MASTER EN SCIENCES INFORMATIQUES

Aspects de la Programmation Visuelle

Jansen, Frank

Award date:
1991

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix de Namur
Institut d'Informatique

Aspects de la ProgrammationVisuelle

Jansen Frank

Promoteur : Fr. Bodart.

Mémoire présenté en vue de l'obtention du diplôme de licencié et maître en Informatique.

Année Académique 1990-1991.

ABSTRACT

Visual programming solves the problem of bringing computing facilities to people who do not have any extensive computer training by using visual representations in the programming process. The first part of this document describes a taxonomy of visual programming. This classification is issued from a complete synthesis of literature. In the second part we will analyse the Labyrinth System in detail in order to present the new tendances in the domain. This system is a general purpose, open core for building graphical and CAD tools. We will end with a little case study to show visual programming techniques can be a support to the specification of a quantitative decision support problem.

En utilisant des représentations visuelles dans le processus de programmation, la programmation visuelle apporte les bienfaits de l'informatique aux personnes qui n'ont pas eu de formation dans ce domaine. La première partie de ce document décrit une taxonomie de la programmation visuelle. Cette classification est née d'une synthèse complète de la littérature. Dans la seconde partie, nous analyserons en détail le système Labyrinth afin de présenter les nouvelles tendances qui apparaissent dans le domaine. Ce système est un logiciel graphique qui peut en construire d'autres du même type. Nous terminerons par une petite étude de cas pour montrer que des techniques de programmation visuelle peuvent être un support à la spécification d'un problème d'aide à la décision quantitatif.

Keywords : visual programming, Quantitative Decision Support Systems, reusability, application development scenario, frameworks, interfaces, visual scripting.

ITHAQUE

Quand tu partiras pour Ithaque, souhaite que le chemin soit long, riche en péripéties et en expérience. Ne crains ni les Lestrygons, ni les Cyclopes, ni la colère de Neptune. Tu ne verras rien de pareil sur ta route si tes pensées restent hautes, si ton corps et ton âme ne se laissent effleurer que par des émotions sans bassesse. Tu ne rencontreras ni les Lestrygons, ni les Cyclopes, ni le farouche Neptune, si tu ne les portes pas en toi-même, si ton coeur ne les dresse pas devant toi.

Souhaite que le chemin soit long, que nombreux soient les matins d'été, où (avec quelles délices !) tu pénétreras dans des ports vus pour la première fois. Fais escale à des comptoirs phéniciens, et acquiers de belles marchandises : nacre et corail, ambre et ébène, et mille sortes d'entêtants parfums. Acquiers le plus possible de ces entêtants parfums. Visite de nombreuses cités égyptiennes, et instruis-toi avidement auprès de leurs sages.

Garde sans cesse Ithaque présente à ton esprit. Ton but final est d'y parvenir, mais n'écourte pas ton voyage : mieux vaut qu'il dure de longues années, et que tu abordes enfin dans ton île aux jours de ta vieillesse, riche de tout ce que tu as gagné en chemin, sans attendre qu'Ithaque t'enrichisse.

Ithaque t'a donné le beau voyage : sans elle, tu ne te serais pas mis en route. Elle n'a plus rien d'autre à te donner.

Même si tu la trouves pauvre, Ithaque ne t'a pas trompé. Sage comme tu l'es devenu à la suite de tant d'expériences, tu as enfin compris ce que signifient les Ithaques.

C. Kavafis

Traduction de Marguerite Yourcenar et Constantin Dimaras.

A la mémoire de deux Professeurs lâchement assassinés le 27 novembre 1990 à Héraklion, Grèce.

Remerciements

Tout d'abord, je tiens à remercier M. François Bodart sans qui ce travail n'aurait jamais pu être accompli. Il en a été l'instigateur et m'a formé pendant près de trois ans pour venir à bout de cette tâche. Que ce labeur puisse être un signe de ma reconnaisance.

Je n'oublie pas Karine Becker, Alain Roginster et Jean-Marie Leheureux qui ont bien voulu me consacrer une partie de leur précieux temps afin de m'aider tout au long de cette aventure.

Celle-ci ayant commencé en Grèce, il me reste à exprimer toute ma reconnaissance au Professeur Yannis Vassiliou, directeur de la Foundation of Research and Technology - Hellas, pour l'accueil qu'il m'a offert dans son centre de recherche.

Je dois encore témoigner toute ma gratitude au Professeur Manolis Katevenis ainsi qu'à ses assistants Christos Georgis et Panos Kalogerakis, pour l'aide qu'ils m'ont apportée tout au long de ma participation au projet Laby.

Table des matières

Introduction

Permettre à des personnes qui n'ont pas eu de formation en informatique de profiter pleinement de la puissance des ordinateurs, tel est le difficile challenge de cette décade. La programmation visuelle représente une approche révolutionnaire pour relever ce défi. Nous allons via ce travail de fin d'études parcourir les aspects importants de la programmation visuelle. Pour ce faire, le travail a été divisé en 3 parties :

Le premier chapitre est le fruit d'une synthèse très complète de toute la littérature éditée sur les différents projets liés à la programmation visuelle. Afin de présenter le résultat de cette enquête sous la meilleure forme possible, nous utiliserons une structure qui situera chaque technique de programmation visuelle. Le but de cette taxonomie est de favoriser notre compréhension en nous concentrant uniquement sur les distinctions fonctionnelles entre chaque technique.

Le second chapitre vous présentera une analyse précise d'un logiciel graphique en cours de développement. Le Système Labyrinth est un logiciel graphique qui permet d'en construire d'autres du même type. Cet outil d'implémentation générique va nous enseigner les nouveaux concepts de la programmation visuelle tels que la technique du scripte visuel. La compréhension de ces techniques récentes sera renforcée par la présentation d'une petite démonstration réalisée à la fin de mon stage de fin d'études.

Le troisième et dernier chapitre consiste en une étude de cas qui décrit l'utilisation du scripte visuel comme support à la spécification d'un Système d'Aide à la Décision quantitative. Nous utiliserons cette nouvelle technique comme support à la spécification d'un problème d'investissement. Les résultats de cette étude ont donné naissance à une simulation qui sera exécutée sur Apple Macintosh à l'aide du logiciel Hypercard. Elle décrit une conversation type entre un Système d'Aide à la Décision quantitative et un utilisateur non expérimenté.

J'ai donc commencé mon travail de fin d'études en récoltant un maximum d'informations sur les différents projets réalisés dans la dernière décénie. Puis, je me suis tourné vers le futur en allant travailler sur le projet Labyrinth qui reprend les nouveaux concepts de la programmation visuelle. Après ces quelques expériences, j'ai essayé d'appliquer ce que j'ai appris dans un domaine assez précis. J'ai donc réalisé un dialogue type représentant bien les apports des techniques visuelles lors de la spécification d'un Système d'Aide à la Décision quantitative.

Une bibliographie complète reprenant la plupart des articles et ouvrages édités dans le domaine de la programmation visuelle vous sera présentée à la fin du document. Vous pourrez y puiser toutes les informations complémentaires qui n'apparaîtraient éventuellement pas dans le contenu de ce travail.

L'ensemble des personnes qui ont participé à cette grande aventure vous souhaite un agréable voyage dans le monde fabuleux de la programmation visuelle.

Dans ce premier chapitre, je vais brièvement décrire chaque tendance présentée dans la structure ci-dessus. Ainsi, on aura une idée assez précise sur l'état de l'art à l'heure actuelle.

Pour des raisons pratiques, vous retrouverez le schéma de la structure à côté de chaque explication d'une des catégories de celle-ci. La case qui sera marquée par des pointillés représentera la catégorie que l'on va décrire.

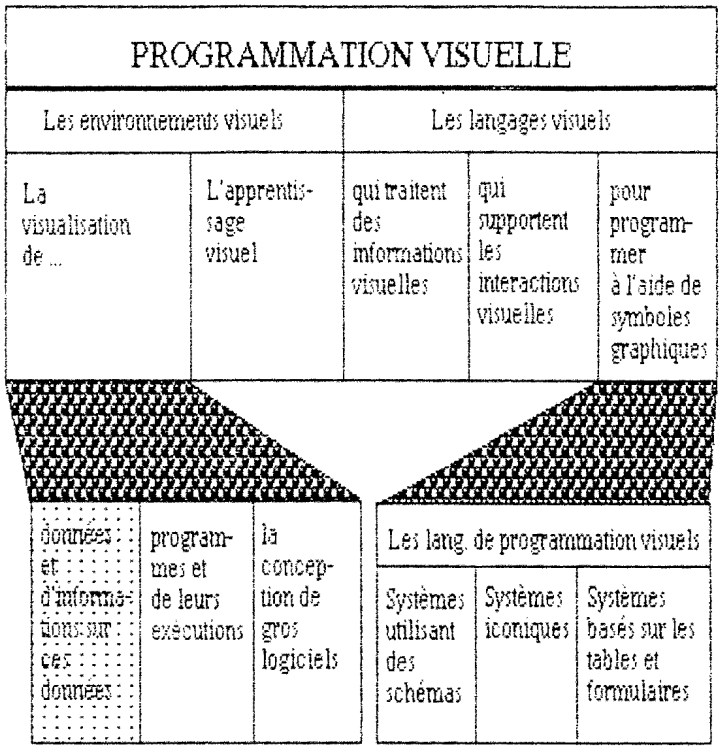


Schéma N° 1-3. : La visualisation de données et d'informations sur ces données dans la structure de la programmation visuelle.

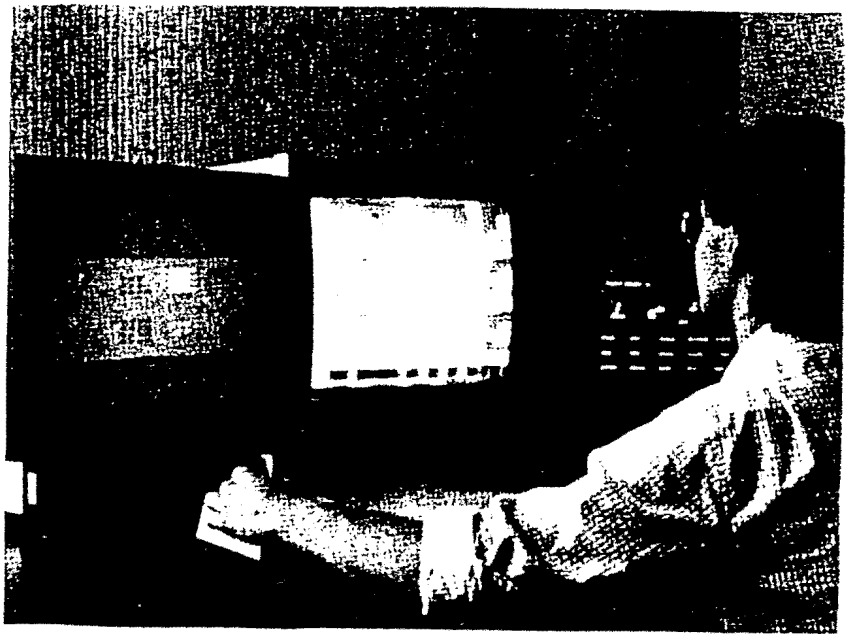


Schéma N° 1-4. : Ce S.G.S.D. [Her80] gère la base de données des bateaux d'une société maritime. L'utilisateur "navigue" parmi les représentations des bateaux (écran de gauche) et "zoome" pour trouver les informations textuelles qui l'intéressent (écran central).

I.2. LES ENVIRONNEMENTS VISUELS

I.2.1. La visualisation de données et d'informations sur ces données

I.2.1.1. La visualisation de données

Un Système de Gestion Spatiale de Données (S.G.S.D.) [Her80] est un système qui utilise une technique d'accès aux données via leur représentation graphique. Elle est née d'un besoin important de communication à l'intention des personnes qui veulent accéder au contenu d'un Système de Gestion de Base de Données (S.G.B.D.), mais qui ne sont pas formées au maniement de tels systèmes. Au contraire des S.G.B.D. traditionnels où les utilisateurs recherchent des informations en posant des questions dans un langage de requête formel, le S.G.S.D. présente graphiquement les données de la base de données dans une structure spatiale. De cette manière, l'utilisateur peut voyager dans l'espace des données graphiques, il peut aussi agrandir ou rétrécir la représentation spatiale pour faire ressortir plus ou moins de détails (schéma N° 1.4.). Cette présentation permet ainsi aux utilisateurs de trouver l'information dont ils ont besoin sans devoir la spécifier précisément ou savoir exactement où elle est stockée dans la base de données.

Les S.G.S.D. ont été développés à l'intention de personnes qui sont des utilisateurs novices ou occasionels, avec peu ou pas d'expérience en programmation. Habituellement, ces utilisateurs emploient le système avec un objectif spécifique en tête. Ils n'ont aucune compétence dans le langage de requête fourni par le S.G.B.D., et ils n'ont aucun intérêt à connaître la manière dont les données sont structurées et stockées dans le S.G.B.D. . Pour ce groupe d'utilisateurs, les facilités de recherche fournies par le système reposent fortement sur la manipulation directe et la visualisation des données [Mors79] [Schn83].

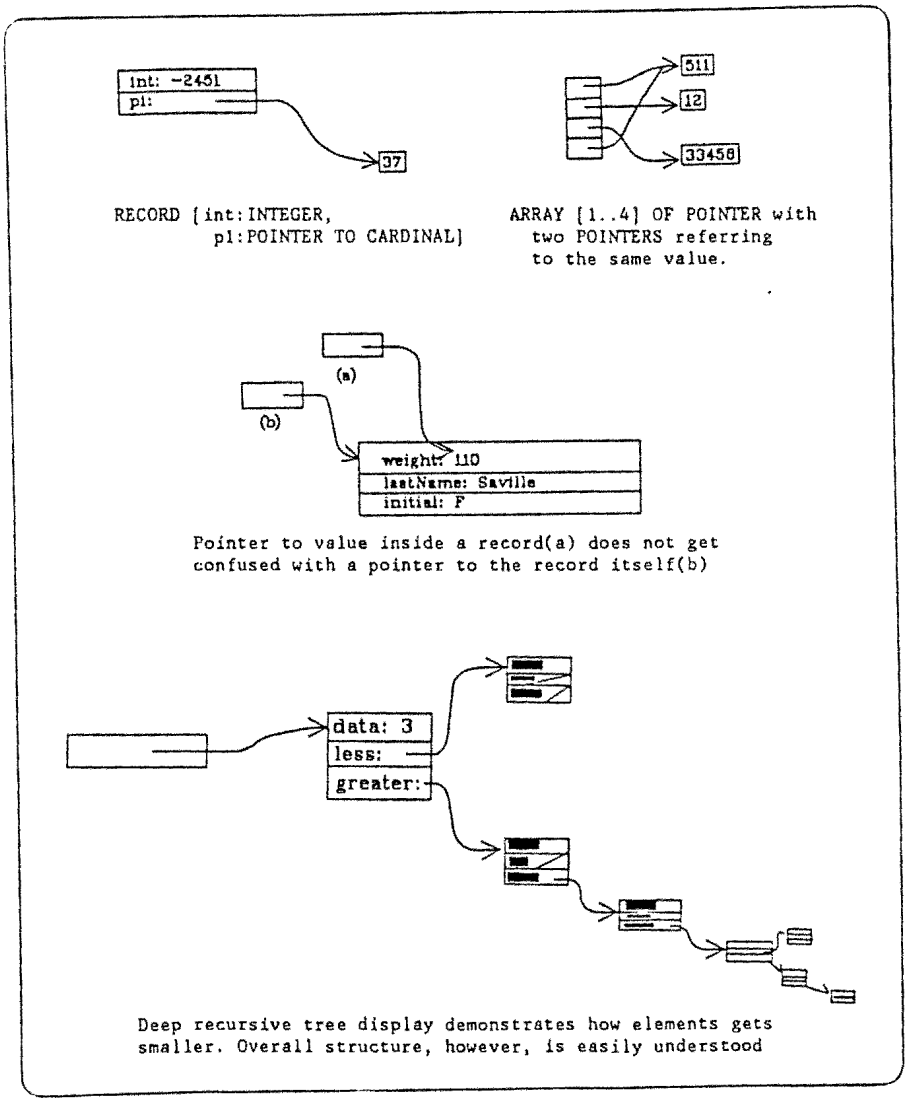


Schéma N° 1- 5. : Un exemple de présentation de structures de données [Mye83].

Cependant, le succès d'un tel système est limité à un petit nombre d'applications utilisant des bases de données de tailles relativement restreintes. Ces limitations sont principalement dues aux ressources importantes requises pour générer et stocker les environnements graphiques car les coûts de présentations graphiques d'une base de données (en espace disque et en surface écran) sont extrêmement importants.

I.2.1.2. La visualisation des structures de données

D'autres systèmes ont été développés mais ceux-ci l'ont été à l'intention d'un autre type d'utilisateurs qui a des objectifs fort différents. Ici, l'utilisateur est un programmeur expérimenté qui doit gérer toute la complexité de la programmation. Le but de ces nouveaux outils est de fournir un environnement de programmation qui favorise la compréhension des programmes en utilisant la visualisation des structures de données [Mye83] [Böc86] . Ils s'adressent donc à des programmeurs chevronnés qui affichent et explorent des structures de données souvent difficiles à représenter symboliquement. Ces représentations graphiques aideront aussi les programmeurs novices à mieux comprendre certains aspects d'un langage de programmation.

Puisque les structures de données jouent un rôle prédominant en programmation, et comme il existe beaucoup de représentations graphiques pour ces structures de données, il est normal que la visualisation de ces structures soient utilisées. D'ailleurs, lorsqu'on observe le contenu du bloc-notes d'un programmeur, on peut voir que celui-ci (le bloc-notes) est rempli de dessins représentant la structure de son problème. Le principe de base des systèmes qui nous préoccupe actuellement est de créer automatiquement des affichages similaires à ceux que le programmeur a l'habitude d'esquisser sur une feuille de papier. La solution habituelle étant que la représentation graphique de la structure des données est générée automatiquement à partir de la section déclarative du code source des programmes. Ensuite, le programmeur peut éventuellement modifier cette représentation graphique à l'aide d'un dispositif de pointage.

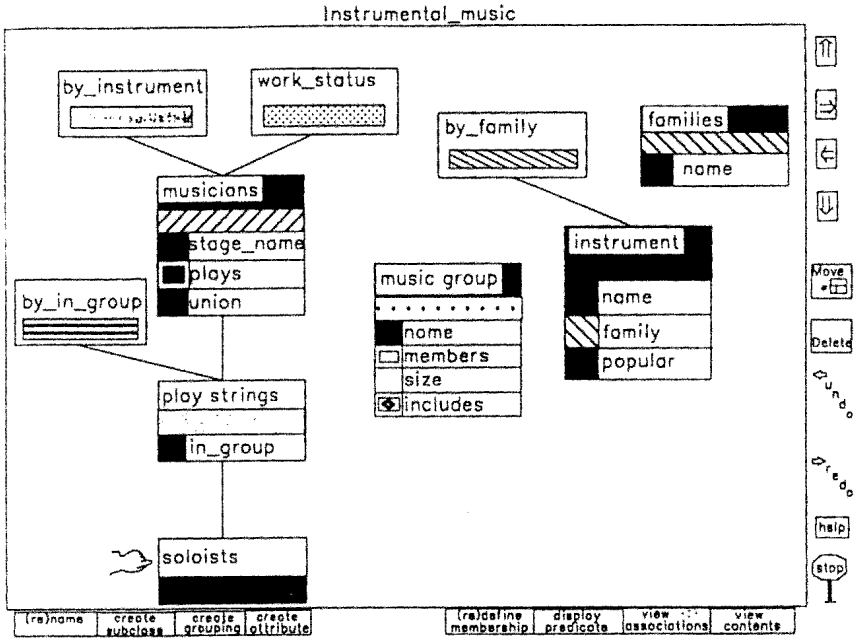


Schéma N° 1-6. : Exemple de schéma de base de données construit interactivement avec le système ISIS de [Gol85]. Grâce à un autre type de représentation de ce schéma, on pourra ensuite effectuer des requêtes graphiques.

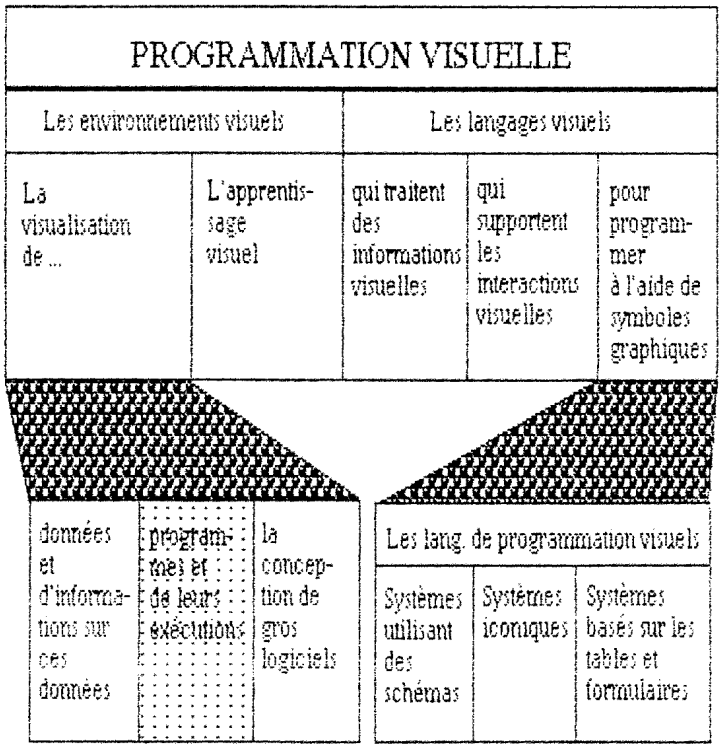


Schéma N° 1-7. : La visualisation de programmes et de leurs exécutions dans la structure de la programmation visuelle.

I.2.1.3. La visualisation des schémas de base de données

Après la visualisation des instances de données (I.2.1.1.) et des structures de celles-ci (I.2.1.2.), l'étape suivante consiste à visualiser les schémas de bases de données.

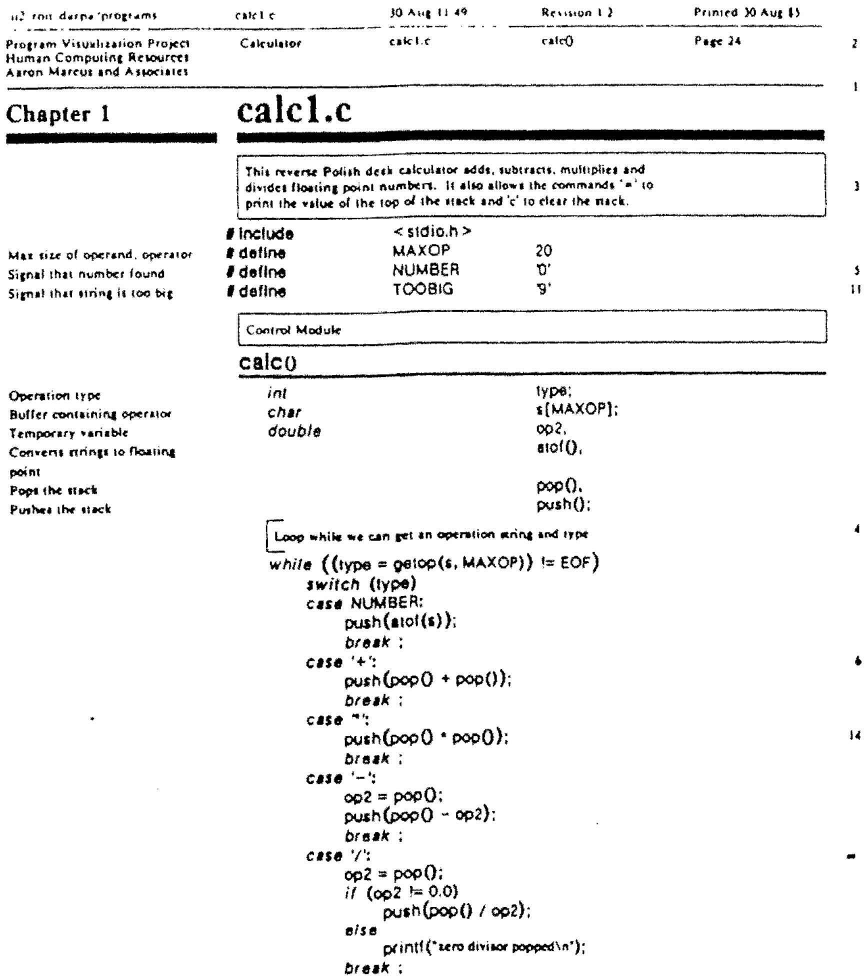
Ces deux dernières décades, des progrès impressionnants ont été réalisés en fournissant aux utilisateurs de gros volumes de données auxquels on peut accéder rapidement et assez facilement. Cependant, on a fait moins de progrès dans la technique qui fournit aux utilisateurs un accès aisé aux schémas des bases de données. Un environnemment graphique interactif permettrait par exemple de construire des schémas (schéma N° 1-6.), de devenir familié avec l'organisation d'une base de données en naviguant graphiquement à travers celle-ci, et de formuler des requêtes visuelles qui peuvent être stockées en tant que partie du schéma et réutilisées plus tard [Gol85]. Les représentations graphiques utilisées sont souvent des variantes du Modèle de Données Sémantique (M.D.S.) [Cze90] [Ham81] [Gys90] [Rous85].

