MASTER EN SCIENCES INFORMATIQUES

Concevoir et réaliser l'interface usager des applications interactives

Waterkeyn, Pascal

Award date:
1985

Awarding institution:
Universite de Namur

Facultés Universitaires N.D. de la Paix - NAMUR

INSTITUT D'INFORMATIQUE

CONCEVOIR ET REALISER L'INTERFACE USAGER DES APPLICATIONS INTERACTIVES

PASCAL WATERKEYN

Mémoire présenté
en vue de l'obtention du grade de
Licencié et Maître en Informatique.

Année académique 1984 - 1985

Je tiens à remercier Monsieur A. Van Lamsweerde pour l'attention dont il a fait preuve lors de la lecture du manuscrit de ce présent travail.

TABLE DES MATIÈRES

1. INTRODUCTION

Ces derniers années ont vu apparaître sur la scène informatique un concept nouveau : celui de **poste de travail individuel.**

Il en résulte une toute nouvelle conception de la relation homme-machine, basée entre-autre sur une plus grande richesse du dialogue entre l'utilisateur et son système.
Sans nule doute, l'**interface usager**, ce composant matériel et logiciel chargé de réaliser et de gérer le dialogue, suscite beaucoup d'intérêt actuellement.
En effet :

- l'informatique ne s'adresse plus aux seuls spécialistes,

- l'utilisation croissante du traitement interactif, de même que les progrès des performances des terminaux, engendrent naturellement de nouvelles ixigences,

- le système "facile-à-apprendre", "facile-à-utiliser" favorise la productivité de l'usager, dont le coût, relativement au matériel, ne cesse de croître,

- l'explosion de l'interactivité est source d'un accroissement considérable de la complexité du code dédié à la communication homme-machine, qui représente, dans le cas notamment des applications de gestion courantes, une part importante du système global.

Sur le plan **matériel** [Yalamanchili 84, Pilxe 85, Gutz 81] le poste de travail actuel offre :

- un **écran graphique** de type bitmap à haute résolution (1024 x 1024 pixels) caractérisé par une importante surface d'affichage, un haut débit de transfert d'informations et une grande liberté quant à la présentation de cette information.

- un **dispositif de désignation** (souris, crayon lumineux, tablette graphique) permettant à l'usager de sélectionner directement une information affichée plutôt qu'en frappant son nom au clavier.

- une puissance de **traitement local** relativement importante (1 Mip, 2 Mb) permettant une utilisation en temps réel du poste de travail.

- des possibilités de **communication** avec les autres postes de travail (réseau local) permettant un partage des ressources matérielles et "informationnelles" (base de données).

Sur le plan du **dialogue interactif** [Smith 82, Tesler 81] la tendance actuelle est d'abandonner les langages de commandes classiques qui ressemblent trop aux langages de programmation. L'écran graphique permet de présenter à l'usager un environnement dynamique constitué d'un ensemble d'objets actifs qu'il peut organiser à sa guise.

Le domaine de la conception et de la réalisation de l'interface usager souffre cependant d'un manque de support méthodologique consistant [Good 81]. Faute d'outils logiciels et conceptuels adaptés, le concepteur se trouve démuni ; il est contraint de raisonner à un niveau d'abstraction trop bas et se forge une connaissance intuitive et informelle du problème. Il n'est donc pas surprenant qu'il tende à "réinventer la roue" et à construire du "sur-mesure".

Les travaux actuels dans ce domaine sont essentiellement basés sur l'indépendance syntaxique de l'application vis-à-vis du langage de commandes [Olsen 82, Pilote 83, Wasserman 79, Hayes 84, Rowe 83, Moran 78] ; mais ils s'intéressent à l'aspect dialogue uniquement et imposent un style d'interaction figé (soit orienté menu, soit dirigé par le formulaire).

(i) La première partie de ce travail introduit les notions qui interviennent lors de la **conception** de l'interface usager.
Le **modèle conceptuel de l'usager** est une représentation simplifiée de l'environnement d'exécution de sa tâche qui lui permet d'expliquer mentalement (et de deviner) les effets de ses actions. Nous proposons que la conception d'une application interactive soit basée, dès le début, sur ce modèle conceptuel (trop souvent, et actuellement encore, l'interface utilisateur est la dernière partie d'un système à être conçue). Nous présentons en 3.5. une méthodologie originale de conception basée sur une telle démarche.

Dans ce cadre, l'utilisation de **métaphores** confère une grande cohérence au modèle en fournissant une référence constante à quelque chose d'existant.

Nous définirons l'**état courant** d'une application comme l'ensemble des commandes et des objets accessibles à l'usager à un moment donné. L'affichage permanent d'une représentation visuelle de l'état courant fournit toute l'information dont l'usager a besoin à court terme et permet donc de diminuer sa charge mentale.
Une telle approche conduit à un style d'interaction du type "voir et pointer plutôt que se souvenir et taper".

(ii) Le but de la seconde partie de ce travail est de fournir au concepteur d'application interactive un certain nombre d'abstractions qui lui seront utiles pour :

- Réaliser une **séparation** claire entre l'application et l'interface usager grâce à des concepts charnières qui permettront d'exprimer leur interaction en termes suffisamment abstraits.

- Favoriser une bonne **structuration** de l'interface usager lui-même en dégageant les différentes fonctions qu'il doit assurer de manière à les rendre indépendantes.

Au plus bas niveau d'abstraction, l'usager interagit avec l'application par la lecture de caractères affichés à l'écran, par la frappe de touches au clavier et par le déplacement du dispositif de désignation.
A ce niveau, nous désirons assurer l'indépendance de l'application vis-à-vis des particularités du matériel utilisé.
Nous définirons le **terminal logique** comme le modèle d'un appareil idéal, cachant à l'application les spécificités du terminal physique.

A un niveau d'abstraction plus élevé, le dialoque s'exprime en termes de commandes et d'objets. Les commandes permettant d'activer les traitements; les objets, représentés visuellement, sont modifiés par les traitements et en font apparaître les effets.

Nous définirons l'**usager logique** comme un représentant simplifié de l'usager réel. Le protocole de **dialogue logique** régit l'interaction entre l'usager logique et l'application. Il permet d'assurer l'indépendance de l'application vis-à-vis du protocole de dialogue réel.

En ce qui concerne la gestion dynamique de la représentation visuelle des objets, nous introduirons une structure d'échange appelée **représentation externe.** Elle permet l'expression du transfert d'informations affichables en termes de concepts abstraits. Sa traduction en primitives de bas niveau pour le terminal logique est invisible de l'application.

Enfin, pour assurer l'indépendance de l'usager vis-à-vis d'un contexte d'exécution particulier, c'est-à-dire lui permettre d'utiliser concouramment plusieurs applications, nous introduirons le concept d'**appareil virtuel.** Du point de vue de l'application, celui-ci possède les mêmes fonctionnalités que le terminal logique. Du point de vue de l'usager cependant, le terminal virtuel ne monopolise pas l'écran du poste de travail ; il est représenté par une fenêtre.

Tout au long de ce texte, nous tenterons de mettre en évidence les bénéfices d'une approche "orientée-objet", tant au niveau de l'ergonomie du dialogue qu'au niveau de sa mise en oeuvre.

2.1. MODELISATION DE L'APPLICATION

2.1.1. Introduction

Nous définirons ici les concepts de base que nous utiliserons tout au long de ce texte. Nous ne tenterons pas de fournir des définitions un tant soit peu formelles ni totalement dépourvues d'ambiguités. Notre but est plutôt d'introduire intuitivement l'approche particulière que nous avons adoptée.

Commençons par deux définitions classiques et très générales:

- **Une application interactive** est une collection de programmes exploités au moyen d'un dialogue entre l'ordinateur et un utilisateur humain.

- **L'interface usager** d'une application interactive est le composant chargé de gérer ce dialogue.

Précisons ces notions :

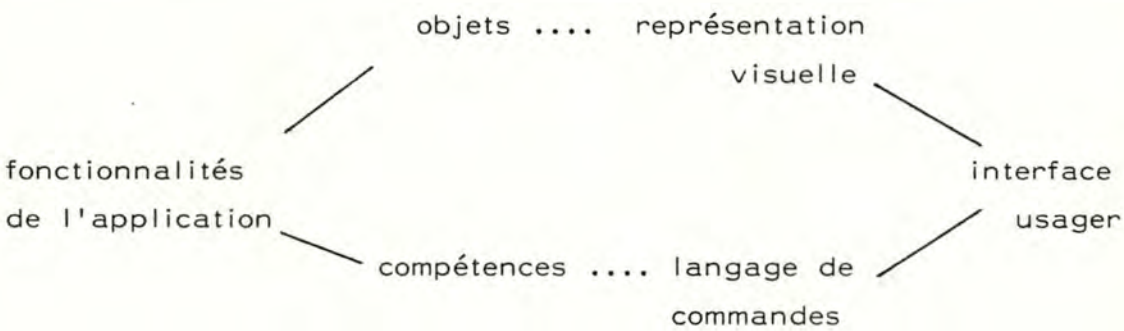
L'application est conçue pour aider l'utilisateur à réaliser une **tâche.** La tâche est une entité complexe structurée en sous-tâches. Une **activité** est une (sous-) tâche qui peut être accomplie à l'aide d'une seule application. Par exemple, l'édition et la compilation sont des activités de la tâche consistant à réaliser un programme.

Nous modéliserons l'application selon une approche "orientée-objet"[Robson 81, Goldberg 83]. L'application est l'implémentation d'un modèle d'activité, représentation simplifiée du monde réel. Ce modèle s'exprime en termes **d'objets** et d'opérations définies sur ces objets. Les caractéristiques des objets du modèle ainsi que des opérations que l'application peut effectuer (appelées compétences dans la suite) constituent les **fonctionnalités** de l'application.

Dans le cas de l'éditeur, par exemple, les objets manipulés sont des textes (en langue naturelle ou dans un langage formel), des lignes, des mots (ou symboles) et des caractères. Les compétences de l'éditeur consistent en différentes opérations d'édition sur ces objets : insertion, suppression, déplacement, recherche, etc...

Concevoir l'interface usager d'une application interactive consiste traditionnellement à définir une représentation visuelle des objets ainsi qu'un langage de commandes permettant d'activer les compétences de l'application.

Donc, toute application interactive est définie par ses fonctionnalités et son interface usager.



L'usager est confronté au système interactif pour accomplir une certaine tâche. Il jugera donc les qualités de l'application en fonction de l'aide effective qu'elle lui apporte dans l'accomplissement de sa tâche.

La conception de l'interface usager d'une application interactive commencera donc par une analyse de la tâche telle qu'elle est effectuée actuellement, c'est-à-dire manuellement ou par un système automatique que l'on désire remplacer.

La description de la tâche actuelle offre alors un point de départ pour la conception d'un nouvel environnement de tâche

pour l'usager, dans lequel il peut accomplir les mêmes buts qu'auparavant, entouré à présent d'un nouvel ensemble d'objets et utilisant de nouvelles méthodes.

Le modèle conceptuel d'une application décrit la manière dont ses fonctionnalités sont présentées à l'usager.

Le modèle conceptuel de l'usager, quant à lui, décrit la manière dont l'usager conçoit l'environnement d'exécution de sa tâche. L'utilité effective de l'application dépend alors de la possibilité d'assimilation de ces deux modèles.
En effet, l'incapacité pour l'usager d'appréhender correctement le modèle conceptuel que lui présente l'application conduira inévitablement à des réactions de rejet et à une utilisation erronnée de l'application, quelles que soient les qualités intrinsèques de ses fonctionnalités.

La tâche de l'interface usager est de représenter le modèle conceptuel de l'application. Il définit une méthode d'accès à ce modèle appelée **modèle d'interaction.**

2.1.2. Choix du modèle conceptuel

Bien qu'on ne sache définir ce qu'est un bon modèle, on s'accorde cependant à dire qu'une application devrait être facilement compréhensible et utilisable si le modèle conceptuel de l'usager et celui du concepteur sont proches.

Le concepteur d'application interactive peut donc choisir d'utiliser des analogies familières (métaphores), ou bien il introduira des fonctions entièrement nouvelles nécessitant une approche originale. Chaque choix a ses avantages et ses inconvénients.

La méthaphore est d'une grande utilité pour l'usager débutant. En effet, l'apprentissage ne se limite pas à un effort de mémorisation, il est également nécessaire que l'usager puisse comprendre. Il construira un modèle mental de l'application au fur et à mesure de son utilisation. Ce modèle lui permettra d'expliquer le comportement de l'application. Or, le mécanisme de la compréhension semble être dominé par le raisonnement par analogie. Si le modèle de l'application est basé sur une métaphore, il fournira une référence constante à quelque chose d'existant. L'utilisateur possèdera alors une pré-connaissance de l'application.

L'utilisation d'une métaphore confère également une grande cohérence au modèle. La référence à quelque chose de connu ainsi que la cohérence du modèle poussent l'usager à expérimenter et à investiguer.

La métaphore fournit donc une base simple à partir de laquelle l'apprentissage peut évoluer de façon incrémentale.

L'usage d'une métaphore n'est cependant pas sans danger. L'application définit pour l'usager une vision nouvelle de l'activité. Les moyens matériels ne sont pas identiques aux

précédents, les objets ne se matérialisent plus de la même manière, ne possèdent plus tout à fait les mêmes propriétés, et leur manipulation ne suit plus le mode opératoire traditionnel. Pour une discussion de ce genre de problème on consultera [Smith 82] .

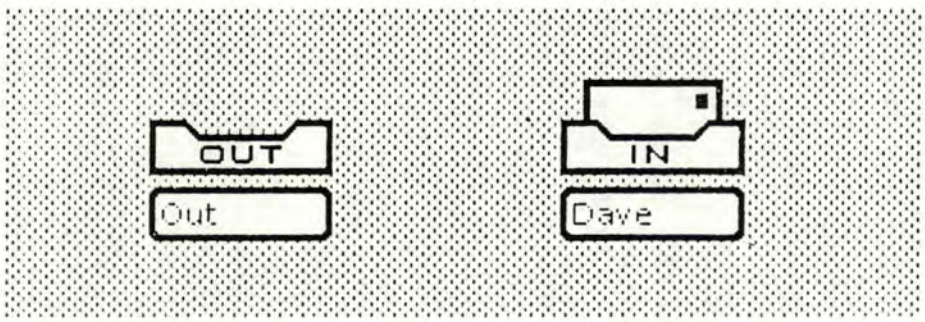
2.1.3. Exemple d'utilisation d'une métaphore

La conception de l'interface usager du Star de XEROX (dont ont hérité Lisa et Macintosh d'APPLE) est basée sur **la métaphore du dessus de bureau** ("desktop metaphor").

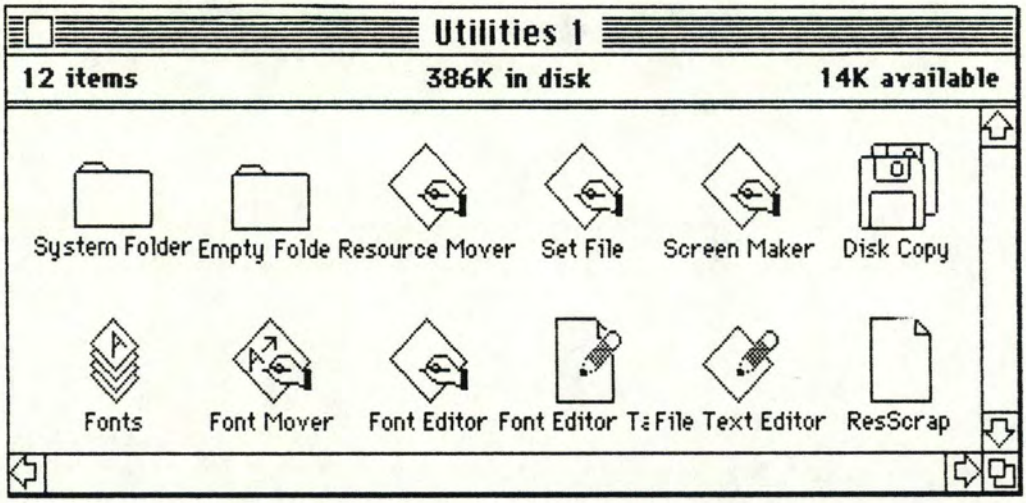
Le modèle sous-jacent à une telle interface contient l'équippement traditionnel de bureau : classeur, fichier, bloc-notes, corbeille à papier, calendrier, calculatrice, etc...

Ces objets sont représentés par des symboles graphiques apparaissant sur l'écran du poste de travail. Ces symboles graphiques, appelés **icônes**, ne sont pas des dessins passifs mais sont utilisés pour contrôler les simulations des activités effectuées sur un bureau.
Les icônes possèdent donc la puissance de traitement nécessaire pour simuler les objets réels qu'elles représentent.
L'usager manipule ces différents accessoires au moyen du dispositif de désignation dont est équipé son poste de travail.

La métaphore du dessus de bureau n'est qu'un cas particulier de la **métaphore physique** conçue par Alan KAY [Kay 77]. Celle-ci consiste à utilier une représentation visuelle (icône) dotée des mêmes capacités que l'objet réel qu'elle représente. La métaphore physique est donc d'une grande généralité.



Icônes de l'interface usager du Star représentant les paniers de départ et d'arrivée du système de de courier électronique.



Icônes de l'interface usager du Macintosh représentant diverses applications utilitaires

Dans cette section, nous avons introduit la notion de métaphore pour présenter à l'usager le modèle conceptuel de l'application de manière à favoriser son **assimilation.**

Dans la section suivante, nous déterminerons quelles sont les informations à présenter à l'usager pour favoriser **l'utilisation** du modèle.

2.2. Modélisation de l'état d'une application interactive

Dans ce chapitre, nous introduisons la notion **d'état** d'une application. Par la visualisation constante de l'état de l'application on diminue la charge mentale et le facteur d'anxiété de l'usager. En lui permettant de revenir dans des états antérieurs on lui fournit un mécanisme général pour annuler les effets des erreurs.

