THESE de DOCTORAT de l'UNIVERSITE PARIS 6

Spécialité :
INFORMATIQUE

présentée

par Mr Philippe PIERNOT

pour obtenir le grade de DOCTEUR de l'UNIVERSITE PARIS 6

Sujet de la thèse :

Un système d'aide pour la mise en oeuvre de la programmation par démonstration

soutenue le 30 Juin 1995

devant le jury composé de :

Mlle BORNE Isabelle, rapporteur

Mr COT Norbert, directeur de thèse

Mr CYPHER Allen, examinateur

Mr LIEBERMAN Henry, rapporteur

Mr PERROT Jean-François, Président

Remerciements

Mes plus sincères remerciements vont :

à Monsieur Perrot, pour avoir accepté de présider mon jury de thèse et de s'être intéressé à mes travaux ;

à Mademoiselle Borne, pour avoir accepté d'être rapporteur et pour m'avoir suggéré de nombreuses améliorations concernant ce manuscrit. Je tiens aussi à remercier Mademoiselle Borne pour les fructueuses conversations que nous avons eues à propos du langage de programmation Smalltalk et des environnements de programmation utilisateur ;

à Monsieur Lieberman, pour avoir accepté d'être rapporteur et pour le temps consacré à lire et à juger ce travail. Le système Mondrian que Monsieur Lieberman a développé a été pour moi une source d'inspiration importante ;

à Monsieur Cypher, pour avoir accepté d'être examinateur. Son l'article sur Eager m'a fait découvrir le domaine de la Programmation Par Démonstration a fortement influencé mes travaux. Je remercie aussi Monsieur Cypher de m'avoir invité à participer au "Programming By Example Workshop" me donnant ainsi l'opportunité de rencontrer la plupart des chercheurs dans le domaine de la Programmation Par Démonstration ;

à mon directeur de thèse, Monsieur Cot, pour avoir encadré mes recherches, prodigué des conseils toujours avisés, relu ce manuscrit des ses premières versions et fait part d'une humeur souriante pendant toutes les années que j'ai passées à l'Université René Descartes.

Je tiens aussi à remercier :

Tous les compagnons de route du LIAP 5 (Laboratoire d'Intelligence Artificielle de l'Université Paris 5) notamment M. Yvon et F. Lefèvre avec qui j'ai débattu de nombreuses idées qui ont trouvé place dans AIDE ;

Monsieur Gruber, du Stanford Knowledge Systems Laboratory, pour avoir partagé ses nombreuses idées sur les applications possibles des mécanismes d'enregistrement de commandes, autres que la Programmation Par Démonstration ;

Monsieur Huyghes pour m'avoir permis d'implanter le précurseur de SPII lorsque j'étais scientifique du contingent à l'Aérospatiale.

Enfin je ne saurais oublier :

Ma femme Fabienne et mon fils Benoît , pour avoir partagé mes efforts tout le long de ces trois années de recherche ;

Mes parents pour avoir toujours soutenu mes entreprises pendant mes études.

Cette thèse a pu voir le jour grâce à un contrat de recherche de longue durée conclu en 1992 entre le LIAP 5, le CNET et le CNRS (projet Cogniscience) et qui avait pour but de développer des interfaces adaptatives et démonstrationnelles dans le domaine des Télécommunications. Ce contrat a permis d'établir une coopération scientifique très fructueuse entre plusieurs organismes de recherche internationaux tels que le département de génie de la connaissance de l'université de Madrid, le Media Lab du MIT et le département KSL (Knowledge Systems Laboratory) de l'université de Stanford. Nous tenons à en remercier les responsables du CNET et du CNRS concernés.

Liste des figures

Chapitre 1

Chapitre 2

Chapitre 3

Chapitre 4

Chapitre 5

Liste des tableaux

Table des matières

Résumé

Avec la naissance de la micro-informatique, l'industrie logicielle a connue un profond bouleversement. Ainsi, par le passé, les logiciels étaient souvent développés sur mesure pour résoudre un problème particulier alors qu'aujourd'hui, avec le passage à un marché de masse, cette approche n'est plus économiquement viable. Au lieu de cela, les utilisateurs doivent avoir recours à des applications génériques mais pas toujours bien adaptées. La Programmation Par Démonstration tente de remédier à cet état de fait en permettant à l'utilisateur d'adapter lui-même une application à ses propres besoins en programmant dans l'interface du système, c'est à dire en montrant au système ce qu'il doit faire sur un ou plusieurs exemples. Malheureusement, les systèmes de Programmation Par démonstration sont difficiles à implanter car ils mettent en jeu des techniques provenant de différents domaines tels que l'Interaction Homme-Machine, les Langages de Programmation et l'Intelligence Artificielle.

Cette thèse décrit la conception et l'implantation d'AIDE ainsi que la recherche qui a aboutie à son développement. AIDE est le premier squelette d'application destiné à aider les développeurs à intégrer les techniques de la Programmation Par Démonstration dans leurs applications. SPII, un prototype d'éditeur graphique développé avec AIDE est ensuite présenté afin d'illustrer l'utilisation d'AIDE. Enfin une évaluation d'AIDE a été effectuée : dans le cas de l'application SPII, l'utilisation d'AIDE a fait gagner un facteur 10 en terme du nombre de lignes de code pour supporter la programmation par démonstration.

Les principales contributions d'AIDE sont :

(1) AIDE représente l'historique sous forme hiérarchique (arbre de commandes) offrant un meilleur support pour l'annulation et la ré-exécution de commandes ainsi que la Programmation Par Démonstration. Cet historique est créé de façon *incrémentale* ;

(2) La seconde contribution est d'avoir développé le premier système de Programmation Par Démonstration *fonctionnant à travers plusieurs applications* et représentatant les commandes à un haut niveau d'abstraction ;

(3) AIDE intègre les mécanismes de correction d'erreur à la Programmation Par Démonstration ;

(4) AIDE fournit une *commande de recherche* qui peut être intégrée aux applications développées et qui permet à l'utilisateur de spécifier ses intentions.

Abstract

The recent personal computer revolution lead to profound changes in the software industry. Thus, in the past, software was custom made to solve a particular problem, whereas today, with the mass market economy this approach is no longer viable. Instead, users have to use generic application, not always well adapted to their needs. Programming By Demonstration is a technique trying to address this problem by letting the user customize herself her application by programming in the user interface of the system, i.e. by showing the system what to do on one or more examples. Unfortunately, Programming By Demonstration Systems are difficult to implement because they draw upon different domains such as Human-Computer Interaction, Programming Languages and Artificial Intelligence.

This thesis describes the design and the implementation of AIDE, as well as the research that led to its development. AIDE is the first application framework dedicated to the implementation of application providing Programming By Demonstration capabilities. SPII a prototype of a graphical editor implemented on top of AIDE is then presented to illustrate AIDE usage. Finally an evaluation of AIDE has been conducted: in the case of the SPII application, using AIDE reduced the number of lines necessary to implement the Programming By Demonstration capabilities by a factor of 10.

The main contributions of AIDE are:

(1) AIDE adopts a hierarchical representation of the application history (command trees) offering an improved support for both undo/redo and Programming By Demonstration capabilities. AIDE builds this history in an incremental way;

(2) AIDE is the first system based on high level commands to allow Programming By Demonstration across applications;

(3) AIDE integrates error recovery capabilities with Programming By Demonstration;

(4) AIDE provides an intuitive search command that can be integrated within application developed with AIDE, and allowing users to specify they intent.

Chapitre 1

Introduction : programmation utilisateur

La mode actuelle du logiciel facile à utiliser tend à accorder une importance excessive aux interfaces à manipulation directe. Pour être vraiment expressifs (et donc réellement faciles à utiliser), les logiciels ont besoin à la fois d'interfaces qui peuvent apprendre et de langages de programmation spécialisés accessibles à l'utilisateur.

Michael Eisenberg [Eisenberg 91]

1 Motivation

Dans les années soixante, la plupart des logiciels étaient développés en interne pour répondre aux besoins particuliers des utilisateurs. Ainsi, un site informatisé possédait souvent ses propres programmeurs qui écrivaient des logiciels spécifiquement pour le reste du personnel. Les grandes banques avaient leur propre programme de comptabilité, les grandes entreprises leur logiciel de gestion de stock etc. Et lorsqu'un logiciel provenant de l'extérieur était utilisé, il était généralement adapté au site dans lequel il était installé.

Aujourd'hui, avec l'avènement de la micro-informatique, les ordinateurs sont largement répandus au bureau comme à la maison. Malheureusement, les sites de taille modeste comme le petit commerce, l'atelier ou le domicile ne possèdent pas suffisamment de ressources pour développer ou faire développer leurs propres logiciels. Au lieu de cela, ils doivent avoir recours à des **applications** génériques disponibles sur le marché mais pas toujours bien adaptées.

Ainsi, la distance entre l'utilisateur et le programmeur d'un logiciel s'est accrue. Un logiciel n'est donc plus écrit en tenant compte des besoins d'un utilisateur particulier mais plutôt en essayant de répondre au plus grand nombre de situations possible. Par exemple, l'équipe qui a programmé le calendrier électronique Now Up To Date ne sait pas que votre commerce est fermé le dernier mardi du mois pour inventaire ; le programmeur qui a développé le logiciel de gestion du courrier électronique Eudora ne sait pas que vous possédez, en plus de votre adresse sur Internet, une seconde adresse sur le réseau Calvacom ; votre traitement de texte ne sait pas que les documents contenant des figures en couleur sont à imprimer sur la "ColorJet" alors que ceux en noir et blanc doivent être dirigés sur la "LaserWriter-3" etc.

Ceci contraint souvent l'utilisateur à mettre en correspondance la tâche qu'il désire accomplir avec les fonctionnalités génériques des applications à sa disposition. Il est inévitable qu'une telle "gymnastique intellectuelle" réclame des étapes superflues, que seul le programmeur de l'application peut automatiser. Par exemple, tous les jours lorsque j'allume mon ordinateur pour la première fois, je lance manuellement le logiciel Eudora pour accéder à mon courrier électronique. Le Macintosh ne permet pas d'automatiser cette tâche de manière satisfaisante dans la mesure où il fournit un dossier particulier ("Ouverture au démarrage") dont le contenu est lancé *à chaque démarrage* de la machine. Cette solution ne me convient pas car je redémarre mon ordinateur plusieurs fois dans la journée mais je souhaite qu'Eudora ne soit lancé automatiquement qu'*une seule fois par jour*. Pour couvrir ce besoin particulier, il faut avoir recours à la programmation.

Ainsi l'utilisateur n'a d'autre solution que d'attendre la version suivante du logiciel en espérant qu'elle procurera les fonctionnalités qui lui font défaut. Lorsque le logiciel en est à sa sixième version, l'utilisateur se trouve confronté à des menus hiérarchiques, des boîtes de dialogue qui s'enchaînent, une pléthore de barres d'icônes ou de palettes et pourtant le logiciel n'en fait toujours pas assez. En effet, pour des raisons économiques, le développeur implante les "améliorations" les plus demandées par les clients dans la version suivante du logiciel, augmentant par la même sa complexité. Ce nombre accru de fonctions est d'ailleurs souvent utilisé comme un argument marketing face à la concurrence. D'autre part, toujours pour des raisons essentiellement économiques, il ne peut satisfaire à toutes les exigences et donc certaines fonctionnalités resteront à l'écart, frustrant ainsi certains utilisateurs.

L'utilisateur se retrouve dans une situation ironique : l'ordinateur est censé être l'outil idéal pour exécuter des tâches répétitives, pourtant c'est l'utilisateur qui doit encore accomplir beaucoup d'entre elles. Une réponse à ce problème est la **Programmation Utilisateur**, c'est-à-dire permettre à l'utilisateur d'adapter lui-même une application à ses propres besoins grâce à la programmation. Bien que l'utilisateur ne soit généralement pas capable de programmer avec la même aisance que quelqu'un dont c'est le métier, il peut employer des solutions qui rendent la programmation plus accessible.

Les trois sections suivantes examinent les applications de la Programmation Utilisateur, les raisons pour lesquelles la programmation est une activité difficile ainsi que les solutions existantes qui peuvent en faciliter l'accès.

2 Applications de la programmation utilisateur

La programmation utilisateur permet au non-programmeur d'automatiser des tâches répétitives pour gagner du temps et diminuer les risques d'erreur, d'intégrer des applications pour développer une solution sur mesure ou encore d'adapter une application à ses besoins spécifiques. Nous passons maintenant en revue chacune de ses applications.

2.1 Automatiser des tâches répétitives

La vocation première de la programmation est l'automatisation de tâches répétitives. Par exemple, si l'utilisateur souhaite changer en caractère gras le mot "Programmation" dans tout ce document, il peut créer un programme accomplissant cette tâche au lieu de chercher manuellement chaque occurrence du mot et changer son style grâce au menu "Format". Ceci se traduit à la fois par un gain de temps et par une diminution du taux d'erreur.

2.2 Intégrer des applications

Un programme peut accomplir des tâches qui mettent en oeuvre plusieurs applications, fournissant ainsi le moyen de créer des solutions sur mesure. Un programme peut en effet envoyer des instructions à une application, recevoir le résultat pour ensuite le passer à d'autres logiciels. Par exemple, un tel programme peut effectuer une requête dans une base de données, copier le résultat dans une feuille de calcul afin de mettre à jour un graphe, puis insérer ce dernier dans un traitement de texte.

2.3 Adapter une application aux besoins de l'utilisateur

Un programme peut également être utilisé pour modifier ou ajouter des fonctions à une application existante. Pour adapter une application à ses besoins, l'utilisateur peut ainsi créer un programme qui sera déclenché à chaque fois qu'une action particulière est accomplie dans l'interface de l'application, comme appuyer sur un bouton ou choisir une option dans un menu. Ainsi, dans un traitement de texte, l'utilisateur peut créer un programme comptant le nombre de mots dans un document et rendre ce programme accessible au même titre que les autres commandes de l'application.

3 Pourquoi est-il difficile de programmer ?

Deux raisons principales font que les personnes ne sachant pas programmer perçoivent la programmation comme une activité difficile qu'ils ne souhaitent pas aborder : la première est liée aux connaissances nécessaires et la seconde est liée au talent ou à l'aptitude requise.

3.1 Quantité de détails à connaître

Le programmeur doit absorber une masse importante de connaissances ayant trait au langage de programmation utilisé : il doit maîtriser à la fois sa *syntaxe*, sa *sémantique* et ses *caractéristiques techniques internes*. Il est de plus fréquent que deux langages expriment une même sémantique de façon différente ; et dans les cas où deux langages possèdent des syntaxes similaires, le programmeur rencontre alors des différences de sémantique subtiles !

Le programmeur doit non seulement apprendre les idiomes liés au langage employé mais aussi ceux liés à la programmation en général (comme les variables, les tableaux, les structures de contrôle etc.). Même le programmeur expérimenté rencontre des difficultés lorsqu'il écrit son premier programme avec un nouveau langage de programmation. Il a souvent recours à des exemples de programmes qu'il essaie de comprendre puis de modifier.

Les connaissances emmagasinées par un programmeur ont été acquises au fil des ans dans des cours d'informatique ou sur le tas par analyse de programmes écrits par d'autres. La connaissance de ces "détails" fait partie du bagage que doit posséder un bon programmeur mais ne fait pas partie des préoccupations de l'utilisateur désirant accomplir sa tâche avec un effort minimal. Fournir à l'ordinateur une description en français de la tâche à réaliser n'est malheureusement pas suffisant aujourd'hui.

3.2 Capacité à créer des plans abstraits

En plus des connaissances requises, un talent particulier est nécessaire pour programmer : il s'agit d'être capable *d'élaborer et de communiquer des plans abstraits sans avoir de retour immédiat* [Soloway 82]. Le programmeur doit garder l'état d'une réalité intangible à l'esprit. Il ne voit pas les effets du programme qu'il rédige tant qu'il ne l'a pas exécuté : au lieu de cela, il doit avoir recours à son imagination.

Une personne sans aucune connaissance de la programmation n'est pas habituée à créer un plan de façon si abstraite. D'autre part, elle est habituée à communiquer avec d'autres personnes en français et non avec une machine à travers un langage de programmation. Ainsi le peintre en bâtiment sait peindre et décrire ce qu'il fait à un apprenti, mais il est improbable qu'il puisse écrire un programme contrôlant un robot

peintre. La plupart des personnes suivent et donnent des instructions sans problème : elles peuvent suivre et transcrire des recettes de cuisine, exécuter des procédures administratives, ou indiquer à leurs amis comment se rendre à leur domicile. Un point important est que toutes ces activités se passent dans une réalité physique familière : il est facile aux gens de prévoir les conséquences de leurs actions car ils sont habitués à leur réalisation physique et peuvent donc imaginer ce qui va se passer.

Une personne suivant ou créant un plan ayant trait à une réalité physique peut essayer ce plan étape par étape, et ainsi le modifier. Ceci peut être accompli mentalement ou physiquement. Par contre, le programmeur doit créer un plan sans pouvoir l'essayer en même temps qu'il l'écrit. Il ne peut pas expérimenter avec des parties du plan car dans les environnements traditionnels de programmation il n'existe souvent pas de moyen simple d'essayer des petites parties du programme. Lorsqu'il exécute le programme dans sa totalité, il lui est rarement possible de faire du pas à pas et modifier le programme simultanément. Tout ceci implique que le programmeur doit être capable de simuler mentalement le plan qu'il vient de créer et d'en imaginer les conséquences avant de l'exécuter.

Enfin, à ces difficultés s'ajoute le fait que la tâche à accomplir possède une difficulté inhérente. Il existe cependant des solutions destinées à faciliter la programmation : ces solutions essaient de réduire la masse de détails à connaître ou l'effort mental pour créer un plan qui est le programme.

4 Solutions pour faciliter la programmation

Les solutions disponibles pour faciliter la programmation couvrent un large domaine, allant des feuilles de calcul à la rédaction d'un programme dans un langage iconique. Nardi [Nardi 93], et Myers [Myers 90d] dressent un tableau complet des solutions disponibles. Nous nous intéressons ici aux solutions permettant de *contrôler ou modifier le comportement d'une ou plusieurs applications*, ce qui constitue un sous-ensemble de l'ensemble des solutions visant à faciliter la programmation au sens général.

4.1 Préférences

Les préférences (exemple donné figure 1.1) sont des alternatives offertes par le concepteur d'une application pour répondre aux différents besoins de l'utilisateur. Elles constituent la façon la plus élémentaire de modifier le comportement d'une application, même si elles ne mettent pas en oeuvre une réelle programmation, dans la mesure où l'utilisateur choisit simplement une alternative parmi d'autres.

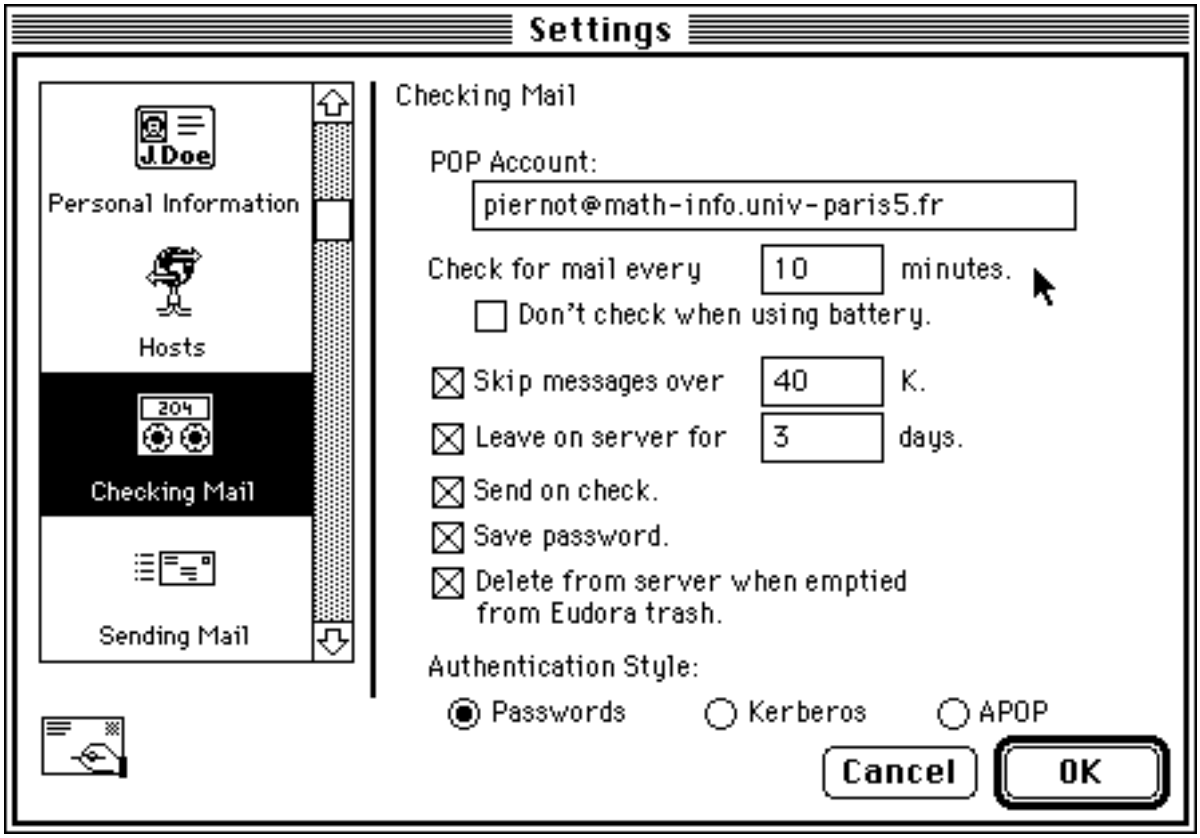


Figure 1.1 : Préférences proposées par Eudora,
un programme de gestion du courrier électronique.

Bien qu'utiles, lorsque les modifications susceptibles d'être apportées à une application sont bien cernées, les préférences présentent deux limitations importantes :

- La première limitation est qu'elles ne permettent d'accommoder qu'un nombre fini de situations prévues par le concepteur de l'application. Du fait des besoins potentiellement variés des utilisateurs, elles ne peuvent répondre à leur totalité. Ainsi, dans un logiciel de gestion du courrier électronique, il est rare de trouver une option permettant d'effacer tous les messages dupliqués.

- La seconde limitation est que pour répondre au plus de besoins possibles, il faut proposer un *nombre important d'options*, rendant difficile la configuration de l'application par l'utilisateur. Ainsi, le programme de gestion du courrier électronique Eudora (figure 1.1), ne possède pas moins de 64 choix de configuration. Un autre exemple est le "bureau" du Macintosh et ses nombreux panneaux de contrôle. Il en découle que dans certaines entreprises et universités, c'est à l'ingénieur système et non à l'utilisateur qu'incombe la tâche de configurer les applications. C'est un retour à la case départ puisque là encore, les sites aux ressources modestes n'ont pas toujours accès à l'expertise requise.

4.2 Langages d'application

Les langages d'application sont des langages de programmation spécialisés de haut niveau permettant de modifier ou d'étendre les fonctionnalités d'une application. Ces langages sont plus accessibles que les langages de programmation traditionnels car ils proposent à l'utilisateur un vocabulaire limité, adapté au domaine de l'application. De tels langages fournissent des équivalents textuels pour les commandes typiquement effectuées avec la souris. Ainsi, dans un éditeur graphique, changer la taille d'un rectangle en tirant un de ses coins, correspondra à l'instruction "`changer-taille (12, 20, 100, 200)`". Dans un éditeur de texte, l'utilisateur pourra insérer, mettre en italique ou effacer des caractères, des mots, des paragraphes etc. (figure 1.2). Dans un tableur, les objets manipulés seront des colonnes, des lignes ou des cellules qui pourront être sélectionnées, copiées, déplacées etc. (figure 1.3). Ces langages sont généralement suffisamment puissants pour permettre à l'utilisateur confirmé de changer de façon drastique le fonctionnement d'une application. Cependant ils possèdent eux aussi leurs problèmes :

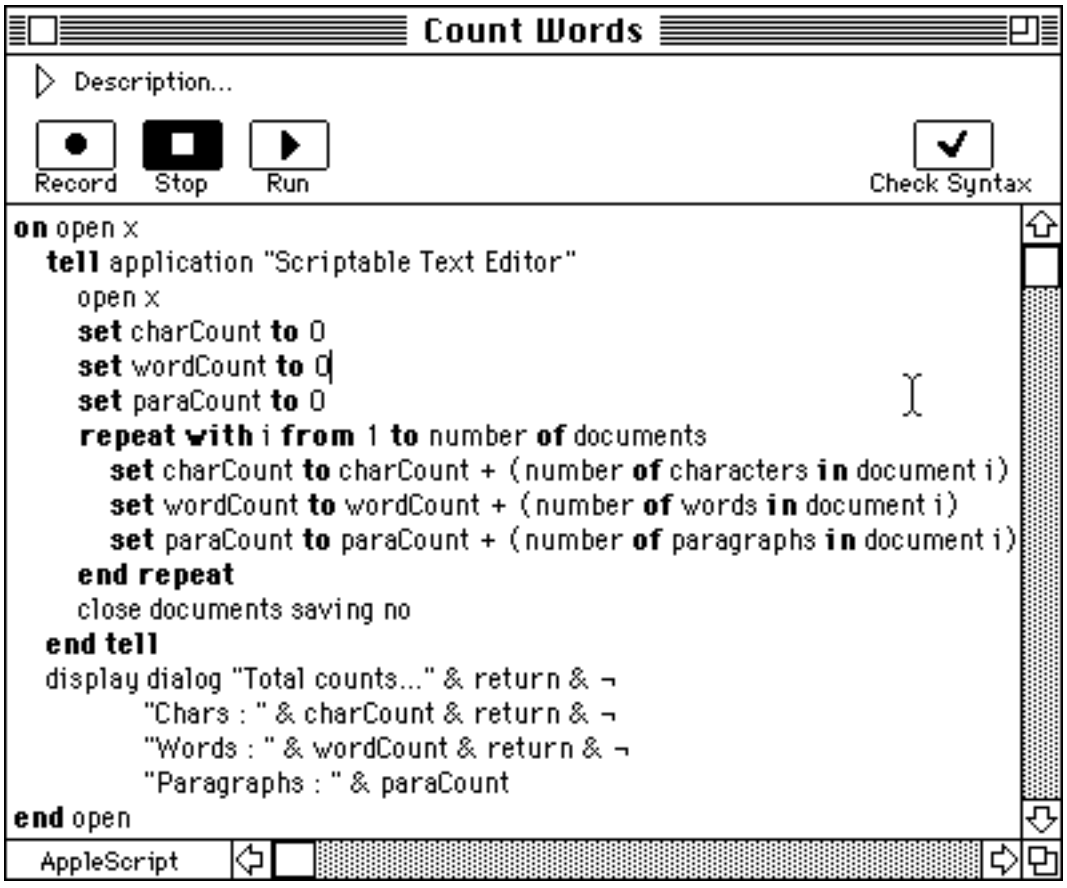


Figure 1.2 : Un programme écrit en AppleScript contrôlant un éditeur de texte afin de compter le nombre de caractères, mots et paragraphes dans la fenêtre courante.

- Un premier problème — aujourd'hui en passe d'être résolu — est que chaque logiciel propose son propre langage, entraînant un effort d'apprentissage croissant avec le nombre d'applications. Un des langages d'application les plus prometteurs est sans doute AppleScript [Apple 93a] (figure 1.2) dans la mesure où il permet de piloter de nombreuses applications de façon uniforme et peut être étendu par chacune d'elles par ajout de nouveaux objets et de nouvelles commandes. Ainsi l'objectif est de pouvoir contrôler toute application à partir d'un *langage de programmation unique*, ce qui limiterait l'effort d'apprentissage requis. Un second avantage lié à cette approche est qu'elle autorise la création de *programmes pilotant plusieurs applications :* par exemple, un tel programme pourrait charger tous les matins les cours de la bourse à l'aide d'un logiciel de télécommunications, pour ensuite demander à un tableur d'effectuer des calculs sur les valeurs boursières qui viennent d'être chargées.

- Un second problème — celui-ci beaucoup plus sérieux — est qu'AppleScript et plus généralement les langages d'application restent malgré tout des *langages de programmation à part entière*. Ils nécessitent donc non seulement la connaissance d'un vocabulaire et d'une syntaxe mais surtout la maîtrise des concepts de programmation tels que les structures de données — listes, tableaux etc. — et les structures de contrôle — boucles, conditions, procédures etc. — (se référer à la troisième section de ce chapitre). A ce titre, même si ils permettent d'étendre une application dans des proportions importantes, il n'en demeure pas moins que pour automatiser des tâches très simples, il ne devrait pas être nécessaire de recourir à de tels langages : en effet, il est possible de rédiger un programme pour dessiner un organigramme à l'écran mais il est tout de même plus simple d'utiliser un logiciel de dessin.

- Enfin, du point de vue du développeur de logiciels, les langages d'application sont *coûteux à mettre en oeuvre*. Ils réclament en effet un effort de conception (choix d'une syntaxe et d'un vocabulaire appropriés) et d'implémentation (écriture d'un interpréteur ou d'un compilateur intégré à l'application) important. Là encore une approche similaire à AppleScript va dans la bonne direction : AppleScript possède une syntaxe prédéterminée, un vocabulaire restreint mais extensible et fourni un ensemble de primitives pour que le développeur n'ait pas à créer un interpréteur de toutes pièces. Une autre approche adoptée par SchemePaint [Eisenberg 91] et AgentSheets [Repenning 93] est de proposer à l'utilisateur le même langage de programmation que celui qui a été utilisé pour créer l'application, éliminant ainsi la nécessité d'écrire un interpréteur. Cela suppose bien sûr que l'application soit écrite dans un langage interprété comme Lisp ou Smalltalk et réclame une certaine maîtrise du langage sous-jacent même si l'utilisateur peut faire directement appel aux commandes de l'application.

4.3 Enregistreurs de macro

Les enregistreurs de macro constituent l'étape suivante après les langages d'application : ils permettent aux utilisateurs de créer un programme — appelé ici **macro** — dans le langage de l'application en *enregistrant* la séquence de **commandes** qu'ils effectuent. La macro ainsi créée pourra être rejouée ultérieurement. En pratique, l'utilisateur amorce le processus avec la commande "Enregistrer Macro", exécute les commandes nécessaires puis finit avec la commande "Stopper Enregistrement" (figure 1.3). Beaucoup de logiciels proposent un enregistreur de macro, des traitements de texte aux logiciels de communications en passant par les tableurs. Certains systèmes comme QuicKeys [CE Software 93], Tempo II [Affinity 91] ou AppleScript [Apple 93a] proposent l'enregistrement de macro *étendue à tout le système* ce qui à l'avantage de présenter un mécanisme unique à l'utilisateur et de pouvoir créer des macros mettant en oeuvre plusieurs applications (avec tous les avantages décrits dans la section précédente). Les enregistreurs de macro permettent à l'utilisateur de créer des macros pour automatiser des tâches simples sans avoir à apprendre un langage de programmation. Par exemple, dans un traitement de texte la tâche peut être d'insérer la date courante à chaque fois qu'un nouveau document est créé, dans un logiciel de télécommunications elle peut être de charger périodiquement le cours d'une action donnée et dans un tableur, de tracer le graphe mensuel d'une valeur boursière. Certains problèmes demeurent néanmoins présents :

- Le problème majeur des enregistreurs de macros est qu'ils sont *trop littéraux* : la séquence de commandes enregistrée est rejouée sans discernement de leur part. Ainsi dans la figure 1.3 l'utilisateur a sélectionné les valeurs boursières pour le mois de janvier puis demandé de tracer le graphe correspondant. Lorsque la macro sera rejouée, les lignes 2 à 9 seront sélectionnées à nouveau. Mais pour le mois de février il y aura peut-être un nombre différent de valeurs boursières et dans ce cas la macro aura un comportement incorrect.

- Un second exemple est fourni par les enregistreurs de macro qui stockent des **commandes de bas niveau** comme `déplacer-souris`, `presser-bouton-souris`,`taper-caractère`, `sélectionner-menu` etc. Supposons que l'utilisateur sélectionne le fichier "a-lire.txt" et le renomme "a-lire.bak". Cette macro ne pourra être ré-exécutée correctement que dans le cas improbable où les coordonnées de l'icône représentant le fichier à renommer sont celles où était précédemment placé le fichier "a-lire.txt".

- Un dernier problème est que pour comprendre ou modifier le fonctionnement d'une macro existante, il faut connaître le langage de programmation de l'application (figure 1.3). D'un point de vue implémentation, ils possèdent donc le même problème que les langages d'application puisque là aussi il faut écrire un interpréteur et en plus un mécanisme pour enregistrer les commandes de l'utilisateur sous forme textuelle.

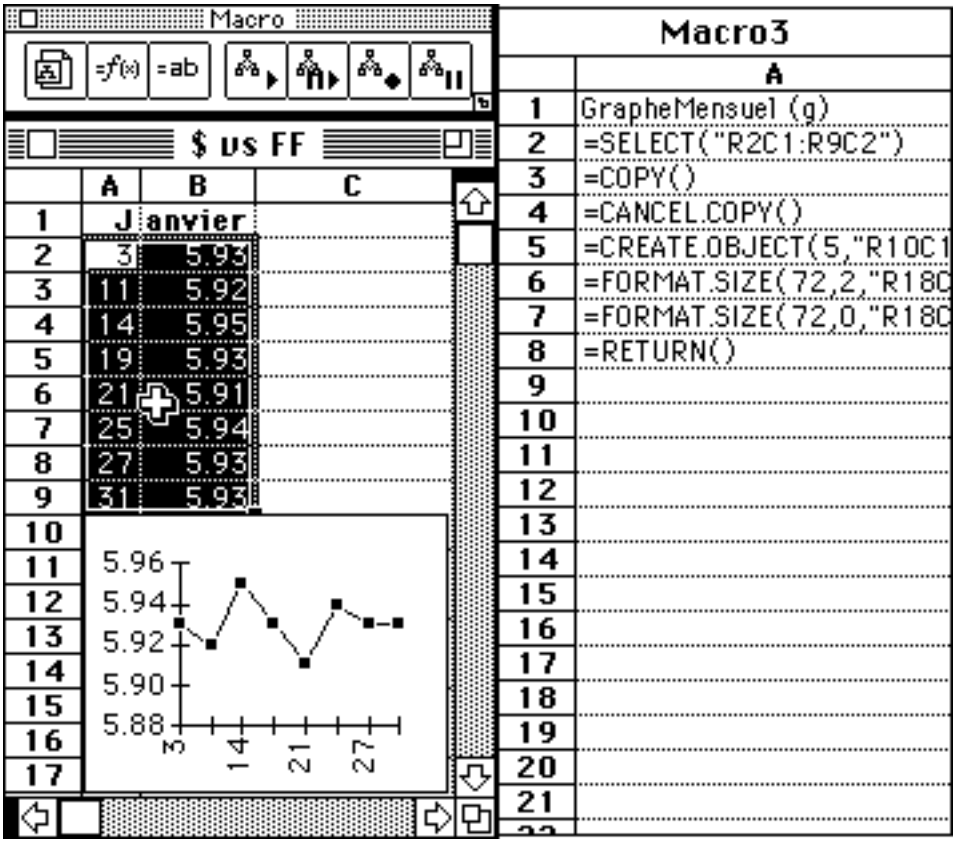


Figure 1.3 : Une macro enregistrée avec le tableur Excel traçant le graphe mensuel d'une valeur boursière.