I.2.2 La visualisation de programmes et de leurs exécutions

Le but, ici, est d'apporter aux programmeurs et aux utilisateurs une compréhension globale sur ce que fait un programme, comment il le fait, pourquoi il le fait et quels sont ses effets. C'est bénéfique non seulement pour les débutants qui apprennent à programmer, mais aussi pour les professionnels qui travaillent avec différents aspects de la programmation tels que la conception, les tests et les modifications.

I.2.2.1. Les "belles impressions"

Au départ, les "belles impressions" dénottaient une présentation améliorée du code source des programmes. La motivation primaire était de mettre en évidence la structure des programmes sous une forme vive pour que le programme soit plus lisible, plus compréhensible et génère ainsi moins



u2 ron darpa 'programs | calc1.c | 30 Aug 11 49 | Revision 1.2 | Printed 30 Aug 85

Program Visualization Project | Calculator | calc1.c | calc() | Page 24 2
Human Computing Resources
Aaron Marcus and Associates

1

Chapter 1

calc1.c

This reverse Polish desk calculator adds, subtracts, multiplies and divides floating point numbers. It also allows the commands '=' to print the value of the top of the stack and 'c' to clear the stack. 3

```
                                      #include   <stdio.h>
Max size of operand, operator         #define    MAXOP      20
Signal that number found              #define    NUMBER     '0'        5
Signal that string is too big         #define    TOOBIG     '9'        11
```

Control Module

calc()

```
Operation type                    int        type;
Buffer containing operator        char       s[MAXOP];
Temporary variable                double     op2,
Converts strings to floating                 atof(),
point
Pops the stack                               pop(),
Pushes the stack                             push();

    Loop while we can get an operation string and type                   4
    while ((type = getop(s, MAXOP)) != EOF)
        switch (type)
        case NUMBER:
            push(atof(s));
            break ;
        case '+':                                                        6
            push(pop() + pop());
            break ;
        case '*':
            push(pop() * pop());                                         14
            break ;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break ;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("zero divisor popped\n");
            break ;
```

Schéma N° 1-8. : Petit exemple de listing bien présenté [Bae86].

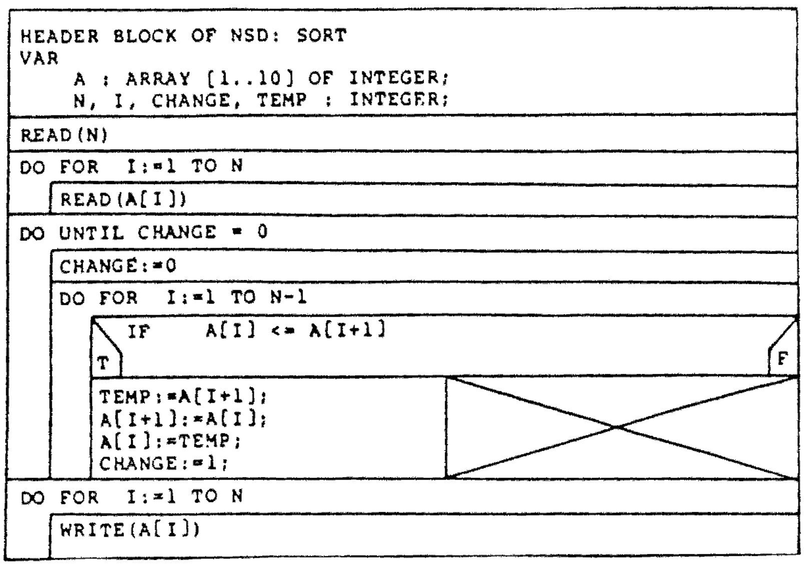


Schéma N° 1-9. : Ce graphe GNS étendu peut être assez facilement réalisé à partir d'un code source PASCAL [Pon83].

d'erreurs [Mia83]. Au début, les outils graphiques étant plutôt limités, le travail sur les "belles impressions" se concentrait principalement sur les différents niveaux d'indentation des programmes, sur les commentaires bien placés, et sur l'insertion de lignes blanches [Opp80] [Mia83].

La nouvelle approche se caractérise par l'emploi des principes de conception graphique concurremment avec les facilités d'affichages pour produire des représentations plus riches des textes de programmes (schéma N° 1-8.) [Bae86]. Cela a été rendu possible grâce à des fontes multiples, des tailles de caractères variables, des largeurs de caractères variables, des espaces proportionnels entre les caractères, des symboles non alphanumériques, des teintes de gris différentes, des normes et des localisations spatiales arbitraires d'éléments sur une page.

I.2.2.2. La visualisation à l'aide de schémas

Les organigrammes et autres schémas ont depuis longtemps fourni un moyen statique pour décrire les flux de contrôle à travers un programme. Cependant, jusqu'au récent déclin des coûts en informatique, la consommation élevée de ressources pour mémoriser une représentation graphique d'un programme garde ces organigrammes et autres schémas sur papier.

Les avantages engendrés par la présentation des programmes sous ces diverses formes sont néanmoins reconnus [Fit79]. Donc, lorsque les coûts des calculs et de mémorisation des graphiques ont chuté, les efforts pour traiter les programmes sous une forme graphique prirent de plus en plus d'importance [Pag83]. La plupart des systèmes puisent dans le code source des programmes les informations nécessaires pour créer une représentation graphique de celui-ci. Les représentations habituelles sont des graphes GNS (schéma N° 1-9.), des organigrammes classiques ou tout autre schéma "fait maison". Il existe aussi des éditeurs syntaxiques [Lev86] qui guident le programmeur dans l'élaboration de son programme. Le langage de programmation n'a pas changé mais le programmeur construit son algorithme avec l'aide constante du système, celui-ci connaissant parfaitement les unités syntaxiques de ce langage.

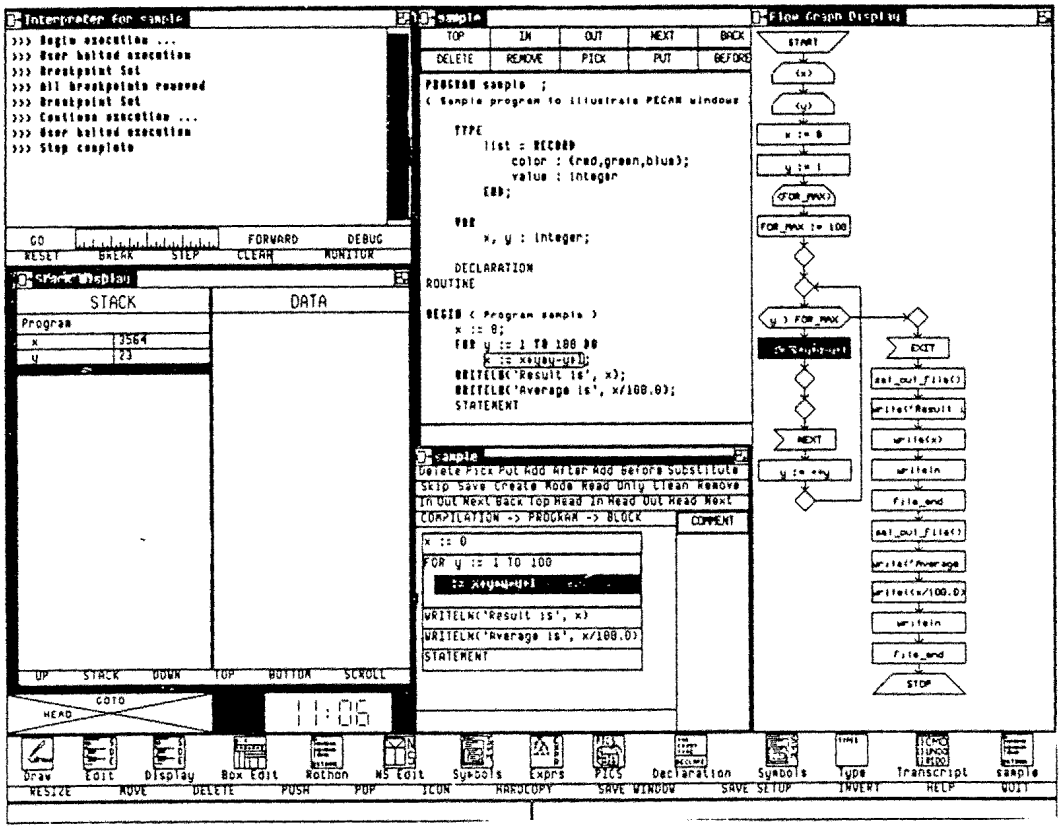


Schéma N° 1-10. : Les vues multiples de l'état du programme après exécution [Reis85]. Nous reconnaissons un organigramme dynamique, un éditeur de code syntaxique, les résultats de l'exécution partielle, un debugger, etc ... . Les icônes situées en bas de l'écran représentent les différents outils qui sont offerts.

I.2.2.3. Les vues multiples d'un programme et de ses états d'exécutions

Dans les deux sections précédentes, nous avons vu que les "belles impressions" augmentent la lisibilité des programmes et que la représentation de la structure d'un programme sous forme d'organigramme ou autres schémas en améliore la compréhension. Cependant, l'analyse, la conception, l'implémentation et la maintenance d'un programme impliquent des activités mentales basées non seulement sur l'apparence du programme mais aussi sur d'autres observations. Celles-ci peuvent être exprimées par une série de questions telles que "Comment le programme travaille-t-il ?", "Comment les composantes sont-elles connectées ensemble ?", "Quels effets ont-elles les unes sur les autres ?", etc... . En d'autres termes, pour assister le processus de programmation, un outil qui procure des vues multiples d'un programme et de ses états d'exécution serait plus efficace qu'un outil qui se concentre uniquement sur le code source d'un programme. On mettrait ainsi en évidence toute une série d'aspects dynamiques de la programmation.

Beaucoup d'environnements de programmation interactifs supportent des vues multiples [Reis85] [Reis86] [Reis87] [Teit81] [Teit85]. En fait, les systèmes à multi-fenêtrages très présents dans les environnements interactifs sont conçus précisément avec le concept de vues multiples à l'esprit [Car84]. Chaque fenêtre correspond à une vue particulière du programme ou de ses états d'exécutions (schéma N° 1-10.). On peut classer ces fenêtres en 3 grandes catégories :

- 1) Les vues concernant les données : où l'on retrouve des outils qui visualisent les données ou les informations sur ces données. Mais dans ce cas, les représentations graphiques sont plus dynamiques car elles varient en fonction de l'exécution et ne sont plus simplement basées sur le code source du programme.

- 2) Les vues concernant le programme : où l'on retrouve des outils qui visualisent les programmes et leurs états d'exécutions. On utilise différentes sortes de schémas pour décrire l'exécution des programmes et non plus seulement leur structure.

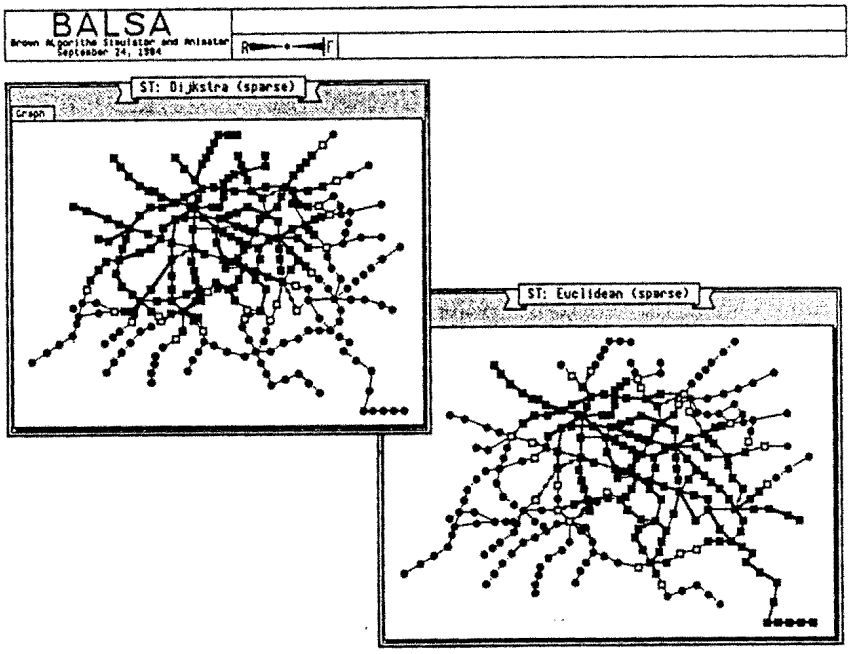


Schéma N° 1-11. : Cette simulation exécute en parallèle deux algorithmes différents de recherche du chemin le plus court sur un graphe représentant le plan du métro de Paris. [Bro85b]. Les cercles noirs sont les stations non encore "visitées", les carrés noirs sont les stations déjà visitées et les carrés blancs sont prêts à être visités.

- 3) Les vues concernant le contrôle : où les outils reprennent les fonctionnalités classiques d'un debugger très puissant (renverser l'exécution d'un programme, exécuter pas à pas, contrôler la vitesse d'exécution, insérer des points d'arrêts, etc ...).

Le programmeur peut alors parfaitement comprendre ce qui se passe lors de l'exécution de l'un de ses programmes en visualisant le contenu des différentes fenêtres, chacune d'entre elles renfermant les résultats d'un des outils de l'environnement.

I.2.2.4. Comprendre par animation d'algorithmes

Dans cette section, on se concentre sur les systèmes qui utilisent l'animation pour accroître la compréhension d'un programme [Böc85] [Bro84] [Bro85b] [Kra88] [Lon85]. Contrôlé par des éléments temporels, l'affichage dynamique de graphiques peut être utilisé pour communiquer beaucoup d'informations de manière plus compacte, plus vivante, et sous une forme plus rapidement assimilable. Il est normal que l'animation ait été adoptée par beaucoup comme moyen de communication (Pensons à toutes les histoires fabuleuses que nous avons littéralement vécues grâce aux dessins animés créés par Walt Disney.). La nouveauté réside dans l'application de l'animation à **l'étude** de la programmation.

Actuellement, les efforts majeurs dans ce domaine concernent deux groupes différents de personnes qui peuvent bénéficier de l'animation graphique.

Le premier groupe est formé d'enfants faisant leurs premiers pas en informatique. Tout le monde a en tête le langage LOGO et sa célebre tortue qui ont aidé de nombreux enfants à entrer dans le monde fabuleux des ordinateurs.

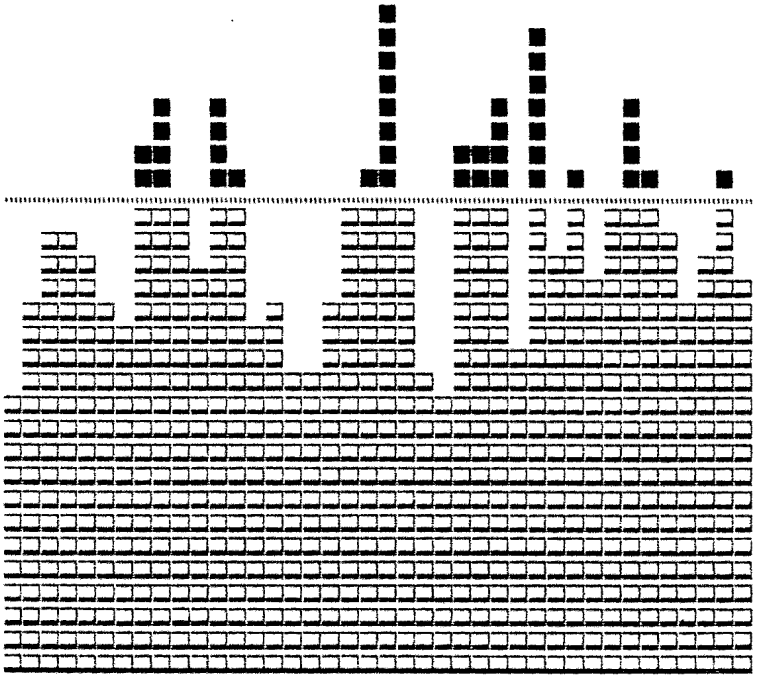


Schéma N° 1-12. : En fonction de la gestion de l'index choisi, du nombre de pages, du nombre d'articles par page et du taux de remplissage en zone primaire, on regarde les articles s'organiser dans l'index du fichier à accès direct.

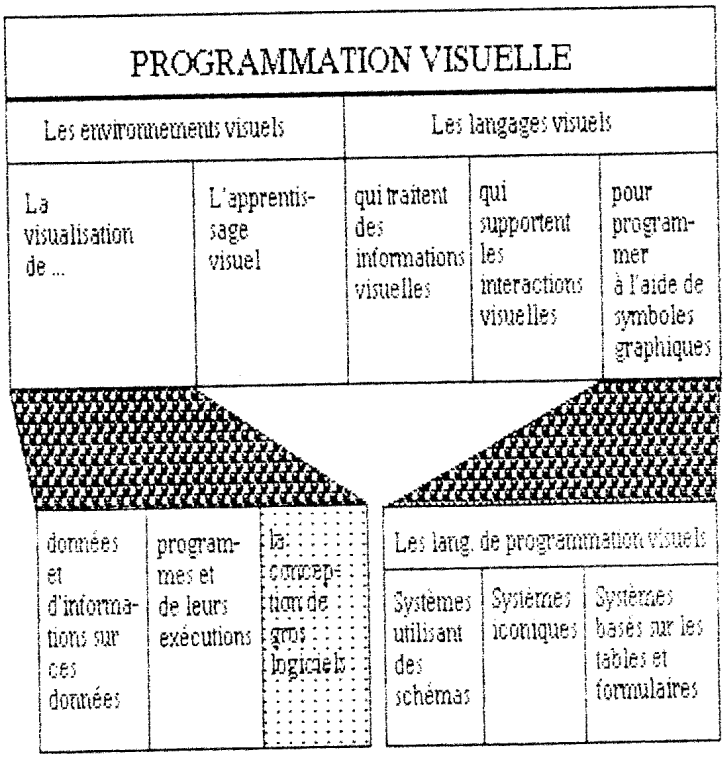


Schéma N° 1-13 : La visualisation de la conception de gros logiciels dans la structure de la programmation visuelle.

L'autre groupe est constitué d'étudiants en informatique et de chercheurs qui désirent une compréhension plus approfondie de la logique interne des algorithmes. Ceux-ci concernent généralement les mathématiques (schéma N° 1-11.) (Euclide, Dijkstra, génération de nombres pris au hasard, etc ...), les tris (tri bulle, Quicksort, tri par insertion, etc ...) et les recherches (séquentielles, arbres balancés, etc ...). D'ailleurs, tous les étudiants de première licence en informatique aux Facultés Universitaires Notre-Dame de la Paix de Namur ont un jour ou l'autre simulé graphiquement différentes gestions d'index concernant des fichiers supportant l'accès direct (rangement calculé ou séquentiel indexé) (schéma N° 1-12.). Ce simulateur partiellement graphique a été élaboré par une équipe conduite par le Professeur J.L. Hainaut dans le cadre de recherches sur les technologies de bases de données.

I.2.3. La visualisation de la conception de gros logiciels

> "*La conception d'un logiciel de grande taille est fort différente de celle d'un simple algorithme : l'interface externe, c'est-à-dire les spécifications, est définie moins précisément, est plus complexe, et est plus sujette aux changements. Le logiciel important a une structure interne plus forte, donc beaucoup d'interfaces internes, et la mesure de son succès est bien moins claire.*"
>
> Butler Lampson [Lam84]

Une compréhension des spécifications, des décisions de conception, des structures du système, des dépendances entre les données et les composantes, etc ..., est cruciale à travers tout le cycle de vie du logiciel. Il devient de plus en plus difficile de maîtriser tous ces éléments lorsque le logiciel augmente en taille et en complexité. Cette observation, couplée avec le succès des techniques graphiques pour la visualisation de programmes à un niveau très bas ("programming in the small"), a stimulé les efforts de la visualisation de la programmation à un niveau plus élevé ("programming in the large") [Bro85a] [Mori85].

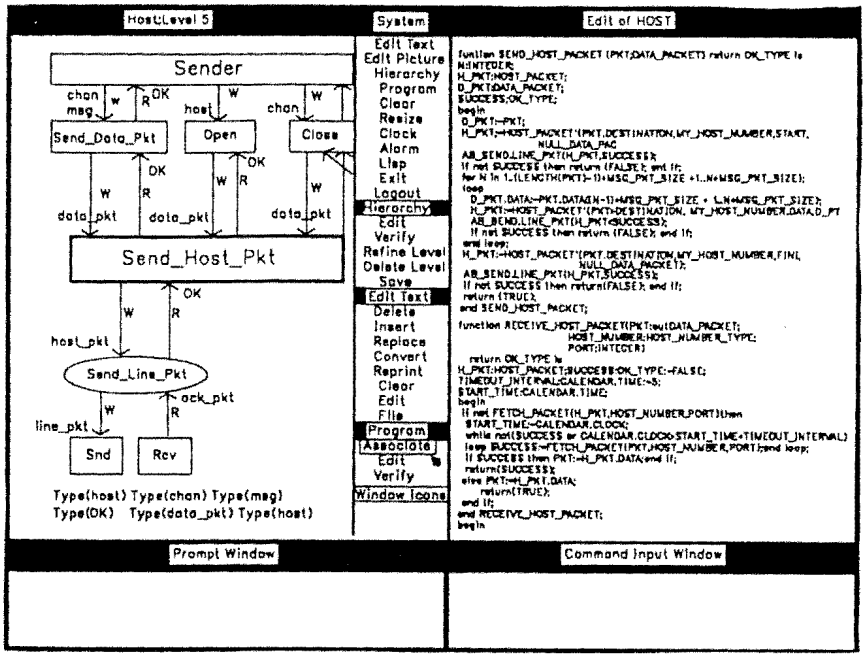


Schéma N° 1-14. : L'utilisateur a choisi dans le dernier niveau de la hiérarchie l'entité "Send-Host-Pkt" et dans le code du programme la procédure Ada s'y rapportant. Par la commande "Associate", le système PegaSys enregistre cette association. Il vérifiera ensuite si la spécification visuelle est logiquement consistante avec la procédure.

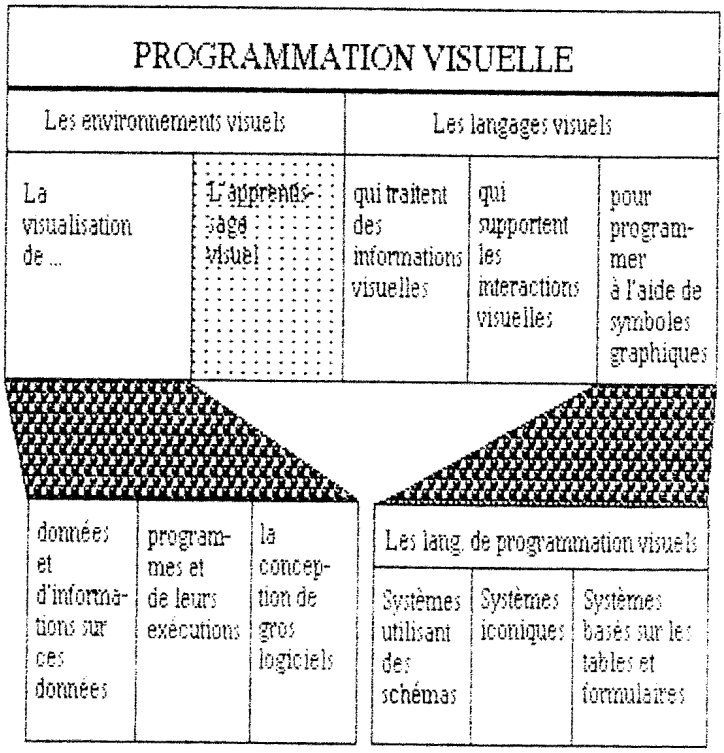


Schéma N° 1-15. : L'apprentissage visuel dans la structure de la programmation visuelle.