Pour l'usager, **l'état courant** de l'application est défini par:

- **l'environnement des commandes** : quelles sont les commandes que l'usager peut effectuer à ce moment.
- **L'environnement des objets** : quels sont les objets susceptibles d'être affectés par l'exécution de ces commandes.

Le concept d'environnement de commandes, encore appelé **mode**, a deux fonctions :

- il détermine à une activité bien précise, sélectionnée dans l'ensemble des activités que l'application permet d'effectuer (par exemple : le mode " insertion " correspond à l'une des activités que permet l'éditeur de texte ; celui-ci possède également un mode " recherche ", un mode " déplacement ", un mode " suppression " ...).
  La notion de mode permet donc de décomposer l'espace des commandes de l'application en sous-ensembles de taille raisonnable (typiquement moins de 10 commandes).

- il permet de pallier la limitation du nombre de touches du clavier du poste de travail physique (le mode courant permet, par exemple, de distinguer les caractères formant le nom d'une commande des caractères à insérer dans le texte et qui n'ont pas de signification particulière pour l'éditeur).

Tout comme l'état courant de l'application détermine les commandes courantes, il détermine également le ou les objets courants.
La notion d'objet courant est liée à celle de centre d'intérêt. L'objet courant est celui que l'on traite actuellement. Il permet de ne pas avoir à spécifier à chaque commande l'objet sur lequel elle s'applique. L'objet courant correspond donc à une sorte de valeur par défaut.

Le comportement dynamique de l'application est décrit par les changements d'état. Ceux-ci sont provoqués par les commandes ; soit directement par certaines commandes qui ont pour unique fonction de modifier l'état de l'application, soit indirectement par des commandes effectuant un traitement et provoquant un changement d'état par effet de bord.

Le comportement de l'application peut ainsi être modélisé par un diagramme de transition d'états. L'application est alors un interpréteur pour un tel diagramme. Celui-ci possède un état interne et un certain comportement d'entrée-sortie.
En "entrée", il accepte les commandes de l'usager. En "sortie", il modifie des objets lors de changements d'état.

Cette approche ne doit pas être confondue avec celle qui consiste à décrire la syntaxe du langage de commandes au moyen de diagrammes de transition. L'application se doit, en effet, d'être indépendante du format des commandes ; la manière dont les caractères frappés au clavier sont transformés en commandes bien formées n'est pas de son ressort. De plus, la tendance actuelle consiste à abandonner les langages de commandes classiques du style langage de programmation au profit d'une utilisation du contenu de l'écran au moyen du dispositif de désignation (menu, sélection d'objet).

Comme l'état courant de l'application interactive détermine l'interprétation du dialogue, il est impératif que l'usager puisse à tout moment déterminer aisément l'état de l'application, et ceci sans devoir modifier cet état.

Une analyse des problèmes rencontrés par les usagers d'applications interactives [Nievergelt 83] fait apparaître certaines situations fréquentes qui s'expriment par les questions suivantes :

- " Où suis-je ? "
  Lorsque l'écran est différent de ce qui était attendu.

- " Que puis-je faire ici ? "
  Lorsque l'usager est incertain à propos des commandes activables.

- " Comment suis-je parvenu ici ? "
  Lorsqu'il pense avoir frappé une mauvaise touche.

- " Où puis-je aller ? et comment aller là ? "
  Lorsqu'il désire explorer les possibilités du système.

Remarquons que les " lieux " mentionnés dans ces questions sont en fait des états de l'application.

Les deux premières questions sont relatives à l'état courant. Aux questions " Où suis-je ? " et " Que puis-je faire ici ? ", l'application répondra en affichant respectivement l'environnement courant des données et l'environnement courant des commandes.

La question " Comment suis-je parvenu ici ? " concerne le passé du dialogue, son historique. Nous définirons l'historique du dialogue comme une suite de couples (état, commande) représentant les différents états par lesquels l'application est passée et les commandes qui ont provoqué les changements d'état. En lui donnant accès à l'historique du dialogue, l'application peut justifier à l'usager comment l'état courant a été atteint. En lui permettant de revenir en arrière dans le dialogue, l'application fournit un mécanisme de base pour annuler les effets de ses erreurs.

Remarquons que dans ce dernier cas, on inclut dans l'état non seulement l'identité des objets courants, mais également leur valeur. Ce mécanisme peut être très lourd s'il permet de revenir dans n'importe quel état antérieur car il faut alors mémoriser toutes les valeurs intermédiaires de tous les objets modifiés, y compris des fichiers arbitrairement grands [Archer 84] . La majorité des systèmes ne mémorisent qu'un seul état précédant l'état courant.

En ce qui concerne les questions "Où puis-je aller ?" et "Comment aller là ?", l'application interactive fournira une commande d'aide ("help") invoquable à tout moment. Comme de telles questions se produiront toujours et que le bon manuel est rarement à la bonne place au bon moment, un système interactif doit être auto-explicatif.

En affichant le sous-diagramme de transition constitué des arcs issus du noeud associé à l'atat courant, la commande "help" visualise l'ensemble des états accessibles à partir de l'état courant. Bien entendu, une aide quant à l'**effet** des différentes commandes est également nécessaire.

En tenant compte de l'état courant, et éventuellement de l'historique du dialogue, cette explication peut être adaptée au contexte

2.2.2. REPRESENTATION DE L'ETAT D'UNE APPLICATION

Dans le paragraphe précédent, nous avons montré la nécessité pour l'usager de pouvoir consulter l'état courant de l'application. On peut aller plus loin, en imposant que cet état soit visible en permanence.
L'écran d'affichage comportera alors, à tout moment, une représentation partielle de l'état courant de l'application ; sous forme d'une liste des commandes activables (menu), une au plusieurs listes des objets accessibles et une représentation visuelle de la valeur de certains objets : ceux qui font partie du centre d'intérêt courant.

Cette approche sous-entend donc que l'**état courant de l'application constitue toute l'information dont l'usager à besoin à court terme.** Cet état ne pouvant pas être représenté entièrement sur l'écran, l'application sélectionnera certains objets en fonction du contexte, en anticipant les besoins de l'usager. La représentation de l'état de l'application doit également être paramétrable par l'usager lui-même. Il pourra alors adapter le mode d'emploi de l'application à ses habitudes particulières et à son degré d'expertise.

Quels avantages peut on retirer d'une telle approche ?

1. Par une meilleure sélection des informations affichées à l'écran, on peut espérer diminuer la charge mentale de l'usager.

   Durant la pensée consciente, le cerveau utilise plusieurs niveaux de mémoire, le plus important étant la mémoire à court terme. De nombreuses études ont analysé la mémoire à court terme et son rôle lors de la réflexion. Deux conclusions émergent :
   1° la pensée consciente utilise les concepts qui sont dans le mémoire à court terme et 2° la capacité de la mémoire à court terme est limitée.

Lorsque le contexte d'exécution de la tâche est rendu visible en permanance, l'écran d'affichage prend le relais de la mémoire à court terme en agissant comme une "mémoire cache visuelle".

2. Par l'utilisation du dispositif de désignation et la visualisation instantanée des effets des commandes, l'usager a l'impression d'agir directement sur son environnement.

Le dispositif de désignation permet à l'usager de sélectionner des objets et des commandes affichés à l'écran. La possibilité de "montrer" fournit toute la puissance de la référence pronominale.

Plutôt que de désigner un objet ou une commande indirectement, par son nom ou son numéro dans un menu, l'usager peut montrer un objet en disant **cette** procédure, **ce** fichier, **cet** article...

En dialoguant avec l'application, l'activité de l'usager consiste à sélectionner les traitements appropriés sur les objets courant de manière à obtenir les résultats voulus. Dans ce cadre, l'état courant de l'application est, pour l'usager, l'état d'avancement de son activité ; il consitue son environnement de travail.

En représentant visuellement cet état et en donnant l'impression à l'usager qu'il agit directement sur les objets qu'il voit et qu'il désigne, on supprime virtuellement l'existence de l'application. C'est l'usager lui-même qui modifie les objets car, comme dans le monde réel, il voit instantanément l'effet de ses actions.

Pour l'application, comme pour l'usager, chaque objet est caractérisé par les opérations que l'on peut lui appliquer. Dans un dialogue orienté-objet, on sélectionne d'abord l'objet et ensuite l'opération.

La sélection de l'objet cible ainsi que celle de l'opération se feront avantageusement à l'aide du dispositif de désignation. Cette approche semble plus naturelle car elle correspond aux habitudes du monde réel où la référence pronominale est constamment utilisée.

Dans cette section, nous avons défini le concept **d'état** d'une application interactive comme l'ensemble des commandes et des objets accessibles à un momment donné. L'exécution des commandes provoquant les changements d'états.

L'état courant de l'application prend donc implicitement en compte les conditions d'applicabilité des commandes futures et les résultats des commandes antérieures, c'est à dire toute l'information dont l'usager a besoin à court terme.

L'affichage permanent d'une représentation visuelle de l'état courant et la possibilité de revenir dans des états antérieurs permettent de diminuer la charge mentale et le facteur d'anxiété de l'usager. Un modèle d'interaction basé sur une telle approche est du type " voir et pointer plutôt que se souvenir et taper ".

2.3. INTERFACE USAGER UNIVERSELLE

Nous avons, jusqu'à présent, implicitement associé l'interface usager à l'application interactive dont il fait partie. Les concepts développés dans les sections précédentes sont néanmoins indépendants de toute application particulière.

Dans cette section, nous introduisons la notion d'interface usager universelle comme une entité à part entière : celle qui est chargée de gérer la communication entre l'utilisateur humain et l'ensemble des applications de son système interactif.

Dans les sections suivantes, nous développerons alors des abstractions permettant de réaliser l'indépendance de l'interface usager.

Malgré l'intérêt actuel pour la communication homme-machine, les découvertes ont, jusqu'à présent, tendance à être appliquées seulement localement, **à l'intérieur d'un programme d'application individuel**, plutôt que globalement, pour un système entier sur lequel fonctionnent beaucoup d'applications.

Les systèmes interactifs actuels se présentent à l'utilisateur comme une collection de programmes d'application sans liens. L'utilisateur est confronté à toute une série d'interfaces distinctes, chacune basée sur ses propres concepts et principes d'interaction.

Alors que la population d'utilisateurs augmente rapidement et comprend une fraction croissante d'usagers novices ou occasionnels, nous devons développer des systèmes dans lesquels tous les programmes d'application **parlent le même langage.**

Pour pouvoir réaliser ce but, nous devons être capables de définir un canevas conceptuel caractérisant l'interaction, qui soit suffisamment général pour s'appliquer à (presque) toutes les tâches interactives actuelles et à venir.

Cela nous permettra de parler d'interface usager dans l'abstrait et, dans une seconde étape, de concevoir une architecture détaillée pour une telle interface.

Cette approche présente un certain nombre d'avantages apparents, tant pour l'usager que pour le réalisateur.

(i) Pour l'usager, l'effet principal de l'unicité de l'interface usager est d'augmenter la cohérence apparente du système. Alors que l'utilisation de métaphores confère une grande cohérence au modèle d'interaction au sein d'une même application, le partage de l'interface usager étend cette cohérence au système tout entier.

L'utilisateur acquiert une vue homogène de son environnement de travail car un mode d'emploi identique (définit par le modèle d'interaction) s'applique à toutes les applications.

En prolongeant cette idée, on peut dire que tout ce qui sera extrait des applications pour être incorporé à l'interface usager unique contribuera à la cohérence du système.

Dans la section consacrée à l'état de l'application interactive, nous avons mis en évidence un certain nombre de commandes devant être activables à tout moment : ce sont les demandes d'information sur l'état courant, les commandes de retour en arrière ("undo") et les demandes d'aide ("help"). De telles commandes, basées sur des concepts indépendants des spécificités fonctionnelles des applications seront appelées commandes universelles ou encore **commandes génériques.**

(ii) Nous analyserons à présent les avantages et les implications d'une telle approche pour la réalisateur de système interactif.

De par les progrès dans le domaine du matériel interactif, le dialogue entre l'homme et la machine devient de plus en plus sophistiqué, introduisant ainsi une complexité croissante du logiciel devant le gérer.

La conception et la réalisation de l'interface usager devient donc de plus en plus longue et coûteuse. Ceci est dû, non seulement à la conception initiale, mais également au long processus de raffinement nécessaire à l'obtention d'une interface réellement utilisable.

D'une part, l'unicité de l'interface usager permet de factoriser **le coût** de sa conception et réalisation par le nombre d'applications l'utilisant.

De plus, grâce au découplage logique et physique de l'interface par rapport aux application, toute modification de celui-ci restera transparente aux applications pour autant que le protocole de dialogue entre l'interface et les applications soit inchangé. Ceci permet donc l'évolution de l'interface usager en cours d'utilisation et favorise **la maintenance.**

Dans cette section nous avons étendu la séparation logique entre l'application et l'interface usager à une séparation physique. Pour l'utilisateur, l'interface usager universelle augmente la cohérence des diverses applications par l'unicité du modèle d'interaction et la définition de commandes génériques. Le concepteur, quant à lui, voit le coût de réalisation et de maintenance de son interface usager divisé par le nombre d'applications l'utilisant.

Les chapitres suivants développeront des abstractions permettant de réaliser une telle indépendance.

3.1. LE TERMINAL LOGIQUE

3.1.1. Introduction

Le terminal logique permet d'assurer l'indépendance de l'application vis-à-vis des caractéristiques du terminal physique.

Le terminal logique est un modèle. C'est un **type abstrait** qui possède les fonctionnalités d'un terminal **idéal.** En cachant la manière dont ces fonctionnalités sont mises en oeuvre, il abstrait les spécificités du terminal physique. L'application est alors assurée que le terminal se comporte selon le modèle du terminal idéal.

La réalisation du concept d'appareil logique passe par deux étapes : le choix d'un modèle et le choix d'une implémentation.

3.1.2. CHOIX DU MODELE DU TERMINAL LOGIQUE

Pour définir le modèle du terminal logique, nous avons pris en compte les qualités suivantes : simplicité, cohérence, pouvoir d'abstraction, correspondance avec le modèle intuitif et correspondance avec les possibilités du matériel actuel.

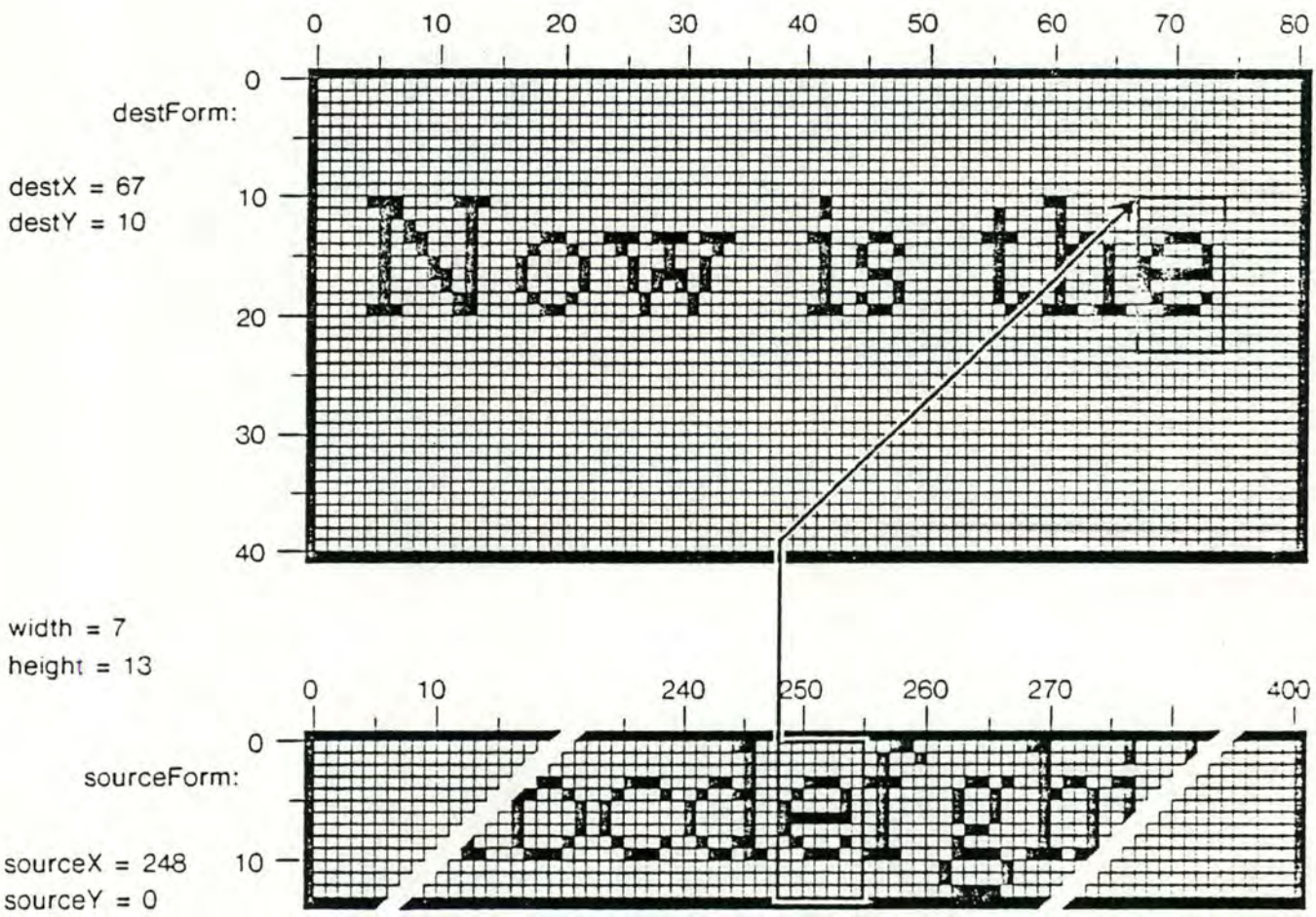
Ces caractéristiques ne sont certes pas indépendantes. Simplicité, cohérence et correspondance avec le modèle intuitif sont nécessaires pour rendre le modèle utilisable et utilisé. Le pouvoir d'abstraction, c'est-à-dire la faculté de cacher de l'information, est, bien entendu, la raison d'être du terminal logique. Les spécifications des fonctionnalités du terminal logique doivent s'exprimer en termes appropriés à son utilisation et cacher les particularités de l'implémentation. Le modèle "flux d'octets" est visiblement trop rudimentaire.

