnant, ou même gênant ? Notre hypothèse est que la reconnaissance de cette variété permet peut-être au contraire de déterminer l'espace dans lequel peuvent se définir des mécanismes spécifiques à telle ou telle forme de programmation par l'utilisateur.

1.2.1.4 Exemples de programmation par l'utilisateur.

Puisqu'il est difficile de partir d'une définition cohérente et globale et unifiant tous les aspects, on peut donner une série d'exemples de sujets pouvant nécessiter de la programmation par l'utilisateur. Ces exemples sont réels, récoltés durant 3 années soit dans les newsgroups (bionet.software par exemple), soit dans les messages envoyés au support technique du service d'informatique scientifique de l'Institut Pasteur, soit dans des entretiens avec des biologistes du campus ou pendant les ateliers de conception participative (voir le chapitre II).

Comme on peut le constater dans la liste qui suit, la nécessité de programmation peut provenir de causes de nature très différentes, et les types de programmation sont assez variables. Certains cas sont assez classiques, comme les fonctions complexes, les manipulations de données, les analyses de résultats ou les reformattages, d'autres, comme les variantes de fonctions, les fonctions trop simples ou inaccessibles, posent le problème de la granularité d'accès aux fonctions ou de leur modifiabilité, et nécessitent souvent une reprogrammation complète.

Exemples

- automatisations, requêtes complexes :
  - rechercher un motif de séquence puis retrouver toutes les structures correspondantes de type structure secondaire dans une banque ;

programmer c'est...	*mais...*
La programmation c'est l'abstraction, la généralisation : écrire un programme c'est modéliser des objets d'une manière suffisamment générale.	Programmer amène parfois aussi à spécialiser, comme par exemple aller directement dans le code pour personnaliser une partie. Écrire un pilote de périphérique, c'est l'opposé d'une généralisation. Prendre un algorithme général (programmation dynamique), et régler ses équations pour l'appliquer à un domaine spécifique (la comparaison de séquences d'ADN), c'est aussi de la programmation.
L'essence de la programmation c'est la conception d'algorithmes.	Une partie infime des programmes écrits comporte un nouvel algorithme. C'est même un lieu commun de dire que la plus grande partie des programmes n'est pas de nature algorithmique, mais est plutôt composée de "colle", d'entrées-sorties, d'interface utilisateur. Ou bien doit-on suivre une certaine idée selon laquelle la simple présence d'une boucle dénote la conception d'un algorithme ?
La programmation c'est l'automatisation : sauvegarder des instructions dans un fichier pour leur exécution ultérieure, c'est programmer.	Cela signifie-t-il que taper des commandes complexes au niveau d'un interpréteur n'est pas de la programmation ?
Programmer, en sauvegardant des instructions pour leur exécution à une date donnée, c'est aussi planifier.	Cela veut-il dire que l'utilisation des commandes "at" ou "cron" sous Unix est de la programmation (commandes pour lancer des jobs à l'avance) ?
La programmation c'est la traduction vers un niveau méta ou symbolique, comme par exemple lorsqu'on nomme un objet (c'est une référence à la dichotomie usage-mention [SUC92]).	Est-ce que définir un "alias" avec un shell Unix c'est programmer ?
Programmer c'est écrire du code.	Est-ce que l'édition d'un fichier de configuration c'est de la programmation ?
Programmer, c'est utiliser un compilateur ou un interpréteur.	Est-ce que soumettre un fichier à l'interpréteur LaTeX ou même utiliser Netscape c'est programmer ?
La programmation c'est la construction d'un logiciel ou de composants logiciels.	Les programmeurs professionnels construisent-ils si fréquemment des logiciels ou des composants ?
Un programme, c'est la représentation statique de quelque chose de dynamique [LF95].	La conception et l'implémentation d'une base de données, n'est-elle pas de la programmation ?
Programmer c'est déléguer [Rep93a].	Programmer c'est contrôler [Rep93a].
Programmer c'est chercher à mieux définir un problème [Rep93a][Nar95].	Programmer c'est implémenter des solutions.

TAB. 1.1 – Définitions de la programmation

  - construire une amorce d'après un alignement multiple : cela consiste à écrire un script qui lance un programme d'alignement, lance un autre programme qui en extrait le consensus, puis un troisième qui construit l'amorce à partir de ce consensus (une amorce est un morceau de séquence d'ADN qui sert à démarrer la construction d'une séquence par appariement avec une autre ; ces séquences doivent être à la fois suffisament spécifiques pour ne pas s'apparier avec n'importe quelle partie de l'autre séquence, mais aussi suffisamment génériques pour tolérer de légères différences) ;
  - donner la liste des numéros d'accession de toutes les séquences d'une banque qui contiennent au moins 12 répétitions de CA, et le moins possible d'ALU ;
  - chercher des EST provenant de certains tissus, et identifier ceux d'entre eux qui correspondent à des gènes connus ; puis rechercher dans une banque bibliographique (PubMed) ou autre si ce gène est associé à un facteur de transcription ; si possible automatiser cette stratégie pour ne pas se "brûler" les yeux sur l'écran ;
  - recherche de tous les CDS (régions codantes) des séquences d'ADN correspondant à des séquences protéiques spécifiées d'après leur identifiant dans un fichier ;
  - recherche des consensus d'un contig : il s'agit d'une procédure qui ouvre chaque banque données dans une liste, y ajoute les résultats d'un assemblage et cherche le consensus du contig s'il est unique ;
    voici par exemple le script pour le faire avec un logiciel interactif et scriptable, Staden [DS91] :

```
set io [open_db -name foobar -version 0 -access rq -create 1]
assemble_new_contigs -io $io -files {read1 read2 read3}
set num_contigs [db_info num_contigs $io]
if {$num_contigs > 1} {
get_consensus -io $io -contigs contigname -outfile filename ...
}
```

  - prendre des séquences dans un fichier Word, les convertir en format FASTA, puis lancer des Blast sur le Web ;

- fonctions tellement simples qu'aucun logiciel ne les réalisent (sauf des commandes Unix) :

  - trouver le nombre d'occurrences d'un caractère ou d'une suite de caractères dans un fichier texte ;
  - chercher dans une séquence les caractères autres que ACGT : c'est tellement simple (à programmer) qu'il faut ... le programmer si les logiciels dont on dispose ne le font pas ;
  - aligner la première séquence d'un fichier avec toutes les autres ;
  - rechercher le meilleur hit d'une séquence sur chaque banque dans un rapport Blast : un rapport Blast contient les "hits" d'une séquence requête dans une ou plusieurs banques de séquences, mais ces hits sont classés de manière globale à toutes les banques ;

- fonctions plus complexe mais inexistantes, ou absentes des logiciels courants :

  - générer toutes les séquences de protéines qui ont un poids moléculaire donné ;
  - détecter des oligos ayant une fréquence particulièrement forte dans une séquence d'ADN (une réponse donnée dans le newsgroup était d'utiliser les "sequence landscape") ;

- contrôle plus fin d'une fonction par l'utilisateur :

  - pouvoir contrôler l'ordre dans lequel les alignements par paire sont réalisés dans un alignement multiple, ou bien pouvoir indiquer des groupes ;

- fonctions présentes car nécessaires au calcul, mais inaccessibles dans l'affichage ou les résultats :

  - pourcentage d'identités entre toutes les paires de séquences dans un logiciel d'alignement ;
  - calculer le nombre effectif de codons dans des génomes pour pouvoir les comparer : une banque

existe avec ces informations, mais seulement pour certaines espèces : on a donc besoin de la procédure de calcul ;

- fonctions de formattage ou de présentation :

  - diviser un fichier selon un motif ;
  - reformatter un fichier PHYLIP (alignement de séquences) pour avoir le nom de la séquence sur chaque ligne (et non seulememt au début du fichier) et en enlevant les blancs entre les séquences (exemple réalisé avec awk) (reformattage nécessaire pour travailler avec le format d'un autre programme) ;
  - regrouper des colonnes par 3 ;
  - renuméroter une séquence de, par exemple, -3000 à +500 au lieu de 0 à 3500 ;
  - insérer une ligne blanche dans toutes les séquences dans un fichier de 20 000 séquences ;
  - visualiser une séquence d'ADN en spécifiant le nombre de bases par ligne et voir la traduction en protéine, avec une numérotation indépendante ;

- variantes de procédures existantes :

  - aligner des séquence avec des coûts *non linéaires* d'extension de gaps ;
  - aligner des séquences qui sont *dans le sens inverse* (il faut en inverser l'une des deux) ;
  - rechercher des motifs dans une séquence *en évitant les motifs répétés* ;
  - trouver dans une banque les séquences dans lesquelles une amorce *n'est pas trouvée* ;

- fonctions simples mais n'existant pas *simultanément* dans le même logiciel : la question posée est donc : existe-t-il *un* logiciel qui fait tout cela ; cela pose la question de la combinaison des logiciels, avec des formats de données différents ; la solution est en général d'éditer le résultat :

  - un programme qui calcule une séquence consensus et en même temps donne la variabilité de chaque acide aminé sur les positions ;
  - avoir la traduction de la séquence consensus en même temps que la séquence consensus elle-même ;

- analyser et extraire une sortie de programmes, ou des entrées de banques peu courantes (les exemples sont légion) ; ou analyser la différence entre les résultats de deux programmes similaires (ou de deux serveurs Web) ;

- manipulation de données :

  - additionner les valeurs de deux colonnes dans un fichier tabulé (dans l'exemple, pour additionner des substitutions synonymes et des substitutions non-synonymes pour avoir les mutations totales) (en awk également) ;
  - boucher un trou dans une série de numéros et remplacer des blancs par des tabulations dans un fichier d'entrée de banque (PDB) ;
  - compter le nombre de barres contigües (signe de d'égalité entre 2 séquences alignées) dans un alignement ;
  - j'ai une liste de fichiers sous cette forme :

    ```
    01-truc
    02-machin
    03-schtroumph
    ...
    ```

    et je voudrais me débarrasser du chiffre et du tiret qui précède chaque nom, comment puis-je faire ?
  -

    ```
    J'ai deux tableaux, 1 liant l'ensemble de mes individus à une série
    ```

```
de données A et l'autre liant une partie de mes individus à une série
de données B. Existe-t'il un moyen sous excel de générer
automatiquement (plus ou moins) un nouveau tableau liant l'ensemble
des données aux données A et B ?
```

- extension des paramètres :
  - créer une nouvelle matrice dans un programme où les matrices sont décrites en C (Clustalw) ;
    1. écrire soi-même un nouveau fichier en C
    2.
       ```
       To create a distance matrix in ClustalW you proceed as follows. Firts
       import or create a multiple alignment in Clsutal. When done, go to the
       menu item 4. "Phylogenetic tree". Select "Exclude positions with Gaps =
       ON "and "Correct for muliple substitutions = On" if you want a PAM
       matrix or select "OFF" in the case you want the distances expressed as
       simple percentages. Now go to the menu item 6. "output options format"
       and select 3. "Toggle Phylip distance matrix output = ON". Now go back
       to the Phylogenetic tree menu and and select 4. "Draw tree now". In your
       directory you will find a file with the suffix ".dst", which is your
       distance matrix. If you also selected "Toggle Phylip tree format =ON"
       you will also find a tree file in your directory as well.
       ```

Ces exemples *réels* montrent, si cela était nécessaire :

- que le paramétrage, si fin soit-il, si graphique ou interactif soit-il, ne peut pas suffire à traiter toutes ces sortes de problèmes ;
- qu'il ne s'agit pas seulement de pouvoir programmer la fonction, mais qu'il faut pouvoir l'intégrer dans un ensemble, et que cela n'est pas le moins difficile, car cela dépend de la manière dont est construit cet ensemble ;
- que les domaines sémantiques sont trop variables pour réaliser les solutions au sein d'un système unique de programmation spécialisée ;
- qu'un logiciel unique *à moins de comporter une fonction générale de programmation* ne peut pas intégrer tous les besoins.

On voit également dans ces exemples que ce qui est *difficile* , ce n'est pas du tout l'aspect algorithmique (aucun des exemples ci-dessus, poutant largement hors de portée de la plupart des biologistes, ne comporte de problèmes algorithmiques, à une exception près pour les coûts non-linéaires de gaps). La difficulté, pour le biologiste, vient plutôt de l'environnement et de la manière dont les logiciels sont conçus, qui les rend insuffisamment flexibles.

1.2.1.5 Les situations de programmation.

On peut également observer que la nature de l'activité de programmation varie beaucoup selon les situations et les objectifs. Du "Just-in-time programming" [Pot93a], à l'écriture d'un logiciel scientifique ou commercial, il y a des variations...

- Manipulation de données, formattage pour la préparation d'une étude.
- Visualisation graphique de résultats.
- Automatisation, écriture de scripts.
- Résolution de problèmes, modélisation, exploration d'alternatives.
- Formation, cours de programmation : exercices, projet.
- Écriture d'un logiciel d'analyse pour le laboratoire.
- Réalisation d'une base de données scientifique.
- Implémentation d'un algorithme associé à une publication.

On comprend par exemple qu'il y a sans doute une grande différence entre les problèmes et les objectifs d'un novice apprenant à programmer et d'une personne ayant besoin d'écrire des lignes de code pour

une utilisation avancée mais occasionnelle d'un logiciel complexe. Pour le premier cas, on fera plutôt appel aux modèles cognitifs, à la psychologie de la programmation, aux outils de la pédagogie, aux techniques de visualisation de programme, ... alors que pour le second, on essaiera de concevoir des outils les plus adaptés possibles à son contexte de travail, en améliorant autant que possible dans ce sens l'environnement d'utilisation et les outils de navigation (quoique que la pédagogie bénéficierait aussi d'une approche customisation). De même, le contexte professionnel dans lequel on programme a une grande importance (figure 2).

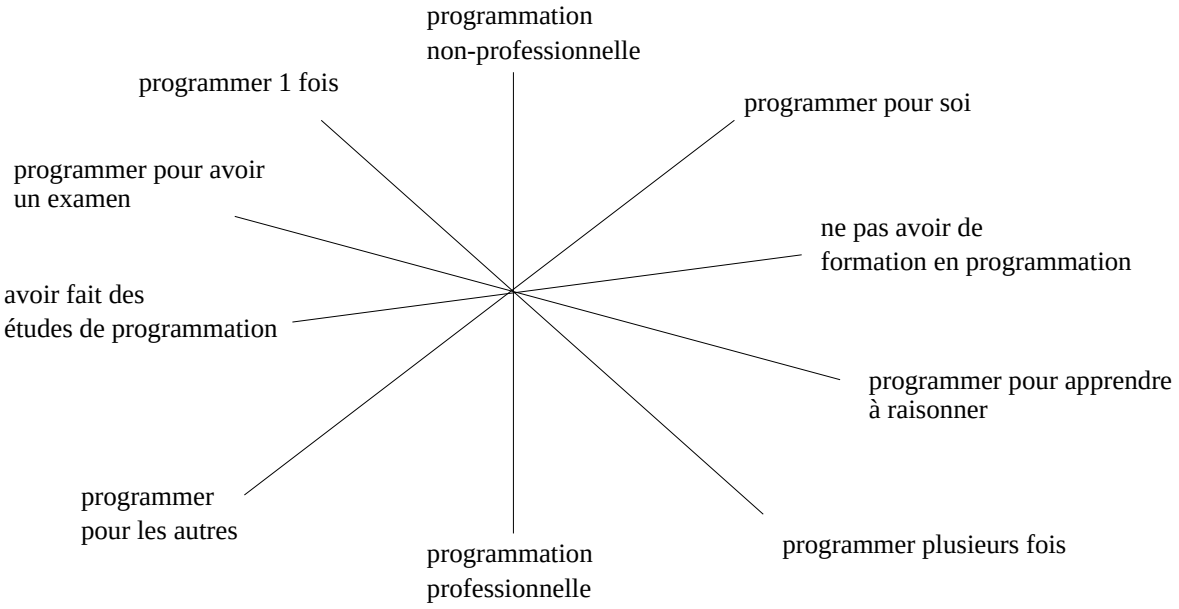


FIG. 1.2 – Situations de programmation.

Il convient cependant de reconnaître que ces différences ne sont pas nécessairement tranchées. [Nar95] évoque l'idée d'un continuum, tout comme [Eis97], selon qui les différences entre professionnels et non-professionnels sont un peu surestimées dans la littérature. Cela se constate à maintes reprises dans le domaine de la bio-informatique, où il est souvent difficile de dire si un bio-informaticien est biologiste ou informaticien d'origine. Il n'est pas rare de voir un biologiste devenu informaticien se voir confier la responsabilité informatique (matériel, logiciel, support technique, développement) d'un laboratoire, et les logiciels scientifiques sont bien souvent écrits par des biologistes de formation.

1.2.1.6 Programmation et interaction, programmer et utiliser.

S'il est difficile de définir la programmation en soi, peut-être pouvons-nous la définir par rapport à autre chose ? Peut-être est-il plus facile de dire ce qu'elle n'est pas ? Voyons comment la programmation peut s'opposer à l'interaction.

Indépendamment de toute incarnation dans un contexte d'utilisation ou dans des techniques d'interaction, y a-t-il une différence de fond entre programmer et interagir qui ferait que l'un est irréductible à l'autre, voire même que ce serait cette limite qui les définirait l'un et l'autre ? On peut dire en effet, que, par définition, il y a toujours au moins deux langages, l'un définissant l'autre. Ainsi, dans un logiciel interactif, il y a le langage de l'interaction et il y a le langage dans lequel est écrit le logiciel, qui décrit et construit la machine capable de comprendre le langage d'interaction. Mais c'est vrai aussi pour les machines non interactives : il y a par exemple le langage dans lequel est écrit le compilateur, et le langage analysé par le compilateur. Ce n'est donc pas cela qui détermine une différence entre interagir et programmer, qui se situent plutot sur un continuum.

On peut introduire de la programmation (écriture d'une instruction) dans un outil interactif (question/réponse) mais elle fait alors partie d'un langage interactif. Ce n'est donc pas toujours non plus la **forme de l'expression** qui définit la différence, mais son **mode d'utilisation** . Le mode interactif

serait discontinu et non déterminé d'avance. Dans le mode programmation, tout doit être déterminé d'avance. C'est le domaine de la définition et de la modélisation. Pourtant dans un tableur, les formules sont entrées et évaluées de manière interactive : le mode est bien discontinu, mais cependant, il ne fait pas de doute qu'il s'agit bien de programmation.

Enfin, il ne faut pas assimiler interaction et manipulation directe. Une interaction peut être effectuée de manière indirecte, par exemple en utilisant un logiciel piloté par une interface conversationnelle.

N'y a-t-il pas en réalité, comme on le voit dans la figure 3, plusieurs axes à distinguer :

- *mode* : l'axe programmation-interaction, sur lequel on décrit la modalité d'utilisation (dialogue riche ou traitement continu) - c'est sans doute à cet axe que correspond la distinction systèmes visuels compilés - systèmes visuels interprétés de [Mye90],
- *expression* : l'axe décrivant le type d'expression (texte, graphique ou plus exactement : manipulation directe),
- *tâche* : l'axe programmation-utilisation qui parle de l'objectif de la tâche effectuée, mais aussi de la limite entre l'interface et l'application sous-jacente, ou de la limite entre le processus d'écriture d'un logiciel et le processus d'utilisation, et qui permet de décrire à quel point l'un peut intervenir sur l'autre. [Mør97c] définit le tailoring comme une forme d'utilisation se dirigeant sans rupture vers une réelle programmation.

On remarque qu'il y a une continuité sur l'axe mode - comme on l'a vu avec la figure 1, de même que sur l'axe expression - il y a des transitions entre les langages et environnements purement textuels et ceux qui sont complètement graphiques, comme on le verra plus loin. La continuité sur l'axe tâche est-elle également envisageable ? C'est une des questions que nous allons poser.

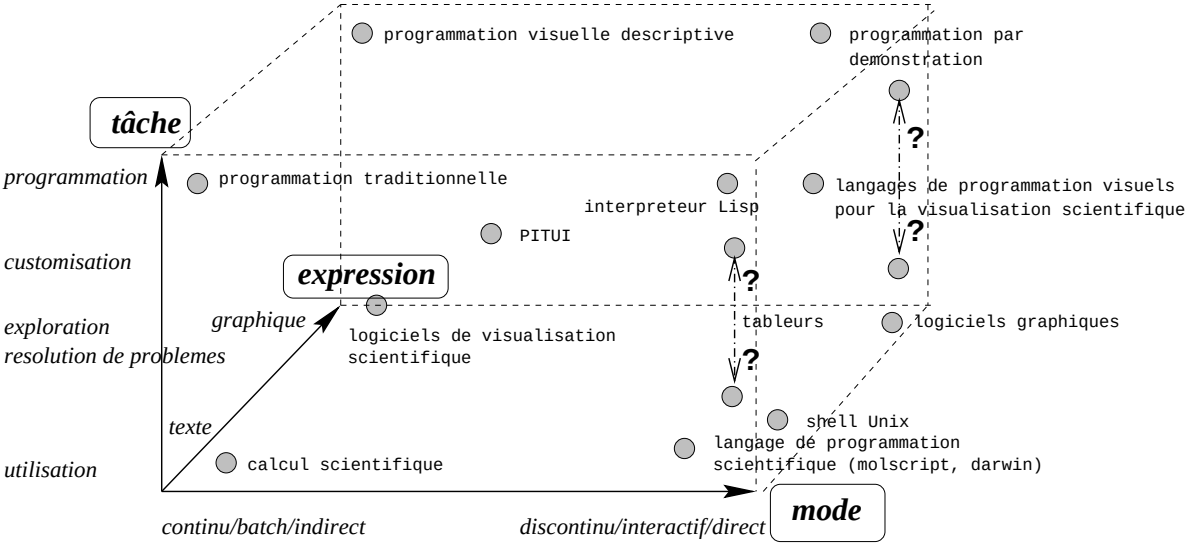


FIG. 1.3 – Axes programmation-interaction.

La nécessité de la différenciation de ces axes peut être illustrée par quelques exemples.

- L'écriture de commandes shell un peu élaborées présente un cas d'utilisation (axe tâche) sous forme textuelle (axe expression) et interactive (axe mode) :