4.4 Programmation par démonstration (PPD)

La Programmation Par Démonstration [Piernot 91], [Myers 92], [Cypher 93] — parfois aussi appelée à tort Programmation Par l'Exemple (voir section 4.5) — vise à étendre le concept des enregistreurs de macros selon deux directions :

- La première direction consiste à améliorer la façon de créer un programme. Tout comme les enregistreurs de macro, les systèmes de PPD enregistrent les commandes de l'utilisateur. Cependant ils n'utilisent pas tous la séquence démarrer enregistrement / stopper enregistrement. Certains enregistrent en permanence ce que fait l'utilisateur et lui proposent de terminer sa tâche lorsqu'ils détectent une répétition. D'autres permettent à l'utilisateur de consulter et de sélectionner des commandes passées pour créer un programme. Enfin, certains utilisent la métaphore enseignant / élève, c'est-à-dire que l'utilisateur explique ce qu'il fait à la machine en plus d'accomplir les commandes.

- La seconde direction vise à résoudre le problème lié au fait que les enregistreurs de macro sont trop littéraux. Les systèmes de PPD créent des programmes en *généralisant* l'enregistrement des actions de l'utilisateur, de telle sorte que le programme puisse être correctement exécuté *dans un contexte différent*. Le

programme n'est plus seulement une séquence de commandes mais peut aussi contenir des variables, des structures de données et des structures de contrôle comme des boucles, des conditions, des appels de procédure etc. Dans l'exemple de la figure 1.3, au lieu de sélectionner systématiquement les lignes 2 à 9, un programme créé par démonstration sélectionnerait les lignes allant de la seconde à la *dernière ligne précédant une cellule vide.* Cette fois-ci, le programme pourrait être exécuté correctement pour le mois de février.

Les systèmes de PPD sont donc plus expressifs que les enregistreurs de macro et permettent de créer une riche variété de programmes *par démonstration*, comme par exemple, un programme soulignant le titre de toutes les sections d'un document dans un traitement de texte. Un autre programme peut tracer un rectangle autour de l'objet sélectionné dans un logiciel de dessin (une commande qui fait souvent cruellement défaut). Enfin, un programme peut chercher dans l'annuaire électronique tous les plombiers situés dans le 18ème arrondissement. La figure 1.4 illustre le système de PPD Eager [Cypher 91] aidant l'utilisateur à créer une colonne de boutons dans le logiciel HyperCard. L'utilisateur ajoute les deux premiers boutons et Eager se charge du reste en générant un programme puis en l'exécutant. Eager est décrit plus en détail dans le chapitre suivant.

L'avantage de la programmation par démonstration est qu'elle permet de *programmer dans l'interface utilisateur* [Halbert 84]. Puisque les systèmes de PPD sont basés sur des interfaces à manipulation directe [Shneiderman 83], ils bénéficient de leurs avantages :

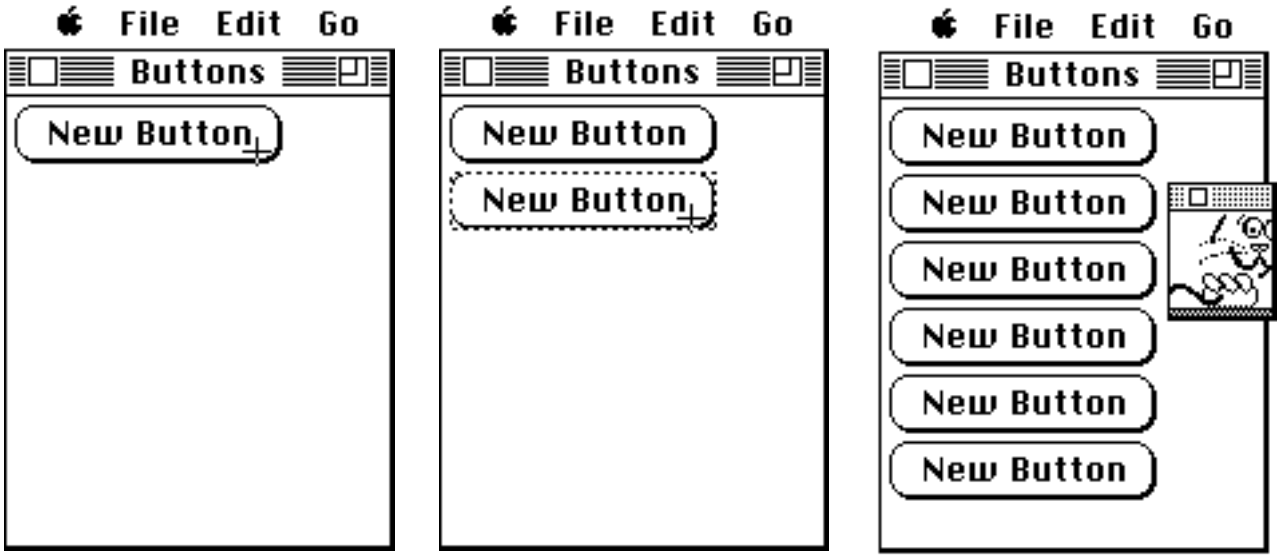


Figure 1.4 : Eager aide l'utilisateur à créer une colonne de boutons dans HyperCard.

- les débutants peuvent apprendre facilement les fonctionnalités de base ;
- les experts peuvent accomplir très rapidement un grand nombre de tâches ;
- un utilisateur confirmé mais intermittent peut se souvenir des concepts ;
- les messages d'erreur sont rarement nécessaires ;
- ils fournissent une réponse immédiate à l'utilisateur, ce qui lui permet de vérifier si les actions qu'il accomplit le mènent bien vers son objectif ;
- l'utilisateur ressent moins d'anxiété car ses actions sont facilement réversibles et parce que le système est compréhensible.

En plus des avantages liés aux systèmes à manipulation directe, les systèmes de PPD possèdent leurs propres avantages :

- les opérations abstraites peuvent être accomplies de façon concrète : pour écrire un programme, l'utilisateur montre le comportement désiré à l'aide d'un ou de plusieurs exemples réels ;
- il n'y a pas besoin de mettre en correspondance un langage textuel avec les objets présents à l'écran : pour faire référence à une action, l'utilisateur effectue simplement cette action au lieu d'avoir recours à une instruction textuelle. *L'utilisateur programme dans l'interface du système* ;
- ils mettent la programmation à la portée du néophyte.

Bien sûr les systèmes de PPD présentent leurs propres difficultés :

- ils sont *difficiles à mettre en oeuvre* car aux difficultés d'implantation liées aux langages d'application et aux enregistreurs de macro s'ajoutent celles qui sont liées à la généralisation d'un programme ;
- comme avec les enregistreurs de macro, la visualisation d'un programme est problématique d'autant plus que non seulement les commandes doivent être représentées mais aussi leur généralisation ;
- lorsqu'ils généralisent les commandes, ils peuvent *commettre des erreurs* car ils doivent deviner les intentions de l'utilisateur. Les utilisateurs peuvent être inconfortables avec un système qui se trompe.

4.5 Programmation par l'exemple (PPE)

La Programmation Par l'Exemple est similaire à la Programmation Par Démonstration dans la mesure où l'utilisateur crée un programme en fournissant des exemples. Cependant, dans les systèmes de PPD un exemple est constitué de la liste des commandes effectuées par l'utilisateur pour accomplir une tâche particulière : c'est cette liste qui est utilisée pour générer le programme. Dans le cas des systèmes de PPE, un exemple est simplement un couple (*état initial, état final*) : c'est au système d'inférer la transformation permettant de passer de l'état initial à l'état final sans avoir recours aux étapes intermédiaires. Pour illustrer la différence entre les deux approches imaginons que l'utilisateur souhaite enseigner à un éditeur de texte comment reformater une liste de numéros de téléphone (tableau 1.1). Dans un système de PPE comme Editing By Exemple [Nix 85] l'utilisateur fournit l'état initial, effectue les commandes nécessaires puis spécifie que l'état final a été atteint : le système doit alors inférer que le deuxième mot de chaque ligne a été remplacé par ":". Dans un système de PPD comme TELS [Mo 90] l'utilisateur enseigne la tâche exactement de la même façon, la seule différence étant que le système enregistre les étapes intermédiaires et utilise celles-ci pour générer la transformation. Chacune de ces approches possède ses propres avantages et inconvénients :

Un système de PPD utilise la séquence de commandes effectuées par l'utilisateur et permet donc de synthétiser des programmes complexes car c'est l'utilisateur qui fournit la trace de l'algorithme. Par contre cette approche est très sensible aux variations de l'utilisateur : dans l'exemple du tableau 1.1, si l'utilisateur n'avait pas édité chaque ligne de façon consistante, le système n'aurait pas pu détecter une répétition.

Programmation Par l'Exemple	**Programmation Par Démonstration**

Etat initial :
 	
`Paul Durant 42 55 34 56`
`Isabelle Durant 43 45 34 56`
`Jean Dupont 45 34 56 99`
 	
Etat final :
 	
`Paul : 42 55 34 56`
`Isabelle : 43 45 34 56`
`Jean : 45 34 56 99`
	`sélectionner mot 2 de la ligne 1`
	`effacer`
	`insérer ":"`
	
	`sélectionner mot 2 de la ligne 2`
	`effacer`
	`insérer ":"`
	
	`sélectionner mot 2 de la ligne 3`
	`effacer`
	`insérer ":"`

Tableau 1.1 : Comparaison entre les exemples enregistrés par un système de Programmation Par l'Exemple et un système de Programmation Par Démonstration.

A l'opposé. les systèmes de PPE ne sont pas sensibles à la façon dont la tâche a été accomplie par l'utilisateur puisque les étapes intermédiaires ne sont pas prises en compte. Ainsi, dans la tâche représentée dans le tableau 1.1, pour la première ligne, l'utilisateur peut avoir frappé six fois la touche effacer puis frappé le caractère " :". Pour le seconde ligne, l'utilisateur peut avoir sélectionné le second mot, frappé une fois la touche effacer puis tapé le caractère ":". Enfin pour la troisième ligne, l'utilisateur peut avoir inséré le caractère ":", sélectionné le second mot et pressé la touche "effacer". La contrepartie est qu'il devient très difficile voir impossible d'inférer la transformation appropriée lorsque celle-ci atteint un certain niveau de complexité.

5 Problématique

5.1 Pourquoi étudier les systèmes démonstrationnels ?

La Programmation Utilisateur connaît un essor assez récent car — comme nous l'avons abordé au début de ce chapitre — les limitations des interfaces à manipulation directe commencent à se faire cruellement sentir. Ainsi, ces interfaces sont difficiles à étendre et les analogies qu'elles entretiennent avec le monde réel sont souvent limitées, ce qui entraîne des inconsistances pouvant troubler l'utilisateur [Repenning 93]. Dans [Negroponte 91], Negroponte prédit que la prochaine génération d'interfaces homme-machine fournira des *agents* permettant à l'utilisateur de *déléguer* certaines tâches. Comme c'est le cas avec des agents humains, l'utilisateur devra leur enseigner la façon d'accomplir une tâche donnée (l'état de l'art en intelligence artificielle ne laisse d'ailleurs pas espérer l'existence d'agents totalement autonomes dans un futur proche).

Cette thèse porte sur les systèmes de Programmation Par Démonstration car ils représentent une évolution significative de la Programmation Utilisateur en permettant à l'utilisateur de *programmer dans l'interface du système*. Les systèmes de PPD fournissent une transition facile entre la manipulation directe et la délégation et devraient se développer dans les années à venir (des applications reprenant certains principes de la PPD commencent déjà à apparaître sur le marché [Cypher 95]). Lorsque l'on prend connaissance de ce domaine, on constate qu'une vingtaine de systèmes de Programmation Par Démonstration ont été développés, chacun séparément. De nombreux projets démarrent dans le domaine de la Programmation Par Démonstration et il est important de minimiser la duplication d'efforts: le point de vue pris dans cette thèse est d'*aider le développeur* à implanter de tels systèmes.

5.2 Problème : les systèmes de PPD sont difficiles à implanter

Implanter un système de Programmation Par Démonstration est une tâche complexe car ces systèmes mettent souvent en jeu des techniques d'intelligence

artificielle (apprentissage symbolique, représentation des connaissances, inférence etc.), en plus des techniques nécessaires à la mise en oeuvre d'un enregistreur de macro (interpréteur d'un langage de programmation, mécanisme de gestion de l'historique) et des techniques d'interaction homme-machine (création, visualisation et invocation d'un programme, correction d'erreur etc.) [Myers 92]. Le travail nécessaire à l'implantation d'un nouveau système est conséquent et aucune fondation n'existe aujourd'hui pour venir bâtir de tels systèmes. Cela à deux conséquences : pour aborder ce domaine de recherche, l'accès est difficile et beaucoup d'effort est perdu à réimplanter des mécanismes qui existent déjà dans d'autres systèmes. D'autre part, tant qu'une plate-forme de développement robuste n'existera pas — probablement sous la forme d'un squelette d'application intégré au système d'exploitation — il n'y aura pas d'applications de la PPD à grande échelle.

5.3 Solution : un squelette d'application pour la PPD

Il paraît donc très important de proposer une telle plate-forme de développement aux chercheurs en Programmation Par Démonstration dans un premier temps, puis éventuellement aux développeurs d'applications dans un second temps. Cela rendrait plus facile l'accès au domaine et permettrait de tester rapidement de nouvelles idées sans avoir à réinventer la roue à chaque fois. Une telle plate-forme aiderait au développement d'une grande variété de systèmes de PPD en fournissant des opérations *génériques* et *réutilisables*. Une seconde caractéristique que doit posséder cette plate-forme est d'être *extensible* de façon à ce le développeur puisse l'adapter à sa situation particulière, incorporer de nouvelles techniques, ou encore proposer les améliorations qu'il a créées à l'ensemble des développeurs.

Coutaz [Coutaz 90] répartit les outils pour la construction d'interfaces homme-machine selon trois catégories : les boîtes à outils, les squelettes d'application et les générateurs d'interfaces.

- Les *boîtes à outils* sont des bibliothèques de procédures adaptées à la programmation d'interfaces homme-machine. Elles offrent de nombreux services tels que la gestion des événements souris et clavier, l'affichage des fenêtres et de leur contenu, la gestion des objets d'interface comme les menus, les boutons et les champs texte etc. Ces boîtes à outils se situent généralement à un niveau proche de la machine et dans les systèmes d'exploitation modernes elles peuvent contenir des milliers de fonctions : il découle de ces deux points que leur apprentissage est difficile et leur utilisation est accompagnée de duplication d'efforts. Le développeur doit en effet réimplanter des fonctionalités similaires d'une application à l'autre (telles que les séquences d'initialisation, le rafraîchissement des fenêtres, la gestion des événements etc.) car les boîtes à outils ne fournissent pas la charpente d'une application mais seulement les briques de base.

- Les *squelettes d'application* sont des assemblages logiciels réutilisables et extensibles qui réalisent une large part des fonctions de l'interface d'une application. Ils remédient à certains des problèmes liés aux boîtes à outils, en particulier la duplication d'efforts et à un degré moindre l'apprentissage. Ainsi, la tâche du développeur consiste à remplir les trous du squelette et à redéfinir les parties prédéfinies du système inadaptées au besoin particulier de son application. La réutilisation, l'extensibilité et la surcharge de logiciel se traduisent directement dans les termes de la programmation par objets. La plupart des squelettes d'application utilisent donc cette technique et sont construits au dessus d'une boîte à outil. L'avantage des squelettes d'application est qu'ils fournissent une architecture logicielle saine : le developpeur n'a plus à déterminer l'assemblage correct des briques logicielles. De plus, le niveau de service procuré est plus proche des besoins du développeur. L'inconvénient majeur tient essentiellement aux difficultés de la réutilisation logicielle : pour réutiliser de façon optimale un squelette d'application il faut bien comprendre son fonctionnement et il faut aussi que le type d'application à développer rentre dans le moule. Enfin, lorsqu'il doit étendre ou surcharger certaines parties du squelette, l'utilisateur retombe inévitablement sur les services de la boîte à outil.

- Les *générateurs d'interfaces* créent l'interface d'une application à partir d'une spécification textuelle ou graphique. Cette interface est reliée à un noyau d'exécution qui s'apparente à un squelette d'application. Cette catégorie en est encore au stade de la recherche dans la mesure où aucun système ne de permet de créer des interfaces avec le même niveau de généralité que les boîtes à outils ou les squelettes d'application.

Parmi ces trois alternatives nous avons retenu les squelettes d'application car ils sont un bon compromis pour le type de plate-forme logicielle que nous souhaitons développer : les boîtes à outils se situent à un niveau d'abstraction trop bas tandis que les générateurs d'interfaces sont d'avantage au stade de la construction d'écrans que de la définition du comportement.

5.4 Méthodologie

Nous avons tout d'abord *analysé et comparé* les systèmes démonstrationnels les plus marquants afin de comprendre leur fonctionnement général et les solutions techniques particulières mises en oeuvre, en nous basant sur la littérature disponible. De plus, un certain nombre de systèmes de Programmation Par Démonstration ont été collectés puis évalués. Nous avons extrait des traits communs entre les divers systèmes de PPD afin de proposer une architecture logicielle.

Dans un second temps, nous avons cherché les technologies susceptibles de faciliter la création d'un tel squelette. Il est apparu que le modèle objet est à la base de nombreux squelettes d'application comme MacApp [Wilson 90] et MotifApp [Young

92] et offre les capacités d'extensibilité et de réutilisabilité nécessaires. Nous nous sommes également inspiré des techniques récentes de communication inter-application permettant de contrôler des applications de l'extérieur et ceci sans intervention humaine (c'est-à-dire sans manipulation clavier ou souris).

Nous avons ensuite implanté notre architecture dans l'environnement Smalltalk [Goldberg 89] car ce dernier regroupe dans un environnement interactif un langage de programmation par objets et un squelette d'application. Cette architecture a été *validée* par la création d'une application utilisant le squelette. Enfin nous avons effectué une évaluation du squelette et tiré des conclusions sur ce que nous avons appris.

6 Conclusion

Dans ce chapitre nous avons présenté les techniques existantes qui permettent de programmer une application afin de modifier son comportement. La Programmation Par Démonstration est une nouvelle technique que nous avons choisi d'étudier car elle offre un certain nombre d'avantages tels que la non nécessité de connaître la programmation et la puissance d'expression. Le chapitre suivant dresse un état de l'art de la Programmation Par Démonstration.

Chapitre 2

Systèmes de programmation par démonstration : état de l'art

Si l'utilisateur sait effectuer une tâche avec l'ordinateur, cela devrait être suffisant pour créer un programme accomplissant cette tâche. Il ne devrait pas être nécessaire d'apprendre un langage de programmation comme C ou Basic. Au lieu de cela, l'utilisateur devrait être à même d'instruire l'ordinateur de "faire ce qu'il fait", et l'ordinateur devrait créer le programme qui correspond aux actions de l'utilisateur.

Allen Cypher [Cypher 93].

1 Introduction

Selon la terminologie de Myers [Myers 92], une interface démonstrationnelle est une interface qui permet à l'utilisateur de construire un programme abstrait en effectuant des actions sur des exemples concrets (souvent par manipulation directe). Myers propose une classification des interfaces démonstrationnelles qui prend en compte deux critères principaux :

Intelligente / non intelligente : une interface démonstrationnelle est intelligente si elle *infère* un programme en utilisant des *heuristiques.* Dans une interface démonstrationnelle non intelligente, l'utilisateur guide le système en effectuant lui-même les généralisations.

Programmable / non programmable : une interface démonstrationnelle est programmable si elle peut générer des programmes contenant des boucles, des conditions et des variables. Une telle interface utilise généralement la liste des commandes effectuées par l'utilisateur pour créer le programme. Les interfaces démonstrationnelles non programmables génèrent des programmes beaucoup plus limités : elles utilisent souvent des couples d'exemples (représentant une liste d'arguments et la valeur de retour correspondante) pour inférer une transformation [Lipton 90], [Myers 91b], [Nix 85].

Cette thèse s'intéresse aux systèmes de Programmation Par Démonstration, c'est-à-dire selon Myers des systèmes à base d'interfaces démonstrationnelles, *programmables, intelligentes ou non.* Afin de dresser un état de l'art il a été nécessaire d'isoler puis de raffiner un ensemble de critères permettant de comparer les différents systèmes de Programmation Par Démonstration existants. Dans ce chapitre, nous présentons tout d'abord les critères retenus, puis nous analysons par ordre chronologique cinq des systèmes qui ont marqué le développement des systèmes de PPD par rapport aux critères choisis. Nous dressons enfin un tableau comparatif de ces systèmes et nous présentons leurs points communs ainsi que leurs avantages respectifs — susceptibles de nous aider dans la conception de notre architecture logicielle.

2 Critères retenus

2.1 Domaine d'application et utilisateurs ciblés

Les systèmes de Programmation Par Démonstration couvrent une large variété d'applications. Certains sont spécialisés dans un domaine plus ou moins large comme le traitement de texte [Mo 90], la robotique [Heise 89], la création d'interfaces graphiques [Myers 89,90a,91b,91c], les opérations d'une calculette de bureau [Witten 81], l'animation graphique [Stasko 91] etc. A l'opposé, d'autres systèmes couvrent plusieurs domaines d'application [Halbert 84] ou sont indépendants de tout domaine [Potter 93].

Lors de l'analyse d'un système démonstrationnel, il est également important de considérer la catégorie d'utilisateurs ciblée car ce paramètre influe sur la conception du système. Ainsi, un système s'adressant à l'utilisateur n'ayant absolument aucune connaissance de la programmation aura tendance à effectuer les généralisations lui-même (cf. Eager ou Mondrian, sections 3.3 et 3.5) tandis qu'un système supposant certaines notions de programmation pourra impliquer d'avantage l'utilisateur dans le processus de généralisation (cf. Smallstar ou Chimera, sections 3.1 et 3.4). Les systèmes de Programmation Par Démonstration s'adressent à une grande variété d'utilisateurs, sachant ou non programmer, membres de diverses professions, utilisateurs d'une application particulière etc.

2.2 Interaction

De façon générale, l'activité de programmation est constituée de plusieurs phases incluant la création, l'invocation, l'exécution, la correction, la visualisation et la modification d'un programme. Comme c'est le cas dans les environnements de programmation traditionnels, les systèmes démonstrationnels enchaînent ces phases de différentes façons. Dans certains systèmes comme Mondrian [Lieberman 93] — descendants directs des enregistreurs de macros — l'utilisateur enregistre le programme pour ensuite l'exécuter. Dans d'autres systèmes, beaucoup plus interactifs, les différentes phases sont enchevêtrées [Maulsby 89a,b]. Nous passons maintenant en revue chacune de ces phases.

2.2.1 Création du programme

La majorité des systèmes de Programmation Par Démonstration descendent des enregistreurs de macro et donc enregistrent *à la demande de l'utilisateur* les commandes effectuées, puis les généralisent pour les rejouer ultérieurement. D'autres systèmes observent *en permanence* les commandes de l'utilisateur afin de *prédire* ses futures actions et terminer la tâche courante. Enfin, une dernière catégorie de systèmes adopte un comportement intermédiaire en enregistrant les commandes de l'utilisateur *en permanence*, et en permettant à ce dernier de *sélectionner* les commandes qu'il souhaite pour créer un programme.

En plus de créer le programme en montrant les commandes à effectuer, l'utilisateur doit souvent guider le processus de généralisation. Dans les systèmes sans inférence (non intelligents), c'est l'utilisateur qui doit effectuer toutes les généralisations à la main. D'autres systèmes (intelligents) infèrent les intentions de l'utilisateur pour déterminer les structures de contrôle et de données. Dans ce cas, la phase de création du programme est parfois accompagnée d'un retour utilisateur afin de montrer les inférences effectuées par le système et permettre à l'utilisateur de les corriger.

2.2.2 Invocation et exécution du programme

Invocation

Pour invoquer un programme — c'est-à-dire déclencher son exécution — plusieurs méthodes sont à la disposition de l'utilisateur. Celle qui est la plus répandue laisse à l'utilisateur le soin d'invoquer lui-même le programme en sélectionnant celui-ci dans un menu, en cliquant sur une icône ou un bouton voir encore en pressant une combinaison particulière de touches. Une autre possibilité est l'invocation automatique d'un programme selon les changements qui interviennent dans le système. De tels changements peuvent être basés sur le temps (on rencontre ce type d'invocation avec le dossier de démarrage du Macintosh par exemple, cf. chapitre 1 section 1), le changement des propriétés d'un objet (par exemple lorsqu'un objet est placé dans la poubelle), ou l'arrivée d'un **événement** extérieur (comme l'insertion d'une disquette) etc. Les méthodes

d'invocation jouent un rôle important car parfois la tâche à effectuer est très simple mais l'important est de savoir quand l'accomplir. Ainsi on peut imaginer que chaque fois que la cote d'une action particulière dépasse un seuil donné, un message soit envoyé au possesseur du portefeuille boursier. De la même façon, on peut vouloir lancer l'exécution d'une application détectant les virus chaque fois qu'une disquette est insérée.

Exécution

A des fins de sécurité et de correction d'erreur, il est important de fournir un mécanisme permettant de contrôler l'exécution du programme comme par exemple effectuer du pas à pas ou encore interrompre puis reprendre son exécution. Lorsque le système ne sait pas quoi faire et a besoin d'information, il peut aussi transférer le contrôle à l'utilisateur. Dans certains systèmes l'utilisateur peut de plus annuler les effets d'un programme, ce qui est très important si le système a généré un programme ne correspondant qu'imparfaitement aux désirs de l'utilisateur, si l'utilisateur a invoqué le programme par mégarde ou si l'utilisateur s'est trompé en créant le programme.

2.2.3 Visualisation du programme

Afin de pouvoir comprendre un programme et le corriger il est nécessaire de fournir une visualisation de celui-ci. Cette visualisation peut être textuelle, dans ce cas elle présente la forme d'un langage de programmation (CUSP pour Smallstar, Smalltalk pour Rehearsal World [Gould 84a,b]) ou graphique sous forme d'une bande dessinée (Mondrian, Pursuit, Chimera). Le problème d'une représentation graphique est qu'il est difficile d'exprimer les généralisations. La visualisation peut être de plus connectée au mécanisme contrôlant l'exécution du programme et représenter ainsi non seulement le programme mais aussi son exécution. Ainsi, Pygmalion [Smith 75] représente le programme comme un film et Eager anticipe la prochaine commande de l'utilisateur sans l'exécuter.

2.3 Inférence et implantation

Certains systèmes enregistrent les commandes de l'utilisateur et laissent à l'utilisateur le soin d'ajouter les structures de contrôle comme les conditions, les boucles et les variables. Une seconde possibilité abordée dans la section 2.2.1 est de laisser le système "deviner" les intentions de l'utilisateur et effectuer les généralisations appropriées. Dans ce cas, le système doit posséder un moteur d'inférence contenant plusieurs heuristiques afin de généraliser les commandes enregistrées.

Les langages utilisés pour l'implantation des systèmes démonstrationnels sont variés allant de langages de bas niveau comme C à des langages de haut niveau comme Smalltalk ou Lisp.

3 Principaux systèmes démonstrationnels

3.1 Smallstar — Halbert 1984

3.1.1 Domaine d'application et utilisateurs ciblés

Smallstar [Halbert 84] est une simulation de l'interface graphique du système documentaire Star de Xerox, à laquelle des capacités de Programmation Par Démonstration ont été rajoutées. Star [Smith 82] est un outil de traitement de documents à vocation généraliste, précurseur du Macintosh. Star utilise la métaphore du bureau avec des icônes représentant les différents objets du système (programmes, imprimantes, documents, formulaires, boîtes aux lettres, tableaux etc.). Dans cet environnement, l'utilisateur peut éditer du texte et des graphiques, envoyer et recevoir des messages, remplir des formulaires, accéder à des bases de données et imprimer des documents. Star possède également un langage d'application appelé CUSP (CUStomer Programming) avec lequel l'utilisateur peut développer des applications sur mesure. Smallstar est le premier système de Programmation Par Démonstration destiné au grand public — c'est-à-dire à des utilisateurs n'ayant pratiquement aucune connaissance de la programmation — dans le but d'automatiser les tâches répétitives souvent rencontrées lorsqu'ils utilisent le Star et ceci sans avoir recours au langage CUSP.

3.1.2 Interaction

Création du programme

Pour créer un programme l'utilisateur commence par ouvrir un bouton ou une icône programme puis presse le bouton "Start Recording" pour démarrer l'enregistrement (figure 2.1). Il effectue ensuite les étapes nécessaires puis presse le bouton "Stop Recording" pour arrêter l'enregistrement. On reconnaît là un descendant des enregistreurs de macro. Une fonctionnalité utile est la possibilité d'éditer un programme existant soit en insérant des commandes à n'importe quel point du programme (en passant en mode enregistrement) soit en effaçant des instructions. Bien sûr, cela peut poser des problèmes : l'utilisateur doit s'assurer que le programme garde sa cohérence après modification. En effet, si une instruction est ôtée du programme il faut s'assurer que les instructions suivantes peuvent être exécutées correctement (par exemple, si l'utilisateur efface une instruction créant un fichier, toutes les instructions suivantes faisant référence à ce fichier ne pourront pas être exécutées). De même, si une commande est insérée, il faut vérifier que les instructions qui la suivent puissent être exécutées (par exemple, si une instruction effaçant un fichier est insérée, toutes les instructions qui font référence à ce fichier provoqueront une erreur). Une fois la phase d'enregistrement du programme terminée, l'utilisateur doit effectuer les généralisations nécessaires. Deux types de généralisations peuvent prendre place :

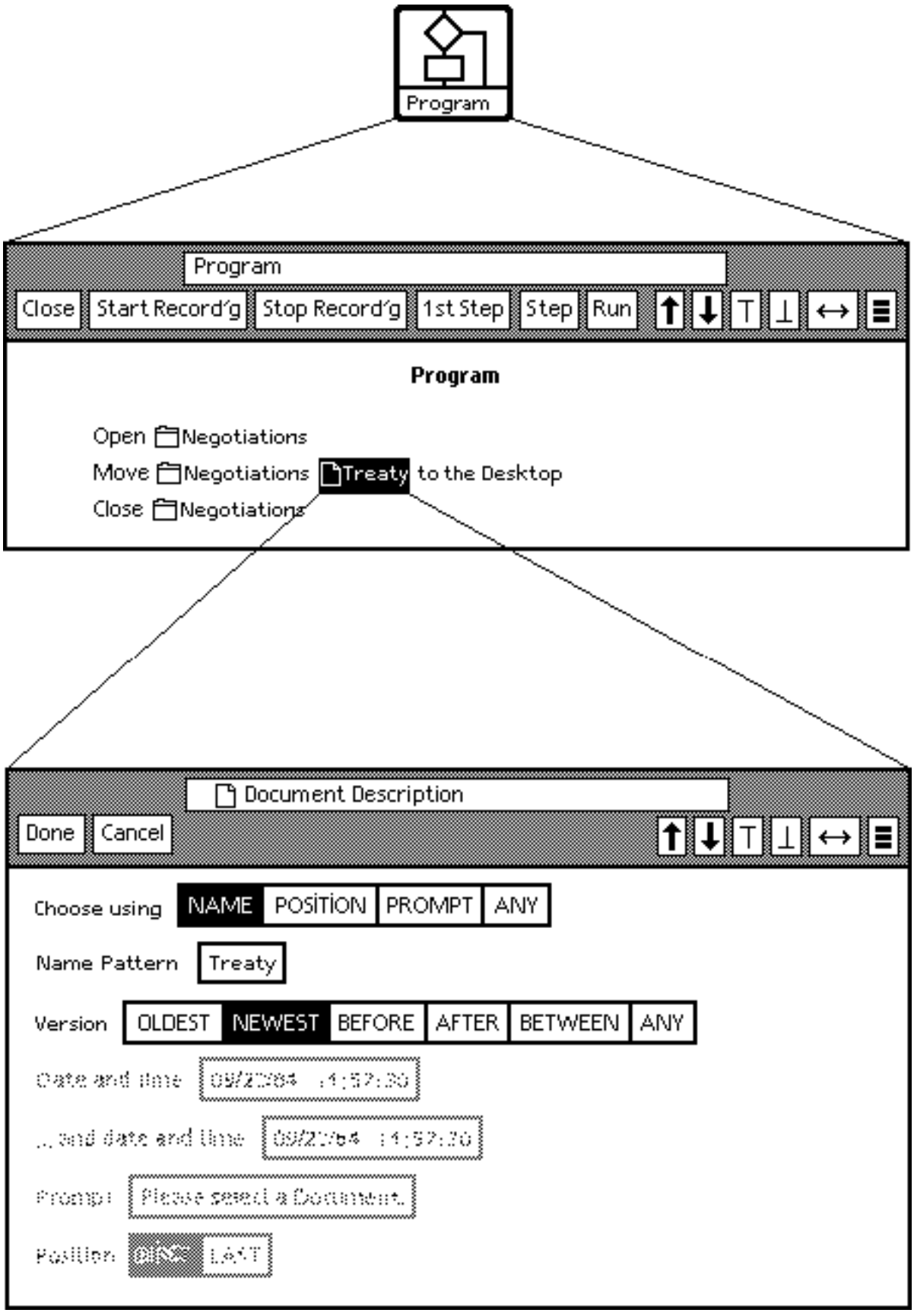


Figure 2.1 : Smallstar : une macro qui place la dernière version du fichier "Treaty" sur le bureau.

Le premier type de généralisation consiste à déterminer les *structures de contrôle* comme les boucles et les conditions. Pour ajouter une boucle, l'utilisateur sélectionne tout d'abord les instructions qui formeront le corps de la boucle puis choisit l'option "Repeat" dans un menu et sélectionne le critère pour choisir les objets. Dans les systèmes de PPD, les conditions sont généralement plus difficiles à rajouter par démonstration que les boucles car il faut montrer les deux chemins que peut emprunter le programme. Pour remédier à ce problème, Smallstar ne permet de créer que des conditions du type si-alors. Pour créer une condition du type si-alors-sinon, l'utilisateur doit créer une seconde condition avec le prédicat opposé. Tout comme pour les boucles, l'utilisateur sélectionne les instructions qui formeront la partie action de la condition. L'utilisateur spécifie la partie condition en ouvrant une boîte de dialogue ressemblant à une calculette de bureau grâce à laquelle il peut spécifier la condition sous forme d'une expression logique.

Le second type de généralisation consiste à généraliser les *arguments des commandes*. La contribution majeure de Smallstar réside dans les "Descripteurs de Données". Après avoir enregistré le programme, l'utilisateur effectue les généralisations en cliquant sur les icônes représentant les objets du programme. Une boîte de dialogue apparaît alors grâce à laquelle l'utilisateur va expliquer pourquoi il a sélectionné un objet donné. Ainsi, dans la figure 2.1, l'utilisateur a ouvert une boîte de dialogue sur le document "Treaty". La boîte de dialogue permet à l'utilisateur de spécifier ses intentions, par là même effectuant une généralisation. Dans cet exemple, l'utilisateur explique au système qu'il a sélectionné ce document car c'est *le plus récent* des documents *ayant pour nom "Treaty"*. L'utilisateur aurait également pu enseigner au système de demander à l'utilisateur de spécifier l'objet à chaque fois que le programme est exécuté ou lui dire de sélectionner l'objet en se basant sur sa position dans la fenêtre. Pour chaque type d'objet présent dans le système, Smallstar propose un ensemble de descripteurs de données présentés à l'utilisateur dans une boîte de dialogue.