Nous avons tenté de définir un modèle qui permette l'utilisation effective des possibilités offertes par le matériel actuel. Le modèle le plus courant actuellement est le terminal à écran alphanumérique. Les écrans graphiques de type **bitmap**, associés à un dispositif de désignation tel que la **souris** ne sont pas encore fort répandus, mais leur avenir semble assuré, vu la baisse des prix dans ce secteur.
Un modèle "ligne à ligne" basé sur le télétype, tel que fournissent la plupart des systèmes d'exploitation est inapproprié ; il conduit inévitablement à une sous-utilisation des possibilités du matériel.

En nous basant sur le fait que l'information textuelle gardera encore longtemps sa place privilégiée, nous proposons un modèle **alphanumérique**, c'est-à-dire basé sur le concept de caractère.
Ceci n'exclut pas une extension future du modèle vers de réelles possibilités graphiques. Une telle extension peut se faire, par exemple, en raffinant le concept de caractère en un tableau de points (bitmap) ou une série de vecteurs (shape).

L'utilisation du modèle alphanumérique sur un écran graphique de type "bitmap" passe par l'utilisation d'une table ("form" en Smalltalk) mémorisant tout l'aphabet affichable sous forme d'un tableau de points blancs et noirs. L'affichage d'un caractère particulier se fait alors en recopiant le sous-tableau de points qui lui correspond dans l'écran d'affichage. (Un écran de type bitmap est, en effet, une partie de la mémoire affichée en permanence , chaque bit correspondant à un point sur l'écran).

En définissant différentes tables de conversion pour un même alphabet, il est possible de réaliser différentes polices de caractères (italiques, gras, ...).



L'unité de coordonnées de position est le **caractère** pour l'écran alphanumérique et le **point** (ou pixel) pour l'écran bitmap. Le rapport entre ces deux systèmes de coordonnés est défini par la taille du tableau de points associé à un caractère de l'alphabet courant. Si ces tableaux n'ont pas une taille identique pour chaque caractère de l'aphabet , alors le passage d'un système de coordonnées à l'autre devient fort complexe (car il dépend du contenu de l'écran).

Le modèle que nous proposons est donc orienté vers les terminaux alphanumériques courants, mais il est compatible avec les écrans graphiques. Une approche basée sur des concepts propres aux écrans graphiques aurait été inutilisable pour les terminaux alphanumériques.

Le terminal logique comprend trois composants : l'écran logique, le clavier logique et la souris logique. Le premier est un dispositif de **sortie**, les deux autres des dispositifs d'**entrée.**

Le clavier et la souris captent des événements extérieurs. Pour l'application, la frappe d'une touche ou le déplacement de la souris sont des **événements asynchrones.** Dans les environnements de programmation habituels, la synchronisation est obtenue au moyen d'un tampon qui mémorise la survenance de ces événements. Les opérations fournies au programmeur sont alors des demandes de lecture du dispositif (en fait, du contenu du tampon).
Dans de tels environnements, on a perdu la notion d'événement. Une telle approche ne convient pas :

- dans un environnement en **temps réel.** Un haut degré d'interactivité nécessite le contrôle du temps de réponse. Les environnements en temps partagé ne le permettent pas.

- dans un environnement à **plusieurs dispositifs d'entrée.** En effet, la lecture sur un tampon vide provoque le blocage du

processus lecteur, ce qui l'empèche de pouvoir traiter le contenu d'autres tampons, par exemple.

En conséquence, nous avons tenu à préserver la notion d'événement. Cette approche est originale.

Dans la méthaphore orientée-objet, le concept de message permet d'associer événement et appel de procédure. Il permet donc de supprimer l'attente explicite exprimée par l'opération de lecture.

La structure générale du programme interactif classique comporte une bouche de lecture explicite :

```
répéter
    lire(commande);
    exécuter(commande);
jusqu'à  commande = fin.
```

Grâce au concept de message, cette structure peut être remplacée par la suivante ; l'application interactive est alors composée d'un ensemble de **serveurs** ; un pour chaque type d'évènement externe.

```
traiter-toouche(t)
...
traiter-déplacement(d)
...
traiter-clic(c)
...
```

Ces **serveurs** sont chargés de reconnaître les commandes et d'appeler les procédures de traitement appropriées. En assignant des priorités plus élevées aux **serveurs**, il est possible d'assurer une réponse rapide à certains événements externes, quel que soit le traitement en cours.

Nous définirons donc le clavier logique et la souris logique comme des émetteurs de messages à destination de l'application et l'écran logique comme un récepteur de messages. La communication entre le terminal logique et l'application est donc réellement bidirectionnelle ; non seulement pour les données, mais également du point de vue du contrôle.

3.1.2.1. L'ECRAN LOGIQUE

L'écran logique permet l'affichage d'informations. L'unité d'information affichable est le **caractère.** Le contenu de l'écran, c'est-à-dire l'ensemble des caractères affichés, est structuré en **page.** Une page est une suite de **lignes.** Une ligne est une suite de caractères.
La position d'un caractère permet d'identifier un élément du contenu de l'écran. Il s'agit d'un couple **(x,y)** où **x** est le numéro d'ordre du caractère dans la ligne et **y** le numéro d'ordre de la ligne dans la page d'écran.

Chaque position ayant mêmes dimensions, la surface d'affichage peut donc être vue comme un tableau bidimensionnel, soit un rectangle. La hauteur du rectangle est égale au nombre de lignes dans la page et la largeur du rectangle est égale au nombre de caractères dans la ligne.

La compétence de base de l'écran logique effectue l'affichage d'un caractère à une position donnée. Il s'agit en fait d'une modification d'un élément du contenu de l'écran.

Sa spécification est :

" La compétence de WriteChar (x,y,c) a pour effet d'afficher le caractère **c** à la position (x,y) " .

Préconditions : La position (x,y) doit être valide, c'est-à-dire :

$o \leqslant x < $ largeur et $o \leqslant y <$ hauteur et c doit être un caractère valide, c'est-à-dire le code ASCII d'un caractère affichable.

L'écran logique permet la prise en compte d'**attributs visuels** associés à chaque caractère affiché. Il s'agit d'effets spéciaux tels que la surbrillance, l'inversion vidéo, le clignottement, le souligné, etc...

En général, la compétence d'affichage s'écrira donc :

WriteChar (x, y, c, a1, a2, ... an)

où a1, a2, ... an seront les valeurs d'attributs qui seront associées au caractère.

Cette approche possède deux désavantages :

1. un nombre élevé d'arguments à spécifier à chaque appel. Le programmeur doit connaître l'existence et la signification de chacun d'entre eux.
   L'ordre de leur spécification (qui est arbitraire) doit également être mémorisé. Une erreur à ce niveau peut être particulièrement difficile à repérer.

2. un manque de flexibilité quant à l'évolution du modèle. Toute modification du modèle (l'ajout de nouveaux attributs, par exemple) nécessite la mise à jour de tous les programmes utilisant le terminal logique (à cause du changement du nombre d'arguments, de leur ordre et de leur signification).

Pour diminuer le nombre d'arguments, nous avons introduit la plupart d'entre eux dans le modèle sous forme implicite, en leur associant des valeurs courantes par défaut. Ces valeurs peuvent être consultées et modifiées par des compétences appropriées. Il y a donc une position courante (valeur par défaut pour les arguments x et y) et des valeurs courantes pour les attributs visuels.

L'écran logique consiste donc en un type abstrait comportant deux catégories de compétences :

- les compétences dont l'effet correspond à une action (comme, par exemple, affichier un caractère).
  L'effet de cette action dépend de la valeur courante de certains arguments implicites au moment de son exécution.

- les compétences qui permettent de consulter et modifier les valeurs par défaut des arguments implicites (comme, par exemple, modifier la position courante d'affichage).

Remarquons qu'il n'est pas exclus que les compétences de la première catégorie modifient également certaines valeurs par défaut, par "effet de bord" (par exemple, l'affichage d'un caractère peut modifier la position courante).

En diminuant le nombre d'arguments explicites nous avons, en contrepartie, augmenté le nombre de compétences.

En ce qui concerne l'accès aux arguments implicites, trois solutions sont possibles :

- pour chaque argument, définir une compétence qui permet de lire sa valeur et une compétence pour modifier sa valeur.

- pour chaque argument et pour chaque valeur, écrire une compétence qui teste si cet argument possède cette valeur et une compétence qui affecte cette valeur à l'argument.

- écrire deux compétences, l'une acceptant le nom d'un argument et renvoyant sa valeur, l'autre acceptant le nom d'un argument et une valeur et affectant la valeur à l'argument.

Remarquons que ces trois approches ne s'excluent pas mutuellement. Elles peuvent être combinées pour un même type abstrait, en fonction de la sémantique des arguments concernés et éventuellement du nombre de valeurs possibles (cas de la 2e solution).

Nous avons adopté la troisième approche, essentiellement parcequ'elle minimise le nombre de compétences. Certains arguments implicites possèdent cependant leur compétence propre (c'est le cas de la position courante pour les arguments implicites x et y).

Attributs de l'écran logique

- Width : largeur de l'écran logique
- Height : hauteur de l'écran logique
- xCursor : position courante : x
- yCurson : position courante : y (ligne courante)
- Emphasize : valeur courante de l'attribut "mise en évidence".
- AutoWrap : Passage à la ligne automatique
- AutoScroll : Défilement du contenu de l'écran (scrolling).

Compétences de l'écran logique

- WriteChar(c) : Ecrit le caractère **c** à la position courante.

- WriteString(s) : Ecrit la chaîne de caractères **s** à partir de la position courante.

- EraseLine() : Efface le contenu de la ligne courante depuis la position courante.

- EraseScreen() : Efface le contenu de l'écran depuis la position courante.

- NewLine() : Effectue un "passage à la ligne".

- CursorAt(x,y) : déplace la position courante en (x, y).

- SetParam(select,value) : Affecte a l'argument implicite de nom **select** la valeur **value.**

- GetParam(select) : Renvoie la valeur courante de l'argument implicite de nom **select.**

Semi-graphiques

Les compétences semi-graphiques permettent de manipuler des représentations graphiques orientés caractères ; elles sont compatibles avec le modèle alphanumérique du terminal logique.

- DrawHLine(w) : Trace une ligne horizontale de largeur **w** à partir de la position courante. Une largeur positive implique un tracé vers la droite, une largeur négative, un tracé vers la gauche.

- DrawVLine(h) : Trace une ligne verticale de hauteur **h** à partir de la position courante.
  Une hauteur positive implique un tracé vers le bas.

- DrawBox(w,h) : Trace un rectangle de largeur **w** et de hauteur **h** à partir de la position courante.

La spécification précise de ces compétences est fort longue à cause des nombreux cas particuliers se produisant aux valeurs limites de la position d'écriture.

Voici une spécification plus complète de la compétence WriteChar, limitée cependant au cas d'un caractère affichable.

Spécification de la compétence WriteChar

1. Cas d'un caractère affichable

1.1. Si **c** est le code ASCII d'un caractère affichable et si la position courante est valide ($0 \leqslant$ xcursor $<$ width et $0 \leqslant$ ycursor $<$ height) alors l'exécution de WriteChar(c) affiche le caractère **c** à la position courante, avec les valeurs courantes d'attributs et déplace la position courante au caractère suivant de la même ligne.

1.2. Si la position courante est invalide c'est-à-dire après la fin de la ligne (xcursor = width) alors si la valeur de l'attribut AutoWrap est égale à "true", alors

1.2.1. Si la position courante n'est pas sur la dernière ligne de l'écran, alors la position courante est déplacée au premier caractère de la ligne suivante et on procède comme en 1.1, c'est-à-dire que le caractère c est affiché en première position

1.2.2. Si la position courante est sur la dernière ligne de l'écran et si la valeur courante de l'attribut AutoScroll est "true", alors :

- on fait défiler le contenu de l'écran d'une ligne vers le haut en supprimant la première ligne et en insérant une ligne blanche en fin d'écran (scroll-up).

- la position courante est déplacée au début de cette dernière ligne et on procède comme en 1.1.

1.3. Dans tous les autres cas, on ne modifie pas l'écran logique (ni aucun de ses attributs).

2. Cas d'un caractère de contrôle

Les caractères de contrôle standards (BS, HT, LF, CR) sont interprétés par le terminal logique. Ceci n'augmente pas les fonctionnalités effectives du terminal logique, mais assure une compatibilité avec le logiciel existant (langage de programmation, format des fichiers, texte ...).

Les caractères de contrôle non standards sont soit ignorés, soit provoquent l'affichage d'un caractère particulier , en fonction de la valeur courante d'un attribut approprié.

3.1.2.2. Le clavier logique

Le clavier logique détecte les événements "frappe d'une touche logique". Chaque fois qu'une touche est frappée, le clavier émet un message à destination de l'application. Le clavier logique possède donc une compétence permettant à l'application de déterminer l'identité du serveur concerné. Celui-ci recevra en argument le code ASCII correspondant à la touche.

Compétence du clavier logique

- SetKeyHandler(kh) : Après exécution de cette compétence, le processus de l'application identifié par **kh** sera activé à chaque frappe de touche du clavier logique, avec pour argument le code de la touche.

3.1.2.3. La souris logique

La souris physique est un dispositif mécanique que l'on peut déplacer sur une surface horizontale. Les déplacements de la souris sont liés à ceux d'un curseur sur l'écran (à ne pas confondre avec le surseur correspondant à la position courante d'écriture sur l'écran). La souris dispose d'un certain nombre de boutons (de 1 à 4). La pression d'un des boutons de la souris s'appelle "clic de désignation" car elle correspond à désigner l'objet couramment affiché à la position du surseur de la souris.

La souris logique transmet ses déplacements et l'état de ses boutons par envoi de messages vers les serveurs de l'application. La souris logique possède des compétences qui permettent à l'application de spécifier l'identité de ses serveurs concernés ainsi que la position et la forme du curseur sur l'écran.

Compétences de la souris logique

- SetMoveHandler(mh) - Après exécution de cette compétence, la procédure de l'application identifiée par **mh** sera appelée à chaque déplacement de la souris logique, avec pour argument un code indiquant la direction du déplacement (haut, bas, gauche, droite).

- SetClickHandler(ch) - Après exécution de cette compétence, la procédure de l'application identifiée par **ch** sera appelée à chaque pression d'un des boutons de la souris logique, avec comme argument le numéro du bouton.

- SetMousePos(x,y) - Positionne le curseur associé à la souris logique en (x, y).

- GetMousePos() - Renvoie la position courante du curseur associé à la souris logique.

- SetMouseForm(c) - Ddéfinit le caractère représentant le curseur associé à la souris logique.

- GetMouseForm() - Renvoie le caractère courant représentant le curseur associé à la souris logique.

3.1.2.4. Conclusion

Le modèle classique du terminal est un modèle de bas niveau d'abstraction ; il est proche des caractéristiques physiques du matériel.

Par sa trop grande dépendance vis-à-vis des particularités des premiers terminaux (télétypes), il n'a pas pu suivre l'évolution des capacités du matériel dans ce domaine.

Il en résulte que la plupart des programmes actuels (y compris les systèmes d'exploitation) sont basés sur le modèle "ligne à ligne" du télétype.

Non seulement les capacités des terminaux actuels sont largement sous-exploitées, mais le mode d'interaction qui en résulte est rigide et inadapté à une tâche réellement interactive (présence d'informations éventuellement périmées sur l'écran, mélange des commandes et des résultats des traitements ...).

Nous proposons un modèle basé sur les fonctionnalités des terminaux à écran alphanumérique. En cachant la manière dont ces fonctionnalités sont mises en oeuvre, le modèle du terminal logique abstrait les spécificités du terminal physique.

3.1.3. REALISATION DU MODELE DU TERMINAL LOGIQUE

3.1.3.1. Alternatives de réalisation

On peut imaginer essentiellement trois techniques pour réaliser le terminal logique :

- La première consiste à réaliser un module émulateur de terminal idéal pour chaque type de terminal réel que l'on désire utiliser. Le module adéquat est alors chargé au moment de l'activation de l'interface usager.
  Une telle solution procédurale est très souple, mais elle nécessite la possibilité de chargement dynamique.

- Une autre technique consiste à implémenter un processus séparé qui agit comme un traducteur entre l'interface usager et le terminal réel.
  Sous UNIX, par exemple, la communication interprocessus peut être réalisée au moyen de **pipes.**
  En fournissant de nouveaux processus traducteurs, il est possible d'utiliser de nouveaux types de terminaux, et ceci sans modifier l'interface usager. Cependant, la plupart des systèmes d'exploitation rendent cette approche trop inefficace.

- Une troisième solution consiste à décrire les particularités de chaque type de te"rminal utilisé dans une base de données appropriée. Cette description sera exploitée par un émulateur unique, capable d'effectuer toutes les traductions nécessaires.
  Contrairement aux deux techniques précédentes, cette approche est plus déclarative. Cependant, le concepteur doit prendre grand soin d'identifier un ensemble de paramètres de description de terminal qui couvrent les caractéristiques de l'ensemble du matériel existant et à venir.

C'est cette approche qui a été adaptée dans le système UNIX. On y trouve un fichier de description de terminaux nommé **termcap** (pour "terminal capabilities") et deux librairies ; l'une fournissant des primitives de lecture du contenu de termcap, l'autre fournissant certaines primitives de gestion d'écran indépendantes du terminal réel.

Les terminaux actuels (dits "intelligents") comportent des fonctions d'édition locale de plus en plus évoluées.
L'usage de ces fonctions est fortement recommandé dans un contexte interactif car elles apportent un confort visuel non négligeable en diminuant le temps de mise à jour de l'écran.