Le but principal est de fournir un environnement visuel intégré qui supporte l'entièreté du cycle de vie d'un logiciel complexe et de grande taille. Les efforts dans ce domaine sont toujours en cours. On essaye de fournir un environnement de développement de logiciel dans lequel les spécifications, les décisions de conception, et les structures du système sont toutes saisies sous forme graphique par les personnes qui conçoivent, utilisent et maintiennent un logiciel de taille importante. L'accent est alors mis sur l'intégration de toute une série d'outils graphiques (reprenant la plupart des fonctionnalités présentées jusqu'à présent) pour former un environnement visuel intégré supportant le cycle de vie complet d'un programme de grande taille.

Le schéma N° 1-14. représente un outil graphique qui vérifie si la spécification visuelle d'un programme de gestion de réseaux est logiquement consistante avec le code source de ce programme. .

I.2.4. "L'apprentissage visuel"

> " *Dans le cas le plus idéal, le programmeur agit comme un professeur et ne donne que quelques exemples à l'ordinateur; celui-ci, comme un élève intelligent, induit l'abstraction qui couvre tous les exemples."*
>
> Brad Myers [Mye86b]

I.2.4.1. Contrastes entre les environnements visuels et les langages visuels.

Dans les 3 dernières sections (I.2.1., I.2.2., I.2.3.), nous avons décrit des systèmes conçus pour fournir un environnement visualisant des données, des programmes ainsi que de leurs états d'exécution, des structures d'un système, ou une combinaison de ceux-ci. Leur but principal est de fournir des aides visuelles dans la programmation conventionnelle. Du point de vue utilisateur, les environnements visuels peuvent être caractérisés par le principe "Montre-moi ce que j'ai !" (en terme de programme, de données ou de conception de systèmes) ou "Montre-moi ce qui se passe !" (en terme d'exécution).

Au contraire, "l'apprentissage visuel" a un autre objectif que l'on peut caractériser par "Fais ce que je te montre !". Les systèmes "d'apprentissages visuels" travaillent dans un environnement hautement visuel. Cependant, un système qui tombe dans cette catégorie va au-delà d'un environnement visuel et essaye actuellement de produire un programme en se basant sur ce qui lui a été montré.

Parce que l'objectif premier de "l'apprentissage visuel" est de construire des programmes, beaucoup de personnes [Cha87] ne font pas la distinction entre celui-ci et les langages de programmation visuels (I.3.). Afin de renforcer notre compréhension, il est important de noter certaines distinctions.

Avec les langages de programmation visuels (qu'ils aient des icônes, des formulaires ou toutes autres représentations graphiques comme unité syntaxique), les programmes sont construits à partir de ces unités visuelles du langage. Ces primitives graphiques (icônes, lignes, boîtes, flèches, etc ...) ont une syntaxe et une sémantique bien définies.

L'apprentissage visuel utilise une autre technique de construction de programme indépendante des langages. Habituellement, un utilisateur montre à l'ordinateur comment il résoud les calculs désirés en les décrivant sur quelques échantillons de données. Ensuite, le système généralise sur d'autres données le comportement montré pour l'application. "Ce style d'interaction mime donc la façon informelle que nous utilisons pour expliquer les programmes" [Rae84].

Bien que l'apprentissage visuel et les langages de programmation visuels ont la même fin, (produire un programme), les moyens mis en oeuvre pour atteindre cet objectif sont sensiblement différents. Tandis que les langages de programmation visuels sont utilisés pour apprendre à l'ordinateur à "Faire comme on lui dit", l'apprentissage visuel utilise l'approche "Fais comme on te montre !".

Exemples de paires "entrées / sorties"

- REVERSE [(x1 x2 x3 ... xn)] = (xn ... x1 x2 x3)

- CONCAT [(A B C), (D E)] = (A B C D E)
  CONCAT [(L M), (N O P)] = (L M N O P)

Exemples de traces de programmes

- SORT [(3 1 4 2)] --> ( )
  (1 4 2) --> (3)
  (4 2) --> (1 3)
  (2) --> (1 3 4)
  ( ) --> (1 2 3 4)

Schéma N° 1-16. : Deux types d'exemples de spécifications de programme par l'exemple.

I.2.4.2. La programmation par l'exemple : La clé du problème

Presque tous les systèmes d'apprentissages visuels sont inspirés de la technique de "programmation par l'exemple ou programmation par démonstration" [Hal84]. Un bref résumé de ce concept s'avère utile.

La majorité des individus sont plus à l'aise lorsqu'ils travaillent avec des objets concrets et spécifiques. Lorsqu'ils sont biens choisis et proprement utilisés, ils aident les gens à comprendre des concepts abstraits. Du point de vue utilisateur, l'idée fondamentale servant de base à la programmation par l'exemple / démonstration est vraiment très simple. L'utilisateur écrit un programme (ex : écrit les spécifications) en donnant des exemples à l'ordinateur sur ce que le programme doit faire. Le système enregistre, et heureusement aussi généralise ce qui lui a été montré. La spécification pourrait consister à donner des exemples de sorties désirées pour chaque entrée, ou des exemples sur la manière dont le programmme devrait traiter les entrées, ou une combinaison de ces deux méthodes [Bie76]. Dans le schéma N° 1-16., nous avons repris quelques exemples tirés de [Bar82].

I.2.4.3. L'émergence de l'apprentissage visuel

A cause de l'attrait naturel des exemples pour expliquer des concepts, la spécification par l'exemple a longtemps attiré l'attention des chercheurs de la communauté de programmation automatique. Cependant, du point de vue implémentation, ce type de programme induit des difficultés considérables. Plus précisément, les exemples doivent être choisis de telle sorte que l'on puisse spécifier sans ambiguïté le programme désiré.

Malheureusement, la spécification par l'exemple est rarement complète car quelques exemples ne décrivent pas complètement le comportement du programme désiré. Le système doit être capable de travailler avec des informations partielles ou fragmentées. Il doit aussi déterminer si les spécifications de l'utilisateur sont consistantes et si le modèle que l'utilisateur décrit est bien le bon programme. De plus, lorsque le traitement est expliqué dans les cas les plus généraux, le système doit énumérer l'ensemble de tous les programmes possibles. Le coût de l'énumération peut être très important, même avec les techniques actuelles de l'intelligence artificielle.

Ces difficultés peuvent être réduites en diminuant le domaine des applications visées. Lorsque celui-ci est relativement petit et précis, les chances de succès sont plus grandes. Les difficultés peuvent être encore plus réduites en fournissant un environnement graphique interactif. Les graphiques rendent les exemples concrets. L'interaction aide l'utilisateur à communiquer avec le système avec une plus grande facilité, ce qui pourrait conduire à un accroissement de la clareté et de la quantité d'information transmise au système. De plus, un environnement graphique interactif encourage la coopération entre l'utilisateur et le système et offre une construction plus dynamique des programmes. Un nouveau style de programmation, que nous appelons "apprentissage visuel" représente les différents essais qui couplent la puissance des exemples avec les bénéfices du travail dans un environnement visuel interactif.

Nous pouvons diviser les systèmes d'apprentissage visuel en deux sous-groupes :

- 1) Pour **le premier groupe**, bien que les interactions graphiques soient utilisées dans le processus de spécification, le centre d'intérêt est la capacité du système à faire des inférences inductives sur base de spécifications qui décrivent partiellement le comportement du programme. Nous trouvons surtout les systèmes qui intéressent l'ensemble des personnes faisant partie de la communauté de programmation automatique [Bie85].

- 2) Au contraire, les systèmes du **second groupe** se basent sur les interactions graphiques comme moyen de programmation [Fin84] [Smi82] [Rub85]. Ils limitent généralement leurs applications à un domaine relativement petit et précis, et n'essayent pas de faire des inférences inductives excessives. Dans l'état actuel de l'art, la plupart des systèmes d'apprentissage visuel appartiennent au second groupe.

Manifestement, il y a beaucoup de caractéristiques communes aux deux groupes. Ils utilisent tous les deux des exemples. L'utilisateur a un contact visuel direct avec des représentations significatives des objets et il peut les manipuler naturellement. Il travaille avec les effets du programme plutôt qu'avec le programme lui-même. Il n'est plus nécessaire d'étudier tous les aspects d'un langage de programmation pour commencer à programmer.

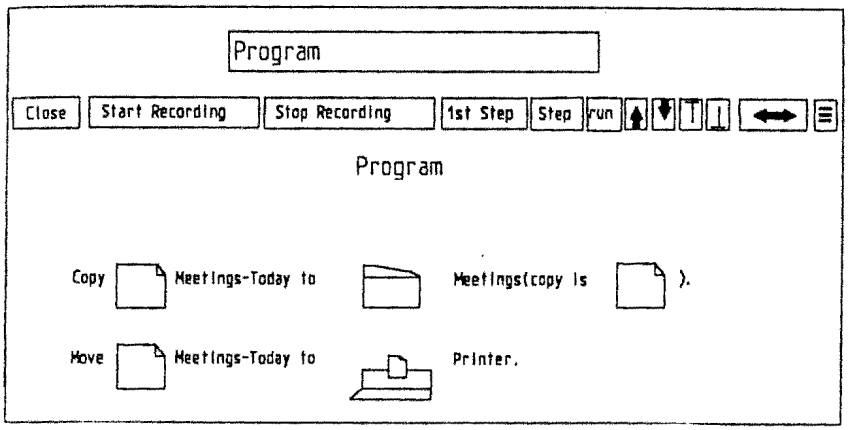


Schéma N° 1-17. : Un petit programme de bureautique spécifié par l'exemple [Smi82].

En même temps, ils ont tous les deux des restrictions. En plus des limitations physiques imposées par l'espace réduit d'un écran, leurs limitations ne sont pas de même nature.

Le premier groupe peut produire un programme à partir d'informations incomplètes, mais il demande que l'utilisateur connaisse la manière exacte dont il veut que les traitements soient effectués. En général, il n'est pas raisonnable d'assumer qu'un utilisateur (tout spécialement un non programmeur) connaisse suffisamment les aspects procéduraux de la programmation lorsqu'il veut faire des calculs plus compliqués.

Le second groupe, d'un autre côté, offre un environnement plus intuitif mais le domaine de problème est plus restreint. Ce style de système travaille mieux lorsque les objets du problème ont une représentation directe et évidente. Mais cela devient difficile lorsqu'on doit exprimer des idées dans des termes plus généraux et abstraits.

En dépit de ces limites, l'apprentissage visuel représente un système intéressant pour construire un programme. Dans le futur, il n'y a aucun doute que l'on assistera à une fusion des techniques utilisées par les deux écoles. Lorsque ces deux disciplines (l'intelligence artificielle et les interactions graphiques) seront mises ensemble, le fruit de leur union produira un système très efficace et convivial.

Pour illustrer cette catégorie, nous avons choisi deux petits exemples. Le premier est un petit programme créé sur le sytème Smallstar, une version prototype du système Star [Smi82]. Supposons qu'un cadre désire écrire un petit programme qui effectue une copie du document "Meetings-Today", qui sauve cette copie dans un classeur appelé "Meetings", et qui imprime le document copié. Il va décrire la séquence d'opérations suivante :

- Pointer sur l'opération "Start recording".
- Pointer sur l'icône de "Meetings-Today".
- Pointer sur l'opération "Copy".
- Pointer sur l'icône de "Meetings".
- Pointer sur l'icône "Meetings-Today".
- Pointer sur l'opération "Move".
- Pointer sur l'icône "imprimante".
- Pointer sur "Stop recording".

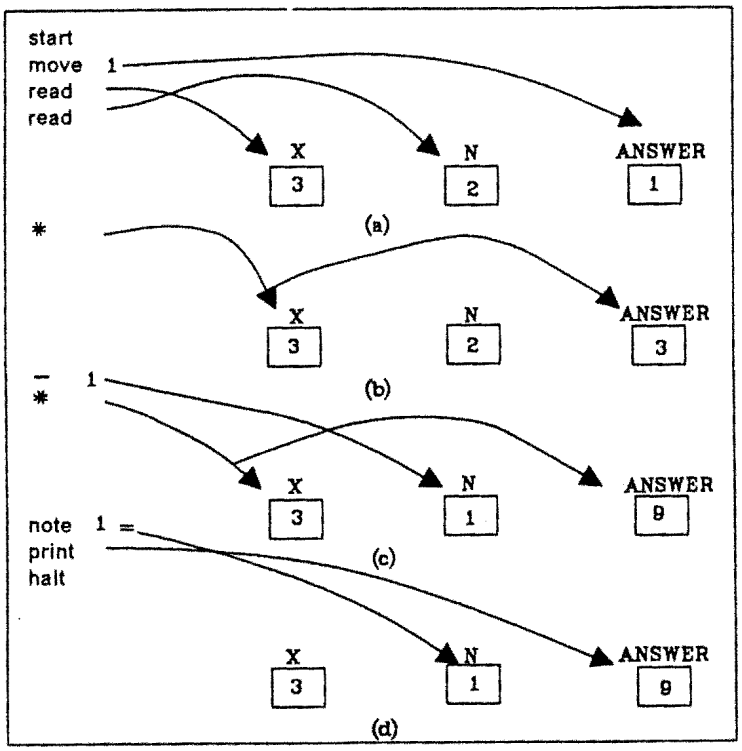


Schéma N° 1-18. : L'exemple qui décrit le programme calculant la puissance "32".**

(a) : L'état que l'on doit retrouver au début du programme.

(b) : Calcul de la première puissance de 3 (1*3).

(c) : Calcul de la seconde puissance de 3 (3 * 3).

(d) : Donner la condition d'arrêt et imprimer le résultat (N=1).

Le programme résultat nous est présenté dans le schéma N° 1-17. Si le cadre effectue ce style d'opération tous les jours, il peut rappeler ce programme via l'icône qui le représente.

Le second exemple est un système de programmation interactif qui construit automatiquement des programmes à partir d'exemples d'exécutions donnés par l'utilisateur. Si nous voulons créer un programme calculant la puissance "X**N", nous devons d'abord déclarer interactivement trois variables (X, N, Answer). Ensuite nous devons réaliser un petit exemple de calcul pour montrer au système le comportement que le programme doit avoir. Le schéma N° 1-18. montre le calcul de 3^2. Le système généralisera ensuite cet exemple donné à tous les cas possibles.

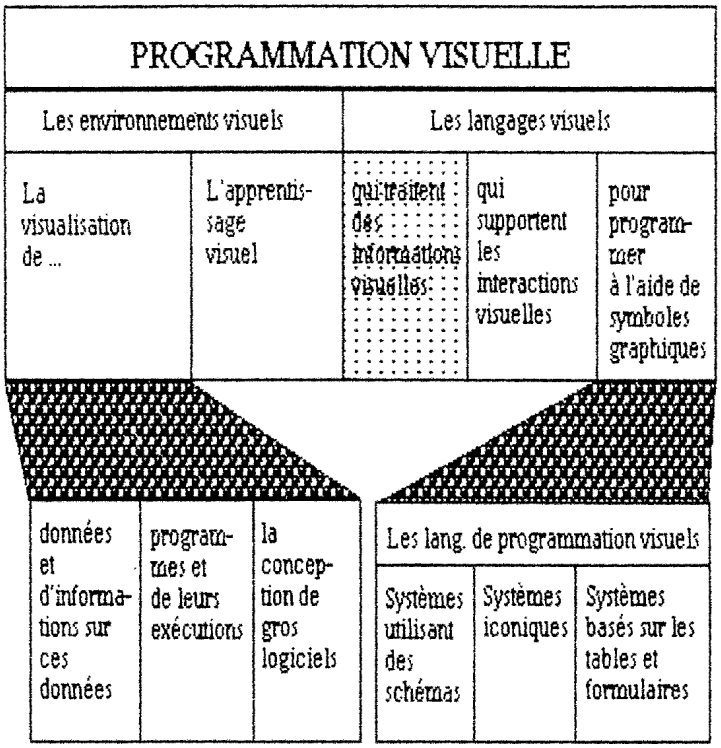


Schéma N° 1-19. : Les langages visuels qui traitent des informations visuelles dans la structure de la programmation visuelle.

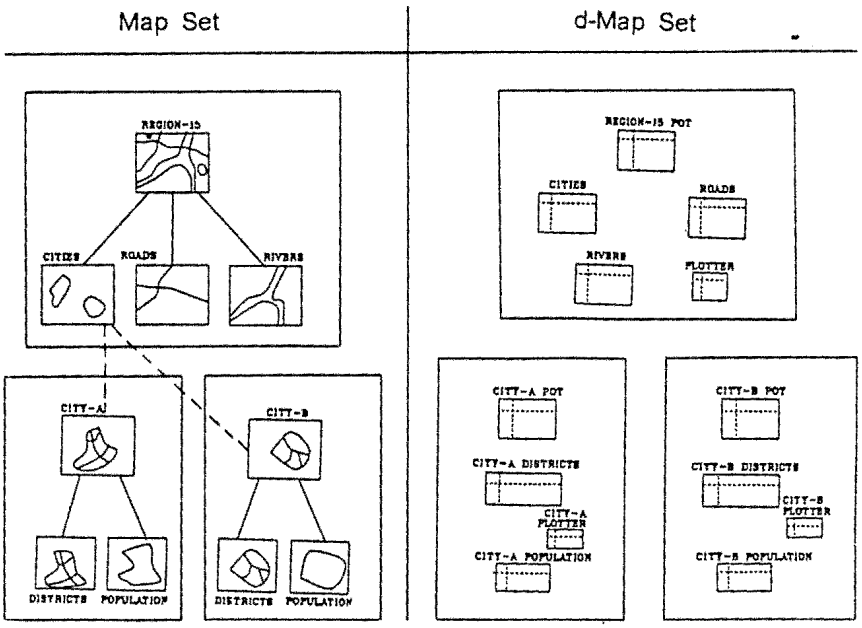


Schéma N° 1-20. : Exemple de représentation des images par des tables relationnelles pour ensuite mieux les traiter [Cha79].

I.3. LES LANGAGES VISUELS

Selon leurs spécificités, les langages visuels peuvent être classés en trois catégories : les langages de traitement des informations visuelles, les langages qui supportent les interactions et représentations visuelles, et les langages pour programmer à l'aide d'expressions visuelles.

I.3.1. Les langages de traitement des informations visuelles

Dans cette section, nous parlons des langages visuels de la première catégorie. Ce sont des langages conçus pour traiter des informations qui se présentent sous forme de dessins ou d'images.

Nous allons mettre en évidence les différences qui existent entre les langages qui traitent des informations visuelles et les langages de programmation visuels. Au début des années 70, les systèmes de traitement d'images et les systèmes de gestion de base de données (S.G.B.D.) étaient développés en parallèle dans deux camps différents.

Les systèmes de traitement d'images étaient principalement dédiés aux scientifiques, aux ingénieurs, aux géographes et aux médecins. Chaque système de traitement d'images à été façonné pour un but spécifique dans un environnement spécial, ce qui rend le partage d'images très difficile, voire impossible.

Les systèmes de gestion de base de données, d'un autre côté, ont été conçus pour des applications de gestion. Leur but principal était de permettre le partage des données tout en maintenant l'indépendance, l'intégrité, et la sécurité de celles-ci. La gestion des données sous forme d'images ne fait pas partie de ce domaine. Il était difficile, sinon impossible, d'utiliser les langages de requête conventionnels des bases de données pour exprimer ou manipuler des relations spatiales.

```
(a)  get  [all                    ]  ;  condition list;  into relation_name.
          [descriptor list        ]

(b)  [paint  ]  [picture                                    ]
     [sketch ]  [picture-relational file  ;  condition list;  into frame_name.
                   (elementary or
                    composite)            ]

(c)  show   frame_name.

(d)  load   database_name.

          (Note) Alternatives are shown in square brackets
```

Schéma N° 1-21. : Les structures typiques de requêtes sur des images [Cha79].

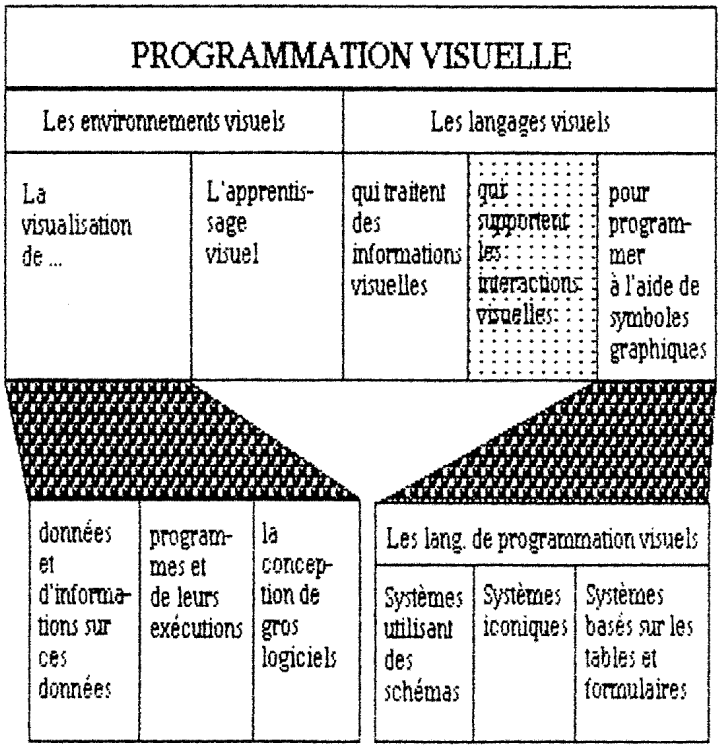


Schéma N° 1-22. : Les langages visuels qui supportent les intéractions visuelles dans la structure de la programmation visuelle.

A la fin des années 70, la technologie qui traite les images et celle qui gère les bases de données commencèrent à fusionner. La quantité croissante d'images devant être générée et analysée, l'extension de ce nouveau domaine d'application, et le besoin sans cesse grandissant de partager des données picturales ont ainsi motivé le développement de systèmes généralisés.

Profitant de la communicabilité des langages conventionnels des bases de données, une approche relativement simple consistait à introduire des capacités de traitement d'images dans ces anciens langages pour en créer des nouveaux. Par conséquent, un certain nombre de langages de requêtes pour des données d'images sont implémentés à partir de langages de requêtes conventionnels (schéma N° 1-20.). Ces langages sont souvent appelés langages de requête "picturaux". Un terme qui peut porter à confusion, car dans la plupart des cas, les langages de requêtes eux-mêmes ne sont pas picturaux (mais bien textuels) (schéma N° 1-21.), alors que les objets qu'ils gèrent le sont.

I.3.2. Les langages qui supportent les interactions et représentations visuelles

Grâce aux progrès récents de la technologie, les différentes catégories d'ordinateurs possèdent généralement un écran haute-résolution. Une conséquence naturelle de ce développement est la naissance d'applications d'un nouveau genre employant des interactions visuelles comme support de dialogue.

L' affichage d'objets graphiques jouent évidemment un rôle important dans les interactions visuelles. L'ancienne méthode pour afficher des objets graphiques quelconques consistait à écrire un programme (dans un langage de programmation traditionnel) qui accepte des entrées paramétrisées, examine une base de données et fait alors appel à des packages de sous-routines graphiques pour créer l'écran désiré. Cette approche nécessite des ressources importantes, l'utilisation de langages traditionnels, qui n'ont pas été conçus pour supporter des interactions visuelles, en est bien sûr la cause principale.

```
icon class cluster (r)
begin
maximum size is (110,60);
position is
  (case r.type begin
        "CV":     800
        "SSN":    1600
        "SSBN":   1600
        "SSGN":   1600
        "CGN":    2500
        "CG":     2500
        "CA":     2500
        "DDG":    3200
        "FF":     3200
        "AGI":    3200
        "AO":     4000
        default: 1600
        end,  case r.nat begin
                    "US":     1200
                    "UR":     2000
                    default: 2500
                    end);
   template icon case r.type begin
        "CV":    carrier
        "SSN":   sub
        "SSBN":  sub
        "SSGN":  sub
        "CGN":   cruiser
        "CG":    cruiser
        "CA":    crusier
        "DDG":   destroyer
        "FF":    destroyer
        "AGI":   destroyer
        "AO":    oiler
        default: cruiser
        end;
    scale is r.beam*2 percent;
    color of region 1 is case r.ready begin
        "1":     green
        "2":     yellow
        "3":     orange
        "4":     red
        default: gray
        end;
    attribute region r.nam   from (5,16) to (70,28);
    attribute region r.ircs  from (5,28) to (70,40);
    attribute region r.conam from (5,40) to (70,52);
end;
```

Schéma N° 1-23. : La description d'une classe d'icônes pour une base de données stockant des informations sur des bateaux [Her80].