Invocation et exécution du programme

Pour invoquer le programme ainsi créé, l'utilisateur sélectionne l'option "Run" dans la fenêtre associée au programme, ou dans le menu attaché à l'icône ou au bouton représentant le programme. Une originalité de Smallstar est de pouvoir exécuter le programme pas à pas afin de suivre son fonctionnement. Dans le cas où une erreur se produit, Smallstar affiche un message d'erreur et indique l'instruction fautive dans la fenêtre représentant le programme.

Visualisation du programme

Au fur et à mesure que les commandes sont enregistrées, une description textuelle agrémentée d'icônes apparaît dans la fenêtre programme (partie supérieure de la figure 2.1). Les icônes indiquent le type d'objet manipulé (dossier, document, sélection, tableau etc.). Le langage utilisé est similaire à CUSP : la partie gauche est un verbe décrivant la commande et la partie droite est une phrase représentant ses arguments.

L'utilisateur ne peut par contre pas programmer en utilisant ce langage textuel : ce langage n'est utilisé que pour la visualisation ce qui est dommage dans le cas où l'utilisateur est déjà familier avec CUSP.

3.1.3 Inférence et implantation

Smallstar n'effectue aucune inférence. Au lieu de cela, comme nous l'avons vu dans la section portant sur la création d'un programme, c'est l'utilisateur qui spécifie les structures de contrôle et les descripteurs de données une fois le programme enregistré, par là même effectuant les généralisations nécessaires. Une heuristique présente dans le système consiste à proposer par défaut les descripteurs de données les plus plausibles (ainsi pour un fichier, le nom est le facteur de choix par défaut). Lorsque Smallstar exécute un programme, il utilise les descripteurs de données pour trouver les objets dont le programme a besoin. La première fois qu'un descripteur de données est rencontré, Smallstar cherche les objets correspondant à cette description. Si plus d'un objet correspondent à la description, l'un d'eux est choisi arbitrairement. Lorsque Smallstar rencontre à nouveau le même descripteur de données il saute l'étape de recherche et utilise l'objet déjà trouvé. Ainsi, quand un même objet apparaît plus d'une fois dans un programme, toutes les références à cet objet utilisent le même descripteur de données ce qui constitue une heuristique importante. Il est cependant possible de changer manuellement les descripteurs de données pour chaque référence faite à un objet de façon à contrer cette heuristique.

Smallstar enregistre les commandes sous forme de haut niveau et non pas comme des opérations souris. Une commande possède un nom et un ensemble d'arguments. Une commande enregistrée peut impliquer plusieurs actions de la part de l'utilisateur. Ainsi dans la seconde ligne du programme de la figure 2.1, l'utilisateur a tout d'abord sélectionné le document "Treaty" puis l'a placé sur le bureau. Le programme est enregistré comme un arbre dont les feuilles sont les commandes de l'utilisateur. Cet arbre est initialement une simple liste, mais après ajout des structures de contrôle il devient un véritable arbre. Smallstar a été implanté en Smalltalk 80 sur une machine Xerox Dorado.

3.2 Metamouse — Maulsby 1988

3.2.1 Domaine d'application et utilisateurs ciblés

Metamouse [Maulsby 89a,b] est un logiciel de dessin du type MacDraw [Cutter 87], programmable par démonstration. Un utilisateur sans connaissance préalable de la programmation peut étendre Metamouse de façon à automatiser certaines tâches répétitives. Il gagne alors en vitesse de travail et en précision. Ce logiciel de dessin possède un nombre restreint de commandes qui permettent de créer, déplacer et effacer des rectangles et des droites. Toute sa puissance réside dans le fait qu'il est programmable.

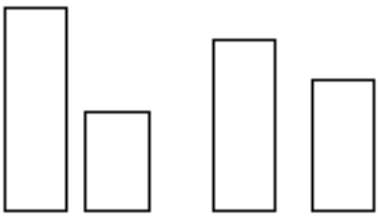
Metamouse utilise la métaphore de l'apprenti et permet à l'utilisateur de créer de nouvelles commandes en utilisant des outils de construction comme c'est traditionnellement le cas en dessin. Un objet graphique ayant la forme d'une tortue appelée Basil (figure 2.2) correspond à un curseur multifonctions, servant à concentrer l'attention du dessinateur sur les capacités d'inférences limitées du logiciel. Basil simule un élève auquel on apprend à dessiner : il prédit les actions suivantes lorsqu'il détecte des répétitions, il demande à l'utilisateur d'effectuer des constructions géométriques et de fournir des paramètres lorsque cela est nécessaire et il génère un programme.

3.2.2 Interaction

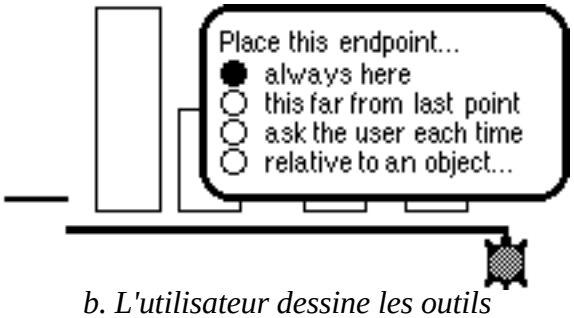
Création du programme

Pour spécifier une procédure, l'utilisateur passe en mode enregistrement puis accomplit la tâche sur un exemple, parfois en créant des constructions géométriques afin de mettre en évidence certaines relations entre objets et clarifier ainsi d'éventuelles ambiguïtés. Metamouse observe alors les actions de l'utilisateur et lorsqu'il détecte une répétition — c'est-à-dire que l'utilisateur a accompli une même séquence d'actions une seconde fois — Metamouse prédit les commandes à venir et offre d'effectuer les itérations suivantes. L'originalité de Metamouse est qu'il utilise la métaphore d'apprentissage : l'utilisateur joue le rôle du professeur enseignant la tâche à accomplir à une tortue représentée à l'écran (figure 2.2). Non seulement l'utilisateur montre la tâche mais, en plus, il peut mettre en valeur des relations entre objets à l'aide d'outils de constructions, corriger certaines inférences faites par le système ou répondre aux questions posées par Basil. Metamouse peut ainsi interagir avec l'utilisateur lorsqu'il a besoin d'informations supplémentaires (pour déterminer les intentions de l'utilisateur). L'utilisateur peut interrompre l'enregistrement pour le reprendre par la suite.

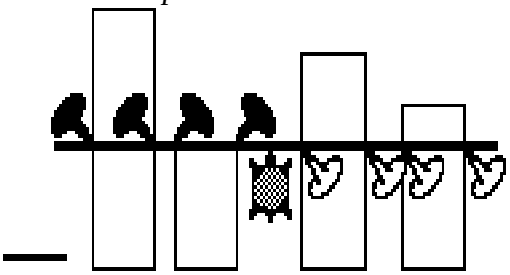
Dans la figure 2.2, l'utilisateur souhaite enseigner à Basil comment trier des rectangles par hauteur croissante. Pour ce faire, il commence par créer deux objets de construction — l'espaceur et la règle — qui seront utilisés par la suite pour exprimer des relations entre objets (a,b). L'espaceur est un segment qui sera utilisé pour déterminer l'espace entre les rectangles dans la configuration finale et la règle est un second segment qui sera utilisé pour désigner chacune des boîtes par hauteur croissante. A l'étape suivante (c), l'utilisateur amène la règle de telle sorte qu'elle vienne toucher le sommet du plus petit rectangle. Apparaissent alors des punaises symbolisant les points d'intersection entre la règle et les autres objets. Les punaises noires expriment les points de contact jugés importants par Basil comme les coins d'une boîte et les milieux et fin de segments. Les punaises blanches expriment les points de contact jugés moins importants. L'utilisateur peut changer lui-même la couleur d'une punaise en cliquant dessus de façon à corriger l'inférence du système. Ainsi (d), il change l'état des punaises de façon à ce que seules les punaises touchant le plus petit rectangle soient noires. L'utilisateur amène ensuite le rectangle sur l'espaceur (e) et seul le point de contact est associé à une punaise noire. Il déplace ensuite l'espaceur de façon à ce qu'il touche le coté droit de la boîte (f). L'utilisateur prend ensuite la règle : c'est une étape



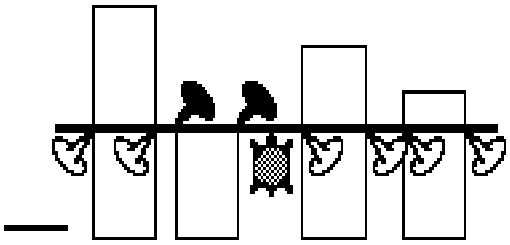
a. Disposition initiale



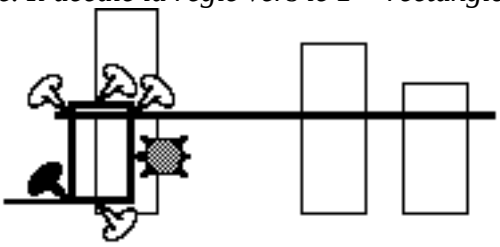
b. L'utilisateur dessine les outils



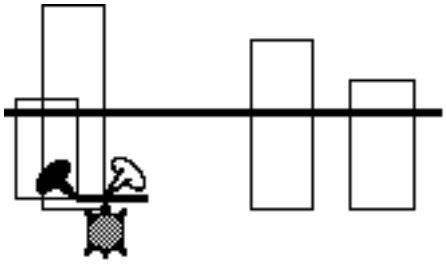
c. Il décale la règle vers le 1er rectangle



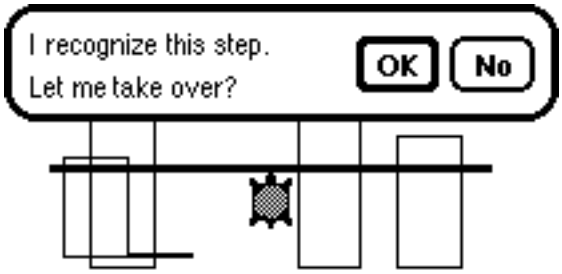
d. Il change les punaises de gauche



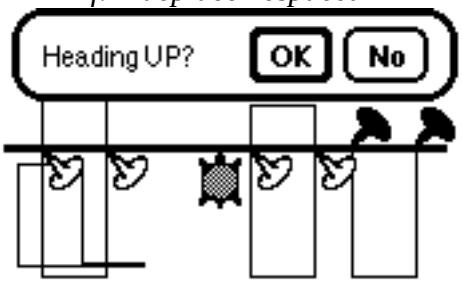
e. Il déplace le rectangle vers l'espaceur



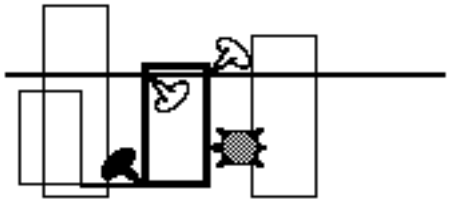
f. Il déplace l'espaceur



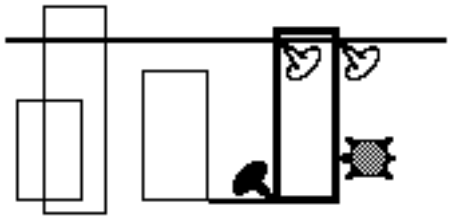
g. Il répète l'action que Basil a prédite



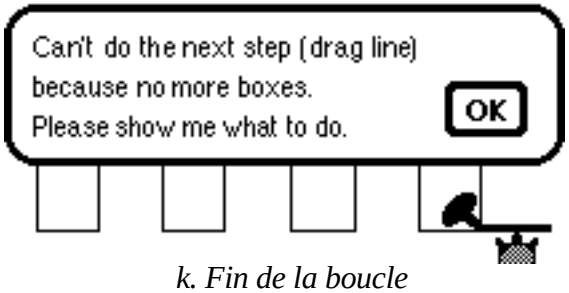
h. Basil déplace la règle vers le haut



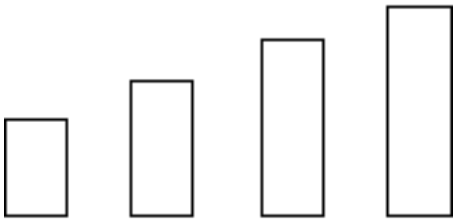
i Basil déplace le rectangle vers l'espaceur



j. Basil traite le 3ème rectangle



k. Fin de la boucle



l. Résultat final

Figure 2.2 : Metamouse : enseigner comment trier des rectangles par hauteur croissante.

que Basil reconnaît et demande s'il peut finir. Basil amène la règle vers la boîte suivante par taille croissante (g) et demande confirmation à l'utilisateur (h). Basil amène ensuite la boîte vers l'espaceur (i). Basil effectue la tâche pour les boîtes suivantes jusqu'à la fin de la boucle (j, k, l).

Invocation et exécution du programme

Dès que Metamouse détecte une répétition il offre à l'utilisateur la possibilité de finir la tâche à sa place (figure 2.2.g). Une fois le programme ainsi créé, l'utilisateur peut l'appeler à nouveau en utilisant le menu "Tasks".

Comme expliqué dans la section précédente, la phase d'exécution est intimement mêlée à celle de création et de généralisation. L'utilisateur peut annuler les commandes et le programme peut demander des arguments à l'utilisateur pour mener la tâche à bien.

Visualisation du programme

Metamouse n'offre pas de visualisation du programme créé à proprement parler. Au lieu de cela il présente l'étape prédite en déplaçant la tortue. Il semble que le seul moyen de savoir ce que fait un programme donné est de l'exécuter ce qui constitue une faiblesse du système. Comment peut-on alors s'assurer que le programme fonctionne dans tous les cas ?

3.2.3 Inférence et implantation

Le système d'apprentissage induit des relations binaires entre les éléments de construction. Le savoir de Basil est très limité : il connaît deux objets primitifs (le rectangle et la ligne) et sait traiter le redimensionnement, la sélection et le déplacement d'un objet. L'inférence est utilisée pour identifier les contraintes géométriques dans les commandes d'édition, pour déterminer les boucles et les branchements et pour différencier les variables des constantes.

Pour chaque commande accomplie par l'utilisateur, Metamouse enregistre les paramètres de la commande ainsi qu'un ensemble d'arguments représentant le contexte dans lequel la commande est exécutée. Ces arguments sont :

- l'ampleur et la direction du mouvement ;
- les événements tactiles (début et fin de contact avec d'autres objets) ;
- la nature des objets touchés ;
- l'emplacement des segments les composant ;

• les relations spatiales établies par les événements tactiles.

La mémoire de travail contient 1) la position et l'orientation de Basil, 2) les événements tactiles, 3) des triplets (précondition, action, postcondition), les conditions étant dérivées de 1 et 2, 4) le graphe reliant ces triplets, représentant le programme en cours de création, 5) l'identité des objets créés et transformés pendant la trace.

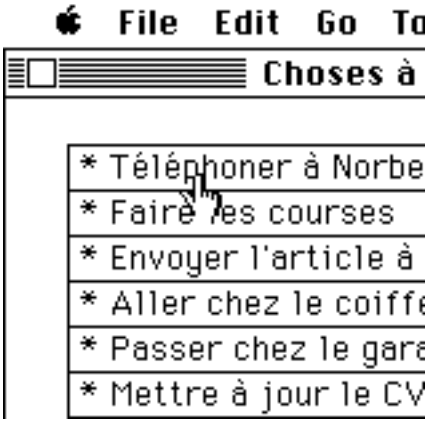
Durant la démonstration de l'algorithme, Metamouse anticipe les commandes, observe et généralise les pré et post conditions tactiles gouvernant les actions. Quand il prédit une action, Basil doit instancier les variables d'objet et de position. Ceci est réalisé en utilisant un résolveur de contraintes, qui génère le vecteur de translation à appliquer à Basil ainsi que les "poignées" à saisir. Si une prédiction échoue après l'application d'un vecteur, d'autres vecteurs sont essayés.

Metamouse a été initialement développé en Lisp sur un Macintosh II mais à cause des temps de réponse trop lents (essentiellement provoqués par le solveur de contraintes) il a du être porté en C++ sur station de travail X-Window. Metamouse est l'un des rares systèmes capables d'inférer des conditions.

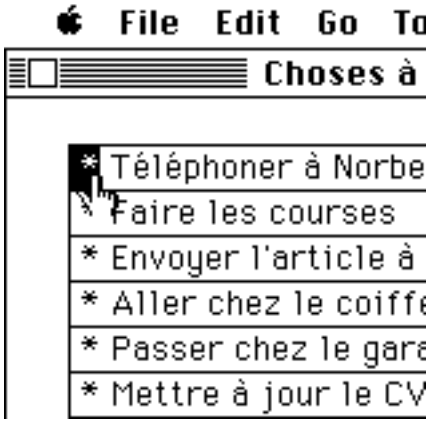
3.3 Eager — Cypher 1990

3.3.1 Domaine d'application et utilisateurs ciblés

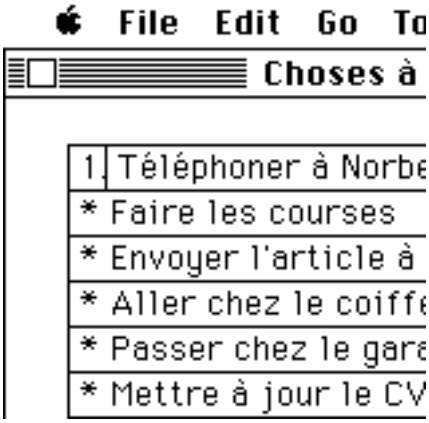
Eager automatise les tâches répétitives dans le cadre de l'environnement auteur HyperCard. Il s'agit d'un système de programmation par démonstration qui s'adresse à la tranche des utilisateurs d'HyperCard la moins expérimentée : absolument aucune connaissance préalable de la programmation n'est nécessaire. Eager répond également au fait que l'utilisateur ne réalise pas toujours immédiatement qu'il peut créer un programme pour accomplir la tâche qui l'intéresse. Souvent l'utilisateur commence sa tâche et au bout de quelques étapes s'aperçoit qu'il aurait pu l'automatiser. Dans la plupart des systèmes de PPD, l'utilisateur doit alors interrompre sa tâche, passer en mode enregistrement, effectuer les étapes nécessaires, stopper l'enregistrement puis invoquer le programme ainsi créé. Ce changement de mode interrompt le flot de l'interaction et peut déranger le novice. Eager adopte une approche différente car il observe en permanence le comportement de l'utilisateur pour essayer de prédire ses prochaines commandes. Eager peut alors proposer de terminer une tâche sans même que l'utilisateur y ait pensé.



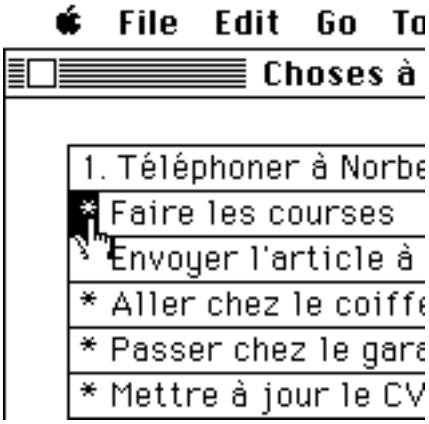
a. Les lignes à modifier.



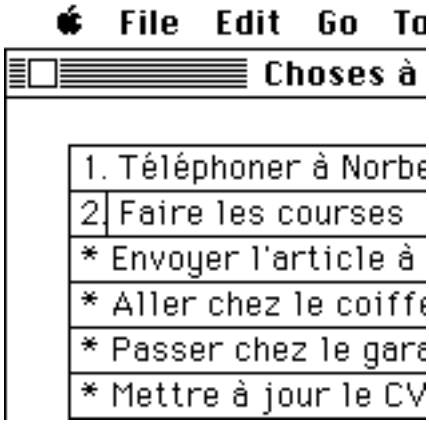
b. Sélection du premier astérisque.



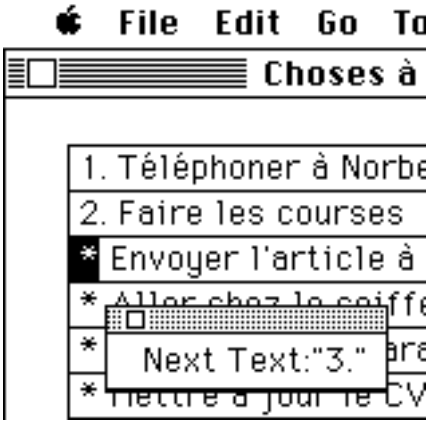
c. Remplacement par "1".



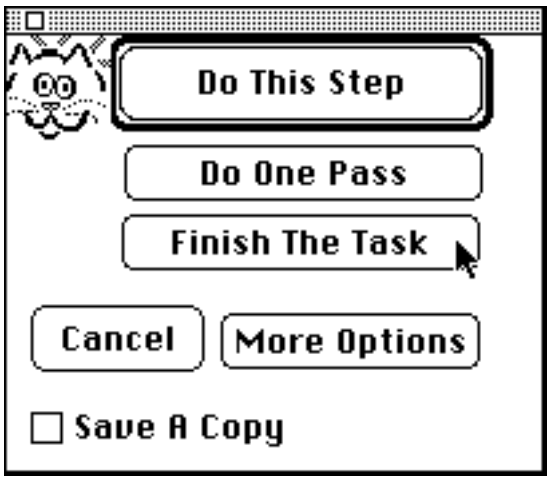
d. Sélection du second astérisque.



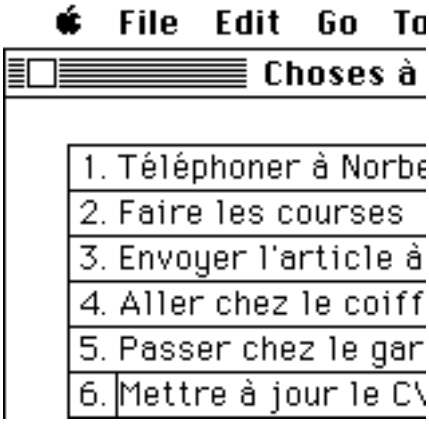
e. Remplacement par "2".



f. Sélection du 3ème astérisque. Eager prédit la commande suivante



g. L'utilisateur demande à Eager de terminer la tâche pour lui.



h. Le résultat final.

Figue 2.3 : Eager remplace les astérisques par des numéros.

3.3.2 Interaction

Création du programme

L'utilisateur ne fait rien de particulier pour créer un programme. Au lieu de cela, Eager enregistre constamment toutes les commandes de l'utilisateur dans l'espoir de trouver des répétitions et automatiser celles-ci. Lorsqu'Eager pense avoir détecté une boucle, il *anticipe la commande suivante sans l'exécuter*. Pour cela il sélectionne d'une couleur verte l'option dans un menu, affiche en vert le texte qu'il pense que l'utilisateur va taper etc. Si l'inférence d'Eager est correcte, la prochaine action que l'utilisateur fera sera déjà présélectionnée en vert. Si Eager se trompe, c'est-à-dire que la commande effectuée par l'utilisateur n'est pas celle à laquelle il s'attendait, cette commande est considérée comme un contre exemple et le programme est révisé en conséquence.

Invocation et exécution du programme

Une fois que l'utilisateur fait confiance à Eager et pense qu'il va mener à bien la tâche, il demande à Eager de finir la tâche pour lui en cliquant sur une icône particulière. L'utilisateur a le choix d'exécuter le programme d'un seul coup, de faire du pas à pas ou encore d'accomplir un certain nombre d'itérations. Avant d'exécuter un programme, Eager sauvegarde la pile HyperCard courante de telle sorte que l'utilisateur ne perde pas tout son travail en cas d'erreur.

A titre d'exemple, la figure 2.3 montre comment Eager aide l'utilisateur à remplacer tous les astérisques présents en début de ligne par des numéros. Pour ce faire, l'utilisateur sélectionne l'astérisque de la première ligne et le remplace par le chiffre 1 (figure 2.3.a-c). Il fait de même pour la seconde ligne mais cette fois remplace l'astérisque par le chiffre 2 (figure 2.3.d-e). Lorsqu'il sélectionne l'astérisque de la troisième ligne Eager prédit alors que l'utilisateur va taper le chiffre 3 (2.3.f). A ce moment l'utilisateur décide de faire confiance à Eager et lui demande de finir la tâche pour lui (2.3.g-h).

Visualisation du programme

Le programme généré par Eager est un script Hypertalk que l'utilisateur peut consulter s'il le souhaite (figure 2.4). L'autre forme de visualisation est l'anticipation des commandes utilisateur.

3.3.3 Inférence et implantation

Ce système — écrit en LISP avant d'être porté en C++ pour des raisons d'efficacité — est une application qui tourne en tâche de fond sur Macintosh et qui reçoit des commandes de haut niveau en provenance d'une version modifiée d'HyperCard. La tâche principale d'Eager est de détecter des boucles dans l'historique de l'utilisateur : ainsi, à

chaque fois qu'une nouvelle commande est reçue, Eager cherche dans l'historique une commande similaire ; deux commandes sont considérées similaires si elles sont de même type (par exemple `copier texte`) et si les données sur lesquelles elles portent partagent des traits communs (par exemple Mardi et Mercredi, bouton n°3 et bouton n°4, dernier mot de la ligne n°18 et dernier mot de la ligne n°19 etc.). La détection de trait commun se fait comme suit :

- Pour les données numériques, Eager cherche les constantes, les entiers consécutifs et les suites arithmétiques avec tolérance (ainsi à la série 50, 72, 91 correspond la suite 20*i+30±2). Des nombres apparaissent dans les numéros de carte, les coordonnées des objets, la position d'un mot dans une ligne etc.

- En ce qui concerne les données textuelles, Eager décompose les chaînes de caractères en sous-chaînes, en se basant sur les séparateurs rencontrés. Il recherche les constantes, l'ordre numérique ou alphabétique, le passage de majuscule en minuscule, les séquences connues comme les jours de la semaine, les mois et les chiffres romains. On rencontre des informations textuelles dans les passages de texte sélectionnés, les chemins d'accès aux fichiers, le nom des cartes, des boutons, des champs etc. De plus, lors de l'édition d'un texte, il met en correspondance le nouveau texte avec l'ancien, mais aussi avec le nouveau texte de l'itération précédente.

Quand deux commandes sont jugées similaires et qu'Eager n'a pas encore trouvé de boucle, il suppose qu'une boucle commence. Si les commandes suivantes de l'utilisateur confirment bien qu'il y a une boucle (c'est-à-dire qu'au moins deux itérations ont été effectuées) Eager anticipe alors les prochaines commandes en se basant sur le programme qu'il a généralisé. Lorsqu'un seuil de confiance est dépassé (c'est-à-dire qu'au moins trois itérations ont été effectuées), Eager affiche une icône particulière que l'utilisateur peut sélectionner pour laisser le soin a Eager de terminer sa tâche.

Si Eager peut détecter des répétitions dans la trace de l'utilisateur, c'est parce qu'il manipule des commandes de haut niveau. Il possède de plus une connaissance du domaine abordé (les cartes font partie de piles, l'abscisse d'un objet est de type "coordonnées d'écran" ainsi Eager - lors de la recherche de similarités - accorde une tolérance de 4 pixels pour tenir compte des inexactitudes de l'utilisateur) et chaque commande est enregistrée avec de l'information contextuelle.

Les séquences de commandes de l'utilisateur sont regroupées pour former des commandes de plus haut niveau : ce sont ces commandes qui sont enregistrées puis généralisées. Par exemple la séquence de commandes utilisateur :

```
sélectionner texte de (11,20) à (13,22)
couper texte
coller texte en (21,1)
```

Figure 2.4 : Script Hypertalk généré par Eager.

sera transformée en :

```
déplacer texte de (11,20) à (13,22) vers (21,1)
```

Cette méthode permet de tenir compte des fluctuations dans les actions de l'utilisateur. A une commande donnée peut en effet correspondre plusieurs séquences d'actions. D'autre part, le fait qu'Eager anticipe les commandes encourage l'utilisateur à effectuer les étapes d'une boucle de manière consistante.

Les limitations d'Eager concernent le fait qu'il n'infère pas de conditions, de boucles imbriquées, de répétitions temporelles et qu'il ne permet pas de sauver un programme créé, excluant ainsi la possibilité de faire un appel de procédure. Une autre limitation est que l'utilisateur ne peut pas créer lui-même un programme si celui-ci ne contient pas de boucle.

3.4 Chimera — Kurlander 1990

3.4.1 Domaine d'application et utilisateurs ciblés

Chimera [Kurlander 90] est un vaste système introduisant plusieurs innovations dans le domaine de l'édition graphique parmi lesquelles figurent : la création de procédures graphiques par l'exemple et par démonstration, le maintien de contraintes graphiques entre objets, un système de recherche d'objets graphiques sophistiqué, une représentation graphique et éditable de l'historique des commandes etc. Chimera se présente comme un éditeur multi-usage permettant d'éditer des dessins en deux dimensions, des interfaces graphiques ainsi que du texte. La partie qui nous intéresse ici est celle qui porte sur la création de macro par démonstration. Le système s'adresse à l'utilisateur débutant mais reste néanmoins difficile à maîtriser du fait du nombre d'outils disponibles et des nouveaux concepts mis en jeu.

3.4.2 Interaction

Création du programme

Chimera enregistre constamment les commandes de l'utilisateur dans un historique graphique. Lorsque l'utilisateur décide de créer une macro, l'utilisateur sélectionne dans l'historique les étapes à inclure puis identifie les arguments de la macro et effectue les généralisations nécessaires. La création de macro dans Chimera se fait donc en deux passes :

- Dans la première passe l'utilisateur démontre la tâche dans l'interface de l'application et ses commandes apparaissent dans l'historique sous forme graphique

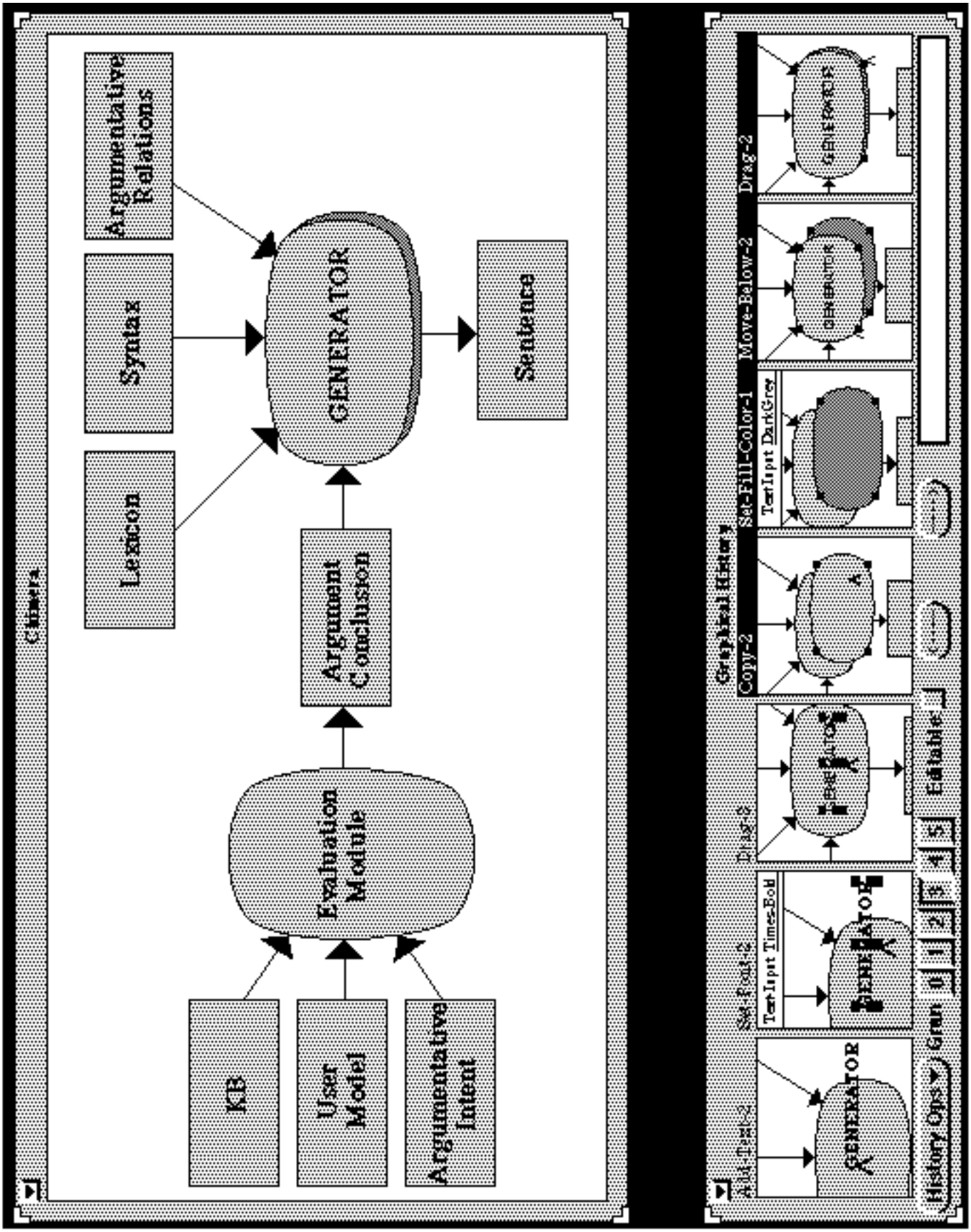


Figure 2.5 : Chimera : l'éditeur graphique et son historique graphique (les cases dont le nom apparaît en inversion vidéo sont sélectionnées pour définir la macro "Créer Ombre").

(partie inférieure de la figure 2.5). Ici l'interaction n'est pas différente de ce qui se passe normalement lorsqu'il utilise Chimera : il n'y a pas de commande spéciale à exécuter et il n'y a pas besoin d'utiliser les opérations démarrer ou stopper enregistrement. Comme expliqué section 3.3.1, souvent l'utilisateur ne pense à définir une macro que lorsqu'il a effectué au moins une fois les étapes : les commandes de la macro peuvent donc avoir déjà été démontrées et pas d'autre répétition n'est nécessaire.

- Lors de la seconde passe, l'utilisateur sélectionne tout d'abord dans l'historique les commandes qui doivent constituer la macro, détermine les arguments de celle-ci, fournit des informations complémentaires pour guider la généralisation puis sauve la macro. Dans cette passe, les commandes utilisées ne sont pas les mêmes que lors d'une interaction normale. Tout comme dans Smallstar l'utilisateur peut ôter ou insérer des commandes dans une macro.

Cette séparation des phases de démonstration et de généralisation est similaire à Smallstar, si ce n'est que Chimera est toujours en mode enregistrement.

Invocation et exécution du programme

L'invocation de la macro est effectuée par l'utilisateur qui sélectionne celle-ci dans un menu particulier. Lorsque la macro est exécutée, Chimera demande à l'utilisateur les arguments nécessaires. Chimera offre la possibilité d'annuler l'effet d'une macro en cas d'erreur.