Considérons, par exemple, l'insertion ou la suppression d'une ligne. Si ces fonctions doivent être réalisées par le module du terminal logique, elles nécessitent alors la réécriture de la partie de l'écran située en dessous de la ligne concernée. Sur un écran de 24 lignes x 80 caractères cela fait en moyenne 1.000 caractères à envoyer sur la ligne, ce qui prend un temps non négligeable (de l'ordre de la seconde) et attire l'attention visuelle de l'usager en dehors du centre d'intérêt courant (la ligne insérée ou supprimée).

Si par contre, le terminal réel est capable d'effectuer localement l'insertion ou suppression de ligne, alors il suffit de déclencher cette fonction par l'envoi de codes appropriés (typiquement une séquence de contrôle de 3 ou 4 caractères).
Pour l'usager, l'effet apparaît alors comme instantané et localisé.

Il semble donc clair que le facteur limitatif à l'utilisation des terminaux classiques soit la faible bande passante de la ligne de transmission les reliant à l'ordinateur. Toute réalisation sérieuse ne peut ignorer les fonctions d'édition locale des terminaux actuels car elles permettent d'éviter partiellement les inconvénients dûs à ce goulot d'étranglement.

Un premier problème provient cependant de la disparité des séquences de contrôle permettant d'activer les fonctions d'édition locale.

Un second problème se pose, dû à la variété de ces fonctions qui, bien que possédant des caractéristiques semblables d'un terminal à l'autre, ne sont jamais totalement identiques.

Un troisième problème provient de l'existence d'erreurs de codage qui apparaissent fréquemment dans le logiciel de contrôle interne au terminal.

Il n'est pas étonnant, dans ces conditions, que la description des caractéristiques des terminaux ne puisse se faire de manière systématique.
Une analyse du fichier **termcap** de UNIX sera éclairante à ce sujet [Annexe A].

Tout d'abord, l'aspect cryptique du contenu apparaît immédiatement (Ce ficier est pourtant construit et mis à jour avec un éditeur de texte habituel). Ceci est dû, tout d'abord à un choix malheureux des identifiants des fonctions (2 caractères), mais surtout aux caractéristiques de la communication avec le terminal, c'est-à-dire du codage complexe des fonctions et arguments en séquences de contrôle.

En pratique, on constate à ce sujet, que l'imagination des constructeurs de terminaux est telle que la prise en compte d'un nouveau type de terminal dans le fichier descriptif **termcap** nécessite fréquemment la définition d'un nouveau format pour décrire le codage particulier à ce type de terminal et donc une modification du code de l'émulateur qui doit interpréter le contenu du fichier ... ce que l'on désirait éviter.

Ce fait est encore renforcé par les anomalies de comportement dont souffrent de nombreux terminaux. Celles-ci varient non seulement d'une marque à l'autre et d'un modèle à l'autre, mais également en fonction du numéro de série (les erreurs sont parfois corrigées après un certain temps). La découverte d'une anomalie de fonctionnement entraîne l'ajout d'un nouveau type de particularité dans le fichier descriptif et également une nouvelle modification de l'émulateur [Annexe A].

Celui-ci doit donc, non seulement pouvoir effectuer tous les types de codage existant, mais en plus prendre en compte toutes les anomalies de fonctionnement découvertes sur tous les types de terminaux. Il devient donc fort complexe, coûteux à réaliser, encombrant ... et comporte éventuellement ses propres erreurs.

En conclusion, plutôt que de tenter de formaliser ce qui ne peut l'être et de réaliser un interpréteur pour un formalisme en constante évolution, nous avons opté pour l'approche procédurale qui consiste à réaliser un module **terminal logique** pour chaque type de terminal réel.

3.1.3.2. Comment cacher les fonctionnalités du terminal réel

Nous décrivons dans cette section, une méthode permettant d'optimiser l'utilisation des fonctions d'édition locale des terminaux dits "intelligents", tout en cachant leurs spécificités et même leur existence à l'application. Certaines idées développés ici sont suggérées dans [Stallman 81]

Plutôt que de reporter instantanément sur l'écran réel les effets des opérations de l'écran logique, nous retardons sa mise à jour en introduisant la notion de **transaction** pratique, on utilise deux écrans, l'un représentant le contenu de l'écran logique, l'autre le contenu de l'écran réel. Entre deux transactions, l'écran réel doit refléter exactement l'écran logique, et les deux contenus sont donc identiques. Durant une transaction cependant, les contenus peuvent différer, l'écran réel n'étant pas mis à jour.

En fin de transaction, un algorithme spécifique est utilisé pour rétablir la cohérence. Celui-ci est adapté aux spécificités du terminal réel en utilisant les fonctions d'édition locales au terminal de manière à réduire le débit d'octets sur la ligne de transmission et optimiser ainsi le temps effectif de mise à jour de l'écran réel.

Pour l'application, le pouvoir d'abstraction d'un tel algorithme provient de la simplicité de sa spécification, malgré la grande variété des implémentations possibles.
Concrètement, cela se traduit par l'ajont d'une nouvelle compétence, soit **UpdateScreen** dont la spécification peut s'énoncer " Après exécution de Update Screen, le contenu de l'écran visible par l'usager reflète exactement l'état de l'écran logique " .

Le prix apparent à payer dans une telle approche est de rendre explicite la distinction entre écran logique et écran réel et de faire apparaître la notion de transaction, avec la nécessité d'appeler aux moments apportuns l'algorithme de réaffichage. Celui-ci est chaque fois réactivé entièrement et non pas à partir de la position d'interruption.

Il est cependant possible de cacher à l'application l'existance de l'algorithme de réaffichage en l'imaginant comme un processus séparé s'exécutant en parallèle. Tant que l'application effectue un traitement, le processus de réaffichage est bloqué, car de priorité inférieur, et les modifications de l'écran logique ne sont pas répercutées sur l'écran réel. Lorsque l'application termine son traitement et se met en attente implicite de message, le processus de réaffichage est activé provoquant ainsi la mise à jour de l'écran.

Remarquons que si une commande survient durant cette mise à jour d'écran, le processus de réaffichage est interrompu au profit de l'application. Cette approche a pour effet de réduire le flux de caractères à destination du terminal en n'envoyant pas des informations qui sont susceptibles de devenir périmées après l'exécution de cette commande.

3.1.3.3. Implémentation de l'algorithme de réaffichage

La séparation entre l'écran logique et l'écran réel est implémentée par deux contenus d'écrans représentés en mémoire. L'écran logique, encore appelé **écran désiré**, est modifié par les compétences du terminal logique. L'écran réel, encore appelé **écran courant** est l'image fidèle de l'écran physique, tel qu'il apparaît aux yeux de l'usager.

La spécification de l'algorithme de réaffichage est alors :

" Etant donné une image d'écran courant et une image d'écran désiré, l'algorithme de réaffichage va transformer l'écran courant en l'écran désiré d'une manière qui soit la plus efficace possible. "

L'invariant est, bien entendu, la cohérence entre le contenu de l'écran courant et celui de l'écran physique. Cet invariant doit être vérifié dans les sections interruptibles de l'algorithme de réaffichage.

Toute la complexité du processus est reportée dans les termes "la plus efficace possible". C'est ici qu'interviennent les spécificités du terminal réel.

L'algorithme de réaffichage pouvant être exécuté fort fréquemment (typiquement lors de l'écho de chaque caractère, pour un éditeur de texte), il faut rechercher un compromis entre le temps d'exécution et le nombre d'octets envoyés vers le terminal.

Le réaffichage comprend deux étapes :

- déterminer les discordances entre l'écran courant et l'écran désiré et
- déterminer la manière optimale de supprimer ces discordances.

Ces deux étapes sont intimement liées. La manière optimale de mettre l'écran courant à jour est d'utiliser les fonctions d'édition locale du terminal réel. La première étape consiste alors à déterminer quelles sont les conditions d'applicabilité de ces fonctions qui se présentent.

Le cas le plus simple est celui de l'écran ne possédant que l'écriture de caractère et l'adressage du curseur. Le réaffichage consiste alors à comparer les deux écrans, caractère par caractère, et à envoyer la différence vers le terminal.

En supposant que les écrans sont représentés par des tableaux bidimensionnels de caractères :

```
pour i:= o à hauteur-1

   pour j:= o à largeur-1

      si courant i,j   désiré i,j

         alors  CurseurEn(i,j);
EcrireCaractère(désiré i,j );
courant i,j :=désiré i,j ;
```

En remarquant que positionner le curseur nécessite également l'envoi d'un certain nombre d'octets, on améliore l'algorithme en ne positionnant le curseur à un endroit que s'il ne s'y trouve pas déjà.

L'algorithme que nous avons implémenté est basé sur ce principe. Il prend en compte les changements d'alphabets et de valeurs d'attributs et optimise le déplacement du curseur par un adressage absolu ou relatif Annexe

Les opérations d'insertion et de suppression de lignes et de caractères permettent de **déplacer** des lignes de texte vers le haut ou vers le bas en supprimant ou en insérant des lignes blanches. Elles permettent également de déplacer du texte vers la gauche ou vers la droite en supprimant ou en insérant des caractères individuels.

La prise en compte de ces fonctions évoluées conduit à un accroissement considérable de la complexité de l'algorithme de réaffichage dont le temps d'exécution devient alors prohibitif.

Le temps d'exécution de l'algorithme précédent est visiblement de l'ordre de O(w.h) si w est la largeur et h la hauteur de l'écran. Le calcul de la séquence optimale d'insertions/suppressions fait appel au problème de "la correction de chaîne à chaîne" (une généralisation du problème consistant à trouver la plus longue sous-séquence). Pour une dimension, ce problème admet une solution en O(l2) où l est la longueur de la chaîne. Des solutions existent, permettant de diminuer légèrement ce temps d'exécution , mais elles ne sont effectives que pour des problèmes de grande taille (ce qui n'est pas notre cas : une ligne comportant typiquement 80 caractères et une page 24 lignes).
Notre problème étant à deux dimensions, le temps d'exécution d'un algorithme calculant le séquence optimale d'insertions/suppressions est de l'ordre de O(w2.h2) ce qui conduit à des performances inacceptables.

Des techniques spéciales doivent, dans ce cas, être mises en oeuvre pour faciliter la comparaison efficace des deux contenus d'écrans [Gosling 81] .

Nous n'avons cependant pas suivi cette voie ; non pas à cause des problèmes d'efficacité, mais parceque les fonctions fournies par les terminaux courants ne correspondent pas aux besoins du gestionnaire de fenêtres décrit plus loin.
Le gestionnaire de fenêtres décompose l'écran physique en plusieurs sous-écrans indépendants. Alors que les fonctions du terminal physique affectent le contenu de l'écran tout entier, les fonctions utiles au gestionnaire de fenêtres devraient porter sur un sous-écran sans effecter les zones non concercées. Par exemple, l'insertion d'une ligne dans un sous-écran ne doit pas affecter les lignes des autres sous-écrans à sa droite et à sa gauche.
Ce problème est encore renfocé par la superposition partielle de sous-écrans ce qui leur confère une représentation visuelle non nécessairement rectangulaire.

Ceci réduit malheureusement considérablement l'utilité des fonctions d'édition locale des terminaux actuels.

Nous avons exploré dans ce chapitre différentes techniques de réalisation du terminal logique. En justifiant les choix relatifs à notre implémentation, nous avons tenté de mettre en lumière les difficultés liées à la réalisation d'un module interface présentant, côté application, un haut niveau d'abstractions pour un matériel doté de caractéristiques fort diverses.

3.2.LA REPRESENTATION EXTERNE

Dans ce chapitre, nous mettons en évidence les difficultés liées à la construction et la mise à jour de la représentation visuelle des objets.
Pour instaurer un dialogue de haut niveau d'abstraction entre l'application et l'interface usager, nous introduisons la notion de représentation externe, implémentée par un arbre de boîtes.

La notion de boîte n'est pas nouvelles [Knuth 79] et a déjà été mise en oeuvre dans différentes réalisations [Chamberlin 81, Hammer 81, Mikelsons 81]. La description que nous présentons ici est basée sur la RE développée pour l'atelier logiciel ADELE [Herrmann 84, Coutaz 84].

Pour chaque objet qu'elle présente à l'usager, l'application doit gérer deux représentatins :

- la **représentation interne (RI)**, structure de données adaptée aux traitements que l'application effectue, et

- la **représentation visuelle (RV)**, adaptée à l'usager. Celle-ci est basée sur des facteurs psychologiques et cognitifs et tient compte de l'insertion de l'objet représenté dans le modèle conceptuel de l'usager.

L'application est, non seulement, chargée de créer et mettre à jour ces deux représentations, mais également d'assurer leur **cohérence** à tout moment.
Dans un contexte hautement interactif, toute modification interne doit être instantanément reflétée à l'usager, et non pas sur demande de ce dernier, comme c'est habituellement le cas.

Le fait que ces deux représentations doivent satisfaire à des exigences totalement différentes leur confère des caractéristiques qui souvent s'opposent.

Il y a donc, en général, conflit de structure entre la représentation interne et la représentation visuelle d'un même objet. Ceci rend la tâche de construction et de maintien de leur cohérence fort complexe [Annexe D].

L'approche traditionnelle à ce problème de multi-représentation consiste à réaliser un traducteur de RI en RV (parfois appelé "décompilateur"), qui est soit activé explicitement par l'usager (mais nous avons vu que ce n'est pas une bonne solution), soit automatiquement, à chaque modification de la RI.

Un tel traducteur devient rapidement complexe dès que l'on désire fournir à l'usager des facilités telles que le polymorphisme (différentes "vues" d'un même objet) et la sélection d'informations au moyen du dispositif de désignation.

Nous proposons un protocole de haut niveau d'abstraction entre l'application et l'interface usager, de manière à :

- diviser la complexité du processus de construction et de mise à jour de la représentation visuelle et

- donner une plus grande autonomie à l'interface usager en y incluant du code indépendant des caractéristiques d'une RI particulière.

Le dialogue entre l'application et l'interface usager est basé sur une structure de donnée intermédiaire appelée **représentation externe** (RE).
Le concept de base constituant le RE est la boîte. La boîte est une structure d'échange; elle permet l'expression du transfert d'informations entre l'application et l'interface usager.

(i) Dans le sens application vers interface usager, la boîte permet l'expression de relations topologiques entre informations affichables etla spécification dynamique d'attributs visuels.

Pour représenter les relations structurelles entre informations visibles, les boîtes sont organisées hiérarchiquement. La RE est donc réalisée sous forme d'un arbre de boîtes.

L'information principale associée à une boîte feuille est sa valeur. Le type de valeur habituellement utilisé est la chaîne de caractères, mais ce pourrait être n'importe quelle entité affichable (symbole graphique, figure géométrique, icône, ...).

La boîte composite décrit principalement les relations de composition entre ses boîtes composantes. C'est essentiellement à ce niveau que les réalisations existantes diffèrent, car elles sont toutes orientées vers une application spécifique (traitement de textes, édition de programmes, graphiques de CAO...).
Les attributs de composition (ou de mormatage) décrivent des relations topologiques. Les boîtes peuvent être composées verticalement (V), horizontalement (H) ou de manière plus complexe comme H et V et H ou V.

- Les boîtes composées par H et V seront concaténées horizontalement jusqu'à une certaine largeur maximale (fixée par la largeur de la boîte composite). Lorsque cette largeur est atteinte, il se produit un " passage à la ligne " et les boîtes restantes sont concaténées horizontalement sur une nouvelle ligne de référence. Ce mode de composition est utilisé pour du texte habituel.

- Les boîtes composées par H ou V seront toutes alignées horizontalement si la largeur de la boîte composite le permet et seront toutes alignées verticalement sinon. Un tel mode de composition est utilisé pour afficher les instructions d'un programme.

Certaines RE possèdent des boîtes **colles** (ou séparateurs) qui ne contiennent pas d'information affichable, mais permettent de spécifier des espacements variables (pour la justification de texte, par exemple).

- Les relations de composition entre boîtes expriment donc des positions **relatives.** La localisation des objets affichés, en termes de coordonnées d'écran, est inconnue de l'application.

- Les relations de composition expriment des positions **dynamiques.** Une modification de la taille d'une boîte feuille ou composite peut entraîner une réévaluation complète de la disposition des boîtes. Ce mécanisme est également transparent à l'application.

La boîte est dotée d'**attributs visuels** exprimant le choix du matériel typographique (police de caractères, couleur, ...) et les effets spéciaux (mise en évidence).

Les **attributs exprimant la répartition** permettent la prise en compte de la notion de " foyer d'attention " grâce au mécanisme d'élision [Mikelson 81].
Ceci permet de présenter l'information avec différents niveaux de détail. C'est ainsi que, lors de l'édition d'un texte, par exemple, plutôt que de présenter une "fenêtre" constituée de n lignes consécutives, l'on pourra présenter la structure gnérale condensée du texte (table des matières) dont un élément seulement (celui que l'on édite pour l'instant) est affiché en entier.

(ii) Dans le sens interface usager vers application, la boîte définit l'unité d'information désignable par l'usager au moyen du dispositif de sélection.

Lorsque l'usager effectue une opération de sélection, l'afficheur de boîtes détermine la boîte feuille couramment visitée par le curseur.
l'**attribut de contrôle d'accès** indique, pour chaque boîte, si celle ci est effectivement désignable. Dans le cas contraire, la structure d'arbre est remontée jusqu'à la rencontre d'un noeud possédant une valeur appropriée pour cet attribut. Ce noeud est la racine d'un sous arbre qui consstitue la représentation sous forme de boîte de l'objet que l'usager a sélectionné.
Ce mécanisme montre comment l'application contrôle la granularité de l'information désignable.

Les opérations de construction de la RV à partir de la RI, et de maintien de leur cohérence se font alors en deux étapes.

Dans un premier temps, un module appelé **Compositeur** ou **Formatteur** effectue la traduction de RI en arbre de boîtes. Ce module fait partie de l'application car il connaît les caractéristiques de la RI. Celles-ci ont cependant disparu dans l'arbre de boîtes, qui ne contient que des informations nécessaires à l'affichage. Le **Compositeur** assure donc l'indépendance de l'IV vis-à-vis de l'application en lui cachant les particularités de la RI.