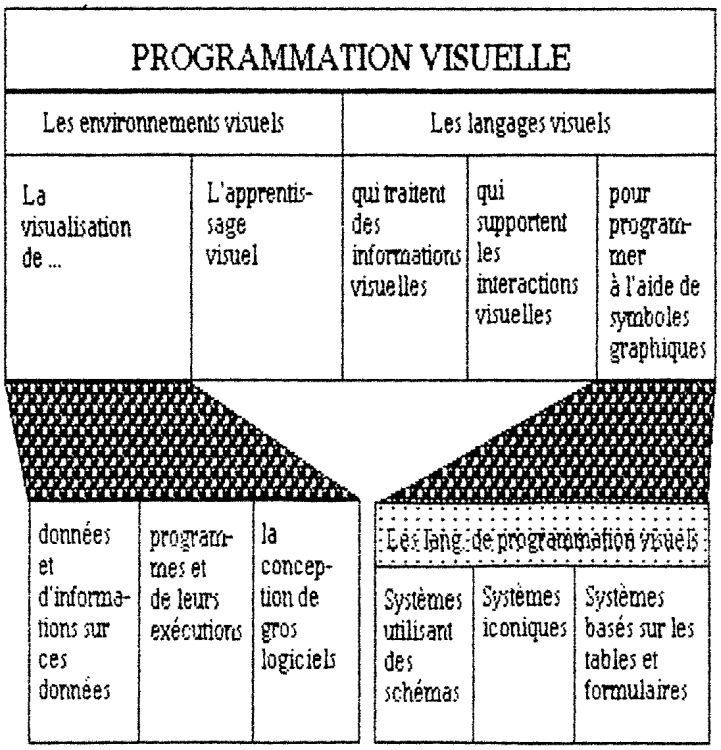


Schéma N° 1-24. : Les langages de programmation visuels dans la structure de la programmation visuelle.

Puisque les interactions visuelles sont devenues une aide indispensable, de nouveaux langages ont été développés pour décrire des interactions visuelles. Il existe de quelques méthodes de conception de ce type de langages mais ces langages ne sont que très rarement visuels [Her80] [Car85].

Dans la section I.2.1.1., nous avions présenté un Système de Gestion Spatiale de Données (S.G.S.D.) qui présentait le contenu d'une base de données dans une structure spatiale. Pour construire cette structure spatiale, le gestionnaire du S.G.S.D. a besoin d'un langage qui décrit la forme graphique des éléments de cette structure. Ce langage s'appele ICDL (Icon Class Description Language). Il se compose d'une série de commandes qui reçoivent des valeurs d'attributs et effectuent des opérations graphiques telles que sélectionner une forme graphique, colorier une région ou insérer du texte. Un exemple de description (schéma N° 1-23.) reprend les commandes nécessaires pour générer les icônes utilisées dans l'exemple déjà présenté de la base de données d'une société maritime (schéma N° 1-4.).

I.3.3. Les langages de programmation visuels

I.3.3.1. Définition

Un langage de programmation visuel peut être défini comme "un langage qui utilise des représentations visuelles (en plus ou à la place des mots et des nombres) pour accomplir ce qui normalement aurait été écrit dans un langage de programmation traditionnel (à une seule dimension)" [Shu88] . Notons que cette définition n'impose aucune restriction sur le type de données ou d'informations traitées. Ce qui est important, c'est que, pour être considéré comme langage de programmation visuel, le langage lui-même doit employer des expressions visuelles significatives (et pas seulement décoratives) comme technique de programmation [Cha86] [Cha87] [Cha89].

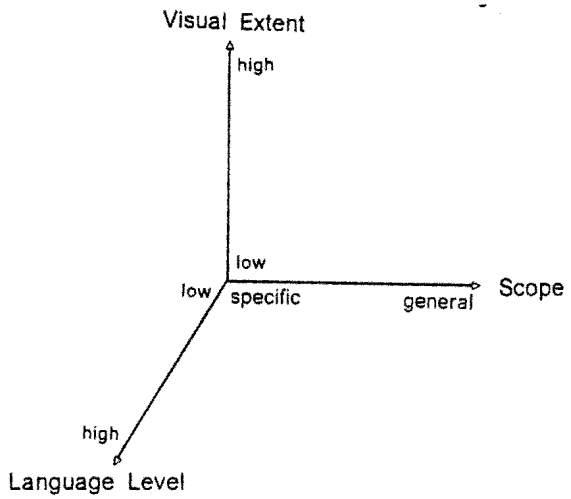


Schéma N° 1-25. : Les trois dimensions définissant le profil d'un langage de programmation visuel [Schu85b].

I.3.3.2. Le profil d'un langage de programmation visuel

Lorsqu'on désire évaluer un langage de programmation visuel, trois aspects importants viennent à l'esprit (schéma N° 1-25.) [Shu85b] :

- 1) **Le niveau du langage** est la mesure inverse de la quantité de détails que l'utilisateur doit donner à l'ordinateur. Le niveau du langage de programmation COBOL est supérieur à celui du langage de programmation ASSEMBLEUR.

- 2) **L'étendue du langage**, allant de générale à spécifique, représente la quantité de travail qu'un langage est capable d'effectuer. On dit que le langage ASSEMBLEUR a un domaine d'application plus large que celui de COBOL car ce dernier est exclusivement dédié à des applications de gestion.

- 3) **L'étendue des expressions visuelles** est une mesure du nombre d'expressions visuelles introduites dans le langage de programmation. Les expressions visuelles sont des représentations visuelles (et non textuelles) significatives (icônes, graphes, organigrammes, images, ...) utilisées comme composantes du langage de programmation. Plus il y a de représentations visuelles, plus l'étendue visuelle est importante.

En se basant sur les représentations visuelles employées pour concevoir des programmes, la plupart des langages de programmation visuels que l'on trouve dans la littérature, tombent dans trois grandes catégories :

- 1) Dans la première, les **icônes** sont délibérement conçus pour jouer le rôle principal dans la programmation. Nous parlerons de cette catégorie dans le chapitre I.3.5. .

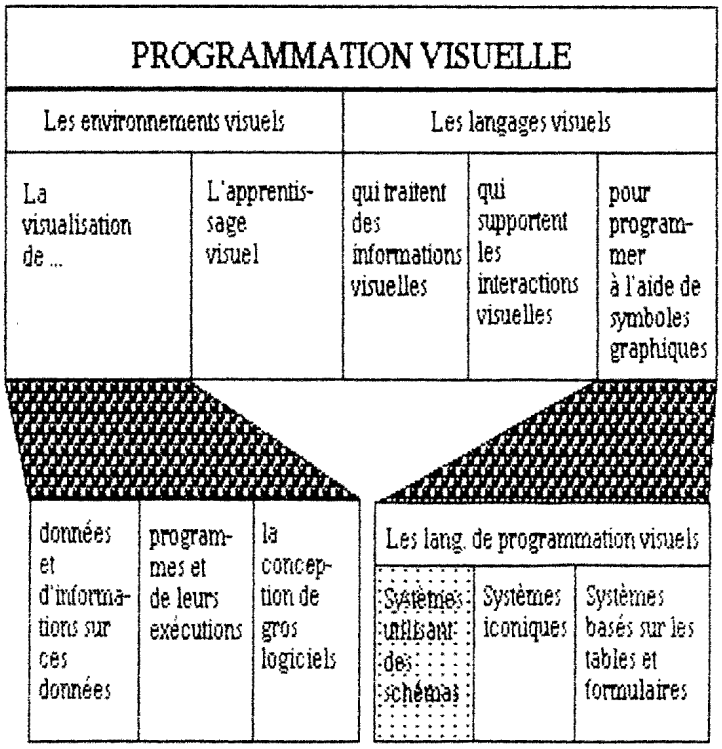


Schéma N° 1-26. : Les systèmes utilisant des schémas dans la structure de la programmation visuelle.

- 2) Dans la seconde, nous trouvons des **organigrammes et des diagrammes**. Ces représentations sont utilisées depuis bien longtemps sur papier. Mais ici, elles sont soit incorporées dans les constructions de programmes en tant qu'extensions des langages de programmation classiques, ou sont soit devenues des unités interprétables par la machine. Les travaux de cette catégorie vous seront présentés dans le chapitre I.3.4. .

- 3) La troisième catégorie se situe entre les deux précédentes. Ici, les représentations graphiques sont conçues comme faisant partie intégrale du langage. Cependant, contrairement aux icônes des systèmes iconiques, elles ne sont pas les "stars" du système et à l'inverse des extensions graphiques, le langage n'est pas conçu pour travailler en liaison avec d'autres langages de programmation classiques et il ne peut fonctionner sans représentations graphiques. Les caractéristiques de ce type de système vous seront détaillées dans le chapitre I.3.6.

I.3.4 Les systèmes utilisant des schémas

Depuis plusieurs années, des schémas de toutes sortes ont été utilisés comme aide visuelle pour l'illustration ou la documentation d'un ou de plusieurs aspects des programmes (ex. diagrammes de flux, schémas entités-associations, organigrammes, ...). Mais, en général, ces aides graphiques n'englobaient pas les programmes eux-mêmes car elles n'étaient pas exécutables. Les coûts élevés des terminaux graphiques haute-résolution et le gros volume de stockage des représentations graphiques ont gardé ces techniques de schémas sur les papiers et les tableaux.

Etant donné la chute rapide des coûts des écrans graphiques, l'émergence des stations de travail très puissantes, et le désir d'augmenter la productivité des programmeurs, des efforts ont été effectués récemment pour introduire des organigrammes, des graphiques ou tout autre schéma comme extension graphique du code exécutable.

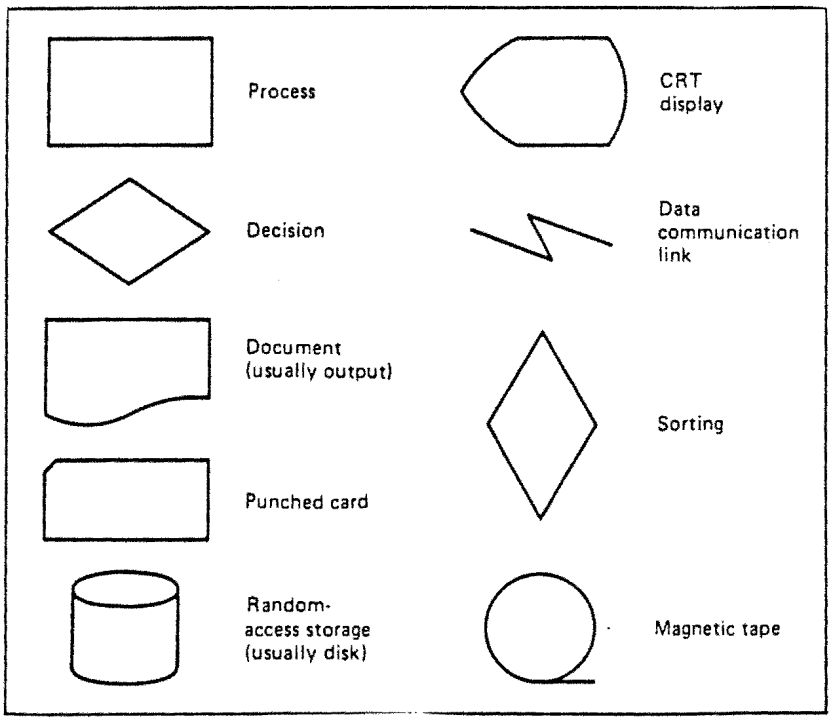


Schéma N° 1-27. : Les symboles classiques formant les premiers organigrammes (IBM Template X20-8020).

Par l'implémentation de schémas exécutables, on crée des avantages dans tout le cycle de vie traditionnel de la conception d'un programme. La programmation induit différentes étapes distinctes : l'analyse d'opportunité, l'analyse conceptuelle, l'encodage, les tests, etc ... . Il est alors évident que nous allons faire appel à différentes sortes de représentations graphiques comme aide visuelle pour abstraire des programmes.

Un problème sérieux avec cette approche est le besoin de garder à jour à la fois les graphiques et le code qui sont tous deux une représentation d'un même programme. Il n'est pas surprenant que quelque part dans le processus de mise à jour, les différents graphiques (qui sont aussi une partie de la documentation) ne représentent plus le code actuel qui est exécuté, et par conséquent, créent des problèmes dans la maintenance future du programme.

En rendant ces schémas exécutables, on essaye de fusionner les deux processus séparés (analyse et conception) en un seul. Cela ne rend pas seulement les programmes plus faciles à comprendre, mais aussi à documenter et à maintenir.

Nous allons brièvement présenter quatre sortes de schémas qui sont couramment utilisés en programmation visuelle. On pense évidemment aux organigrammes, aux graphes GNS, aux diagrammes de flux de données et aux diagrammes états-transitions. La plupart des projets dans cette catégorie de langage essayent de rendre l'un de ces schémas "exécutable".

I.3.4.1. Les organigrammes

Utiliser des graphiques ou des schémas en relation avec la programmation est une technique presque aussi vieille que la programmation elle-même. Les organigrammes ont été développés, au départ, comme aide pour les programmeurs utilisant le langage assembleur. Celui-ci n'a malheureusement pas été conçu pour faire de la programmation structurée. Les programmeurs ont alors appris à dessiner des organigrammes pour rendre plus claires les structures de programmes dans leur esprit; ainsi ils ne se perdront plus dans une mer de détails de bas niveaux.

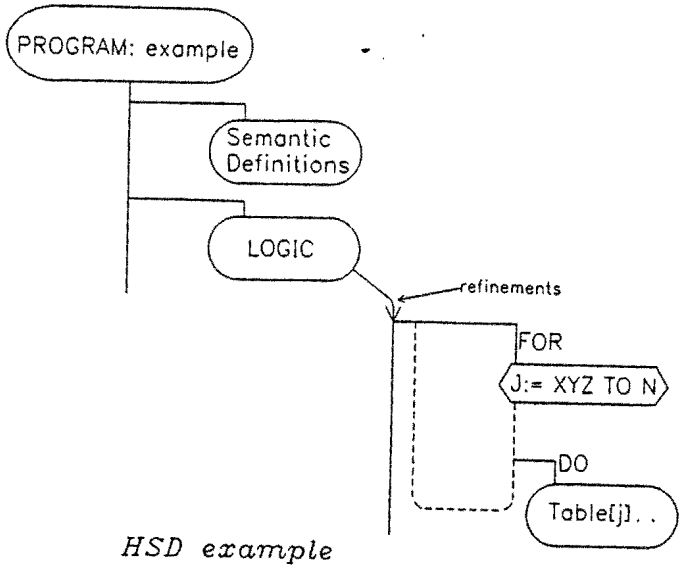


Schéma N° 1-28. : Un exemple d'organigramme évolué [Dia80].

PRIME : PROCEDURE; SPEC : this program computes the first n (>2) prime numbers			
PARAMTERS : NAME	SIZE	TYPE (I/R/C/CV)	USE (I/O/M)
LOCAL / GLOBAL VARIABLES : NAME p v n, x, lim, square, pr	SIZE 1000 35	TYPE (I/R/C/CV) integer integer integer	USE (L/G) local local local
p (1) = 2; x = 1; lim = 1; square = 4; /*initialization*/			
get list (n) ; put skip edit (p(1), (f(10)) ;			
LOOP i = 2 to n			
pr = 0;			
LOOP while (pr = 0) /* loop while not prime */			
x = x + 2 ;			
IF square < = x TRUE ? FALSE			
v(lim) = square ; lim = lim + 1; square = p (lim) * p (lim) ;			
pr = 1 ;			
LOOP k = 2 to lim - 1 while (pr = 1)			
IF v (k) < x TRUE ? FALSE			
v (k) = v (k) + p (k) ;			
IF x = v (k) TRUE ? FALSE			
pr = 0 ;			
p (i) = x ; put skip edit (x) (f (10)) ;			

Schéma N° 1-29. : Un programme exprimé par un graphe GNS étendu [Ng79].

Pour décrire le flux de contrôle des programmes, toutes sortes de symboles (schéma N° 1-27.) sont connectés par des lignes flèchées. Avec la venue de langages de programmation de plus hauts niveaux et les concepts de programmation structurée, les techniques d'organigrammes sont aussi influencées par l'approche structurée. Cette évolution a surtout créé de nouveaux symboles représentant de nouveaux concepts (schéma N° 1-28.).

Des systèmes de programmation sont alors nés en rendant ces nouveaux types d'organigrammes directement exécutables [Cun86] [Dia80].

I.3.4.2. Les variantes des Graphes de Nassi-Shneiderman (GNS)

Parmi les diagrammes de flux structurés, les graphes de Nassi-Shneiderman sont probablement les plus connus [Nas73]. Comparée aux diagrammes de flux conventionnel, l'utilisation des GNS ne met pas seulement en évidence la structure logique des composantes d'un programme, mais elle esquisse aussi un portrait du programme sous une forme plus compacte (schéma N° 1-29.). La forme rectangulaire des symboles fait aussi que les GNS prennent moins de place. Pour ces raisons, les GNS (ou leur dérivés) ont été choisis par un certain nombre de chercheurs comme forme d'exécution graphique [Alb84] [Gli84] [Ng79] [Pon83].

Notons que l'ensemble des langages faisant partie de cette catégorie peut être caractérisé comme des extensions graphiques de langages de programmation conventionnels existants (appelés langages de bases). Dans le cadre de l'analyse dimentionnelle (selon les trois dimensions précédemment présentées dans I.3.3.2.), nous observons que les extensions graphiques ont la même étendue de langage et le même niveau de langage que leur langage de base respectif. L'introduction d'expressions visuelles dans la programmation sert principalement à rendre la structure des programmes plus claire.

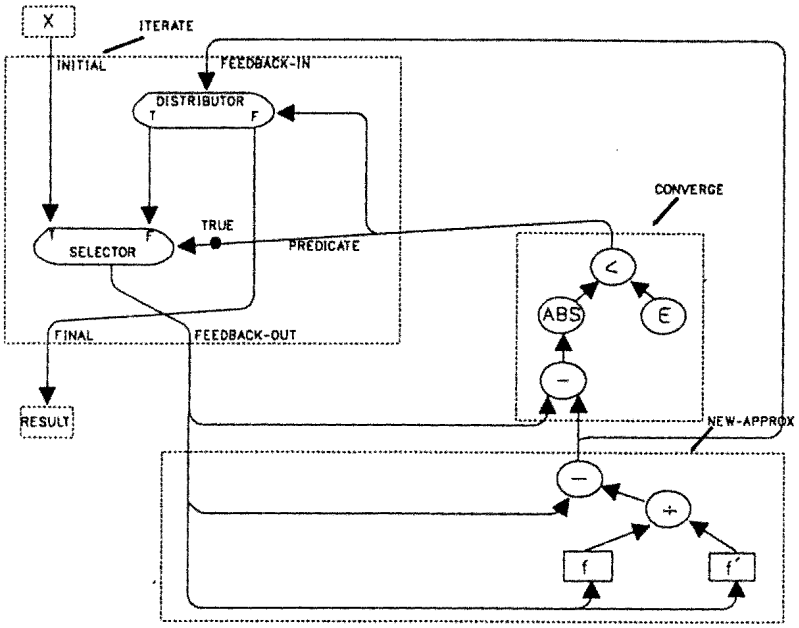


Schéma N° 1-30. : Ce diagramme de flux calcule la valeur nulle d'une fonction par la méthode de Newton [Dav82]. Il suffit de donner une valeur à la variable X et le programme calcul la valeur nulle ("RESULT") de la fonction représentée par f.

I.3.4.3. Les diagrammes de flux de données

Les organigrammes et les GNS expriment la structure du flux de contrôle des programmes mais ignorent le flux des données. D'un autre côté, les diagrammes de flux de données sont utilisés en rapport avec des langages du même type. Ces langages se basent principalement sur le modèle du flux de données qui ne se préoccupe que des valeurs. Une opération est en état de faire quelque chose si et seulement si toutes les valeurs d'entrées requises ont été calculées. Ce type de langage n'introduit pas de contraintes de séquence autres que celles imposées par les dépendances de données. Un programme dans un langage de haut niveau de flux de données est directement convertible en un graphe dont les noeuds représentent les fonctions et dont les arcs représentent les dépendances de données entre ces fonctions.

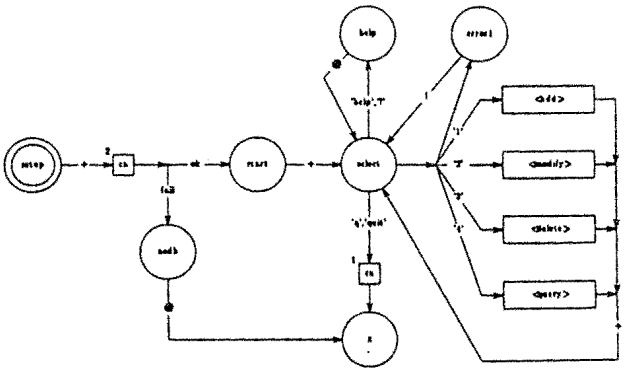
De très bonnes raisons nous permettent de représenter les langages de flux de données par des représentations graphiques. Les langages de flux de données gèrent les actions des programmes par une simple règle de déclenchement basée sur la disponibilité des données. Lorsque les arguments d'un noeud sont disponibles, on dit qu'il est "tirable". Après l'avoir tiré, les résultats du noeud sont envoyés à d'autres fonctions qui nécessitent ces résultats comme arguments. Ce comportement suggère que le programme puisse être parfaitement décrit par un graphe orienté dans lequel chaque noeud représente une fonction et chaque arc orienté, un moyen conceptuel par lequel les données circulent (schéma N° 1-30.) [Dav82] [Rod79].

Le modèle le plus couramment usité pour représenter ces flux de données est "le modèle du jeton". Dans ce modèle, les données sont toujours vues comme voyageant grâce aux arcs d'un noeud vers un autre dans un flux de jetons.

I.3.4.4. Les diagrammes état-transition

Les organigrammes et les diagrammes de flux dont nous vous avons déjà parlés portent une attention toute particulière sur les fonctions du programme et sur les données sur lesquelles ces fonctions travaillent. Ils n'ont donc aucun rapport avec la description des interactions entre l'utilisateur et l'ordinateur.

main



Actions

```
1    call shutdown
2    call startup


diagram main entry setup exit x

database 'usedddb'
library  '../scripts'

tab t_0 15
tab t_1 20

message header
  cs,r2,c0,c_ 'USE Data Dictionary'

message lastline
  r$,c0,'Hit any character to continue.'

node setup

node select
        tomark_A,cs,r+3,t_0,'Please choose ',
        r+2,t_1,'1: Add a dictionary entry.',
        r+2,t_1,'2: Modify a dictionary entry.',
        r+2,t_1,'3: Delete a dictionary entry.',
        r+2,t_1,'4: Query data dictionary',
        r+2,t_1,'help: Information on use of program',
        r+2,t_1,'quit: Exit USE/Data Dictionary',
        r+2,t_0,'Your choice: '

node help
        cs,r$-3,c0,'For more information about a command, enter',
        r$-2,c0,'the command number, press return and then type "help" or "?"',
        r$,c0,'Hit any key to continue'

node nodb
        cs,r$,c0,'Could not open database directory'

node start
        header,mark_A

node x
        cs

node error1
        r$-1,c0,rv,bell,'Please type a number from 1 to 4.',rv,
        lastline
```

Schéma N° 1-31. : Ce diagramme état-transition est le niveau supérieur d'un système de gestion d'un dictionnaire de données [Was86b].

Par contre, les diagrammes état-transition sont bien adaptés pour décrire des interfaces. Une des vertus principales des notations du diagramme état-transition est qu'en donnant les règles de transition pour chaque état, elles rendent explicite ce que l'utilisateur peut faire à chaque point d'un dialogue, et ce que les effets seront.

Les notations de base de ces diagrammes suivent des conventions très largement utilisées. Graphiquement, un diagramme état-transition consiste en un réseau de noeuds (représentant les états) ainsi que d'arcs orientés et nommés (représentant des transitions d'états). La traversée d'un chemin (path) fait déclencher l'opération y étant associée. Cependant, le diagramme état-transition ne représente pas toutes les informations dans chaque état (Ex: des contraintes sur les entrées, les fonctions à exécuter sur les données, etc ...). Il est donc parfois nécessaire de compléter les diagrammes avec des descriptions textuelles sur les actions à exécuter (schéma N° 1-31.).

Beaucoup d'extensions du diagramme état-transition ont été proposées pour spécifier le comportement des sytèmes interactifs [Jac85] [Was85] [Was86] [Was86b]. Presque toutes les propositions soulignent le désir d'exprimer des spécifications précises d'une interface et d'accroître la participation de l'utilisateur dans les premières étapes de la conception. Peu d'efforts, cependant, se sont tournés vers l'exécution directe de ces spécifications.