Visualisation du programme

Une macro est représentée sous forme d'un historique graphique, *éditable* par l'utilisateur. Il s'agit là d'une contribution majeure de Chimera. Chimera utilise la notion d'historique graphique pour visualiser le contenu d'une macro, mais aussi pour visualiser les commandes accomplies et pour proposer l'annulation à l'utilisateur. Un tel historique supporte la création de macro : lors du processus de généralisation, l'utilisateur peut faire une référence temporelle à une commande passée. En cas d'erreur d'exécution, Chimera sélectionne le panneau de l'historique qui a créé l'erreur. Enfin l'historique peut être utilisé pour visualiser, modifier ou annuler des commandes passées. Chimera possède un ensemble d'heuristiques pour afficher l'historique dans les meilleures conditions surtout si la fenêtre de dessin comporte de nombreux objets. Certaines de ces heuristiques sont destinées à ôter du panneau les objets qui ne sont pas importants ainsi qu'à effectuer un zoom et un cadrage sur la situation courante. Un autre ensemble d'heuristiques permet de regrouper des panneaux ensemble afin d'aboutir à une représentation plus compacte de l'historique.

Un avantage majeur de ce type de représentation est qu'il est inutile de connaître un langage de programmation pour prendre connaissance d'une macro. Par contre une des

limitations de cette approche est qu'il est difficile de représenter graphiquement les généralisations (Chimara ne représente d'ailleurs pas les généralisations).

3.4.3 Inférence et implantation

Chimera ne fait pas de réelle inférence : c'est l'utilisateur qui effectue les généralisations nécessaires à la main. Comme dans Smallstar, l'utilisateur peut effectuer les généralisations en sélectionnant des options dans une boîte de dialogue associée à la commande. De façon similaire à Smallstar, Chimera possède des heuristiques qui prennent la forme de valeurs de défaut dans les boîtes de dialogue. Chimera ne permet pas de créer de boucles ou des conditions mais autorise l'appel de procédure. Chimera est écrit en Lisp, C et Postscript sur une station de travail Sun Sparc.

3.5 Mondrian — Lieberman 1991

3.5.1 Domaine d'application et utilisateurs ciblés

Mondrian [Lieberman 93] est un éditeur graphique orienté objet avec lequel l'utilisateur peut définir de nouvelles commandes par démonstration. Cet éditeur est très simple et offre une représentation graphique originale des commandes sous forme de *dominos :* le côté gauche d'un domino est une miniature de l'écran *avant exécution* de la commande, tandis que le côté droit montre l'écran *après exécution* (partie gauche de la figure 2.6). Parmi les commandes proposées on trouve la sélection, le déplacement et le groupement d'objets, la création de rectangles, le choix de couleurs etc. Ce logiciel s'adresse aux personnes accomplissant des tâches graphiques et désireuses de créer de nouvelles commandes sans pour autant posséder des notions de programmation.

3.5.2 Interaction

Création d'un programme

La phase de création débute par la *sélection* des objets graphiques qui constitueront les *arguments* du programme. Ensuite, l'utilisateur choisit le domino intitulé "Nouvelle Commande" puis fournit le nom de la commande qu'il souhaite définir. Un nouveau domino apparaît alors dans la fenêtre, avec une miniature de l'écran courant en partie gauche et un point d'interrogation en partie droite. L'utilisateur démontre ensuite les étapes nécessaires en faisant appel soit aux commandes de l'éditeur soit à des commandes qu'il a précédemment définies (y compris celle qu'il est en train de créer, autorisant ainsi non seulement l'*appel de procédure* mais aussi la *récursivité*). Enfin, il sauvegarde la commande ainsi définie en cliquant à nouveau sur le domino "Nouvelle Commande". Le côté droit du nouveau domino est mis à jour en conséquence : le point d'interrogation est remplacé par une miniature de l'écran représentant l'état final.

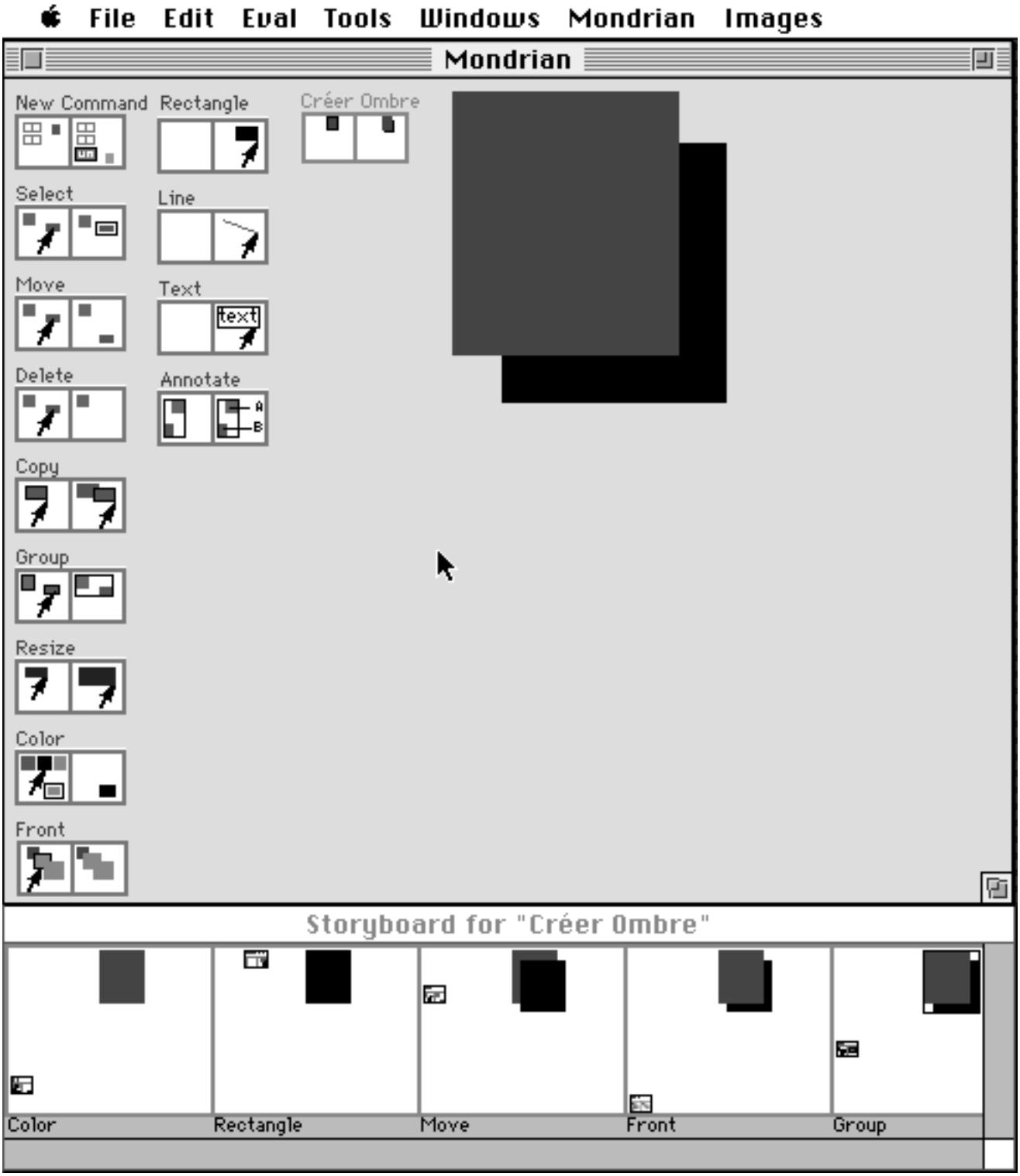


Figure 2.6 : Mondrian : définition de la macro "Créer Ombre".

Mondrian confirme les commandes de l'utilisateur sous forme d'une *voix synthétique* de façon à ne pas interférer avec la visualisation. Dans la figure 2.6, l'utilisateur vient de définir le domino "Créer Ombre" qui ajoute une ombre noire au rectangle sélectionné.

L'utilisation de la sélection est une solution très élégante au problème du passage d'arguments car elle ne brise pas la métaphore de l'éditeur. Il n'est pas mentionné si la sélection peut aussi être utilisée pour passer des valeurs de retour ce qui paraît également important.

Invocation et exécution d'un programme

Pour invoquer un programme, l'utilisateur commence par sélectionner les objets qui doivent être les arguments, puis clique sur le domino représentant le programme à exécuter. Il n'y a aucune différence entre invoquer une commande prédéfinie et une commande créée par l'utilisateur. Ainsi, pour invoquer la commande "Créer Ombre" de la figure 2.6, l'utilisateur sélectionne le rectangle qui l'intéresse et clique ensuite sur le domino "Créer Ombre". Mondrian exécute alors les commandes associées au domino "Créer Ombre" :

1. sélectionner la couleur noire (couleur de l'ombre) ;
2. créer un rectangle (noir) ayant les mêmes coordonnées que le rectangle sélectionné (ce sera l'ombre) ;
3. décaler le nouveau rectangle (pour créer un effet d'ombrage) ;
4. amener le rectangle original au premier plan ;
5. grouper les deux rectangles pour former un objet unique.

Mondrian n'offre pas de contrôle de l'exécution de la macro.

Visualisation d'un programme

Lorsque l'on souhaite visualiser un programme, il suffit de cliquer sur le domino correspondant, en maintenant une touche particulière du clavier appuyée. Apparaît alors une nouvelle fenêtre contenant une représentation graphique du programme à la manière de Chimera (partie inférieure de la figure 2.6) : le programme est représenté comme une succession de cases montrant le résultat de chaque commande, donnant le nom de celle-ci ainsi qu'une description textuelle (non représentée ici). Tout comme dans Chimera, il arrive que pour en améliorer la clarté, une case représente une *abstraction* de l'état dans lequel se trouve l'écran. Ainsi, lorsque l'écran possède trop d'objets, seuls ceux qui sont liés à la commande sont affichés. D'autre part, Mondrian élargit le curseur

```
(DEFUN CREER-OMBRE   (INTERACTOR SELECTION)
   (COLOR-COMMAND '(0 0 0)
                  INTERACTOR
   )
   (RECTANGLE-COMMAND (LEFT-TOP (NTH 0 SELECTION))
                      (BOTTOM-RIGHT (NTH 0 SELECTION))
                      INTERACTOR
                      'RECTANGLE-1
   )
   (MOVE-COMMAND '(10 10)
                 INTERACTOR
   )
   (GROUP-COMMAND (LIST (NTH 0 SELECTION)
                        (FIND-OBJECT-NAMED
                            INTERACTOR 'RECTANGLE-1)
                  )
                  INTERACTOR
                  'GROUP-1
   )
)
```

Figure 2.7 : Code Lisp généré par Mondrian pour la macro "Créer Ombre".

pour insister sur sa position ou, au lieu d'offrir une représentation détaillée d'un objet complexe, n'affiche que son contour. Une différence avec Chimera, est que la vue graphique du programme ne peut être manipulée en aucune façon. L'utilisateur familier avec le langage Lisp peut aussi visualiser le code Lisp correspondant au programme créé (figure 2.7).

3.5.3 Inférence et implantation

Comme nous venons de l'aborder, Mondrian implante un certain nombre d'heuristiques afin d'améliorer la lisibilité des miniatures d'écran. Il possède également un algorithme d'apprentissage qui garde trace des relations géométriques entre les objets ainsi que des relations de dépendance entre différentes commandes. Ainsi, au lieu d'enregistrer la seconde commande du programme "Créer Ombre" comme `(créer-rectangle 10, 110, 80, 180)`, Mondrian repère l'objet par rapport aux autres objets présents *dans la sélection*. La commande sera enregistrée comme : `(créer-rectangle allant du coin Nord-Ouest du premier élément de la sélection au coin sud-est du premier élément de la sélection)` (cf. figure 2.7). Ceci se base sur l'hypothèse que les étapes d'une procédure doivent dépendre des arguments de la procédure. Mondrian connaît une liste de relations privilégiées comme gauche, droite, centre etc. Lorsque des contacts interviennent avec des objets situés en dehors de la

sélection, des valeurs absolues sont utilisées, correspondant à des constantes dans un langage de programmation.

Mondrian ne détecte ni les boucles ni les conditions : une commande n'est qu'une séquence linéaire d'instructions ce qui limite sa force d'expression. Ainsi, il n'est pas possible d'ajouter automatiquement une ombre à tous les rectangles sélectionnés.

De façon interne, une commande est représentée comme un objet dont les variables donnent le nom de la commande, décrivent l'image du domino, pointent sur une fonction à appeler lorsque la commande doit être exécutée et pointent sur une liste décrivant la classe des arguments de la commande et l'interaction associée (c.-à-d. drag and drop, sélection).

A la macro "Créer Ombre" de la figure 2.6, correspond la fonction Lisp CREER-OMBRE (figure 2.7) qui prend deux arguments, le premier étant l'éditeur graphique et le second la liste des éléments sélectionnés. Mondrian nomme automatiquement les objets créés au cours de la procédure pour permettre des références ultérieures. Mondrian est écrit en Lisp et fonctionne sur un Macintosh.

4 Résumé

Au vu des différents systèmes que nous venons d'étudier il est clair qu'ils présentent à la fois des caractéristiques communes et des différences certaines. Nous reprenons maintenant les principaux critères énoncés dans la deuxième section et pour chacun d'eux nous comparons les différentes approches prises par les systèmes de PPD étudiés. Cette comparaison guidera la conception de notre squelette d'application. Le tableau 2.1 récapitule les caractéristiques de ces cinq systèmes de Programmation Par Démonstration.

4.1 Création du programme

Plusieurs méthodes très différentes sont utilisés pour créer des programmes par démonstration. La plus répandue consiste à créer un programme de manière similaire à une macro, c'est-à-dire déclencher l'enregistrement des commandes, démontrer la tâche puis stopper l'enregistrement (Smallstar, Mondrian, Metamouse) : cette méthode est bien adaptée si l'utilisateur est déjà familier avec le concept de macro et s'il a une tâche précise à automatiser. Une seconde méthode consiste à enregistrer en permanence les commandes de l'utilisateur et à proposer de créer un programme lorsqu'une répétition est rencontrée (Eager) : cette approche est bien adaptée aux utilisateurs débutants dans la mesure où (1) elle ne requiert aucune connaissance de la programmation et (2) l'utilisateur n'a pas toujours conscience ou réalise parfois trop tard qu'il peut créer un programme pour automatiser la tâche qu'il est en train d'accomplir. Enfin une troisième

	Smallstar	**Metamouse**	**Eager**	**Chimera**	**Mondrian**
Domaine	traitement de documents	édition graphique	HyperCard	édition graphique	édition graphique
Création d'une macro	manuelle : démarre/stoppe enregistrement puis généralisation	manuelle : démarre/stoppe enregistrement et *interaction avec Basil*	automatique : *anticipation des commandes*	manuelle : *sélection d'étapes dans l'historique* puis généralisation	manuelle : démarre/stoppe enregistrement *la sélection représente les arguments*
Généralisation d'une macro	utilisateur *après* enregistrement en créant des *descripteurs de données*	système et utilisateur *pendant* enregistrement	système (et utilisateur lorsqu'il accepte ou refuse une commande anticipée)	utilisateur *après* enregistrement	système : une *voix synthétique* énonce les généralisations
Invocation d'une macro	manuelle (clique sur une icône ou un bouton)	automatique pendant la création, manuelle après	manuelle *après suggestion du système*	manuelle (choix dans un menu)	manuelle (clique sur un domino)
Exécution	pas à pas	annulation	pas à pas ou par itération	*annulation*	—
Visualisation d'une macro	texte et icônes	aucune	anticipation ou scripte Hypertalk	*historique graphique éditable*	*historique graphique* ou code Lisp
Structures de contrôle	boucles, conditions, variables	boucles, conditions, variables	boucles, variables	appel de procédure *avec arguments*, variables	appel de procédure *avec arguments*, variables
Inférence	non	oui (trouve les boucles et les conditions)	oui (trouve des boucles)	non	oui (variables et relations entre objets)
Connaissances	uniquement des défauts	contraintes graphiques	séquences	uniquement des défauts	relations géométriques
Interface hôte	Smalltalk	Mac puis X-Windows	Macintosh	Postscript et OpenWindows	Macintosh
Langage	Smalltalk	Lisp puis C++	Lisp puis C++	Lisp et C	Lisp
Plate-forme	Xerox Dorado	Mac puis Apollo	Macintosh	Sun Sparc	Macintosh

Tableau 2.1 : Comparaison des systèmes de Programmation Par Démonstration.

méthode adopte un comportement intermédiaire dans la mesure où elle enregistre toutes les commandes de l'utilisateur et permet à celui ci de créer une macro après coup en sélectionnant les commandes à inclure dans la macro (Chimera). Chacune de ces méthodes propose donc ses propres avantages et à l'heure actuelle il n'est pas clair de savoir quelle est la méthode la plus adaptée pour la PPD. Notre squelette d'application doit être capable de supporter ces trois méthodes puisqu'un même mécanisme d'inférence peut être utilisé pour ces trois approches.

Un autre point concerne la définition des arguments d'une procédure et le retour d'une valeur : l'utilisation de la sélection que fait Mondrian est très élégant dans la mesure où le flot de l'interaction reste très naturel. Il n'y a pas de commandes spécifiques à invoquer pour définir et passer les arguments d'une macro. Cette approche est par contre plus limité que Chimera puisque les arguments ne peuvent être que des objets sélectionnables.

4.2 Invocation du programme

Dans tous les systèmes sauf un (Eager), c'est à l'utilisateur qu'incombe la tâche d'invoquer une macro au travers de l'interface graphique. Dans notre squelette nous adopterons une approche conservatrice, c'est-à-dire que nous laisserons le soin à l'utilisateur d'invoquer une macro. Ceci n'exclue pas l'invocation automatique dans le futur.

4.3 Exécution du programme

La plupart des systèmes proposent d'interrompre l'exécution d'un programme, certains autorisent le pas à pas (Smallstar) et enfin d'autres permettent d'annuler les effets d'une macro (Chimera, Eager, Metamouse). Ce dernier point parait très important dans la mesure où trois types d'erreurs peuvent se présenter: (1) une macro ou une commande peut avoir été invoquée par mégarde, (2) le système peut s'être tromper dans la généralisation du programme, (3) l'utilisateur peut avoir créé un programme erroné. Il est donc primordial de fournir un outil de correction d'erreur robuste. L'approche de Chimera qui intègre le mécanisme de création de programme à l'historique de commandes semble aller dans la bonne direction et être un bon modèle pour notre squelette.

4.4 Visualisation du programme

Deux approches principales ont été prises pour la visualisation d'un programme. La première consiste à représenter le programme sous une forme textuelle parfois agrémentée d'icônes : dans ce cas la macro est représenté dans le langage de programmation de l'application (CUSP pour Smallstar, Hypertalk pour Eager et Lisp

pour Mondrian). La représentation graphique des commandes parait être la plus adaptée même si il est difficile de représenter les généralisations; par contre il est difficile à un squelette d'application de supporter cette représentation de façon générique. En effet, la représentation graphique d'une commande est étroitement liée à l'interface du système ainsi qu'à la nature de la commande. Pour la première version du squelette nous représenterons les commandes sous forme textuelle par soucis de simplicité. Du fait de la nature modulaire et extensible du squelette, ajouter une telle représentation restera toutefois possible dans le futur.

4.5 Inférence

La majorité des systèmes enregistrent des commandes de haut niveau ainsi que le contexte associé. Certains systèmes possèdent de plus des connaissance ayant trait au domaine de l'application (Metamouse, Mondrian — connaissance de la géométrie) alors que d'autre détectent des répétitions dans la trace (Eager et la représentation des séquences — connaissance générales) et enfin certains ne possèdent aucune connaissances spécifiques (Chimera, Smallstar). La représentation des variables comme des descripteurs de donnée est une idée très intéressante car elle fourni un moyen de contrôle à l'utilisateur, par contre elle requiert une commande supplémentaire. Un autre moyen pour parvenir à un effet similaire serait de fournir une fonction de recherche d'objets qui se baserait sur les propriétés des objets : les utilisateurs sont déjà familiers avec de telles fonctions (par exemple le Système 7.5 du Macintosh possède une telle commande et le logiciel graphique Canvas 3.5 permet aussi de chercher des objets dans une fenêtre graphique).

4.6 Implantation

Tous les systèmes abordés ont été développé avec un langage orienté objet, que ce soit C++, Smalltalk ou CLOS (extension objet de Lisp). Ceci est du au fait que les systèmes de PPD sont difficiles à implanter et que toute aide dans la phase de développement est bienvenue. Un second avantage — uniquement proposé par les langages interprétés de haut niveau comme Hypertalk ou Smalltalk — est qu'ils fournissent un moyen immédiat d'encoder une macro de façon interne : la macro est alors un programme que le langage peut exécuter. Il faut cependant noter que deux systèmes — Metamouse et Eager — initialement écrits en Lisp ont du être portés en C++ pour des raisons d'efficacité et que Chimera est écrit en Lisp et C.

Chapitre 3

AIDE : un squelette pour les systèmes de programmation par démonstration

Les systèmes démonstrationnels sont difficiles à construire. Tous les programmes existants ont été implantés séparément et laborieusement "à la main". Des boîtes à outils et autres composants logiciels sont nécessaires pour les interfaces démonstrationnelles.

Brad Myers [Myers 92]

1 Introduction

Comme nous l'avons exposé dans le chapitre précédent, les systèmes de Programmation Par Démonstration sont des logiciels complexes, difficiles à implanter, et pourtant la plupart d'entre eux possèdent des éléments en commun parmi lesquels on trouve : une représentation des commandes utilisateur et du contexte associé, un historique des commandes, et un algorithme d'apprentissage symbolique opérant sur l'historique. Le projet AIDE [Piernot 93] vise à faciliter la création de tels systèmes en fournissant une couche logicielle écrite en Smalltalk sur laquelle les développeurs peuvent venir bâtir, avec un minimum d'effrot, des applications mettant en oeuvre des techniques démonstrationnelles.

Nous isolons tout d'abord les principaux critères auquel doit satisfaire un tel squelette d'application c'est-à-dire : être extensible, enregistrer les commandes utilisateur

à un niveau d'abstraction élevé, fournir un outil de correction d'erreur et structurer l'historique des commandes. Nous présentons ensuite l'architecture qui a été retenue ainsi que ses principaux composants : les *commandes*, les *macros* et le *gestionnaire de commandes*. Enfin, nous montrons comment nous avons étendu ce squelette afin de supporter le dernier critère (concernant la structuration de l'historique) qui n'avait pas été identifié initialement. Nous introduisons ainsi les **arbres de commandes** comme un modèle d'historique hiérarchique.

2 Principes de conception

2.1 Extensibilité

La première qualité que doit posséder un squelette d'aplication est bien sûr d'être extensible de façon à ce que le développeur puisse l'enrichir et l'adapter aux besoins de l'application qu'il développe. De plus, le squelette doit être capable d'évoluer avec l'apparition de nouvelles techniques de Programmation Par Démonstration. Nous avons retenu le langage Smalltalk (la version 1.1 de Digitalk puis par la suite SmalltalkAgents 1.2 de QKS) pour développer notre squelette car il s'agit d'un langage à objets dynamique facilement extensible et comportant un grand nombre de classes predéfinies. De plus, Smalltalk bénéficie d'un environnement de programmation très riche pour écrire, consulter et corriger le code.

2.2 Enregistrement de commandes de haut niveau

Lorsque l'utilisateur interagit avec l'interface d'une application, il génère une séquence d'opérations souris et clavier élémentaires, aussi appelées **commandes de bas niveau** (par exemple `déplacer-souris`, `presser-bouton-souris`, `relacher-bouton-souris`, `presser-touche-clavier`, `relacher-touche-clavier` etc.). Certains systèmes comme les enregistreurs de macros QuicKeys et Tempo II ou les interfaces prédictives Reactive Keyboard [Darragh 92] et Predictive Calculator [Witten 81] enregistrent les commandes de l'utilisateur à un tel niveau. Le problème avec cette approche réside dans le fait que les commandes de bas niveau véhiculent une information trop fine qui ne correspond pas à l'intention première de l'utilisateur, rendant ainsi difficile l'extraction d'information exploitable à un autre niveau, la visualisation de l'interaction passée ou encore la ré-exécution d'une commande dans un contexte différent (chapitre 1, section 4.3). Ainsi, l'action de sélectionner la poubelle sur le bureau est plus facile à rejouer et à comprendre quand elle est exprimée comme `sélectionner (Poubelle)` plutôt que `presser-bouton-souris (623, 383)`. Les historiques de bas niveau présentent néanmoins trois avantages de taille : ils sont faciles à adjoindre au système d'exploitation, ils ne réclament aucune modification des applications existantes et ils peuvent enregistrer les commandes utilisateur de façon *globale* (c'est-à-dire que l'historique peut contenir des commandes en provenance de différentes applications).

Potter a ainsi basé son système de Programmation Par Démonstration Triggers [Potter 93] uniquement sur des commandes de bas niveau.

Une meilleure approche consiste à enregistrer des *commandes de haut niveau* comme c'est le cas dans certains systèmes démonstrationnels (Smallstar, Chimera, Mondrian, Pursuit [Modugno 94], Mike [Olsen 88]), outils de correction d'erreur (Chimera, WeMet [Rhyne 92]), enregistreurs de macro (AppleScript, HP NewWave [Fuller 89]) et interfaces prédictives (Eager, Metamouse, Edward [Bos 92]). Ainsi dans l'architecture des AppleEvents [Apple 93a,b], lorsque l'utilisateur manipule une application, son interface envoie des commandes de haut niveau (appelées AppleEvents), via un composant du système d'exploitation (le gestionnaire d'AppleEvents), au corps de l'application pour exécution. Ce modèle est bien adapté à la programmation par démonstration étendue à tout le système car le gestionnaire de commandes reçoit les commandes de n'importe quelle application. Une commande de haut niveau (comme `dessiner-rectangle (18, 93, 345, 240)`) est une représentation plus riche qu'une simple série de commandes de bas niveau (ici : `déplacer-souris`, `presser-bouton-souris`, `déplacer-souris`, ..., `déplacer-souris`, `relacher-bouton-souris`). De plus, une même série de commandes de bas niveau peut avoir plusieurs interprétations suivant l'application qui la reçoit, conduisant ainsi à une ambiguïté. A titre d'exemple, une autre interprétation de la séquence de commandes de bas niveau précédente aurait pu être la commande `mettre-à-la-poubelle ("fichier.txt")`. Enfin, une commande de haut niveau ne se base pas seulement sur la position des objets à l'écran (ainsi, enregistrer simplement la commande de bas niveau consistant à cliquer sur le fichier "fichier.txt" fait que la commande ne peut être rejouée que si les coordonnées de l'icône représentant le fichier restent les mêmes), mais possède d'avantage de sémantique et peut donc être manipulée plus facilement par un algorithme d'apprentissage symbolique.

2.3 Fournir un outil de correction d'erreur

Les systèmes de programmation par démonstration permettent d'automatiser des tâches répétitives : du fait de cette puissance placée entre les mains de l'utilisateur, il est primordial de fournir un outil de correction d'erreur pour permettre d'annuler les dommages engendrés par un programme. En effet, l'exécution d'un programme peut avoir des conséquences dévastatrices car un programme peut affecter un nombre important d'éléments avant d'être interrompu. Annuler manuellement l'effet d'un programme peut donc être très coûteux. L'utilisateur peut vouloir annuler les effets d'un programme pour deux raisons. La première est qu'il peut avoir invoqué le programme par erreur. La seconde est que le programme peut être erroné : l'erreur peut avoir été introduite par mégarde soit par l'utilisateur, soit par le système lors de l'inférence les intentions de l'utilisateur. Il semble donc naturel et nécessaire de coupler le mécanisme d'historique avec celui de récupération d'erreur.

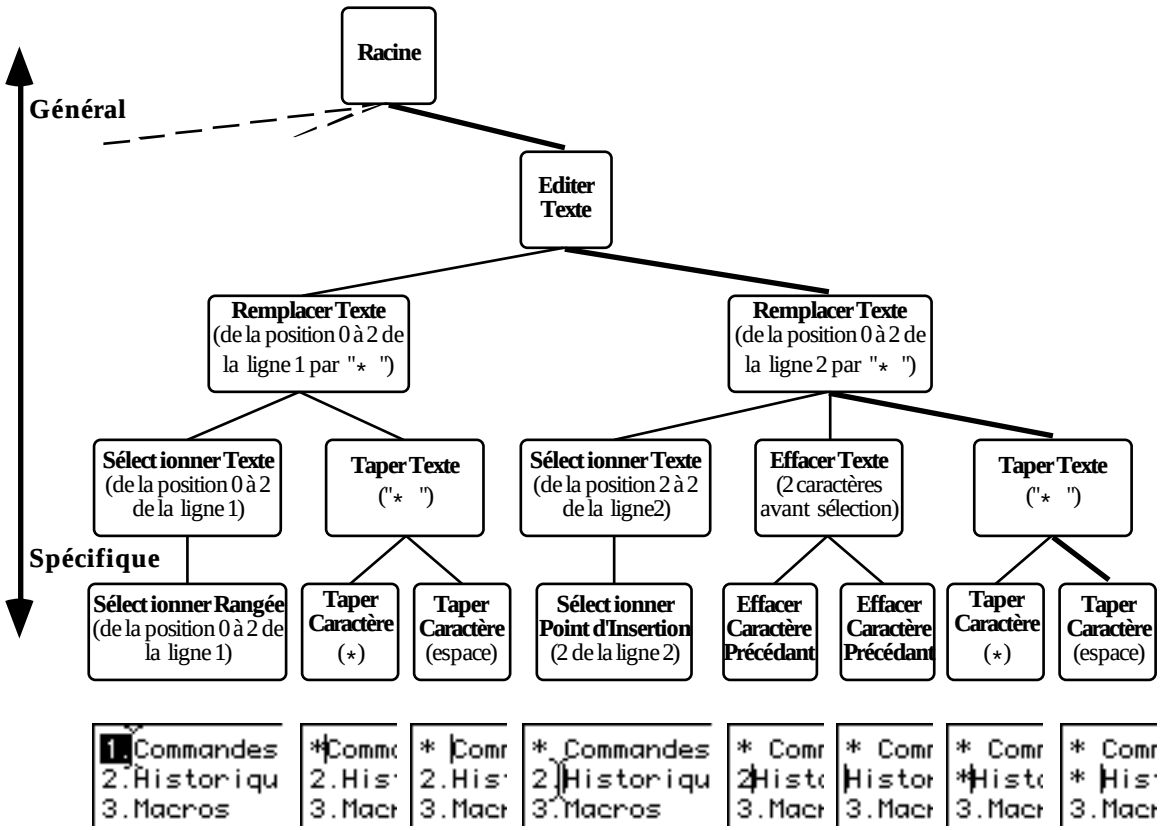


Figure 3.1 : Arbre des commandes pour la tâche "Remplacer Numéros par Astérisques".

2.4 Structuration de l'historique des commandes

Observant que très peu de systèmes (comme [Vitte 86]) ont essayé de représenter l'historique des commandes sous une forme plus structurée qu'une liste, Kosbie [Kosbie 93] a introduit la notion d'agrégats de commandes — commandes composées d'autres commandes — conduisant à une représentation arborescente de l'historique. Dans ce modèle, les commandes de bas niveau (par exemple `taper-caractère`) sont regroupées ensemble pour former une commande plus abstraite (par exemple `taper-texte`) qui, à son tour, peut être regroupée avec d'autres commandes et conduire à une commande encore plus abstraite (par exemple `remplacer-texte`) etc. De tels agrégats de commandes sont plus proches de la structure de la tâche de l'utilisateur qu'une série de commandes de bas niveau car ils correspondent à la *décomposition* d'une tâche en sous tâches : les commandes situées vers le sommet de l'arbre sont plus générales et représentent des tâches de haut niveau, tandis que les commandes au bas de l'arbre sont plus spécifiques et représentent des tâches de plus bas niveau. De plus, elles permettent à l'utilisateur ainsi qu'au système de choisir la granularité à laquelle ils veulent manipuler et visualiser l'historique. En conséquence, les historique hiérarchiques fournissent une bonne base pour l'annulation de commandes à plusieurs niveaux ; pour les interfaces démonstrationnelles (en enregistrant des commandes de haut niveau, la détection de similitudes est moins sensible aux variations d'accomplissements de la tâche) ; pour capturer les intentions de l'utilisateur (en conservant aussi les commandes de bas niveau

le système sait comment la commande de haut niveau a été accomplie : dans certains cas, une commande de bas niveau représente correctement l'intention de l'utilisateur) ; et pour une meilleur visualisation (en fournissant un affichage plus représentatif de la tâche accomplie).

De tels historiques offrent de plus une approche intermédiaire entre la Programmation Par Démonstration — qui enregistre toutes les commandes— et la Programmation Par l'Exemple — qui ne conserve que l'état initial et final — (voir chapitre 1, section 4.5).

Un exemple d'un tel historique hiérarchique pour un éditeur de texte est donné figure 3.1, le bas de la figure montrant des captures d'écran miniatures représentant les effets des commandes primitives sur l'interface. La tâche présentée dans cette exemple (remplacer numéros pas astérisques) est de remplacer les deux premiers caractères de la ligne 1 et 2 par "`*` ". Le sommet de l'arbre est la commande racine, un de ses enfants est `éditer-texte` qui consiste lui-même de deux sous-commandes `remplacer-texte`. Il faut noter que le parcours de l'arbre des commandes s'effectue classiquement de haut en bas et de gauche à droite représentant ainsi la séquentialité des commandes.

3 Architecture générale

3.1 Présentation

Notre architecture (figure 3.2) est semblable à celle des AppleEvents dans la mesure où une application développée avec AIDE est responsable de la transformation d'une série de commandes de bas niveau générées par l'utilisateur en une commande de haut niveau. Le *gestionnaire de commandes* enregistre la commande dans l'historique et demande à l'application de l'exécuter. Jusqu'à ce point, le flot de données est similaire à celui du modèle des AppleEvents. Mais dans le modèle des AppleEvents, la fonction du gestionnaire de commandes est d'assister les applications à résoudre des spécificateurs d'objets complexes (similaires aux descripteurs de données de Smallstar), tandis que dans notre cas, le gestionnaire de commande est chargé de la création d'un programme à partir de l'historique des commandes utilisateur. Ainsi, le dialogue entre une application développée avec AIDE et le gestionnaire de commande est différent selon que nous enregistrons ou exécutons une macro.

En *mode enregistrement* (c.-à-d. une macro est enregistrée), le gestionnaire de commandes demande à l'application d'enrichir les commandes de haut niveau reçues avec de l'information contextuelle. Ainsi, dans un traitement de texte le contexte associé à la commande `sélectionner-texte` pourrait contenir le texte sélectionné, sa position par rapport au début de la ligne, son style, etc. Ensuite, le gestionnaire de commandes recherche les boucles dans l'historique, effectuant les généralisations nécessaires.