Un second module, appartenant à l'IV et appelé **Afficheur** ou **Visualiseur** parcourt alors l'arbre de RE et le traduit en séquence d'opérations de construction et de mise à jour de la RV. Ce sont des opérations du terminal logique.
L' **Afficheur** réalise l'indépendance de l'application vis-à-vis des caractéristiques du support d'affichage, et en particulier, des dimensions de l'écran logique.

Le sens des flèches indique le flux d'information de **visualisation** (Application Usager). Ce sens s'inverse pour les informations de **désignation** (Usager Application).

La structuration hiérarchique des boîtes permet la propagation des valeurs d'attributs au mayen d'un mécanisme d'héritage.
Tout attribut dont la valeur n'est pas spécifié à un certain niveau, hérite de la valeur du niveau spupérieur. Il s'agit donc d'une "valeur par défaut" qui peut éventuellement être modifiée à un niveau inférieur.

L'évaluation des dimensions des boîtes fait cependant appel à un mécanisme plus complexe. Ce fait n'est pas mis en évidence dans la littérature.

Les dimensions (largeur, hauteur, point de référence) des boîtes feuilles sont déduites de leur valeur.
Par exemple, la largeur d'une boîte feuille contenant une chaîne de caractères est fixée par la longueur de cette chaîne. Ces dimensions remontent vers les boîtes composites lors de l'application des règles de composition.

Les dimensions de la boîte racine sont fixées par la taille du dispositif d'affichage (écran, fenêtre, ...). Ces dimensions sont propagées vers le bas, conditionnant ainsi l'application des règles de composition et provoquant éventuellement élisions ou

troncatures.

L'évaluation d'un arbre de boîtes, effectuée par l'Afficheur, consiste donc en un parcours de l'arbre nécessitant des choix (de type de composition) pour descendre, éventuellement invalidés lorsqu'on remonte. Le processus d'évaluation effectuant des retours en arrière (backtracking) est fort coûteux en temps d'exécution.
La représentation externe, en tant qu'arbre de boîtes, est quant à elle grande consommatrice de mémoire.

La réalisation de la RE ne sera donc envisagée que sur un poste de travail de haut de gamme (2M octets de mémoire centrale, processeur d'au moins 1 Mip).

En conclusion, voici les avantages présentés par l'utilisation des concepts développés dans ce chapitre :

- La représentation externe permet une séparation claire entre l'application et l'interface usager.

- La description de la représentation visuelle des objets est définie en termes de concepts abstraits permettant d'exprimer les relations structurelles, le polymorphysme et la répartition. La traduction de ces propriétés en termes de primitives de bas niveau pour le terminal logique est invisible de l'application.

- L'application et l'interface usager se partagent la tâche de construction et mise à jour de la RV.
  Le module effectuant la traduction RE - RV ne doit être réalisé qu'une seule fois.

- La représentation externe est une structure d'échange. Elle

unifie les concepts d'entrée et de sortie de l'inforlation en réalisant un lien bidirectionnel entre RI et RV.

- La représentation externe que nous avons décrite est très générale. L'arbre de boîtes est indépendant du choix de la RI et permet de décrire une gamme étendue de RV.

Dans ce chapitre, nous avons présenté la notion de Représentation Externe en tant que structure d'échange. La description de l'arbre de boîtes est basée su la RE développée pour l'atelier logiciel ADELE [Coutaz 84] . Celle-ci nous semble trop limitée à l'affichage de texte - de programmes. Des mécanismes doivent ètre ajoutés pour permettre la visualisation de tableaux et de textes en langue naturelle. La littérature ne fournit actuellement aucune évaluation précise quant à la réalisation et l'utilisation de l'arbre de boîtes.
Devant un tel "état de l'art", nous pensons que l'élaboration d'une structure d'échange est un projet de longue haleine; nous ne ne l'avons donc pas approfondi dans le cadre de ce mémoire.

3.3. L'USAGER LOGIQUE

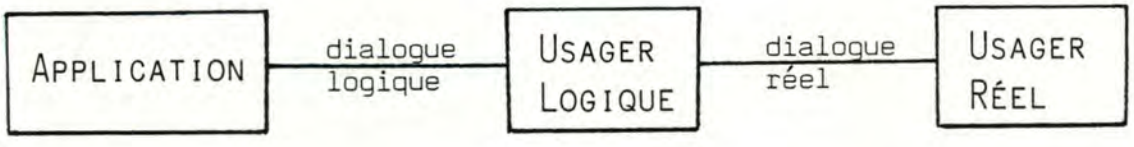
3.3.1. Description

Dans ce chapitre, nous introduisons le concept d'usager logique implémenté par le dialogueur pour assurer l'indépendance de l'application vis-à-vis du dialogue interactif.

Dans l'approche poursuivie jusqu'à présent, nous tentons de séparer ce qui est du ressort de l'application (partie fonctionnelle) de ce qui concerne l'interface usager.

Pour réaliser cela, nous introduisons des abstractions permettant d'exprimer et de réaliser clairement cette séparation. Tout comme nous avons abstrait le transfert de valeurs entre l'application et l'usager au moyen du concept de représentation externe, nous allons abstraire la manière dont l'usager contrôle l'enchaînement des différents traitements que l'application peut effectuer.

L'**usager logique** est un modèle, représentation simplifiée et idéalisée de l'usager réel.
Le protocole de **dialogue logique** régit l'interaction entre l'usager logique et l'application.
Le **dialogue réel** est défini par le choix d'une technique d'interaction qui caractérise la forme du dialogue effectivement utilisé par l'usager réel.



Parmi les différentes techniques d'interaction les plus utilisées, citons le type question - réponse, le formulaire, le menu et les langages du type "shell" de UNIX. Leurs avantages et inconvénients respectifs dépendent des particularités de la tâche (plus ou moins bien structurée, ...), et l'usager (degré d'expertise, ...) et éventuellement du matériel (existence d'un dispositif de désignation, ...).

L'approche classique consiste à définir un ensemble d'outils d'acquisition spécialisés chacun dans une technique d'interaction.
Ils permettent à une application de gérer les périphériques nécessaires à la réalisation d'une certaine forme de dialogue avec l'usager. C'est ainsi qu'il existe des "gestionnaires de menus", "gestionnaires de formulaires", etc...

Dans l'architecture qui résulte de ce type de démarche, le flot de contrôle réside dans l'application : la succession des interactions est déterminée lors de la specification de l'application et les appels aux outils d'interaction résident dans le corps de l'application.
C'est donc le logiciel qui, dans ce cas, contrôle explicitement le séquencement des opération. Ceci se traduit, pour l'usager, par une certaine rigidité du dialogue.

Comme il n'existe pas de technique d'interaction nettement préférable aux autres à tout points de vue, il est souhaitable de pouvoir disposer simultanément de plusieurs d'entr'elles de manière à pouvoir adapter dynamiquement la forme du dialogue aux particularités de l'application et de l'usager.

Une telle adaptation de la structure du dialogue réel devra rester transparente à l'application.

Pour réaliser ce but, nous allons extraite le flot de contrôle de l'application et le confier à un composant spécialisé

de l'interface usager : le **gestionnaire de dialogue** ou **dialogueur.**

3.3.2. Exemple de dialogue logique

Dans cette section, nous allons élaborer un dialogue logique basé sur le modèle orienté-objet de l'application.

Pour le dialogueur, l'application consiste en un ensemble de compétences correspondant chacune à un traitement atomique sur un certain type d'objet. Pour déclencher l'exécution d'une compétence, le dialogueur effectue une requête de l'application (envoi de message) spécifiant la compétence à exécuter (sélecteur) ainsi que les objets éventuels (arguments) sur lesquels portent le traitement.

L'ensemble des compétences activables à un moment donné, ainsi que l'ensemble des objets, arguments éventuels de ces compétences, sont déterminés par l'état courant de l'application. Cet état n'est pas directement accessible au dialogueur, et donc à l'usager logique.
Alors que la représentation externe fournit un moyen d'accès aux objets courants de l'application, il faut définir un mécanisme permetant au dialogueur de déterminer les compétences (ou commandes) courantes. Un tel mécanisme peut être implémenté par une liste, mise à jour par l'application à chaque changement d'état et consultée par le dialogueur chaque fois qu'il désire effectuer un traitement. C'est le procédé utilisé dans l'atelier logiciel ADELE [Herrmann 84].

Dans le dialogue logique, nous défirons un mécanisme plus souple et plus général permettant à l'usager logique de déterminer les compétences activables dans l'état courant et sur un objet donné. Ceci permet, entre-autres, la prise en compte de différents types d'objets par une même application.

Le protocole de dialogue logique est défini par un certain nombre de compétences (activés par messages en provenance du dialogueur) que doivent posséder toutes les applications interactives en plus de celles définies par leurs fonctionnalités propres. Ces compétences ont la particularité d'avoir une sémantique idépendante des applications et des objets. Elles définissent des commandes universelles ou **commandes génériques.**

En termes de programmation orientée-objet, le dialogue logique est l'ensemble des compétences de la classe **application.** Chaque application étant définie comme une sous-classe de la classe application, elle hérite donc du protocole de dialogue logique tout en rajoutant les compétences qui lui sont propres.

L'exécution d'une application est alors une instance de la classe de cette application. Il peut donc y avoir plusieurs instances d'une même application (l'éditeur de texte, par exemple), chacune possédant les mêmes fonctionnalités mais différant par les valeurs des variables d'instances (le contenu du tampon d'édition, la position courante ...).

L'ensemble minimal de compétences du dialogue logique comprend :

- La compétence **Initialiser()** effectue l'activation d'une application. Après création d'une nouvelle instance de l'application, celle-ci effectue l'initialisation de ses structures de données et déclare au dialogueur celles qui sont nécessaires à la communication (structures d'échanges).

- La compétence **Terminer()** signifie la fin de la session interactive avec cette application. Celle-ci effectue alors les sauvegardes nécessaires. L'instance de l'application est ensuite détruite ainsi que les objets qui lui ont été associés (fenêtre...).

- La compétence **QuellesOpérations?(obj)** qui permet à l'usager logique de connaître la liste des compétences que l'application peut effectuer dans son état courant et sur l'objet **obj** fournit en argument.

Dans un environnement orienté-objet, l'activation d'une compétence se fera par envoi à l'objet **obj**, d'un message dont le sélecteur est un élément de la liste renvoyée par **QuellesOpérations?(obj)**.
Dans un cadre plus général, on définira une compétence **Activer(compétence)**.

On y adjoindra, bien entendu, les commandes génériques **Défaire()** ("undo") et **Aide()** ("help").

Construire l'ensemble des commandes génériques consiste à (1) déterminer leur nombre optimal, ce qui fait appel à des facteurs psychologiques, et (2) déterminer les "verbes génériques" (le nom des commandes génériques), ce qui dépend de la classe d'applications considérée (système d'aide à la décision, atelier logiciel, bureautique,...).
Par exemple, les commandes génériques du Star de XEROX sont au nombre de huit. Elles se nomment MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, AGAIN, UNDO, HELP.

3.3.3.Conversion du dialogue réel en dialogue logique

Dans la section consacrée à la représentation de l'état de l'application, nous avons présenté les caractéristiques générales d'un modèle d'interaction du type "voir et pointer plutôt que se souvenir et taper".
Nous présentons ici un dialogue orienté-objet basé sur ce modèle et abordons le problème de sa traduction en dialogue logique.

Dans un dialogue orienté-objet, l'usager sélectionne d'abord l'objet et ensuite l'opération.

Le gestionnaire de dialogue traite les "clics de désignation" en provenance de la souris logique.

Voici un exemple de dialogue réel, et sa traduction en dialogue logique :

1) L'usager sélectionne un caractère sur l'écran.

2) Le module afficheur détermine le plus petit sous-arbre de boîtes désignable contenant ce caractère (cfr 3.2.2).

3) Le dialogueur effectue la mise en évidence de ce sous-arbre.

4) Le dialogueur envoie à l'application le message QuellesOpérations?, avec le sous-arbre de boîtes comme argument.

5) La réponse est une liste de noms. Le dialogueur fait apparaître cette liste sous forme d'un menu (qui peut également être décrit par un arbre de boîtes).

6) L'usager sélectionne un élément du menu.

7)Le dialogueur envoie à l'application un message d'activation de la compétence associée à l'élément sélectionné. Le menu disparaît.

Dans le cas d'un dialogue par menus hiérarchisés, au message QuellesOpérations?, l'application répondra par une liste arborescente. En parcourant cet arbre, le dialogueur affichera les différents niveaux de menus. Il y a alors répétition des étapes (5) et (6), sans interaction avec l'application.
Un usager expert, ayant mémorisé la structure des commandes de l'application, aura la possibilité d'activer directement toute compétence, en frappant son nom au clavier, quel que soit son niveau dans l'arbre des commandes. Ce fait restant transparent à l'application.

Dans le cas d'un langage de commandes en langue naturelle, le dialogueur effectuera par exemple une analyse syntaxique de la phrase, avec repérage de mots clés.
Les mots clés correspondant à des actions seront associés à des noms de compétences (en utilisant éventuellement des tables de synonymes) tandis que ceux qui correspondent à des entités seront associés aux objets concernés (après déréférenciation des pronoms, en utilisant la notion d'objet courant : le dernier objet sélectionné).

Dans tous les cas, l'interaction entre l'application et le dialogueur peut être basée sur le protocole de dialoguelogique introduit plus haut.

L'état de l'art dans ce domaine se limite, à notre connaissance, à une seule réalisation Herrmann 84 , fortement orientée vers un environement particulier.

La définition d'un dialogue logique qui soit à la fois suffisamment souple pour permettre la prise en compte d'une grande variété de dialogues réels et suffisamment abstrait pour cacher les particularités de ce dialogue réel est une entreprise à longue échéance.

L'activation d'une compétence de l'application peut requérir, en plus du nom de la compétence et de l'identité de l'objet à traiter, la désignation d'un certain nombre de paramètres.
Par exemple, la compétence "insérer" de l'éditeur nécessite comme paramètre la position d'insertion.

La sélection de plusieurs paramètres pour une même compétence se fait par remplissage d'un formulaire dans Smalltalk Tesler 81 et dans ADELE [Herrmann 84] . Le dialogueur de l'atelier logiciel Concerto [Carton 84] connait, pour chaque compétentence, le type et la position de ses paramètres.
Lors de la désignation d'un objet, le dialogueur détermine son type et en déduit sa position en tant que paramètre de la commande. Lorsque tous les paramètres ont été sélectionnés, la commande complète est envoyée à l'application.
Ce mécanisme a le désavantage de ne pas prendre en compte les paramètres optionnels.

Remarquons que l'absence d'une confirmation explicite est compensée par l'existence d'une commande générique permettant d'annuler les effets de la dernière opération.

3.4.L'APPAREIL VIRTUEL

En introduisant le concept d'appareil virtuel, nous montrons, dans ce chapitre, comment utiliser le poste de travail pour gérer simultanément plusieurs activités concourrantes.

Les systèmes d'exploitation habituels imposent un mode de travail séquenciel : l'exécution d'une tâche complexe doit se décomposer en une **suite** d'activités ; une activité devant être terminée pour pouvoir commencer l'activité suivante (exemple classique : le cycle édition, compilation, liaison, exécution).

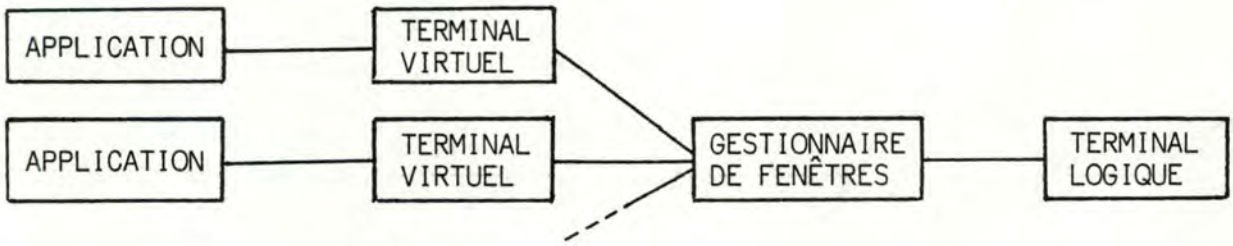
On constate cependant qu'en l'absence de telles contraintes, l'usager adapte spontanément un mode de travail consistant en un passage fréquent entre plusieurs activités à différents états d'avancement. Ces activités sont continuellement interrompues pour des causes internes (comme la nécessité d'un résultat intermédiaire) ou des causes externes (comme la survenance d'une tâche plus urgente).

Pour briser le lien application - terminal, et ainsi libérer le poste de travail du contexte d'exécution unique, on associe à chaque application un terminal virtuel, représenté par une fenêtre sur l'écran réel. Il est alors possible de visualiser simultanément plusieurs contextes d'exécution, chacun correspondant à une activité en cours. Le passage d'une fenêtre à l'autre permet alors de simuler les changements d'activité qui surviennent durant l'exécution d'une tâche.

Pour utiliser une métaphore de programmation, nous dirons que les différentes activités sont gérées comme des corantives plutôt que comme des procédures. Elles peuvent donc être interrompues et reprises ultérieurement dans le même état, contrairement aux procédures qui sont gérées de manière LIFO.

Le concept **d'appareil virtuel** permet d'effectuer le découplage de l'application vis-à-vis du terminal. De manière à rendre le dispositif transparent à l'application, le terminal virtuel possède, au vu de celle-ci, les mêmes fonctionnalités que le terminal logique.

Du point de vue de l'usager cependant, le terminal virtuel ne monopolise pas le poste de travail ; il est représenté par une fenêtre sur l'écran physique. Le découplage du clavier et de la souris est obtenu grâce au concept de fenêtre courante. C'est vers l'application s'exécutant dans la fenêtre courante que sont envoyés les caractères frappés au clavier et les **clics** de désignation. En changeant de fenêtre courante, il est ainsi possible de changer d'application courante.



Le gestionnaire de fenêtres, implémentant le concept de terminal virtuel, a une place particulière dans l'architecture générale de l'interface usager que nous développons.