Avoir des spécifications directement exécutables donne un moyen très efficace pour construire des prototypes d'interface utilisateur. Un prototype permet non seulement de faire évaluer l'interface désiré par les utilisateurs futurs. Il permet aussi au développeur d'évaluer les performances de l'utilisateur et la satisfaction de celui-ci durant les premières étapes du processus de développement d'un système. De plus, le prototypage facilite l'expérimentation d'un nombre d'alternatives et rend les modifications et les révisions moins difficiles.

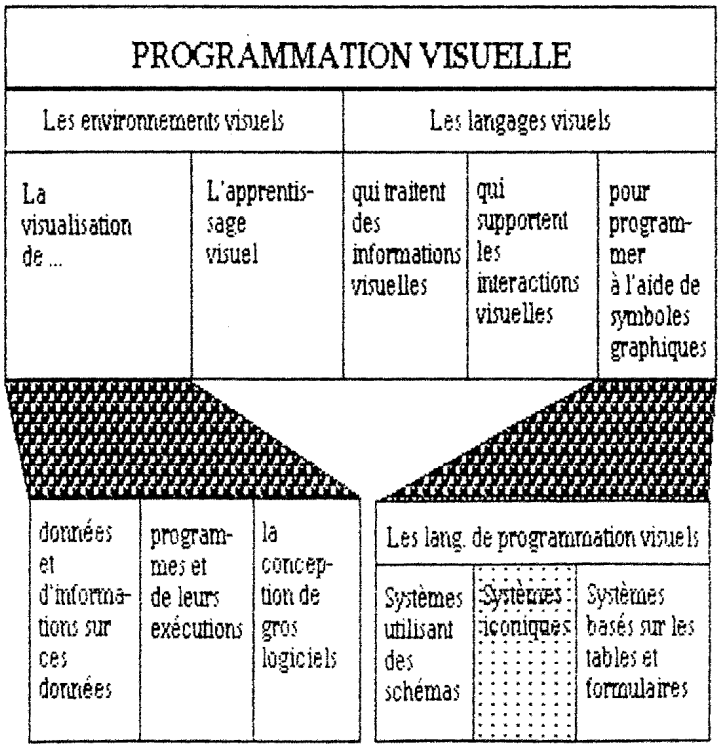


Schéma N° 1-32. : Les systèmes iconiques dans la structure de la programmation visuelle.

I.3.4.5. Conclusion sur les systèmes utilisant des schémas

Il est utile de mentionner qu'il y a en fait deux groupes types différents d'extensions graphiques des langages de programmation textuels. Le premier groupe (basé sur les organigrammes, les Graphes de Nassi-Shneiderman et les diagrammes de flux de données) se concentre sur la description de la structure d'un programme en terme de flux de contrôle / de données. Le second groupe (utilisant des diagrammes état-transition) se concentre sur les spécifications interactives d'interfaces utilisateurs.

Lorsqu'on rend les organigrammes et les diagrammes exécutables, les objectifs pricipaux diffèrent eux aussi. En général, le premier groupe s'efforce d'éliminer les écarts qui peuvent exister entre le programme et sa documentation. Le second groupe essaye de produire des prototypes qui facilitent l'expérimentation des alternatives avant qu'un système soit construit.

I.3.5. Les icônes et les systèmes iconiques

I.3.5.1. Réflexion sur le système idéographique chinois

Les images représentent sûrement la plus vieille forme d'écriture à l'heure actuelle. Les hiéroglyphes sont une forme ancienne d'écriture (égyptienne, maya, zapotèque, ...) dont les caractères sont, à un certain degré, des images reconnaissables. Le chinois, le plus ancien des systèmes d'écriture connu, s'est développé à partir de pictogrammes qui sont apparus il y a quelques 4000 ans, et sont utilisés dans leur forme actuelle depuis plus de 2000 ans. Avant de parler des icônes et des systèmes iconiques actuels, il serait intéressant de jeter un oeil sur ces idéogrammes chinois [Wan73].

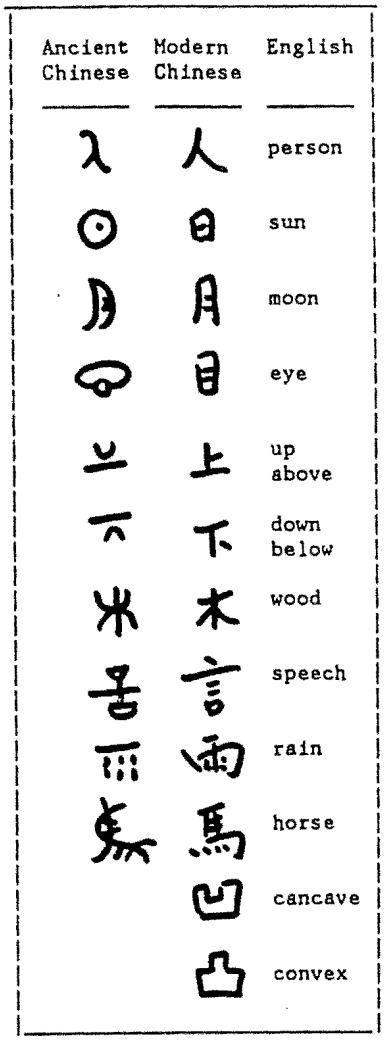


Schéma N° 1-33. : Chaque caractère représente un simple concept dans le système d'écriture chinois.

Schéma N° 1-34. : Le pictographe chinois pour le mot "cheval" ressemble réellement à un cheval. Pour peu, on l'entendrait galoper sur la feuille de papier.

Il y a des milliers de caractères idéographiques dans le système d'écriture chinois. Chaque caractère représente une simple syllabe chinoise et habituellement un simple concept. Les caractères chinois peuvent paraître étranges aux yeux des occidentaux. Mais avec un peu d'imagination, il est possible de "voir" des objets concrets ou des concepts abstraits dans les anciennes formes des pictogrammes (schéma N° 1-33.). Un caractère chinois a un rapport plus direct avec sa signification qu'un mot écrit en français [Liu89]. En français, la séquence de lettres signifiant "cheval" a un sens uniquement par les sons qu'elles représentent. La forme des lettres n'a aucune relation avec le concept qu'elles expriment. Pour un chinois, le caractère pour "cheval" ressemble à un cheval avec une crinière et quatre jambes (schéma N° 1-34.). L'image est tellement vivante que l'on peut presque entendre l'animal galoper à travers la page. On peut dire la même chose lorsqu'on compare les langages de programmation iconiques avec les textuels.

En d'autres termes, la forme d'expression picturale n'est pas la panacée pour tous les problèmes de communication, mais les images semblent fournir une motivation supplémentaire pour apprendre. Le défi, la fantaisie, et la curiosité (les facteurs les plus importants qui rendent les jeux informatiques si captivants) sont tous là lorsqu'on travaille avec des systèmes picturaux [Schn83].

I.3.5.2. Le contraste avec l'apprentissage visuel

Il serait peut-être utile de rappeler que la programmation par l'exemple ou programmation par démonstration est une technologie qui essaye aussi de rendre la programmation plus simple. L'utilisateur décrit un programme en donnant des exemples à l'ordinateur sur ce qu'il désire que le programme fasse. Le système enregistre, et on espère aussi, généralise ce qui lui a été montré pour le réutiliser plus tard.

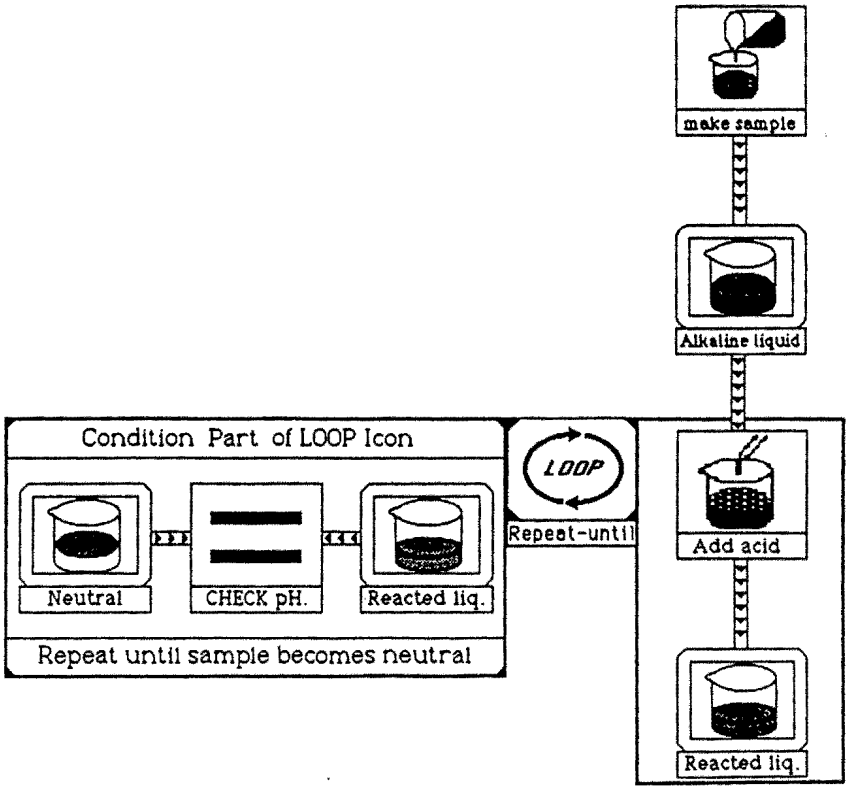


Schéma N° 1-35. : Petite expérience chimique simulée par un système iconique [Hir86].

On doit cependant garder à l'esprit que la programmation par l'exemple est une méthodologie de développement de programme et non un langage de programmation. Bien qu'à la fois la programmation par l'exemple et les langages de programmation sont utilisés pour produire des programmes, il y a une profonde différence entre ces deux approches. Dans la première approche, l'ordinateur est chargé de "faire comme on lui montre" où la façon de lui montrer se base fortement sur des interactions. Dans la seconde approche, l'ordinateur est chargé de "faire comme on lui a dit" où la façon de le dire est exprimée par des unités de construction propres aux langages de programmation.

Introduire des graphiques ou des images dans le processus de programmation ajoute une dimension intéressante et utile à la communication homme-machine. Donc, lorsqu'un langage de programmation utilise des expressions visuelles, il devient un langage de programmation visuel. Lorsqu'un système de démonstration utilise des représentations visuelles, il devient un système "d'apprentissage visuel". Mais les principes fondamentaux de ces deux approches différentes ("Fais comme je te le montre !" et "Fais comme je te le dis !") ne sont pas altérés par l'inclusion de symboles graphiques.

I.3.5.3. Les langages de programmation iconiques

Ces dernières années, un nombre significatif de langages iconiques ont été présentés dans la littérature [Broc86] [Cha86] [Ede88] [Gli84] [Hal84] [Hir86] [Ich90] [Kar88] [Mcl86] [Rouk90] [Smi82]. En gros, ils ont tous le même objectif : utiliser des icônes comme briques de construction d'un langage de programmation. Dans cette catégorie de langage, on "construit" les programmes au lieu de les écrire.

La philosophie générale de conception de programme par un langage de programmation iconique est la suivante :

- Choisir dans une palette d'images celles qui représentent visuellement les structures de données et les variables nécessaires pour l'élaboration du programme désiré.

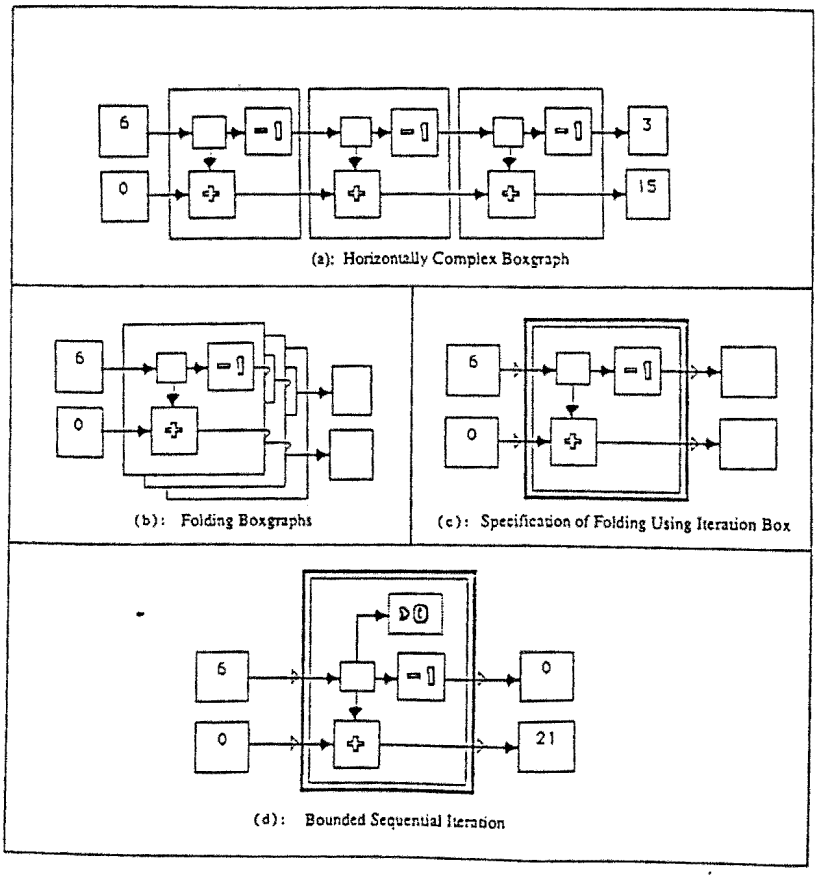


Schéma N° 1-36. : Exemples d'abstractions possibles avec des icônes [Mcl86].

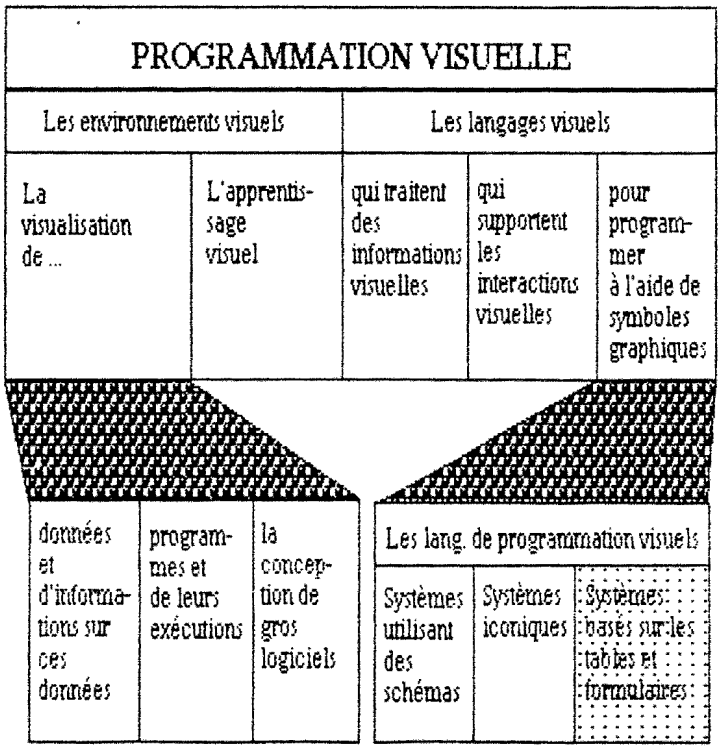


Schéma N° 1-37. : Les systèmes basés sur les tables et formulaires dans la structure de la programmation visuelle.

- Exprimer l'algorithme désiré par un dessin multi-dimensionnel et logiquement structuré.

- Regarder le programme pendant qu'il s'exécute et voir les résultats générés.

- Si le programme ne fait pas ce qui était prévu, situer où et quand les erreurs se produisent.

Dans le schéma N° 1-35., un chimiste désire simuler une expérience chimique. Il a choisi parmi l'ensemble des icônes représentant des données, des primitives ou des contrôles, celles qu'il doit utiliser. Il a assemblé ces icônes pour composer l'expérience désirée. Elle consiste à partir d'un échantillon, de le mettre dans une solution alcaline et d'y ajouter une petite quantité d'acide jusqu'à ce qu'elle soit neutre.

Le schéma N° 1-36. nous présente différents types d'abstractions d'icônes avec le système Show and Tell [Mcl86].

I.3.6. Les systèmes basés sur les tables et les formulaires

En 1979, le "CODASYL End-User Facilities Committee" (E.U.F.C.) a déclaré que "l'approche utilisant des formulaires (ou formes) est considérée comme l'interface la plus naturelle entre un utilisateur et les données" [Lef79]. Ceci est dû au fait qu'un très grand nombre d'utilisateurs emploient des formulaires (Ex: formulaire d'ordre d'achat, formulaire de rapport des dépenses, etc ...) ou des versions de formulaires (Ex: des rapports, des mémos, etc ...) dans leurs activités de travail de tous les jours ainsi que dans leur vie personnelle (formulaire des contributions, etc ...).

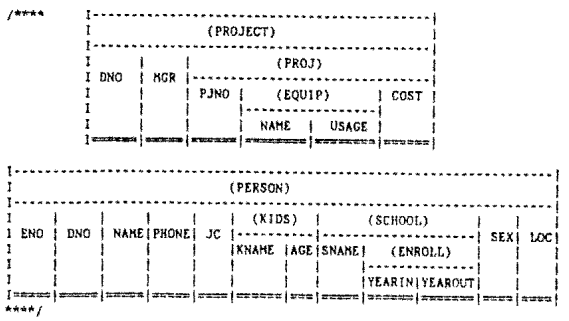


Schéma N° 1-38. : Deux formulaires de données exprimant la stucture de données des projets et des personnes y travaillant [Shu85a].

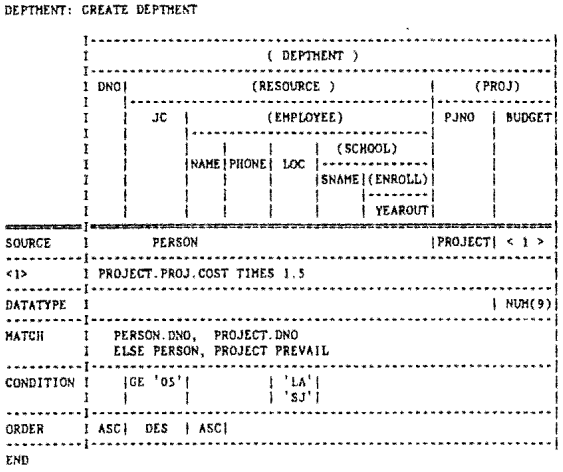


Schéma N° 1-39. : Un formulaire de programme exprimant la requête qui crée un nouveau formulaire de données à partir des deux précédents [Shu85a].

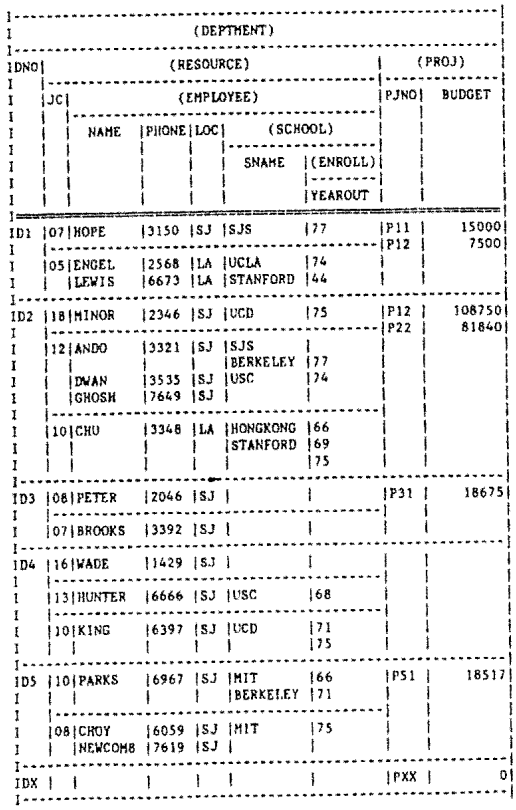


Schéma N° 1-40. : Le formulaire de données résultant de l'exécution de la requête présentée ci-dessus [Shu85a].

Le comité CODASYL n'est pas le seul groupe à considérer le formulaire comme "l'interface la plus naturelle". Le succès historique des tableurs témoigne de l'attrait de l'approche orientée formulaires ou tables. A l'heure actuelle, le remplissage de forme est utilisé couramment dans de très nombreux domaines en informatique (Les logiciels de bureautique, comme dispositif d'introduction et d'affichage de données, dans des requêtes sur des bases de données , etc ...). Il y a beaucoup de raisons qui nous poussent à utiliser cette approche. La plus importante de toutes vient du résultat d'une analyse des besoins en traitements d'informations des non-programmeurs. Elle révèle que la plupart des manipulations de données peuvent être naturellement exprimées ou pensées comme des traitements sur des formulaires [Fik85] [Row85] [Shu82] [Shu85a] [Yao84].

Les formulaires sont généralement utilisés dans deux classes de représentations :

- Les formulaires que l'on emploie comme représentation visuelle **des structures de données et des instances de ces données** (communément appelés formulaires de données) (schéma N° 1-40.). L'entête d'un formulaire peut représenter précisément une structure hiérarchique quelle que soit sa profondeur et sa largeur. Pour représenter les instances des données, leurs valeurs sont affichées sous l'entête du formulaire.

- Les formulaires que l'on emploie comme représentation visuelle **des programmes eux-mêmes** (dénommés aussi formulaires de programmes) (schéma N° 1-39.). En général, le traitement des données peut être exprimé par un (ou plusieurs) formulaire(s) de ce type qui prend (prennent) un ou deux formulaires de données comme argument d'entrée et produit (produisent) un autre formulaire de données comme résultat.

Un programme dans cette section consiste alors en une ou plusieurs spécifications de formulaire de données et de programme. Pour créer le formulaire "département" (schéma N° 1-40.), l'utilisateur est parti des deux formulaires "projet" et "personne" (schéma N° 1-38.). Pour ce faire, un programme représenté également par un formulaire (schéma N° 1-39.) extrait les données requises en fonction de plusieurs sortes de conditions.

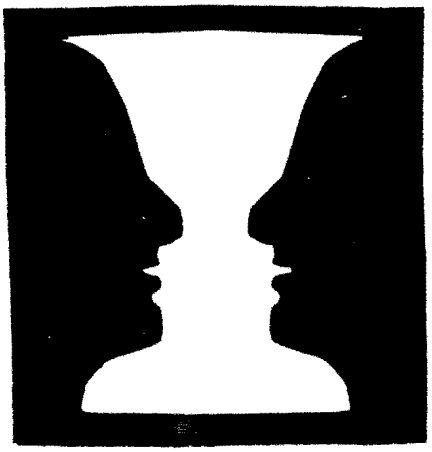


Schéma N° 1-40. : Une représentation graphique ambiguë.

I.4. Conclusion

Un certain nombre d'avantages ont été mis en évidence lorsqu'on a utilisé des représentations visuelles tout au long du développement d'un logiciel. Mais il existe quand même des problèmes non encore résolus qui pourraient faire l'objet de recherches complémentaires [Mye86b] [Shu88a]. Passons-les rapidement en revue :

- 1) Des confusions ou malentendus peuvent se produire lorsqu'on travaille avec des représentations graphiques (schéma N° 1-40.). Les confusions peuvent disparaître si l'on établit une compréhension commune suffisante. Il faudra encore découvrir des unités visuelles bien adaptées.

- 2) Difficultés lors du traitement de programmes ou de données de grandes tailles. Le problème se situe surtout au niveau de la surface réduite de l'écran. Des recherches doivent encore étendre les notions d'abstractions, de "scrolling" et de multi-fenètrages.

- 3) Un manque de généralité : La plupart des systèmes travaillent dans des domaines fort restreints. Le système Labyrinth que nous présenterons dans le chapitre suivant essaye de réduire ce défaut. La difficulté réside dans le choix des représentations visuelles qui sont suffisamment générales pour pouvoir s'appliquer à un plus large domaine.

- 4) Inefficacité : Les systèmes actuels sont souvent lents. Les progrès techniques et le choix judicieux des représentations internes accéléreront les systèmes de la future génération.

- 5) Les programmes créés ne sont pas souvent structurés (Ex. Une flèche peut parfois faire office de GOTO). Les représentations graphiques et le système de composition doivent faire l'objet de recherches supplémentaires.

- 6) Pas de place pour les commentaires : Des systèmes judicieux de commentaires interactifs pourraient être implémentés.

- 7) Les systèmes basés sur la programmation par l'exemple (avec inférences) ne reçoivent aucune information sur la stucture des programmes qu'ils doivent générer automatiquement. Un exemple supplémentaire peut faire varier radicalement la structure d'un programme. Avant de donner les exemples, il faudrait communiquer naturellement au système un minimum d'informations sur la structure du programme qu'il doit générer (Ex. : Quelle valeur est une constante ?, Quelle valeur est une variable ?, les conditions éventuelles, etc ...).