```
for f in `prog1 | grep xx` do
    cat $f | awk 'if {$1 == "0"} {print $2}'
done
```

  Il s'agit bien d'une interaction, puisque c'est une ligne de commande du shell, mais exprimée dans un langage (textuel) de programmation. C'est plutôt de l'utilisation, dans le contexte du shell, parce qu'on ne construit pas nécessairement un programme. Et pourtant c'est la spécification d'une itération, d'un test, comportant des variables (même si ce sont des variables automatiques

de la commande awk) qui sont souvent les critères descriptifs de la programmation.

- A l'inverse, la spécification d'un programme peut prendre un caractère non textuel et interactif, comme c'est le cas dans les systèmes proposant la programmation par démonstration. D'une part le programme démontré est issu d'une suite d'actions interactives, et d'autre part, comme on l'a déjà vu, une procédure d'inférence peut être complétée par un dialogue avec l'utilisateur. Là où cela devient plus ambigu, c'est que le système avec inférence ou enregistrement de programme fonctionne alors même que l'utilisateur n'en est pas conscient : dans ce cas des interactions "naturelles" servent en même temps pour l'utilisation "normale" et pour la construction de quelque chose car l'on se trouve alors à deux positions simultanément sur l'axe tâche. Cela est un peu différent de l'action de *démontrer* , plus consciemment constructive.

- Enfin, il y a aussi de l'interactivité dans certaines formes de programmation incrémentale, comme dans les tableurs ou dans Self [CUS95] où on programme par interaction graphique, ou bien tout simplement en essayant des morceaux de code dans un interpréteur. Peut-être le fait que le résultat de toutes les interactions soit conservé et accumulé, qu'il y ait une forme d'anticipation et de pensée à moyen ou long terme, joue-t-il un rôle important dans la question de savoir si oui ou non on est en train de programmer, apprendre à programmer, ou utiliser [GW98].

Les deux types de logiciels décrits dans la figure 1 : systèmes de visualisation post-traitement et langages de programmation visuels, se situent dans le graphe de la figure 3 à l'opposé à la fois de l'axe mode et de l'axe tâche. Ils sont à l'opposé de l'axe mode car les premiers fonctionnent en continu et les seconds permettent à l'utilisateur d'intervenir à toutes les étapes de l'analyse, par exemple en posant des contrôles sur des variables ou des moniteurs sur certains résultats intermédiaires, ou même en programmant des tests par des techniques de manipulation directe. Ils sont aussi éventuellement à l'opposé sur l'axe tâche, car les définitions de contrôle ou de points d'arrêts peuvent être conservées d'une session à l'autre, et constituent par conséquent une customisation, une adaptation constructive du logiciel à une situation particulière.

Ce qui complique notre raisonnement, c'est que l'opposition réelle ou théorique entre programmer et utiliser ou interagir s'est historiquement concrétisée dans une discontinuité entre les outils de programmation et les outils applicatifs, reflétant une discontinuité entre les utilisateurs des uns et des autres. Mais la progression de la programmation par l'utilisateur final montre qu'il y a de plus en plus un continuum au niveau des capacités techniques, des types de tâche et des structures organisationnelles.

1.2.2 Répartir le travail

Mais cette discontinuité n'est pas nécessairement à rejeter. En effet, comme le remarque [Fis99], les systèmes informatiques sont aujourd'hui trop complexes pour être conçus par une seule profession. D'un côté, on épargne bien sûr à l'utilisateur la fabrication des parties du logiciel qui sont éloignées de son domaine de compétences et d'intérêt, mais de l'autre, on reconnaît l'utilité de lui laisser l'opportunité d'intervenir dans les composantes qui relèvent de son expertise. Cela peut se présenter sous plusieurs formes :

- Dans le type d'architecture et le modèle de calcul : certains modèles comme les systèmes experts tout comme de nombreux systèmes déclaratifs supportent explicitement cette complémentarité.
- Dans le processus de conception : dans ce type de démarche [Gre93], on essaye d'intégrer les utilisateurs au cours des étapes de conception, afin d'abord que les outils soient mieux adaptés à leurs besoins, voire aussi pour établir les conditions d'une meilleure créativité sur le plan de la conception ou même de l'innovation technologique (les tableurs n'ont-ils pas été inventés par un non-informaticien ?).

On verra que les systèmes personnalisables (au-delà de la simple adaptation d'attributs de présentation) relèvent parfois des deux catégories : ils supportent explicitement et techniquement la contribution de l'utilisateur dans la construction et la finalisation de l'outil et s'intègrent souvent dans une démarche de conception coopérative.

1.2.2.1 Modèles de calcul pour la répartition du travail.

On peut proposer deux façons de poser la question des manières de répartir les tâches entre informaticiens professionnels et utilisateurs :

- Quelles sont les parties du logiciel ou du processus de construction qu'il serait approprié de rendre accessibles à des non-informaticiens et quels sont les problèmes soulevés ? La réponse à cette question dépend bien sûr du domaine : certains modèles de calcul sont plus adaptés à certains types de tâche.
- Quels sont les aspects de la construction d'un logiciel qui sont inaccessibles aux non professionnels ?

Admettons que si c'est la première question - qui présente pourtant le problème de manière positive - qui devrait seule déterminer les solutions, c'est parfois le deuxième aspect qui l'emporte dans les fonctions effectivement proposées : on limite l'accès à telle partie de l'outil parce que ce serait techniquement inabordable. On décide par exemple sans étude préalable que la ré-implémentation d'un composant n'est pas à la portée de l'utilisateur simplement parce qu'il aurait à écrire du code. Mais c'est parfois à ce niveau précisément que l'expertise de l'utilisateur peut être utile, comme c'est le cas en biologie où c'est souvent dans les décisions fines prises au niveau de l'heuristique d'un algorithme que réside la sémantique d'une méthode. Ce type de limitation relève d'autre part souvent d'une acception générique des difficultés de la programmation qui ne tient pas toujours compte des variations entre les utilisateurs. Nous citerons quelques-unes des études concernant les difficultés spécifiques des différents modèles de calcul dans le chapitre II.

Quels sont donc les tâches qu'il serait approprié de rendre accessible à l'utilisateur ?

- Une partie souvent accessible à la programmation par l'utilisateur est la *construction de l'interface utilisateur* . Ce sont souvent des ajouts dans les menus, la création de boutons, la définition de raccourcis, le placement, l'affichage ou l'édition de variables, ainsi que l'édition des attributs de présentation. C'est aussi la composition d'interface graphique à partir de boîtes à outils de widgets. Il nous semble qu'il ne faut pas minimiser cet aspect. Les constructeurs d'interface comme Visual-Basic sont rejetés avec mépris du champ d'investigation des chercheurs en programmation visuelle (ici à juste titre puisqu'ils n'ont rien de visuel) mais aussi en programmation par l'utilisateur final. Et pourtant, ils remportent un très grand succès auprès des non-informaticiens. Pourquoi ? Une des raisons ne pourrait-elle pas être qu'ils libèrent le programmeur de l'écriture d'une partie à la fois familière et fastidieuse de l'application [PRM00] ? On peut se dire à cet égard qu'un environnement de programmation par l'utilisateur doit pouvoir fournir ce type d'objets, et peut-être ne pas imposer une interface, surtout si elle est générique, comme le font certains environnements de programmation visuelle (à base de diagrammes dataflow tout au moins) [Rep93a].

- Une autre est l'écriture de *scripts* combinant des fonctions de l'interface utilisateur ou construites dans le langage interactif (comme avec la programmation par démonstration à partir d'actions interactives).

- Certains systèmes permettent l'écriture d'une partie des *procédures ou fonctions* qui servent de paramètres au système : c'est le cas des tableurs ou des systèmes experts. C'est le principe de la fonction paramètre ou de certains patrons de conception, comme la stratégie [GHJV95]. Par exemple, les procédures de tri permettent souvent de passer la fonction de comparaison de deux éléments en paramètre, et les handlers d'évènements permettent d'associer une commande à un bouton ou à un clic de la souris [PRM00].

On peut se poser la question de manière inverse : quels sont, selon les systèmes, les composants du logiciel qui sont *épargnés* à l'utilisateur (tableau 2) ?

Tout comme les langages spécialisés, l'intérêt des modèles présentés dans le tableau 2 est d'avoir déjà réalisé une partie importante du code, notamment la partie difficile qu'est le contrôle.

Système	*Composants épargnés à l'utilisateur*
Tableurs	interface utilisateur, contrôle, variables
Constructeurs d'interface	programmation graphique (le système fournit les widgets standards)
Programmation par démonstration	le contrôle (si inférence ou si à base de règle), la traduction action-instruction
Programmation orientée domaine	bibliothèques, composants réutilisables
Langages de prototypes	la modélisation
Langages à contraintes	le contrôle
Langages dataflow	le contrôle (si s'agit de dataflow à gros grain)
Programmation évènementielle	le contrôle [PRM00]

TAB. 1.2 – Composants du logiciel épargnés à l'utilisateur

Ces modèles sont certainement plus ou moins adaptés à un domaine, comme le dataflow pour les domaines à fort traitement ou manipulation de données, les agents pour les environnements de simulation et les jeux, les contraintes pour les bases de données, les systèmes experts, ou les constructeurs d'interface graphiques [MMM99].

La programmation déclarative.

La programmation déclarative est une forme de spécification qui s'intègre assez bien dans l'idée de système interactif. Pour [Weg98], les modèles interactifs sont d'ailleurs de nature déclarative tout comme les modèles logiques, mais ils remplacent l'interprétation de formules par des propriétés du monde. La programmation déclarative est une technique qui permet de répartir la programmation entre le développeur du système et son utilisateur, en donnant pour tâche à l'utilisateur de définir ce qu'il veut en termes de résultats (le quoi), et en laissant la description de la partie contrôle permettant de réaliser ce résultat (le comment) au concepteur du système. Dans le modèle dataflow, ce sont les données et les relations de dépendances entre elles que l'utilisateur doit décrire ; dans le modèle logique, ce sont des prédicats logiques. La programmation par démonstration, lorsqu'elle sait faire de l'inférence, peut également être classée parmi les techniques déclaratives, par opposition aux techniques procédurales [Gir00], puisque ce qu'on en attend c'est que l'algorithme soit déduit d'une suite de faits descriptifs (actions, états, données, ...). Correspondent également à ce type de programmation déclarative : les tableurs, les techniques de programmation à partir de règles de production condition-action [SCT00][OFL97][BRL91][Rep93b] ou celles qui fonctionnent sur la définition de contraintes, comme [Bor81] ou [MGD+90]. Il reste bien sûr qu'on peut encore différencier les techniques déclaratives selon le mode de spécification : soit l'utilisateur sélectionne parmi des possibilités pré-définies, soit il construit lui-même la spécification (règle logique, formule, ...).

De manière générale dans ces techniques, c'est toujours le contrôle, la définition procédurale, qu'on essaye d'éviter à l'utilisateur. Ces systèmes ont pour avantage de permettre une spécification de l'application à partir de *descriptions indépendantes* les unes des autres, données *dans un ordre quelconque* . Il est beaucoup plus simple de définir des informations locales à de petits composants en comptant sur les moteurs internes du langage pour en vérifier la cohérence, que de définir un système de contraintes au contrôle complexe dans son entier [AGK+97].

Mais déléguer la *spécification du contrôle* ne devrait pas signifier qu'on retire le *contrôle de l'exécution* à l'utilisateur. Décider quand est invoqué tel ou tel fragment du code définirait les différents degrés de l'échelle interaction-programmation selon [Rep93a] : la manipulation directe ne délègue rien et permet à l'utilisateur de contrôler complètement ces invocations, tandis que le fonctionnement batch consiste à déléguer complètement le contrôle au programme. La solution de théâtre participatif que [Rep93a] propose, qui consiste à laisser l'utilisateur intervenir de manière asynchrone sur des agents munis d'un script, fait partie de ces solutions intermédiaires, qui consistent à déléguer au système une partie des éléments de contrôle, tout en laissant la possibilité à l'utilisateur d'intervenir selon un mode ou un autre, différents selon les modèles de calcul.

1.2.2.2 Paramétrabilité.

Comme le montre la figure 4, on peut aller plus loin dans cette réflexion et se dire que la répartition des tâches de spécification peut-être mesurée selon un degré plus ou moins grand de paramétrabilité [Mør97a].

- (a) représente les logiciels "algorithmiques" qui traitent une entrée et produisent une sortie, en proposant un paramétrage initial ; dans ce cas, la surface de paramétrabilité est assez étroite ;
- (b) les tableurs, avec les formules, définissent au contraire une surface de paramétrabilité importante et complexe, mais limitée à un modèle de calcul imposé (fonctionnel) ;
- (c) avec un compilateur ou un interpréteur classiques, qui n'imposent pas d'autre modèle que celui de Turing, la part de paramétrage est très grande, tous les comportements sont possibles ;
- (d) un framework laisse un espace de spécification important, mais dans le cadre d'une architecture pré-établie ;
- (e) les interactions proposées par un logiciel interactif définissent un espace de paramétrage qui peut être important lorsque le logiciel n'impose pas d'ordre dans les actions.

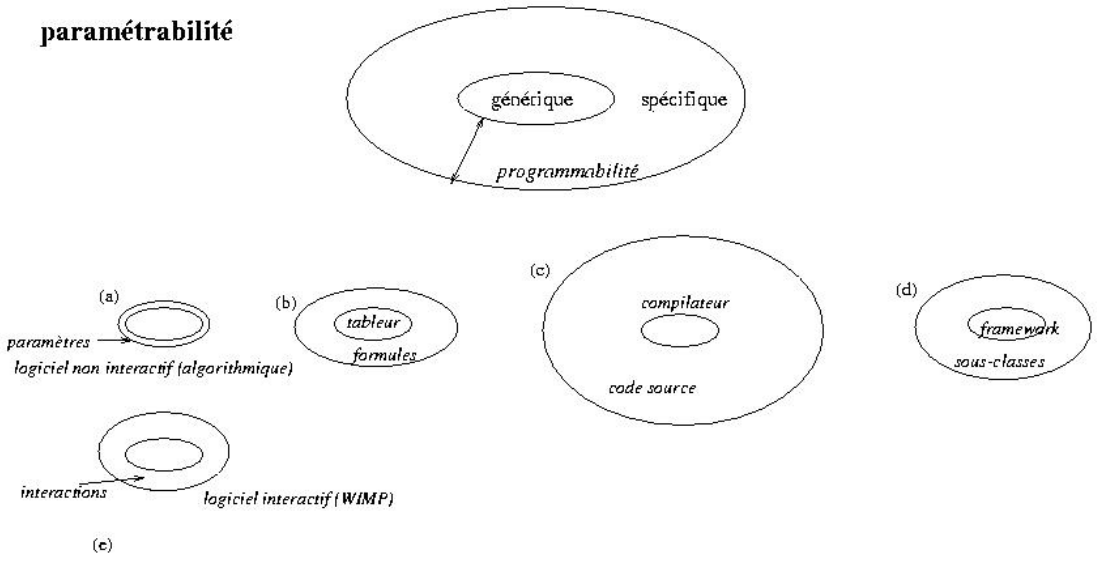


FIG. 1.4 – Paramétrabilité.

Un système spécifique programmable dans l'interface, à la fois extensible et modifiable peut donc fort bien être pensé sur le même modèle que tout système réalisable à l'aide d'un interpréteur ou d'un compilateur ((c) dans la figure 4). Le système Elody [OFL94], qui utilise la structure Lisp de liste à la base aussi bien des données que des fonctions, montre l'importance de la structure interne du programme pour la paramétrabilité, pour permettre l'abstraction. Pour opérer une abstraction, dire : *lambda(x)* à partir d'un corps de fonction, il faut pouvoir isoler le paramètre x . Ce mécanisme ne marche pas avec n'importe quel élément ou partie d'un code, comme par exemple avec une structure conditionnelle dans son entier : seule la condition ou l'une des deux branches peut être paramétrée, la structure elle-même ne l'est pas. C'est l'objet de [GHJV95] ou [Pre94] d'identifier les structures, du moins dans l'approche objet, qui peuvent faire l'objet d'un paramétrage, et de décrire quelles sont les mécanismes qui permettent de l'implémenter.

Les possibilités et les choix techniques qui offrent diverses possibilités d'articulation entre les informations fournies par l'utilisateur et le système lui-même sont abordées de manière plus détaillée dans le chapitre III.

1.2.3 Sur quels aspects doit-on mettre l'accent ?

La section précédente a essayé de montrer que c'est moins la programmation par l'utilisateur en elle-même qui constitue un but en soi que l'idée générale d'une place aménagée pour les spécifications de l'utilisateur dans les systèmes informatiques. Cette place peut être définie en vertu de critères de difficulté technique, par des techniques de programmation adaptées, ou par une forme plus ou moins complexe de paramétrisation, ou encore, indépendamment de ces critères, par le choix d'une méthode de conception et d'un modèle de calcul adapté au domaine et au contexte d'utilisation.

Mais la situation n'est pas si simple. On peut en effet considérer la programmation comme une forme poussée d'utilisation, ou comme une participation possible à la construction du logiciel, c'est-à-dire comme une activité dont on ne cherche pas à définir une nature universelle, et qui se présentera peut-être sous des aspects différents selon le contexte et les techniques du domaine et du type d'utilisateur. Dans cette optique, la programmation par l'utilisateur final signifie tout d'abord : programmation par l'utilisateur *de quelque chose* , à savoir le logiciel dont il est l'utilisateur. Ce n'est pas la même chose que la programmation en général, destinée à tout le monde.

Mais on peut aussi s'intéresser à la programmation en elle-même, et éventuellement l'adapter à des utilisateurs non-professionnels, au nom par exemple d'une culture informatique générale que toute personne devrait posséder [DiS99].

Cette ambiguité est assez manifeste dans le domaine de la bio-informatique. La programmation sous une forme ou une autre constitue une forme de spécification scientifique de premier plan qui, en général, est confiée à des bio-informaticiens, c'est-à-dire souvent à des biologistes ayant appris l'informatique et la programmation beaucoup plus qu'à des informaticiens d'origine. Elle est donc un *moyen* indispensable pour la réalisation effective de la recherche en biologie *par des biologistes* . Mais, pour des raisons socio-professionnelles, elle est aussi devenue un but en soi (en même temps qu'un rejet en soi) : maîtriser le langage informatique est devenu aujourd'hui un atout professionnel significatif soit pour obtenir un meilleur salaire soit tout simplement pour trouver un travail. Il n'est pas étonnant qu'elle provoque aussi un rejet de la part de nombreux biologistes qui y percoivent un changement de nature de leur activité, alors que, une fois de plus, elle ne devrait constituer qu'un moyen supplémentaire, si ce n'est normal, du moins normalement supporté, d'utilisation des logiciels.

Les deux dernières sections présentaient les approches de programmation par l'utilisateur et de répartition du travail parce qu'il nous a semblé qu'elles constituaient les objectifs *implicites* de démarches techniques permettant une activité significative de l'utilisateur non professionnel de l'informatique dans le contrôle des logiciels. Il nous semble que ces objectifs peuvent être décrits comme implicites à partir de moment où ils sont établis et supposés *par défaut* . Mais il nous semble utile aussi d'expliciter plus précisément les objectifs que nous pensons être importants, indépendamment si cela est possible des aspects de complexité d'utilisation et de conception logicielle. Ces objectifs se partagent entre :

- La programmation comme une forme évoluée d'interaction.
- La culture informatique (computer literacy).
- La participation à la construction des systèmes.

1.2.3.1 La programmation comme une forme évoluée d'interaction.

Nous avons évoqué ce type d'utilisation au début du chapitre : il recouvre toutes les approches qui permettent à l'utilisateur d'utiliser un logiciel comme un medium dynamique, au service de la réflexion et de l'exploration intellectuelle [Nar95], [Weg97b][DA89][Rep93b]. Que la surface décisionnelle de l'utilisateur soit augmentée par des techniques de type programmatique, pour ajouter du contrôle ou spécifier des contraintes, cela est important, mais il est également important à nos yeux que la possibilité prévue d'avance par le logiciel d'un dialogue où les résultats successifs du calcul sont itérativement améliorés par des informations données par l'utilisateur qui ne sont ni déterminables au démarrage du calcul, ni explorables par le calcul lui-même pour des raisons combinatoires. [Weg99] donne un exemple amusant de l'intérêt de ce type d'interaction pour l'obtention de meilleurs résultats de calcul (cet argument est par ailleurs formalisé dans de nombreux articles de cet auteur). Selon lui, si Ken Starr interviewe Bill Clinton, il apprendra plus par un questionnement interactif que par un formulaire

préparé d'avance (follow-up, relance, insistance). Ce que cet exemple tente de montrer, c'est que chaque interaction (action, calcul), est enrichie d'informations qui tiennent compte d'un état passé.

C'est là une idée de la programmation comme processus de résolution de problèmes, c'est-à-dire comme exploration de problèmes mal ou incomplètement définis, qui s'écarte résolument de la notion de programmation comme traduction d'un problème bien défini dans les mécanismes de la programmation [Rep93b].

Ce type d'activité est parfois compris comme du "bricolage". En somme, il se place à un niveau équivalent des croquis diagrammatiques, avec pour différence et avantage qu'il opère sur un support informatique qui oblige à une certaine formalisation [PL92]. Cette activité doit pouvoir s'accompagner d'un apprentissage indispensable des concepts, comme le soulignent notamment [BA99][LD89] en montrant que le bricolage seul ne permet pas de résoudre beaucoup de problèmes. Notre position - assez raisonnable et assez classique au demeurant, mais alors pourquoi est-ce si polémique ? se résume de cette façon : ni le bricolage ni la formation conceptuelle seuls ne constituent une solution générale pour l'apprentissage et la maîtrise de la programmation.

Ce type d'approche interactive est très important en bio-informatique, où les logiciels implémentant des algorithmes combinatoires sont la majorité, mais où la difficulté des techniques de paramétrage nécessaires à ce type de fonctionnement empêche la plupart du temps leur réalisation. Parce que cet aspect recouvre un aspect crucial de la problématique programmation/interaction, et parce qu'une grande partie des objectifs de flexibilité que nous nous sommes donnés sont de ce type - une sorte de flexiblité algorithmique - nous avons exploré cette approche dans le prototype que nous avons réalisé, et cela est décrit dans le chapitre V.

1.2.3.2 Computer literacy.

Pourquoi tout le monde ne programme-t-il pas ? Pourquoi la programmation ne fait-elle pas partie de l'utilisation courante des ordinateurs ? Est-ce un problème de difficulté technique ou cognitive ? Est-ce parce que les langages de programmation sont inadéquats ? Pourquoi les programmes que nous utilisons sont-ils des boîtes noires que nous ne pouvons pas changer et dont le fonctionnement nous est pour la plupart du temps inconnu [Dou97] ? Pourquoi le software est-il du hardware pour la grande majorité des utilisateurs ? Y a-t-il une différence de nature entre programmer et utiliser ou interagir, une difficulté particulière, une prédisposition à la programmation que tout le monde n'aurait pas [Pir84] ? Déjà, ne serait-ce que pour utiliser les nombreuses fonctions de customisation existant dans les logiciels, [Mac91a] montre que les utilisateurs sont très réticents. De même [Rep93a] montre que l'apprentissage réussi de la programmation nécessite un message très clair de l'environnement quant au compromis entre le contrôle que l'utilisateur y gagnera et l'effort qu'il devra fournir - autrement dit, tout le monde ne *veut* pas programmer, loin de là. [vR99] ou [DiS99] parlent de 'computer literacy' (Knuth) : selon ces derniers, maîtriser l'ordinateur signifiera un jour ou l'autre pour les utilisateurs savoir le programmer, du moins dans la limite de leurs besoins.

S'agit-il pour autant de la programmation *pour tout le monde* , dans un sens équivalent à l'alphabétisation [DiS99] ? S'agit-il aussi de *programmation facile* , un vieux rêve... ? Il faut peut-être distinguer la facilité (théorique) et l'accessibilité (pratique), celle des systèmes programmables, des environnements de programmation, ou même des programmes qui auront à être plus compréhensibles [vR99].

Il est important de distinguer ces deux approches de l'alphabétisation et de la programmation comme interaction, pour lesquelles les objectifs et les mécanismes d'un accès à une forme plus ou moins limitée de la programmation sont sans doute un peu différents. Mais il n'est pas exclu de les rendre compatibles : si l'apprentissage de la programmation, comme un objectif en soi, peut être facilité par une approche contextualisée dans l'utilisation d'un logiciel particulier, comme le propose [SBMP97] qui décrit un cours de programmation partant du contexte des logiciels Word et Excel, ou si, dans l'autre sens, la pratique exploratoire d'une programmation d'abord limitée à la modification d'un outil de travail peut amener à des notions de programmation suffisamment générales pour être réutilisables dans un autre contexte, c'est encore mieux. Il nous semble assez difficile d'affirmer sans raisons particulières, qui seraient dûes à des problèmes d'organisation de l'entreprise ou de commercialisation d'un logiciel, que la programmation par l'utilisateur offerte par un système informatique doive être *limitée a priori* , alors qu'on affirme par ailleurs que l'accès à la programmation pour tout le monde serait un bien.

Face à une situation où cependant ces deux objectifs de programmation pour tout le monde et de systèmes informatiques modifiables sont assez rarement atteints, il faut s'interroger sur les raisons et les remèdes :

1. La programmation est une forme de pensée difficile à comprendre, à apprendre et à enseigner : quelles sont les principales difficultés et les solutions proposées ? La programmation nécessite-t-elle un enseignement préalable [BA99] ? Cet apprentissage est-il possible dans des conditions d'utilisation où la programmation n'est pas un but en soi [CR87] ?
2. Les formes traditionnelles de programmation ne conviennent pas : en existe-t-il d'autres, soit du point de vue de la notation, soit du point de vue du niveau de langage ? Les environnements de programmation sont-ils adaptés à une forme occasionnelle programmation ? (ou à la programmation elle-même d'ailleurs [LF95])

Ces aspects cognitifs et pédagogiques sont abordés plus en détail dans le chapitre II.

1.2.3.3 La personnalisation de logiciels.

Contexte

Il est utile de se poser le problème du contexte dans lequel peut se produire l'activité de programmation. Lorsqu'on adopte ce type d'approche, comme la conception contextuelle [BH98] ou l'approche par scénarios [Car95], et lorsqu'on observe le contexte de travail des utilisateurs non-informaticiens avec les outils dont ils disposent, on constate très fréquemment ce qui est décrit dans [Mør97a] comme un problème de panne-rupture (breakdown) ou d'incident critique [Fla54]. Les outils sont limités, il faudrait pouvoir les modifier. Prenons des exemples :

1. Le logiciel de dessin permet des rotations à 30/60 et 90 degrés : que faire si on veut des rotations à 45 degrés ? ([DiG96b] fournit de tels exemples en nombre)
2. Le logiciel permet d'éditer un alignement de séquences, mais il ne sait pas afficher les acides aminés pour les codons.

Ce type de situation se produit quotidiennement avec les logiciels utilisés par les chercheurs en biologie, et il n'est pas certain que la meilleure solution soit d'enrichir continuellement les logiciels afin de prévoir toutes les situations. Ainsi, pour [BHHW99] qui décrit ORBIT, un environnement pour les outils bio-informatique, la personnalisation constitue une bonne alternative à la conception d'interfaces multi-fonctionnelles, et un logiciel *flexible* est bien supérieur à un logiciel *riche en fonctions* , surtout dans un domaine où les utilisateurs sont très différents. Comme le montre [Eis97], nous avons le choix entre deux sortes de complexités : celle de la programmation et celle de la "profusion". Aucune tâche n'est en effet impossible à réaliser en manipulation directe, mais cela peut demander tellement d'interactions ou la disponibilité d'un nombre tellement important de fonctions dans le logiciel, qu'on arrive à une complexité, différente, mais finalement équivalente à celle de la programmation.

Le problème que nous nous posons peut être reformulé autrement : il faut que l'utilisateur puisse modifier le système informatique, mais d'une façon qui n'est *pas nécessairement prévue par ce système* . Le problème qui se pose donc c'est de savoir comment faire pour prévoir ces besoins par définition imprévus.

Ce qui nous intéresse également, d'un point de vue plus général, dans cet aspect de la programmation par l'utilisateur, c'est qu'il correspond à l'idée qu'un logiciel ne soit pas toujours un produit achevé, mais un medium évolutif [FG90], [DB98b], à la programmation duquel l'utilisateur peut participer - dans le prolongement direct des techniques de conception participative ([SN93]). S'il est important qu'un logiciel soit bien adapté aux tâches des utilisateurs, [NJ94] remarque qu'un logiciel trop précisément adapté peut par là même être trop déterminé et fermé à la créativité. Pour [BD95] c'est en étant un peu sous spécialisé, non fini, qu'un système permet l'utilisation la plus riche et la plus inventive ; et c'est à l'utilisateur, à travers des mécanismes d'adaptation correctement conçus, qu'il revient s'il le veut de "finir" le produit. Peut-être cela éviterait-il au logiciel cet aspect d'idéalisation de la procédure remarquée par [Dou97] qui rend les logiciels pénibles voire impossibles à utiliser : les logiciels sont en effet une forme informatisée de pratiques de travail, mais à travers l'informatisation, des décisions ont été prises, qui visent souvent à introduire des règles peu ou pas pratiquées dans la réalité, ou qui éliminent un certain nombre de cas qui, selon l'utilisateur ou le moment pouvait avoir une grande

importance. C'est donc aussi ce type de limites que tentent de faire disparaître les solutions du type de celle proposée par [Mør97b], qui consiste à concevoir le logiciel comme un outil générique évolutif qui peut être spécialisé par ses utilisateurs, depuis la définition de préférences jusqu'à la programmation de nouvelles classes et de nouvelles fonctions.

Définitions

Il existe plusieurs définitions de la personnalisation (ou : customisation, tailorabilité, adaptation). Selon [Mac91b] customiser c'est changer le logiciel de manière persistente sans écrire de code. [HK91] parle de customisation dès lors qu'il s'agit de modifier l'outil lui-même, ou d'effectuer un changement imprévu ; par opposition à une action dont l'effet est immédiat, il caractérise aussi une action de customisation comme ayant un effet *différé* .

Ce qui apparaît de manière constante dans les différents points de vue c'est l'effet persistant de la modification, et le fait que c'est le système lui-même qu'on modifie (on n'en construit pas un autre), ce qui nécessite parfois une discussion sur la sémantique de ces changements : est-ce toujours le même système si on a ajouté des fonctions ou modifié le comportement de l'une d'entre elles ? D'autres aspects importants mais sujets à discussion sont :

- La portée des modifications : [Cyp91a] inclut les préférences, les templates (feuilles de style) et les macros alors que pour d'autres cela comprend aussi l'adaptation au niveau des composants. Pour certains cela comprend la modification des composants mêmes du système, pour d'autres cela n'est possible que par extension, ou par ajout de composants.

- L'écriture de code : contrairement à [Mac91b] la personnalisation s'étend à l'écriture de code pour [FG90],[BD95],[Tri92], [KM95] et [Mør97a].

- L'aspect plus ou moins prévu des modifications : à première vue, ce point n'est pas indépendant des deux autres, dans la mesure où il n'est pas évident d'envisager des changements imprévus sans modifier ou ajouter de composants, ou sans écrire de code.

La programmation par l'utilisateur dans un contexte d'utilisation et de personnalisation

En réalité notre démarche s'inscrit à la fois dans le domaine de la programmation par l'utilisateur final et dans celui de la tailorabilité, parce que ces deux thématiques ont leur place dans une réflexion générale sur la flexibilité. Il y a cependant une précision importante à formuler : autant la tailorabilité se définit d'abord par la tâche à réaliser, l'objectif poursuivi, puis ensuite par des techniques qui permettent de l'implémenter, autant la programmation par l'utilisateur se définit au contraire d'abord par des techniques (programmation par démonstration, programmation visuelle, tableurs, ...) puis ensuite par des activités auxquelles ces techniques peuvent s'appliquer. Contrairement au terme de personnalisation, qui décrit l'action d'adapter un logiciel, le terme de programmation par l'utilisateur ne présume pas *ce* qui peut être programmé : le système lui-même, de nouvelles fonctions, ...

A titre d'illustration du fait que la programmation par l'utilisateur est non seulement caractérisée, mais reconnue comme une technique, [Mør97a] par exemple écarte la programmation par l'utilisateur comme *technique* de réalisation de la customisation. Selon cet auteur, ce type de programmation ne répond pas à tout : c'est un *langage* de conception et non de re-conception ou d'intégration (plutôt que d'extension), et à ce titre doit s'opérer dans des langages différents ; ou encore dans [Mør97c] : la programmation par l'utilisateur ne suffit pas, les macros et scripts sont des langages d'intégration, pas d'implémentation. Cette distinction recoupe celle de [Eis97] entre scripting et programmation : l'un se définit par l'utilisation du langage de l'interface, l'autre par l'utilisation du langage sous-jacent : mais il n'en déduit pas la différence entre programmation par l'utilisateur et extension. Il nous semble qu'il serait utile de montrer à quel point ces deux domaines peuvent être amenés à mettre en oeuvre des techniques similaires, qui nous intéressent dans ce chapitre.

La différence principale se situe au niveau des objectifs et du mode d'opération :

- Mode d'utilisation : lorsqu'on parle de systèmes de programmation par l'utilisateur final, on parle d'outils dans lesquels en général la programmation est l'*utilisation normale* : dans un tableur, l'écriture de formule est le mode normal d'utilisation. Au contraire, dans les logiciels tailorables, l'activité d'adaptation est *optionnelle et périphérique* : en principe tout doit déjà fonctionner

d'une manière satisfaisante sans que l'utilisateur n'ait à spécifier de préférence ou de détails spécifiques (du moins est-ce la meilleure manière selon nous de concevoir la tailorabilité).

- Aspect cognitif de l'activité : nous n'avons pas encore décrit les travaux de [Gre96] qui seront abordés dans le chapitre II, mais parmi les activités génériques qui selon lui décrivent toutes les activités possibles dans l'utilisation d'un système informatique (incrémentation, transcription, modification, conception exploratoire) l'activité qui correspond à la programmation par l'utilisateur est une activité d'incrémentation (du moins lorsqu'il s'agit d'utiliser un tableur) alors que la customisation est plutôt une activité de modification, ce qui implique des propriétés assez différentes dans l'artefact informatique - bien que ces différences s'atténuent lorsqu'il s'agit plus de "blended-users" que de "end-users" [Gre99]. Ainsi, dans le cadre de la programmation par l'utilisateur, il n'y a pas en principe d'application pré-existante à adapter. Dans un tableur, il faut créer une feuille de calcul et non modifier des feuilles de calcul par défaut - même s'il y a peut-être des exemples fournis dans la documentation. Prenons encore l'écriture de scripts, c'est bien de l'incrémentation.

Dans un cas (programmation par l'utilisateur) on crée, on incrémente, mais on n'est pas dans l'extension alors que dans l'autre (personnalisation) on modifie, on étend, on rentre dans l'implémentation. Il semble donc que les techniques, la responsabilité et l'objectif soient complètement différents (comme le sont aussi manifestement les domaines de recherche). Dans le cas de la customisation on peut parler de protocoles méta-objets (MOP) [Dou95], on modifie le système. Dans le cas de la programmation par l'utilisateur, il ne faut surtout pas le faire. Inversement dans les MOP on ne parle pas tellement d'incrémenter le système, d'en modifier l'interface, si l'on excepte l'évocation par [Kic92] de l'idée d"'open behaviour". Pourquoi une telle distinction ? En gros la question c'est : quels MOP pour la programmation par l'utilisateur, qu'il s'agisse d'incrémentation ou de conception exploratoire ? Cette activité n'est pas limitée à l'intégration : c'est aussi ajouter une nouvelle fonction. Fournir des formules à un tableur ou une méthode de type stratégie [GHJV95] à un mécanisme de calcul, c'est à la fois plus de l'utilisation que de la customisation. En effet, le système ne fonctionne pas sans cela - on le complète plutôt qu'on ne le change, mais cela utilise les mêmes techniques que la customisation de [Mør97b] ou les MOP comme celui de [Rao91] ou de [Dou96b].

Ce que nous essayons de montrer dans la figure 5 est que la "place" de la programmation par l'utilisateur, comme l'écriture de formules, n'est pas la même dans le système tant du point de vue de la fonction (aspect cognitif ou mode d'utilisation) que du point de vue des niveaux de langage. Ce que cette figure illustre également, c'est l'aspect additionnel de la programmation par l'utilisateur au sens des techniques connues par rapport à l'aspect optionnel et "in situ" des techniques de customisation.

Cependant, on peut aussi constater que programmation par l'utilisateur et personnalisation sont des activités qui se rejoignent, notamment sur l'aspect nécessairement spécifique, dans le sens où la customisation intervient forcément dans un domaine - celui de l'application, et la programmation par l'utilisateur a tout intérêt à coller au domaine. Ce sont enfin des activités complémentaires qui peuvent s'effectuer dans une suite cohérente : pourquoi la programmation par l'utilisateur ne serait pas modifier la système et pourquoi la customisation ne serait pas de l'intégration ou de l'incrémentation aussi (faire une macro et s'en servir comme d'un bouton : c'est bien personnaliser l'interface) ? [Mør97b] le reconnaît : le tailoring n'est ni du développement, ni de l'utilisation, mais quelque chose entre les deux. Enfin, et c'est notre hypothèse, la programmation par l'utilisateur si on la considère isolément et du point de vue de son accessibilité, gagne aussi à être située dans un contexte d'application par opposition à un environnement qui laisserait l'utilisateur dans une totale créativité, ne serait-ce que parce qu'elle est plus accessible dans un contexte riche d'exemples, si ce n'est dans un contexte riche d'occasions réelles de programmer.

En somme, nous nous situons bien en partie dans le cadre de la programmation par l'utilisateur (composition) mais dans un cadre où cette programmation s'effectue concrètement "comme" une modification d'un système existant (extension, ré-implémentation). Ce que nous reprenons essentiellement de la démarche de personnalisation, c'est donc son caractère *optionnel* , comme pour les protocoles méta-objets [KLL+97] : l'application présente un comportement par défaut qui remplit correctement le cahier des charges de ce type de système. Mais comme nous avons aussi pour objectif de permettre à

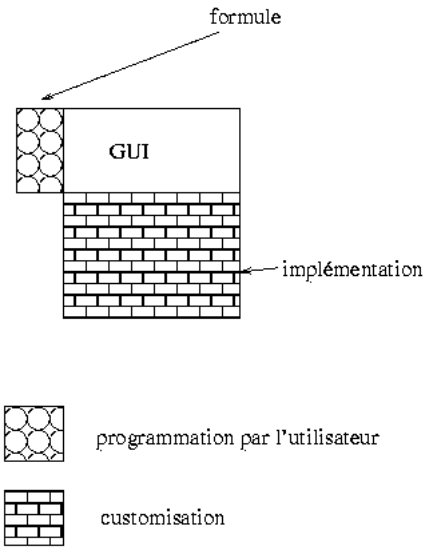


FIG. 1.5 – Programmation par l'utilisateur et personnalisation.

la programmation par l'utilisateur de se développer pour elle-même, ce cadre n'implique en rien pour nous les limites habituelles, tant techniques que sémantiques, des approches de personnalisation ; nous ne voyons aucun inconvénient par exemple à un éventuel changement de sémantique, voire à l'évolution indépendante de plusieurs "versions" du système. Ce que nous reprenons de la démarche programmation par l'utilisateur ce sont les aspects de création et d'activité incrémentale, sans que cela implique nécessairement un niveau de langage différent. Enfin notre approche n'est pas non plus incompatible avec l'apprentissage de la programmation, voire la construction de systèmes - qui de moyens deviennent alors fins, du moins si l'utilisateur en a la curiosité.

Niveaux de langage.

Comme nous l'avons vu plus haut, certains auteurs comme [Mør97c] considèrent que le niveau complexité des langages de programmation par l'utilisateur n'est pas adéquat pour les techniques d'extension. Par ailleurs, la principale raison d'être des techniques de programmation par l'utilisateur *final* est l'accès à la programmation sans programmer, comme nous l'avons décrit plus haut. Mais pour [Mac91b], la personnalisation doit aussi se faire sans programmer. Bien qu'il semble que la modification directe d'un système se fasse plus aisément dans le langage dans lequel il est spécifié, cela n'est pas obligatoire notamment si un protocole particulier est prévu pour cela, ou si les spécifications de l'utilisateur pour modifier le système sont fournies sous forme de sélection parmi des possibilités prédéterminées. Il est vrai que dans ce cas il manque la dimension d'imprévu qui selon [Dou96c] est une caractéristique importante d'un système adaptable, et que la personnalisation par sélection est nécessairement moins puissante que la personnalisation par création ; par exemple, [Dou96c] constate que dans certaines boîtes à outils de systèmes CSCW, le langage du système dont le contrôle est paramétrable est restreint à des modes opératoires prédéfinis (on ne pourra pas en créer un nouveau).

Nous pouvons donc considérer que cette dimension n'est pas vraiment essentielle pour différencier les deux activités de personnalisation et de programmation par l'utilisateur. Le choix du niveau de complexité du langage se fera donc certainement plus en fonction du type d'utilisateurs ciblé (enfants, chercheurs en biologie, menuisiers, compositeurs, ...) que du type de tâche. Dans le prototype que nous avons réalisé, biok (chapitre V), les formules et les méthodes sont en XOtcl, même si par nature la formule est plus simple (elle ne prend pas de paramètres, il n'y a pas de contrôle, etc... ce qui s'explique par quelques facilités syntaxiques fournies par l'interface d'évaluation des formules).

Personnalisation et paramétrabilité.

Il y a une différence importante entre les problématiques de customisation et de paramétrabilité. Les

deux se construisent autour de l'idée de l'apport de l'utilisateur dans la manière d'utiliser un logiciel et posent le même type de question d'architecture. Mais la caractéristique de persistence d'une utilisation à l'autre situe la personnalisation dans une problématique de construction du logiciel par l'utilisateur.

Pour défendre cette idée de continuité entre programmation/modélisation et programmation/customisation, comme cela est illustré par une courbe unique dans la figure 6, il faut accepter l'idée que l'utilisateur non-informaticien soit partie prenante dans la construction de l'outil ou du système, et ce, aussi bien au début, par la conception participative, que par la suite en ouvrant à la tâche de customisation toutes les possibilités techniques de la programmation.

On peut considérer aussi que cette courbe représente les deux étapes du lambda-calcul : abstraction (lambda) et application (beta-réduction) [OFL97]. La customisation peut être considérée comme une étape intermédiaire d'application *partielle* . C'est exactement à ce niveau là du "processus de programmation" que se situent les logiciels scientifiques permettant un contrôle : à mi-chemin entre un programme trop général et un programme trop déterminé. Tout l'art du développement d'un logiciel customisable ou programmable serait donc de définir correctement le niveau d'abstraction où il faudrait s'arrêter. L'idéal serait cependant de permettre un retour en arrière dans la courbe. On pourrait spécialiser, mais pas généraliser un logiciel, du moins de manière prévue ? Il y aurait une sorte de mur de verre ou de frontière mouvante mais aussi irréversible que le temps ou l'entropie (il serait plus facile de re-construire que de dé-construire, c'est que que l'histoire des problèmes de la ré-utilisation semble dire) ?

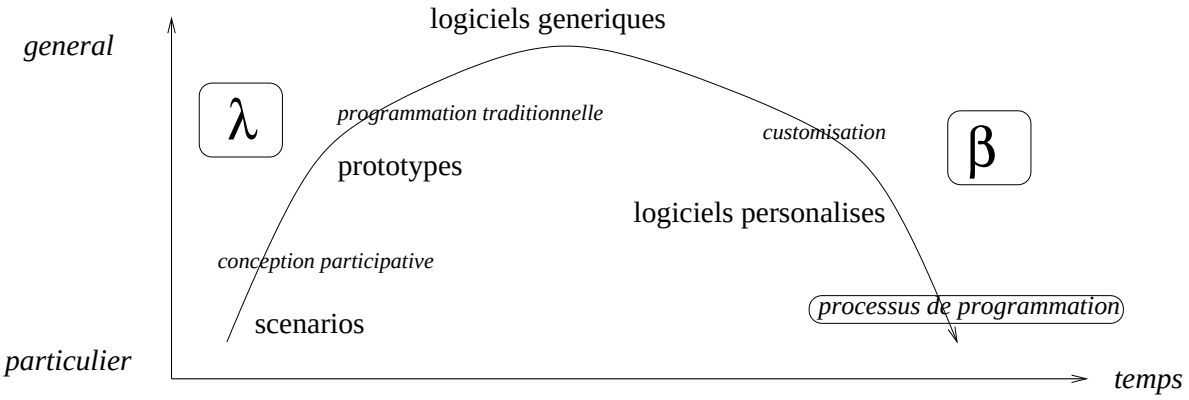


FIG. 1.6 – Généralisation/spécialisation.

1.2.3.4 Construction de systèmes interactifs.

Bien que cela puisse sembler hors sujet, les recherches sur la programmation par l'utilisateur et les systèmes ouverts ont, nous semble-t-il également un intérêt quant à la question plus générale de la construction de systèmes interactifs. Elles permettent d'une part d'explorer d'autres manières de construire le logiciel qui intéresseraient aussi les informaticiens professionnels, notamment pour la construction de logiciels interactifs, dont les théories fondatrices de l'informatique comme calcul rendent compte assez mal [BL93a], [Weg97a]. Et d'autre part, sans doute des logiciels construits selon des techniques apparentées à celle qu'un utilisateur non-professionnel en informatique utiliserait seront plus naturellement adaptés à ce type de problématique. C'est en fait ce que nous avons pu expérimenter en construisant le prototype biok : sa construction a procédé d'une approche mixte, avec une partie classique de conception et d'implémentation de l'architecture, mais pour une grande part, c'est l'environnement lui-même qui a été utilisé pour expérimenter des idées, comme les idées de co-visualisation (la comparaison visuelle d'analyses) ou de paramètres actifs (exemple donné dans le chapitre V à propos de la recherche de motifs), sans parler bien sûr des aspects de déboguage et d'observation de l'exécution qui ont été très utiles.

Construction interactive et directe.

Si on y réfléchit, l'idée de construction interactive et directe est presque aussi paradoxale que

l'expression "programmation par l'utilisateur" : en un sens, rien ne s'oppose plus que la programmation, cette activité planifiée, et l'interaction, où chaque action de l'utilisateur dépend de ce qui vient juste de se passer. Que peut-on entendre par construction interactive ?

1. Le mode de spécification est interactif, mais l'objet spécifié n'est pas directement le système interactif : on peut mettre dans cette catégorie un très grand nombre de langages visuels dont le niveau de spécification est la description indirecte, même si les techniques utilisées sont visuelles [BL93a]. Le terme plus juste pour décrire ces approches est plutôt programmation visuelle ou plus généralement par manipulation directe, mais pas programmation interactive.
2. Le mode de spécification est interactif et l'objet spécifié est directement le système lui-même. C'est le cas de manière évidente pour les systèmes permettant la construction de l'interface utilisateur par manipulation directe (c'est donc direct dans deux sens du terme).
3. La programmation a un caractère incrémental : l'entrée de l'instruction K s'effectue après avoir eu le résultat de l'instruction K-1, voire après l'obtention d'informations supplémentaires venant du monde extérieur au système : c'est la définition de l'interactivité de [Weg98].
4. La programmation se fait de manière interactive parce qu'elle se produit dans un contexte d'utilisation [Mør97b][FG90].

Les cas 3 et 4 ont été traités dans les sections précédentes et le cas 1 ne correspond pas à ce qui nous intéresse. Le cas 2 est-il limité à la construction de la partie interface graphique ? Peut-on sérieusement envisager l'utilisation des techniques de programmation par démonstration ou sur exemples pour la définition de *tout* le système ?

Auto-amorçage.

En somme, est-il possible de construire un système interactif de manière totalement interactive ? ou en d'autres termes : peut-on concevoir des techniques de construction de logiciel qui soient plus adaptées à la nature différente des systèmes interactifs par rapport aux systèmes algorithmiques fermés, et qui aillent peut-être en même temps dans le sens d'un accès plus large à la conception de logiciels ? D'après [BL93a], on peut tenter d'apporter une validation théorique supplémentaire à l'ingénierie des systèmes interactifs soit par une description de ces systèmes comme un calcul déterminé par un langage, celui de l'interaction, soit par une approche directe qui construit le système sans langage intermédiaire. L'inconvénient des approches descriptives est la trop grande complexité de l'interaction. Les techniques constructives comprennent les générateurs d'interfaces (génération d'après un dessin de l'interface), les techniques déclaratives, avec construction par démonstration ou par programmation visuelle. Elles se différencient des techniques de description dans le sens où la définition de l'interface est faite directement, sans passer par un codage intermédiaire. Ces techniques sont souvent incomplètes : les générateurs, comme les boîtes à outils, s'arrêtent à la couche présentation (même avec les valeurs actives ou les connexions entre objets). Pour la programmation visuelle, il nous semble qu'elle relève le plus souvent des techniques descriptives : la plupart du temps, ce ne sont pas du tout les objets du système interactif qui sont "construits", mais ceux de la description elle-même ; simplement, le langage de description est graphique. Mais les techniques constructives ont pour avantage de permettre un développement itératif, à base de prototypage, voire même de langage de prototypes, dans lequel l'utilisateur a sa place. Cette approche, que l'on peut considérer comme de la programmation interactive, permet également l'auto-amorçage, et la personnalisation inter active. Tout comme nous essayons de le faire ici, elles envisagent la programmation, même si c'est à des fins de production de logiciel, dans un contexte d'utilisation, et en cela, elles sont liées à la programmation par des non-informaticiens. Ce n'est donc peut-être pas par hasard que les deux sujets de la programmation par l'utilisateur final et de la construction d'interfaces utilisateur soient abordés dans un même livre : "Languages for Developing User Interfaces" [Mye92], un peu comme si la clé et le seul objectif de la programmation par l'utilisateur était l'interface graphique ou inversement comme si le meilleur langage pour la construction d'interface était nécessairement un langage pour l'utilisateur.

Ce qui est souhaitable, néanmoins, même si les techniques interactives ne permettent pas un jour de programmer des applications entières, c'est que le développement des techniques et des réflexions concernant la programmation par l'utilisateur puisse influencer la manière dont les logiciels sont programmés [Eis97]. Et cela peut se produire aussi bien sous la forme de possibilités de personnalisation, par des langages proches de l'interaction, que sous la forme d'une participation plus importante des utilisateurs aux différentes étapes de fabrication.

1.2.3.5 Conclusion.

Nous avons tenté dans cette partie de décrire les problèmes que nous avons rencontrés au cours de nos réflexions sur la programmation par l'utilisateur. Nous avons essayé de le faire en présentant d'abord les orientations et les idées couramment admises dans la littérature ou dans la pratique quotidienne, mais ces idées ne nous ont pas éclairé suffisamment. C'est pourquoi il nous a semblé intéressant de dégager des axes problématiques comme les axes décrivant les tâches ou le mode d'action moins souvent traités que l'axe décrivant la forme de l'expression. Nous avons également essayé de re-situer la problématique de la programmation par l'utilisateur dans un contexte qui est celui de l'utilisation de logiciels existants : situation plus réaliste du moins dans le domaine de la recherche en biologie que celle où l'utilisateur aurait tout à construire. C'est l'idée générale de partage du travail qui nous a guidé comme un cadre dans lequel on peut réfléchir à la fois aux questions de de programmation interactive (avec la programmation comme une forme poussée d'interaction), de paramétrabilité - comportant des limites qui sont à l'origine de la nécessité de la programmation par l'utilisateur - et de construction logicielle - avec les problématiques de personnalisation et de construction interactive. On note que, placée dans cette perspective de programmation située dans un contexte, qui selon nous est la perspective pertinente dans le cas des biologistes qui ne sont pas devenus programmeurs, la difficulté de la programmation prend un peu moins d'importance, et que l'accent est plutôt mis *sur la manière dont les logiciels sont construits* , c'est-à-dire leur inadaptation à ce type d'utilisation, pourtant fréquente.

Cette mise en place problématique aura été complexe a établir car il y a de nombreux recoupements entre les techniques utilisées et les objectifs poursuivis, comme si le but de la programmation était toujours le même mais par des moyens différents, et comme si les caractères discriminants de ce qui constitue la programmation étaient des invariants.

Il existe de nombreuses recherches concernant l'un ou l'autre des aspects que nous avons abordés, et, maintenant que nous avons établi les axes problématiques qui nous conviennent, il est possible de les décrire comme des solutions au moins partielles.

1.3 Approches.

Après avoir défini notre espace du problème, c'est donc l'espace des solutions que nous nous proposons d'aborder maintenant. Celles-ci se partagent en deux grands ensembles :

- Les solutions techniques qui partent du principe que la forme de notation habituellement utilisée en programmation ne convient pas et qui explorent des types de notations différents.
- Les solutions qui modélisent le problème de la programmation dans le cadre d'un système de navigation ou de communication.

1.3.1 L'importance de la notation.

[DiS99] montre, sur l'exemple d'une démonstration mathématique avant l'invention de la notation algébrique, à quel point la notation peut avoir une importance cruciale dans la conceptualisation et le raisonnement. Comme en tout, il faut penser dans le langage pour arriver à penser : c'est l'argument de *medium d'expression* [Eis97]. Grâce à la formalisation dans un langage de programmation, on parvient à une nouvelle compréhension, potentiellement plus riche, des idées et des méthodes. [GBC97] montre aussi à quel point, pour les novices en programmation, ce sont des éléments de niveau lexical, fortement liés à la notation, qui prennent une grande importance dans la compréhension. A ce titre, il peut être intéressant de considérer les approches de la programmation visuelle dont l'objectif essentiel est d'explorer les possibilités d'un type de notation différent du texte : la notation graphique.

1.3.1.1 Formes textuelles, formes visuelles.

Le problème d'une nouvelle notation n'est pas simple et semble ne pas pouvoir se réduire à un simple passage du textuel au visuel. [DiS97] est assez efficace dans son argumentation pour nous convaincre que Boxer, parce qu'il rend les objets et les concepts de programmation plus visibles et

surtout plus manipulables, rend la programmation et l'abstraction plus facile à apprendre qu'avec Logo où les objets doivent être désignés indirectement. Mais la littérature sur la programmation visuelle abonde de polémiques sur les avantages cognitifs, pratiques ou pédagogiques du tout, du un peu ou du pas du tout visuel. On reproche souvent aux langages visuels d'être arbitraires, immatures et non suffisamment orientés domaine [Rep93a]. Ainsi, [FJG95],[Nar95] ou [Cor92] pensent que la forme visuelle n'enlève rien à la complexité conceptuelle, et qu'elle ne fait que la transposer. A l'argument selon lequel la pensée s'aide souvent de croquis [BWGP98] et [Rep93a] répondent qu'on ne construit jamais tout un programme avec cela, que ces schémas ont une durée de vie limitée, et qu'ils deviennent vite incohérents entre eux au cours du développement (on notera cependant l'utilisation de croquis graphiques dans les design rationale de [Mør94] comme justification historicisée des prises de décisions de conception successives). Quant à [Bro87], il affirme que la réalité du logiciel n'est pas dans l'espace : une représentation cartographique ou avec des diagrammes comme pour le matériel n'est pas possible : il y en aurait trop, les uns superposés sur les autres (flot de contrôle, de données, schémas de dépendances, séquences temporelles, espaces de nommage) et ces graphes ne seraient pas toujours planaires, ni même hiérarchiques. De plus, comme le montre [Bla96], de nombreuses approximations sont faites par les auteurs quant aux critères d'évaluation de l'intérêt cognitif de ces systèmes ou de l'approche en général, défaut méthodologique qu'essaye de pallier [GP97a] en déterminant une liste de variables pertinentes et opératoires, que nous détaillons dans le chapitre II.

Le formalisme de spécification du langage est, avec le modèle de calcul, la caractéristique qui intéresse le plus souvent les auteurs de taxonomies des langages visuels. Bien qu'il soit difficile d'isoler une présentation des formalismes choisis par les différents langages du modèle de calcul sous-jacent ou du domaine d'application, et surtout du niveau de langage de la machine virtuelle décrite [dS00], il est cependant intéressant de présenter certaines de ces idées de notation dans le cadre d'une réflexion sur la notation.

1.3.1.2 Métaphores spatiales.

[Mye90] répertorie les types de notation suivants : organigrammes et variantes, réseaux de Petri, graphes dataflow, graphes orientés, autres variantes de graphes, matrices, formulaires, icônes et phrases icôniques (la position de l'icone compte), pièces de puzzle (cf BLOX Pascal où la syntaxe est guidée par la forme des pièces : on ne peut pas assembler n'importe quoi ensemble), tableurs, démonstration, rien (quand la démonstration est faite sur des données). [BB94], plus succintement, identifie (niveau II.B. de sa classification) : les diagrammes, les icones, les séquences d'images statiques. [Nic96] fait aussi une étude détaillée de l'usage des diagrammes en informatique.

Sans doute classés parmi les langages à formulaires, ou alors non classifiés, les systèmes comme Boxer [DA89], VEX, un outil de programmation visuel pour la lambda calcul [CHZ95] (figure 7), LiveWorld [Tra94], inspiré de Self, où les objets sont des agents, représentés graphiquement par des boîtes (figure 8), Forms/3 [BAD+00], un tableur visuel (figure 9), ou MAP [FJG95], un langage visuel qui permet de voir la structure des programmes comme un emboîtement de blocs qui correspondent soit au blocs de code, soit à des structures de données (figure 10), ont pour caractéristique commune de traduire l'un des concepts de programmation les plus importants, la notion d'environnement lexical - que ce soit par le biais de l'abstraction procédurale ou du bloc, par la forme de la *boîte* et des emboîtements : qu'elle soit ronde ou carrée, il s'agit bien d'une figure représentant la fermeture.

Ce qui est plutôt bienvenu, c'est que cette forme a une correspondance dans le domaine des interfaces utilisateur : celle du formulaire, ou même de la cellule de tableur, ou encore plus généralement celle de l'objet graphique souvent constitué d'éléments imbriqués. Dans [PL92], on montre par exemple qu'un programme Boxer de modélisation d'une molécule, utilisant le formalisme de boîtes imbriquées, peut se construire en plaçant le code correspondant à chaque sous-structure de la molécule au niveau de chaque sous-structure graphique.

La traduction de la dimension temporelle est, elle, beaucoup plus difficile à réaliser. [LF95], qui définit la programmation comme la description d'un processus dynamique par une description statique, a bien décrit l'ampleur du problème. Dès qu'il s'agit de décrire une évolution ou un changement d'état, mis à part les approches par exemples comme [SCT00] ou ChemTrains [BL93b], les formalismes visuels rivalisent d'arbitraire et de complexité [Han94] et supposent souvent la compréhension préalable d'un

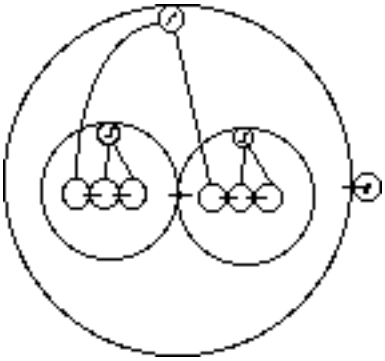


FIG. 1.7 – VEX [CHZ95]

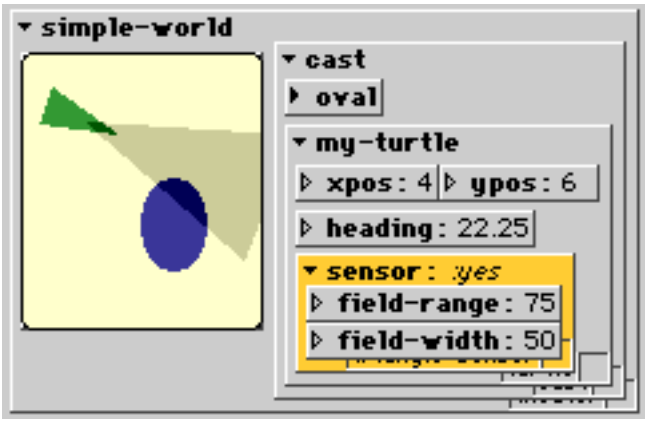


FIG. 1.8 – LiveWorld [Tra94]

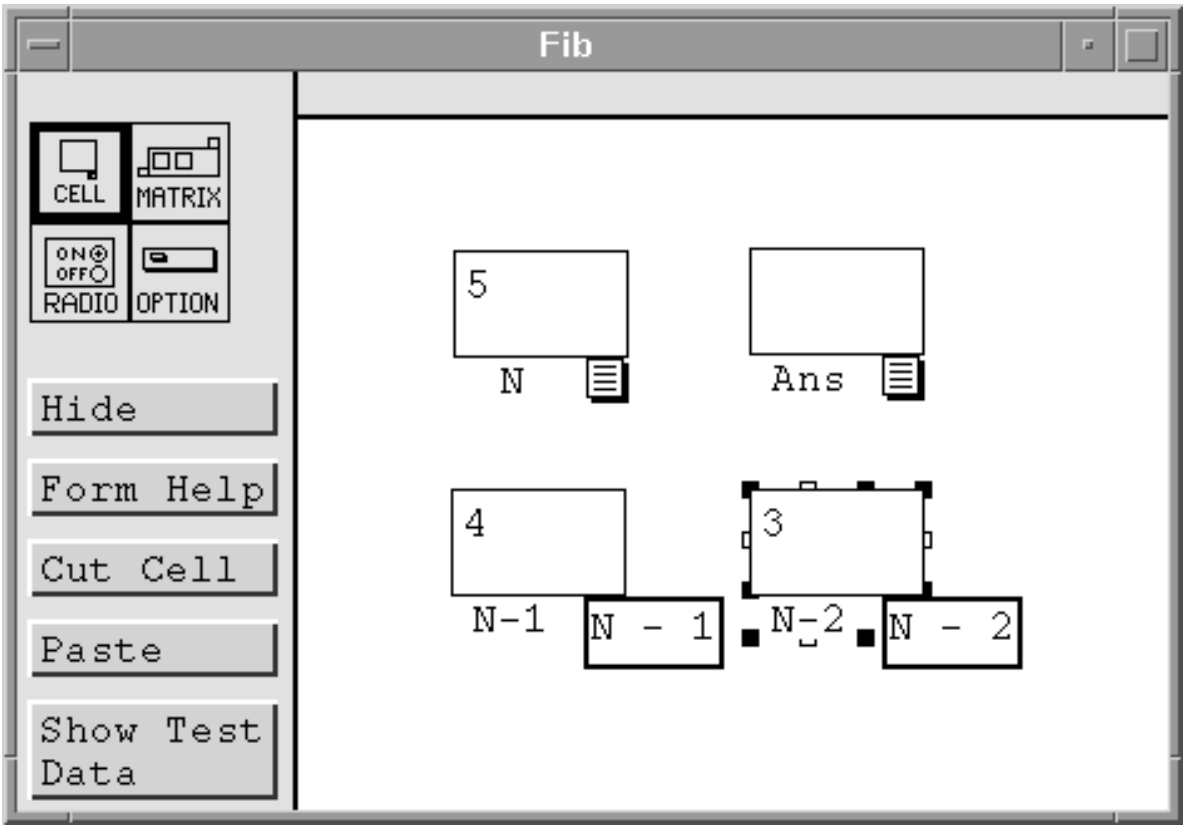


FIG. 1.9 – Forms/3 [BAD+00]

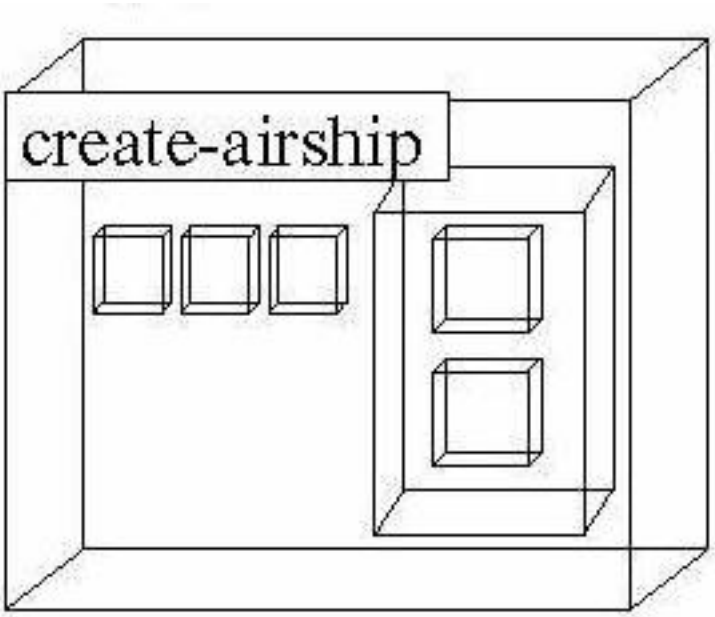


FIG. 1.10 – MAP [FJG95]

modèle non trivial de la part du programmeur. Quelques exemples :

- Hi-Visual [HMY+86] avec l'envoi de messages par superposition de boîtes hybrides méthodes/données (paradigme objet).
- Cantata [RASW90] : mécanisme data ou demand driven selon la présence ou non d' éléments interactifs dans le programme.
- VIPEX [HM88] qui propose trois sortes de boîtes : processing, composition, regroupement lexical - avec des principes d'exécution différents.
- Show&Tell (STL) [KCM90] dont le principe de contrôle repose sur le contrôleur d'incohérence.

La difficulté inhérente à la spécification des structures de contrôle en programmation visuelle est analysée dans plusieurs articles [CK00][CDZ93][GMP98][AGK+97][Han94].

1.3.1.3 Notation active.

On peut parler de notation active pour les approches de programmation par démonstration fonctionnant sur le principe de l'enregistrement de macros. La représentation est ici invisible, ou cachée [Eis97], puisqu'elle est constituée d'actions dans l'interface utilisateur.

Il reste que, avec ce type de notation, la modification de la spécification déduite devient difficile sans représentation éditable, ce qui change complètement la nature de la notation. Certains systèmes donnent une représentation de la macro ou du script enregistré sous une forme visuelle : comme un film pour Pygmalion [Smi75] ou un historique graphique pour Chimera [CKM93b]. L'intérêt de ces exemples, est que la notation reste familière à l'utilisateur, mais ce n'est pas toujours le cas.

La notation active pose en cela clairement la question de savoir si l'activité de programmation doit reposer sur de l'implicite ou sur de l'explicite (le code généré par opposition à la spécification interactive), ou sur du déclaratif par opposition au procédural [Gir00].

1.3.1.4 Différence description/exécution.

Comme le soulignent les auteurs de taxonomies des langages visuels [BHP+94] [BB94] [BGL95], [Dra92] [Mye90] [PBS93], il est important de distinguer les systèmes qui permettent une visualisation de ceux qui fournissent des mécanismes de construction. En outre, parmi les systèmes de visualisation, il faut aussi distinguer ceux qui font de la visualisation statique de programmes, de ceux qui permettent une visualisation de l'exécution.

Il résulte cependant de cette distinction que ces systèmes visuels sont présentés dans des taxonomies différentes et donc dans des articles différents, ce qui est bien dommage. On vient de voir en effet la programmation par démonstration comme l'une des "notations" pertinentes pour une programmation par l'utilisateur, et l'une des techniques les plus courantes dans ce domaine, la programmation sur exemples [Gir00] permet justement à l'utilisateur de réutiliser une exécution pour programmer [Lie92]. Plusieurs systèmes visuels ont d'ailleurs songé à l'importance de la cohérence entre la notation utilisée à la construction et à l'exécution ("shape retention" [FJG95]) pour la compréhension du programme en cours de construction, comme MAP [FJG95] (figure 6), VEX [CHZ95] (figure 3), où le modèle du lambda-calcul permet de penser l'exécution d'un programme comme un processus de ré-écriture qui se traduit tout naturellement par un processus de transformation d'images graphiques. Les systèmes à base de règles comme Cocoa [SCT00] qui permet de créer des jeux et des animations, Elody [OFL97], un système de programmation fonctionnelle visuelle de composition musicale, Peridot [CKM93c] pour la création d'interface ou ChemTrains [BRL91], un langage pour créer des animations, bien qu'il n'utilisent pas nécessairement une exécution réelle mais une image analogue, fonctionnent aussi sur ce principe que le programme doit ressembler à ce qu'on voit pendant l'exécution.

1.3.1.5 Réalisme naïf.

Est-ce par pur parti pris idéaliste que certains systèmes comme Self [CUS95], ARK [Smi86], Boxer [DA89] ou ToonTalk [Kah00] essayent, dans l'apparence même des objets du langage, de viser le littéralisme ou le réalisme le plus concret possible ? Dans ces environnements, on veut que l'utilisateur

programmation		utilisation	
code textuel	formalismes visuels explicites	formalismes visuels implicites	application

TAB. 1.3 – Explicite et implicite

ait l'impression de programmer directement les objets, et on donne même parfois à ces derniers une apparence qui donne cette impression : une ombre pour ARK et un aspect gravé pour Self. [DiS97] explique que le réalisme naïf consiste à donner l'impression à l'utilisateur que ce qu'il voit, c'est le système lui-même, et [SUC92] explique que le problème de la programmation, de manière générale, c'est la distance créée par la mention - le nommage - plutôt que l'utilisation directe. La dernière affirmation est très radicale, mais dans les deux cas, il est mis en avant que ce qui importe, c'est que l'utilisateur ait l'impression d'avoir une prise concrète sur le programme dans l'interface. Selon [Smi86], tout ce qui ne relèverait pas de l'analogie avec le monde réel pourrait apparaître à l'utilisateur comme de la magie.

1.3.1.6 3.1.6 Langages hybrides.

Si les langages hybrides sont intéressants ici, c'est qu'ils tentent, plutôt que d'appliquer systématiquement un principe de présentation (le tout visuel notamment), d'isoler les éléments d'un programme qui justifient réellement un aspect graphique. C'est ce qu'explique [Gre97] : si la notation est reconnue pour avoir un rôle important dans les processus cognitifs, il n'y a guère de généralisations possibles quant à l'avantage du tout visuel : chaque notation est utile à des types de compréhension différents. Il existe plusieurs exemples de langages mixtes, sans parler des tableurs, comme [RCB+98], ou [EM95] qui propose un langage textuel incorporant, pour les structures qui le nécessiteraient (arbres, graphes, ...), un formalisme visuel indépendant du langage hôte et associé à un traducteur. D'autres systèmes comme [Mye83] [Boe86] [BAD+00] [Ibr98] proposent un mécanisme visuel pour définir ou accéder aux structures de données.

1.3.1.7 Explicite et implicite : trouver le bon formalisme ?

Le tableau 3 tente de décrire l'axe - est-ce un continuum ? - sur lequel peuvent s'inscrire les langages de programmation visuels : les langages que [Lie92] qualifie de type 'Icons on Strings', à vocation généraliste, se classeraient dans la catégorie *programmation* , alors que les environnements de programmation dans l'interface, comme les systèmes comportant de la programmation par démonstration ou les applications programmables, comme celles de [DE95a], [Guz98], [JNZM93] (ACE), [Mør97c], [Tra94] (LiveWorld), ou [DA89] (Boxer), capables d'utiliser l'interface comme support pour la programmation, se classent plutôt du coté de l'*utilisation* .

Selon [Rep93a] ou [Lie92] les langages de programmation visuelle surchargent ainsi inutilement l'interface de programmation avec des relations explicites, faisant trop peu appel aux relations spatiales implicites. Mais, d'un autre côté, [Rep93a] donne une analyse intéressante du rapport inverse entre l'explicitation et la flexibilité : rendre explicites des éléments comme les relations entre objets permet de les manipuler en tant que tels dans le mode utilisation. L'exemple, cité dans ces travaux, de l'interface de la machine à café, dont le menu est constitué de codes numériques qui n'ont aucun rapport avec les boissons, montre que cette complexité pour le moins accidentelle aurait quand même pour avantage de rendre le système aisément extensible, puisqu'il ne serait pas nécessaire d'effectuer de grands changements dans l'interface utilisateur pour ajouter une nouvelle boisson. Les notations implicites sont donc moins flexibles que les notations explicites dans la mesure où, comme elles ne sont pas présentes dans l'interface, on ne peut pas changer leurs caractéristiques, comme leur label, leur couleur, etc...

Concernant ce compromis entre l'implicite et l'explicite, [NZ93] et [NJ94] montrent que l'outil visuel et interactif adéquat est un intermédiaire neutre entre les formalismes informatiques explicites, trop éloignés des utilisateurs non professionnels, et les représentations implicites très spécialisées et ancrées dans un domaine d'utilisation, mais insuffisamment souples. Au fond, ces derniers outils seraient plus porteurs de solutions toutes faites que support d'une activité de résolution de problèmes.

1.3.1.8 Formalismes visuels

[NZ93] montre que la question de la notation visuelle est trop abstraite en elle-même : ce n'est pas dans l'aspect visuel seul, par exemple dans le nombre de dimensions de la notation, que réside l'amélioration apportée par un changement de notation : c'est plutôt dans la combinaison d'une notation et d'un niveau de représentation conforme au domaine de l'application. On trouve de tels formalismes par exemple en musique avec PatchWork ou OpenMusic [ARL+99], des systèmes d'aide à la composition, qui utilisent une notation cohérente avec la composition acoustique, construite à partir de "patches" qui représentent des modules acoustiques à la manière de composants électroniques (il n'est pas indifférent à cet égard que le laboratoire où sont développés ces systèmes s'appelle "Représentations musicales") ; on en trouve également en informatique, avec les réseaux de Petri ou les Statecharts de [Har87].

1.3.1.9 Conclusion sur la notation

La notation a donc une grande importance, mais, comme le montrent les exemples que nous venons de voir, plutôt en tant qu'elle constitue un medium éditable sur lequel l'utilisateur peut travailler pour représenter sa problématique que si elle est comprise comme une solution syntaxique.

La direction qui nous intéresse est également celle qui consiste à pouvoir *passer* de l'implicite à l'explicite : chaque type de notation a son intérêt, mais il semble, ce qui n'est pas une découverte, que la flexibilité soit au prix de cette complexité apportée par l'explicitation des structures. Cela est vrai pour n'importe quelle composante d'un système, comme nous le verrons avec le principe de réification dans les protocoles méta-objets.

1.3.2 Niveaux de langage : structure et navigation.

Cette discussion sur les formalismes visuels avec d'une part leur relation avec la programmation visuelle et d'autre part leur ancrage dans les mécanismes supportant la réflexion dans un domaine particulier nous conduit à aborder la question de la distance entre les langages - non plus langage pour la programmation utilisateur ou langage d'implémentation, mais cette fois langage de l'interface et du système.

1.3.2.1 Orthogonalité.

D'après [dS00], les problèmes de la conception d'interface et de la conception de langages de programmation sont orthogonaux : d'un côté il s'agit d'écrire de bons algorithmes et de tenir compte des contraintes des compilateurs ou des interpréteurs, de l'autre il s'agit de résolution de problèmes et de critères d'utilisabilité. Ce sont même deux activités qu'on peut considérer comme devant être indépendantes. De même pour [Mør97b], la distance entre le langage de l'interface et le langage de l'implémentation est irréductible car l'un est orienté vers la tâche, et doit suivre les règles de la psychologie alors que l'autre est orienté vers l'efficacité et doit suivre les règles d'une grammaire formelle. Cette orthogonalité est décrite, dans le graphe de la figure 3, par l'axe utilisation-programmation, plutôt que par les axes textuel-graphique ou continu-discontinu.

Cette question est qualifiée de paradoxe par [Rep93a]. Comment faire pour qu'un environnement de travail approche le plus possible un domaine particulier mais en même temps permette une exploration suffisamment générale ? Elle est d'autant plus cruciale si on choisit une approche 'programmation dans l'interface', car là, bien sûr, on est directement et nécessairement dans le domaine. Mais cela est-il nécessairement contradictoire avec la généralité ?

Cette question peut également être décrite comme un problème de compromis entre programmabilité et utilisabilité : [Mør97b] démontre que plus la distance entre le code et l'interface utilisateur est réduite, afin de donner accès à l'utilisateur au code des fonctions tailorables, plus la distance entre la tâche et l'interface utilisateur augmente. La solution à cette difficulté est d'autant plus importante que l'utilisateur doit pouvoir réaliser quelque chose avec le langage dans un temps suffisamment court (critère d'approchabilité de [Nar95]).

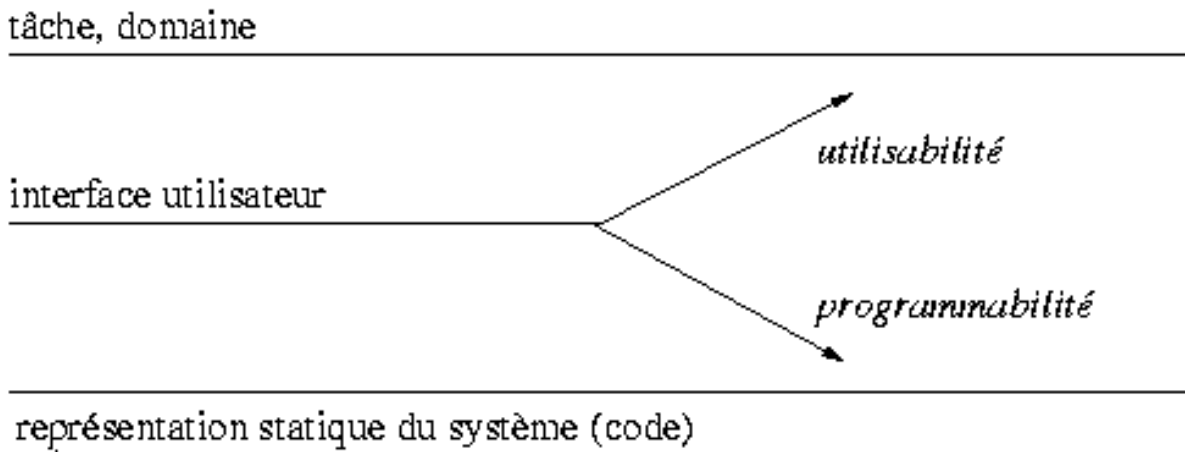


FIG. 1.11 – Distance code interface utilisateur.

1.3.2.2 Programmation visuelle, construction d'interfaces et programmation dans l'interface.

Si l'on y regarde bien, ce qui différencierait les langages de programmation visuels généralistes des approches 'programmation dans l'interface' c'est que les premiers sont en fait ... des éditeurs de programmes, alors que les autres sont des applications. Ces éditeurs sont soit complètement graphiques, soit semi-graphiques, mais ils interviennent au niveau du modèle, pas du programme en train de s'exécuter. A cet égard, on peut parler de bi-modalité : soit on est dans le cas d'une bi-modalité pure (programmation/utilisation), soit, si on peut programmer dans l'interface, on peut faire l'hypothèse d'un continuum entre des actions interactives et programmatives, voire même, comme [Mye90] ou [BAD+00] le disent des tableurs, on est dans un système non modal.

Les éditeurs d'interface, de ce point de vue, sont dans une position plus ambigüe. D'une certaine manière, on y programme dans l'interface, puisqu'on manipule celle-ci directement afin de la définir, et que le résultat de cette manipulation est un résultat réutilisable.

Il y a visiblement une distinction de type forme/fond, confirmée par les modèles d'architecture comme Seeheim [PtH85], selon laquelle il y a nécessairement deux niveaux de langage dans une application interactive :
- le langage de programmation,
- le langage d'interaction.

C'est le cas sans aucun doute pour la très grande majorité des applications. C'est même nécessaire par exemple pour les accès aux bases de données, et les guichets de toutes sortes. Mais il y a aussi un très grand nombre de logiciels dont la qualité interactive bénéficierait d'une appropriation plus large, ou même totale, par l'utilisateur.

Suffirait-il pour cela de considérer qu'il n'y a qu'un seul niveau de langage [Der92] : si l'interface est le langage alors on peut tout programmer avec, aussi bien l'interface que le noyau fonctionnel ? Mais d'autres comme [Cor92] démontrent que, par exemple, l'interface visuelle nécessaire à la construction d'un arbre binaire est un langage à part entière (machine de Turing compatible), et par là même tout aussi complexe que son équivalent textuel. Peut-être peut-on se demander si dans ces cas-là, il n'y a pas illusion sur la nature d'interface utilisateur de l'environnement de description visuelle - mais cette interface ne serait pas au bon niveau de machine virtuelle, à moins d'être de nature pédagogique.

1.3.2.3 Prise en compte de l'orthogonalité

On peut tenir compte de cette nécessaire distance de point de vue entre l'utilisateur et le système informatique principalement de deux manières complémentaires - qui rappellent la façon dont nous avions posé la question de la répartition du travail entre l'informaticien et l'utilisateur :

- En considérant ce qui intéresse l'utilisateur dans la conception et la programmation, et en distinguant ce qui relève du métier d'informaticien professionnel et du domaine d'expertise de l'utilisateur non-professionnel. Tout le monde ne peut pas tout faire.
- En étudiant comment faciliter le passage du langage de l'interface - univers des objets d'intérêt - au langage d'implémentation - univers plus informatique.

Ces deux axes se rejoignent puisqu'ils induisent nécessairement une certaine manière de structurer le logiciel pour le rendre le plus paramétrable possible. Le premier décrit de quoi se compose et comment se structure un système informatique qui laisse la place à la programmation par l'utilisateur. Le deuxième cherche comment utiliser ou adapter cette structure pour rendre une navigation possible entre l'interface et l'intérieur du logiciel et d'un composant à l'autre du logiciel. Ainsi, selon [vR99], il faut à la fois que le code soit manipulable, mieux découpé, en modules de taille humaine ou du moins d'une manière navigable (il fait à ce sujet référence à l'AOP - Aspect Oriented Programming, référence faite également par [Mør98]) et en même temps laisser à l'utilisateur la partie du code qui peut être spécifique à la manière des tableurs qui se chargent de l'affichage, des entrées-sorties et du flot de contrôle.

On a vu dans la section précédente qu'il existait, au niveau de l'architecture des logiciels, du modèle de calcul qu'il propose, des moyens de répartir intelligemment les tâches de programmation entre ce qui relève des connaissances de l'utilisateur, souvent plus proches de l'interface et des résultats attendus, comme c'était le cas avec la programmation déclarative et ce qui relève de l'architecture générale d'un système informatique. Mais, quelque soit le choix qui a été fait sur l'étendue des possibilités de programmation par l'utilisateur, il reste à définir l'articulation entre l'interface et le code pour réduire la distance entre les deux. En effet, l'accès à ces zones de programmation doit être fait selon les règles de l'utilisabilité - puisque cela fait désormais partie de l'interface entre l'utilisateur et le logiciel.

De manière générale on distingue les tendances "magiques" et les tendances "rationnelles". On pourrait en effet vouloir trouver une solution qui *élimine* cette distance - nous en avons cité quelques unes plus haut sous la rubrique réalisme naïf, mais certains spécialistes de ces questions de distance nous mettent en garde. Peut-être en effet ne faut-il pas, par une apparence trop réaliste, tromper l'utilisateur sur la nature purement informatique de certains objets [dS00], et peut-être aussi faut-il éviter toute illusion d'ordre métaphorique qu'il faut ensuite combattre pour permettre à l'utilisateur de dépasser son plateau de compétence [CR87]. Ainsi pour l'analogie, telle que préconisée par exemple par [DiS97], qui peut présenter un intérêt pédagogique, mais à condition qu'on en percoive les limites. En somme, il ne faut surtout pas faire disparaître la distance - au contraire, il faut l'expliciter, la rendre navigable, manipulable afin de ne pas donner l'illusion de compréhension.

Les approches rationnelles consistent donc à considérer un logiciel comme un produit explicable, pouvant rendre compte de son fonctionnement [Dou97], dont il faut fournir des descriptions explicites [GP97a]. On retrouvera également cette idée d'explicitation dans les principes de conception des protocoles méta-objets qui se fondent sur la notion d'interface méta-niveau explicite et documentée, conçue comme une réification des principes de fonctionnement des mécanismes sous-jacents du système [KdRB91]. Amener l'utilisateur à programmer l'application ne signifie donc pas seulement lui faciliter l'accès au code, ou lui faciliter la spécification par une notation adéquate. Il faut aussi lui permettre de comprendre le raisonnement du concepteur, de passer du modèle de la tâche au modèle du système. Plusieurs orientations de recherche tentent de décrire les implications de cette approche et d'expliquer pourquoi un logiciel comme Emacs, alors qu'il est complètement customisable et re-programmable, ne fait pas l'objet de tels développements [DE95a] - ou pourquoi l'idée séduisante de lentille sémantique qui, en se promenant sur l'interface révèle le code des objets qu'elle traverse [HRS97] n'est pas suffisante.

[Dou97] explique l'importance de cette rationalisation par rapport à la problématique de l'accessibilité à travers la notion d'*"accountability"* qu'on peut comprendre sinon traduire par la notion de responsabilité, de compte rendu : un système informatique doit pouvoir être lisible en termes de causes et d'effets, rendre compte des causes de son fonctionnement apparent. Ce type de lisibilité suppose comme on l'a dit au début de ce chapitre une certaine remise en question de l'idée de boîte noire : cette approche a donc sa place dans une réflexion sur l'articulation entre l'interface et le fonctionnement interne, puisqu'il s'agit de rendre visible ce dernier. Ainsi, l'expression "to account for" signifie : expliquer quelque chose ou la cause (ou les raisons) de cette chose. Si l'"accountability" dans le domaine du

Cause	*Effet*
modèle	instance
programme	exécution
algorithme	action
graphe de contrôle (organigramme, réseau de Petri, automate à états)	graphe d'états atteints
classe	objet
plan	maison
description	construction

TAB. 1.4 – Modèle et copie

logiciel consiste à pouvoir accéder à son code, ou du moins à une représentation symbolique cohérente par rapport à ce code, c'est donc parce que le code est la cause explicative et mécanique de l'exécution du programme, de l'affichage de l'interface, de la même manière que la classe explique l'instance, le modèle explique l'exemple, le plan explique la construction.

Parmi les idées de conception d'interfaces qui permettent de réaliser cette lisibilité, on peut citer les explications de conception ("design rationale") comme ceux de [Mør94], conçus explicitement pour mettre en correspondance les objets du domaine figurés dans l'interface utilisateur et le code. Ils n'ont volontairement pas le statut de documentation officielle - dont le message est souvent peaufiné et idéalisé - mais, sous la forme par exemple d'un historique des questions qui se sont posées et des décisions qui ont été prises, ils permettent à l'utilisateur de comprendre les raisons non nécessairement officielles mais en tous cas objectives de tel ou tel aspect du fonctionnement du système : c'est pour ces différentes raisons qu'ils ne sont pas exécutables et qu'ils sont si possible spécifiés dans un média différent du code. Ils sont aussi le moyen de navigation pour en voir les aspects concrets au niveau du code lui-même. Leur avantage par rapport aux patrons de conception (design pattern) est que ces derniers sont, du point de vue de l'utilisateur, trop axés sur la structure du code lui-même dont ils constituent une description schématique. D'ailleurs [Joh92][LN95][Pre94] en montrent l'utilisation possible pour décrire les frameworks. Une autre approche de ce type, c'est-à-dire de type "sincère" dans le sens où son objectif est de "laisser voir" ce qu'il en est réellement du système en utilisant des informations objectives, est celle des systèmes réflexifs qui font l'objet d'un développement plus détaillé dans le chapitre III.

D'autres approches sont plus construites, comme notamment les approches sémiotiques et pédagogiques. Selon le modèle sémiotique [dS99][dS00] l'interface utilisateur est un vecteur de communication entre le concepteur et l'utilisateur. Les signes qui constituent l'interface sont des messages que l'utilisateur doit essayer d'interpréter. Comme il y a nécessairement une différence entre les langages du design et de l'interaction, la conception de l'interface doit appliquer des stratégies qui aident à limiter les problèmes. Dans le cas particulier des systèmes qui proposent à l'utilisateur un niveau d'accès programmatique, systèmes ayant par conséquent une double interface, les solutions de conception qui en résultent sont axées autour de l'idée de *correspondance cohérente* entre les deux niveaux de langage. [dS00] montre comment, en utilisant les connaissances de la sémiotique (c'est-à-dire de la communication par des messages et des problématiques sémantique/pragmatique ou représentation/communication), on peut améliorer le passage de l'un à l'autre. On ne peut pas s'appuyer uniquement sur les syntaxes et sémantiques respectives du langage de programmation et de l'interface : l'homomorphisme et l'isomorphisme sont difficiles à maintenir - il y aurait trop de contraintes, la programmation visuelle l'a montré. Parmi les types de messages qui doivent passer du concepteur à l'utilisateur (affordances de modification et d'exploration), celui qui suggère comment modifier et étendre l'application est le plus difficile à communiquer ; il dépend du degré d'articulation des signes (leur structure interne et divisibilité en autres éléments).

Les approches pédagogiques réflechissent sur le rôle des exemples : si on part d'une approche de programmation dans l'interface, on peut faire en sorte que le code existant, bien adapté au domaine, soit en même temps un répertoire d'exemples - non pas nécessairement réutilisables tels quels, mais dont on peut s'inspirer. La mise en place et la définition d'exemples ne sont pas une chose simple et le problème est abondamment étudié [BG96][KG98][LC92][Nea89][vR99]. Ces exemples peuvent illustrer

une approche d'apprentissage de conception [May89]. Un autre type de mécanisme pédagogique pouvant aider l'utilisateur est celui de "disclosure" (révélation, découverte) décrit par [DE95a], qui consiste à révéler à l'utilisateur ce qu'il se passe au fur et à mesure des interactions : son avantage par rapport aux exemples est d'être situé dans une utilisation. Un exemple d'application de ce mécanisme est parfois proposé dans les systèmes de programmation par démonstration lorsqu'ils montrent à l'utilisateur le programme généré par ses actions : cela peut tout à fait devenir un moyen d'apprentissage de la programmation situé dans un contexte pertinent pour l'utilisateur. Ces aspects sont développés dans le chapitre II.

1.3.3 Méthode : générale ou spécifique, abstraite ou concrète ?

Nous l'avons vu avec le problème de l'orthogonalité, la difficulté de la question : faut-il une méthode générale ou spécifique, abstraite ou concrète ? est dans la tension entre deux buts apparemment contradictoires mais également importants :

- adhérer au domaine de l'utilisateur permet d'aider ce dernier à mieux établir une correspondance entre le problème et l'outil,
- fournir un outil suffisamment flexible et peu contraint permet des utilisations imprévues et constitue un meilleur support pour la résolution de problèmes.

Nous présentons cette question sous trois aspects différents :

1. L'élaboration de niveaux intermédiaires permet-elle de moduler les termes de cette tension et peut-on envisager des solutions mixtes ?
2. Quels sont les avantages et les inconvénients des langages orientés domaine ?
3. Y a-t-il moyen de combiner des objectifs génériques comme la construction interactive de systèmes interactifs avec des objectifs spécifiques comme la conception d'un environnement programmable d'analyse de séquences ?

1.3.3.1 Niveaux intermédiaires

Le seul moyen pour résoudre cette tension semble dans la constitution de niveaux intermédiaires qui articulent le niveau de programmation générale et l'ancrage dans un domaine particulier. La solution de type de [Mør97c] consiste à fournir une aide à la navigation entre l'interface utilisateur et le code de l'application par des explications attachées aux unités applicatives et qui en sont une des composantes (comme la vue et le modèle au sens du schéma MVC). L'approche de [Rep93a] consiste plutôt à construire un paradigme de construction intermédiaire, proche des formalismes visuels de [NZ93], pour la résolution de problèmes.

Ce type d'approche a pour avantage de traiter en même temps le problème des niveaux de langages différents de l'application et de l'implémentation, en fournissant des solutions de soutien à la navigation et à l'articulation entre les deux.

1.3.3.2 Langages génériques ou spécifiques ?

Le débat langages spécialisés ou langages généraux n'est pas nouveau. [NJ94][Nar95] jugent les langages spécialisés plus coûteux, et pensent qu'ils induisent une variété des langages d'un système à un autre qui n'est pas souhaitable, et enfin qu'il n'est pas toujours facile de définir les spécificités d'une tâche. On insiste sur l'accessibilité et le haut niveau des langages spécialisés : écrire des pages Web en C serait nettement plus compliqué qu'en HTML [vR99] ; mais s'ils ne comportent pas les constructions minimales pour programmer [CIS92], ils restent limités, même à l'intérieur de leur domaine. C'est sans doute vers une solution mixte qu'il faut tendre, soit par un langage unique comportant à la fois des instructions spécialisées et générales, soit en prévoyant plusieurs langages, comme dans [Mør97a] où à chaque niveau de customisation correspond un langage différent : feuilles de propriétés, langages de script, langage classique (ici Beta).

1.3.3.3 Combiner les objectifs

Poser en même temps les questions de la programmation par l'utilisateur, de l'apprentissage de la programmation et de la construction interactive de systèmes présente une difficulté. Dans tous les cas, on est amené à réfléchir sur un langage commun à la pédagogie, à la construction des logiciels et à leur utilisation - ce qui est déjà un problème très difficile. Mais, de plus, une technique de construction de logiciels, ainsi que l'enseignement de la programmation exigent un minimum de généralité, alors que la programmabilité d'un logiciel gagne à être pensée dans les termes et les concepts empruntés au domaine de son utilisation.

Tout d'abord, par la recherche d'une solution spécifique, appliquée dans un domaine bien précis, on verra peut-être que les meilleures solutions sont sans doute les solutions mixtes. Il existe déjà, que ce soit dans le domaine des outils visuels ou des techniques intelligentes, une palette très riche de solutions dont on peut s'inspirer.

Ensuite, l'intérêt de poser ces questions en même temps est de pouvoir mieux définir la limite entre le générique et le spécifique, et ce sur un cas concret plutôt que selon des lignes générales. De la même façon que [KdRB91] définit un protocole pour les langages objets, il est peut-être possible de définir un protocole méta pour les systèmes interactifs (MIP : meta-interaction protocol ?) qui englobe toutes les composantes de ces systèmes. Plus précisément, la question : "quels sont les caractéristiques d'un logiciel interactif adaptable" peut éventuellement permettre de trouver des éléments de réponse à la question : "quelles sont les éléments minimum nécessaires à la construction interactive d'un système interactif ?". Par rapport à la question du bootstrap présentée dans la section 1.2.3.4 [BL93a] nous avons opté pour une approche inverse : plutôt que de répondre à la question de la construction interactive de système interactifs par une technique d'amorçage, dont le point de départ serait par exemple un élément d'interface générique fait pour se cloner et se différencier de manière analogue au développement de la vie biologique, comme les boutons de [MCLM90], nous avons préféré une solution qui mette l'utilisateur dans un contexte d'utilisation habituel - une application avec des fonctions d'analyse correctement conçues - mais proposant des mécanismes de programmabilité non limités. On peut dire qu'il s'agit d'une solution "par le plein" par opposition à une solution "par le vide" (figure 12).

D'un coté, en rendant un système interactif non seulement scriptable, mais aussi complètement re-programmable, on en améliore l'interactivité [vR99][BHP+94]. De l'autre côté, on en tire des leçons pour améliorer la conception et la programmation des systèmes interactifs, dans la ligne des démarches de conception participative [SN93] ou de modélisation par l'utilisateur [Fis00][EF94]. On a déjà vu que l'activité de customisation participe d'une tâche constructive. La programmation dans l'interface peut tout aussi bien servir à cette activité qu'à une activité de nature interactive - ne laissant aucun résultat persistant. Les deux dimensions sont orthogonales.

1.3.3.4 Une solution particulière.

Au terme de ce chapitre, nous pouvons préciser notre approche. Soit le système informatique suivant :

- C'est un outil spécialisé dans un domaine particulier : l'analyse de séquences biologiques.
- Il se présente sous la forme d'outils graphiques, comme un éditeur de séquences, d'alignements, d'arbres phylogénétiques, de texte ; ces outils sont les outils habituels des biologistes et sont configurés pour pouvoir effectuer la plupart des tâches standards qu'on trouve dans ce type d'outils.
- L'éditeur d'alignement est un exemple typique de formalisme visuel suffisamment neutre [NZ93] pour permettre des traitements de toutes sortes, comme en témoigne déjà de nombreux exemples de développement de feuilles de calcul ou de piles HyperCard basées sur les alignements de séquences biologiques [AG00] [Ber87] [Del00] [Dop90] [Eer92] [Hay93] [MG98] [G97][Usd92].
- C'est un environnement programmable : les objets du domaine présents dans l'interface sont décrits par des formules, qui renvoient à des objets internes qui sont éditables.
- Cet environnement fonctionne sur un modèle de type tableur (les objets de l'interface sont définis par une formule) et donc dataflow - dont le contrôle doit être maîtrisable par l'utilisateur.
- Il n'y a pas de limite à la programmabilité de cet environnement : c'est un environnement de

programmation en même temps qu'un outil d'analyse.

Notre hypothèse est que ce type de système remplit un ensemble de buts, qui sont, pour les biologistes :

1. effectuer son travail,
2. adapter et étendre l'outil pour d'autres tâches,
3. apprendre à programmer,
4. construire cet environnement.

Le dernier objectif ne doit pas surprendre : un des intérêts principaux de la programmabilité de ce prototype est la définition par les biologistes eux-mêmes, en complément et dans la suite d'ateliers de conception participative qui ont déjà eu lieu [Let99a][Let99b][Let00a], des fonctions et des interactions utiles de cet environnement.

Le niveau de langage nécessaire doit être à la fois celui des langages de script [BHS96] [Dub99] [NZ00b], c'est-à-dire des langages de "colle", d'assemblage de composants, mais aussi celui des langages d'implémentation [Mør97b], c'est-à-dire un langage capable de décrire une classe, un objet une méthode. Si un langage possède ces deux propriétés, il convient (comme c'est le cas pour Xotcl [NZ00b]).

1.4 Conclusion.

Dans ce chapitre, notre objectif était de décrire les termes du problème de la programmation par l'utilisateur, et parcourir les différentes approches qui le traitent. Ce que nous avons voulu montrer, c'est que les aspects de ce problème se décrivent difficilement en termes de définitions (qu'est ce que programmer, qu'est-ce qu'un utilisateur final, ...), très sujettes à discussion car nécessairement mal définies dans une situation multi-dimensionnelle, sociale, culturelle et historique. Il nous a semblé plus utile, plutôt que de partir de définitions, d'éclaircir des objectifs généraux qui sont pertinents *selon le contexte* où l'on se place (type de problème, type d'utilisateur, situation professionnelle), quitte à ce que ces objectifs ne soient pas nécessairement ce que nous appelons communément programmer. C'est l'objet du chapitre II de préciser ce contexte et de montrer comment la démarche participative est un outil efficace pour cela.

Il nous semble qu'à travers le raisonnement proposé dans ce chapitre, nous sommes arrivés à une sorte d'équation dont les termes sont :

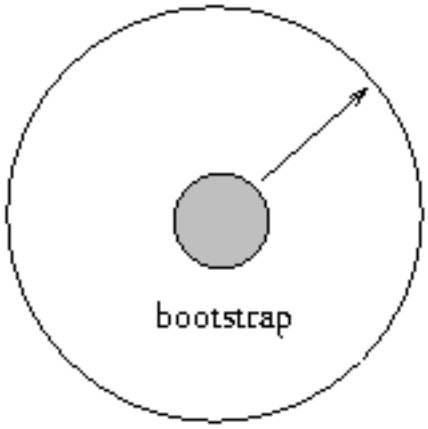
programmation généraliste + approche contextuelle = environnement de travail complètement programmable

En effet, notre premier objectif est la flexibilité de l'application, et nous avons vu que la flexibilité s'obtient par une forme ou l'autre de programmation, qui soit suffisamment *générale* pour "prévoir qu'on ne peut pas tout prévoir". D'autre part, l'objectif d'apprentissage de la programmation nous semble pertinent pour la biologie : cela ne signifie pas que nous pensons que tous les biologistes doivent savoir programmer, mais que, pour ceux qui le désirent, cela ne peut qu'être très utile du point de vue scientifique. Il semble cependant, et c'est là précisément que notre approche se situe bien comme une approche de programmation *par l'utilisateur* , que cet apprentissage, et l'activité de programmation, soient facilités par un environnement proche du travail de l'utilisateur. Cette proximité se décrit de deux façons : l'environnement de programmation doit être le même que l'environnement de travail (l'analyse de séquences par exemple), c'est-à-dire qu'il n'est pas nécessaire de sortir du programme ou de changer de contexte pour programmer, et les objets de l'environnement de programmation doivent être proches de ceux du domaine (ces proximités sont étudiées dans la seconde partie du chapitre III).

Nous n'avons pas pas trouvé de solution technique unique correspondant à notre situation, mais ce qui semble probable, c'est que plusieurs des démarches comportent des éléments intéressants, et que nous parvenions à une solution hybride (figure 11).

Ce qui est également important selon nous, c'est de mettre l'accent sur une démarche méthodologique : l'exploration technique apporte en effet des réponses importantes, mais il est également intéressant de situer ces hypothèses, et ce, non pas seulement parce que ce travail se déroule en entreprise, mais aussi parce que les situations spécifiques, comme nous avons essayé de le montrer, apportent leur propre lot de réponses ou plutôt *permettent de mieux comprendre les problèmes* .

A titre de conclusion, la figure 12 illustre la manière dont nous pensons que les concepts abordés dans ce chapitre sont liés entre eux.



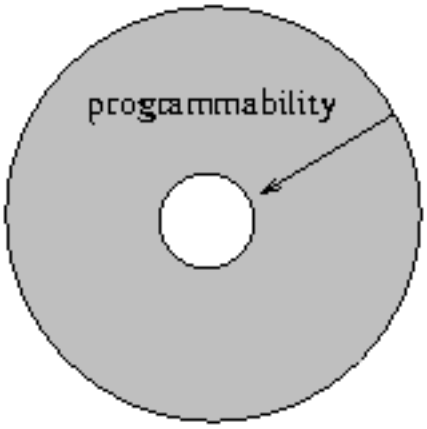


FIG. 1.12 – Solutions par le plein (PITUI) ou par le vide (bootstrap).

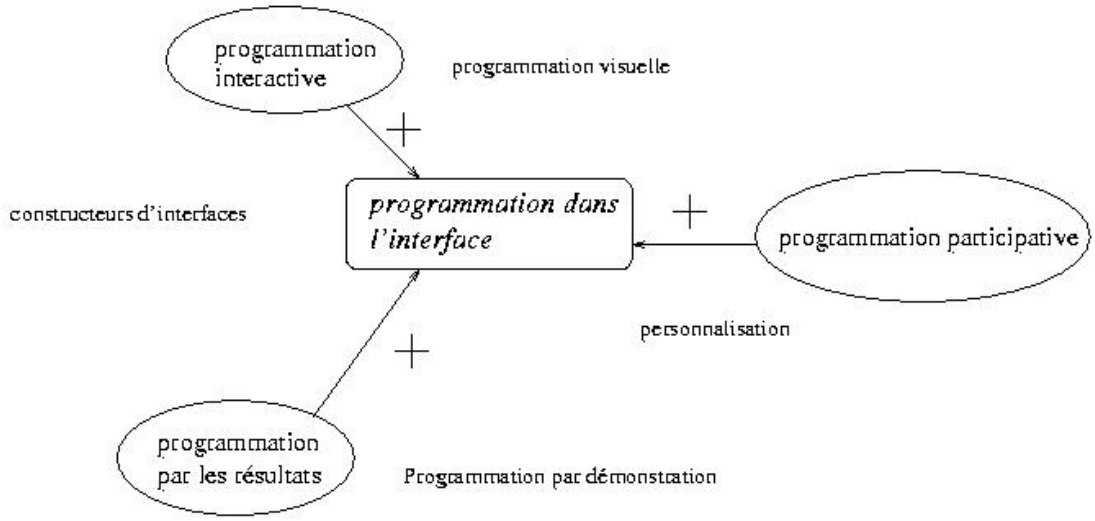


FIG. 1.13 – Programmation dans l'interface.

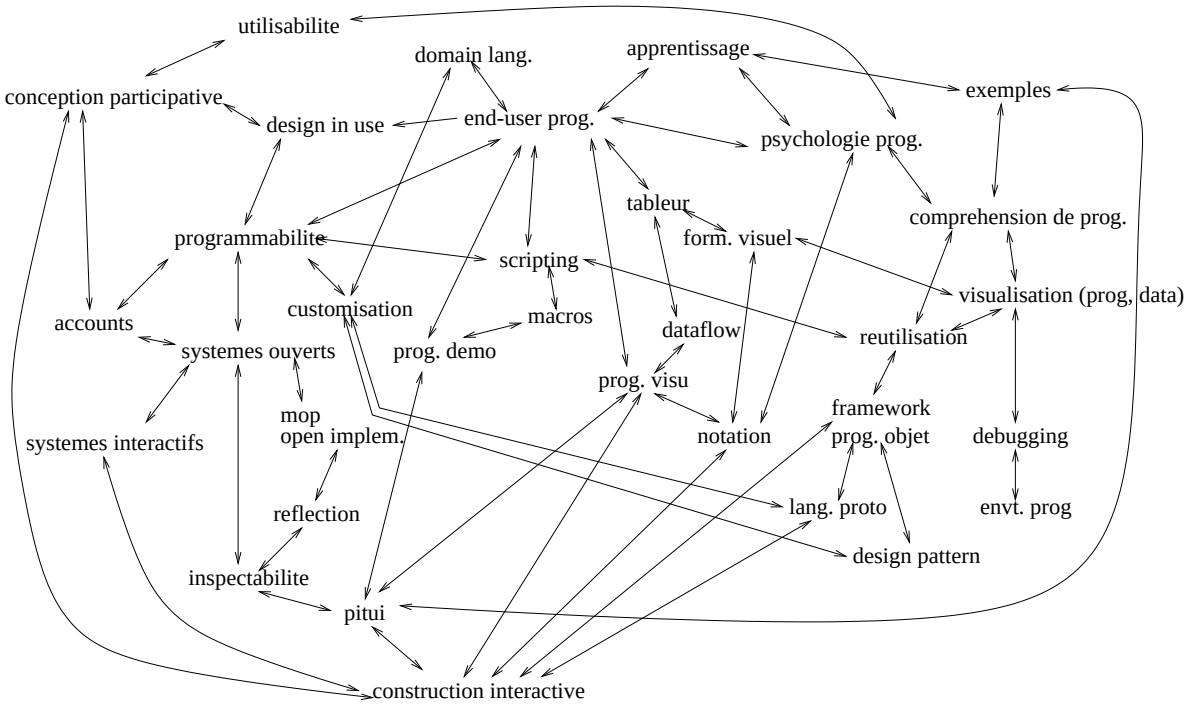


FIG. 1.14 – Carte conceptuelle.

Chapitre 2

Facteurs humains et programmation par l'utilisateur.

2.1 Introduction.

Ce chapitre a pour objet de décrire les méthodes qui orientent la problématique de la programmation par l'utilisateur vers l'utilisateur lui-même, par opposition aux approches plus techniques. Cette orientation recouvre des démarches très différentes, qui vont de la psychologie de la programmation à la programmation coopérative, en passant par les études d'utilisabilité dont l'objet est d'améliorer le logiciel sur un certain nombre de points définis d'avance.

Ces approches sont riches de renseignements, et il semblait utile de les évoquer en les situant dans une perspective où les utilisateurs ne sont ni des étudiants, ni des enfants, mais des chercheurs et des ingénieurs scientifiques. Une section est consacrée à la description précise de ce contexte, à partir des données d'une enquête réalisée sur la campus de l'Institut Pasteur. Parmi les études dont nous faisons part dans ce chapitre figurent également deux séries de tests logiciels, portant sur des développements locaux, dont nous avons voulu montrer l'utilité tant technique qu'organisationnelle.

L'approche que nous avons particulièrement développée est l'approche "participative", dont le principe est de faire émerger les informations utiles au développement du logiciel à partir de situations impliquant les utilisateurs plutôt qu'à partir de principes cognitifs ou ergonomiques. Cette démarche, peut-être paradoxale à première vue dans le contexte de la conception d'outils pour la programmation par l'utilisateur, est cependant riche de renseignements, car elle permet de placer la programmation par l'utilisateur dans une perspective plus globale.

2.1.1 Problèmes et difficultés de la programmation.

La prise en compte des facteurs humains dans le domaine de la programmation est assez ancienne, sous la forme d'études en psychologie de la programmation faisant appel aux connaissances en psychologie cognitive et en pédagogie.

En ce qui concerne la question qui nous intéresse - l'accès des non-professionels de l'informatique à l'une des techniques réputées les plus difficiles de cette discipline - ces études apportent des points de repère, d'abord parce qu'elles modélisent les différentes sortes de difficultés associées à l'écriture ou à la compréhension de programmes, ensuite parce qu'elles étudient expérimentalement les effets des différentes solutions.

Niveaux de problèmes

Pour mieux isoler les problèmes qui se posent dans la programmation occasionnelle par des non-professionnels, il faut distinguer les niveaux de difficulté. Ainsi, pour [LD89], on distingue trois niveaux : résolution de problèmes, conception (construction, découpage, organisation du logiciel), langage (syntaxe et modèle de calcul). [Dét98] précise que la programmation se modélise non seulement par la

résolution de problèmes, mais également par la problématique de la compréhension de texte. Pour [NZ93] il semble que la difficulté qu'il faut prendre en compte est plus celle de la modélisation du problème lui-même, que celle de sa traduction dans un langage. Le langage de programmation, supporté par un environnement correctement conçu et proposant les formalismes adéquats, constitue une aide à la réflexion, et non un objectif en soi.

Dans un autre ordre d'idées, plus pragmatique, [SBMP97] propose une démarche pédagogique où on distingue les difficultés de mise en oeuvre des difficultés théoriques. Il nous semble en effet que les difficultés auxquelles se heurtent les biologistes lorsqu'ils abordent la programmation sont d'ordre assez concrets. Il est tout d'abord plutôt rare d'avoir à traiter d'emblée un problème algorithmique du moins en dehors des structures d'apprentissage : dans la très grande majorité des cas, les problèmes pour lesquels on fait appel à la programmation sont des problèmes de formattage, d'analyse de texte, d'extraction de données et d'enchaînement de programmes.

D'autre part, nous avons pu constater que des personnes ayant suivi une formation de trois mois en programmation à l'Institut Pasteur, pendant lesquels elles ont appris et réellement apprécié les principaux concepts de programmation à travers l'apprentissage de langages comme Scheme ou Java, n'arrivent pas toujours à programmer dans leur contexte de travail par la suite. Sans vouloir généraliser sur un cas particulier, l'observation que nous pouvons en déduire est qu'il est effectivement difficile de mettre en oeuvre les outils et les concepts adéquats dans un contexte où il faut tout reconstruire à partir de rien. S'y ajoute certainement la difficulté de modélisation d'un problème dans un formalisme de calcul, mais notre hypothèse est que cette modélisation ne peut être que facilitée par un environnement conçu pour faciliter ce type de mise en oeuvre.

2.1.1.1 Tâches

Y a-t-il une spécificité de la programmation par l'utilisateur concernant les différentes tâches inhérentes à l'activité de programmation ?

La psychologie de la programmation distingue les tâches de construction et de compréhension de programmes. Sans doute, dans le cadre de la programmation occasionnelle et non professionnelle, la tâche de compréhension prend-elle une importance particulière (figure 1). En effet, et c'est d'ailleurs une approche que nous préconisons, ce type d'utilisateurs est plus souvent amené à lire des programmes et à en modifier de petites parties qu'à en créer à partir de rien [Bla00].

Compréhension

Les informations pertinentes sur la question de la compréhension de programme ont été décrites par [Pen87]. Ce sont : les opérations, les états du programmes (avec les tranches de temps), le flot de contrôle (la séquence des évènements), le flot de données (les relations entre les données ou les structures de données) et la fonction générale du programme.

Il y a des systèmes ou des langages qui s'attachent particulièrement à faciliter cette tâche, comme VEX (Visual EXpressions) [CHZ95] qui propose une formalisme visuel pour comprendre le lambda-calcul, ou Boxer [DiS97], un langage pédagogique fonctionnant sur la représentation explicite et spatiale des structures de données et sur l'analogie. Des approches rationnelles comme celle de [Mør94] sont aussi construites sur l'idée que pour modifier un programme il faut en comprendre le fonctionnement et matérialisent ce principe en établissant des design rationale (explications de conception) comme outil d'articulation et de navigation entre l'application et son code source. De nombreuses études portent spécifiquement sur cet aspect. [Dav00] étudie les objets sur lesquels se porte l'attention lorsqu'on regarde un programme pendant un temps très court, et remarque que les novices voient plus spécifiquement le flot de contrôle alors que les experts notent d'abord le flot de données. [Goo99] constate qu'il n'y a pas de méthode unique de compréhension, et que celle-ci est très influencée par la forme de la représentation du programme, qu'elle soit sémantique ou syntaxique. [Gre97] le confirme, tout en insistant sur l'importance de l'environnement et des outils de navigation, ainsi que sur l'opportunisme des sujets : les gens font ce qui est le plus facile et non le plus efficace. [OBCG99] compare, sur la question également de la compréhension, deux langages visuels, l'un fonctionnant sur le modèle flot de données, l'autre sur le modèle flot de contrôle et constate que le premier est plus long à apprendre mais qu'il donne de meilleurs résultats du point de vue du niveau d'abstraction et de la complétude des descriptions

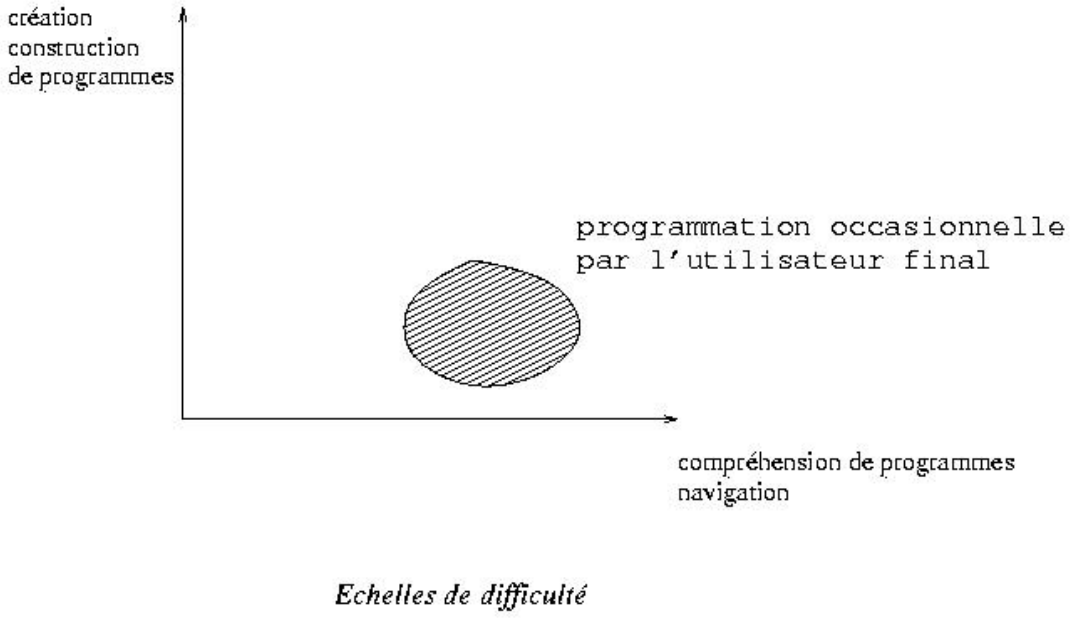


FIG. 2.1 – Tâche de compréhension, ou construction de programmes ?

(on a demandé un résumé aux sujets de l'étude), sans pour autant gêner la compréhension fonctionnelle. [Goo99] reconnaît également l'importance du paradigme de programmation mais [Gre97], avec le raisonnement du "match- mismatch" explique qu'aucune des solutions n'est globalement meilleure, et reconnaît l'adéquation de certaines "notations" à certains objets et non à d'autres. La prise en compte de l'importance de cet aspect a donné lieu au développement d'un outil d'analyse que nous présentons plus loin, dit cadre conceptuel de dimensions cognitives ("cognitive dimensions framework" ou CDF).

Réutilisation

La réutilisation est une activité majeure de la programmation par l'utilisateur, notamment l'utilisateur bio-informaticien, dans la mesure où ce dernier va assembler des éléments logiciels dans des scripts ou bien construire une application à partir d'une boîte à outils graphiques ou d'une bibliothèque de fonctions scientifiques. C'est pour cette raison que [vR99] insiste sur l'importance d'enseigner l'architecture logicielle dans un cours minimal de programmation : le besoin de plus important dans le futur sera selon lui l'assemblage de composants et la programmation de code colle. Cet aspect, aussi important selon lui que la possibilité d'adapter le logiciel, serait ainsi plus central que l'algorithmique. [WB97] décrit comment supporter la réutilisation en intégrant l'archive de code sous une forme évolutive et informelle dans l'environnement lui-même (ici Forms/3 [BAD+00]), avec des outils de recherche, des jeux de tests et de la documentation.

Résolution de problèmes

Comme nous l'avons évoqué dans le chapitre d'introduction, l'activité de programmation par l'utilisateur peut consister pour une part importante dans la définition itérative et exploratoire d'un problème mal ou insuffisamment défini.

2.1.1.2 Apprentissage

Comme [PAF84], qui montre que la récursion s'apprend difficilement par approximation, [BA99] qui constate l'inefficacité du bricolage sans l'acquisition préalable d'un modèle ou [LD89] qui souligne l'importance de l'apprentissage de la conception, nous pensons que pour la partie de la programmation concernant la formalisation de problèmes - l'activité de modélisation du problème ainsi que sa traduction dans un paradigme de calcul - un apprentissage conceptuel est nécessaire.

Limites et paradoxes de l'apprentissage.

Plus que la différence entre novices et experts qui intéresse la plupart des études, dans la mesure où leur objectif est souvent pédagogique, c'est la différence entre les personnes ayant l'apprentissage comme objectif explicite, comme les étudiants, et les personnes ayant besoin de l'apprentissage dans la poursuite d'un autre objectif qui nous semble importante.

D'après [CR87], la cognition humaine pose en effet deux problèmes pour l'acquisition de compétences techniques nécessaires à l'utilisation d'un système informatique, et cette analyse s'applique dans le contexte où la programmation n'est pas un objectif en soi. L'un est le biais de productivité : les adultes ont tendance à échapper à l'apprentissage pour obtenir le résultat attendu plus vite, même si ce résultat serait obtenu plus rapidement en lisant la documentation. L'autre est le biais d'assimiliation, qui fait que les compétences techniques tendent à une asymptote de niveau assez médiocre. Les utilisateurs des systèmes informatiques ne cherchent pas du tout à profiter des opportunités d'acquisition de nouvelles compétences, mais au contraire tendent à ré-utiliser le plus possible leurs connaissances, même si cela est inapproprié. Les solutions préconisées par les auteurs à ces problèmes cognitifs sont, non pas de changer l'utilisateur - ils insistent même sur le caractère inhérent et inévitable de ces deux biais - mais d'agir sur le plan de la conception des systèmes en développant des stratégies par rapport à ces tendances. Ils décrivent ainsi trois possibilités pour chacun des biais :

1. attaquer la tendance (développer les aspects ludiques, le mode aventure, le défi, faire valoir la récompense intellectuelle, détruire l'impression que les connaissances acquises sont utilisables, casser la métaphore s'il y en a une, ...),
2. en atténuer les effets (réduire la motivation nécessaire pour apprendre et réduire les risques de l'apprentissage, abaisser le niveau d'assimilation nécessaire, ...)
3. en tirer avantage (intégrer l'apprentissage dans la tâche, profiter de l'échec de la métaphore de manière explicite, ...).

Ces constatations concernant les biais cognitifs s'appliquent tout particulièrement au cas des programmeurs occasionnels, qui n'ont pas, comme ce serait le cas pour des étudiants en informatique devant obtenir leur diplôme et un travail, de raison de s'intéresser à l'apprentissage de la programmation. Par exemple, pour un scientifique en train de mener des analyses biologiques qui nécessitent la spécification de calculs originaux ou de manipulation de données, programmer n'est pas une activité intéressante en elle-même, sauf s'il désire se reconvertir professionnellement.

Dans une modélisation qui n'est pas cognitive, mais plus socio-organisationnelle, [Mac91a] analyse le compromis que constitue pour l'utilisateur la personnalisation de logiciel entre du temps perdu à ne pas faire son travail et l'anticipation du temps gagné à améliorer l'outil. Cet article décrit le processus de décision et identifie les facteurs qui favorisent ou empêchent la customisation : la pression sociale (exemples suggérés par ou vus chez d'autres), des évènements extérieurs (travail, déménagement, voyages, tremblements de terre ...), ainsi que des facteurs internes (temps libre, ennui, agacement), le changement de logiciel (souvent, on se mets à customiser pour gommer les changements). Le principal facteur de customisation est la découverte que quelque chose peut être automatisé ou pour faire disparaître un problème ennuyeux - nous avons souvent noté à quel point la programmation n'est adoptée qu'en cas de problèmes, au point qu'à notre avis cette composante doive faire explicitement partie d'un modèle de la programmation par l'utilisateur. Le principal facteur qui empêche la customisation est effectivement le manque de temps - d'autant plus qu'on ne veut pas risquer un investissement en temps inutile voire négatif. C'est aussi le manque de connaissance ou bien d'intérêt.

Ce que nous pouvons tirer de ces analyses, c'est que, pour autant qu'il s'agisse effectivement d'apprentissage - même si cet objectif n'est pas au premier plan, il reste toujours un peu présent dans une activité technique complexe - les approches doivent privilégier un apprentissage de la programmation contextualisé et pensé comme une activité continue tout au long de la vie professionnelle [DE95b]. Par contextualisé nous entendons un apprentissage situé dans un environnement, une application dont le fonctionnement est familier à la personne afin de ne pas superposer les problèmes de compréhension du domaine et ceux de la technique et de proposer un contexte motivant sur le plan professionnel [SBMP97][RCB90].

Exemples

Dans le cadre de la programmation non professionnelle, l'activité de réutilisation est assez proche

de l'utilisation d'exemples. En effet, les composants existants ont souvent une fonction d'exemples, et, s'il ne sont pas réutilisés comme tel, ils peuvent en revanche servir partiellement, par copier-coller. Cela peut paraître choquant de vouloir supporter une telle façon de programmer, qui entraîne du code dupliqué et qui n'est pas optimal de point de vue des mises à jour de versions, mais ce type de pratique est courant, et ne présente pas les mêmes inconvénients dans le cadre de programmes individuels ad-hoc et temporaires. D'ailleurs, la mise en oeuvre de pratiques industrielles de réutilisation logicielle est complexe et doit, pour fonctionner correctement, reposer sur un organisation solide et contraignante. Enfin, l'utilisation d'exemples a une fonction pédagogique, surtout si elle est explicitement supportée dans l'environnement, comme dans [Nea89], où l'éditeur est divisé en deux fenêtres, l'une pour le programme, l'autre pour l'exemple, et où les outils de navigation sont optimisés pour la recherche de patterns algorithmiques. Les exemples fournis par cet environnement sont pris dans des cours de programmation, mais dans une application personnalisable et extensible, les meilleurs exemples sont ceux de l'application elle-même, qui ont pour avantage d'être centrée sur le domaine [Mør97b].

Les bibliothèques de cas ("case libraries") généralisent cette idée, comme STABLE (Smalltalk Apprenticeship-Based Learning Environment) [KG98], qui est une bibliothèque d'études de cas de conception et de programmation objet. Cette bibliothèque est surtout utilisée dans un cadre d'apprentissage il est vrai, plus exactement l'apprentissage fondé sur un projet (Project-Based Learning). [LC92] explique l'intérêt des études de cas pour initier les novices à la conception, souvent délaissée selon cet auteur. Chaque étude de cas comprend le titre du problème, une description pédagogique de la manière dont un expert a résolu le problème, le code de l'expert - ce qui fait d'ailleurs fortement penser aux patrons de conception avant l'heure [GHJV95]. Cette démarche s'oppose à l'enseignement de l'informatique orienté syntaxe mais en même temps à l'inefficacité de la démarche descendante qui est souvent enseignée mais irréaliste. Un des résultats d'études menées dans le cadre de cette recherche montre qu'on apprend plus avec un commentaire d'expert qu'en programmant tout seul, notamment pour la capacité à généraliser. Ainsi, dans les domaines de la médecine et de la gestion, les études de cas se pratiquent beaucoup pour apprendre les idées complexes et interdépendantes du contexte. Cette démarche cadre par ailleurs assez bien avec l'idée défendue par les approches de personnalisation de logiciel, qui insistent sur l'importance, quand cela est possible, de maintenir un contact entre les auteurs et les utilisateurs.

2.1.1.3 Psychologie ou design ?

Ce qui caractérise l'approche cognitive, c'est d'être focalisée sur les capacités humaines, mais en s'orientant assez nettement vers l'adaptation de l'humain au modèle calculatoire, à travers une démarche pédagogique (figure 2).

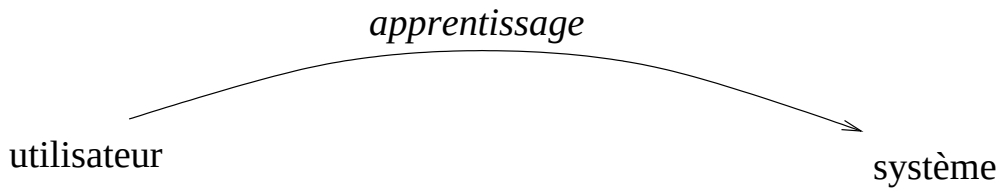


FIG. 2.2 – Adaptation par l'utilisateur

Une autre approche, qui ne serait d'ailleurs pas nécessairement opposée, mais complémentaire, serait de considérer que c'est au système informatique de s'adapter le plus possible à l'utilisateur (figure 3).

FIG. 2.3 – Adaptation par le système

Il y a plusieurs sortes d'arguments pour appuyer cette orientation ou accentuation différente :

- une motivation de principe, suivant par exemple l'idée de la conception centrée sur l'utilisateur,
- une motivation de type cognitive, comme celle que nous avons décrite plus haut, qui reconnaît les limites de l'apprentissage et pour laquelle l'approche préférable est dans la conception de systèmes mettant en oeuvre des stratégies de contournement des biais cognitifs.

Hypothèse de la complexité accidentelle : facilité ou accessiblité ?

Le premier argument n'est pas seulement un argument de principe, mais repose aussi sur l'hypothèse qu'il y a toujours, dans un système informatique, une part de la complexité qui est accidentelle et qui peut être éliminée, ou atténuée [Bro87]. On pourrait ainsi distinguer trois niveaux de complexité :

1. la complexité intrinsèque (celle du problème),
2. la complexité extrinsèque (celle de l'outil),
3. la complexité accidentelle (celle qu'on pourrait éviter au niveau de l'outil).

C'est l'approche de [Rep93a] qui élargit le cadre d'analyse des difficultés de la programmation à un modèle à trois composantes : l'outil, le problème, la personne (figure 4).

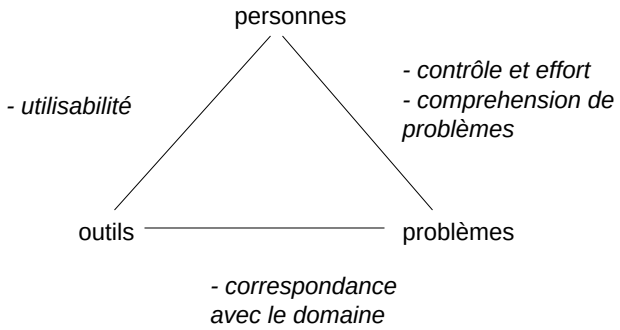


FIG. 2.4 – Personnes, outils, problèmes

La relation entre personnes et outils caractérise l'utilisabilité de l'outil, indépendamment de son utilité par rapport au problème à résoudre. C'est sur cet axe que se situent les solutions d'ordre syntaxique ou visuel, ou encore le caractère plus ou moins flexible des notations comme celle des tableurs. La relation entre outils et problèmes décrit l'orientation par rapport au domaine, c'est-à-dire l'adaptation d'un outil à la résolution d'un problème spécifique, comme dans les langages orientés domaine (PostScript, HTML, ...).

Les langages à objets se placent assez bien à la fois sur l'axe personnes-outils et sur l'axe outils-problèmes, parce que leur notation permet un "mapping" plus naturel - bien que cela soit fortement discuté - entre les entités de l'outil (les classes) et la compréhension des problèmes par les personnes.

Enfin la relation personnes-problèmes décrit la compréhension et la résolution d'un problème par les personnes : c'est ici que se placeraient les problématiques de psychologie de la programmation, mais on voit que ce modèle intègre une interaction potentielle avec les outils, puisque souvent c'est par son implémentation avec un outil qu'une modélisation peut progresser.

2.1.1.4 Orientation choisie

Pour les raisons que nous venons d'expliquer, mais aussi parce que cette étude n'a pas été réalisée dans un laboratoire de psychologie cognitive ou expérimentale, mais dans une entreprise dont les utilisateurs de l'informatique sont des collègues et des scientifiques de haut niveau, nous avons opté pour une approche qui privilégie une hypothèse plus pragmatique. Selon cette hypothèse (figure 5), il y a d'autres démarches à tenter avant d'invoquer les sciences cognitives ou éducatives :

- concevoir un environnement de programmation utilisable,
- concevoir cet environnement avec les utilisateurs afin d'en optimiser la pertinence,

– situer la programmation dans le contexte où elle se produit généralement, c'est-à-dire dans le cadre d'un travail qui n'est pas nécessairement la construction d'un programme.

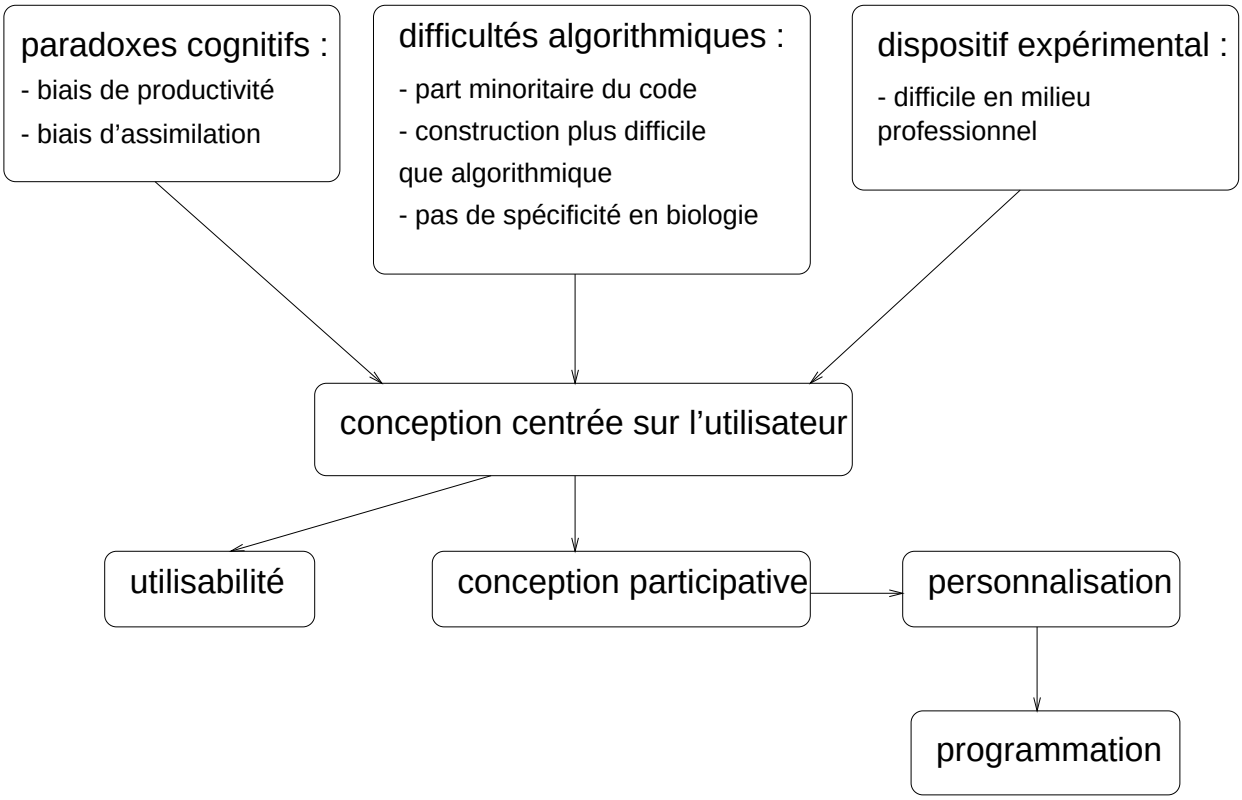


FIG. 2.5 – Choix

2.2 Point de vue de l'utilisabilité.

2.2.1 Études d'utilisabilité en programmation.

2.2.1.1 Études au niveau des notations

L'article qui fait référence est [Gre89] qui décrit les *dimensions cognitives des notations* (CDF : Cognitive Dimension Framework), un cadre conceptuel multi-dimensionnel de variables pertinentes pour l'évaluation de l'utilisabilité d'artefacts informationnels, autrement dit de tout système d'information incarné à travers une notation ou une structure. Le terme "cognitif" est employé pour situer les variables comme étant pertinentes pour l'apprentissage ou l'utilisation, mais c'est bien les systèmes eux-mêmes qui sont décrits par ces variables, et non les mécanismes intellectuels.

Les dimensions sont les suivantes :

- niveau d'abstraction : nombre de concepts de haut niveau qu'il faut apprendre avant de pouvoir utiliser un système (exemple : expressions régulières) ("abstraction level"),
- proximité entre le monde du problème et le monde du programme ("closeness of mapping"),
- consistance : des sémantiques similaires sont exprimées dans des formes syntaxiques similaires ("consistency"),
- caractère diffus des notations : répandues dans toutes les directions, en opposition à la concision, verbosité du langage ("diffuseness"),
- risque d'erreur dûe à la notation (risque d'erreur) ("error-proneness"),
- difficulté des opérations mentales : au niveau des notations, comme au niveau de leur enchevêtrement ("hard mental operations"),

- dépendances cachées (perceptives ou symboliques) et localité ("hidden dependencies"),
- décisions prématurées : contraintes dans l'ordre pour faire les choses ("premature commitment"),
- évaluation progressive : possibilité de s'arrêter pendant une tâche, par exemple en cours d'édition, et possibilité de contrôler ce qui a été fait jusque là (progressive evaluation),
- expressivité des rôles : lisibilité des rôles d'un composant ("role-expressivness"),
- notations secondaires : notations ne faisant pas strictement partie de la structure syntaxique, mais introduites pour aider à la perception de la sémantique ; exemple : commentaires, indentation ("secondary notation"),
- viscosité : effort nécessaire pour le moindre changement ? ("viscosity"),
- actions provisoires : degré de décision par rapport à des actions ou des notes ; possibilité de noter quelque chose dans une notation intermédiaire et non définitive ("provisionality"),
- visibilité et juxtaposabilité : possibilité de visualiser des composants aisément, ou de les comparer ("visibility", "juxtaposability").

Ces dimensions cognitives n'ont de sens que situées dans le contexte d'une activité, pour laquelle elles prendront plus ou moins d'importance. Les activités, décrites comme génériques, sont :

- l'incrémentation : ajout d'information sans altérer la structure,
- la modification : changement d'une structure existante, éventuellement sans ajout d'information,
- transcription : copie de contenu d'une structure dans une autre structure,
- conception exploratoire : combinaison des activités d'incrémentation et de modification, sans que le résultat final ne soit connu d'avance,
- recherche d'information,
- compréhension/découverte exploratoire (de la base d'une classification, de la structure d'un algorithme, ...).

Les artefacts sont composés génériquement de la manière suivante [Gre00] :

- d'une *notation* ou, lorsque cette dernière est invisible, d'un *langage d'interaction* (ce que nous avons appellé "notation active" dans le chapitre d'introduction),
- d'environnements d'*édition* : les éditeurs, chacun possédant sa propre ontologie (réseau de concepts pertinents), sont le support de la création et de la manipulation des abstractions (équivalents des instruments au sens de [BL00]),
- un *médium d'interaction* : le papier, l'écran, ... qui se caractérise par son degré de persistence et le degré de contrainte sur l'ordre ("constrained-order/free-order").

Les activités spécifiques correspondantes dans la programmation par l'utilisateur ou la personnalisation sont par exemple :

- incrémentation : ajout d'une formule dans une feuille de calcul, écriture d'un script pour composer des fonctions fournies par l'environnement avec un mécanisme de composition prêt à l'emploi ou en reprenant un autre script similaire (il y a alors aussi transcription) ;
- modification : lorsqu'il s'agit d'adapter un composant existant ; l'activité de modification nécessite des mécanismes d'abstraction et suppose résolus les problèmes de compréhension de logiciel ; pour pouvoir modifier, la fluidité (non viscosité) prend une importance particulière ;
- conception exploratoire : c'est le cas dès qu'il s'agit de construire une fonction nouvelle qui ne ressemble à aucune autre, ce qui relève d'une activité de modélisation ou d'invention ;
- transcription : conversion de format (si cela est possible, ce type d'activité doit être supporté par l'environnement).

Les variables importantes, dans le cadre de la programmation novice ou occasionnelle (correspondant à une activité de conception exploratoire essentiellement) sont [Gre00] :

- une faible viscosité : un changement simple et incrémental doit être possible sans induire de complications par ailleurs,
- pas trop de décisions prématurées : il faut pouvoir commencer quelque chose sans connaître tous les éléments d'avance,
- haute visibilité : il est important de pouvoir visualiser tous les éléments entrant en jeu : les composants nécessaires, le changement induit par une action,
- une bonne expressivité de la notation : cela est nécessaire pour la compréhension du rôle des objets.

On peut y ajouter la proximité problème-programme si l'on considère qu'il est important que le langage soit orienté vers le domaine du problème.

L'étude de [Gre00] porte sur les dimensions de viscosité et d'expressivité des rôles du langage Prolog, dans le contexte d'une tâche typique de programmation incrémentale : modifier une fonction qui calcule une moyenne en une fonction qui calcule une moyenne filtrée (on retire les valeurs nulles). Pour un langage, l'expressivité des rôles c'est la possibilité de remonter du code aux schémas cognitifs originaux du programmeur (les "plans" qu'il avait en tête et qu'il a traduit sous forme d'une implémentation [Dav90]). Il s'avère que, si un code en Prolog est relativement peu "visqueux" (on peut en modifier une partie bien localisée sans avoir à changer d'autres choses), il est cependant beaucoup plus difficile à décomposer qu'un programme écrit en Pascal : il y a très peu d'éléments notationnels en Prolog, notamment pour les structures de contrôle, mais plutôt des réarrangements différents de ces éléments notationnels. L'absence de distinction lexicale entre des structures de calcul très différentes rend difficile la détection et la différenciation des schémas algorithmiques, alors qu'en Pascal des indices lexicaux comme while et for aident beaucoup.

Concernant les tableurs, toujours dans cette étude, le verdict est assez dur : il bloquent l'abstraction, ont une mauvaise expressivité quant aux rôles, ils sont l'occasion de nombreuses dépendances cachées, et rendent la compréhension exploratoire difficile. Cependant, ils sont très utiles pour l'incrémentation.

Une autre étude, proposée pour un workshop sur la programmation par l'utilisateur *final* ou *intermédiaire* ("blended") porte sur les tableurs et sur la programmation de scripts [Gre99] :

- Tableurs : il s'agit surtout d'une activité d'incrémentation (ajouter un même type de matériau que sont les données et formules), et il y a peu de modifications (c'est-à-dire de changement de la structure). Il s'agit d'une activité similaire à la programmation par des novices mais avec une forte orientation cependant vers un but ou une tâche. Dans ce type de programmation, il est conseillé d'éviter d'obliger l'utilisateur à prendre des décisions prématurées ou à maîtriser des abstractions qui posent rapidement un problème de compréhension - surtout pour la communication de feuilles de calcul au sein d'une organisation. Les propriétés essentielles sont la proximité au problème ; les dépendances cachées et la verbosité (comme dans HyperTalk) ne sont pas trop gênantes car les activités de modification et de recherche sont rares.
- Pour l'activité d'écriture de scripts par des utilisateurs intermédiaires, c'est un peu différent : la compréhension est importante notamment pour le partage de scripts. Les actions qui ne sont pas orientées directement et immédiatement vers un but sont possibles (l'utilisateur peut investir dans le futur) mais des modifications importantes sont inacceptables, à cause de l'investissement que cela représenterait. L'abstraction n'est pas à éviter, surtout si cela permet une meilleure compréhension. Comme la compréhension est importante, de bonnes possibilités de notation secondaire sont utiles. S'il s'agit de réutilisation, la proximité programme-problème a moins d'importance, mais les aspects de la notation induisant des erreurs ou créant des dépendances cachées moins tolérables.

Dans notre contexte, il y a plusieurs types d'artefact à évaluer : cela va de l'interface graphique, c'est-à-dire l'interface de programmation (éditeur, navigation), à l'architecture elle-même et aux concepts sous-jacents dont nous avons parlé dans le chapitre d'introduction, dans la mesure où les possibilités (au sens de faisabilité) de programmabilité dépendent de ces derniers.

2.2.1.2 Études au niveau des paradigmes

L'objectif n'est pas ici de donner un tableau même général de ce type d'études, mais d'en donner un ou deux exemples, portant notamment sur la programmation par objets, modèle que nous avons choisi pour le prototype biok. Ces exemples montrent que le sujet reste assez polémique, ce qui se produit bien sûr inévitablement lorsqu'on recherche une solution générale pour toutes les situations de programmation.

De nombreuses études comparent la programmation par objets avec l'approche procédurale. [ADST00] étudie l'utilisabilité du paradigme objet pour la construction et la compréhension. Les tests sont différenciés selon le type de tâche et l'expérience. Les tâches sont classées comme orientées objet ou comme procédurales selon l'importance accordée au processus et à leur séquencement et selon la densité et

la complexité des structures de données. Les conclusions sont que, pour les novices, la construction objet d'un problème objet est supérieure à la construction objet d'un problème procédural, mais que du point de vue de la compréhension, la représentation procédurale est plus facile que la représentation objet pour les sujets semi-expérimentés. Selon les auteurs, l'introduction de méthodes objet dans les entreprises nécessite de bien choisir les exemples de tâches pour la formation.

Une autre étude, [Dav00] se donne pour objectif de répondre à la question : où porte-t-on son attention quand on a très peu de temps (les programmes sont donc présentés aux sujets pendant un très court instant). Résultats : les experts extraient plus d'informations, mais cela dépend des catégories (surtout pour la fonction, l'opération, l'état, plus que pour les flots de données ou de contrôle). Les novices sont plutôt perceptifs au flot de contrôle. Dans le même ordre d'idées [DGG95] évalue l"'importance" des objets pour la manipulation et la compréhension de programmes. Les sujets doivent classer librement des fragments de codes. La conclusion est que la programmation objet n'a rien de "naturel", et que les experts, tout comme les bons joueurs d'échecs, sont capables de reconnaître des patterns par fragments entiers ; par ailleurs une des prédictions est que les experts ont beaucoup de connaissances communes (souvent axées sur des entités dérivables syntaxiquement : constructeurs, gérant d'évènements, inlines,...). Les niveaux de programmation semblent être corrélés à la capacité d'organisation. Une des surprises de l'enquête est que les experts perçoivent plus ou s'intéressent plus à l'aspect fonctionnel (type de valeur de retour, action effectuée, calcul), à la structure interne, qu'à l'aspect objet (relations entre les objets, classification, hiérarchie, appartenance d'un fragment à une classe). Mais les fragments proposés étaient essentiellement des méthodes, moins des définitions de classes. Inversement, les novices ont eu une approche plus orientée objet, peut-être pour faire plaisir à l'expérimentateur, ou parce que le cours n'était pas loin. Peut-être la classification objet est-elle plus évidente pour les experts, qui passent rapidement à autre chose une fois que la décomposition de l'application aurait été mentalement effectuée (afin de décharger la mémoire).

De manière générale, les méthodes objets ne semblent pas beaucoup plus faciles à aborder pour les novices, bien qu'elles aient la réputation de permettre une meilleure correspondance entre le domaine du problème et les techniques de conception et de programmation. Cela est lié d'abord aux difficultés d'abstraction, mais beaucoup aussi aux problèmes de navigation dans les relations entre les objets, dont les dépendances sont le plus souvent cachées, à cause des mécanismes d'héritage, du polymorphisme et de la liaison dynamique [CSBA90]. S'y ajoutent dans notre cas un niveau d'héritage supplémentaire dû à la possibilité de définir les méthodes au niveau de l'objet (peu utilisé cependant pour les objets appartenant à une classe applicative), et des dépendances associées aux filtres et aux mixins. Pour cette raison, nous avons développé des outils de navigation dans la hiérarchie de classe et des outils de visualisation des interactions entre objets, similaires à ceux de [CSBA90].

Des études comme [GBC97] étudient l'impact du modèle de calcul (en opposant flot de données et flot de contrôle) sur la compréhension. Cette étude montre par exemple que les novices ont tendance à utiliser des informations de détails et de bas niveau ; c'est ensuite, après un certain temps, qu'on arrive à une représentation du modèle du domaine du problème. Plus précisément, les explications procédurales dominent, puis fonctionnelles, puis dataflow. Le langage influe effectivement sur ce résultat, mais le procédural est quand même présent dans les expériences avec des langages logiques ou dataflow.

2.2.1.3 Études au niveau d'un langage

[Mye00] applique le principe de l'évaluation heuristique de [NM94] aux langages C++, Visual Basic, C et Python. Les variables mesurées sont :

- la conception graphique, qui s'applique à la disposition et aux couleurs par exemple dans un éditeur syntaxique, et au support de l'indentation, soit dans l'éditeur, soit dans le langage (Python) ;
- la simplicité :
  - en C++, il y a trop de fonctions et des interactions imprévisibles entre elles ;
  - en C, les 16 niveaux de précédence sont complexes à apprendre ou à mémoriser, et il y a trop de moyens d'exprimer la même chose (exemple : a++ et a+=1) ;
  - des constructions comme void pour non, ou char pour byte, ou encore des listes qui commencent à 0 et non 1, les deux signes == et = ne sont pas vraiment dans le "langage de l'utilisateur",
  - l'existence d'espaces de noms communs pour les fonctions, tableaux, scalaires peut induire des

confusions ;
- correspondance avec les habitudes ou le monde réel : l'expression a = a+1 est très surprenante ;
- facilité pour la mémorisation : dans tous les langages, il faut se souvenir de toutes les fonctions et de leurs paramètres ;
- cohérence : static veut dire 3 choses différentes en C++ ; les séparateurs en C++ sont différents : " ;" ou "," qu'il s'agit des en-têtes de procédures ou du passage de paramètres ; ces derniers se font par référence ou par défaut selon les types de variables ;
- "feedback" : il n'y en a aucun dans la plupart des environnements avant la compilation ou l'exécution, sauf en Visual Basic où des boîtes de dialogue de confirmation peuvent apparaître pendant qu'on tape du code ;
- gestion des erreurs : il faut bien sûr les empêcher, et s'il en a, afficher des messages conpréhensibles et informatifs ;
- raccourcis : il y en a trop peu, et il y a en général trop de choses à faire pour créer un petit programme.

2.2.1.4 Études de langages et d'environnements de programmation visuelle

Le domaine des langages visuel est un haut lieu de débats et de compétitions, comme celle qui est décrite dans [Han94]. Les études d'utilisabilité sont donc nombreuses. [BRL91] applique la technique du walkthrough à l'évaluation de la facilité de traduction d'une tâche dans un programme visuel. [MCM97] effectue une évaluation empirique d'un shell graphique. [Bla00] présente une évaluation d'idées de programmation visuelle : la représentation visuelle et la génération interactive d'expressions régulières. C'est une idée à appliquer sans hésiter d'ailleurs en analyse de séquences : les biologistes voudraient par exemple pouvoir choisir un ensemble corrélé de données (par exemple certains sites d'un alignement de séquences) et s'en servir pour effectuer une recherche du motif inféré soit dans le reste de l'alignement, soit dans une banque de motifs répertoriés.

Plusieurs études dans ce domaine visent à comparer les techniques visuelles et les techniques textuelles (et n'ont peut-être pas vraiment pour but d'évaluer l'utilisabilité d'un système particulier, mais plutôt la pertinence d'une idée). [GP97a] applique le framework de dimensions cognitives à la programmation visuelle, en comparant notamment LabView et Prograph aux langages C et Pascal. On peut citer également : [PB93] qui compare la manipulation textuelle et visuelle de matrices, [GB97] qui compare la programmation d'objets complexes dans un tableur sous forme de gestes directs et de formules, [GP92] qui analyse les cas où la notation visuelle est plus difficile à comprendre que la notation textuelle, [GB96], qui cherche à déterminer si les représentations graphiques facilitent l'apprentissage de la récursivité aux novices.

2.2.1.5 Études de langages et d'environnements de programmation pour l'utilisateur

Sans faire l'objet de l'article, l'utilisabilité est mentionnée dans [vR99] qui soulève le problème des outils et des environnements de programmation qu'exigeraient une programmation accessible à tous. Selon cet auteur (c'est l'auteur du langage Python), il ne s'agit pas tant de rendre la programmation facile que de la rendre accessible. Ce texte évoque notamment le développement d'un environnement de programmation : IDLE pour Python, avec des possibilités diverses, dont un "Undo" incrémental ou même sélectif dans le code et dans l'exécution, et souligne l'importance des exemples spécifiques très orientés domaine. Un plan de cours de programmation est proposé, dans lequel l'accent est mis sur les connaissances générales, comme l'architecture matérielle ou logicielle, afin de maîtriser les techniques de composition logicielle considérées comme l'une des activité de programmation les plus importantes, ainsi que sur les aspects pratiques et exploratoires de la programmation (débogage et résolution de problèmes). On remarque l'absence inhabituelle de l'algorithmique et l'intérêt accordé aux aspects pragmatiques. L'importance de l'environnement de programmation et des aspects pratiques de la programmation est souligné également par [RCB90], qui remarque que les livres pour apprendre Smalltalk parlent assez peu de l'environnement et pratiquement pas de l'interface utilisateur. Le résultat, c'est qu'il faut des semaines d'apprentissage avant de pouvoir commencer à faire quelque chose. [CSBA90] est une mise en application de ces idées à travers l'outil ViewMatcher, utilisé au cours de sessions progres-

sives destinés à des programmeurs amateurs connaissant le modèle fonctionnel, mais pas la construction d'applications. Ces sessions sont organisées en 5 blocs correspondant aux étapes suivantes :

1. interaction avec le Workspace, essais d'envoi de messages ;
2. étude du modèle MVC ; aucun code n'est encore visualisé, la trace des envois de messages dans la pile d'exécution est inspectée ;
3. exploration de la partie modèle de MVC ; compréhension et essais de modification du code des méthodes ;
4. les parties vue et contrôle sont abordées ; modifications de méthodes ;
5. interactions libres, essais de plusieurs applications, avec le code ; tâche libre.

[Gir93] expose les résultats d'études utilisateurs qui proposent des tâches libres de modification d'un logiciel, Modifier, pour des concepteurs (ici, l'exemple porte sur la conception de cuisines). Le logiciel est construit sur une base de connaissances, dans laquelle l'utilisateur peut entrer de nouvelles règles. Dans les tests, il s'agit de voir si les utilisateurs proposent des modifications impossibles, étant données la structure de cette base de règles. Ce fut le cas, il a été notamment impossible d'ajouter une règle : "il faut une porte", car toute règle devait être associée à un objet. Une autre modification s'est montrée impossible : ajouter une grille de dessin. Le logiciel Modifier permet la création de nouveaux objets de conception (un freezer par exemple), par recopie ou non d'autres unités et avec une aide à la définition du nouvel objet dans la hiérarchie de classes. Les tests ont cependant montré que les concepteurs ne sont pas très à l'aise avec les termes de classe et relation.

[Hen95] est une étude basée sur la technique de l'incident critique dans le domaine des tableurs : les utilisateurs avaient pour tâche de diviser une feuille de calcul en plusieurs blocs, auxquels une fonction différente devait être appliquée.

[TCJ92] est une étude d'utilisabilité sur l'environnement HyperCard, qui constate de nombreuses incohérences et qui remarque que l'orientation objet, telle qu'elle est appliquée dans HyperCard, ne résoud pas tous les problèmes.

[PRM00] présente une étude générale en vue de la recherche sur la programmation naturelle. La définition de la programmation qui correspond à cette étude est l'activité de transfert d'un plan mental en code. Le but de cette recherche est d'utiliser les formulations de non informaticiens comme indications d'améliorations possibles des langages de programmation actuels afin de réduire la transformation nécessaire pour programmer. Les tests (2 séries) consistent à demander à des non-programmeurs de formuler des solutions à des problèmes. Le premier groupe était constitué d'enfants et le problème était autour d'un logiciel de jeux. Le second était un problème de rangement dans une liste et les participants des adultes de formation avancée en sciences. Dans cette étude, on a veillé à ne pas influencer les participants par une formulation verbale des problèmes qui aurait influencé l'expression des solutions. Aucune contrainte formelle n'a été donnée pour les solutions (nombre d'étapes, longueur des phrases,...). Un ensemble de concepts représentatifs de la programmation a été déterminé pour définir le matériel de l'étude : variables, affectation, initialisation, comparaisons, incrémentation de compteurs, arithmétique de base, itération, recherche, tri, animation, parallélisme, collisions et interactions entre objets, input utilisateur. Des innovations potentielles ont été relevées, en remarquant par exemple l'importance des opérateurs sur les aggrégats, des conditions avec exception (if A unless B serait mieux compris que if A and not B). En général, la négation est évitée. Les auteurs constatent que, dans les expressions mathématiques, la variable désignant l'objet concerné est souvent omise et que les variables d'état sont remplacées par des verbes au passé ou au futur. 2/3 des participants ont utilisé des formulations graphiques (diagrammes ou images) dans l'expression de leur solution. Une remarque : l'étude constate une nette dominance du style de programmation évènementiel, ou à base de règles de production, et reconnaît aux environnements HyperCard, VisualBasic et Lingo d'avoir adopté ce style qui correspond bien au modèle mental "naturel" des non-informaticiens. Le langage de programmation d'HyperCard est d'ailleurs cité comme un des exemples de langages naturels. Mais seules des mises en garde et des critiques sont énoncées à son sujet. La base des programmes écrits dans ces environnements adoptés par des dizaines de milliers de programmeurs amateurs ne serait-elle pas aussi un puit d'informations sur la programmation naturelle ? Ne faudrait-il pas se demander pourquoi ces langages sont adoptés par tant de personnes ?

Catégorie	Nombre	Hommes	Femmes
Technicien	61	27.9%	72.1%
Ingénieur	49	30.6%	69.4%
Chercheur	238	58.8%	41.2%
Stagiaire	189	46.6%	53.4%
Administratif	22	4.5%	95.5%

TAB. 2.1 – Catégorie professionnelle

Macintosh	557	96.0%
PC	164	28.0%
terminal X	149	25.6%
station de travail	63	10.8%

TAB. 2.2 – Type de terminal utilisé

2.2.2 Études réalisées dans le cadre de la thèse.

2.2.2.1 Enquête sur l'utilisation de l'informatique scientifique

Cette étude s'est déroulée durant l'hiver 1996 à l'occasion d'un projet pour l'obtention d'une unité de valeur d'analyses de données au CNAM. Il s'agissait de réaliser une enquête à l'aide d'un questionnaire sur l'utilisation de l'informatique à l'Institut Pasteur et d'en faire, pour le mémoire, une analyse avec des outils comme l'analyse de correspondances multiples, adaptée au dépouillement d'enquêtes.

Le questionnaire a été préparé avec l'aide de deux ergonomes professionnels de la CERMA/IMASSA (Bretigny) qui ont rencontrés une quinzaine de biologistes du campus et aidé à l'établissement d'un questionnaire de test, évalué auprès d'une vingtaine de personnes.

Le questionnaire final, envoyé nommément à un millier de personnes, comportait 3 pages de questions portant sur l'expérience en informatique, avant et depuis l'arrivée à l'Institut Pasteur, la formation, avant et depuis l'arrivée à l'Institut Pasteur, le type de logiciels utilisés, les problèmes rencontrés, l'environnement informatique utilisé, l'utilisation de la documentation, la démarche pour trouver de l'aide, les opinions. Le questionnaire se terminait par un petit Quizz. Les personnes pouvaient répondre de manière anonyme en retirant la première page du questionnaire qui contenait leur adresse.

La saisie des réponses (600 réponses) a pris plusieurs semaines et a mis à contribution une bonne partie des membres du service informatique. L'analyse en elle-même a pris environ un mois, en incluant le traitement préalable des données, la mise au point des variables pertinentes, l'apprentissage du logiciel (il fallait se connecter sur la machine VAX/VMS d'un institut (IIE) partenaire du CNAM et utiliser un logiciel assez complexe qu'il fallait paramétrer avec des fichiers de commandes), et le traitement des résultats. Cela a été l'occasion de constater la difficulté de manipulation des paramètres de ce type d'outil d'analyse. Les résultats finaux ont été publiés dans le bulletin du service d'informatique, le B6 [Let99c].

Données

Les tableaux 1, 2, 3 et les figures 6, 7, 8 et 9 représentent une partie des données recueillies, et nous semblent pertinents pour situer le reste des études décrites dans ce chapitre puisqu'ils décrivent grossièrement l'environnement de travail des biologistes du campus : plate-forme, logiciels scientifiques, ainsi que leurs problèmes et leur façon d'apprendre l'informatique (aide, documentation, ...). Il ressort de ces tableaux que la principale ressource en cas de problèmes est un "gourou" local, que les problèmes sont fréquents, qu'il se situent plutôt du côté de l'informatique technique (réseau, éditeur de texte), mais que presque la moitié des personnes notent leur "trucs" et lisent la documentation.

Résultats

La figure 10 représente la meilleure projection en deux dimensions des variables de l'enquête, calculée par la méthode de l'analyse des correspondances multiples qui est une méthode qualitative fondée sur les corrélations entre les variables. Cette figure laisse apparaître une répartition des utilisateurs en sept

Analyse de séquences	292	50.9%
Assemblage de séquences	134	23.1%
Modélisation moléculaire	40	6.8%
Phylogénie	54	9.3%
Analyse génétique	66	11.3%
Recherche bibliographique	438	75.5%
Statistiques	71	12.2%
Outils Unix	105	18.1%

TAB. 2.3 – Logiciels utilisés

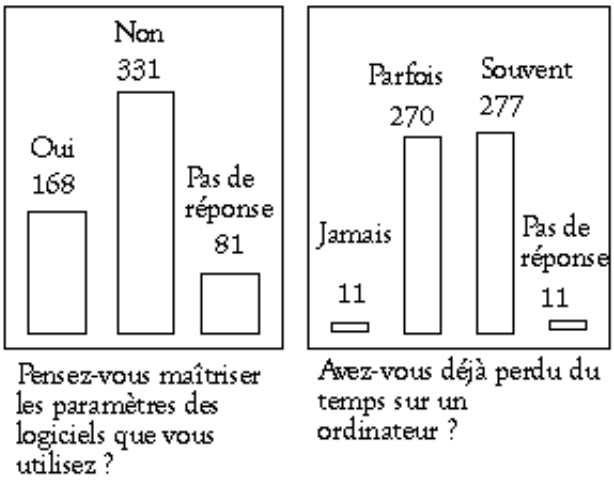


FIG. 2.6 – Les problèmes

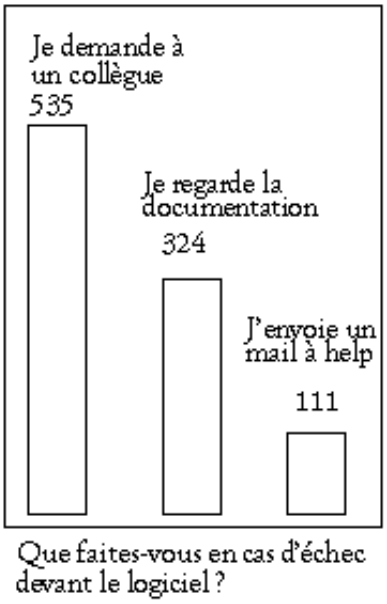


FIG. 2.7 – Comment trouver de l'aide

catégories :

1. Les occasionnels (210 personnes) : cette catégorie regroupe les utilisateurs qui sont essentiellement usagers de la bureautique, et qui n'ont ni le but, ni le temps de s'investir dans l'informatique

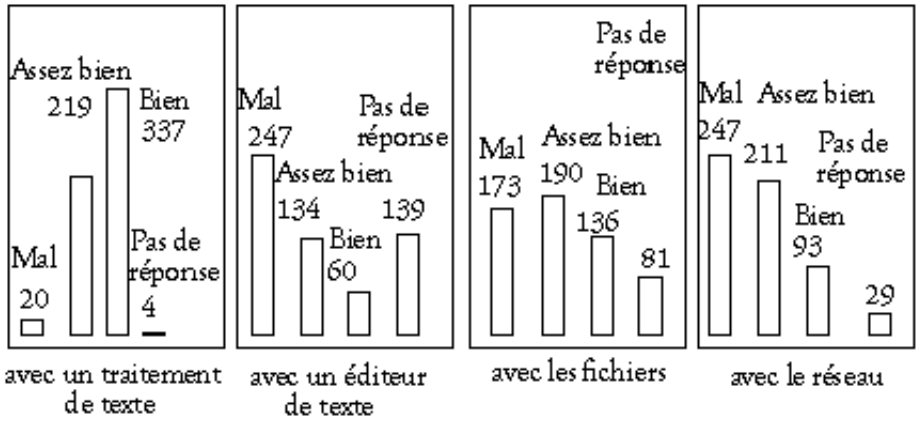


FIG. 2.8 – Comment vous débrouillez-vous avec ...

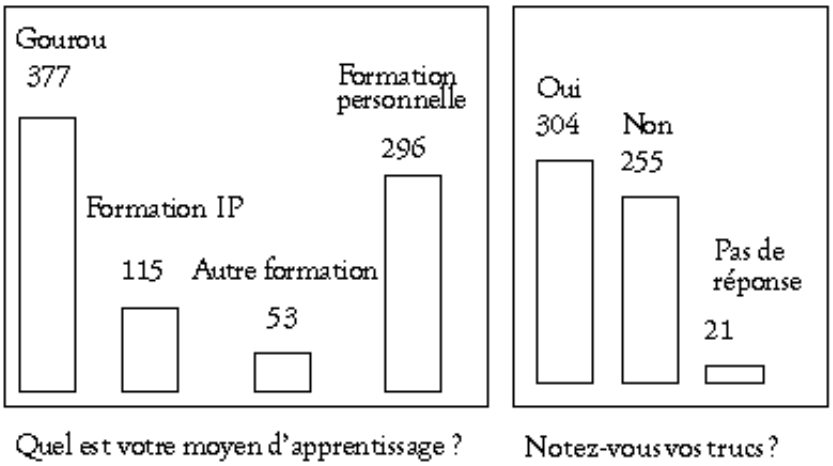


FIG. 2.9 – Formation.

par la consultation de documentation, par une formation ou par l'exploration des possibilités des logiciels. D'ailleurs, 64% de ces personnes n'ont pas de terminal attitré. D'où un certain nombre de difficultés, notamment avec les fichiers ou avec le réseau (pour 65%) ; le mail est consulté moins d'une fois par jour, voire jamais. Les utilisateurs de cette classe, plutôt des femmes (61%), ont l'impression de mal maîtriser les paramètres des logiciels, et de ne connaître que 20% de leurs fonctions. Pour s'aider, 100% d'entre eux font appel à leurs collègues et 75% considèrent que leur moyen d'apprentissage passe par la consultation d'un gourou. Mais ce qui caractérise le plus cette classe, c'est de ne jamais consulter de documentation en cas d'échec (72%) et de ne pas s'investir dans une formation personnelle (76%).

2. Les "non-unixiens" (90) : ce type d'utilisateur est bien décrit par deux caractéristiques : une grande autonomie avec l'informatique, mais une informatique ni "biologique" ni "Unix". Il utilise plutôt des logiciels de statistiques (34%) ou de recherche bibliographique (91%), même si c'est assez fréquent pour les autres classes, et 49% de ces personnes considèrent les statistiques comme faisant partie de leur métier. Inversement, 81% n'utilisent pas de logiciels d'analyse de séquences ou d'assemblage de séquences (98%) et considèrent que ces activités ne font pas partie de leur métier (72%). Il s'agit de personnes utilisant l'informatique depuis longtemps (plus de 6 ans après leur entrée à l'Institut Pasteur pour 44%), connaissant bien les logiciels qu'ils utilisent (la moitié estiment connaître 60% des fonctions des logiciels). L'informatique utilisée est apparemment très peu "unixienne" : 42% d'entre eux sont équipés de PC, et 90% n'utilisent pas d'outils Unix, ni ne programment de routines (98%). Ils n'ont pas besoin de formation (91% n'en ont d'ailleurs pas

suivi), et, en cas de problèmes, consultent la documentation.

3. Les apprentis (90) : une classe un peu à part qui rassemble ceux qui ont suivi les formations du service d'informatique scientifique (l'analyse factorielle regroupe les personnes qui ont des points communs moins courants que par exemple le fait de posséder un Macintosh). Mais on peut remarquer que ce n'est pas le seul point commun dans ce groupe : plus informées des efforts du service d'informatique scientifique pour communiquer avec les utilisateurs, ces personnes lisent le B6 (90% le gardent) et ont répondu à la plupart des questions du questionnaire ! S'agit-il d'autres effets de la formation qu'ils ont suivie, les personnes se débrouillent bien avec le réseau, explorent par le Web, utilisent les outils Unix (40%), se débrouillent assez bien avec un éditeur de texte (40%), lisent les News (50%), utilisent un menu (58%), envoient un mail à help en cas de problème (48%) et ont décroché la meilleure note à la question Quizz sur le prix d'abonnement à l'Internet de l'Institut (23%). Il se peut que la possession d'un terminal X (56%) soit aussi un facteur favorable. Si 76% d'entre eux sont une aide pour quelqu'un et 81% ont déjà installé un logiciel, aucun d'entre eux ne programme. La formation ne résoud pas tout : sur le plan factoriel (figure 10), ces personnes sont tout de même du côté de ceux qui ont encore des difficultés, même si elles utilisent l'informatique depuis quelques années.
4. Les plus jeunes (85) : aussi autonomes que les personnes de la classe "non-unixiens", ces utilisateurs sont plus jeunes (41% ont entre 25 et 30 ans, 53% sont stagiaires). Ils tiennent leurs connaissances d'une pratique avant leur entrée à l'Institut Pasteur (plus de 6 ans pour 25%), d'une formation antérieure et d'un apprentissage personnel (98% n'ont pas suivi les formations SIS, mais 60% connaissent les logiciels par la documentation ou par l'exploration). A la différence de la classe précédente, ces personnes font de l'analyse de séquences (92%), de l'analyse génétique ou de la phylogénie. Ils connaissent les joies des réseaux, du Web par exemple, pour la recherche dans les banques (98%), la recherche de documents (83%), l'analyse de séquences (66%) ou du courrier électronique (33% le lisent dès qu'il arrive). Ils se débrouillent bien avec les fichiers aussi, et même assez bien avec un éditeur de texte. Enfin, 63% sont des hommes.
5. les gourous (35) : dans cette classe ont trouve les quasi "informaticiens" (97% d'hommes...). Leur différence est la pratique de la programmation (61% considèrent la programmation comme faisant partie de leur métier, 82% programment des routines, 20% des programmes plus complexes - ce qui fait environ 7 programmeurs pour l'Institut Pasteur en dehors du service d'informatique scientifique). Ils se débrouillent bien avec à peu près tout, ont une station de travail (73%), ou un terminal X, utilisent beaucoup le réseau (45% ont réalisé une page Web, 48% ont posté un message dans les News), explorent souvent (42%) et pratiquaient l'informatique avant d'entrer à l'Institut Pasteur (plus de 6 ans pour 55%). C'est dans cette classe que les personnes ont signalé des problèmes de performances ou demandé l'installation de logiciels.
6. Les discrets (30) : c'est ainsi que nous avions dénommées les personnes ayant rempli le questionnaire mais ne se sentant pas très concernées par les questions d'informatiques.
7. Les administratifs (22).

Cette enquête a été réalisée il y a maintenant 5 ans, ce qui en relativise les résultats puisque l'utilisation de l'informatique se développe très rapidement en biologie, et le nombre de biologistes pratiquant l'informatique a fortement augmenté. La situation des utilisateurs occasionnels a également été significativement améliorée par le développement d'interfaces Web pour utiliser les logiciels Unix.

2.2.2.2 Tests pour un générateur d'interfaces Web, Pise

Cette étude a été réalisée au cours de l'hiver 1997 à l'occasion du développement d'un générateur d'interfaces Web pour les programmes d'analyse biologique fonctionnant sur Unix, Pise [Let00b] (décrit dans le chapitre V). L'interface générée par ce programme, écrite en HTML simple, venait d'être mise en place et les techniques de conception centrée sur l'utilisateur avaient été vues (cours de Wendy Mackay) à l'automne précédent dans le cadre du DEA d'Informatique d'Orsay.

Les sujets des tests ont été recrutés par envoi d'un message sur la liste électronique du campus, auxquel une quinzaine de personnes ont répondu. Les tests ont été précédés d'un petit questionnaire par courrier électronique demandant aux personnes de préciser leur niveau en informatique (débutant, occasionnel, habitué) et le type de problèmes scientifiques traités par informatique qu'elles connaissaient.

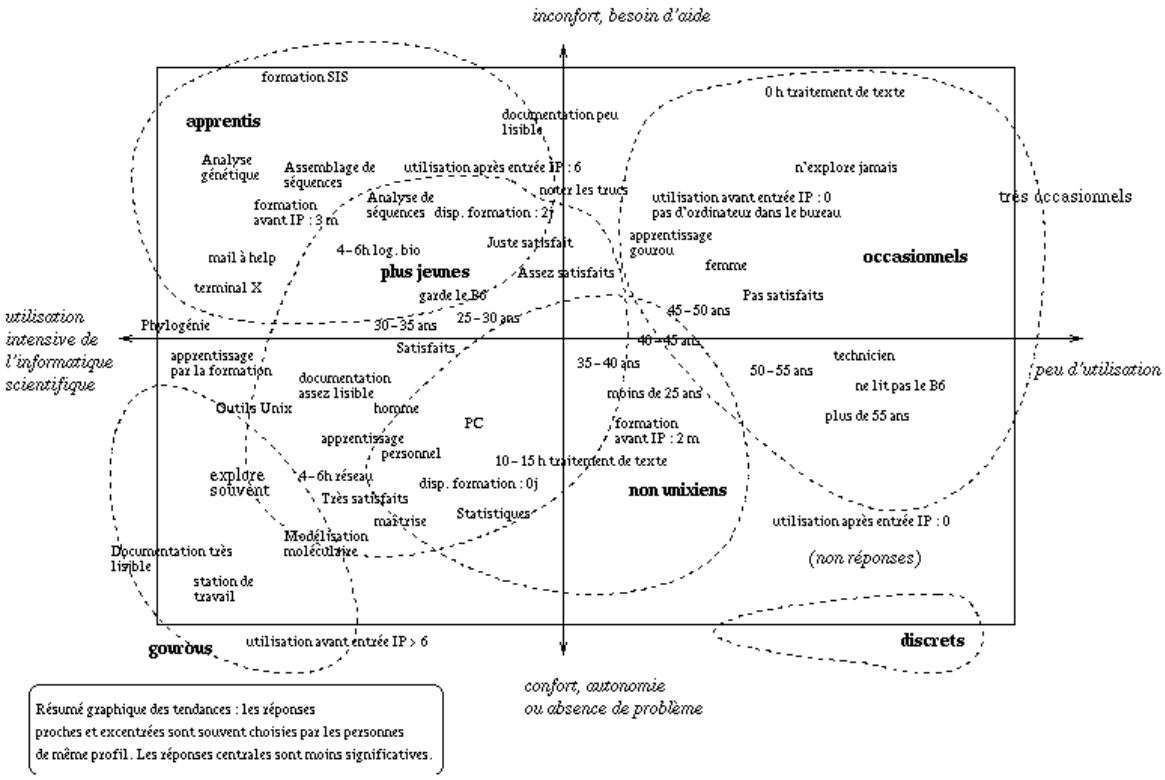


FIG. 2.10 – Analyse multi-dimensionnelle.

La procédure de test utilisée était le "think aloud" par paire, une personne décrivant l'interface et les interactions en cours, l'autre personne "dictant" ce qu'il fallait faire. 6 groupes ont pu être constitués, chaque séance durant environ une heure.

Le résultat de ces séances de test a été une amélioration des interfaces, et une mise en évidence des éléments sur lesquels il fallait faire des compromis. La procédure de test était, comme on le voit, extrêmement simple (beaucoup plus que pour la première étude), mais suffisamment utile (également beaucoup plus que pour la première étude) : ces interfaces sont maintenant utilisées par des dizaines de milliers de personnes à raison de plus de 500 soumissions par jour, et le logiciel est distribué et installé sur une quinzaine de sites.

Un résultat annexe était la confirmation de l'intérêt de la démarche pour interagir avec les biologistes. Le retour de la part de ces derniers a été très positif : de nombreux messages spontanés manifestant leur accord sur l'idée de faire participer les utilisateurs aux tests d'interface ou même expliquant pourquoi la séance avait été intéressante. Pour certains, c'était aussi l'occasion de "toucher" aux logiciels en présence d'informaticiens, ou de se remettre à une activité d'analyses de séquences après un pause un peu longue. C'est depuis ces séances que ma conviction quant à l'importance de cette "attitude" s'est enracinée. [KAD+96] montre que l'approche centrée sur l'utilisateur s'implante souvent dans les entreprises - du moins celles qui n'ont pas une vocation de production logicielle - à travers une personne convaincue. Curieusement, ces méthodes ne sont pas enseignées dans les cursus techniques d'informatique et ne font pas non plus vraiment partie de la culture informaticienne. L'enseignement du DEA d'Informatique d'Orsay a donc été déterminant pour l'introduction de ces méthodes dans le cadre de l'informatique de l'Institut Pasteur.

2.2.2.3 Tests pour un annuaire interrogeable, le BioNetbook

Le BioNetbook est une base d'adresses de pages Web concernant la biologie (le nom vient de "notebook" qui veut dire carnet d'adresses, "net" pour Internet et "bio" pour biologie). Sa première version consistait en une dizaine de pages HTML accessibles à partir d'une page de menu intitulée "Biologie Moléculaire sur le Web". Cette version a été complètement remaniée à l'automne 1998, en collaboration avec une collègue biologiste (Irène Wang) pour transformer ces pages en base de données relationnelle. Cette base d'adresses contient à l'heure actuelle plus de 6000 références classées selon différents critères (type de ressource, espèce ou organisme et domaine biologique). Elle est également interrogeable par mot. La base est régulièrement mise à jour et les nouvelles entrées sont consultables. Une notice d'aide explique comment utiliser l'interface Web et donne de nombreux exemples d'interrogations.

Ces séances avaient pour but d'améliorer la lisibilité de l'interface Web (et non de refaire toute la conception), c'est-à-dire de faire en sorte que des biologistes, censés connaître les bases de Netscape, de l'Internet et ce qu'est un outil de recherche de mots, puissent comprendre le fonctionnement du formulaire.

La procédure suivie était analogue à celle utilisée pour les interfaces Web générée par Pise, bien que seulement 6 personnes, réparties en 3 groupes de 2, aient pu participer aux tests. Durant la séance, l'utilisation de l'interface commençait en général avec une requête qui correspondait au thème de travail d'une des deux personnes. Un court entretien préalable permettait donc de déterminer des thèmes qu'on pouvait suggérer comme exemple pour démarrer. Exemples :

- recherche d'outils de calcul du point isoélectrique d'une protéine ;
- recherche de pages sur la vidéo-microscopie ;
- données sur les structures transmembranaires ;
- recherche d'informations sur traitement de la leucémie chez l'homme ;
- pages sur Listeria.

Les résultats des tests sont décrits dans le détail dans l'article publié dans le bulletin du service. Ils ont essentiellement permis de lever les ambiguïtés de l'interface, notamment entre le formulaire et la notice d'aide (un des sujets avait pris la notice pour une version "browser" du formulaire). Les requêtes effectuées pendant les tests ont également mis en évidence le fait que le type de contenu, les données de la base, n'étaient pas clairement perçues : certains pensaient y voir une base de connaissances, ou bien une base de séquences biologiques, alors qu'il s'agit d'une base d'adresses ! Certains termes et la langue de l'interface sont apparues comme étant difficiles à comprendre.

Résultats des améliorations de l'interface Web

Des mesures ont été effectuées à la suite des améliorations apportées à l'interface en tenant compte des tests. Sur une période de deux mois s'étendant de mi-février à mi-avril 2000, le nombre de requêtes par jour a quadruplé par rapport à la période précédant les tests (il est passé à 360 par jour). Cela peut s'expliquer par d'autres facteurs, notamment par le fait que la base de données était déjà référencée sur plus de 250 sites spécialisés en biologie. Une optimisation du temps de réponse (modification indépendante des tests dont il est question ici, mais effectuée à la même époque) peut également expliquer une utilisation plus enthousiaste. L'augmentation du nombre de requêtes peut encore s'expliquer par deux autres facteurs : la notice d'aide comporte beaucoup plus de liens hypertextes qui sont autant de requêtes effectuées sur la base d'une part, et la classification des réponses est devenue cliquable (ce qui correspond approximativement à une fonction de recherche des réponses voisines).

L'effet des changements dans l'interface effectués d'après les informations recueillies lors des tests se voit probablement plus à la proportion de questions sans réponses qui a diminué (33% au lieu de 42%). En particulier, la médiane a changé : alors que 50% des questions obtenaient auparavant de 0 à 2 réponses, 50% des questions obtiennent actuellement de 0 à 10 réponses.

2.2.2.4 Conclusion sur les études réalisées

Ces études ne sont pas fortement centrées sur le sujet de la thèse, qui est la programmation par l'utilisateur, mais il nous semble qu'elles présentent un intérêt méthodologique, d'une part pour situer le contexte de ces travaux, mais aussi parce que nous pensons que, dans un environnement réaliste,

toute solution à la programmation par l'utilisateur doit reposer sur un *ensemble de solutions graduées*

2.3 Programmation par l'utilisateur et développement participatif.

2.3.1 Utilité et limites des études objectives.

Le type d'études que nous avons présentées dans la section précedente, qu'elles portent sur les aspects cognitifs ou sur l'utilisabilité, et qu'on peut caractériser comme "objectives", est d'une grande utilité par rapport au domaine de la programmation par l'utilisateur, plutôt technique [Gre95], cherchant souvent une justification objective dans les domaines non techniques. Des mises en garde contre l'utilisation injustifiée d'affirmations cognitives, comme celle de [Bla96], ou contre les opinions de superlativisme (avantage évident de tel ou tel type de programmation) ou de naturalisme (affirmations sur le fait qu'un langage ou un modèle de calcul est naturel) comme celle de [Gre97] sont ainsi très utiles.

Un modèle incontournable

Mais ce qui sous-tend ces approches, c'est l'idée tout de même qu'il y a un modèle incontournable, le calcul par ordinateur, pour lequel des spécialistes théoriciens ont inventé des modèles de calcul, et auquel l'utilisateur doit se conformer. Cela est détectable dans la démarcation très floue qu'on trouve dans plusieurs de ces recherches entre la pratique d'un test, comme en psychologie cognitive et en pédagogie, où ce sont les capacités cognitives du programmeur qui sont évaluées par rapport à tel ou tel langage ou modèle, et la conduite d'une étude d'utilisabilité. Très souvent, comme dans [DP95] ou [ADST00], la conclusion comporte une affirmation concernant la capacité ou l'incapacité de l'utilisateur à changer de modèle mental. C'est plus souvent l'apprenabilité que l'utilisabilité qui est évaluée et la question explicite est bien : les utilisateurs vont-ils comprendre le système ? Nous avons vu pourtant que la programmation recouvre des activités très diverses, sur lesquelles un accent différent pourrait être mis selon les contextes particuliers. Il y a notamment une différence de but et de type d'activité entre la programmation comme spécification et résolution de problèmes et la programmation comme traduction de plans mentaux en mécanismes de programmation, qui est une vue de la programmation très tournée vers la technologie [Rep93a].

Objectivité des études et aspects pragmatiques

Un des aspects qui nous semble *déterminant* et qui n'est pas souvent pris en compte par ces études, est celui des considérations pragmatiques et sociales ([BHP+94] intègre cependant un aspect pratique dans sa taxonomie des langages visuels avec la dimension décrivant le support d'applications existantes). Ce qui détermine bien souvent le choix d'un environnement ou d'un logiciel, ce sont des éléments très contingents :

- un collègue connaît l'outil,
- l'outil était déjà installé sur la machine [G97],
- des éléments subjectifs, ou de conviction (le simple mot de programmation ou de variable peut influencer négativement [Bla00][BRL91]),
- présence d'une bibliothèque ou d'une archive maintenue de logiciels,
- l'outil est déjà connu (même s'il a été utilisé dans un contexte différent).

Ces aspects ont pris bien sûr une grande importance dans le contexte de cette thèse, pour laquelle il n'était pas envisageable de travailler exclusivement sur des outils de recherche, fermés à l'environnement technique existant, mais nous croyons par ailleurs que le développement *effectif* de l'accès à la programmation pour les non-professionnels doit prendre en compte ces dimensions contingentes.

Vers un développement plus coopératif.

La question qui nous intéresse maintenant est alors de savoir si, dans le cadre d'environnements ou de langages pour la programmation par l'utilisateur, cela a un sens d'envisager une analyse centrée sur l'utilisateur qui intervienne dès la conception, et si l'idée d'abandonner le laboratoire pour passer au terrain [HL91] a un sens dans ce domaine.

2.3.2 Les méthodes participatives.

2.3.2.1 Démarche générale.

L'objectif général de la démarche participative [BGK93][BESS00] est de faire des outils contrôlables par leurs utilisateurs qui améliorent à la fois la productivité et la qualité du travail. Le principe de la démarche consiste à anticiper les problèmes organisationnels ou techniques qui se posent immanquablement lors de l'introduction d'un système informatique. La méthodologie consiste, à cette fin, à intégrer l'utilisateur jusque dans les phases dites techniques de la fabrication, phases durant lesquelles il lui est possible de travailler (imaginer, prototyper, simuler) sur des représentations du système futur [Kyn95].

Perspectives : pragmatique, théorique et politique

La démarche participative - ou coopérative - peut être présentée selon trois perspectives [Gre93] : pragmatique, théorique et politique.

1. Pragmatique : sur le plan de l'efficacité, il y a un gain certain à adapter un système informatique aux besoins des utilisateurs ; pour les développeurs, ce sont des systèmes qui marchent mieux et qui sont plus utilisés ; pour le management, cela signifie plus de productivité.
2. Théorique :
   (a) jeux de langage : "if a lion could speak, we would be able to understand it" (Wittgenstein) ; l'idée est donc de développer les moyens d'expression pour la description et la création de représentations ;
   (b) action et réflexion : notre manière d'être est dans l'action plus souvent que dans la réflexion détachée (Heidegger) ; en faisant de la conception en situation, on suit cette idée.
3. Politique : la conception participative est née en Scandinavie à l'époque de la démocratisation ("folkhemmet", dans les années 1980 [BESS00]) : l'idée était donc de démocratiser la conception des systèmes informatiques et de mettre les éléments décisifs de cette conception dans les mains des travailleurs.

Si, pour notre contexte, l'aspect politique n'a pas de réelle importance (encore que...), les aspects pragmatique et théorique sont intéressants. L'aspect théorique, avec la question du détachement réflexif, nous donne un cadre conceptuel pour situer la programmation dans l'activité générale d'utilisation, comme cela est développé plus loin.

Emprunt aux principes des études ethnographiques

La démarche participative emprunte aux études ethnographiques un certain nombre de principes [BGMSW93][MNG+93].

- utiliser l'environnement naturel : cela consiste à étudier les pratiques de travail des utilisateurs dans leur laboratoire et non dans un bureau du centre informatique ;
- compréhension du contexte et des relations internes : l'utilisation, le choix, les difficultés associées à un système informatique ne sont pas purement cognitives, mais situées dans un contexte, en relation avec des outils et des problèmes spécifiques, comme nous l'avons vu plus haut [Rep93a][Gre89] ; s'y ajoutent les questions d'organisation et d'espace de travail [KM95] ;
- la démarche est plus descriptive que normative : ce n'est pas l'informatique qui dicte les méthodes de travail, mais le contraire ;
- la grille d'analyse (les catégories descriptives) doivent être celles du sujet et non de l'observateur : cette idée, qui suppose également que les catégories descriptives ne sont pas nécessairement les mêmes d'un groupe à l'autre, induit des difficultés éventuelles sur les comparaisons de groupes différents et pose un problème pour les méthodes "objectives" (comme les méthodes d'inspection d'utilisabilité [NM94] ou même les dimensions cognitives de [Gre89]) ; il faut sans doute considérer que les démarches sont complémentaires : la démarche participative permettant de laisser émerger des variables [Nar97b], les méthodes "objectives" permettant de les évaluer ;
- approche évolutive et adaptative [Nar97b] : c'est une des raisons pour lesquelles il n'y a pas de règles dans la conception participative - il faut s'adapter à chaque situation particulière ;
- identification du chercheur aux intérêts des sujets étudiés ;

psychologie cognitive	*ethno-méthodologie*
repose sur un modèle des buts et intentions du sujet	se fonde sur une approche de phénoménologie descriptive
données quantitatives	données contextualisées
modèle causal, corrélations	non-causal ; critique des conditions artificielles de l'expérimentation en laboratoire
explications cognitives	explications sociales et interprétatives

TAB. 2.4 – Psychologie cognitive et ethno-méthodologie

- participation des sujets à la formulation des descriptions.

Les techniques empruntées à l'ethnographie concernent notamment :

- l'observation : faire la différence entre ce que les gens font et disent qu'ils font (comportement idéal versus manifeste, connaissances tacites, comportements automatiques) ; lorsqu'il s'agit d'informatiser une tâche, on en donne souvent une description idéalisée et rationalisée en oubliant involontairement les détails dont la non prise en compte peut rendre un système informatique inutilisable ;
- l'attention, qui doit être portée sur les évènements, les personnes, les lieux, les objets, les artefacts, les incidents.

Les arguments pour adopter l'approche participative dans le cadre de la conception de systèmes sont :

- une meilleure compréhension du terrain ;
- l'idée de ne pas imposer une vision des choses ;
- une réponse au problème des technologies en quête d'applications - on n'en connait pas les utilisateurs potentiels ;
- comprendre l'importance du contexte d'utilisation d'une technologie ;
- les difficultés pour les utilisateurs à visualiser les technologies futures ou le comportement d'un système futur ;
- reconnaître les limites des approches traditionnelles : dans ces approches, l'utilisateur reste virtuel, le lieu de travail n'est jamais vu, le focus est sur la technologie, pas le travail.

On peut comparer la démarche participative à celles que nous avons décrites auparavant (études cognitives et d'utilisabilité) en suivant l'argumentation de [MNG+93] lors d'un panel de comparaison des approches ethno-méthodologiques et de la psychologie cognitive (table 4).

La critique essentielle des ethno-méthodologistes par rapport aux conditions de laboratoire est qu'elles ne correspondent pas au monde réel et qu'on ne peut pas contrôler toutes les variables qui affectent un comportement ; on veut s'isoler du contexte alors que c'est lui le plus important. D'autre part, les tâches données aux sujets sont très contraintes, artificielles, et sont à effectuer en un temps limité et insuffisant. Enfin, on ne considère pas les idées des sujets dans une expérimentation (dans une étude sur les conséquences cognitives de l'enseignement de la programmation, [LD89] mesure l'intérêt pour la programmation par un score sur l'envie de lire des articles en fonction de leur sujet : 12 sur l'informatique, 8 sur la science, 4 non techniques ; mais c'est une mesure, pas une idée).

Les psychologues répondent à cela qu'il faut faire la différence entre l'expérimentation pure et l'expérimentation appliquée et qu'il faut reconnaitre l'apport méthodologique pour la généralisation, qui est plus délicate à effectuer en ethno-méthodologie.

Cependant, [Nar97b] insiste avec humour sur la nécessité de ne pas se lancer dans l'ethnographie quand on n'est pas formé pour cela. Il faut savoir poser les bonnes questions, ne pas se contenter de certaines réponses, il faut surtout savoir quoi faire des données, et gérer le biais de l'observateur. Mieux vaut proposer un projet à un étudiant en ethnographie que de le faire soi-même (même s'il ne "cause" pas Derrida ou Latour). Elle souligne également l'importance d'une perspective théorique, comme celles de la théorie de l'activité [Bød96][Nar97a], de la cognition distribuée [Hut93] ou de l'action située [ST93].

"Work of the work, work of the tool".

[HJ93] apporte une distinction à rattacher à la question du détachement réflexif : si quelqu'un est en train d'écrire et qu'on lui demande ce qu'il fait, il dira : *j'écris un article ...* Il ne dira pas : *j'utilise un stylo et du papier* . Ce n'est que si l'outil ne fonctionne plus qu'il va s'y intéresser. Or, très souvent, les études d'utilisabilité s'intéressent à l'outil plus qu'au travail - ce qui est quand même compréhensible aussi - mais du coup les études portent plus sur les dernières étapes du développement. C'est également la référence que [Mør94] fait aux modes expérimental/réflexif de Norman : le mode réflexif (les comparaisons, les prises de décision) est une prise de distance permettant la remise en cause d'une conception.

Conception et technologie

Le langage dans lequel un système futur est décrit lors de sa spécification, c'est-à-dire par exemple un diagramme d'ordonnancement ou un schéma relationnel, est souvent difficile à interpréter pour les utilisateurs (ou clients, commanditaires). Il est alors impossible pour ces derniers d'anticiper les fonctions réelles de ce système et de vérifier que les modalités leur conviennent. Un exemple est donné par [BG91] : lors du changement d'un système de paye, alors que les secrétaires géraient les dossiers par types de contrat et pouvaient aider les utilisateurs, cela est devenu impossible avec l'arrivée d'un système qui effectuait le classement par ordre alphabétique. Cet aspect apparemment "technique" de la conception, dont les analystes du projet n'ont sans doute pas perçu l'importance pour n'avoir pas observé les utilisateurs de l'ancien système, a eu des effets importants sur l'organisation, effets qu'il était de toutes façons difficile d'anticiper à la simple lecture d'un document de spécification.

C'est l'un des arguments pour l'implication de l'utilisateur dans les phases "techniques" de la fabrication : ces dernières sont en réalité le lieu d'un certain nombre de décisions importantes qui n'ont pas été revues ou étaient peu perceptibles lors des phases de spécification [Dou96b].

Flexibilité

[KM95] et [Tri92] montrent que la conception participative est un bon moyen de spécifier des systèmes flexibles, car elle est conçue pour gérer l'inattendu, le non-standard dans la mesure où cela se produit dans des pratiques réelles.

2.3.2.2 Techniques.

Il n'y pas de techniques officielles ni de règles explicites dans la démarche participative, mais [MWW93], bien qu'il en soit conscient, consacre cependant une revue à vocation pratique pour conseiller les concepteurs dans le choix de méthodes ou techniques, et pour promouvoir ces pratiques. Les approches sont recensées avec des exemples selon leur objectif, les phases de développement où elles s'appliquent, le processus (communication, décision), le matériel qu'elles utilisent ("object model"), le modèle de participation (qui participe), la taille des groupes, le type de résultat et les méthodes formelles qui leur sont complémentaires.

Nous détaillons un peu dans ce qui suit les techniques que nous avons utilisées.

Entretiens

[HJ93] donne un bon aperçu des éléments qui sont importants pour réussir un entretien : ils recoupent, comme on peut le constater, certains des principes des études ethnographiques :

- Intérêt d'être présent dans le bureau, le laboratoire : on voit les objets, les cahiers, livres, documents qui servent, les post-it et aussi les collègues. On peut donc poser des questions sur des choses concrètes et les choses à raconter arrivent plus facilement.
- Principe fondamental des entretiens : donner le contrôle à la personne; pour cela, poser des questions ouvertes, saisir les opportunités d'approfondir une question, écouter.
- Gérer le focus : l'enquêteur a forcément un focus, ou un filtre quand il arrive pour l'entretien et il vaut mieux en être conscient; il a une idée sur la nature du travail, des opinions sur le design, ou sur l'utilisabilité, ainsi que sur les buts de l'enquête. Il faut donc saisir toutes les occasions de prise de conscience; par exemple quand la personne fait un "oui-oui" approbateur on croit qu'on comprend - mais cela peut signifier qu'elle pense à une autre idée. Quand on croit qu'on comprend, il est donc utile de vérifier en posant une question.
- Les interviewés on toujours une liste de desiderata, des opinions : ce qui est (aussi) intéressant

c'est de comprendre pourquoi ils veulent cela.
- Il est utile d'utiliser ce schéma de questions : que faites-vous, pourquoi le faites-vous, est-ce que c'est ce que vous attendiez (résultat à l'écran).
- Après l'entretien : chercher les structures et les flux de travail, les problèmes pour faire le travail, les problèmes pour utiliser le système, les descriptions de problèmes causés par le système et des contournements. Noter les aspects du système et du processus de travail qui supportent la tâche.

Scénarios

Les entretiens sont une bonne occasion de relever des scénarios, c'est-à-dire une situation liée à un problème particulier et à des données, et la suite d'actions correspondant à la recherche de solutions.

Selon [Car95], les scénarios ont pour avantage d'être à un niveau de notation que les utilisateurs comprennent. Contrairement aux techniques de conception classiques, qui visent à être formelles, rigoureuses et exhaustives, les techniques de scénario sont ouvertes, informelles, fragmentaires, redondantes et concrètes. Elles permettent de prendre en compte les aspects imparfaits, les erreurs, les interruptions. Elles aident à se poser des questions, notamment des questions imprévues. Elles sont utiles à toutes les étapes : du cahier des charges à l'évolution future en passant par les spécifications, les rationales, l'évaluation, la documentation, le travail d'équipe en général. Nous ajoutons que les scénarios sont également utiles pour le prototypage réalisé au cours des ateliers de conception.

Ateliers

Il y a plusieurs sortes d'ateliers :

- des ateliers de brainstorming, qui consistent à générer le plus d'idées possibles, en suivant quelques règles du jeu :
  - choix pour chaque groupe d'un modérateur qui veille à ce que tout le monde se soit exprimé, et d'un scribe qui note les idées (sur une grande feuille),
  - pas de censure, ni même d'opinion sur ce que dit quelqu'un d'autre,
  - proposer au moins une idée considérée comme stupide par son auteur.

  Ces règles du jeu ont pour objectif de favoriser la génération des idées, car une idée apparemment hors-sujet ou sans intérêt pour susciter, par association d'idées, d'autres idées plus utiles.
- des ateliers de prototypage où il s'agit de construire une maquette sur la base d'un scénario ou d'un brainstorming.

Prototypage

[Bro87] signale l'importance du prototypage. Selon lui, l'aspect le plus difficile dans la construction d'un système informatique est de décider précisément quoi construire :

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Il fait également remarquer que, pour qu'un système soit bien spécifié, l'idéal est même de faire un prototype qui simule les parties importantes de l'interface et effectue les fonctions les plus importantes.

Mais la définition d'un prototype est très variable : pour une personne faisant une étude centrée utilisateurs, cela peut-être un storyboard en papier, pour un programmeur un programme de test, pour un spécialiste des interactions graphiques des images écran avec un comportement prédéfini, pour une organisation une preuve suffisante de faisabilité [HH97].

Dans la démarche participative, un prototype est une représentation du système futur qui sert à effectuer des simulations [Kyn95]. Mais un prototype peut avoir plusieurs rôles, qu'il est important d'identifier clairement [HH97] :

- outil exploratoire,
- crédibiblité,
- démonstration,
- visualisation,
- manipulations effectives,
- système final,

– spécification exécutable.

[HH97] modélise également les objectifs d'un prototype autour de trois problèmes : rôle de l'artifact pour l'utilisateur, look and feel (comment l'artifact va-t-il être utilisé, manipulé, avec quelles interactions ? expérience concrète de l'utilisateur) et implémentation (faisabilité avec un langage, performances, ...).

Pour [BG91] le statut du prototype peut être :

- le prototype destiné à devenir le système final (sert à recevoir le feedback de l'utilisateur - on est alors proche d'une étude d'utilisabilité),
- le prototype comme spécification exécutable (certains CASE-tools peuvent générer des prototypes à partir de spécifications),
- le prototype exploratoire,
- les maquettes peuvent aussi servir à l'apprentissage par l'utilisateur, à confronter leurs compétences avec des techniques nouvelles [BGK93].

A titre d'exemples, deux prototypes sont décrits dans [BG91] : un premier, utilisé dans un atelier auprès d'assistants dentaires, construit autour de piles HyperCard et un deuxième, construit avec Oracle et de la programmation, pour un atelier plus orienté autour d'un problème d'organisation. Le premier exemple consistait plutôt en un prototypage de l'interface où les utilisateurs ont pu expérimenter des éléments d'interface ou même faire du dessin libre, alors que le deuxième était moins souple. L'auteur pose le problème de la modification du prototype en cours d'atelier : de légères modifications de code ont eu lieu pendant les sessions, à la demande d'un utilisateur qui voulait copier un bouton et modifier le script - ce qui a notamment pour intérêt de montrer aux utilisateurs que le prototype est souple et a pour but le maquettage. Il est de toutes façons difficile d'arrêter le processus d'exploration/évaluation pour faire une modification. Une autre question est la possibilité ou non d'utiliser certaines fonctions pendant un atelier, soit parce qu'elles sont trop complexes (comme dans cet exemple la formulation de requêtes pour l'outil de génération de rapports qui a nécessité l'aide du concepteur), soit parce qu'elles prennent trop de temps à s'exécuter. Ces remarques concernant la manipulabilité et modifiabilité des maquettes nous semblent potentiellement transposables à la manipulation d'éléments de programmation lors d'un atelier : est-ce faisable, cela n'interrompt-il pas l'atelier ? Ou au contraire l'accès à la programmation peut-il aider à résoudre ces problèmes de rupture ?

La maquette peut être construite avec divers types de matériaux :

- du matériel de bureau, comme du papier, des post-it, des transparents, du papier collant (couverture de livres), des étiquettes, ...
- une cassette vidéo,
- des programmes informatiques adaptés au prototypage rapide, comme HyperCard [BG91] ou des systèmes ad-hoc, comme un storyboard en Lisp [EK91] ou une version du système lui-même s'il s'agit d'un prototype proche du système final ou d'une partie de ce système,
- une combinaison des trois : une vidéo de maquettes papier, une maquette papier superposée à une vidéo ou à l'interface d'un système informatique [EK91], ...

La littérature signale quelques difficultés concernant le statut du prototype. Pour que le prototype serve de support à la discussion et non de produit de démonstration, il ne doit pas y avoir de confusion possible avec le système réel. Ainsi, il est important que les difficultés matérielles rencontrées avec le prototype apparaissent comme telles. C'est un problème avec un prototype informatique : s'il y a une "erreur", on ne sait pas toujours si c'est une "vraie" erreur ou un bug du système, alors qu'il n'y a aucune ambiguité si un post-it tombe par terre : on comprend ce qui se passe, ce n'est pas une boîte noire [BGK93].

Les prototypes informatiques favorisent les techniques qui marchent bien et la participation des utilisateurs est plus réduite - ils laissent faire les programmeurs (cf exemple de maquettage de storyboard en Lisp : ceux qui ne pouvaient pas programmer étaient peu participatifs, et n'avaient que des discussions vagues) [EK91]. Sinon les ordinateurs ont bien sûr de nombreux avantages pour faire des maquettes : on n'a pas besoin de tout recopier, de tout refaire quand il y a quelque chose à changer ; on peut sauver un design intermédiaire et on peut tester l'efficacité et les performances. Par exemple, l'utilisation d'Hypercard permet en faisant succéder des images, de simuler des changements dynamiques, de l'entrée de texte, une sélection. Technique mixte a consisté, pour pallier un manque d'Hypercard, à

utiliser du papier sur l'écran ! Dans le même ordre d'idée, [Mul91] pense que les techniques de prototypage rapide ou technologiques donnent un avantage aux concepteurs.

[CCRN00] est plutôt critique par rapport à ces approches centrées sur le prototypage et montre que ce n'est pas uniquement en faisant du prototypage dans des ateliers de courte durée qu'un développement participatif peut se dérouler. Il est vrai que la durée des ateliers ne pouvant pas être trop longue, il faut faire quelques compromis pour que les participants des ateliers aient le temps de réaliser quelque chose :

- préparer une partie du prototype avant l'atelier, éventuellement avec un ou deux participants,
- faire choisir aux participants un problème isolé et de taille réduite, par exemple une idée issue d'un brainstorming, et réaliser une vidéo dans un temps très court.

Vidéo

La vidéo a plusieurs applications dans les techniques participatives [Mac00] :

1. observation de personnes en train d'utiliser un logiciel, comme des étudiants utilisant un éditeur de réseaux de Petri [MRJ00],
2. observation de personnes en train d'effectuer une tâche, comme dans [CRC97] qui filme les activités d'une classe (travaux pratiques de physique, cours, discussions, ou exposés) ;
3. pour filmer des entretiens ;
4. pour filmer des artefacts : livres, cahiers de laboratoire, feuille d'instructions, devoirs ;
5. pour construire la maquette d'une fonction ou d'une interaction ou d'un sous-système : la vidéo est consiste alors à filmer des interactions avec des maquettes non informatiques [MRJ00] ; [Mul91] décrit le dispositif : la caméra est idéalement au-dessus d'une surface plane sur laquelle tout le monde peut intervenir ; une maquette ainsi construite devient souvent une spécification, qui constitue un bon outil de communication dans le cas notamment où designers et développeurs ne sont pas ensemble ; on y voit notamment les arguments différents dans les discussions avec les utilisateurs ;
6. la présentation de scénarios (la vidéo, tout comme l'enregistrement audio, s'avère très utile pour rappeller des détails oubliés) ; [EK91] explique que la vidéo et, tout comme l'utilisation d'un rétroprojecteur fait un effet visuel plus proche d'une maquette technologique pour le look&feel, mais cependant différent ; le but est de créer l'illusion d'un système futur ;
7. pour convaincre des utilisateurs : [CCRN00] raconte comment l'implication a changé à la fois lors d'un workshop d'analyse détaillée de scénarios vidéo comme base d'une session de demandes (claim) (mais aussi beaucoup quand le financement du projet par la NSF a officiellement reconnu les enseignants comme représentants de l'administration) ;
8. pour convaincre le management [BB99].

Nous avons utilisé la vidéo pour plusieurs des fonctions énumérées ci-dessus : pour observer des biologistes utilisant un éditeur d'alignement, ou effectuer une tâche, pour filmer des entretiens (dont certains en compagnie de Wendy Mackay), pour construire une spécification de sous-système (correspondant au prototype biok développé par la suite) et filmer des prototypes construits pendant un atelier et pour convaincre des collègues (ou du moins essayer).

2.3.2.3 Conception participative et programmation par l'utilisateur.

Situation dans la démarche globale

Nous situons l'utilisation de la conception participative dans notre travail dans le cadre du principe général d'*anticipation de l'imprévisible* [Gre93], aussi bien sur le plan de la spécification du système, de la conception comme sur celui de la technique et qui est illustré par la figure 11.

Le terme de "conception réflexive" mentionné dans la figure 11 se réfère aux techniques dont nous pensons qu'elles favorisent la flexibilité, et qui sont présentées au chapitre III.

[KM95] décrit ainsi la démarche participative comme un moyen de prise en compte des variations et des nécessités de flexibilité, par constraste avec les approches de conception orientées objets (abstraction, intérêt d'une approche générique et abstraite [Boo94]) ou herméneutiques (approche qui consiste

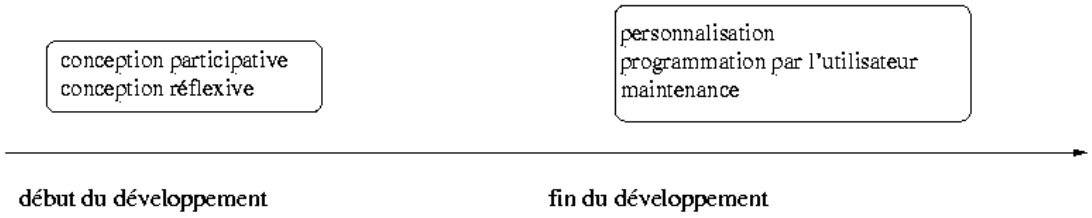


FIG. 2.11 – Flexibilité et programmation par l'utilisateur

à dégager des patterns récurrents). [HK91], dans le cadre de l'ouvrage "Design at work" entièrement consacré aux approches participatives, décrit l'approche de personnalisation comme un façon de continuer la conception participative, à condition que cela ait été prévu aussi bien au niveau des outils informatiques que durant la période de conception. [Tri92] précise cette idée en montrant qu'il faut se méfier des systèmes plein de "hooks" servant à la personnalisation, et que la tailorabilité est quelque chose qui se conçoit à l'avance. La conception participative est un lieu où se révèlent les fluidités, les diversités dans les usages, et permet de définir les régions de customisation. Ainsi, on peut repérer les ambiguités dans la façon de réaliser certaines tâches, comme les ambiguités procédurales. Cela concerne par exemple l'ordre dans lequel effectuer des actions, qui n'est pas toujours le même soit pour des utilisateurs différents, soit pour un même utilisateur : pour gérer son courrier, on voudra parfois pouvoir lui appliquer un système de règles (ruleset) *après* , et non *avant* , de l'avoir lu. Cela concerne en somme les incertitudes de l'utilisateur sur ses propres pratiques de travail, comportant de nombreux contre-exemples, des conflits, voire des problèmes politiques. Or, ces problèmes qui sont souvent écartés de la conception comme non pertinents, sont précisément ceux qui permettent de définir les régions flexibles.

Défense d'une approche non quantative et non expérimentale

Comme on vient de le voir, la conception participative consiste pour une grande part à laisser émerger des idées, ce qui se situe très en amont des études d'utilisabilité. Est-il possible d'appliquer la conception participative à la conception de langages ? Cela paraît assez envisageable pour les environnements de programmation, qui sont des interfaces utilisateurs au sens classique du terme. Mais qu'en est-il des niveaux paradigmes et notation ? Est-il possible que des non-professionnels inventent de nouveaux paradigmes ? De nouvelles notations ? Dans l'étude citée plus haut sur la programmation naturelle [PRM00], des innovations technologiques potentielles ont été identifiées, même si les variables définissant la programmation était déterminées d'avance, et même si seule la partie de la programmation consistant à traduire des plans en mécanismes de programmation est étudiée.

La réponse que nous apportons se situe en réalité sur un plan plus global que celui des outils reconnus de la programmation et de leurs caractéristiques.

2.3.2.4 Participation à la conception et programmation : une approche intégrée (la programmation comme un moyen).

La programmation par l'utilisateur est parfois considérée comme un outil de conception participative. C'est le cas pour [MA93], avec une utilisation de la programmation à base de storyboards informatiques (CISP : Cooperative Interactive Storyboarding Prototyping), une technique pour générer un consensus par un concept de système interactif tangible. Le problème qui est traité est celui des changements du prototype d'une session de conception participative à l'autre qui sont souvent un peu pénibles [BG91]. L'objectif est donc de réduire le cycle "use-eval-modify" en permettant l'utilisation directe du storyboard par l'utilisateur. L'utilisateur, d'évaluateur (reviewer) devient co-développeur. L'environnement fournit des blocs de construction orientés domaine qui ont une apparence réaliste. Le système enregistre et peut rejouer les actions, ce qui rend possibles les conceptions alternatives. Ce qui est important c'est que la manière dont une solution est trouvée a été enregistrée : on peut la retrouver sans avoir à tout recommencer. Les interactions (tâches de modification : placement, graphique, commentaires) sont enregistrées avec HyperCard.

Sur un autre plan, [Win95], dans le cadre d'une analogie générale entre programmation et environnements de conception, considère le prototypage interactif comme le pendant de la programmation interactive (comme le confirment ces fonctions similaires à de la programmation par démonstration dans l'exemple précédent) : il cite à cet égard les outils pour construire des "façades programmées" ("programmed façades") : Hypercard, Supercard, Macromind Director, Toolbox pour simuler les fonctions à travers l'interface, ainsi que les langages orientés prototypage (Hypercard, Visual Basic, Smalltalk) qui associent du code à des éléments d'interface.

2.3.2.5 Contradictions potentielles (la programmation comme un but).

Pour nous, la programmation par l'utilisateur n'est cependant pas limitée à un moyen pour spécifier des éléments de conception. L'accès à la programmation constitue un objectif, même si la programmation n'est pas un but en soi pour l'utilisateur.

Orientation technologique.

Sur le plan des principes sous-jacents aux techniques participatives, il y aurait plusieurs contradictions entre l'idée de faire programmer des non-informaticiens et l'approche à partir de maquettes. Un des principes importants du maquettage est dans l'immédiateté de la maquette, qui doit se faire oublier, et dont le faible contenu technologique aide l'utilisateur à se concentrer sur la conception sans s'attarder aux détails techniques [BGK93]. Notre projet a une forte orientation technologique, et le focus sur la technologie plutôt que sur le travail peut selon [BGMSW93] constituer une barrière à l'innovation.

Type de maquette.

D'autre part, quel type de prototype peut servir à prototyper la programmation ? Nous verrons dans la description des ateliers qu'il n'est pas simple de manipuler toutes les sortes de données et de maquettes, pour des questions de volumes des données (les séquences biologiques sont un peu lourdes à manipuler) mais aussi pour des questions de granularité : quel est le niveau pertinent de manipulation pour une activité de programmation ? Les outils de l'environnement, les instructions du programme, les concepts abstraits ?

Nature de la programmation.

Se pose aussi la question de la nature de la programmation elle-même : c'est une activité à vocation plutôt générique, s'exerçant en principe dans un détachement réflexif, ce qui va à l'encontre des principes d'immédiateté de la maquette et de l'idée de réflexion dans l'action [Sch90] qui sous-tend toute la démarche.

Apprentissage et participation à la conception.

Enfin, si la conception participative partage un certain nombre de points communs avec l'approche minimaliste [CR87][RCB90][WZ97], comme le fait de situer l'activité dans un contexte spécifique, et de tenir compte des tâches du domaine, il n'en reste pas moins qu'il peut y avoir une ambiguïté entre le rôle du concepteur et celui d'enseignant. Lors d'un atelier, s'agit-il d'apprendre à programmer ? La question se pose si les participants ne connaissent pas le langage de programmation, ce qui arrive nécessairement s'ils ne sont pas des programmeurs expérimentés.

2.3.2.6 Un cadre conceptuel et méthodologique pour la programmation par l'utilisateur ?

La programmation par l'utilisateur peut aussi n'être considérée ni comme un but, ni comme un moyen, mais comme un mode d'utilisation particulier qui arrive lorsqu'il n'y a pas d'autres solutions. Ce statut doit à la fois être supporté par l'environnement et soutenu par la méthodologie, comme c'est le cas dans la conception participative : la rupture, la panne sont modélisés comme détachement réflexif [EK91] [Mør97c]. Une approche comme celle de la customisation correspond d'ailleurs exactement à ce problème et le traite comme quelque chose de normal : il y a besoin de programmer quand il y a échec ou insuffisance de l'interface utilisateur ou de l'application.

La conception participative n'est donc pas seulement un espace d'exploration de la diversité et des exceptions par rapport aux procédures standards [KM95][Tri92], mais elle est aussi une oppor-

tunité, par les procédés d'effectuation, pour faire émerger des situations de programmation, mais en permettant d'identifier quand celle-ci est nécessaire *et de quelle manière* : par édition directe de code, par programmation par démonstration, par un éditeur spécialisé ? Comme les ateliers sont un lieu où s'évaluent les priorités - une des fonctions de la conception de systèmes informatiques n'est-elle pas de hiérarchiser les besoins ? - cela se produit également pour les différents techniques et approches de la programmation que nous avons recensées dans le chapitre I. [Gir00] remarque ainsi, à propos de la programmation sur exemples appliquée au contexte de la conception de composants mécaniques, que les paramètres ne devraient pas avoir à être déduits par le système, mais qu'ils peuvent au contraire être spécifiés par l'utilisateur qui les connaît bien de par son expertise : c'est ce type d'informations, concernant la pertinence de tel ou tel mécanisme, que la démarche participative permet de mettre en lumière, sans même que les participants aient d'ailleurs à connaître les concepts associés, comme ici la notion de paramètre. C'est dans l'*utilisation* de la maquette par le participant que le concepteur identifie le concept associé à l'objet. Pour rester dans le cadre de la programmation par démonstration, c'est aussi en interagissant avec des utilisateurs qu'on peut découvrir ou vérifier s'il est utile ou non de passer par une représentation symbolique du programme ou du modèle inféré : telle forme d'expression, comme celle des expressions régulières ou d'une grammaire non contextuelle, très adaptée aux structures d'ARN notamment, qu'on peut considérer hors contexte comme illisible pour un non spécialiste, peut s'avérer au contraire correspondre parfaitement à des connaissances du domaine dont le concepteur n'avait pas conscience, et ne poser aucun problème.

Dans un cadre plus "positif", nous avons également eu l'occasion de constater, ou du moins de pressentir, l'utilité d'un atelier pour explorer des techniques de type programmation interactive pour la résolution de problèmes. Cet atelier, qui a servi à spécifier un problème, est décrit dans ce chapitre.

2.3.3 Études réalisées.

2.3.3.1 Contexte, immersion : support technique, cours.

Ces études se situent dans un contexte à la fois de recherche dans le cadre de la thèse et de travail quotidien (installation de logiciels de biologie, support technique) apportant une connaissance pratique des outils utilisés par les biologistes. Une participation importante à un cours de programmation pour des biologistes (plusieurs mois de participation sur trois ans comprenant un cours de programmation en Ada et des travaux pratiques en Scheme) a également permis d'appréhender l'approche de la programmation par des non-professionnels scientifiques.

2.3.3.2 Analyse : études de terrains, entretiens, scénarios.

Une vingtaine d'entretiens ont été menés entre la fin de l'année 1998 et la fin de l'année 2000, soit sur le campus de l'Institut Pasteur, soit auprès de biologistes appartenant à d'autres organismes.

Les personnes rencontrées variaient du technicien effectuant une tâche ciblée, par exemple d'alignement de séquences de virus avec un outil bien précis, au chercheur menant de front plusieurs problématiques et utilisant de multiples techniques, allant de la paillasse à la programmation en passant par les statistiques. Certains des biologistes vus (un tiers environ) sont nettement des bio-informaticiens, c'est-à-dire des biologistes pratiquant une informatique technique de manière quotidienne (nous n'incluons pas l'activité d'annotations dans l'informatique technique) : soit en programmant dans des environnements classiques (C, C++), soit utilisant des langages visuels (LabView), outils de programmation dans l'interface (tableurs, HyperCard) ou des constructeurs d'interface (Visual Basic). Certaines parmi les personnes interviewées sont auteur de logiciels distribués, dont certains en C, d'autres en Visual Basic, et un autre sous la forme d'un ensemble de macros pour un tableur. L'une des personnes était responsable de l'informatique de son unité, une autre enseignante à l'université (elle avait enseigné Visual Basic à des étudiants en biologie). L'autre partie des personnes étaient soit des utilisateurs experts de l'informatique, connaissant bien les méthodes et les outils, mais ne pratiquant pas la programmation, soit des programmeurs très occasionnels, ayant dans leurs fichiers et dans leur documentation un certain nombre de scripts en shell, en perl ou en awk pouvant resservir à l'occasion.

Ces entretiens ont pour la plupart été réalisés avec une prise de notes écrites. Quelques séances (4)

ont donné lieu à un enregistrement vidéo, réalisé avec l'aide d'une chercheuse en conception participative (Wendy Mackay). Cette technique est d'une utilisation délicate auprès de collègues chercheurs, même si l'on précise bien que seul l'écran sera filmé et que les images ne seront pas diffusées. La rédaction des notes de l'entretien a systématiquement été envoyée aux personnes rencontrées peu de temps après l'entretien, afin de vérifier l'exactitude des points scientifiques abordées pendant l'entretien.

Ces entretiens ont permis de mieux comprendre les difficultés rencontrées par les biologistes dans l'utilisation de l'informatique et sont une source d'informations très précises et très concrètes, indispensable pour la préparation des ateliers, la détermination d'éléments directeurs pour la problématique de la programmation par l'utilisateur et la réflexion sur la conception du prototype informatique.

2.3.3.3 Conception : ateliers participatifs.

Démarche générale.

Nous décrivons ici 6 ateliers :

1. premier atelier : prototypage pour le logiciel Alice (16 janvier 1998),
2. deuxième atelier : brainstorming pour l'éditeur d'alignement (4 avril 1999),
3. troisième atelier : prototypage pour l'éditeur d'alignement (5 mai 1999),
4. quatrième atelier : prototypage autour de la programmation (29 mars 2000),
5. cinquième atelier : prototypage à partir de biok (14 février 2001),
6. sixième atelier : prototypage d'un problème à résoudre (19 juin 2001).

La démarche participative a été choisie dès le début de la thèse, par conviction a priori mais aussi parce que, dans le cadre d'un travail sur la programmation par l'utilisateur dans l'interface d'une application existante, il fallait que la conception de cette application ait le moins d'incohérence possible par rapport au contexte de travail réel des biologistes. Un outil de travail mal fait n'est pas un environnement très engageant pour une approche en douceur de la programmation. Il fallait aussi choisir le type d'application dans lequel déployer ces capacités de programmation, et, en facilitant la communication entre des personnes de professions différentes, l'approche participative semblait un bon moyen d'exploration. Le premier atelier a eu cette fonction : son objet n'était pas exactement un éditeur d'alignement - outil choisi par la suite pour construire le prototype - mais une interface pour un logiciel d'extraction de motifs [SVS97]. Ce premier atelier a aussi servi d'initiation à ce type de technique, que j'avais seulement expérimenté rapidement lors d'une école d'été quelques mois avant (juillet 1997). Les deux ateliers suivants ont eu lieu un an après à un mois d'intervalle, après un séjour au laboratoire DAIMI (S. Bodker) de l'Université d'Aarhus, où j'avais pu participer à des entretiens avec enregistrement vidéo d'utilisateurs d'un éditeur de résaux de Petri sur le projet CPN2000 [MRJ00]. Ces deux ateliers ont concerné plus directement l'éditeur d'alignement : d'abord un atelier de brainstorming pour générer des idées et récolter des informations, puis un atelier de prototypage de deux des fonctions importantes de ce type d'outils. Un atelier de prototypage plus orienté programmation a suivi un an plus tard (mars 2000). Enfin, les derniers ateliers reposaient sur la manipulation d'un prototype informatique (février et juin 2001).

Entre les ateliers, qui se sont donc étalés sur 3 ans, des phases d'entretiens ont eu lieu, soit pour montrer des prototypes, soit pour préparer l'atelier, comme pour les deux derniers ateliers : l'atelier de prototypage des fonctions de programmation (fin mars 2000) a été précédé de démonstrations individuelles du prototype vidéo fait juste avant, l'atelier de protypage utilisant le prototype informatique (février 2001) a été également précédé de démonstrations de ce prototype. Il s'agit plus précisément de discussions autour des interactions et des fonctions qui sont montrées, et ces entretiens furent notamment l'occasion de préparer les données et les scénarios possibles pour l'atelier. D'autres entretiens intermédiaires ont été menés début 1999 pour montrer un premier prototype fait pour explorer des techniques de programmation et s'appuyant sur un scénario vu dans un atelier, et discuter autour des techniques correspondantes.

Des mini-ateliers avec un ou deux biologistes et organisés de façon plus impromptue ont également eu lieu afin par exemple de développer un point insuffisamment vu lors de l'atelier précédent. C'est ainsi qu'un scénario d'exportation de résultats de programmes d'analyse a été prototypé après l'atelier consacré à la programmation (fin mars 2000), et un mini-atelier sur la création et la programmation de

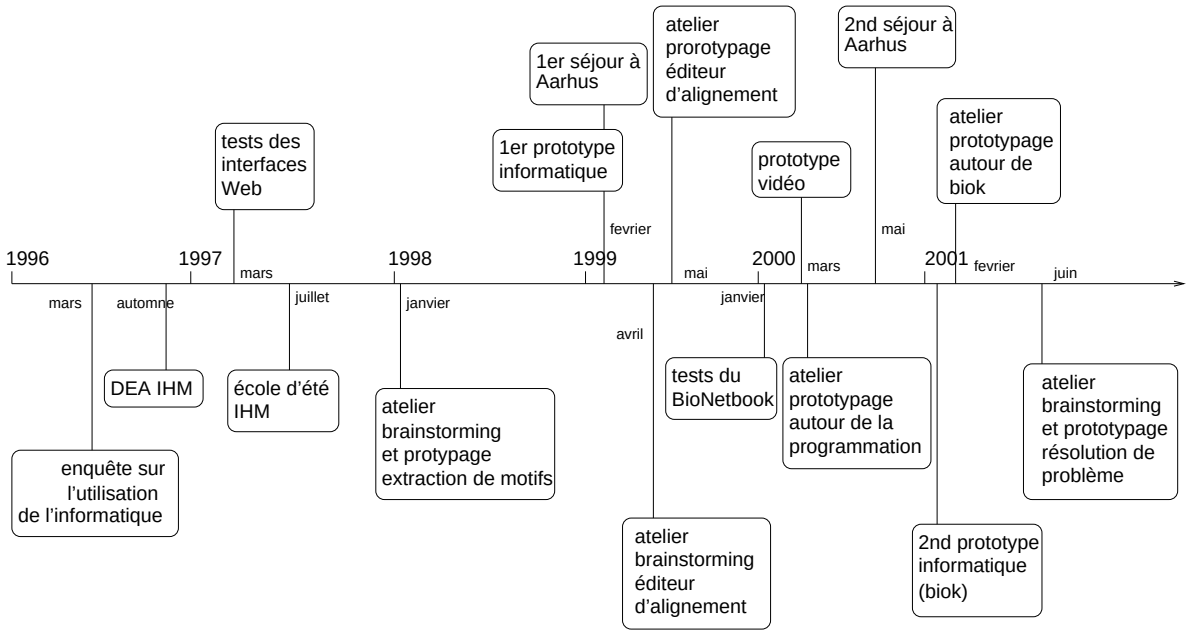


FIG. 2.12 – Calendrier

tags a eu lieu deux semaines après l'atelier sur le prototype informatique avec deux biologistes (février 2001).

La description qui suit se veut à la fois critique et pratique. En effet, un des problèmes les plus difficiles pour appliquer la conception participative est qu'il n'y a pas beaucoup de règles énonçables afin de laisser à cette approche son caractère hautement adaptatif. Il en résulte que le seul moyen de formation est alors l'apprentissage sur le tas, ce qui est l'occasion de bien des erreurs. Une autre difficulté vient du fait que les descriptions des expériences publiées dans la littérature sont souvent un peu militantes et idéalisées, et qu'on se retrouve un peu perdu lorsque tout ne marche pas comme dans les publications. Enfin, l'analyse critique des enregistrements et la synthèse de ces réflexions dans une rédaction sont justement le meilleur moyen d'améliorer l'approche.

Premier atelier : brainstorming et prototypage autour d'une interface pour un logiciel d'extraction de motifs.

16 janvier 1998

Le premier atelier s'est déroulé en janvier 1998 autour de l'interface d'un logiciel d'extraction de motifs à partir de séquences, Alice, écrit par Marie-France Sagot, qui était présente durant l'atelier. La séance a duré environ 2 heures et a été enregistrée (enregistrement audio), mais le début a été perdu (15 premières minutes).

Participants

Les participants étaient des biologistes (4) et des informaticiens (2), dont l'auteur du logiciel Alice, algorithmicienne, et moi-même. Parmi les biologistes, l'un était un programmeur entraîné, un deuxième avait déjà programmé en Fortran ou en C. Un troisième était un habitué des logiciels d'analyse de ce type. Les autres participants étaient moins habitués.

Objectifs de l'atelier

L'atelier était présenté comme une séance de discussion et d'évaluation constructive d'idées d'interfaces graphiques pour le logiciel Alice.

Description du logiciel Alice

Alice est un logiciel implémentant un algorithme d'extraction de motifs non connus à partir d'un jeu de séquences biologiques [SVS97]. Le résultat du logiciel est un texte comportant une liste de modèles sous forme d'expressions régulières, auxquels sont associés la longueur du modèle et le quorum, et la

liste de ses occurrences (avec leur position de début) dans les séquences (repérées par leur numéro d'ordre dans le fichier).