Alors que les concepts d'appareil logique, d'usager logique et de représentation externe assurent l'indépendance de l'application vis-à-vis de l'interface usager, le terminal virtuel, quant à lui, concerne l'indépendance des applications entr'elles.

En effet, l'appareil virtuel permet le partage d'une ressource unique, tout en cachant ce fait aux applications. Celles-ci ne doivent donc pas tenir compte l'une de l'autre lors des accès concourrants à la ressource terminal réel.

Du point de vue de réalisateur d'interface usager, le gestionnaire de fenêtres effectue donc le lien entre les diverses applications concourrantes et l'unique poste de travail.

Pour l'usager, le gestionnaire de fenêtres est une application ; celle qui lui permet d'organiser le contenu de son écran de manière à refléter l'organisation de sa tâche.

3.5.UNE METHODOLOGIE DE CONCEPTION DE L'APPLICATION INTERACTIVE

Dans les chapitres précédents, nous avons développé un certain nombre de concepts ou abstractions, utiles pour la conception d'une architecture interactive.

Dans le but de mettre en évidence les relations existant entre ces concepts, nous développerons ici une approche constructive de la réalisation d'une application interactive. Par opposition à une tendance classique qui consiste à réaliser l'interface usager en dernier lieu, l'approche que nous proposons considère la relation usager - application comme un point de départ pour la définition et la mise en oeuvre des fonctionnalités de l'application interactive.

Cette approche comporte cinq étapes :

3.5.1.L'analyse de la tâche

La première étape de la conception d'une application interactive consiste à analyser le contenu de la tâche telle qu'elle est effectuée actuellement. Cette analyse permettra de déterminer un mode opératoire courant, exprimé en termes de manipulation (traitements) de certaines entités (objets, concrets ou abstraits).

Il s'agit en fait d'un modèle de la tâche telle qu'elle est perçue par l'usager. Ce modèle conceptuel ne peut être ignoré à aucun moment de la conception de l'application car l'usager y fera constamment référence durant l'utilisation de l'application.

3.5.2. Définition du modèle conceptuel de l'application

Le modèle conceptuel élaboré à l'étape précédente est à la base d'un autre modèle de tâche, celui qui décrit la tâche telle qu'elle sera effectuée en utilisant l'application.

On y définira les objets accessibles à l'usager ainsi que les traitements qu'il pourra commander, c'est-à-dire, les fonctionnalités de l'application.

3.5.3. Définition du modèle d'interaction

Le modèle d'interaction de l'application définit un nouvel envirionnement de tâche. Il décrit la manière dont l'usager peut interagir avec le modèle conceptuel défini à l'étape précédente.

C'est ici que l'on décidera de l'utilisation éventuelle d'une métaphore, de la représentation visuelle et du nom des commandes.

3.5.4. Implémentation des fonctionnalités de l'application

La réalisation "interne" du modèle conceptuel de l'application consiste à définir les structures de données représentant les différents objets du modèle et à concevoir les algorithmes implémentant les traitements du modèle.

Si cette réalisation est découpée en fonction des objets du modèle, elle consistera alors à un ensemble de "type abstrait de donnée".

Remarque : les étapes 3 et 4 sont concourrantes, mais cependant bien distinctes. Il est cependant préférable de définir le modèle d'interaction avant de réaliser les fonctionnalités de

l'application pour ne pas être tenté de faire apparaître des solutions d'implémentation.

3.5.5 Liaison de la partie fonctionnelle à l'interface usager

La dernière étape dans la réalisation de l'application interactive consiste à instaurer la communication entre l'implémentation des fonctionnalités de l'application et l'interface usager, implémentant le modèle d'interaction.

C'est ici qu'interviennent les abstractions développées aux chapitres précédents.

La représentation externe instaure la communication au niveau de la représentation visuelle des objets. Le concepteur établira, pour chacun d'eux, une structure d'arbre de boîtes adaptée à la fois à sa RI et sa RV. Il devra alors réaliser le module "Compositeur", qui effectue la génération de la RE de l'objet à partir de sa RI.

L'usager logique gère le dialogue au niveau des traitements. La partie fonctionnelle de l'application doit reconnaître le protocole de dialogue logique et, en particulier, implémenter les commandes génériques.

3.6.L'APPLICATION GESTIONNAIRE DE FENETRES

Nous développerons, dans ce chapitre, la réalisation de l'application **gestionnaire de fenêtres** de manière à illustrer les différentes étapes de conception décrites au chapitre précédent.

3.6.1. L'analyse de la tâche

Comme nous l'avons expliqué au paragraphe introduisant le concept de terminal virtuel, le gestionnaire de fenêtres (GF) doit permettre à l'usager d'effectuer plusieurs activités de manière concourante.

La tâche analysée à cette étape est en fait une méta-tâche : celle qui consiste à organiser les différentes activités constituant la tâche de l'usager.
De ce point de vue, le gestionnaire de fenêtres est donc un gestionnaire d'activités.
Analyser cette tâche de gestion d'activités se réduit alors à mettre en évidence les déficiences des systèmes d'exploitation courants qui interdisent à l'usager d'utiliser simultanément plusieurs applications.

3.6.2.Définition du modèle conceptuel de l'application GF

La solution au problème de monopolisation du terminal par l'application courante consiste à utiliser le concept d'appareil virtuel. Chaque application, ayant l'impression de posséder son propre terminal, peut ignorer l'existence d'autres applications.

Le modèle conceptuel du GF décrit ses fonctionnalités en termes d'applications, de terminal virtuel, d'application courante, etc...

3.6.3.Définition du modèle d'interaction

La fenêtre est la représentation visuelle de l'écran virtuel. Elle comprend :

- Le contenu : visualisant une partie du contenu de l'écran virtuel qu'elle représente.
- Le titre : permettant d'identifier l'application associée au terminal virtuel.
- Le cadre : qui délimite l'étendue de la fenêtre.

La partie visualisation du gestionnaire de fenêtres gère la composition des différentes fenêtres sur l'écran du poste de travail.
Certaines réalisations existantes permettent le recouvrement partiel au total des fenêtres ; d'autres composent une mosaïque formée de la juxtaposition des différentes fenêtres.

La première approche présente certains avantages :

- L'analogie des fenêtres se recouvrant sur l'écran du poste de travail avec les feuilles de papier sur un bureau est à la base d'un mode d'emploi plus naturel por l'usager.

- La taille et la position d'une fenêtre ne sont pas contraintes par l'existence des autres fenêtres, augmentant ainsi l'indépendance des applications.
  L'usager à toutes libertés pour déplacer et modifier les dimensions des fenêtres ; ces possibilités sont nettement plus limitées dans un système ne permettant pas le recouvrement.

- Cette approche permet une augmentation virtuelle de l'espace d'affichage en introduisant une troisième dimension.

Le dialogue avec le gestionnaire de fenêtres concerne :

- La sélection d'une fenêtre courante, et donc d'une application courante. Il s'agit en fait d'un méta-dialogue (dialogue sur le dialogue).

- L'activation des compétences liées aux caractéristiques des fenêtres : modification des dimensions, déplacement dans l'écran d'affichage, défilement du contenu (scrolling), etc...

Il existe à tout moment une et une seule fenêtre courante. Celle-ci est sélectionnée au moyen du dispositif de désignation. La fenêtre courante est totalement visible (elle est au-dessus des autres fenêtres) et son titre est mis en évidence.

Un "clic" dans la partie contenu de la fenêtre courante est transmis à l 'application courante ; il y correspond un "clic" dans son écran virtuel.

Un "clic" dans le titre ou sur le cadre d'une fenêtre fait apparaître le menu des compétences du gestionnaire de fenêtres (conceptuellement, celles de la fenêtre elle-même).

Un "clic" en dehors de toute fenêtre est assimilé à un "clic" dans l'écran de l'application GF. Son effet est identique à un "clic" dans le titre ou le cadre d'une fenêtre.

3.6.4. Implémentation des fonctionnalités de l'application GF.

Les concepts de **terminal virtuel**, de **fenêtre** et de **composeur** apparaissent comme des objets distincts dans notre réalisation.

En référence avec les concepts décrits au chapitre consacré à la représentation externe, on peut décrire le terminal virtuel comme la RI de l'application gestionnaire de fenêtres [Annexe C], sa

RV étant l'image d'une fenêtre telle qu'elle apparaît sur l'écran et la RE est alors constituée par l'objet "fenêtre" (en tant que type abstrait de donnée).

Le "composeur" (à ne pas confondre avec le "compositeur" de boîtes) est un module du GF chargé de construire l'image physique résultant de la composition des RV des différentes fenêtres.
L'implémentation du terminal virtuel fait appel aux techniques décrites dans la section 3.1.3 consacrée à la réalisation du terminal logique et est détaillée dans l'annexe B.

L'implémentation de la fenêtre et du composeur sera présentée à l'étape suivante consacrée à l'interface usager du gestionnaire de fenêtres.

Le terminal virtuel est basé sur le même modèle que celui du terminal logique. Il comporte un écran virtuel, un clavier virtuel et une souris virtuelle. Donc, du point de vue de l'application, l'écran virtuel possède les compétences de l'écran logique (telles que WriteChar, EraseLine, CursorAt...).
Cependant, l'effet de ces compétences n'est pas répercuté sur un écran réel, mais bien sur une image (en mémoire) du contenu de l'écran virtuel.

Cette image contient toutes les informations nécessaires à l'affichage du contenu de l'écran, y compris une position courante et des valeurs courantes pour les divers paramètres de l'écran logique.

L'écran virtuel possède une compétence supplémentaire permettant de recopier une partie rectangulaire de son contenu sur un autre écran. C'est la seule compétence qui permet de "lire" le contenu d'un écran virtuel.

3.6.5 Liaison de la partie fonctionnelle à l'interface usager

En ce qui concerne l'interface usager, l'application gestionnaire de fenêtres présente quelques particularités.

Tout d'abord, comme nous l'avons signalé au début de l'étape précédente, l'objet fenêtre est en fait la RE de l'application. L'arbre de boîtes ne convient (exceptionnellement) pas pour cette réalisation puisque nous désirons représenter le recouvrement des fenêtres. Pour une gestion de l'espace d'affichage basée sur une mosaïque de fenêtres, l'arbre de boîtes aurait parfaitement convenu.
Le "composeur de fenêtres", dont nous discuterons l'implémentation, correspond donc à l'"afficheur" de boîtes.

Un second point caractéristique de cette application est que l'usager n'a accès qu'aux compétences de la fenêtre et du composeur. Les compétences du terminal virtuel sont utilisées exclusivement par les applications.

Voici donc un schéma représentant de manière simplifiée la structure du gestionnaire de fenêtres, en termes des concepts que nous avons développés dans ce travail. (pp 87)

3.6.5.1.L'objet "fenêtre"

Un objet fenêtre a pour attributs toutes les informations nécessaires à la construction de la représentation visuelle d'un écran virtuel : les dimensions, la position et le titre de la fenêtre, la position du contenu affiché, ...

Du point de vue de l'usager, les compétences de la fenêtre lui permettent de modifier sa taille, sa position et les diverses caractéristiques visuelles.

Pour le composeur, la fenêtre possède une compétence qui construit sa représentation visuelle sur un écran donné.

3.6.5.2.Compétences de la fenêtre

- Move(x,y)	Déplace la coin supérieur gauche de la fenêtre à la position (x, y) de l'écran.
- Resize(w,h)	Modifie les dimensions de la fenêtre : largeur = w ; hauteur = h.
- MakeView(s)	Construit la représentation visuelle de la fenêtre sur l'écran **s**.

Remarque :

Ces compétences sont décrits selon une approche orientée-objet ; la fenêtre concernée est celle qui reçoit le message. Pour un langage classique de type Pascal ou C elles seront implémentées par des procédures possédant un argument supplémentaire permettant d'identifier l'objet cible (cet argument correspond au "self" de Smalltalk).

3.6.5.3.Le composeur

Le composeur est chargé de construire l'image physique résultant de la composition des représentations visuelles des différentes fenêtres. Le composeur gère la superposition éventuelle des fenêtres et maintient constamment l'écran de composition à jour de manière à ce que son contenu reflète fidèlement l'état des terminaux virtuels.

Le composeur est chargé de gérer l'interaction éventuelle des différentes fenêtres. Les fenêtres elles-mêmes s'ignorent mutuellement de manière à maximiser l'indépendance des diverses applications entr'elles.
Dans notre réalisation, l'interaction entre les fenêtres correspond au recouvrement.

Le composeur gère une liste des fenêtres potentiellement visibles. Cette liste est ordonnée selon l'axe de profondeur, la première fenêtre étant celle du fond et la dernière celle du dessus (la fenêtre courante).

Remarquons qu'imposer un ordre total sur l'ensemble des fenêtres est une facilité d'implémentation. Un ordre partiel permettant de comparer les fenêtres dont l'intersection est non vide est suffisant.

Les compétences du composeur permettent, d'une part, d'inserer et de supprimer des fenêtres dans la liste et, d'autre part, de construire l'écran de composition.

3.6.5.4.Compétences de gestion de la visibilité des fenêtres

- Ontop(w) — Place (ou déplace) la fenêtre **w** au dessus des autres fenêtres, la rendant ainsi totalement visible. Par effet de bord, d'autres fenêtres peuvent devenir partiellement ou totalement cachées.

- Bottom(w) — Place (ou déplace) la fenêtre **w** en dessous des autres fenêtres. Elle est alors éventuellement cachée (peut-être partiellement) par

d'autres fenêtres, tandis que d'autres peuvent de ce fait devenir visibles.

- Remove(w) — Supprime la fenêtre **w** de l'écran d'affichage.

Des compétences plus fines, basées explicitement sur l'ordre de profondeur des fenêtres (comme par exemple, l'insertion d'une fenêtre à un ième niveau) sont inutiles pour l'usager. Comme nous l'avons signalé, cet ordre est partiellement arbitraire et ne peut être déduit du simple examen de l'écran.

De plus, nous pensons qu'il est préférable de posséder un nombre limité de compétences aisément activables et dont l'effet est facile à décrire et donc à mémoriser plutôt qu'un grand nombre de fonctions complxes.
Pour l'éditeur de texte, par exemple, nous avons constaté que lorsque la compétence "NextChar" est aisément activable (par la touche), celle-ci est utilisée pour tous les déplacements vers la droite, plutôt qu'une compétence plus complexe, mais plus appropriée. C'est ainsi que l'usager préférera utiliser plusieurs fois "NextChar" pour déplacer le curseur au mot suivant plutôt que "NextWord" dont l'activation est un peu plus difficile et dont l'effet est plus compliqué à prévoir.

3.6.5.5. Compétence de construction de l'écran

- BuildScreen() — Construit, sur l'écran de composition, l'image résultant de la superposition des fenêtres.

Théoriquement, BuildScreen() doit être appelée chaque fois que l'écran de composition est susceptible d'être modifié, c'est-à-dire :

- lors d'une modification de la liste des fenêtres

- lors d'une modification des paramètres d'une fenêtre

- lors d'une modification d'un écran virtuel associé à une fenêtre.

Une telle manière de procéder possède au moins deux désavantages :

- il rend l'existence du composeur explicite aux fenêtres et aux terminaux virtuels.

- il ne semble pas raisonnable de reconstruire tout l'écran de composition à chaque modification du contenu d'une fenêtre. Des techniques incrémentales doivent être envisagées pour diminuer la fréquence d'exécution de Build Screen, et éviter ainsi des dégradations du temps de réponse.

Pour cacher l'existence du composeur aux autres objets et diminuer le nombre de reconstructions d'écran, on peut utiliser le même procédé que pour l'algorithme de réaffichage du terminal logique. On peut, en effet, considérer que le composeur de fenêtres est au terminal virtuel ce que le module de réaffichage et au terminal logique.

Le composeur de fenêtres est donc un processus parallèle aux applications, qui périodiquement met à jour l'écran de composition de manière à ce qu'il reflète l'état courant des différents écrans virtuels. Le composeur n'est pas appelé

explicitement, mais il se synchronise sur les périodes d'activité des applications.
Chaque fois qu'une application interactive termine un traitement, elle se met en attente implicite de caractères et de "clics". Le composeur est alors activé et met à jour la fenêtre associée à cette application.

3.6.5.6.Construction de l'écran de composition

Le principe de l'algorithme de construction de l'écran de composition consiste à parcourir la liste des fenêtres à afficher, en commençant par celle du fond et en "remontant", et à construire successivement leur représentation visuelle.

Soit Winlist, la liste des fenêtres du composeur, et S, l'écran de composition.

```
BuildScreen()
    var w : window;
    begin
        EraseScreen(S);
        w := First(Winlist);
        while (w     nil)
        do begin
            MakeView(w,s);
            w := Next(Winlist)
            end
    end
```

Telle quelle, cette manière de proceder est pariculièrement naïve car elle conduit à construire la représentation visuelle
(1) d'une fenêtre qui sera de toute façon cachée
(2) d'une fenêtre visible qui n'a pas été modifiée.

Les algorithmes présentés dans la littérature [Meyrowitz 81] ne nous ayant pas paru suffisamment convaincant, nous décrirons les algorithmes originaux qui ont été développés dans le cadre de notre implémentation du GF.

En considérant que le contenu des fenêtres change bien plus fréquemment que leur position ou leurs dimensions, nous allons développer un algorithme incrémental qui permet de répercuter les modifications du contenu des fenêtres sans nécessiter une reconstruction de tout l'écran de composition. La difficulté provient, dans ce cas ci, du recouvrement éventuel du contenu des fenêtres.

Si pour chaque caractère d'une fenêtre, on peut déterminer s'il est visible ou non sur l'écran de composition, alors il est aisé de mettre à jour le contenu d'une fenêtre. En effet, pour chaque caractère modifié (par l'application) depuis la dernière mise à jour, s'il est visible, on calcule sa position absolue sur l'écran (à partir de sa position dans la fenêtre et de la position de la fenêtre dans l'écran) et on l'y affiche.
Si le caractère en question n'est pas visible, il ne faut rien faire.

Il reste alors deux sous-problèmes à résoudre : (1) comment déterminer si un caractère est visible et (2) comment déterminer si un caractère a été modifié.