Comme vous avez pu le voir, la programmation visuelle est un domaine où la recherche est très intense. L'objectif principal consiste à améliorer l'interface des environnements de programmation. Cette synthèse a donc tenté de classer les systèmes actuels et a présenté les problèmes généraux en espérant clarifier votre perception du domaine.

Chapitre II : LABY : Un logiciel graphique d'implémentation

II.1. Introduction

Jusqu'à présent, nous vous avons présenté les grandes tendances qui guident les chercheurs dans le vaste domaine de la programmation visuelle. Cette présentation a eu le mérite de retracer les différentes étapes parcourues pour arriver aux projets actuels. Ceux-ci sont en effet fortement inspirés des divers succès engendrés par les prototypes développés ces 10 dernières années.

Les progrès de la technologie des interfaces homme-machines (manipulation directe, écrans bitmaps, multi-fenêtrage, toolkits conviviaux, menus déroulants, etc ...) se répandent rapidement dans tous les milieux. Cette expansion réserve certainement un avenir prospère aux logiciels qui communiquent via diverses interactions visuelles. Maintenant, les objets interactifs supportant cette interaction (icônes, menus déroulants, barres de défilements, schémas, etc ...) ne sont plus simplement "décoratifs" mais ils participent activement aux traitements désirés (programmation visuelle, conception de circuits électroniques, élaboration d'un plan d'un immeuble, etc ...). Au sein de cette nouvelle génération de logiciels interactifs, on retrouve la plupart des outils de C.A.O. (Conception Assistée par Ordinateur) que l'on utilise dans des domaines différents tels que l'architecture, l'électronique ou l'ingénierie. Il y a aussi les outils graphiques faisant partie d'une des catégories décrites dans le premier chapitre.

Pour développer efficacement ce type d'applications, il va falloir utiliser une autre technique que la programmation classique. Celle-ci n'ayant pas été développée dans ce but, le travail de conception de tels systèmes ressemble à un véritable parcours du combattant. La première étape de simplification consiste à utiliser des toolkits pour concevoir l'interface du

logiciel. Mais malgré cette aide, il reste quand même un grand travail de programmation et de formation. Alors **pourquoi ne pas créer une fois pour toute un logiciel graphique qui en construira d'autres du même type ?** On réaliserait tout ce travail difficile de programmation une fois pour toute avec, par exemple, un langage de programmation orienté-objet et un toolkit. Le programme ainsi créé est générique et ne concerne plus un domaine particulier. On peut dès lors élaborer aisément les logiciels graphiques que l'on désire, sans devoir passer par une programmation de longue haleine. Ce principe intéresse tout particulièrement les utilisateurs non expérimentés qui désirent garder leur indépendance vis-à-vis des programmeurs professionnels.

Le système Labyrinth [1] (Laby) [Kat90b] est précisément un de ces logiciels graphiques qui peut en construire d'autres. Je vais tout d'abord décrire en quelques lignes les concepts fondamentaux de ce système en cours de développement.

Laby est un logiciel graphique qui n'est lié à aucun domaine particulier (il est "general-purpose"). Il offre la possibilité de développer très facilement des outils graphiques ou des logiciels de CAOs. Dans sa plus simple utilisation, il fait office d'**éditeur graphique** qui travaille sur des **dessins hiérarchiques** et qui mémorise les **connexions** entres les différentes composantes de ces dessins. Ainsi, lorsque l'une d'entre elles se déplace, les autres la suivent en maintenant automatiquement les relations qui existaient. Les composantes des dessins hiérarchiques, appelées **cellules** (ou objet), ont des paramètres d'entrées et de sorties, appelés **ports**. Ceux-ci vont façonner les objets selon les besoins. Toutes les cellules ne sont pas nécessairement visibles sur le dessin, quelques-unes d'entre elles peuvent calculer des valeurs ou faire des actions arbitraires (cellules invisibles). Toutes les valeurs des ports n'ont pas besoin d'être des coordonnées géométriques (des points), elles peuvent être d'un tout autre

[1] Le système Labyrinth est actuellement développé à l'institut d'informatique du FORTH (FOndation of Research and Technology - Hellas) où j'ai eu la chance de faire mon stage de fin d'études. Ce centre de recherche est installé à Héraklion, capitale de la Crète : berceau de la civilisation méditerranéenne. Le nom de ce système vient du labyrinthe mytique crétois composé d'un ensemble compliqué de corridors. Ce labyrinthe forme le splendide Palais de Knossos (1500 av. JC) érigé à 1 Km au Sud du centre de recherche. Les auteurs espèrent que ce nom évoquera la capacité du système à traiter rapidement des graphiques complexes.

type (nombres, caractères, listes, etc ...). Le système mémorise aussi les connexions entre les ports des cellules. Pour Laby, une connexion signifie l'égalité des valeurs des ports et pour maintenir ces égalités, c'est-à-dire pour propager un changement à toutes les cellules affectées, Laby possède un "**évaluateur d'événements**" ("event-driven evaluator"). Un utilisateur habitué peut définir de nouveaux types de valeurs de ports, de nouvelles cellules visibles ou invisibles avec les fonctionalités qu'il désire. Laby peut donc être adapté à des besoins fort spécifiques. Il sert de noyau à des outils graphiques ou de CAO pour des environnements différents.

Cette rapide présentation ne donne qu'une bien petite idée des possibilités offertes par le système. Avant d'aborder la description précise de chacun des concepts mis en évidence ci-dessus (en caractères **gras**), nous allons exprimer les motivations qui ont poussé le FORTH (FOndation of Research and Technology - Hellas) à développer un tel système.

II.2. Les applications visées

La motivation initiale qui a conduit au développement de Laby est née d'une demande croissante d'outils de Conception Assistée par Ordinateur. En plus des outils de CAO courants (pour ingénieurs, architectes ou physiciens), les auteurs pensaient que ce système s'avèrerait très utile pour d'autres sortes de conception telles que : la construction d'algorithmes et des unités algorithmiques qui les composent, la création de formulaires administratifs interactifs, la mise en page de textes, la construction d'interfaces, etc ... . Nous verrons plus tard que les auteurs avaient sous-estimé le potentiel de leur projet.

II.2.1. Un système basé sur les contraintes

Lors de conceptions assistées par ordinateur, un des concepts communs à travers toutes les disciplines (électricité, mécanique, architecture, etc ...) est l'utilisation d'unités graphiques et symboliques pour représenter l'objet en construction. Sur le marché actuel, il y a beaucoup "d'éditeurs graphiques", "d'éditeurs de schémas", ou de systèmes apparentés. La plupart d'entre eux représentent les informations graphiques par une collection de primitives graphiques (Ex. droites, cercles, etc ...) avec des coordonnées définissant leurs emplacements. Une représentation aussi simple ne peut décrire explicitement les relations et les dépendances entre les différents éléments qui forment l'objet en construction. Un des défauts majeurs dû à cette déficience se manifeste lorsque l'on doit modifier le dessin, car l'utilisateur doit lui-même propager les contraintes à la main. Le concepteur remarquera assez facilement s'il respecte ou non des contraintes de type graphique en regardant le dessin. Mais pour celles d'un autre type (donc invisibles), l'utilisateur doit avoir une très bonne mémoire pour ne rien oublier.

C'est pour cela que le système Labyrinth utilise une représentation interne qui inclut les relations entre les composantes. Donc, lorsque l'on modifie un dessin, ces relations sont automatiquement maintenues par le système (schéma N° 2-1.) [Kat89a].

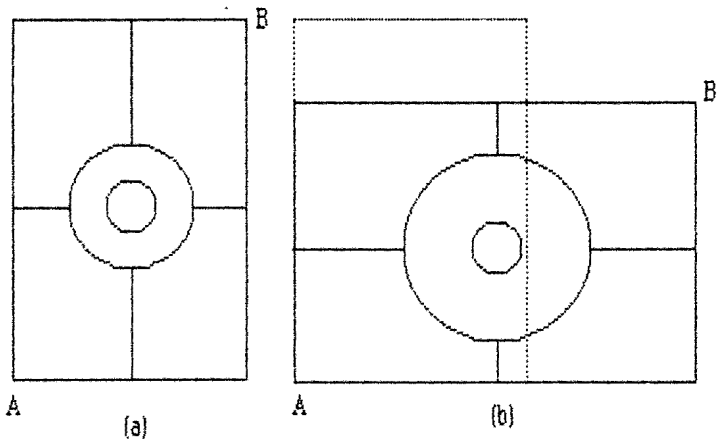


Schéma N° 2-1. : Laby mémorise les relations entre les composantes.

Dans l'exemple du schéma N° 2-1., l'utilisateur a décrit non seulement les coordonnées des composantes du schéma (un rectangle, deux cercles et quatre droites), mais aussi les relations, les contraintes et les dépendances qu'elles ont entre elles. Ainsi, le schéma (a) a été défini de sorte que :

- les cercles sont centrés par rapport au rectangle, lui même défini par les points A et B,

- le plus petit des deux cercles est de diamètre constant,

- le plus grand cercle a un diamètre égal à la moitié de la droite horizontale du rectangle,

- et les quatre lignes rejoignent les milieux des côtés du rectangle aux points correspondants sur le plus grand cercle.

Ainsi, si l'on déplace le point B en bas à droite, le schéma (a) se transforme automatiquement en schéma (b). Si l'on avait effectué la même opération sur un éditeur de type classique, seuls les côtés du rectangle auraient bougé et les autres transformations seraient à charge de l'utilisateur.

Il existe un autre cas important où l'on doit conserver des informations sur les relations entre les composantes. C'est lorsqu'un programme d'analyse doit interpréter, d'une façon ou d'une autre, un schéma. En cas d'absence de telles informations, le programme d'analyse devrait entre autre réaliser un traitement de comparaison des coordonnées. Ce traitement risque d'être coûteux en calcul lorsqu'il faudra découvrir des équivalences de points ou de lignes. Par exemple, on pourrait exiger d'un CAO pour architecte d'effectuer une analyse structurelle sur les immeubles ou maisons que l'utilisateur doit dessiner. Le système nécessite alors des informations supplémentaires différentes des coordonnées des objets (résistance des matériaux, pressions, poids de l'objet, etc ...).

Il y a évidemment d'autres systèmes d'édition graphique qui tiennent compte de la structure et des connexions de ce qu'on édite. Par exemple, la plupart des systèmes de production de circuits électroniques connaissent les connexions (fils) entre les transistors, diodes, etc ... pour tester l'efficacité du circuit. Cependant, tous les systèmes de ce type que nous connaissons ne s'appliquent pas à un domaine très large. Ils ne peuvent être utilisé que dans le champ très spécifique pour lequel ils ont été créés au départ. Laby espère combler ce trou en offrant un outil de conception graphique applicable à tous les domaines.

II.2.2. Les facultés d'adaptation du système face à l'environnement qui l'entoure

Pour arriver à ce résultat, les concepteurs du système ont bien séparé le noyau du programme des objets sur lequel il doit travailler. Ce noyau restant constant pour tous les CAO et outils graphiques que l'on désire employer, il suffit de changer de **librairie de cellules** pour adapter ce noyau à tous nos désirs. On peut ainsi comparer le corps de Laby à un magnétoscope sans cassette vidéo. Sans elle, le magnétoscope est inutile comme Laby sans ses librairies de cellules. Le magnétoscope peut devenir une source de peur si l'on visionne un film d'horreur (avec du suspens, des monstres, des vampires ou des extra-terrestres). Le même magnétoscope peut aussi devenir une source de joie en regardant un film comique (avec des gags, des tartes à la crème, des quiproquos ou des situations cocasses). On rencontre donc dans chaque type de films des concepts qui leur sont propres.

Summary of Long Commands	
Command	*Function*
`:edit cellName`	Start editing another cell – existing or not yet existing
`:defip portName`	Make the *to* selection be a new input port of the edit cell
`:defop portName`	Make the *from* selection be a new output port of the edit cell
`:undefp portName`	Undefine a port of the current edit cell
`:use [cellName]`	Get another of these cells; connect it to the *to* selection
`:set value`	Modify the value feeding the *from* selection to the new value. Examples: `:set "hello"`, `:set 3.14`, `:set <2.5, -3>`
`:const value`	Create a constant and make it *to* selection
`:offst <x,y>`	Create a constant-offset rel. to the origin (make it *to* sel.)
`:seeall`	fit the entire edit cell into the window

`:zoom [number]`
`:ZOOM`
scrolling
	Zoom-in (*number*>1.0) or zoom-out (*number*<1.0) by that factor
	equivalent to `:zoom 1.414` (zoom-in)
	Use scroll-bars, or `:right :left :up :down`

`:grid number`	Set the grid spacing (in mm) and turn the grid on
`:snap number`	Snap mouse-selected coordinates to integer multiples of *number*
`:getorigin` `:setorigin`	The *to* selection goes to the origin The origin goes to the *to* selection
`:geometry ...`	Set the size and location of the graphics window
`:redraw`	Erase and redraw the graphics window
`:flashControl ...`	Set the flashing mode for the current selections
`:load "file"`	Read and execute long commands from that file
`:path [...]`	Change or add to the directory search path
`:lscells`	List the composite cells in the path directory(ies)
`:getdef cellName`	Print documentation or the definition of a cell
`:postscript [...]`	Generate PostScript code for the current edit cell
`:cellname newName`	Change the name of the current edit cell
`:save ["directory"]`	Save edit cell into disk file
`:savestate "file"`	Save cell instance state information
`:loadstate "file"`	Load cell instance state information
`:define ...` `:connect ...`	Used in the textual description language Used in the textual description language
`:exit`	Exit from Laby

Schéma N° 2-2. : Tableau récapitulatif des commandes de Laby [Kat90b].

On peut dire la même chose pour chaque type de librairies de Laby :

- Une librairie de cellules géométriques fait de Laby un CAO pour élaborer des graphiques quelconques.

- Une librairie de cellules électriques offre à l'utilisateur un CAO pour concevoir et tester des circuits électriques (en connectant les ports du circuit avec ceux de la cellule "générateur").

- Une librairie d'objets bureautiques créera un outil graphique qui réalisera un grand nombre d'opérations courantes en gestion.

- etc ...

On peut ainsi imaginer toutes de sortes de librairies selon les besoins des utilisateurs.

II.2.3. Les utilisateurs du système

On peut d'ores et déjà préciser qu'il y a deux types d'utilisateurs de Laby :

II.2.3.1. Les producteurs

Le premier type adapte le système en définissant des librairies de nouvelles cellules (primitives ou composées). Ce sont des programmeurs connaissant parfaitement le **Langage de Description Textuel** (L.D.T.) propre au système Labyrinth. Grâce à ce langage de haut niveau composé principalement des commandes de Laby (schéma N° 2-2.) et des cellules primitives (schéma N° 2-3.) ou composées, le programmeur peut définir des **classes de cellules** qu'il stocke ensuite dans des fichiers. L'exemple qui suit illustre les caractéristiques du langage de description textuelle.

Summary of Primitive Cells	
Name	*Function*
Visible Cells	
`line`	Draw a straight line segment between the input points `p`, `q`, of color `col`, width (thickness) `wid`, and style `sty` (p. 15-16).
`arrow`	Draw an arrow from input point `from` to point `to`, with given color, width, and style. The tip is of length `l` and inclination `r`.
`circle`	Draw a circle with center `c`, passing from the point `p`, with given color, width, and style.
`marker`	Draw a little mark at point `p`.
`mmline`	Experimental min-max-finding cell
`rectmf`	Manhattan filled rectangle. Fill a rectangular array with a color `col` and a fill pattern `pat`, given two opposite corners of the area, `p` and `q`. Output the other two corners, five middle points, the width and the heigth.
`text`	Print a given string `s` at a given point `p`, with a given color `col`, style (font) `sty`, and point-size `ps` (p. 15).
Invisible Cells	
`textSize`	Given a string, its style (font), its point-size, and one of its (thirteen) "characteristic" points, find the rest of these points. Nothing is drawn.
`floatAscii`	Convert between a number `n` and its ASCII representation `s`
`trans`	Parallel translation of the vector `p0`→`p1` to `pIn`→`pOut`
`offset`	Vector offset `v` relative to point `p` gives the output `q`
`offsetXY`	Offset by <x, y> (in mm) relative to point `p` gives the output `q`
`xy`	`out` is the intersection of the vertical line passing by input point `x` with the horizontal line passing by input point `y`.
`rot90`	Rotate left by 90° input `p1` around `p0` to find `pOut`
`intersection`	`pOut` is the intersection of *line-0* defined by input points `l0p` and `l0q`, with *line-1* defined by `l1p` and `l1q`. When one of the input pairs `p`≡`q`, `pOut` is the *perpendicular projection* of that point on the other line.
`ratio`	the vector `p0`→`pr` is equal to `r` times the vector `p0`→`p1`; `pr`, `r` are I/O
`length`	`p1` is on the line `p`→`q`, at a distance `l` from `p`; `p1`, `l` are I/O
`extremum`	Find the minimum or the maximum of the x's or y's of some points

Schéma N° 2-3. : Tableau récapitulatif des primitives de Laby [Kat90b].

Le producteur désire créer une cellule composée représentant un simple carré. Les composantes de cette nouvelle cellule seront quatre primitives visibles "ligne" (schéma N° 2-5. (c)), une primitive invisible "rotation" (schéma N° 2-5. (a)) et une primitive "translation" également invisible (schéma N° 2-5. (b)). La description textuelle de cette nouvelle cellule nous est présentée dans le schéma N° 2-4.

Cette cellule composée possède quatre ports : deux ports d'entrée (p0, p1) et deux ports de sortie (p2, p3). On reconnaît un port d'entrée par la valeur par défaut qui le suit directement. Les deux ports d'entrée représentent les coordonnées des coins inférieurs du carré et les coordonnées des coins supérieurs seront calculées en fonction des deux premiers. On calcule d'abord la valeur du point "p2" grâce à la primitive "rotation de 90 degrés". La valeur du point "p3" est calculée par la primitive "translation". On connecte ensuite ces deux valeurs à leur port de sortie respectif et on trace les quatre lignes pour former un carré. Une présentation graphique de la construction de la cellule composée nous est proposée dans le schéma N° 2-6.

```
define manFig1 ( p0 = < 0 , 0 >, p1 = < 10 , 0 >, p2, p3 ) ;
   use rot : rot90 ( p0=p0 , p1=p1) ;
   use t : trans ( p0=p0 , p1=p1 , pIn=rot.pOut ) ;
   connect p2 = rot.pOut ;
   connect p3 = t.pOut ;
   use line ( p=p0, q=p1 ) ;
   use line ( p=p0, q=p2 ) ;
   use line ( p=p2, q=p3 ) ;
   use line ( p=p1, q=p3 ) ;
enifed ;
```

Schéma N° 2-4. : La description textuelle de la nouvelle cellule.

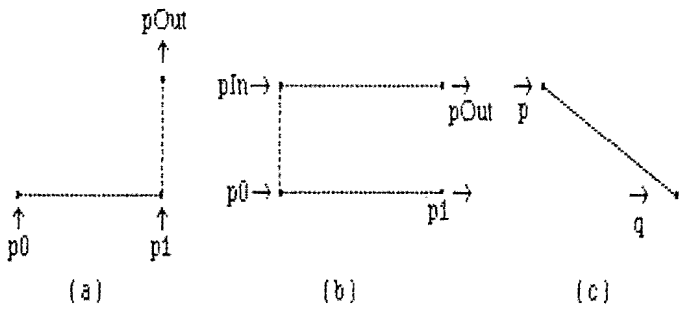


Schéma N° 2-5. : Les trois types de primitives utilisées pour créer un carré.

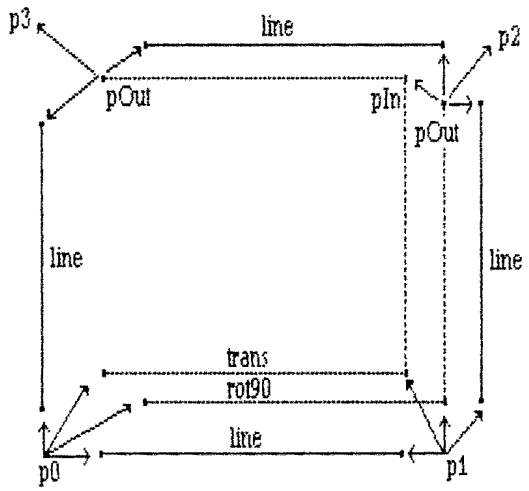


Schéma N° 2-6. : La construction de la cellule composée appelée "carré".

II.2.3.2. Les consommateurs

Ces utilisateurs n'ont généralement que peu (ou pas) d'expérience en programmation. Ils utilisent les classes de cellules définies précédemment pour éditer graphiquement des schémas hiérarchiques, pour déplacer des objets ou changer des valeurs de ports, etc ... . S'ils ont besoin d'une classe de cellules qui n'existent pas encore dans une librairie, il doivent demander aux responsables de la programmation des cellules de la leur fabriquer. Ils peuvent également apprendre à utiliser le langage de description textuel qui se compose d'une quarantaine de commandes (schéma N° 2-2. et 2-3.) facilement assimilables car fort proches de la langue parlée.

Nous reparlerons plus tard de cette relation producteur-consommateur qui est à la base d'un nouveau scénario pour le développement d'applications orientées objets. Comme promis, nous allons décrire minutieusement les différents éléments qui caractérisent le système Labyrinth.

II.3. Description détaillée des caractéristiques du système.

II.3.1. L'organisation hiérarchique

Le système Labyrinth travaille aisément sur de grands graphiques, souvent très complexes, en utilisant une organisation hiérarchique naturelle : de petites composantes sont employées pour construire des **cellules** (objets), et celles-ci constituent des composantes pour créer d'autres cellules de plus grande taille. Par exemple, des morceaux de bois, de verre et de métal sont utilisés pour faire des portes et des fenêtres. Les portes, fenêtres et murs peuvent être employés pour former des appartements, ceux-ci constituent éventuellement des buildings, etc ... .

II.3.2. Les ports

Les cellules sont plus souples lorsqu'elles ont des paramètres dont les valeurs peuvent être changées pour faire des cellules sur mesure. Par exemple, la largeur et la hauteur d'une fenêtre varient en fonction des besoins. Pour ce faire, les cellules de Laby ont des paramètres d'entrées et des paramètres de sortie appelés **ports**. Les ports d'entrées d'une cellule lui fournissent des informations, tandis que ses ports de sorties distribuent les "résultats" de la cellule au reste du monde. Il existe aussi des ports qui ont une polarité mixte (entrée / sortie) : selon les circonstances, ils ont une polarité entrée ou une polarité sortie.

Pour chaque définition de classe de cellule et pour chaque port d'entrée ou port mixte, une **valeur par défaut** doit être spécifiée. Ainsi, tous les ports d'entrées d'une cellule n'ont pas besoin d'être connectés à d'autres ports pour que la cellule puisse fonctionner correctement. Lorsque l'un d'entre eux n'est pas connecté, sa valeur par défaut est prise automatiquement en compte.

Il y a aussi des ports qui sont connectés à des valeurs importantes du système. Etant donné que ces connexions sont un peu spéciales, on les appelle des **connexions d'environnement** et elles sont traitées différemment.

II.3.3. Les primitives

Les cellules de plus bas niveau, qui ne contiennent aucune autre cellule, sont appelées **primitives** (schéma N° 2-3.). Laby travaille sur deux types de primitives, les visibles qui apparaissent à l'écran ou qui s'impriment sur papier (via PostScript [Ado87a] [Ado87b]), opposées aux invisibles qui calculent les valeurs des ports de sorties mais qui n'affichent ou n'impriment rien. Cependant, puisque l'édition graphique joue un rôle central dans Laby et puisque l'objectif principal du système est que l'on puisse éditer la structure de ce que l'on veut concevoir en utilisant des graphiques interactifs, toutes les cellules, mêmes les invisibles, ont un symbole graphique qui peut être affiché à la demande.

II.3.4. Les cellules composées

Les cellules **composées** peuvent être construites en utilisant d'autres cellules primitives ou composées qui ont déjà été définies. Dans les éditeurs graphiques classiques, l'opération de composition est la simple juxtaposition de cellules à des coordonnées géométriques spécifiques. Laby a une opération de composition supplémentaire : la connexion entre les ports de cellules. Pour le système Laby, une connexion signifie l'égalité des valeurs des ports. Dans le cas spécial où les valeurs de ports connectés sont du type point géométrique, leurs égalités impliquent la notion habituelle de connexion entre cellules à la place (point) où ces ports se situent. Si un des ports bouge, les autres qui y sont connectés suivront.