```
MODEL   14      50      [YWF][RKH][PG][TSA][VMLI][VMLI][VMLI][VMLI][RKH][VMLI][QN][RKH][GA][GA]
        3       85      W K P T L V I L R I N R A A
        4       85      W K P T L V I L R I N R A A
        5       96      Y K P T L V L L R I N R A A
```

Le but de l'interface graphique est essentiellement de permettre une meilleure navigation parmi les résultats, notamment pour visualiser les occurrences de chaque modèle dans les données et pour sélectionner les modèles intéressants en vue d'analyses ultérieures.

Préparation

La préparation de l'atelier a consisté à construire une maquette en papier et à choisir un scénario comportant des données et des actions sur ces données, similaires à ce qu'on peut faire avec le logiciel Alice.

Le choix du scénario s'est fait sur la base d'entretiens effectués au cours des mois précédents auprès de biologistes effectuant des analyses similaires à celles du logiciel Alice. Marie-France avait assisté à deux de ces entretiens. Le scénario choisi était une recherche de fonction analogue parmi des séquences assez différentes. Ce sont les données de ce scénario qui ont été utilisées pendant l'atelier, car elles se prêtaient bien à l'extraction de motifs ; ce sont également les résultats réels du logiciel qui sont utilisés pour la maquette.

Un prototype papier avait été préparé avec l'aide de l'algorithmicienne pour éclaircir les points clés de l'algorithme et pour discuter des utilisations potentielles des résultats. Des éléments assez précis de la conception avaient été prévus, comme une ébauche de langage d'interaction (simple clic = affichage dans l'autre fenêtre, double clic = sélection, shift-clic = modification de la sélection, etc...), ou bien des types particuliers de navigation entre les fenêtres, ainsi que des boutons ou des menus avec des fonctions plus ou moins précises.

FIG. 2.13 – Maquette du 1er atelier

La photo de la figure 13 montre la maquette en papier utilisée pendant l'atelier. Elle est composée de deux grandes fenêtres principales : l'une contenant de vraies données (les séquences de travail d'un des biologistes), l'autre contenant les résultats - également des motifs réellement trouvés par le logiciel Alice. Des composants en papier avaient été préparés : barre de défilement, palette de couleurs, boîtes de dialogue, sélecteur de fichiers, fenêtres contenant des informations venant des banques de données

(annotations sur les données), vue zoomée.

Une mini-maquette en HTML avait été conçue pour explorer les possibilités de navigation dans les données et résultats du logiciel ou tout simplement comme exemple pour discuter avec l'auteur du logiciel. Elle n'a pas été utilisée lors de l'atelier.

Séance

L'atelier a duré 2h30. Il a commencé par une description des fonctions du logiciel par l'auteur, puis par l'explication du fonctionnement de l'atelier, et la présentation du matériel de maquettage. Ensuite, il a principalement reposé sur une revue des spécifications informelles et non-informatiques matérialisées par la maquette. Cette revue - c'était un de ses objectifs - a donné lieu à de nombreuses discussions concernant l'algorithme, son utilisation ainsi que la conception de l'interface graphique. Le prototype préparé d'avance servait donc de support de brainstorming, à titre de spécification incomplète ; il ne s'agissait pas de le valider. La discussion se voulait ouverte, et plusieurs des sujets abordés n'étaient pas prévus.

Tous les aspects discutés ne peuvent pas être mentionnés ici. L'atelier a été l'occasion de discuter de l'algorithme et pouvait susciter quelques idées, concernant plus son interface que le noyau lui-même. Ainsi par exemple, la possibilité de réitérer les recherches sur des zones dans lesquelles des occurrences de modèles avait été trouvées. Une des possibilités intéressantes de post-traitement des résultats qui a été évoquée est la réduction de l'ensemble des symboles d'un groupe en cas de non-occurrence.

Il a été intéressant notamment d'explorer la zone mixte visualisation / algorithme, où certaines demandes de visualisation correspondaient potentiellement à une légère modification de l'algorithme, et où la méthode participative a montré son intérêt comme support pour générer des idées à faible coût de développement. Parmi ces demandes, souvent toutes simples : celle sur l'ordre d'affichage évoquait un tri par score (non prévu), ou bien la demande sur l'étendue des groupes de lettres à partir desquels sont construits les modèles.

Commentaires

Objectifs de l'atelier

Il s'agissait d'un premier atelier, donc d'une sorte de démarrage d'une activité participative avec des biologistes du campus. Les objectifs de l'atelier n'étaient sans doute pas suffisamment définis. Ainsi, la description de l'atelier dans le message envoyé aux participants potentiels - qui avaient été contactés par téléphone auparavant ou qui avaient été interviewés - n'en donne qu'une idée assez vague :

Je suis en train d'organiser une séance de "conception participative" pour une interface graphique au logiciel d'extraction de motifs de Marie-France Sagot. Il s'agit d'une séance où un scénario de travail (observé) est déroulé avec une maquette (papier) devant des utilisateurs. Les réactions et discussions que cela provoque permettent de mieux réaliser la conception de l'interface. Il n'est pas nécessaire d'être un expert sur le sujet, au contraire.

Inutile de dire qu'il y a eu des réactions à la dernière phrase - qui constituait plutôt un contresens, puisqu'au contraire les méthodes participatives ont pour but de partager les expertises et de reconnaître celles des utilisateurs ! Il reste cependant que cette mention s'était auparavant avérée utile à plusieurs reprises dans les interactions avec les utilisateurs, qui pensent souvent, dès qu'il s'agit d'informatique, qu'ils sont incompétents. De plus, le type d'analyse effectué par ce logiciel correspond souvent à une utilisation experte des logiciels. Ce n'est pas ce qu'on apprend en premier.

D'autres descriptifs - notamment le sujet du message adressé aux biologistes pour organiser effectivement l'atelier - confirment l'aspect un peu vague des objectifs de l'atelier :

- "séances maquette logiciel extraction de motif"
- "séance de test sur maquette"

A posteriori, on peut dire que les objectifs effectifs de l'atelier étaient les suivants :

- discussions sur l'algorithme et l'utilisation de l'algorithme,
- brainstorming et prototypage de l'interface,
- formation.

Pour les deux premiers, il s'agit des deux aspects noyau fonctionnel et interface graphique, sur lesquels en effet l'atelier a porté, avec une zone mixte algorithmique-interaction dans les phases d'uti-

lisation des résultats et de réitération des analyses. L'objectif de formation - non explicite - peut être considéré comme double : formation des biologistes, aussi bien à ce type d'analyse de données - de nombreux échanges ont eu lieu concernant le sens des paramètres ou des résultats - qu'à l'activité de conception d'interface, mais surtout formation du chercheur-concepteur (moi-même) aux techniques participatives.

Rôle du prototype

L'atelier était très vivant et très riche en idées. Il a aussi été l'occasion d'une prise de contact entre l'auteur du logiciel et de potentiels utilisateurs.

Cependant, même si les discussions étaient ouvertes et le fil de la séance relativement libre, le prototype était un peu perçu comme une spécification à discuter, sinon à valider - c'est ce que montre le message envoyé aux participants, cité ci-dessus. Les descriptions du prototype pendant l'atelier étaient, un peu comme s'il s'agissait d'une véritable démonstration (d'un prototype en papier pourtant !), assez affirmatives et factuelles comme :

- il n'est pas prévu de ...
- ce bouton sert à ...

Le sens de tels énoncés ne se voulait pas contractuel, mais c'est pourtant ainsi qu'il avait été perçu par les participants. On a pu constater ce statut du prototype dans les questions des participants :

- est-ce que c'est possible de définir des régions ?
- avec ton file browser, tu entends charger une ou plusieurs séquences ?

Peut-être la préparation de l'atelier avec l'auteur a-t-elle pu jouer le rôle - involontairement - d'une définition de contraintes à respecter, ce qui ce serait ressenti dans le ton. Autre explication : l'habitude d'être mis devant le fait accompli avec les logiciels développés entraîne ce mode de questions : des demandes plutôt que des discussions.

Participation des biologistes

Bien que des maquettes en papier aient été utilisées pendant la séance, il ne s'agissait pas d'une séance de prototypage participatif puisque les manipulations n'étaient effectuées que par moi-même. à plusieurs reprises cependant, des composants d'interfaces ont été ajoutés sur la suggestion d'un des participants : sortir la palette de couleurs pour créer une fonction d'annotation des données, créer un champs "commentaires" pour la séquence. Ce type de participation quand même assez asymétrique évoque un peu l'utilisation *a posteriori* des méthodes participatives dans [HJ93]. Ainsi, plusieurs occasions de prototypage par un utilisateur on été ratées :

- changement des positions de début dans les séquences,
- exportation d'un modèle ou d'un profil pour faire une autre analyse (de la part d'un participant qui a l'habitude de combiner des résultats de différents programmes),
- manipulation des blocs alignés par le logiciel Macaw [SAL91] : le participant a décrit les interactions, ce qui était compliqué, alors qu'il aurait été plus simple pour lui de les dessiner au tableau,
- sélections : bloquer un niveau de l'interface pour faire une sélection dans une sélection (cela aurait eu l'intérêt de montrer ce que le participant voulait vraiment faire avec ce type d'interactions)

Les interventions et questions des biologistes auraient pu être un peu mieux prises en compte. à plusieurs reprises, au lieu de saisir l'occasion d'une question ou d'une remarque pour appronfondir un point, j'ai préféré reporter à l'étape prévue pour ce sujet plus tard lors de l'atelier. Peut-être cela permet-il d'avoir une discussion plus structurée et plus organisée, mais cela fait perdre le contexte de la question qui était probablement utile. Par exemple, au moment où l'on parlait du chargement des séquences, un des participants a posé une question sur l'édition d'une séquence trop longue et dont seulement la fin est intéressante pour l'analyse : je lui ai répondu que c'était prévu à la fin de l'atelier, à deux reprises puisque très peu de temps après il a demandé si on pouvait faire l'analyse sur seulement une partie des séquences. Or ce problème se posera toujours au moment où le biologiste chargera ses données, car les données sont extraites de sources diverses et d'ensembles éventuellement plus grands que les zones d'analyse de motifs, comme c'est le cas si on veut analyser uniquement un domaine de protéines dans un gène plus grand, ou bien une zone particulière de l'ADN où l'on sait qu'il y aura des motifs significatifs comme des promoteurs. Il a donc été très utile d'avoir un enregistrement audio de

la séance pour repérer ces défauts.

Il y avait aussi trop d'explications sur la méthode suivie ou sur la conception d'interfaces, qui étaient trop générales et hors contexte, soit au niveau des logiciels (utilité ou non de l'aide en ligne, d'un mode d'emploi de 10 pages, ...) soit au niveau de la méthode participative (il faut faire ceci ou cela, exemples : "il faut utiliser un cas concret pour discuter", "j'ai introduit une complexité en parlant de la fonction search", "il faut bien décrire toutes les possibilités et voir celles qui ont un sens", ...).

Conception de l'interface

Du point de vue de la conception de l'interface, il y a eu des idées très intéressantes. Certains utilisateurs sont de bons concepteurs même si ce n'est pas leur rôle officiel dans les méthodes participatives (bien qu'il y ait divers points de vue sur cette question : [CCRN00] en fait au contraire un objectif explicite). Il s'agit d'une compétence qui vient de l'utilisation fréquente et experte de nombreux logiciels existants, principalement sur Macintosh pour ce qui est des interfaces graphiques, mais pas seulement.

Certaines remarques ou idées venaient de manière évidente d'une grande habitude des logiciels :

- utilité des menus pour signaler l'existence de raccourcis clavier,
- aspect standard des fonctions de type "Edit" (avec copier-coller),
- des standards plus bio-informatiques comme les codes couleurs pour les acides aminés,
- la coloration avec des couleurs suffisamment pâles pour une meilleure visualisation du texte ou des séquences,
- l'action de tirer sur la règle pour faire une sélection verticale de toutes les lignes,
- marquer les couleurs déjà utilisées dans la palette (si on veut utiliser des couleurs proches pour des modèles ou des données proches).

D'autres idées semblaient moins issues d'une culture logicielle :

- représentation différente par rapport aux autres symboles des modèles pour les jokers,
- ligne clignotante pour voir s'il y a des occurrences non visibles d'un modèle dans une phase de sélection.

On pourrait dire que l'atelier a peut-être eu légèrement tendance à porter un peu trop sur la conception d'interfaces plutôt que sur la tâche elle-même, ce qui est explicable par la présence d'un utilisateur expert d'applications de même type parmi les participants (quelqu'un qui voit autant le stylo que ce qu'il est en train d'écrire - voir plus haut [HJ93]) et qui connaît le langage de conception informatique.

Conception participative et génération d'idées nouvelles

Pour l'auteur de l'algorithme, en tant qu'utilisateur de cet algorithme sur des données réelles étudiées en collaboration avec des chercheurs, la plupart des idées générées n'étaient pas nouvelles. Mais peut-être la conception participative (et certainement d'autres méthodes de conception, du moment qu'elles tiennent compte des utilisateurs) sert-elle souvent à identifier parmi les idées de l'informaticien celles qui sont vraiment les bonnes, plus souvent qu'à trouver de nouvelles idées. L'informaticien a en effet toujours beaucoup d'idées qui viennent plus des possibilités techniques que de ce dont les utilisateurs ont vraiment besoin, ce qui donne des logiciels riches mais compliqués dont une toute petite partie seulement sert vraiment. Par conséquent, la conception participative sert paradoxalement plus à éliminer des idées qu'à en créer de nouvelles. Mais générer de nombreuses idées n'est pas non plus inutile ni forcément le seul fait de possibilités techniques, c'est certainement le travail du chercheur.

Autres remarques

- L'enregistrement audio est très utile mais insuffisant pour une séance de conception et de réflexion sur une interface, notamment pour toutes les discussions qui référencent les fenêtres pour les questions de navigation ou pour les discussions concernant les interactions avec la souris. La présence d'une caméra aurait cependant gêné certains des participants.
- Il y a quelques difficultés à faire du prototypage en papier d'applications qui n'ont de sens qu'avec des données volumineuses : il faut soit décider de se placer au niveau au-dessus, en courant le risque d'être hors contexte, soit prévoir une préparation compliquée avec beaucoup de manipulation de données avant l'atelier (cf données pour maquettes). Ceci étant, même avec des données issues d'un scénario réel, ce n'était pas suffisant pour que les biologistes trouvent une idée de motif à rechercher autour de ce qui était affiché.

- Les notes rédigées après l'atelier sont des notes qui décrivent les discussions et les idées de l'atelier. Une rédaction ainsi qu'une analyse de l'enregistrement plus méthodologiques auraient été utiles (notamment pour l'objectif de formation personnelle).

Développements à l'issue du premier atelier.

Le thème de cet atelier et quelques-unes des idées qui en sont ressorties ont servi de scénario pour la programmation d'un prototype informatique dont le but était plutôt technologique. Il s'agissait avant tout d'un terrain d'essais pour explorer les mécanismes d'introspection, de persistence et de programmation dans l'interface, mais ce prototype s'appuyait volontairement sur des idées émises lors de cet atelier, avec des biologistes, afin de correspondre à une situation réaliste. Ce prototype a été développé un an plus tard, lors d'un séjour d'un mois à l'Université d'Aarhus.

Ce prototype (décrit plus précisément dans le chapitre V) n'est de toute évidence pas conçu selon l'architecture prévue dans l'atelier : au lieu d'avoir des fenêtres indépendantes, il est contruit sur le principe d'incorporation de composants graphiques dans du texte. Parmi les idées de l'atelier qu'il reprend, on peut citer :

- l'annotation (avec des actions et des couleurs associées aux tags),
- la recherche dans les occurrences des modèles.

Mais surtout, ce prototype a servi de support pour des entretiens individuels ultérieurs avec plusieurs des biologistes qui avaient participé à l'atelier. Les discussions autour de ce prototype ont porté aussi bien sur les analyses possibles avec un tel outil, que sur les possibilités de programmation dans son interface. Cela a permis de montrer en quoi cette activité de programmation peut consister : construction d'éléments graphiques par clonage, modification ou création de scripts, création de tags et de traitants d'évènements graphiques. Le fait que ces tâches de programmation soient situées dans un logiciel d'analyse de séquences qui avait fait l'objet d'un prototypage - même si la séance avait eu lieu un an avant - lui donnait un contexte plus pertinent.

Conclusion

On ne peut que constater une grande tolérance des utilisateurs aux erreurs de communication ... (avec les informaticiens ils sont habitués !).

Deuxième atelier : brainstorming autour des éditeurs d'alignements.

4 avril 1999

Objectifs de l'atelier

L'outil éditeur d'alignement, bien que difficile à implémenter, avait été identifié au cours des différents entretiens comme un outil central dans le travail de l'analyse de séquences. Non pas que les éditeurs d'alignement fassent défaut dans le parc logiciel, au contraire, il en existe de très nombreux, décrits dans le chapitre IV (nous en avons recensé 40 pour une revue logicielle : `http://bioweb.pasteur.fr/cgi-bin/seqanal/review-edital.pl`). Ces outils proposent tous des fonctions très utiles, mais le problème essentiel pour le biologiste est que ces éditeurs ne proposent jamais un ensemble *minimum* de fonctions indispensables au travail d'alignement de séquences. La raison principale de cette situation vient selon nous d'une très faible pratique des démarches d'utilisabilité dans le domaine bioinformatique, aussi bien pour les logiciels lignes de commande, traditionellement destinés aux biologistes aimant la technique, que pour les logiciels prévus pour des personnes sans compétences informatiques particulières que sont les applications sur le Web ou surtout les applications graphiques. La simple notion d'effectuer des tests en dehors du laboratoire où a été développé le logiciel avec d'autres sujets que des collègues apparaît souvent comme assez saugrenue et surtout comme une perte de temps. Les logiciels sont souvent développés dans un but pratique personnel, par goût de la technique, ou très souvent dans le cadre d'une recherche scientifique fondamentale. C'est sans doute dans des départements informatique de grosses institutions à vocation de service à la collectivité, et dans lesquels l'activité de développement d'outils à vocation générale est reconnue, que les logiciels les plus ergonomiques ont pu être développés, comme au NCBI (National Center for Biotechnology Information du NIH), à Infobiogen ou bien à l'EBI (European Bioinformatics Institute).

L'atelier se proposait donc de faire le point de manière assez large sur ce type d'outils, et de générer des idées ou plus simplement de recenser les points qui semblent particulièrement importants aux biologistes.

Participants

14 biologistes participaient à cet atelier. La plupart d'entre eux avaient été consultés lors d'entretiens individuels. Presque tous étaient des utilisateurs du serveur Web d'analyses de séquences (chapitre V), et avait été contactés sur ce critère.

Séance

L'atelier a duré environ 2 heures. Deux groupes se sont formés avec pour but de générer une liste d'idées, de remarques, de fonctions, etc... en suivant les règles du brainstorming (voir la section 2.3.2.2 dans ce chapitre).

Chacun des groupes a rempli deux grandes feuilles de papier contenant pour l'un 25 points pour l'autre presque 40. A la fin, chacun coche les 3 idées qui lui semblent les plus importantes, ce qui détermine les 3 idées du groupe qu'un orateur présente à l'ensemble des participants en quelques transparents. Il n'y avait pas de thème affecté à chaque groupe, c'était donc complètement ouvert.

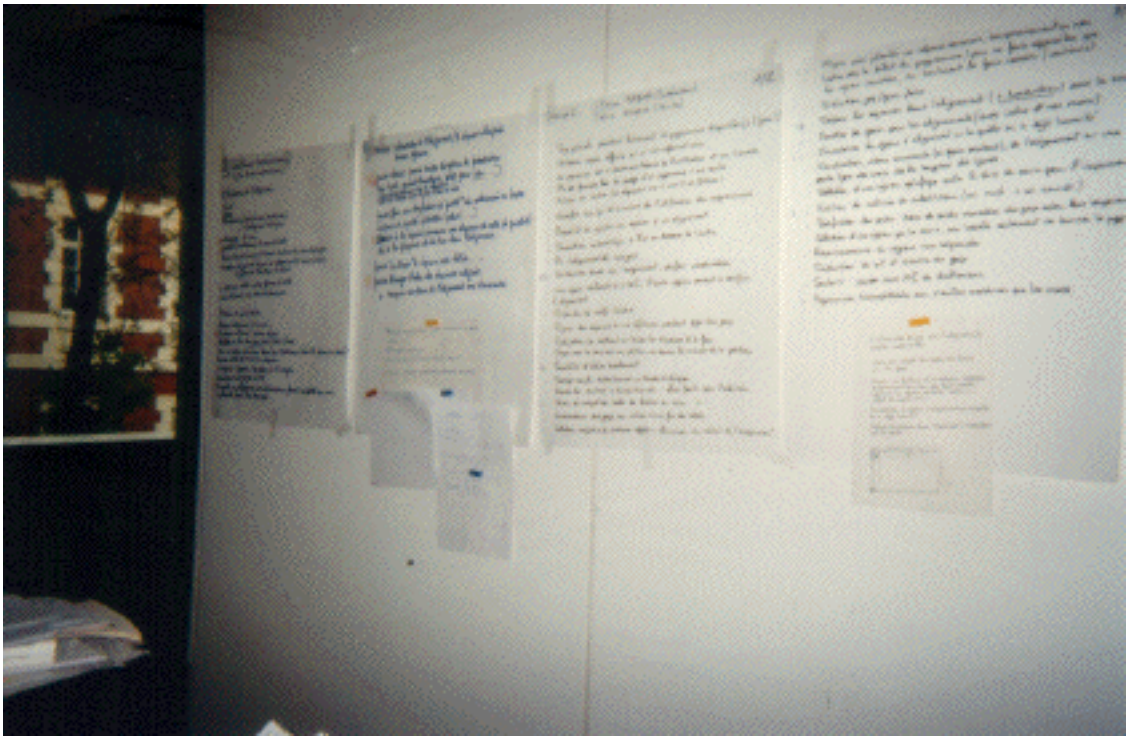


FIG. 2.14 – Résultats du brainstorming

Groupe 1

– A) réalisation de l'alignement :

1. clustal, pileup, Wconsensus (ce dernier génère la matrice de substitution et permet le réalignement de régions)
2. découpage à la main
3. possibilité de sélectionner les zones d'intérêt
4. Sequence Navigator : permet d'aligner directement les chromatographes
5. Macaw : alignements locaux en sélectionnant la zone d'intérêt
6. Macaw : signification statistique à la Blast
7. le programme idéal : tous formats d'entrée
8. pas de limites au nombre de séquences

- B) édition et présentation :
  1. retaper l'alignement à la main
  2. alignement en fixant certaines régions
  3. fixer des zones dans le fichier d'entrée
  4. Mise en évidence de zones choisies par l'utilisateur (dans la séquence et)
  5. pouvoir modifier le "titre" de la séquence
  6. Interface de SeqVu, souplesse de l'interface
  7. sauvegarde du projet initial (pouvoir revenir aux séquences de départ après toutes les éditions et tous les alignements)
  8. ressortir un alignement intermédiaire en un format compatible avec une recherche dans les banques
  9. MacBox : présentation de l'alignement par rapport à une séquence désignée comme référence
  10. pouvoir choisir parmi toutes les options de présentation (soit tirets quand homologie, points pour gaps,...)
      - indiquer toutes les bases ou tous les acides aminés
      - nombre de bases par ligne, blocs ou non, ...
  11. pouvoir faire une classification de "qualité" des acides aminés ou bases ; existence de qualités pré-établies (polarité)
  12. associer à la séquence consensus une séquence de notes de qualité liée à la fréquence de la base dans l'alignement
  13. pouvoir "surcharger" la séquence avec hélice
  14. pouvoir changer l'ordre des séquences alignées
  15. masquer une zone de l'alignement non intéressante

Groupe 2

1. page générale présentant brièvement les programmes disponibles (plus que 2 lignes !)
2. utilisation rapide difficile pour un non informaticien
3. le programme doit s'adapter aux besoins de l'utilisateur et non l'inverse
4. problème des formats lors du passage d'un programme à un autre
5. pratique de rentrer des séquences une à une
6. exemples au fur et à mesure de l'utilisation des programmes
7. possibilité de rajouter une séquence à un alignement
8. traduction automatique + l'un en dessous de l'autre (et associer la ligne de traduction avec la séquence, pas comme dans SeqEd)
9. problème d'alignement dû aux gaps
10. contraintes dures sur l'alignement, parfois modulables
11. une région contrainte à 100%, l'autre région servant à vérifier l'alignement
12. recherche de motifs itérative
13. aligner les séquences à une référence possédant déjà des gaps
14. modification sur toutes les séquences à la fois (par exemple : insertion)
15. cliquer avec la souris sur un position : donne le numéro de la position
16. possibilité d'éditer localement
17. passage souple acides nucléiques -> acides aminés

18. garder les couleurs + acides aminés = plus facile pour l'édition
19. choix du respect du cadre de lecture ou non
20. localisation des gaps au milieu ou en fin de codon
21. sélection négative de certaines régions, pour les éliminer du calcul de l'alignement
22. aligner puis présenter une séquence consensus, temporairement ou non (option dès le début du programme, contrairement à certains logiciels qui ne le proposent qu'en option d'impression) : sert à faire apparaître uniquement les régions variables; ou les faire ressortir (couleurs)
23. déplacer les séquences dans l'alignement (en compagnie de la traduction) avec la souris
24. fonction de zoom pour les alignements (avec icône et non menu)
25. immobiliser les régions de l'alignement sur lesquelles on a déjà travaillé
26. visualisation, même minuscule (si zoom existant) de l'alignement sur une seule ligne -> choix de la longueur des lignes
27. sélection d'une région spécifique entre 2 clics de souris pour l'imprimer
28. édition de matrice de substitution (acides nucléiques ou acides aminés)
29. modification des poids : choix de poids variables des gaps selon leur longueur
30. sélection à la souris de la région à aligner
31. reconnaissance des régions non séquencées (les 'N' : nettoyage automatique)
32. traduction de part et d'autre des gaps
33. couleurs : penser aux 10% de daltoniens (une palette spéciale :-) ?)
34. programmes portables sur d'autres machines que les Macs

Les deux groupes ont présenté leurs idées de manière assez différentes. L'un avait classé ses idées par thèmes - ce qui n'était pas demandé - c'est d'ailleurs le groupe qui a généré le moins d'idées sur le plan quantitatif.

Les idées exprimées par chacun des groupes montrent une certaine habitude de ce type d'outil. Plusieurs fois, ce sont des fonctionalités d'un outil précis qui sont demandées, soit pour des calculs particuliers, comme wconsensus [HrHS90], Macbox ou Macaw [SAL91], soit pour des fonctions interactives comme pour Sequence Navigator, soit pour la qualité générale (Seqvu [seq96]), soit comme exemple négatif (seqed, de GCG [Wom00]). Les idées proposées sont donc souvent fondées sur une expérience, ce qui fait qu'elles sont assez précises et assez pratiques. On peut être frappé par la simplicité des demandes les plus fréquentes : pouvoir choisir le caractère qui représente un gap (point ou tiret), changer l'ordre d'affichage des séquences, pouvoir sélectionner des zones dans l'alignement. D'autres idées sont moins classiques, mais aussi souvent demandées qu'elles sont peu offertes par les logiciels : fixer certaines régions de l'alignement, c'est-à-dire donner en paramètre au calcul des contraintes à respecter a été demandé par les deux groupes, ainsi que la possibilité de verrouiller contre l'édition certaines régions de l'alignement une fois alignées par le logiciel et éditées par le biologiste. Deux des fonctions les plus importantes choisies par le deuxième groupe étaient la prise en compte du cadre de lecture et l'affichage d'une séquence consensus à la demande : aucune de ces deux fonctions ne présente de difficulté particulière d'implémentation. Etait enfin présenté comme le programme idéal (mais avec un seul vote) celui qui accepterait tous les formats de séquences en entrée.

Discussions après l'atelier

Bien qu'ayant eut lieu par messages électroniques et page Web interposées, c'est pour cet atelier que les discussions ont été les plus actives. La page Web de compte-rendu de l'atelier contenait une liste assez complète d'adresses référençant la plupart des logiciels comparables à celui dont l'atelier avait été l'objet. La page contenait également les listes d'idées des deux groupes avec les votes marqués de points bleus et de points rouges pour les idées choisies pour l'exposé. Certaines de ces idées pointaient sur des questions en bas de page, auxquels plusieurs des biologistes ont répondu par email - d'où l'idée

d'utiliser un serveur Web de discussions ("wiki") pour les ateliers suivants (qui est fait pour ça mais qui a toutefois moins bien marché que les échanges par messages électroniques).

Troisième atelier : prototypage autour des éditeurs d'alignements.

5 mai 1999 Un troisième atelier a eu lieu en mai 1999 pour commencer à faire du prototypage d'interfaces de fonctions d'un éditeur d'alignement. La séance était filmée par une chercheuse en philosophie qui s'intéressait à la conception participative.

Participants 11 participants, dont 9 biologistes, une philosophe et moi-même.

Objectifs de l'atelier L'objectif de ce troisième atelier était de prototyper deux des fonctions parmi celles qui se dégagaient du brainstorming de l'atelier précédent.

Déroulement

L'atelier, qui a duré environ 2 heures, a commencé par une phase préparatoire de présentation du "matériel" et du "vocabulaire", sous forme de composants en papier (souris en carton, post-it, boîtes de dialogue, barres de défilement...) et sous forme de transparents représentant des widgets (doubles listes, ascenseur, fenêtres à panneaux, zoom avec barre de défilement, onglets, fonction crop). Les données papier, des séquences biologiques, avaient également été préparées en abondance.

Une "démonstration" de la technique de maquettage papier avait été préparée avec une des biologistes, qui avait déjà participé à un atelier de prototypage sur le thème des cahiers de laboratoires augmentés. Nous avions préparé un maquettage autour d'un petit scénario simple mais très réaliste : pré-aligner des séquences d'ADN de façon à les présenter dans une même phase de lecture, pour pouvoir par la suite effectuer des interactions, comme des sélections multiples, qui aient un sens biologiquement parlant - la possibilité de choisir le cadre de lecture avait été une des fonctions plébiscitées lors du brainstorming de l'atelier précédent. La maquette était un ensemble de séquences imprimées alignées sur un tableau magnétique. Les séquences affichées étaient des séquences d'ADN, mais dans des phases de lecture différentes : l'affichage des acides aminés, dans lesquels la biologiste pouvait repérer un motif connu, a permis de recaler les séquences. Un problème s'est produit : il fallait enlever un gap à un endroit donné, et le papier était assez peu pratique pour cette manipulation ! Cette démonstration a montré deux choses : qu'on pouvait simuler une tâche biologique avec du papier - la biologiste ne s'est absolument pas laissée perturber par les problèmes de manipulations et a déroulé l'action jusqu'à la fin - et que l'important n'était pas de faire une maquette parfaite. Cependant il apparaît, comme dans le premier atelier, qu'il est difficile de maquetter avec des données complexes comme des séquences.

Les fonctions à prototyper ont ensuite été choisies dans la liste établie lors de la séance précédente, reclassée par thèmes : alignement, visualisation, traductions, annotation et publication, et édition. J'ai proposé certaines fonctions qui me semblaient intéressantes, dont la recherche de motifs connus et l'annotation qui sont des sujets récurrents, comme on l'a vu dans le premier atelier. Ces thèmes apparaissaient également bien placés dans les votes de la fin du brainstorming. Deux groupes se sont constitués avec pour tâche, après une préparation de trois quarts d'heure, de présenter un petit scénario autour de la fonction choisie.

La phase de discussion des deux groupes n'est pas bien captée par l'enregistrement video - la caméra ayant été parfois orientée vers un groupe, parfois vers un autre, mais à chaque fois un peu loin - mais on constate deux manières de fonctionner assez différentes. Le groupe ayant choisi le thème de la recherche de motifs connus a discuté pendant 30 minutes assis autour d'une table avec papier et stylos ; le groupe ayant choisi l'annotation, activité peut-être plus interactive - plus "manuelle" dira le présentateur - était debout autour d'un tableau magnétique (photo de la figure 15) et discutait avec des fenêtres, des couleurs et des dessins (un chromatographe). Chaque groupe a ensuite présenté son scénario, qui était filmé.

Le premier groupe avait établi un protocole complexe de recherche incrémentale de motif, avec des contraintes, des marges d'erreur et un langage graphique pour délimiter des zones de l'alignement. Le matériel utilisé est montré sur la photo 16.

La figure 17 schématise les fenêtres, matérialisées par des post-its, nécessaires à la définition des seuils d'erreurs. Il était possible de faire varier le nombre d'erreurs autorisées.

Dans la figure 18 sont représentés la fenêtre affichant la liste des motifs recherchés, ansi que leurs occurrences dans l'alignement - il pouvait y en avoir plusieurs - avec des dégradés de couleur pour

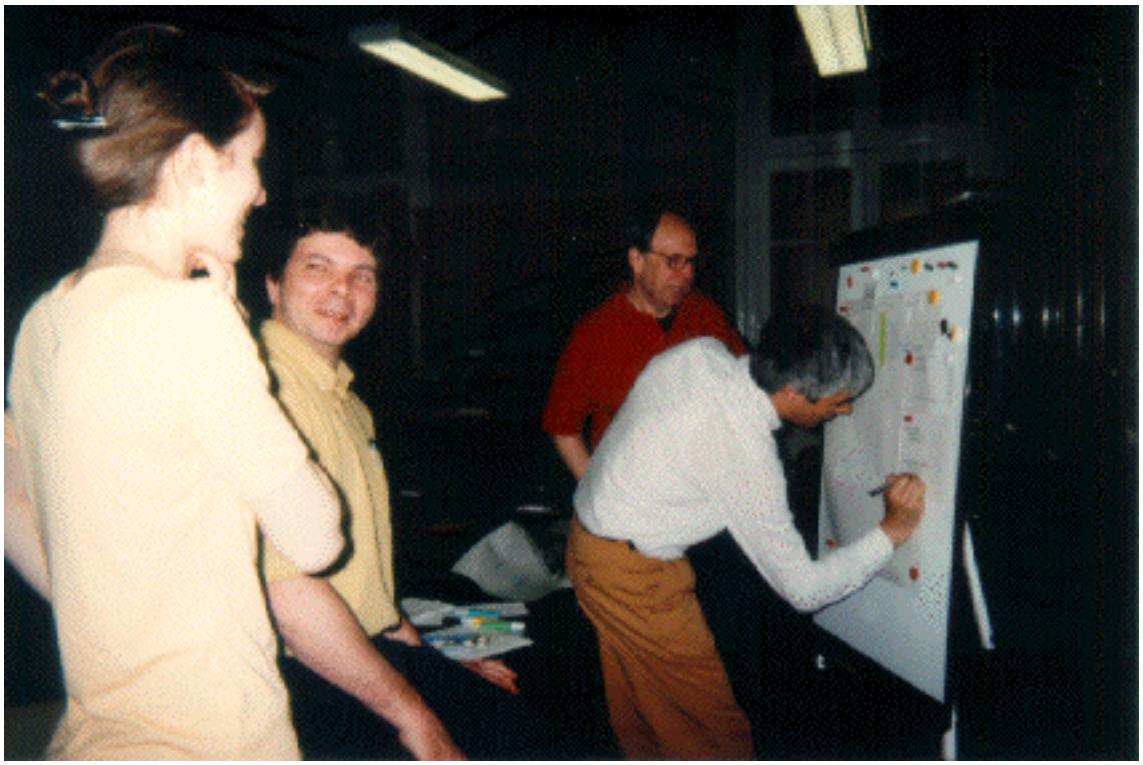


FIG. 2.15 – Le groupe "annotations"

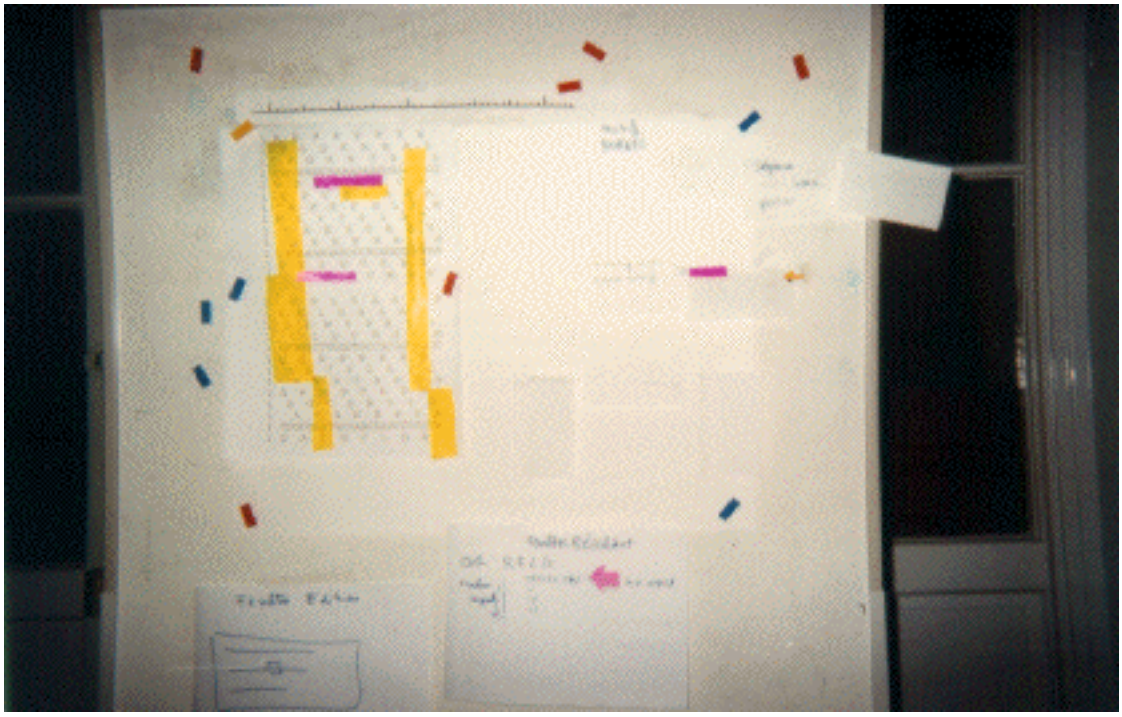


FIG. 2.16 – Le groupe "recherche de motif"

marquer le plus ou moins bon score de l'occurrence. La notion de contexte d'une recherche était également spécifiée, comme on le voit dans la figure 16 à travers les deux bandes verticales jaunes qui indiquent les limites de la zone de recherche. Ces limites peuvent être spécifiées à la souris, mais

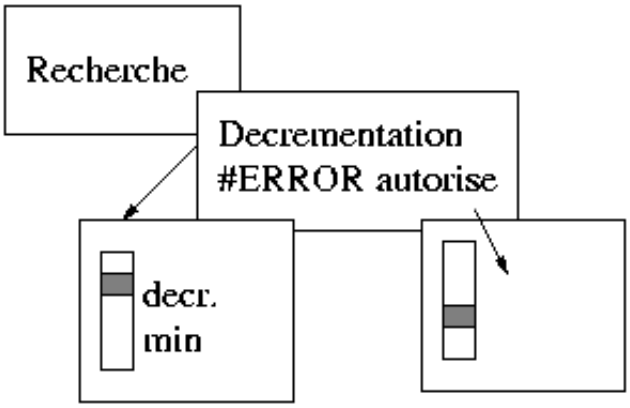


FIG. 2.17 – Recherche avec erreurs

également par une recherche précédente ou encore par un motif complexe composé de plusieurs blocs, comme celui qui est disponible dans le prototype actuel (chapitre V, section "Recherche de motifs"). C'est ce qu'un des participants avait dénommé une définition des bornes par leur contenu (exemple : TGGTGG).

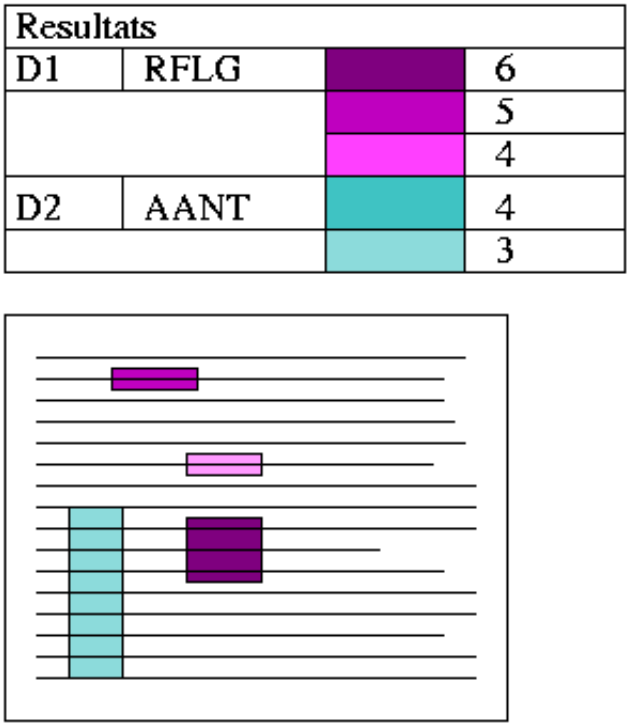


FIG. 2.18 – Dégradés de couleurs pour hiérarchiser les résultats

Le groupe "annotations" (figure 20) a exploré les différentes manières de rajouter des commentaires sur les données, ainsi que de structurer les liens entre ces notes : imbrication, chaînage. Notamment, c'est à travers le problème pratique de l'impression que s'est déroulé le scénario et que la question de la structure adéquate a été posée. Or l'impression est une fonction qu'on relègue souvent à la phase

des additions finales sans doute parce qu'elle est techniquement pénible et qu'elle relève de la catégorie moins passionnante des entrées-sorties ou des exportations.

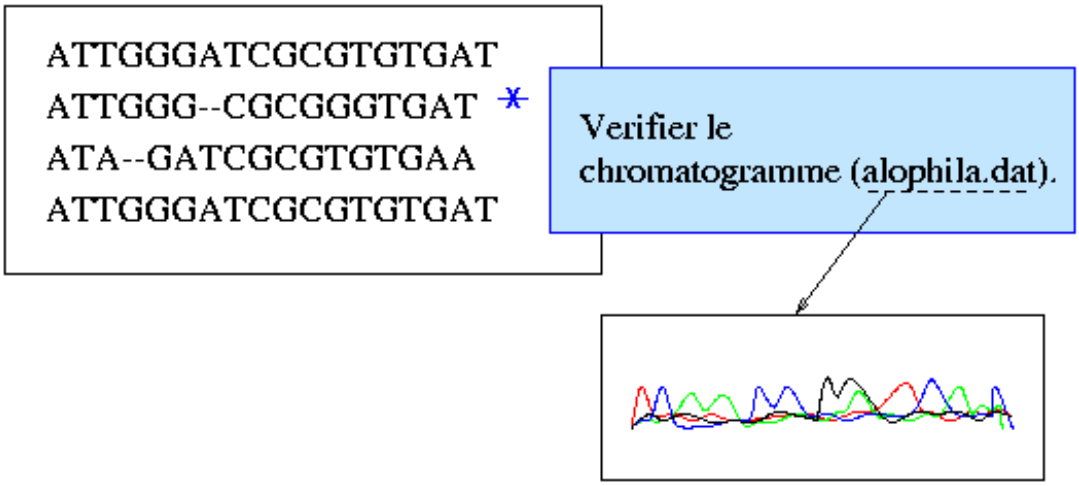


FIG. 2.19 – Commentaires

Commentaires

Objectifs de l'atelier

Si l'un des objectifs était de se servir du prototypage pour susciter des discussions, cet objectif aura été rempli. Les participants m'ont fait part de l'intérêt que représentaient pour eux ces ateliers pour discuter ensemble de questions scientifiques. Mais il aurait fallu qu'il y ait dans chaque groupe deux animateurs au courant du sujet, qui se seraient retrouvés ensuite pour échanger leurs remarques, ou bien deux caméscopes placés plus près car toutes les conversations des groupes ont été perdues. Une autre solution aurait été de ne faire qu'un seul groupe et un seul thème, et cette remarque est générale pour tous les ateliers, sauf peut-être pour l'atelier de brainstorming.

Les deux groupes étaient en fait très complémentaires - on fera la même remarque pour un autre atelier. Par exemple, il n'est pas inutile de pouvoir annoter une occurrence de motif dans un alignement, en particulier si cette occurrence est trouvée dans un contexte qui lui donne une meilleure signification, information qu'aucun programme ne pourra donner. Inversement, une recherche de motif doit pouvoir être effectuée dans le contexte d'une zone préalablement annotée, et pourquoi pas identifiable au niveau d'un script en fonction de mot-clés figurant dans ces notes. Ou encore, une note peut être associée automatiquement à une zone ayant servi au paramétrage d'une recherche.

Cette complémentarité des éditions et des résultats de programmes, illustrée dans la figure 20, est représentative d'une bipolarité constante dans l'analyse de séquences biologiques entre ce qui relève de l'expertise et ce qui relève des algorithmes.

Programmation

Des fonctions ou des objets ayant trait à la programmation se sont présentés à trois reprises dans l'atelier :

1. scripts : dans le cadre du thème annotations, c'était implicite, puisque, comme on le voit sur la figure 19, les notes constituent parfois une sorte de "script" graphique qui mémorise les actions faites ou à faire sur telle et telle sous-zone ; cette fontion de l'annotation avait été observée dans un entretien préalable : lors de l'édition d'un alignement, à la fois pour corriger les séquences et réaligner visuellement les parties incorrectement alignées par le logiciel, le chercheur avait noté dans une fenêtre "bloc-notes" extérieure à l'application les bouts de séquences (quelques positions) qu'il restait à corriger ou à repérer ;
2. langage de définition de zones : dans le cadre de la recherche de motifs connus, il a été question

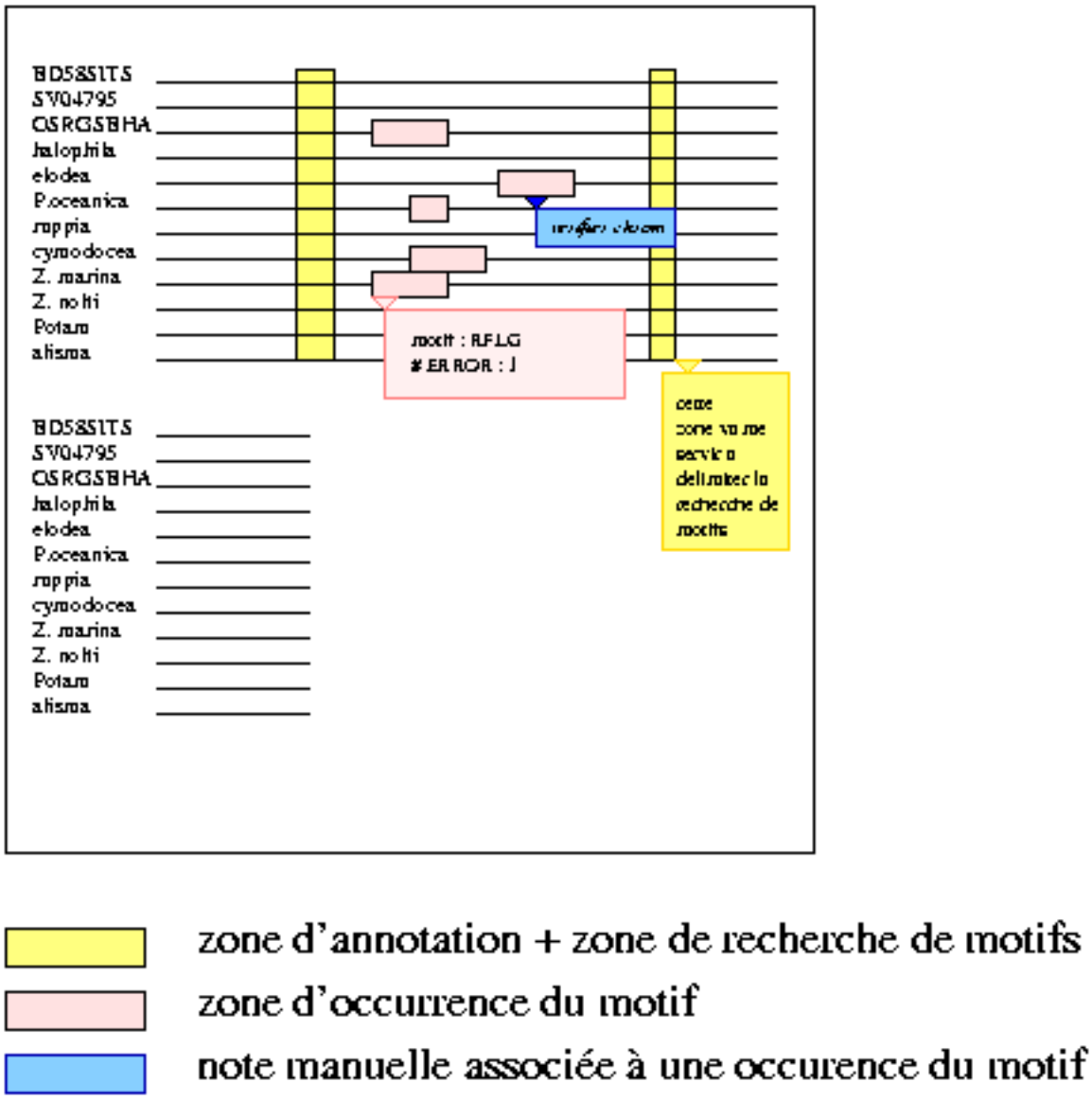


FIG. 2.20 – Annotations et recherche de motifs

de pouvoir spécifier la zone de recherche par des positions, ansolues ou relatives, ou par des intervalles ;

3. manipulation des motifs : il y avait nettement une notion variables pour les identifiants de motifs recherchés.

Exemples technologiques

Pour illustrer l'idée de recherche incrémentale, j'ai décrit le prototype de recherche de texte de [BL00], dont on voit un exemple dans la figure 21, en le dessinant sur un transparent.

Dans ce prototype, toutes les occurrences sont présentées dans le texte et dans la barre de défilement, et sont mises à jour à chaque modification de la chaîne de recherche. La démonstration de ce prototype a souvent un effet très positif car il illustre un type d'interaction qui serait très utile en analyse de séquences mais qui est peu utilisé : les biologistes ont en effet besoin de pouvoir explorer interactivement

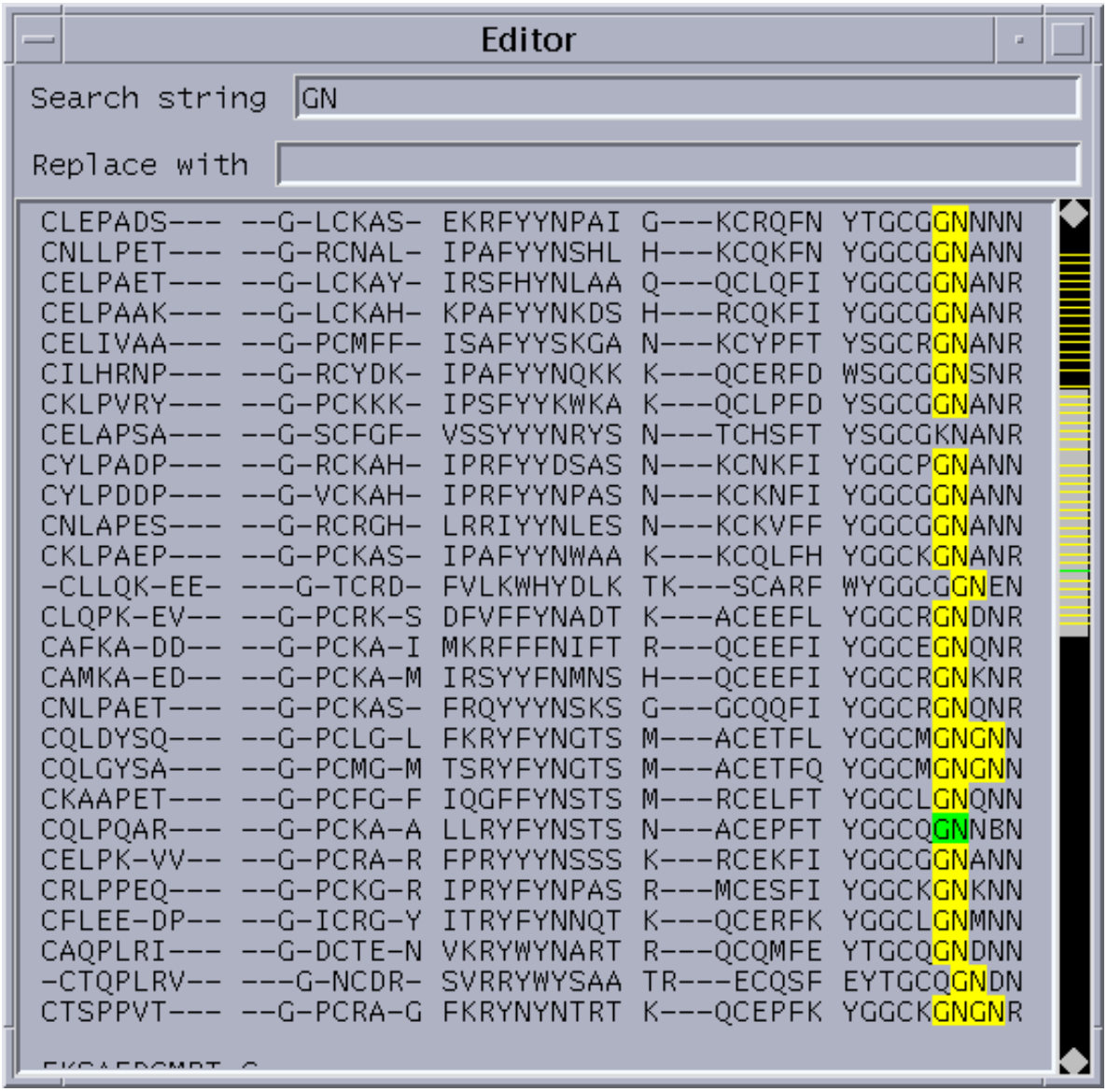


FIG. 2.21 – Requêtes dynamiques

un espace important de données en visualisant immédiatement le résultat pour éventuellement adapter la recherche.

Un autre exemple tiré de la recherche en techniques d'interaction est celui des liens fluides [ZCM98] dont on pourrait appliquer l'idée pour l'annotation de séquences comme dans la figure 22.

Le principe des liens fluides est de permettre de prévisualiser une information concernant le document associé à un lien hypertexte, ce qui évite d'avoir à rapatrier le document ou l'afficher dans une fenêtre supplémentaire. La possibilité de visualiser des annotations de manière transitoire et légère présente un intérêt pour l'analyse de séquences dans la mesure où les annotations peuvent être assez nombreuses, prendre beaucoup de place et ne pas toutes présenter un intérêt général.

Développements à l'issue de l'atelier.

Un certain nombre des techniques de recherche de motifs ainsi que certaines fonctions d'annotation ont été implémentées dans le prototype biok. Le lien entre annotation et zone de recherche et d'occurrence de motifs a également été exploité, puisque le mécanisme pour réaliser ces deux fonctions est à peu près le même (mécanisme de tag, présenté dans le chapitre V). L'accès programmatique a été

FIG. 2.22 – Liens fluides

réalisé à travers une classe permettant de manipuler ces zones.

Quatrième atelier : prototypage autour de la programmation.

29 mars 2000

Participants

Neuf participants, soit huit biologistes et moi-même. Parmi les biologistes, définis comme tels par leur formation initiale, sept sont des anciens élèves du cours de programmation de l'Institut Pasteur ; au moins trois pratiquent la programmation couramment (en Scheme), trois occasionellement (C, Scheme, perl ou scripts shell et pages Web). **Préparation** Durant les semaines qui ont précédé cet atelier, un prototype vidéo de l'environnement d'objets graphiques reposant sur un modèle dataflow a été élaboré pour illustrer les fonctions envisagées pour la programmation du prototype biok. Ce prototype vidéo a été construit en filmant des maquettes en papier, en suivant un storyboard précis. Les fonctions filmées ont été implémentées effectivement dans biok :

- les objets graphiques et leur structure,
- les formules,
- les commandes,
- l'éditeur d'alignement,
- les fonctions de programmation, incluant la modification simple d'une méthode, la création d'une nouvelle méthode simple, la modification de la structure de l'objet pour ajouter une fonction

(par exemple, pour le filtrage des colonnes de l'éditeur d'alignement, il faut des variables d'état supplémentaires), la modification de l'interface graphique.

Cet atelier s'est préparé au cours de plusieurs entretiens avec comme support ce prototype video. Comme le film dure en tout environ 1h30, il avait été montré en affichage accéléré. L'idée de participation à un atelier avait été proposée au cours de ces entretiens. **Séance** L'atelier a duré environ 3 heures, et a commencé par un visionage rapide du prototype vidéo. Le petit brainstorming qui a suivi a eu pour résultat une liste de sujets possibles au tableau, tournant autour de la présentation ou de la navigation, du déboguage, de la documentation, du nommage et des fonctions de l'éditeur d'alignement. Deux thèmes ont été choisis et deux groupes se sont formés : l'un autour de la question de la documentation et de la recherche d'aide, l'autre autour de la réutilisation des commandes. On remarque que, même si le thème annoncé de l'atelier était le prototypage autour de la programmation, ni les sujets issus du brainstorming ni les thèmes choisis ne sont particulièrement ou exclusivement centrés sur la programmation.

Il était mentionné que le prototype issu de la discussion serait filmé.

Les prototypes

La première présentation a fait ressortir l'importance d'une aide à plusieurs niveaux et de bons outils de navigation à partir d'un exemple simple : comment faire pour rassembler des séquences dans un groupe avant de les aligner ? Les différents niveaux d'aide ou points d'entrée suggérés étaient les suivants :

1. aide générale, sous la forme d'un petit clip ou de textes illustrés,
2. manuel utilisateur ou tutorial "de A à Z" avec toutes les commandes et leurs paramètres ; c'est le mot tutorial qui a été prononcé mais "de A à Z" fait penser à un exposé plus systématique voire ordonné de manière non nécessairement pédagogique,
3. documentation contextuelle, accessible par le menu de chaque objet, avec un sous-menu "Description" pour chaque menu et sous-menu,
4. documentation personnelle, qui apparaît toujours en premier dans les menus de description,
5. recherches contextuelle (pour l'objet courant) et générale sur mot-clés,
6. liens hypertexte dans la documentation.

Ces idées nous inspirent quelques remarques (a posteriori). L'idée de textes illustrés (idée 1) pourrait faire penser à un storyboard : ceux qui auraient servi lors de la conception pourraient être ré-utilisés de cette manière, à l'instar des design rationale de [Mør94]. L'aide personnelle (idée 4) pourrait être construite autour d'un exemple qui a bien marché (éventuellement pris dans l'historique).

Il y a eu une intervention sur l'aide en cas d'erreur, à propos de laquelle les opinions étaient partagées. Une des personnes voulait des avertissements et des explications en cas d'erreur, par exemple dans le choix des paramètres, une autre pensait que c'est très ennuyeux d'avoir tout le temps des messages d'aide en cas d'erreur comme dans Microsoft Word (cela pourrait être désactivable), un troisième participant assurait que l'utilisateur devait savoir ce qu'il faisait et que ce n'était pas au programme de lui expliquer qu'il avait fait quelque chose d'aberrant. Ceci était plutôt de l'ordre du débat d'idées que du prototypage, mais clairement il y avait ceux qui étaient en position d'utilisateurs finaux et ceux qui se comportaient en concepteurs de logiciels ! En tous cas, l'intervention venait d'une situation vécue par cette participante qui insisté un petit moment jusqu'à admettre diplomatiquement que c'est en faisant des erreurs qu'on apprend...

L'idée du deuxième exposé-prototype était celle d'un historique éditable, montré sur un scénario de recherche de motifs consensus dans des séquences alignées. Ce scénario venait de la thématique de travail d'une des participantes. L'interface de l'historique comportait des cases à cocher pour sélectionner les commandes voulues pour l'action suivante : sauver sous forme d'une macro, ou ré-exécuter, soit sur l'objet courant, soit sur un autre objet, soit sur le même objet mais sur une sélection de zone différente. Les commandes désélectionnées pouvant être indispensables à une exécution correcte, il est possible d'évaluer le script avant de l'enregistrer. En fait l'historique devient un véritable formulaire puisque quelqu'un a aussi suggéré que l'on puisse y éditer les paramètres des commandes comme par exemple le paramètre d'une recherche de motif. L'éditeur d'historique est aussi un support pour manipuler d'autres objets. C'est la question qui a été posée par l'une des participantes : est-ce qu'on perd une commande

si on ne la coche pas ? La réponse donnée a été négative, et qu'il y a de toutes façons deux choses différentes : l'historique qui reste, et les objets qu'on en extrait en les sauvant ou en les réappliquant. C'est donc un objet très interactif comme support éventuel pour la programmation. L'idée de compléter l'historique afin d'y ajouter des données a également été suggérée :

```
J'ai mes 10 séquences, j'en rajoute 2, parce qu'elles sont importantes
pour moi, et ça ne change pas le consensus, mais je veux pouvoir changer
l'historique - rajouter une étape dans la fabrication du consensus
parce que j'ai besoin de ces 12 séquences.
```

On voit que l'historique est vraiment considéré comme un outil de programmation - ici on y modifie les données traitées comme si on relançait le programme, tout en sachant que cela ne serait pas utile puisque le résultat n'en serait pas changé. Mais ce qu'on veut, c'est quelque chose de réutilisable plus tard, qui soit cohérent : un programme, en somme.

Il y avait deux niveaux qui se superposaient dans la maquette : un niveau interface utilisateur avec l'éditeur et les séquences, ou bien avec l'éditeur d'historique, et un niveau descriptif. Le niveau descriptif était un diagramme flot de données, qui indiquait avec des flèches en papier d'où provenaient telle et telle entrée, ou qui décrivait les différentes possibilités de sorties d'une étape de calcul, comme le calcul de consensus. En somme, au lieu de représenter des éléments de l'interface, les composants de la maquette représentaient des symboles. En principe, dans un atelier participatif, on évite de schématiser et de passer au niveau de la description. Mais ce style de présentation avait l'avantage d'être synthétique, et peut-être le diagramme avait-il aidé le présentateur à mémoriser l'exposé ? Peut-être aurait-il fallu proposer de faire un mini-storyboard ?

Le deuxième groupe a en réalité présenté deux idées connexes : l'historique des commandes comme on vient de le voir, mais aussi l'historique des données, ou versionnement. Le sujet, à savoir la recherche itérative d'un motif consensus dans un ensemble de séquences, s'y prêtait bien puis qu'il était important de pouvoir retrouver la version des données ayant servi au calcul à une étape donnée. Il a été très intéressant de constater ce lien dual très fort entre la notion d'historique et celle de versionnement, et indirectement de persistence : histoire des actions, histoire des données, sachant que cette question est cruciale pour les analyses effectuées par les biologistes, qui sont des manipulations de données incrémentales et exploratoires. Il y a très peu de support pour cela dans les environnements informatiques et l'implémentation - ou du moins le début d'implémentation - de cette fonction de persistence dans le prototype biok a toujours suscité des réactions très positives.

Les états successifs d'un objet sont aussi des "dates" pour l'historique :

```
E: tu peux fixer "je veux le consensus à l'étape 10 séquences" et
pouvoir dans l'historique pointer sur les commandes qui ont servi à le
faire.
S: y aller un peu à tatons ; vérifier - oui c'est bien celle (la
commande) de 9 séquences, la ré-essayer...
```

Dans l'optique histoire des données, peut-être l'historique des commandes sert-il plus de repère pour comprendre les versions successives des résultats que de base de programmation. Ce qui est aussi important, c'est de pouvoir comparer deux étapes, afin de comprendre l'effet d'une commande, par exemple pour comprendre ce qui se passe au niveau du calcul d'un consensus lors de l'ajout d'une nouvelle séquence. C'est également intéressant dans un contexte dataflow, puisque la formule d'entrée du nouvel objet peut être définie à partir d'un ancien objet et d'une commande prise dans l'historique. Ainsi, cette fonction serait réalisable assez simplement dans le prototype biok à travers un "empilement" des données intermédiaires reliées par des formules.

Une autre idée a été de distinguer par des couleurs différentes les commandes de l'historique qui relèvent de la création de l'objet de celles qui relèvent des fonctions appliquées à l'objet ou l'utilisant. En somme on distingue les constructeurs des autres types de méthode, afin de permettre une réutilisation plus simple de ces dernières. C'est sans doute une idée à reprendre pour les mécanismes de persistence : faudrait-il un support de cette distinction dans l'implémentation ?

```
F: On pourrait faire dans l'history des sous-parties - virtuelles mais
bon - qui concernent le fonctionnement des fonctions de cadre

C: qu'est-ce que tu entends par cadre ?

F: un objet si tu veux ... et une sous-partie création de l'objet

C: comment tu les isoles ?

F: les fonctions de création de l'objet seraient en bleu par exemple,
et les autres fonctions de l'objet en rouge

C: tu sépares ce qui est pertinent par rapport à l'objet du reste ?

F: tu peux le ré-adapter à un autre objet

C: tu as un exemple ?

F: appliquer le même calcul à un cadre où les bornes seront différentes

E: sélection d'une sous-partie de l'alignement ou exclure une séquence:
c'est indépendant des commandes et ça fait partie plutôt des "créations",
des choses "spécifiques" du cadre, tu ne vas pas le garder dans ta
commande.
```

Tout le monde était un peu fatigué à la fin de cet exposé-prototype, ce qui fait que des questions n'ont sans doute pas été posées (certaines l'ont été dans les notes sur la page du site wiki-web) :

- comment déclenche-t-on l'historique (évoqué avec la question de l'historicisation par objet, et question posée sur la page wiki-web, mais il était difficile d'y répondre) ?
- différence évaluer-tester et appliquer : retour arrière possible pour tester (d'autant plus que la présentation comportait une partie persistence) ?
- une représentation des interactions - comme les sélections - est-elle possible sous forme de commandes dans l'historique ? d'autant plus qu'il y a eu une référence explicite à la notion de commande de shell :

  ```
  Une fois que tout ceci a été fait - une fois toutes les commandes ont
  été appliquées, on peut obtenir un historique de toutes les commandes
  - un peu comme les commandes du shell sous forme d'un
  historique éditable avec des cases à cocher pour garder ou ne pas
  garder la commande.
  ```

  La liste des commandes était en effet explicite :

  1. introduire les sequences
  2. les aligner
  3. determiner le consensus
  4. colorier en rose
  5. colorier en vert

- représentation du contexte des commandes, surtout si ce sont des définitions visuelles qui ne sont pas "définies" ni "nommées", et qui constituent des sortes de variables automatiques ou d'objets par défaut.

Commentaires

Influence du prototype video, programmation

Le prototype vidéo visionné en début de séance et certains entretiens individuels ont clairement influencé le prototypage. Les mots "objet" et "méthode" était souvent utilisés, mais plutôt pour mentionner les objets du domaine. C'est plutôt très bon signe, signe que les termes sont acceptés, et associés à des concepts à la fois de l'interface et du domaine. A titre d'exemple, les mots sont utilisés dans le déroulement de l'exemple pour la documentation :

```
On a une aide à différents niveaux : d'abord une démo très simple
(un texte illustré ou petit film), qui montre la structure du
programme
, les objets
, comment entrer les séquences, ainsi que les
actions les plus basiques, puis une aide plus sophistiquée : un
tutorial avec des exemples pour toutes les actions possibles, de A à
Z, avec tous les objets
 déjà existants.
```

Les "objets déjà existants" sont sans doute les objets "système", c'est-à-dire probablement les classes . Ou encore :

```
Cliquer sur un des objets
, par exemple cliquer sur une
séquence, amène
une première fenêtre de menu avec d'abord les méthodes
 les plus
courantes que l'on peut faire a partir de cet objet
, puis ensuite les
autres : les méthodes
 plus complexes ou moins souvent utilisées.
```

Autre mentions :

```
...maintenant on voit la méthode
 Align avec un menu de description
des diverses choses qu'on peut faire avec cette méthode
 :
il y a une description simple et avancée. La description simple c'est
juste une phrase qui explique ce que fait la méthode
 en 2
lignes sans explication sur les paramètres.
```

On voit dans cette dernière citation que "méthode" signifie sans équivoque commande avec paramètres. Il faut donc qu'il y ait une continuité entre la notion de méthode au sens de la programmation objet et celle de commande accessible dans l'interface, d'autant plus que dans l'environnement biok, ces méthodes sont éditables.

Un autre aspect très important qui relie les fonctions de documentation avec la programmation et que le prototype vidéo a sans doute bien fait passer, est l'idée développée par ce groupe, de formulaire d'édition de la documentation d'une méthode créée par l'utilisateur : ce formulaire permettrait de remplir les cases prévues pour les différents niveaux d'aide, sans être obligatoire toutefois.

Enfin il est également possible que la notion de documentation personnelle - consistant à compléter l'aide système pour faire ressortir les points importants - ait été amenée par les fonctions de personnalisation montrées dans le prototype vidéo, notamment la réécriture de méthodes par l'utilisateur, non pas par sous-classement, mais par simple réévaluation d'un code source *local* à l'utilisateur. C'est d'ailleurs exactement ce qu'a exprimé une des participantes dans la page wiki-web :

```
Question (Catherine) : Est-ce qu'on peut modifier la documentation originale ?
        Remarque (Katja) : Si je réfléchis bien, il n'y a pas de
        différence entre documentation personelle et documentation
        originale. Par exemple il n'y a pas de différence entre les
        méthodes fournies par défaut et celles créées par
        l'utilisateur pour lesquelles lui il fera la
        documentation. Si les deux sont au même niveau, une édition de
        la documentation originale ne devrait plus poser de
        problèmes. Mais peut-être qu'il faudra garder des versions
        anciennes.
```

Le terme de *cadre* est apparu plusieurs fois, de la part de plusieurs personnes ; il était même sur un des post-it, et semblait signifier à la fois environnement, fenêtre ou objet.

Même si les sujets n'étaient pas centrés sur la programmation, on a pu noter quelques occurrences de ce thème. Par exemple, concernant les manipulations de l'historique (comme lorsqu'on fait une sélection de commandes à réappliquer), il y a eu plusieurs remarques concernant le niveau d'action et de responsabilité sous-jacent, ainsi que des idées proches de celles des environnements de programmation :

- `il faut une fonction de "test" de l'historique avant de le sauver` ;
- `le fait de désélectionner une commande de l'historique peut "bazarder" tout le reste : on est au même niveau de responsabilité que quand on programme (autrement dit, voilà un cas où décocher une case est une action de programmation)` ;
- `on peut avoir un raisonnement débogueur qui contrôle qu'on n'enlève pas n'importe quoi - le même qui disait juste avant que le programme ne pouvait pas tout contrôler...` ;
- l'idée de définition et nommage de la macro pour construire une abstraction.

On retrouve aussi cette idée, vue dans Labview, qu'il y a une continuité entre programmation et interaction aussi bien dans l'idée que les paramètres d'une méthode peuvent en être le formulaire, ou que les commandes exécutées auparavant peuvent être les champs d'un formulaire, ou encore l'idée que les commandes sélectionnées dans l'historique peuvent être un objet graphique glissé sur un autre objet.

Enfin les deux thèmes abordés dans cet atelier sont "méta" ("quelque chose sur ...") :

- la documentation c'est quelque chose *sur* le logiciel,
- l'historique est une réification des commandes *pour les manipuler en tant que telles* (pour les réutiliser, en construire une autre, les sauver, etc...) ; dans les deux cas, on "voit le stylo", pour reprendre l'exemple tiré de [HJ93].

Lien avec d'autres ateliers

On peut constater des liens avec l'atelier prototypage où il avait été question d'annotations (troisième atelier) :

- annoter l'historique,
- annoter un exemple comme documentation personnelle,
- pouvoir créer une documentation personnelle comme une annotation.

L'activité d'annotation est centrale en analyse de séquences et peut-être peut-on en tirer une métaphore à réutiliser pour la programmation ? Cela aussi a été abordé dans la page wiki-web de compte-rendu :

```
Question (Catherine) : Quelle différence y a-t-il entre l'annotation
     d'objets et l'annotation de commandes ? on peut imaginer par
```

```
exemple une annotation indiquant comment un motif a été cherché,
avec quelle méthode, quels paramètres, etc... ce qui n'est pas
très différent d'une documentation personnelle sur une méthode ?