3.6.5.7.Détermination des caractères visibles

Le premier sous-problème est résolu de manière efficace par l'utilisation d'une table possédant autant d'entrées qu'il y a de caractères sur l'écran de composition et qui, pour chaque entrée, fournit un identifiant de la fenêtre à laquelle appartient le caractère correspondant.
Cette table a la même structure qu'un écran virtuel. Cependant, plutôt que de mémoriser, pour chaque position, la valeur du caractère qui y est affiché, on mémorise le numéro de la fenêtre. Cette approche permet de limiter la taille de la table à celle du contenu d'un écran virtuel (environ 2K octets) et permet d'utiliser les mêmes compétences d'accès au contenu.

Donc, pour déterminer si un caractère **c**, à la position relative (xc,yx) dans une fenêtre **f** dont l'origine est en (xf,yf) sur l'écran, est visible, on vérifie dans la table si l'entrée correspondant à la position (xc+xf,yc+yf) fournit bien l'identifiant de **f**.
Le problème est alors ramené à la construction de la table en question.

La table de visibilité des caractères est contruite en même temps que l'écran de composition. Lorsqu'une fenêtre construit sa représentation visuelle, à chaque position de caractère écrit sur l'écran de composition est associé l'identifiant de la fenêtre en question. Un caractère appartient donc à la dernière fenêtre qui a écrit à sa position.

Comme la table de visibilité des caractères est reconstruite à chaque construction de l'écran de composition, il est d'autant plus avantageux de l'utiliser que le rapport du nombre de mises à jour d'écran au nombre de reconstructions est grand.
Une reconstruction de l'écran de composition n'est à présent nécessaire que lors d'une modification du nombre ou de la disposition des fenêtres.

3.6.5.8.Autre utilisation de la table de visibilité

Non seulement, la table de visibilité permet une mise à jour efficace du contenu des fenêtres, mais elle est également d'une grande utilité lors de l'utilisation du dispositif de désignation.

Au niveau du terminal logique, un **clic** de désignation est caractérisé par la position courante du curseur associé à la souris logique sur l'écran.
Lors d'une désignation sur l'écran de composition, le gestionnaire de fenêtres doit pouvoir déterminer la fenêtre désignée de manière à changer de fenêtre courante ou bien, à dérouter le **clic** vers l'application concernée.

La difficulté du problème provient, à nouveau, du recouvrement éventuel des fenêtres. La table de visibilité y fournit une solution élégante.
En effet, étant donné une position sur l'écran, la table de visibilité nous donne immédiatement l'identifiant de la fenêtre contenant le caractère affiché à cette position.

3.6.5.9.Détermination des caractères modifiés

Une manière élémentaire pour déterminer les caractères modifiés est celle que nous avons décrite pour l'algorithme de réaffichage. Elle consiste à disposer de deux copies du contenu de la fenêtre : un contenu courant et un contenu désiré.
L'application modifiant le contenu désiré, il suffit de le comparer au contenu courant pour déterminer les changements.

Un tel procédé est cependant coûteux en place mémoire occupée (car il nécessite deux copies du contenu de chaque fenêtre) et en temps d'exécution (car il faut comparer tous les caractères du contenu alors qu'éventuellement un seul d'entre

eux seulement a été modifié).

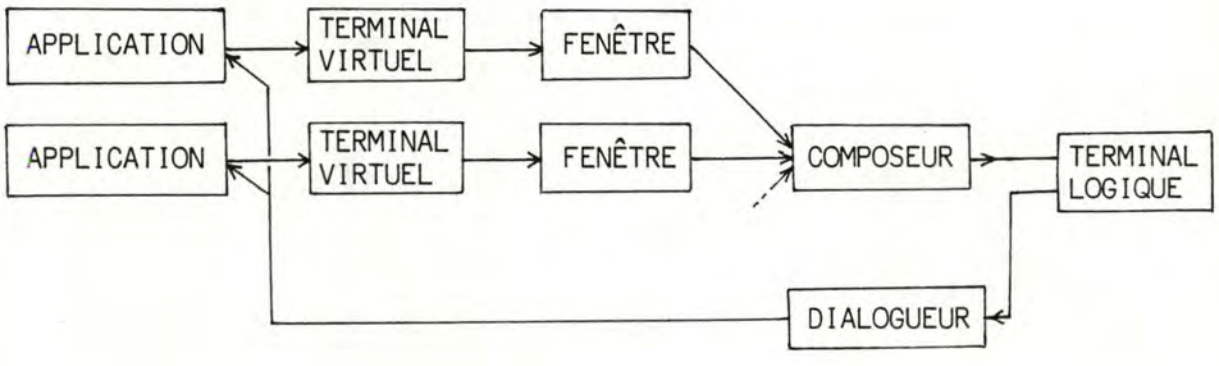
La solution adoptée consiste à ne mémoriser qu'un seul contenu, comportant des indications permettant d'accélérer la mise à jour.

On disposera donc, par exemple, d'un indicateur par fenêtre, indiquant si son contenu a été modifié ou non, ainsi qu'un tel indicateur par ligne de contenu de l'écran virtuel associé.

L'algorithme de mise à jour s'écrit alors :

```
pour chaque fenêtre
si elle a été modifiée
    alors pour chaque ligne
        si elle a été modifiée
            alors pour chaque caractère
                s'il est visible
                    alors recopier le caractère dans
                        l'écran de composition
```

Le problème est alors ramené à la mise à jour de ces indicateurs. Celle-ci est effectuée par l'écran virtuel à chaque modification de son contenu.



4. CONCLUSION

Dans ce travail, nous avons tenté de mettre en évidence des concepts propres à structurer la conception et la réalisation de l'interface usager des applications interactives.

La prise en compte du **modèle conceptuel de l'usager** conduit à organiser la manière dont l'application sera présentée à l'utilisateur.

La notion d'**état de l'application interactive** permet de présenter celle-ci comme un ensemble d'objets et d'opérateurs effectuant des manipulations sur ces objets. En représentant visuellement cet état et en donnant l'impression à l'usager qu'il agit directement sur les objets qu'il voit et qu'il désigne, on supprime virtuellement l'existence de l'application en tant qu'intermédiaire abstrait.

Pour le réalisateur d'application, nous avons introduit des abstractions qui permettent de découpler la partie fonctionnelle de l'application et le logiciel de dialogue avec l'usager.
Ceci conduit à la notion d'**interface usager universelle**, réalisée une seule fois pour un système tout entier, plutôt qu'à l'intérieur de chaque programme d'application individuel.

L'acquisition des commandes est réalisée par le **dialogueur**, implémentant l'**usager logique.** Il interagit avec l'application selon un protocole indépendant du dialogue réel.
Le dialogueur peut alors présenter différents styles d'interaction à l'utilisateur (menu, formulaire, langage de commandes).

La **représentation externe**, implémentée par un **arbre de boîtes**, met à la disposition des applications des mécanismes qui masquent les problèmes liés à l'affichage et à la gestion des informations visuelles.

Le **terminal logique** cache à l'application les particularités du terminal physique. Il possède les fonctionnalités d'un terminal idéal, indépendant du matériel utilisé.

Enfin, le **gestionnaire de fenêtres**, implémentant le **terminal virtuel**, assure l'indépendance de l'usager vis-à-vis d'un contexte d'exécution particulier en lui permettant d'utiliser concouramment plusieurs applications.

Apports, limites et perspectives

Les notions que nous avons développées ne sont pas fondamentalement nouvelles. Il nous a cependant semblé important de les réunir dans un seul travail et d'en montrer les interdépendances de manière à proposer au concepteur d'application un ensemble cohérent de concepts représentatifs de l'état de l'art en la matière.

Le domaine de la conception et de la réalisation de l'interface usager étant en pleine évolution, les concepts que nous avons présentés n'ont pas encore d'interprétation unanime et leurs alternatives de réalisation sont insuffisamment explorées.

Dans ce travail nous lions bi-univoquement l'application et son terminal virtuel. Cette approche est originale. La plupart des systèmes existants tels que Smalltalk Goldberg 83 , Interlisp Teitelman 79 , Macintosh Williams 84 , Adèle et Concerto Herrmann 84 , CWSH Weiser 85 , permettent à l'application de créer plusieurs fenêtres et lui donnent accès aux fonctionnalités de toutes les fenêtres, y compris celles créées par les autres applications. Les systèmes qui ne le permettent pas, tel Bruwin Meyrowitz 81 , le considèrent comme une extension valable.

Non seulement cette approche tend à provoquer une abandance de fenêtres sur l'écran, mais, plus fondamentalement, elle ignore la signification de la fenêtre en tant que terminal virtuel au profit d'un gadget visuel permettant d'augmenter artificiellement la surface d'affichage.

Nous postulons donc que les caractéristiques et l'existence même des fenêtres doivent être cachées à l'application. Bien entendu, nous reconnaissons l'utilité pour l'application de pouvoir décomposer la surface d'affichage en différentes zones, de manière à refléter une certaine structuration de l'information visible.
Nous proposons qu'une telle décomposition soit effectuée **à l'intérieur** de la fenêtre associée à l'application. Ceci peut être réalisé en augmentant le modèle du terminal logique/virtuel au moyen de la notion de "panneau", surface rectangulaire possédant les mêmes fonctionnalités que l'écran logique. Les différents panneaux formant alors une découpe en mosaïque de l'écran virtuel associé à l'application.

Cette approche a l'avantage de préserver l'indépendance des applications entre elles et de visualiser le lien sémantique qui existe entre les différents panneaux d'une même application.

En ce qui concerne le terminal logique, nous avons défini un modèle indépendant du concept de fenêtre. Les réalisations existentes soit reconnaissent la nécessité de l'indépendance vis-à-vis du matériel mais ne définissent aucun modèle (cfr termcap de Unix), soit confondent la notion de terminal logique et celle de terminal virtuel (et mélangent alors les fonctionnalités de l'écran logique et celles de la fenêtre. Ceci rejoint la discussion précédente).

En réalisant le terminal logique, nous avons soulevé les problèmes d'efficacité qui surviennent dans un environnement hautement interactif. Nous avons alors présenté des techniques incrémentales permettant de circonvenir ces inefficacités, au prix d'une complexité accrue des algorithmes.

Une extension intéressante à notre réalisation consiste à implémenter le modèle du terminal logique sur un matériel doté d'un écran graphique (tel le Macintosh).

Nous n'avons pas tenté d'implémenter l'arbre de boîtes et le dialogueur. Nous pensons, en effet, que tels qu'ils sont décrits, ils n'offrent pas encore la généralité à laquelle ils prétendent.

L'arbre de boîtes ne permet de définir que des relations hiérarchiques entre les constituants d'une image et les seules contraintes que l'application peut exprimer sont celles de positionnement des informations (composition) et de largeur maximum d'une boîte feuille. Ces possibilités permettent l'affichage de programmes, de formulaires ou de documents élémentaires, mais sont nettement insuffisantes dans un cadre plus général. Il est, par exemple, impossible de définir des tableaux ; en effet, la largeur d'un élément n'est pas déterminé par le contenu de la boîte feuille qui y correspond mais par la largeur de la colonne.

L'implémentation de la représentation externe posera également des problèmes d'efficacité. En effet, les structures arborescentes sont fort gourmandes en mémoire dès que le nombre de noeuds devient important. Il n'est donc pas raisonnable d'associer une boîte feuille à chaque caractère d'un texte affiché ; mais l'éditeur de textes doit néanmoins permettre la sélection d'un seul caractère. Un compromis doit être trouvé à ce niveau là.

De plus, l'évaluation de la position et des dimensions des boîtes est un processus fort coûteux en temps d'exécution. Ceci provient des "retours en arrière" (backtracking) qui se produisent lorsqu'un choix de composition est invalidé (par exemple dans le cas HauV, lorsque la composition horizontale échoue). Des techniques d'évaluation incrémentale devront certainement être développés.

Les limites du dialogueur proviennent du protocole de dialogue logique. Tel que nous l'avons défini, celui-ci ne peut raisonnablement être utilisé que pour un dialogue arienté-objet du type opérande opérateur .
Pour pouvoir réaliser un dialogue plus riche, il faut définir un protocole permettant de décrire le transfert d'informations relatives à l'application (par exemple, la structure hiérarchique de ses commandes) et spécifiques à la tâche (pour mettre en oeuvre des techniques de l'intelligence artificielle).

La définition de mécanismes permettant l'expression du transfert d'informations relatives à l'affichage et au dialogue est nécessaire à l'indépendance de l'interface usager vis-à-vis de l'application.

La représentation externe et l'usager logique constituent une étape intermédiaire dans la réalisation de mécanismes d'échange réellement généraux.

5. REFERENCES

Archer J.E., Conway R. (1982), "Display Condensation of Program Text.", IEEE Transactions on Software Engineering, Vol. 8, N° 5, September 1982, pp. 526-529.

Archer J.E., Conway R., Schneider F.B. (1984), "User Recovery and Reversal in Interactive Systems.", ACM Transactions on Programming Languages and Systems, Vol. 6, N° 1, January 1984, pp. 1-19.

Arnold K. (1981), "Screen Updating and Cursor Movement Optimization : A Library Package.", UNIX Programmers Manual, Seventh Edition, Berkeley Release 4.1, 1981.

Bailes P.A. (1985), "A Low-Cost Implementation of Coroutines for C.", Software-Practice and Experience, Vol. 15, N° 4, April 1985, pp. 379-395.

Barach D.R., Taenzer D.H., Wells R.E. (1981), "Design of the PEN Video Editor Display Module.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 130-136.

Brown P.J. (1983), "Error Messages : The Neglected Area of the Man/Machine Interface ?", CACM, Vol. 26, N° 4, April 1983, pp. 246-249.

Broze H. et al (1982), "Evaluation critique du Système UNIX.", Technique et Science Informatiques, Vol. 1, N° 4, 1982, pp. 315-324.

Carton J.L. (1984), "Contribution à l'Etude du Dialogueur de l'Atelier Logiciel Concerto : une Interface Intégrée, Interactive et Evolutive.", Mémoire, CNAM IIE, juin 1984.

Clanton C. (1983), "The Future of Metaphor in Man-Computer Systems.", Byte, Vol. 8, N° 12, December 1983, pp. 263-279.

Conchon A. (1984), "Le Poste de Travail Concerto", Rapport Interne CNET, juin 1984.

Coutaz J. (1984), "The Box, A Layout Abstraction for User Interface Toolkits.", Technical Report CMU-CS-84-167, December 1984.

Cox B.J. (1983), "The Object Oriented Pre-Compiler.", Sigplan Notices, Vol. 18, N° 1, January 1983, pp. 15-22.

Dutta K. (1985), "Modular Programming in C : An Approach and an Example.", Sigplan Notices, Vol. 20, N° 3, March 1985, pp. 9-15.

Goldberg A., Robson D. (1983), "Smalltalk-80 : the Language and its Implementation.", Addison-Wesley, Reading Mass., 1983.

Good M. (1981), "Etude and the Folklore of User Interface Design.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 34-43.

Gosling J. (1981), "A Redisplay Algerithm.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 123-129.

Gutz S., Wasserman A.I., Spier M.J. (1981), "Personal Development Systems for the Professional Programmer.", Computer, April 1981, pp. 45-53.

Hammer M. et al (1981), "The Implementation of Etude, An Integrated and Interactive Document Production System.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 137-146.

Hayes P.J. (1984), "Executable Interface Definitions using Form-Based Interface Abstractions.", Technical Report CMU-CS-84-110, March 1984.

Herrman M. (1984), "Interface Usager, Application dans un Atelier de Génie Logiciel.", Thèse, IMAG, 1984.

Ingalls D.H.H. (1981a), "Design Principles Behind Smalltalk.", Byte, Vol. 6, N° 8, August 1981, pp. 286-298.

Ingalls D.H.H. (1981b), "The Smalltalk Graphics Kernel.", Byte, Vol. 6, N° 8, August 1981, pp. 168-194.

Kay A., Goldberg A. (1977), "Personal Dynamic Media.", Computer, Vol. 10, N° 3, March 1977.

Kernighan B.W., Ritchie D.M. (1978), "The C Programming Language.", Prentice Hall Inc., 1978.

Mélèse B. (1983), "Mentor Rapport : Manipulation de Textes Structurés sous Mentor.", Rapport Technique N° 23, INRIA, avril 1983.

Mélèse B. (1984), "Edition Structurée-Edition Non Structurée, Coopération et Complémentarité.", AFCET 2è Collaque de Génie Logiciel NICE, 4-6 juin 1984.

Merritt D.R. (1984), "TERMCAP Unveiled.", UNIX Review, September 1984, pp. 42-48.

Meyer B. (1982), "Principles of Package Design.", Comm. ACM, Vol. 25, N° 7, July 1982, pp. 419-428.

Meyrowitz N., Moser M. (1981), "BRUWIN : An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems.", ACM SIGOPS Conference, 1981, PP. 180-189.

Mikelson M. (1981), "Prettyprinting in an Interactive Programming Environment.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 108-116.

Moran T.P. (1978), "Introduction to the Command Language Grammar.", Xerox PARC Technical Report SSL-78-3, October 1978.

Morgan C., Williams G., Lemmons P. (1983), "An Interview with Wayne Rosing, Bruce Daniels, and Larry Tesler.", Byte, Vol. 8, N° 2, February 1983, pp. 90-114.

Nievergelt J. (1983), "Design of Man/Machine Interfaces : Towards the Integrated Interactive System.", Technical Report, ETH Zurich Informatik, 1983.

Norman D.A. (1983), "Design Rules Based on Analyses of Human Error.", Comm. ACM, Vol. 26, N° 4, April 1983, pp. 254-258.

Olsen D.R., Dempsey E.P. (1983), "Syntax Directed Graphical Interaction.", Proceedings Sigplan 83, Symposium on Programming Languages Issues in Software Systems, pp. 112-117.

Perq (1984), "User's Guide to the Sapphire Window Manager.", PERQ Systems Corporation, January 1984.