II.3.5. L'évaluateur d'événements

Pour respecter les connexions, c'est-à-dire l'égalité des valeurs des ports, lorsque les valeurs varient, Laby utilise un **évaluateur d'événements**. Un changement de valeur est un événement. Lorsque l'un de ceux-ci se présente à un des port d'entrée d'une cellule, celle-ci a besoin d'être réévaluée. Ces calculs se font jusqu'à ce que tous les événements aient été pris en compte. Notons que la nature événementionnelle de ce traitement garantit que seules les parties du graphe qui ont besoin d'être réévaluées le sont réellement. Les cellules composées sont évaluées récursivement en fonction des composantes qui les forment. Par ce type de traitement, on peut qualifier Laby de système basé sur la satisfaction de contraintes [Bor86]. Il n'est évidemment pas nécessaire de restreindre ce système de propagation de contraintes à des buts purement graphiques.

II.3.6. Exemples de cellules, de ports et de connexions

En plus des unités géométriques, il est possible de définir d'autres sortes de cellules avec leurs propres types de valeur de ports.

Par exemple, des cellules électriques peuvent être des transistors, des diodes, des condensateurs, des ampoules, des générateurs, etc ... . Celles-ci ont un symbole graphique qui les représentent, mais elles ont aussi une signification electrique qui leur est propre. Leurs ports pourraient être leurs bornes électriques (ayant une valeur électrique et une position géométrique), des nombres définissant la force des transistors, et des entrées symboliques définissant leurs technologies (TTL, CMOS, ...).

Des exemples de cellules de type formulaires pourraient être des champs noms, des champs adresses, des champs salaires, des arbres ou des tableaux. Les paramètres peuvent être des chaînes de caractères ou des nombres définissant les contenus des boîtes, des points géométriques

définissant l'emplacement des boîtes à l'intérieur d'un formulaire plus grand, le style de fonte pour afficher ou imprimer le contenu des champs, le nombre de lignes et de colonnes d'un tableau, etc ... .

Un exemple de cellule de composition de texte pourrait être un "paragraphe de texte". Ses ports d'entrées pourraient être un point définissant la marge de gauche, un point définissant la marge de droite, un point de début de texte, l'espace d'interligne, un numéro / symbole définissant le style de fonte, et une chaîne de caractères représentant le texte que l'on désire composer; le port de sortie peut être le point de fin de texte. De tels paragraphes peuvent être mis en cascade les uns derrière les autres pour former un paragraphe plus grand en connectant le point de fin de texte de l'un avec le point de début de texte du suivant.

Lorsqu'on compose des cellules, notons qu'il y a des paramètres d'entrée qui tendent à être communs parmi plusieurs cellules (Ex. Le style de fonte courant, les marges courantes, la largeur courante des lignes, la technologie courante des transistors utilisés, etc ...). Laby les appelle des **variables d'environnements** et comme on l'a déjà dit, il les traite différemment.

II.4. L'architecture du système Labyrinth

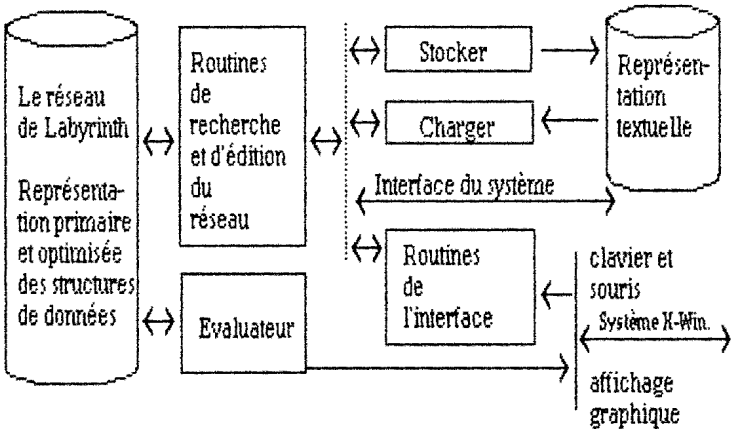


Schéma N° 2-7. : L'architecture du Système Labyrinth.

Le noyau du système est composé de structures de données qui représentent le réseau de Labyrinth, d'un ensemble de routines de recherche et d'édition du réseau, et de l'évaluateur. Le système communique directement via les routines de recherche et d'édition, qui activent automatiquement l'évaluateur. L'utilisateur communique via des routines spéciales de l'interface. Toutes les réactions du système viennent de l'évaluation de cellules graphiques.

Le système travaille donc sur des structures de données qui représentent ce qu'on conçoit (le réseau de Laby). Ces structures de données utilisent couramment des pointeurs pour permettre un accès rapide aux cellules et aux ports de celles-ci.

Tout ce qui a été conçu avec Laby est aussi représenté sous forme textuelle. Ceci donne un moyen pratique de stockage sur disque. Ces fichiers forment ce que l'on appelle les librairies de cellules (les fameuses "cassettes vidéos").

II.5. Les cellules "Text" et "TextSize" : Un exemple concret

Pour clôturer ce second chapitre, nous allons passer à des aspects plus pratiques. J'espère ainsi aboutir à une plus grande compréhension des concepts vus précédemment. Parmi l'ensemble des cellules existantes à l'heure actuelle, j'en ai choisi deux qui me semblent particulièrement représentatives. La première est une primitive visible appelée **Text**. La seconde se nomme **TextSize**, c'est également une primitive mais celle-ci fait partie des invisibles. Mon choix n'est pas vraiment le fruit du hasard car c'est sur ces deux cellules que j'ai travaillé durant mon stage au FORTH. Avant mon arrivée, les textes ne pouvaient être affichés (ou imprimés) qu'en noir, avec un seul style de fonte et une seule taille de caractères. Le calcul de la taille d'un texte ainsi produit était réduit à sa plus simple expression. Pour donner une idée du travail exécuté, nous commencerons simplement par la définition brute de ces deux primitives.

II.5.1. La primitive Text

Actuellement, Laby gère tout ce qui touche au texte à l'aide de deux primitives différentes qui seront probablement fusionnées en une seule dans un avenir relativement proche. La première des deux, Text, affiche simplement une chaîne de caractères ("s") en un point donné ("p") avec le style de fonte "sty", la taille de caractères "ps" et la couleur "col". Le tableau suivant décrit les caractéristiques de cette primitive.

Text			
Imprime une chaîne de caractères à un point donné			
Nom du port	Type du port	Polarité	Fonction du port
p	Point	I	Le point où commence la ligne de base de la chaîne de caractères.
s	String	I	La chaîne de caractères à imprimer.
col	Number	I	La couleur du texte (0 par défaut).
sty	Number	I	Le style de font du texte (0 par défaut).
ps	Number	I	La taille des caractères (0 par défaut).

Schéma N° 2-8. : Table descriptive de la primitive Text.

Avec les cinq ports d'entrées de la primitive Text, on retrouve les trois types de ports que l'on peut utiliser actuellement avec Laby :

- 1) Le type **Nombre** : les nombres à virgule flottante.

  Ex. 10, -1.7, 2e-3, -.55, etc ...

- 2) Le type **String** (ou chaîne de caractères) : toute chaîne de caractères ASCII.

  Ex. "Un texte entouré de quotes", etc ...

- 3) Le type **Point** : des paires d'entiers représentant des coordonnées en millimètres.

  Ex. <-14,22>, <12.5,-.5>, etc ...

Dans la prochaine version (3.x), il y aura plus de types prédéfinis et l'utilisateur sera libre d'élaborer ses propres types. Cette version supportera aussi des listes de valeurs, de cellules et de ports dont la taille pourra varier pendant l'exécution.

Grâce aux trois derniers ports de la primitive, l'utilisateur peut faire varier la présentation de ce qu'il veut afficher (ou imprimer) :

- 1) Le port "**col**" fixe la couleur désirée. On a le choix entre 16 couleurs différentes (0 = noir, 1 = blanc, 2 = rouge, 3 = vert, ..., 15 = jaune or). Si l'utilisateur ne donne aucune valeur à ce port, la couleur noire est prise par défaut.

- 2) Le port "**sty**" définit le style de fonte que l'on désire utiliser. Pour l'instant, il existe cinq types de fonte (0 = le fonte du système, 1 = Times Roman, 2 = Times Italic, 3 = Times Bold, 4 = Times Bold-Italic).

- 3) Et pour finir, on peut choisir la taille des caractères (7 possibilités) via le port "**ps**". La plus petite taille autorisée est huit points et la plus grande est vingt-quatre points.

Avec le petit tableau ci-dessus (schéma N° 2-8.), l'utilisateur connaît tout ce dont il a besoin pour travailler avec cette primitive. Ce qui spécifie donc une cellule, c'est sa fonctionnalité générale et la fonctionnalité de chacun de ses ports.

II.5.2. La primitive TextSize

La seconde primitive qui manipule du texte est invisible et n'affiche (ou n'imprime) donc rien. Elle calcule simplement les coordonnées d'un certain nombre de points représentant la "taille" qu'aurait une chaîne de caractères si on l'affichait(ou l'imprimait). La raison qui fait de TextSize une primitive séparée de Text, est qu'il est parfois nécessaire de l'activer sans imprimer la chaîne de caractères. Observons à présent un exemple qui décrit les différents points importants représentant la "taille" de la chaîne de caractères (schéma N° 2-9.). Ces points sont ensuite mieux détaillés dans le tableau qui suivra (schéma N° 2-10.).

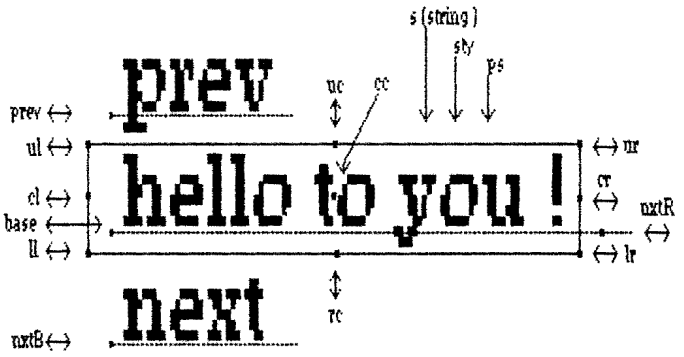


Schéma N° 2-9. : Les 13 coordonnées calculées par la cellule TextSize.

TextSize			
Etant donné un string, un font, une taille de font, et un de ses points caractéristiques; elle trouve la valeur des autres points caractéristiques.			
Nom du port	Type du port	Polarité	Fonction du port
s	String	I	La chaîne de caractères.
sty	Number	I	Le style de font (0 par défaut).
ps	Number	I	La taille des caractères (0 par défaut).
base	Point	I/O	Début de la ligne de base de s, c'est le point qui doit être passé au port p de Text pour l'afficher à cette position.
nxtR	Point	I/O	C'est le point "base" de la chaîne de caractères qui pourrait suivre celle-ci, sans blanc entre elles.
prev	Point	I/O	C'est le point de base de la chaîne de caractères qui pourrait s'afficher au dessus de la courante.
nxtB	Point	I/O	C'est le point de base de la chaîne de caractères qui pourrait s'afficher en dessous de la courante.
ul, ll, ur, lr	Point	I/O	Les quatre coins du plus petit rectangle englobant la chaîne de caractères.
cl, uc, lc, cr	Point	I/O	Les quatre centres des côtés du plus petit rectangle englobant la chaîne de caractères.
cc	Point	I/O	Le centre du plus petit rectangle englobant la chaîne de caractères.
ys	Number	O	La hauteur en mm.
dx	Number	O	La longueur de la chaîne de caractères.
bx	Number	O	La longueur du rectangle englobant.

Schéma N° 2.10. : La table descriptive de la cellule TextSize.

La valeur du port "s" est obligatoire et les fonctionnalités de "sty" et "ps" sont les mêmes que dans la primitive Text (cinq styles de fontes et sept tailles de caractères). Pour les treize ports d'entrées / sorties, il faut que l'un d'entre eux soit un port d'entrée en lui attribuant une valeur. Puis, étant donné la valeur de ce port d'entrée et la taille du plus petit rectangle englobant la chaîne de caractères ("s"), la position des douze autres points est calculée. Ces douze ports deviennent alors des ports de sorties. L'intérêt de la primitive consiste à connecter intelligemment les quinze résultats obtenus (12 coordonnées et 3 longueurs) avec d'autres cellules. Au moindre changement (taille des caractères, style de fonte, position de la ligne de base, etc ...), l'ensemble des points est recalculé et toutes les cellules affectées sont modifiées en conséquence. L'exemple qui va suivre nous aidera à mieux comprendre ce mécanisme.

L'utilisation de cette cellule a l'air un peu rébarbative à première vue mais ce n'est vraiment pas le cas. Parmi les treize ports d'entrées / sorties de la primitive, un seul doit avoir une valeur d'entrée et les autres deviennent instantanément des ports de sorties. De plus, les valeurs par défaut simplifient encore plus l'utilisation comme va nous le montrer le petit exemple de composition de cellules qui suit.

II.5.3. Petit exemple de compositions de cellules

Pour prouver que les deux primitives réalisées durant mon stage fonctionnaient parfaitement bien, j'ai réalisé une petite démonstration. Cet exemple n'a pas vraiment d'utilité pour un système existant mais il a le mérite d'exprimer pleinement les possibilités offertes par les deux nouvelles cellules.

Le principe utilisé sera sûrement appliqué si plus tard on veut concevoir des outils de traitement, de formatage, et d'édition de textes. Il sera également employé avec des outils de CAO classiques ou des logiciels graphiques pour la bureautique.

La démonstration consiste à créer une nouvelle cellule composée que j'ai nommé "TableDeCaractères". Elle emploie cinq primitives différentes pour réaliser son travail (Text, TextSize, Line, Extremum et Ratio). Je vais tout d'abord présenter le "code source" (via le langage de description textuelle) (schéma N° 2-11.) de cette nouvelle cellule puis j'expliquerai le rôle de chaque primitive dans la cellule composée.

```
define TableDeCaractères (ul=<-50,-10>, s11="Je suis centrée.",
s12="Je suis cadrée à gauche.", s21="Je suis aussi cadrée à gauche.",
s22= "Moi, je suis cadrée à droite.", s31="Bonjour",
s32="ABCDEFGHIJKLMNOPQRSTUVWXYZ", lr);

    use ts11 : textSize (s=s11, sty=1, ps=3, ul=ul);
    use ts21 : textSize (s=s21, sty=1, ps=3, base=ts11.nxtB);
    use ts31 : textSize (s=s31, sty=1, ps=3, base=ts21.nxtB);
    use ex1  : extremum (mode="X", in0=ts11.ur, in1=ts11.lr,
               in2=ts21.lr, in3=ts31.lr);

    use ts12 : textSize (s=s21, sty=1, ps=3, ul=ex1.out0);
    use ts22 : textSize (s=s22, sty=1, ps=3, base=ts12.nxtB);
    use ts32 : textSize (s=s32, sty=1, ps=3, base=ts22.nxtB);
    use ex2  : extremum (mode="X", in0=ts12.ur, in1=ts12.lr,
               in2=ts22.lr, in3=ts32.lr);

    use c11  : ration (p0=ul, p1=ex1.out0, r=0.5);
    use ts11c : textSize (s=ts11.s, sty=ts11.sty, ps=ts11.ps, uc=c11.pr);
    use t11  : text (p=ts11c.base, s=ts11c.s, sty=ts11c.sty, ps=ts11c.ps,
               col=0);

    use t21 : text (p=ts11.nxtB, s=ts21.s, sty=ts21.sty, ps=ts21.ps, col=0);
    use t31 : text (p=ts21.nxtB, s=ts31.s, sty=ts31.sty, ps=ts31.ps, col=0);
    use t12 : text (p=ts12.base, s=ts12.s, sty=ts12.sty, ps=ts12.ps, col=0);

    use ts22r : textSize (s=s22, sty=1, ps=3, lr=ex2.out2);
    use t22 : text (p=ts22r.base, s=ts22r.s, sty=ts22r.sty, ps=ts22r.ps,
              col=0);
    use t32  : text (p=ts22.nxtB, s=ts32.s, sty=ts32.sty, ps=ts32.ps, col=0);
    use vert : line (p=ex1.out0, q=ex1.out3);
    use hor1 : line (p=ts11.ll, q=ex2.out1);
    use hor2 : line (p=ts21.ll, q=ex2.out2);
    connect lr = ex2.out3;

  enifed;
```

Schéma N° 2-11. : La description textuelle de la cellule "TableDeCaractères".

Avant d'expliquer clairement les différentes commandes utilisées pour composer la nouvelle cellule, j'aimerais vous présenter le résultat graphique de la cellule TableDeCaractères dans le schéma N° 2-12. .

Je suis centrée.	Je suis cadrée à gauche.
Je suis aussi cadrée à gauche.	Moi, je suis cadrée à droite.
Bonjour	ABCDEFGHIJKLMNOPQRSTUVWXYZ

Schéma N° 2-12. : Résultat graphique de la cellule "TableDeCaractères".

Pour créer soi-même une cellule, il faut d'abord avoir travaillé un minimum avec le système Labyrinth. Ainsi, on se familiarise rapidement avec les commandes et les primitives de Laby. On utilise ensuite un éditeur de texte pour encoder un petit "programme" (une procédure) qui définit **la classe de cellule** que l'on désire. Pour créer son programme, le programmeur a à sa disposition un langage de description textuelle qui est principalement formé par l'union de l'ensemble des commandes de Laby (schéma N° 2-2.) avec celui des primitives (schéma N° 2-3.) et des cellules existantes. Il suffira alors "d'exécuter" la procédure pour analyser les effets de celle-ci (schéma N° 2-12.).

En plus des commandes et cellules courantes de Laby, on utilise trois commandes supplémentaires. **Define ... enifed** qui délimite la définition de la procédure, **use** qui crée une nouvelle instance de classe de cellules et **connect** pour assembler ces instances (les connexions peuvent être aussi directement réalisées avec la commande "use").

Analysons plus en détail la description textuelle (schéma N° 2-11.) de notre exemple. Comme pour chaque création de classe, on commence par la commande "define" suivie du nom de la classe de cellule et de la liste des noms de ports. Cette liste est entourée de parenthèses et les noms sont séparés par des virgules. Certains d'entre eux sont suivis d'un signe égal et d'une constante; ce sont des ports d'entrées et les constantes sont des valeurs par défaut. Le type des ports est déduit du type des constantes. Par défaut, les autres ports sont des ports de sorties car il n'est pas encore permis d'utiliser des ports d'entrées / sorties avec des cellules composées.

Nous définissons donc une cellule appelée "TableDeCaractères" qui a sept ports d'entrées et un port de sortie. Parmi les ports d'entrées on donne la position du coin supérieur gauche de la table ("ul") et le contenu des six cases de celle-ci ("s11, ..., s32"). Le résultat calculé est le coin inférieur droit de la table ("lr").

Le corps de la procédure est essentiellement composé de commande "use" qui définissent des instances de classe de primitives. Le mot-clé "use" est suivi d'un ou deux identifiants (lorsqu'il y en a deux, ils sont séparés par "deux points"). S'il y a un seul identifiant après le mot-clé, c'est le nom de la classe dont on veut créer une nouvelle instance. S'il y a deux identifiants, le second est le nom de la classe et le premier un **nom de référence**. Celui-ci est un identifiant textuel de la nouvelle référence et il doit être unique dans la classe que l'on définit. Lorsqu'il y a des parenthèses après la commande "use", elles contiennent une liste de connections séparées par des virgules. Une connection est représentée par un identifiant de gauche, un signe égal et un identifiant de droite. Elle indique que le port identifié par l'élément de gauche doit être connecté au port identifié par l'élément de droite. Celui-ci est appelé la tête de la connection car c'est lui qui fournit la valeur de la connection tout comme dans une assignation classique.

La première instruction ("use ts11 : textSize (s = s11, sty = 1, ps = 3, ul = ul);") calcule les coordonnées encore inconnues du plus petit rectangle englobant la chaîne de caractères "s11" ("Je suis centré.") en fonction du style de fonte, de la taille des caractères et du coin supérieur gauche du rectangle. On associe alors ces résultats à l'identifiant "ts11" pour pouvoir y accéder facilement. Si l'on désire connaître la valeur du point central du rectangle, on y accède par la composition suivante : "ts11.cc".

Les deux instructions suivantes font le même calcul pour les chaînes de caractères "s21" et "s31" ("Je suis aussi cadrée à gauche." et "Bonjour"). Pour résoudre le problème qui nous intéresse, on doit trouver parmi l'ensemble des points calculés, celui ou ceux qui sont les plus à droite. On réalise facilement cette opération grâce à la primitive "extremum" qui trouve le minimum ou le maximum des "x" ou des "y" des coordonnées de plusieurs points. Son premier port d'entrée ("mode" de type string), accepte les valeurs suivantes : "x" pour calculer le minimum des "x", "X" pour le maximum des "x", "y" pour le minimum des "y" et "Y" pour le maximum des "y". Elle a aussi quatre autres ports d'entrées de type "Point" ("in0", ..., "in3"). Et pour terminer, elle a qua-

tre ports de sorties ("out0", ..., "out3") dont les coordonnées sont dérivées de celles des ports d'entrées après avoir remplacé toutes les coordonnées "x" ou "y" par leur valeur maximale ou minimale en fonction du mode choisi. On projette en fait tous les points sur un même axe (x ou y selon les cas) et on regarde celui qui est le plus à droite ou le plus à gauche, le plus haut ou le plus bas. Dans notre cas, on connaît l'abscisse du point le plus à droite parmi l'ensemble des points calculés précédemment.

On effectue exactement la même opération pour les trois autres chaînes de caractères "s12", "s22", "s32" à droite dans le tableau ("Je suis cadrée à gauche.", "Moi, je suis cadrée à droite." et "ABCDEFGHIJKLMNOPQR-STUVWXYZ") mais ici le coin supérieur gauche du premier rectangle englobant "s12" est égal au point le plus haut à droite calculé précédemment.

Après avoir placé les six cases qui vont acceuillir les chaînes de caractères, il faut placer ces dernières convenablement dans les cases en fonction de leurs caractéristiques (centrée, cadrée à gauche ou à droite). On commence par "s11" qui doit être centrée. On utilise pour cela une primitive appelée "ration" qui possède deux ports d'entrées ("p0" et "p1") et deux ports d'entrées / sorties ("pr" et "r"). Lorsque "r" est choisi comme port d'entrée, la primitive calcule "pr" tel que le vecteur "p0 -> pr" est égal à "r" fois le vecteur "p0 -> p1". Si "pr" est le port d'entrée, elle calcule "r" qui implique la même propriété. Comme on a choisi "r" égal à 0,5, on a trouvé le milieu ("pr") du côté supérieur du rectangle englobant "Je suis centrée.".

Il ne reste plus qu'à trouver la ligne de base où l'on écrira la chaîne de caractères grâce au "textSize ts11c" dont le centre supérieur ("uc") est égal à "pr" déjà calculé. On a alors tout ce qu'il faut pour écrire la phrase au bon endroit avec la primitive "text t11". On affiche également "s21", "s31", "s12" aux coordonnées déjà trouvées.

Il faut encore calculer la position exacte où l'on va afficher la chaîne de caractères "Moi, je suis cadrée à droite" en respectant sa propriété. Pour effectuer cela, on utilise encore un "textSize" qui connaît le point le plus à droite de tout le dessin en le connectant avec les résultats du second "extremum ex2". Il découvre la ligne de base où j'écris la chaîne de caractères ("t22"). On achève le traitement des strings par "s32" qui s'affiche simplement sans calcul supplémentaire.

Pour améliorer la présentation et bien comprendre les différentes propriétés des chaînes de caractères, on trace judicieusement trois lignes de séparation. Les points étant connus, il suffit de les connecter avec trois primitives "line".

Pour clôturer la procédure, il faut encore connecter le port de sortie de la nouvelle classe de cellules composée. Cette opération est réalisée par une commande "connect" entre "lr" le port de sortie de la cellule composée et la coordonnée représentant le coin inférieur droit.

La fin de la procédure est signalée par la seconde partie de la commande de définition que l'on écrit "enifed" ("define" écrit à l'envers). Une fois terminée, on peut regarder les résultats en faisant exécuter la nouvelle procédure (schéma N° 2-12.). L'utilisateur peut ainsi faire varier interactivement des valeurs de ports d'entrées pour voir l'évaluateur d'événements transformer automatiquement le dessin afin de garder les propriétés initiales. Parmi les changements habituels on retrouve la modification des chaînes de caractères, des styles de fontes utilisés, de la taille des caractères, du coin supérieur gauche, etc ... . Le système Labyrinth réagit très rapidement aux différentes modifications, c'est principalement dû à la représentation interne des données qui a été parfaitement imaginée et implémentée.