    Remarque (Katja) : Ca introduit des niveaux différents de
    documentation (annotation). Par exemple pour une méthode
    "recherche pattern" :
       1. description générale de méthode (utilisation,
       paramètre, algo etc.)
       2. description avancée de méthode (plus de détails sur les
       paramètres, exemples)
       3. des annotations des occurences de pattern
       (spécification des parametres, particularité)
```

Organisation, méthodologie

Il y avait deux groupes, comme dans d'autres ateliers, ce qui pose un problème de temps et d'organisation, surtout s'il n'y a qu'un seul animateur. Cependant cela a permis des croisements entre les deux thèmes :

- dans le groupe documentation : pouvoir retrouver les parties de l'aide qui ont été consultées la fois précédente (notion d'historique au niveau de la documentation) ;
- dans le groupe historique de commandes : la notion de réutilisation pour ré-appliquer peut s'étendre à celle de réutilisation comme *exemple* illustrant une manière de faire marcher les fonctions ; les deux types de mémorisation sont similaires, avec notamment la possibilité d'avoir un exemple utilisable, qu'on puisse faire tourner ("documentation par l'exemple"?) - associé à l'idée de documentation personnelle "une fois que l'utilisateur a compris" ;
- ajouter un commentaire sur un historique sauvegardé, c'est de la documentation.

On retrouve ce problème, peut-être lié à la formation des participants ou des orateurs, du passage en mode "design" au lieu du mode "utilisation" avec des interactions concrètes, même simulées : dans la deuxième présentation, l'historique est sauvé sous le nom "toto.consensus" - on est dans un mode assez général. J'ai donc posé la question : "tu l'appelles comment ce fichier ?" pour concrétiser - mais les réponses étaient de l'ordre du schéma de nommage plutôt qu'un nom précis et réaliste.

En fait, les orateurs des deux exposés étaient plutôt des programmeurs (un développeur et un enseignant en programmation). Le style était donc celui de l'exposé de fonctionalités, bien que ces fonctionalités aient été déroulées - très vite pour le deuxième - pas à pas. Par exemple, le sujet de l'interaction était parfois dénommé : "l'utilisateur", "la personne" ou : "on", mais pas vraiment "je" (contrairement à la la démonstration faite par la biologiste lors du 3ème atelier qui avait été faite sur le mode "je" bien qu'il se soit explicitement agit d'une démonstration).

Exemples technologiques

L'exemple des boutons de [MCLM90], réimplémentés partiellement en Tcl/Tk (M. Beaudouin-Lafon) a été montré pour illustrer les possibilités de personnalisation. Les boutons servent à exécuter une action ; ils sont éditables, et on peut les cloner. Pour créer un nouveau bouton, il faut donc en cloner un premier et l'éditer.

Autres remarques

- La liste des participants n'est pas mentionnée dans les notes : c'est un peu gênant parfois même avec l'enregistrement video : on ne sait pas forcément qui parle.
- C'était une séance assez longue (l'après-midi, disons 3 heures) et fatigante - il y avait même des boissons et des gateaux pour une pause conséquente, ce qui explique sans soute que les exposés de prototype sont un peu accélérés vers la fin. Le visionage de la vidéo a probablement pris un peu moins d'une heure, et c'est sans doute cela qui aurait dû être évité, d'autant plus que plusieurs des personnes l'avaient déjà vu individuellement (les 2/3).
- C'est difficile d'être seul pour filmer et animer surtout s'il y a deux groupes : il faudrait peut-être proposer de nommer un scribe dans chaque groupe et faire le point avec lui sur les étapes de la discussion ?

– Il n'a pas été suffisamment dit qu'on peut interrompre la personne qui présente le prototype.

Cinquième atelier : prototypage à partir d'un prototype informatique.

14 février 2001

Ce cinquième atelier différait des autres principalement parce qu'il reposait sur l'utilisation d'un programme réel, même s'il s'agissait toujours d'un prototype. Le prototype utilisé était une première version de biok (chapitre V).

Objectifs de l'atelier

Le but général de l'atelier a été explicité au départ : se servir de l'utilisation d'un prototype logiciel comme point de départ à un prototypage de quelques fonctions et interactions particulières.

Participants

Volontairement peu nombreux pour pouvoir chacun disposer d'un terminal, ils étaient cinq. Quatre d'entre eux programment couramment (Scheme, C, Java et perl), deux enseignent la programmation à d'autres biologistes, un autre développe des applications utilisées par d'autres personnes et une autre personne écrit des scripts et des programmes de calcul scientifique (mais avec l'aide de collègues informaticiens). Une des participantes avait utilisé Tcl/Tk pour un projet (une simulation de maladie dans une population) et avait même pour cela utilisé l'environnement de construction d'interfaces VTcl (Visual Tcl) [All00].

Préparation

Comme pour l'atelier précédent, la séance avait été préparée par des entretiens individuels autour du prototype avec chacun des participants. Ces derniers avaient également, sur leur demande, reçu un certain nombre de pointeurs pour se familiariser avec le langage de programmation (Tcl).

Séance

L'atelier était divisé en deux parties : manipulations du prototype, puis discussions et prototypage autour d'idées apparues lors des manipulations. La séance se voulait ouverte, mais s'appuyait sur des exercices destinés à se familiariser avec le prototype qu'aucune des personnes n'avait eu l'occasion de manipuler avant l'atelier, sauf une qui avait "testé" la page d'instructions, et connaissait déjà le langage. Les instructions étaient détaillées, afin de ne pas trop mettre les participants en situation "pédagogique". Leur fonction était seulement de faire pratiquer quelques manipulations avec l'environnement. Toutes les actions suggérées n'ont pas été réalisées mais le but était plutôt de susciter les réactions et les idées pour une table ronde prévue dans la deuxième partie de l'atelier. Le passage à cette partie a été un peu difficile à cause des difficultés rencontrées dans l'utilisation du prototype logiciel, mais une fois démarré la discussion a été très riche et s'est même prolongée après l'atelier.

Commentaires

Utilisabilité

Il ressort en premier lieu qu'une courte séance préalable de tests d'utilisabilité ("think aloud", par paire) aurait été très utile. Cela aurait évité à certains participants de se décourager à chercher pourquoi telle action ne marchait pas et de se trouver en situation d'échec. La difficulté relevait à la fois de la compréhension générale du modèle sous-jacent et de problèmes plus pratiques de navigation, d'identification des élements de l'interface et des liens entre eux, comme pour la perception des zones d'interaction - notamment celles prévues pour l'entrée de code.

Objectif de l'atelier et du prototype

Il aurait sans doute aussi fallu insister plus sur le rôle de l'atelier et le statut du prototype : l'atelier a été perçu par certains comme une séance de test du logiciel lui-même ce qui n'était évidemment pas le but, le prototype étant en phase très préliminaire. Il a aussi été compris par d'autres comme un cours - un cours de programmation, ce qu'il n'était pas non plus. Il aurait sans doute fallu insister sur le fait que les "instructions" de la page Web n'étaient pas un défi ou un exercice à réaliser et qu'on pouvait poser toutes les questions qu'on voulait. Cela a été dit au début de l'atelier, mais peut-être la programmation suscite-t-elle parfois ce type de motivations - assez ludique somme toutes !

Participation des biologistes

La première partie a été un peu frustrante - on n'arrivait pas toujours à faire une chose bien et complètement, et même s'il n'y avait aucune contrainte pour finir ce qui était décrit dans la page Web, c'était difficile pour les participants d'avoir des idées de prototypage pendant ce temps alors qu'ils essayaient de résoudre des problèmes de sélection ou de navigation et d'appréhender en même temps un environnement assez complexe. Il aurait peut-être fallu aussi se limiter à une tâche plus simple (comme l'a suggéré une personne) qui fasse intervenir un cas intéressant de programmation pour des biologistes.

Néanmoins, même s'il n'y a pas eu beaucoup de prototypage papier - sauf pour une fonction de sélection multiple de zones dans l'alignement qui a été montrée en utilisant des transparents sur le rétroprojecteur, la discussion a duré plus d'une heure et plusieurs sujets ont été abordés, concernant par exemple :

- La disposition des différents composants du programme : dans plusieurs fenêtres, dans des fenêtres à panneaux, dans des boîtes refermables (comme c'est fait pour les attributs graphiques dans Vtcl d'après l'expérience d'une des participantes). à cette occasion, il a été utile de montrer les mécanismes de rangement implémentés dans le prototype.
- La disposition des données, comme de pouvoir partitionner l'alignement en plusieurs tableaux.
- La simplification des zones de programmation : il y en avait trop qui n'étaient pas bien identifiables, ce qui rendait encore plus difficile la compréhension du modèle de fonctionnement des évaluations de code dans l'environnement. Il a été suggéré par exemple de placer "l'interpréteur" non pas dans la console, mais dans la boîte principale ouverte par l'application, ou encore de mieux identifier l'objet d'appartenance de tel ou tel composant de programme et de mieux les nommer. L'idée de programmabilité *généralisée* a en fait été réellement remise en question.

Programmation

Il nous a semblé que l'aspect objet de l'architecture et les notions de méthodes n'ont pas posé de problèmes particuliers à l'utilisation. Une des fonctions de visualisation du prototype repose sur un mécanisme de tag, qui est un étiquettage graphique associé à des valeurs qu'on relie à des positions dans l'alignement de séquences. La création d'un nouveau tag a posé plus de questions : les participants ayant atteints cette étape ont eu quelques difficultés concernant les interactions nécessaires pour créer la classe et les méthodes à redéfinir : trouver la classe modèle, les indications concernant les différences classe/instance, ou même comprendre plus globalement le schéma général de fonctionnement des tags. Ces difficultés sont donc plus liées à l'application qu'aux mécanismes de programmation objet (qui induit il est vrai souvent des interactions complexes entre objets).

Le prototype permettait la modification des attributs graphiques du tag en passant par du code. Cela a été critiqué à juste titre, et a donné lieu à un développement spécifique par la suite.

Un cas type pour la programmation par démonstration a été rencontré : un des participants a en effet proposé ce type de fonction, en donnant des idées d'interaction pour la réaliser. Une boîte de dialogue permettrait de modifier l'attribut d'une cellule selon son contenu, puis un autre moyen permettrait de recopier cette modification à d'autres cellules *de même contenu* . L'exemple est donc ici le "contenu" de la cellule, ce qui n'est pas toujours simple à déduire. Cela peut être le contenu littéral, ou bien une propriété, comme l'appartenance à groupe physico-chimique commun.

Discussions et autres activités de prototypage à la suite de l'atelier

La page wiki-web contient quelques remarques d'une des participantes sur la difficulté de certaines actions attendues ou bien, plus générales, sur les environnements programmables :

```
J'ai l'impression que pour commencer à programmer dans un tel
environnement sans connaître beaucoup du langage de programmation il
est indispensable de bien connaître l'architecture et la structure de
l'environnement.
```

Un mini-atelier a eu lieu deux semaines plus tard pour affiner le prototypage autour de la fonction centrale de l'application, la visualisation de propriétés avec le mécanisme des tags. C'est principalement sur cette partie que les participants avaient achoppé, et nous en avons conclu qu'il était impossible de laisser les définitions pour faire marcher ce mécanisme à un niveau de programmation de code.

Cet atelier, pour lequel un compte-rendu a également été fait, reposait, tout comme le premier atelier trois ans plus tôt, sur une maquette préparée à l'avance. Il s'agissait d'un storyboard dont les cases étaient des feuilles A3 avec des étiquettes numérotées indiquant les séquences d'actions et d'états. Mais chaque étape, au lieu d'être un dessin, était une maquette papier rééditable. La figure 23 montre une des pages de ce storyboard-maquette, où l'on voit comment définir un tag non plus à partir d'une liste de valeurs, mais à partir d'une formule.

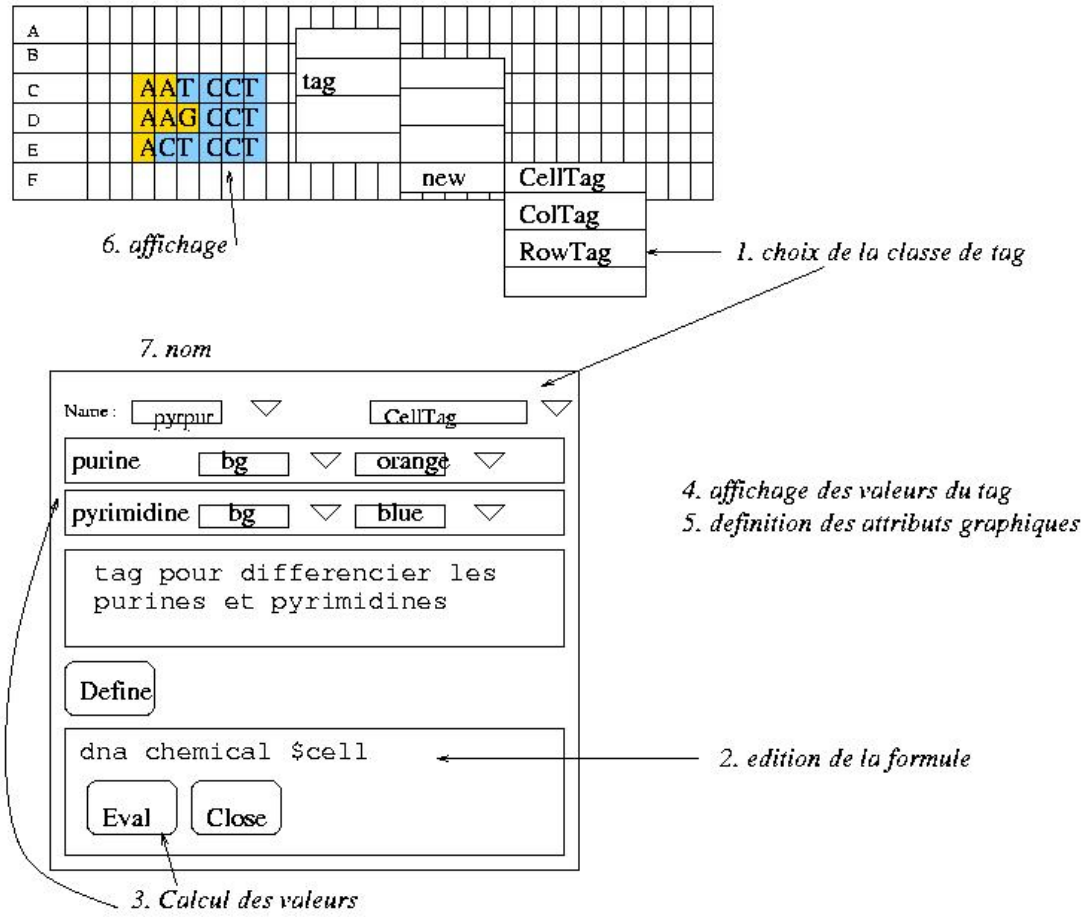


FIG. 2.23 – Définition d'un tag avec une formule

Sixième atelier : prototypage pour la résolution de problèmes.

19 juin 2001

Cet atelier a été organisé à l'occasion de deux stages d'étudiants, l'une biologiste et l'autre bio-informaticien, qui travaillaient sur un algorithme de prédiction de segments transmembranaires dans des protéines (ces stages sont décrits à la fin du chapitre V).

Objectifs de l'atelier

L'une des personnes avait implémenté dans l'environnement biok l'algorithme publié dans la littérature [Hei92], en y ajoutant une interface intégrée dans les mécanismes de visualisation fournis par l'outil (chapitre V). L'autre étudiant avait pour objectif de concevoir et d'implémenter un protocole d'interaction avec ce même algorithme, qui permette d'incorporer l'interaction et les informations venant de l'utilisateur dans l'heuristique de ce dernier (mais sans toucher à son code). Il fallait donc explorer ces interactions.

Participants

En plus de ces deux étudiants, trois autres personnes ont participé à cet atelier, dont deux bio-informaticiennes et moi-même. L'une des deux bio-informaticiennes est spécialisée dans l'étude des structures des protéines et nous a beaucoup aidés pour situer le problème dans une perspective scientifique concrète, l'autre était biochimiste de formation, et informaticienne de profession (elle travaille dans le centre informatique).

Préparation

Il fallait essentiellement préparer des données, c'est-à-dire des séquences de protéines transmembranaires pour lesquelles le type d'interaction que nous voulions explorer était justifié. Ce travail de préparation a été réalisé par l'étudiant et lui a pris une semaine de recherches bibliographiques. Il s'est avéré que cette préparation était indispensable : sans scénario, aucune interaction, surtout en présence de vrais biologistes, n'aurait pu commencer.

L'autre partie de la préparation a consisté à construire des parties de maquette avec l'étudiante biologiste représentant l'état actuel de l'interface, sur des exemples réels de résultats d'affichage (des segments de séquences colorés de différentes couleurs selon qu'il s'agissait de segments transmembranaires certains ou putatifs ou de boucles intra- ou extra-cellulaires).

Séance

L'atelier a commencé par une présentation des données et une maquette illustrant l'affichage actuel. Ensuite, nous avons commencé une simulation de diverses interactions que nous avions imaginées pouvoir être intéressantes pour l'utilisateur : modifications de la topologie générale de la protéine, changement de la taille des segments, ... en utilisant des post-its ou des bandes de papier collants pour faire des sélections. Chaque topologie était matérialisée par une feuille différente et collée sur une grande feuille de papier. Pendant ces simulations, une discussion s'est engagée avec la biologiste spécialiste des structures qui connaissait bien le problème. Nous avons ensuite, grâce à un exemple approprié, rapidement convergé vers une situation plus focalisée sur l'indication par l'utilisateur de position de début de segments que l'algorithme n'avait pas détecté. C'est cet aspect, décrit dans le chapitre V, qui a pu être implémenté par l'étudiant dans la suite de son stage.

2.3.3.4 Remarques générales sur les ateliers.

Méthodologie

- L'organisation d'un atelier par un seul intervenant est difficile : il faut en même temps filmer (correctement) les interactions avec le prototype et écouter ce qui se dit pour pouvoir réagir. Du coup, certaines questions n'ont pu être posées que dans la rédaction du compte-rendu, ce qui montre l'intérêt d'un site Web collaboratif de type wiki. Ce site n'a d'ailleurs pas toujours fonctionné de manière très dynamique : les remarques aux questions posées dans ces notes ont été faites par un petit nombre des participants - peut-être aurait-il fallu faire une petite démonstration wiki pendant l'atelier ?
- La réticence à maquetter diminue nettement lorsqu'il est prévu de filmer le prototype.
- L'utilisation d'exemples technologiques donne parfois l'impression qu'il s'agit d'un cours (de programmation d'interfaces). Cependant, il n'est pas inutile de montrer ce qu'il est techniquement possible de faire, car cela peut susciter des idées.
- En combien de groupes faut-il répartir les participants ? Dans les ateliers que nous avons organisés, il y en avait un ou deux. Si cela dépend d'abord du nombre d'animateurs, il semble cependant qu'il soit intéressant d'avoir plusieurs groupes afin de pouvoir bénéficier des interconnexions qu'il y a entre les thèmes de chaque groupe. Cela permet également de répartir les personnalités leader qui ont une forte influence sur le travail du groupe.
- Les orateurs pour les exposés sont toujours des hommes, à une exception près, alors qu'il y a toujours au moins un tiers de participantes.

La conception participative et la programmation par l'utilisateur : enseignements apportés par les ateliers.

Deux conclusions semblent se dégager de l'utilisation de la conception participative dans un contexte de développement d'un environnement programmable.

Il y a d'abord sans doute une particularité dans la participation aux ateliers de personnes qui par ailleurs savent programmer : ces personnes ont des compétences en conception et peut-être ont-elles déjà abandonné la vision "utilisateur". Non pas que ces personnes ne connaissent pas les besoins des utilisateurs, bien au contraire, mais plutôt, elles en rendent compte sous la forme d'une spécification générique, soit assez conventionnelle - avec un respect des conventions et des pratiques de design des applications courantes, soit assez normatives, ce qui peut arriver lorsque c'est quelqu'un qui enseigne la programmation. Ceci étant dit, il y a une certaine diversité parmi les biologistes "programmeurs" : des éternels recommençants aux professionnels. Ceux auxquels s'adressent ces ateliers sont plutôt entre les deux. Mais une autre particularité vient du côté des débutants ou occasionnels, qui risquent de ressentir une situation d'échec plus vivement peut-être qu'avec d'autres types d'applications.

La conception participative fait reculer la programmation. En théorie, la programmation n'étant pas un but en soi sauf si l'environnement est aussi considéré comme pédagogique, la conception participative sert à trouver une conception qui soit "la bonne" : construire les bonnes fonctions, mais aussi éliminer les fonctions inutiles. En principe, dans un logiciel idéalement bien conçu, la programmation ne devrait donc plus être nécessaire. C'est la "programmabilité" comme une fonction marketing qui est ainsi évitée, puisqu'elle n'est pas un but en soi, et puisque l'espace de conception est complètement recouvert par l'interface utilisateur. S'il reste de la programmation, c'est d'abord par principe, puisque la flexibilité maximale n'est réalisable que par ce moyen. C'est aussi parce que les ateliers permettent d'identifier les zones de variabilité, composée d'une zone générale et d'une zone variable pour laquelle la programmation doit être accessible : c'est clairement de cas des mécanismes de visualisation. Ainsi, le prototype fournit les principales fonctions de visualisation, mais fournit au même niveau la possibilité d'en créer de nouvelles, car on ne peut pas supposer les avoir toutes imaginées d'avance. C'est enfin aussi parce que les ateliers permettent d'identifier des niveaux de manipulation d'ordre symbolique, comme on l'a vue pour les "langages" autour des motifs.

On voit d'autre part qu'on ne distingue plus ici, comme on l'avait fait sur un plan *théorique* dans la discussion du chapitre I, ce qui est programmation de ce qui ne l'est pas. Au contraire, les *pratiques* sont bien repérées : il faut remarquer dans les ateliers ce qui relève de la programmation afin d'en tirer le plus d'informations possibles pour définir et concevoir un environnement programmable, c'est-à-dire comportant des sorties de secours en cas de défaut dans la conception. L'avoir prévu aussi bien par des ateliers participatifs que par une programmabilité pourrait faire toute la différence.

Intérêt suscité par la démarche participative en biologie.

Un peu de publicité pour la démarche participative ayant été faite à l'occasion de présentations publiques, et un article ayant été publié dans une revue interne du service d'informatique scientifique [Let99a], il n'est pas rare que cette approche suscite une certaine curiosité de la part de collègues bio-informaticiens, qui en voient l'application pour leur propre travail. Ainsi, un atelier de prototypage d'interface utilisateur a été organisé récemment dans un laboratoire du campus pour une application spécifique - autour des "Bacterial Interspersed Mosaic Elements" (BIME) [BCH99], avec enregistrement d'une vidéo. D'autres personnes ont montré leur intérêt pour ces méthodes, notamment dans des groupes de développement de composants pour la bio-informatique (bioperl, biopython).

L'idée de l'atelier organisé avec le bio-informaticien du campus était venue à l'occasion d'une question qu'il m'avait posée concernant les différents outils et méthodes de spécification de logiciels. En effet, ce collègue était souvent confronté à des utilisateurs indécis quant à ces spécifications, et il cherchait un moyen, une méthode, pour parvenir à discuter avec eux et trouver un accord. Je lui ai donc proposé l'idée d'atelier participatif, après lui avoir montré quelques vidéos et des maquettes en papier, et après avoir préparé un ou deux scénarios avec lui.

Il semble que les biologistes ou les bio-informaticiens biologistes d'origine sont beaucoup plus souvent intéressés par la démarche participative que les informaticiens professionnels (cela se vérifie sans exception dans notre cas). Il est difficile de donner une explication objective à ce phénomème : est-ce l'intérêt pour les approches pratiques, expérimentales ? Est-ce parce qu'ils comprennent plus facilement l'utilité des ateliers *pour eux-mêmes* , en tant qu'utilisateurs des logiciels ? Peut-être les informaticiens pensent-ils que seules les méthodes formelles sont valables ? Peut-être craignent-ils d'être ensuite astreints à des spécifications trop nombreuses ou d'être limités dans leur créativité (c'est l'argument que l'un d'eux a avancé après avoir lu [Gre93]) ?

2.4 Conclusion.

Notre propos n'est pas de démontrer que la démarche participative est une panacée, mais qu'elle est complémentaire, en tant que source d'informations différentes, des méthodes "objectives" et généralistes. Il nous semble en effet que, tout particulièrement pour un domaine se donnant comme enjeu de mettre dans les mains de non-techniciens l'un des outils les plus techniques de l'informatique, la programmation, il est crucial non seulement de confronter les explorations techniques à leurs utilisateurs potentiels, ce que font déjà de nombreuses études, mais aussi de travailler avec ces utilisateurs pour comprendre *leur* façon d'appréhender ce que les informaticiens définissent comme étant la programmation, c'est-à-dire de répondre, mais pas à leur place, à ce type de questions :

- à quel(s) *niveau(x)* d'une application va-t-on situer les approches par inférence, par programmation de code, par diagrammes visuels ?
- *Sur quels objets* doivent porter les fonctions de ces différents niveaux ? A quels objets ne faut-il surtout pas donner accès par programmation ?
- Quel est le *mode* d'accès à tel niveau de programmation : échec du logiciel, apprentissage incrémental et contextualisé, exploration, ... ?
- Etc...

Ces questions sont en réalité assez simples ! Mais supposer qu'on en connaît les réponses d'avance est à notre avis une erreur de point de vue, par simple hypothèse qu'un informaticien et un non-informaticien ne peuvent pas avoir les mêmes priorités, les mêmes difficultés, les mêmes objets d'intérêts.

De plus ces approches s'apprennent tout de même assez bien. On notera certes qu'elle s'apprennent plutôt sur le tas que dans les livres, dont les conseils sont utiles, mais a posteriori, une fois qu'on a fait sa propre expérience. C'est le sens de la réflexion dans l'action de [Sch90] qui s'applique également aux animateurs d'ateliers. Il est certain que la participation de professionnels de l'animation de groupe, de la conception graphique, de l'ethno-méthodologie ou de la vidéo peut constituer une aide significative, et [Nar97b] suggère utilement de faire appel à ces spécialistes fussent-ils étudiants, mais il est déjà très utile d'organiser des ateliers sans ces moyens-là.

Nous voudrions également dire que l'approche participative est plutôt sympathique. Si la participation à un atelier est certainement fatigante, aussi bien pour les animateurs que pour les participants, c'est que la concentration, l'implication des personnes est en général assez forte : il est difficile de s'endormir au milieu d'une construction de maquette ou d'une prise vidéo car tous les participants ont quelque chose à faire !

Enfin, le matériau rassemblé pendant ces ateliers étant important, le temps pris aux participants est bien rentabilisé. De plus, ayant participé aux ateliers, ils sont plus à même d'utiliser l'application finale et peuvent aider d'autres personnes à l'utiliser.

Chapitre 3

Techniques pour la programmabilité.

3.1 Introduction

Ce chapitre a pour objectif de définir "techniquement" l'espace de variation logicielle dans lequel nous pouvons situer la programmation par l'utilisateur. Un certain nombre de techniques ont été explorées pendant la thèse, ainsi que des principes de conception. Ils ont été annoncés dans le premier chapitre et nous voulons ici en présenter une synthèse plus détaillée orientée en fonction de l'objectif que nous poursuivons.

La première partie concerne les approches générales et les concepts concernant la flexibilité logicielle et nous discutons le cas échéant leur utilité par rapport à notre démarche ainsi que les applications que nous avons pu en faire. La seconde partie s'attache à décrire les conditions dans lesquelles ces idées et ces principes peuvent s'adapter à la programmabilité pour l'utilisateur non professionnel et explore différentes modalités d'accès à l'implémentation par l'interface utilisateur.

3.2 Flexibilité et systèmes modifiables

Dans cette partie, nous essayons d'abord de passer en revue quelques notions générales sur la flexibilité logicielle puis nous verrons quelles sont les propriétés d'un système flexible pour ses utilisateurs. C'est ce qui nous amènera à envisager l'idée de conception réflexive qui étend la flexibilité d'un système à la possibilité de sa propre modification.

3.2.1 Flexibilité logicielle

C'est le propre des techniques de réutilisation d'avoir pour objectif une factorisation des éléments selon un degré plus ou moins grand de généricité et de rendre également ces derniers hautement paramétrables. Un composant générique qui n'aurait qu'un seul fonctionnement possible perdrait de son potentiel de réutilisabilité et d'application à des cas variés. Alors que les ouvrages de génie logiciel fournissent des conseils sur l'aspect générique de ces composants, c'est plutôt leur paramétrabilité qui nous intéresse ici. Les termes sont cependant souvent interchangeables dans la mesure où la généricité se traduit par un polymorphisme paramétrique.

On peut distinguer deux grands types de flexibilité - d'un coté ce qui *permet* la flexibilité et de l'autre ce qui la *caractérise* :

- la flexibilité interne : cele porte sur la manière dont est construit le logiciel ou le composant ; c'est ce qui rend l'architecture d'un logiciel maintenable, modifiable, lisible ; les types de solutions sont la modularité, l'utilisation de patrons de conception [GHJV95] ;

- la flexibilité externe : c'est la flexibilité d'utilisation, la programmabilité ; le système est paramétrable, mais aussi adaptable à un nouvel usage ; cela porte sur la manière dont est conçue l'interface du composant.

Il est important de distinguer ces différents types de flexibilité (même si le deuxième n'est pas facile à réaliser sans le premier) : ce ne sont pas les mêmes critères de qualité, les mêmes utilisateurs ou contextes, ni la même partie du processus de développement.

Ainsi [GHJV95] juge que la flexibilité et la réutilisation se conçoivent de manière différente selon qu'on doit construire :

- une application : il s'agit dans ce cas de réutilisation interne ("code reuse"), et le but est qu'il n'y ait pas trop de dépendances entre les classes ;
- une toolkit (flexibilité externe) ;
- un framework : il s'agit là plutôt de réutiliser une conception, qui se caractérise en général à la fois par une flexibilité interne ainsi qu'une flexibilité externe maximisée sous la forme d'une bonne généricité et d'une bonne paramétrabilité.

3.2.1.1 Flexibilité relative au type de langage

Vus sous l'angle de la paramétrabilité plutôt que sous celui de la réutilisabilité, en quoi le paradigme de programmation lui-même influe-t-il sur la flexibilité ?

- dans certains langages fonctionnels, comme Lisp, les fonctions sont des valeurs calculables et peuvent être des paramètres ;
- dans le modèle par objets, avec l'héritage, on peut considérer que ce sont les méthodes qu'on fait varier ; tant que la granularité est au niveau de la méthode, on peut donc dire que les méthodes sont des paramètres, soit statiques, soit dynamiques selon l'environnement, dans la suite de l'idée de fonction citoyen de 1ère classe dans les langages fonctionnels, mais là, le principe se généralise ; avec la composition, ce sont les objets eux-mêmes qu'on fait varier ; les classes peuvent aussi être un paramètre.

D'après [Weg89], indépendamment du paradigme de langage, la flexibilité conceptuelle est très encouragée par les langages sans typage fort et surtout sans abstraction de données (accès aux variables des objets uniquement par des opérations). Une anecdote concernant une bio-informaticienne utilisant notre prototype biok confirme cette hypothèse : elle a pu imaginer de construire une courbe qui était la soustraction de deux autres grâce à la possibilité d'inspecter les variables des objets courbes alors que l'accesseur pour ces données n'avait pas été prévu.

Il ne s'agit pas non plus de rejeter tout mécanisme d'abstraction, au contraire. L'abstraction est bien sûr un outil de flexibilité conceptuelle. D'une part l'abstraction de données ne recouvre pas tous les mécanismes d'abstraction, il y a aussi par exemple l'abstraction comme regroupement de plusieurs entités sous une seule de plus haut niveau, notamment la définition de procédure ou de fonction regroupant plusieurs instructions. Un des aspects très utile du langage XOtcl [NZ00b], qui est une extension objet de Tcl [Ous98] que nous utilisons pour l'implémentation de notre prototype biok, est de permettre la redéfinition isolée d'une méthode, cette redéfinition devenant active immédiatement pour toutes les instances. Ce type de mécanisme, tout comme l'héritage en général d'ailleurs, favorise l'utilisation de l'abstraction "méthode" et évite le défaut de viscosité relevé par [Gre96] qui obligerait par exemple à ré-instancier toutes les instances pour que la modification soit prise en compte (viscosité de répétition). L'abstraction "classe" elle-même (construire un modèle pour des objets identiques) doit pouvoir aussi être utilisée à bon escient, et non obligatoirement comme dans de nombreux langages objets où il faut définir une classe pour construire un objet, même unique. En XOtcl, il est possible de déclarer des objets sans classe. En réalité, ils sont alors déclarés dans la classe Object, fournie par le langage. Ce mécanisme est utilisé dans prototype biok pour les objets à instance unique, comme le fabriquant d'éditeurs editors, l'objet responsable de la gestion des points d'arrêts et des demandes de trace debug, le serveur de noms names, le menu général biok, les serveurs d'informations biologiques : protein, codegen, etc...
Il permet d'élargir l'outillage de modélisation en ne considérant les classes que pour ce qu'elles sont : un mécanisme de fabrication d'objets identiques qui ne devrait être utilisé que dans ce cas - ce qui rend

inutile le patron de conception "singleton". Dans sa présentation des langages à objets, c'est ainsi que [Weg89] présente les choses : il y a d'abord les objets, vient ensuite l'idée de classe, comme un moule pour des objets identiques, puis vient l'idée d'héritage, pour factoriser des propriétés communes.

Un autre mécanisme de niveau objet existant dans XOtcl est le "mixin" : un mixin est une sorte de super-classe qui s'insère dans la hiérarchie des super-classes d'un objet, de manière dynamique comme un filtre. Cependant, dans ce langage, le mixin est définissable au niveau d'un objet, ce qui permet de faire varier le comportement d'un objet particulier sans perturber les autres. C'est cet outil qui est utilisé dans le prototype biok, par exemple pour définir les dépendances de données entre les objets, pour implémenter le dataflow, ainsi qu'un protocole pour explorer les variantes d'un algorithme de prédiction biologique.

Il aurait également été possible d'utiliser le mécanisme de délégation au lieu du mécanisme d'héritage, ce qui aurait permis d'assouplir l'implémentation de certains services, en les déléguant à d'autres classes, et ce à la demande. C'est alors le problème *inverse* du problème classique de communication de self [Lie87] qui se pose : c'est le self du délégué qu'on voudrait pouvoir utiliser (la délégation à proprement parler n'est pas prévue dans XOtcl, mais on peut implémenter une fonction équivalente avec des filtres et des mixins).

3.2.1.2 Flexibilité interne

Patrons de conception

Les patrons de conception sont d'abord destinés à la flexibilité interne puisqu'en eux-mêmes ils ne constituent pas des composants réutilisables - ce sont seulement des schémas de conception. Mais ils sont une base riche et solide pour construire des composants flexibles à l'utilisation. [GHJV95] explique ainsi que la clé pour maximiser la réutilisation est de prévoir les nouveaux besoins et les changements dans les besoins déjà définis, et que l'intérêt des patrons de conception est justement de permettre des variations importantes, mais localisées, c'est-à-dire n'amenant pas de changements dans toute l'application ou encore *variant de manière indépendante* . Qu'il y ait en effet flexibilité et variations semble une idée raisonnable, toute la question revient à chercher comment ne pas varier n'importe comment. En somme, c'est l'idée qu'il faut *réduire les dépendances* parmi lesquelles [GHJV95] cite par exemple :

- La dépendance entre la création d'un objet et le nom de sa classe : il est conseillé d'utiliser plutôt la fabrique abstraite ou le prototype.
- La dépendance de l'application envers a) des opérations spécifiques (requêtes explicites) : les patrons "chaîne de responsabilité" ou "commande" permettent de faire varier ces opérations ; b) un algorithme : l'itérateur, la stratégie, la template méthode ou le visiteur rendent cet élément variable.
- La dépendance par rapport à une implémentation, à une représentation de l'objet.
- Les dépendances entre classes, qu'on trouve souvent dans les systèmes monolithiques : le "layering" et le couplage abstrait constituent des solutions plus flexibles (fabrique abstraite, pont, chaîne de responsabilités, commande, façade, médiateur, observateur).
- Les dépendances entre les méthodes de classes apparentées - il faut éviter d'étendre les fonctions par sous-classement : les méthodes de création ont souvent une initialisation ou une finalisation ; il vaut mieux utiliser la composition et la délégation (pont, chaîne de responsabilité, composite, décorateur, observateur, stratégie).

En résumé, les aspects qui peuvent varier se caractérisent ainsi - et on voit que c'est bien plus que du paramétrage :

- création des objets : objets composites, classe ou sous-classe à instantier, familles d'objets, objet singleton,
- accès aux objets : interface, responsabilités, sous-système,
- structure des objets : composition, implémentation, mémorisation,
- interactions et dépendances entre objets,
- algorithme, étapes d'un algorithme, opérations à appliquer.

3.2.1.3 Flexibilité externe

La flexibilité externe concerne la souplesse d'utilisation. [NT95] caractérise ainsi les opérations de composition logicielle à partir d'un découpage de ce qui est fixe et de ce qui varie, par degrés de flexibilité croissante (figure 1) :

- *paramétrisation fonctionnelle* : tous les paramètres doivent avoir été prévus (sélection) ; on peut apporter quelques nuances : le nombre de paramètres est-il variable ? les paramètres peuvent-ils être des fonctions ? c'est une flexibilité plus compositionnelle, mais cela doit quand même avoir été prévu et rentrer dans un schéma ;
- *composition logicielle* : la structure est un paramètre ; cela n'est pas nécessairement difficile d'utilisation : le scripting ou la programmation par démonstration rentrent dans cette catégorie ; si c'est un framework, c'est plus complexe ; la vraie limite est dans la flexibilité des composants, c'est-à-dire leur paramétrabilité et leur intégrabilité ; dans le cas de la programmation par démonstration, c'est la puissance d'expression du langage : peut-on tout décrire ? quelle est la granularité ?
- *programmation* : définie à partir du moment où le langage est général.

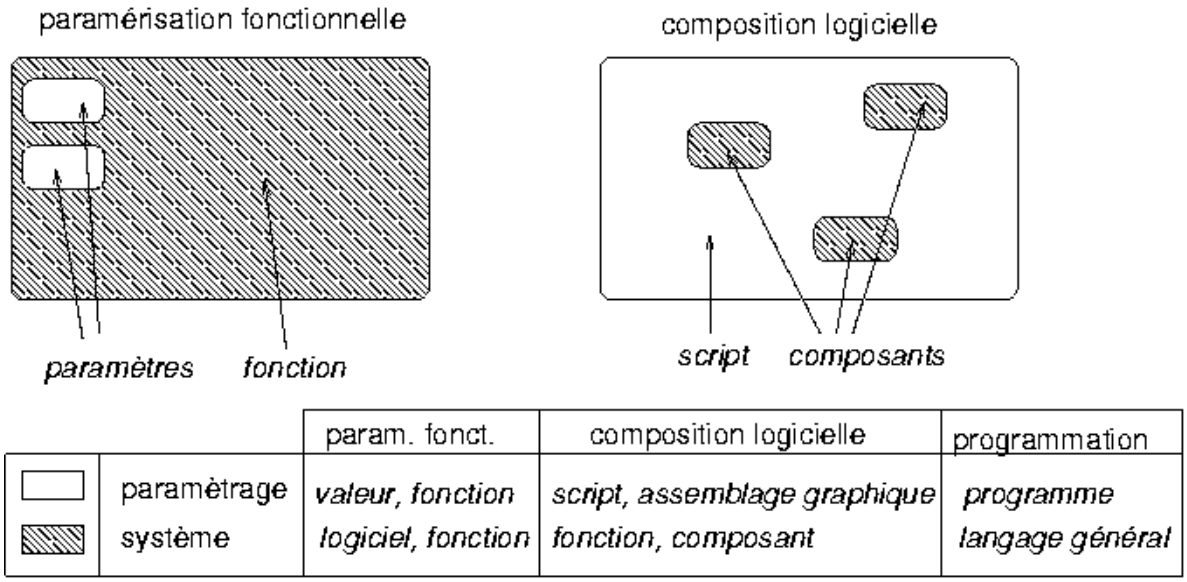


FIG. 3.1 – Paramétrisation.

Un framework se classerait par exemple à la fois dans la première catégorie, puisque la structure globale est donnée et n'a pas à être déterminée par l'utilisateur, mais aussi dans la catégorie suivante, puisqu'en sous-classant l'utilisateur doit structurer, ainsi que dans la troisième du fait que la programmation des sous-classes se fait dans le langage d'implémentation.

La nature des paramètres a aussi de l'importance. Dans le cadre de la programmation déclarative, par exemple, le caractère *programmatique ou non* de la paramétrabilité apporte une plus ou moins grande flexibilité (de la même manière que la possibilité de manipuler des valeurs fonctionnelles ou de manipuler dynamiquement des classes et des objets). [KLL+97] décrit 4 degrés de paramétrabilité dans l'utilisation d'un module de manipulation d'ensembles, où le paramétrage donne des indications au module pour choisir une stratégie d'implémentation des ensembles :

1. choix de la stratégie adaptatif : l'utilisateur ne spécifie *aucun paramètre* ;
2. un paramètre déclaratif à la création d'un ensemble donne un profil d'usage de chaque fonction du module et une indication sur la taille de l'ensemble ; le choix est donc *implicite* , et l'utilisateur ne contrôle pas ce choix ;
3. choix *explicite* d'une stratégie d'implémentation de l'ensemble (LinkedList, HashTable, BTree, ...) ; suppose une connaissance par l'utilisateur des différentes stratégies, mais le contrôle est plus précis ;

4. le client peut donner comme stratégie explicite *sa propre implémentation* (un nom de classe) ; cela est utile pour les cas où on sait d'avance qu'il est impossible de prévoir tous les cas possibles.

Ces distinctions sont à rapprocher aussi de la distinction que fait [Dou96b] sur la flexibilité comme sélection ou comme création, car dans la spécification déclarative, on ne fait que choisir parmi des possibilités fixes. C'est d'ailleurs, selon [Mør97c], ce qui distingue la programmation par l'utilisateur final de la personnalisation de logiciel : les macros et scripts sont des langages d'intégration, et non d'implémentation, et à ce titre, permettent seulement de sélectionner des actions ou des objets parmi des choix prédéfinis. à l'inverse, [Nic96] considère plutôt comme une force de certains environnements visuels (Visual Basic ici) de fonctionner par sélection : toutes les opérations disponibles sont accessibles dans l'environnement et cela facilite la programmation par l'utilisateur non-spécialiste.

3.2.2 Flexibilité pour l'utilisateur

L'aisance de construction apportée par la flexibilité interne, ainsi que l'aisance et la puissance d'utilisation dûes à la flexibilité externe ne sont pas des conditions suffisantes pour qu'un système soit flexible pour son utilisateur non informaticien. S'il s'agit d'un logiciel à paramétrisation fonctionnelle, c'est la puissance d'expression qui n'est pas suffisante, s'il s'agit d'une bibliothèque, le choix d'un bon niveau de granularité est crucial. Une boîte à outils n'est pas simple à utiliser, et un framework l'est encore moins ; il arrive cependant que des biologistes peu expérimentés en programmation utilisent des modules spécialisés dans leurs scripts comme par exemple ceux de bioperl [CFD+97], un ensemble de modules d'analyse de séquences et d'accès aux banques de données. Mais l'écriture de scripts ne suffit pas à toutes les tâches d'analyse biologiques : il faut aussi pouvoir visualiser des données et implémenter des méthodes. Or, la construction des outils de visualisation et d'interaction qui sont nécessaires à la réflexion scientifique assistée par ordinateur ne sont pas du domaine de compétences - ni d'intérêt - des biologistes.

3.2.2.1 Choix d'une architecture.

Au regard de ces contraintes et des définitions données plus haut, notre choix d'architecture s'établit autour de deux questions principales :

1. Quelle architecture est-elle préférable pour notre approche : une bibliothèque de fonctions, des composants graphiques adaptés au domaine (schéma *"you have to call me"*) ou bien une hiérarchie de classes génériques (framework) (schéma dit de type "Hollywood" : *"we call you"*) ?
2. Doit-on partir d'un ensemble générique (framework) qui nécessite un travail préalable de spécialisation avant de pouvoir être utilisé ou d'un système *qui marche déjà* , c'est-à-dire une application déjà spécialisée, qu'on peut reprogrammer ou compléter ?

Le schéma de type framework est à conserver comme principe d'architecture interne pour notre système. Mais il ne constitue pas une solution suffisante pour des raisons de difficultés évidentes d'utilisation pour des non-spécialistes. Ces difficultés sont par exemple, si on reprend le cadre des dimensions cognitives de [Gre96], dûes aux dépendances cachées et au niveau d'abstraction des composants. Un framework est ainsi moins facile à utiliser qu'une bibliothèque où chaque composant est en principe plutôt conçu pour fonctionner indépendamment.

La figure 2 illustre ce recoupement de techniques qui nous semble convenir :

- existence de composants réutilisables,
- framework mais *instancié* , déjà spécialisé,
- pas de limitation sur l'accès au code (pas de zones hachurées),

[Mør97b] présente ainsi la tailorabilité comme une manière de rendre la réutilisation accessible aux non-professionnels - les types de réutilisation adaptées à la personnalisation étant les bibliothèques, l'héritage, la composition et les générateurs. [Mør97c] explique aussi que les questions de personnalisation et de réutilisation sont très proches dans ce contexte, car pour aider à la réutilisation et pour rendre en même temps la personnalisation possible, il faut améliorer la navigation et l'ergonomie : faciliter la localisation, la compréhension, l'intégration et l'extension de nouveau code.

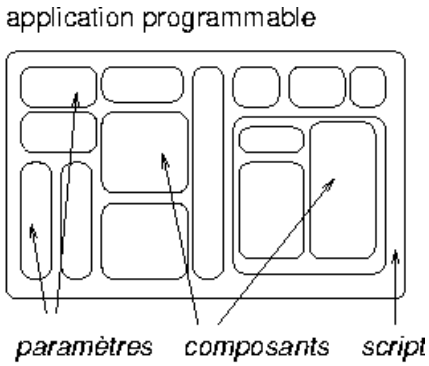


FIG. 3.2 – Programmabilité.

On peut, dans le contexte de la flexiblité pour l'utilisateur, donner une échelle des degrés de flexibilité légèrement différente de celle de [NT95] détaillée plus haut, qui se situait nettement sur le plan de la programmation. En reprenant [TMH87], les niveaux successifs en sont :

- flexibilité (le système s'adapte parce qu'il est générique),
- paramétrabilité (le comportement du système peut varier selon le paramétrage),
- intégrabilité (le système s'intègre correctement dans un cadre de composants),
- tailorabilité (le système est modifiable).

Nous allons maintenant poursuivre notre analyse de la flexibilité pour l'utilisateur d'abord pour les activités de personnalisation, et ensuite pour les activités de programmation par l'utilisateur.

3.2.2.2 Flexibilité et personnalisation

Quelles sont les possibilités des techniques de personnalisation ? Quel degré de flexibilité peut-on obtenir par ce type d'approche ?

Si [BD95] regrette qu'il n'existe, dans les logiciels actuels, que peu de possibilités de personnalisation en dehors soit de la personnalisation de la présentation (pour tous les autres cas il faut écrire du code), la personnalisation peut cependant s'échelonner sur des niveaux différents.

Un système tailorable, dans l'échelle de [Tri92] peut être modifié :

- par ajout de composants existants,
- par spécialisation des composants eux-mêmes,
- ou par création de nouvelles classes de composants.

Selon [KM95] la tailorabilité comprend trois niveaux, qui recoupent en fait les niveaux de flexibilité de [TMH87] :

1. choix d'alternatives prédéfinies,
2. construction de comportements/actions nouvelles à partir d'actions existantes (macros, scripts)
3. modification du système, du code, extensions.

Enfin [Mør97a] voit également trois niveaux :

- la personnalisation classique (fichier de définitions) ;
- l'intégration, qu'il définit ainsi : il s'agit de créer une séquence d'exécution de programmes et d'en faire une fonction nouvelle, enregistrée dans l'application avec un nom ;
- l'extension radicale, définie par trois types d'extension :
  - simple : ce type d'extension est accessible depuis les objets de présentation,
  - complexe : création de nouveaux éléments,
  - restructuration : par exemple de la hiérarchie de classes - factorisation en partie automatisable.

[Mør97a] remarque qu'il y a deux types d'intégration : l'intégration de type "hard" (plug-in, recompilation), et celle de type "soft" (agents, macros, scripts), dont la différence tient au fait que les

scripts sont éditables et ne sont pas seulement formés d'actions interactives, mais aussi de composants de haut-niveau. Enfin, selon lui, les niveaux intégration et extension se distinguent aussi par le niveau de langage. En effet, le langage d'intégration n'est pas le même que le langage d'implémentation.

Eléments personnalisables

Pour [Mør97c], le principe général n'est pas celui de la reprogrammation, mais celui de l'extension, on ne peut pas changer le code d'origine. Le logiciel décrit, BasicDraw, écrit en BETA, n'est d'ailleurs prévu que pour les extensions. L'objectif est d'être capable d'étendre l'application à tous les niveaux *pertinents pour un utilisateur* , c'est-à-dire ceux qui sont visibles dans le modèle du domaine qu'on peut percevoir à travers l'interface utilisateur. Les possibilités d'extension s'articulent à trois niveaux : la présentation (attributs graphiques), les design rationales (explications concernant la conception comme un moyen de comprendre réellement une conception à travers les raisons des choix et non par une description rationalisée), et le code lui-même.

Procédures

Concrètement, pour une extension, il faut :

1. trouver le nom du pattern BETA,
2. trouver le fichier correspondant : un éditeur apparait avec deux fenêtres : à gauche l'original, à droite le template d'extension.
3. quatre opérations sont ensuite nécessaires pour ajouter une extension dans l'interface :
   (a) créer la classe,
   (b) créer un item de menu,
   (c) créer un menu pour instancier l'item,
   (d) créer une extension pour instancier ce menu au démarrage.

Si l'on reprend le cadre conceptuel de dimensions cognitives décrit par [Gre96], on remarque qu'en général l'activité de modification requiert plus de précaution dans l'architecture d'un logiciel et de compétences que l'activité d'incrémentation : il est plus facile de reconstruire que de déconstruire.

Si l'on résume, les questions importantes concernant l'étendue de l'approche par personnalisation sont :

- Doit-on se limiter à un langage d'intégration (composition, scripts, macros) ou bien faut-il aussi un *langage d'implémentation* , permettant la création de composants ?
- Faut-il procéder par *extension* seulement à partir d'un système existant ? ou peut-on aussi permettre à l'utilisateur de modifier des composants ?

3.2.2.3 Flexibilité et programmation par l'utilisateur

La propriété de flexibilitité est plus difficile à appliquer dans le cadre de langages dont le but est plus que la flexibilité : la créativité, voire la construction d'application. La flexibilité maximale est en effet atteinte dans une activité d'incrémentation ou même de conception exploratoire [GP97a] puisque c'est l'utilisateur qui définit l'application. Cependant, si le langage pour l'utilisateur permet de faire de la composition, on peut parler de la flexibilité comme intégrabilité de [TMH87], qui s'arrête donc au composant lui-même.

Enfin, puisqu'on parle de langage, il existe alors un méta-langage sur lequel pourrait porter des possibilités d'extension : cela pourrait être ajouter un élément au vocabulaire, comme le vocabulaire des composants graphiques, ou modifier une méta-règle indiquant un ordre d'évaluation des règles. Comme le constate [MMW00], cela peut certes sembler un peu excessif, voire impossible en tous cas pour un utilisateur ayant besoin de ce type de système. Par contre il me semble que ces possibilités sont intéressantes dans la mesure où un utilisateur plus expert (un parent, un professeur, ...) peut les exploiter pour mettre de nouvelles possibilités à la disposition des autres utilisateurs. De façon générale, il n'y a pas de raison de bloquer la flexibilité "vers le haut".

Rappelons également le lien mis en évidence par [Rep93a], entre la flexibilité et le fait de rendre certains objets explicites du point de vue du formalisme ou de la notation. Cela n'est pas utile dans

l'usage familier, mais dès qu'on veut donner plus de contrôle à l'utilisateur, lui laisser la possibilité de redéfinir certaines caractéristiques, cela consiste en général à faire émerger des structures au niveau de l'interface, à passer de l'implicite à l'explicite.

3.2.2.4 Conclusion sur la flexibilité

Nous avons décrit dans cette partie un certain nombre de dimensions concernant des concepts relatifs mais provenant de contextes distincts (génie logiciel des composants, logiciels personnalisables). Le tableau 1 tente de résumer les différentes dimensions qui nous ont semblé pertinentes pour mesurer la flexibilité dans le cadre de la programmation par l'utilisateur.

Il nous a également semblé utile de placer la flexibilité souhaitable pour les utilisateurs dans la perspective de la flexibilité logicielle en général, puisque vraisemblablement cette dernière ne peut que lui être profitable. Mais, alors que souvent la flexibilité est pensée dans le cadre de la construction de composants génériques, on voit que, dans le contexte de la flexibilité pour la programmation par l'utilisateur, il en est tout autrement. Il faut ici pouvoir concevoir qu'un système soit à la fois spécifique et flexible, ce qui ajoute une difficulté et nécessite peut-être de nouveaux mécanismes.

3.2.3 Conception réflexive

3.2.3.1 Systèmes réflexifs

Un système est dit réflexif s'il agit sur lui-même avec les connaissances qu'il a de lui-même. Pour [Mae88] un système réflexif est un système qui raisonne, agit, etc ... sur lui-même de manière causale (a system "about" himself).

Relation représentation-comportement (causalité)

Ce qui nous intéresse en effet particulièrement dans l'idée de réflexivité, c'est la relation causale entre la représentation interne statique d'un système en train de s'exécuter qu'est son code, et le fonctionnement actuel de ce système (figure 3). Un système réflexif contient donc des structures qui représentent des aspects de lui-même et qui y sont liées de manière causale (de manière précise, plus ou moins synchrone, et avec un retour : il peut aussi se modifier lui-même - d'où le lien avec les protocoles méta-objets) [Mae88].

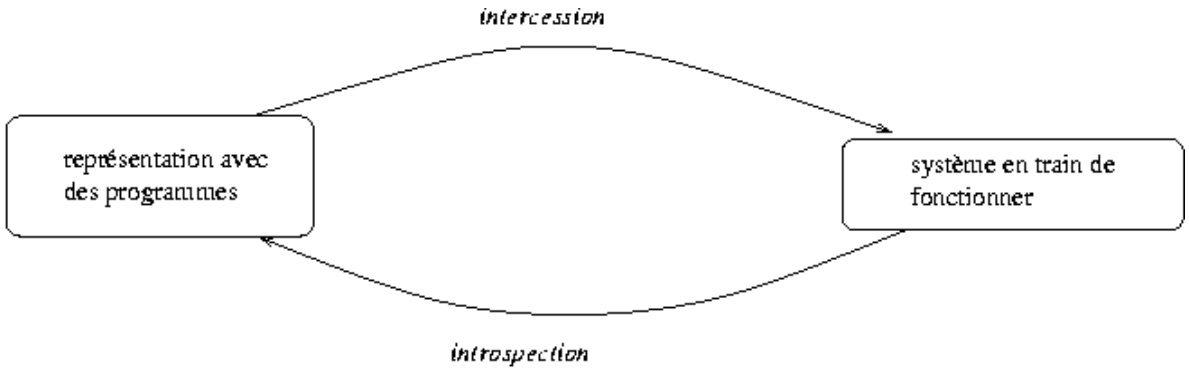


FIG. 3.3 – Causalité.

De même, pour [Dou96a] la réflexion computationnelle est une représentation explicite du système causalement liée à ce système et à son comportement effectif par une relation bi-directionnelle : dans le sens représentation-comportement (relation d'intercession) et dans le sens comportement-représentation (relation d'introspection).

Comme on le voit dans la figure 3, cela représente, dans un sens (introspection) la possibilité d'une réutilisation maximale des informations qui sont déjà présentes dans les structures internes, et dans l'autre, en modifiant cette représentation, de modifier le fonctionnement du système en cours

Degré de flexibilité	[TMH87]	1. flexibilité : logiciel générique (exemple : tableur) 2. paramétrabilité : plusieurs comportements au choix 3. intégrabilité : composants recombinables 4. tailorabilité : évolutivité, modifiabilité (a) ajout de composants dans l'interface (b) spécialiser leur comportement (c) créer de nouvelles instances
Type d'activité	[Dou96b]	1. sélection (correspond aux trois premières catégories de flexibilité) 2. création [Gre96] (a) incrémentation i. intégration ii. ajout de nouvelles fonctions (b) modification i. extension [Mør97b] A. simple (accessible depuis les objets de présentation) B. complexe (création de nouveaux éléments) C. restructuration ii. modification du système (c) conception exploratoire i. même chose que restructuration
Déclaratif/programmatique	[KLL+97]	1. déclaratif (a) adaptatif (b) choix implicite (c) choix explicite 2. programmatique (a) sous-classe, fonction, composant
Durée		1. transitoire (une session) (a) exploration 2. permanent (toutes les sessions à venir) (a) d'autant plus nécessaire si modification ou complexité
Mode d'utilisation	[KLL+97]	1. standard 2. optionnelle
Complexité structurelle	[NT95]	1. paramétrisation fonctionnelle 2. composition logicielle : la structure est un paramètre, mais les mots du langage ne sont pas généraux (composants) 3. programmation
Puissance d'expression (degrés de libertés, généralité,...)		1. langage spécialisé (exemple : HTML) 2. langage généraliste
Niveau technique du langage		1. interaction 2. scripting 3. implémentation

TAB. 3.1 – Flexibilité

d'exécution (intercession). C'est une base technique idéale pour la programmation dans l'interface, car la causalité garantit la cohérence entre les deux représentations.

Types de réflexion

On distingue la réflexion *comportementale* de la réflexion *structurelle* , la première consistant pour le système à réfléchir et utiliser ce qu'il sait de son comportement, la deuxième s'intéressant aux structures internes plus statiques du système (les classes par exemple). [Mae88] différencie la réflexion *procédurale* de la réflexion *déclarative* , qui ont une portée différente : dans la réflexion procédurale, la représentation interne ("self-représentation") coïncide complètement avec le programme qui implémente le système alors que dans la réflexion déclarative, on n'a que des spécifications de contraintes partielles. La réflexion est *explicite* si elle est utilisée ponctuellement, par exemple pour déboguer ; elle est *implicite* si tout le système repose sur la réflexion [Mae88], ce que visiblement [Rao91] distingue comme réflexion complète (architectures reflexives, avec lecture et écriture du code évalué, modification, exploration ; en bref, il y a une connexion causale entre la représentation et le processus) de la réflexion partielle (courante dans de nombreux systèmes sous forme de fonctionalités diverses). [Rao91] distingue également la réflexion *implémentationnelle* de la réflexion *computationnelle* , plus générale : la première concerne les cas où on laisse l'utilisateur du système fournir sa propre implémentation d'une partie localisée du système, par exemple l'implémentation de stratégies de dessin de fenêtres pour le cas particulier d'un tableur.

Utilité de la réflexion

Il existe plusieurs raisons pour un système d'utiliser les connaissances qu'il a de lui-même.

- [Mae88] décrit l'utilisation de la réflexion sur la plan pratique : pour déboguer, analyser les performances, interfacer, bref pour l'organisation interne et l'interface. On trouve des outils réflexifs dans les langages, mais l'utilité de la réflexion n'est pas souvent reconnue, ce qui a pour conséquence que le programmeur applique des solutions ad-hoc et que le résultat est moins propre, moins fiable, moins modulaire et plus coûteux en terme de lignes de code.
- Un autre exemple d'utilisation donné par [GNZ00] est la possibilité de configuration dynamique de composants par la connaissance de leur interface.

Dans notre prototype biok, la réflexion est utilisée de plusieurs manières : de manière comportementale pour détecter les dépendances entre les objets induites par les formules, mais aussi de manière plus structurelle pour les outils de programmation. Dans la mesure où le programme ne coïncide pas avec la connaissance qu'il a de lui-même, c'est une réflexion déclarative et explicite.

3.2.3.2 Protocoles méta-objets

Des systèmes modifiables

Les protocoles méta-objets (MOP pour meta-object protocol) se définissent assez bien à partir des éléments vus dans la section précédente. Ils correspondent à la réflexion implémentationnelle, puisqu'il s'agit de laisser à l'utilisateur la possibilité de définir sa propre implémentation : simplement, les représentations explicites à partir desquelles fonctionnent les mécanismes de réflexion se font à base d'objets ou plutôt de méta-objets [Dou96a]. Comme le dit [HK91] : on peut non seulement apporter des modifications au système, mais les les changements en tant que tels sont "objectifiés". Dans la terminologie de la réflexion, les MOP correspondent surtout aux protocoles d'intercession qui modifient le comportement du système en utilisant cette représentation.

Plus spécifiquement, les protocoles méta-objets se sont développés dans le domaine des langages à objets : leur but était de rendre accessible et modifiable par l'utilisateur les objets qui définissent les objets, d'où leur nom. Ainsi par exemple [Coi88] décrit les méta-classes comme un mécanisme permettant d'ouvrir les langages et construire des architectures ouvertes. Objvlisp a pu servir à simuler plusieurs sortes de langages objets différents, en définissant dans une méta-classe la stratégie d'héritage, la représentation interne des objets ("make-object" pour implémenter l'objet soit avec des vecteurs, hashtables, structures,...), l'accès aux méthodes avec cache, la liaison des variables d'instance soit par un "tree-walker" soit par un "send", l'accès aux variables d'instance avec distinction privé/publique ou par valeur active. [KdRB91] a systématisé et répandu cette technique.

Techniquement parlant, définir un *protocole* , c'est :

1. définir des *types d'objets* ,
2. leur associer des *opérations* ,
3. c'est aussi définir un *comportement par défaut* ,
4. et des changements *incrémentaux* .

Pourquoi s'intéresser aux MOP dans le cadre de la programmation par l'utilisateur ?

- une partie de la démarche est similaire, et correspond au principe de système modifiable ou re-programmable ;
- la variabilité et les requêtes des utilisateurs sont prises en compte [Kic92] ;
- les protocoles méta-objets apportent de bons principes de conception pour la flexibilité ;
- ils apportent aussi de bons concepts de construction d'interface, comme la réification, l'aspect explicite et documenté ;
- ils critiquent l'idée de boîte noire et de barrière d'abstraction : nous verrons que cette critique est transposable à la problématique de la programmation par l'utilisateur à travers les aspects de barrière technique, et d"'accounts".

Rien n'interdit d'appliquer les protocoles méta-objets à un autre système qu'un langage de programmation par objets. [Rao91] l'applique à un système de fenêtres, mais on pourrait l'appliquer à chaque fois qu'il s'agit de réifier la définition d'un objet dans un système pour le rendre modifiable. Ainsi par exemple, un style dans un traitement de texte qui devient manipulable en lui-même constitue un tel méta-objet. Cette idée peut se généraliser à travers l'idée d'interaction instrumentale [BL00], qui consiste à systématiser cette notion de manipulation directe non plus de l'objet d'intérêt, mais de l'instrument opérant sur cet objet. C'est une démarche qui permet de prendre en compte cette situation que nous avons déjà décrite de "work of the tool", par opposition à "work of the work" qui se produit lorsque l'instrument ne marche plus [HJ93].

La figure 4 montre qu'une transposition entre le domaine des langages objets et un autre domaine s'effectue aisément. Dans notre contexte, il faut par contre retirer le terme "objet" de l'expression puisque nous ne construisons pas un langage à objets. Si nous conservons le terme de "protocole", c'est parce que nous pensons qu'un protocole n'est pas très différent par sa fonction d'une interface, on peut même affirmer qu'il *est* une interface. Dès lors, si nous voulons choisir un terme pour notre démarche, cela peut être : MAP (méta-application protocol) ou encore : MAI (méta-application interface).

If faut distinguer la méta-programmation des protocoles méta-objets : la méta-programmation concerne la définition de méta-programmes, c'est-à-dire des programmes qui décrivent mais surtout génèrent des programmes [RAZ99][Rid00]. Ceci étant dit, cela concerne tout de même l'idée de *description* d'un programme dans un autre langage. Mais du côté de la méta-programmation, il n'y a pas nécessairement de réflexivité, ni cette idée d'un système qui se modifie lui-même par des méthodes réifiées dans un protocole. Il n'y a pas non plus l'idée de méta-circularité (quand le modèle et le méta-modèle ou le langage et le méta-langage sont dans le même langage).

Barrière d'abstraction, barrière technique

Les auteurs pronant les protocoles méta-objets et les systèmes ouverts considèrent l'argument de boîte-noire et de barrière d'abstraction, destiné à protéger l'utilisateur des détails de l'implémentation, comme une limitation pour la flexibilité des systèmes informatiques, et avancent que la démarche de transparence et de manipulabilité n'est en rien incompatible avec l'abstraction et la notion d'interface, bien au contraire. Nous pensons que ce raisonnement est tout à fait transposable à la problématique de la programmation par l'utilisateur, en terme non plus de barrière d'abstraction mais de barrière technique.

- Il faut protéger le système pour en assurer la robustesse.
- Derrière l'idée qu'il faut protéger le système, il y a aussi l'idée de barrière technique. Il faudrait protéger l'utilisateur des complexités de l'implémentation et de la présentation de code. Tout comme [KAR+93] montre que l'utilisateur peut au contraire, par une connaissance explicite des mécanismes et des possibilités du système, aider à en optimiser le fonctionnement, nous pensons que la connaissance du code peut guider l'utilisateur qui le souhaite, de manière documentée, à accéder à un niveau de programmation plus avancé. Par exemple, parmi les systèmes de programmation par démonstration [Cyp93b][Lie00], nombreux sont ceux qui cachent le code généré, alors que l'accès à cette représentation pourrait être très utile à l'utilisateur pour four-

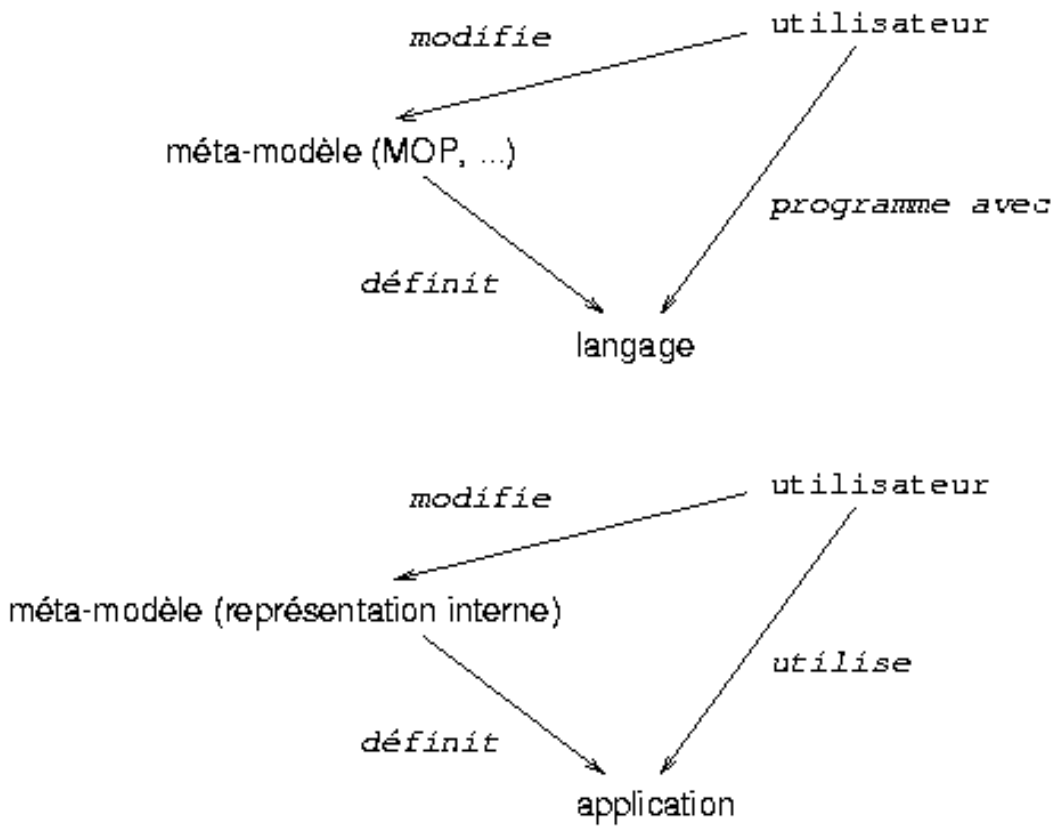


FIG. 3.4 – Transposition de l'idée de MOP.

nir des informations précises au système. La programmation par démonstration pourrait ainsi servir d'interface pour l'apprentissage, du moins si l'utilisateur s'y intéresse. Certains auteurs, comme [DA89], pensent que l'accessibilité de la représentation interne est bien plus important que d'autres qualités plus souvent avancées des langages de programmation, comme la simplicité formelle.

C'est pourquoi, de même que la conclusion de la critique de la séparation entre abstraction et implémentation conduit à l'idée d'interface duale [Kic92], l'idée d'aménager la barrière technique entre interface utilisateur et programmation amène l'idée de programmation dans l'interface, ou encore de niveaux d'interface d'une application, mais accessible à partir du même contexte (celui de l'utilisation), par opposition à l'aspect bi-modal environnement de programmation / interface utilisateur. On remarque notamment que l'idée d'interface duale est graphiquement au même niveau :

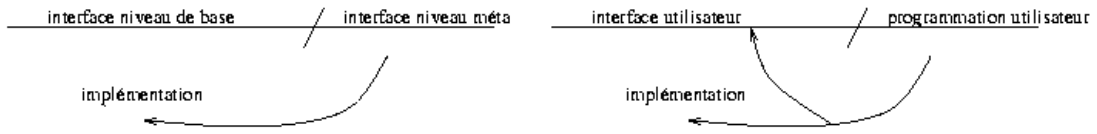


FIG. 3.5 – Interface duale.

La figure 5 montre l'analogie des concepts de programmation méta-objet et de programmation dans l'interface. Il y a une transposition vers l'interface utilisateur, ainsi qu'une différence notable : il s'agit aussi par la programmation de modifier l'interface utilisateur, et de rajouter des fonctions.

Comportement par défaut

Les protocoles méta-objets reposent en partie sur l'idée que leur utilisation est optionnelle. Le système possédant un protocole possède en effet un fonctionnement par défaut qui a été étudié pour convenir aux situations les plus courantes. Comme nous l'avons expliqué dans le chapitre II, notre démarche concernant la programmation par l'utilisateur, surtout lorsqu'il s'agit de personnalisation, est tout à fait similaire. Un système comportant des fonctions de programmation doit pouvoir fonctionner dans la plupart des cas sans nécessiter de programmation. Un argument parfois invoqué contre la programmation par l'utilisateur, considère à juste titre que celle-ci aurait avant tout un effet marketing, similaire à celui de l'ajout de nombreuses fonctionalités, qui en rendrait l'utilisation plus complexe, sans y ajouter rien de fondamental. On peut aussi penser le contraire, comme [BHHW99] qui pense que la possibilité de personnalisation peut *remplacer* la trop grande richesse de fonctions, mais il n'est pas sûr que la critique citée ci-dessus soit infondée et que la programmation par l'utilisateur ne soit pas affichée afin d'augmenter les ventes du produit. Il est tout à fait certain également que les fonctions de programmation ne doivent pas constituer un prétexte pour ne pas implémenter les fonctions importantes et laisser faire ce travail de conception et de programmation à l'utilisateur. C'est toute la différence entre donner la possibilité de faire varier le comportement d'un système dont le fonctionnement par défaut est soigneusement étudié, et fournir un grand nombre d'outils qui, s'ils étaient à la portée de l'utilisateur, permettraient d'arriver au même but. C'est pour ces raisons qu'il est important de combiner ces techniques de protocoles réflexifs avec une approche participative, car elles permettent de déterminer [Tri92] :

1. quelle est la part de variation à prévoir dans un système informatique - là où il faudra prévoir de la programmabilité sans doute
2. quelle est le fonctionnement par défaut souhaitable.

Ce qu'on constate dans les ateliers de conception participative, c'est l'importance d'un bon comportement par défaut, complet, suffisant pour les situations les plus courantes. C'est aussi la raison pour laquelle il faut fournir des exemples dans un système avec programmation pour non-informaticiens. Ainsi, pour [Tri92], le comportement par défaut doit aussi être suffisamment bien conçu pour faire fonction d'exemple de code.

Mais un comportement par défaut n'est pas nécessairement *générique* , et cette approche s'oppose à celle de [Mør97c] qui propose au contraire une approche de tailorabilité pour des applications génériques possédant les mécanismes nécessaires à la spécialisation à la portée d'un non professionnel.

Des principes de conception

Dans la même ligne que les patrons de conception ou les dimensions cognitives en tant qu'elles concernent les aspects de l'artefact programmable qu'est un logiciel, les protocoles méta-objets suivent des principes de conception qui nous semblent tout à fait utiles pour la construction d'un système flexible même s'il ne comporte pas nécessairement et précisément de MOP :

- Le principe des objets graphiques, comme dans Amulet [MMM+97], ThingLab [Bor77], Self [SMU95], et plusieurs autres systèmes nous semble correspondre à l'idée d'avoir un *noyau de base* , des éléments primitifs de construction et des mécanismes génériques, autour desquels on peut construire des variantes de systèmes.
- [Kic92] décrit certains de ces principes :
  - le principe de *portée* consiste à faire connaître et contrôler la portée des modifications ;
  - le principe de *séparation conceptuelle* permet d'utiliser l'interface meta-niveau sans avoir à tout comprendre ;
  - le principe d'*incrémentalité* consiste à ne pas avoir à tout refaire sous prétexte de modifier certaines propriétés (ce qui se fait assez bien en programmation par objets puisqu'on peut sous-classer) ; ces deux derniers principes rappellent le critère d'*approchabilité* de [Nar95], qui préfère les langages où il n'y a pas besoin de tout connaître ni de tout comprendre pour commencer à programmer).
  - le principe de *robustesse* , qui est plus difficile à maintenir, et qui peut poser un problème dans le cas de modification directe ou de prototocoles de type procéduraux qui, contrairement aux protocoles fonctionnels, peuvent agir par effet de bord sur l'état du système (on peut envisager d'absorber les erreurs depuis le run-time) [Tri92].

- [KAR+93] décrit encore plus précisément ces principes de *localité* dans la conception d'un protocole :
  - localité des fonctionalités (features) : il faut pouvoir accéder à des fonctionalités individuelles ;
  - localité textuelle : cela consiste à pouvoir identifier l'utilisation du MOP dans le code ;
  - localité des objets : pouvoir agir au niveau d'un objet (instance) ;
  - localité des stratégies : pouvoir ne changer qu'une seule chose ;
  - localité d'implémentation : changement incrémentaux et proportionnels en taille de code.

Les principes de portée, de séparation conceptuelle et d'incrémentalité sont en effet tous relatifs à une problématique de localité autour de laquelle tournent différents choix de regroupement dans les systèmes à méta-objets existants (classe, fonction, groupe, ...). [Kic92] donne les éléments d'une discussion sur la localité : réfléchir sur le découpage base/méta revient à parcourir un espace à deux dimensions dans lequel il n'est pas évident de localiser les points.

Protocole pour heuristiques

Un des objectifs des MOP est de laisser l'utilisateur redéfinir une implémentation pour optimiser le système dans un cas spécifique pour lequel le comportement par défaut n'est pas optimal. Dans le cadre de l'application des algorithmes d'analyse biologique, il est fréquent que le biologiste ait des connaissances précises qui pourraient permettre de réduire l'espace combinatoire du calcul. Mais il est en général impossible de communiquer ces connaissances à l'algorithme. En effet, le logiciel ne peut pas prévoir tous les paramètres imaginables ne serait-ce que pour tenir compte de toutes les connaissances biologiques existantes, ce qui le rend d'ailleurs bien trop complexe à utiliser. On le voit bien dans l'utilisation de certains logiciels de phylogénie, très peu de personnes parviennent par exemple à utiliser les fonctions complexes de pondération des données. Pourtant, et dans de nombreux cas, le biologiste pourrait apporter des connaissances qui aideraient le système à converger vers une solution.

Les idées directrices, que nous avons déjà évoquées dans le chapitre d'introduction (en 2.3.1 - la programmation comme une forme évoluée d'interaction), et qui concernent la problématique interaction/algorithmes nous ont été suggérées par [Weg97b] pour qui restreindre le calcul par ordinateur aux systèmes algorithmiques fermés réduit la possibilité de résolution de problèmes et de modélisation des systèmes complexes. La technique que nous avons appliquée pour augmenter la surface d'interaction de l'utilisateur consiste, sans modifier le code de l'algorithme lui-même, à :

1. mettre en place un mécanisme de mixin (dans XOtcl, un mixin fonctionne comme un filtre mais agit au niveau d'un objet particulier et non pour tous les objets de la classe) interceptant parmi les méthodes celles qui interviennent dans l'heuristique,
2. mettre en place un mécanisme interactif générique de spécification d'informations par l'utilisateur à partir des résultats intermédiaires.

Ce dispositif dynamique et générique, qui suppose bien sûr un découpage et une coordination au niveau de l'implémentation originale de l'algorithme, permet à l'utilisateur de simuler des comportements différents sans alourdir la structure du programme original. Nous avons exploré ce type d'interaction avec l'algorithme dans l'implémentation d'une méthode de prédiction de structure transmembranaire décrite dans le dernier chapitre.

Ce type d'intervention reprend l'idée des protocoles méta-objets de prise en compte des connaissances de l'utilisateur, mais ici, il s'agit moins d'optimisation du temps de calcul que d'une sorte d'"optimisation scientifique".

Des objectifs distincts

Comme on l'a vu, l'utilisation des techniques de protocoles méta-objet est intéressante dans le contexte de la programmation par l'utilisateur, particulièrement lorsqu'il s'agit de tailorabilité. Cependant, l'objectif classique de ces protocoles est l'implémentation ouverte. [Dou96b] (ou [Mae87]) rappelle les relations entre réflexivité, protocoles méta-objets et implémentation ouverte :

- la réflexivité est un *principe de conception* ,
- les protocoles méta-objets sont une *technique de programmation* , ne serait-ce que pour le choix de la programmation par objets,
- l'implémentation ouverte est un *objectif* .

Or cet objectif ne recouvre pas exactement celui de la programmation par l'utilisateur, qui sert

à d'autres choses, comme une modification sémantique du système, ou bien un ajout de fonctions. Il faut peut-être analyser en quoi cela modifie les principes de conception des protocoles méta-objets. Il est possible que cela les ouvre encore plus, avec une perte éventuelle au niveau des propriétés qu'il est possible d'assurer (robustesse, ...) - ce qui est un peu normal dans une situation où on donne à l'utilisateur une marge importante voir illimitée de créativité.

MOP et interfaces

Définir un protocole c'est, selon [KdRB91], réifier et rendre explicite une partie de l'implémentation du système afin d'en construire une interface qui en ouvre l'accès à l'utilisateur. Pourquoi ne pas aller jusqu'au bout de cette idée afin de rendre la démarche applicable à l'utilisateur "final", l'utilisateur de l'application ? C'est d'ailleurs l'idée d'interaction instrumentale développée par [BL00] où c'est l'instrument d'interaction qui devient objet de manipulation. Cela est également lié à l'idée qu'un formalisme explicite - par opposition à un formalisme implicite [Rep93a] - permet plus de flexibilité.

3.2.3.3 Persistance

La persistance est la conséquence technique et logique d'un système modifiable : pour l'être réellement, il faut bien sûr que ce soit de manière permanente. C'est d'ailleurs l'une des définitions de la personnalisation : un moyen de modifier le système *de manière permanente* (sans programmer) [Mac91b]. Comme le montre la figure 6, la représentation statique en mémoire externe est le prolongement technique de la représentation statique interne du programme qui est modifiée par les mécanismes réflexifs que nous avons vus ci-dessus. La difficulté tient au fait que la persistance ne correspond pas uniquement au code source sous forme d'instructions, de classes et de méthodes. Il y a un lien direct entre l'état du système en train de fonctionner et sa représentation sous une forme réinitialisable lors d'une session ultérieure. Cette représentation peut être de plusieurs types :

- la suite exacte des actions effectuées depuis le dernier cliché qui seront à ré-exécuter lors d'une session suivante ; le problème, c'est que cette représentation grossit au cours du temps et peut comporter de nombreuses actions inutiles, voire redondantes ;
- un vidage strict de la mémoire : tous les objets, leur état, etc... : le problème ici est que ce vidage de la mémoire n'en est pas tout à fait un ; il s'agit aussi d'une suite d'actions - mais de granularité différente - qui sont des affectations de variables et des créations objets, et il est très lourd de les ordonner dans un ordre correspondant à celui de leur vie effective ; cependant, pour tout autre choix, se pose alors des problèmes de dépendances, sans compter que le contexte peut changer à chaque session, rendant l'image mémoire invalide (un numéro de socket change à chaque fois par exemple) ;
- une procédure spécifique de sauvegarde pour chaque type d'objet, restreinte à certains aspects, jugés plus important que d'autres : mais alors lesquels choisir ? ce choix constitue *une réelle programmation* , dans le sens où il faut prendre des décicions, anticiper, etc... ; ce qui sera permanent doit être fixé d'avance, et donc les modifications du système opérées par les interactions ne constituent pas à elles seules une programmation de ce système - à moins de prévoir un protocole méta-objet, qui consisterait simplement à permettre à l'utilisateur de modifier la méthode adéquate ; c'est ce choix qui a été fait dans le prototype biok.

La persistance est bien une relation de causalité réflexive, comme cela est illustré dans la figure 6.

On peut remarquer la similarité de cette problématique avec celle de la programmation par démonstration [Cyp93b][Lie00]. En effet, il s'agit dans les deux cas d'extraire de l'information à partir d'un système en cours d'utilisation pour en déduire quelque chose qui soit réutilisable de façon permanente. Dans un cas, ces informations sont de nature dynamique (les actions exécutées) et temporelle, dans l'autre, les informations sont de nature statique, mais tout aussi complexe à traiter, puisqu'il faut inférer une description d'état initial à partir d'un état qui ne l'est pas.

Le fait de décrire l'état à inférer comme devant avoir un caractère d'état initial a son importance. Il s'agit d'éviter d'avoir une initialisation du système qui réexécute toutes les étapes, ce qui pour certains calculs rendrait l'utililisation des fonctions de persistance rédhibitoire. Il faut donc à la fois enregistrer l'état courant de l'objet, c'est-à-dire son apparence et son état interne, de façon à ne pas avoir à les recalculer, mais la spécification des actions qui amènent à cet état doit être présente aussi pour une

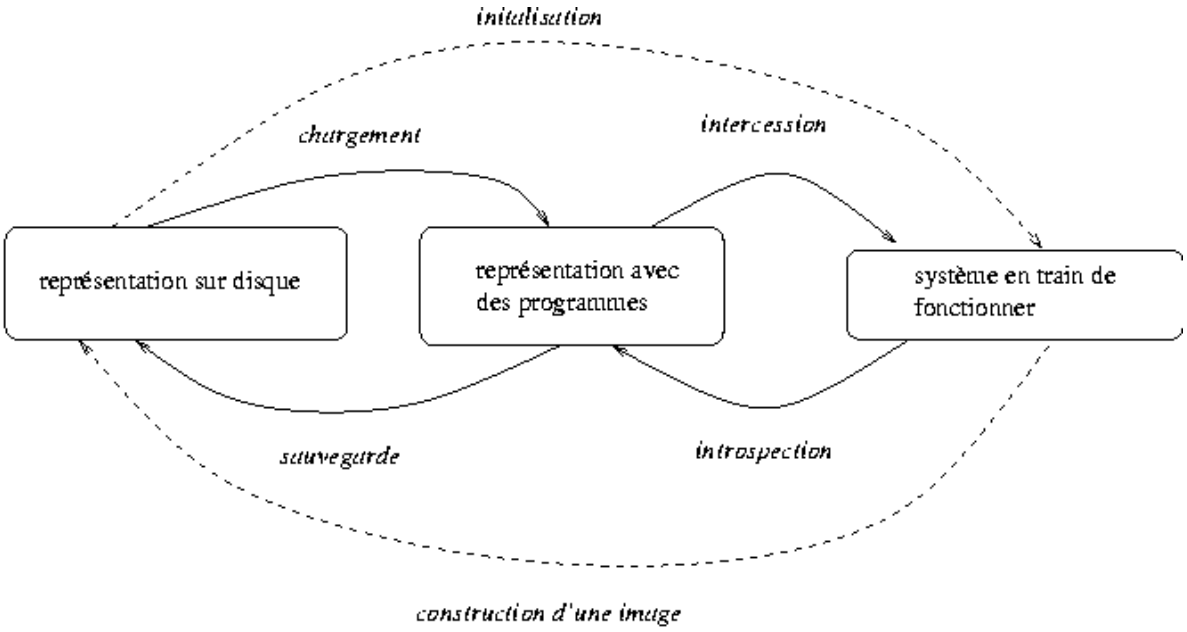


FIG. 3.6 – Persistance.

question de cohérence. Par exemple :

- conserver la courbe calculée *et* la formule qui spécifie la courbe ;
- conserver les tags graphiques affichés dans l'éditeur d'alignement *et* les instructions de chargement qui ont permis de les afficher, ou du moins l'effet de leur chargement (éditeur de tag ouvert, état interne de l'éditeur d'alignement cohérent, etc...).

Un autre aspect important de cette question de caractère initial de l'image (c'est-à-dire ne nécessitant pas de répéter les actions successives qui y ont amené lors d'une session précédente), est dû à l'architecture dataflow qui induit de nombreuses dépendances entre les objets. La granularité des dépendances n'est en effet pas celle des objets : il ne suffit pas de créer les objets ni dans l'ordre de leur création par l'utilisateur, ni dans l'ordre du graphe des dépendances de niveau formule, car le premier ordre n'est pas le même que le second, et les dépendances définies par les formules, considérées au niveau des objets, forment un cycle. Il suffit pour cela que l'objet A dépende dans sa formule d'*entrée* de l'objet B, mais que l'objet B ait un *attribut graphique* qui dépende d'une valeur présente dans l'objet A : alors que, au moment des actions de l'utilisateur il n'y avait aucune contradiction dans ces spécifications, car la granularité des actions était plus fine que celle du chargement d'un objet, il est impossible de choisir entre les deux ordres de création possibles à la réinitialisation, sauf si on décide de *ne pas réexécuter* les formules.

Enfin, comme le fait remarquer [Weg89], la persistance nécessite une notion très forte de l'*identité* de l'objet qui soit indépendante de sa clé interne d'accès et qui survive d'une session à l'autre. La détermination de cette identité doit donc faire l'objet d'une conception, et à ce titre, figure parmi les sujets potentiels de prototypage participatif décrit dans le chapitre II. Une solution par défaut qui semble raisonable est d'identifier un objet par le contexte de son utilisation, c'est-à-dire par exemple le sous-répertoire d'exécution du prototype où il figure à côté des données relatives, ainsi que par son aspect visuel et la position qu'il occupe sur l'écran et qui le rendent identifiable pour l'utilisateur. A cette fin, le comportement par défaut du prototype consiste à enregistrer les objets persistents dans un sous-répertoire Objects du répertoire courant, ou bien à pouvoir désigner ce répertoire d'objets persistents par une variable d'environnement BIOKUSER ; ce comportement est défini dans une méthode userdir de l'objet sources : cette méthode peut parfaitement être modifiée par l'utilisateur. Pour l'aspect visuel, les attributs graphiques et la géométrie sont repris d'une session d'utilisation à une autre.

Il y a aussi la question du choix du mode de persistance, de la manière dont celle-ci s'intègre dans l'utilisation du système : est-elle systématiquement appliquée à tous les objets, spécifique à certaines

classes d'objets, ou bien au choix de l'utilisateur ? Doit-elle être effective à tout moment (à la moindre modification) ou seulement lorsque l'utilisateur le demande ? Ou encore : peut-on imaginer qu'il y ait un mode persistent pendant lequel tout serait enregistré et un mode temporaire ? Une fois un objet déclaré persistent, l'enregistrement doit-il devenir automatique à chaque modification ? Tout comme pour l'aspect d'identité, ces questions doivent faire l'objet d'une conception, du choix d'un comportement par défaut et d'un protocole d'adaptation. Un comportement par défaut pourrait être :

- un choix par instance, modifiable interactivement par l'utilisateur avec une valeur initiale pour un nouvel objet définie par classe - par sélection ou plutôt par élimination (exemple : les objets éditeurs ou fenêtre de trace ne seraient pas persistents) ;
- ce choix serait visualisable : une couleur, un switch ;
- la persistance n'a pas besoin d'être continue pour tous les objets, mais pour certains, comme la partie utilisateur des fenêtres d'aide où ce dernier peut saisir son propre texte, elle pourrait l'être (en fait selon la nature plus ou moins durable du type d'information associé).

3.2.3.4 Conclusion sur les protocoles réflexifs

Les éléments que nous apportent les principes que nous avons décrits dans cette section sont essentiellement :

- La réflexivité permet de bénéficier d'informations présentes et garantit la cohérence entre l'état du système et sa représentation sous forme de code.
- Les protocoles méta-objets constituent une bonne technique pour moduler l'interface d'accès à ces informations.

Pour reprendre le cadre conceptuel cité ci-dessus, nous optons pour la réflexivité comme principe de conception mais notre utilisation des protocoles méta-objets reprend plus leurs lignes de conception et le principe même de système modifiable que la technique à proprement parler (ne serait-ce que parce que nous les appliquons à un système qui n'est pas un langage de programmation). Enfin notre objectif est la programmabilité, ce qui n'est pas complètement la même chose que l'implémentation ouverte.

De manière plus générale, il semble que les principes de réflexivité soient fortement liés à ceux de flexibilité et d'extensibilité. Les systèmes donnant une priorité dans leur architecture à la flexibilité comportent souvent de tels mécanismes [RAZ99][IdFF96][JNZM93]. Pour [Coi96] il y a en effet un lien nécessaire entre flexibilité et réflexivité, puisque qu'un système réflexif construit par définition les informations qui lui permettent d'évoluer. C'est l'intérêt des MOPs du point de vue méthodologique : on n'a plus besoin de choisir entre générique et spécifique : un comportement spécifique standard est fourni, on peut s'en servir pour en créer d'autres dans la mesure où ce comportement est décrit de manière suffisamment générique quelque part dans l'interface de niveau méta.

3.3 Quels niveaux d'interface pour un système programmable par l'utilisateur ?

Cette dernière partie a pour objectif de recenser et de décrire les axes qui nous semblent importants pour développer ces idées de flexibilité et de réflexivité dans le cadre d'un système programmable par l'utilisateur.

Nous avons décrit les problèmes de distance dans le chapitre I et évoqué les grandes directions des solutions existantes et souhaitables face à ce problème. Nous avons vu dans ce chapitre que ce problème est modélisé comme un problème de compromis utilisabilité/programmabilité matérialisé d'un côté par une distance entre la tâche et l'interface, et de l'autre par une distance entre le code source et l'interface [Mør97b].

Nous avons également vu dans la section précédente que les protocoles réflexifs reviennent non pas à plonger directement (figure 7) dans les structures internes et la représentation du système, comme le code source, mais à réifier les mécanismes définissant la structure de cette représentation sous la forme d'une interface de niveau méta. L'intérêt des approches de protocoles réflexifs vient donc non pas du fait qu'elles ouvrent les langages ou les systèmes, mais du fait qu'elles *ajoutent de l'interface* à leur

implémentation. Après tout, n'y a-t-il pas de nombreux systèmes livrés avec leur code source (bien plus que de systèmes comportant des mécanismes réflexifs) ? Pourquoi, comme cela est suggéré par la figure 7, et comme cela se produit dans la réalité, ne pas s'en contenter pour modifier une implémentation ? Les protocoles réflexifs possèdent comme on l'a vu des propriétés intéressantes : ils constituent un mécanisme réutilisable et modulaire, et font l'objet d'une conception et d'une architecture logicielle prévues pour que cela fonctionne bien. Ils sont conditionnés par une action explicite de réification documentée et résultent de la part de l'utilisateur d'une réflexion et d'un modèle qui s'y intègrent bien. Donc, ils ajoutent de l'interface. Le problème est alors moins : "comment ouvrir" que : "comment ne pas tout ouvrir n'importe comment". L'objectif de la programmabilité est différent de celui de l'implémentation ouverte, mais les principes de conception d'interface de méta-programmation sont similaires. Simplement, il s'agit non pas seulement de faire varier les implémentations, mais aussi de faire varier la sémantique en modifiant ou en ajoutant de nouvelles fonctions.

Enfin, si cette question d'une interface à l'implémentation se pose pour des programmeurs professionnels, on comprend qu'elle est d'autant plus importante pour des programmeurs non-professionnels dans un environnement où la programmation est occasionnelle et conçue pour cela.

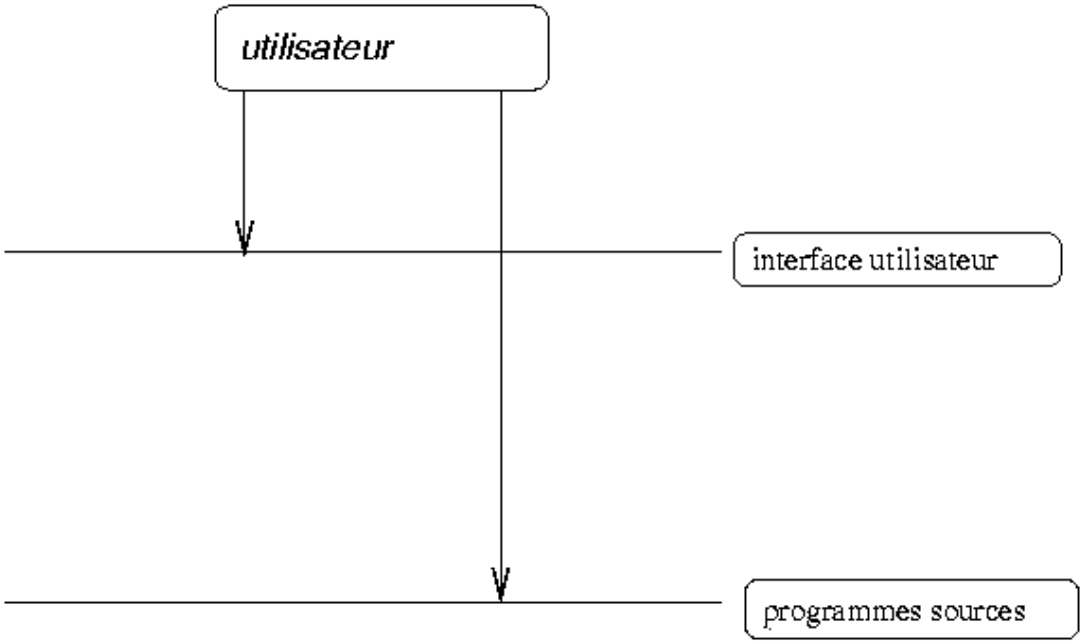


FIG. 3.7 – Accès direct au code.

Donc, ce dont il question ici, c'est d'ajouter de l'interface et de considérer la programmation non plus seulement comme un problème de traduction ou de compréhension de texte, mais aussi comme un problème de *navigation* dans un ensemble si possible bien structuré d'informations. [Gre97] insiste sur cet aspect, tout particulièrement pour la compréhension de programmes, tâche importante dans l'activité d'adaptation.

L'objet de cette dernière partie est donc de définir de quelles manières aménager la navigation (figure 8) :

1. par une mise en correspondance de l'interface et de la représentation sous-jacente,
2. par la détermination de niveaux intermédaires progressifs, et du passage de l'un à l'autre,
3. par la conception d'une interface utilisable pour le niveau programmation lui-même.

La question qui se pose à la vue de ce schéma c'est bien sûr de savoir si on ne peut pas unifier les deux parties "interfaces" ou "intermédiaires" de la figure 8 et faire en sorte que les interfaces graphiques "collent" aux interfaces méta, et inversement, que les interfaces méta soient réifiées par une interface

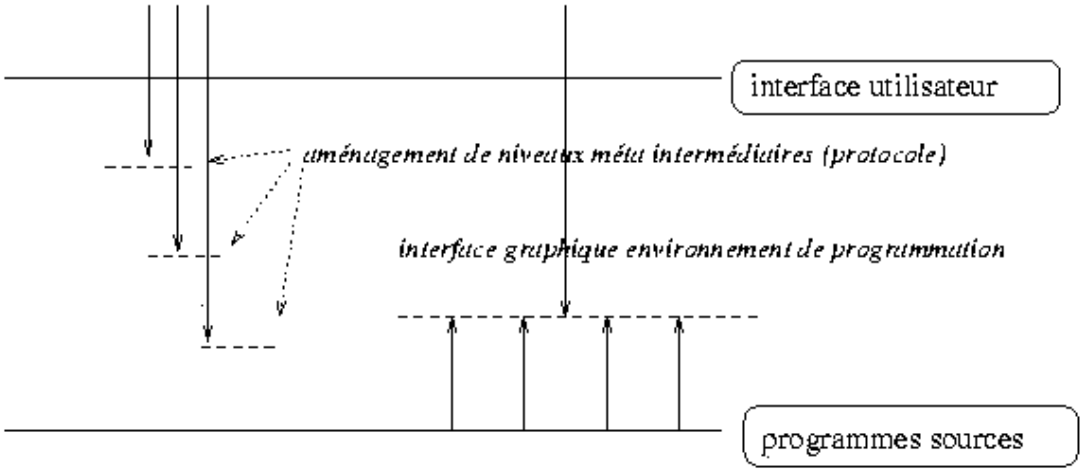


FIG. 3.8 – Niveaux et interfaces.

graphique. Comment jouer avec l'hypothèse d'orthogonalité entre les deux types d'interface [Mør97b] que nous avons rappelée dans le chapitre I ?

Nous proposons d'examiner les questions suivantes :

1. Articulation du langage de l'interface et du langage de programmation : quels sont les principes à suivre et les techniques à appliquer pour à la fois distinguer les deux niveaux mais rendre compte de la correspondance entre les deux ?
2. Indépendamment des niveaux de langage, est-il pertinent de concevoir des couches d'utilisation intermédiaires, et lesquelles ?
3. Peut-on utiliser les informations sur les structures internes pour la navigation, dans la mesure où elles sont accessibles sans trop de constructions supplémentaires ?
4. Les langages de programmation ne mériteraient-ils pas de meilleures interfaces d'utilisation ? Quelles sont les améliorations qu'on peut apporter aux environnements de programmation et à la programmation elle-même pour la rendre plus directe ?

3.3.1 Articulation des niveaux de langages

Correspondance entre les niveaux

Une première idée est de répertorier les éléments de part et d'autres entre lesquels on peut établir une correspondance.

L'idée centrale de l'approche sémiotique [dS00] est qu'il doit y avoir une articulation cohérente entre les niveaux de langages dans un système avec programmation par l'utilisateur : par exemple chaque action interactive doit correspondre à une commande du langage programmation par l'utilisateur, et il y doit y avoir la même granularité dans les actions interactives et dans l'éxécution pas à pas. Cela peut d'ailleurs servir de tutorial pour la programmation et cela permet de créer des scripts [BCC+97]. De la même manière, un enregistrement d'historique pour un environnement de programmation sur exemples ou tout simplement pour des macros doit comporter notamment tous les éléments existants dans les boîtes de dialogue [dS00].

[dS00] propose une étude détaillée de fonction de recherche/remplacement et de leur enregistrement par une fonction macro et donne des éléments concrets sur cette approche : les différences de niveau d'articulation constatées dans cette étude sont très gênantes. Par exemple, le sens en avant/en arrière est proposé dans l'interface mais n'a aucun équivalent dans le langage de macros. L'article propose quelques recommendations pour la production de signes : 1) reconnaître plutôt qu'inventer de nouveaux signes, 2) choisir des types expressifs habituels, 3) les signes qui appartiennent au domaine de l'ordinateur doivent

être présentés comme tels 4) articuler les signes jusqu'au point de manipulation auquel les utilisateurs pourront aller. [BCC+97] présente une autre étude concernant les langages de l'interface utilisateur (UIL) et le langage de macro de Word. On y détecte par exemple un problème de co-référentialité : les formattages sont de sens différents dans l'éditeur de texte et dans l'éditeur de macros.

Dans la logique de cette démarche, [Mør97c] propose d'indexer l'aide non pas sur les mot-clés du langage, mais sur ceux de la tâche. Les techniques de "disclosure" [DE95a] vont également dans ce sens, puisqu'il s'agit de faire apparaître dans l'interface utilisateur des informations sur les instructions exécutées au fur et à mesure des interactions ou sur l'état du système. Pour cela, des techniques comme l'interception d'actions graphiques [DE95a] sont à explorer.

Un autre type de correspondance qui nous paraît fondamental est la correspondance bi-latérale entre les spécifications faites dans l'interface et celles qui doivent pouvoir être faites dans le cadre plus classique d'un éditeur de fichiers : il faut pouvoir passer de l'un à l'autre. Une des techniques utilisables - pour les deux aspects - est la réflexion, qui permet de trouver l'équivalent en code de n'importe quel composant de l'application, eut-il été changé en cours de route.

Une autre question est de décider s'il faut des langages de programmation différents pour l'utilisateur et le système ? Il peut soit y avoir plusieurs langages de programmation pour les différents niveaux de programmation de l'application, comme les langages C et Tcl dans les architectures d'intégration décrites par [Ous98], soit un seul.

Transition entre les niveaux

Il faut également prendre en compte le contexte et les modes de transition d'un niveau à l'autre.

Une telle transition peut s'opérer par une navigation exploratoire, éventuellement initiée par un tutorial : il faut alors pouvoir tout parcourir et savoir où on est par rapport aux deux niveaux d'interfaces. Mais sur le plan pratique, c'est surtout lors de l'occurrence d'un problème qu'il faut pouvoir se repérer : [Mør97c] montre à ce sujet que c'est une situation où la distinction entre utilisation, adaptation et programmation s'efface. Ainsi, comme expliqué dans [Mør95], les unités applicatives doivent être indexées sur les parties de l'interface qui ont causé la panne et non sur la structure du système.

Les techniques de "disclosure" peuvent opérer dans des modes différents [DiG96a] :

- l'"echoing", qui consiste à afficher les expressions symboliques correspondant à une interaction à la souris,
- le "querying", c'est-à-dire la présentation de toutes les expressions valides dans un contexte, qui peut être déterminé par exemple par l'objet graphique courant,
- et le "monitoring" qui donne des informations pour composer.

3.3.2 Approche modèle en couches

Nous présentons ici deux solutions dont l'objectif est de moduler la distance entre les niveaux de représentations par un niveau intermédiaire : la première méthode vise à relier les deux niveaux, la seconde, que nous avons implémenté dans le prototype biok, crée un niveau de programmation de type "tableur" intermédiaire entre les objets du domaine et l'implémentation.

3.3.2.1 Transition rationnelle entre l'interface et l'implémentation

L'objectif de [Mør94] est d'articuler le niveau de l'interface utilisateur et celui de l'implémentation par une couche intermédiaire qui les décrit l'un par rapport à l'autre sous la forme d'explications de conception. Dans cette architecture, les différentes vues d'un objet, dans un sens architectural légèrement inspiré du modèle MVC, sont : la présentation (les attributs graphiques), l'explication de conception (*design rationale*) et le code lui-même. Pour aider à la navigation, les objets de l'interface sont des accesseurs ("handle") à travers lesquels il est possible d'accéder aux deux autres vues (cf Hypercard, [Gre97] ou [WB97]). Ensuite, les rationales servent à mettre en correspondance les objets du domaine et le code. Si on regarde les exemples de design rationale donnés dans [Mør97b], ils se comportent de manière similaire à des "bulles d'aide", à ceci près que l'information qu'ils donnent porte sur les décisions de conception qui ont été prises relativement à des fonctions de l'interface utilisateur

et constituent presque des explications, au sens de justification, ou même d"'account", sous la forme d'un historique des questions qui se sont posées. Mais selon [Dou97], avec l'idée qu'il est illusoire de prétendre que toutes les décisions de conception ou d'implémentation sont équivalentes, c'est aussi ce type d'information qui permet de comprendre comment fonctionne un logiciel.

Les questions auxquelles doivent pouvoir répondre ces éléments de documentation sont :

- que peut-on faire avec ce programme ? (but),
- qu'est-ce que c'est ? (description),
- comment faire ceci ? (procédure),
- pourquoi cela est-il arrivé ? (historique)
- qu'est-ce que cela veut dire ? (interprétation),
- où suis-je ? (navigation).

Les bulles d'aide sont donc des menus correspondants à ces différents types de questions. D'autres catégories de questions peuvent être utiles :

- quelles sont les alternatives ? (stratégie),
- montrer un exemple (illustration),
- qui contacter si rien d'autre ne marche ? (expertise).

On voit que ces explications sont plutôt prévues pour un contexte de panne, considéré comme une des conditions pour l'apparition d'un mode réflexif (comparaisons, prises de décision) et une prise de distance permettant la remise en cause de la conception [Mør94].

Les design rationale sont spécifiés dans un médium éventuellement non textuel - avec un éventuel éditeur spécialisé. Pour l'auteur, il est important que le media soit différent du texte, et en cela, ces documents deviennent en quelque sorte dépositaires de cet aspect "moins syntaxique" et plus visuel des croquis que l'on dessine pour s'aider à programmer et que la programmation visuelle cherche à retrouver [LM95]. Un exemple de rationale est donné dans [Mør98] : l'interface est une application de dessin (BasicDraw), et le rationale pour le dessin d'un rectangle à la souris est une *figure* expliquant les angles et les axes de rotation, ainsi que les dimensions (largeur, hauteur,...) tels qu'ils sont gérés par le code. Ces éléments correspondent sans équivoque aux *variables* du code, ce qui permet de faire la correspondance entre l'interface et le programme. On peut donc aussi comprendre les design rationale comme une explicitation des plans, au sens de la psychologie de la programmation [Dav90], constitués des idées qui ont amené à l'implémentation. Un utilisateur, lors d'une séance d'évaluation de l'application, formule implicitement ce rôle en disant que le rationale correspond bien au code grâce aux termes équivalents, mais en donnant une perspective différente, celle de l'idée génératrice de l'implémentation, représentée par une figure.

3.3.2.2 Modèle de calcul et interfaces : combinaison de la programmation par objets et du modèle dataflow

Cette approche consiste à construire au sein du système lui-même des couches d'utilisation de complexité progressive. C'est le choix que nous avons fait dans notre prototype biok, où il y a deux couches principales :

1. couche objets graphiques qui sont les objets d'intérêt de l'utilisateur (graphe, séquence d'ADN, entrée d'un banque de données, carte génétique, ...) : les objets graphiques peuvent interagir entre eux par le moyen de formules ;
2. couche d'objets XOtcl.

Une des raisons justifiant de n'avoir pas respecté le modèle strict du tableur où toute valeur est définie par une formule ("value rule") [Kay84], comme le fait [BAD+00], est d'abord que cela n'est pas facile. On se heurte en effet à des difficultés de conception voire potentiellement de compréhension pour l'utilisateur et ce modèle souffre des exceptions. Ainsi, dans le système Penguims (Programmable Environment for GUI Management and Specification) [Hud94], qui permet de construire des interfaces graphiques, tout objet de l'interface est défini par des cellules qui spécifient des contraintes. Les éléments graphiques à proprement parler, comme les lignes, ne sont pas des cellules, mais des items attachés. Les cellules fonctionnent comme des contraintes unidirectionnelles, et contrairement à la disposition classique d'un tableur, elle peuvent être regroupées librement. Les cellules comportent

un nom, un indicateur d'équation ('='), un bouton ouvrir/fermer, un indicateur d'héritage ('Like' = idée de programmation incrémentale à partir d'un code plus expert), un indicateur d'erreur (plus une bulle de message) dans la formule, et un indicateur d'activité. Un mécanisme de verrouillage déblocable permet de conserver l'héritage des méthodes de la cellule source. Le mécanisme de liaison entre éléments attachés et cellules est soit à sens unique, soit à double sens : dans le sens graphique -> cellule, il permet de déterminer la valeur d'une cellule selon un critère graphique (par exemple la position de la ligne déterminée par l'utilisateur). Il est possible de partager des objets par références indirectes, ce qui permet la composition ou bien d'avoir des objets qui en paramétrent d'autres. Mais ce système ne respecte pas la règle de calcul des valeurs par formules uniquement, car il emploie des interacteurs ou des macros qui peuvent modifier la formule d'autres cellules. De plus il emploie des valeurs 'code' pour les effets de bords comme les affectations. De même NoPumpG [Lew90] a pour but de fournir des primitives graphiques sans sortir du paradigme tableur (sans instructions impératives, macros, ni code en dehors des formules). Mais les types graphiques sont prédéfinis : l'utilisateur ne peut pas définir les éléments de l'interface par des formules ni créer des types complexes.

L'autre raison du non respect du modèle "tout tableur" est dans l'idée de passage progressif de l'utilisateur au niveau d'implémentation, celui où sont définies les primitives utilisables dans les formules. Il se trouve d'ailleurs que le langage des deux niveaux est le même dans biok, puisque c'est XOtcl dans les deux cas.

Modèle dataflow

Le paradigme de programmation reposant sur la spécification des flots de données [Jag95] est très présent dans le domaine de la programmation par l'utilisateur, notamment à travers les tableurs dont le principe de programmation est de définir la valeur d'un objet à partir d'autres objets sources, mais aussi dans les langages de programmation visuels. L'intérêt de ce type de système pour des non-informaticiens réside dans le niveau de spécification en principe assez élevé, de nature déclarative puisque le système se charge de définir le contrôle, et portant sur les objets théoriquement familiers à l'utilisateur que sont les données. De plus, et c'est particulièrement vrai en biologie et en analyse de séquences, le principe de composition de fonctions est particulièrement utile dans des contextes où les outils existant sont plutôt spécialisés dans un certain type de calcul et ne traitent par conséquent qu'une étape. La fonction d'enchaînement des fonctions d'analyse est par exemple très appréciée par les utilisateurs de Pise [Let00b], notre système générateur d'interfaces qui permet, à partir d'un résultat de calcul, de choisir interactivement (par un menu) le programme suivant dont les données d'entrées sont ainsi prédéfinies. Comme le constate [BAD+00], le modèle fonctionnel (quand des valeurs sont calculées par des fonctions qui renvoient à leur tour des valeurs) qui est le modèle général sous-jacent aux tableurs et au paradigme dataflow, est très bien adapté à la composition, d'autant plus que, selon [Gre95] il permet d'avoir des primitives sur des aggrégats, puisque les données arrivent en paquets.

Le modèle dataflow est donc plutôt une très bonne idée, mais il présente tout de même un certain nombre de difficultés, notamment du fait de la diversité des sémantiques possibles attachées au formalisme dans lequel il est spécifié. Dans le cas des tableurs, le formalisme est complètement implicite : l'utilisateur n'a pas à spécifier les relations entre les données par une notation particulière autre que la mention d'un objet dans la formule d'un autre. De même dans Pise, c'est le choix d'utiliser telle ou telle partie du résultat pour continuer les traitements qui construit implicitement ce graphe. Dans le cas des formalismes visuels, comme ceux de LabView [lab87], de ProGraph [CGP89] ou de Khoros/Cantata [RASW90], l'utilisateur doit spécifier les connexions de manière explicite. Des systèmes comme [Boy98a], où l'action de rapprocher deux boîtes crée une connexion et l'exécute fournissent un mécanisme qu'on peut qualifier de semi-explicite, puisque le contexte d'interaction est bien un ensemble de noeuds explicites, mais c'est par une interaction et non par une spécification que la connexion s'effectue, et ce, dynamiquement.

La sémantique de ces formalismes visuels (les noeuds, les arcs, la manière dont ils sont connectés et les attributs supplémentaires qui leur sont associés) est variable d'un système à l'autre, et n'est pas toujours très simple. D'après [Han94], les programmes visuels sont assez difficiles à lire, car il n'y a pas de connaissance acquise et commune de la signification des icones. [GP92] donne ainsi des mesures objectives de cas où un programme visuel est plus difficile à comprendre qu'un programme textuel. Dans certains cas, comme celui de Prograph, VIPERS [BM94] ou LabView, les noeuds représentent majoritairement des fonctions, et dans d'autres, ils peuvent en plus représenter des données (distinction

dataflow fonctionnel ou objet [Kim95]). Assez souvent, les noeuds représentent plusieurs choses différentes : des éléments d'interface, des fichiers, des variables, du texte, des structures de contrôle. Souvent, le type des noeuds est mixte, c'est-à-dire qu'ils peuvent être soit des données soit des fonctions : dans Piper [Biz00] ils sont des documents, des programmes ou des terminateurs, dans LabView, ils peuvent aussi être des données (tableaux), dans Prograph, il y a les "persistents" et dans STL [KCM90] les "variable/memory box". Dans IShell [Bor90], il y a à la fois des objets actifs (les fenêtres qui sont des applications actives) et des objets passifs (des fichiers et documents). Dans VIVA [Tan90] il y a a trois sortes de boîtes : sources, opérations et moniteurs, ainsi que dans VIPEX : traitement (code en Lisp), composition (assemblage de boîtes, exécution des sous-boîtes), group (sorte d'environnement pour partager des variables). Enfin, dans Visual Toolset [BJ90], selon que les boîtes sont juxtaposées horizontalement ou verticalement, elles servent à spécifier la construction ou le flot de données.

La sémantique de l'exécution peut également être assez complexe : elle est parfois contrôlable par l'utilisateur, comme dans InterCONS [Smi87], dans lequel les itérations sont contrôlées par un bouton. Certains langages ont plutôt un contrôle orienté données ("data-driven" : la fonction est invoquée dès que les données sont disponibles), d'autres sont dépendants d'une demande ("demand-driven" : en plus des données, il faut un jeton pour démarrer l'exécution) ; certains sont mixtes, comme Khoros/Cantata dans lequel, si on a mis des noeuds d'affichage ("glyph"), ce sont eux qui déterminent quelles boîtes doivent être exécutées.

Un trait important est celui de la vivacité du graphe : est-ce un graphe informatif, exécutable, dynamique ou à ré-affichage permanent ? Il nous semble également important de différencier les systèmes dans lesquels le formalisme représente la spécification, de ceux dans lesquels le graphe représente une instance en cours d'exécution. Dans LabView, l'interface est divisée en deux parties : le panneau frontal est l'interface utilisateur, et le bloc de diagrammes représente le programme source. [Bor90], dans le cadre d'un shell Unix visuel, IShell, souligne ainsi diverses difficultés sémantiques non résolues par la représentation visuelle (ici iconique) : par exemple, la machine représentée par l'icone connecté doit rester disponible, c'est-à-dire réinstanciable, pour d'autres usages - avec des connexions différentes : autrement dit, l'icone doit à la fois représenter un classe et une instance. La différence entre un qualifieur de commande (qui fait partie de l'exécution) et un bouton dans un panneau de contrôle (qui fait partie de la machine) n'est pas très facile à clarifier non plus.

Il est difficile d'évaluer l'intérêt de ces langages en dehors du domaine dans lequel ils sont utilisés [BH95]. Les utilisateurs de LabView avec lesquels nous avons mené des entretiens apprécient beaucoup cet environnement, mais considèrent que son avantage principal réside dans la richesse de sa bibiothèque de fonctions mathématiques et de traitement du signal. Comme cela est expliqué dans [OBCG99] ou [GPB91], il n'est guère possible d'évaluer globalement les fonctions de ces systèmes indépendamment du type de problème à résoudre. On le voit bien dans [Han94], rapportant une compétition de quatre langages sur trois types de problèmes de programmation : algorithmique (un générateur de nombre premiers), gestion et formulaire (un carnet de chèques) et graphique avec animation (une roue de wagon qui descend une pente dont l'inclinaison est réglée en temps réel par l'utilisateur). Prograph et VisaVis [PVM94] sont les plus efficaces pour l'algorithme de génération de nombre premiers, tandis que ChemTrains [BL93b] résoud facilement le problème d'animation car ce langage anime automatiquement les mouvements entre états. LabView utilise ses primitives sur les tableaux pour le générateur de nombres premiers et Prograph le type liste et ses opérations. De manière générale, d'après [Hil92], les langages visuels dataflow sont particulièrement bien adaptés aux domaines avec beaucoup de manipulation de données, comme le traitement d'image, la visualisation scientifique ou le graphisme.

Mais parmi les avantages du formalisme boîtes-flèches on doit tout de même citer la possibilité de mettre en place de moniteurs de visualisation, comme dans LabView où on peut placer ce type d'outils sur n'importe quel connexion, ainsi que la diminution des dépendances cachées [GB98].

Dans notre prototype, nous avons opté pour le principe de flot de données reposant sur des formules pour la composition des objets graphiques. Il nous a cependant semblé que ces formalismes explicites, par ailleurs coûteux à développer, n'apporteraient rien de particulier, ni sur le plan de la simplicité, ni sur le plan de la puissance d'expression, et qu'il était plus important de développer des formalismes visuels [NZ93][Rep93a] fortement attachés à la nature des données, comme l'éditeur d'alignement de séquences.

Modèle objet

Le deuxième niveau du prototype, celui où sont définis et implémentés les objets graphiques, leur composition et leurs méthodes, suit un paradigme à objets. Plusieurs aspects ont orienté ce choix.

Il semble que ce paradigme convienne assez bien à la fois à l'encapsulation et au regroupement de données, ainsi qu'à leur manipulation dans une interface graphique. Les méthodes orientées objets sont d'ailleurs particulièrement appréciées dans le domaine de l'IHM [MOO94] [Rob95][Col95][GHJV95] et de la programmation visuelle [BGL95][Dra92]. Ce regroupement à la fois conceptuel et visuel semble en effet convenir à une approche de programmation dans l'interface pour laquelle les fonctions de *localisation de code* sont essentielles, tout comme le sont les fonctions de *modification incrémentale* : la viscosité de ce type d'architecture est minimisée [Gre96]. Par ailleurs, ces langages permettent de construire des applications ayant une bonne flexibilité interne [GHJV95].

Nous avons vu dans le chapitre II que les études d'utilisabilité du paradigme objet, si elles reconnaissent son intérêt pour le support de la modification incrémentale, en signalent néanmoins la difficulté pour la maîtrise des liens et des interactions entre entités. Mais l'aspect programmation dans l'interface, surtout si l'interface est bien étudiée du point de vue de la navigation vers et dans le code, change sans doute un peu les données du problème. Il y a plusieurs aspects dans la programmation objet dont certains sont sans doute plus difficiles en tant qu'activités de modélisation, comme la conception d'une hiérarchie de classes ; mais d'autres sont peut-être plus abordables, voire simplifient la tâche de l'utilisateur, comme l'encapsulation qui est un principe plutôt pratique pour ranger et regrouper du code - et qui correspond à cette idée d'emboîtement qui fait l'intérêt des environnements comme Self [SMU95] ou Boxer [DA89]. D'autres types de relations que l'héritage sont aussi possibles et peut-être plus adéquats dans le cadre d'une conceptualisation exploratoire, comme la similarité, décrite dans [DB98a] qui permet de définir des relations 'is-like' au lieu de 'is-a', avec l'héritage de méthodes notamment. La variante des langages de prototypes [LSU88] [Lie87] offre des possibilités intéressantes pour la programmation expérimentale : l'utilisateur peut essayer de modifier le comportement des objets en modifiant les méthodes d'un objet et non de sa classe. [KMK95] présente une tentative d'exploiter cette nature "objectale" des interfaces utilisateur, dans un environnement qui exploite et étend le modèle MVC et où existent deux sortes d'objets : les objets graphiques et les objets de classe.

Enfin, il existe d'autres systèmes, notamment parmi ceux que nous avons cités plus haut dont [Kim95], qui combinent les paradigmes dataflow et objet sans que cela semble poser de problème particulier.

3.3.2.3 Granularité

Cette question est très dépendante bien sûr du choix de paradigme qui a été fait et du découpage en niveaux d'interface, mais dans le cadre de la programmation pour l'utilisateur, le niveau d'intérêt de l'objet manipulé par rapport au domaine d'application est particulièrement important. [Mør95] résume bien les termes généraux du problème : quelle doit être la taille d'une unité applicative (blocs constructifs de base des applications tailorables), afin qu'elle soit à la fois compréhensible et utile (puissante) : c'est un compromis entre l'utilisabilité, l'accessibilité, la modularité fonctionnelle, la compréhensibilité (pour les non-programmeurs), l'utilité (powerful pour les programmeurs), la non destructibilité, la pédagogie (support de l'apprentissage incrémental). Pour cet auteur, ce qui est également important, c'est que la granularité de l'unité d'interface soit la même que celle de l'unité de programmation (principe que nous avons suivi dans le prototype biok).

3.3.2.4 Pas trop de couches...

Pour conclure sur cette partie dont l'objectif était de décrire les possibilités d'établir plusieurs niveaux d'interface, il nous semble utile de préciser que s'il est important de fournir à l'utilisateur des contextes adaptés à son niveau d'utilisation et d'intérêt, il peut cependant être peu efficace de vouloir en ajouter trop.

[DP95], par exemple, évalue l'utilisabilité d'un environnement de programmation pour le langage Dylan, un langage objet compilé et dynamique (avec développement incrémental, liaison dynamique et objets auto-identifiants). L'objectif des tests est de voir si les programmeurs comprennent les fonctions de cet environnement, qui, par opposition au modèle fichiers, propose un modèle d'organisation du code

autour d'objets appelés "définitions", stockés sous forme de "source records" dans une base de données (une définition est une classe, une variable, une méthode, ...). Le principe de cet environnement est à la fois une continuité entre le mode de test de l'application et le mode programmation, tout en voulant maintenir une démarcation claire entre les deux. L'architecture repose donc sur deux processus, l'un servant les données de développement ("interactive cross-development model"), l'autre l'application. L'interface graphique est sous forme de panneaux présentant les différents aspects : listes de classes, méthodes, code d'une méthode. Le type de panneau (ce qu'il contient) est décidé par l'utilisateur. On peut le diviser en deux, en changer la taille, ainsi que lier dynamiquement des panneaux par des liens interactifs (hot links, créés par drag&drop de la flèche de l'un sur la flèche de l'autre). Les résultats des tests démontrent que cette architecture n'est pas comprise par les sujets des tests : les liens interactifs ne sont pas compris, de même que le modèle reposant sur des définitions par opposition au modèle plus classique utilisant des fichiers. Le modèle cross-développement n'est pas compris non plus, et pour les auteurs, cela est dû à trois raisons : l'environnement était pré-démarré lors des tests, l'interface n'est pas assez informative et il y a eu des problèmes de déconnexions. Les idées de l'environnement n'étaient pas dans la culture habituelle. Enfin les notions de développement incrémental et interactif ont induit des confusions probablement dûes aux termes indifférenciables dans les menus (update, recompile, ...) et aux notions de projets.

Cette architecture assez sophistiquée est cependant à notre avis beaucoup trop complexe, et ajoute un niveau supplémentaire qui n'a pas de correspondance forte avec des concepts existants dans les méthodes de travail des programmeurs. En somme, nous pensons qu'il vaut mieux rendre plus manipulables et plus navigables les objets ou les concepts qui existent d'emblée à la fois dans l'environnement et les concepts de programmation et dans l'environnement de travail de l'utilisateur, par exemple en utilisant d'un côté la réflexivité et de l'autre la conception participative.

3.3.3 Structures et interfaces

Il s'agit maintenant d'évaluer s'il est possible de faire reposer les aspects navigationnels de l'interface sur les structures internes de l'application. Ici, nous suivons toujours l'idée selon laquelle réifier (rendre explicite une structure), augmente la flexibilité d'un système. La propriété de reflexivité et les outils d'instrospection qui permettent de la réaliser fournissent un outil puissant pour supporter cette démarche.

3.3.3.1 Modèles d'architectures (Seeheim, ARCH)

Un des principes admis en ingénierie de l'IHM, décrit par les modèles d'architecture comme Seeheim [PtH85] ou ARCH [BCK98], est de minimiser le couplage entre le noyau fonctionnel et les niveaux interactifs (dialogue et présentation) : ce couplage est-il au contraire augmenté par les possibilités d'accès au code et à la structure tels que nous les préconisons ? [Rep93a] critique les UIMS qui créent plus des interfaces homme-widget que des interfaces homme/résolution de problèmes : selon lui, il n'y a rien pour construire des artefacts visuels. Selon l'auteur il faut, contrairement à ce qu'en dit le génie logiciel en ingénierie des interfaces, augmenter le couplage interface/application au niveau des objets d'intérêt, dans le sens où les éléments graphiques génériques que sont les boutons ou les menus sont trop génériques (on ne conduit pas une voiture avec des boutons).

Il s'agit cependant d'autre chose ici : plutôt d'aménager le passage d'un niveau à l'autre par une réification, sans pour autant renforcer les dépendances ni contredire l'idée de découplage interface-implémentation. La question qui se pose est : comment peut-on utiliser le modèle de Seeheim pour définir des couches ? On peut s'en inspirer pour spécialiser des éditeurs correspondant à chaque niveau :

- En créant une interface pour la présentation : édition d'attributs graphiques.
- Par une interface pour le dialogue : liaisons d'évènements, accès aux méthodes qui leur sont associées, spécification d'automates à états (ces aspects n'ont pas encore été traités dans le prototype réalisé). Ce type d'éditeurs, ainsi que le précédent, existe d'ailleurs dans certains environnements comme Vtcl [All00].
- C'est, nous semble-t-il surtout l'accès au noyau fonctionnel qui intéresse les biologistes : c'est à ce niveau que se définissent les fonctions d'analyse et leur sémantique, partie correspondant aux

connaissances des biologistes. Les étudiants en biologie qui ont utilisés le prototype biok pour leur projet semblent assez à l'aise avec ce niveau de programmation, et bien comprendre l'idée que cela concerne "tout ce qui n'est pas lié à l'affichage et à la souris" (c'est l'explication qui leur est donnée). La démarche pédagogique de [RCB90] confirme d'ailleurs cette idée que les aspects interactifs et de construction d'interfaces ne sont pas les plus simples, puisque ce ne sont pas eux qui sont abordés dans une première exploration de l'environnement ViewMatcher.

Cependant, comme tout chercheur, le biologiste doit pouvoir présenter ses résultats, et des problèmes se produisent fréquemment sur des questions de niveau présentation, comme dans ce cas où le logiciel avait prévu d'appliquer une palette décrivant les ressemblances entre des séquences, mais où le biologiste voulait, lui, plutôt colorer les familles de séquences ; le choix des objets ou des données sur lesquels appliquer une palette de couleur est donc aussi important que le choix des couleurs elles-mêmes.

3.3.3.2 Réflexion et interfaces

Comme l'on peut dans un sens utiliser l'introspection pour construire des boîtes de dialogue sur des objets (inspecteur de Smalltalk), et pour faire de la navigation, et, on peut dans l'autre sens introduire des informations concernant l'utilisation interactive au niveau même de la définition des objets de l'application, avec pour objectif de les rendre utilisables par introspection.

L'introspection est en effet un outil pratique pour créer de l'interface, du moins dans la partie environnement de programmation ou programmation directe. Il peut servir dans les cas suivants :

- pour la construction de feuilles de propriétés pour les variables et les structures de données, construites à partir de leur type et de leur structure interne (c'est le rôle de l'inspecteur d'objets dans Smalltalk),
- l'éditeur est en même temps une formulaire pour essayer une méthode avec un ou des champs prévus pour entrer les paramètres (dans biok, l'éditeur de la fonction est en même temps un formulaire pour l'utiliser, comme dans LabView [lab87]).

Intégrer le niveau interface utilisateur dans la définition des objets et classes

L'idée réciproque est d'augmenter le langage, éventuellement par le moyen de méta-classes, de façon à encoder les informations utiles à l'interface utilisateur dans le langage lui-même pour les utiliser ensuite de manière réflexive. Cela évite la redondance d'information et la double gestion de code et de structures de données. Nous décrivons une implémentation possible de cette idée en XOtcl dans le chapitre V (5.4.3.4).

Modèle MVC

Dans le prototype biok, la partie visible des objets graphiques correspond aux parties vue et contrôle et ils possèdent pour la plupart une partie donnée. Cette structure se reflète dans la composition des menus d'accès aux méthodes qui sont départagés entre vue et données. Cette structuration ne semble pas poser de problèmes particuliers à l'utilisation et permet de diminuer le nombre d'éléments dans ces menus.

Hiérarchie des classes

Cette question recouvre deux aspects :

- les possibilités de navigation utilisant la hiérarchie de classe et les relations de composition,
- la tailorisation par sous-classement.

Navigation

Les techniques sont par exemple :

- des menus hiérarchiques déroulant ou des listes de sélection comme dans l'environnement de programmation de Smalltalk (un IDE de ce type a d'ailleurs été développé pour le langage XOtcl),
- des accès aux super-classes à partir du menu d'édition d'un objet.

La première solution est moins accessible que la deuxième pour l'utilisateur, car elle va de l'abstrait au concret et est moins contextualisée que la seconde.

Composition

Le langage XOtcl fournit la notion d'aggrégat aussi bien pour les classes que pour les objets, supportant ainsi la relation de composition [NZ00a]. Les classes qui constituent les composantes de cet aggrégat (comme les classes Cell pour les cellules ou Area pour les zones du tableur ou les courbes dans un outil d'affichage de courbes), peuvent être déclarées comme :