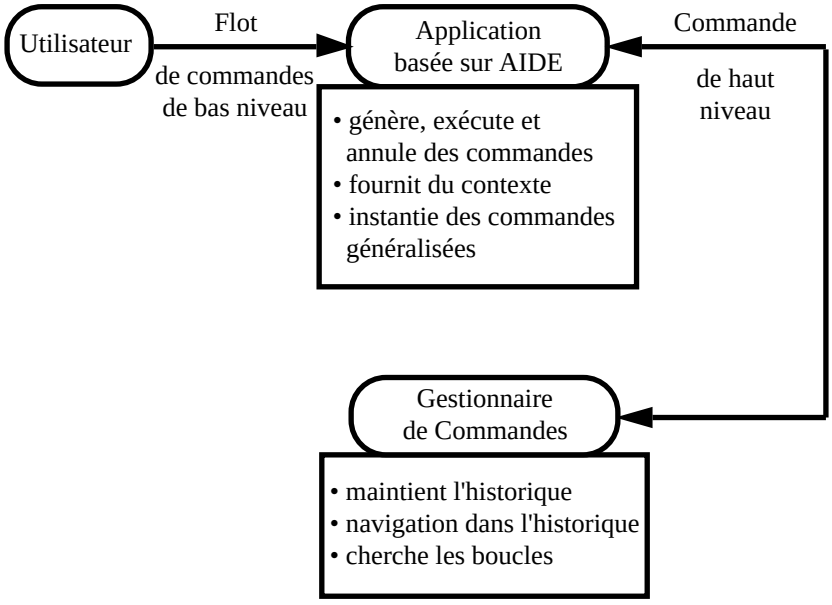


Figure 3.2 : Architecture d'AIDE.

En *mode exécution* (c.-à-d. une macro est exécutée), la macro obtient des commandes généralisées à partir de sa définition et demande à l'application d'instancier la commande (une commande dont les arguments sont normaux, obtenue à partir de la commande généralisée, et que l'application peut exécuter), l'enregistre et la passe à l'application pour son exécution.

Cette architecture supporte soit un historique global (un gestionnaire de commandes unique pour tout le système), soit un historique associé à chaque application (un gestionnaire de commandes pour chaque application). Le dialogue entre une application et le gestionnaire de commandes se fait par échange de commandes. Celles-ci, le gestionnaire de commandes, les macros, ainsi que le processus de généralisation sont ensuite décrits.

3.2 Commandes de haut niveau

Les commandes de haut niveau forment le coeur de notre système car elles fournissent une description sémantiquement riche des actions effectuées par l'utilisateur. Dans notre squelette, une telle commande est implantée par un objet Smalltalk — instance de la classe `Commande` — possédant un `nom` (qui peut être affiché par l'interface graphique du système), un `propriétaire` (application à qui appartient la commande) et un ensemble d'arguments incluant des *paramètres* (utilisés pour exécuter et annuler une commande) ainsi que de l'*information contextuelle* (information décrivant l'état de l'application au moment de l'exécution de la commande, utilisée lors de la généralisation). De plus, chaque commande fait référence à des méthodes spécifiques définies dans l'application pour l'exécution et l'annulation, pour la généralisation et l'instanciation et pour l'obtention du contexte. Les commandes sont envoyées au gestionnaire de commandes par des méthodes de déclenchement implantées dans

Attribut	Valeur	Description
`nom`	`'Changer Taille'`	nom de la commande
`propriétaire`	`unEditeurGraphique`	application à laquelle appartient la commande
`exécArg`	`#(4 9 94 79)`	argument de `exécMth` : nouveau rectangle
`annuArg`	`#(4 9 54 29)`	argument de `annuMth` : ancien rectangle
`contArg`	`#(90 70 9/5 7/2)`	contexte fourni par `contMth` : le nouveau rectangle et le rapport entre le nouveau et l'ancien rectangle
`exécMth`	`#changerTailleExéc:`	nom de la méthode de l'application exécutant la commande : change la taille de l'objet choisi avec `exécArg`
`annuMth`	`#changerTailleAnnu:`	nom de la méthode de l'application annulant la commande : rétablit la taille de l'objet choisi grâce à `annuArg`
`contMth`	`#changerTailleCont:`	nom de la méthode de l'application remplissant l'attribut `contArg`
`généMth`	`#changerTailleGéné:`	nom de la méthode de l'application retournant la généralisation de la commande en la groupant à une autre commande
`instMth`	`#changerTailleInst:`	nom de la méthode de l'application retournant l'instantiation de la commande généralisée

Tableau 3.1 : Exemple de commande de haut niveau pour un éditeur graphique.

l'application. Ces méthodes ne sont pas définies dans la classe `Commande` car leur rôle est précisément de générer ces commandes.

Un exemple de commande de haut niveau pour un éditeur graphique est donné dans le tableau 3.1 : il s'agit de la commande consistant à changer la taille d'un objet, effectuée en déplaçant une des "poignées" de l'objet sélectionné. Dans cet exemple la méthode de déclenchement associée testerait si l'utilisateur a cliqué sur une des "poignées" de l'objet sélectionné, et dans l'affirmative dessinerait une zone rectangulaire spécifiée avec la souris jusqu'à ce que le bouton de la souris soit relâché. La méthode générerait ensuite la commande de haut niveau `changer-taille` et l'enverrait au gestionnaire de commande.

Les commandes ont un comportement très simple : lorsqu'elles reçoivent un message en provenance du gestionnaire de commande comme `exécute`, `annule`, `obtiensContexte`, `généralise` ou `instancie`, elles envoient à leur `propriétaire` le message référencé par `exécMth`, `annuMth`, `contMth`, `généMth` ou `instMth` avec respectivement comme arguments le contenu de `exécArg`, `annuArg` ou `contMth` (`généMth` et `instMth` ne prennent pas d'arguments).

3.3 Gestionnaire de commandes

Chaque fois qu'une application développée avec AIDE reçoit une commande de bas niveau, elle vérifie si il est possible de générer une commande de haut niveau en utilisant l'une de ses méthodes de déclenchement. Si c'est le cas, elle crée une nouvelle instance de la classe `Commande`, remplit ses variables `propriétaire`, `exécArg` et `annuArg` (éventuellement en interagissant avec l'utilisateur comme c'est le cas dans l'exemple de la commande `changer-taille`) et ensuite passe la commande au gestionnaire de commandes. Ce dernier enregistre la commande dans l'historique et renvoie la commande à l'application pour son exécution. L'application exécute la

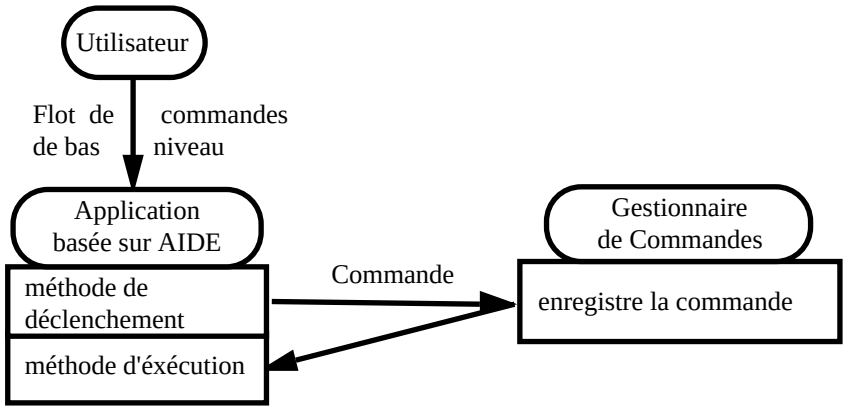


Figure 3.3 : Dialogue entre l'application et AIDE lors de l'exécution d'une commande.

commande en appelant la méthode dont le nom et les arguments sont respectivement stockés dans les variables `exécMth` et `exécArg` de la commande. L'application redonne ensuite le contrôle au gestionnaire de commandes (figure 3.3).

Lorsque le système est en *mode enregistrement*, les étapes supplémentaires suivantes ont lieu (cf. la figure 3.4 qui illustre le flot de données entre une application basée sur AIDE et le gestionnaire de commandes). Tout d'abord le gestionnaire de commandes demande à l'application de l'information contextuelle (en appelant la méthode `contMth`), reçoit en retour une commande enrichie, puis recherche les boucles et essaie de généraliser la commande en la comparant avec les commandes précédemment enregistrées (en utilisant une généralisation par défaut ou bien la méthode `généMth`). L'information contextuelle joue un rôle très important car c'est elle qui aide le gestionnaire de commandes à déterminer les intentions de l'utilisateur. Finalement, la commande généralisée ainsi obtenue est enregistrée dans la définition de la macro et le gestionnaire de commande attend une autre commande.

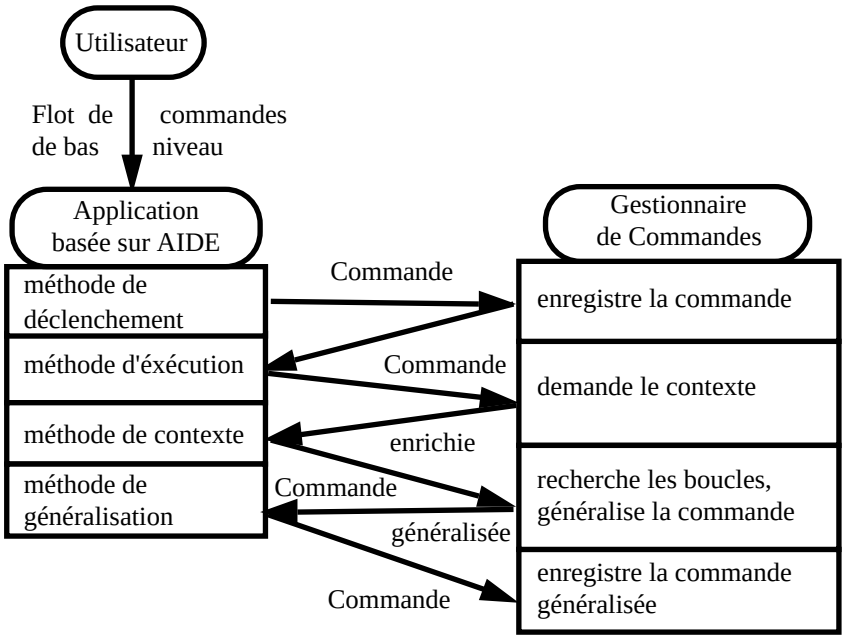


Figure 3.4 : Dialogue entre l'application et AIDE en mode enregistrement de macro.

Le gestionnaire de commandes fournit également une interface ressemblant à celle d'un magnétoscope permettant de naviguer dans l'historique et d'exécuter et d'annuler des commandes entres applications (figure 3.5). Cliquer sur une commande située avant la commande sélectionnée annule toutes les commandes jusqu'à celle-ci. Cliquer sur une commande située après la commande sélectionnée ré-exécute toutes les commandes jusqu'à celle-ci. Cette interface est expliquée plus en détail dans le chapitre suivant.

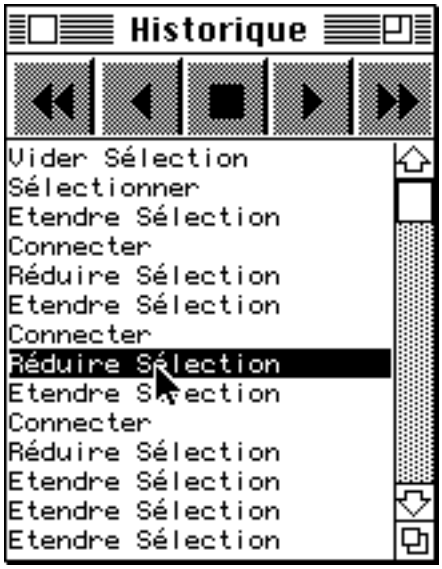


Figure 3.5 : Interface pour naviguer dans l'historique.

3.4 Macros : des commandes définies par l'utilisateur

Dans AIDE, les macros sont des *procédures* définies par l'utilisateur opérant sur une collection *ordonnée* d'arguments et éventuellement retournant une valeur. Une macro est une commande définie par l'utilisateur possédant également un nom et des arguments, et pouvant être exécutée, annulée etc. En fait, les macros sont implantées par la classe `Macro` qui est elle-même une sous-classe de la classe `Commande`. Pour enseigner une macro au système, l'utilisateur sélectionne les objets qui formeront ses arguments, choisit ensuite "Enregistrer Nouvelle Macro" dans le menu "Macros", entre le nom de la macro, démontre les étapes de la macro en effectuant les commandes appropriées et enfin sélectionne l'option "Stopper Enregistrement" dans le menu "Macros". Les objets sélectionnés à la fin de la macro constituent la valeur retournée par la macro. Une fois la macro définie elle apparaît dans le menu "Macros" et peut être invoquée par l'utilisateur à tout moment comme n'importe quelle autre commande de l'application.

Notre système est similaire à Mondrian [Lieberman 93] dans la mesure où les arguments de la macro sont les objets sélectionnés avant l'invocation de la macro, et la valeur de retour consiste en la liste des objets sélectionnés après exécution de la macro. En conséquence il est possible de chaîner des macros, les arguments étant passés au travers de la sélection. Une macro peut appeler une autre macro ce qui correspond à la notion d'appel de procédure et permet de supporter la récursion. Lorsqu'une macro est exécutée, les commandes intermédiaires générées sont passées au gestionnaire de commandes qui les enregistre en tant que *sous-commandes* de la macro-commande. Ainsi, les résultats de l'invocation d'une macro peuvent être annulés en annulant successivement ses sous-commandes. Afin d'éviter la double exécution d'une macro lorsqu'elle est ré-exécutée, ses sous-commandes sont ré-exécutées au lieu de la macro elle-même. Comme nous l'avons déjà mentionné, la capacité de pouvoir annuler les effets

d'une macro est cruciale car il arrive que les généralisations effectuées par le système ne soient pas celles voulues par l'utilisateur.

En *mode exécution de macro*, le dialogue entre AIDE et l'application est représenté figure 3.6. Ici, la macro choisit la prochaine commande généralisée dans sa définition, l'envoie à l'application pour instanciation (par un appel à la méthode `instMth` qui donne des valeurs aux arguments), reçoit en retour une commande normale et l'enregistre comme une de ses sous-commandes. Elle passe ensuite la commande à l'application pour exécution puis choisit la commande généralisée suivante jusqu'à ce que la fin de la macro soit atteinte.

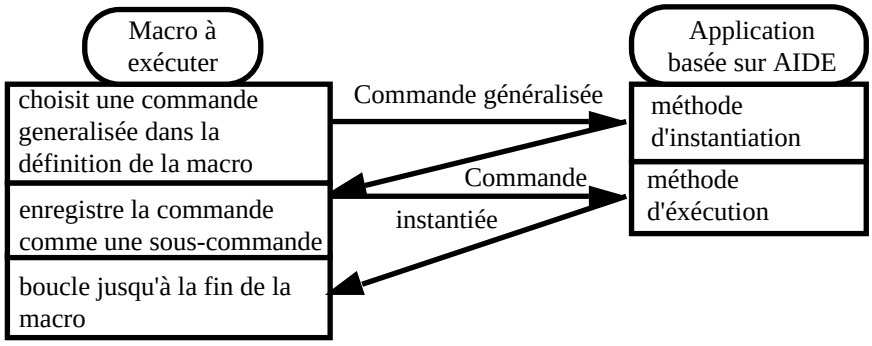


Figure 3.6 Dialogue entre l'application et AIDE en mode exécution de macro.

3.5 Processus de généralisation

3.5.1 Détection des boucles

Lorsque le gestionnaire de commandes généralise l'historique pour créer une macro, il cherche d'abord les boucles en essayant de mettre en correspondance les commandes. Deux commandes peuvent être mises en correspondance si elles sont de même type (c.-à-d. qu'elles ont le *même nom*), si elles possèdent le *même propriétaire*, et si cela ne crée pas une branche sortant d'une boucle déjà formée. Une limitation courante d'AIDE est qu'il ne détecte pas les boucles imbriquées : pour arriver à un effet similaire, l'utilisateur doit tout d'abord créer une macro accomplissant la première boucle et ensuite une seconde macro appelant la première à l'intérieur d'une boucle. Il est aussi important de noter qu'un détecteur de boucle plus sophistiqué peut être facilement incorporé dans le squelette.

3.5.2 Généralisation des arguments

La seconde étape consiste à généraliser la commande c'est-à-dire généraliser ses arguments. Pour ce faire, le système tente de détecter diverses *séquences* en utilisant les valeurs précédentes des arguments pour chaque étape de la boucle. Tout comme Eager, AIDE reconnaît des séquences numériques (constantes, linéaires, croissantes,

décroissantes), des séquences de chaînes de caractères (sous-chaîne en commun, jours, mois), des séquences de points (consistant de deux séquences numériques) et des séquences de classes. L'ensemble des séquences reconnues peut-être facilement étendu pour prendre en compte de nouveaux objets.

Commande	exécArg	annuArg	contArg
changer-taille	#(4 9 94 79)	#(4 9 54 29)	#(90 70 9/5 7/2)
changer-taille	#(2 2 94 58)	#(2 2 25 18)	#(92 56 4 7/2)
changer-taille	#(6 3 94 73)	#(6 3 50 23)	#(88 70 2 7/2)
▼ ▼ ▼ ▼	▼ ▼ ▼ ▼	▼ ▼ ▼ ▼	▼ ▼ ▼ ▼
changer-taille	#(? ? 94 ?)	#(? ? ? ?)	#(? ? ? 7/2)

Tableau 3.2 : Généralisation de trois commandes.

Dans le tableau 3.2 nous montrons comment trois commandes `changer-taille` peuvent être généralisées en une quatrième commande. La commande généralisée est obtenue en combinant tous les arguments (`exécArg`, `annuArg` et `contArg`) de la première commande avec les arguments correspondant des seconde et troisième commandes. Dans le cas présent, la commande généralisée signifie que l'utilisateur a changé la taille de l'objet de telle sorte que l'abscisse du coin en bas à droite soit égale à 94 et que la nouvelle hauteur soit 7/2 fois plus haute que la précédente. Des exemples plus complexes sont donnés dans le chapitre suivant.

3.5.3 Les séquences

La puissance du mécanisme de généralisation réside dans la façon dont les séquences sont traitées, qui est inspiré par Eager [Cypher 91] et par les "descripteurs" de Michalski [Michalski 83]. Les différents types de séquences sont implantés comme des classes comprenant les méthodes `enseigneValeur`, `prochaineValeur` etc. Une séquence est créée en fournissant ses deux premières valeurs, puis en ajoutant ses valeurs suivantes. Par exemple, dans le cas d'un descripteur linéaire, donner 4 et 9 comme premier et second termes crée la séquence de `4 à 9 par pas de 5`. Ajouter 14 comme troisième valeur mettrait à jour la séquence à `4 à 14 par pas de 5`. Maintenant si 15 était la quatrième valeur, la séquence deviendrait une simple `séquence croissante de 4 à 15`. De plus, les séquences peuvent être mises en correspondance avec d'autres séquences, permettant ainsi la généralisation entre plusieurs macros. Ainsi, la séquence résultant de la mise en correspondance d'une séquence linéaire de pas 2 avec une séquence linéaire de pas 3 est une séquence croissante.

4 Un modèle d'historique hiérarchique

4.1 Motivation

Dans la troisième section de ce chapitre, nous avons présenté les principaux éléments d'AIDE qui permettent aux développeurs d'incorporer des techniques démonstrationnelles dans leurs applications écrites en Smalltalk. Nous avons vu que ce squelette possède une structure modulaire qui peut être facilement augmentée pour répondre à des besoins particuliers. Après avoir utilisé AIDE pour créer une application réelle (voir chapitre 4), nous nous sommes aperçus que l'application était très sensible à l'ordre dans lequel les commandes étaient enregistrées. Ceci nous a amené à réfléchir sur la nature des historiques de commandes et il est apparu qu'AIDE (comme la plupart des autres systèmes) représentait l'historique sous la forme d'une simple liste de commandes. Malheureusement, les listes ne fournissent qu'une vue linéaire de l'interaction passée, reflétant pauvrement la structure de la tâche de l'utilisateur. En nous inspirant de la proposition de Kosbie [Kosbie 93] nous avons établi un nouveau critère auquel doit répondre AIDE : structurer l'historique des commandes (voir section 2.4). Dans les deux premiers chapitres nous avons vu que les historiques constituent un composant essentiel permettant à l'utilisateur de réutiliser des activités passées pour corriger une erreur, réduire les tâches répétitives où simplement consulter ce qu'il a fait [Greenberg 93]. Proposer un nouveau modèle pour l'historique va donc bien au delà de la Programmation Par Démonstration.

Maintenant que Kosbie a proposé une nouvelle approche pour représenter les historiques (c.-à-d. les agrégats de commandes) nous abordons une nouvelle question : comment *créer incrémentalement* de tels historiques hiérarchiques à partir des commandes de bas niveau de l'utilisateur ? Cette question est importante, car à chaque fois qu'une commande est exécutée, l'historique doit être mis à jour en conséquence et pour des raisons d'efficacité il ne doit pas être totalement recréé de toutes pièces à chaque fois. Pour résoudre ce problème nous introduisons un nouveau modèle [Piernot 95b] où (1) chaque commande possède un parent prototypique qui représente son abstraction naturelle ; où (2) la construction de l'historique est accomplie par l'adoption de commandes ; et où (3) l'éventuelle ambiguïté soulevée par la présence de plus d'un parent possible est résolue en créant des commandes partielles.

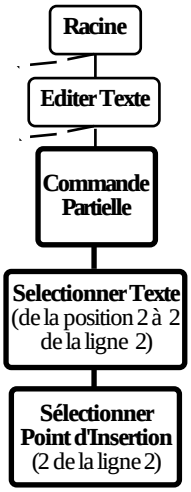
La section suivante offre tout d'abord une présentation générale des deux composants de ce modèle : le constructeur d'historiques incrémental et les arbres de commandes. Dans un second temps, elle illustre comment ce modèle a été intégré dans AIDE de façon extensible. Pour preuve du concept, les exemples sont tirés d'un éditeur de texte qui a été développé avec AIDE.

4.2 Présentation générale

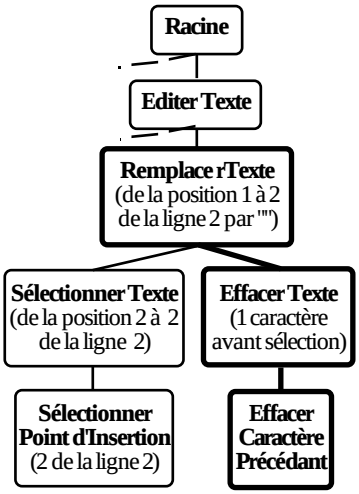
4.2.1 Constructeur d'historiques incrémental

Le constructeur d'historiques est responsable du groupement de commandes en commandes plus générales. Le constructeur développé est *incrémental*, c'est-à-dire qu'ajouter une nouvelle commande à l'historique n'entraîne pas une réinterprétation totale de l'historique — ce qui serait très inefficace — et suppose que chaque commande possède un *parent prototypique*. Le constructeur d'historiques essaie d'insérer une nouvelle commande dans l'historique en demandant à ce dernier d'adopter la commande. Dans le cas où la commande est une continuation naturelle (c'est à dire que son parent figure dans la dernière branche de l'historique) elle est adoptée ; dans le cas contraire, puisque la commande connaît son abstraction naturelle, le constructeur génère le parent de la commande et essaie d'inclure celui-ci dans l'historique de la même façon. Pour certaines commandes, il peut y avoir une ambiguïté concernant leur parent prototypique ; dans ces cas, l'identité du parent est seulement connue à l'arrivée des commandes suivantes. Deux approches sont possibles pour résoudre ce problème : la première est de choisir un des parents potentiels à titre d'essai et de revenir en arrière en cas d'erreur ; la seconde approche consiste à laisser l'ambiguïté en créant une *commande partielle*, dont l'identité sera déterminée ultérieurement, levant ainsi l'ambiguïté. La seconde option a été retenue afin que l'arbre des commandes ne reflète que ce que l'utilisateur a effectué jusqu'au moment présent.

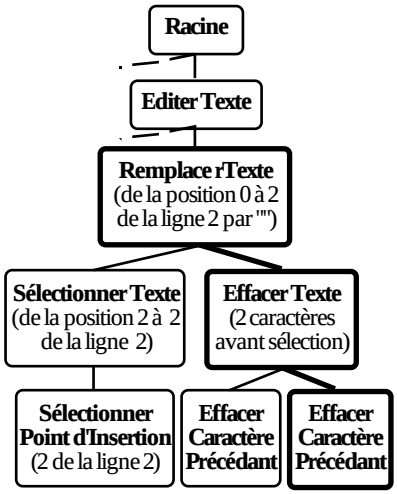
A titre d'exemple, la figure 3.7 fournit une description étape par étape de la façon dont les commandes d'édition de texte ont été ajoutées une par une à l'historique pour former le côté droit de l'arbre de la figure 3.1. Pour ajouter une nouvelle commande C à l'historique, le constructeur commence sa recherche par la branche la plus à droite de l'arbre et détermine, en parcourant la branche de bas en haut, si l'une des commandes peut adopter C. Dans l'affirmative, le constructeur accomplit l'adoption. Deux cas peuvent se présenter à ce moment-là : si l'adoption génère une nouvelle commande (c'est-à-dire une commande partielle adopte une nouvelle commande comme dans la figure 3.7.b), alors l'ancienne commande est ôtée de l'arbre et remplacée par la nouvelle ; autrement, C est simplement ajoutée aux enfants de la commande. En conséquence de quoi, un par un, chacun des ancêtres de la commande met à jour ses arguments de façon à refléter le changement qui touche à l'un de ses enfants (figure 3.7.c). Si C n'est adoptée par aucune des commandes sur la dernière branche (i.e. figure 3.7.d `Taper Caractère` ne pouvait pas être adopté par les commandes `Effacer Texte`, `Remplacer Texte` ou `Editer Texte`), alors le constructeur génère le parent de la commande et répète le processus avec la commande parent jusqu'à ce qu'une adoption soit effectuée (dans la figure 3.7.d, le parent de `Taper Caractère` — `Taper Texte` — a été adopté par `Remplacer Texte`). Le processus prend toujours fin car la commande `Racine` doit toujours être présente dans la chaîne des ascendants possibles de chaque commande.



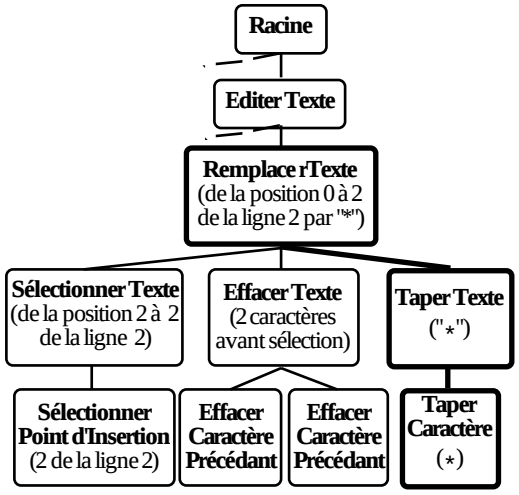
a. Sélectionner Point d'Insertion et son parent n'ont pas été adoptés. Son grand parent l'a été.



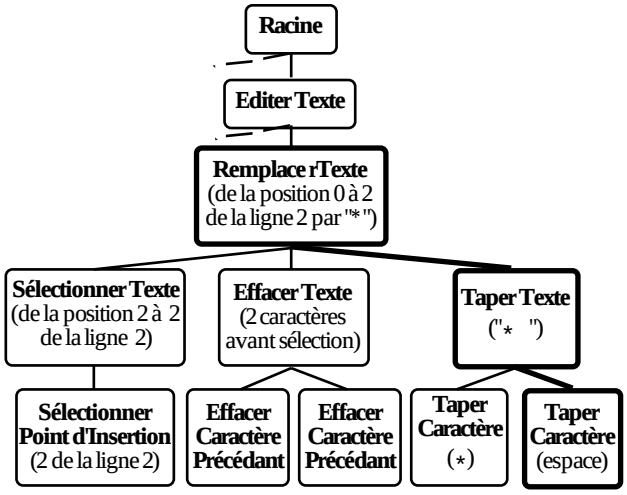
b. Effacer Caractère Précédent n'a pas été adopté. Son parent a été créé et adopté par Commande Partielle. L'adoption a généré une nouvelle commande (Remplacer Texte)



c. Effacer Caractère Précédent a été adopté par Effacer Texte. Effacer Texte et Remplacer Texte ont mis à jour leurs arguments en conséquence.



d. Taper Caractère n'a pas été adopté. Son parent a été créé et adopté par Remplacer Texte. Remplacer Texte a mis à jour ses arguments.



e. Taper Caractère a été adopté par Taper Texte. Taper Texte et Remplacer Texte ont mis à jour leurs arguments.

Figure 3.7. Description étape par étape de la façon dont les commandes sont ajoutées à l'historique (nous montrons ici comment la partie droite de l'arbre de la figure 3.1 a été générée).

4.2.2 Arbre de commandes

Les arbres de commandes sont en fait des commandes, composées elles-mêmes d'autres commandes. Comme dans MacApp [Wilson 90] et MotifApp [Young 92], les commandes sont des objets comprenant entre autres les messages `exécute` et `annule`. Le tableau 3.3. illustre l'ensemble des commandes définies par le développeur dans l'éditeur de texte développé avec AIDE. La première colonne représente la classe de la commande, la seconde présente la grammaire déterminant comment la commande peut adopter une nouvelle commande et la dernière fournit la classe du parent prototypique. Par exemple, la commande `Sélectionner Texte` peut avoir pour enfants les commandes `Sélectionner Point d'Insertion`, `Sélectionner Rangée` ou `Déplacer Curseur` et pour parent `Remplacer Texte` ou `Formater Texte`. Le parent de `Sélectionner Texte` est `Commande Partielle` qui est une commande temporaire, attendant l'adoption d'une commande pour changer d'identité : ceci est dû au fait que `Sélectionner Texte` peut avoir deux parents possibles, `Remplacer Texte` ou `Formater Texte`. Si `Commande Partielle` adopte `Taper Texte` ou `Effacer Texte`, elle génère la commande `Remplacer Texte`. Si elle adopte `Changer Jeu de Caractères`, `Changer Couleur`, `Changer Taille` ou `Changer Style` elle génère la commande `Formater Texte`. Lorsque le processus d'adoption génère une nouvelle commande, l'ancienne (ici `Commande Partielle`) est enlevée de l'arbre et la nouvelle commande est ajoutée. Pour définir la grammaire des commandes, le développeur doit définir les méthodes `classeDuParent` et `peutAdopter` (voir section 4.3.2).

Les arbres de commandes facilitent le travail des mécanismes de généralisation utilisés en Programmation Par Démonstration car la détection de boucles devient moins sensible aux variations d'accomplissement de la tâche. Ainsi dans l'exemple de la figure 3.1, l'utilisateur remplace les deux premiers caractères de la première et seconde ligne en utilisant différentes commandes primitives, mais les deux cas elles aboutissent à la même commande de haut niveau `Remplacer Texte`.

Les arbres de commandes offrent également un meilleur support pour les mécanismes de correction d'erreur en réduisant le temps mis pour naviguer dans l'historique (en annulant et ré-exécutant des commandes avec une interface similaire à celle de la figure 3.6). Puisque l'historique enregistre des commandes intermédiaires, le système n'a pas besoin d'annuler ou de refaire toutes les commandes primitives mais plutôt annule ou refait les commandes intermédiaires. Par exemple pour annuler les cinq dernières commandes primitives de la figure 3.1, il suffit d'annuler la commande `Remplacer Texte`. Cela suppose bien sûr que la commande `Remplacer Texte` ait redéfini les méthodes `annule` et `exécute`. Pour l'éditeur de texte nous avons redéfini ces méthodes pour toutes les commandes (ces méthodes utilisent les arguments de la commande), ce qui est plus rapide que de demander à chacun des enfants de s'annuler lui-même.

Commande	**Règles gouvernant l'adoption des enfants**	**Commande Parente**
Sélectionner Point d'Insertion Sélectionner Rangée Déplacer Curseur	Pas d'enfants (commande primitive)	Sélectionner Texte
Sélectionner Texte	(Sélectionner Point d'Insertion | Sélectionner Rangée | Déplacer Curseur)$^{+}$	Commande Partielle
Commande Partielle	Sélectionner Texte. Peut adopter une commande dont le grand parent peut-être Editer Texte. Dans l'affirmative, le résultat de l'adoption est le parent de la nouvelle commande adoptée (c.-à-d. Remplacer Texte ou Formater Texte)	Editer Texte
Taper Caractère	Pas d'enfants (commande primitive)	Taper Texte
Taper Texte	Taper Caractère^{+}	Remplacer Texte
Effacer Sélection Effacer Caractère Précédent Effacer Caractère Suivant	Pas d'enfants (commande primitive)	Effacer Texte
Effacer Texte	Sélectionner Texte$^{?}$, (Effacer Sélection | Effacer Caractère Précédent | Effacer Caractère Suivant)$^{+}$	Remplacer Texte
Remplacer Texte	Sélectionner Texte$^{?}$, (Taper Texte | Effacer Texte)$^{+}$	Editer Texte
Changer Jeu de Caractères Changer Couleur Changer Style Changer Taille	Pas d'enfants (commande primitive)	Formater Texte
Formater Texte	Sélectionner Texte$^{?}$, (Changer Jeu de Caractères | Changer Couleur | Changer Style | Changer Taille)$^{+}$	Editer Texte
Editer Texte	(Commande Partielle | Remplacer Texte | Formater Texte)$^{+}$	Racine
Racine	Accepte toute commande dont le parent peut être Racine	Aucune

Tableau 3.3. Commandes utilisées dans l'éditeur de texte
($c^{?}$ signifie que c est présent 0 ou 1 fois ; c,d implique c puis d ;
c|d implique c ou d ; c^{+} signifie que c est présent au moins une fois).

4.3 Implantation du modèle dans AIDE

4.3.1 Algorithme du constructeur d'historiques

La première chose à noter est que le constructeur d'historiques garde un pointeur sur la dernière commande exécutée. Lorsque les commandes sont annulées, l'arbre des commandes n'est pas modifié, mais au lieu de cela, le pointeur est mis à jour. Quand une nouvelle commande arrive, les commandes précédemment annulées sont ôtées de l'arbre avant que la nouvelle commande ne soit ajoutée (c'est ce que Archer, Conway, Schneider appellent le "truncate/redo" [Thimbleby 91]). Le constructeur est encapsulé dans la classe `Commande` — comme un ensemble de méthodes — et son point d'entrée est la méthode `ajoute` (se reporter à la figure 3.8). Cela signifie que pour ajouter une nouvelle commande dans l'arbre, on envoie le message `ajoute` à la racine de l'arbre, avec la commande à ajouter comme argument. Nous en avons profité pour déplacer les méthodes du gestionnaire de commande dans la commande `Racine`. Cet objet joue maintenant le double rôle de gestionnaire de commande et d'historique.