Pike R., Locanthi B. (1985), "Hardware/Software Trade-offs for Bitmap Graphics on the Blit.", Software-Practice and Experience, Vol. 15, N° 2, February 1985, pp. 131-151.

Pilote M. (1983), "A Programming Language Framework for Designing User Interfaces.", Proceedings Sigplan 83, Symposium on Programming Languages Issues in Software Systems, pp. 118-136.

Robson D. (1981), "Object-Oriented Software Systems.", Byte, Vol. 6, N° 8, August 1981, pp. 74-86.

Rowe L.A., Shoens K.A. (1983), "Programming Language Constructs for Screen Definition.", IEEE Transactions on Software Engineering, Vol. 9, N° 1, January 1983, pp. 31-39.

Shani U. (1983), "Should Program Editors not Abandon Text Oriented Commands ?", Sigplan Notices, Vol. 18, N° 1, January 1983, pp. 35-40.

Shaw M., Borison E., Horowitz M., Lane T., Nichols D., Pausch R. (1983), "Descartes : A Programming Language Approach to Interactive Display Interfaces.", Proceedings Sigplan 83, Symposium on Programming Languages Issues in Software Systems, pp. 100-111.

Siegel M.L. (1984), "Towards Standardized Video Terminals. ANSI X3.64 Device Control.", Byte, Vol. 9, N° 4, April 1984, pp. 365-374.

Smith D.C., Irby C., Kinball R., Verplank B. (1982), "Designing the Star User Interface.", Byte, Vol. 7, N° 4, April 1982, pp. 242-282.

Sneeringer J. (1978), "User-interface Design for Text Editing : A Case Study.", Software-Practice and Experience, Vol. 8, 1978, pp. 543-557.

Snodgrass R. (1983), "An Object-Oriented Command Language.", IEEE Transactions on Software Engineering, Vol. 9, N° 1, January 1983, pp. 1-8.

Stallman R.M. (1981), "EMACS, the Extensible, Customizable Self-Documenting Display Editor.", Proceedings Sigplan Symposium on Text Manipulation, Vol. 16, N° 6, June 1981, pp. 147-156.

Stowe B. (1984), "Free and Bound Generics : two Techniques for Abstract Data Types in Modular C.", Sigplan Notices, Vol. 19, N° 3, March 1984, pp. 12-20.

Stroustrup B. (1983), "Adding Classes to the C Language : An Exercise in Language Evolution.", Software-Practice and Experience, Vol. 13, 1983, pp. 139-161.

Symbolics (1981), "Scroll Windows", Symbolics Inc., August 1981.

Teitelman W. (1979), "A Display Oriented Programmer's Assistant.", Int. J. Man/Machine Studies, Vol. 11, 1979, pp. 157-187.

Tesler L. (1981), "The Smalltalk Environment.", Byte, Vol. 6, N° 8, August 1981, pp. 90-147.

Tuthill B. (1984), "Terminal Interface Manipulations", UNIX Review, July 1984, pp. 56-58, 95.

Tuthill B. (1984), "Screen Handling from C Programs.", UNIX Review, August 1984, pp. 69-73.

Wasserman A.I., Stinson S.K. (1979), "A Specification Method for Interactive Information Systems.", Proceedings Specification of Reliable Software, Cambridge Mass., IEEE Computer Society, 1979, pp. 68-79.

Waters R.C. (1982), "Program Editors Should not Abandon Text Oriented Commands.", Sigplan Notices, Vol. 17, N° 7, July 1982, pp. 39-46.

Weiser M. (1985), "CWSH : The Windowing Shell of the Maryland Window System.", Software-Practice and Experience, Vol. 15, N° 5, May 1985, pp. 512-522.

William J., Mark M. (1981), "TERMCAP.", UNIX Programmers Manual, Seventh Edition, Berkeley Release 4.1, 1981.

Williams G. (1984), "The Apple Macintosh Computer.", Byte, Vol. 9, N° 2, February 1984, pp. 30-54.

Yalamanchili S., Malek M., Aggarwal J.K. (1984), "Workstations in a Lokal Area Network Environment.", Computer, Vol. 17, N° 11, November 1984, pp. 74-86.

ANNEXE A

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3X). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P•) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P•)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P•)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P•)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P•)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P•)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P•)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode

ei	str		End insert mode; give ":ei=:" if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give ":im=:" if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue

ul	bool	Terminal underlines even though it doesn't overstrike
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like ce \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

```
d0|vt100|vt100-am|dec vt100:\
        :cr=^M:do=^J:nl=^J:bl=^G:co#80:li#24:cl=50\E[;H\E[2J:\
        :le=^H:bs:am:cm=5\E[%i%d;%dH:nd=2\E[C:up=2\E[A:\
        :ce=3\E[K:cd=50\E[J:so=2\E[7m:se=2\E[m:us=2\E[4m:ue=2\E[m:\
        :md=2\E[1m:mr=2\E[7m:mb=2\E[5m:me=2\E[m:is=\E[1;24r\E[24;1H:\
        :rf=/usr/lib/tabset/vt100:\
        :rs=\E>\E[?3l\E[?4l\E[?5l\E[?7h\E[?8h:ks=\E[?1h\E=:ke=\E[?1l\E>:\
        :ku=\EOA:kd=\EOB:kr=\EOC:kl=\EOD:kb=^H:\
        :ho=\E[H:k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:ta=^I:pt:sr=5\EM:vt#3:xn:\
        :sc=\E7:rc=\E8:cs=\E[%i%d;%dr:
```

ANNEXE B : IMPLÉMENTATION DU TERMINAL LOGIQUE (EXTRAITS).

1°) VARIABLES D'INSTANCE (ATTRIBUTS).

```
struct vs_desc
{
        char    *scrmap;
        short   width;
        short   height;

        short   xmin,xmax;
        short   ymin,ymax;
        char    *sbeg,*send;

        short   xcursor,ycursor;
        char    *curpos;
        short   scrflags;
};

typedef struct vs_desc *virtscreen;

/* screen flags */
#define autowrap        1
#define autoscroll      2
#define graphic         4
#define emphasize       0200    /* ne pas modifier ! */
#define scrdefaults     (autowrap | autoscroll)
```

2°) CRÉATION D'INSTANCE.

```
virtscreen create(x, y, w, h)
int x, y, w, h;
{
        register virtscreen s;
        s = (virtscreen) malloc(sizeof (*s));
        if (s == nil)
           then return(nil);
        s->scrmap = malloc(w * h);
        if (s->scrmap == nil)
           then { free(s);
                  return(nil);
                }
        s->width = w;
        s->height = h;
        s->xmin = 0;
        s->ymin = 0;
        s->xmax = w;
        s->ymax = h;
        s->sbeg = s->scrmap;                    /* = MapChar(s, s->xmin, s->ymin) */
        s->send = s->scrmap + w * h;            /* = MapChar(s, 0, s->ymax) */

        vsinit(s);
        return(s);
}
```

```
writeChar(s, c)
virtscreen s; byte c;
{
        if (c >= ' ' && c <= '~')
           then vswrtCh(s, c);
           else switch (c)
           {
                case '\n': vswrtLn(s);
                           break;
                case '\t': s->xcursor = (s->xcursor & ~7) + 8;
                           if (s->xcursor > s->xmax)
                              then s->xcursor = s->xmax;
                           break;
                case '\r': s->xcursor = s->xmin;
                           break;
                case '\b': if (s->xcursor > s->xmin)
                              then --s->xcursor;
                           break;
           }
}


 vswrtCh(s, c)
 register virtscreen s;
 char c;
 {
        if (s->xcursor < s->xmax)
           then { *s->curpos++ = Encode(s, c);
                  ++s->xcursor;
                  return;
                }
        if (s->scrflags & autowrap)
           then { vswrtLn(s);
                  if (s->xcursor < s->xmax)
                     then { *s->curpos++ = Encode(s, c);
                            ++s->xcursor;
                            return;
                          }
                }
 }


writeString(s, st)
register virtscreen s;
register string st;
{
        register char c;

        while (c = *st++)
           vswrtCh(s, c);
}


#define Encode(s, c)    (((c) & ((s)->scrflags & graphic? 037 : 0177)) \
 | (s)->scrflags & emphasize)
```

```
Home(s)
register VirtScreen s;
{
        s->xcursor = s->xmin;
        s->ycursor = s->ymin;
        s->curpos = s->pbeg;
}



CursorAt(s, x, y)
register VirtScreen s; int x, y;
{
        if (x <= 0)
           then s->xcursor = s->xmin;
           else { x += s->xmin;
                  if (x < s->xmax)
                     then s->xcursor = x;
                     else s->xcursor = s->xmax - 1;
                }
        if (y <= 0)
           then s->ycursor = s->ymin;
           else { y += s->ymin;
                  if (y < s->ymax)
                     then s->ycursor = y;
                     else s->ycursor = s->ymax - 1;
                }
        s->curpos = MapChar(s,s->xcursor,s->ycursor);
}



Emphasize(s, b)
VirtScreen s; bool b;
{
        if (b)
           then s->scrflags |= emphasize;
           else s->scrflags &= ~emphasize;
}



NewLine(s)
register VirtScreen s;
{
        if (s->ycursor < s->ymax)
           then { s->xcursor = s->xmin;
                  ++s->ycursor;
                  s->curpos = MapChar(s,s->xcursor,s->ycursor);
                }
        else if (s->scrflags & autoscroll)
           then { s->xcursor = s->xmin;
                  s->curpos = MapChar(s,s->xcursor,s->ycursor);
                  vsScroll(s,1);
                }
}



#define MapChar(s, x, y)        ((s)->scrmap + (y) * (s)->width + (x))
```

4°) ALGORITHME DE RÉAFFICHAGE.

```
redisplay(nscreen, oscreen)
virtscreen nscreen, oscreen;
{
        register char *newc, *oldc, *rlimit;
        char *lastc, *llimit;
        int y, lasty, incr;

        y = nscreen->ymin;
        newc = nscreen->body;
        oldc = oscreen->body;
        incr = nscreen->width - (nscreen->xmax - nscreen->xmin);
        rlimit = newc + (nscreen->xmax - nscreen->xmin);
        lasty = -1;

        while (y++ < nscreen->ymax)
        {
            llimit = newc;
            do {
                lastc = newc;
                while (newc < rlimit && *oldc == *newc)
                    {   ++newc; ++oldc; }
                if (newc < rlimit)
                   then {
                        if (lasty == y)
                            then CursRight(newc - lastc);
                            else {
                                CursorAt(newc - llimit, y);
                                lasty = y;
                                }
                        do {
                            Decode(oscreen, *newc);
                            *oldc++ = *newc++;
                           }
                        while (newc < rlimit && *oldc != *newc);
                        }
                }
            while (newc < rlimit);
            newc += incr;
            oldc += incr;
            rlimit += nscreen->width;
        }
}


Decode(vs, c)
register virtscreen vs; register char c;
{
        if ((vs->scrflags ^ c) & emphasize)
           then if (c & emphasize)
                   then { EmphOn(); vs->scrflags |= emphasize; }
                   else { EmphOff(); vs->scrflags &= ~emphasize; }
        c &= 0177;
        if (c >= ' ')
           then { if (vs->scrflags & graphic)
                      then { GraphicOff(); vs->scrflags &= ~graphic; }
                  WriteChar(c);
                }
           else { if (!(vs->scrflags & graphic))
                      then { GraphicOn(); vs->scrflags |= graphic; }
                  WriteChar(c | 0140);
                }
}
```

ANNEXE C : IMPLÉMENTATION DU GESTIONNAIRE DE FENÊTRES (EXTRAITS).

Processus : implémentation sous UNIX du concept d'application. Le processus communique avec l'interface usager par deux flux de caractères (pipes).

Tâche : permet la synchronisation des différentes activités du gestionnaire de fenêtres. Une tâche est associée à chaque flux d'octets dirigé vers le gestionnaire de fenêtres (une tâche par processus, une tâche pour le clavier logique, une tâche pour la souris logique). La tâche convertit l'opération de lecture de caractères en évènements "arrivée d'un caractère".

```
struct proc_desc
{
        int     procid;         /* process-id */
        byte    pread;          /* pipe for receiving bytes */
        byte    pwrite;         /* pipe for sending bytes */
};

typedef struct proc_desc *process;

extern int StartProcess();

#define sendsignal(p, s)        (kill((p)->procid, (s)))
#define sendbytes(p, a, n)      (write((p)->pwrite, a, n))
#define readbytes(p, a, n)      (read((p)->pread, a, n))

#ifndef _IOCTL_
#include <sys/ioctl.h>
#endif

#define wakeup(p)               ioctl((p)->pread, FIONREAD, 0)
```

```
#include <stdio.h>
#include <signal.h>
#include "../h/stdefs.h"
#include "../h/process.h"

#define READ    0
#define WRITE   1

int StartProcess(proc, cmd)
process proc; string cmd;
{
        int pid,rpI2I,wpI2I;

        if (pipe(rp) < 0)
           then return(-1);
        if (pipe(wp) < 0)
           then {
                close(rpIREADI);
                close(rpIWRITEI);
                return(-1);
                }
        if ((pid = fork()) == 0)
           then {                       /* child */
                close(READ);
                dup(wpIREADI);
                close(wpIREADI);
                close(wpIWRITEI);

                close(WRITE);
                dup(rpIWRITEI);
                close(rpIWRITEI);
                close(rpIREADI);

                {   int i;
                    for( i=2; i < _NFILE; close(i++));
                }
                dup(WRITE);

/*              execl("/bin/csh","-Scoubidou","-c",cmd,nil); */
                execl("/bin/csh","-Scoubidou","-i","-f");
                _exit(1);
                }
        if (pid < 0)
           then {
                close(rpIREADI);
                close(rpIWRITEI);
                close(wpIREADI);
                close(wpIWRITEI);
                return(-1);
                }
        close(wpIREADI);
        close(rpIWRITEI);

        proc->procid = pid;
        proc->pwrite = wpIWRITEI;
        proc->pread  = rpIREADI;
        return(pid);
}


KillProcess(proc)
process proc;
{
        kill(proc->procid, SIGTERM);
        close(proc->pread);
        close(proc->pwrite);
}
```

TÂCHE : VARIABLES D'INSTANCE ET DE CLASSE.

```
struct task_desc
{
        int     procid;
        byte    pread;
        byte    pwrite;
        window  win;
        int     (*byte_handler)();
};

typedef struct task_desc *task;

#define max_task        16

extern task task_list[max_task];
extern int nbr_task;

extern task actv_task;
extern task curnt_task;

#define actv_proc       actv_task
#define actv_win        (actv_task->win)
#define setbytehandler(h)       actv_task->byte_handler = (h)
```

CRÉATION D'INSTANCE.

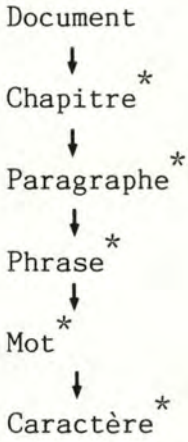
```
#define max_task        16

task task_list[max_task];
int nbr_task;


task NewTask(cmd, win, handler)
string cmd;
window win;
int (* handler, ();
{
        task t;

        if (nbr_task >= max_task)
           then Error("nombre maximum de processus");
        if ((t = task_list[nbr_task]) == nil)
           then {
                t = (task) malloc(sizeof (*t));
                if (t == nil)
                   then Error("plus de place, processus");
                }
        if (StartProcess(t, cmd) < 0)
           then Error("le processus ne demarre pas");
        t->win = win;
        t->byte_handler = handler;

        task_list[nbr_task++] = t;
        return t;
}


FreeTask(t)
task t;
{       /* pas encore implemente */
}
```

```
Kbdwait(delay)
unsigned delay;
{
        char c;
        alarm(delay);                       /* les signaux sont traites ailleurs
        if ((read(0, &c, 1)) == 1)
           then {
                ioctl(0, TIOCSTI, &c);  /* ce code n'est pas portable */
                }
}


scanTasks()
{
        int i;
        extern UpdateScreen();

        while (true)    /* on n'en sort jamais !? */
        {
            for (i=0; i < nbr_task; ++i)
                ProcessTask(task_list[i]);
            UpdateScreen();
            Kbdwait(0);
        }
}


task activ_task;

ProcessTask(t)
task t;
{
        int n_reads; byte b; long byte_count;

        activ_task = t;
        kbready(activ_proc, &byte_count);
        while (byte_count-- > 0)
        {
                n_reads = readbytes(activ_proc, &b, 1);
                if (n_reads != 1) then break;
                (*activ_task->byte_handler) (b);
        }
        if (byte_count >= 0)
           then UpdtFlag(activ_task->win, modified);
}


task curnt_task;

InitTask()
{
        extern virt_keyboard();

        task t = (task) malloc (sizeof(*t));
        t->pread = 0;                   /* le clavier */
        t->byte_handler = virt_keyboard;

        t->procid = -1;
        t->portid = -1;
        t->win = nil;
        task_list[0] = curnt_task = t;
        nbr_task = 1;
```

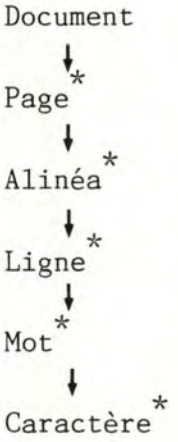
ANNEXE D : CONFLIT DE STRUCTURE ENTRE RI ET RV.

1º) Cas d'un document en langue naturelle.

Structure interne (RI)



Structure externe (RV)



Il est évident qu'on ne peut pas, en général, faire correspondre ces deux structures. Un chapitre, par exemple, peut tenir entièrement sur une page, être à cheval sur deux pages ou prendre un grand nombre de pages.
Bien que la forme visuelle du document doir refléter sa structure interne, cette forme visuelle est également dépendante d'autres contraintes de représentation comme la taille des pages,...etc.
Le fait que ces deux structures peuvent être mises en correspondance au niveau du caractère entraîne que le problème de la construction de la RV à partir de la RI est habituellement abordé au niveau du caractère également. Considérer la structure interne et la structure externe d'un document comme des suites de caractères est raisonner à un niveau d'abstraction trop bas.

2º) Cas d'un programme (en langage formel).