```
Spreadsheet::Cell Spreadsheet::Area
```

afin que leur appartenance aux composantes de la classe Spreadsheet soit encodée et introspectable pour la construction des outils de navigation (au même niveau que les composantes vue et données par exemple). La commande :

```
Spreadsheet info children
```

permet en effet de retrouver ces composantes. C'est la même idée que les conventions de nommage dans les Java Beans comme support de l'utilisation des composants, mais cela va un cran plus loin puisque cela a un effet sur la structure effective des objets.

Spécialisation par sous-classement

La structure adoptée par [Mør97b] est adaptée à l'idée de personnalisation par extension en conservant la couche système. Le schéma général d'une extension correspond à celui du langage BETA dans lequel le système est implémenté :

```
code à exécuter avant

appel super-classe

code à exécuter après
```

En BETA, on ne peut pas changer le code d'origine, mais seulement l'étendre. Celui-ci est prévu uniquement pour les extensions (grâce aux extensions type-safe du langage BETA). C'est en effet la super-classe qui définit où se trouve l'extension, avec appel de cette dernière par l'instruction inner.

Mais la question qui se pose surtout est : faut-il systématiquement imposer à l'utilisateur de sous-classer pour étendre une classe ? Il nous semble à la fois plus facile et plus logique d'intégrer des nouvelles fonctions au même niveau. Une sous-classe ne doit de préférence être créée que s'il y a un réel changement sémantique induit par l'extension. Il peut cependant être utile que l'environnement et l'utilisateur soient conscients des changements (ajouts ou modifications) effectués par rapport à l'environnement standard, en prévoyant par exemple un mécanisme qui repère la redéfinition des méthodes du système sans sous-classer et prévient l'utilisateur. Ce mécanisme peut d'ailleurs aussi proposer à l'utilisateur de définir une sous-classe lors de la première redéfinition.

Le sous-classement n'a de sens que si l'utilisateur créée quelque chose de nouveau, comme c'est le cas par exemple lorsqu'il définit un nouveau tag de visualisation, correspondant à une nouvelle fonction. Alors que l'adaptation, comme par exemple d'ajouter un bouton dans le panneau de commandes pour rendre une commande plus accessible, ne constitue pas une création. C'est toujours de la classe concernée qu'il s'agit, comme par exemple la classe Plot, et il n'y a pas de raison "logique" de l'appeller autrement. Nommer est un problème plutôt difficile. C'est d'ailleurs un autre problème que poseraient les frameworks dans lesquels c'est une opération standard. Mais sous-classer, c'est un travail de modélisation, et un surcroît de modélisation surtout sans nécessité est à éviter (systèmes "abstraction hungry" dans les dimensions cognitives de [Gre96]). Sous-classer, construire une abstraction, est très utile quand on sait ce qu'on veut, mais peut être gênant quand on veut simplement expérimenter et prototyper [Weg89]. D'après [GB98], pour des programmeurs novices ou occasionnels, il faut faire très attention à ne pas les contraindre à des décisions prématurées, comme ce serait le cas s'il fallait absolument trouver un nom de classe, en prévoyant des étapes intermédiaires, en enlevant si possible les contraintes dans l'ordre des actions (premature commitment [Gre96]).

D'autre part ce travail de conceptualisation peut être réalisé par des utilisateurs plus expérimentés, comme les jardiniers [Nar95], ou les informaticiens qui maintiennent l'environnement.

En résumé, il faut donc *pouvoir* sous-classer, mais sans que cela soit *obligatoire* [GB98].

3.3.3.3 Relations entre la représentation statique et la représentation dynamique

Réduire la distance entre le code et l'application, c'est aussi réduire la distance entre la représentation statique et la représentation dynamique, et, de même qu'il y a de la navigation suivant la définition statique des objets (classe, composition, ...), il doit y avoir des possibilités de navigation suivant les interactions entre objets.

D'après [AGK+97], la dimension temporelle est une des dimensions les plus difficiles à spécifier dans un programme. D'ailleurs selon [LF95] c'est la la définition même de la programmation : spécifier un processus dynamique par une représentation statique. L'idée de représenter déclarativement un programme par un exemple d'exécution [Lie92], qui est analogue en programmation visuelle à celui des approches de programmation par démonstration, tente d'ailleurs d'aplanir cette difficulté. [FJG95], avec les cartes temporelles ("time-maps"), ramène la définition des séquences temporelles sur un plan analogue et dans un formalisme équivalent à la définition des relations statiques. [BAD+00] prend également en compte de la visualisation de la dimension temporelle par une fonction d'animation.

C'est aussi pourquoi le déboguage devrait être beaucoup plus étudié et développé dans les environnements de programmation courants [LF95]. [ULF97] confirme cette nécessité : les erreurs sont dûes à trois types de distances : spatiale, temporelle et sémantique. Pour minimiser la distance temporelle, il faut notamment avoir la possibilité de revenir en arrière.

Pour faciliter la compréhension des relations dynamiques, [LN95] propose un schéma de conception dont le but est de pouvoir combiner les éléments concrets (objets, invocations) et abstraits (classes et relations), et de filtrer la trace d'exécution par une sélection verticale (objets/classes) puis horizontale (relations/interactions).

3.3.4 Langage et interfaces

Il faudrait, de manière générale, améliorer l'utilisabilité des interfaces graphiques de programmation. Sans aller jusqu'à la définition d'une interface visuelle, il existe plusieurs approches pour rendre la programmation, c'est-à-dire la manipulation d'objets de programmation, plus directe, plus *palpable*.

3.3.4.1 Environnements de programmation et programmation dans l'interface

[Guz98] constate avec amertume que le modèle de l'environnement que l'utilisateur pouvait changer autant qu'il le voulait, pour le personnaliser, et dans lequel il est considéré comme un apprenant, s'est perdu avec les interfaces utilisateurs modernes. Reconnaissons en effet que, dans notre optique, ces environnements sont extraordinaires : le seul exemple de Thinglab, comme c'est décrit dans [Bor77] est exemplaire. On peut, d'un seul clic de souris, passer de la vue graphique d'un objet, designée par le terme 'picture' (l'interface, en somme) à son code, appelé 'structure'. On trouve ce type d'interface aussi dans Self [SU95], dans [GAL97] ou Boxer [DA89]. Ces systèmes ont pourtant une fonction bien définie et bien particulière qui explique cette souplesse dans l'architecture qui, indéniablement, facilite la manipulation du code : ce sont des environnements d'*apprentissage* , pour lequel la programmation est un but en soi. Le niveau de machine virtuelle est donc clairement celui de la programmation, voire de la construction d'interfaces.

Il existe plusieurs environnements de programmation permettant la programmation dans l'interface utilisateur :

- Smalltalk (cf environnement [GAL97]),
- l'interface ViewMatcher pour Smalltalk de [CSBA90],
- Self [SU95],
- Amulet [MMM+97] (surtout pour la recherche en construction d'interfaces),
- ThingLab [Bor81] : bien que ThinLab permette la simulation et la modélisation, il a surtout été utilisé pour construire des interfaces, comme pour la maintenance de cohérence entre vue(s) et données. Ainsi tous les exemples de résolutions de contraintes avec ThingLab sont des objets 2D.

Ces environnements constituent certainement un modèle pour notre approche, mais ils sont *avant tout* des environnements de programmation. La différence entre ces environnements et notre approche tient plus précisément aux aspects suivants :

- Un environnement de programmation est générique (permet de programmer n'importe quelle application).
- Un environnement de programmation sépare les objets graphiques de l'application des objets graphiques de la programmation. Dans biok, nous avons essayé de supprimer la frontière entre ces deux modes (le shell d'un objet appelle directement les méthodes, l'éditeur de méthode sert de formulaire, l'aide est modifiable pendant l'utilisation, etc...).
- Une application programmable, par opposition à un environnement de programmation, sait faire bien d'autres choses que définir des objets de programmation.
- A propos de Smalltalk, [GB98] remarque les difficultés dûes au niveau élevé d'abstraction de l'environnement : il y a beaucoup de classes et d'interactions entre ces classes à comprendre avant de pouvoir commencer à utiliser l'environnement ("abstraction hungry").

C'est toute la différence entre l'approche personnalisation et l'approche programmation, et ce, bien que les environnements cités ci-dessus comportent des exemples correspondant à une démarche pédagogique très étudiée. Dans notre approche, c'est d'abord le niveau de l'application destinée à l'analyse de séquences qui compte, la programmation est *en plus* . La programmation n'est pas le type d'utilisation standard de ce système, mais c'est un mécanisme standard de l'environnement.

C'est en fait un peu la même critique que celle que nous faisions à l'égard des approches frameworks : ce sont certes des systèmes modèles pour nous, mais leur utilisation dans le cadre de la programmation par l'utilisateur non-professionnel nécessiterait au moins quelque chose de plus, et ce plus est - et ce n'est pas rien - une vraie application qui constitue le premier niveau d'utilisation.

Literate programming

Il existe des techniques tout à fait intéressantes pour rendre l'activité de programmation plus pratique.

Par exemple, [Coc97] explore l'idée de *literate programming* (l'idée d'origine vient de [Knu84]), qui consiste à considérer le code source comme un texte qu'on peut disposer commme on le désire, et selon un découpage n'ayant aucun rapport avec la structure interne du programme, à des fins de lisibilité ou de compréhension logique. La figure 9 est une image d'un éditeur de code Java (l'éditeur étant lui-même écrit en Tcl !), appliquant, en plus des principes de découpage de niveau utilisateur, les techniques de visualisation contextuelle (holophraste). Tout le code est affiché, mais certains morceaux peuvent être compressés - pas complètement cependant : les fontes sont de taille très petite, ce qui fait qu'on garde une idée de la taille du morceau comprimé - selon un paramètre "degré d'intérêt" que l'utilisateur peut régler. On peut comprimer/étendre ces morceaux en cliquant sur le nom de la méthode ou le mot-clé de la structure syntaxique.

TFPDRAW [MT86] est un formatteur de code qui ajoute des symboles graphiques (triangles, flèches, semi-cercles,...) dans le texte pour mettre en valeur la structure (pour indiquer un exit, un bloc, un test, un goto). Les composants sont accessibles par un diagramme des relations entre modules (processus, données ou package).

Navigation

En plus des possibilités de navigation évoquées au sujet des structures internes, il y a des principes plus généraux de navigation comme ceux qui sont décrits par exemple par [Gre96], comme ceux de juxtaposabilité et de visibilité : cette dimension cognitive décrit la possibilité de visualiser des éléments simultanément et de pouvoir les comparer ; dans le cadre de la programmation, c'est notamment la possibilité d'ouvrir plusieurs éditeurs simultanés sur des méthodes différentes, ce qui doit théoriquement avoir pour effet de diminuer le nombre de méthodes faisant la même chose dans une application. Ou bien, comme dans [Nea89] qui décrit un système d'apprentissage à base d'exemples, il y a deux fenêtres juxtaposées, l'une pour le programme, l'autre pour l'exemple.

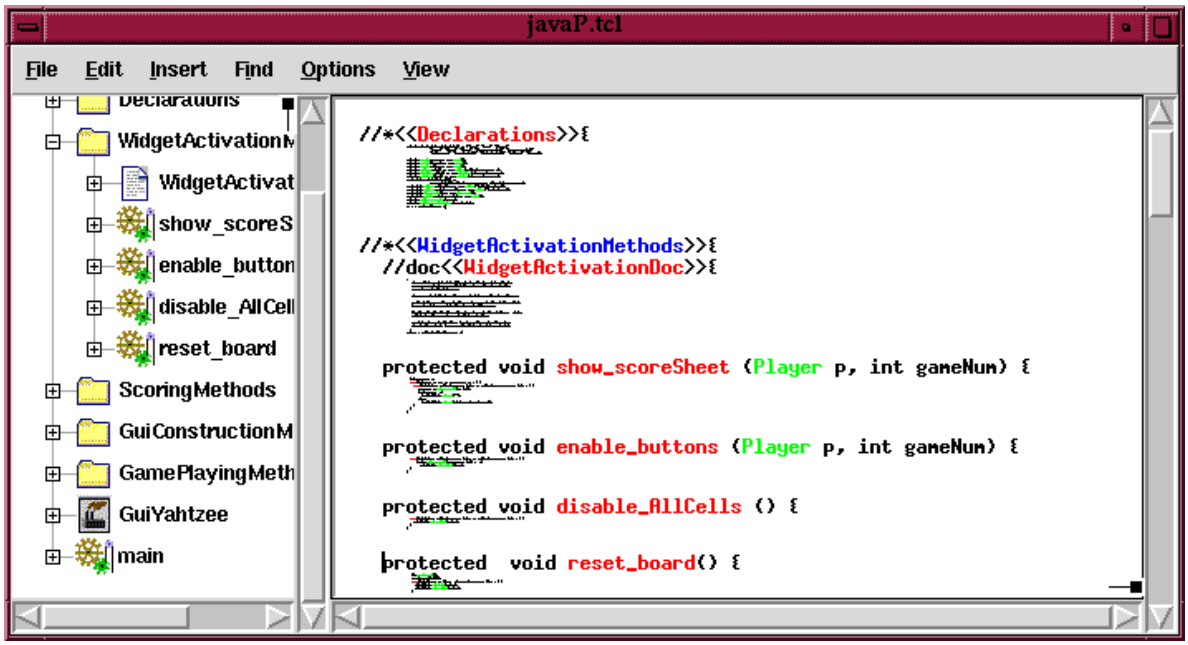


FIG. 3.9 – Literate programming.

3.3.4.2 Programmation directe

L'étape suivante dans l'idée de gommer les frontières entre programmation et utilisation consiste dans l'idée de programmation directe, où, d'une part, l'effet d'un changement dans le code est immédiatement perceptible, et où, de plus, le programmeur a l'impression d'atteindre directement les objets du programme. Cette approche a un lien important avec l'idée de causalité et de réflexivité puisqu'elle consiste à associer le changement du code directement avec le changement du comportement visible dans l'interface et à l'inverse permet de trouver le code (la cause) directement à partir de l'effet (l'objet affiché). Dans cet ordre d'idées, nous avons déjà cité les lentilles de déboguage magiques de [HRS97]. Il s'agit d'une "toolglass" (fenêtre semi-transparente dont la position est contrôlée par la main non-dominante) comportant plusieurs types de lentilles qui permettent de visualiser les classes qui interviennent dans un objet graphique, l'arborescence des widgets (ici ce sont de interacteurs). Cet outil fonctionne à la manière des requêtes dynamiques (dynamic queries) avec le choix d'attributs à visualiser ou le choix du niveau de l'arborescence. Bien sûr, l'intégration du code définissant les lentilles est invisible aux classes d'objets graphiques.

C'est aussi la propriété de "pokabilité", de pouvoir "toucher" les objets du programme [DA89], de matérialité du code [Kah00], ou encore celle de réalisme [CUS95][ULF97][KMK95], voire de réalisme naïf, qui consiste à donner l'illusion que les objets de l'interface *sont* les objets du système [DA89]. Ou encore, dans [SUC92], on cherche à supprimer la distance cognitive entre la réalité et sa description dans l'interface : la distinction entre *mentionner* et *utiliser* est modélisée comme une rupture cognitive. Nous pourrions citer également, comme autant d'exemples de systèmes de programmation directe :

- LiveWorld [Tra94] (figure 10) ;
- Le système décrit dans [PM99] (figure 11), qui est un système reposant sur l'idée de récursivité graphique. Le composant de base est formé de quatre zones rectangulaires : nom, entrée, puis modificateur et résultat. Chacune des zones peut elle-même provenir d'un composant dont elle forme la partie résultat. L'accès à des niveaux plus fins des opérations s'opère par ouverture de la zone correspondante (figure 11).
- Les boutons de [MCLM90] ou de [RHC91].

Une autre possibilité du côté des mécanismes des langages est d'avoir une couche de programmation **instance** pour travailler uniquement sur les objets présents dans l'interface. Le langage Otcl, ainsi que XOtcl qui est utilisé dans le prototype est un langage permettant la programmation au niveau des

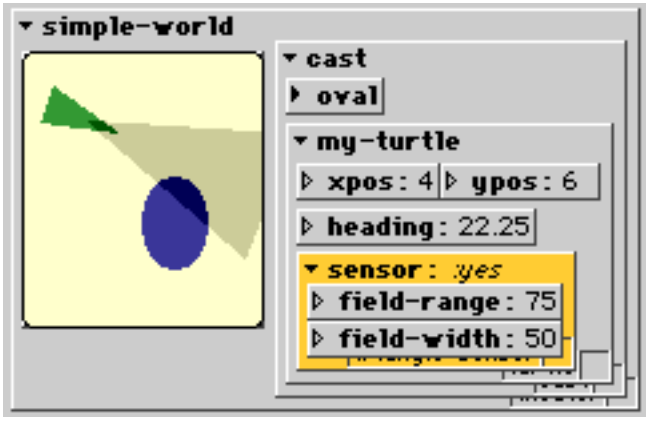


FIG. 3.10 – LiveWorld [Tra94]

FIG. 3.11 – Nested User Interface Components [PM99]

objets : il est donc possible d'associer une méthode à une instance unique. Soit par exemple une instance f de la classe Field :

```
Field f
```

On peut lui associer une méthode m

```
f proc m {x} {
}
```

qui n'est pas définie pour les autres instances de la classe Field. Ce mécanisme est particulièrement intéressant pour redéfinir une méthode de manière isolée, afin de l'essayer sans perturber les autres instances potentiellement présentes dans l'interface. La méthode redéfinie de cette manière continue cependant à exister sous forme de méthode d'instance, ce qui pose un problème lorsque le corps de la méthode contient un appel à sa version dans la superclasse (next). En effet, le schéma d'appel devient alors :

méthode d'instance ->
méthode de classe ->
méthode de super-classe

La possibilité déjà mentionnée de créer des objets sans définir de classe en XOtcl contribue à notre avis également à cette idée de programmation-prototypage.

Éditeurs de données : déboguage ou formulaire ?

La manipulabilité des données (visualisation et édition), considérée comme un élément important d'un langage visuel par [Mye90], est mise en oeuvre dans plusieurs systèmes, comme [Ibr98], [Boe86], [CHZ95], [Mye83] ou [Ede90].

Quand toutes les structures de données sont affichables et modifiables, on est certes dans un mode de fonctionnement de type débogueur, tel que celui décrit par [LF95]. Mais pour peu que la simple présence dans l'interface d'une boîte ou d'une feuille d'édition fasse office de déclaration de variable, c'est pourtant aussi le modèle d'accès du tableur ou du formulaire et ce qui en fait le succès [DA89][BAD+00][CUS95] (tout est interface et tout est paramètre).

3.3.4.3 Éditeurs graphiques spécifiques, formalismes visuels.

A mi-chemin entre le niveau d'édition du code et l'interface utilisateur, on peut concevoir des outils spécialisés à la fois pour l'*édition* et l'*utilisation* de certains objets. Ce type de développement n'est pas nécessaire pour toutes les formes de programmation, mais, s'il s'agit d'une fonction nécessitant une spécification procédurale, même partielle, et si c'est en même temps une véritable fonction de l'application, souvent utilisée, cela devient indispensable. Ce type de solutions pourrait s'apparenter à l'idée de formalismes visuels [Nar95] qui sont une forme de notation appropriée. Ils aident l'utilisateur à réaliser une tâche qui pourrait être effectuée dans une autre notation - écrire du code - mais constituent une notation plus effective.

Un exemple

Dans le prototype biok, l'éditeur de tags est un exemple d'un tel outil (un tag est un étiquettage graphique de valeurs associées à des positions dans l'alignement, valeurs différentes selon la fonction de visualisation et calculée par une méthode éditable par l'utilisateur - voir le chapitre V). Il s'agit en effet à la fois d'un éditeur, puisque le code d'une des méthodes est directement éditable, et de l'interface standard d'utilisation du tag.

3.3.5 Conclusion sur les niveaux d'interface.

Nous avons essayé de dégager plusieurs dimensions pertinentes pour articuler deux niveaux de représentation a priori orthogonaux : soit en explicitant cette différence de manière rationnelle, soit en établissant des degrés intermédiaires, soit en utilisant la structuration interne là où cela est utile pour l'incorporer dans l'interface utilisateur, ou enfin en ramenant la partie code source un peu plus près de l'interface utilisateur on lui apportant des propriétés d'utilisabilité ou de manipulation directe.

Nous avons exploré et implémenté un certain nombre de ces idées dans le prototype biok, en particulier celles qui consistent à utiliser le matériel statique et dynamique accessible par introspection, et

qui rendent l'environnement plus navigable. Nous avons également suivi l'idée de permettre un double accès graphique et programmatique aux éléments de l'interface. Enfin, les niveaux de programmation sont progressifs : ceux que l'utilisateur est amené à utiliser plus souvent sont aménagés pour une utilisation simplifiée : les formules sont le niveau le plus simple (pas de paramètre, facilités syntaxiques, historique), les tags viennent ensuite avec une seule méthode de format standard, puis les méthodes d'analyse de données, en principe de taille raisonnable. La programmation de l'interface et les changements de structure ne bénéficient d'aucun support particulier, puisqu'on peut considérer qu'à ce niveau les outils professionnels de développement sont connus de l'utilisateur.

3.4 Conclusion

Des solutions qui convergent

La figure 12 montre comment nous comprenons et prenons en compte l'ensemble des approches que nous avons analysées. Ce qui est mentionné en italiques, c'est ce qui selon nous leur manque pour s'appliquer à notre situation spécifique.

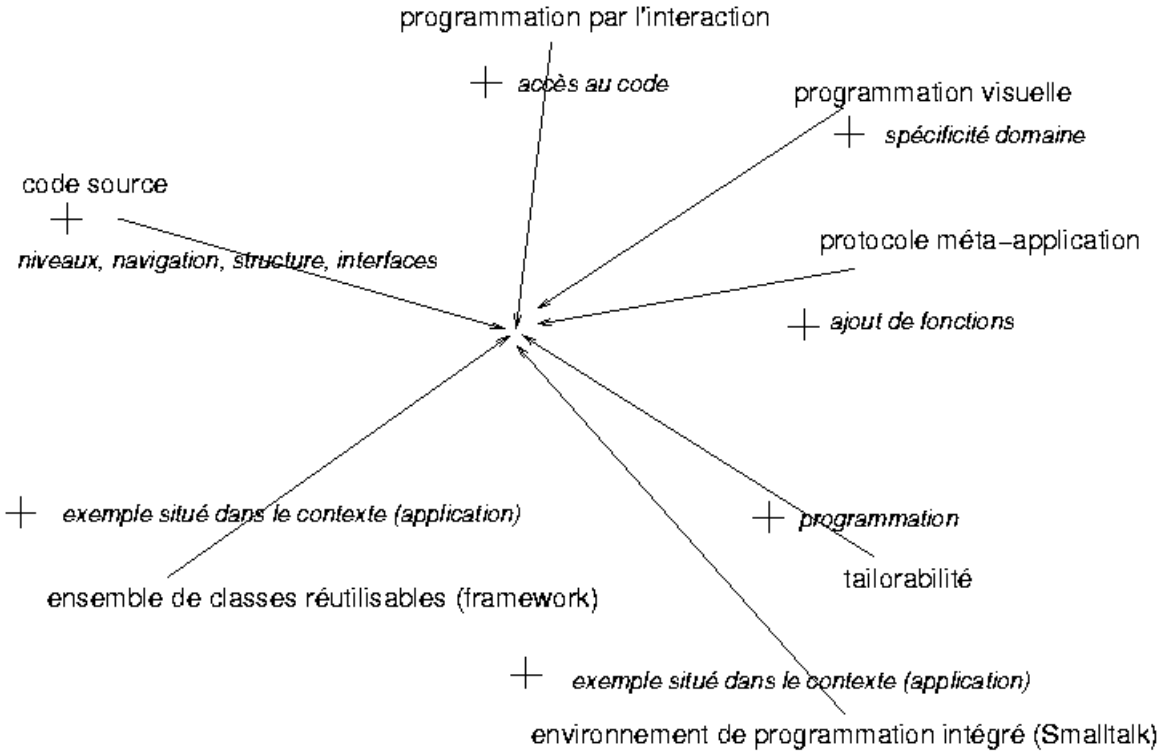


FIG. 3.12 – Convergence.

Programmation par l'utilisateur et flexibilité

On peut tenter de donner maintenant une vue de la programmation par l'utilisateur qui tienne compte des concepts abordés dans ce chapitre. La programmation par l'utilisateur consiste à :

- utiliser ce qu'on programme,
- programmer ce qu'on utilise [CIS92].

L'intérêt de cette définition est de s'affranchir de tout présupposé sur le "niveau" technique de l'utilisateur, et de toute limitation que cela peut induire quant aux possibilités de programmation. Nous avons préferré utiliser ce terme pour parler des niveaux d'interface, parmi lesquels l'utilisateur peut choisir celui qui convient à ses connaissances et à ses besoins. Le principe de réflexivité tel que nous l'avons utilisé conduit à une programmabilité totale du système. Ce qui "limite" le danger potentiel de cette ouverture, c'est le fait que si l'application est bien adaptée au travail des biologistes, ces

derniers, dans la plupart des cas, n'auront pas besoin de s'aventurer, ils feront simplement leur travail. Le prototype biok est effectivement conçu, grâce à la démarche participative, pour que les tâches courantes soient possibles sans programmation. L'accès au niveau de programmation ne se produit a priori que dans certains cas :

- lorsque la programmation est un objectif en soi, comme lors des stages de bio-informatiques (chapitre V) ;
- lorsque il y a une "panne" : une fonction ne marche pas, n'existe pas ou est mal adaptée ;
- lorsque l'utilisateur se perd dans les menus en cherchant quelque chose qu'il ne trouve pas : c'est aussi une panne, mais plutôt dans le sens d'une mauvaise utilisabilité.

Nous avons eu l'occasion d'observer, lors des stages de bio-informatique et des ateliers qui se sont déroulés ces derniers mois c'est-à-dire pour l'instant avec quatre ou cinq utilisateurs seulement, que les utilisateurs ne s'aventurent pas spontanément dans le code de l'application. Ils préfèrent explorer ce qu'ils ont l'impression de maîtriser.

Chapitre 4

Flexibilité en bio-informatique.

4.1 Introduction

Ce chapitre a pour objectif de recenser parmi les techniques et méthodes utilisées dans le domaine de la bio-informatique celles qui concernent notre problématique. Ce sont d'une part les outils, techniques et méthodes visant à améliorer la flexibilité des logiciels, et d'autre part les approches qui privilégient une forme d'interactivité où le biologiste et l'algorithme peuvent échanger des informations.

Outre une description de l'état de l'art concernant la flexibilité en bio-informatique, ce chapitre devrait également permettre de réfléchir sur ce que le contexte spécifique de la programmation par l'utilisateur dans un domaine de recherche peut apporter à la problématique générale de la programmabilité des logiciels et de la programmation par l'utilisateur final. En effet, à l'instar de [Dou96b] qui montre que la question de l'adaptabilité des logiciels se pose de manière spécifique dans le cadre de logiciels supportant la coopération entre plusieurs personnes, nous pouvons nous attendre à ce que la la programmabilité soulève des questions particulières dans un domaine de recherche scientifique. Ce chapitre a donc également pour but de montrer quels sont la place et la particularité potentielles du développement de méthodes informatiques par le biologiste lui-même dans le développement de logiciels pour la bio-informatique.

Ce chapitre est organisé de la manière suivante : une première grande section (4.2) répertorie les développements logiciels réalisés en bio-informatique pour parvenir à la constitution de ressources de type composants, considérés comme le meilleur moyen pour obtenir des logiciels flexibles (4.2.1) ; cette partie décrit également les problèmes d'intégration communément rencontrés dans le domaine (4.2.1 et 4.2.2) ; la deuxième grande partie (3) s'intéresse aux outils que les biologistes ont à leur disposition pour mieux contrôler leurs analyses et leurs données, allant des outils de visualisation interactive ou d'édition de résultats aux langages spécialisés, en passant par des macros pour tableur ou traitement de texte. Si les deux sections se recoupent, c'est parce que les solutions utilisées par les biologistes comprennent également l'utilisation de bibliothèques, et parce que les langages et les environnements de programmation utilisateur peuvent aussi être des composants. Mais les deux perspectives sont différentes, et il est nécessaire de les présenter, ne serait-ce que pour illustrer par des exemples pris dans la bio-informatique ce point de vue que nous avons développé tout au long de cette étude, selon lequel la flexibilité ce n'est pas seulement la possibilité de construire des outils avec des éléments de base (section 4.3.2) : c'est aussi l'interactivité dans le contrôle (section 4.3.3) et la possibilité de changer un logiciel (section 4.3.1).

4.1.1 Pourquoi les biologistes ont-ils besoin de flexibilité ?

Les raisons de poser la question de la flexibilité pour l'utilisateur dans le contexte de la biologie sont les suivantes :

- L'idée de l'évolutivité d'un programme n'est plus seulement une idée vague d'amélioration des fonctions et des qualités générales ; la description d'un programme comme une théorie empirique

et réfutable [Mør97b] [NDdM+94] est plus que pertinente dans le domaine de la recherche en biologie, en constante évolution.

- Les biologistes, forts de plusieurs années d'études et de plusieurs années d'expériences dans un domaine très savant et en continuelle expansion, ont bien plus de capacités que des informaticiens à formuler, spécifier, modéliser et *créer* leur problématique, parce qu'ils ont une idée de ce que la biologie peut leur donner comme informations : le principe de la division des tâches en fonction de la répartition des compétences semble ici pertinent [Fis99]. On entend souvent cet argument selon lequel il est plus facile pour un biologiste d'apprendre les concepts de base suffisant pour programmer (suffisamment logiques et généraux pour être applicables dans la plupart des contextes), que pour un informaticien de connaître assez de biologie (ce qui prend des années d'études). De fait, les bio-informaticiens sont plus souvent des biologistes d'origine, sauf ceux qui s'occupent de la création de bases de données et d'interfaces utilisateurs.
- Dans un domaine où le logiciel et les langages sont un moyen d'implémenter une idée et de la rendre active, il serait dommage d'abandonner sinon l'idée de programmation du moins celle de programmablité par découragemment devant l'inadaptation des outils existants. Il n'est pas possible pour chaque chercheur en biologie d'avoir un informaticien pour traduire ses pensées. Comme en tout, il faut penser dans le langage pour arriver à penser (argument de la programmation comme *medium d'expression* [Eis97]), phénomène souvent observé dans l'histoire des notations mathématiques [DiS99]. Dans le même ordre d'idée, les programmes constituent également un support de communication entre les scientifiques, dépositaire d'un savoir commun, comme par exemple dans la communauté des utilisateurs de Mathematica (argument de la programmation comme *medium de communication* [Eis97]).
- Une bonne mesure de la nécessité de flexibilité est donnée par un certain nombre d'études ou d'analyses qui montrent que les utilisateurs préfèrent souvent des logiciels génériques, même s'ils sont mal adaptés, à des logiciels orientés domaine, uniquement parce qu'ils sont plus souples et permettent de modéliser ce qui n'est pas prévu par les logiciels spécialisés [NJ94][DiG96b]. On verra par exemple que, dans la pratique de l'analyse de séquences biologiques, ce cas se produit avec les éditeurs graphiques spécialisés d'alignements de séquences auxquels les biologistes préfèrent bien souvent un traitement de texte.

4.2 Flexibilité logicielle

L'analyse des approches existantes concernant la flexibilité en bio-informatique (comme dans d'autres domaines) peut être abordée différement, selon que l'on choisisse le point de vue de la construction logicielle ou celui de l'utilisation. Dans le premier cas, on met plutôt l'accent sur les techniques d'intégration et les bibliothèques de composants en vue d'une construction de logiciel, dans l'autre, on s'intéresse aux fonctions documentées de l'interface utilisateur permettant à ce dernier d'intervenir par rapport au comportement du logiciel (re-conception, personnalisation, décomposition). Notre démarche est plutôt centrée sur le second aspect, mais nous allons toutefois évoquer aussi le premier aspect pour plusieurs raisons :

- afin d'une part de donner une idée des outils à la disposition de la communauté scientifique pour la construction des logiciels et du niveau de langage jugé utile dans la plupart des cas (quelles sont les primitives d'un langage bio-informatique) ;
- parce que la frontière entre ceux des biologistes qui voudraient juste utiliser ces composants (par exemple pour écrire des scripts "jetables") et les constructeurs d'outils n'est au fond pas si nette ; il arrive fréquemment qu'un script personnel devienne un outil partagé par des collègues, puis par des personnes ayant demandé un type de traitement similaire par l'intermédiaire des newsgroups ;
- dans le cadre de notre problématique, il est important de pouvoir disposer de composants bien faits pour construire rapidement des prototypes et pour les implémenter de manière souple ; le biologiste-utilisateur est le premier à bénéficier de l'existence de telles ressources si le ou les informaticiens avec lesquels il travaille peuvent rapidement prototyper une application pour en faire un support de spécification ;
- enfin, comme on l'a vu dans un autre chapitre, notre approche de programmabilité par l'utilisateur bénéficie pleinement de la conception de logiciels respectant les critères nécessaires à la

construction d'un composant et d'un framework.

Cette première section se propose donc de décrire les méthodes et les techniques utilisées en bio-informatique concernant les propriétés logicielles que sont à la fois la flexibilité interne et externe, c'est-à-dire les propriétés qui font d'un logiciel un objet générique, paramétrable et réutilisable.

4.2.1 Composants

4.2.1.1 Motivations

Il existe d'ores et déjà de nombreuses bibliothèques de composants dans plusieurs domaines de la biologie :

- en analyse et prédiction de structures de protéines, avec BALL [KL00], une bibliothèque de composants C++ pour l'analyse de structures moléculaires ;
- pour la manipulation d'entrées de banques de données, comme avec PDBlib [CSPB94], une bibliothèque de classes C++ pour manipuler les entrées de la banque PDB ;
- en analyse de séquences : SCL [VHKW96] propose des classes C++ pour traiter les formats, les collections de séquences, les alignements par paires ainsi que des classes génériques, tandis que bioperl [CFD+98][CFD+97][Che98] fournit des modules perl objet pour l'analyse de séquence, le chargement de fiches de banques, l'analyse et l'extraction de résultats ; SEQALN est une bibliothèque en C spécialisée dans les algorithmes d'alignement, et SEQIO une bibliothèque d'entrées-sorties pour les séquences, dont nous avons réalisé une interface en Tcl, et que nous utilisons dans le prototype biok ;
- dans le domaine des chemins métaboliques, comme avec ce système qui propose un modèle de bases de données et des composants [EM98] ;
- en génétique, en génomique et en génomique comparative, avec des composants graphiques de navigation [O'M98], ou des outils de simulations génétiques [CL99], pour n'en citer que quelques-uns.

Les arguments pour le développement de composants en bio-informatique, assez classiques, sont essentiellement, comme dans tous les domaines : la qualité du code, la construction d'autres composants à partir de composants existants [Bir98], la possibilité de construire rapidement une application pour avoir un retour rapide des utilisateurs [RF97]. [Bir98] mets en avant l'*utilité* du composant aussi importante que sa portabilité ou sa réutilisabilité (et cite à ce propos les composants réutilisables faits par des fanatiques du composant mais qui ne sont ... jamais utilisés). Parmi les alternatives aux solutions à base de composants, la plus pratiquée est celle qui consiste à tout refaire soi-même. L'approche d'adaptation de composants, utilisée lorsqu'il faut un peu de code supplémentaire pour réaliser l'adaptation, est parfois évoquée - mais comme étant limitée [GRS95a] ou même à proscrire [BLL+99]. [GRS95b] insiste sur l'importance des standards et la promotion de composants de niveau scientifique : un des problèmes des composants serait que leur production ne donne lieu à aucune reconnaissance scientifique. D'ailleurs, comme l'explique [PRT00], c'est justement parce que les tâches intermédiaires et fastidieuses de nettoyage de banques de données de séquences (erreurs et redondance) ne sont jamais publiés et sont pour cette raison ré-implémentées dans plusieurs centres bio-informatiques, qu'il est utile d'en faire des composants réutilisables.

Il existe d'autres formes de réutilisation très pratiquées, comme l'utilisation de programmes exécutables externes accessibles depuis l'interface utilisateur [SOW+94] ou de services réseaux, mais, sans doute parce qu'il ne s'agit pas de composants statiques logiciels, ces formes d'assemblage ne sont pas souvent citées comme faisant partie de l'ingénierie du composant.

4.2.1.2 Qu'est-ce qu'un composant ?

Si les avantages du développement de l'ingénierie du composant en bio-informatique ne font pas de doute, la définition de ce qu'est un composant donne cependant lieu à certaines variations. Ainsi, pour [GRS95a] un composant est un sous-système significatif, comme le sont par exemple, typiquement :

- une base de données (ou des fichiers simples, des données),

- un programme d'analyse biologique,
- une interface utilisateur, ou tout service de présentation comme un "report writer", ou un tableur,
- une passerelle vers des services externes.

Pour [Bir98], un composant se définit de manière simple comme étant *un morceau de code réutilisable* (une fonction, un script, ...). Dans le système Piper [Biz00], qui est un environnement visuel de composition d'objets développé dans le contexte de la bio-infomatique mais se voulant général, les objets composables peuvent-être un document, une application, un widget, un convertisseur de format, un "viewer", une "décision", c'est-à-dire une structure de contrôle, un terminateur (bouchon pour arrêter un workflow), ou un objet composite.

Ces trois définitions ne sont pas incompatibles, mais l'imprécision ou la diversité qui en résulte nous montre qu'il est utile de suivre la démarche de [ND95] selon lequel il faut distinguer les aspects méthodologiques des aspects techniques. Ainsi [Bir98] insiste sur le fait qu'un composant n'a ni à être écrit dans un langage particulier, ni à correspondre à un paradigme de programmation particulier. Cela peut-être aussi bien un module, une classe, ou un programme exécutable (même si ce n'est pas idéal) comme GDE [SOW+94]. Du point de vue méthodologique, un composant est simplement quelque chose qui n'est pas conçu isolément, mais pour faire partie d'un framework de composants qui collaborent. Il encapsule n'importe quelle abstraction informatique. Ce qui fait la définition d'un composant est par conséquent non pas sa nature en tant qu'élément logiciel, mais le fait qu'il interagisse dans un ensemble. Ainsi, une spécification, comme le modèle de données en ASN.1 du NCBI [Ost95], une documentation, comme celle qui est distribuée avec les modules bioperl [CFD+98][CFD+97][Che98], des données de tests ou des exemples d'applications comme celles qui sont distribuées avec la bibliothèque SEQIO, peuvent constituer des composants, comme le sont également des mixins, des templates ou des macros [ND95].

4.2.1.3 Interfaces entre composants

Ce qui fait la difficulté de l'écriture de composants en général c'est la spécification de leur interface ou plus généralement la mise en place de ce qui permet à ces composants d'interagir, car celle-ci dépend souvent d'une coordination entre des concepteurs, et doit s'intégrer dans un processus de développement logiciel dont l'importance est parfois minimisée et pour lequel il n'existe pas beaucoup d'environnement de support technique. Le projet bioperl, un des plus actifs en bio-informatique, fait intervenir un grand nombre de développeurs partout dans le monde, et, à titre illustratif des processus de développement dans le milieu scientifique, est essentiellement supporté techniquement par une liste de courrier, un site Web (`http://bio.perl.org`) comprenant quelques pages wiki-web, et le système d'archivage CVS.

La mise en place des interfaces des composants comprend d'une part le *procédé de liaison* entre composants et d'autre part la *modélisation et la représentation des données* partagées entre les composants. Le premier type de problème n'est pas particulièrement spécifique à la bio-informatique, mais il est néanmoins important, puisqu'il implique des choix concernant le type technique du composant. Le deuxième point repose sur l'idée que les données forment cette part "non syntaxique" de l'interface des composants logiciels. Cet aspect est à la fois très important et particulièrement peu simple en bio-informatique. C'est en effet un problème rendu difficile par la nature des données biologiques : elles sont abondantes, dans un sens quantitatif mais surtout dans le sens où il y en a beaucoup de natures différentes, chaque type de donnée étant reliée aux autres selon un schéma complexe [AVB01].

[GRS95b] résume bien la situation : les standards doivent porter essentiellement sur : la représentation des données, la communication entre programmes, les systèmes de bases de données (DBMS), les interfaces graphiques.

Nous allons d'abord décrire brièvement les problèmes liés à la représentation des données en bio-informatique, puis nous donnerons un aperçu des types techniques de composant et des solutions existantes dans le domaine pour leur assemblage.

4.2.1.4 Représentation des données

Il faut tout d'abord distinguer les problèmes de modélisation et de représentation des données, les premiers étant plus précisément abordés dans la section qui présente les aspects sémantiques. Pour ce qui concerne la représentation des données, les questions importantes sont :

- la *mise en place de standards* , portant selon [GRS95a] sur des choix de représentation de bas niveau et des protocoles, avec une réflexion sur l'extensibilité de l'ensemble des types d'objets biologiques et des positions de replis si le logiciel client ne sait pas les traiter ou les afficher ; un problème connexe mais très sensible sur le plan pratique est celui de la *robustesse* des composants par rapport aux problèmes de représentation et de format des données [KL00][PHB98] : sans doute considéré comme trivial et sans intérêt tant du point de vue informatique que du point de vue scientifique, ce problème est le cauchemar quotidien des biologistes analysant leurs séquences.
- l'*intégration des données entre elles* (navigation, regroupements) :
  - selon [BLL+99], le problème posé est celui des inter-relations entre les données, matérialisées par des relations biologiques comme l'orthologie, le "gene mapping", la régulation de gènes : pour découvrir toutes ces relations, manipuler ces données, il faut des outils standards, ainsi qu'une vue unique et opérationnelle ; dans l'état actuel, il faut encore intégrer des données de nature différente (cartographie, séquence, fonction, protéine, chemins métabolique), de différentes espèces biologiques, et de sources différentes - avec des bases de données qui ont des formats et des schémas différents ;
  - [PRT00] confirme la nécessité d'outils implémentant ces relations ; il faut pouvoir accèder aux données par fonction, phylogénie, position cartographique, localisation cellulaire, régulation d'expression, maladie associée au gène, similarité d'annotation et chemin métabolique ;
- la *fragmentation des données* : selon [STM98], le problème qui se pose souvent avec les bases de données relationnelles, et qui fait qu'elles ne peuvent constituer une solution autonome à la manipulation des données génomiques, est que, pour des raisons de performance, les données doivent être fragmentées dans plusieurs enregistrements ou plusieurs tables ; cela rend leur utilisation directe par le biologiste difficile ;
- la *représentation évolutive* : c'est une idée similaire à celle de "content scalability" [Let95], sur le plan des choix de représentation ; cela peut être réalisé par des formats auto-descriptifs, ou par la définition d'interfaces, à condition que l'interface puisse évoluer.

Parmi les représentations ou les systèmes d'interfaces de données qui font l'objet de travaux actuellement, on peut citer essentiellement :

- Corba : cette architecture est souvent considérée comme la solution technique parfaite pour les problèmes d'intégration, de répartition des données sur des sites différents, d'hétérogénéité des systèmes et des langages, mais aussi de modélisation grâce à la possibilité techniquement supportée de définir des interfaces explicites entre les objets. Dans Piper [Biz00], Corba est utilisé pour la communication client-serveur et pour la communication entre les couches du système. Ainsi, le système décrit dans [Mun98] facilite la communication client-objet par un pont CORBA dans la couche des objets domaine d'une base de données de cartographie génétique. Dans la même idée, [O'M98] utilise Corba pour canaliser les données depuis les bases des données d'espèces biologiques vers des outils graphiques à travers l'Internet. Un dernier exemple est celui de JESAM [PRT00] où Corba intervient dans une architecture répartie comme serveur des alignements préalablement calculés sur un réseau de PCs liés en PVM, et aussi comme serveur pour fournir des cluster (groupes classés par ressemblance) calculés à partir de ces alignements. Corba est donc utilisé ici pour structurer un calcul réparti. D'après les auteurs, le bilan de l'utilisation de Corba est d'une part que cette technologie nécessite plus d'expertise informatique et de bibliothèques à installer, mais qu'elle est très utile pour la navigation intégrée d'une information à l'autre. Le problème de l'utilisation de Corba avec l'Internet, dû au fait que le protocole IIOP ne passe pas les coupe-feus, paraît gênant, ainsi que le problème de signature multi-applets, nécessaire pour accéder à des données réparties entre plusieurs serveurs, puisque certaines ne seront pas à la même adresse que le fournisseur de l'applet. Mais [BLL+99] critique l'argument fréquement invoqué contre le choix Corba selon lequel ses performances seraient mauvaises : ce n'est pas selon cet article le problème prioritaire. Ce qui est le plus important, c'est de créer des méthodes flexibles, confortables et complètes d'accès aux données et aussi d'avoir un bon niveau d'expressivité et

une bonne extensibilité en échelle (scalability).

- ASN.1 est le standard de représentation de l'un des plus gros centre bio-informatiques mondiaux, le NCBI, qui est le producteur de la banque Genbank, une des principales banques de séquences nucléiques de la planète. Le NCBI favorise donc bien sûr l'utilisation de ce format de représentation, qui comme XML a l'avantage d'être auto-descriptif, mais qui, à l'inverse de XML, a l'inconvénient majeur d'être sous forme binaire, donc non lisible et non manipulable autrement que par des bibliothèques logicielles fournies par le NCBI.
- XML est de plus en plus utilisé en biologie, que ce soit pour modéliser les objets biologiques : CML pour la chimie, BSML (Bioinformatic Sequence Markup Language), BioML (Biopolymer Markup Language), GAME (Genome Annotation Markup Elements), ou pour produire des résultats de calcul analysables par la machine [Let00b] ou pour spécifier des interface de programmes [Let00b][Biz00] [Cha00]. D'après [Bir98], XML a l'avantage d'être simple, sans état, et d'accompagner l'objet (comme ASN1, mais en texte). Parmi les solutions distribuées, XML est plus réaliste que Corba du point de vue des performances. [AVB01] remarque que XML peut modéliser les liens (XLL), qu'il est facile à manipuler : on peut changer un modèle en "mode texte" ("content scalability") et qu'il constitue un support de standardisation. Mais XML ne résoud pas le problème de l'accès rapide aux données et de l'indexation.

Corba et XML ne sont bien sûr pas des techniques équivalentes, puisque CORBA définit un protocole de communication entre objets avant de définir une représentation ; notre mise en parallèle se limite donc à l'aspect de spécification d'interfaces entre objets et de support technique pour la réaliser.

Analyse de texte

Moins lourde de conséquences que les décisions concernant les représentations des données en biologie, la question de la structure et de la forme des résultats des programmes d'analyse est cependant assez importante pour faire souvent l'objet d'un traitement particulier dans l'industrie du composant en bio-informatique. Ainsi, plusieurs outils proposent un format de sortie tabulé (tbob, ...) et, plus récemment, un résultat sous forme XML : le logiciel readseq de D. Gilbert, un des plus utilisés pour les conversions de format, propose depuis quelque temps une sortie XML, le script gb2xml, écrit par P. Bouige, bio-informaticien à l'Institut Pasteur, convertit une fiche Genbank en XML, le logiciel Blast, pourtant développé au NCBI, bastion de la technologie "concurrente" ASN.1, propose maintenant une sortie XML.

Il arrive souvent que les résultats de programmes deviennent des données, de manière plus ou moins directe. Ces résultats sont dans la très grande majorité des cas sous forme de texte, et leur utilisation comme donnée suppose parfois un travail d'analyse et d'extraction. Par exemple, ProSearch [KLS92] permet d'identifier dans des séquences des motifs connus, répertoriés dans la banque Prosite. Le résultat est une liste de références dans cette banque et de positions dans la séquence fournie en entrée par l'utilisateur :

```
Access#     From->To                    Name    Documentation#
-------     --------                    ----    --------------
PS00005       39->41        PKC_PHOSPHO_SITE    PDOC00005
        Pattern [ST].[RK] matched
        Site    39 TTR 41
PS00005       71->73        PKC_PHOSPHO_SITE    PDOC00005
        Pattern [ST].[RK] matched
        Site    71 TSK 73
PS00005       84->86        PKC_PHOSPHO_SITE    PDOC00005
        Pattern [ST].[RK] matched
        Site    84 STK 86
PS00005     136->138        PKC_PHOSPHO_SITE    PDOC00005
        Pattern [ST].[RK] matched
        Site    136 TCK 138
PS00006        8->11        CK2_PHOSPHO_SITE    PDOC00006
        Pattern [ST]..[DE] matched
        Site    8 TLIE 11
```

```
PS00006       51->54        CK2_PHOSPHO_SITE    PDOC00006
        Pattern [ST]..[DE] matched
        Site    51 TAVD 54
PS00006       71->74        CK2_PHOSPHO_SITE    PDOC00006
        Pattern [ST]..[DE] matched
        Site    71 TSKE 74
PS00006       79->82        CK2_PHOSPHO_SITE    PDOC00006
        Pattern [ST]..[DE] matched
        Site    79 TSSE 82
PS00008       20->25                MYRISTYL    PDOC00008
        Pattern G[^EDRKHPFYW]..[STAGCN][^P] matched
        Site    20 GILAAI 25
PS00008       77->82                MYRISTYL    PDOC00008
        Pattern G[^EDRKHPFYW]..[STAGCN][^P] matched
        Site    77 GLTSSE 82
```

On constate aisément dans cet exemple que les positions des occurrences, le motif lui-même ou les identifiants des motifs trouvés sont réutilisables pour effecteur d'autres traitements, comme la recherche de toutes les protéines possédant tel ou tel motif, puis leur comparaison ou leur utilisation pour d'autres analyses, etc... Les "données-résultats" sont ainsi souvent relatives ou similaires aux "données-entrées", et peuvent constituer à leur tour des "données-entrées", avec parfois un peu d'extraction de texte.

Ainsi [HFA99] propose des modules perl pour analyser efficacement et avec robustesse des entrées de la banque de protéines Swissprot, [RGG00] est un module écrit en Python pour analyser les banques de séquences et SPEM [PHB98] fournit des routines d'analyse d'entrées de la banque EMBL.

4.2.1.5 Liaisons entre composants

Concernant la composition, comme nous l'avons dit plus haut, il y a deux aspects qui sont à considérer : un aspect méthodologique qui s'intéresse à la conception des composants, c'est-à-dire au découpage qui détermine leur granularité et leur fonction, et un aspect technique qui porte sur la nature physique du composant et qui est déterminant quant à l'utilisabilité du composant dans un contexte ou dans un autre. Après avoir traité ces deux aspects, comme le prototype que nous avons réalisé repose pour une grande part sur ces techniques, nous décrivons brièvement quelques composants s'appuyant sur les technologies et conception par objets dans le domaine de la bio-informatique. Enfin, nous posons la question de l'intégration *technique* souvent déclarée incontournable par les auteurs de logiciels en bio-informatique, et nous défendons une approche plus ouverte.

Découpage et granularité

Les mêmes principes s'appliquent en bio-informatique qu'ailleurs : une bonne granularité permet au composant d'être utilisable dans plus de contextes différents - du moins si l'on entend par granularité non pas la taille physique (octets, lignes de code, ...) mais la taille en nombre de fonctions différentes. La propriété d'indépendance entre les composants est également invoquée par [SFT+01] : elle permet aux composants du système ISYS, un système d'intégration de ressources bio-informatiques, de pouvoir interagir même s'ils ne se connaissent pas d'avance et sont développés indépendamment. [Bir98] prone également cette propriété classique de couplage faible.

Types physiques de composants

Le type physique approprié, c'est-à-dire en fait le niveau de granularité pratique, est l'objet d'un certain débat. [Bir98] défend l'idée qu'un module est plus souple et plus évolutif qu'un programme exécutable, qui est plus monolithique et moins réutilisable. Mais [SOW+94] soutient qu'il vaut quand même mieux faire appel à des programmes externes déjà existants que d'attendre l'intégration des algorithmes qu'ils implémentent dans une bibliothèque logicielle ou un progiciel commercial comme GCG, avec parfois la nécessité de tout retraduire dans le langage ou l'architecture de ce logiciel. Nous pourrions même aller plus loin en disant que la réutilisation la plus économique et la plus synchronisée est celle qui consiste à utiliser un service déjà présent sur le réseau sous la forme d'un serveur Web ou d'un service CORBA. C'est cette forme de réutilisation qui est majoritairement pratiquée sous

une forme non programmatique par les biologistes aujourd'hui, et l'on pourrait s'en inspirer plus pour accentuer les diverses formes de réutilisation, du moins si l'on étend la notion de réutilisabilité au-delà de la simple notion d'intégration au niveau d'une bibliothèque. Ainsi, [GRS95a] invoque parmi les aspects de l'ingénierie des composants sur lesquels trouver un consensus le transfert d'informations par messages électronique ou par le Web, placé dans la continuité de l'interopérabilité entre systèmes d'exploitation. Une autre possibilité de réutilisation est d'utiliser du code envoyé par un serveur de code dynamique - comme les applets. Cette dernière solution a un double avantage pour le biologiste : elle permet de ne pas envoyer ses données confidentielles sur un site externe, tout en lui évitant d'avoir à installer le logiciel ou le composant chez lui.

Interfaces programmatiques (API)

Il est également intéressant de mentionner une approche qui n'est pas techniquement (assez) courante en bio-informatique pour réutiliser des programmes, mais qui, dans le contexte de la bio-informatique où la technologie et la production de composants ne sont pas encore mûres, peut apporter des solutions intermédiaires très utiles.

Ce qui compte pour beaucoup dans l'intérêt de la technologie des composants, c'est la possibilité d'utiliser du logiciel au niveau de granularité d'un langage de programmation : c'est ce qui permet d'en faire un usage automatisé et répétitif sans avoir à manipuler l'interface de commande du système. Ainsi, ce qui est très séduisant dans Corba, c'est d'une part cette possibilité d'accéder à des objets répartis sur des plate-formes différentes et implémentés dans des langages différents, mais surtout à travers une interface réellement programmatique : un objet avec des méthodes.

Une des approches possible pour parvenir à un résultat similaire à celui de la technologie des composants est de prévoir, éventuellement après coup, une interface programmatique dans les applications. Ainsi, pour des systèmes qui existaient de manière plus "monolithique" comme Acedb [DTM91] ont s'est aperçu qu'il serait utile d'avoir un niveau de manipulabilité plus puissant pour les requêtes ad-hoc, soit complexes soit répétitives, c'est-à-dire une interface programmatique [STM98][SCTMTM98]. C'est d'ailleurs de manière générale une bonne idée de prévoir plusieurs niveaux de manipulabilité, aussi bien pour une application que pour un serveur Web : les propriétés de scriptabilité et de parsabilité sont particulièrement appréciées pour des analyses non prévues d'avance, et sont une réelle forme de manipulabilité et d'utilisabilité. Parmi les système proposant une telle interface, on peut citer :

- [CFGO98] qui définit une API java et un modèle de données pour s'intégrer à d'autres applications ;
- SSAHA [NCM01] : un algorithme d'alignement disponible à la fois comme application et comme une bibliothèque ;
- [Cha98] : une API JAVA pour accéder à la banque IMGT/LIGM-DB ;
- [GGRK98] : une interface visuelle reposant sur une API Prolog et un langage de requêtes (Daplex) pour une banque de structures (P/FDM).

De manière générale, comme on a pu le constater, il ne semble pas utile d'en rester à des notions classiques (et techniques) de composants statiques de bibliothèque pour décrire ce qui se fait effectivement en bio-informatique en terme de fabrication et d'utilisation de composants. [ND95], à propos de la distinction entre computationnel (ce qui fait un calcul) et compositionnel (ce qui constitue une partie d'une application), suggère de ne pas systématiquement assimiler les entités compositionelles avec l'étape de composition et de construction statique. Il existe bien sur des possibilités de liaison dynamique à l'exécution dont l'activation d'un service réseau ou l'utilisation d'une interface programmatique ne sont guère plus que des extensions délocalisées.

Langages orientés objets en bio-informatique.

De très nombreux systèmes sont développés en technologie objet dans le domaine de la bio-informatique, qu'il s'agisse d'applications complètes comme [PD00] pour un éditeur d'alignement, de bases de données objet [KM98], de modèles de données [BLL+99] ou de bibliothèques de composants comme BALL [KL00], PDBlib [CSPB94], [EM98] ou SCL [VHKW96]. Les langages supports varient de Java à Python [RGG00] en passant par C++ ou [incr Tcl].

Comme l'observent très justement [Bir98] ou [ND95], la technologie des composants n'est pas complètement recouverte par la technologie objet. L'idée de la programmation orienté-objet de masquer l'implémentation par une interface est très utile, mais cela n'est pas suffisant pour penser les com-

posants car un composant encapsule n'importe quelle abstraction informatique, et sa granularité est d'ailleurs soit plus fine soit plus grosse que celle d'une classe. De même, encore selon [Bir98], qui tente de convaincre les réticents à l'ingénierie des composants de ne pas trop être influencés par les aspects techniques, objet ne veut pas nécessairement dire héritage, polymorphisme, etc... et l'objet n'est pas un but en soi. Pour [MRDV99], un des intérêts de l'approche objet est l'isomorphisme avec les objets de l'interface, qui permet selon les auteurs à l'utilisateur de manipuler les représentations à travers les objets de l'interface.

Plug&play ou adaptation ?

Pour certains auteurs de logiciels comme [CFGO98], l'intégration des composants doit reposer sur une intégration technique assez forte. Certains, par exemple, voient dans les JavaBeans une solution à tous les problèmes de la fabrication des composants, sans penser au fait que cela suppose un langage unique de programmation qui ne convient pas dans toutes les situations - par exemple pour prendre en compte les logiciels existants - ainsi qu'un modèle unique de communication par évènements. Plusieurs systèmes ont déjà fait ce choix technique. Dans ISYS [SFT+01], les composants graphiques sont synchronisables par un protocole de type abonnement à des évènements (Listener). Les JavaBeans sont également proposés comme standard de communication entre composants dans [KL98] et [Boy98a]. Même si la liaison de composants graphiques pose un problème un peu particulier, c'est la solution intégration forte contre la souplesse et l'économie en terme de réécriture de composants.

Il nous semble d'une part que d'autres solutions techniques existent, qu'elles sont plus ouvertes (aussi bien au sens topologique qu'au sens de l'hétérogénéité des plate-formes ou de l'évolutivité [ND95]) et sont d'ailleurs déjà utilisées en bio-informatique : les passerelles entre langages, les interfaces programmatiques d'exécutables et les services réseau à sortie grammaticalement structurée (figure 1).

D'autre part, même si les composants s'intègrent dans une architecture souple et à plusieurs niveaux, il reste un problème que l'on rencontre souvent lorsqu'on gère les logiciels scientifiques d'un centre informatique en biologie : dans le monde réel, rares sont les logiciels qui s'intègrent et plus rares encore sont les composants adaptables. Très souvent, il faut écrire des adaptateurs, des convertisseurs, soit de données soit de résultats de programmes, afin de permettre des analyses plus complexes. Il serait dommage de ne pas tenir compte de l'aspect inévitable de cette situation dans le domaine de la recherche académique : il existe de très nombreux outils qui implémentent des algorithmes très astucieux et très efficaces mais qui, faut d'accessibilité technique, ne sont jamais utilisés. Les composants "existent" déjà sous diverses formes, et tout en accélérant le développement de composants sous une forme directement utilisable, il semble raisonnable et réaliste de développer des techniques d'adaptation, car les biologistes ont besoin de pouvoir faire leur analyse quelque soit l'état de qualité logicielle de ces objets. On peut même anticiper le fait qu'étant donné le statut fréquent de mise en oeuvre provisoire et jetable d'hypothèses qu'ont les logiciels dans la recherche en bio-informatique, cette situation ne semble pas prête de disparaître définitivement.

La flexibilité des composants passe donc à notre avis par le support des outils d'adaptation, c'est-à-dire plus précisément les *langages de script* et les outils de conversion de format. Sur le plan de la flexibilité, le scripting est une façon, puisqu'il s'agit de programmation au sens large du terme, de *laisser la place à l'imprévu* , là où le plus souvent la liaison de composants échoue (figure 1). Comme le dit [NDdM+94], l'hypothèse d'applications stables en univers fermé ne tient plus ; il faut par exemple combiner les approches objets et 4GL, ou scripting, en tous cas plus orientés utilisateurs.

L'idée que le scripting étend les possibilités de construction d'application que les langages objets ne suffisent pas à assurer est expliquée par exemple dans [AN00], pour qui une application est faite de composants et de scripts pour les assembler, ou dans [GNZ00] pour lequel il faut combiner composants et scripting (avec éventuellement de l'introspection) pour faire une application flexible. C'est aussi une critique des composants boîte-noires, qui s'ils ne marchent pas, ne sont pas réparables, et qui sont difficiles à faire évoluer. En résumé, pour la composition comme pour les autres techniques de programmation, le paramétrage est une technique insuffisante. [NDdM+94] propose un modèle visuel de scripting de type tableur, où le script joue le rôle de formule.

Ce qui nous paraît intéressant, c'est d'une part que le scripting est une technique qui par divers aspects est accessible à l'utilisateur averti, surtout si le langage de script possède des extensions pour le domaine d'application (comme le conseille aussi [Dub99]). Cette question laisse il est vrai place à

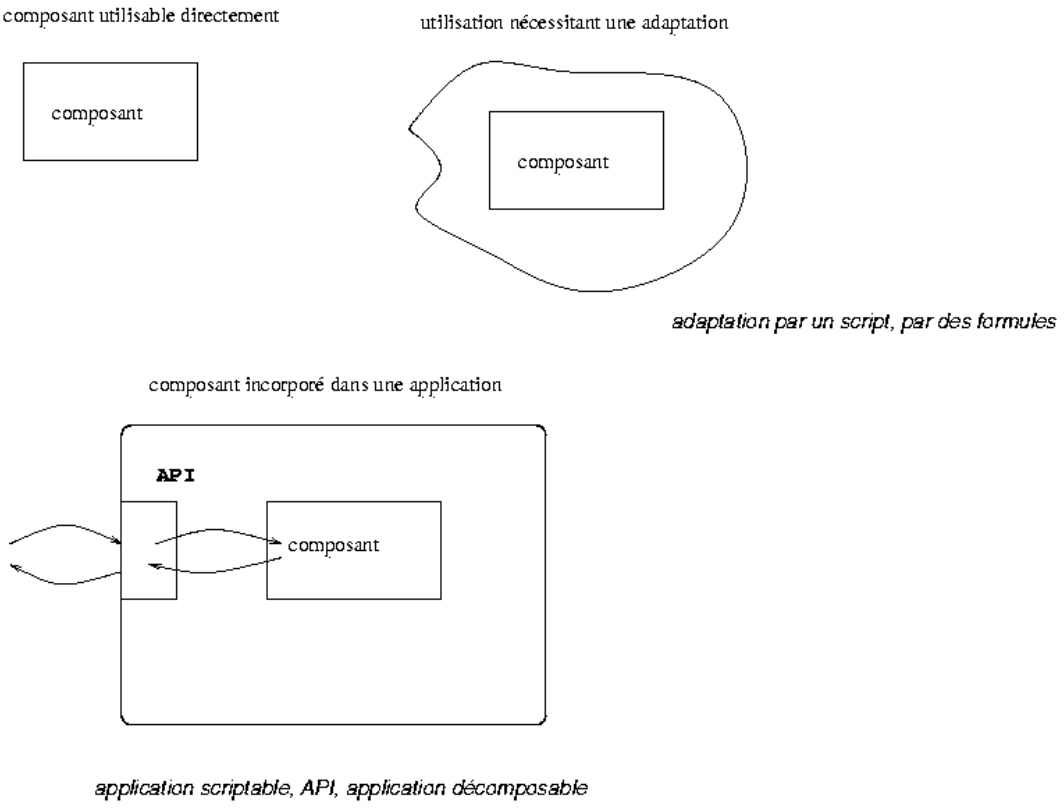


FIG. 4.1 – Flexibilité et composants

discussion, comme pour [SN99] pour qui les langages de colle sont *trop orientés domaines* et pas assez formalisés, et qui propose la réflexion comportementale comme alternative aux adaptateurs ("wrappers") pour l'extensibilité et la reconfiguration dynamique. Dans le même ordre d'idées, [Now97] critique le fait que l'architecture soit trop souvent implicite : il faudrait faire des éléments d'architecture des citoyens de première classe.

4.2.1.6 Sémantique des composants

Nous plaçons dans cette section la description des cadres dans lesquels l'utilisation de composants prend son sens, tel qu'il se produit dans le domaine de la bio-informatique.

Couplage avec le domaine

C'est d'abord la question de la modélisation des données qui doit être abordée. Cette question relève surtout de décisions scientifiques qui sont par exemple traitées par le groupe LSR (Life Sciences Research) de l'OMG (Object Management Group), ainsi que par les producteurs de grandes banques de données comme l'EBI (European Bioinformatics Institute) ou le NCBI (National Center for Biotechnology Information). En attendant, [GRS95a] propose un consensus sur les types de données de base :

- séquence (DNA, protéine, raw, finished), cartes de toutes sortes,
- données de type "laboratory workflow",
- gènes, allèles, phénotypes,
- résumés d'articles, documentation scientifique.

tandis que [SFT+01] choisit un modèle de données à base d'objets biologiques stables :

- séquences d'ADN et de protéine,
- gènes,
- enzymes,
- chemins métaboliques,
- cartes,
- éléments cartographiables,
- désignations taxonomiques.

Mais pour des auteurs de logiciels, il s'agit d'une part de décider de l'isomorphisme entre les objets du programme et les objets biologiques [KL00][FCB+99] et d'autre part de déterminer à quel point la sémantique des objets doit entrer dans l'implémentation. En général, la tendance est plutôt vers la généricité voire la neutralité de l'implémentation par rapport à la biologie. Ainsi, [Sea95] et [FCB+99], auteurs de composants graphiques pour l'analyse de séquences, promeuvent l'idée de neutralité, qui consiste en l'indépendance entre le langage d'utilisation du composant et l'objet biologique. Les fonctions proposées par leurs composants sont donc très générales et portent par exemple sur les propriétés de sélectivité ou de visibilité. L'argument est de dire que les données biologiques, surtout en génétique et en biologie moléculaire, sont évolutives, et qu'il vaut mieux proposer une interface générique qui fonctionne de la même façon pour tous ces objets. D'ailleurs, le choix de données stables et d'un schéma simple et générique, avec des objets de granularité assez élevée [SFT+01][BLL+99] vont bien dans le sens de la fléxibilité. C'est la même idée qu'on trouve dans [KL00] : des classes sont prévues pour les objets de base comme les atomes ou les molécules, avec des schémas récursifs composites pour certains des objets biologiques comme les protéines, chaines, résidus et atomes. On retrouve cette idée de récursivité dans [BLL+99] pour le schéma de classes de bases de données cartographiques : il est récursif afin d'autoriser un niveau quelconque de description.

Une autre idée récurrente est la séparation entre données et fonctions, comme dans [BLL+99] où les cartes ne contiennent que les informations de localisation, déportant les autres informations sur les objets eux-mêmes, ou dans [KL00] qui préfère implémenter une méthode "apply" (start/operator/finish) pour éviter d'encombrer l'interface des objets avec de nombreuses méthodes.

Classification des composants

Il est aussi intéressant de modéliser la syntaxe et la sémantique du "langage" d'utilisation des composants que les composants eux-mêmes. Cela permet de leur donner un profil général, de raisonner globalement sur leur interface, de déterminer leur granularité, dans un contexte particulier, la biologie. C'est aussi nécessairement dans ce cadre conceptuel que doivent se situer les critères déterminants pour un système ouvert à l'utilisateur. Sur ce plan, l'état de l'art, tout autant que l'observation des pratiques, fournit des informations importantes, mais ce sont des synthèses comme [SGB+01] qui permettent un raisonnement générique sur les composants.

Dans cette classification des tâches un programme, un composant est modélisé génériquement comme une *requête* , c'est-a-dire comme une phrase ayant une syntaxe. La figure 2 illustre ce modèle : les composants sont classés selon qu'ils sont plutôt des collections, que ce soit en entrée ou en sortie (**séquences** , **banques** , **critères** , **alignements** , **motifs conservés**), des transformateurs (programme d'**alignement multiple** , de **phylogénie**), des filtres (**recherche dans une banque**), des filtres-transformateurs ou des sélecteurs (comme la fourche parallèle). La figure 2 illustre cette classification : 3 collections (des séquences, une banque et des critères de recherche) figurent en entrée d'un filtre transformateur, le tout constituant une recherche de ces séquences dans la banque en respectant certains critères. Le résultat est une collection (des séquences), en entrée d'un transformateur qui va les aligner, et produire une nouvelle collection, l'alignement. Cet alignement est ensuite en entrée d'une bifurcation parallèle, permettant de calculer une phylogénie ou une extraction de motifs.

Ce qui peut être éventuellement intéressant ici, c'est l'utilisation de ce cadre syntaxique et sémantique pour la construction d'un langage utilisateur.

4.2.1.7 Exemple : composants graphiques.

Il y a une demande de composants graphiques proportionnelle à l'importance de la visualisation de données et à la nécessité d'outils interactifs pour les analyses biologiques (nous en parlons plus dans la

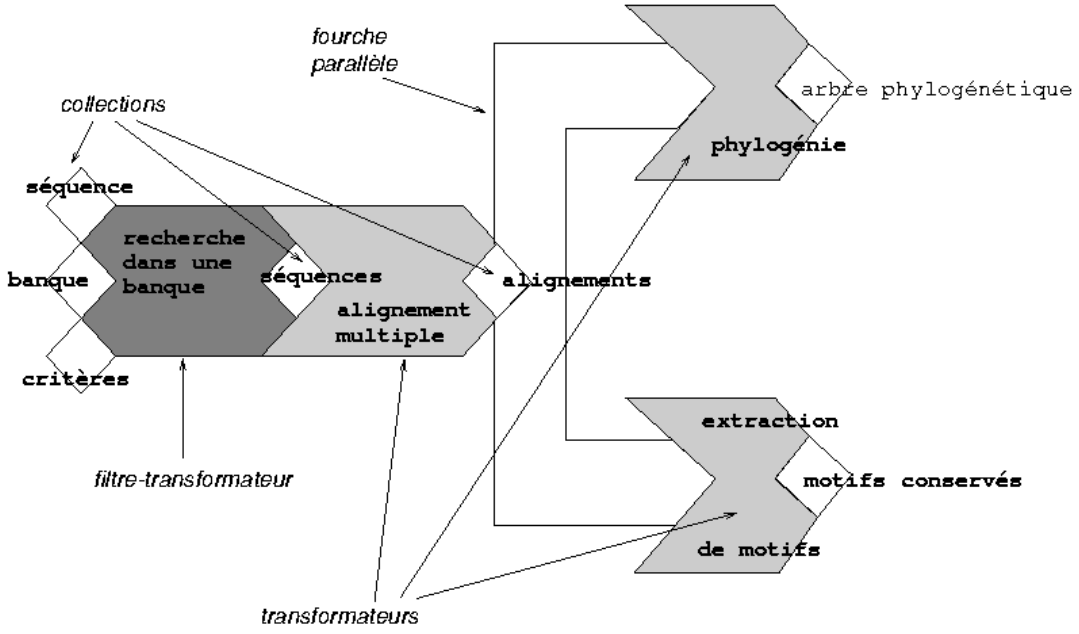


FIG. 4.2 – Classification des tâches

deuxième partie du chapitre). Les exemples sont très nombreux :

- ISYS [SFT+01] est un système permettant la navigation entre des données de nature et d'échelle différente, par exemple : depuis la carte génomique, vers les séquences annotées correspondantes, puis par similarités successives, vers les chemins métaboliques.
- bioTk [Sea95] est un ensemble de widgets écrits en Tcl/Tk (non maintenu) qui permet d'afficher des informations sur des séquences, des cartes génétiques ou des chromosomes.
- Les bioWidgets [CFGO98][FCB+99] reposant sur une architecture JavaBeans, ont pour objectif de créer des composants non seulement pour la visualisation mais aussi pour l'interaction, avec des fonctions de filtrage, de sélection, d'analyse et d'édition) ; une API est définie pour l'intégration à d'autres applications ; les widgets proposés sont : la carte, la séquence, l'alignement multiple, BlastView (visualisation de résultats de recherche de séquences similaires dans les banques avec le logiciel Blast) et AnnotView pour visualiser des annotations.
- Zldb [Zel94] est un système d'objets graphiques (suivant le modèle que nous avons adopté : un objet égale une fenêtre), avec des interfaces de commande pour manipuler directement les objets. Ce système semble malheureusement abandonné.
- JaMBW [Tol97] est un ensemble d'applications graphiques en Java. Ce ne sont pas des composants au sens propre du terme, mais des petits programmes ne réalisant qu'une seule fonction, disponibles sous forme d'applets.
- biowish [SP97] est une extension à Tcl/Tk pour l'analyse de séquences ; cet outil s'intègre donc dans des scripts et fournit une application graphique pour l'édition et la visualisation d'une séquence, ainsi que quelques fonctions d'analyse.
- BioSymphonybeans [KL98] est un ensemble de widgets et de composants implémentant un modèle de données pour les informations chimiques, structurelles et liées aux séquences. Il est implémenté avec des JavaBeans, et permet d'effectuer plusieurs sortes d'analyses classiques comme la recherche de similarité dans les banques ou la prédiction de structure secondaire.

4.2.1.8 Conclusion sur les composants.

Dans le cadre d'une réflexion sur l'extension potentielle des mécanismes de flexibilité à l'utilisateur, on ne peut pas ne pas mentionner les contraintes pour la fabrication de composants citées par certains

auteurs pourtant informaticiens. Ainsi [FCB+99] mentionne à propos du développement des Biowidgets la difficulté des techniques : il faut faire une API claire et complète, avec tous les évènements documentés et les interfaces "listener". Il note une certaine rigidité dans les framework JavaBeans et le graphe des évènements ou dans le mécanisme de notification MVC. Il souligne aussi la nécessité d'une conception mûrement pensée. [PRT00] reconnaît également que l'utilisation de Corba nécessite plus de travail et d'expertise professionnelle que d'autres approches plus classiques (comme la mise en place de scripts perl lançant des calculs et analysant les résultats).

Il semble également que des solutions d'intégration fondées sur des formats à la fois lisibles par l'homme et par la machine constituent un meilleur potentiel d'extensibilité pour l'utilisateur, même informaticien. C'est en partie ce qui explique le succès des langages XML en bio-informatique face au standard de représentation ASN.1 : ils n'ajoutent rien à ce dernier sur le plan conceptuel mais ils sont lisibles et manipulables. Peut-on en dire autant du scripting par rapport aux solutions purement programmatiques ? Peut-être faut-il plutôt préférer les approches multi-modèles s'il n'y a pas d'incompatibilité.

Enfin les questions portant sur la correspondance entre les composants et les objets biologiques ou sur la neutralité des composants par rapport aux données se poseront peut-être d'une manière différente dans la perspective d'une utilisation par les biologistes, ces derniers ayant probablement besoin de manipuler des objets proches de leur domaine.

4.2.2 Intégration

On associe souvent à l'idée d'environnement intégré l'idée de système dit "monolithique", qu'on oppose à la modularité des bibliothèques de composants. Mais la qualité d'intégration de ces environnements, dont l'objet est d'aider le biologiste à effectuer des tâches d'analyse en facilitant l'accès aux outils, leur enchaînement ainsi que la manipulation des données biologiques, est pourtant d'une grande utilité.

Ces systèmes vont de la machine-outil, ce qu'on appelle parfois les "genome crunchers", qui exécutent une chaîne de traitements d'un bout à l'autre (MAGPIE [GS96], GeneQuiz [SSC+94], PEDANT [FAH+01]), aux couteaux suisses et aux outils graphiques d'édition, en passant par des systèmes d'aide à la décision ou à la résolution de problèmes comme Imagene [MRDV99], le Biology Workbench du NCSA [Sub98], les outils de modélisation comme Swiss-PdbViewer [GP97b] et les outils d'analyse de séquences comme MACAW [SAL91]. Ce qui nous permet de tous les classer dans une catégorie d'environnements intégrés - alors qu'ils sont assez différents, c'est cette *coordination prévue d'avance* - bien sûr plus ou moins réussie - entre tâches et données qui va permettre au biologiste, une fois préparées ses données, d'obtenir des résultats sans avoir à effectuer de tâches intermédiaires sans intérêt par rapport au travail à réaliser, et sans exiger des connaissances techniques qu'il n'a pas et qu'il n'a pas envie d'avoir. [MRDV99] remarque ainsi que les méthodologies d'intégration ou de représentations de données sont rarement traitées ensemble ou implémentées avec le même soin dans le même environnement. L'approche du système Imagene [MRDV99] est donc d'intégrer étroitement gestion de données (entrées et sorties) et analyses biologiques, accompagnées d'outils de visualisation et d'évaluation pour le biologiste.

Nous aimerions montrer ici que la qualité d'intégration de ces systèmes "monolithiques" est indépendante des propriétés de flexibilité ou d'interactivité. Il est en effet tout à fait possible qu'un environnement intégré soit complètement inchangeable et en mode batch, ou bien au contraire qu'il soit très interactif, construit sur une architecture modulaire et personnalisable. Ainsi, certains de ces environnements intégrés [MRDV99] permettent une interaction importante de l'utilisateur concernant le déroulement des tâches, donnant par exemple la possibilité de reprendre une tâche à une étape intermédaire en fonction des résultats déjà obtenus. Un autre exemple est SeWer [Bas01], un des outils offrant le plus de possibilités de personnalisation parmi tous ceux que nous avons vus, mais il est en même temps très intégré puisqu'à partir d'une interface utilisateur unique il permet de lancer autant de programmes d'analyses répartis sur le Web qu'il est souhaitable en fonction du type de données de la requête (séquence, alignement, ...). Il est aussi un joli exemple d'intégration de services hétérogènes, et cela sans passer par des techniques ou des architectures lourdes et sophistiquées, puisque la seule technique utilisée est JavaScript.

Il est vrai que ces environnements "monolithiques" sont souvent et à juste titre critiqués sur leur manque de souplesse. Parmi ces systèmes nombreux sont ceux qui ne permettent aucune modification ou adaptation ni aucune manipulation en dehors de l'interface graphique standard : il est impossible de les utiliser indirectement ou de toute autre manière non prévue, et il est tout autant impossible de les ré-utiliser pour écrire d'autres programmes - le plus souvent parce que le code source n'est pas disponible mais aussi parce qu'ils ne sont pas conçus pour cela.

4.2.3 Conclusion

Cette revue des techniques utilisées dans le domaine de la bio-informatique pour améliorer la flexibilité des logiciels a voulu s'adapter aux tendances saillantes des développements en bio-informatique actuels. Le centre de la discussion a donc été l'ingénierie du composant souvent opposée aux démarches "monolithiques" moins flexibles, ainsi que le problème de la représentation des données sans lequel la tâche la construction de composants est rendue difficile puisque ces derniers ont du mal à communiquer. Nous notons cependant un certain glissement hors du problème puisque parler de standards par exemple lorsqu'on traite de flexibilité peut sembler un peu paradoxal. Sans doute la problématique de la flexibilité pour l'utilisateur, abordée dans la section suivante, se situera-t-elle un peu différemment.

Il se trouve également que les approches qui pourraient figurer au titre de programmation par l'utilisateur sont - dans ce domaine - en général présentées dans les conférences traitant de génie logiciel et de réutilisabilité, pour la raison sans doute que les techniques correspondantes sont "orientées-objets" voire "orientées-composants", et essentiellement tournées vers la construction (les kits de construction par exemple, encore qu'il y en ait très peu en dehors de Visual Basic). Mais, de manière générale, ces approches sont souvent orientées par la technique.

4.3 Flexibilité pour l'utilisateur

Après cette description des concepts et techniques utilisées en bio-informatique pour la flexibilité logicielle, nous allons maintenant examiner la flexibilité dans le contexte de la recherche en biologie :

- du point de vue du *pouvoir d'expression* requis pour des utilisateurs dont il est particulièrement difficile de prévoir les attentes et les techniques d'analyse adéquates ;
- du point de vue de *l'usage* réel, voire, en suivant une réflexion semblable à celle de [Dou96b], en observant les techniques susceptibles de laisser émerger les pratiques de travail.

On peut définir la flexibilité nécessaire dans ce contexte de manière très générale : elle englobe tout ce qui permet à l'utilisateur de maximiser sa surface de décision par rapport au système informatique. Cela peut-être :

- pouvoir *guider* finement un algorithme existant par des indications complexes et précises, comme le fait le logiciel SwissPdb-Viewer [GP97b] ;
- spécifier tout ou partie du *contrôle* dans un traitement ou dans une série de calculs, c'est-à-dire spécifier les conditions d'exécution, les conditions d'arrêt et les bornes d'itération, comme le permet le langage Darwin [GHKB00] ;
- modifier, ou plutôt *corriger des résultats* fournis par un programme en fonction de connaissances biologiques, fonction remplie par tous les éditeurs d'alignement ;
- *adapter* une configuration logicielle à l'aide d'outils de personalisation fournis par le logiciel, comme dans l'environnement GDE [SOW+94] ;
- *modifier* le système lui-même ; ce type de technique n'est pas proposé explicitement ni supporté dans les outils actuels à notre connaissance, mais c'est une pratique courante : le code source étant accessible, et l'auteur ayant donné son accord, on peut adapter le source aux besoins spécifiques.

Nous avons classé ces différents aspects selon trois axes (figure 3) :

- *flexibilité du système, ou modifiabilité* : décrit les possibilités effectives d'adaptation et de modification du logiciel ;
- *flexibilité de l'algorithme, ou interactivité* : décrit les systèmes donnant un contrôle important au biologiste sur le déroulement du calcul ou sur le résultat final : cela va des éditeurs de résultats,

aux environnements interactifs de résolution de problème, ou même aux systèmes scriptables et aux langages de programmation, qui font l'objet d'une section à part (cette section comprend les tableurs et les logiciels de programmation interactive comme Hypercard) ;

– *flexibilité des interfaces, ou intégrabilité* : cette dimension, qui fait le lien avec la section 4.2, a pour fonction de décrire les caractéristiques de l'intégration et de l'intégrabilité : qu'est-ce qu'un système intégrable pour un biologiste, quelle différence cela a-t-il avec l'intégration et comment cela se situe-t-il par rapport à la question de la flexibilité ? selon nous, du point de vue de l'utilisateur biologiste, l'intégrabililité est caractérisée par la possibilité, si cela est nécessaire, d'améliorer avec les outils appropriés l'intégration des composants de l'application qu'il est en train d'utiliser.

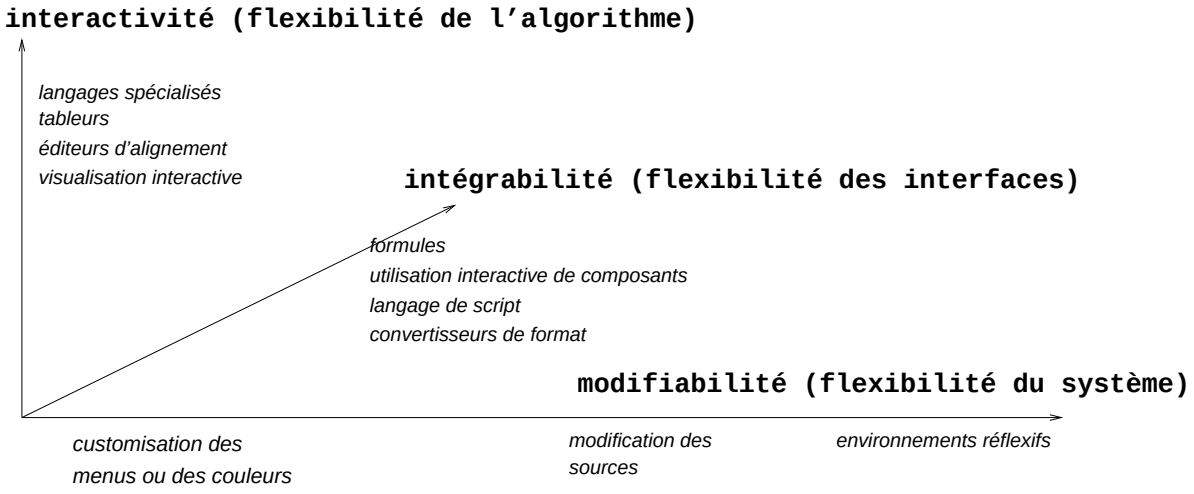


FIG. 4.3 – Dimensions de la programmabilité

Intégrabilité La caractéristique intégrabilité décrit également, que ce soit dans le cas d'une application ou d'un composant, le travail de programmation que le biologiste devra effectuer dans le cas où l'intégration n'est pas bonne :

– conversion de formats de séquence,
– analyse et extraction de résultats de programmes,
– etc...

Elle est directement dépendante des décisions par rapport aux standards de représentation des données et des résultats dont nous avons parlé dans la section précédente. On voit cependant que c'est sur cet axe que doivent se placer les outils accessibles à l'utilisateur pour remédier à un défaut d'intégrabilité. Ainsi, la possibilité de spécifier un objet par une transformation d'un autre, avec une formule, qui est fournie dans notre prototype, correspond à cela. C'est typiquement une zone d'intervention de l'utilisateur.

Telle que nous l'avons décrite, cette caractéristique ne dépend pas vraiment de la localité des composants : un système intégré peut fonctionner avec des services distribués sur le réseau. Ce qui fait sa plus ou moins bonne intégration, c'est la façon dont sont préparées les données et sont traités les résultats de ces services. On peut y situer aussi l'interopérabilité, mais l'intégrabilité est une caractéristique plus souple : s'il y a moyen de réparer les décalages, si on n'est pas dans du "plug and play", si les outils sont prévus, on peut parler d'intégrabilité.

Enfin, c'est sur cette dimension que l'on place la description des environnements dataflow assez nombreux en biologie, puisqu'il gèrent à la fois le flux de données et l'intégration programme-programme ou fonction-fonction.

Interactivité L'axe interactivité est celui où se décrivent les décisions prises par le biologiste : dans un tableur, un éditeur d'alignement, ou un traitement de texte, elles sont maximales, mais il y a également une certaine liberté par rapport aux outils d'analyses. On constate en effet que les biologistes sont parfois opportunistes par rapport aux résultats des analyses : s'ils confirment l'idée du chercheur,

ils servent d'illustration à cette idée, sinon, le biologiste considère souvent que l'algorithme utilisé ne correspond pas au bon modèle théorique et il essaye autre chose.

C'est aussi plutôt sur cet axe qu'on peut situer les approches explicitement programmatiques, ou l'utilisateur dispose d'un langage de programmation, fut-il de script ou très spécialisé, mais qui surtout lui permet de contrôler de très près le processus de résolution de problèmes.

Modifiabilité C'est la propriété qui décrit la capacité d'un système à changer. Pratiquement, comme il n'y a quasiment pas d'environnements re-programmables en bio-informatique, la modifiabilité des logiciels n'est effective que lorsqu'il y a des fonctions de personnalisation ou bien si le code source est disponible.

Ce qui différencie l'axe modifiabilité de l'axe interactivité, c'est d'abord l'objet sur lequel porte la programmabilité : dans le premier cas c'est l'objet logiciel, dans l'autre, c'est le processus de résolution de problèmes. De plus, on peut considérer que les deux ne se prévoient pas de la même manière : la programmabilité de l'algorithme est un problème qui relève des algorithmes et de la théorie des systèmes interactifs [Weg97b], alors que la programmabilité du logiciel relève des environnements de programmation, des principes de conception et de l'architecture de l'environnement.

4.3.1 Construction : flexibilité du système

Cette première partie s'intéresse aux techniques utilisées pour améliorer la flexibilité sur l'axe modifiabilité (flexibilité du système), c'est-à-dire modification du système lui-même de manière permanente, même si c'est pour un seul utilisateur. C'est la partie qui s'approche le plus de la section sur la flexibilité logicielle puisqu'elle s'intéresse encore à l'aspect constructif de la flexibilité, même si c'est dans le cadre d'un usage. Nous verrons essentiellement ce qui existe en personnalisation.

4.3.1.1 Personnalisation

Les fonctions de personnalisation permettent à l'utilisateur de définir le comportement du logiciel qui lui convient, et ceci de manière permanente [Mac91b] - c'est la raison pour laquelle nous en parlons dans la section "construction". Bien que certains systèmes enregistrent définitivement les choix de configuration des utilisateurs déduits de leur utilisation sans attendre qu'ils le demandent, c'est en principe une volonté explicite de l'utilisateur.

Dans les logiciels de bio-informatique, comme souvent, la personnalisation n'est souvent possible qu'au niveau des menus, particulièrement pour lancer des analyses externes, ou des couleurs pour la visualisation de propriétés prédéfinies. Quelques exemples :

- Le logiciel ORBIT [BHHW99] permet la personnalisation de formulaires pour lancer des analyses sur le web ; l'outil se veut assez puissant bien que l'architecture semble un peu lourde à manipuler, surtout pour un non-informaticien : par exemple, pour ajouter un service, il faut remplir un source en Java à partir d'un template et en utilisant des méthodes spécifiques à chaque type de widget.
- GDE (Genetic Data Environment) [SOW+94] montre un réel souci de l'extensibilité, sous la forme de personnalisation de menus pour lancer des analyses externes. Ces menus sont définis dans des fichiers texte et sont configurables en principe par l'utilisateur (l'environnement en fournit plusieurs exemples). L'outil comporte une interface graphique pour la définition des menus.
- tkGDE [PD00] est un environnement d'analyses de séquences (successeur du précédent) écrit en [incr Tcl]/[incr Tk] , qui comprend un éditeur de séquences et un éditeur d'annotations avec fonctions de zoom. Comme dans GDE, on peut configurer l'environnement pour le lancement d'analyses externes, en éditant un fichier de propriétés.
- Dans SeWer [Bas01] , la personnalisation est possible à deux niveaux :

  1. L'utilisateur peut conserver un formulaire personnalisé avec les valeurs qu'il a utilisées dans une requête précédente comme valeurs par défaut ; le générateur d'interfaces Pise [Let00b] propose une fonction équivalente, mais où, contrairement à SeWer, les valeurs par défaut sont éditables.
  2. De plus, l'outil est conçu pour être extensible à d'autres services, un peu comme ORBIT, GDE ou tkGDE. L'extension est plutôt du niveau technique de l'administrateur du site

Web local. Un petit protocole est à suivre : il faut donner des informations comme l'url du service à ajouter, le nom de variable du paramètre principal (la séquence requête) qui est une sorte de paramètre générique, et le programme se débrouille pour fabriquer l'adaptateur en javascript, qui nécessite encore quelques opérations de copier-coller.

- DCSE, un éditeur d'alignement spécialisé pour les structures d'ARN [RW93] utilise un fichier de ressources spécifiant des valeurs par défaut pour certains éléments de l'interface.

En revanche, des fonctions de personnalisation qui seraient très utiles, comme par exemple la possibilité de changer la place de la règle dans un éditeur d'alignement sont pratiquement inexistantes.

On peut s'interroger sur les raisons de ces limitations.

- les menus et les couleurs sont techniquement faciles à isoler ;
- ils ont une fonction importante, notamment pour lancer les analyses externes (cf pour [GRSS98] les analyses externes sont un des types de composants significatifs ;
- la personnalisation est souvent ce qui est pensé en dernier dans la conception ; or, la flexibilité cela se conçoit d'avance (c'est pour cette raison qu'il faut aussi de la flexibilité logicielle pour que cela marche) [Tri92].

4.3.2 Intégration : flexibilité des interfaces

C'est le deuxième axe, concernant l'intégration et les possibilités pour le biologiste d'intervenir sur les interfaces entre composants, que nous décrivons ici, en évoquant les outils reposant sur des mécanismes de flots de données, puis sous l'angle des rapports entre composants et interaction.

Que faut-il à un composant pour être utilisable par un non-professionnel ? Les composants sont-ils une bon point de départ pour les systèmes programmables, soit en tant qu'éléments du langage pour l'utilisateur, soit en tant qu'éléments graphiques ? Si ce qui importe en priorité c'est le travail du biologiste, qu'est-ce que cela implique sur la qualité logicielle ?

Nous développons l'idée que, si du point de vue de l'utilisateur c'est sur l'intégration qu'il faut insister, lorsqu'il s'agit de flexibilité pour l'utilisateur, c'est aussi à l'intégrabilité qu'il faut s'intéresser dans la mesure où si il y a des objets ou des opérations non prévues, ou bien qui ont évolué, il faut que le biologiste puisse accéder à des outils de conversion qui lui permettent d'adapter le logiciel à une situation imprévue. Cela doit pouvoir être réalisable en peu de lignes de code, ou par une édition simple, éventuellement automatisable par un mécanisme de macro. Par exemple, les utilisateurs des logiciels sur le serveur de l'Institut Pasteur sont très gênés par certains programmes qui ajoutent une information dans un fichier de données qui n'est pas acceptée par les autres outils. Ils peuvent perdre un temps précieux à comprendre d'où vient le problème, à trouver les outils pour le réparer, ... et plus tard à se souvenir de la procédure qu'ils ont effectuée pour trouver une solution. Un environnement flexible du point de vue de l'intégration devrait supporter ce type de manipulations : on voit ici qu'il ne s'agit ni d'algorithmique ni de personnalisation, mais simplement de manipulations semi-interactives, semi-programmatiques et qui relèvent pourtant de plein droit de la problématique de la flexibilité. On peut aussi se référer aux exemples de "cas" de programmation du chapitre I (1.2.1.4) qui comportaient un grand nombre d'exemples de ce type de manipulations.

Ce qui est important à l'échelle du biologiste ce sont souvent les formats de données, et la possibilité si cela fait défaut de spécifier, au bon niveau et avec les bons outils, les transformations et extractions nécessaires pour manipuler des résultats. Ainsi [Dop90] propose une macro d'entrée-sorties en format fasta, jugée suffisamment importante pour faire l'objet d'une figure dans l'article. C'est ici d'intégrabilité *pour l'utilisateur* qu'il s'agit, c'est-à-dire celle qui est nécessaire après que le logiciel soit terminé.

Certaines questions subsistent :

- Peut-on parler de composants "utilisateur" ? On voit dans [Boy98a] que les composants ne se conçoivent que dans une logique de construction de logiciels. Quelle sorte d'utilisation de composants (même non interactifs) peut-on imaginer pour des utilisateurs qui ne construisent pas de logiciels ?
- Le biologiste a-t-il pour rôle de fabriquer les composants (dans le cas où c'est lui qui connaît le do-

maine) ou de les utiliser ? [NDdM+94] distingue clairement les ingénieurs d'application qui créent les composants des développeurs qui les utilisent. Mais en bio-informatique, les deux possibilités se justifient.

- Comment se partagent les tâches de construction graphique et de conception d'algorithmes ? [SOW+94] répond clairement que le graphique n'est pas du ressort du biologiste, ainsi que [G97] selon qui c'est l'intérêt des tableurs de permettre d'écrire ses propres programmes sans se soucier de l'interface utilisateur, mais nous avons vu plus haut, sur les questions de personnalisation, qu'il peut être utile également de modifier ou d'étendre les fonctions graphiques pour des besoins de visualisation ou de présentation.
- Quels choix de granularité doit-on faire et doit-on prévoir plusieurs niveaux ? Les objets déterminés par [SFT+01] et [GRS95a], cités plus haut, ne sont-ils pas sujets à évolution selon les avancées de la recherche et l'orientation scientifique du laboratoire ?

4.3.2.1 Dataflow

La technique dataflow est très bien adaptée au travail d'analyse de données biologiques, ce qui explique qu'il existe plusieurs systèmes comportant cette fonction. Les plus connus sont le "NCSA Biology Workbench" [Sub98] et BioNavigator, fourni par la compagnie eBioinformatics dans lesquels l'utilisateur peut manipuler des données provenant des banques de données puis traitées et transformées par diverses étapes, telles que celles qui sont décrites par [SGB+01], sans avoir à se soucier des compatibilités de formats et de données : le système se charge de ne lui proposer, dans le contexte d'un objet, que les programmes qui sauront s'en servir. Piper [Biz00] (figure 4) ou [Boy98b] (figure 5) permettent aussi de connecter visuellement des composants reliés par des flux de données.

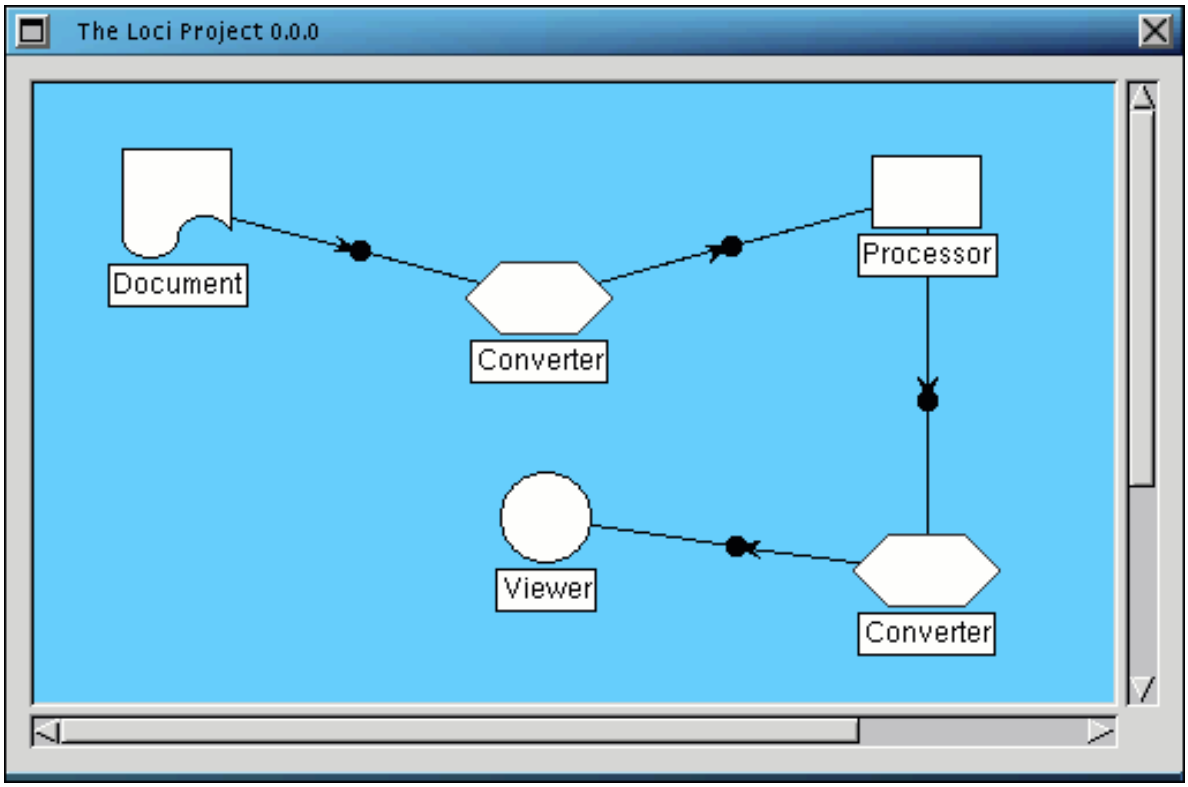


FIG. 4.4 – Piper

D'autres outils, comme par exemple tkGDE, un éditeur d'alignement [PD00] ou notre outil Pise [Let00b] font du flux de données mais plus implicite, dans le sens où ils proposent des menus dynamiques en fonction des types de données ou de résultats.

4.3.2.2 Composants et interaction

Nous avons vu dans la section 4.2 que le domaine de la bio-informatique donne lieu à de nombreux développements de composants, et que ces composants doivent pourvoir répondre à des situations complexes du point de vue des standards de protocoles et de données.

On peut ici faire le point sur ce qu'il en est de l'utilisation de ces composants par un biologiste non-informaticien et sur ce que cette utilisation a de spécifique. Par exemple, y a-t-il des dimensions interactives particulières qui changent l'utilisation de ces objets pour un non-professionnel de l'informatique ? Deux dimensions principales nous sont apparues comme importantes :

- la composition interactive,
- l'utilisation interactive de composants.

La composition interactive, c'est l'utilisation de composants pour construire une application, alors que l'utilisation interactive correspond à une situation où l'objectif de l'utilisateur n'est pas de construire, mais d'utiliser directement le composant pour lui-même.

Composition interactive.

Cette partie recouvre essentiellement les techniques reposant sur les JavaBeans ou sur Visual Basic, mais correspond également au système Piper [Biz00] (figure 4).

Les systèmes utilisant les JavaBeans sont déjà assez nombreux. L'argument qui sous-tend toute la conception du système proposé par [Boy98b], BioBeans (figure 5), est qu'il faut aider l'utilisateur à construire des applications par un design qui simplifie la composition logicielle (avec une focalisation sur les fonctions plutôt que le processus de fabrication). Les problèmes courants que ce système vise à résoudre sont qu'il n'y a en général pas d'accès aux méta-données des composants, que l'adaptation est impossible et qu'on manque d'archives de composants. Le système se propose donc d'aider l'utilisateur à l'identifier les fonctions des composants et à décider des moyens de les combiner. Il est aussi capable de traduire une configuration en application indépendante. Les composants communiquent par un transfert automatique d'informations ; l'utilisateur doit juste trouver l'orientation de la connexion, qui se réalise en rapprochant les boîtes de chaque élément à l'aide de la souris avec un retour visuel ("intelligent docking"). Java a été choisi pour le mécanisme "Adapter" qui permet d'implémenter le dataflow. Des fonctions sont disponibles (par un menu) pour choisir le type de transformation voulu (ils sont chargés automatiquement par le système lorsque l'utilisateur charge ses données).

[KL98] propose un environnement de même type : BioSymphonybeans. Enfin, bien connus également, les bioWidgets [CFGO98], comme c'est décrit dans [FCB+99], sont construits avec l'environnement Java Studio. L'image de la figure 6, extraite de l'article, montre un exemple de programme écrit dans cet environnement.

D'après les auteurs, l'intégration à l'environnement de développement (Java Studio) s'est révélée coûteuse en temps de travail initial. Il semble qu'un biologiste ait cependant pu mettre en place une des applications (BlastView) en peu de temps à partir d'un environnement déjà installé et de composants réutilisables.

Visual Basic est également très utilisé pour construire des interfaces d'applications interactives, notamment à l'Institut Pasteur (Taxotron de P. Grimont). Il est aussi utilisé dans certains cursus de biologie ou de bio-informatique pour l'apprentissage de la programmation.

Utilisation interactive de composants.

La notion d'utilisation interactive de composant que nous abordons ici est très proche des idées qui ont guidé la conception de notre prototype. Il est donc un peu difficile de lui faire correspondre des conceptions existantes, mais on peut le tenter avec deux idées un peu différentes :

1. l'idée de manipulation du composant directement dans l'interface,
2. l'idée d'utilisation par opposition à construction.

Il y a très peu d'exemples en bio-informatique d'environnements de composants conçus de telle sorte que les composants soient à la fois utilisables interactivement en tant que composants, c'est-à-dire des morceaux de code réutilisables avec des opérations de manipulation et d'édition de composant.

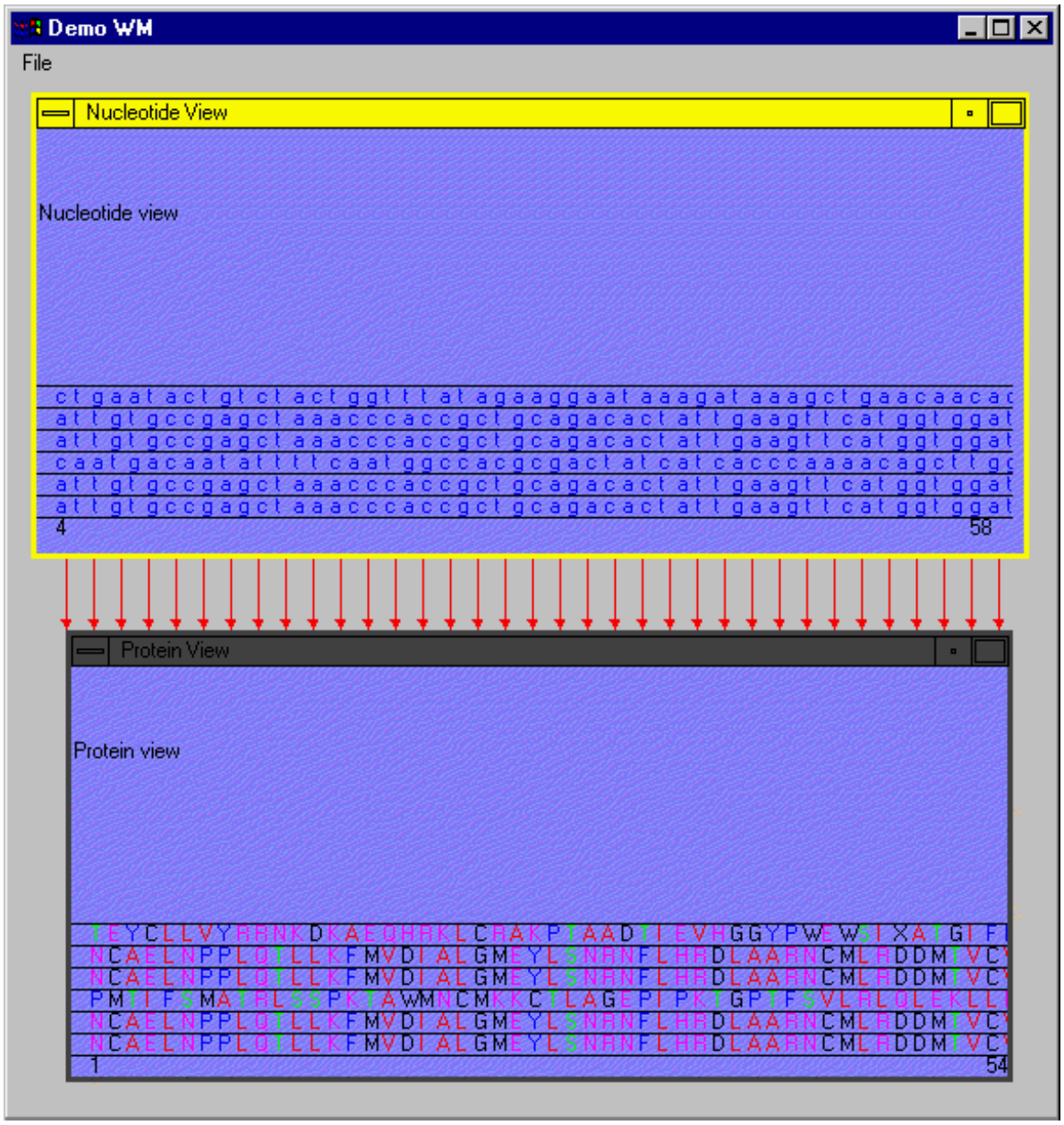


FIG. 4.5 – BioBeans

BioBeans et Piper décrits plus haut sont certes très interactifs, mais ne sont pas faits pour la définition des composants eux-mêmes à partir de l'environnement.

L'autre acception de l'idée d'utilisation interactive de composant est l'utilisation *sans but constructif* . Or les deux environnements cités sont par contre explicitement décrits comme des outils de construction d'application, sans remettre en cause l'idée que le biologiste voudrait à tous prix construire des applications plutôt que faire des analyses.

4.3.2.3 Systèmes décomposables

Un système décomposable est une application, c'est-à-dire d'abord un ensemble d'élements qui interagissent pour réaliser des fonctions et dont le but premier n'est pas de servir de brique de base pour la construction d'un autre objet, mais qui laisse transparaître et rend accessible et manipulable les

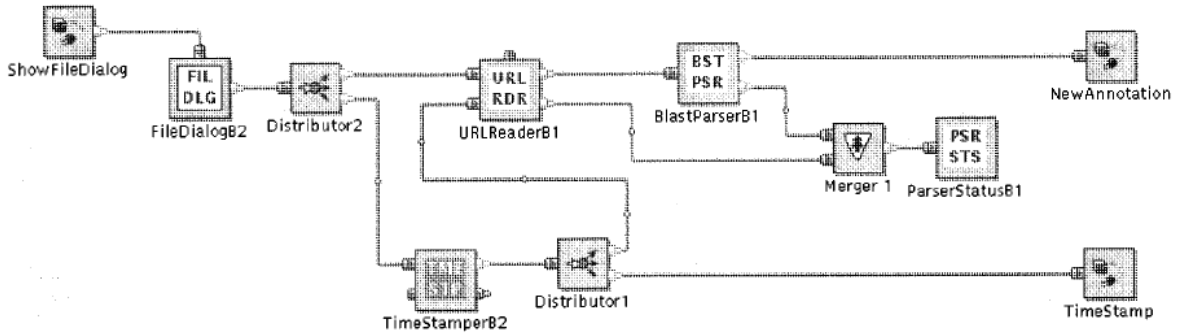


FIG. 4.6 – BioWidgets (d'après [FCB+99])

éléments dont il est composé. C'est dans ce contexte que nous comprenons la notion d'intégration, ou plutôt d'intégrabililité. Une application intégrée se distingue ainsi d'une application intégrable par le fait que dans le deuxième cas, il est possible d'interagir au niveau en-dessous, le composant lui-même, qui devient manipulable en tant que tel, soit pour être modifié, soit pour servir à d'autres compositions que celles dans lesquelles il est participe déjà.

Cette notion de re-design, de modification d'un système existant par remplacement de composants, est d'ailleurs invoquée par [Boy98a] qui cite [FL88]. La figure 7, empruntée à [FL88] montre comment se situe le re-design entre la construction d'applications monolithiques, les kits de construction et la réutilisation de composants.

Le problème que nous essayons de décrire se situe donc quelque part entre les environnements rigides et les univers difficiles à comprendre de composants qui nécessitent une construction. La réponse que nous apportons, celle d'application décomposable, essaye de fournir une solution intermédiaire, en :

- apportant une solution intégrée,
- permettant l'accès aux composants au sein de cette solution, et ce, *sous forme de composants* .

4.3.3 Interaction : flexibilité de l'algorithme

Cette section porte sur les approches qui couplent l'interaction et les algorithmes, c'est-à-dire celles qui permettent au biologiste d'intervenir dans les décisions ou le déroulement du programme. Parmi ces outils particulièrement interactifs, on trouve par exemple les éditeurs de résultats, comme les éditeurs d'alignement, dont nous parlons plus loin. On y trouve également les environnements permettant une itération entre les résultats intermédiaires des calculs afin d'ajuster les paramètres en fonction des résultats successifs, ainsi que les feuilles de calcul. L'objectif ici n'est donc pas uniquement la visualisation de résultats, mais aussi d'augmenter le degré de contrôle du programme par le biologiste, comme cela est décrit dans la taxonomie de [BHP+94].

Cette partie concerne donc l'interactivité : il ne s'agit pas nécessairement de l'interactivité au sens d'interactivité *graphique* , bien que cela soit souvent corrélé, mais de l'interactivité en tant qu'un meilleur contrôle est donné au biologiste sur la méthode d'obtention de ses résultats. Il y a un recoupement avec la composition de tâches en sous-tâches, qui est une des possibilités importantes de contrôle pour l'utilisateur, puisqu'il contrôle dans quel ordre, combien de fois, et dans quelles conditions sont effectuées les sous-tâches, mais ce qui est abordé ici est plutôt l'interactivité en tant que possibilité de dialogue entre l'utilisateur et le programme, en tant que l'utilisateur travaille sur un retour du programme et inversement le programme travaille sur une entrée de l'utilisateur. Ce ne sont pas nécessairement les mêmes mécanismes qui entrent en ligne de compte dans les deux cas, et le pouvoir d'expression ne se mesure pas sur la même dimension.

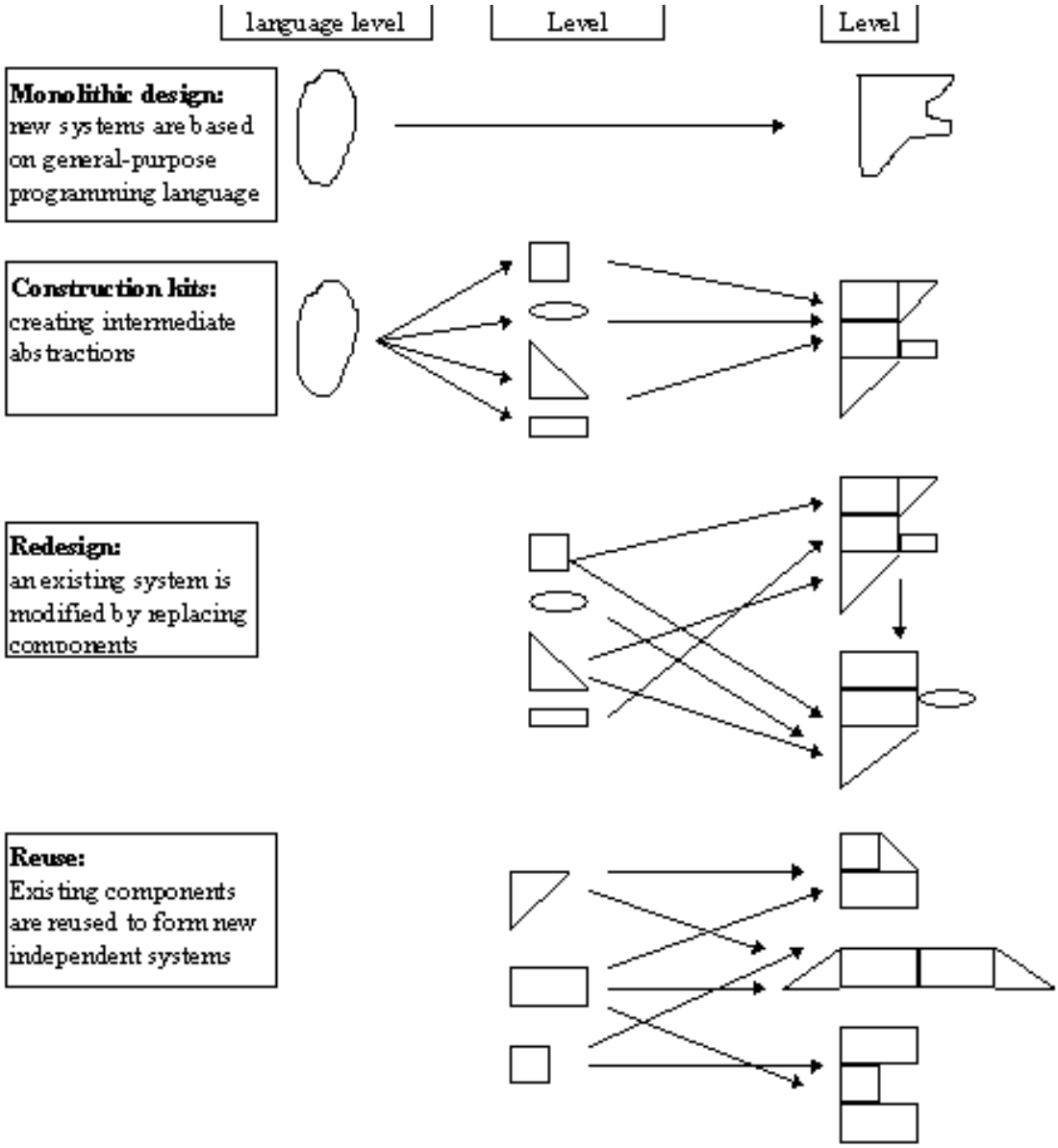


FIG. 4.7 – Construction d'applications (d'après [FL88])

4.3.3.1 Importance de l'informel

Pour introduire la problématique de la flexibilité algorithmique, nous commençons par les supports d'interaction les moins algorithmiques qui soient, comme modèle de surface de décision maximale du biologiste par rapport à un logiciel.

Texte

L'importance du texte comme format support de l'utilisation de l'informatique a été évoquée plus haut à propos des analyseurs de résultats de programmes. Cela reste vrai pour les autres types d'utilisation, et est avéré par le succès d'HTML en biologie, et ce dès les débuts du Web pour sa facilité et son absence de contraintes [AVB01]. Le format texte est limité mais très malléable. Nous avons ainsi pu observer qu'il pouvait être utilisé comme interface utilisateur dans certains laboratoires de l'Institut Pasteur, ne serait-ce que sous forme tabulée. L'exemple était une procédure de "validation" d'un "hit"

de recherche dans une banque directement dans le fichier des résultats - un peu reformatté pour pouvoir ensuite être fourni à un tableur. Nous avons également vu une utilisation de BBedit sur Macintosh pour maintenir une base de données, contenant des données et des analyses réparties sur plusieurs années de travail. De même, la très grande majorité des biologistes préfèrent utiliser un traitement de texte pour travailler sur la présentation de leurs alignements que l'un des nombreux logiciels qui existent.

Le visuel est beaucoup plus puissant mais plus rigide, non par nature, mais parce qu'écrire des logiciels graphiques ayant la même souplesse qu'un éditeur de texte (qui est d'ailleurs aussi un logiciel graphique) est plus difficile.

Édition de résultats

Dans la continuité de ce type d'utilisation de l'informatique, on trouve des logiciels dont le but *avoué* est de modifier les résultats des autres programmes. L'éditeur d'alignement est de tous ces outils d'édition de résultats le plus utilisé. L'alignement de séquences se situe en effet au centre des tâches d'analyses de séquences. Il permet notamment de comparer des objets (les sites, c'est-à-dire les colonnes de l'alignement qui représentent une position particulière dans la molécule), de constituer des profils types et des échantillons de données pour de très nombreuses analyses dont il est le tableau de données (au sens strict des statistiques).

Un alignement de séquences consiste à mettre en correspondance les éléments d'une séquence que sont les lettres représentant les acides aminés ou les acides nucléiques avec des éléments d'une autre séquence voire de plusieurs autres séquences. Comme on le voit dans la figure 8, lorsqu'il n'y a aucune correspondance dans une séquence données pour une lettre existant dans les autres séquences, on mets un "gap", une marque signifiant : "il n'y pas d'objet correspondant à cette position là pour moi".