Du fait de la nature récursive de l'algorithme et de l'encapsulation du constructeur dans la classe `Commande`, une application peut facilement *étendre* le constructeur en surchargeant la méthode `tenteDajouter` (figure 3.9) dans ses propres commandes. Ainsi, le comportement par défaut que nous avons décrit dans la section 4.2.1 peut être modifié pour répondre aux besoins particuliers d'une application. Ce point est spécialement important dans le cas d'un historique global (un historique unique pour toutes les applications ouvertes), où une application pourrait avoir besoin de modifier le comportement du constructeur pour ses propres commandes, sans avoir à modifier son comportement reste du système. De la même façon, puisque les méthodes du gestionnaire de commandes sont maintenant dans la classe `Commande`, il est possible de modifier le comportement de tout le système.

```
Définition ajoute  uneCommande
  Tant-que je tenteDajouter   uneCommande et pas d'adoption,
    Je demande à uneCommande de créerSonParent
    et de l'inclure dans uneCommande
  Fin-Tant-que
  Si l'adoption a généré une nouvelle commande,
    J'enlève  l'ancienne commande
    et j'ajoute  la nouvelle
  Fin-Si
Fin-Définition
```

Figure 3.8. Algorithme de le constructeur de commande incrémental.

```
Définition tenteDajouter   uneCommande
  Si je ne suis pas une feuille,
    Je demande à mon dernier enfant de tenterDajouter
    uneCommande
  Fin-Si
  Si une adoption s'est produite,
    Si elle n'a pas généré une nouvelle commande,
      Je metsAjour   mes Arg uments
    Sinon,
      Je retourne le résultat de l'adoption
    Fin-Si
  Sinon,
    Si je peuxAdopter   uneCommande,
      J'adopte  uneCommande et retourne le résultat
    Sinon,
      Je retourne qu'aucune adoption ne s'est passée
    Fin-Si
  Fin-Si
Fin-Définition
```

Figure 3.9. Pseudo code pour la méthode `tenteDajouter`.

4.3.2 Représentation des commandes

La classe `Commande` (cf. tableau 3.4) est une sous-classe de la classe `Arbre`. Contrairement à la version précédente d'AIDE où toutes les commandes étaient des instances de la classe `Commande`, ici la classe `Commande` est abstraite — ce qui signifie qu'elle ne possède pas d'instances — et les commandes sont des instances de sous-classes de la classe `Commande`. Ainsi, elle hérite de toutes les méthodes de la classe `Arbre` (elle peut avoir des enfants — aussi appelés sous-commandes — et fournit un moyen d'accès à ses descendants) et possède des méthodes qui lui sont propres. Parmi ces méthodes on retrouve `exécute`, `annule` et `nom` qui gouvernent l'exécution d'une commande et retourne son nom sous une forme intelligible par l'utilisateur. Pour implanter une nouvelle commande, le développeur crée tout d'abord une sous-classe de `Commande`, ajoute les arguments de la commande en tant que variables de l'objet puis redéfinit certaines des méthode mentionnées ci-dessus. De plus, le développeur doit décider de la façon dont les commandes seront regroupées pour former des commandes de plus haut niveau. Ceci est fait en définissant les commandes qu'elle peut adopter (méthode `peuxAdopter`), en définissant la classe de la commande parente (méthode `classeDuParent`), et en définissant la façon dont la commande doit mettre à jour ses arguments lorsque : (a) un enfant lui est ajouté, (b) retiré ou (c) met à jour lui aussi ses

Classe `Commande` sous-classe d'`Arbre`	
`nom(cmd)`	Retourne le nom de `cmd`. Doit être surchargé.
`exécute(cmd)`	Exécute `cmd`. Doit être au moins surchargé par les commandes primitives (feuilles).
`annule(cmd)`	Annule `cmd`. Doit être au moins surchargé par les commandes primitives (feuilles).
`classeDuParent(cmd)`	Retourne la classe à laquelle le parent de `cmd` doit appartenir. *Doit être surchargé.*
`metsAjourArgs(cmd)`	Met à jour les arguments de `cmd` (en utilisant ses enfants). *Doit être surchargé.* Appelé quand ses enfants changent.
`peuxAdopter(cmd,cmd2)`	Retourne si `cmd` peux adopter `cmd2` (utilise `classeDuParent`).
`adopte(cmd,cmd2)`	Détermine la commande résultant de l'adoption (*cmd* ou une nouvelle commande), puis ajoute `cmd2` aux enfants de la commande résultant (utilise `metsAjourArgs`) pour finalement la retourner.
`créeParent(cmd)`	Génère le parent de `cmd` (utilise `classeDuParent` et `adopte`).
`ajoute(cmd,cmd2)`	Ajoute `cmd2` aux descendants de `cmd` (cf. figure 3.8).
`enlève(cmd,cmd2)`	Enlève `cmd2` des descendants de `cmd`.

Tableau 3.4. Description de la classe Commande.

arguments (méthode `metsAjourArgs`). Certaines méthodes comme `exécute`, `annule` et `peuxAdopter` n'ont pas besoin d'être redéfinies lorsque le comportement par défaut est correct. Ainsi par défaut, les méthodes `exécute` et `annule` parcourent les enfants de la commande et demandent à chacun d'eux de s'exécuter ou de s'annuler. Ceci implique bien sûr que ces deux méthodes doivent être au moins implantées pour les commandes primitives (commandes sans enfants, c'est-à-dire qui sont les feuilles de l'arbre). D'une

façon similaire, la méthode `peuxAdopter(cmd)` vérifie si le receveur peux avoir `cmd` comme enfant en vérifiant si la classe du parent de `cmd` est la même que la classe du receveur, en conséquence de quoi il est rare d'avoir à modifier cette méthode.

L'implantation d'historiques pose le problème de l'espace mémoire occupé par ceux-ci après une longue période d'utilisation. Notre solution consiste à oublier les enfants des plus vieilles commandes. Pour les commandes intermédiaires (par exemple `Taper Texte`), ôter leurs enfants ne les empêche pas de s'exécuter ou de s'annuler puisqu'elles stockent leurs propres arguments (ici le texte qui a été tapé) et définissent leurs propres méthodes `exécute` et `annule`. La granularité avec laquelle les anciennes commandes pourront être annulées ou exécutées sera simplement moindre. En dernier ressort, des commandes de haut niveau peuvent bien sûr être ôtées de l'arbre si elles ont dépassé un certain âge.

5 Conclusion

Nous avons consacré la première partie de ce chapitre à expliquer les principes qui ont présidé à la conception du squelette d'application AIDE. Ces principes portent sur l'extensibilité du squelette, une riche représentation des commandes et de l'historique et un mécanisme de récupération d'erreur. Ces principes nous ont ensuite amené à articuler l'architecture d'AIDE autour de commandes de haut niveau et d'un gestionnaire de commandes gérant l'historique et la création de macros.

Dans un second temps, nous avons souligné l'intérêt de représenter l'historique sous une forme arborescente plutôt que linéaire car cela permet d'offrir un meilleur support pour la récupération d'erreur ainsi que les techniques démonstrationnelles. Nous avons ensuite introduit notre modèle d'arbre de commandes (les parents prototypiques, le mécanisme d'adoption et les commandes partielles) et montré qu'il est une réponse pratique et élégante au problème soulevé par la *création incrémentale* d'historiques hiérarchiques. Nous avons également présenté comment nous avons implanté avec succès ce modèle en étendant AIDE et illustré son utilisation au travers d'un éditeur de texte.

Dans son modèle, Kosbie [Kosbie 94] adopte une approche déclarative au lieu d'une approche procédurale comme la nôtre qui est plus lourde à mettre en oeuvre. Ainsi, pour définir la grammaire des commandes, le développeur crée pour chaque commande un "hieractor" qui décrit sous quelles conditions cette dernière doit être générée. De telles conditions incluent une commande déclenchante, une commande intermédiaire et une commande terminale. A titre d'exemple, le "hieractor" pour un bouton possèdera `presser-bouton-souris` comme commande déclenchante, `déplacer-souris` comme commande intermédiaire et `relacher-bouton-souris` comme commande terminale. Malheureusement cette approche ne prend pas les arguments des commandes en compte mais seulement leur classe : ceci constitue une limitation sévère car, par exemple, dans un programme de dessin, il serait impossible de spécifier que la commande `pivoter-`

`objet` peut être composée d'autres commandes `pivoter-objet` pourvu que ces sous-commandes s'appliquent au même objet et qu'elles aient même centre de rotation. Notre modèle permet d'exprimer de telles relations. Bien que notre approche requiert plus d'effort de la part du développeur que celle du système de Kosbie, ce problème peut être allégé si le développeur structure correctement la hiérarchie des classes représentant les commandes. Il est aussi important de noter que le modèle de Kosbie n'offre aucun avantage pour l'annulation de commandes. Un dernier avantage de notre approche est qu'elle n'est pas limitée à des interfaces graphiques mais fonctionne aussi avec des interfaces vocales ou à base de lignes de commandes.

Nous dressons ci dessous un tableau récapitulatif permettant de situer AIDE par rapport aux autres systèmes de programmation par démonstration étudiés dans le second chapitre (voir tableau 2.1).

	Aide
Domaine d'application	Indépendant de tout domaine
Création d'une macro	Manuelle dans la version courante du squelette : démarre / stoppe enregistrement. La sélection représente les arguments
Généralisation d'une macro	Effectuée par le système et l'utilisateur (voir fonction de recherche, figure 4.6)
Invocation d'une macro	Manuelle dans la version courante du squelette : l'utilisateur sélectionne les arguments puis choisit la macro dans un menu
Exécution d'une macro	Possibilité d'annuler et de ré-exécuter des commandes à plusieurs niveaux (voir fenêtre historique, figure 3.6)
Visualisation d'une macro	Sous forme textuelle (voir fenêtre historique, figure 3.6)
Structures de contrôle	Boucles non imbriquées, variables et appel de procédure
Inférence	Oui (boucles et variables)
Connaissances	Séquences, création de boucles bien formées
Interface hôte	Macintosh
Langage	Smalltalk/V puis SmalltalkAgents
Plate-forme	Macintosh Quadra 660AV

Tableau 3.5 : Tableau récapitulatif pour AIDE.

Chapitre 4

SPII : un éditeur graphique développé avec AIDE

1 Introduction

Dans le chapitre précédent nous avons présenté AIDE, un squelette d'application aidant les développeurs à incorporer des capacités de Programmation Par Démonstration dans leurs applications écrites en Smalltalk. Maintenant qu'un tel outil a été développé, il est important de vérifier si il rempli bien sa fonction. Pour ce faire, nous avons utilisé la *première version* d'AIDE (celle dépourvue des arbres de commandes) pour implanter SPII [Piernot 93], un éditeur graphique utilisant le paradigme objet du type MacDraw [Cutter 87], programmable par démonstration. Les raisons pour lesquelles nous avons opté pour ce type d'application sont multiples. Les éditeurs graphiques sont des logiciels qui mettent en oeuvre des interfaces à manipulation directe et qui peuvent être utilisés pour de nombreuses tâches comme le dessin industriel, la création de graphes, de diagrammes de réseau, d'organigrammes etc. Le domaine de l'édition graphique présente des types variés de répétitions, ce qui en fait un candidat idéal pour la PPD. Il est ainsi souvent utile d'apporter des changements répétitifs à la forme ou aux propriétés graphiques d'un ensemble d'objets. L'édition graphique requiert aussi parfois que des objets soient disposés d'une façon répétitive. De plus, d'autres systèmes démonstrationnels ont étés bâtis dans de ce domaine ce qui permet d'effectuer des

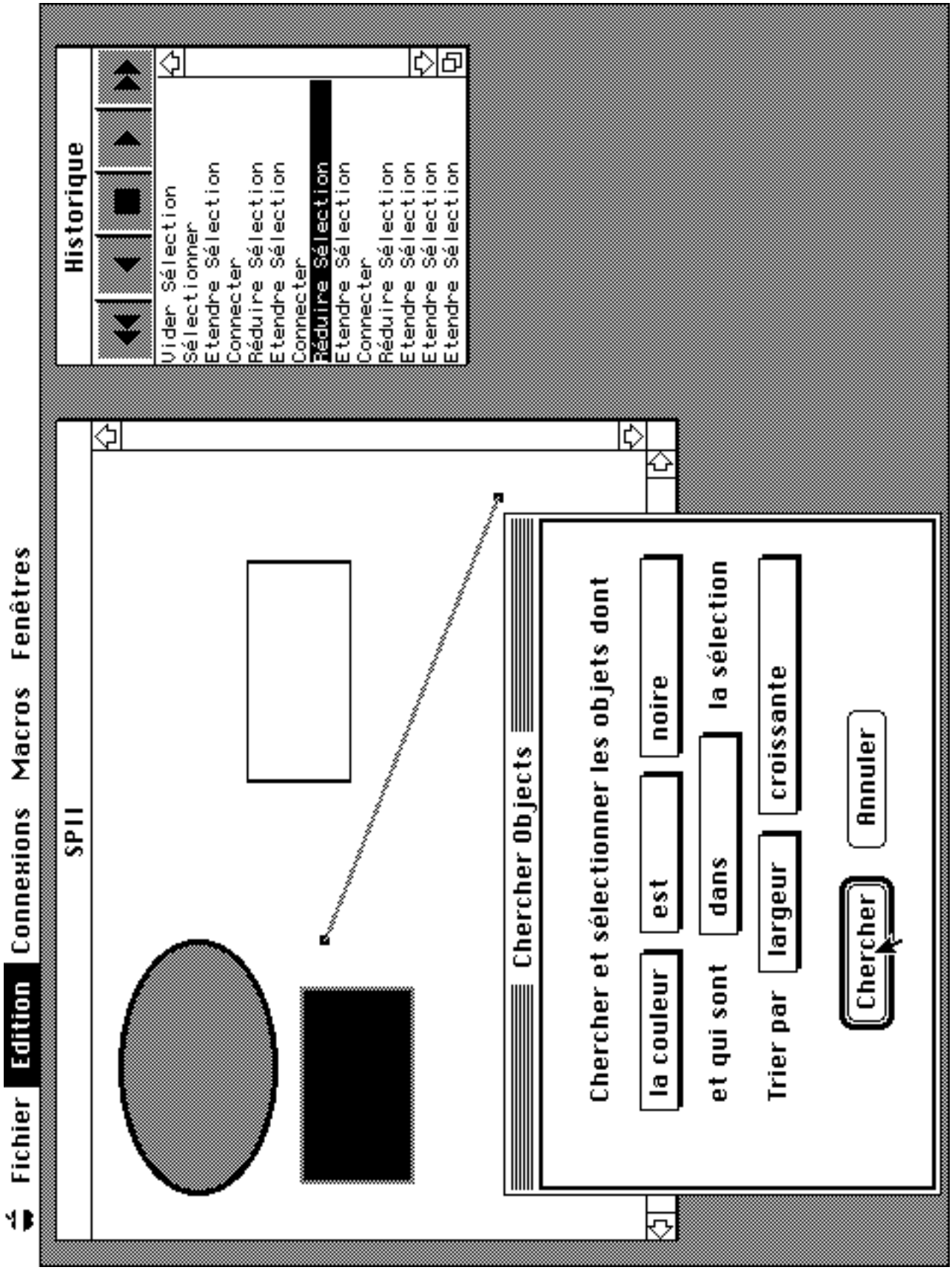


Figure 4.1 Vue générale de l'interface de SPII.

comparaisons. Enfin, l'édition graphique est juste l'une des nombreuses tâches où l'utilisateur vient placer des éléments sur une page électronique. D'autres applications comme les logiciels de Publication Assistée par Ordinateur, les éditeurs d'interfaces graphiques, les applications de CAO électronique possèdent des traits communs avec l'édition graphique, ce qui rend les techniques décrites par la suite également applicables à ces différents domaines.

Dans ce chapitre nous présentons les fonctions de l'éditeur graphique, puis nous montrons à travers deux exemples comment l'utilisateur crée des macros par démonstration dans SPII. Un problème important survenant dans le domaine de la Programmation Par Démonstration est comment permettre à l'utilisateur d'exprimer ses intentions. Certaines caractéristiques de SPII sont particulièrement adaptées et sont également décrites. Enfin, nous détaillons l'implantation de l'éditeur et la façon dont les macros sont généralisées.

2 Présentation de l'éditeur

SPII (figure 4.1) est un prototype d'éditeur graphique utilisant le paradigme objet permettant de créer des interfaces graphiques, des dessins techniques et des graphes. Dans cet éditeur, l'utilisateur peut créer, sélectionner, déselectionner, déplacer, changer la taille, effacer, éditer et connecter des objets par manipulation directe. Ces objets peuvent être des figures géométriques ou des éléments d'interface graphique. SPII a été utilisé dans le cadre d'une application aidant l'utilisateur à gérer la topologie d'un réseau local dans une salle machine, ceci en respectant des contraintes de compatibilité, de placement et de coût (figure 4.2). SPII a été développé au-dessus d'AIDE et utilise ses fonctions de génération de macro et de récupération d'erreur. La fenêtre principale est l'espace de travail dans lequel l'utilisateur place et manipule les objets. Une barre de menu (en haut à gauche) contient les commandes de l'éditeur. A droite figure la fenêtre contenant l'historique des commandes effectuées par l'utilisateur (figure 4.1).

2.1 Navigation dans l'historique

Chaque fois que l'utilisateur accomplit une commande, celle-ci est ajoutée dans la fenêtre contenant l'historique, puis sélectionnée. Pour annuler la dernière commande effectuée, l'utilisateur choisit l'option "Annuler" dans le menu "Edition" ou presse le bouton "flèche arrière" dans la fenêtre historique. L'utilisateur peut également cliquer sur l'une des commandes passées et, dans ce cas, toutes les commandes sont annulées jusqu'à la commande choisie, et cette dernière est sélectionnée (à ce moment la, les commandes annulées figurent toujours dans l'historique). De façon symétrique, l'utilisateur peut ré-exécuter une commande qu'il avait annulée en choisissant l'option "Refaire" dans le menu "Edition" ou en cliquant sur la flèche avant. Cliquer sur une commande située après la commande sélectionnée (c'est-à-dire une commande qui a été annulée), a pour effet de ré-exécuter toutes les commandes jusqu'à la commande

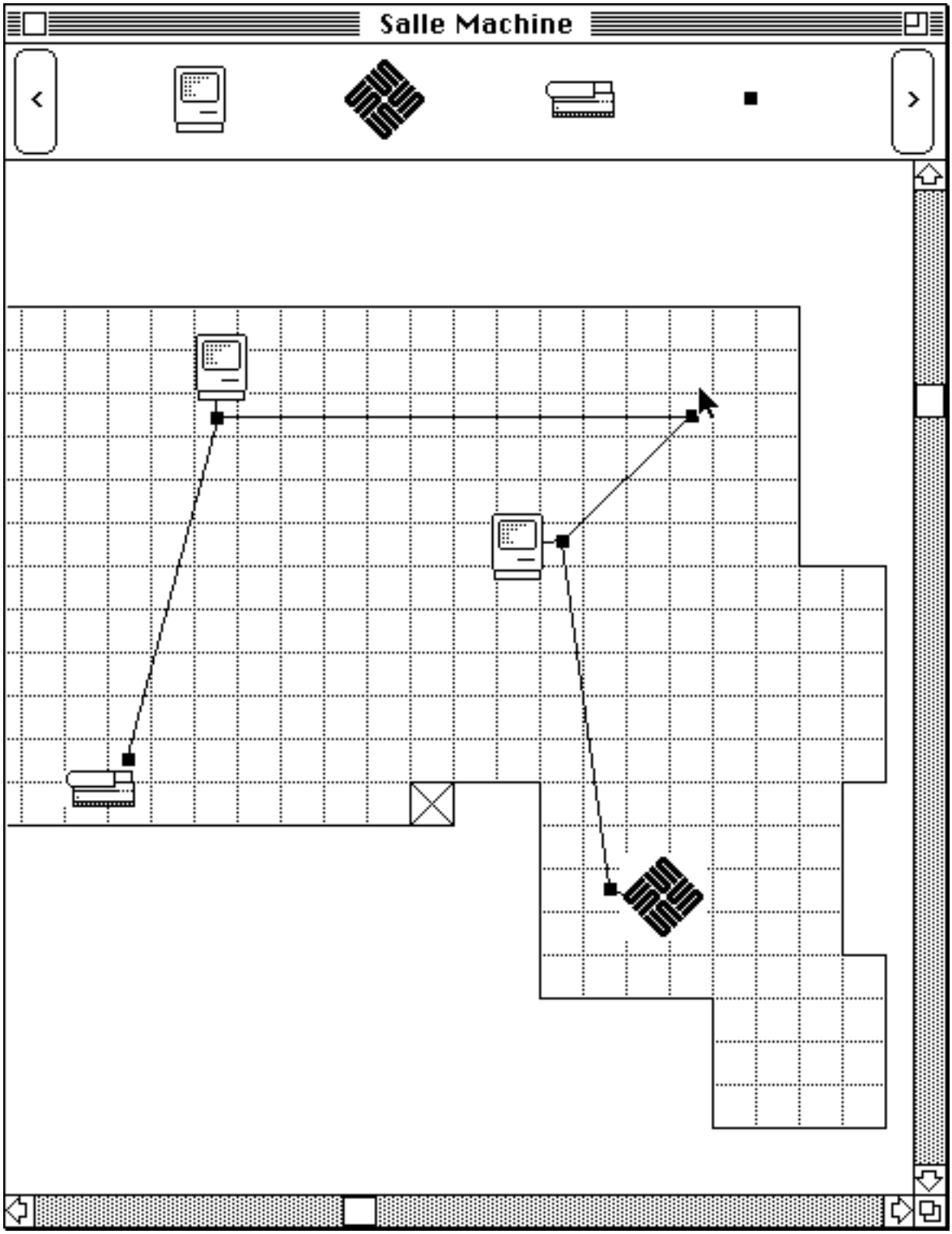


Figure 4.2 : Une application développée avec SPII gérant la topologie d'un réseau local dans une salle machine.

nouvellement sélectionnée. Lorsqu'une série de commandes est annulée ou ré-exécutée, la fenêtre principale affiche un film où l'on voit toutes les commandes se faire ou se défaire. L'utilisateur peut interrompre ce film en pressant le bouton "Stop". Cela fournit un moyen visuel d'identifier l'endroit où une erreur s'est produite sans avoir à lire la séquence de commandes. Deux autres boutons (double flèches) permettent d'annuler et de ré-exécuter toutes les commandes présentes dans l'historique. Notons que dans la version courante, annuler ou exécuter une macro ne montre pas les étapes intermédiaires, ceci dans le but d'accélérer le déroulement du film et être consistant avec l'exécution des autres commandes. La possibilité d'exécuter ou d'annuler pas à pas une macro est également présente dans AIDE, mais elle n'a pas encore été implantée au niveau de l'interface.

Lorsque l'utilisateur a annulé un certain nombre de commandes et qu'il exécute une nouvelle commande, toutes les commandes annulées sont retirées de l'historique comme dans le modèle d'Archer, Conway et Schneider [Thimbleby 91]. En outre, pour maintenir la cohérence de l'historique il n'est pas possible d'insérer ou d'effacer manuellement une commande. Ceci est dû au fait que les commandes suivantes ne pourraient pas être rejouées correctement dans tous les cas. En effet, imaginons que l'utilisateur insère une commande effaçant un objet et que la commande suivante porte sur cet objet. Cette commande ne pourra bien sûr pas être ré-exécutée. De façon similaire, si une commande créant un objet est retirée, toutes les commandes qui suivent portant sur cet objet ne pourront pas être ré-exécutées. D'autres systèmes comme Chimera ou Smallstar sont plus tolérants et autorisent l'édition de l'historique et en cas de problème une erreur sera alors simplement signalée. Dans la version courante de SPII nous avons opté pour la simplicité au prix d'une certaine perte de flexibilité.

2.2 Les commandes de SPII

2.2.1 Ajouter et enlever des objets

SPII permet d'ajouter divers objets (commande `ajouter objets`) grâce à un processus en deux étapes. Tout d'abord, l'utilisateur presse la touche commande et clique avec la souris dans la fenêtre courante, révélant ainsi une liste d'objets à choisir (figure 4.3). Cette liste comprend des objets graphiques (ellipse, ligne, rectangle) ainsi que des objets d'interface (barre de défilement, boîte à cocher, bouton poussoir, bouton radio). Dans un second temps, l'utilisateur trace un rectangle avec la souris dans lequel viendra s'inscrire le nouvel objet choisi. SPII étant au stade de prototype, nous avons limité le nombre d'objets disponibles (il est néanmoins possible de rajouter de nouveaux objets sans avoir à modifier le code de l'éditeur).

Pour ôter des objets de la fenêtre, l'utilisateur sélectionne (voir section suivante) les objets en question puis presse la touche "effacer" ou choisit option "Effacer" dans le menu "Edition" (commande `effacer sélection`).

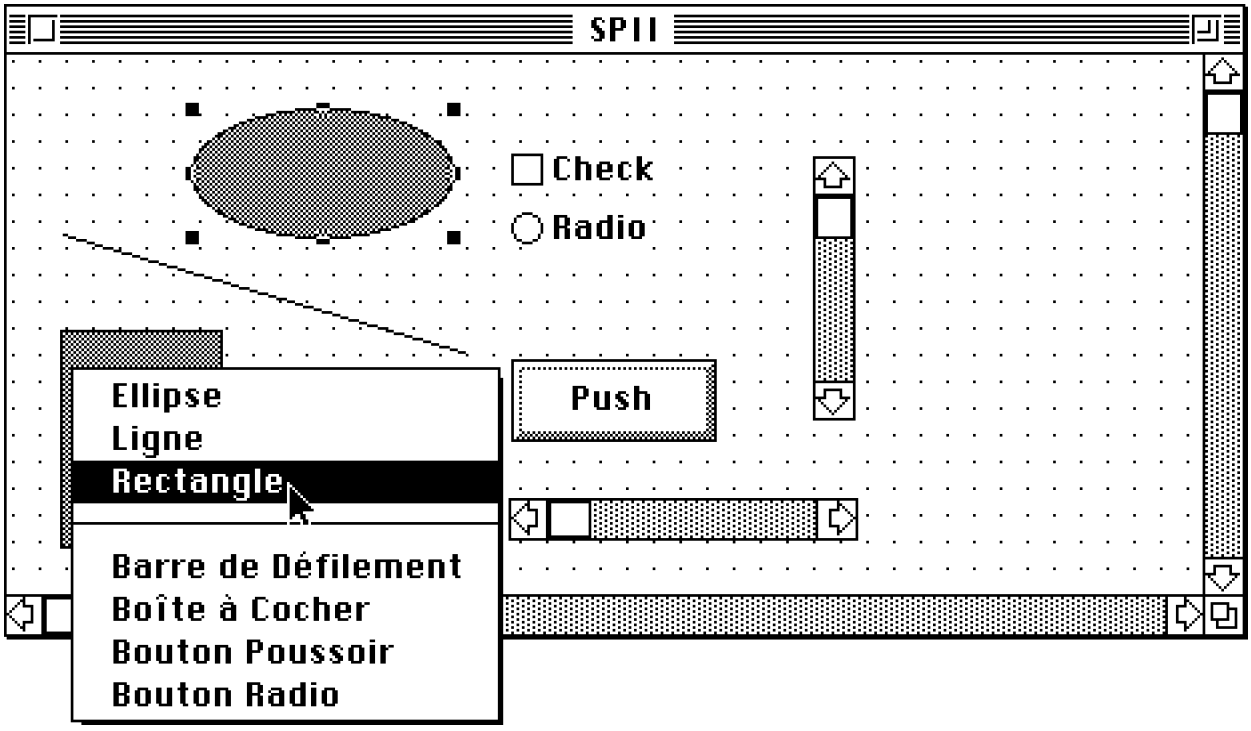


Figure 4.3 Objets disponibles dans SPII.

2.2.2 Sélectionner et désélectionner des objets

Nous appelons sélection l'ensemble des objets sélectionnés dans la fenêtre courante. La sélection correspond à l'ensemble des objets sur lesquels la prochaine commande portera. Un objet sélectionné apparaît entouré de huit "poignées" (voir l'ellipse dans la figure 4.3). Cliquer sur un objet non sélectionné (commande `sélectionner`), remplace la sélection courante par l'objet cliqué (ce qui a pour effet de désélectionner les objets précédemment dans la sélection). Cliquer sur un objet non sélectionné en maintenant la touche "shift" enfoncée (commande `étendre sélection`), ajoute l'objet à la sélection. Cliquer sur un objet sélectionné, en maintenant la touche shift enfoncée (commande `réduire sélection`), ôte l'objet cliqué de la sélection. Cliquer sur le fond de la fenêtre (commande `vider sélection`) a pour effet de désélectionner tous les objets.

Une autre possibilité pour sélectionner ou désélectionner un ensemble d'objets d'un coup est de choisir la commande "Outil de Sélection" et de tracer un rectangle autour des objets concernés (comme avec un lasso).

2.2.3 Redimensionner et déplacer des objets

Nous avons déjà mentionné le fait qu'un objet sélectionné était entouré de huit "poignées". Cliquer et déplacer l'une des poignées change la taille de l'objet (commande `changer taille`). Cliquer sur l'une des poignées situées sur le côté a pour effet de

redimensionner l'objet selon un axe unique. Certains objets ont une taille minimum et ce facteur est pris en compte lors du redimensionnement. De plus, si plus d'un objet est sélectionné, tous les objets sont redimensionnés avec un même facteur.

Pour déplacer un objet sélectionné, l'utilisateur clique dessus, le déplace avec la souris et le relâche à la position désirée (commande `déplacer`). Là encore, si plusieurs objets sont sélectionnés, ils sont tous translatés d'un même vecteur.

2.2.4 Amener des objets au premier plan/arrière plan

Pour amener des objets au premier plan (commande `mettre au dessus`), c'est-à-dire amener des objets au dessus de tous les autres, l'utilisateur sélectionne ceux-ci puis choisit le menu "Mettre au Dessus". De façon similaire, pour amener des objets en arrière plan (commande `mettre en dessous`), l'utilisateur les sélectionne puis choisit le menu "Mettre au Dessous".

2.2.5 Connecter et déconnecter des objets

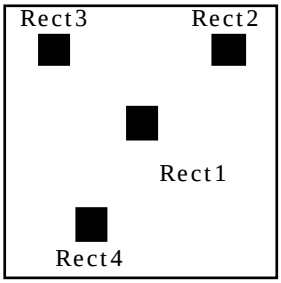
Deux objets sélectionnés peuvent être connectés en sélectionnant le menu "Connecter" (commande `connecter`). Lorsque deux objets sont connectés, une ligne les relie comme le montre la figure 4.2. Déplacer un objet connecté a pour effet de déplacer les connexions en conséquence. Effacer un objet le déconnecte automatiquement de tous les autres objets. Pour déconnecter deux objets connectés, l'utilisateur choisit le menu "Déconnecter" (commande `déconnecter`).

3 Programmation par démonstration dans SPII

3.1 Création et invocation de macros

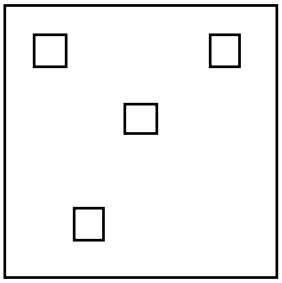
La figure 4.4 comment l'utilisateur crée la macro "Etoile" : la tâche consiste à connecter le premier objet de la sélection à tous les autres objets sélectionnés. Pour cela l'utilisateur sélectionne un certain nombre d'objets qui formeront les arguments de la macro (4.4.a), choisit l'option "Enregistrer Macro" dans le menu "Macros" puis fournit le nom de la macro. Le nom de la macro est alors ajouté à la liste des macros affichées sous le menu "Macros". Ensuite l'utilisateur désélectionne tous les objets (commande `vider sélection`, figure 4.4.b) et sélectionne le rectangle Rect1, qui était auparavant le premier objet de la sélection (4.4.c). Puis, pour chaque objet restant (c.-à-d. Rect2, Rect3 et Rect4), l'utilisateur sélectionne successivement l'objet, appelle la commande `connecter` et désélectionne l'objet (4.4.d-f). Enfin (4.4.g), lors de la dernière étape l'utilisateur sélectionne à nouveau les objets Rect2, Rect3 et Rect4 de telle sorte que la valeur de retour de la macro soit la même que sa valeur d'entrée (nous verrons pourquoi dans la macro suivante). A partir de la trace des commandes utilisateur le système

généralise un programme qui connectera successivement le premier élément de la sélection aux éléments restants (première boucle), puis qui sélectionnera à nouveau tous les éléments (deuxième boucle). Des détails sur la façon dont le programme est synthétisé seront fournis dans la quatrième section.



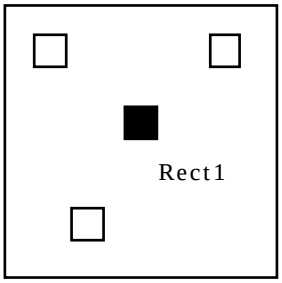
a.

Entrées :
#(Rect1 Rect2 Rect3 Rect4)
Une collection ordonnée de rectangles.



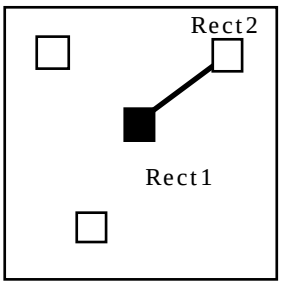
b.

`Vider Sélection`



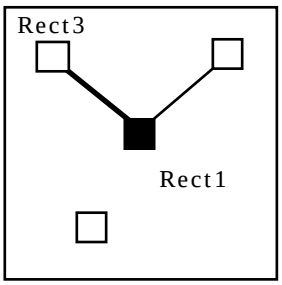
c.

`Sélectionner Rect1`



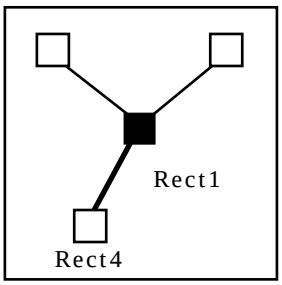
d.

```
Etendre Sélection Rect2
Connecter
Réduire Sélection Rect2
```



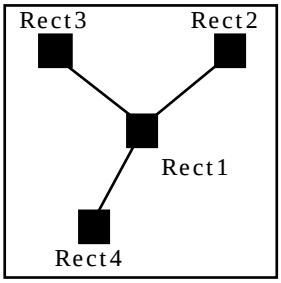
e.

```
Etendre Sélection Rect3
Connecter
Réduire Sélection Rect3
```



f.

```
Etendre Sélection Rect4
Connecter
Réduire Sélection Rect4
```



g.

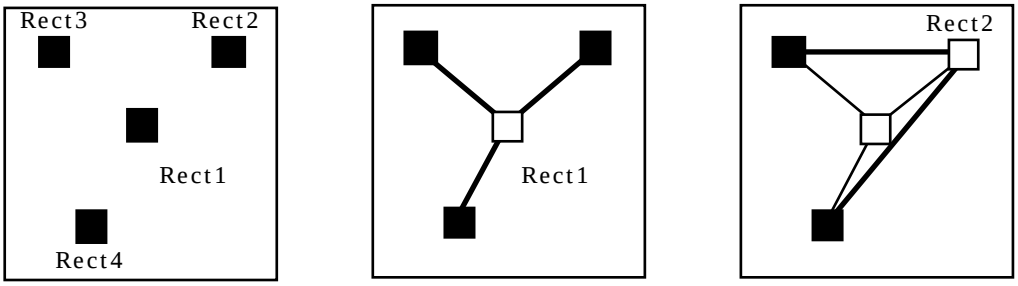
Sorties :
#(Rect1 Rect2 Rect3 Rect4)

```
Etendre Sélection Rect2
Etendre Sélection Rect3
Etendre Sélection Rect4
```

Figure 4.4 : Macro "Etoile" : connecter le premier rectangle de la sélection à tous les autres rectangles de la sélection (les objets sélectionnés sont représentés en noir).