```
---MLACMAGHSNSMALFSFSLLW-LC--SGVLG-------TDT
-------MRG-TPLLLVSLFSLLQ-DG--DCRL--------ANA
-------MAG--ALLGALLLLTL-----FGRSQG-------KNE
-------MQG--GQRPHLLLLLLA-VC-LG-AQS-------RNQ
-------MAG--PVLTLGLLAALV-VCALPGSWG-------LNE
MTLSHSALQF-WTHLYLWCLLLVP-AV-LTQQGSH----T--HA
MANSGPGAP--PPLLLLPLLLLLG-TG-LLPASSHI--ETRAHA
-----------MVQLLAGRWRPTG-AR-RGTRGGLPELSSAAKH
MTGFLRVFLV-LSATLSGSWVTLT-A----TAG----LSSVAEH
```

Figure 8. Exemple d'alignement

Aligner des séquences consiste donc à placer les lettres correspondantes dans une même colonne. C'est une tâche qui peut être effectuée soit par un programme, soit par le biologiste, soit le plus souvent par les deux. Le programme calcule les lettres correspondantes en fonction de leur ressemblance, définie par des matrices de scores de similarité, et en fonction d'un coût associé aux gaps et à leur prolongation. Il n'entre pas dans notre propos de décrire ici ces algorithmes, car ils sont assez nombreux, mais de dire simplement que ces algorithmes appliquent des heuristiques (l'espace combinatoire étant important) et que ces heuristiques ne peuvent prendre en compte tous les paramètres biologiques connus. C'est la raison pour laquelle en général, les biologistes rectifient les résultats "à la main" après le calcul, parce qu'ils savent mieux que le programme que tel acide aminé a toutes les chances de correspondre à tel autre, et aussi parce qu'ils ont la vision globale du contexte d'une position donnée.

Comme l'éditeur d'alignement est l'outil central de notre prototype, nous avons réalisé une revue logicielle de ces types d'outil (`http://www-alt.pasteur.fr/~letondal/review-edital.html`). Pour résumer cette revue, on peut dire que les éditeurs d'alignement servent à trois grands types de tâches :

1. la visualisation, par exemple avec des colorations qui mettent en avant des propriétés diverses,
2. l'édition de l'alignement (dont nous venons de parler),
3. le lancement de programmes d'alignement,
4. le formattage et la publication.

Ces outils combinent souvent les fonctions, comme celles de visualisation et de publication, où l'on mets en valeur certaines propriétés des données pertinentes par rapport au sujet de la publication.

L'édition d'alignement peut également s'aider de la visualisation, lorsque cette dernière mets en valeur une cohérence (similarité, co-variabilité, ...) soit au niveau d'un site, soit au niveau d'un domaine. De même, le lancement de méthodes d'alignement peut s'opérer sur certaines parties uniquement des séquences, qui peuvent avoir été sélectionnées selon les propriétés visualisées. Enfin, l'édition d'alignement se produit en général après une ou plusieurs phases d'alignement automatiques.

Annotations

L'annotation peut être réalisée automatiquement par des programmes, mais lorsqu'il s'agit d'annotations manuelles, c'est la possibilité d'avoir ses propres structures de données et analyses *informelles* , quitte à les formaliser ensuite. C'est également un réel processus de description des données, de définition de méta-données pourrait-on dire - comme définir des variables dans un programme ou des champs dans une base de données - mais totalement libre. Ainsi, dans [STM98], l'utilisateur peut associer des marques à des positions dans les séquences, les utiliser pour les recherches et la manipulation des données comme des citoyens de première classe, les retrouver par introspection, etc. Cette forme d'utilisation des données à travers des variables qui sont au choix de l'utilisateur est également utilisée dans Acedb [DTM91] sur lequel repose AcePerl, ainsi que dans SeqLab, VISTA [MBS+00] ou GeneDoc, des logiciels éditeurs d'alignement.

Codages non standards

Programmer c'est aussi la possibilité de codifier les données selon les nécessités des calculs que l'on veut faire. Peu d'outils permettent à l'utilisateur une codification complètement libre, comme une codification booléenne selon la présence ou non d'une propriété, ou une codification selon la composition chimique, ou encore, comme dans le logiciel DCSE [RW93] qui permet de codifier et contrôler les structures secondaires, par l'insertion de caractères supplémentaires (accolades et accent circonflexe) dans la séquence pour marquer les début et fin d'hélices.

On doit pourtant pouvoir utiliser les outils, comme ceux qui permettent de manipuler les alignements, sans être contraint par la codification standard des séquences biologiques. Il faudrait par exemple une hiérarchie de classes : générique, ADN, protéine, ARN. Un alignement de séquences, c'est aussi un tableau de données similaire à ceux qu'on trouve en analyse de données classique. Cela n'exclut pas qu'il soit possible d'effectuer des contrôles sur les données si cela est nécessaire, par exemple pour corriger les fautes de frappe, ou que l'éditeur contrôle les données interactivement (mode libre/contrôlé).

4.3.3.2 Visualisation interactive et contrôle du logiciel

Avant d'aborder la section concernant la programmation utilisateur, il faut citer comme particulièrement important du point de vue de la flexibilité - et surtout de l'interactivité - les systèmes qui rendent maximal le dialogue avec l'utilisateur aussi bien pour le contrôle de l'obtention des résultats que pour la manipulation des résultats eux-mêmes [BHP+94].

Des techniques de visualisation avancées sont de plus en plus utilisées pour manipuler et visualiser les informations biologiques et les résultats d'analyse. Par exemple, [hCRS+96], [CKBR97], [CRBK98] et [CRSB00] combine des techniques tableur et 3D pour observer des similarités entre séquences. [PVB98] propose l'utilisation de "marking menus" et d'interfaces zoomables pour naviguer sur des cartes génétiques et des chromosomes. [RF97] répertorie ainsi les techniques de "fisheye", de lentilles ou de zoom sémantiques pour visualiser des ensembles de taille importante, ainsi que les "information murals" et les filtres déplaçables. Ces techniques contribuent fortement à l'*analyse interactive* des données par l'utilisateur et sont un complément utile aux outils d'analyse algorithmique.

Une autre approche est d'affiner le *dialogue* entre le biologiste et l'algorithme, principe théorisé par exemple par [Dwo96]. Le logiciel MACAW [SAL91], par exemple, inclut des algorithmes d'extraction de motifs et d'alignement, et permet à l'utilisateur de choisir par les motifs trouvés ceux qu'il veut garder pour construire l'alignement. Ce principe très simple et très apprécié est pourtant très rarement proposé dans les logiciels. Dans le même ordre d'idée, [GGRK98] permet de réutiliser les résultats d'une requête dans une banque pour construire une nouvelle requête, par une technique de "copy-and-drop". Le logiciel Swiss-PdbViewer [GP97b] est un exemple typique d'outil interactif où la visualisation permet de guider les décisions de l'utilisateur sur le plan méthodologique, avec des étapes "manuelles" permettant l'édition de l'alignement préalable avant de le resoumettre pour l'optimisation de la construction d'un modèle

3D.

Une autre sorte d'algorithmes où l'application de ce type d'interaction peut être utile est l'inférence phylogénétique, où il est parfois possible pour l'utilisateur de choisir parmi les résultats précédents ceux des arbres phylogénétiques qui pourraient guider la construction d'un nouvel arbre. La granularité de l'information est en général un arbre entier : peut-être serait-il intéressant de l'étendre à des unités plus fines, comme des sous-arbres ? Lors d'un entretien avec un biologiste, l'idée de pouvoir mieux contrôler le déroulement d'un alignement multiple procédant par une première étape de comparaison des séquence par paires avait été discutée : d'après le chercheur, cette fonction peut gagner à une certaine interactivité, par exemple en faisant glisser la séquence successivement sur toutes les autres afin d'évaluer la qualité de alignement visuellement, notamment sur certains segments connus par le biologiste comme plus important que d'autres, surtout si on peut voir que les différences.

C'est ce principe d'utilisation des résultats pour fournir de l'information en retour à l'algorithme dans une itération contrôlée par le biologiste que nous avons essayé de généraliser dans le prototype biok à l'aide des techniques de mixin et de "user data".

4.3.4 Programmation par l'utilisateur

Les logiciels, aussi sophistiqués soient-ils, ne peuvent pas inclure ni prévoir toutes les fonctions et tous les paramétrages nécessaires au travail scientifique. Comme le disent [SOW+94] ou [GHKB00], il n'est pas possible pour la fabrication de logiciel d'aller aussi vite que la science et de tout intégrer au fur et à mesure - les outils seraient d'ailleurs trop complexes et leurs interfaces utilisateur trop chargées. C'est pour cette raison que les scientifiques doivent pouvoir s'ils le souhaitent - mais sans nécessairement devenir des informaticiens - créér des programmes de toute pièce grâce à des environnements conçus pour cela. Il existe - pas seulement en biologie bien sûr - toute une gamme d'outils comme les tableurs ou les logiciels comme HyperCard qui permettent de réaliser un certain nombre de choses. Nous nous proposons ici d'en faire une revue afin de repérer les raisons et le contexte de leur utilisation.

4.3.4.1 Programmation inutile

Mais nous voudrions, avant de commencer, citer d'abord quelques exemples de ce que les biologistes sont souvent encore *obligés de programmer* alors qu'ils pourraient l'obtenir autrement avec les outils adéquats (la programmation utilisateur, ce serait quand tous ces cas-là sont éliminés).

On peut par exemple distinguer, parmi les approches flux de données, celles qui permettent une utilisation interactive des fonctions dataflow, sans définitions préalables [PD00], [Sub98] ou [Let00b], par opposition aux approches qui demandent de la part de l'utilisateur une construction explicite du graphe dataflow [Biz00] ou [Boy98b]. Dans le premier cas, l'utilisateur n'a pas à modéliser son travail à l'avance puisque c'est le programme qui propose les fonctions disponibles en fonction du type de donnée que l'utilisateur veut traiter.

Nous avons également évoqué plus haut l'importance de l'intégration des données et des programmes pour le travail du biologiste. Si, en prévision d'un défaut d'intégration, nous pensons que les outils doivent fournir un support pour la programmation des routines "de secours" nécessaires à la poursuite du travail, on peut considérer cependant qu'il s'agit bien de "programmation inutile".

4.3.4.2 Tableurs

Il est très instructif de constater une large utilisation des feuilles de calcul dans de très nombreux domaines de la bio-informatique. Ainsi, on peut énumérer un certains nombre d'exemples en analyse de séquences :

- TransMem [ACO+97] prédit des segments transmembranaires avec un réseau de neurones (l'algorithme repose sur la distribution des acides aminés) ; [CCQ94] prédit également des caractéristiques de structures de protéines (de manière assez calculatoire, mais on est conscient des étapes intermédiaires) ;
- [AG00] effectue des alignements de séquences pour contruire un contig ;

- le programme proposé par [Ber87] a un but pédagogique : faire comprendre les algorithmes d'alignement en partant du principe que la notation tableur clarifie les méthodes et qu'elle favorise l'expérimentation ;
- parmi les programmes de visualisation : [hCRS+96] permet la visualisation de similarités dans un tableur 3D, tandis que [GRPM99] affiche des distances entre des séquences nucléiques ; [Hay93] et [Del00] permettent aussi la visualisation d'alignements multiples par des colorations diverses ;
- [Dop90] est un éditeur d'alignements multiples ;
- [MG98] effectue des statistiques sur les séquences : distribution des codons, distribution des 3ème positions de codon ; il peut aussi servir à traduire la séquence ou à construire une table des usages de codons.

Il existe également des feuilles de calcul pour gérer des données expérimentales [SP96], sachant que la plupart des macros pour feuilles de calcul ne font probablement pas l'objet d'une publication. A titre d'exemple, sur le campus de l'Intitut Pasteur, nous avons eu connaissance de feuilles de calcul pour l'extraction de résultats de recherche dans les banques.

Or, les tableurs sont assez mal adaptés et posent de nombreux problèmes pour réaliser ces fonctions - problèmes probablement résolus dans les versions plus récentes - mais ce qui nous intéresse ici c'est d'une part qu'ils sont cités dans les articles (même si ce sont des notes techniques, c'est surprenant), et c'est d'autre part la volonté d'utiliser quand même ces outils bien qu'ils soient si mal adaptés :

- [CCQ94] remarque que la taille maximum dans une cellule est insuffisante pour entrer une séquence (de taille supérieure à 255 lettres) ;
- le nombre de cellules horizontales insuffisant oblige [AG00] à disposer les séquences en colonnes ;
- [Hay93] constate les mêmes limites de 255 caractères par cellule et de 256 colonnes par feuille de calcul, mais aussi que le tiret - est un caractère spécial pour Excel, ce qui est réellement malencontreux pour éditer des alignements où ce même caractère représente un gap ;
- [Hay93] et [AG00] décrivent des manipulations très compliquées pour l'interaction : pour le premier, il faut sélectionner la séquence d'une certaine façon selon le format dans lequel elle a été obtenue par ailleurs ; si c'est une séquence venant de la banque Genbank, par exemple, il ne faut prendre que la partie avec les numéros, puis appeler la fonction 'Parse', puis enfin entrer son nom dans une cellule ; pour le deuxième, il faut deux feuilles intermédiaires uniquement pour les transformations de formats ; c'est la même chose pour [G97], qui décrit ainsi les procédures d'utilisation (toujours assez précises et longues dans ces types d'articles, surtout pour le formattage des données) :

  1. recherche dans la banque,
  2. insertion dans un fichier Word,
  3. édition des positions et autres décorations ,
  4. transformation de la séquence en colonne (par un menu Table->TextToTable ou par une macro pour ajouter un \n à chaque acide aminé)
  5. analyses.

  Pour [AG00], la construction de contigs se fait même à la main.

Sur le plan des types d'opérations et d'interactions, on peut citer un certain nombre de fonctions réalisables avec des feuilles de calcul :

- entrée de données : entrée d'une séquence dans une cellule [AG00] ;
- visualisation de données ;
- lancement de calculs, d'analyses ;
- édition d'alignements : décalage des séquences à gauche ou à droite, changement de place de toute une section de séquence [Hay93] ;
- sélection : sélections non contigües [Hay93] ;
- présentation des résultats d'un programme [GRPM99] ;
- sortie de transparents, présentation [Hay93].

Enfin, assez souvent, les auteurs de ces feuilles de calcul s'emploient à défendre les tableurs comme outil de programmation très pratiques.

- [Hay93] juge les tableurs plus simples et plus souples d'utilisation que le logiciel GDE que nous avons cités plusieurs fois dans ce chapitre [SOW+94].
- Dans [G97] l'utilisation d'un tableur pour faire ces analyses est évaluée : elle est pratique pour intégrer les analyses dans d'autres documents, pour ses possibilités graphiques et la manipulation des paramètres. Les tableurs sont également utiles pour l'apprentissage (voir plus loin). Si l'auteur préfère ne pas utiliser les programmes plus classiques, c'est parce qu'ils produisent des formats incompatibles d'une plate-forme à l'autre, parce qu'ils sont limités aux formats de sortie prévus par le programmeur, parce qu'il est parfois difficile de comprendre comment les faire marcher. Enfin, selon cet auteur, les algorithmes sont souvent modifiés par rapport à la publication initale (nous l'avons effectivement constaté pour les calculs de score du programme Blast). Enfin, pourquoi choisir le logiciel Excel ? tout simplement parce que le PC a été livré avec !

Il semble que nous devons tenir compte de ces réalisations pour comprendre à quel point il est important pour les chercheurs en biologie :

1. d'avoir un outil de travail *flexible* ,
2. et d'avoir un instrument dont on a le *contrôle* .

Et ce à un point tel que les limites de ces outils - tant sur le plan génie logiciel que sur le plan des interactions - ont moins d'importance que le productivité et l'intérêt intellectuel qu'ils suscitent.

4.3.4.3 Traitement de texte

Un phénomène similaire se produit avec les logiciels de traitement de texte, où il est possible d'effectuer des analyses réalisées sous forme de macros.

- [RSUZ91] décrit des macros permettant une visualisation de motifs en transformant les symboles A, C, G et T en symboles géométriques. Les motifs à base de lettres sont en effet trop monotones (ceci est d'ailleurs un argument pour dire que le format "lettre" choisi pour l'analyse de séquence est un bon compromis de format entre la machine et l'homme, mais que pour la visualisation, ce n'est pas toujours idéal) ; l'article invoque aussi l'argument que le logiciel est déjà installé dans tous les laboratoires (c'est aussi même chose que pour le Web, mais le Web n'est pas interactif ni éditable) ;
- [Sha93] reprend [RSUZ91] en ajoutant une fonction de recherche/remplacement et d'autres fonctions d'édition, qui rendent la visualisation de motifs plus interactive : on peut par exemple retirer un motif pour mieux voir les autres etc... ; et on peut aussi juste utiliser des caractères ASCII, plus flexibles ;
- [OCL93] décrit des macros pour la conversion de formats et l'édition d'alignements, ou encore pour le contrôle d'erreur, les calculs de similarité, des statistiques, une fonction d'alignement ADN/protéine sur les séquences de l'alignement converties en blocs d'édition Word ; propose également un utilitaire externe pour contrôler les décalages.

Très peu de ces réalisations sont bien sûr publiées, mais nous avons pu en constater d'assez similaires sur le campus de l'Institut Pasteur, notamment pour la fonction de présentation d'alignements. Aucun outil actuel ne permet par exemple de disposer un alignement librement - avec des espaces, des annotations, en plaçant la règle où l'on veut, etc... Encore une fois, le chercheur est prêt à perdre les fonctions d'édition spécifiques d'un alignement (notamment l'édition des gaps) pour travailler avec un outil dont il connaît le fonctionnement et qui n'impose pas trop de contraintes. Il semble par exemple que Word sache faire de l'édition par colonnes, du fenêtrage et utiliser des bookmarks et que cela soit considéré comme suffisant pour l'utiliser comme éditeur d'alignements. Ce phénomène est d'ailleurs décrit par [NJ94] à propos des logiciels de fabrication de transparents. Même s'il existe des programmes très sophistiqués pour manipuler de manière cohérente l'ensemble des transparents d'une même présentation, on utilise souvent un simple programme de dessin ou de traitement de texte. Les outils spécifiques demandent souvent plus de formation, d'efforts, et sont souvent moins disponibles ; de plus ils peuvent restreindre la créativité si on veut faire des choses qu'ils n'ont pas prévues. Par contre, les outils spécifiques sont appréciés des professionnels, et vont être préférés si la tâche est fréquente.

4.3.4.4 Programmation dans l'interface

Nous abordons maintenant les logiciels ouvrant à des fonctions de programmation vraiment générales, permettant notamment de spécifier le contrôle et de construire l'interface utilisateur. C'est le cas par exemple du système Hypercard [Goo87], mais nous parlerons brièvement aussi de LabView qui est utilisé en bio-informatique [lab87].

HyperCard/Hypertalk est utilisé pour faire de la phylogénie [Eer92], [Bro91] (un programme qui identifie des covariations et des changements compensatoires dans des alignements de séquences), de l'analyse d'ARN [Bro93], de l'analyse de séquences en général [Eer92] (alignements, traduction, ...), de la visualisation de similarités de séquences [Fro94], de la construction de plasmides [Fro99], des calculs moléculaires divers [Bey87] ou [Gil90] (tailles de fragment de restriction, ...), de l'analyse de liaison génétiques [NSS93], ou diverses autres fonctions [Usd92] (gestion et édition de données de séquences, évaluation d'oligonucléotides pour l'hybridisation,...). Nous avons également connaissance de l'utilisation d'HyperCard/HyperTalk à l'Institut Pasteur par un chercheur désirant écrire ses propres outils d'analyse de séquences pour ne pas être dépendant du centre de calcul. Les tâches programmées dans HyperCard étaient essentiellement des routines de reformattage, pour pallier les insuffisances des logiciels existants, mais aussi pour faire des alignements de séquence.

Les références à l'utilisation d'HyperCard/HyperTalk en bio-informatique se font il est vrai de plus en plus rares. Cela est-il dû à la disparition progressive de ce produit sur le marché ? Les anciens utilisateurs le conservent, mais il y en a apparemment peu de nouveaux. Pourtant, [Gil90] exprime l'intérêt de l'utilisation d'HyperCard pour le travail des biologistes, de part sa simplicité d'utilisation, et pour la possibilité qu'il offre à un utilisateur individuel de développer des outils flexibles et utiles. De même [NSS93] définit cet environnement comme un pont organisationnel entre les données (ici des marqueurs polymorphiques) et les programmes (analyses de liaison génétiques). Hypercard sert ici à la fois pour le formattage, l'organisation (petite base de données, comme dans [Usd92]) et l'édition de ces données.

LabView [lab87] est utilisé en biologie, mais dans des domaines plutôt spécialisés dans le traitement du signal, comme la neurobiologie, ou nécessitant de piloter des appareils de paillasse. Il s'agit d'un type d'utilisation un peu différent du précédent, que nous dirons beaucoup plus professionnel. Les 3 ou 4 programmeurs LabView du campus maîtrisent parfaitement cet environnement et sont de bons programmeurs, et programment par ailleurs également dans des langages plus classiques, comme C, perl ou Fortran.

Ces systèmes sont des outils de travail idéaux pour le biologiste non-informaticien qui aime bricoler par lui-même ses outils de travail sans pour autant s'investir dans la discipline informatique (ce dont il n'a pas nécessairement besoin pour écrire des petites routines). Il serait intéressant d'analyser pourquoi ils ne sont pas plus répandus. Listons quelques hypothèse :

- ils utilisent un langage propriétaire,
- ils sont payants (LabView est assez cher),
- ils ne sont pas promus par les centres informatiques,
- ils ne sont pas enseignés à l'université.

4.3.4.5 Langages spécialisés

Les langages décrits ici se différencient des composants dont nous avons parlé dans la première partie du chapitre principalement du fait qu'ils sont des outils de modélisation algorithmique. Ils favorisent l'interaction, et le résultat qu'en attend en général le biologiste est plus la réflexion scientifique au vu des résultats successifs que la construction d'un logiciel. Certains de ces langages sont simplement des langages de script associés à certains logiciels, comme alscript, molscript, Rasmol, Acnuc, SplitsTree, PAUP, ... ou des langages déclaratifs come TexShade [Bei00], une extension de LateX pour la visualisation et la publication d'alignements multiples. Ce langage possède des fonctions adaptées à la visualisation des séquences. Par exemple, le code :

```
\begin{texshade}{AQP.MSF}
        \shadingmode[allmatchspecial]{similar}
```

```
\end{texshade}
```

colore les résidus similaires (figure 9).

FIG. 4.8 – TexShade.

Il existe également des extensions de langages de script, comme biowish [SP97] que nous avons déjà cité. D'autres, comme Dynamite [BD97] sont des langages de modélisation scientifique. Enfin des langages comme Darwin [GHKB00] sont des langages de programmation généralistes avec du contrôle et des variables.

Ces langages privilégient en général une programmation de type *exploratoire* : c'est encore une autre façon, même si elle est ici plus difficile à maîtriser, d'interagir avec un algorithme, de jouer sur des variantes. Dynamite sert par exemple à construire des algorithmes de programmation dynamique à partir de la spécification d'un automate à états finis. Selon la spécification, on obtient ainsi soit des algorithmes d'alignement local, soit des algorithmes d'alignement global. Un système flexible de "labelling" pour les transitions permet de différencier les sous-séquences et d'ajouter de l'information pour l'algorithme. Les données sont de type séquence, mais avec des sous-types (protéine, génomique, DNA, coding, codontable et compmat) et des types utilisateur. Ce langage peut servir à autre chose qu'aligner des séquences, comme "aligner" des symboles d'affectation de structure secondaire sur une séquence de protéine. Il a servi également à intégrer de l'information sur des parties codantes ou des HMM (Modèles de Markov cachés) dans les critères d'un alignement, ou à combiner des modèles probablistes exon/intro avec des modèles probabilistes pour les domaines de protéines. Le labelling permet ainsi de combiner l'homologie et la prédiction de gènes, et permets généralement de modéliser des phénomènes biologiques. A chaque label correspond une interprétation spécifique.

[VHKW96] qui consiste en une bibliothèque de classes C++ pour les formats, les collections de séquences, les alignements par paires et qui comporte aussi des classes génériques (ensembles, matrices, ...), propose également, explicitement, un modèle de programmation exploratoire.

Le langage Darwin [GHKB00] est un langage spécialisé dans l'analyse de séquences pour les biologistes. Il s'utilise par l'intermédiaire d'un langage de commandes, et comprend des bibliothèques pour les alignements et leur l'évaluation, les arbres phylogénétiques, les structures secondaires, la manipulation de matrices, les statistiques, la visualisation, ainsi que plusieurs fonctions d'analyse originales. Il contient des routines d'accès aux banques de données et de conversion de formats. C'est un outil très complet. En particulier, il fournit des itérateurs spécialisés pour les opérations de comparaison (Align1ToAll, AllAll). L'utilisateur peut également fournir des fonctions sous forme de code C ou Java, ou même (s'il est expérimenté) modifier les routines de bibliothèque écrites dans le langage Darwin ou en créer de nouvelles. C'est l'un des systèmes les plus flexibles actuellement en bio-informatique, et, comme par hasard, il comporte quelques fonctions d'introspection. Il fournit enfin quelques mécanismes utiles pour la programmation, comme la documentation de programmes et une syntaxe associative d'accès aux sélecteurs.

Alors que le langage Darwin semble assez utilisé (le site Web affiche 400 copies distribuées), Dynamite souffre probablement d'une interface d'utilisation plutôt incompréhensible.

4.3.4.6 Langages visuels

On décrit ici les logiciels dont le langage est très proche des concepts informatiques (schéma de données, séquencement, connexions dataflow explicites, ...). Il y a un recoupement apparent avec la section "Composants et interaction" mais cette dernière relève plutôt des interfaces de composition interactive ou les composants eux-mêmes interactifs, pour lesquels l'utilisateur n'a a priori pas d'accès au langage d'implémentation qui lui permettrait de fabriquer lui-même ces composants. Apparaissent également ici les langages dont le but n'est pas constructif, mais qui permettent simplement de faciliter l'utilisation d'un système.

- [GGRK98] est une interface visuelle reposant sur une API Prolog et un langage de requêtes (Daplex) pour une banque de structures (P/FDM) ; le choix fait pour le niveau conceptuel de l'interface utilisateur est le schéma entités-relations montrant les cardinalités des relations, les classes et les sous-classes, etc... Les requêtes se font en cliquant sur les entités et en spécifiant les contraintes par menus et boîtes de dialogue. La requête est ensuite affichée (but pédagogique).
- [BAL99] permet des analyses de biodiversité sur le Web par l'intermédiaire d'un langage visuel.
- Dans la mesure où [Boy98b] ou [Boy98a] propose une approche de la construction d'applications par composition visuelle, il peut être cité ici. Mais la programmation s'arrêtant à la composition, il ne constitue pas un exemple de langage visuel complet.
- Dans [MRDV99] l'utilisateur peut contrôler l'exécution des sous-tâches d'une stratégie (une tâche) par un script visuel et interactif. Le code Lisp est éditable, mais nous n'avons pu déterminer de quelle manière d'après l'article.

Ces systèmes de programmation visuelle sont très sophistiqués, et très utiles dans le domaine où ils s'appliquent. Mais, comme nous l'avons plusieurs fois remarqué à propos des langages visuels, ils souffrent soit d'un manque de généralité parce qu'ils ne s'appliquent qu'à un domaine, soit d'une trop grande généralité qui les ramène au niveau conceptuel d'un vrai langage de programmation. Autrement dit, s'ils sont utiles, c'est plus par les fonctions qu'ils proposent que par l'aspect visuel comme solution générale au problème de la programmation.

4.3.4.7 Apprentissage et exploration

Un autre objectif important de la programmation, mais où celle-ci devient plus que dans les autres cas une activité intéressante en soi, est l'apprentissage. Sans doute parce que ces outils sont créés par des biologistes qui auraient bien aimé avoir ce type d'environnement, ils favorisent l'apprentissage et l'exploration de manière explicite.

[Ber87] pense ainsi que la technologie des tableurs peut encourager l'apprentissage de la programmation et la modification du code, et [Hay93] introduit volontairement des commentaires dans ses macros afin de les rendre modifiables. Nous avons déjà cité Dynamite [BD97] comme un système exploratoire, mais qui peut également servir à l'apprentissage des algorithmes classiques de comparaison de séquences, tout comme [VHKW96] pour qui expérimenter les algorithmes et les implémentations alternatives est encouragé et non puni. Dans [MG98] les calculs (calculs statistiques) sont conçus pour être extensibles, à titre d'exemple pour en faire d'autres plus complexes mais également comme une bonne introduction aux techniques des tableurs pour les étudiants. Enfin, pour [G97] l'utilisation d'un tableur permet de tester et de comprendre les algorithmes publiés dans la littérature.

Une des techniques reconnue comme très efficace dans l'apprentissage de la programmation dans un contexte d'utilisation par un non-informaticien est la technique de *"disclosure"* [DE95a] qui consiste à révéler des informations sur la programmation au fur et à mesure des interactions plutôt que d'apprendre la programmation séparément. Il existe plusieurs exemples d'utilisation de cette technique, comme par exemple dans [GGRK98] où la requête spécifiée visuellement est affichée ou dans notre outil Pise [Let00b] où la commande Unix correspondant au lancement du job sur le serveur Web est affichée. [MRDV99] parle également de transparence quant à l'exécution des différentes étapes des tâches d'analyse.

Nous aimerions également citer [PL92], une utilisation de Boxer pour modéliser des molécules. Il s'agit d'un projet assez original, mais qui montre l'intérêt et les objectifs possibles de la programmation en général et de la programmation directe dans l'interface pour des utilisateurs scientifiques. Cette

étude porte sur les relations entre l'apprentissage de la programmation et l'apprentissage scientifique à travers le langage de programmation par boîtes Boxer. Le sujet scientifique était l'apprentissage de la structure atomique de molécules (glucose, galactose). La programmation est ici considéré comme un outil de formalisation d'un problème : les contraintes sont acceptées car l'étudiant veut que le programme marche. L'étude porte essentiellement sur l'acquisition de l'abstraction procédurale, qui convient particulièrement aux structures moléculaires, souvent très semblables - différenciées par exemple par un ordre différent des mêmes atomes. Boxer rend cette tâche plus simple par le formalisme d'inclusion spatiale qui aide à l'organisation des programmes, d'autant plus que les boîtes peuvent être fermées ou ouvertes pour gérer la place sur l'écran. Le sujet, la biochimie, tout comme les mathématiques ou la physique, favorise la connaissance structurelle de la programmation (comment le programme fonctionne-t-il ?), par opposition à la connaissance fonctionnelle (que fait le programme ?).

4.4 Conclusion

Dans ce chapitre, nous avons essayé de brosser un tableau des éléments et des réalisations qui nous semblent intéressants sur le plan de la flexibilité des logiciels qu'utilisent les biologistes. Parmi ces descriptions, nous avons pu constater l'existence de système très intéressants (et souvent très utilisés) à cet égard, qui justifient notre démarche (tableurs, systèmes à contrôle interactif, langage orientés domaine, ...).

Malgré cette profusion d'outils, les possibilités de contrôle fin des outils d'analyse et d'adaptation des divers objets informatiques par les biologistes sont encore assez minces (nous parlons bien sûr de ceux des biologistes qui ne sont pas devenus bio-informaticiens et qui n'ont pas l'intention de le devenir). Parmi les problèmes principaux, illustrés par nos exemples du chapitre I, figurent essentiellement les problèmes d'adaptation dûs soit au manque de souplesse des logiciels, soit au caractère inaccessible des outils d'intégration, qui, lorsqu'ils existent, sont la plupart du temps sous la forme de bibliothèques nécessitant une mise en ouvre non-négligeable d'outils de programmation divers. C'est pourquoi nous avons insisté sur l'aspect d'intégrabilité, terme sous lequel nous classons les techniques permettant au biologiste d'adapter les mécanismes permettant l'intégration (convertisseurs, adaptateurs, ...) si celle-ci fait défaut. C'est aussi pour cette raison que nous préconisons une approche de la réutilisabilité plus souple, en ajoutant aux composants de type bibliothèque l'utilisation de programmes exécutables, d'interfaces programmatiques, et de services réseau. Cette acception de la réutilisation passe par une utilisation plus développée de techniques de scripting.

Si les langages de script ont un grand succès, c'est justement pour la simplicité de leur mise en oeuvre et le bon niveau d'abstraction des opérations qu'ils proposent. Mais les langages de script ne permettent pas de tout faire, notamment si le problème à résoudre se pose dans le cadre d'un logiciel existant qu'il est impossible de ré-implémenter.

C'est pourquoi il nous semble qu'une démarche globale et homogène allant à la fois vers un meilleur contrôle du biologiste sur son environnement d'analyse informatique et vers la programmation intégrée dans des outils efficaces et cohérents, sans que la programmation constitue un but en soi, est encore à améliorer.

Chapitre 5

Description des prototypes

5.1 Introduction

Ce chapitre présente les implémentations réalisées pendant la thèse pour mettre en oeuvre les idées exposées dans les chapitres précédents. Nous évoquons tout d'abord le logiciel Pise, qui n'est plus réellement un prototype puisqu'il est en production depuis environ quatre ans, puis un premier prototype qui a été réalisé pour explorer des idées de programmation dans l'interface, et enfin, biok, un environnement d'analyse de séquences programmable dans l'interface.

5.2 Un générateur d'interfaces, Pise

Le logiciel Pise, qui est décrit dans [Let00b], ainsi que sur la page Web `http://www-alt.pasteur.fr/~letondal/Pise/`, est un générateur d'interfaces pour des programmes tournant sous Unix. Cet outil est destiné aux logiciels non graphiques et non interactifs, à interface ligne de commande ou conversationnelle, qui effectuent des calculs plus ou moins longs sur des données biologiques, comme il en existe en très grand nombre dans le domaine de la biologie. La fonction de ces programmes est essentiellement méthodologique et scientifique : il s'agit d'implémenter un algorithme de manière efficace, mais sans souci de l'interface utilisateur. Ces logiciels sont la plupart du temps développés par les biologistes ayant déterminé la méthode, ou par des stagiaires de leur laboratoire. Bien souvent, les implémentations sont d'abord destinées à un usage personnel, puis sont ensuite mis à disposition de la communauté, soit tels quels, soit avec une documentation minimale. Ce sont parfois les relecteurs des journaux scientifiques, comme ceux du journal Bioinformatics, qui imposent pour la publication la mise à disposition du logiciel, même de manière commerciale. Au rythme où ils sont produits (plusieurs par mois), ces logiciels ne peuvent bien sûr pas être ajoutés dans des paquetages intégrés et cohérents [SOW+94].

C'est pourquoi nous avons eu l'idée de créer un logiciel générateur de formulaires et d'adaptateurs pour l'exécution de ces programmes. Ce générateur, à partir d'une définition en XML de l'interface du programme, qui décrit les caractéristiques de chaque paramètre ainsi que celles des résultats, génère deux formulaires en HTML, l'un minimal, l'autre comportant la liste complète des paramètres documentés du logiciel, ainsi qu'un programme CGI, en perl, capable de lancer l'exécution du programme et de récupérer ses résultats.

Un paramètre est décrit par un certain nombre d'attributs. A titre d'exemple, voici la définition d'un paramètre (-quicktree) pour le programme clustalw (programme d'alignement multiple) :

```
1 <parameter ismandatory=''1'' issimple=''1'' type=''Excl''>
2       <name>quicktree</name>
3       <attributes>
4
5          <prompt>Toggle Slow/Fast pairwise alignments (-quicktree)</prompt>
```

```
6          <vlist>
7             <value>slow</value> <label>Slow</label>
8             <value>fast</value> <label>Fast</label>
9          </vlist>
10         <vdef><value>slow</value></vdef>
11         <comment>
12            <value>slow: by dynamic programming (slow but accurate)</value>
13            <value>fast: method of Wilbur and Lipman (extremely fast but approximate)</value>
14         </comment>
15         <group>2</group>
16         <format>
17            <language>perl</language>
18            <code>($value eq ``fast'')? `` -quicktree'' : ``'' </code>
19         </format>
20         <precond>
21            <language>perl</language>
22            <code>($actions =~ /align/ )</code>
23         </precond>
24
25      </attributes>
26 </parameter>
```

Les attributs qui sont spécifiés dans cet exemple sont, en plus du type (Excl pour liste à choix unique) et du caractère obligatoire (attribut ismandatory, ligne 1), le texte associé au champs dans l'interface (prompt, ligne 5), la liste des valeurs qu'il peut prendre (vlist, lignes 6-9), la valeur par défaut (vdef, ligne 10), le commentaire d'aide (comment, lignes 11-14), une pré-condition spécifiée en perl pour prendre en compte ce paramètre (precond, lignes 20-23), et le code perl pour transformer le choix de l'utilisateur en une chaîne de caractères à placer sur la ligne de commande. La figure 1 montre comment le champs correspondant est affiché dans le formulaire généré.

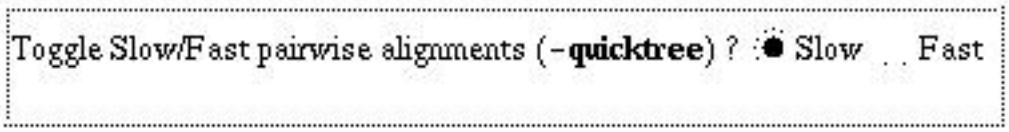


FIG. 5.1 – Champs généré dans le formulaire

Une des originalités de ce générateur est de prendre en compte, à la manière du 'pipe' d'Unix, les interconnexions potentielles entre les sorties et les entrées des programmes, réalisant par là une fonction flot de données. Lorsque l'utilisateur obtient ses résultats, par l'intermédiaire d'une page HTML, les programmes susceptibles d'utiliser les résultats comme données d'entrée lui sont proposés dans un menu et la préparation de ces données pour le formulaire suivant est automatique. Comme cela est montré dans la figure 2, le fichier de sortie outfile présenté dans la page de résultats du programme (2), peut-être redirigé par l'utilisateur dans le formulaire d'un autre programme, neighbor (3), par le moyen d'un menu affiché à côté du fichier (4), qui propose tous les programmes pouvant prendre ce type de données en entrée (neighbor, molphy, pyramids, ...).

Ce mécanisme est implémenté par un attribut associé aux paramètres concernés définissant le type du fichier.

1. dans la définition du programme dnadist : le fichier de sortie est défini comme étant de type **dist_matrix** (ligne 6).

```
1   <parameter type="Results">
2         <name>outfile</name>
3         <attributes>
4              <attribute>
```

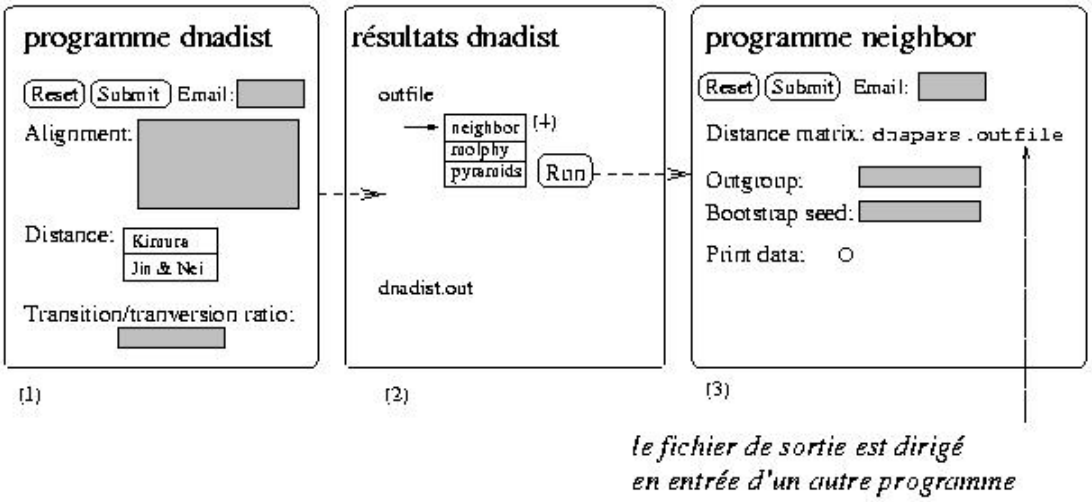


FIG. 5.2 – Connexions de programmes

```
5                <pipe>
6                   <pipetype>dist_matrix</pipetype>
7                </pipe>
8             </attribute>
9         </attributes>
10  </parameter>
```

2. dans la définition du programme neighbor, le même type est associé au fichier d'entrée (ligne 6) :

```
1   <parameter type="InFile">
2        <name>infile</name>
3        <attributes>
4             <attribute>
5                 <pipe>
6                    <pipetype>dist_matrix</pipetype>
7                 </pipe>
8             </attribute>
9        </attributes>
10  </parameter>
```

Le système est capable de générer d'autres types d'interfaces : il comporte également une interface programmatique en perl et en Tcl, ainsi qu'un générateur de client Tcl/Tk. A travers les interfaces programmatiques, qui servent aussi bien pour l'initialisation d'un job (paramètres, données) que pour l'enchaînement des programmes, le serveur Web devient scriptable. En effet, les pages de résultats issues du programme CGI sont formattées en XHTML, ce qui en permet une analyse très simple pour déterminer les enchaînements possibles.

Enfin, il est possible de créer des macros à partir de l'interface interactive. Cette fonction repose sur l'interface programmatique et utilise les étapes interactives suivies par l'utilisateur pour construire la macro.

Pise est utilisé dans le prototype biok, l'environnement programmable que nous décrivons plus loin, à travers les générateurs d'interface de type Tcl/Tk.

Des tests utilisateurs ont été effectués lors de la phase de mise en place du logiciel. Ils sont décrits dans le chapitre II. Ce logiciel est actuellement installé sur le serveur Web de plusieurs sites de bio-informatique, en partie parce qu'il a été sollicité pour servir d'interface générique à une suite logicielle

d'outils d'analyse, EMBOSS, développé au Sanger Centre (EMBOSS possède maintenant plusieurs logiciels d'interfaces). Pise est utilisé par de nombreux biologistes, qui apprécient particulièrement la possibilité d'explorer les nombreux outils d'analyse existants à travers une interface simple et homogène. La possibilité de combiner des programmes par le système de "pipe" facilite également beaucoup le travail d'analyse.

5.3 Premier prototype

Ce premier prototype (figure 3), écrit en Tcl/Tk, s'appuie sur des idées d'interface générées lors d'un premier atelier de conception participative pour un logiciel d'extraction de motifs dans les séquences biologiques [SVS97] (décrit dans le chapitre II).

Dans la figure 3, les deux fenêtres centrales représentent le résultat du programme d'extraction de motifs, Alice [SVS97] et les séquences qui nt été analysées. Le résultat est un texte similaire à celui qui était présenté dans le chapitre II comportant la liste des modèles (motifs) trouvés, et la liste de leurs occurrences dans les séquences. Une petite fenêtre de commande (en rose) permet d'invoquer des commandes sur ce résultat (recherche de mots, affichage d'une modèle dans les séquences, etc...). Ces commandes sont expliquées dans le texte figurant sous le shell, partiellement montré dans la figure. Des boutons, en haut de l'image, correspondent à des commandes d'affichage de modèles précedemment exécutées afin de les re-exécuter ultérieurement. L'objectif de ce prototype est une exploration technique d'idées de programmation dans l'interface, s'appuyant sur les capacités d'introspection du langage. Parmi celles-ci :

- l'édition des attributs des widgets Tk, et des actions associées aux événements (bindings) ;
- la navigation dans l'arborescence des widgets ;
- la création de widgets ou d'ensemble de widgets par clonage (fortement inspiré des boutons de [MCLM90]) : celle-ci est réalisée par un copier dans le tampon de la sélection X puis par un coller dans la commande eval après des translations de noms adaptés à l'espace de nommage cible ; la cible, en général un widget conteneur, est déterminée par l'utilisateur ; comme le code de clonage est disponible dans la sélection X, il est également possible de la recopier dans un fichier pour un usage ultérieur ou dans un message électronique pour la faire partager à d'autres personnes ; l'opération de coller peut ainsi être utilisée à partir d'un code source sélectionné dans l'environnement : son résultat est l'apparition d'un nouvel élément graphique dans l'interface ;
- la création interactive d'étiquettes graphiques (tags), qu'on peut considérer comme des variables graphiques, pour naviguer dans les résultats du programme d'extraction de motifs : l'action par défaut associé à un tag est l'affichage successif de ses occurrences dans les séquences qui ont servi à l'extraction, mais cette action est modifiable ; la création de l'étiquette est réalisée par une commande tag permettant de spécifier une couleur, et marquant la sélection courante ;
- ce prototype explore également l'idée d'interface éditable comme un texte, ou l'interface est en même temps un document dans lequel on peut placer et déplacer des applications (ce qui est aisé à réaliser en Tk, grâce à la notion de fenêtre, qui existe également dans le widget canvas et dans le widget tktable) ; ainsi, pour supprimer un widget, il suffit d'utiliser la touche Backspace ; cette idée s'applique assez bien à la fonction de présentation et d'annotations de résultats de programmes pour la biologie : en effet, il suffit de considérer que le texte juxtapose les notes et les programmes - initalisées avec les données pertinentes, selon une disposition totalement libre du moment qu'elle est compatible avec les algorithmes de disposition dans un widget de texte ;
- le shell de commandes est ainsi un widget, muni d'une documentation présente dans le texte et éditable, ainsi que d'un historique (affiché en fenêtre séparée dans l'image) ;
- les modifications graphiques de l'utilisateur sont persistentes ; la persistance est ici implémentée de manière assez peu efficace sous la forme d'un enregistrement explicite de toutes les commandes ainsi qu'un vidage du widget texte principal (fonction dump du widget Tk) ;
- il est possible, en protégeant les instructions d'entrées-sorties, de générer une tclet, équivalent en Tcl d'une applet java (l'effet sur l'auditoire est garanti !) ;
- le code source est introspectable et éditable (par une commande edit).

Ce prototype est assez peu réaliste du point de vue biologique, mal structuré du point de vue de l'architecture et des concepts, et les outils de navigation ne portent que sur les widgets graphiques,

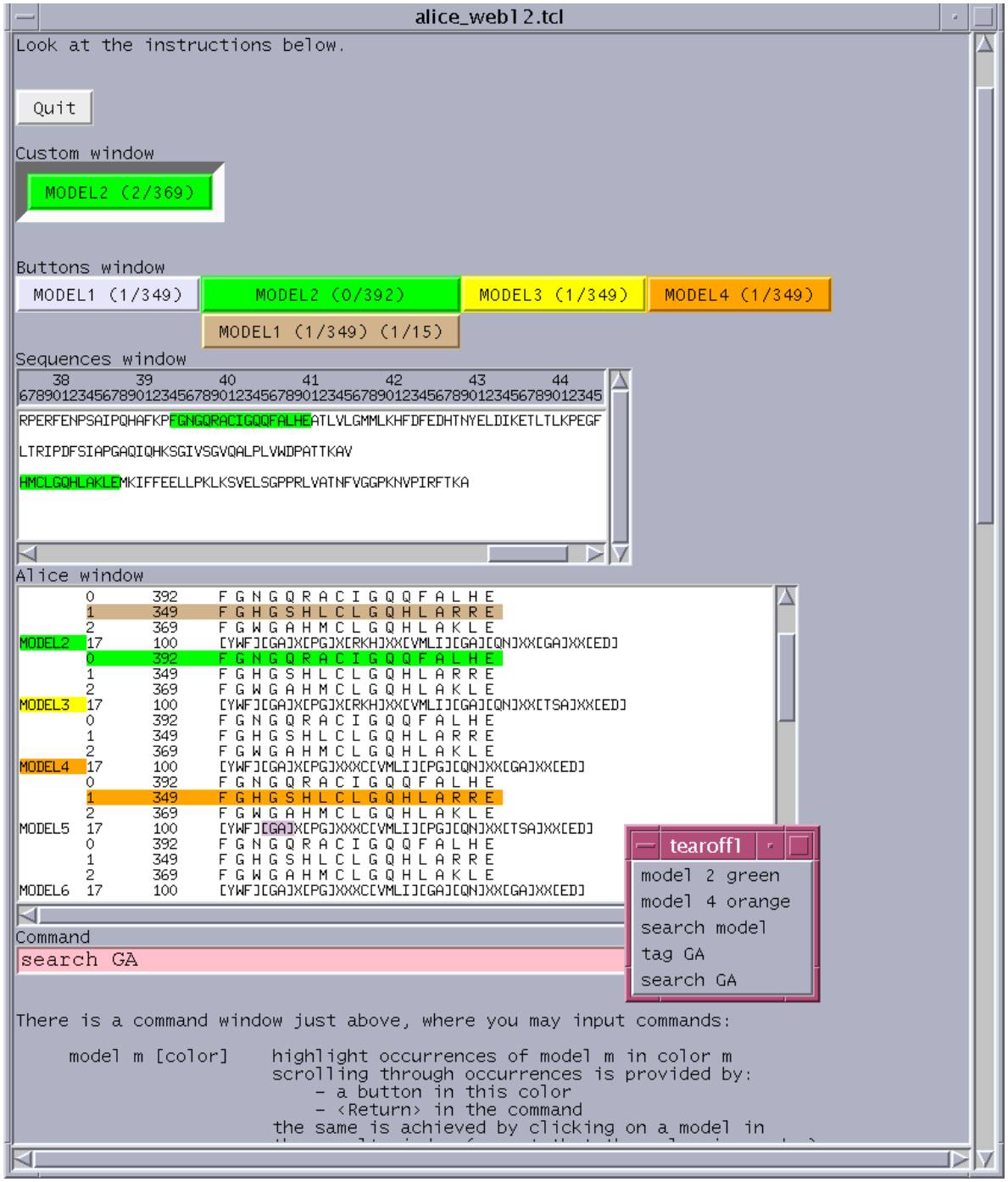


FIG. 5.3 – Premier prototype

en excluant toute autre abstraction comme des objets ou des classes (ce prototype est écrit en simple Tcl). C'est pourquoi nous avons repris plusieurs de ces idées, mais en repartant sur une implémentation complètement nouvelle pour le prototype final.

5.4 Prototype biok

Le prototype biok (Biology Interactive Objects Kit) est à la fois un environnement de travail pour l'analyse de séquences, et un environnement de programmation. Les idées ayant déterminé ce principe ont été longuement expliquées dans les chapitres précédents, mais se résument ainsi : il s'agit de

- supporter la programmation comme une *activité secondaire* par rapport à l'activité principale, l'analyse de séquences (ou tout autre analyse biologique),
- supporter la *programmation incrémentale* : cela consiste soit à modifier le code existant, soit à ajouter de nouvelles méthodes à un objet, soit à créer de nouveaux objets ; en réalité, nous n'imposons volontairement pas de limites ;
- supporter néanmoins la programmation tout court avec des outils corrects comme un débogueur facile à invoquer et à désactiver, et des outils d'observation et de trace des appels de méthodes ;
- supporter la *programmation interactive* , dans le sens restreint d'un interpréteur, mais avec l'idée tout de même que l'interpréteur ou le formulaire sont rattachés à un objet graphique, de manière tangible ; rien par ailleurs ne s'oppose à l'utilisation de techniques de programmation par démonstration ou sur exemple dans cet environnement ;
- supporter l'*apprentissage dans le contexte d'une application réelle* : cet environnement pourrait tout à fait être utilisé pour un cours de programmation par objets ;
- supporter le modèle de calcul *flot de données* et quelques-unes des idées inhérentes aux *tableurs* , comme la double désignation graphique et symbolique, qui produit une équivalence entre les langages d'interaction de de commandes ; l'environnement permet de définir la valeur d'un objet à partir de la valeur d'un autre objet, mais ne respecte pas la règle stricte de [Kay84] (value rule) : toute valeur est définie par une formule.

5.4.1 Présentation générale

Après une présentation générale du langage XOtcl et du principe d'objet graphique, nous présentons l'outil principal de biok, l'éditeur d'alignement. Nous développons ensuite les principes de l'architecture, et montrons leur lien avec les idées développées dans cette thèse. La description de deux stages d'étudiants en bio-informatique, qui se sont déroulés dans le cadre du prototype biok et qui nous ont beaucoup encouragé dans notre démarche, termine cette section.

5.4.1.1 Le langage XOtcl

Le langage choisi pour l'implémentation est XOtcl (`http://www.xotcl.org`), une extension objet de Tcl [Ous98] basée sur Otcl, créé au MIT [WL94]. Le système d'objets de XOtcl reprend les idées principales d'Otcl : possibilité de définir des objets, des classes et des méta-classes, héritage multiple. Les classes sont juste des objets particuliers permettant de définir le comportement d'autres objets, notamment pour la création, la destruction et l'héritage. Le comportement de tout objet, qu'il soit une instance, une classe, ou une méta-classe, peut être modifié par l'ajout de méthodes. Il s'agit donc pratiquement d'un langage de prototypes, même si la réutilisation de méthode repose sur l'héritage et non la délégation.

XOtcl a été développé par un laboratoire de recherche en informatique pour bénéficier à la fois des avantages des langages de script (notamment le prototypage) et des langages à objets. Les fonctions ajoutées par XOtcl sont l'aggrégation dynamique d'objets et le concept de classes imbriquées. Ainsi l'expression :

```
Spreadsheet::Area
```

définit une classe Area, comme composante de la classe Spreadsheet et :

```
pise0::clustalw0
pise0::clustalw1
```

est l'identifiant de deux objets clustalw0 et clustalw1 aggrégés dans un objet englobant, pise0.

Toutes les structures sont introspectables, selon la philosophie du langage Tcl. Le langage fournit des outils d'introspection supplémentaires comme les filtres et les mixins, que nous utilisons dans l'architecture du prototype. Un filtre est une méthode qu'on peut associer dynamiquement à une classe, et qui est invoqué lors de tous les appels de méthodes de cette classe. Un mixin est une classe dont les méthodes sont similaires à des filtres. Contrairement aux filtres, les mixins sont associés à des objets et non à des classes, et les méthodes filtrées sont uniquement celles pour lesquelles la classe mixin a définit une méthode. Le langage comporte également des mécanismes de méta-données, d'assertions, et de chargement dynamique de composants.

Les classes possèdent des paramètres, mécanisme très utiles pour initialiser les variables d'instance dans la commande d'instanciation. La commande suivante : :

```
Sequence $seq -name 1a02 -file /home/letondal/data/human/1a02.fasta
```

crée une instance de la classe Sequence dont les variables d'instance name et file sont pré-initalisées. Les paramètres sont déclarés par la commande de déclaration de la classe et sont hérités :

```
Class Sequence -superclass GraphObject -parameter {file}
```

XOtcl est également le langage choisi comme langage de programmation utilisateur, pour la programmation des formules. La discussion entre le choix d'un nouveau langage plus facile et plus adapté au domaine, comme dans Mathematica ou Hypercard, et spécialement étudié pour des non-professionnels et l'utilisation d'un langage général, comme Lisp dans AutoCAD ou Basic dans Word, montre qu'il il y a des avantages des deux côtés [Eis97]. Mais nous pensons qu'un langage comme Tcl, susceptible d'être retrouvé dans d'autres applications, et utilisé par une communauté importante de programmeurs constitue un investissement plus profitable pour les biologistes, d'autant plus qu'il y a déjà de nombreux outils écrits dans ce langage (même si la tendance est plutôt vers l'utilisation de perl ou de Python).

5.4.1.2 Les objets graphiques

Le composant central du prototype est ce que nous avons appellé un objet graphique, représentation graphique d'un objet significatif de l'application : une séquence, un graphique, un alignement. Comme tous les objets sont par principe éditables, l'objet graphique est également un éditeur : un éditeur de séquence ou d'alignement, un outil d'affichage et d'édition de courbes, un éditeur de code. Les objets graphiques sont des sous-classes de la classe GraphObject qui définit les caractéristiques graphiques standards, illustrées dans la figure 4, ainsi que les méthodes générales de manipulation des objets graphiques.

La partie graphique principale est la partie widgets, qui peut comporter tous les widgets nécessaires à l'objet, ainsi que des objets graphiques internes. Le panneau de boutons peut contenir autant de boutons qu'il est nécessaire, et des commandes permettent de les manipuler. L'alias est le "nom" de l'objet graphique, que l'utilisateur peut utiliser dans les formules. Il y a deux autres parties génériques : le shell (figure 5) et la zone d'entrée de la formule (figure 6). Le shell permet d'invoquer les méthodes de l'objet. Les commandes sont mémorisées dans un historique spécifique à la classe de l'objet, sous la forme d'un fichier éditable. Une méthode standard %w permet de manipuler directement le widget Tk central.

Le contenu d'un objet peut être défini par une formule, entrée dans la zone en bleu qui s'ouvre en double-cliquant sur le nom de l'objet.

La syntaxe des formules est de style objet :

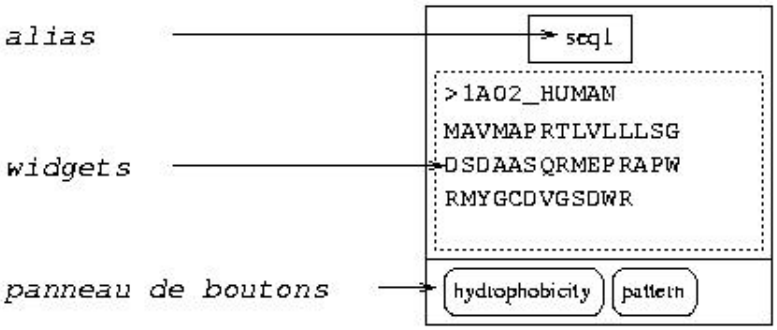


FIG. 5.4 – Object graphique.

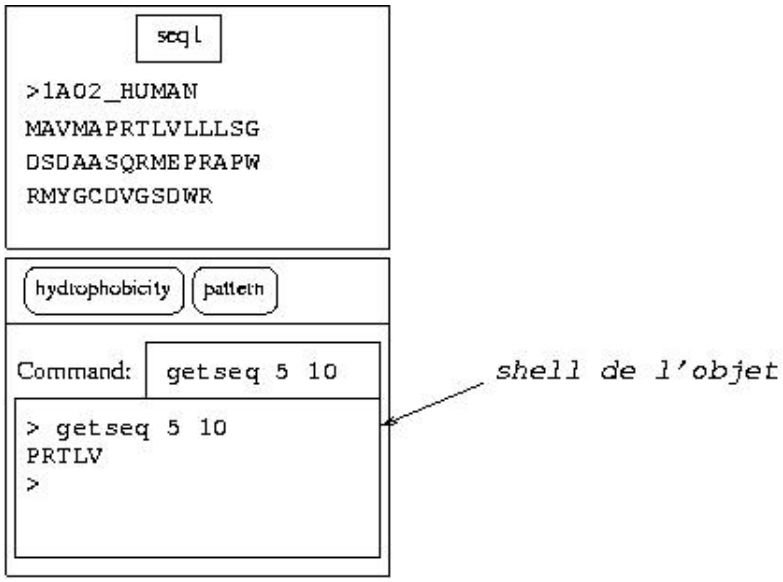


FIG. 5.5 – Shell de l'objet.

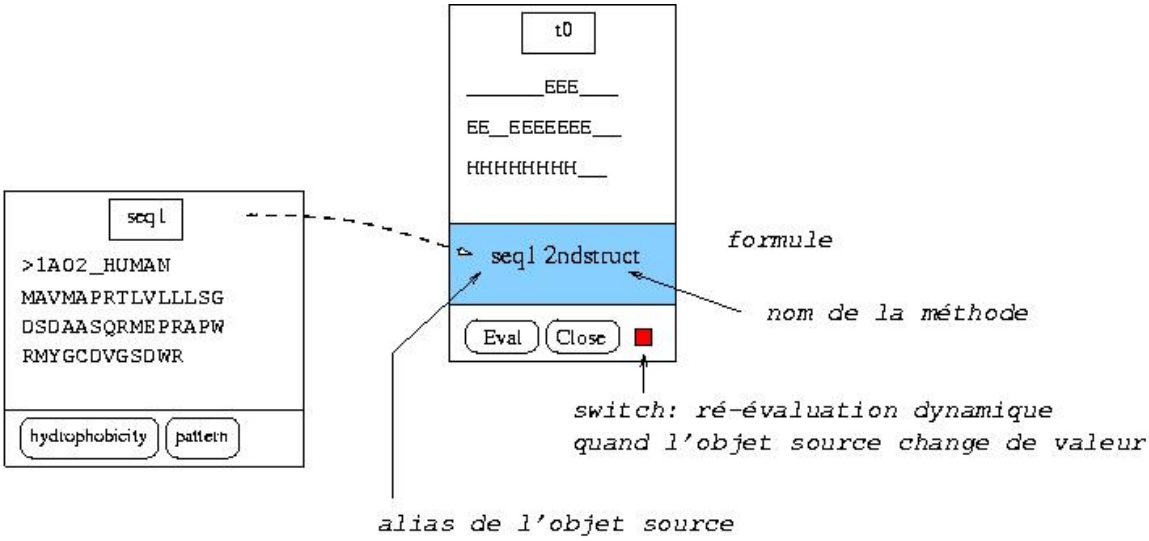


FIG. 5.6 – Formule.

```
objet méthode arguments
```

La formule est en XOtcl, avec des raccourcis syntaxiques pour certaines classes, comme pour les nombres, où il n'est pas nécessaire d'entrer la commande Tcl expr, ou encore pour un objet chargé depuis un fichier ou depuis une banque de données où le nom du fichier ou le code identifiant suffisent. On peut, à la demande, visualiser les dépendances entre les objets, qui apparaissent sous la forme d'une coloration du nom des objets appartenant à la chaîne de dépendances. Il est possible de visualiser plusieurs chaînes car elles sont affichées avec des couleurs différentes. La mise à jour est dynamique : dès que les objets sources changent, la formule est ré-évaluée. L'utilisateur peut désactiver cette mise à jour dynamique. Les formules utilisant un objet sont mémorisées au niveau de la classe de cet objet, et sont disponibles dans son menu. Si par exemple la formule d'une courbe (classe Plot) utilise une séquence (classe Sequence), cette formule apparaîtra dans le sous-menu connect des objets de classe Sequence. Cela permet à l'utilisateur, en utilisant ce menu, de créer un nouvel objet initialisé avec cette formule. La méthode standard d'évaluation d'une formule est evalscript, qui a deux fonctions :

1. appeller la méthode update de calcul effectif et de propagation des mises à jour,
2. initier la mise à jour des données relatives aux dépendances dataflow.

L'architecture sous-jacente aux objets graphiques et à l'implémentation des mécanismes dataflow est décrite dans les sections 5.4.3 et 5.4.4.

5.4.1.3 L'éditeur d'alignement

L'outil central sur lequel a porté le développement est un éditeur d'alignement. D'autres composants, comme un outils d'affichage de courbes et des éditeurs de texte ou de séquence ont été développés et sont succintement décrits dans cette section. Les outils de programmation sont abordés dans les sections 5.4.4 (réflexivité) et 5.4.6 (programmation par l'utilisateur).

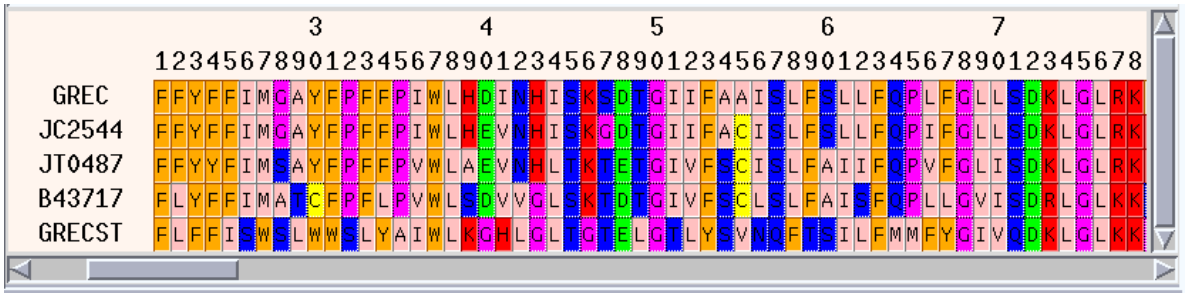


FIG. 5.7 – Éditeur d'alignement.

Fonctionalités

1. **Visualisation**
   L'éditeur fournit plusieurs fonctions de visualisation :
   - plusieurs schémas de coloration (la figure 7 affiche un schéma de propriétés physico-chimiques, de Zappo, emprunté à l'éditeur d'alignement jalview).
   - le pourcentage d'identité sur un site :
   - le contenu en GC ;
   - la nature de purine ou pyrimidine des bases d'ADN ;
   - les segments transmembraires d'une séquence de protéine, affichés ici en bleu, selon l'algorithme Toppred de [Hei92] (figure 9) ; cette visualisation repose sur l'implémentation de cet algorithme par une étudiante biologiste, dont le stage est décrit à la fin de ce chapitre ;
   - les résultats d'une analyse lancée par Pise dont le résultat est un texte contenant des indications de positions dans la séquence ;

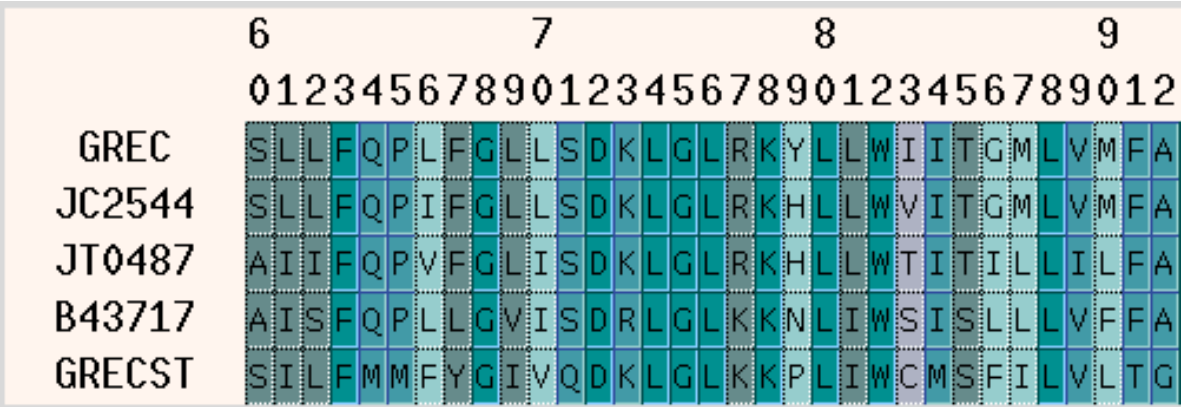


FIG. 5.8 – Pourcentage d'identité.

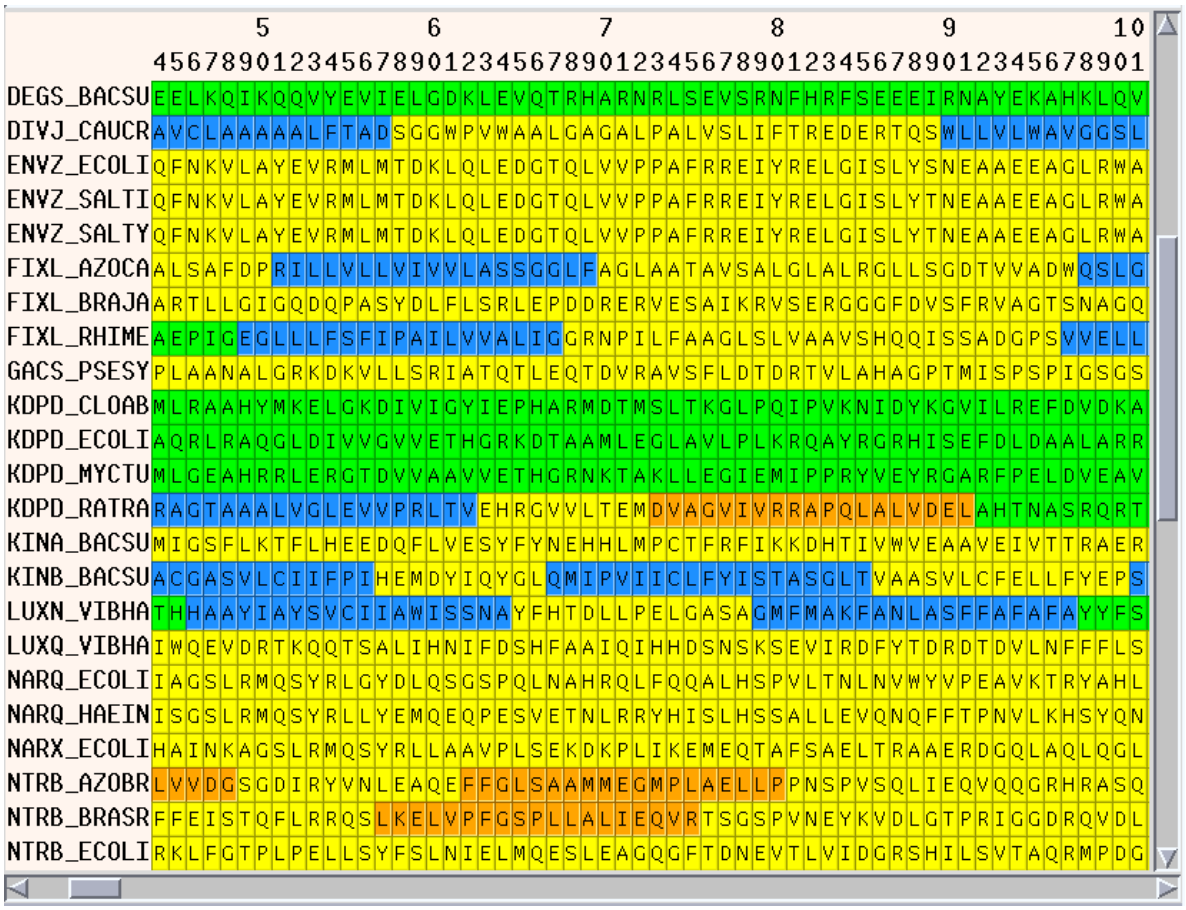


FIG. 5.9 – Segments transmembranaires (Toppred).

- l'occurrence de motifs selon des algorithmes divers ;
- l'occurrence d'une lettre dans une ligne ou une colonne ;
- des codons non synonymes par rapport à une séquence de référence ;
- etc...

Cette liste n'est pas limitée, car l'éditeur fournit un mécanisme de programmation des fonctions de visualisation.

2. **Tags**

Ces fonctions de visualisation sont définies comme des "tags", qui sont un étiquettage graphique de valeurs associées à des positions dans l'alignement. Ces valeurs dépendent du tag, cela peut être : des niveaux d'identité, une valeur dans une liste (purine, pyrimidine) ou (certain, putatif), une valeur booléenne, un entier correspondant aux nombre d'erreurs dans une occurrence de motif. Ces valeurs sont calculées par une méthode, de nom value, que l'utilisateur peut éditer.

L'utilisateur peut modifier un tag, par exemple en changeant les propriétés graphiques associées aux valeurs, ou créer un nouveau tag, en passant par un éditeur de tags (figure 10).

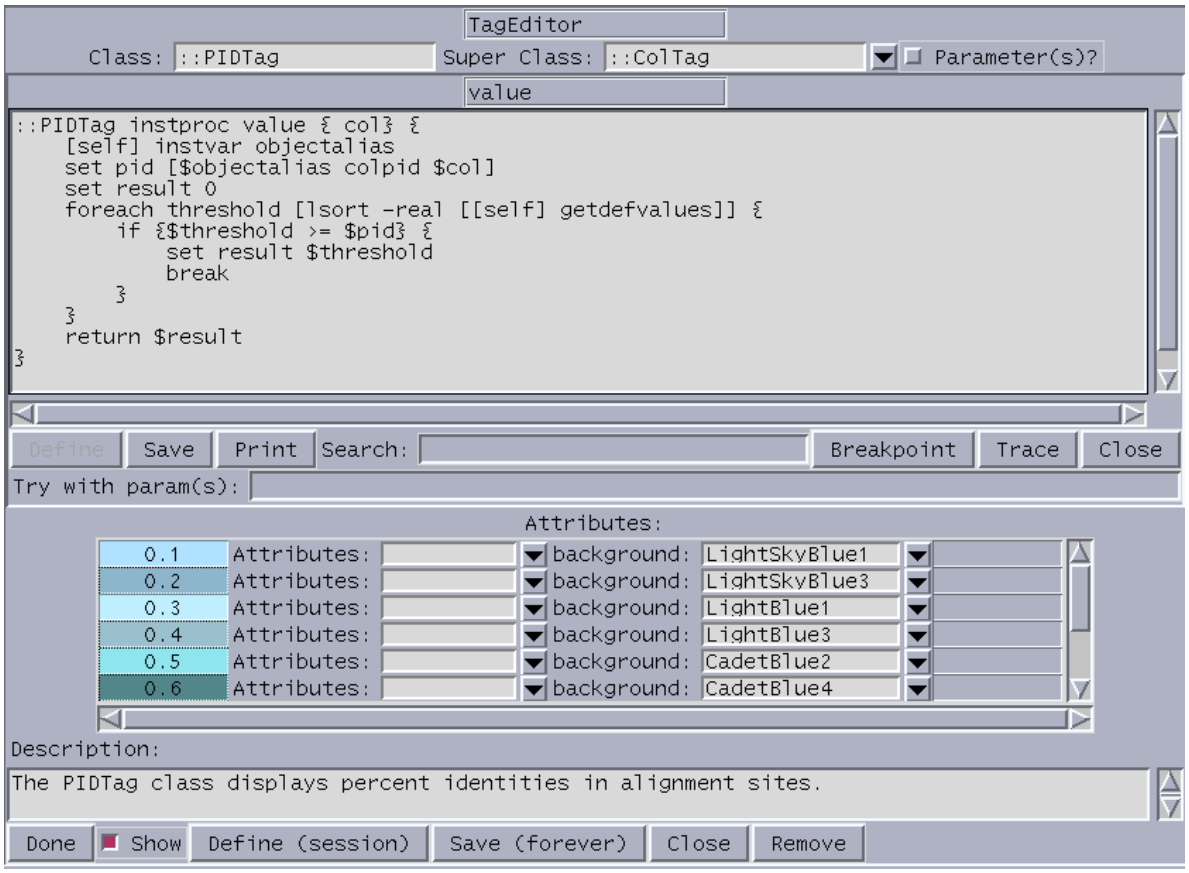


FIG. 5.10 – Editeur de tags.

Dans cet exemple, la valeur (une liste prédéterminée de seuils d'identités) associée à une cellule est obtenue par la méthode affichée dans l'éditeur de code, qui prend pour paramètre le numéro de colonne. Ces méthodes sont plus complexes qu'une formule, puisqu'elles prennent un paramètre et comportent en général plusieurs instructions. Mais elles sont de petite taille, les variables à utiliser sont pratiquement toujours les mêmes, et leur sémantique est toujours la même. Ce sont des occasions typiques de s'inspirer des exemples existants.

3. **Zoom**

Une fonction de zoom permet d'obtenir une vue globale sur les caractéristiques de l'alignement, comme l'occurrence de motifs. La figure 11 montre l'occurrence d'un motif de TATA box dans des séquences d'E. Coli.

FIG. 5.11 – Occurrences de TATA box.

4. **Selection, annotation**
   Il est possible de sélectionner des zones de forme libre - non rectangulaires : cette fonction est très demandée dans les ateliers de brainstorming (figure 12).

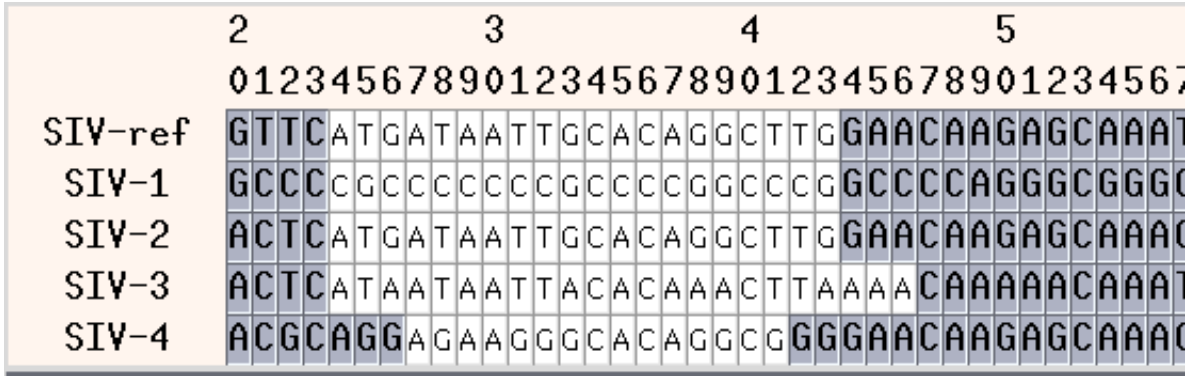


Fig. 5.12 – Sélection non rectangulaire.

On peut associer un nom à une sélection (figure 13).

Fig. 5.13 – Nom pour une sélection.

ce qui la rend accessible, sous forme d'une liste de valeurs et de leurs coordonnées, par d'autres objets. Il s'agit en somme d'un tag "utilisateur".
Il est également possible d'annoter une telle zone (figure 14).

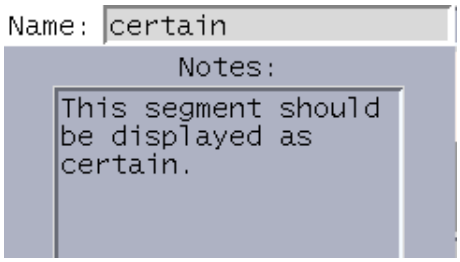


Fig. 5.14 – Annotation.

et ces notes sont interrogeables par une commande :

```
searcharea motif
```

qui retourne la liste des zones dont la note contient ce motif (figure 15), tandis que la commande :

```
searcharea motif couleur
```

la mettra en évidence avec la couleur demandée (cette fonction est en cours de développement).

5. **Recherche de motifs**
   La recherche de motifs, qui est une des fonctions d'analyses les plus étudiées au cours des ateliers de conception, repose dans le prototype sur ce mécanisme général de tags. Cette souplesse a permis d'explorer plusieurs sortes de recherche de motifs :

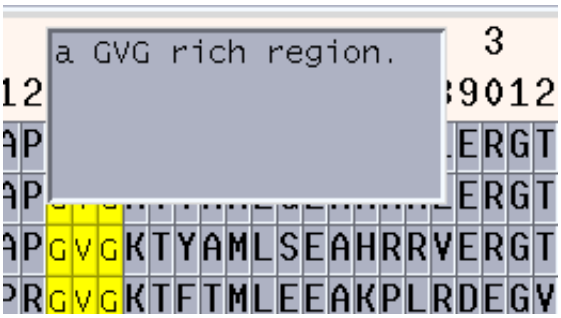


FIG. 5.15 – Recherche dans les annotations.

(a) **Recherche par expressions régulières** (en utilisant la commande regexp de Tcl).

(b) **Recherche avec erreurs** et spécification de plusieurs blocs séparés par un certain intervalle (en utilisant le programme grep+ [MS01]). Par exemple (figure 16), ce motif :

```
2 {1 GAGA} 1-2 {1 AG}
```

décrit la recherche de deux blocs séparés par une ou deux positions ; chaque bloc peut contenir une erreur, et l'ensemble de l'occurrence peut comporter deux erreurs au maximum :
.

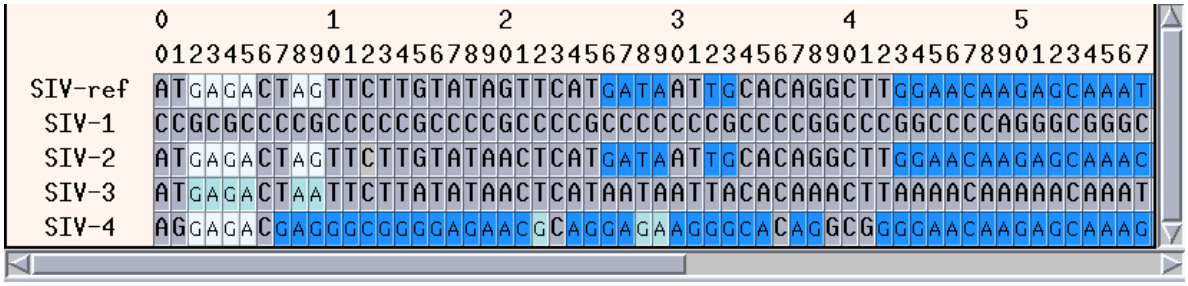


FIG. 5.16 – Recherche de motifs avec erreurs.

La couleur souligne le nombre d'erreurs (figure 17). La visualisation du motif de TATA box :

```
3 {2 TTGACA} 15-19 {2 TATAAT}
```

de la figure 11 a été obtenu par cette fonction.

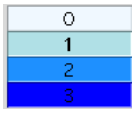


FIG. 5.17 – Nombre d'erreurs.

On remarque que la syntaxe proposée ici n'est pas des plus simples. Parmi les développements qui pourront être réalisés, figurent notamment :

- une fonction d'inférence du motif en fonction des données sélectionnées par l'utilisateur dans l'alignement (fonction de type programmation par démonstration) ; cette fonction avait été suggérée par l'un des participants de l'atelier autour du prototype informatique (cinquième atelier) ; elle pourrait même être étendue à une interaction plus générale avec l'algorithme d'extraction de motifs vu à l'occasion du premier atelier, où en plus des possibilités de navigation, l'utilisateur pourrait sélectionnner par les motifs inférés ceux pouvant servir à des recherches plus ciblée ;
- une interface utilisateur tenant compte de la maquette réalisée pendant l'un des ateliers de prototypage (chapitre II, troisième atelier) : ce qui manque dans l'implémentation actuelle par rapport à cette maquette, c'est la possibilité de synthétiser plusieurs recherches

dans une fenêtre unique ainsi qu'un instrument de type "slider" pour manipuler le paramètre numérique indicant le nombre maximum d'erreurs ; sur le plan fonctionnel, la fonction implémentée est d'une puissance équivalente à celle qui a été spécifiée pendant l'atelier, notamment pour donner un contexte de recherche à un motif. Un des participant avait également suggéré la possibilité recherche avec erreur, sauf sur certains caractères (indiqués par exemple en majuscules) :
`LPXtgX(*)[kr].`

De manière générale, dans la mesure où l'éditeur de tags est à la fois un éditeur donnant accès à la programmation ou à la personnalisation et une interface utilisateur pour l'utilisation des tags, il faudra envisager la possibilité de classes plus diversifiées pour la partie vue et contrôle de cet éditeur, permettant des interactions plus appropriées selon chaque type de fonction de visualisation. Une des possibilités est d'utiliser les fonctions dataflow qui permettent de lier la zone générique de paramètrage qui existe actuellement à d'autres objets graphiques, comme l'outil d'affichage de courbes. Cet outil comprend en effet une fonction qui permet d'associer une ligne droite à un objet numérique : en faisant glisser la droite à la souris, par exemple pour déterminer visuellement un seuil dans une courbe, on modifie simultanément le paramètre du tag, à la manière d'un slider (figure 18).

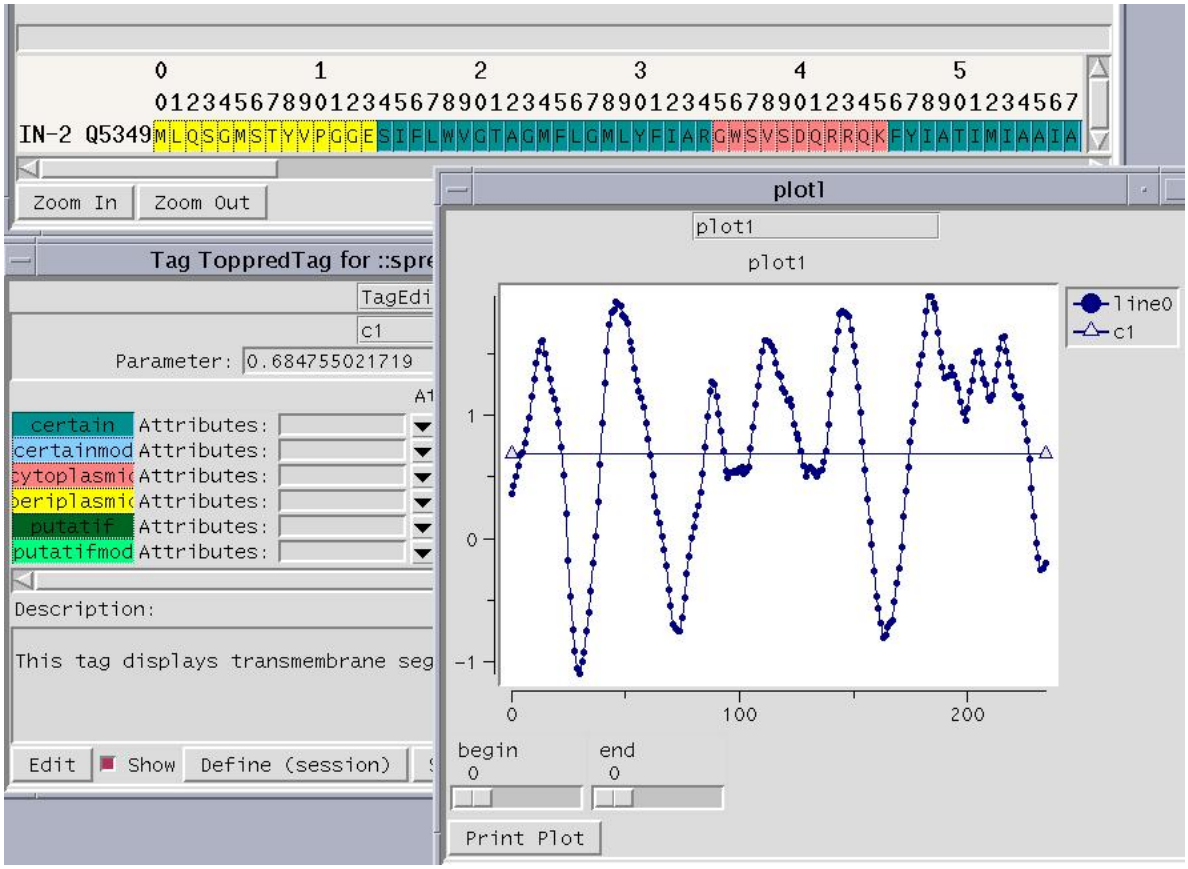


FIG. 5.18 – Interface utilisateur pour les fonctions de visualisation.

(c) **Recherche de motifs connus**, par exemple répertoriés dans une banque de données (ici Prosite [Bai91]) ; dans la figure 19, l'occurrence du motif de la banque (PS00890) est soulignée en bleu-vert, et se superpose à l'occurrence d'un segment transmembranaire en bleu foncé.

Le motif de la banque est assez complexe :

```
[LIMST]-x(2)-[LIMW]-x(2)-[LIMCA]-[GSTC]-x-[GSAIV]-x(6)-[LIMGA]\
 -[PGSNQ]-x(9,12)-P-[LIMFT]-x-[HRSY]-x(5)-[RQ]
```

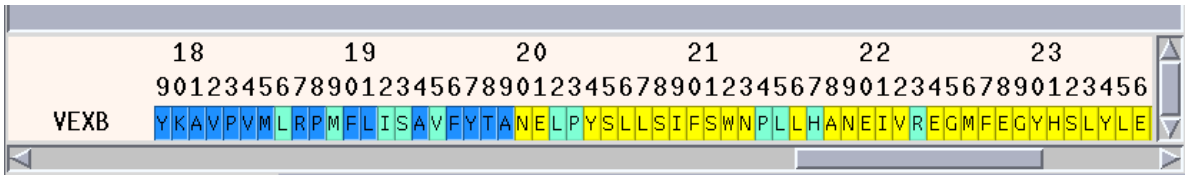


FIG. 5.19 – Motifs Prosite.

L'intérêt de ces motifs est d'être bien documentés. La banque signale également les autres protéines qui possèdent une occurrence de ce motif.

(d) recherche de motifs sur un cadre de lecture particulier (figure 20).

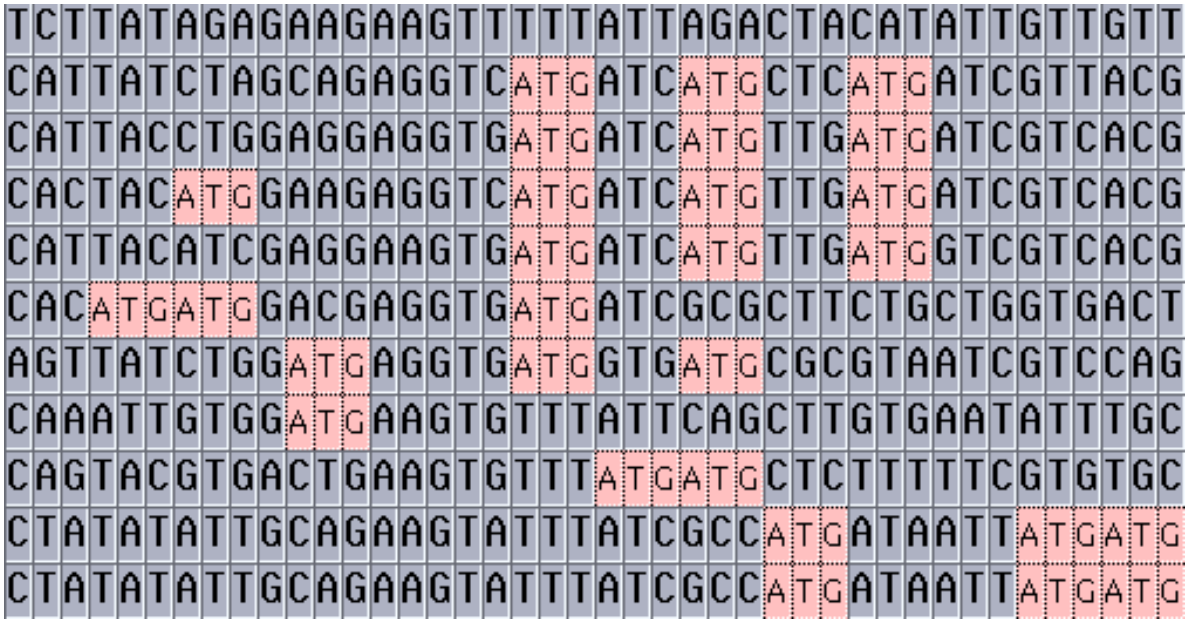


FIG. 5.20 – Recherche sur un cadre de lecture.

(e) **Recherche de lignes ou de colonnes contenant une occurrence** (permet par exemple de filtrer les sites d'un alignement avec l'occurrence d'un acide aminé particulier).

(f) **recherche dans les annotations**.

6. **Analyse de séquences**

(a) **Analyse multiple de séquences**
Tous les programmes d'analyse disponibles par le générateur d'interface Pise sont accessibles. L'analyse peut être appliquée soit à tout l'alignement, soit sur la sélection, soit sur les zones correspondant à un tag.

(b) **Analyse d'une séquence**
L'objet graphique Sequence est un éditeur en mode texte permettant de lancer des analyses, comme l'affiche du brin complémentaire (programmé au bout de 2 jours dans l'environnement et sans connaissance préalable du langage par un étudiant) (figure 21) ou la traduction en protéine (figure 22).

(c) **Chargement depuis une banque**
La séquence peut être chargée depuis une banque de séquences en mettant son identifiant dans la formule (programme GOLDEN, de Nicolas Joly) (figure 23).

(d) **Extraction d'une sortie de programme**

Fig. 5.21 – Brin complémentaire.

Fig. 5.22 – Traduction.

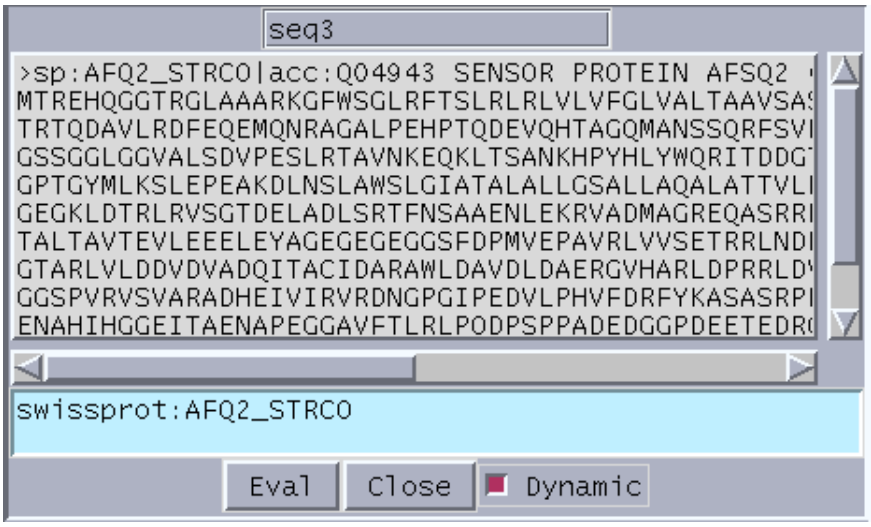


FIG. 5.23 – Chargement depuis une banque.

Les résultats d'un programme peuvent être affichés sous la forme d'un tag. Dans la figure 24, les hélices et les feuillets trouvés dans la séquence par le programme PREDATOR [FA96][FA97], appelé par Pise, sont soulignés en bleu clair et en rose.

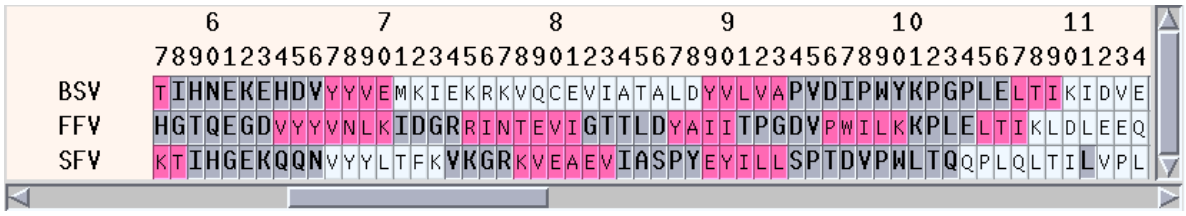


FIG. 5.24 – PREDATOR.

Pour obtenir cet affichage, il faut extraire les positions correspondantes dans la séquence en analysant le résultat du programme.

(e) **Comparaison visuelle d'analyses**

On peut afficher des résultats de plusieurs analyses effectuées sur la même séquence (figure 25). Cette fonction, inventée lors de la préparation d'un atelier, est plutôt simple à implémenter, mais ce qui est important, c'est que l'outil, l'éditeur d'alignement qui est en réalité une sous-classe de tableur, n'est pas trop contraint pour que ce type d'invention puisse avoir lieu. JaMBW [Tol97] propose également ce type de co-visualisation avec l'applet "Feature viewer".

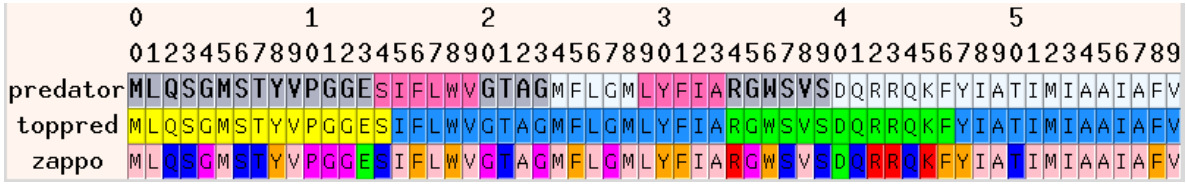


FIG. 5.25 – Analyses juxtaposées.

7. **Alignments**

Les programmes d'alignement multiples accessible par Pise (clustalw, dialign2, dca, ...) peuvent être utilisés pour ré-aligner tout ou partie des séquences dans l'éditeur (figure 26).

```
         1         2         3         4         5
345678901234567890123456789012345678901234567890123
LLHPYPHKMPALRPLQVEIKGNHLKGYWDSGAEITCVPAIYIIEEQPVG
LKTKLKMDLLKPLTVERKGVKIKGYWDSQADITCVPKDLLQGEEPVRQQ
VPRPPQMNPLQLLQPLPAEVKGTKLLAHWDSGATITCIPESFLEDEQPI
         1         2         3         4         5
345678901234567890123456789012345678901234567890123
LLHPYPH-KMPALRPLQVEIKGNHLKGYWDSGAEITCVPA--IYIIEEQP
LKTKLKM---DLLKPLTVERKGVKIKGYWDSQADITCVPKDLLQGEEPVR
VPRPPQMNPLQLLQPLPAEVKGTKLLAHWDSGATITCIP---ESFLEDEQ
```

FIG. 5.26 – Ré-alignement de la zone sélectionnée.

La zone graphique correspondant au résultat est accessible sous le nom du programme choisi, comme clustalw, et il est disponible sous forme programmatique ; par exemple, la commande :

```
[spread3 clustalw] fasta
```

renvoie la zone qui vient d'être alignée en format FASTA (spread3 est l'alias de l'objet éditeur). Ce format est un des formats les plus simples : il comprend une ligne commençant par un caractère '>' et contenant le nom et les informations diverses, suivie d'une ou plusieurs lignes contenant la séquence.

8. **Filtrage, extraction**

   Il est possible d'extraire le contenu d'un éditeur pour le charger dans un autre, en utilisant un tag, par exemple (figure 27) le tag filterM qui colore en orange les sites contenant un "M".

```
                 8         9         10        11        12        13
                 901234567890123456789012345678901234567890123456789013
      ColE1      PVAKLNLFMLYNLLCLVLAIPFGLLELFISIKNN
BACTERIORHODOPSINTMVPFGGEQNPIYWARYADWLFTTPLLLLDLALLVDADQGTILALVGADGIMI
```

FIG. 5.27 – Filtrage.

a servi pour initialiser un autre tableau :

9. **Tableur**

   Comme cet éditeur est aussi un tableur, il est possible de définir le contenu d'une cellule ou d'une ligne par une formule. L'exemple de la figure 29 montre une formule pour calculer sur la 4ème ligne le consensus des trois premières lignes. La formule est :

```
cons C(0..i-1,j)
```

   où cons est une méthode qui calcule le consensus de ses arguments, et où l'expression C(0..i-1,j) représente les cellules de la même colonne (j) des lignes 0 à la ligne juste avant la ligne de la formule (i-1). Pour entrer la formule, on peut double-cliquer sur la cellule, ou bien faire un copier-coller de plusieurs cellules. Les mécanismes de saisie de formules et la syntaxe de désignation des cellules sont à améliorer.

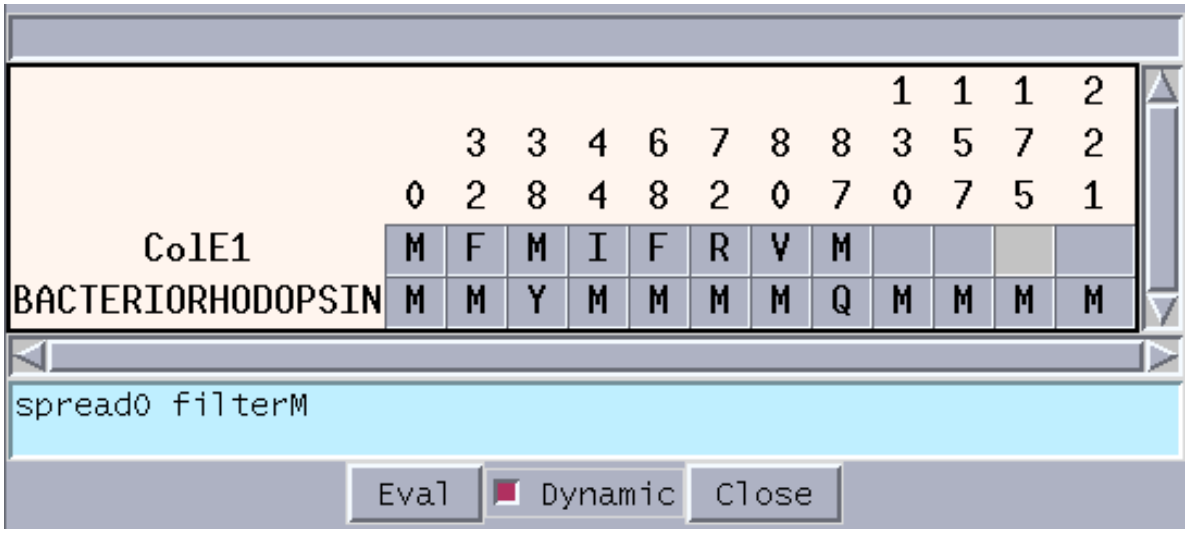


FIG. 5.28 – Extraction.

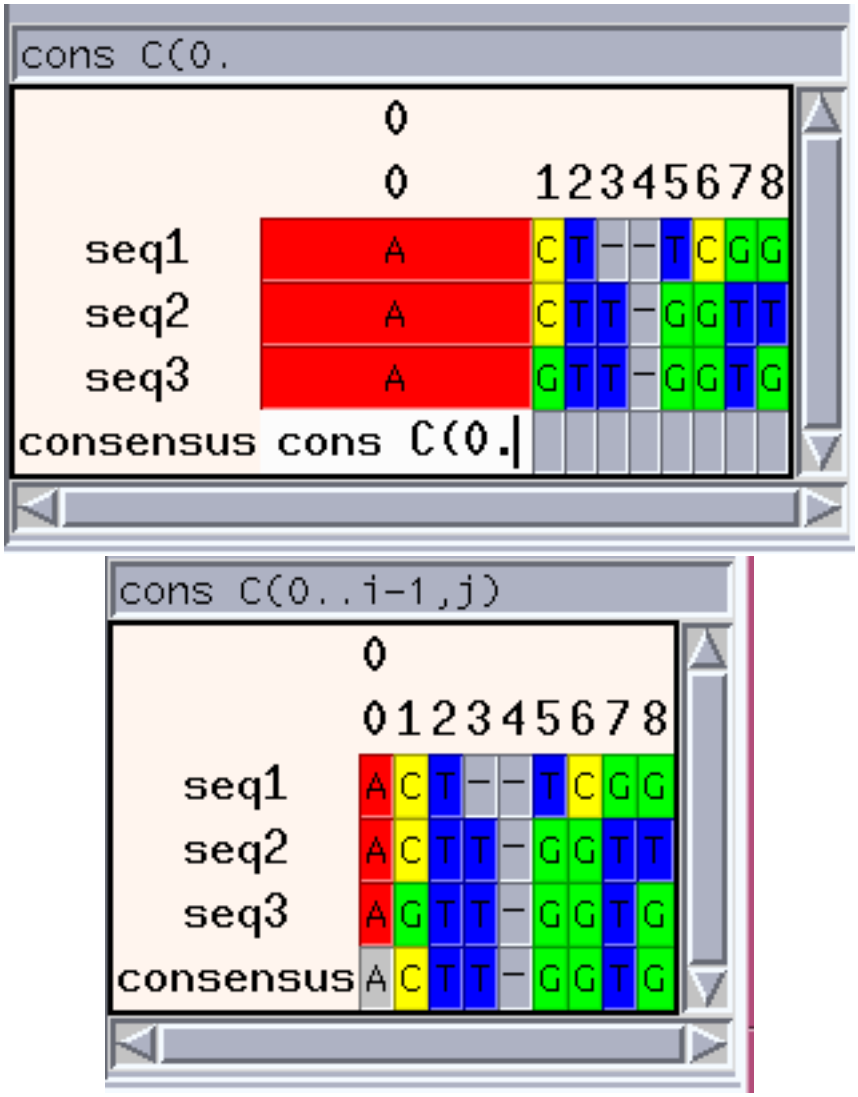


FIG. 5.29 – Formule pour calculer le consensus.

10. **Manipulation des structures de données**

L'éditeur d'alignement, comme tous les objets graphiques dans biok, comporte une interface programmatique pour manipuler l'éditeur et les données de l'alignement. On peut notamment extraire les blocs correspondant à la zone définie par un tag sous la forme d'un objet XOtcl et les manipuler par des méthodes. Ici, on extrait les segments transmembranaires de niveau certain, et on affiche le titre de la séquence et les coordonnées de chaque segment :

```
% set area [spread0 ToppredTag certain]
% foreach block [$area blocks] {
   puts "row: [$area blockrow \$block] title: [$area blocktitle $block] \
    range: [$area blockrange $block]"
}
row: 0 title: ColE1 range: 6 24
row: 0 title: ColE1 range: 32 50
row: 0 title: ColE1 range: 91 109
row: 1 title: BACTERIORHODOPSIN range: 23 41
row: 1 title: BACTERIORHODOPSIN range: 56 74
row: 1 title: BACTERIORHODOPSIN range: 97 115
row: 1 title: BACTERIORHODOPSIN range: 123 141
row: 1 title: BACTERIORHODOPSIN range: 151 169
row: 1 title: BACTERIORHODOPSIN range: 186 204
row: 1 title: BACTERIORHODOPSIN range: 218 236
%
```

Une méthode a é prévue pour obtenir ces mêmes segments en format FASTA :

```
% $area fasta
> ColE1 (6-24)
IKNILFGLYCTLIYIYLIT
> ColE1 (32-50)
FLVSDKMLYAIVISTILCP
> ColE1 (91-109)
LLCLVLAIPFGLLELFISI
> BACTERIORHODOPSIN (23-41)
IWLALGTALMGLGTLYFLV
> BACTERIORHODOPSIN (56-74)
AITTLVPAIAFTMYLSMLL
> BACTERIORHODOPSIN (97-115)
DWLFTTPLLLLDLALLVDA
> BACTERIORHODOPSIN (123-141)
LVGADGIMIGTGLVGALTK
> BACTERIORHODOPSIN (151-169)
AISTAAMLYILYVLFFGFT
> BACTERIORHODOPSIN (186-204)
LRNVTVVLWSAYPVVWLIG
> BACTERIORHODOPSIN (218-236)
LLFMVLDVSAKVGFGLILL
```

On a vu plus haut la variété des types de recherche de motifs, qui justifierait certainement la conception d'un outil de spécification générique, reposant sur un langage et muni d'un niveau de langage plus interactif. A l'issue de l'atelier de prototypage de ces fonctions de recherche de motifs, une discussion a été menée sur le site wiki aboutissant d'ailleurs à des suggestions conernant un tel langage. Il s'agissait d'un langage de manipulation de régions, qu'elles soient définies par l'utilisateur ou par des fonctions de visualisation :

```
area8=between(1,area1)
x=inside(1..10,15..25) (positions)
```

C'est en réalité ce niveau d'expression qu'on obtient avec cette interface programmatique pour les zones graphiques, puisque le langage est Tcl et que toutes les informations nécessaires, dont les positions, sont accessibles.

11. **Exportation dans d'autres formats**
    biok utilise le paquetage SEQIO (`http://www.cs.ucdavis.edu/~gusfield/seqio.html`), ce qui permet d'opérer des conversions depuis et vers la plupart des formats connus (figure 30). Cet aspect peut sembler d'un intérêt secondaire : c'est bien au contraire l'une des fonctions sollicitées en tout premier lieu par les biologistes (voir par exemple les choix prioritaires à l'issue de l'atelier de brainstoming sur l'éditeur d'alignement décrit dans le chapitre II - 2ème atelier). Le problème des biologistes, c'est que justement ces fonctionalités sont souvent considérées comme inintéressantes par le bio-informaticiens. Or, la moitié du temps passé à l'analyse de séquences consiste à effectuer ces conversions de formats, qui sont des opérations sans intérêt scientifique pour les biologistes. La situation n'est par ailleurs pas facilitée par l'absence de standardisation et les variations d'interprétation des formats d'un logiciel à l'autre. Des solutions de type XML, bénéficiant de la lisibilité du texte, de la simplicté de la syntaxe, et de l'aspect explicite et auto-descriptif du format sont de plus en plus adoptées pour traiter ce problème.

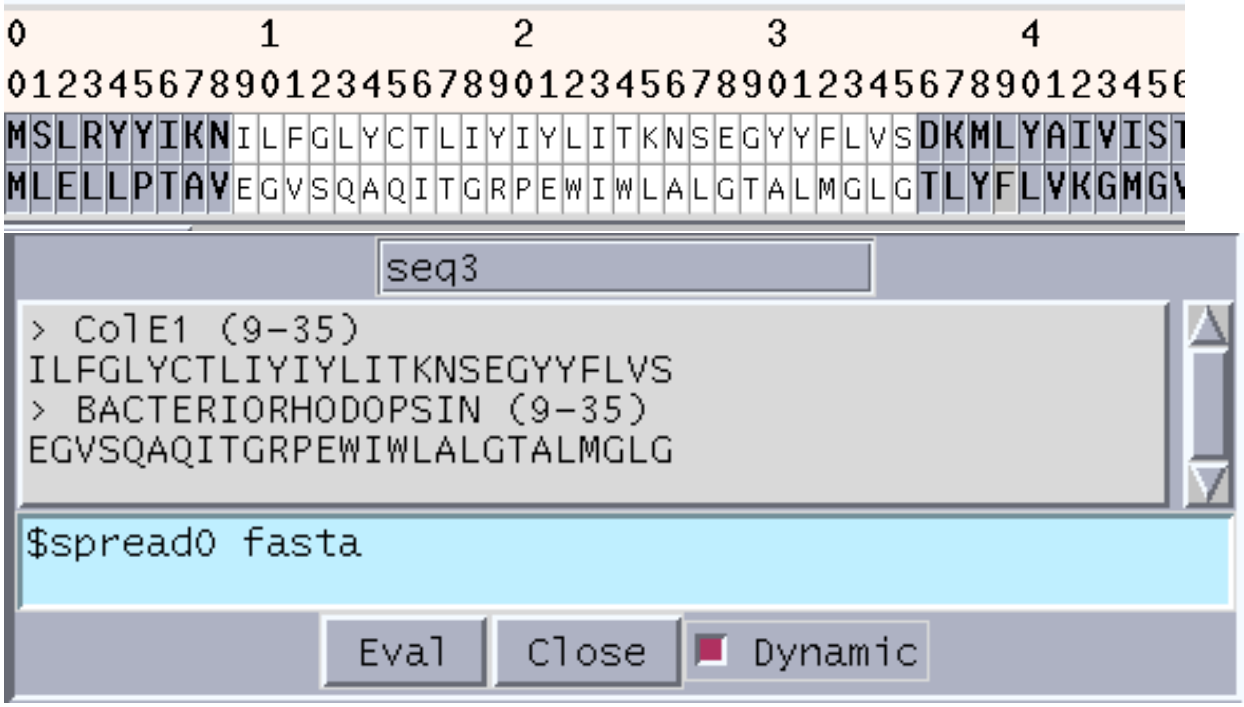


FIG. 5.30 – Conversion de format.

12. **Autres fonctions**
    Le widget à partir duquel l'éditeur est implémenté, tkTable, est lui-même assez riche, et nous n'avons pas encore exploité toutes ses possibilités, comme la possibilité de créer des zones composites, d'incorporer des fenêtres (cela pourrait servir à afficher une courbe, comme une trace ABI), etc...

5.4.1.4 Conception

1. **Classes**

L'éditeur d'alignement est une sous-classe des classes Spreadsheet et AligData. AligData utilise à son tour la classe SequenceData pour lancer les analyses (figure 31).

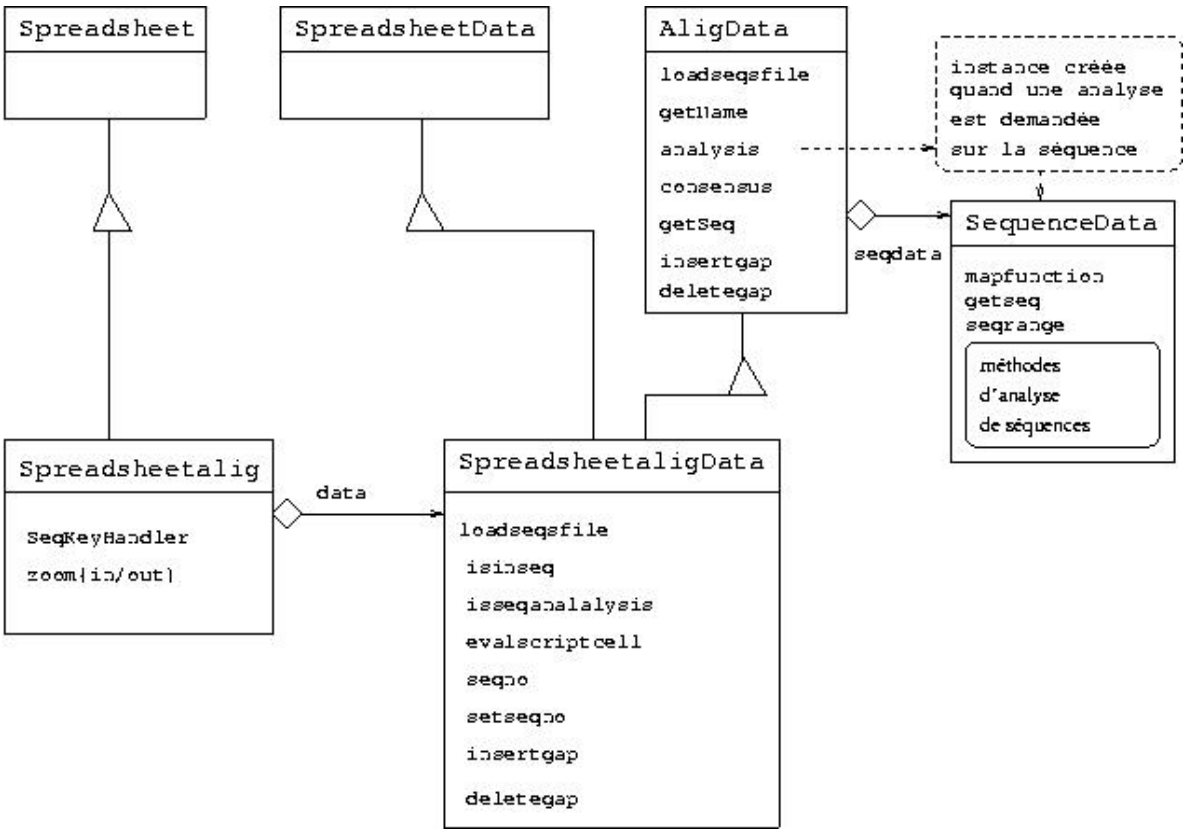


FIG. 5.31 – Classes de l'éditeur d'alignement.

2. **Tags**

   Le diagramme de la figure 32 montre les classes concernées par la définition et la visualisation de tags. La classe TagEditor est à la fois une interface utilisateur et un éditeur (comme tous les objets graphiques de biok). Il est instancié par le tableur à la demande de chargement ou d'édition d'un tag. La visualisation effective se produit quand on évalue le tag (par un appel à sa méthode evalscript), ce qui est fait soit au chargement, soit par l'éditeur de tag dès que l'utilisateur a défini suffisamment d'informations pour cela.

   Une fois le tag chargé, l'utilisateur peut, si par exemple il en a besoin pour programmer, obtenir la zone correspondant au tag par une commande de type :

   ```
   spread0 tagname a_value
   ```

   qui active la méthode gettag et crée une instance de la classe SpreadsheetArea (non montrée dans le diagramme). Dans le diagramme de la figure 32, les flèches en pointillés représentent certaines des invocations de méthodes sur lesquelles repose le chargement des tags.

3. **Configuration du widget tkTable**

   Le widget tkTable n'est pas vraiment configuré pour fonctionner comme un tableur (malgré ce qui est dit dans la documentation). Il a fallu essayer un certain nombre de combinaisons des variables de configuration du widget, ainsi que des valeurs de retour des fonctions liées aux évènements clavier avant de pouvoir afficher dans une cellule un résultat différent de ce qui avait été entré par l'utilisateur.

   Les différents éléments de configuration sont les options suivantes :

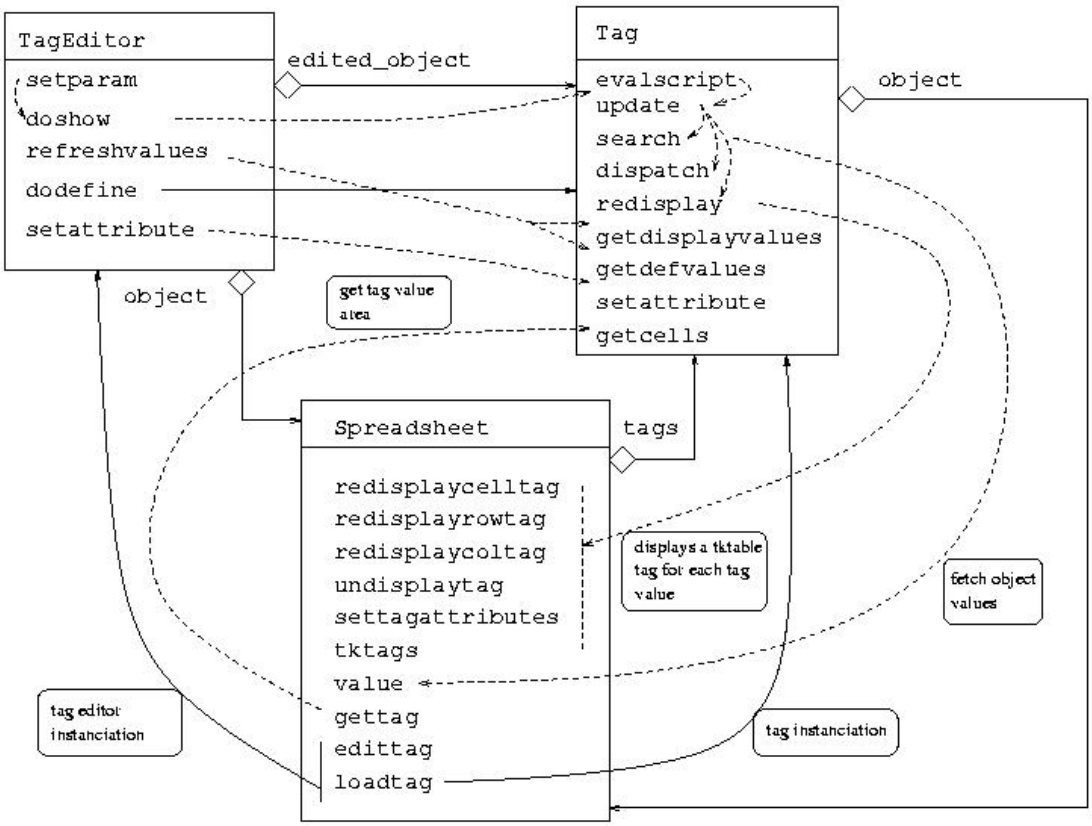


FIG. 5.32 – Classes des tags.

(a) variable : un array Tcl (un tableau associatif) ;
(b) browsecommand : commande qui est exécutée quand on se place sur une cellule ;
(c) selectioncommand : commande pour le copier-coller de cellules ;
(d) validatecommand : commande exécutée pour valider le contenu d'une cellule ; cette commande doit renvoyer une valeur booléenne indiquant si l'entrée est acceptée ou non, et si non, la nouvelle saisie est annulée ; la figure 33 illustre la façon dont nous l'avons utilisé :

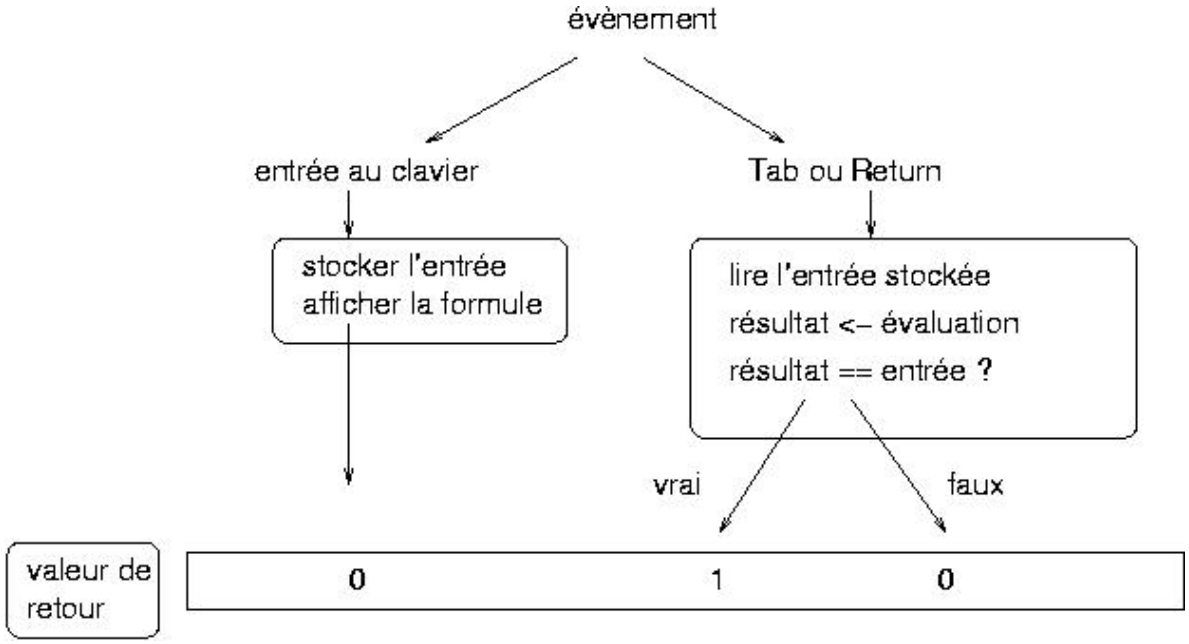


FIG. 5.33 – Configuration de tkTable.

5.4.2 Architecture

Cette section décrit de manière plus détaillée les mécanismes de fonctionnement de biok.

5.4.2.1 Vues et données

Un objet graphique est une "vue", au sens de MVC, et a un objet "modèle" associé, que nous avons appelé data avec une convention de nommage selon laquelle la classe pour la partie données est suffixée par Data : SequenceData pour Sequence, etc...

Les classes de données héritent toutes de la classe Data, qui définit les protocoles génériques pour gérer les dépendances (figure 34). Le protocole de mise à jour des vues lors de la modification des données repose sur l'appel de la méthode redisplay par la méthode générique update définie dans la classe Data. La méthode draw sert pour le dessin initial de l'objet, et est appellée par la méthode init, qui est automatiquement appellée, si elle existe, par XOtcl lors de la création d'une instance.

Les procédures génériques de gestion et de détection des données de dépendances entre les objets sont définies par la classe Data et sont décrites plus loin.

5.4.2.2 Création des objets

Les objets sont créés de plusieurs façons :

1. par un sous-menu new du menu général, construit par introspection de toutes les sous-classes de la classe GraphObject ;

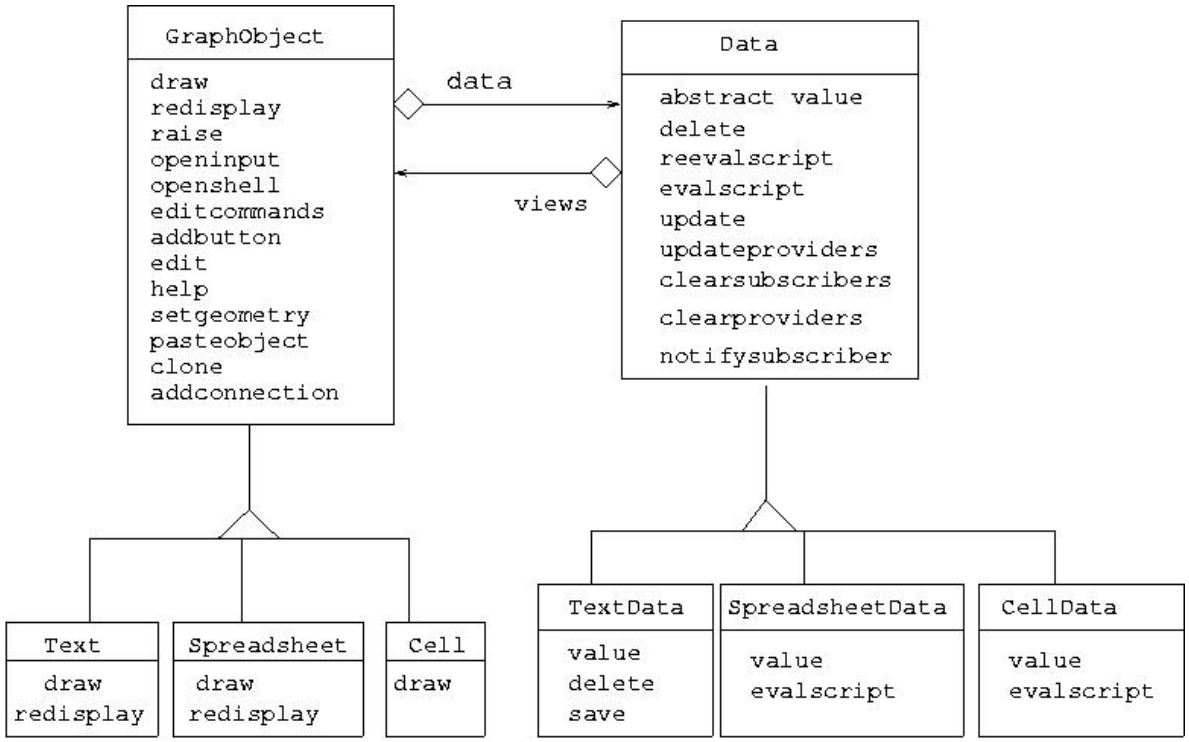


FIG. 5.34 – Vues et données.

2. par un objet builder, sorte de fabrique concrète, qui gère la création automatique de la fenêtre ;
   – la commande suivante :

   ```
   set plot [builder new Plot -name hydro -title "Hydrophobicity plot"]
   ```

   entrée depuis l'interpréteur ou invoquée depuis un script, crée un objet graphique de classe Plot avec pour alias hydro ; la commande renvoie l'identifiant interne de l'objet, ce qui permet de le manipuler dans la suite du script le cas échéant (ici par la variable plot) ;
   – la méthode new de l'objet builder possède des options permettant d'incorporer le nouvel objet dans un autre objet : -into, ainsi qu'une option -data pour associer une nouvelle vue à un objet existant ;
3. par le menu connect des objets, qui crée un nouvel objet avec une formule pré-initalisée ; ce menu résume toutes les formules utilisant ce type d'objet ;
4. par clonage : il est possible de cloner un objet ou de le copier-coller dans un autre, par exemple dans un texte ou dans une fenêtre à panneaux ; la copie peut-être un objet indépendant ou une vue supplémentaire sur l'objet copié ;
5. par l'instruction d'instanciation standard de XOtcl en précisant explicitement la fenêtre où placer l'objet graphique :

   ```
   Plot [names autoname hydro] -into $window
   ```

   l'objet names est le serveur de noms qui gère la correspondance entre les différents espaces de nommage (widgets Tk, alias et noms internes XOtcl), décrit plus loin ; autoname est une fonction de XOtcl pour générer de nouveaux noms ;
6. il est possible de créer uniquement la partie données sans contrepartie graphique, notamment pour n'utiliser que l'interface programmatique ; la commande suivante crée une instance d'interface programmatique pour lancer des jobs clustalw (un logiciel d'alignement multiple) par le système Pise :

```
PiseData $pise -command clustalw
```

5.4.2.3 Dépendances

Dans l'exemple de la figure 35, l'objet de nom plot0 dessine la courbe d'hydrophobicité de la séquence seq0. La formule de l'objet plot0 :

```
[seq0 seqrange] [seq0 window_hydrophobicity]
```

définit deux listes : la première liste contient les valeurs de l'axe des x, la seconde définit les valeurs de la courbe. Pour superposer d'autres courbes, il suffit de spécifier des listes supplémentaires.

Lorsque la valeur de seq0 change, il faut re-calculer la courbe.

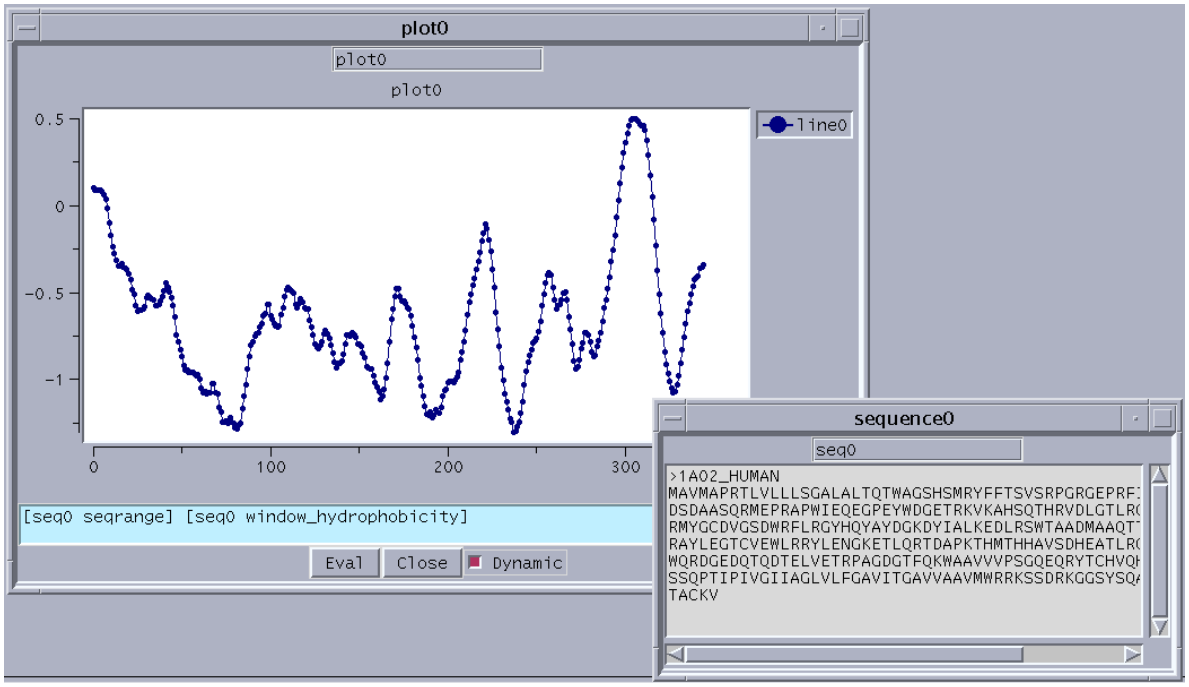


FIG. 5.35 – Courbe d'hydrophobicité.

A cette fin, un patron de conception observateur a été implémenté, et pour cela, nous avons utilisé le mécanisme de mixin fourni dans XOtcl [NZ99c].

Par exemple, pour associer un mixin UpdateMixin à un objet de nom seq0, il faut le déclarer de cette façon :

```
seq0 mixin UpdateMixin
```

La figure 36 montre les étapes de mise à jour.

1. le script de seq0 est modifié, sa formule est ré-évaluée (méthode evalscript),
2. la valeur de l'objet est mise à jour par l'appel de la méthode update,
3. l'appel de next, supposé invoquer la méthode update de la super-classe Data, est intercepté par le mixin UpdateMixin,
4. ce dernier appelle d'abord la méthode update de la super-classe,

5. qui a pour fonction de rafraîchir les vues de seq0 en appelant leur méthode redisplay ;
6. le mixin propage ensuite la mise à jour aux objets clients (subscribers) de seq0, ici plot0, en invoquant leur méthode update, qui dans l'exemple consiste à recalculer la courbe,
7. et qui va également ré-afficher les vues, ici ré-afficher la courbe.

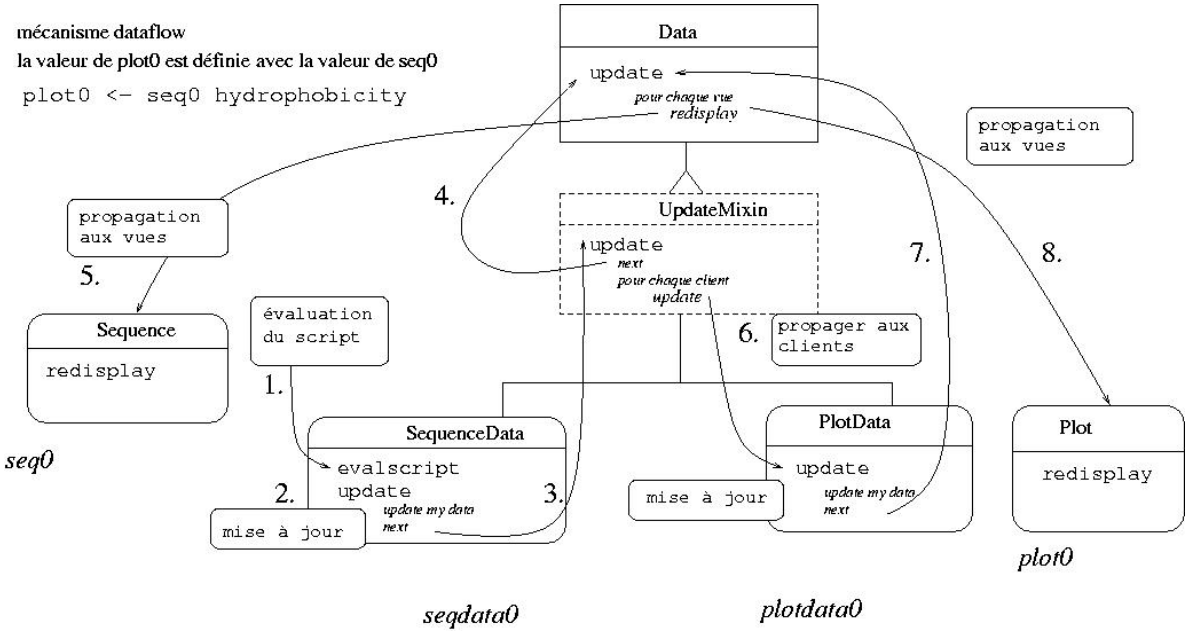


FIG. 5.36 – Mises à jour automatiques.

Si la formule de plot0 est modifiée, et que cet objet n'utilise plus seq0 comme base de calcul, il faut alors simplement supprimer le mixin de la liste des mixins de l'objet seq0.

Le système de détection des dépendances repose sur des techniques réflexives, décrites dans la section suivante.

5.4.2.4 Nommage

Il y a plusieurs espaces de nommage dans biok : l'espace des alias, un espace global et plat, l'espace des objets XOtcl et l'espace de nom des widgets Tk.

La figure 37 montre comment les différents espaces de nommage sont mis en correspondance à travers l'objet names qui est une sorte de serveur de noms :

1. la méthode actualnames permet de retrouver le nom interne correspondant à un alias ;
2. deux tableaux gèrent la correspondance actualname et aliasname ; les adaptateurs Tcl sont décrits dans la section 5.4.2.1 ;
3. un widget Tk connaît l'objet graphique auquel il appartient par la méthode object de l'objet names ; ce lien est indispensable pour relier les évènements graphiques et les menus à un objet particulier ;
4. un objet graphique connaît les widgets Tk qui le composent par l'intermédiaire d'une structure qui suit le patron composite, et dont la racine est un objet appellé frame ; cette structure composite est constituée de deux classes : Widgets et Widget.

5.4.3 Réflexivité

L'architecture de biok repose en partie sur une conception réflexive, et ce, essentiellement pour trois aspects :

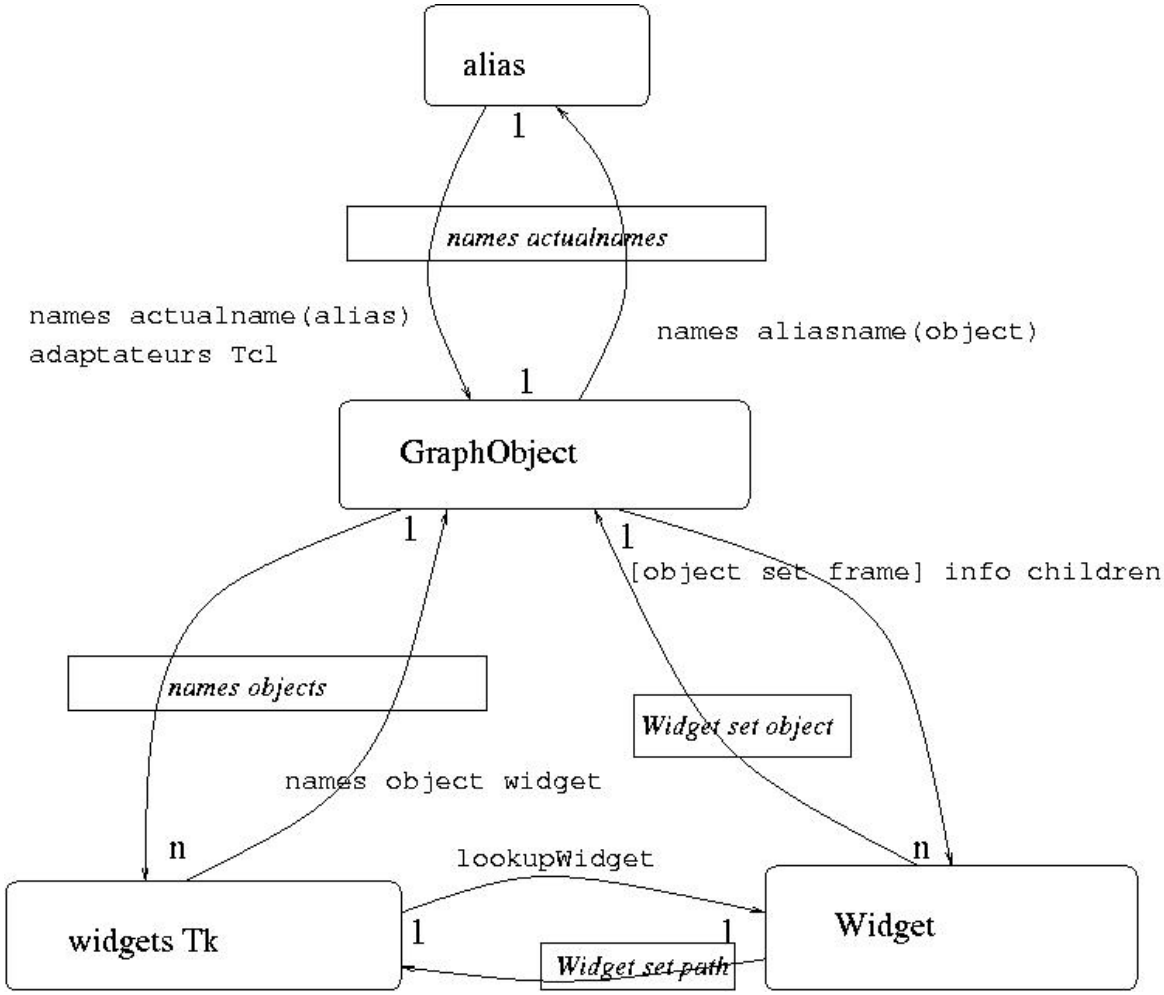


FIG. 5.37 – Correspondance des espaces de nom.

1. la mise en place des dépendances dataflow entre les objets, déterminées par les formules,
2. l'environnement de programmation,
3. certains éléments de l'interface utilisateur.

5.4.3.1 Détection des dépendances

[SN99] suggère d'utiliser la réflexion comportementale pour la construction d'architectures de composition, par l'interception de messages pour la redirection, la délégation, le blocage, etc comme alternative aux adaptateurs (wrappers) pour l'extensibilité et la reconfiguration dynamique. Ce qu'il appelle les ADL (Architectural Description Languages) pallient le manque de formalisation des langages de script modernes, qui par ailleurs possèdent suffisamment de fonctions : eval, chargement dynamique, introspection restreinte.

Dans cet ordre d'idée, nous utilisons une conception réflexive pour détecter les dépendances entre les objets reliés par des formules et mettre en place les éléments de mise à jour automatique. Nous avons exploré plusieurs techniques :

1. analyse du texte de la formule pour extraire les alias des objets source (providers) ; cette approche est assez limitée car :
   - il faut utiliser l'objet par son alias,
   - les dépendances en dehors de formules ne sont pas détectées,
   - aucune indirection n'est permise (utilisation d'un nom de variable qui contient le nom de l'objet).
2. mettre en place un adaptateur pour l'accès à chaque objet par son alias, c'est-à-dire :
   (a) associer une procédure adaptateur Tcl pour les objets XOtcl (implémentée par la méthode createaliasproc de l'objet names),
   (b) associer également une variable globale Tcl aux objets scalaires, à laquelle on associe une procédure de trace (_biok_read_trace)

   Fonctionnement :

   (a) dans le premier cas, l'adaptateur recherche l'objet appelant par la méthode introspective de XOtcl info callingobject ;
   (b) dans le deuxième cas, la procédure de trace fouille _biok_read_trace dans la pile ;
   (c) une fois repéré, l'objet appelé reçoit un signalement d'utilisation par l'objet appelant et une dépendance potentielle est détectée.

   Les limites de cette technique sont les suivantes :
   - il faut utiliser l'objet par son alias,
   - il faut déclarer ces alias comme variables globales avant de les utiliser dans une méthode (en utilisant la méthode globalias de l'objet names).
3. ajouter un mixin ValueMixin à la méthode standard d'accès value de chaque objet ; c'est cette méthode qui est toujours utilisée pour accéder à la valeur d'un objet ; cette technique ne s'applique pas aux autres accesseurs que value, à moins de spécifier d'autres méthodes spécifiques dans la classe ValueMixin ; l'approche est donc très lourde à mettre en place, et n'est pas suffisamment flexible, car elle interdit à l'utilisateur de créer autant d'accesseurs qu'il le désire, à moins de penser lui-même à ajouter la méthode équivalente dans la classe ValueMixin.

Nous avons finalement opté pour la deuxième solution, qui repose sur des mécanismes assez techniques (variables tracées, inspection de la pile, ...), mais qui est la plus souple et qui recouvre le plus de cas. La figure 38 montre les étapes de cette détection et de la mise en place des structure de données gérant les dépendances :

1. la formule est lue par la méthode evalscript,
2. evalscript appelle la méthode update qui évalue effectivement la formule,

3. l'évaluation de la formule :

   `seq0 hydrophobicity`

   invoque l'adaptateur seq0 qui retrouve l'objet appelant (plotdata0) dans la pile,
4. l'adaptateur prévient seqdata0
5. afin qu'il se signale comme fournisseur à plotdata0 ;
6. grâce à cette information, evalscript peut mettre à jour ses données de dépendance et mettre en place le mixin sur l'objet seqdata0.

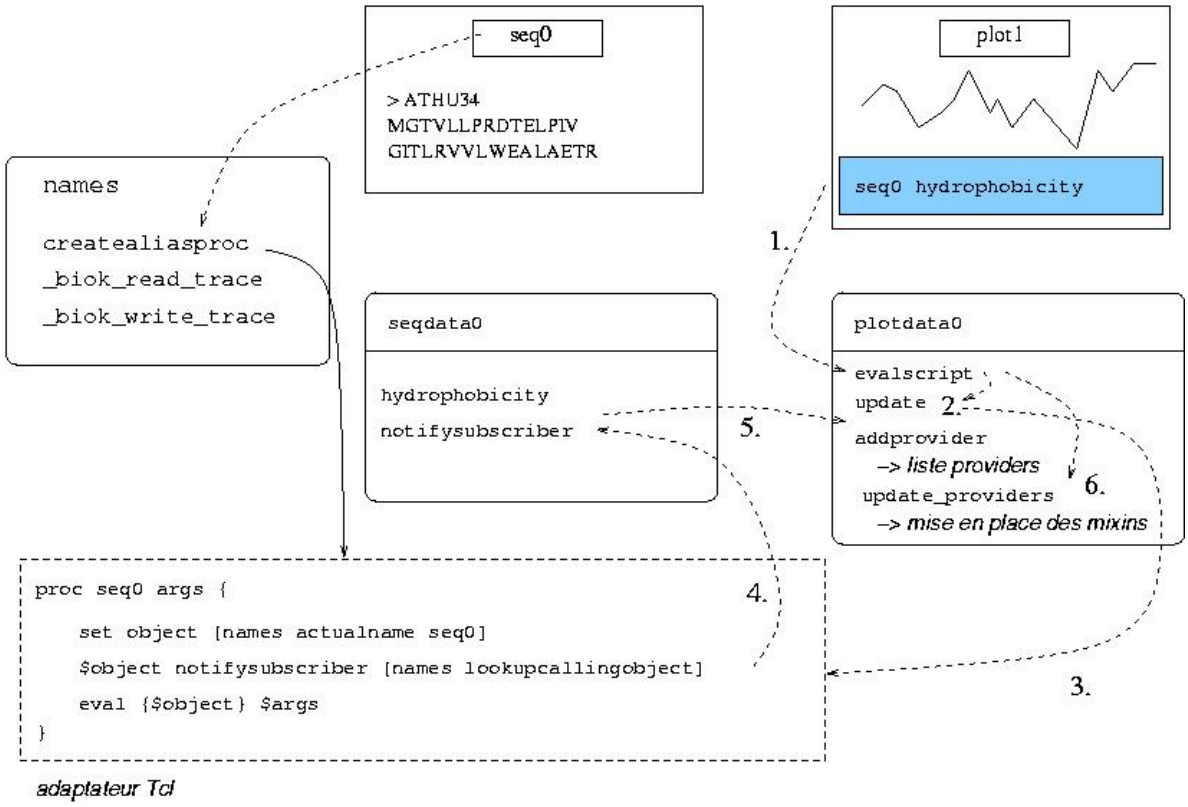


FIG. 5.38 – Détection et gestion des dépendances.

5.4.3.2 Interface utilisateur

C'est surtout la génération des menus qui repose sur l'introspection :

- pour composer la liste des objets graphiques dans le sous-menu de création d'objets,
- pour déterminer les menus d'accès au code en distinguant vue et données, et en donnant accès aux classes imbriquées,
- pour les menus hiérarchiques d'accès aux classes et à leurs instances,
- pour le sous-menu d'accès aux objets (debug, names, codegen,...),
- etc...

5.4.3.3 Programmation

L'environnement de programmation repose sur plusieurs outils d'introspection :

1. éditeurs :

– l'éditeur de méthodes obtient le code par l'intermédiaire de fonctions introspectives (classe Script), la modification étant réalisée par la commande eval de Tcl ;

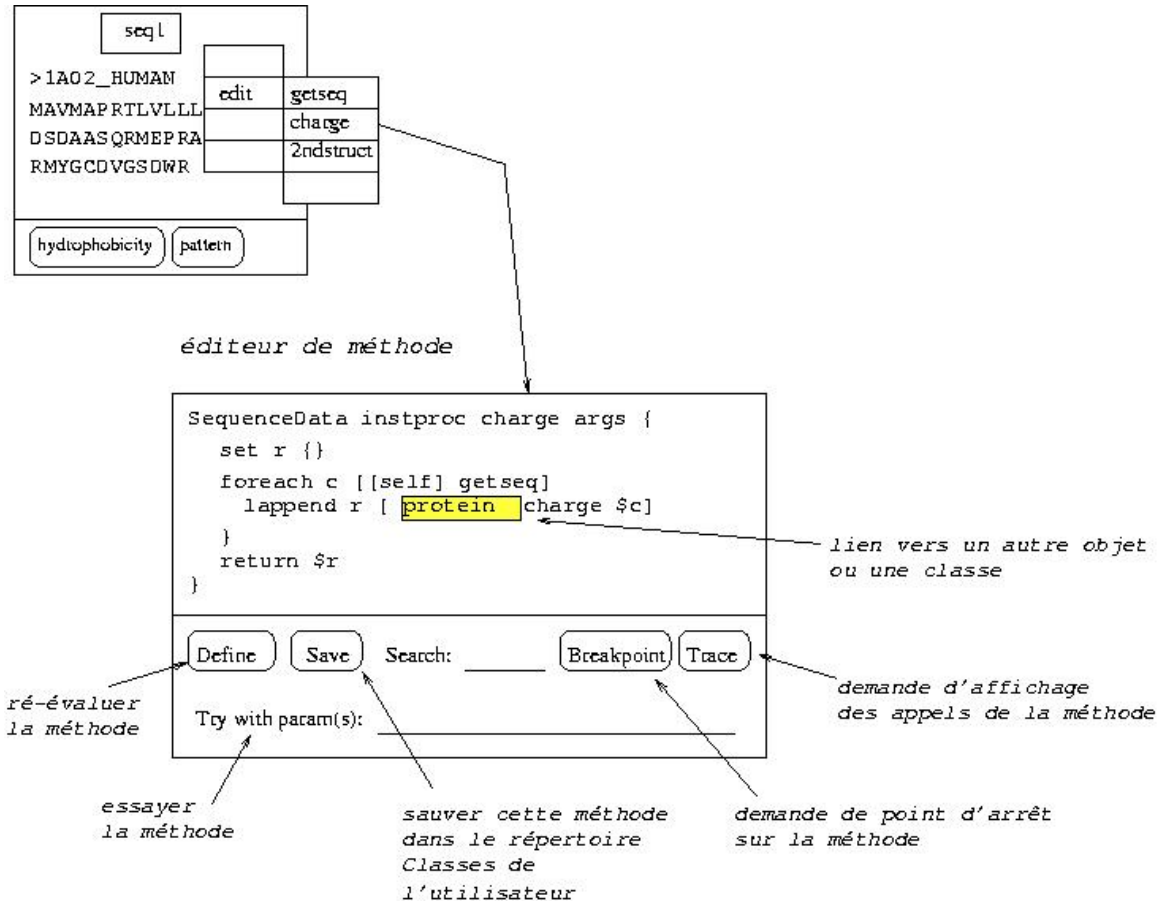


FIG. 5.39 – Editeur de méthodes.

– l'éditeur d'attributs graphiques utilise les fonctions introspectives des widgets Tk (cget), et leur commande configure pour les modifier ;
– la recherche par mots dans les noms de méthodes ou dans le corps des méthodes utilise également ces fonctions introspectives ; le mot à rechercher peut être la sélection courante, qu'elle vienne d'un éditeur de code ou de tout autre zone de texte de l'interface.

2. la correspondance entre fichiers sources et édition dans l'environnement biok est gérée de manière introspective :

   (a) le code d'une méthode est sauvé dans un fichier dont le nom est structuré ainsi : BIOKUSERCLASS/classname/methodname.tcl
   Il faut donc détecter le nom de la méthode et de la classe dans l'éditeur, que l'utilisateur est libre de changer ;

   (b) un filtre est optionnellement activé pour détecter le fichier depuis lequel est chargé une classe, avec la commande info script de Tcl ;

   (c) la persistance est implémentée par introspection sur les divers éléments de la structure d'un objet.

3. Les fonctions de déboguage et de trace reposent entièrement sur les filtres XOtcl :
   – le filtre debugFilter implémente le point d'arrêt,
   – le filtre spyFilter implémente l'espionnage des appels,
   – les filtres ${class}Filter implémentent la fonction de trace pour une classe particulière.

On note un problème cependant avec Tcl ou XOtcl : le code, une fois obtenu, n'est pas pour autant suffisamment structuré pour une manipulation facile, contrairement aux langages Lisp où les lambda sont des listes (ce problème est singalé par [Foo90]).

5.4.3.4 Intégrer le niveau interface utilisateur dans la définition des objets et classes

Nous avons évoqué cette possibilité dans le chapitre III (3.3.3.2), qui consiste à encoder des informations sur l'interface utilisateur dans le langage lui-même.

En XOtcl, nous pouvons par exemple imaginer une nouvelle méthode instcommand pour les classes :

```
classe instcommand procname args {
  ...
}
```

dont le rôle est le même que celui de la méthode standard instproc de définition de méthode, mais qui en plus ferait en sorte que :

1. ces méthodes apparaissent automatiquement dans le menu de l'objet,
2. les méthodes de ce type sont les réponses des requêtes de type "Quelles sont toutes les commandes ?" :

   ```
   info commands
   ```

   au nivau du shell associé à chaque objet (comme le shell est un moyen direct d'invocation des méthodes de l'objet, la question de savoir quelles sont les commandes disponibles revient systématiquement de la part des utilisateurs du prototype biok au cours des ateliers),
3. elles sont également le niveau de granularité correspondant à l'enregistrement de macros.

On peut aussi imaginer une façon de spécifier la catégorie de la commande pour savoir dans quel sous-menu la placer ou pour associer ce mot-clé à la requête : info commands category :

```
classe instcommand procname category args {
  ...
}
```

ou associer des mot-clés, le premier pouvant servir de categorie pour être classé dans un menu, les autres pouvant servir pour des recherches de rubriques :

```
classe instcommand procname keywords args {
  ...
}
```

Les implémentations possibles consistent soit à définir une méta-classe, par exemple : GraphClass et à définir toutes les classes d'objets graphiques avec cette méta-classe au lieu de Class, ou plus simplement - le langage l'autorise techniquement et conceptuellement - à ajouter la méthode instcommand directement à l'objet Class :

```
Class proc instcommand procname keywords args {
  ...
}
```

La solution définissant une méta-classe est plus lourde, mais pourrait par ailleurs servir à automatiser les mécanismes d'initialisation graphique des objets (draw) pour la partie générique - actuellement les objets doivent explicitement appeller draw depuis init, et ce dans un certain ordre (après avoir appelé next de init).

On pourrait également imaginer un système d'annotations qui structurerait ces informations, qui sont en réalité orthogonales à la notion de commande. Ce mécanisme générique pourrait s'intégrer assez aisément dans un environnement dont les utilisateurs sont particulièrement familiers avec l'activité

d'annotation. Puisque nous tentons d'intégrer utilisation et programmation dans un même environnement, un même mécanisme pourrait être utilisé pour les informations de nature programmatique (dans lesquelles des connaissances scientifiques sont incorporées) et de nature biologique.

5.4.3.5 Autres utilisations

Les fonctions d'introspection sont également utilisées dans les cas suivants :

- redirection d'appels de méthode : depuis le shell de l'objet, on peut appeller soit les méthodes de la vue, soit les méthodes de la partie données si la méthode n'est pas définie pour la vue : c'est la méthode unknown de la classe GraphObjects qui effectue la redirection ;
- un mécanisme similaire rend possible l'appel des méthodes de l'objet édité depuis le shell de l'éditeur de méthodes de cet objet (nous avions en effet constaté qu'un des utilisateurs de biok essayait de le faire) ;
- l'introspection est utilisée pour retrouver la structure des objets composite (fonctions info, children, parent, ...) reposant sur le mécanisme d'aggrégats d'XOcl, utilisé notamment par la méta-classe Composite (gestion des widgets, des menus, des objets incorporés, ...).

Il sera sans doute utile de vérifier que les redirections du premier type, mais surtout du second type ne sont pas trop sources de confusion pour les utilisateurs.

5.4.4 Flexibilité

Nous avons décrit, dans le chapitre III, différents types de flexiblité : la flexibilité interne, qui concerne la souplesse intrinsèque de l'architecture, et qui rend le code plus facile à faire évoluer et à comprendre, ainsi que la flexibilité externe, qui concerne la facilité d'utilisation d'un élément logiciel dans un environnement externe.

Nous avons notamment illustré la notion de flexibilité interne par les patrons de conception [GHJV95], dont plusieurs sont utilisés dans biok :

1. fabrique abstraite : editors est un objet serveur des outils d'édition, qui lance et gère les éditeurs de code ;
2. par contre, l'objet builder est une fabrique concrète : il prend en effet en paramètre le nom de la classe de l'objet graphique à créer :

   ```
   builder new Plot -name plot_hydro
   ```

   mais c'est une fabrique quand même puisque tous les objets graphiques sont créés de la même manière.
3. prototype : il est possible de cloner un objet ou de le copier-coller dans un autre (commande copy de XOtcl) ;
4. singleton : XOtcl fournit le mécanisme, puisqu'il n'est pas nécessaire d'associer une classe à un objet ; il suffit de le déclarer comme objet :

   ```
   Object debug
   ```

   Plusieurs singletons existent dans biok :
   - names : le serveur de noms,
   - builder, editors, widgets : les fabriques abstraites et concrètes,
   - debug : le débogueur,
   - sources : le gestionnaire des fichiers sources,
   - biok : l'objet "menu principal",
   - codegen : le code génétique,
   - dna et protein : les serveurs d'informations concernant l'ADN ou les protéines,

– etc...

5. composite : les commandes pour les menus, ainsi que les widgets sont des composites, et leur classe est une méta-classe (Composite) dont la fonction principale est de définir un filtre qui permet d'appeller récursivement certaines méthodes, explicitement enregistrées, sur toutes les composantes de l'objet ; l'implémentation de patrons de conception est une des applications des filtres et des mixins selon les auteurs du langage XOtcl [NZ99c][NZ99a] ; nous avons effectivement implémenté le patron composite en suivant [NZ99b] avec un filtre associé à une méta-classe Composite :

```
Class Composite -superclass Class

Composite instproc compositeFilter args {
    set m [self calledproc]
    set c [self regclass]
    set r [next]
    if {[info exists ${c}::ops($m)]} {
[self] instvar compcommand
foreach child [[self] info children] {
    eval $child $m $args
}
    }
    return $r
}
```

Il suffit ensuite de définir la super-classe AbstractNode :

```
Composite AbstractNode
```

et les sous-classes spécifiques d'objets composites :

```
Class Widgets -superclass AbstractNode
Class Widget  -superclass Widgets
```

Ces classes doivent implémenter la méthode iterate, qui prend comme premier argument un visiteur :

```
Widgets instproc iterate args {
    set v [lindex $args 0]
    set args [lrange $args 1 end]
    eval {$v visit [self]} $args
}
```

Un visiteur peut alors être créé pour les différentes itérations possibles et ce visiteur doit définir une méthode visit :

```
Class TreeVisitor

TreeVisitor deleteWidget
deleteWidget proc visit args {
    set node [lindex $args 0]
    names deleteWidget [$node set path]
```

```
}
```

6. itérateur : les classes de tags, présentées dans la partie concernant l'éditeur d'alignement, définissent différents types d'itérateurs : par colonne, par ligne, par cellule, ...
7. observateur : l'architecture autour du mixin pour gérer les dépendances est un observateur [NZ99c] ;
8. adaptateur : plusieurs mécanismes déjà décrits, dont le générateur d'interfaces Pise et les adaptateurs Tcl ;
9. pont : la fonction de redirection des analyses depuis l'éditeur d'alignement vers un objet séquence ;
10. poids mouche : les cellules dans l'éditeur d'alignement sont de la classe Cell et CellData, mais aucune instance n'est effectivement créée tant qu'il n'y a pas de dépendances sur cet objet ;
11. décorateur : la zone d'entrée de formule peut avoir un décorateur qui affiche un bouton pour ouvrir un sélecteur de fichiers dans le cas d'objets ayant un paramètre file ; ce décorateur peut être un mixin (non implémenté).

Sur le plan de la flexibilité externe, plusieurs mécanismes sont disponibles :

- l'appel de programmes externes par l'intermédiaire de la classe Pise qui permet l'utilisation soit interactive, soit programmatique de près de 200 programmes d'analyse ;
- biok ne comporte pas pour l'instant d'interface programmatique qui permettrait de l'utiliser de manière non graphique ; il est vrai que son architecture privilégie l'aspect interactif de la programmation, et c'est moins pour des raisons techniques (il suffirait de changer la procédure de lancement pour éviter l'affichage), que pour des raisons fonctionnelles que nous n'avons pas développé cet aspect.

5.4.5 Programmation par l'utilisateur

5.4.5.1 Deux niveaux de programmation

Comme cela est illustré dans la figure 40, biok repose sur deux couches principales :

1. une couche d'objets graphiques qui sont les objets d'intérêt de l'utilisateur (graphe, séquence d'ADN, entrée d'un banque de données, carte génétique, ...) ;
2. une couche d'objets XOtcl.

A cette distinction correspond la superposition de deux modèles de calcul : dataflow (c'est-à-dire fonctionnel auto-déclenchable) et objet, qui se situent à des niveaux d'utilisation différents. La programmation par objets est accessible par l'édition du code des méthodes, et pas autrement, alors que la programmation dataflow se fait par les formules, c'est-à-dire soit dans les zones d'entrée de formules prévues par l'interface, soit dans des scripts, soit par l'intermédiaire de commandes de connexion proposées dans les menus et qui proviennent d'un historique des formules. Pour éviter à l'utilisateur la saisie de code en Tcl lors d'une initiation à l'utilisation, on peut imaginer que des fichiers d'historique pré-configurés, comportant par exemple les formules les plus courantes par ordre d'importance décroissant soient fournis à l'utilisateur peu expérimenté.

Le prototype comporte également un double niveau de nommage : les alias des objets graphiques et les identifiants des objets XOtcl avec une correspondance assurée par le système. Le nommage des objets graphiques est contrôlable par l'utilisateur ; c'est un espace plat de variables globales, s'opposant au nommage structuré et hiérarchique du niveau programmation par objets. Des fonctions de l'interface permettent à l'utilisateur de connaître le nom interne des objets graphiques, ainsi que ceux d'ailleurs des widgets Tk pointés par la souris.

5.4.5.2 Navigation

L'environnement de programmation a pour objectif de retrouver plus facilement le code associé à une fonction ou à un composant graphique de l'interface. Pour cela, nous avons mis en place les outils suivants :

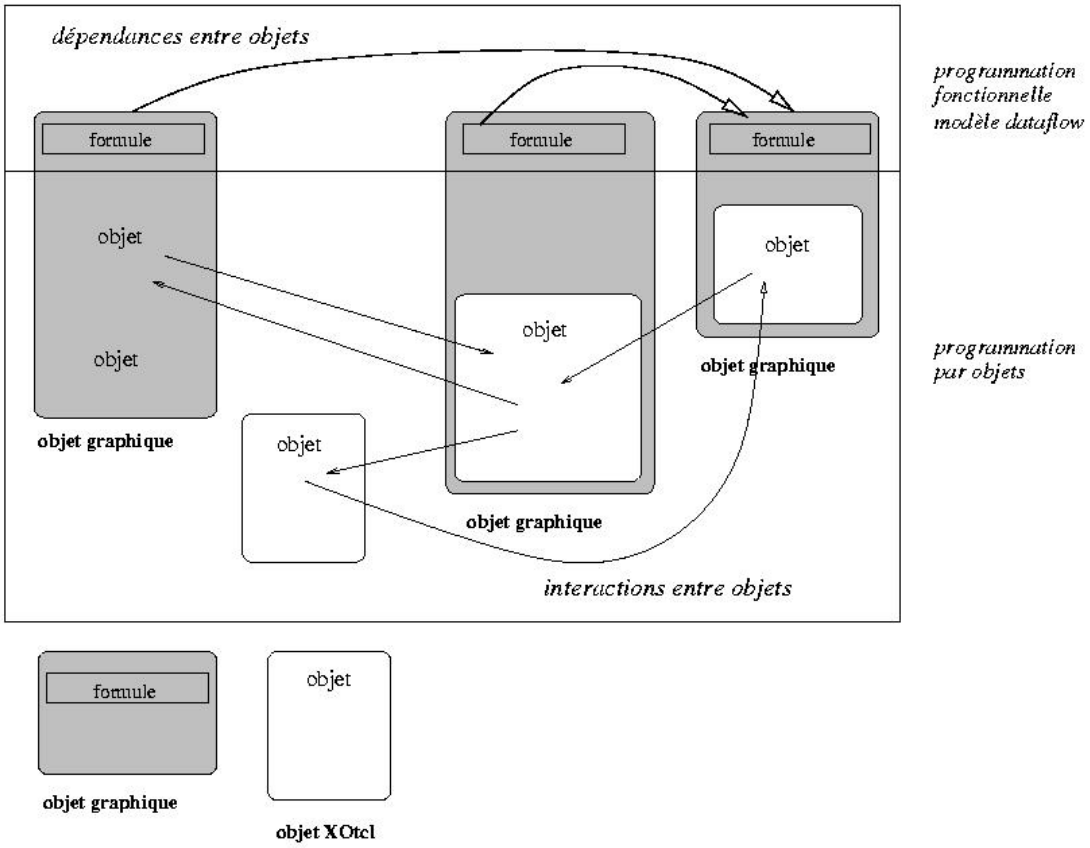


FIG. 5.40 – Couches dataflow et objet.

- l'accès au code par des menus structurés : menus des classes, des objets, et pour chaque objet affiché : menus des méthodes de la vue, menu des méthodes de la partie données, menu des méthodes des classes composantes ;
- des outils de recherche :
  - recherche par nom de classe avec ou sans troncature,
  - recherche par nom de méthode (idem),
  - recherche dans le corps des méthodes.

  ces outils sont notamment utiles pour accéder au code des objets non graphiques ;
- des outils de déboguage (points d'arrêt et trace des appels, c'est-à-dire affichage de la pile des messages dans une fenêtre) ;
- une fonction d'espionnage permet de savoir quelles sont toutes les méthodes qui ont été appellées durant un laps de temps dont l'utilisateur définit le début et la fin ; lorsqu'il en demande la liste, il lui est alors possible de demander à tracer ces appels dans la fenêtre d'affichage de pile des messages.

Il doit également être possible de modifier directement le comportement de l'application. Outre l'édition de toutes les composantes de l'application, il est également important de pouvoir essayer les modifications apportées. Pour cela, il y a plusieurs possibilités d'appeller une méthode :

- sur l'objet édité depuis l'éditeur, ce qui fait de l'éditeur un formulaire pour la méthode (comme dans LabView),
- depuis le shell,
- depuis l'interpréteur Tcl.

5.4.5.3 Déboguage

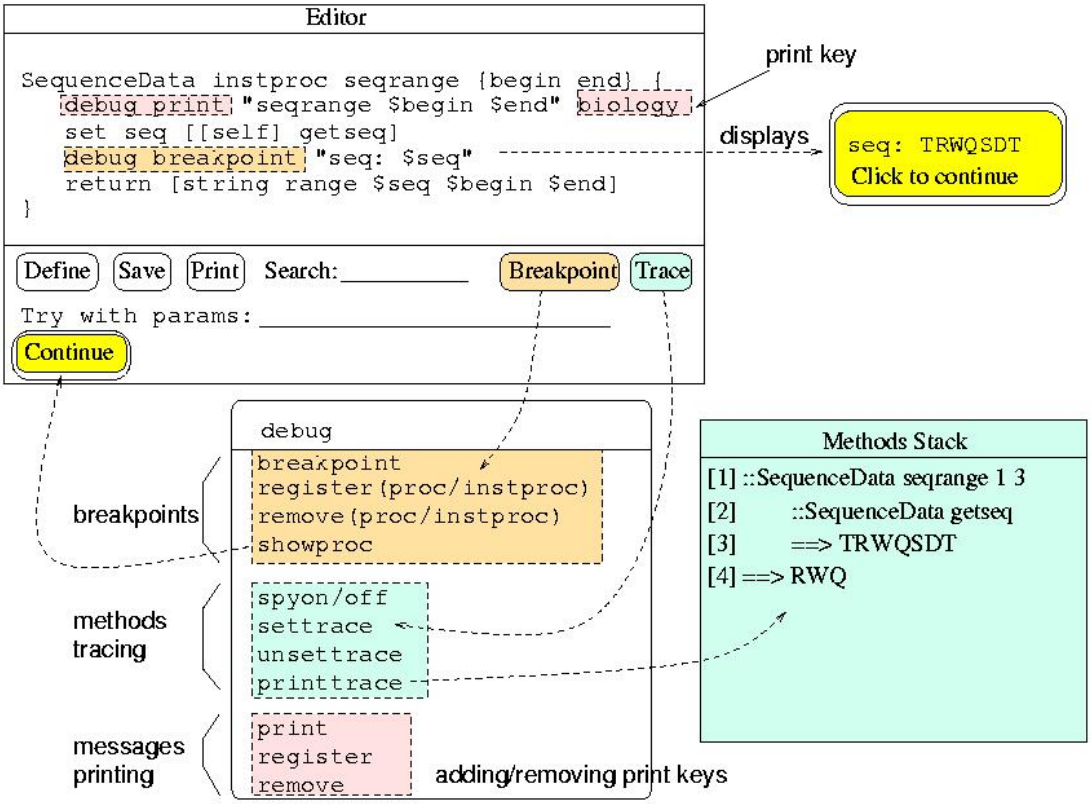


FIG. 5.41 – Déboguage.

Les fonctions de déboguage (figure 41), comme la demande de points d'arrêt ou de trace sur une méthode invocables depuis l'éditeur (figure 39). La trace de l'appel des méthodes permet de visualiser les interactions entre objets, avec les paramètres d'appel et la valeur de retour.

5.4.5.4 Environnement général

Lorsque l'utilisateur sauvegarde le code d'une méthode, ce n'est pas le système lui-même qui est modifié, ni sa copie locale, mais la vue personnelle de l'utilisateur sur le système. Cette version se matérialise par une arborescence de fichiers, par exemple : $HOME/bio/Classes, comportant un répertoire par classe XOtcl, dans lequel chaque méthode figure dans un fichier différent, comme cela est illustré par la figure 42. A l'initialisation du système, ce sont d'abord les classes "système" qui sont chargées, puis les classes utilisateurs, ce qui a pour effet l'ajout ainsi qu'une éventuelle redéfinition de certaines méthodes. Il n'y a pas pour l'instant de message d'avertissement dans le cas d'une redéfinition, mais cela peut être aisément implémenté grâce aux mécanismes de filtre. Il aurait été possible, comme dans d'autres systèmes ou comme dans la philosophie des frameworks, de forcer l'utilisateur à créer une sous-classe pour les cas de redéfinition. Cependant, comme nous l'avons expliqué dans le chapitre III, il n'est pas toujours logique, sur le plan sémantique, de procéder de la sorte. Si l'utilisateur juge qu'une méthode d'analyse associée à la classe SequenceData n'est pas appropriée à son cas spécifique, faut-il nécessairement qu'il créée une sous-classe MySequenceData, MySequenceData ou NomExperienceSequenceData, etc... ? Une des orientations à explorer à ce sujet est peut-être celle de la programmation par aspects, dont l'idée est d'isoler des éléments variant indépendamment, d'ailleurs référencée par [vR99] ou [Mør98] dans le cadre d'une approche de personnalisation.

Une autre question que nous avons eu l'occasion d'explorer lors du premier stage de bio-informatique, est celle de la programmation coopérative. Comme cela est expliqué dans la section où ce stage est décrit, il faut à la fois que l'utilisateur-programmeur sente qu'il peut coopérer au développement du système par une contribution de code, et qu'il peut aussi faire ce qu'il veut, faire des erreurs, explorer des solutions qu'il n'a pas nécessairement l'intention de divulguer, etc... Les outils comme CVS sont un peu contraignants pour ce double objectif, et nous avons préférré pour l'instant rester à une gestion "manuelle" de ces opérations. S'il y a un "protocole", c'est donc ici un protocole plus organisationnel de partage de code entre utilisateurs et concepteurs. Ce protocole est pour l'instant informel, car nous ne sommes pas sûrs de l'intérêt d'une instrumentalisation. Mais on pourrait très bien imaginer une méthode spéciale publish applicable au niveau de chaque méthode, avec des mécanismes de versionnement, de gestion des conflits ou de vues.

Nous avons cité le problème de la robustesse des systèmes comportant un protocole d'auto-modification (MOP, chapitre III, 3.2.3.2) à propos de la nécessité d'une barrière d'abstraction. Sur le plan technique, si l'utilisateur veut revenir à la version officielle d'une méthode, il lui suffit de supprimer le fichier correspondant, procédé dont l'effet sera visible lors de la session suivante (il n'y a pas de moyen de dé-définir une méthode à notre connaissance dans XOtcl).

Mais la question de la diversité des versions induites par ce type d'organisation ne laisse pas de déranger certains informaticiens, qui y voient un potentiel désordre, et ce, parce qu'ils veulent voir un système soit comme un outil de programmation, soit comme une application, *mais pas les deux en même temps* .

5.4.5.5 Persistance

Le choix que nous avons fait est explicité dans le chapitre III, et, comme une solution générique semblait complexe à concevoir, la fonction de persistance a été explorée pour certaines classes d'objets seulement. Ces classes définissent optionnellement une méthode "save" pour les parties vue et donnée, dans lesquelles l'utilisateur peut spécifier des méthodes ad-hoc à appeler. La classe générique GraphObject effectue les sauvegardes générales et appelle les méthodes ad-hoc.

Son effet est très apprécié par les premiers utilisateurs du système, mais on remarque que, lorsque cette fonction leur est montrée, la première question qu'ils posent est : comment peut-on supprimer l'objet persistent Ceci nous donne à penser qu'une persistance automatique par défaut n'est peut-être pas le choix le plus approprié.

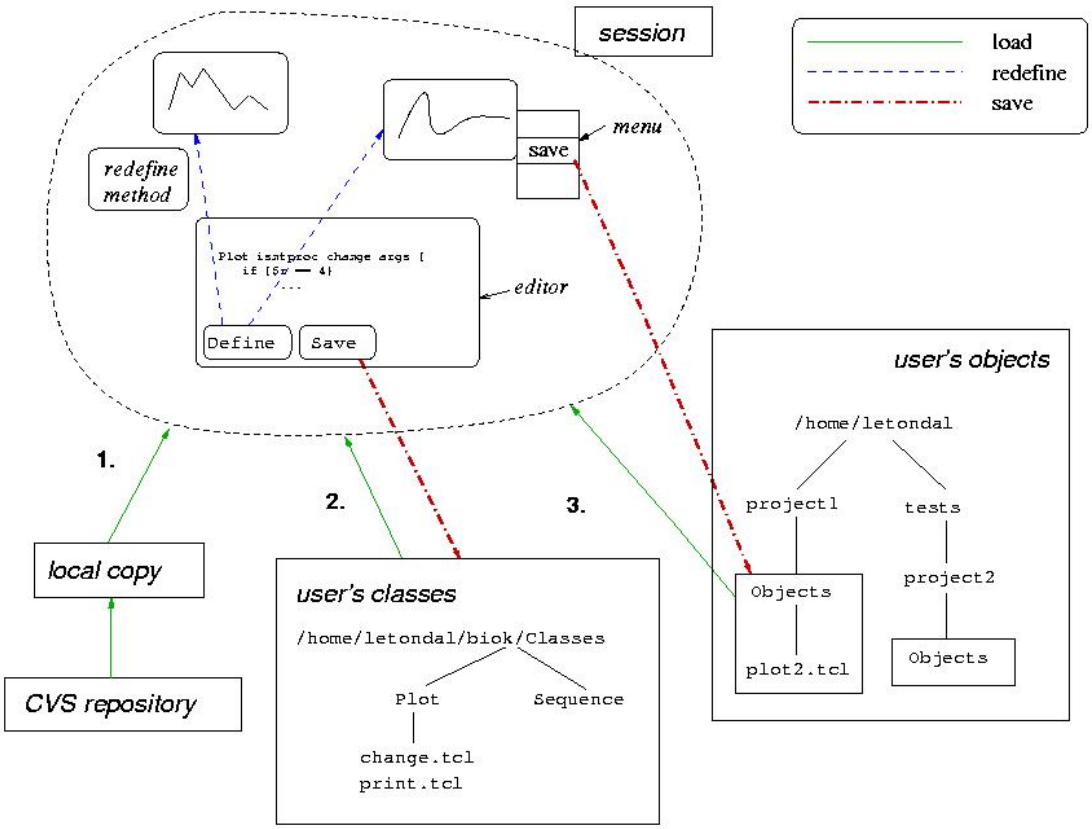


FIG. 5.42 – Environnement.

Les objets persistents sont sauvegardés dans le répertoire de travail (figure 42), concrétisant par les moyens classiques du système Unix la notion de projet, contrairement aux classes utilisateurs qui sont dans un seul répertoire visible de tous les projets.

Sont également appréciées les fonctions d'historique, qui sont également une forme de mémoire, et surtout un bon moyen de faire gagner du temps à l'utilisateur, ainsi que la possibilité pour ce dernier de créer sa propre documentation.

L'idée d'historique thématique complètement éditable a été suggérée à plusieurs reprises, soit durant les entretiens, soit durant les ateliers (chapitre II, 4ème atelier).

La documentation est disponible en trois niveaux :

1. description simple,
2. description complète,
3. documentation personnelle.

La documentation personnelle est conservée au niveau de la classe, tout comme le sont les historiques.

En réalité, il y a un recoupement entre documentation et historique : quand il n'y a pas d'historique, ainsi que nous l'avons observé avec l'étudiante du premier stage, c'est le "bloc-notes" que constitue la documentation qui sert à la fois de documentation, mémo, presse-papier, etc... Un outil très utile concrétisant cet aspect serait un espace de travail, au sens du workspace de Smalltalk, de nature persistente, avec le fonctionnement par défaut suivant :

- ce workspace pourrait être activable sur demande,
- une fois demandé dans le contexte d'un projet (c'est-à-dire d'un répertoire), rendu persistent,
- à la première ouverture du workspace, ce dernier pourrait être initialisé par la valeur du dernier workspace utilisé.

Ce fonctionnement pourrait faire l'objet à la fois d'une séance de prototypage et d'un découpage modulaire pour le rendre modifiable.

5.4.6 Stages d'étudiants

5.4.6.1 Suivi d'une biologiste : programmation d'un algorithme biologique intégré dans l'éditeur d'alignement

Une étudiante en biologie a suivi un stage de deux mois et demi dans le service avec pour sujet de stage d'implémenter un algorithme de prédiction de segments transembranaires dans une séquence protéique [Hei92] dans l'environnement biok.

L'article donne le principe de l'algorithme, qui consiste à calculer un profil pondéré d'hydrophobicité de la séquence. Les pics de ce profil sont de bons indicateurs de segments internes à la membrane, particulièrement hydrophobes. Le calcul du profil s'effectue avec une fenêtre glissante de taille fixe, avec une particularité pour pondérer différemment les bords du segment : la fenêtre est trapézoïdale (figure 43).

L'algorithme consiste à calculer une somme pondérée $h_i * w_i$ sur cette fenêtre, avec :

h_i = une valeur d'hydrophobicité différente pour chaque acide aminé composant la protéine, et
$w_i = i/S$ pour $1 <= i <= n - q + 1$; $(n - q + 1)/S$ pour $(n - q + 1) < i < (n + q + 1)$; $(2n + 2 - i)/S$ pour $(n + q + 1) <= i <= 2n + 1)$
avec le facteur de normalisation $S = (1 + n)2 - q2$ pour obtenir :

$$\sum_{i=1}^{2n+1} w_i = 1$$

Les grandes lignes de l'algorithme défini par ces formules ont été implémentées en 6 semaines environ, mais le démarrage a été difficile. Ce sujet a été donné aux 15 étudiants du cours de programmation

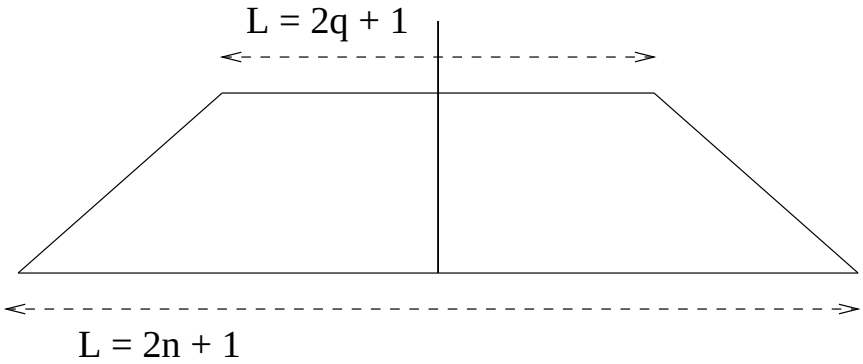


FIG. 5.43 – Fenêtre glissante pour le calcul du profil d'hydrophobicité

de l'Institut Pasteur de la même année, qui, après deux mois d'une formation solide, suffisamment encadrée, l'ont implémenté en trois semaines. Notre stagiaire, en revanche, avait suivi à l'université un cours de programmation de 20 heures (cours magistral devant 40 étudiants), assorti de séances de 20 heures de travaux pratiques en binômes, avec un seul enseignant pour 15 élèves. Le cours suivi était par ailleurs en Python, alors que biok utilise Tcl. Le cours, ciblé pour des non-informaticiens, comprenait une introduction aux structures de données et à la récursivité, appliquée aux arbres :

- Objets de base : nombres, chaines, listes ; désignation
- Fonctions et méthodes
- Conception de fonctions récursives
- Itération.
- Construction de nouveaux objets : types abstraits et classes
- Liaison dynamique et surcharge d'opérateurs
- Illustration : les arbres

Mais l'environnement informatique - ce qu'est un interpréteur de commandes, un gestionnaire de fenêtre, un système de fichiers, un éditeur de texte - rien de tout cela n'était connu ni au niveau conceptuel, ni au niveau pratique. Enfin l'environnement biok n'était pas finalisé du tout, il n'y avait pas encore de documentation, de nombreuses incohérences subsistaient, et des erreurs se produisaient très fréquemment, ... et le maître de stage était très occupé !

Description des principales difficultés

Nous avons pris des notes pendant les 6 premières semaines du stage afin d'identifier les principaux problèmes et de mesurer la progression de la stagiaire (la première semaine - j'étais absente - a été consacrée à la bibliographie). La description qui suit est une synthèse de ces notes, et ne prétend absolument à aucun caractère objectif, scientifique, ou généralisable. C'est une expérience, qui a eu une certaine importance dans la vie du prototype biok car de nombreuses erreurs ont pu être corrigées, et des fonctions ont été soit retirées parce que jugées trop complexes, soit ajoutées après avoir observé l'étudiante en train de travailler.

L'environnement

Au début, sans doute à cause d'une absence de formation sur l'architecture des machines [vR99], mais aussi à cause de la non-finition du prototype, la stagiaire confondait les accès aux fichiers avec un chemin d'accès (path) et l'accès à un objet de programmation. Voici par exemple un cas de schéma de nommage spontané (il s'agissait d'accéder à la valeur d'un tableau) :

```
$biok/biokwish/sourcebiok/protein/hydrophobicity(A)
```

Or ces objets se situent à des niveaux et dans des contextes extrêmement différents :

- biok est à la fois une variable d'environnement Unix et un fichier tcl à évaluer (source) ;
- biokwish est le nom de l'interpréteur wish recompilé avec toutes les bibliothèques nécessaires au fonctionnement de biok ;

- source biok est une instruction Tcl ;
- protein est un objet Xotcl ;
- hydrophobicity est une variable désignant un tableau assiociatif.

Il reste que l'instruction attendue était certainement trop difficile à écrire à cette étape, même après lecture des premiers chapitres d'un livre sur Tcl [Wel99] :

```
protein set hydrophobicity(A)
```

Cependant, une fois le principe expliqué, l'étudiante a spontanément utilisé un autre type d'information de l'objet protein un ou deux jours plus tard.

Les boucles

Le deuxième problème qui a persisté un certain temps concernait l'écriture de boucles, qui n'avait sans doute pas été vues ou très rapidement dans le cours universitaire, plutôt axé sur la récursivité. L'étudiante ne comprenait pas le principe syntaxique et confondait les boucles et les fonctions : selon elle, une instruction écrite en dehors de la boucle pouvait en faire partie. Le principe de construction d'une itération à partir d'un problème et la notion de pas d'itération n'étaient pas non plus acquises.

Force est de constater que ce sont là en effet les difficultés reconnues de la programmation, et que l'acquisition de ces techniques peut être plus difficiles pour certains - qu'ils soient étudiants en informatique ou biologistes. Ces difficultés venaient probablement d'une part d'une non connaissance des mots anglais (while, for, ...) et d'autre part d'une confusion entre les types d'objets d'un programme (variable, mot réservé, ...) : tout cela n'aidait pas à trier les difficultés.

L'affichage de la courbe : une étape dans la progression

L'objet d'affichage de courbes a été vu au début de la 3ème semaine et la possibilité d'afficher graphiquement les valeurs, très appréciée, a sensiblement accéléré la progression : la première fonction de calcul du profil d'hydrophobicité était finie à la fin de la semaine.

Les fonctions

Il restait néanmoins à ce stade des difficultés concernant la définition et l'utilisation de fonctions, notamment pour le fonctionnement du passage de paramètres, et pour la distinction entre une valeur de retour et une instruction d'affichage, qui font le même effet sous l'interpréteur Tcl.

Le passage de paramètres

C'est une notion qui a mis du temps à être assimilée : la stagiaire pensait qu'il fallait donner le même nom de variable aux paramètres formels et effectifs.

Modification de l'objet graphique

Nous avons fait ensemble une modification de la classe Plot (méthodes draw et redisplay) pour ajouter un champs d'affichage donnant la valeur en x d'un point de la courbe ; l'étudiante a plutôt bien suivi ce qui se passait (malgré sa remarque contraire) puisqu'elle m'a rappellé un des points que j'oubliais en résumant tout ce que j'avais fait (c'était des bind et autres instructions similaires).

Modélisation

Un petit travail de conception a été fait ensemble vers la 6ème semaine pour réfléchir sur les différentes manières d'utiliser et d'afficher les résultats, notamment les différentes topologies induites par la détermination des segments transmembranaires. Il s'agissait donc de définir des classes et la manière d'interagir entre elles. Au départ, l'étudiante ne connaissait pas les notions de base de classe, d'instance et d'héritage : après un petit cours, nous avons pu établir les grandes lignes de la conception. Elle a également lu deux ou trois chapitres d'un ouvrage d'introduction à ces notions [BL92], et la discussion que nous avons eu à ce sujet, les questions qu'elle a posées semblait montrer une compréhension correcte.

Conclusion

Utilisabilité

L'environnement biok a été nettement amélioré durant ce stage, outre la correction de nombreuses bogues :

- l'historique du shell a été implémenté,
- des fonctions simples, comme la fonction "Print" dans l'éditeur de méthodes ont été ajoutées (l'étudiante utilisait un éditeur externe uniquement dans le but de pouvoir imprimer),
- l'historique des formules a été introduit : cette fonction permet de créer un nouvel objet à partir d'un menu de formules associé aux objets qui sont potentiellement utilisés dans ces formules (c'est une sorte de guidage pour le dataflow équivalent à celui qui est proposé dans le logiciel générateur d'interfaces Web).

Programmation coopérative

Cette expérience, bien qu'il s'agisse d'un projet très encadré qui n'est pas complètement représentatif des situations futures d'utilisation du prototype, est également un point de départ pour la réflexion sur les questions de développement coopératif. Le code de l'étudiante a été intégré au système, sans que des mécanismes automatiques aient été mis en place pour autant : nous ne sommes pas sûrs que les outils existants, comme les outils de développement collectif de type CVS, soient adéquats pour cette situation, comme nous l'avons dit plus haut.

Apprentissage

Un certain nombre d'opinions, qui doivent être vérifiées de manière plus rigoureuse, sont issues de cette expérience.

Il va de soi à notre avis qu'une bonne formation pratique, mais aussi conceptuelle est nécessaire même pour écrire ce type de petits programmes (200 lignes de code), et l'informatique est déjà enseignée aux futurs biologistes à l'université. Nous sommes bien conscients qu'il est très difficile d'enseigner rapidement les concepts de base de l'informatique, avec peu de ressources. Pourtant, il faudrait simplement qu'il puissent pratiquer très rapidement et avec un encadrement suffisant les concepts au fur et à mesure qu'il les apprennent. Et si la formation est courte, il nous semble par exemple que l'apprentissage sur des objets biologiques intéressants pour l'étudiant accélèrerait l'acquisition (l'effet "affichage de courbes" était impressionnant). La progression de l'étudiante, qui commencé le stage avec un bagage informatique assez faible, a été en effet très rapide. Cela est certainement dû à la possibilité de demander une aide personnelle en cas de difficultés, ce qui n'était pas possible à l'université apparement. Mais nous pensons que l'environnement a également pu motiver l'étudiante pour apprendre : c'est une opinion subjective, mais dont elle nous a cependant fait part elle-même. Enfin, s'il est vrai que la programmation est difficile à apprendre, [Eis97] émet cependant l'hypothèse que cette affirmation, commune à de nombreux chercheurs, est peut-être trop pessimiste : la programmation est en effet une technique acquise chaque année par des milliers de personnes. L'Institut Pasteur dispense depuis 8 ans un cours de programmation de 3 mois suivi chaque année par une quinzaine de biologistes, et ce cours n'a rien de démagogique : on y enseigne la programmation en Scheme et en Java, ainsi que les concepts fondamentaux de la programmation. Les élèves de ce cours, pour la plupart, assimilent bien les concepts et semblent apprécier les exercices de programmation, ce qui nous conduit à envisager l'hypothèse que l'algorithmique et les concepts de programmation ne sont pas nécessairement les aspects les plus difficiles à maîtriser pour ces étudiants scientifiques.

Un cas représentatif

Enfin, le sujet du stage est assez représentatif des programmes qu'un chercheur en biologie peut être amené à écrire pour sa recherche : il consiste dans une activité de *traduction* , au sens propre des activités génériques de [Gre89], puisque la spécification de l'algorithme était donnée dans l'article, sous forme d'équations de récurrence. Cet objectif de programmation d'algorithmes donnés dans la littérature est un des arguments donné par [G97] pour l'apprentissage des tableurs par les biologistes. Si l'un des objectifs de la programmation par l'utilisateur est bien la proximité du langage de spécification du programme par rapport au domaine, nous sommes ici exactement dans ce cas, puisque le domaine de la bio-informatique est un domaine de méthodes scientifiques pour la prédiction, même si cela est plus récent que pour les mathématiques ou les statistiques. C'est pour ce type de raisons d'ailleurs que le stage a profité à l'étudiante : elle était plutôt enthousiasmée, après quelques jours d'effort, à l'idée de pouvoir expérimenter elle-même des variations de l'algorithme et des calculs similaires (elle a d'elle-même programmé d'autres petits calculs de courbes pour analyser ses résultats).

5.4.6.2 Flexibilité algorithmique

Le deuxième stage a pris la suite du premier, puisqu'il s'agissait d'explorer les possibilités d'injecter des informations venant de l'utilisateur dans les étapes heuristiques de l'algorithme, mais sans modifier ce dernier (`http://www-alt.pasteur.fr/~letondal/biok/usermixin.html`). Plus précisément, l'objectif était de permettre à l'utilisateur de donner des indications au programme, soit d'emblée, soit de manière plus interactive au vu des premiers résultats, tout en gardant à l'esprit qu'il fallait être capable de différencier les influences de ces indications sur les résultats.

Redonnons une brève description du problème. L'algorithme prédit la structure d'une protéine transmembranaire, qui ressemble en principe au schéma de la figure 44 (empruntée à l'étudiant).

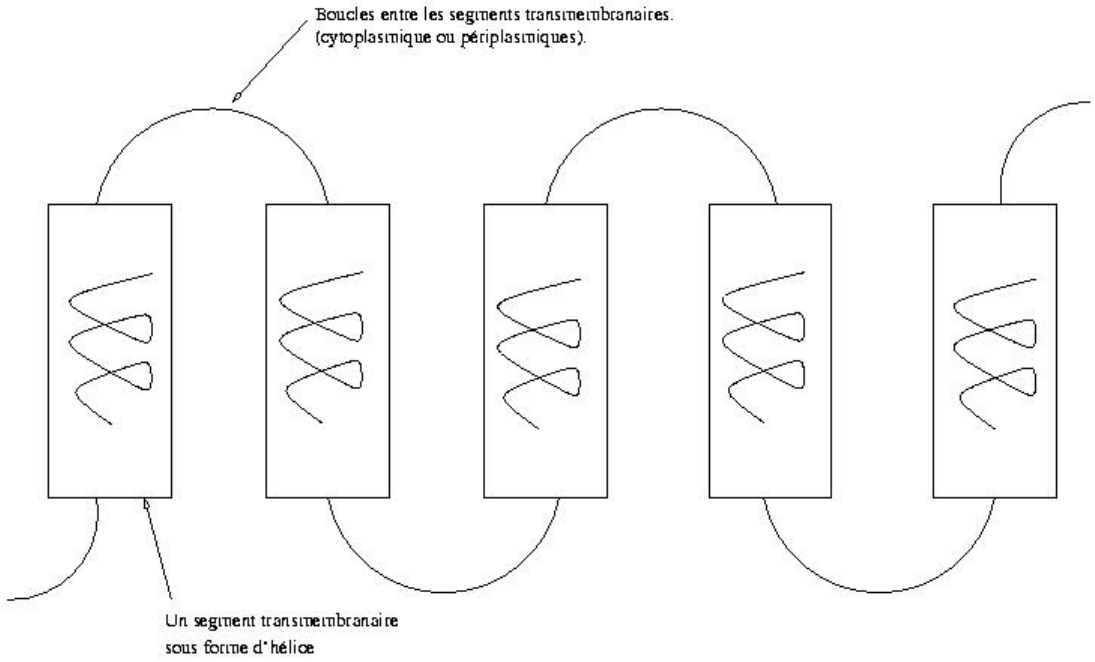


FIG. 5.44 – Une protéine transmembranaire.

Ces segments sont de forme hélicoïdale, et leur particularité est d'être fortement hydrophobes (ce qui rend la paroi de la cellule étanche). L'idée de l'algorithme est par conséquent de situer les segments hydrophobes dans la séquence protéique par calcul de son profil d'hydrophobicité. La figure 45 montre une telle prédiction (les segments commencent là où il y a des pics). Mais l'on voit que, à cause d'une des contraintes de l'algorithme qui spécifie une taille constante pour chaque segment et donc une distance minimum entre les segments, un des segments probables n'a pas été pris en compte par l'algorithme.

Si nous nous plaçons maintenant dans la situation d'un biologiste consultant les banques de données de structures, comme PDB (Protein Data Bank), nous pourrions par exemple observer que le premier segment est placé par l'algorithme bien après le début de la structure hélicoïdale, et que, très vraisemblablement, ce segment doit débuter un peu en amont. La figure 46 montre que l'utilisateur, en indiquant au programme un début de segment à cet endroit, peut permettre à l'algorithme de voir le segment qui était caché. Ce scénario a été choisi à la suite d'un atelier de prototypage décrit dans le chapitre II.

La technique qui a été utilisée est un mixin, qui intercepte deux méthodes parmi les étapes de l'analyse :

1. la méthode segments, qui détermine les segments à prendre en compte,
2. la méthode result_segments, qui prépare les données pour l'affichage.

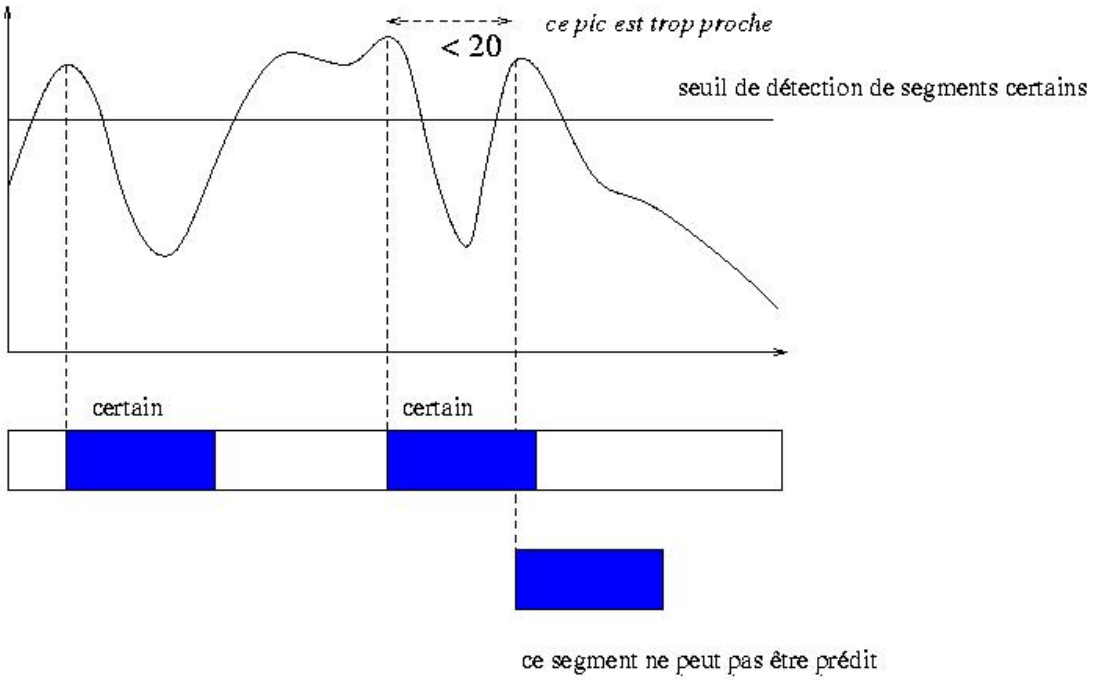


FIG. 5.45 – Segments trop proches.

Dans le mixin interceptant la première méthode, deux actions sont effectuées :

1. un premier appel de la méthode d'origine (segments),
2. un second appel de cette méthode, avec une structure de données modifiée, dans laquelle les positions indiquées par l'utilisateur sont marquées comme prioritaires.

Il ne reste plus à la méthode d'affichage des résultats result_segments qu'à comparer les deux calculs afin d'afficher de manière visible les différences entre les résultats de l'algorithme et ceux dûs à l'intervention de l'utilisateur.

Nous avons fait en sorte que ce paramétrage supplémentaire de l'utilisateur se fasse dans la syntaxe de l'interface graphique : par sélection de cellules dans l'alignement et activation d'un bouton "User data", dont le nom volontairement vague (et générique) veut montrer que cette méthode peut s'appliquer à d'autres problèmes, comme celui de la spécification de contraintes dans un alignement multiple. La figure 47, extraite du document de l'étudiant, montre l'interface de l'éditeur d'alignement pour effectuer ces opérations : les boutons "SetData" et "RemoveData" servent à désigner les positions de début de segment de l'utilisateur. La palette indique quelles couleurs sont associées aux segements : les segments qui n'apparaissaient pas sans l'interaction avec l'algorithme sont "certainmod" et "putatifmod". Le segment coloré en rouge est celui qui a été sélectionné par l'utilisateur : il est suivi d'un segment "certainmod", c'est-à-dire un segment certain qui n'apparaissait pas sans l'information venant de l'utilisateur.

L'intérêt de cette technique réside dans les points suivants :

- l'algorithme d'origine reste intact (il est possible qu'une re-structuration qui ne change pas la sémantique soit négociée avec l'auteur de l'implémentation d'origine),
- on peut revenir à tout moment au comportement d'origine : les mixins sont dynamiques,
- le mixin ne porte que sur les objets sur lesquels il a été spécifiquement placé ; par conséquent, celles des séquences des l'alignement qui n'ont pas d'indication venant de l'utilisateur sont analysées normalement.

La question de la flexibilité algorithmique n'est bien entendu pas réglée par ce type de dispositif tech-

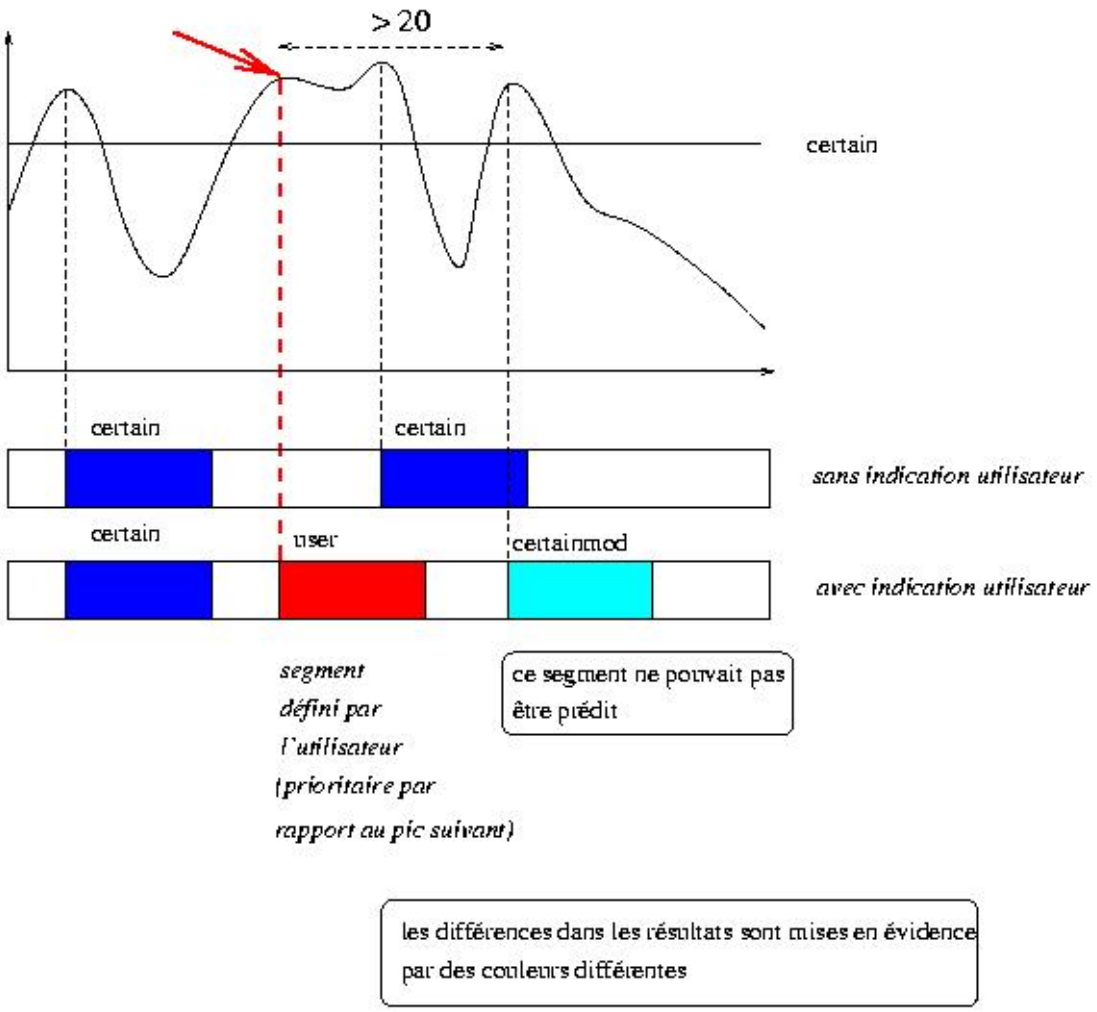


FIG. 5.46 – Segment indiqué par l'utilisateur.

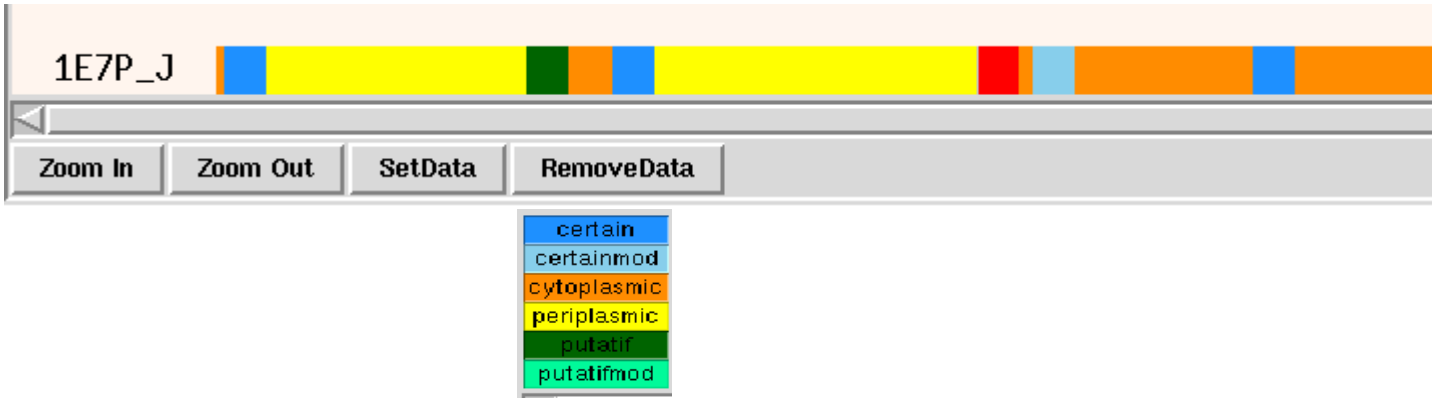


FIG. 5.47 – Sélection des "User data" et visualisation zoomée des segments.

nique, et ne doit pas être prise à la légère. Elle fait appel à une complexité, sans doute d'une autre nature que la complexité algorithmique, sur laquelle les travaux de Peter Wegner [Weg97b][Weg97a][Weg98][WG99a] donnent un éclairage théorique. Cette complexité doit pouvoir faire l'objet d'une réflexion théorique, et son application aux algorithmes biologiques nécessite certainement une coopération étroite avec l'auteur de l'algorithme d'origine, afin que l'interaction ait un sens défini par rapport à cet algorithme. Ce que nous avons voulu montrer à travers ce travail de stage, c'est qu'il y avait quelque chose à explorer, qui relève pleinement de notre problématique interaction-programmation.

5.4.7 Environnements similaires

Si l'on prend séparément chacun des aspects de cet environnement, il existe de très nombreux systèmes similaires :

1. Langages embarqués : les logiciels comme Mathematica, Hypercard, permettent non seulement le scriptage, mais également l'ajout de nouvelles fonctions. La principale différence avec notre prototype tient dans le choix volontaire d'un langage général et très utilisé, Tcl, dont les utilisateurs pourront se servir dans d'autres contextes. Il nous semble également que le choix de l'extension objet XOtcl, de bonne qualité et particulièrement adaptée à ce type de développement, apporte une bonne structuration.
2. La programmation dans le contexte de l'interface utilisateur a été à l'origine de nombreux systèmes, comme LabView ou MatLab, mais ces derniers ne sont pas adaptés au domaine de l'analyse de séquences, particulièrement axés sur le traitement de chaînes de caractères et l'analyse de texte (d'où le succès du langage perl en bio-informatique).
3. Parmi les environnements de programmation comportant des outils de navigation et de déboguage sophistiqués, Smalltalk est exemplaire, ainsi que Self [SMU95]. Pour le premier, nous n'avons d'une part pas eu le courage de convaincre les biologistes et les bio-informaticiens d'utiliser ce langage. Smalltalk, par ailleurs, comme nous l'avons déjà dit dans d'autres chapitres, est très orienté développement de logiciels (même si nous aurions pu développer un environnement applicatif complémentaire), et ne permet pas la programmation d'objets sans classes. En ce qui concerne Self, le problème était essentiellement pratique : ce système (génial) est peu porté et nécessite une machine très performante. Un environnement de programmation, XotclIDE, inspiré de celui de Smalltalk a été développé par un des utilisateurs du langage XOtcl, Artur Trzewik. Il comporte un navigateur pour les classes, un inspecteur d'objets, un débogueur, un gestionnaire de configuration ainsi qu'un outil de gestion de versions (`http://www.xdobry.de/xotclIDE/`).
4. Un système analogue au notre, plus spécialisé dans le domaine de l'analyse de données et des statistiques, est le système R (`http://www.R-project.org/`). Ce système est un ensemble de fonctions de bibliothèques et de composants graphiques, complètement extensibles (dû à sa nature de logiciel libre) et bénéficiant d'ailleurs effectivement de contributions de nombreux utilisateurs. Il nous a été fait part de son utilisation dans un laboratoire de bio-informatique.
5. Il existe un certain nombre de logiciels scriptables en biologie, comme ceux que nous avons décrits dans le chapitre IV. Cette technique permet de piloter le logiciel et d'accéder à son interface programmatique, mais non de modifier le logiciel lui-même.
6. Les outils de visualisation, comme les éditeur d'alignement, du type de celui que nous avons développé, sont également extrêment nombreux, et souvent très riches. La principale différence avec biok concerne les possibilités de programmation, qui sont plus étendues que les fonctions de personnalisation des menus ou des palettes de couleurs associées aux propriétés.
7. Le modèle de tableur étendu a été exploré à plusieurs reprises :
   - ACE (Application Construction Environment) [JNZM93] est un logiciel qui permet, à des niveaux différents selon les utilisateurs, de construire un logiciel à travers son interface. Le niveau de base correspond à l'idée d'un tableur ; le langage des formules est adapté au formalisme visuel ; les formules peuvent être saisies et définies de manière interactive. Le système comporte également quelques aspects introspectifs car tout est programmé pour cela au niveau des bibliothèques en C++. Nous n'avons pas pu essayer ce système, qui a cependant constitué un modèle pour nous. L'implémentation en C++ nous semble cependant être un choix qui ne permets pas une flexibilité totale : une surcouche a dû être écrite pour permettre l'instrospection,

d'une part, et d'autre part l'écriture en C++ semble d'un accès plus difficile que Tcl, a priori, surtout s'il faut à chaque extension penser à compléter cette surcouche. Mais il y a aussi dans biok un certain nombre de composants écrit en C, comme la fonction de recherche de motifs avec erreurs.

– Forms/3 [BAD+00] est également un système utilisant le modèle tableur et permettant la construction d'applications par l'utilisateur. Il utilise Lisp, qui est un langage général et enseigné dans certains établissements ou organismes de biologie. Il comporte de plus un formalisme correspondant bien à l'alignement de séquences, la matrice, avec des outils de manipulation de régions. La règle des valeurs (value rule), consistant à respecter le fait que toute valeur, y compris graphique, soit le résultat d'une formule, nous a cependant semblé trop contraignante.

On voit que l'argument de transférabilité de la connaissance du langage a une importance considérable, soulignée d'ailleurs par [Eis97] comme une limite potentielle des langages de programmation utilisateur. Les biologistes sont parfois prêts à faire l'effort d'apprentissage de la programmation, parce qu'ils en perçoivent les bénéfices, mais, c'est souvent explicite, à condition que cet investissement soit profitable. Enfin, cette thèse se déroulant en entreprise, il nous a semblé préférable de faire des choix de développements réalistes afin que le prototype puisse être réellement utilisé et distribué.

5.5 Conclusion

Les trois programmes présentés dans ce chapitre n'ont pas un statut équivalent.

– Le générateur d'interface Pise est un système en production sur plusieurs sites Web. Son objectif n'est pas d'explorer des aspects de recherche, mais de fournir une aide concrète aux biologistes pour effectuer leur travail quotidien. Ce système nous a cependant permis de comprendre l'importance d'une interface utilisateur permettant aux biologistes de combiner des outils, ce qu'ils ne peuvent généralement pas faire s'ils ne savent pas écrire de scripts. Ceci appartient à la catégorie "programmation inutile" dont nous avons parlé dans les chapitres précédents. Ce système de combinaison est cependant limité : il n'est pas possible sans programmer de modifier l'ordre de combinaison, ni de supprimer une étape, ni généralement de désigner les étapes. La fonction de génération de macro, encore un peu expérimentale sur le plan de l'exécution réelle, donne cependant accès au script perl généré : il suffit donc de le sauvegarder et de l'éditer (cette fonction est réservée aux utilisateurs locaux).
– Le prototype présenté dans la section 5.3 n'est pas du tout utilisé, ni utilisable : il a servi à l'exploration technique et comme support pour des entretiens menés avant un atelier de brainstorming.
– biok est encore un prototype, qui nécessite un travail supplémentaire d'optimisation, de conception et de documentation. Il a déjà été utilisé par 5 ou 6 personnes, et est installé sur plusieurs systèmes Unix (Linux, Compaq/Tru64 UNIX et SunOs). L'installation est assez complexe, car l'environnement, qui est aussi un environnement d'intégration, utilise un grand nombre de bibliothèques. Nous espérons pouvoir améliorer ces aspects dans un futur proche. Le système a été présenté publiquement lors d'un séminaire local à l'Institut Pasteur, où il semble avoir suscité un intérêt réel auprès des biologistes, ainsi que lors d'une conférence de bio-informatique (BOSC'2001) [Let01] axée sur le logiciel libre et la fabrication de composants pour la bio-informatique.

Du point de vue de notre démarche de recherche, l'environnement biok ne comporte pour l'instant que les aspects "langage embarqué", "programmation coopérative" et "programmation dans l'interface", dans un sens étendu par rapport à l'acception de [Hal93], pour qui l'approche PITUI ne concerne que la programmation dans le langage de l'interface, mais dans un sens cohérent avec l'idée de programmation et d'apprentissage situé dans un contexte [SBMP97][CSBA90]. Plusieurs améliorations sont possibles, comme l'introduction de techniques d'inférence de motifs pour la spécification d'expressions régulières, ou de modèles plus généraux comme ceux de [SVS97], qui s'apparente à de la programmation par démonstration, et dont une des applications avait fait l'objet d'un atelier. Le développement de nouveaux formalismes visuels est prévu, et un stage de bio-informatique de deux mois est en cours pour en explorer une première idée, autour de la notion de carte génétique éditable et programmable, la carte étant un concept évidemment pertinent en biologie, voire une bonne métaphore pour la navigation.

Ce sont les techniques de construction d'interface qui sont un peu rudimentaires pour l'instant. Certains mécanismes existent pour composer des objets graphiques ou étendre l'interface.

- Il est possible d'ajouter des boutons dans le panneau et de les sauver.
- On peut intégrer un objet graphique dans un autre (implémenté pour le texte, les fenêtres à panneaux (panedwindow) et les dossiers à onglets (tabnotebook), soit par copier-coller, soit par un menu adapté associé à l'objet imbriquant.

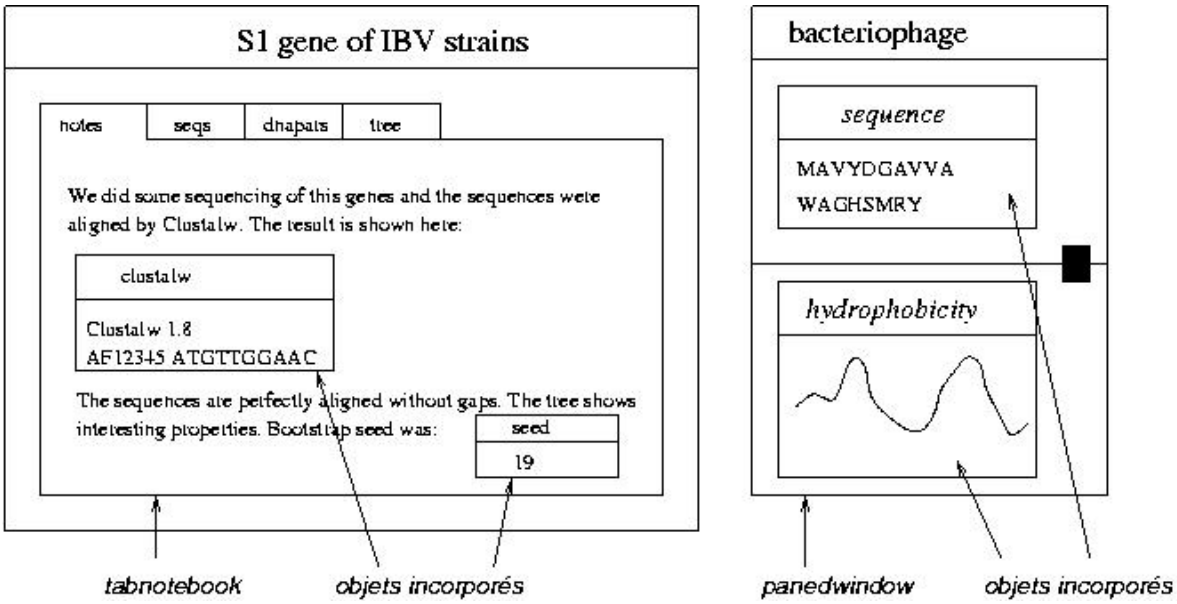


FIG. 5.48 – Objets imbriqués.

Mais la construction s'arrête pour l'instant à une instance particulière d'objet, qu'il est possible de rendre persistent (figure 48) ; il n'y a pas pour l'instant de mécanismes particuliers, à part l'édition de la méthode draw, pour effectuer un changement de composition graphique d'une classe, ou pour en créer une nouvelle par composition. Deux ou trois exemples de ce type de fonctions se sont présentés avec des utilisateurs :

1. ajouter une champ non-éditable pour indiquer la position courante dans l'alignement (non implémenté), ou la position d'un point de la courbe dans l'outil d'affichage de courbes (implémenté par l'étudiante biologiste avec un peu d'aide),
2. ajouter des sliders dans l'outil d'affichage de courbes pour régler l'intervalle affiché.

D'autres techniques d'intégration de la programmation dans l'interface, ou inversement de l'interface dans la programmation peuvent être explorées, comme par exemple l'introduction d'éléments graphiques dans le code à la manière de [EM95], un langage visuel hétérogène contenant du texte pour les structures de contrôle ou de données complexes, et des figures pour les spécifications visuelles. Ce système comporte un analyseur d'images qui transforme les dessins en graphe, tandis qu'une grammaire attribuée permet de le transformer en son équivalent textuel. Le système requiert un éditeur hybride spécialisé.

Ce type d'outils peut servir utilement non seulement pour la spécification, ne serait-ce que sous forme de menus pour aider à choisir une valeur initiale parmi un ensemble de valeurs (figure 49), mais aussi pour la visualisation : ainsi un arbre phylogénétique spécifié sous forme de liste pourrait apparaître sous forme de dessin dans le corps de la méthode qui utilise la structure de données, comme une vue supplémentaire sur cet objet. Cela pourrait être un moyen pour l'utilisateur d'intervenir dans l'algorithme, couper une branche d'exploration sans intérêt, etc...