Cette même macro peut être réutilisée dans un autre contexte avec un nombre différent d'objets. Pour ce faire, l'utilisateur sélectionnera les objets à connecter et appellera la macro "Etoile" dans le menu "Macros". L'utilisateur a la possibilité d'annuler les effets d'une macro grâce à la fenêtre historique. Lors de la phase de création de macro, il peut aussi visualiser l'historique des commandes pour s'assurer qu'il accomplit chaque itération de manière consistante de façon à faciliter la tâche du système.

Nous allons maintenant examiner comment des macros plus complexes peuvent être créées en appelant des macros plus simples et en passant les arguments et la valeur de retour au travers de la sélection. Dans cet exemple, la tâche est de connecter tous les objets de la sélection ensemble de telle sorte qu'ils forment un graphe complet. Pour cela, l'utilisateur peut utiliser la macro Etoile qu'il vient de définir. L'algorithme de la macro "Graphe Complet" consistera à appliquer successivement la macro Etoile sur les



a.

Entrées :

#(Rect1 Rect2 Rect3 Rect4)

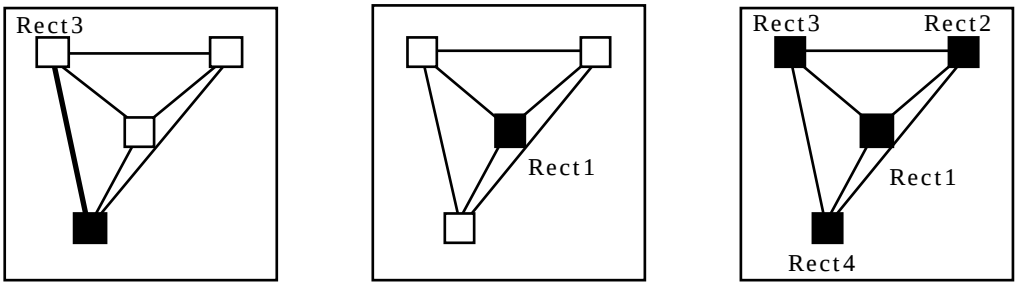
Une collection ordonnée de rectangles

b.

```
Appeler macro Etoile
Réduire Sélection Rect1
```

c.

```
Appeler macro Etoile
Réduire Sélection Rect2
```



d.

```
Appeler macro Etoile
Réduire Sélection Rect3
```

e.

```
Sélectionner Rect4
```

f.

```
Etendre Sélection Rect2
Etendre Sélection Rect3
Etendre Sélection Rect4
```

Sorties :

#(Rect1 Rect2 Rect3 Rect4)

Figure 4.5 : Macro "Graphe Complet" : connecter tous les rectangles de la sélection pour former un graphe complet.

objets de la sélection. La figure 4.5.a montre les arguments de la macro (les objets Rect1, Rect2, Rect3 et Rect4). En 4.5.b, l'utilisateur appelle la macro "Etoile" : Rect1 est alors connecté à Rect2, Rect3, et Rect4 et la sélection reste inchangée. L'utilisateur désélectionne ensuite Rect1. En 4.5.c, il appelle la macro "Etoile" ce qui connecte Rect2 (le premier élément de la nouvelle sélection) à Rect3 et Rect4. Il désélectionne ensuite Rect2. A l'étape 4.5.d, il appelle à nouveau la macro "Etoile" connectant ainsi Rect3 (le premier élément de la nouvelle sélection) à Rect4. En 4.5.e-f il resélectionne les objets un par un. Une fois généralisée, cette macro fonctionne dans des cas où le nombre de rectangles n'est pas égal à quatre. Cet exemple montre l'appel de procédure, l'utilisation des valeurs de retour et comment des boucles imbriquées peuvent être créées.

3.2 Spécifier les intentions de l'utilisateur

Spécifier les intentions de l'utilisateur est un problème crucial pour la Programmation Par Démonstration. Néanmoins, dans certains cas effectuer l'inférence correcte est impossible, même si plus d'un exemple sont fournis. C'est pourquoi plusieurs techniques ont tenté de permettre à l'utilisateur de guider le système. Metamouse [Maulsby 89a,b] utilise des outils de construction comme une ligne mobile afin de montrer les relations géométriques entre objets. Smallstar [Halbert 84] et Leda [Mima 91] autorisent l'utilisateur à expliquer la façon de sélectionner un objet en utilisant des descripteurs de données et lui permettent d'expliciter les structures de contrôle après enregistrement de la macro. Dans Moctec [Maulsby 92] et KidSim [Cypher 95], l'utilisateur peut guider l'inférence du système en établissant des relations entre objets en faisant apparaître un menu contextuel contenant un ensemble de relations prédéfinies ou en entrant des phrases en langage naturel. Nous présentons maintenant comment l'utilisateur peut tirer parti de certaines caractéristiques d'une application basée sur AIDE afin de spécifier son intention.

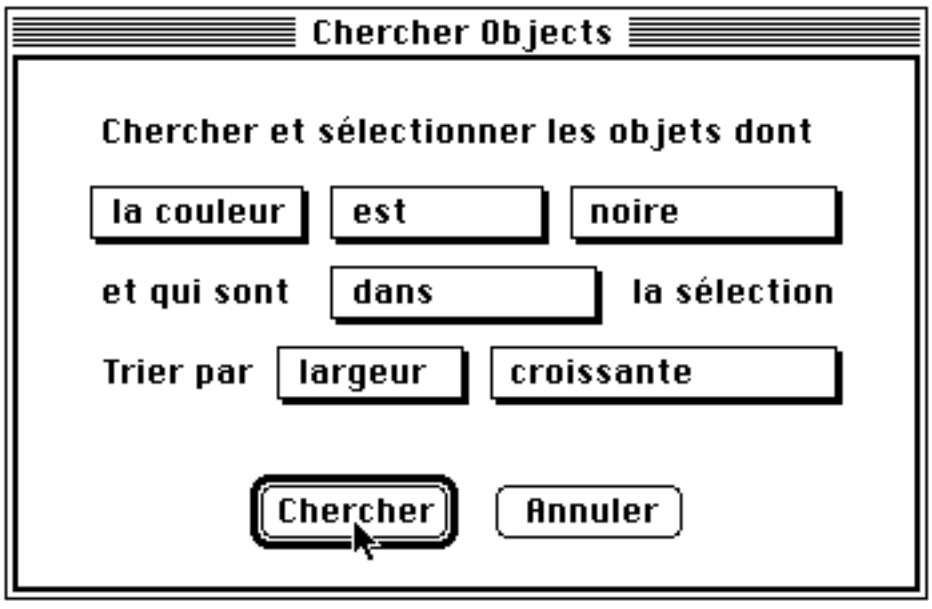


Figure 4.6 : Boîte de dialogue pour chercher des objets.

3.2.1 Montrer comment choisir un objet

Nous avons décidé d'encourager le développeur à implanter une puissante fonction de recherche dans son application en fournissant une par défaut. Avec une telle fonction, l'utilisateur est à même de sélectionner des objets vérifiants certains critères. Pour ce faire, l'utilisateur doit spécifier les critères de recherche en composant une phrase à l'aide de menus dans une boîte de dialogue (comme dans Système 7 et dans Chimera [Kurlander 90]). Dans SPII (cf. figure 4.6), la première ligne décrit le critère de recherche et est composée de trois éléments : une propriété (par exemple nom, position, couleur, taille, classe etc.), un prédicat (est, n'est pas, commence par, finit par, contient, à gauche de, à droite, est une sorte de, =, >= etc.) et enfin une valeur. A la ligne suivante, l'utilisateur spécifie où la recherche doit se dérouler (dans la sélection ou en dehors de la sélection). La dernière ligne autorise le tri des éléments ainsi sélectionnés suivant un critère donné : le premier mot spécifie la propriété à prendre en compte pour le tri tandis que le second mot indique le type de tri. Il faut noter ici que lorsque l'utilisateur sélectionne un menu donné, cela entraîne des modifications des autres menus : ainsi dans la figure 4.6, l'utilisateur a décidé d'effectuer la recherche sur la propriété couleur ; le second menu ne propose que les prédicats "est" ou "n'est pas". Si l'utilisateur avait choisi la propriété classe, le second menu aurait proposé "est", "n'est pas", "est une sous-classe de", "n'est pas une sous-classe de" etc.). Des requêtes complexes (composées de *et*, et de *ou*) peuvent être composées en combinant plusieurs appels à la fonction de recherche. Lorsque cette commande est invoquée, une commande de haut niveau est générée et enregistrée dans l'historique. Tout comme avec les descripteurs de données rencontrés dans Smallstar, cette commande véhicule de l'information concernant la façon dont un ou plusieurs éléments ont été sélectionnés. Par exemple, pour désigner la plus petite ellipse noire dans la fenêtre courante l'utilisateur pourrait effectuer les commandes du tableau 4.1.

Etape	Commande	Commentaire
1	`Vider Sélection`	la sélection est vidée

2	`Chercher Objets`
	`dont la [classe] [est] [ellipse]`
	`et qui sont [en dehors de] la sélection`
		de telle sorte que la recherche ait lieu dans toute la fenêtre

3	`Chercher Objets`
	`dont [la couleur] [est] [noire]`
	`et qui sont [dans] la sélection`
	`Trier par [largeur] [croissante]`
		maintenant, la recherche doit être effectuée parmi les ellipses déjà sélectionnées

Tableau 4.1 : Macro : trouver la plus petite ellipse noire.

3.2.2 Spécifier les structures de contrôle

Les structures de contrôle telles que les boucles imbriquées, les appels de procédure ou les récursions sont difficiles à inférer, c'est pourquoi guider le système est important. Dans une application basée sur AIDE, la possibilité d'appeler une macro à partir d'une autre permet à l'utilisateur de scinder son algorithme en parties plus petites, rendant ainsi la tâche plus facile au système. Par exemple, l'utilisateur aurait pu enseigner la macro "Graphe Complet" sans créer la macro "Etoile", mais la taille de l'exemple aurait été plus grande et le système aurait eu à inférer des boucles imbriquées au lieu de boucles simples. Nous pensons que "découper" les macros est un moyen plus naturel que d'ajouter des structures de contrôle *après* enregistrement de la macro comme dans Smallstar ou Leda. De plus, ce découpage correspond au concept de programmation structurée, favorisant la réutilisation du code.

3.2.3 Fournir les bons arguments et limiter la recherche

Cependant, fournir explicitement les arguments d'une commande est parfois la seule solution. Mais puisque le mécanisme d'enregistrement de macros fonctionne au travers de toutes les applications, des relations complexes entre des objets peuvent être spécifiées en copiant et collant des objets d'une application à une autre. Par exemple, comme c'est le cas dans Smallstar, l'utilisateur pourrait utiliser l'application calculatrice pour créer des formules reliant entre elles des coordonnées de plusieurs objets.

Comme nous l'avons expliqué dans la section 3.1, la sélection courante — avant l'enregistrement d'une macro — constitue le centre d'intérêt pour le mécanisme de généralisation. L'inférence du système se fait par rapport à cette liste ce qui permet de réduire l'espace de recherche (ainsi dans SPII, la recherche des objets se fait dans la sélection plutôt que dans toute la fenêtre).

4 Implantation de SPII

4.1 Classes de l'éditeur

La figure 4.7 illustre la hiérarchie des principales classes utilisées par SPII. Les classes en grisé sont celles fournies par l'environnement Smalltalk de base ; celles en gras sont celles qui ont été implantées spécifiquement pour l'application SPII ; les classes restantes font partie d'AIDE. Dans le reste de cette section nous présentons brièvement les principales classes qui ont été implantées pour SPII ainsi que les relations qu'elles entretiennent avec celles qui font partie d'AIDE.

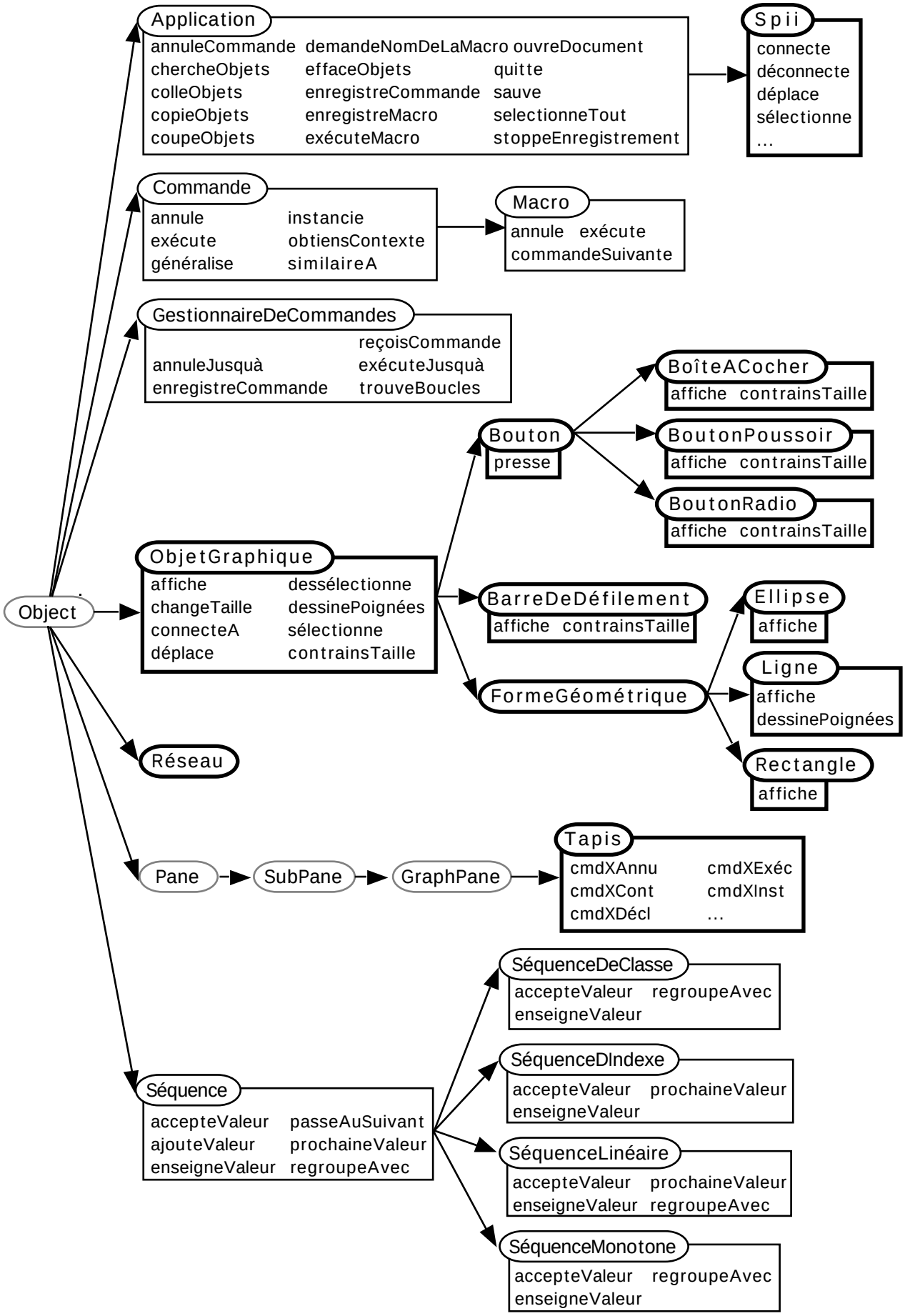


Figure 4.7 : Hiérarchie des classes dans AIDE et SPII.

4.1.1 Classe `Spii`

Cette classe représente l'application SPII dans son intégralité : son rôle est de coordonner les différents éléments de l'éditeur (c'est-à-dire les fenêtres — instances de la classe `Tapis` —, les objets graphiques — descendants de `ObjetGraphique` — et les connexions — représentées par l'objet `Réseau`) et de communiquer avec AIDE. `Spii` est une sous-classe d'`Application`, une classe d'AIDE offrant un nombre limité de services communs à toutes les applications (enregistrement de macro, gestion de documents, couper/coller, fermeture de l'application etc.).

4.1.2 Classe `ObjetGraphique`

`ObjetGraphique` est une classe abstraite (c'est-à-dire qu'elle n'a pas d'instances) décrivant le comportement des objets affichés par l'éditeur. Ainsi, un objet graphique doit savoir s'afficher, se redimensionner, dessiner ses poignées etc. Cette classe possède trois sous-classes qui sont `Bouton`, `BarreDeDéfilement` et `FormeGéométrique`. De même, `Bouton` possède trois sous-classes (`BoîteACocher`, `BoutonPoussoir` et `BoutonRadio`) et `FormeGéométrique` possède trois sous classes (`Ellipse`, `Ligne` et `Rectangle`). Ces classes implantent ou réimplantent certains des services fournis par leur super classe.

4.1.3 Classe `Réseau`

SPII représente de façon interne les objets affichés et leurs connexions sous la forme d'un objet Smalltalk, instance de la classe `Réseau`. Les services offerts par cette classe incluent l'ajout et le retrait de connexions ou d'objets. Lorsque l'utilisateur manipule les objets dans la fenêtre, les changements sont transmis à l'objet `Réseau` qui est mis à jour en conséquence.

4.1.4 Classe `Tapis`

Cette dernière classe forme le coeur de SPII : elle implante la fenêtre dans laquelle l'utilisateur manipule les objets. Ainsi, l'objet `Tapis` reçoit-il toutes les commandes de bas niveau en provenance de l'utilisateur, détermine la commande qu'il doit générer (grâce aux méthodes de déclenchement) puis la passe au gestionnaire de commandes (instance de la classe `GestionnaireDeCommandes`) pour exécution. C'est aussi dans cette classe que les méthodes d'annulation, d'exécution, d'instanciation et d'obtention de contexte sont définies pour chacune des commandes. Dans la version actuelle de SPII, les commandes de l'éditeur sont toutes des instances de la classe `Commande` (classe définie dans AIDE). Dans la dernière version d'AIDE chaque commande est associée à une classe différente ce qui permet de déplacer les méthodes d'annulation, d'exécution, d'instanciation et d'obtention de contexte dans la classe de la commande elle-même.

4.2 Processus de généralisation

Le tableau 4.2 montre l'historique généré par l'utilisateur lors de la définition de la macro "Etoile", c'est-à-dire est une collection de commandes enrichies — des commandes dont l'argument de contexte est rempli. L'étape 0 représente la sélection avant l'enregistrement de la macro. Pour les autres étapes, nous fournissons le `nom` de la commande, les variables `exécArg` (objets manipulés) et `contArg` (arguments de contexte). Dans la colonne `contArg`, les index depuis le début et depuis la fin font référence à la position des objets sélectionnés ou désélectionnés en partant depuis le début ou la fin de la sélection initiale (étape 0).

Etape	**Figure**	`nom` (de la commande)	`exécArg`	`contArg` (indexe dans les arguments)	
				depuis le début	depuis la fin
0	4.4.a	`#(Rect1 Rect2 Rect3 Rect4)`			
1	4.4.b	`Vider Sélection`			
2	4.4.c	`Sélectionner`	`Rect1`	1	4
3	4.4.d	`Etendre Sélection`	`Rect2`	2	3
4	4.4.d	`Connecter`			
5	4.4.d	`Réduire Sélection`	`Rect2`	2	3
6	4.4.e	`Etendre Sélection`	`Rect3`	3	2
7	4.4.e	`Connecter`			
8	4.4.e	`Réduire Sélection`	`Rect3`	3	2
9	4.4.f	`Etendre Sélection`	`Rect4`	4	1
10	4.4.f	`Connecter`			
11	4.4.f	`Réduire Sélection`	`Rect4`	4	1
12	4.4.g	`Etendre Sélection`	`Rect2`	2	3
13	4.4.g	`Etendre Sélection`	`Rect3`	3	2
14	4.4.g	`Etendre Sélection`	`Rect4`	4	1

Tableau 4.2 : Historique pour la macro "Etoile".

D'autres arguments de contexte existent pour les commandes affectant la sélection, mais ils ont été laissés de côté car ils n'étaient pas significatifs dans cet exemple. De tels arguments incluent des propriétés comme la classe, la position, la couleur, la taille, etc. de l'objet sélectionné. Dans la macro "Etoile", le critère pour sélectionner ou déselectionner un objet était sont index par rapport à la sélection initiale.

Le tableau 4.2 illustre la généralisation des commandes enrichies. Le gestionnaire de commandes trouve une boucle en groupant ensemble les étapes 3/6/9, 4/7/10 et 5/8/11, ainsi qu'une seconde boucle en regroupant les étapes 12, 13 et 14.

Etapes Groupées	`nom` (de la commande)	`contArg` (indexe dans les arguments)		
		depuis le début	depuis la fin	
1	`Vider Sélection`			
2	`Sélectionner`	de 1 à 1 par 0	de 4 à 4 par 0	▶ constante : 1
3 6 9	`Etendre Sélection`	de 2 à 4 par 1	de 3 à 1 par -1	▶ de 2 à nombre d'arguments par 1
4 7 10	`Connecter`			
5 8 11	`Réduire Sélection`	de 2 à 4 par 1	de 3 à 1 par -1	▶ de 2 à nombre d'arguments par 1
12 13 14	`Etendre Sélection`	de 2 à 4 par 1	de 3 à 1 par -1	▶ de 2 à nombre d'arguments par 1

Tableau 4.2 : Généralisation de l'historique correspondant à la macro "Etoile".

Dans le cas de la dernière boucle (dernière ligne de le tableau 4.2), la généralisation de l'argument de contexte "depuis le début" signifie que :

- les objets de la seconde à la quatrième position de la sélection (comptés à partir du début de la sélection) sont successivement sélectionnés.

Tandis que la généralisation de l'argument "depuis la fin" signifie que :

- les objets de la troisième à la première position de la sélection (comptés à partir de la fin de la sélection) sont successivement sélectionnés.

En combinant ces deux généralisations lorsque la commande est exécutée, le gestionnaire de commandes génère des commandes sélectionnant les objets allant de la seconde à la *dernière* position de la sélection (en partant du début).

Chaque étape du programme est stockée dans l'historique comme une liste, et une fois détectées, les boucles sont stockées comme des sous-listes. Dans notre exemple, la macro est représenté de façon interne par la liste suivante :

```
(
  Vider Sélection
  Sélectionner   <1>
  (
    Etendre Sélection    <2 à nombre d'arguments par 1>
    Connecter
    Réduire Sélection    <2 à nombre d'arguments par 1>
  )
  (
    Etendre Sélection    <2 à nombre d'arguments par 1>
  )
)
```

Figure 4.7 : Représentation interne de la macro "Etoile".

Les figures 4.8, 4.9 et 4.10 illustrent le flot de communication entre AIDE et l'application SPII dans les trois modes décrits dans le chapitre 3 section 3. Les méthodes en caractère gras sont celles qui appartiennent à SPII tandis que celles en caractère normal font partie d'AIDE. Il est apparent au vu de ces figures qu'AIDE et SPII se partagent le travail et que le développeur doit écrire le minimum de méthodes pour supporter la programmation par démonstration.

```
...
Tapis>>sélectionnerDécl(    unTapis ,unObjetCliqué  )
  cmd := Commande>>créer(unTapis,unObjetCliqué,...)
   ** crée une commande et remplit les variables propriétaire (unTapis), exécArg (unObjetCliqué) et
   annuArg (les objets précédemment sélectionnés) **
  GestionnaireDeCommandes>>reçoisCommande(LeGestDeCmd,cmd)
    GestionnaireDeCommandes>>enregistreCommande(LeGestDeCmd,cmd)
    Commande>>exécute(cmd)
       Tapis>>sélectionnerExéc(    unTapis ,unObjetCliqué  )
         ObjetGraphique>>sélectionne(     unObjetCliqué  )
...
```

Figure 4.8 : Trace d'exécution en mode exécution de commande.

```
Application>>enregistreMacro(applicationSPII)
  nomDeLaMacro := Application>>demandeNomDeLaMacro(applicationSPII)
  GestionnaireDeCommandes>>enregistreMacro(nomDeLaMacro)
...
Tapis>>sélectionnerDécl(    unTapis ,unObjetCliqué  )
  cmd := Commande>>créer(unTapis,unObjetCliqué,...)
   ** crée une commande et remplit les variables propriétaire (unTapis), exécArg (unObjetCliqué) et
   annuArg (les objets précédemment sélectionnés) **
  GestionnaireDeCommandes>>reçoisCommande(LeGestDeCmd,cmd)
    GestionnaireDeCommandes>>enregistre(LeGestDeCmd,cmd)
    Commande>>exécute(cmd)
      Tapis>>sélectionnerExéc(    unTapis ,unObjetCliqué  )
        ObjetGraphique>>sélectionne(     unObjetCliqué  )
    Commande>>obtiensContexte(cmd)
      Tapis>>sélectionnerCont(    unTapis ,cmd )
        cmd.contArg := SéquenceDIndexe>>créer(position de
              unObjetCliqué depuis le début et la fin de la sélection)
    GestionnaireDeCommandes>>trouveBoucles(LeGestDeCmd)
      Appelle la méthode généMth pour regrouper les commandes
...
Application>>stoppeEnregistrement(applicationSPII)
  GestionnaireDeCommandes>>stoppeEnregistrement(LeGestDeCmd,
                                                nomDeLaMacro)
    commandes := GestionnaireDeCommandes>>commandes(LeGestDeCmd)
    macro := Macro>>créer(nomDeLaMacro,commandes)
```

Figure 4.9 : Trace d'exécution en mode enregistrement de macro.

```
Application>>exécuteMacro(nomDeLaMacro)
 macro := Application>>trouveMacro(nomDeLaMacro)
 Macro>>exécute(macro)
   ...
   cmd := Macro>>commandeSuivante(macro)
   Commande>>instancie(cmd)
     Tapis>>sélectionnerInst(    unTapis ,cmd )
       cmd.exécArg :=
         Tapis>>ObjetAyantPourIndexDansLaSélectionInitiale(
         SéquenceDIndexe>>prochaineValeur(cmd.contArg))
   Commande>>exécute(cmd)
     Tapis>>sélectionnerExéc(    unTapis ,cmd )
       ObjetGraphique>>sélectionne(     unObjetCliqué   )
   Macro>>ajouterSousCommande(macro,cmd)
   ...
```

Figure 4.10 : Trace d'exécution en mode exécution de macro.

5 Conclusion

Dans ce chapitre nous avons présenté l'application SPII, un éditeur graphique bâti sur AIDE utilisant le paradigme objet. Les caractéristiques majeures de SPII sont étroitement liées à celles d'AIDE : utilisation de l'historique à des fins de correction d'erreur et de création de macro, utilisation de la sélection pour le passage d'arguments et de valeurs de retour entre macros, fonction de recherche d'objets pour spécifier les intentions de l'utilisateur.

D'un point de vue implantation, SPII réutilise beaucoup de éléments présents dans AIDE pour généraliser les macros. Le seul travail supplémentaire demandé au développeur est de définir le comportement complet des commandes de l'application. Les fonctionalités de base de l'application et les capacités de Programmation Par Démonstration ont été bien séparées. L'implantation de SPII a montré dissociation entre AIDE et l'application utilisant ses services. La communication entre AIDE et SPII est effectuée au travers des commandes échangées.

Chapitre 5

Conclusion : résultats et évaluation

La macrocommande est aux langages d'interaction ce que la procédure est aux langages de programmation. C'est un mécanisme d'abstraction et une technique d'extension. En tant que mécanisme d'abstraction, elle répond au phénomène cognitif de l'apprentissage qui, avec l'expérience, encapsule dans un mnène un ensemble organisé de connaissances plus élémentaires. En tant que technique d'extension, elle permet de concilier le particularisme et la généralité.

...

La construction par démonstration est une façon de raccourcir la distance sémantique qui sépare l'objet conceptuel à réaliser de l'expression de son élaboration. Cette technique, bien qu'attrayante, n'est pas toujours facile à mettre en oeuvre. Dans Emacs, les constituants de la construction sont des commandes textuelles non ambiguës. Dans le cas des interfaces graphiques, la gestuelle est plus difficile à interpréter. Il est clair que des travaux méritent d'être entrepris dans cette direction.

Joelle Coutaz [Coutaz 90]

1 Introduction

Le propos de cette thèse est de proposer une couche logicielle sur laquelle des développeurs peuvent venir construire des applications écrites en Smalltalk mettant en oeuvre des techniques démonstrationnelles. AIDE est une telle couche, fournissant un certain nombre d'éléments pour faciliter et accélérer la création de systèmes intégrant la Programmation Par Démonstration. AIDE étant un squelette d'application destiné aux

développeurs implantant des applications proposant à l'*utilisateur* la création de macros par démonstration, son évaluation a été effectuée sous deux angles différents. Dans un premier temps nous avons mené une évaluation empirique du squelette du point de vue du développeur, en nous basant sur l'expérience acquise lors du développement d'une application conséquente (SPII, chapitre 4) et en étudiant en quoi l'architecture d'AIDE a supporté cet effort. Dans un second temps, nous avons effectué une évaluation de l'application SPII grâce à des tests menés auprès des utilisateurs dans le but de vérifier que les fonctionnalités proposées par AIDE satisfont aux besoins des utilisateurs.

Dans la seconde section de ce chapitre nous évaluons donc ce squelette du point de vue du développeur en vérifiant qu'il aide effectivement les développeurs dans leur tâche puis dans la troisième section nous l'évaluons du point de vue de l'utilisateur en testant si les fonctionnalités qu'il propose sont adaptées aux besoins de l'utilisateur. Dans la quatrième section nous concluons en résumant l'importance des contributions et en indiquant de nouveaux axes de recherche.

2 Evaluation empirique coté développeur

Evaluer l'utilisation du squelette revient à déterminer le gain de productivité fournit au développeur dans son travail de programmation. Cette évaluation est une tâche difficile pour deux raisons majeures. La première raison est logistique et tient au fait que pour mesurer le gain de productivité il faut non seulement avoir un échantillon conséquent de développeurs ayant utilisé le système sur une période importante, mais aussi comparer le temps mis pour implanter une application donée avec et sans le squelette. Dans le cas particulier d'AIDE, seul l'auteur a utilisé le squelette durant une période suffisamment longue pour être représentative (une autre personne utilise le squelette depuis). La seconde difficulté est liée à l'absence de méthodes rigoureuses pour évaluer un squelette d'application : Ainsi Coutaz [Coutaz 90] passe en revue un certain nombre de squelettes et les évalue en termes d'extensibilité et de réutilisabilité mais ne fournit aucunes données quantitatives. L'auteur a décidé d'évaluer le squelette selon deux angles: le premier consiste à justifier les choix d'implantation effectués et le second à mesurer le gain de productivité fournit par l'utilisation d'AIDE pour l'implantation d'une application particulière (SPII).

2.1 Evaluation qualitative de l'architecture

Le premier bénéfice procuré par AIDE est qu'il montre au développeur la façon dont une application mettant en oeuvre la Programmation Par Démonstration est structurée. Il est cependant important de noter que, comme c'est le cas avec la plupart des squelettes d'application, AIDE ne fait gagner du temps au développeur que lorsque celui-ci écrit une application en utilisant le squelette *dès le départ*. AIDE n'est pas un outil permettant d'ajouter facilement des capacités de Programmation Par Démonstration à

une application existante. Nous passons maintenant en revue les principales décisions effectuées en les justifiant.

2.1.1 Représentation des commandes sous forme d'objets

Dans de nombreux squelettes d'application tels que MacApp ou MotifApp [Young 92], une commande d'application est représentée comme un objet gérant l'invocation, l'exécution et l'annulation de celle ci. En plus d'être une notion familière pour les développeurs, la représentation des commandes sous forme d'objets présente de nombreux avantages. Certaines commandes possèdent un état ou des données associées tandis que d'autres peuvent impliquer un certain nombre de fonctions. Dans les deux cas, une représentation sous forme d'objet permet de regrouper les données et les fonctions associées à une commande dans une entité unique. Cette représentation répond également aux principes d'enregistrement de commandes de haut niveau, d'extensibilité et de correction d'erreur spécifiés dans la seconde section du troisième chapitre.

2.1.2 Représentation des commandes sous forme arborescente

SPII ayant été implanté avec une ancienne version d'AIDE n'incluant pas les arbres de commandes, nous avons rapidement prototypé un éditeur de texte (en adaptant à nos besoin celui fourni par l'environnement SmalltalkAgents) dont les commandes sont représentées comme des arbres. Il s'est avéré qu'une telle représentation offre un support amélioré pour la correction d'erreur, et ceci pour deux raisons principales :

- l'utilisateur est à même d'annuler ou de ré-exécuter des commandes à des niveaux d'abstraction différents ;

- la navigation dans l'historique est grandement accélérée lorsque les commandes de niveau intermédiaire implantent elles-mêmes les méthodes `exécute` et `annule`. Nous avons effectué un test comparant le temps mis pour annuler les 100 dernières commandes effectuées dans l'éditeur de texte avec et sans définition de la méthode `annule` dans les commandes intermédiaires. Dans le second cas, le gain a été d'un facteur 5 ce qui s'explique, d'une part, par le fait que l'annulation d'une commande de haut niveau requiert moins d'étapes que l'annulation de ses sous-commandes une par une (puisqu'elle stocke l'état initial avant exécution) et que, d'autre part, le rafraîchissement de l'écran n'a pas lieu à chaque fois qu'une commande feuille est annulée.

Nous n'avons par contre pas validé notre hypothèse comme quoi une telle représentation offre un meilleur support pour l'inférence. Kosbie [Kosbie 95] et Nevill-Manning [Nevill-Manning 94] ont depuis confimé cette hypothèse.

2.1.3 Gestionnaire de commandes

Comme nous l'avons déjà vu, le gestionnaire de commande prend en charge la gestion de l'historique ainsi que la création et l'exécution des macros. Le fait que toutes les applications doivent envoyer leurs commandes à une même entité — le gestionnaire de commandes — permet de créer des macros contrôlant plusieurs applications. Ceci serait impossible si chaque application gérait elle-même son historique de façon indépendante.

Dans le chapitre précédent (figures 4.8 à 4.10) nous avons présenté la trace lors de l'enregistrement et l'exécution d'une macro : il apparaît que les tâches sont biens distribuées entre AIDE et l'application : AIDE implante les mécanismes de base comme l'enregistrement de commandes, la création et l'exécution de macro, la navigation dans l'historique et fait appel à SPII pour les comportements spécifiques à l'application. Cette séparation facilite le travail du développeur qui se concentre sur son application et écrit un nombre minimal de méthodes pour que son application puisse tirer parti d'AIDE.

2.2 Gain de productivité dans le cas de SPII

Tout comme cela aurait été le cas avec les squelettes MacApp ou MotifApp, nous avons implanté pour chaque commande (`cmdX`) les commandes de SPII (voir chapitre 4, section 4) en définissant les méthodes `cmdXAnnu` (annulation), `cmdXExéc` (exécution), et `cmdXDécl` (déclenchement) dans la classe `Tapis`. Le seul effort supplémentaire requis pour incorporer les capacités de programmation par démonstration a été d'implanter les méthodes supplémentaires requises par AIDE, comme `cmdXInst` et `cmdXCont` ce qui constitue environ 5% du code de SPII.