```
tag2 proc initattributes { args} {
    [self] instvar attrscripts
    set attrscripts(0.1) [list [list background "LightSkyBlue1"]]
    set attrscripts(0.2) [list [list background "LightSkyBlue3"]]
    set attrscripts(0.3) [list [list background "LightBlue1"]]
    set attrscripts(0.4) [list [list background "LightBlue3"]]
    set attrscripts(0.5) [list [list background "CadetBlue2"]]
    set attrscripts(0.6) [list [list background "CadetBlue4"]]
    set attrscripts(0.7) [list [list background "SteelBlue2"]]
    set attrscripts(0.8) [list [list background "SteelBlue4"] \
     [list Attributes: background  background Color: FloralWhite  "FloralWhi
    set attrscripts(0.9) [list [list background "DeepSkyBlue2"]]
    set attrscripts(1.0) [list [list background "DeepSkyBlue4"]]
}
```

FIG. 5.49 – Menu dans le code.

Conclusion générale

Les questions qui ont été posées dans l'introduction peuvent être succintement reformulées de cette manière :

1. Y a-t-il des moyens informatiques complémentaires aux moyens techniques pour faciliter l'accès des biologistes à la programmation ?
2. Est-il nécessaire de faire une thèse en informatique pour que les biologistes puissent mieux contrôler les systèmes informatiques dont ils ont besoin ?

La première question ne signifie pas que nous n'avons utilisé aucune des solutions techniques, au contraire, nous en avons utilisé certaines (la réflexivité, essentiellement). Elle signifie que nous avons choisi de chercher dans une autre direction. Les observations, les constatations qui ont été faites proviennent de la prise en compte du contexte de travail et de détails pragmatiques. Une de ces constatations est que de nombreux biologistes ne désirant pas devenir programmeurs mais ayant besoin de plus de flexibilité et de contrôle sont prêts à faire un effort d'apprentissage de la programmation comme une activité utilitaire, mais que les outils qu'ils utilisent ne supportent pas du tout cet effort.

La réponse à la deuxième question sera plus personnelle. Sans la rencontre de chercheurs qui réfléchissent sur les moyens d'améliorer les interactions entre les personnes et les ordinateurs, sans leur influence et les pistes qu'ils ont pu me donner, sans l'intérêt et les discussions qu'a pu susciter mon travail, sans leur exigence aussi, les choix conceptuels et techniques qui soutiennent mon approche n'auraient peut-être pas été explorés, car, sans être toujours récents ni spectaculaires, ils ne sont pas dans la culture des informaticiens. Un chercheur en informatique, surtout en interaction homme-machine, est quelqu'un qui n'admet pas l'état des choses, pour qui les difficultés techniques ont une histoire et pour qui cette histoire est loin d'être finie. Cet enthousiasme, s'il a bousculé ma façon de travailler, est essentiel pour aborder le sujet qui m'a préoccuppée pendant ces années comme un problème de fond, tout en étant sans cesse illustré par l'expérience quotidienne.

A titre d'illustration, citons ces autres questions qui nous sont parfois posées lors de la présentation de notre travail ou de nos idées :

1. Avec l'idée de flexibilité algorithmique, n'est-il pas potentiellement dangereux de laisser l'utilisateur donner des indications erronées à un algorithme, dont les résultats n'auront plus aucun sens ?
2. Avec les mécanismes permettant à l'utilisateur de modifier le système, et d'avoir sa propre version, ne va-t-on pas avoir de nombreuses versions incompatibles ?
3. Avec les méthodes participatives, n'a-t-on pas peur de devoir satisfaire tous les besoins forcément innombrables de tous ces utilisateurs différents ?

Ces questions, qui s'inquiètent de ce que les frontières entre programmation et interaction, programmation et utilisation ou conception et utilisation ne sont plus si nettes, et ce, pas seulement sur le plan technique, mais sur le plan de la répartition des rôles, montrent que l'approche que nous avons choisie : interroger ces frontières et utiliser des méthodes de conception qui les effacent, n'est pas trop inappropriée car elle a le mérite de poser *une question directe* . Face à la recherche de moyens, et il en existe de très nombreux, pour augmenter la flexibilité et la programmabilité des outils, le problème qui est posé par ces questions est ainsi : veut-on *vraiment* cette flexibilité, veut-on *vraiment* que les non-professionnels puissent programmer, sans être cantonnés dans des régions prévues d'avance dans les couches superficielles du logiciel, c'est-à-dire pas nécessairement là où sont les problèmes qu'ils

rencontrent réellement, comme ceux que nous avons cités dans le premier chapitre ? La difficulté technique, les différences souvent soulignées au cours de workshops entre professionnels et non-professionnels [GRS92], programmeurs formels ou informels [GKMT99], etc... ne sont-elles pas surestimées par rapport à la réalité, qui montre qu'il y a aussi une informatique utilitaire, qui n'est ni un art ni une science, mais simplement un moyen de travailler ?

Il y a une sorte de paradoxe à vouloir d'une part aider les utilisateurs à programmer et à penser que cela leur est suffisamment utile pour créer des outils astucieux ou mettre en place des structures d'enseignement soigneusement conçues pour eux, et à penser d'autre part qu'il faut limiter la flexibilité et la programmabilité des logiciels qu'ils utilisent, et qui sont le lieu principal où la programmation leur est *utile et accessible* , sous prétexte que cela va créer du désordre algorithmique ou logiciel.

Les premiers essais qui ont été faits autour du prototype biok, à l'occasion des stages d'étudiants qui se sont déroulés ces derniers mois, semblent confirmer l'hypothèse de l'efficacité de la démarche de programmation dans le contexte de l'interface, et ce, malgré une utilisabilité encore insuffisante de cet environnement. La possibilité de programmer interactivement est reconnue depuis longtemps à travers les environnements comme Lisp ou Smalltalk, ainsi que l'intérêt pour des non-professionnels de travailler sur des concepts proches de leur domaine. Les méthodes participatives, encore peu utilisées en France mais très populaires dans le nord de l'Europe et aux États-Unis, sont également reconnues comme efficaces pour créer des systèmes informatiques acceptables, c'est-à-dire à la fois utiles et utilisables, mais aussi praticables et socialement pertinents [Nie93]. Ce que nous avons essayé de faire, c'est de rapprocher ces trois idées, et le retour que nous avons sur cette approche est très positif, particulièrement de la part des biologistes.

Un des résultats les plus dérangeants de la démarche participative aura été la constatation, pourtant prévisible, que la programmation n'est pas au centre des objectifs des utilisateurs : elle est, le plus souvent, la solution en cas d'échec de l'application, et cela se constate dans les ateliers par le fait qu'elle est toujours périphérique, secondaire par rapport aux fonctions importantes d'analyse biologique. Elle est également une démarche a posteriori : ce résultat me convient, comment en conserver la procédure ? Si nous voulons rendre la programmation utile, utilisable et accessible, c'est en reconnaissant ces situations-là et tout ce qu'elles impliquent techniquement, sur le plan de l'accessibilité et sur le plan de la programmation par interaction. Une des techniques qui mériteraient à notre avis une réflexion de fond est celle, en réalité proche des problématiques de programmation par démonstration, de la déduction d'une classe à partir d'une instance, mais sans restriction à certaines interactions ou parties de l'interface prévues d'avance. Puisque dans notre environnement, l'utilisateur peut sans restriction créer de nouveaux éléments à tous les niveaux, ce type de mécanisme doit être suffisamment général. Nous avons commencé à explorer les possibilités de persistance, mais nous nous sommes heurté à une difficulté théorique, celle qui consiste à distinguer les aspects statiques des aspects dynamiques d'une application. Peut-être y a-t-il des éléments de réponse du côté des études sur la réflexivité comportementale et structurale, ainsi que du côté des travaux sur les rapports entre persistence et interaction [GW98] [GW99] [GSW00].

Bibliographie

[AADR98] Carlos Agon, Gérard Assayag, Olivier Delerue, and Camilo Rueda. Objects, time and constraints in openmusic. In *Proc. ICMC 98, Ann Arbor, Michigan, I.C.M.A., San-Francisco, 1998.*, 1998.

[ACO+97] P Aloy, J Cedano, B Oliva, FX Aviles, and E Querol. Transmem : a neural network implemented in excel spreadsheets for predicting transmembrane domains of proteins. *Bioinformatics*, 13(3) :231–234, June 1997.

[ADST00] Ritu Agarwal, Prabuddha De, Atish P. Sinha, and Mohan Tanniru. On the usability of oo representations. *CACM*, 43(10) :83–89, October 2000.

[AG00] R. Anbazhagan and E. Gabrielson. Spreadsheet-based program for alignment of overlapping dna sequences. *Biotechniques*, 26(6) :1180–1185, June 2000.

[AGK+97] A. Ambler, T. Green, T. D. Kimura, A. Repenning, and T. Smedley. 1997 visual programming challenge summary. In *IEEE Symposium on Visual Languages. Capri, Italy.*, pages 11–18, September 1997.

[All00] Stewart Allen. Vtcl. unpublished, 2000.

[AN00] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts - a tour of piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology.* Kluwer, 2000, 2000.

[ARL+99] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and O. Delerue. Computer assisted composition at ircam : Patchwork and openmusic. *Computer Music Journal*, 23(3), 1999.

[AVB01] Frédéric Achard, Guy Vaysseix, and Emmanuel Barillot. Xml, bioinformatics and data integration. *Bioinformatics*, 17(2) :115–125, February 2001.

[BA99] Mordechai Ben-Ari. Bricolage forever! In T. Green, R. Abdullah, and P. Brna, editors, *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, 1999.

[BAD+00] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3 : A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 2000. to appear.

[Bai91] A. Bairoch. Prosite : a dictionary of sites and patterns in proteins. *Nucleic Acids Res*, 25(19) :2241–2245, 1991.

[BAL99] James Beach, Allen Ambler, and Jennifer Leopold. Formulate and specify : Using visual programming to enable web-based biodiversity analysis. Technical report, DesignLab, 1999. Supported by NSF grant DBI 99-05760.

[Bas01] Malay Kumar Basu. Sewer : a customizable and integrated dynamic html interface to bioinformatics services. *Bioinformatics*, 17(6) :577–578, June 2001.

[BB94] Margaret Burnett and Marla Baker. A classification system for visual programming languages. Technical report, Department of Computer Science, Oregon State University, 1994. also published in Journal of Visual Languages and Computing, September 1994, 287-300.

[BB99] Jakob Buur and Kirsten Bagger. Replacing usability testing with user dialogue. *CACM*, 42(5) :63–66, May 1999.

[BCC+97] S. Barbosa, M.P Cara, J.R. Cereja, C.K.V. Cunha, and C.S. de Souza. Interactive aspects in switching between user interface language and end-user programming environment : a case study. In *Proceedings of the III Workshop on Multimedia, Hypermedia and Human-Computer Interaction. WOMH'97. São Carlos,SP Maio de 1997.*, pages 107–118, 1997.

[BCH99] S. Bachellier, J.M Clément, and M. Hofnung. Short palindromic repetitive dna elements in enterobacteria : a survey. *Res. Microbiol.*, 150 :627–639, 1999.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[BD95] Richard Bentley and Paul Dourish. Medium versus mechanism : Supporting collaboration through customization. In *In Proceedings of ECSCW'95*, pages 133–148, 1995.

[Bød96] S. Bødker. Applying activity theory to video analysis : how to make sense of video data in hci. In *Context and Consciousness : Activity Theory and Human-Computer Interaction*, pages 147–174. MIT Press, 1996.

[BD97] E. Birney and R. Durbin. Dynamite : a flexible code generating language for dynamic programming methods used in sequence comparison. *Proc Int Conf Intell Syst Mol Biol*, 5 :56–64, 1997.

[Bei00] E. Beitz. Texshade : shading and labeling of multiple sequence alignments using latex2 epsilon. *Bioinformatics*, 16(2) :135–139, February 2000.

[Ber87] M. Bernstein. Using spreadsheet languages to understand sequence analysis algorithms. *Comput Appl Biosci*, 3 :217–221, April 1987.

[BESS00] Susanne Bødker, Pelle Ehn, Dan Sjögren, and Yngve Sundblad. Co-operative design perspectives on 20 years with the scandinavian it design model. In *Proceedings of NordiCHI 2000, Stockholm.*, October 2000. Keynote presentation.

[Bey87] RJ Beynon. A macintosh hypercard stack for calculation of thermodynamically-corrected buffer recipes. *Comput. Appl. Biosci.*, 4(4) :487–490, November 1987.

[BG91] S. Bødker and K. Grønbæk. Design in action : From prototyping by demonstration to cooperative prototyping. In *Design at Work : Cooperative Design of Computer Systems, Chapter 11.*, pages 197–218. Hillsdale, New Jersey Lawrence Erlbaum Associates, 1991.

[BG96] Paul Brna and Judith Good. Searching for examples : An evaluation of an intermediate description language for a techniques editor. Technical report, Computing Department Lancaster University and Department of Artificial Intelligence The University of Edinburgh, 1996.

[BGK93] S. Bødker, K. Grønbæk, and M. Kyng. Cooperative design : Techniques and experiences from the scandinavian scene. In *Participatory Design : Pinciples and Practices*, pages 157–175. Hillsdale, NJ : LEA, 1993.

[BGL95] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors. *Visual Object-Oriented Programming, Concepts and Environments.* Prentice Hall, 1995. 274 pages.

[BGMSW93] J. Blomberg, J. Giacomi, A. Mosher, and P. Swenton-Wall. Ethnographic field methods and their relation to design. In *Participatory Design : Principles and Practices*, pages 123–156. Hillsdale, NJ : LEA, 1993.

[BH95] Ed Baroth and Chris Hartsough. Visual programming in the real wold. In *Visual Object-Oriented Programming, Concepts and Environments*, pages 21–42. Prentice Hall, 1995.

[BH98] Hugh Beyer and Karen Holtzblatt. *Contextual Design : Defining Customer-Centered Systems.* San Francisco Morgan Kaufmann Publishers ISBN 1-55860-411-1; QA76.9.S88B493, 1998. 472 pages.

[BHHW99] MI Bellgard, HL Hiew, A Hunter, and M Wiebrands. Orbit : an integrated environment for user-customized bioinformatics tools. *Bioinformatics*, 15(10) :847–851, 1999.

[BHP+94] Margaret M. Burnett, Richard Hossli, Timothy Pulliam, Brian VanVoorst, and Xiaoyang Yang. Toward visual programming languages for steering in scientific visualization : a taxonomy. *IEEE Computational Science and Engineering.*, pages 44–62, 1994.

[BHS96] B.B. Bederson, J.D. Hollan, and J. Stewart. An agnostic approach to scripting languages. Technical report, Department of Computer Science, University of Maryland, 1996. unpublished.

[Bir98] Ewan Birney. Software components in bioinformatics. Technical report, Sanger Center, Cambridge, dec 1998.

[Biz00] J.W. Bizzaro. Distributing scientific applications with gnu piper. unpublished, 2000.

[BJ90] J.A. Borges and R.E. Johnson. Multiparadigm visual programming language. In *in IEEE Workshop on Visual Languages. Skokie, Illinois. 4-6 October 1990*, pages 233–240, October 1990.

[BL92] M. Beaudouin-Lafon. *Les langages à objets.* Armand Colin, 1992.

[BL93a] Michel Beaudouin-Lafon. La construction interactive de systèmes interactifs. In *Actes Journées du GDR-PRC Programmation Avancée et Outils pour l'IA, Orsay (France)*, pages 181–190, October 1993.

[BL93b] B. Bell and C. Lewis. Chemtrains : A language for creating behaving pictures. In *IEEE Workshop on Visual Languages, Bergen, Norway*, pages 188–195, 1993.

[BL00] Michel Beaudouin-Lafon. Instrumental interaction : An interaction model for designing post-wimp user interfaces. In *Proceedings of the ACM CHI 2000 Conference on Human Factors in Computing Systems, The Haag (NL), 1-6 April 2000*, April 2000.

[Bla96] Alan F. Blackwell. Metacognitive theories of visual programming : What do we think we are doing ? In *Proceedings IEEE Symposium on Visual Languages*, pages 240–246, 1996.

[Bla00] Alan Blackwell. Swyn : A visual representation for regular expressions. In *Your Wish is My Command : Giving Users the Power to Instruct their Software.* Morgan Kaufmann, 2000.

[BLL+99] E. Barillot, U. Leser, P. Lijnzaad, C. Cussat-Blanc, K. Jungfer, F. Guyon, G. Vaysseix, C. Helgesen, and P. Rodriguez-Tome. A proposal for a standard corba interface for genome maps. *Bioinformatics*, 15(2) :157–169, February 1999.

[BM94] Massimo Bernini and Mauro Mosconi. Vipers : A data flow visual programming environment based on the tcl language. In *in Proceedings of the workshop on Advanced visual interfaces June 1 - 4, 1994, Bari Italy.*, 1994.

[Boe86] Heinz-Dieter Boecker. The enhancement of understanding through visual representation. In *SIGCHI 1986 Proceedings*, pages 44–50. ACM Press, 1986.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications, @nd edition.* Addison-Wesley Object Technology Series, 1994.

[Bor77] Alan Borning. Thinglab - an object-oriented system for building simulations using constraints. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 497–498, 1977.

[Bor81] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *TOPLAS - ACM Transactions on Programming Languages and Systems*, 3(4) :353–387, October 1981.

[Bor90] Kjell Borg. Ishell : a visual unix shell. In *In Human Factors in Computing Systems, CHI'90 Conference Proceedings, New York : ACM*, pages 201 – 207, 1990.

[Boy98a] John Boyle. Component design for biological applications. unpublished, 1998.

[Boy98b] John Boyle. A visual environment for the manipulation and integration of java beans. *Bioinformatics, Volume 14, Issue 8, September 1998*, pages 739–748, September 1998.

[BRL91] Brigham Bell, John Rieman, and Clayton Lewis. Usability testing of a graphical programming system : things we missed in a programming walkthrough. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '91)*, pages 7–12. ACM Press, 1991.

[Bro87] Frederick P. Brooks. No silver bullet,essence and accidents of software engineering. *Computer Magazine*, April 1987. l'url pointe sur un texte scanné.

[Bro91] JW Brown. Phylogenetic comparative analysis of rna structure on macintosh computers. *Comput Appl Biosci*, 7(3) :391–393, July 1991.

[Bro93] JW Brown. A macintosh hypercard compilation of small subunit ribosomal rna sequences. *Comput. Appl. Biosci.*, 9(4) :473, August 1993.

[BWGP98] Alan F. Blackwell, Kirsten Whitley, Judith Good, and Marian Petre. Programming in pictures, pictures of programs. In *Thinking with Diagrams : an Interdisciplinary Workshop*, 1998.

[Car95] John M. Carroll, editor. *Scenario-Based Design : Envisioning Work and Technology in System Development.* Wiley, ISBN 0-471-07659-7, 1995. 408 pages.

[CCQ94] J Clotet, J Cedano, and E Querol. An excel spreadsheet computer program combining algorithms for prediction of protein structural characteristics. *Bioinformatics*, 10(5) :495–500, September 1994.

[CCRN00] J.M. Carroll, G. Chin, M.B Rosson, and D.C. Neale. The development of cooperation : Five years of participatory design in the virtual school. In *Proceedings on Designing Interactive Systems : Processes, Practices, Methods, and Techniques*, pages 239–251. New York : Association for Computing Machinery, 2000.

[CD97] Luca Cardelli and Rowan Davies. Service combinators for web computing. In *Proc. of the First Usenix Conference on Domain Specific Languages*, 1997. SRC Research Report 148.

[CDZ93] Wayne Citrin, Michael Doherty, and Benjamin G. Zorn. Control constructs in a completely visual imperative programming language. Technical report, Department of Computer Science, University of Colorado at Boulder., 1993. Technical Report CU-CS-672-93.

[CFD+97] Stephen A. Chervitz, Georg Fuellen, Chris Dagdigian, Richard Resnick, and Steven E. Brenner. Bioperl : Object-oriented perl modules for bioinformatics. In *Oib'97, Objects in Bioinformatics- Reusable Software Components and Distributed Computing for the Biological Sciences.*, 1997.

[CFD+98] S.A. Chervitz, G. Fuellen, C. Dagdigian, R. Resnick, and S.E. Brenner. Bioperl response to the life science research group request for information. In *Object Management Group Meeting. (1997).*, 1998. Addendum : (Mar. 1998), Bioperl Object and Framework Requirements.

[CFGO98] Jonathan Crabtree, Steve Fischer, Mark Gibson, and Chris Overton. Cbil's biowidgets : Data interaction components for bioinformatics. In *Oib'98, Objects in Bioinformatics '98*, 1998. poster.

[CGP89] P.T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph : A step towards liberating programming from textual conditioning. In *Proc. IEEE Workshop on Visual Languages, Rome*, pages 150–156, 1989. Also appearing in Visual Object-Oriented Programming : Concepts and Environments, M.M. Burnett, A. Goldberg, T.G. Lewis (Eds), Prentice-Hall (1995).

[Cha98] Denys Chaume. A java(tm) application programming interface for accessing imgt/ligm-db database. In *Oib'98, Objects in Bioinformatics '98*, 1998. poster.

[Cha00] Brad Chapman. Getting programs and libraries to work with piper. unpublished, 2000.

[Che98] Steve A. Chervitz. Bioperl : Standard perl modules for bioinformatics. In *ISMB'98. Intelligent Systems for Molecular Biology. (1998). Halkidiki, Greece*, 1998. poster.

[CHZ95] Wayne Citrin, Richard Hall, and Benjamin Zorn. Programming with visual expressions. In *IEEE Symposium on Visual Languages*, 1995.

[CIS92] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *In IFIP 12th World Computer Congress, Madrid.*, volume 3, pages 54–60, 1992.

[CK00] Jarnie Chattratichart and Jasna Kuljis. An assessment of visual representations for the 'flow of control'. In Alan Blackwell and Eleonora Bilotta, editors, *Collected Papers of the 12th Annual Workshop of the Psychology of Programming Interest Group (PPIG-12), Corigliano Calabro, Cosenza, Italy*, pages 45–60, 2000.

[CKBR97] Ed H. Chi, Joseph Konstan, Phillip Barry, and John Riedl. A spreadsheet approach to information visualization. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'97)*, pages 79–80. ACM Press, 1997.

[CKM93a] Allen Cypher, David S. Kosbie, and David Maulsby. Characterizing pbd systems. In *Watch What I Do. Programming by Demonstration, Part III Perspectives.*, pages 467–484. MIT Press, 1993.

[CKM93b] Allen Cypher, David S. Kosbie, and David Maulsby. Chimera : Example-based graphical editing. In *Watch What I Do. Programming by Demonstration, Part I Systems*, pages 270–290. MIT Press, 1993.

[CKM93c] Allen Cypher, David S. Kosbie, and David Maulsby. Peridot : Creating user interfaces by demonstration. In *Watch What I Do. Programming by Demonstration, Part I Systems*, pages 124–153. MIT Press, 1993.

[CL99] J Conery and M Lynch. Genetic simulation library. *Bioinformatics*, 15(1) :85–86, 1999.

[Coc97] A Cockburn. Supporting tailorable program visualisation through literate programming and fisheye views. Technical report, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 1997.

[Coi88] P. Cointe. The objvlisp kernel : A reflexive lisp architecture to define a uniform object-oriented system. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 155–176. North-Holland, 1988.

[Coi96] P. Cointe. Reflective languages and metalevel architectures. *ACM Computing Surveys*, 28(4), 1996. article no 151, position paper.

[Col95] D Collins. *Designing object-oriented user interfaces.* Redwood City, CA : Benjamin Cummings, 1995.

[Cor92] James R. Cordy. Why the user interface is not the programming language and how it can be. In *Languages for Developping User Interfaces.* Jones and Bartlett, 1992.

[CR87] J. M. Carroll and M. B. Rosson. *Interfacing Thought : Cognitive Aspects of Human-Computer Interaction*, chapter The paradox of the active user., pages 80–111. Cambridge, Mass : MIT Press, 1987.

[CRBK98] Ed H. Chi, John Riedl, Phillip Barry, and Joseph Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 18, jul/aug 1998.

[CRC97] G. Chin, M. B. Rosson, and J. M. Carroll. Participatory analysis : Shared development of requirements from scenarios. In *In Human Factors in Computing Systems, CHI'97 Conference Proceedings, New York : ACM*, pages 162–169, 1997.

[CRSB00] Ed H. Chi, John T. Riedl, Elizabeth Shoop, and Phillip Barry. A novel visualization method for biological sequence similarity reports. *Journal of Electronic Imaging : Special Issue on Visualization and Data Analysis, (to appear). SPIE, Bellingham, WA.*, 2000.

[CSBA90] John M. Carroll, Janice A. Singer, Rachel K. E. Bellamy, and Sherman R. Alpert. A view matcher for learning smalltalk. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pages 431 – 437. ACM Press, 1990.

[CSPB94] W Chang, IN Shindyalov, C Pu, and PE Bourne. Design and application of pdblib, a c++ macromolecular class library. *Comput. Appl. Biosci.*, 10(6) :575–586, December 1994.

[CUS95] Bay-Wei Chang, David Ungar, and Randall B. Smith. Getting close to objects : Object-focused programming environments. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming, Concepts and Environments*, pages 185–198. Prentice Hall, 1995.

[Cyp91a] Allen Cypher. Customising application programs. In *In Proc. First International HCI'91 Workshop, Moscow*, pages 152–157, August 1991.

[Cyp91b] Allen Cypher. Eager : programming repetitive tasks by example. In *In Proc. CHI '91 Conference Companion.*, pages 33–39. ACM Press, 1991.

[Cyp93a] Allen Cypher. Introduction : Bringing programming to end users. In *Watch What I Do. Programming by Demonstration, Part III Perspectives.*, pages 1–11. MIT Press, 1993.

[Cyp93b] Allen Cypher. *Watch What I Do. Programming by Demonstration.* MIT Press, 1993. 652 pages.

[DA89] Andy DiSessa and H. Abelson. Boxer : a reconstructible computational medium. In *Studying the Novice Programmer.* Lawrence Elbaum Associates, 1989.

[Dav90] S. P. Davies. The nature and development of programming plans. *nternational Journal of Man-Machine Studies*, 32(4) :461–481, 1990.

[Dav00] Simon P. Davies. Expertise and the comprehension of object-oriented program. In Alan Blackwell and Eleonora Bilotta, editors, *Collected Papers of the 12th Annual Workshop of the Psychology of Programming Interest Group (PPIG-12), Corigliano Calabro, Cosenza, Italy*, pages 61–66, 2000.

[DB98a] Rebecca Walpole Djang and Margaret M. Burnett. Similarity inheritance : A new model of inheritance for spreadsheet vpls. In *IEE Symposium on Visual Languages.* ACM Press, 1998.

[DB98b] Paul Dourish and Graham Button. On "technomethodology" : Foundational relationships between ethnomethodology and system design. *Human-Computer Interaction 1998 13(4)*, pages 395–432, 1998.

[DE95a] Chris DiGiano and Michael Eisenberg. Self-disclosing design tools : a gentle introduction to end-user programming. In G. Olson and S. Schuon, editors, *In Proc. DIS'95 Symposium on action Systems*, pages 189–197. ACM Press, Ann Arbor, Michigan, 1995.

[DE95b] Chris DiGiano and Michael Eisenberg. Supporting the end-user programmer as a lifelong learner. Technical report, Department of Computer Science, University of Colorado at Boulder., 1995. Technical Report CU-CS-761-95.

[Del00] Christian Delamarche. Color and graphic display (cgd) : programs for multiple sequence alignment analysis in spreadsheet software. *Biotechniques 2000 Jul ;29(1) :100-4, 106-7*, 29(1) :100–4, 106–7, July 2000.

[Der92] Michael Dertouzos. The user interface is the language. In *Languages for Developping User Interfaces.* Jones and Bartlett, 1992.

[DGG95] S. P. Davies, D. J. Gilmore, and T. R. G Green. Are objects that important ? the effects of expertise and familiarity on the classification of object-oriented code. *Human-Computer Interaction*, 10 :227–248, 1995. volumes number 2–3.

[DiG96a] Chris DiGiano. A vision of highly-learnable end-user programming languages. In *Child's Play 1996 Position Paper*, 1996.

[DiG96b] Christopher J. DiGiano. *Self-disclosing design tools : an incremental approach toward end-user programming.* PhD thesis, Department of Computer Science, University of Colorado at Boulder., 1996. Technical Report CU-CS-822-96.

[DiS97] Andy DiSessa. Twenty reasons why you should use boxer (instead of logo) - especially for logo folks, but also a general description of boxer's advances over other programming environments. In *reprint from Learning and Exploring with Logo : Proceedings of the Sixth European Logo Conference, Budapest, Hungary, 7-27*, 1997.

[DiS99] Andy DiSessa. *Changing Minds : Computers, Learning, and Literacy.* MIT Press, 1999.

[Dop90] J Dopazo. Multiple sequence editing by spreadsheet. *Comput. Appl. Biosci.*, 6(4) :401–402, October 1990.

[Dou95] Paul Dourish. Developing a reflective model of collaborative systems. *ACM Transactions on Computer-Human Interaction*, 2(1) :40–63, March 1995.

[Dou96a] Paul Dourish. *Computational reflection and Open Implementation*, chapter 3, pages 35–48. 1996.

[Dou96b] Paul Dourish. *Open Implementation and Flexibility in CSCW Toolkits.* PhD thesis, Dept of Computer Science, University College, London, 1996.

[Dou96c] Paul Dourish. *Open Implementation and Flexibility in CSCW Toolkits*, chapter Flexibility in CSCW Toolkits. 1996.

[Dou97] Paul Dourish. Accounting for system behaviour : Representation, reflection and resourceful action. In Kyng and Mathiassen, editors, *Computers and Design in Context*, pages 145–170. Cambridge : MIT Press, 1997.

[DP95] Joseph Dumas and Paige Parsons. Discovering the way programmers think about new programming environments. *CACM*, 38(6) :45 – 56, June 1995.

[Dra92] Nikos Drakos. Object orientation and visual programming. In *OOPSLA'92*. ACM Press, 1992.

[DS91] S Dear and R Staden. A sequence assembly and editing program for efficient management of large projects. *Nucleic Acids Res*, 19(14) :3907–3911, 1991.

[dS99] C.S. de Souza. Semiotic engineering principles for evaluating end-user programming environments. Technical report, Computer Science Department, PUC-Rio, Brazil, 1999. In C.J.P. de Lucena (ed.) Monografias em Ciência da Computação.

[dS00] C.S. de Souza. Leading users from interaction into programming : The teaching-centered approach of semiotic engineering. Technical report, Computer Science Department, PUC-Rio, Brazil, 2000. unpublished - 2000 ?

[Dét98] Françoise Détienne. *Génie logiciel et psychologie de la programmation.* Hermes, 1998. 184 pages.

[DTM91] R. Durbin and J. Thierry-Mieg. A c. elegans database. unpublished, 1991.

[Dub99] Paul F. Dubois. Ten good practices in scientific programming. *Computing in Science and Engineering*, pages 7–11, 1999.

[Dwo96] Garett Dworman. Homer : a pattern discovery support system. In *Proceedings of the CHI '96 conference companion on Human factors in computing systems : common ground. April 13 - 18, 1996, Vancouver Canada*, pages 305 – 306, April 1996. short paper.

[Ede90] M Edel. The tinkertoy graphical programming environment,. In *Visual Programming Environments : Paradigms and Systems, Glinert, E.* IEEE Comp Sci Press, Los Alamitos, CA, 1990.

[Eer92] D.J. Eernisse. Dna translator and aligner : Hypercard utilities to aid phylogenetic analysis of molecules. *Comput Appl Biosci*, 8(2) :177–184, April 1992.

[EF94] Michael Eisenberg and Gerhard Fischer. Programmable design environments : integrating end-user programming with domain-oriented assistance. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '94 Boston United States)*, pages 431 – 437. ACM Press, April 1994.

[Eis97] M Eisenberg. End-user programming. In *Handbook of Human Computer Interaction 2ed complety revised.*, pages 1127–1146. North-Holland, 1997.

[EK91] Pelle Ehn and Morten Kyng. Cardboard computers : Mocking-it-up or hands-on the future. In *Design at Work : Cooperative Design of Computer Systems, Chapter 11.*, pages 169–196. Hillsdale, New Jersey Lawrence Erlbaum Associates, 1991.

[EM95] Martin Erwig and Bernd Meyer. Heterogeneous visual languages - integrating visual and textual programming. In *In Proceedings of VL'95, 11th International IEEE Symposium on Visual Languages, September 5 - 9, 1995 Darmstadt, Germany*, 1995.

[EM98] Matthew Eldridge and Renato Mancuso. Applying object-oriented database and component technology to metabolic pathway data. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[FA96] D. Frishman and P. Argos. Incorporation of long-distance interactions into a secondary structure prediction algorithm. *Protein Engineering*, 9 :133–142, 1996.

[FA97] D. Frishman and P. Argos. 75 *Proteins*, 27 :329–335, 1997.

[FAH+01] Dmitrij Frishman, Kaj Albermann, Jean Hani, Klaus Heumann, Agnes Metanomski, Alfred Zollner, and Hans-Werner Mewes. Functional and structural genomics using pedant. *Bioinformatics*, 17(1) :44–57, January 2001.

[FCB+99] S. Fischer, J. Crabtree, B. Brunk, M. Gibson, and G.C. Overton. biowidgets : data interaction components for genomics. *Bioinformatics*, 15(10) :837–846, October 1999.

[FG84] W Finzer and L Gould. Rehearsal world : Programming by rehearsal. *BYTE*, 9(6) :187–210, June 1984.

[FG90] G. Fischer and A. Girgensohn. End-user modifiability in design environments. In *In Human Factors in Computing Systems, CHI'90 Conference Proceedings, New York : ACM*, pages 183–191, 1990.

[Fis99] Gerhard Fischer. Social creativity, symmetry of ignorance and meta-design. In L. Candy and E. Edmonds, editors, *Proceedings of Creativity and Cognition 1999*, pages 116–123. ACM Press, 1999. other title ? Symmetry of Ignorance, Social Creativity and Meta-Design.

[Fis00] Gerhard Fischer. User modeling in human-computer interaction. *Contribution to the 10th Anniversary Issue of the Journal User Modeling and User-Adapted Interaction (UMUAI)*, 2000.

[FJG95] Elisabeth Freeman, Suresh Jagannathan, and David Gelernter. In search of a simple visual vocabulary. In *IEEE Symposium on Visual Languages*, 1995.

[FL88] Gerhard Fischer and A. Lemke. Construction kits and design environments : Steps toward human problem-domain communication. *Human-Computer Interaction Journal*, 3(3) :179–222, 1988.

[Fla54] J. C. Flanagan. The critical incident technique. *Psychological bulletin*, 54(4) :327–358, 1954.

[Foo90] Brian Foote. Object-oriented reflective metalevel architectures : Pyrite or panacea ? In *A Position Paper for the ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures*, 1990.

[Fro94] K.U. Frohlich. Sequence similarity presenter : a tool for the graphic display of similarities of long sequences for use in presentations. *Comput Appl Biosci*, 10(2) :179–183, April 1994.

[Fro99] KU Frohlich. Plasmid maker stack. Technical report, Physiologisch-chemisches Institut, Tübingen, Germany, 1999. unpublished.

[G97] Shaw G. Protein sequence interpretation using a spreadsheet program. *Biotechniques*, 19(6) :978–983, June 1997.

[GAL97] Adele Goldberg, Steven T. Abell, and David Leibs. The learningworks development and delivery frameworks. *Communications of the ACM (CACM)*, 1997.

[GB96] Judith Good and Paul Brna. Novice difficulties with recursion : Do graphical representations hold the solution ? In *In Proceedings of the European Conference on AI in Education, Lisbon, Portugal, September 30 – October 2, 1996.*, 1996.

[GB97] H. Gottfried and M. Burnett. Programming complex objects in spreadsheets : An empirical study comparing textual formula entry with direct manipulation and gestures. In *Empirical Studies of Programmers : Seventh Workshop 1997*, 1997.

[GB98] T R G Green and A F Blackwell. *A tutorial on cognitive dimensions*, 1998. This tutorial has 3 parts : an introduction to the framework ; analysis of two real-life examples ; and a set of interactive virtual devices, accessible via the web, which illustrate different design decisions and their trade-offs.

[GBC97] J Good, P Brna, and R Cox. Novices and program comprehension : does programming language make a difference ? In *Procs of Cognitive Science*, 1997. envoyé par l'auteur.

[GGRK98] Ignacio Gil, Peter Gray, Chris Robertson, and Graham Kemp. A java-based visual interface and navigator for the p/fdm protein structure database. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GHKB00] H.H. Gonnet, M.T. Hallett, C. Korostensky, and L. Bernardin. Darwin v. 2.0 : an interpreted computer language for the biosciences. *Bioinformatics*, 16(2) :101–103, 2000.

[Gil90] DG Gilbert. Two hypercard calculators for molecular biology. *Comput. Appl. Biosci.*, 6(2) :113–116, April 1990.

[Gir93] A. Girgensohn. Modifier : Improving an end-user modifiable system through user studies. In *Human-Computer Interaction, Vienna Conference, Berlin : Springer-Verlag*, pages 141–152, 1993.

[Gir00] Patrick Girard. *Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur.* PhD thesis, Université de Poitiers, 2000. Habilitation à diriger les recherches.

[GKMT99] Howie Goodell, Sarah Kuhn, David Maulsby, and Carol Traynor. Report of the end-user programming/informal programming workshop at chi 99. In *End User Programming / Informal Programming*, 1999.

[GMP98] Elena Ghittori, Mauro Mosconi, and Marco Porta. Designing and testing new programming contructs in a data flow vl. In *In Proceedings of the 14th IEEE Symposium on Visual Languages, Halifax, Canada, September 1-4, 1998*, 1998.

[GNZ00] Michael Goedicke, Gustaf Neumann, and Uwe Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000), Erfurt, Germany, Oct 9-12, 2000.*, October 2000.

[Goo87] Danny Goodman. *The Complete HyperCard Handbook.* Bantam Books, 1987.

[Goo99] Judith Good. *Programming Paradigms, Information Types and Graphical Representations : Empirical Investigations of Novice Program Comprehension.* PhD thesis, University of Edinburgh, 1999.

[GP92] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction : Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). CUD : Rome.*, 1992.

[GP97a] T. R. G. Green and M. Petre. Usability analysis of visual programming environments : a 'cognitive dimensions' framework. *J. Visual Languages and Computing, 7*, pages 131–174, 1997.

[GP97b] N Guex and MC Peitsch. Swiss-model and the swiss-pdbviewer : an environment for comparative protein modeling. *Electrophoresis*, 18(15) :2714–2723, December 1997.

[GPB91] T. R. G. Green, M. Petre, and R. K. E. Bellamy. Comprehensibility of visual and textual programs : A test of superlativism against the 'match-mismatch' conjecture. In *Empirical Studies of Programmers : Fourth Workshop 1991 p.121-146*, 1991. à trouver.

[Gre89] T.R.G Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *Conference on People and Computers V.* Cambridge University Press, 1989.

[Gre91] T. R. G. Green. The nature of programming. In *The Psychology of Programming*, pages 21–44. London : Academic Press, 1991.

[Gre93] J. Greenbaum. Pd, a personal statement. *CACM*, 36(6) :47, June 1993.

[Gre95] T. R. G Green. Noddy's guide to visual programming. *Interfaces (Newsletter of the British Computer Society Human-Computer Interaction Group)*, 1995.

[Gre96] T.R.G Green. An introduction to the cognitive dimensions framework. In *MIRA workshop, Monselice, Italy.*, 1996. Extended abstract of invited talk.

[Gre97] T.R.G Green. Cognitive approaches to software comprehension : Results, gaps and limitations. In *Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97, University of Limerick, Ireland.*, 1997.

[Gre99] T R G Green. Language design and informal programmers. In *CHI'99 Workshop on End-User Programming and Blended-User Programming*, 1999. position paper.

[Gre00] T. R. G. Green. Instructions and descriptions : some cognitive aspects of programming and similar activities. In V. Di Gesù, S. Levialdi, and L. Tarantino, editors, *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000)*, pages 21–28, 2000. invited paper.

[GRPM99] I Goncalves, M Robinson, G Perriere, and D Mouchiroud. Jadis : computing distances between nucleic acid sequences. *Bioinformatics*, 15(5) :424–425, May 1999.

[GRS92] M Guzdial, J. Reppy, and Randall Smith. Report of the user / programmer distinction workshop. In *Languages for Developping User Interfaces*. Jones and Bartlett, 1992.

[GRS95a] Nathan Goodman, Steve Rozen, and Lincoln Stein. The case for componentry in genome information systems. Technical report, Whitehead Institute for Biomedical Research, 1995.

[GRS95b] Nathan Goodman, Steve Rozen, and Lincoln Stein. The importance of standards and componentry in meeting the genome informatics challenges of the next five years. In *Second Meeting on the Interconnection of Molecular Biology Databases, July 20-22, 1995, Cambridge, United Kingdom*, July 1995.

[GRSS98] N Goodman, S Rozen, LD Stein, and AG Smith. The labbase system for data management in large scale biology research laboratorie. *Bioinformatics*, 14(7) :562–574, July 1998.

[GS96] T Gaasterland and CW Sensen. Fully automated genome analysis that reflects user needs and preferences. a detailed introduction to the magpie system architecture. *Biochimie*, 78(5) :302–310, 1996.

[GSW00] Dina Goldin, Scott Smolka, and Peter Wegner. Turing machines, transition systems, and interaction. Technical report, Department of CS, University of Massachusetts, Boston, October 2000. UMB CS Technical Report 00-7.

[Guz98] Mark Guzdial. What got lost in the translation. In *CHI 98 Learner-Centered Design Workshop*, 1998.

[GW98] Dina Goldin and Peter Wegner. Persistence as a form of interaction. Technical report, Brown University, 1998. Brown Technical Report CS 98-07.

[GW99] Dina Goldin and Peter Wegner. Behavior and expressiveness of persistent turing machines. Technical report, Brown University, 1999. Brown Technical Report CS 99-14.

[Hal93] Daniel C. Halbert. Smallstar : Programming by demonstration in the desktop metaphor. In *Watch What I Do. Programming by Demonstration, Part I Systems.*, pages 238–269. MIT Press, 1993.

[Han94] Wilfred J. Hansen. The 1994 visual languages comparison. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, August 1994. Moderator : Wilfred J. Hansen, Director, Andrew Consortium, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891.

[Har87] D Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[Hay93] M. G Haygood. Spreadsheet macros for coloring sequence alignments. *Biotechniques*, 15(6) :1084–1089, 1993.

[hCRS+96] Ed Huai hsin Chi, John Riedl, Elizabeth Shoop, John V. Carlis, Ernest Retzel, and Phillip Barry. Flexible information visualization of multivariate data from biological sequence similarity searches. In *In Proc. of IEEE Visualization '96, pp. 133–140, 477. IEEE CS Press, 1996*, pages 133–140, 1996.

[Hei92] Gunnar Von Heijne. Membrane protein structure prediction. hydrophobicity analysis and the positive-inside rule. *J. Mol. Biol*, 225(2) :487–494, 1992.

[Hen95] David G. Hendry. Display-based problems in spreadsheets : A critical incident and a design remedy. In *In Proceedings of VL'95, 11th International IEEE Symposium on Visual Languages, September 5 - 9, 1995 Darmstadt, Germany*, 1995.

[HFA99] H Hermjakob, W Fleischmann, and R Apweiler. Swissknife - 'lazy parsing' of swiss-prot entries. *Bioinformatics*, 15(9) :771–772, September 1999.

[HH97] Stephanie Houde and Charles Hill. What do prototypes prototype ? In *Handbook of Human Computer Interaction 2ed complety revised.*, pages 367–381. North-Holland, 1997.

[HH99] Jed Harris and Austin Henderson. A better mythology for system design. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems*. ACM Press, 1999.

[Hil92] Daniel D. Hils. Visual languages and computing survey : Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(4) :69–101, 1992.

[HJ93] Karen Holtzblatt and Sandra Jones. Contextual inquiry : A participatory technique for system design. In *Participatory Design : Principles and Practices*, pages 177–210. Hillsdale, NJ : LEA, 1993.

[HK91] Austin Henderson and Morten Kyng. There's no place like home : Continuing design in use. In *Design at Work : Cooperative Design of Computer Systems*, pages 219–240. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1991.

[HL91] Christian Heath and Paul Luff. Work, interaction and technology : Empirical studies of social ergonomics. Technical report, Rank Xerox Research Centre, 1991. Technical Report EPC-1991-108.

[HM88] V. Haarslev and R. Möller. Visualization of experimental systems. In *IEEE Workshop on Visual Languages, Pittsburgh/PA, Oct. 10-12, IEEE Computer Society Press*, pages 175–182, October 1988.

[HMY+86] M. Hirakawa, N. Monden, I. Yoshimoto, M. Tanaka, and T. Ichikawa. Hi-visual : A language supporting visual interaction in programming. In S. K. Chang, T. Ichikawa, and A. Ligomenides, editors, *Visual Languages*, pages 233–259. Plenum Press, 1986.

[HrHS90] GZ Hertz, GW 3rd Hartzell, and GD Stormo. Identification of consensus patterns in unaligned dna sequences known to be functionally related. *CABIOS. Comput Appl Biosci*, 6(2) :81–92, April 1990.

[HRS97] Scott E. Hudson, Roy Rodenstein, and Ian Smith. Debugging lenses : A new class of transparent tools for user interface debugging. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'97)*, pages 179–187. ACM Press, 1997.

[Hud94] Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3) :209–239, July 1994.

[Hut93] E Hutchins. Learning to navigate. In Jean Lave Seth Chaiklin, editor, *Understanding Practice*. Cambridge University Press, 1993.

[Ibr98] Bertrand Ibrahim. Diagrammatic representation of data types and data manipulations in a combined data- and control-flow language. In *Proceedings IEEE Symposium on Visual Languages*, 1998.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. *Software : Practice and Experience*, 26(6) :635–652, June 1996.

[Jag95] R. Jagannathan. Dataflow models. In E. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.

[JNZM93] Jeff A. Johnson, Bonnie A. Nardi, Craig L. Zarmer, and James R. Miller. Ace : building interactive graphical applications. *CACM*, 36(4) :40–55, 1993.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA'92*, pages 63–76. ACM Press, 1992.

[KAD+96] John Karat, Michael E. Atwood, Susan M. Dray, Martin Rantzer, and Dennis R. Wixon. User centered design : quality or quackery ? In *Proceedings of the CHI '96 conference companion on Human factors in computing systems : common ground. April 13 - 18, 1996, Vancouver Canada*, pages 161 – 162, April 1996. panel.

[Kah00] Ken Kahn. Generalizing by removing detail. *CACM*, 43(3) :104–106, March 2000.

[KAR+93] Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G Bobrow. Metaobject protocols : Why we want them and what else they can do. In A. Paepcke, editor, *Object-Oriented Programming : The CLOS Perspective.*, pages 101–118. The MIT Press, Cambridge, MA, 1993., 1993.

[Kay84] Alan Kay. Computer software. *Scientific American*, 251(3) :52–59, September 1984.

[KCM90] T.D. Kimura, J. W. Choi, and J. M. Mack. Show and tell : A visual language. In E.P. Glinert, editor, *Visual Programming Environments : Paradigms and Systems*, pages 397–404. IEEE Comp Sci Press, Los Alamitos, CA, 1990.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[KG98] Colleen M. Kehoe and Mark Guzdial. Case libraries for learning object-oriented design. Technical report, College of Computing at Georgia Tech, 1998. unpublished.

[Kic92] G. Kiczales. Towards a new model of abstraction in the engineering of software. In *IMSA'92*, 1992.

[Kim95] Takayuki Dan Kimura. Object-oriented dataflow. In *In Proceedings of VL'95, 11th International IEEE Symposium on Visual Languages, September 5 - 9, 1995 Darmstadt, Germany*, 1995.

[KL98] Anatoli Krassavine and Gerald Loeffler. Biosymphony beans : Software components and data models for bioinformatics. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[KL00] Oliver Kohlbacher and Hans-Peter Lenhof. Ball : rapid software prototyping in computational molecular biology. *Bioinformatics*, 16(9) :815–824, September 2000.

[KLL+97] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *In proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.*, May 1997.

[KLS92] L.F. Kolakowski, J.A. Leunissen, and J.E. Smith. Prosearch : fast searching of protein sequences with regular expression patterns related to protein structure and function. *Biotechniques*, 13(6) :919–921, December 1992.

[KM95] A. Kjær and K.H Madsen. Participatory analysis of flexibility. *CACM*, 38(5) :53–60, May 1995.

[KM98] A. Kaps and H. W. Mewes. Migrating to an object-oriented protein sequence database. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[KMK95] Yuichi Koike, Yasuyuki Maeda, and Yoshiyuki Koseki. Improving readability of iconic programs with multiple view object representation. In *In Proceedings of VL'95, 11th International IEEE Symposium on Visual Languages, September 5 - 9, 1995 Darmstadt, Germany*, 1995.

[Knu84] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2) :97–111, May 1984.

[Kyn95] Morten Kyng. Making representations work. *CACM*, 38(9) :46 – 55, September 1995.

[lab87] Labview : a demonstration. unpublished, 1987.

[LC92] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems. *CACM*, 35(3) :121–132, March 1992.

[LD89] Marcia C. Linn and John Dalbey. Cognitive consequence of programming instruction. In *Studying the Novice Programmer*, pages 57–81. Lawrence Elbaum Associates, 1989.

[Let95] Stanley Letovsky. Beyond the information maze. *J Comput Biol*, 2(4) :539–546, 1995.

[Let99a] Catherine Letondal. Atelier brainstorming éditeur d'alignement. Technical report, Institut Pasteur, Paris., 1999.

[Let99b] Catherine Letondal. Atelier prototypage éditeur d'alignement. Technical report, Institut Pasteur, Paris., 1999.

[Let99c] Catherine Letondal. Résultats de l'enquête sur l'utilisation de l'informatique à l'institut pasteur. Technical report, Institut Pasteur, Paris., 1999.

[Let00a] Catherine Letondal. Atelier du 29 mars 200 : prototypage autour de la programmation. Technical report, Institut Pasteur, Paris., 2000.

[Let00b] Catherine Letondal. A web interface generator for molecular biology programs in unix. *Bioinformatics*, 17(1) :73–82, 2000.

[Let01] Catherine Letondal. biok : Biology interactive object kit. In *BOSC2001, Bioinformatics Open Source Conference*, 2001. oral presentation.

[Lew90] C. Lewis. Nopumpg : Creating interactive graphics with spreadsheet machinery. In E. P. Glinert, editor, *Visual Programming Environments : Paradigms and Systems*, pages 526–546. IEEE Computer Society Press, Los Alamitos, 1990.

[LF95] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *ACM conference on Human Factors in Computing Systems (Summary, Demonstrations) (CHI '95)*, pages 480–486. ACM Press, 1995.

[Lie87] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *Object-Oriented Computing*, 1987. also in First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, Oregon, September 1986.

[Lie92] Henry Lieberman. Dominos and storyboards : Beyond icons on strings. In *IEEE Conference on Visual Languages, Seattle, September 1992.*, 1992.

[Lie93] Henry Lieberman. Tinker : A programming by demonstration system for beginning programmers. In *Watch What I Do. Programming by Demonstration*, pages 49–64. MIT Press, 1993.

[Lie00] Henry Lieberman, editor. *Your Wish is My Command : Giving Users the Power to Instruct their Software.* Morgan Kaufmann, 2000.

[Lit99] Jim Little. The will and the word, a philosophy of programming. unpublished, 1999.

[LM95] James A. Landay and Brad A. Myers. Just draw it ! programming by sketching storyboards. Technical report, Human-Computer Interaction Institute, CMU, 1995. Technical Report CMU-HCII-95-106.

[LN95] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help framework understanding. In *OOPSLA'95*, pages 342–357. ACM Press, 1995.

[LNW98] Henry Lieberman, Bonnie A. Nardi, and David Wright. Grammex : Defining grammars by exemple. In *ACM conference on Human Factors in Computing Systems (Summary, Demonstrations) (CHI '98), Los angeles, California, USA*, pages 11–12. ACM Press, April 1998.

[LSU88] Henry Lieberman, Lynn Andrea Stein, and David Ungar. Of types and prototypes, the treaty of orlando. In *Addendum to the Proceedings of OOPSLA'87*, pages 43–44. ACM Press, 1988.

[MA93] K.H. Madsen and P. Aiken. Some experiences with cooperative interactive storyboard prototyping. *CACM*, 36(6) :57–64, June 1993.

[Mac91a] Wendy E. Mackay. Triggers and barriers to customizing software. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 153–160. ACM Press, 1991.

[Mac91b] Wendy E. Mackay. *Users and Customizable Software : A Co-Adaptive Phenomenon.* PhD thesis, Massachusetts Instititute of Technology, 1991.

[Mac00] Wendy E. Mackay. *In Situ Design*, 2000. Master Class notes. Søderberg, Denmark.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proc. of the OOPSLA-87 : Conference on Object-Oriented Programming Systems*, pages 147–155, Languages and Applications, Orlando, FL, 1987.

[Mae88] P. Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland, 1988.

[Mas00] Tetsuya Masuishi. A reporting tool using "programming by example" for format designation. In *Your Wish is My Command : Giving Users the Power to Instruct their Software.* Morgan Kaufmann, 2000.

[Mau93] David Maulsby. The turvy experience : Simulating an instructible interface. In *Watch What I Do. Programming by Demonstration, Part I Systems.*, pages 238–269. MIT Press, 1993.

[May89] Richard E. Mayer. The psychology of how novices learn computer programming. In *Studying the Novice Programmer*, pages 129–159. Lawrence Elbaum Associates, 1989.

[MBS+00] Chris Mayor, Michael Brudno, Jody R. Schwartz, Alexander Poliakov, Edward M. Rubin, Kelly A. Frazer, Lior S. Pachter, and Inna Dubchak. Vista : visualizing global dna sequence alignments of arbitrary length. *Bioinformatics*, 16(11) :1046–1047, November 2000.

[MCLM90] Allan MacLean, Kathleen Carter, Lennart Lovstrand, and Thomas Moran. User-tailorable systems : Pressing the issues with buttons. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pages 175–182. ACM Press, 1990.

[MCM97] Francesmary Modugno, Albert T. Corbett, and Brad Myers. Graphical representation of programs in a demonstrational visual shell - an empirical evaluation. *ACM Transactions on Computer-Human Interaction*, 4(3) :276–308, September 1997.

[MG98] N. R. Mcewan and D. Gatherer. Adaptation of standard spreadsheet software for the analysis of dna sequences. *Biotechniques*, 24(1) :131–138, January 1998.

[MGD+90] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet : Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11) :71–85, November 1990.

[MM97] Richard G. McDaniel and Brad A. Myers. Gamut : demonstrating whole applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology October 14 - 17, 1997, Banff Canada*, pages 81 – 82. ACM Press, October 1997.

[MMM+97] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment : New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6) :347–365, June 1997.

[MMM99] Brad A. Myers, Rich McDaniel, and Rob Miller. The amulet prototype-instance framework. In Mohamed Fayad and Ralph E. Johnson., editors, *Domain-Specific Application Frameworks*. John Wiley and Sons, 1999.

[MMW00] Brad A. Myers, Richard McDaniel, and David Wolber. Intelligence in demonstrational interfaces. *CACM*, 43(3) :82–89, March 2000.

[MNG+93] Andrew Monk, Bonnie Nardi, Nigel Gilbert, Marilyn Mantei, and John McCarthy. Mixing oil and water ? : Ethnography versus experimental psychology in the study of computer-mediated communication. In *Conference proceedings on Human factors in computing systems April 24 - 29, 1993, Amsterdam The Netherlands*, pages 3 – 6. ACM Press, April 1993.

[MOO94] Susan E. McDaniel, Gary M. Olson, and Judith S. Olson. Methods in search of methodology–combining hci and object orientation. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '94 Boston United States)*, pages 145 – 151. ACM Press, April 1994.

[Mør94] Anders Mørch. Designing for radical tailorability : Coupling artifact and rationale. *Knowledge-Based Systems*, 7(4) :253–264, December 1994. in thesis.

[Mør95] Anders Mørch. Application units : Basic building blocks of tailorable applications. In *Proceedings 5th Int'l East-West conf. on HCI, Moscow, July 1995. Lecture Notes in Computer Science 1015. Springer Verlag, 68-87.*, pages 68–87. Springer Verlag, 1995. in thesis.

[Mør97a] A. Mørch. Three levels of end-user tailoring : Customization, integration, and extension. In M. Kyng and L. Mathiassen, editors, *Computers and Design in Context.*, pages 51–76. The MIT Press, Cambridge, MA, 1997. in thesis + In Computers and Design in Context.

[Mør97b] A.I. Mørch. *Method and Tools for Tailoring of Object-oriented Applications : An Evolving Artifacts Approach.* PhD thesis, Department of Informatics, University of Oslo, April 1997.

[Mør97c] Anders Mørch. Evolving a generic application into a domain-oriented design environment. *Scandinavian Journal of Information Systems*, 8(2) :63–89, 1997. in thesis.

[Mør98] Anders Mørch. Aspect-oriented tailoring of object-oriented applications. In *Proceedings of the 21st Information System Research Seminar in Scandinavia (IRIS 21). Department of Computer Science, Aalborg University,*, pages 641–655, 1998.

[MRDV99] C. Medigue, F. Rechenmann, A. Danchin, and A. Viari. Imagene : an integrated computer environment for sequence annotation and analysis. *Bioinformatics*, 15(1) :2–15, January 1999.

[MRJ00] W.E. Mackay, A. Ratzer, and P. Janecek. Video artifacts for design : Bridging the gap between abstraction and detail. In *Proceedings of ACM DIS 2000, Conference on Designing Interactive Systems. Brooklyn, New York.* ACM Press, 2000.

[MS01] L Marsan and MF Sagot. Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory site consensus identification. *J. of Comput. Biol.*, 7 :345–360, 2001.

[MT86] Kazuo Matsumara and Suichi Tayama. Visual man-machine interface for program design and production. In *IEEE Workshop on Visual Languages*, pages 71–80, 1986.

[Mul91] Michael J. Muller. Pictive - an exploration in participatory design. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 225 – 231. ACM Press, 1991.

[Mun98] Chris Mungall. The ark genome databases : Migrating to a multilayer corba-based architecture. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[MWK89] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse : Specifying graphical procedures by example. In *Proceedings of SIGGRAPH '89, Vol. 23, No. 3, ACM, Boston, August 1989*, volume 29, pages 127 – 136, August 1989.

[MWW93] Michael J. Muller, Daniel M. Wildman, and Ellen A. White. Taxonomy of pd practices : A brief practitioner's guide. *CACM*, 36(6) :29–37, June 1993.

[Mye83] B. A. Myers. Incense : A system for displaying data structures. *ACM Computer Graphics*, 17(3) :115–125, July 1983.

[Mye90] Brad Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1) :97–123, March 1990. BLOX image below.

[Mye91] Brad A. Myers. Text formatting by demonstration. In *CHI'91*, pages 251–256. ACM Press, 1991.

[Mye92] Brad A. Myers, editor. *Languages for Developing User Interfaces.* Jones and Bartlett, 1992. 457 pages ; based on CHI'91 workshop on language for programming user interfaces and language for end-users.

[Mye00] Brad A. Myers. Usability issues in programming languages. Technical report, School of Computer Science, Carnegie Mellon University, 2000. Part of the Natural Programming Project.

[Nar95] Bonnie A. Nardi. *A small matter of programming : perspectives on end user computing.* MIT Press, 1995. 162 pages.

[Nar97a] Bonnie A. Nardi, editor. *Context and Consciousness : Activity Theory and Human-Computer Interaction.* MIT Press, 1997. 400 pages.

[Nar97b] Bonnie A. Nardi. The use of ethnographic methods in design and evaluation. In *Handbook of Human Computer Interaction 2ed complety revised.*, pages 361–366. North-Holland, 1997.

[NCM01] Zemin Ning, Anthony J. Cox, and James C. Mullikin. Ssaha : A fast search method for large dna databases. *Genome Research*, 2001. submitted.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[NDdM+94] Oscar Nierstrasz, Laurent Dami, Vicki de Mey, Marc Stadelmann, Dennis Tsichritzis, and Jan Vitek. Visual scripting - towards interactive construction of object-oriented applications. unpublished, 1994.

[Nea89] Lisa Rubin Neal. A system for example-based programming. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '89)*, pages 63–68. ACM Press, 1989.

[Nic96] Jeffrey Nickerson. *Visual Programming.* PhD thesis, New York University, 1996. UMI 9514409.

[Nie93] Jakob Nielsen. *Usability engineering.* Academic Press Inc., Boston., 1993.

[NJ94] Bonnie A. Nardi and Jeff A. Johnson. User preferences for task specific vs generic application software. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '94)*, pages 392–398. ACM Press, 1994.

[NM94] Jakob Nielsen and Robert L. Mack, editors. *Usability Inspection Methods.* John Wiley and Sons, New York, NY, 1994.

[Now97] Palle Nowack. Frameworks - representations and perspectives. In *Workshop on Language Support for Design Patterns and Frameworks, ECOOP'97*, 1997.

[NSS93] BE Nichols, VC Sheffield, and EM Stone. A user-friendly hypercard interface for human linkage analysis. *Bioinformatics*, 9(6) :757–759, December 1993.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition.* Prentice Hall, 1995. 361 pages.

[NZ93] Bonnie A. Nardi and Craig L. Zarmer. Beyond models and metaphors : Visual formalisms in user interface design. *Journal of Visual Languages and Computing*, 4(1) :5–33, 1993.

[NZ99a] Gustaf Neumann and Uwe Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC'99), Takamatsu, Japan, Dec 6-10, 1999.*, December 1999.

[NZ99b] Gustaf Neumann and Uwe Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems, San Diego, May 3-9 1999*, May 1999.

[NZ99c] Gustaf Neumann and Uwe Zdun. Implementing object-specific design patterns using per-object mixins. In *NOSA'99, Proceedings of the Second Nordic Workshop on Software Architecture, Ronneby, Sweden, Aug 12-13 1999*, August 1999.

[NZ00a] Gustaf Neumann and Uwe Zdun. Towards the usage of dynamic object aggregations as a form of composition. In *Proceedings of Symposium of Applied Computing (SAC'00), Como, Italy, Mar 19-21, 2000*, May 2000.

[NZ00b] Gustaf Neumann and Uwe Zdun. Xotcl, an object-oriented scripting language. In *Proceedings of 7th Usenix Tcl/Tk Conference (Tcl2k), Austin, Texas, Feb 14-18*, February 2000.

[OBCG99] Jon Oberlander, Paul Brna, Richard Cox, and Judith Good. The match-mismatch conjecture and learning to use data-flow visual programming languages :final report. Technical report, University of Edinburgh, University of Leeds, University of Sussex, 1999.

[OCL93] MO Ortells, VB Cockcroft, and GG Lunt. Cedit : a c interface and macro facility for protein sequence alignment editing in colour with microsoft word 5.0 for pcs. *CABIOS. Comput Appl Biosci*, 9(6) :741–744, December 1993.

[OFL94] Y. Orlarey, D. Fober, and S. Letz. Lambda calculus and music calculi. In *Proceedings of the International Computer Music Conference 1994, Computer Music Association, San Francisco.*, pages 243–250, 1994.

[OFL97] Yann Orlarey, Dominique Fober, and Stéphane Letz. Elody : a java+midishare based music composition environment. In *Proceedings of the ICMC 97, Thessaloniki*, 1997.

[O'M98] Andrew O'Malia. Comparative genome analysis using corba and components. In *Oib'98, Objects in Bioinformatics '98*, 1998.

[Ost95] J.M Ostell. Integrated access to heterogeneous biomedical data from ncbi. *IEEE Eng. Med. Biol.*, 14 :730–736, 1995.

[Ous98] John K. Ousterhout. Scripting : Higher level programming for the 21st century. *IEEE Computer*, 31(3) :23–30, March 1998.

[PAF84] P.L. Pirolli, J.R. Anderson, and R. Farrell. Learning to program recursion. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society, Hillsdale, NJ.*, pages 277–280, 1984.

[PB93] Rajeev Pandey and Margaret Burnett. Is it easier to write matrix manipulation programs visually or textually ? an empirical study. In *IEE Symposium on Visual Languages*, pages 344–351. ACM Press, 1993.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. Principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4 :211–266, 1993.

[PD00] Hans-Peter Pohle and Bernd Drescher. A flexible and easy to use molecular biology workbench efficiently developed in tcl/tk. *Software : Practice and Experience*, 30(12) :1433–1445, October 2000.

[Pen87] N. Pennington. Stimulus structures and mental representations in expert comprehension of programs. *Cognitive Psychology*, 19 :295–341, 1987.

[PHB98] M. R. Pocock, T. Hubbard, and E. Birney. Spem : a parser for embl style flat file database entries. *Bioinformatics*, 14(9) :823–824, 1998.

[Pir84] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance. An Inquiry into Values.* Bantam Book, 1984.

[PL92] D. Ploger and Ed. Lay. The structure of programs and molecules. *Journal of Educational Computing Research*, 8(3) :347–364, 1992.

[PM99] Ken Perlin and Jon Meyer. Nested user interface components. In *Proceedings of the 12th annual ACM symposium on User interface software and technology November 7 - 10, 1999, Asheville United States*, pages 11 – 18. ACM Press, November 1999.

[Pot93a] Richard Potter. Just-in-time programming. In *Watch What I Do. Programming by Demonstration, Part III Perspectives.*, pages 513–526. MIT Press, 1993.

[Pot93b] Richard Potter. Triggers : Guiding automation with pixels to achieve data access. In *Watch What I Do. Programming by Demonstration, Part III Perspectives.*, pages 361–380. MIT Press, 1993.

[Pre94] W. Pree. *Design patterns for object-oriented software development.* ACM Press, 1994. 268 pages.

[PRM00] J.F. Pane, C.A. Ratanamahatana, and Brad Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 2000. to appear.

[PRT00] J. D. Parsons and P. Rodriguez-Tomé. Jesam : Corba software components to create and publish est alignments and clusters. *Bioinformatics*, 16(4) :313–325, April 2000.

[PtH85] G. Pfaff and P. J. W. ten Hagen, editors. *Seeheim Workshop on User Interface Management Systems.* Springer-Verlag, Berlin, 1985.

[PVB98] S Pook, G Vaysseix, and E Barillot. Zomit : biological data visualization and browsing. *Bioinformatics*, 14(9) :807–814, October 1998.

[PVM94] Jorg Poswig, Guido Vrankar, and Claudio Moraga. Visavis : a higher-order functional visual programming language. *Journal of Visual Languages and Computing.*, 5 :83–111, 1994.

[Rao91] Ramana Rao. Implementational reflection in silica. In *ECOOP '91 (LNCS 512), p. 251 ff.*, pages 251–267. ACM Press, July 1991.

[RASW90] J. Rasure, D. Argiro, T. Sauer, and C. S. Williams. A visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology*, 2 :183–199, 1990.

[RAZ99] Reza RAZAVI. Building an end-user-oriented application framework by meta-programming. a case study. In *Position Paper for OOPSLA'99 Metadata and Dynamic Object-Model Pattern Mining Workshop*, 1999.

[RCB90] Mary Beth Rosson, John M. Carrol, and Rachel K. E. Bellamy. Smalltalk scaffolding : a case study of minimalist instruction. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pages 423 – 430. ACM Press, 1990.

[RCB+98] Cyndi Rader, Gina Cherry, Cathy Brand, Alexander Repenning, and Clayton Lewis. Designing mixed textual and iconic programming languages for novice users. In *Proceedings IEEE Symposium on Visual Languages*, 1998.

[Rep93a] Alexander Repenning. *Agentsheets : A Tool for Building Domain-Oriented Dynamic, Visual Environments.* PhD thesis, University of Colorado at Boulder, 1993.

[Rep93b] Alexander Repenning. Agentsheets : Applying grid-based spatial reasoning to human-computer interaction. In *in Proceedings of 1993 IEEE Workshop on Visual Languages, Bergen, Norway, IEEE Computer Society Press*, 1993.

[RF97] AJ Robinson and TP. Flores. Novel techniques for visualising biological information. In *Proc Int Conf Intell Syst Mol Biol 1997;5 :241-9*, pages 241–249, 1997.

[RGG00] Chenna Ramu, Christine Gemünd, and Toby J. Gibson. Object-oriented parsing of biological databases with python. *Bioinformatics*, 16(7) :628–638, July 2000.

[RHC91] George G. Robertson, D. Austin Henderson, and Stuart K. Card. Buttons as first class objects on an x desktop interactive components. In *Proceedings of the ACM Symposium on User Interface Software and Technology 1991 (UIST)*, pages 35–44. ACM Press, 1991.

[Rid00] François-René Rideau. Métaprogrammation et libre disponibilité des sources. deux défis informatiques d'aujourd'hui. Technical report, CNET DTL/ASR (France Telecom), 2000.

[Rob95] S. P. Robertson. Generating object-oriented design representations via scenario queries. In John M. Carroll, editor, *Scenario-Based Design : Envisioning Work and Technology in System Development.* Wiley, 1995.

[RSUZ91] Jack Rawls, Marion L. Spell, Thomas R. Unnasch, and Peter. A. Zimmerman. Transformation of dna sequence data into geometric symbols. *Biotechniques*, 11(1) :50–52, July 1991.

[RW93] P De Rijk and R De Wachter. Dcse, an interactive tool for sequence alignment and secondary structure research. *CABIOS. Comput Appl Biosci*, 9(6) :735–740, December 1993.

[SAL91] GD Schuler, SF Altschul, and DJ Lipman. A workbench for multiple alignment construction and analysis. *Proteins*, 9(3) :180–190, 1991.

[SBMP97] Michèle Soria, Anne Brygoo, Michelle Morcrette, and Odile Paliès. *Initiation à la programmation par Word et Excel.* Vuibert, 1997. 516 pages.

[Sch90] Donald Schon. *Educating the Reflective Practitioner : Toward a New Design for Teaching and Learning in the Professions.* Jossey-Bass, 1990.

[SCT00] David Canfield Smith, Allen Cypher, and Larry Tesler. Novice programming comes of age. *CACM*, 43(3) :75–81, March 2000.

[SCTMTM98] L.D. Stein, S. Cartinhour, D. Thierry-Mieg, and J. Thierry-Mieg. Jade : an approach for interconnecting bioinformatics databases. *Gene 209(1-2) :GC39-GC43*, 16 :39–43, March 1998.

[Sea95] David B. Searls. biotk : componentry for genome informatics graphical user interface. *Gene*, 163(2) :GC1–GC16, October 1995.

[seq96] Seqvu : Dna sequence matching/visualisation software. shareware, unpublished, 1996.

[SFT+01] A. Siepel, A. Farmer, A. Tolopko, M. Zhuang, P. Mendes, W. Beavis, and B. Sobral. Isys : a decentralized, component-based approach to the integration of heterogeneous bioinformatics resources. *Bioinformatics*, 17(1) :83–94, January 2001.

[SGB+01] Robert Stevens, Carole Goble, Patricia Baker, , and Andy Brass. A classification of tasks in bioinformatics. *Bioinformatics*, 17(2) :180–188, February 2001.

[Sha93] Basavaraju Shankarappa. Transformation of sequence data into genometric symbols. *Biotechniques*, 14 :990–995, June 1993. BioComputing-Technical Report.

[Shn00] Ben Shneiderman. Foreword to your wish is my command. In *Your Wish is My Command : Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2000.

[Smi75] D. C Smith. *PYGMALION : A Creative Programming Environment*. PhD thesis, Stanford University, 1975.

[Smi86] R. B. Smith. The alternate reality kit : an animated environment for creating interactive simulations. In *Proceedings 1986 IEEE Computer Society workshop on visual languages. Dallas.*, pages 99–106, June 1986.

[Smi87] D. N. Smith. Intercons : Interface construction set. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY USA. RC 13108, 1987.

[SMU95] Randall B. Smith, John Maloney, and David Ungar. The self-4.0 user interface : Manifesting a system-wide vision of concretness, uniformity, and flexibility. In *in Proc. OOPSLA '95*, pages 47–60, 1995.

[SN93] Douglas Schuler and Aki Namioka. *Participatory Design : Principles and Practices*. Hillsdale, NJ : LEA, 1993. 312 pages.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer, 1999.

[SOM87] William C. Sasso, Judith Reitman Olson, and Alan G. Merten. The practice of office analysis : Objectives, obstacles, and opportunities. *IEEE Bulletin on Office Knowledge Engineering*, May 1987.

[SOW+94] S.W. Smith, R. Overbeek, C.R. Woese, W. Gilbert, and P.M. Gillevet. The genetic data environment an expandable gui for multiple sequence analysis. *CABIOS*, 10(6) :671–675, December 1994.

[SP96] R. P. Stowe and D. L. Pierson. Spreadsheet macro for setting up pcr assay tubes. *Biotechniques*, 20(6) :1088–1089, June 1996.

[SP97] T Sicheritz-Ponten. Biowish : a molecular biology command extension to tcl/tk. *CABIOS*, 13(6) :621–622, December 1997.

[SSC+94] M. Scharf, R. Schneider, G. Casari, P. Bork, A. Valencia, C. Ouzounis, and C. Sander. Genequiz : a workbench for sequence analysis. *Proc Int Conf Intell Syst Mol Biol*, 2 :348–353, 1994.

[ST93] L Suchman and R Trigg. Artificial intelligence as craftwork. In Seth Chaiklin and Jean Lave, editors, *Understanding Practice*. Cambridge University Press, 1993.

[STM98] L.D. Stein and J. Thierry-Mieg. Scriptable access to the caenorhabditis elegans genome sequence and other acedb databases. *Genome Research*, 8(12) :1308–1315, December 1998.

[SU95] Randall B. Smith and David Ungar. Programming as an experience : The inspiration for self. In *in Proc. ECOOP '95*, 1995.

[Sub98] S Subramaniam. The biology workbench : A seamless database and analysis environment for the biologist. *"Bioinformatics", Proteins, 32 :, 1998.*, 32(1) :1–2, July 1998. editorial.

[SUC92] Randall B. Smith, David Ungar, and Bay-Wei Chang. The use-mention perspective on programming for the interface. In *Languages for Developping User Interfaces*. Jones and Bartlett, 1992.

[SVS97] M.-F. Sagot, A. Viari, and H. Soldano. Multiple sequence comparison — A peptide matching approach. *Theoretical Computer Science*, 180(1–2) :115–137, June 1997.

[Tan90] S. L. Tanimoto. Viva : a visual language for image processing. *Journal of Visual Languages and Computing*, 1 :127–139, June 1990.

[TCJ92] H Thimbleby, A Cockburn, and S Jones. Hypercard : An object-oriented disappointment. In P Gray and R Took, editors, *Building Interactive Systems : Architectures and Tools*, pages 35–55. Springer-Verlag, 1992.

[TJC91] Allan Tuchman, David Jablonowski, and George Cybenko. Run-time visualization of program data. In *Proceedings of Visualization '91, San Diego, CA*, October 1991.

[TMH87] R. Trigg, T. Moran, and F. Halasz. Adaptability and tailorability in notecards. In *Bullinger and Shackel (eds.), INTERACT'87, North Holland.*, 1987.

[Tol97] LI Toldo. Jambw 1.1 : Java-based molecular biologists' workbench. *CABIOS. Comput Appl Biosci*, 13(4) :475–476, August 1997.

[Tra94] Michael Travers. Recursive interfaces for reactive objects. In *ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '94)*, pages 379–385. ACM Press, 1994.

[Tri92] Randall H. Trigg. Participatory design meets the mop : Accountability in the design of tailorable computer systems. In *Previously appeared in : Proceedings of the 15th IRIS (Information systems Research seminar In Scandinavia) Gro Bjerknes, Tone Bratteteig, Karlheinz Kautz (eds.), August 1992, Larkollen, Norway.*, 1992. other subtitle : Informing the design of tailorable computer systems.

[ULF97] David Ungar, Henry Lieberman, and Christopher Fry. Debugging and the experience of immediacy. *CACM*, 40(4) :38–43, April 1997.

[Usd92] K Usdin. Hypercard-based data management tools for molecular biologists. *Comput Appl Biosci*, 8(2) :107–111, April 1992.

[VHKW96] W Vahrson, K Hermann, J Kleffe, and B Wittig. Object-oriented sequence analysis : Scl–a c++ class library. *Bioinformatics*, 12(2) :119–127, April 1996.

[vR99] Guido van Rossum. Computer programming for everybody. Technical report, CNRI : Corporation for National Research Initiatives, 1999.

[WB97] Rebecca A. Walpole and Margaret M. Burnett. Supporting reuse of evolving visual code. In *IEE Symposium on Visual Languages.* ACM Press, 1997.

[Weg89] Peter Wegner. Learning the language. *BYTE Magazine*, pages 245–253, March 1989.

[Weg97a] Peter Wegner. Interactive fundations of computing. Technical report, Brown University, April 1997.

[Weg97b] Peter Wegner. Why interaction is more powerful than algorithms. *CACM*, 40(5) :80–91, May 1997.

[Weg98] Peter Wegner. A research agenda for interactive computing. Technical report, Brown University, 1998. unpublished.

[Weg99] Peter Wegner. Draft of ecoop'99 banquet speech. In *ECOOP'99, Lisbon, Portugal*, 1999. Draft of Banquet Speech.

[Wel99] Brent B. Welch. *Practical Programming in Tcl and Tk, 3rd Edition.* Prentice Hall, 1999.

[Wer92] A Werth. Tourmaline : Formatting document headings by example. Master's thesis, Information Networking Institute, Carnegie Mellon University, 1992.

[WG99a] Peter Wegner and Dina Goldin. Interaction, computability, and church's thesis. unpublished, 1999.

[WG99b] Peter Wegner and Dina Goldin. *Models of Interaction*, 1999. ECOOP '99 Course Notes.

[Win95] Terry Winograd. From programming environments to environments for designing. *CACM*, 38(6) :65 – 74, June 1995.

[Wit93] Ian H. Witten. A predictive calculator. In *Watch What I Do. Programming by Demonstration, Part I Systems.*, pages 67–76. MIT Press, 1993.

[WL94] David Wetherall and Christopher J. Lindblad. Extending tcl for dynamic object-oriented programming. Technical report, Telemedia, MIT lab. of CS, 1994. unpublished.

[Wol96] David Wolber. Pavlov : Programming by stimulus-response demonstration. In *Conference proceedings on Human factors in computing systems, CHI'96, Vancouver Canada*, pages 252 – 259, April 1996.

[Wom00] DD Womble. Gcg : The wisconsin package of sequence analysis programs. *Methods Mol Biol*, 132 :3–22, 2000.

[WZ97] Susan Wiedenbeck and Patti L. Zila. Hands-on practice in learning to use software : A comparison of exercice, exploration, and combined formats. *ACM Transactions on Computer-Human Interaction*, 4(2) :169–196, June 1997.

[ZCM98] P. Zellweger, B-W. Chang, and J. Mackinlay. Fluid links for informed and incremental link transitions. In *Proceedings of Hypertext'98, Pittsburgh, PA, June 20-24, 1998.*, pages 50–57, June 1998.

[Zel94] J. Zelinka. Zldb, graphical user interfaces in bioinformatics. In *Workshop, National Institute for Medical Research, The Ridgeway, Mill Hill, London, NW7 1AA. UK, 1st July 1994, organized by Tom Flores*, 1994.

Table des matières

Introduction générale **3**

1 Introduction **5**

1.1 Ouverture et interactivité des systèmes informatiques. 5

1.2 Objectifs. 7

1.2.1 Programmation par l'utilisateur final. 7

1.2.1.1 La programmation sans programmer. 8

1.2.1.2 Qui sont les utilisateurs finaux ? 10

1.2.1.3 La programmation, c'est quoi ? 11

1.2.1.4 Exemples de programmation par l'utilisateur. 11

1.2.1.5 Les situations de programmation. 15

1.2.1.6 Programmation et interaction, programmer et utiliser. 16

1.2.2 Répartir le travail 18

1.2.2.1 Modèles de calcul pour la répartition du travail. 19

1.2.2.2 Paramétrabilité. 21

1.2.3 Sur quels aspects doit-on mettre l'accent ? 22

1.2.3.1 La programmation comme une forme évoluée d'interaction. 22

1.2.3.2 Computer literacy. 23

1.2.3.3 La personnalisation de logiciels. 24

1.2.3.4 Construction de systèmes interactifs. 28

1.2.3.5 Conclusion. 30

1.3 Approches. 30

1.3.1 L'importance de la notation. 30

1.3.1.1 Formes textuelles, formes visuelles. 30

1.3.1.2 Métaphores spatiales. 31

1.3.1.3 Notation active. 34

1.3.1.4 Différence description/exécution. 34

1.3.1.5 Réalisme naïf. 34

1.3.1.6 3.1.6 Langages hybrides. 35

1.3.1.7 Explicite et implicite : trouver le bon formalisme ? 35

1.3.1.8 Formalismes visuels 36

1.3.1.9 Conclusion sur la notation 36

1.3.2 Niveaux de langage : structure et navigation. 36
1.3.2.1 Orthogonalité. 36
1.3.2.2 Programmation visuelle, construction d'interfaces et programmation dans l'interface. 37
1.3.2.3 Prise en compte de l'orthogonalité 37
1.3.3 Méthode : générale ou spécifique, abstraite ou concrète ? 40
1.3.3.1 Niveaux intermédiaires 40
1.3.3.2 Langages génériques ou spécifiques ? 40
1.3.3.3 Combiner les objectifs 41
1.3.3.4 Une solution particulière. 41
1.4 Conclusion. 42

2 Facteurs humains et programmation par l'utilisateur. 47
2.1 Introduction. 47
2.1.1 Problèmes et difficultés de la programmation. 47
2.1.1.1 Tâches 48
2.1.1.2 Apprentissage 49
2.1.1.3 Psychologie ou design ? 51
2.1.1.4 Orientation choisie 52
2.2 Point de vue de l'utilisabilité. 53
2.2.1 Études d'utilisabilité en programmation. 53
2.2.1.1 Études au niveau des notations 53
2.2.1.2 Études au niveau des paradigmes 55
2.2.1.3 Études au niveau d'un langage 56
2.2.1.4 Études de langages et d'environnements de programmation visuelle . . 57
2.2.1.5 Études de langages et d'environnements de programmation pour l'utilisateur 57
2.2.2 Études réalisées dans le cadre de la thèse. 59
2.2.2.1 Enquête sur l'utilisation de l'informatique scientifique 59
2.2.2.2 Tests pour un générateur d'interfaces Web, Pise 62
2.2.2.3 Tests pour un annuaire interrogeable, le BioNetbook 64
2.2.2.4 Conclusion sur les études réalisées 64
2.3 Programmation par l'utilisateur et développement participatif. 65
2.3.1 Utilité et limites des études objectives. 65
2.3.2 Les méthodes participatives. 66
2.3.2.1 Démarche générale. 66
2.3.2.2 Techniques. 68
2.3.2.3 Conception participative et programmation par l'utilisateur. 71
2.3.2.4 Participation à la conception et programmation : une approche intégrée (la programmation comme un moyen). 72
2.3.2.5 Contradictions potentielles (la programmation comme un but). 73
2.3.2.6 Un cadre conceptuel et méthodologique pour la programmation par l'utilisateur ? 73

2.3.3 Études réalisées. 74
2.3.3.1 Contexte, immersion : support technique, cours. 74
2.3.3.2 Analyse : études de terrains, entretiens, scénarios. 74
2.3.3.3 Conception : ateliers participatifs. 75
2.3.3.4 Remarques générales sur les ateliers. 101
2.4 Conclusion. 103

3 Techniques pour la programmabilité. 105
3.1 Introduction 105
3.2 Flexibilité et systèmes modifiables 105
3.2.1 Flexibilité logicielle 105
3.2.1.1 Flexibilité relative au type de langage 106
3.2.1.2 Flexibilité interne 107
3.2.1.3 Flexibilité externe 108
3.2.2 Flexibilité pour l'utilisateur 109
3.2.2.1 Choix d'une architecture. 109
3.2.2.2 Flexibilité et personnalisation 110
3.2.2.3 Flexibilité et programmation par l'utilisateur 111
3.2.2.4 Conclusion sur la flexibilité 112
3.2.3 Conception réflexive 112
3.2.3.1 Systèmes réflexifs 112
3.2.3.2 Protocoles méta-objets 114
3.2.3.3 Persistance 119
3.2.3.4 Conclusion sur les protocoles réflexifs 121
3.3 Quels niveaux d'interface pour un système programmable par l'utilisateur ? 121
3.3.1 Articulation des niveaux de langages 123
3.3.2 Approche modèle en couches 124
3.3.2.1 Transition rationnelle entre l'interface et l'implémentation 124
3.3.2.2 Modèle de calcul et interfaces : combinaison de la programmation par objets et du modèle dataflow 125
3.3.2.3 Granularité 128
3.3.2.4 Pas trop de couches... 128
3.3.3 Structures et interfaces 129
3.3.3.1 Modèles d'architectures (Seeheim, ARCH) 129
3.3.3.2 Réflexion et interfaces 130
3.3.3.3 Relations entre la représentation statique et la représentation dynamique 132
3.3.4 Langage et interfaces 132
3.3.4.1 Environnements de programmation et programmation dans l'interface . 132
3.3.4.2 Programmation directe 134
3.3.4.3 Éditeurs graphiques spécifiques, formalismes visuels. 136
3.3.5 Conclusion sur les niveaux d'interface. 136
3.4 Conclusion 137

4 Flexibilité en bio-informatique. 139
4.1 Introduction . . . 139
4.1.1 Pourquoi les biologistes ont-ils besoin de flexibilité ? . . . 139
4.2 Flexibilité logicielle . . . 140
4.2.1 Composants . . . 141
4.2.1.1 Motivations . . . 141
4.2.1.2 Qu'est-ce qu'un composant ? . . . 141
4.2.1.3 Interfaces entre composants . . . 142
4.2.1.4 Représentation des données . . . 143
4.2.1.5 Liaisons entre composants . . . 145
4.2.1.6 Sémantique des composants . . . 148
4.2.1.7 Exemple : composants graphiques. . . . 149
4.2.1.8 Conclusion sur les composants. . . . 150
4.2.2 Intégration . . . 151
4.2.3 Conclusion . . . 152
4.3 Flexibilité pour l'utilisateur . . . 152
4.3.1 Construction : flexibilité du système . . . 154
4.3.1.1 Personnalisation . . . 154
4.3.2 Intégration : flexibilité des interfaces . . . 155
4.3.2.1 Dataflow . . . 156
4.3.2.2 Composants et interaction . . . 157
4.3.2.3 Systèmes décomposables . . . 158
4.3.3 Interaction : flexibilité de l'algorithme . . . 159
4.3.3.1 Importance de l'informel . . . 160
4.3.3.2 Visualisation interactive et contrôle du logiciel . . . 162
4.3.4 Programmation par l'utilisateur . . . 163
4.3.4.1 Programmation inutile . . . 163
4.3.4.2 Tableurs . . . 163
4.3.4.3 Traitement de texte . . . 165
4.3.4.4 Programmation dans l'interface . . . 166
4.3.4.5 Langages spécialisés . . . 166
4.3.4.6 Langages visuels . . . 168
4.3.4.7 Apprentissage et exploration . . . 168
4.4 Conclusion . . . 169

5 Description des prototypes 171
5.1 Introduction . . . 171
5.2 Un générateur d'interfaces, Pise . . . 171
5.3 Premier prototype . . . 174
5.4 Prototype biok . . . 176
5.4.1 Présentation générale . . . 176
5.4.1.1 Le langage XOtcl . . . 176

5.4.1.2 Les objets graphiques 177
5.4.1.3 L'éditeur d'alignement 179
5.4.1.4 Conception 192
5.4.2 Architecture 195
5.4.2.1 Vues et données 195
5.4.2.2 Création des objets 195
5.4.2.3 Dépendances 197
5.4.2.4 Nommage 198
5.4.3 Réflexivité 198
5.4.3.1 Détection des dépendances 200
5.4.3.2 Interface utilisateur 201
5.4.3.3 Programmation 201
5.4.3.4 Intégrer le niveau interface utilisateur dans la définition des objets et classes 203
5.4.3.5 Autres utilisations 204
5.4.4 Flexibilité 204
5.4.5 Programmation par l'utilisateur 206
5.4.5.1 Deux niveaux de programmation 206
5.4.5.2 Navigation 206
5.4.5.3 Déboguage 208
5.4.5.4 Environnement général 209
5.4.5.5 Persistance 209
5.4.6 Stages d'étudiants 211
5.4.6.1 Suivi d'une biologiste : programmation d'un algorithme biologique intégré dans l'éditeur d'alignement 211
5.4.6.2 Flexibilité algorithmique 215
5.4.7 Environnements similaires 218
5.5 Conclusion 219

Conclusion générale **223**

Table des figures

1.1 Classification des systèmes de visualisation scientifique (emprunté à [BHP+94]). 6
1.2 Situations de programmation. 16
1.3 Axes programmation-interaction. 17
1.4 Paramétrabilité. 21
1.5 Programmation par l'utilisateur et personnalisation. 27
1.6 Généralisation/spécialisation. 28
1.7 VEX [CHZ95] 32
1.8 LiveWorld [Tra94] 32
1.9 Forms/3 [BAD+00] 33
1.10 MAP [FJG95] 33
1.11 Distance code interface utilisateur. 37
1.12 Solutions par le plein (PITUI) ou par le vide (bootstrap). 44
1.13 Programmation dans l'interface. 45
1.14 Carte conceptuelle. 45

2.1 Tâche de compréhension, ou construction de programmes ? 49
2.2 Adaptation par l'utilisateur 51
2.3 Adaptation par le système 51
2.4 Personnes, outils, problèmes 52
2.5 Choix 53
2.6 Les problèmes 60
2.7 Comment trouver de l'aide 60
2.8 Comment vous débrouillez-vous avec 61
2.9 Formation. 61
2.10 Analyse multi-dimensionnelle. 63
2.11 Flexibilité et programmation par l'utilisateur 72
2.12 Calendrier 76
2.13 Maquette du 1er atelier 77
2.14 Résultats du brainstorming 82
2.15 Le groupe "annotations" 86
2.16 Le groupe "recherche de motif" 86
2.17 Recherche avec erreurs 87
2.18 Dégradés de couleurs pour hiérarchiser les résultats 87

2.19 Commentaires 88
2.20 Annotations et recherche de motifs 89
2.21 Requêtes dynamiques 90
2.22 Liens fluides 91
2.23 Définition d'un tag avec une formule 100

3.1 Paramétrisation. 108
3.2 Programmabilité. 110
3.3 Causalité. 112
3.4 Transposition de l'idée de MOP. 116
3.5 Interface duale. 116
3.6 Persistance. 120
3.7 Accès direct au code. 122
3.8 Niveaux et interfaces. 123
3.9 Literate programming. 134
3.10 LiveWorld [Tra94] 135
3.11 Nested User Interface Components [PM99] 135
3.12 Convergence. 137

4.1 Flexibilité et composants 148
4.2 Classification des tâches 150
4.3 Dimensions de la programmabilité 153
4.4 Piper 156
4.5 BioBeans 158
4.6 BioWidgets (d'après [FCB+99]) 159
4.7 Construction d'applications (d'après [FL88]) 160
4.8 TexShade. 167

5.1 Champs généré dans le formulaire 172
5.2 Connexions de programmes 173
5.3 Premier prototype 175
5.4 Object graphique. 178
5.5 Shell de l'objet. 178
5.6 Formule. 178
5.7 Éditeur d'alignement. 179
5.8 Pourcentage d'identité. 180
5.9 Segments transmembranaires (Toppred). 180
5.10 Editeur de tags. 181
5.11 Occurrences de TATA box. 182
5.12 Sélection non rectangulaire. 183
5.13 Nom pour une sélection. 183
5.14 Annotation. 183

5.15 Recherche dans les annotations. 184
5.16 Recherche de motifs avec erreurs. 184
5.17 Nombre d'erreurs. 184
5.18 Interface utilisateur pour les fonctions de visualisation. 185
5.19 Motifs Prosite. 186
5.20 Recherche sur un cadre de lecture. 186
5.21 Brin complémentaire. 187
5.22 Traduction. 187
5.23 Chargement depuis une banque. 188
5.24 PREDATOR. 188
5.25 Analyses juxtaposées. 188
5.26 Ré-alignement de la zone sélectionnée. 189
5.27 Filtrage. 189
5.28 Extraction. 190
5.29 Formule pour calculer le consensus. 190
5.30 Conversion de format. 192
5.31 Classes de l'éditeur d'alignement. 193
5.32 Classes des tags. 194
5.33 Configuration de tkTable. 195
5.34 Vues et données. 196
5.35 Courbe d'hydrophobicité. 197
5.36 Mises à jour automatiques. 198
5.37 Correspondance des espaces de nom. 199
5.38 Détection et gestion des dépendances. 201
5.39 Editeur de méthodes. 202
5.40 Couches dataflow et objet. 207
5.41 Déboguage. 208
5.42 Environnement. 210
5.43 Fenêtre glissante pour le calcul du profil d'hydrophobicité 212
5.44 Une protéine transmembranaire. 215
5.45 Segments trop proches. 216
5.46 Segment indiqué par l'utilisateur. 217
5.47 Sélection des "User data" et visualisation zoomée des segments. 217
5.48 Objets imbriqués. 220
5.49 Menu dans le code. 221

Liste des tableaux

1.1 Définitions de la programmation 12
1.2 Composants du logiciel épargnés à l'utilisateur 20
1.3 Explicite et implicite 35
1.4 Modèle et copie 39

2.1 Catégorie professionnelle 59
2.2 Type de terminal utilisé 59
2.3 Logiciels utilisés 60
2.4 Psychologie cognitive et ethno-méthodologie 67

3.1 Flexibilité 113

Glossaire

Alignement Processus par lequel deux séquences sont comparées afin d'obtenir le plus de correspondances (identités ou substitutions conservatives) possibles entre les lettres qui les composent.

ALU Une famille de séquences répétitives qu'on trouve dispersées dans le génome humain. Le nom vient du fait que ces séquences sont reconnues par l'endonuclease de restriction Alu.

Amorce (primer) Courte séquence d'ADN qui, hybridée avec une molécule simple brin d'acide nucléique, permet à une polymérase d'initier la synthèse du brin complémentaire.

Contig Ensemble de fragments d'ADN contigus et ordonnancés, utilisé pour reconstituer la carte physique d'un chromosome ou du génome.

Dataflow Dans une architecture ou un langage dataflow les calculs sont déclenchés dès que des données sont disponibles ou modifiées.

EST Expressed Sequence Tag. Marqueurs de séquences exprimées. Séquences venant de clones, essentiellement de banques d'ADN complémentaire (issu d'ARN messager, produit lors de la traduction de l'ADN pour produire la protéine).

Facteur de transcription Protéine qui régule la transcription d'un gène en se fixant sur son promoteur (courte séquence d'ADN nécessaire à l'initiation de la transcription). La transcription est synthèse d'une molécule d'ARN complémentaire à une séquence d'ADN, et constitue une des étapes de l'expression d'un gène.

Feuille de propriétés Une boîte de dialogue qui permet à l'utilisateur de modifier les caractéristiques d'un objet en éditant les valeurs de ses attributs (propriétés).

GUI Graphical User Interface.

IDE (Integrated Development Environment) : environnement de programmation comportant des outils intégrés (débogueur, éditeur, outils de trace, ...).

langage de prototypes Langage sans classes dans lequel les objets sont créés individuellement ou par clonage et où la délégation d'appel remplace l'héritage de méthodes.

langage de script Un langage de programmation simple et interprété conçu pour le développement rapide d'applications constituées d'enchaînements de commande.

langage à contraintes Formalisation d'un problème en un réseau de contraintes, structure déclarative qui permet d'exprimer des relations entre des objets ; le calcul procède par résolution de ces contraintes.

MVC Model View Controller. Une méthode pour partitionner la conception d'un logiciel interactif. Le modèle est la partie qui représente le fonctionnement interne, la vue celle qui gère l'affichage (ce que l'utilisateur voit du modèle), et le contrôle celle qui s'occupe des interactions de l'utilisateur (pour modifier le modèle ou la vue).

Persistance Propriété d'un langage ou d'un système qui permets aux objets créés et aux variables de survivre entre les différentes exécutions du programme.

Personnalisation (customization) Adaptation par l'utilisateur du système informatique à ses besoins spécifiques.

Programmation évènementielle Programmation où le calcul procède par réaction à des évènements, comme des interactions de l'utilisateur.

Séquence Représentation linéaire des acides aminés d'une protéine ou des acides nucléiques d'une molécule d'ADN. Du point de vue informatique, les séquences sont représentées comme des chaînes de caractères.

Résumé

Cette thèse a pour objet l'amélioration des possibilités de contrôle et d'adaptation des outils informatiques par des biologistes à travers une réflexion sur la flexibilité logicielle, la programmation par l'utilisateur et les démarches de conception participative. Le manque de flexibilité des outils disponibles limite souvent leur utilité. La solution la plus connue à ce problème est la programmation. Mais comment donner accès à cette discipline complexe à ceux des biologistes qui n'ont pas le temps d'apprendre ? Nous précisons d'abord la problématique programmation et interaction. Nous réfléchissons ensuite à la question de la flexibilité logicielle et complétons l'idée de programmation par l'utilisateur par celle de participation à la conception, deux manières de donner un contrôle sur le logiciel. Parallèlement à cette réflexion, nous avons mené des études de terrains et organisé des ateliers de conception avec des biologistes permettant à l'utilisateur de participer activement à la conception d'un logiciel. Nous avons observé que ce n'est pas tant l'écriture de code, que la construction de tout un logiciel qui pose problème. L'idée d'application programmable permet à l'utilisateur d'effectuer son travail sans avoir à programmer, tout en fournissant un accès guidé mais total au code. Pour les biologistes désirant apprendre la programmation, un tel environnement constitue un support d'apprentissage adapté comportant des exemples centrés sur leur domaine. Un prototype, biok, a été réalisé comportant des composants pour l'analyse de séquences comme un éditeur d'alignement ou un outil d'affichage de courbes. L'éditeur d'alignement fonctionne comme un tableur, et dispose d'un mécanisme programmable d'étiquettage graphique pour visualiser des propriétés biologiques. L'architecture de cet environnement repose sur la notion d'objet graphique, permettant la composition d'objets biologiques par des formules, l'accès structuré au code de l'application.

Abstract

This thesis aims to provide biologists with a better control over the software they use through a reflection on software flexibility and end-user programming, as well as a user-centred design approach. A lack of flexibility among available tools often limits their usefulness. Programming is the most general solution, but how do we provide access to this complex technique to the biologists who do not want to spend too much time learning how to build software ? We first address this issue by a discussion on programmation, interaction and software flexibility. We then extend the idea of user-programming to the idea of participatory design, as two complementary means for giving the users control over the software they use. We also conducted field studies and organized design workshops. Participatory design enables the user to actively participate in the design of software. One of the most important facts to emerge was that writing code may not be as difficult for non-professional programmers as building a whole software, designing an architecture or a graphical user interface. This leads us to the ideas of programmable applications which enable the user to accomplish his task while at the same time providing him with a guided yet full access to the code. Moreover, for those biologists who wish to learn programming, such an environment provides an appropriate support with domain examples as well as incremental programming features. A prototype has been built which comprises several components for biological sequence analyses, such as an alignment editor or a plot tool. The alignment editor behaves as a spreadsheet, and provides a programmable graphical tag mechanism that enables the user to visualize biological properties. The architecture of this environment relies on the concept of graphical objects, which provides the composition of application objects through formula and structured access to the internal representation of the application.