	Nombre de lignes
SPII	
Corps de l'application	2000
Objets graphiques	2200
PPD	100
Total	4300
AIDE	
Commandes & macros	400
Gestionnaire de commandes	500
Séquences	500
Interface	400
Total	1800
Total Général	6100

Tableau 5.1 Tailles respectives du code pour les différentes parties d'AIDE et de SPII.

Le tableau 5.1 montre le nombre de lignes correspondant à SPII et AIDE. Il apparaît de façon criante que le code de SPII chargé de la création de macros par démonstration représente moins de 2.5% du code total ou 5% si l'on ne tient pas compte des divers classes d'objets graphiques qui ont été crées. Il faut également contraster ce chiffre avec la taille de code nécessaire pour implanter AIDE. En effet si le développeur devait implanter SPII sans AIDE, le nombre de lignes de code nécessaires à l'implantation des capacités ce PPD serait du même ordre de grandeur que le code pour le corps d'AIDE.

3 Evaluation de SPII auprès des utilisateurs

3.1 Objectifs

L'évaluation d'un système de Programmation Par Démonstration doit se faire selon deux axes : la facilité d'utilisation et la qualité de l'inférence.

Evaluer la facilité d'utilisation

La facilité d'utilisation a été testée en déterminant si l'utilisateur est capable de construire un programme : ceci est testé en mesurant le temps mis pour créer une macro donnée mais aussi en observant les difficultés rencontrées par l'utilisateur. Un autre facteur important est la rapidité du système ainsi que le retour qu'il fournit a l'utilisateur. Les points que nous désirons tester sont :

- *utilisation de la sélection* : la sélection est-elle une bonne solution pour passer les arguments à une macro et retourner une valeur ?

- *fenêtre de l'historique* : l'utilisateur utilise-t-il cette fenêtre pour naviguer dans l'historique et pour vérifier les commandes qu'il vient d'effectuer ?

- *rapidité d'exécution* : le temps mis par le système pour généraliser le programme est-il suffisamment court ?

Evaluer si la qualité de l'inférence est suffisante

Un second élément très important est la qualité des programmes généralisés par le système. Les programmes générés sont-ils assez suffisamment généraux, sont-ils corrects? Nous désirions de plus tester les points qui suivent :

- le découpage d'une macro en une macro plus simple est-il un moyen approprié de créer des boucles imbriquées ?

3.2 Méthodologie

Afin de d'étudier la perception qu'ont les utilisateurs de SPII, nous avons mené à bien une série de tests. Les tests auprès des utilisateurs constituent une méthode empirique pour découvrir les principaux problèmes d'utilisation d'une interface. Ces types de tests possèdent cinq caractéristiques communes :

1. Le but est de trouver des problèmes d'utilisation ;
2. Les sujets des tests font parti des utilisateurs potentiels du produit ;
3. Les sujets effectuent des tâches typiques que le produit est supposé tester ;
4. L'organisateur du test observe l'utilisateur et note ses performances ;
5. L'organisateur effectue un diagnostique des problèmes d'utilisation découverts et propose des solutions.

Après l'expérience acquise lors de la compétition d'interface graphique organisée par Apple Computer [Piernot 95a], nous avons décidé de tester l'interface de SPII auprès des utilisateurs en suivant la même méthode de test. Dans cette méthode dite d'"Intervention Active" [Dumas 93], la personne organisant le test s'assoit a coté du sujet et observe ce qui se passe. Il est demandé au sujet de penser à haute voix en même temps qu'il effectue sa tâche. Pendant les tâches destinées à tester l'interface, l'organisateur du test se contente d'observer le sujet. Cependant à des moments prédéfinis, l'organisateur pose des questions pour élucider certains points. Cette méthode était d'autant plus adaptée pour tester SPII que l'application manquait de robustesse et donc la supervision de l'auteur était requise.

La première étape a consisté à choisir des sujets : pour ce faire l'auteur a identifié cinq sujets dans son entourage allant de la personne n'ayant jamais programmé au développeur confirmé. Les sujets appartiennent aux deux sexes et font partie de la tranche d'age 21-35 ans. Des études récentes ont montré que 80% des problèmes d'utilisation sont découverts lorsque les tests sont effectués avec cinq sujets [Virzi 90], [Virzi 92].

Les tests ont été conduits de la façon suivante. Dans un premier temps, nous avons laissé l'utilisateur se familiariser avec l'éditeur graphique, puis nous avons vérifié que toutes les commandes étaient maîtrisées. Ce dernier point est important car nous ne cherchons pas à tester les fonctions de base de l'éditeur mais les capacités de Programmation Par Démonstration qu'il procure. Dans un second temps, nous avons fourni à l'utilisateur un feuillet expliquant comment créer et invoquer des macros dans SPII et décrivant les inférences dont le système est capable (figure 5.1). Nous lui avons ensuite demandé de créer trois macros données allant en difficulté croissante après lui avoir montré comment créer une macro sur un exemple particulier. Pour chaque tâche, l'utilisateur devait créer le programme automatisant cette tâche puis l'exécuter sur un exemple de son choix. Durant le test les commentaires de l'utilisateur étaient notés. Enfin, dans un dernier temps nous avons interrogé l'utilisateur pour savoir ce qu'il pensait du système et quelles étaient les difficultés qu'il avait rencontrées.

Instructions pour SPII

1) Création d'une macro:

Dans SPII une macro est une commande *définie par l'utilisateur* opérant sur les objets sélectionnés dans la fenêtre courante, comme n'importe quelle autre commande. Pour créer une macro, commencer par sélectionner les objets sur lesquels celle-ci portera, choisir l'option "**Enregistrer Macro**" dans le menu "**Macros**", puis entrer le nom de celle-ci lorsque SPII le demande (une fois le nom entré il apparaît sous le menu "**Macros**"). Démontrer ensuite les étapes de la macro en accomplissant les commandes nécessaires sur les objets de la sélection. Terminer en choisissant l'option "**Stopper Enregistrement**" dans le menu "**Macros**". La macro est alors prête à être utilisée. Note : en cas d'erreur il est possible d'annuler ou de ré-exécuter des commandes en utilisant soit la fenêtre historique soit les options "**Annuler**" ou "**Refaire**" sous le menu "**Edition**".

2) Invocation d'une macro

Pour invoquer une macro, sélectionner les objets sur lesquels elles doit s'appliquer, puis sélectionner son nom dans la liste des macros présentée sous le menu "**Macros**".

3) Fonctionnement de SPII

Une fois une macro enregistrée, SPII crée un programme généralisant celle-ci en essayant de trouver des répétitions dans la liste des commandes qui viennent d'être enregistrées. Lorsqu'une répétition est trouvée, il génère une boucle automatisant la répétition. Pour créer une boucle, SPII se base sur la position des objets par rapport à la sélection initiale. Note : il est conseillé d'accomplir les commandes de façon consistante de façon à ce que SPII puisse détecter des répétitions. Observer la fenêtre historique permet de vérifier si c'est le cas.

Figure 5.1 : Feuillet fourni aux utilisateurs décrivant le fonctionnement des macros dans SPII.

Tâche N° 0 : effacer tous les objets sélectionnés

Lors de cette étape, l'utilisateur se voit montré comment créer une macro par démonstration. Il est en effet apparu après plusieurs tests que la description du système fournie (figure 5.1) n'était pas suffisante et qu'il fallait montrer à l'utilisateur comment contourner des "bugs" du système. La macro créée est volontairement très simple (il s'agit d'effacer tous les objets sélectionnés) et cette démonstration pourrait être inclue dans le système sous la forme d'un tutoriel.

Etat initial : *Etat final :*

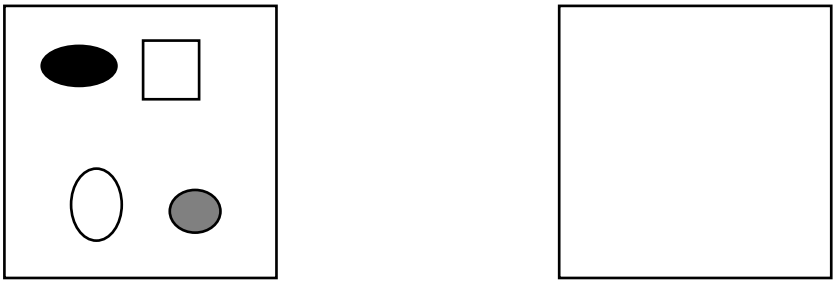


Figure 5.2 : Macro n° 0 - effacer tous les objets sélectionnés (effectuée avec l'utilisateur).

Tâche N° 1 : connecter les objets en chaîne

Dans ce premier test il est demandé de créer un programme connectant chaque élément de la sélection au suivant de façon à former une chaîne d'objets reliés entre eux comme le montre la figure 5.3.

Etat initial : *Etat final :*

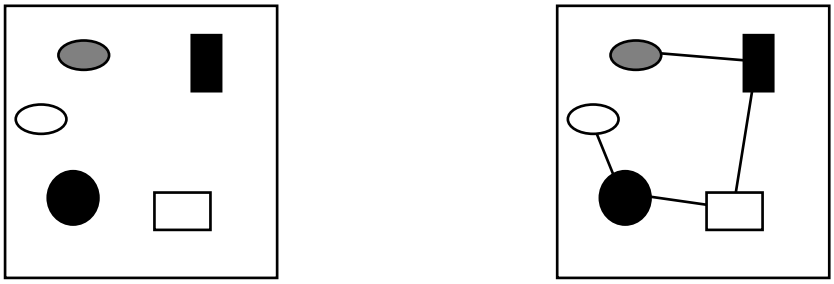


Figure 5.3 : Macro n° 1 - connecter les objets en chaîne.

Tâche N° 2: connecter les objets en étoile

Le deuxième exercice consiste à créer un programme connectant le premier élément de la sélection aux éléments restants (voir figure 5.4). Il est de plus demandé que la sélection pour l'état final soit la même que pour l'état initial (mêmes éléments et même ordre).

Etat initial : *Etat final :*

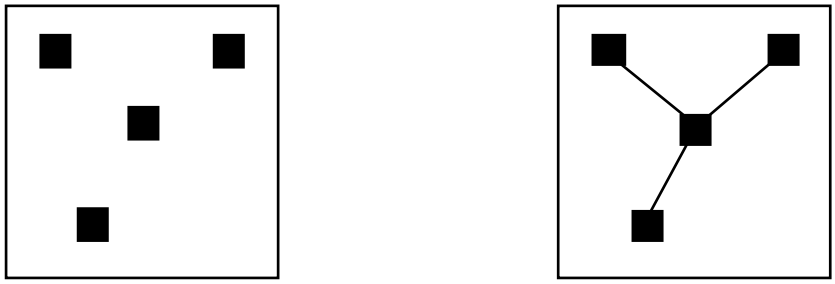


Figure 5.4 : Macro n° 2 - connecter les objets en étoile.

Tâche N° 3 : connecter tous les objets entre eux

Dans cette dernière partie, il s'agissait de connecter tous les éléments de la sélection entre-eux pour former un graphe complet (voir figure 5.5). Aucun utilisateur n'a pensé à réutiliser de lui-même la macro précédemment créée. Nous avons donc dû demander explicitement à l'utilisateur de créer cette macro en utilisant la précédente. Ceci pourrait ne pas être un problème en pratique lorsque l'utilisateur a compris que pour créer des boucles imbriquées il faut avoir recours à plusieurs macros. Cela reste néanmoins une limitation importante du système.

Etat initial : *Etat final :*

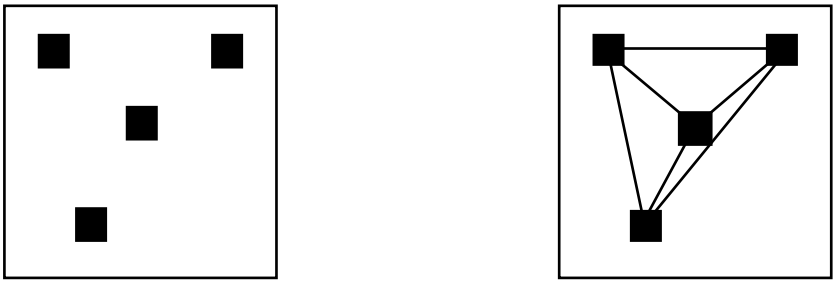


Figure 5.5 : Macro n° 3 - connecter tous les objets entre eux.

3.3 Résultats

Les tests auprès des utilisateurs ont soulignés divers problèmes :

1) L'un des problèmes les plus important que nous avons rencontré — en dehors des bugs du système — est que SPII ne représente pas l'ordre dans lequel les objets ont été sélectionnés. Ceci est une problème car AIDE est sensible à l'ordre des objets dans la sélection et ne pardonne pas des variations dans la façon dont une tâche est accomplie. Ceci pourrait être corrigé en utilisant des dégradés de couleur ou des chiffres pour indiquer l'ordre des objets dans la sélection.

2) Le seconde problème majeur réside dans le fait que SPII ne pardonne pas des erreurs dans l'ordre dont les commandes sont exécutées. C'est ce qui nous a conduit à développer d'historique hiérarchique. Une solution consisterait à réimplanter SPII avec la dernière version d'AIDE incluant les arbres de commandes.

3) Certains utilisateurs se sont plain de ne pas pouvoir visualiser le code d'une macro (seul le nom des commandes apparaît dans la fenêtre historique mais les boucles et les arguments ne figurent pas). Ce point peut être facilement corrigé en fournissant au niveau d'AIDE un langage textuel (à la Smallstar) pour représenter les macros puisque les boucles et les arguments ont une représentation interne. Il serait par contre plus difficile de fournir un mécanisme générique pour représenter l'historique et les macros sous forme graphiques dans la mesure ou AIDE devrait faire appel a l'application pour déterminer les objets à représenter.

4) SPII ne permet pas de modifier une macro existante. Une fois la macro enregistrée, il faut la recréer de toute pièce en cas d'erreur. AIDE ne supporte pas l'édition de macro.

Les problèmes 3 et 4 réclament une extension d'AIDE.

	Tâche N° 2	**Tâche N° 3**	**Tâche N° 4**
Utilisateur 1	5 min	9 min	–
Utilisateur 2	6 min	8 min	10 min
Utilisateur 3	3 min	6 min	9 min
Utilisateur 4	1 min	6 min	7 min
Utilisateur 5	1 min	3 min	4 min

Tableau 5.2 : Temps mis par les utilisateurs pour accomplir chacune des tâches.

4 Conclusion

4.1 Nouvelles voies de recherche

4.1.1 Inférence et structures de contrôle

AIDE peut être amélioré dans plusieurs directions, la plus significative étant d'enrichir les structures de contrôle qu'il est capable d'inférer. Ainsi, dans l'état actuel, AIDE n'est pas capable d'inférer la présence de boucles imbriquées, de conditions et donc il n'est pas capable de créer des procédures récursives. Un second travail important consiste à intégrer le mécanisme de généralisation à la nouvelle représentation de commandes hiérarchiques. Jusqu'à présent nous avons prouvé l'intérêt de ce modèle pour l'annulation de commande, mais du travail reste à faire pour montrer qu'il est aussi utile à la Programmation Par Démonstration.

4.1.2 Exploitation de l'historique

La fenêtre historique permet à l'utilisateur de prendre connaissance des commandes qui viennent d'être effectuées. La représentation actuelle présente néanmoins certaines limitations : seul le nom des commandes est affiché, les arguments étant omis, ce qui ne permet pas de représenter leur généralisation. D'autre part, le système pourrait montrer qu'il a détecté une boucle peut-être en soulignant les commandes dans chaque itération. Une autre amélioration consisterait à rendre cet historique éditable de telle sorte que l'utilisateur puisse modifier les inférences du système. Enfin, un représentation graphique à la matière de Chimera, Pursuit ou de Mondrian serait bien venue. La représentation des commandes sous forme d'arbre semble bien adaptée au groupement de commandes dans un même panneau.

Yvon [Yvon 94] utilise cet environnement pour ajouter des facilités de programmation par démonstration à un logiciel de télécommunication et prévoit d'étendre le modèle pour que l'utilisateur puisse chercher des informations sur l'historique en fonction de la date etc.

4.2 Adoption auprès des développeurs

Dans le cadre de cette recherche Smalltalk a été utilisé car il s'agit d'un langage de programmation par objets dynamique qui permet de prototyper rapidement une application. Du point de vue implantation, pour que l'architecture développée dans AIDE connaisse une adoption conséquente il faudrait :

- dans un premier temps, réécrire AIDE dans un langage à objets à large audience comme C++. En effet, peu d'applications disponibles dans le commerce sont écrites

en Smalltalk. Si AIDE était fourni sous la forme d'une librairie C++, chaque application pourrait bénéficier *individuellement* d'AIDE ;

- dans un second temps, intégrer AIDE aux systèmes d'exploitation existants (l'architecture des AppleEvents semble être le candidat idéal) de telle sorte que toutes les applications puissent bénéficier *simultanément* de cette architecture et ainsi avoir des capacités de Programmation Par Démonstration *étendues à tout le système*.

Dans l'éventualité d'une implémentation dans un langage statique, la représentation des commandes prendrait une dimension supplémentaire (en plus d'implanter la récupération d'erreur et la Programmation Par Démonstration) puisqu'elle fournirait le moyen de contrôler une application écrite dans un langage statique comme C++ à partir d'une autre partie du système.

Dans un environnement dynamique intégré comme Smalltalk ou Lisp, tout objet peut envoyer un message à n'importe quel autre objet, ce qui rend aisé le contrôle d'une application : il n'est pas forcément nécessaire d'avoir recours à des commandes. L'avantage d'une telle approche est que le développeur n'a pas à décider à l'avance quelles sont les fonctions de l'application qu'il désire rendre publiques. Dans un environnement statique, cette possibilité n'est pas offerte et les commandes sont la seule solution pour contrôler une application. Malheureusement la réalité commerciale veut que la majorité des applications soient écrites dans un langage comme C ou Pascal.

4.3 Résumé des contributions

AIDE présente cinq contributions au domaine de la programmation par démonstration.

Premier squelette d'application pour la PBD

La première concerne le fait qu'AIDE est le premier squelette d'application dédié aux applications de programmation par démonstration. Jusqu'à présent chaque système de PPD a été développé séparément à partir de zéro.

Création incrémentale des arbres de commandes

Le mécanisme de création de l'historique hiérarchique des commandes apparaît comme une contribution importante pour le domaine de la programmation par démonstration.

Système fonctionnant à travers les applications

La seconde contribution est d'avoir fournit le premier système permettant la programmation par démonstration fonctionnant à travers plusieurs application et intégrant une représentation des commandes à un haut niveau. Le seul système semblable est Triggers [Potter 93], encore que celui-ci manipule seulement des événements de bas niveau.

Intégration de la correction d'erreur à la PPD

AIDE est l'un des seuls systèmes avec Chimera à intégrer l'outil d'annulation de commandes avec la programmation par démonstration. Comme nous l'avons vu précédemment, la possibilité d'annuler les effets d'une macro est primordiale est totalement supportée par AIDE sans effort supplémentaire pour le développeur.

Navigation dans l'historique

L'interface que propose AIDE pour naviguer dans l'historique est intuitive pour les utilisateurs car elle utilise la métaphore familière d'un magnétoscope. Du point de vue du développeur aucun effort d'implantation n'est nécessaire puisque l'interface pour la navigation est complètement gérée par AIDE.

Commande de recherche

Enfin, la commande de recherche proposée (voir figure 4.6) est une alternative aux boîtes de dialogues fournies par Smallstar ou Leda. Les utilisateurs sont déjà familiers avec de telles commandes (par exemple avec le bureau du Macintosh) et leurs utilisation est donc plus naturelle.

Références bibliographiques

[Affinity 91] Affinity Microsystems Ltd. *Tempo II User Manual*, Boulder, CO, 1991.

[Apple 93a] Apple Computer Inc. *AppleScript Language Guide*, Addison-Wesley, Reading, MA, 1993.

[Apple 93b] Apple Computer Inc. *Inside Macintosh: Interapplication Communication*, Addison-Wesley, Reading, MA, 1993.

[Bos 92] E. Bos. "Some Virtues and Limitations of Action Inferring Interfaces", *Actes de ACM Symposium on User Interface Software and Technology, UIST '92*, ACM Press, Novembre 1992, pages 79-88.

[CE Software 93] CE Software Inc. *QuicKeys User Manual*, West Des Moines, IA, 1993.

[Coutaz 90] J. Coutaz. *Interfaces Homme-ordinateur : Conception et réalisation*, Dunod informatique, Paris, 1990.

[Cutter 87] M. Cutter, B. Halpern & J. Spiegel. *MacDraw*. Apple Computer Inc., 1987.

[Cypher 91] A. Cypher. "Eager: Programming Repetitive Tasks by Example", *Actes de CHI '91* (New Orleans, Apr. 28-May 2), ACM 1991, New York, pages 33-39. Une nouvelle version est publiée dans [Cypher 93].

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

[Cypher 95] A. Cypher, D. Smith & J. Spohrer. "End-user Programming of Simulations", *actes de CHI '95*, Denver, Colorado, Mai 1995, pages 27-34.

[Darragh 92] J. Darragh & I. Witten. *The Reactive Keyboard,* Cambridge University Press, New York, 1992.

[Dumas 93] J. Dumas & J. Redish. *A Practical Guide to Usability Testing.* Ablex Publishing Corporation, Norwood, NJ, 1993.

[Eisenberg 91] M. Eisenberg. "Programmable Applications: Interpretor Meets Interface", *A.I. Memo No. 1325,* MIT, Artificial Intelligence Laboratory, Octobre 1991.

[Fuller 89] I. Fuller. "An Overview of the HP NewWave Environment", *HP Journal,* Août 1989, pages 6-8.

[Goldberg 89] A. Goldberg & D. Robson. *Smalltalk 80: The Language,* Addison-Wesley, Reading, Mass., 1989.

[Gould 84a] L. Gould & W. Finzer. "Programming by Rehearsal", *Technical Report SCL-84-1,* Xerox Parc, Mai 1984.

[Gould 84b] L. Gould & W. Finzer. "Programming by Rehearsal: An Environment for Buiding Educational Software", *Byte,* Juin 1984. Egalement publié dans [Cypher 93].

[Greenberg 93] S. Greenberg. "Techniques for reusing activities", *The Computer User as Toolsmith: the Use, Reuse, and Organization of Computer-based Tools,* Cambridge University Press, New York, 1993, pages 40-64.

[Halbert 84] D. Halbert. "Programming by Example", *Ph.D. Thesis,* Department of Electrical Engineering and Computer Science, University of California Berkeley, 1984. Aussi publié en tant que : D. Halbert, "Programming by Example", *Technical Report OSD- T8402,* Office Systems Division, Xerox Corporation, Décembre 1984. Une description apparait également dans [Cypher 93].

[Heise 89] R. Heise. "Demonstration Instead of Programming: Focussing Attention in Robot Task Acquisition", *Research Report 89/360/22,* Department of Computer Science, University of Calgary, Calgary, Canada, Septembre 1990.

[Kosbie 93] D. Kosbie & B. Myers. "A System-Wide Macro Facility Based on Aggregate Events: A Proposal", *Watch What I Do: Programming by Demonstration,* Cypher A., ed., MIT Press, Cambridge, MA, 1993, pages 433-444.

[Kosbie 94] D. Kosbie & B. Myers. “Extending Programming By Demonstration With Hierarchical Event Histories,” *Actes de East-West Human Computer Interaction '94*, St. Petersburg, Russia, August 1994.

[Kosbie 95] D. Kosbie. “Hierarchical Event Histories”, *Thèse* en préparation à l'université de Carnegie Mellon, 1995.

[Kurlander 90] D. Kurlander & S. Feiner. “A Visual Language for Browsing, Undoing and Redoing Graphical Interface Commands”, *Visual Languages and Visual Programming*, Chang S. ed., Plenum Press, New York, NY, 1990, pages 257-275.

[Lieberman 93] H. Lieberman. “Mondrian: A Teachable Graphical Editor”, *Watch What I Do: Programming by Demonstration*, Cypher A., ed., MIT Press, Cambridge, MA, 1993, pages 340-358.

[Lipton 90] R.J. Lipton & R. Sedgewick. “NOTECH: Typesetting Without Formatting”, *Research Report*, Princeton University, 1990.

[Maulsby 89a] D. Maulsby & I. Witten. “Teaching a Mouse how to Draw”, *Proceedings of Graphics Interfaces'89*, pages 130-137.

[Maulsby 89b] D. Maulsby, K. Kittlitz & I. Witten. “Metamouse: Specifying Graphical Procedures by Example”, *Actes de SIGGRAPH 89*, Vol. 23, No. 3, ACM, Boston, Août 1989, pages 127-136.

[Maulsby 92] D. Maulsby, “Prototyping an Instructible Interface: Moctec”, *Actes de CHI '92*, ACM, Monterey, Mai 1992, pages 153-154.

[Michalski 83] R. Michalski. “A Theory and Methodology of Inductive Learning”, *Machine Learning, An Artificial Intelligence Approach*, Tioga, Palo Alto, 1983.

[Mima 91] Y. Mima. “A Visual Programming Environment for Programming by Example Abstraction”, *Proceedings of 1990 IEEE Workshop on Visual Languages*, Octobre 1991, pages 132-137.

[Mo 90] D. H. Mo. “Learning Text Editing Procedures From Examples”, *Research Report 90/387/11*, Department of Computer Science, University of Calgary, Calgary, Canada, May 1990.

[Modugno 94] F. Modugno & B. Myers. “Pursuit: Visual Programming in a Visual Domain”, *Technical Report CMU-CS-94-109*, Carnegie Mellon University, Pittsburgh, Janvier 1994.

[Myers 89] B. Myers, B. Vander Zanden & R. Dannenberg. “Creating Graphical Interfaces Application Objects by Demonstration”, *Actes de ACM SIGGRAPH UIST'89*, pages 95-104.

[Myers 90a] B. Myers et al. “Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment”, *IEEE Computer 23.11*, Nov. 1990, pages 71-85.

[Myers 90b] B. Myers. “Taxonomies of Visual Programming and Program Visualization”, *Journal of Visaul Languages and Computing, 1.1*, Mars 1990, pages 97-123.

[Myers 91a] B. Myers. “Text Formatting by Demonstration”, *Actes de CHI'91*, ACM 1991, pages 251-256.

[Myers 91b] B. Myers. “Graphical Techniques in a Spreasheet Specifying User Interfaces”, *Actes de CHI'91*, ACM 1991, pages 243-249.

[Myers 91c] B. Myers. “The Lapidary Graphical Interface Design Tool”, *Actes de CHI'91*, ACM 1991, pages 465-466.

[Myers 92] B. Myers. “Demonstrational Interfaces: A Step Beyond Direct Manupulation”, *IEEE Computer*, Vol. 25, No. 8, Août 1992, pages 61-73. Republié dans [Cypher 93].

[Nardi 93] B. Nardi. *A Small Matter of Programming: Perspectives on End User Programming*, MIT Press, Cambridge, MA, 1993.

[Negrompote 91] N. Negrompote. “Beyond the Desktop Metaphore”, *Reserach Directions in Computer Science: An MIT Perspective*, A. Meyer, J. Guttag, R. L. Rivest et P. Szolovits, Eds. MIT Press, Cambridge, MA, 1991, pages 183-190.

[Nevill-Manning 94] C. Nevill-Manning & D. Maulsby. “Modelling Sequences using Grammars and Automata”, *Rapport interne*, Université de Waikato, Nouvelle Zelande, Juillet 1994.

[Nix 85] R. Nix. “Editing by Example”, *ACM Transactions on Programming Languages 7.4*, Octobre 1985, pages 600-621.

[Olsen 88] D. Olsen & D. Dance. “Macros by Example in a Graphical UIMS”, *IEEE Computer Graphics & Applications*, Janvier 1988.

[Piernot 91] P. Piernot & M. Yvon. “Interfaces Graphiques Intelligentes : le Cas des Interfaces Démonstrationnelles”, *Rapport interne LIAP-5*, Université Paris 5, 1991.

[Piernot 93] P. Piernot & M. Yvon. “The AIDE Project: An Application-Independent Demonstrational Environment”, *Watch What I Do: Programming by Demonstration*, Cypher A., ed., MIT Press, Cambridge, MA, 1993, pages 382-401.

[Piernot 95a] P. Piernot et al. “Designing the PenPal: Blending Hardware and Software in a User-Interface for Children”, *actes de CHI '95: Human Factors in Computing Systems*, Denver, Colorado, 7-11 Mai 1995.

[Piernot 95b] P. Piernot & M. Yvon. “Incremental Command Tree Construction For Hierarchical Histories”, à paraître dans les *actes de HCI '95*, Université d'Huddersfield, Août 1995.

[Potter 93] R. Potter. “Triggers: Guiding Automation with Pixels to Achieve Data Access”, *Watch What I Do: Programming by Demonstration*, Cypher A., ed., MIT Press, Cambridge, MA, 1993, pages 360-380.

[Repenning 93] A. Repenning. “AgentSheets: A Tool for Building Domain-Oriented Dynamic Visual Environments”, Thèse CU-CS-693-93, University of Colorado at Boulder, Department of Computer Science, Décembre 1993.

[Rhyne 92] R. Rhyne & C. Wolf. “Tools for Supporting the Collaborative Process”, *Actes de ACM Symposium on User Interface Software and Technology, UIST '92*, ACM Press, Novembre 1992, pages 161-170.

[Shneiderman 83] B. Shneiderman. “Direct Manipulation: A Step Beyond Programming Languages”, *IEEE Computer*, Vol. 16, No. 8, Aout 1983, pages 57-69.

[Smith 75] D. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*, Birkhauser, Basel, 1977. Also in [Cypher 93].

[Smith 82] D. Smith, C. Irby, R. Kimball, B. Verplank & E. Harslem. “Designing the Star User Interface”, *Byte*, Vol. 7, No. 4, April 1982, pages 242-282.

[Soloway 82] E. Soloway, K. Ehrlich, J. Bonar & J. Greenspan. “What do novices know about programming?”, dans Badre & Shneiderman (eds), Directions in human-computer interaction, Norwood, NJ, Ablex Publishing Corporation, 1992.

[Stasko 91] J. Stasko. “Using Direct Manipulation to Build Algorithm Animations by Demonstration”, *Actes de CHI'91*, ACM 1991, pages 307-314.

[Thimbleby 91] H. Thimbleby. *User Interface Design*, ACM Press Frontier Series, 1991, chapitre 12, pages 261-286.

[Virzi 90] R. Virzi. “Streamlining the Design Process: Running fewer subjects”, *Proceedings of the Human Factors Society 34th Annual Meeting*, 1990, pages 291-294.

[Virzi 92] R. Virzi. “Refining the test pahse of usability evaluation: How many subjects is enough?”, *Human Factors 34*, 1992, pages 457-468.

[Vitte 86] J. S. Vitte. “Undo, Skip and Redo: a new framework for redoing”, *IEEE*, October 1986, 39–52.

[Wilson 90] D. Wilson, L. Rosenstein & D. Shafer. *Programming With MacApp*, Addison-Wesley, Reading, MA, 1990.

[Witten 81] I. Witten. “Programming by Example for the Casual User: A Case Study”, *Actes de Seventh Conference of the Canadian Man-Computer Communication Society*, CMCCS, Waterloo, Juin 1981, pages 105-113.

[Young 92] D. Young. *Object-Oriented Programming with C++ and OSF/motif*, Prentice Hall, Englewoods Cliffs, NJ, 1992.

[Yvon 94] M. Yvon, F. Lefèvre & P. Piernot. “Adaptive and Demonstrational Interfaces Applied to Telecommunication Services”, *Actes du East-West International Conference on Human-Computer Interaction*, Saint Petersbourg, 2 - 6 août 1994, pages 49-61, Volume II.

Glossaire

Application — Logiciel pour ordinateur personnel disponible dans le commerce comme Excel, DBase etc.

Arbre de commandes — **Commande** représentée comme un objet composé d'autres (sous-)commandes formant ainsi un arbre. La commande racine représente l'**historique** du système tandis que les commandes feuilles sont des **commandes de bas niveau**.

Commande — Option offerte par une **application** à l'**utilisateur**. Des exemples de commandes sont copier fichier, effacer le dernier caractère, déplacer un objet etc.

Commande de bas niveau — Action élémentaire effectuée par l'**utilisateur** comme déplacer ou cliquer sur un des boutons de la souris, presser ou relâcher une touche du clavier etc. Parfois également appelée **commande primitive**.

Commande primitive — Voir **commande de bas niveau**.

Evénement — Action élémentaire entraînant une modification du système. Inclue l'**invocation** d'une **commande de bas niveau** mais aussi le changement de date, l'insertion d'une disquette etc.

Historique — Ensemble des **commandes** que l'utilisateur a effectué. L'historique peut être lié à une application particulière ou être étendu à tout le système.

Invocation — Acte déclenchant l'exécution d'une **commande** ou d'un **programme**.

Macro — Une séquence de **commandes** qui peut être rejouée. Dans le contexte de la **Programmation Par Démonstration**, une procédure qui a été créée en enregistrant les **commandes** de l'utilisateur.

Programmation Utilisateur	Mécanisme destiné à faciliter la programmation pour les non programmeurs (**utilisateurs**). Les langages d'application, les enregistreurs de macro, les langages de programmation visuel, la **Programmation Par Démonstration** et la **Programmation Par l'Exemple** sont des réponses à la Programmation Utilisateur.
Programme	Voir **macro**.
Programmeur	Personne dont le métier est de programmer. A contraster avec l'**utilisateur** qui lui ne sait pas programmer.
Programmation Par Démonstration (PPD)	Création d' un **programme**, aussi appelé **macro**, en (dé) montrant au système, sur un ou plusieurs exemples, les étapes successives que doit effectuer le programme. Un système de PPD enregistre les commandes effectuées par l'utilisateur.
Programmation Par l'Exemple (PPE)	Création .d'un **programme** en donnant un ou plusieurs exemples du résultat de l'exécution du programme. Un système de PPE enregistre l'exemple de départ (état initial) et l'exemple modifié (état final).
Sélection	Liste des objets de l'**application** qui sont sélectionnés. Par exemple, dans un traitement de texte le contenu de la sélection est une liste de caractères. Dans un programme de dessin la sélection est souvent marquée par des "poignée" ou des pointillés entourant les objets sélectionnés.
Trace	Voir **historique**.
Utilisateur	Personne utilisant une **application**. Typiquement une personne qui n'est pas un **programmeur**.

Annexe 1

“The AIDE Project: An Application-Independent Demonstrational Environment”

P. Piernot et M. Yvon., *Watch What I Do : Programming by Demonstration,* Cypher A., ed., MIT Press, Cambridge, MA, 1993, pages 382-401.

Annexe 2

“Adaptive and Demonstrational Interfaces Applied to Telecommunication Services”

M. Yvon, F. Lefèvre et P. Piernot., *Actes du East-West International Conference on Human-Computer Interaction (EWHCI '94)*, Saint Petersbourg, 2 - 6 août 1994, pages 49-61, Volume II.

Annexe 3

“Designing the PenPal: Blending Hardware and Software in a User-Interface for Children”

P. Piernot et al., *actes de CHI '95: Human Factors in Computing Systems,* Denver, Colorado, 7 - 11 Mai 1995, pages 511-518.

Annexe 4

“Incremental Command Tree Construction For Hierarchical Histories”

P. Piernot et M. Yvon., à paraître dans les *actes de Human-Computer Interaction (HCI '95)*, University of Huddersfield, Angletere, 29 août - 1er septembre 1995.