Croyez-moi, il m'a fallu beaucoup plus de temps pour expliquer aussi clairement que possible le petit exemple ci-dessus que pour l'encoder et le mettre au point. A première vue, ça a l'air difficile à implémenter mais avec un tout petit peu d'expérience, on peut arriver à des résultats étonnants. Coupler la primitive "text" avec "textSize" offre une puissance incroyable de traitement des chaînes de caractères pour tous les outils graphiques liés de près ou de loin à la notion de texte.

II.6. Conclusion

En guise de conclusion, nous devons vous rappeller que le système Labyrinth est toujours en cours de développement. A l'heure actuelle il ne possède pas toutes les fonctionnalités qui lui sont destinées. Parmi celles-ci, nous retrouverons des polarités de ports variables durant l'exécution, le type "liste" de valeurs, de ports ou de cellules, les variables d'environnement, le multi-fenètrage, "l'undo", des interruptions et une interface encore plus conviviale (utilisant les "widgets" de MOTIF). Toutes ces caractéristiques seront bientôt implémentées dans les versions futures de Laby.

Un autre problème est en passe d'être résolu. Il s'agit de trouver une représentation graphique qui permettra d'éditer facilement les cellules invisibles. Le problème ne se pose pas pour les cellules visibles qui se représentent elles-mêmes via leurs aspects. Mais si l'utilisateur veut éditer graphiquement des cellules composées, il faut que les cellules invisibles aient également une représentation graphique. Il n'est vraiment pas facile de trouver un système de représentation qui puisse être adapté à tous les types de cellules invisibles. Mais nous faisons confiance aux chercheurs du FORTH (FOndation of Research and Technology - Hellas) qui trouveront rapidement une solution efficace.

Une fois le système terminé, il faudra encore développer des ensembles de cellules formant les librairies. On attend avec impatiance le contenu de la librairie "bureautique" [Mey91] [Nie90]. On retrouvera certainement des cellules qui représenteront les concepts classiques de la bureautique (Ex. : Tables, hiérarchies, formulaires, classeurs, documents, calculatrice, équations, etc ...). Comme nous le verrons, la petite étude de cas que nous vous présenterons dans le chapitre suivant pourrait être implémentée grâce à cette librairie.

Chapitre III : Etude de cas : Support à la Spécification d'un Système d'Aide à la Décision Quantitative

III.1. Introduction

Après la synthèse de la littérature liée à la programmation visuelle et l'analyse d'un outil graphique d'implémentation, nous terminons la revue des principaux aspects de la programmation visuelle par une petite étude de cas. Nous nous écarterons donc encore un peu plus des aspects théoriques pour affronter la réalité de la pratique.

Jusqu'à présent, nous avons surtout utilisé les différentes techniques visuelles comme support au développement de programmes en nous concentrant essentiellement sur les dernières phases du cycle de vie d'un logiciel (la conception, l'encodage et les tests). Mais nous sommes également persuadés que l'on pourra rapidement profiter des avantages des représentations graphiques dans les premières étapes de conception d'un programme (l'analyse, la structuration et la formulation d'un problème). Celles-ci (les premières étapes) s'appliquant à des domaines fort vastes, nous aborderons principalement l'analyse des problèmes résolus habituellement par des Systèmes Interactifs d'Aides à la prise de Décision (S.I.A.D.). Pour être encore plus précis, seuls les S.I.A.D. quantitatifs liés à la gestion financière seront pris en compte. Nous verrons d'ailleurs dans la première partie du chapitre (III.2.) que la résolution des problèmes quantitatifs de gestion se prête très bien à un support graphique.

Précisons l'objectif poursuivi en introduisant une aide graphique efficace dans un S.I.A.D.Q. . Il s'agit en fait de simplifier profondément le maniement de ce type d'outil. Les utilisateurs pourront alors participer activement à l'entièreté du développement des outils qu'ils désirent, sans avoir pour autant une expérience en programmation. Ils seront enfin indépendants face à tout effort de programmation (formation, encodage, tests, etc ...). Les professionnels de

l'informatique ne travailleront pas moins pour la cause car il faudra quand même préparer les S.I.A.D. aux domaines auxquels on les destine. Nous présenterons plus en détail le rôle de chacun des acteurs (professionnel de l'informatique et utilisateur novice) dans la seconde partie de ce chapitre (III.3.).

Comme promis, nous terminerons cet exposé par la réalisation d'une petite étude de cas reprenant les concepts introduits dans ce même chapitre. L'exemple choisi est un problème classique de gestion budgétaire. Une entreprise désire faire un investissement de capital mais plusieurs possibilités s'offrent à elle. Pour réaliser le bon choix parmi l'ensemble des investissements proposés, il faut faire l'évaluation de la rentabilité de chacun d'eux. Généralement, on utilise un tableur pour modéliser ce type de problème mais le décideur doit alors faire appel à un professionnel de l'informatique ou apprendre lui-même à programmer un tel système. Même s'il est un utilisateur bien entraîné, une étude de [Bro87] montre que dans 44% des cas, l'utilisateur fait au moins une erreur de programmation lorsqu'il crée un programme avec un tableur. Les chiffres sont encore plus catastrophiques lorsqu'il doit modifier un programmme existant. On imagine aussi la perte de temps encourue si le décideur formule, encode et teste lui-même le programme avec un tableur classique. C'est pourquoi nous proposons un outil graphique convivial avec lequel les utilisateurs spécifieront graphiquement le modèle d'évaluation d'investissement qu'ils utilisent couramment. Ils participeront alors au développement des outils qui leur sont nécessaires sans devoir apprendre à programmer. Cette participation augmentera également la qualité des outils ainsi créés.

Par manque de place et de temps, nous nous limiterons aux aspects significatifs du problème car le but ici n'est pas de décrire dans les moindres détails un système exécutable mais de prouver qu'une technique graphique est un support efficace à la spécification d'un S.I.A.D. quantitatif (S.I.A.D.Q.).

Expliquons tout d'abord les raisons qui nous ont poussés à choisir les S.I.A.D. quantitatifs comme cobaye pour notre étude de cas.

III.2. La nécessité d'un support graphique lors de la modélisation de problèmes de gestion

III.2.1. Profil de l'utilisateur

Avant de parler de solutions, il serait peut-être intéressant d'analyser brièvement le profil et les besoins des utilisateurs visés. Ils appartiennent à l'ensemble des cadres supérieurs d'une entreprise. Il sont généralement âgés et n'ont que très rarement des connaissances techniques en informatique. Leurs exigences sur les outils qu'ils utilisent peuvent être classées en six grandes catégories :

- 1) Il faut que les outils soient rapides et simples à utiliser en mettant l'accent sur la **convivialité**.

- 2) L'information accessible doit être **à jour**.

- 3) L'information accessible doit être **pertinente**.

- 4) Le système et son contenu doivent pouvoir être aisément **individualisés** car chaque cadre a ses propres habitudes.

- 5) L'utilisateur doit avoir un **accès direct aux banques de données** si le système ne lui donne pas l'information qu'il recherche.

- 6) L'outil doit pouvoir **communiquer** avec le monde extérieur (avec simplicité et intégrité).

Mais à l'heure actuelle, bien peu d'outils arrivent à respecter le cahier des charges inspiré par ces six catégories. Ceux qui s'en rapprochent le plus sont vraisemblablement les systèmes "W" et "1up".

III.2.2. Les S.I.A.D. actuels.

Les systèmes d'aide à la décision les plus répandus n'offrent qu'un support adéquat pour rechercher des informations sur un problème et ensuite l'analyser. Le décideur possède alors une série de renseignements pour le réconforter dans le choix qu'il doit effectuer. Cependant, ce type de S.I.A.D. offre une aide insuffisante lors de la première phase de l'aide à la décision : **la structuration et la formulation du problème** [Pra90]. Les chercheurs dans le domaine des S.I.A.D. se sont donc plus tournés vers l'aide à la résolution de problèmes que vers l'aide à la structuration de ceux-ci. De ce fait, les S.I.A.D. se composent de techniques de recherches d'informations, de modèles de décision prédéfinis et des procédures d'analyses. Mais aucun outil d'aide à la formulation de problèmes n'est vraiment proposé. On impose donc des structures préétablies pour résoudre des problèmes de gestion alors qu'on pourrait découvrir l'organisation mentale du problème telle qu'il est perçu par l'utilisateur [Liv85]. Celui-ci ne peut qu'adapter le modèle qu'il a en tête aux diverses structures imposées par le système. L'utilisateur se trouve alors au pied d'un mur souvent infranchissable surtout s'il ne possède pas la notion de base pour le franchir : la connaissance de la programmation du système pour se libérer des structures préétablies.

Ne serait-il pas plus intelligent d'offrir un S.I.A.D. [Pra90] :

- a) permettant aux gestionnaires de participer personnellement à la structuration des éléments du modèle qu'il désire utiliser;

- b) offrant une forme de communication entre l'utilisateur et le système qui utilise plus les concepts habituellement rencontrés par l'utilisateur;

- c) pouvant être utilisé par des gestionnaires qui connaissent parfaitement le domaine dans lequel ils ont l'habitude de travailler mais qui ont une expérience quasi nulle en informatique;

- d) permettant à l'utilisateur de comparer les différentes décisions possibles dans un environnement graphique interactif ?

En fait, pour développer ce nouveau type d'outil, il faudrait surtout retravailler ce qu'on appellera plus tard la "présentation" de celui-ci (l'interface). Nous proposons pour cela une solution basée sur l'utilisation de représentations graphiques rappelant les objets couramment employés par les gestionnaires. La suite de cette section montrera ce qui nous a poussé à choisir cette solution.

III.2.3. Un peu de psychologie cognitive

On sait déjà que le cerveau humain est relativement limité lorsqu'il doit traiter des problèmes complexes avec beaucoup de variables et beaucoup de relations entre celles-ci. Mais le gestionnaire financier doit justement faire face à ce type de problème. Alors, pour mieux comprendre les relations qui existent entre l'homme et l'ordinateur, faisons un peu de psychologie cognitive.

Des recherches sur le cerveau humain et les images montrent bien l'importance de celles-ci lors de résolutions de problèmes et tout spécialement dans la première phase : la formulation (spécification) du problème (ce qui nous intéresse tout particulièrement) [Pai71] [Kid84] [Mora81]. En effet, la compréhension de la structure d'un problème de taille complexe est directement liée au processus de construction du modèle mental ou interne qu'on perçoit du problème. Une représentation visuelle du problème dans le processeur d'informations informatique tend à stocker plus efficacement la connaissance dans la mémoire à long terme du processeur d'informations humain. Par conséquent, l'utilisateur peut facilement réappliquer son expérience (sa connaissance) lors de situations nouvelles. Les images représentent donc le modèle mental qu'a l'utilisateur du domaine du problème. Si l'on donne alors à l'utilisateur la possibilité de travailler directement sur la représentation visuelle du problème, la puissance des ordinateurs peut être alors appliquée à l'acquisation et la structuration de la connaissance d'une manière apparentée à celle du cerveau humain. De plus, une autre étude [Pra90] suggère l'importance de l'intégration des techniques de manipulations de symboles (représentés par des images) conjointement avec des données quantitatives. C'est la conclusion de cette étude qui nous a orientée vers les S.I.A.D. quantitatifs (S.I.A.D.Q.).

III.2.4. Les S.I.A.D.Q.

Lorsqu'on parle des S.I.A.D.Q., nous pensons directement aux logiciels tableurs qui ont fleuri dans bon nombre d'entreprises du monde entier. Les personnes qui ont imaginé ce type de logiciels suivaient déjà les mêmes objectifs que nous : apporter à un plus grand nombre de personnes les services de l'informatique sans pour autant devoir perdre trop de temps en formation. Le succès de ce type de logiciel prouve qu'ils sont arrivés près du but escompté mais on peut encore faire mieux en améliorant la représentation graphique utilisée.

Un tableur est constitué de lignes et de colonnes de cellules qui peuvent contenir des chaînes de caractères, des nombres, des formules ou des fonctions. Les utilisateurs travaillent ainsi directement sur la représentation visuelle du tableur. Malheureusement, la véritable structure du modèle en terme d'éléments et de relations entre ces éléments n'est pas vraiment bien représentée par un format style tableau. Pour que l'utilisateur puisse manipuler directement les composantes du modèle, on doit réaliser un système utilisant encore plus des objets graphiques interactifs.

C'est ce que nous découvrirons plus tard mais nous allons tout d'abord revenir aux rôles joués par les deux acteurs principaux du scénario décrivant le nouveau cycle de développement de logiciels.

III.3. Vers un nouveau modèle de développement d'applications

III.3.1. Introduction

La méthode traditionnelle de conception de programmes fonctionne assez bien pour des applications stables sur une longue échéance (Ex. : La gestion des salaires, du personnel, des stocks, etc ...). Mais dans le domaine

qui nous préoccupe (la gestion financière), les spécifications des programmes ne sont pas toujours stables et bien définies car nous ne travaillons plus dans un univers clos. Les programmeurs professionnels sont alors débordés de travail de modification de programme créé par les demandes incessantes des utilisateurs. Ceux-ci n'ont généralement pas la patience d'attendre et les ordinateurs personnels installés sur leur bureau ne servent plus à grand chose car ils ne sont jamais au goût du jour. La solution idéale exigerait des utilisateurs qu'ils apprennent à programmer les outils qu'ils utilisent. Mais croyez-vous que des cadres supérieurs débordés de travail puissent s'offrir le plaisir de suivre une formation sur la programmation des S.I.A.D.Q. classiques ? De plus, après une formation prolongée, la programmation elle-même est une tâche de longue haleine qui se clôture dans près de la moitié des cas par un résultat erroné [Bro87].

Pour offrir aux décideurs les outils qu'ils désirent sans pour cela perdre de leur temps si précieux, il faudrait développer des S.I.A.D.Q. radicalement différents basés sur de nouveaux concepts. Il faudra même revoir la méthode traditionnelle de conception de programmes en fonction des nouvelles exigences des utilisateurs.

III.3.2. La réutilisation

Nous partons d'une supposition que nous estimons fausse : Chaque application est unique. Cette supposition est la cause principale qui justifie la création des anciennes méthodes de développement de logiciels. En fait, un grand nombre d'applications superficiellement différentes exécutent des fonctions fort identiques. Des personnes qui avaient déjà compris cet état de fait, ont alors imaginé d'enfermer ces fonctionnalités communes dans des entités sémantiques de haut niveau : les classes d'objets [Gib90] [Tsy89] [Nie90].

Une classe d'objet peut être définie comme une abstraction d'un éléments du domaine tel qu'il est perçu par le programmeur. Chaque classe d'objet peut être remodelée selon les besoins des programmeurs via des paramètres. Une classe ressemble à un moule qui fabrique des objets de même type. Elle dissimule sous une interface tous les aspects techniques qui pourraient gêner le programmeur et elle offre entre ses semblables une composition intéressante via l'héritage. L'ensemble de ces propriétés permet de partager des parties de code source entre plusieurs applications faisant partie du même domaine. Cette idée a donné naissance à la programmation orientée-objet.

La réutilisation de classes d'objets n'affecte pas vraiment le scénario de développement d'applications mais elle produit des systèmes plus flexibles, plus robustes, facilement modifiables et extrêmement modulaires. Mais cette nouvelle technique n'est pas d'un grand secours pour nos gestionnaires financiers qui désirent mettre la main à la pâte sans perdre trop de temps. Nous possédons une bonne base de travail mais il faudrait encore un peu plus l'étendre. Au lieu de se partager des parties de programmes modelables (des éléments de base du problème), pourquoi ne pas se partager des structures entières d'applications [Wir90] que l'on pourrait adapter facilement ? Ces Structures Génériques d'Applications (ou S.G.A.) [Nie90] guideront l'utilisateur depuis la spécification jusqu'à l'implémentation du problème à informatiser. Ce type de solution intéresse évidemment nos gestionnaires qui n'en demandaient pas autant. Mais entre la théorie et la pratique, il existe un gouffre que l'on va devoir franchir.

III.3.3. Les nouveaux rôles des acteurs

Comme nous l'avons déjà souligné, deux types d'acteurs jouent des rôles différents dans le nouveau scénario de développement de logiciel. Ce nouveau modèle de développement a été introduit par le projet ITHACA [1] [Pro89]. Les rapports qui lient ces deux types d'acteurs sont semblables à ceux qui lient les clients à leurs fournisseurs.

III.3.3.1. Le programmeur professionnel ou concepteur d'application (C.A.)

C'est lui qui joue le rôle du fournisseur dans le scénario. Sa marchandise ira directement remplir les magasins de l'utilisateur non expérimenté. Ses stocks sont composés de toutes sortes de Structures Génériques d'Applications (S.G.A.) qu'il répartit selon le domaine auquel les structures se rapportent. Il est temps d'analyser plus en détail cette mystérieuse marchandise produite par ce fournisseur.

Elle est composée de trois sous-produits [Bec91] :

- a) Le squelette : l'ensemble des classes pertinentes lié au domaine que l'on désire traiter ainsi que les relations qui existent entre ces éléments.

- b) Le guide : un système d'aide qui guide le développeur lors de la spécification d'une application. L'aide se porte surtout sur le choix des classes nécessaires et sur la connection de celles-ci.

[1] ITHACA (an Integrated Toolkit for Highly Advanced Computer Application) est un projet d'intégration technologique (# 2121) faisant partie de la section bureautique du programme ESPRIT II. Les principaux partenaires sont Nixdorf (Berlin), Bull (France), Datamont (Milan), Tècnics en Automatitzacio d'Oficines (Barcelone), **la FOndation pour la Recherche et la Technologie d'Hellas (Iraklion) et le Centre Informatique de l'Université de Genève.**

- c) La présentation : C'est la partie qui nous intéresse le plus car c'est elle qui permet à des utilisateurs non expérimentés de profiter des S.G.A. . Elle cache entre autres les aspects techniques des deux premières composantes qui pourraient gêner l'esprit du développeur. C'est ici que nous allons utiliser la richesse des représentations visuelles adaptées au monde cognitif de l'utilisateur.

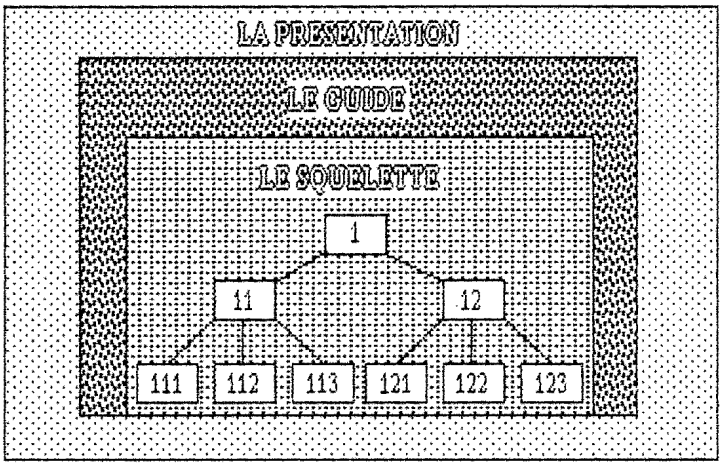


Schéma N° 3-1. : Les 3 composantes d'une structure générique d'application.

Le rôle du concepteur d'application peut être résumé par la phrase suivante : Il est chargé de représenter un ensemble de spécifications de classes relatives à un domaine et son mode d'emploi par des images rappelant le monde cognitif de ce domaine. Cette tâche n'est pas des plus simples. La principale difficulté réside dans la découverte des invariants liés à un domaine. La qualité des structures d'applications étant proportionnelle à la qualité des invariants, on comprend l'importance du travail réalisé par le concepteur d'application.

III.3.3.2. L'utilisateur non expérimenté ou développeur d'application (D.A.)

C'est lui qui joue le rôle du client en se fournissant directement dans le stock du concepteur d'application. Il "achète" ainsi les S.G.A. qui se rapportent à son domaine de travail pour développer à l'aide du mode d'emploi les applications dont il a besoin. Ce nouveau processus de développement peut s'effectuer en six étapes distinctes :

- 1) *Choix de la (des) structure(s) générique(s)* en se basant sur le domaine et les fonctionnalités estimées de la future application.

- 2) *Choix des classes* qui s'avéreront réellement utiles parmi celles qui sont proposées par le S.G.A. . Ce choix se fait à l'aide du guide du S.G.A. .

- 3) *Adapter le comportement des classes* en les instanciant via les paramètres de la classe ou, si nécessaire, via l'héritage.

- 4) *Spécifier l'application* recherchée en manipulant directement les représentations visuelles des classes choisies et remodelées.

- 5) Tester l'application en observant son comportement global.

- 6) Faire éventuellement modifier les composantes des S.G.A. en fonction de l'évolution du domaine car la fabrication de S.G.A. est un processus itératif.

Comme vous l'avez déjà remarqué, la représentation des concepts du domaine joue un rôle capital dans l'exécution des six étapes ci-dessus. L'ensemble de ces représentations et la manière de les manipuler forment ce qu'on a appelé la "présentation" des structures génériques d'application.

III.4. L'importance de la "présentation" dans une structure générique d'application

Nous rentrons dans le vif du sujet. C'est ici que les techniques graphiques vont supporter l'utilisateur dans la spécification du modèle solutionnant les problèmes auxquels il doit faire face. L'aide ainsi offerte est de plusieurs types :

- 1) Tout d'abord, la présentation d'une structure générique d'applications **cache** les aspects techniques du squelette et du guide. L'utilisateur se retrouve alors face à un monde qui ne lui est pas inconnu et il se sent parfaitement à l'aise pour interpréter les concepts qui lui sont proposés.

- 2) Ce sentiment est amplifié car l'utilisation d'unités visuelles pour représenter les classes d'objets et les dépendances entre ces classes, **réduit** considérablement **la complexité** de la procédure de spécification.

- 3) Il y a également un troisième facteur à prendre en compte. La présentation apporte une **communication plus naturelle** entre le système et son utilisateur. Par un choix judicieux des représentations graphiques, on peut reproduire un environnement de travail semblable à celui que le cadre rencontre tous les jours.

- 4) Une présentation apporte en plus une **documentation** naturelle sur les éléments que l'on désire utiliser. Lorsque je vois concrètement un concept, je reconnais les opérations qui, logiquement, sont réalisées sur ce concept. (Ex. Si je vois la représentation d'une horloge, je sais que je peux lire l'heure, que je peux avancer ou reculer l'heure, lire ou modifier la date, etc ...). Pour rassurer encore un peu plus l'utilisateur, on peut associer à tout concept et à toutes parties de ce concept une aide directe pour expliquer le comportement de chacun d'eux.

- 5) Les représentations visuelles autorisent aussi l'utilisation de la **manipulation directe** [Schn83]. Que ce soit avec une souris ou un écran tactile, l'utilisateur manipule naturellement les concepts comme s'il les avait en main.

- 6) Grâce aux représentations visuelles, on a la possibilité de passer aisément **de l'édition à l'exécution** des concepts. L'utilisateur profitera de cet avantage lorsqu'il corrigera les applications qu'il a spécifiées. Le lien entre la formulation et l'exécution ne perturbe absolument pas l'utilisateur. On travaille alors avec la programmation (spécification) par l'exemple (I.2.4.2.). Ce système facilite également l'implémentation automatique des applications à partir de spécifications visuelles.

- 7) La **quantité d'information** communiquée par une représentation graphique est supérieure à celle communiquée par une représentation textuelle. Sur un espace de travail délimité, la communication est donc plus intense.

Ces sept points représentent les principaux supports à la spécification d'une application. La selection des S.G.A. (première étape) et des classes (seconde étape) est ainsi grandement facilitée. Il en ira de même pour les trois étapes suivantes (l'instanciation, la spécification et la vérification). Mais on peut encore améliorer l'aide si l'on choisi une bonne technique visuelle pour manipuler les objets graphiques.

Notre choix s'est porté sur la technique du "scripte visuel" (visual scripting) [Nie90] [Kap89] qui est une extension de sa petite soeur, la technique du "scripte" [Rya90] [You87]. Reprenons tout d'abord quelques définitions extraites de [Kap89] :

- "**La technique du scripte** est une technique de programmation utilisée pour "coller" des composantes logiciels l'une à l'autre (où chaque composante possède un comportement local afin d'obtenir un comportement global : celui de l'application.)."

- "**Le modèle du scripte** détermine quelles sortes de composantes logiciels peuvent être scriptées ainsi que la procédure à suivre pour le faire. Il décrit également l'interprétation des scriptes".

La technique du scripte se combine évidemment bien avec la programmation orientée-objet. Mais on doit encore l'améliorer si l'on veut aider des utilisateurs non expérimentés. Pour l'instant, on voit encore apparaître trop de concepts techniques surtout dans l'interface des composantes. Comme nous l'avons déjà dit, la solution à ce problème consiste à cacher ce qu'on ne veut plus voir par une interface visuelle. La technique étendue s'appelle "scripte visuel". Elle servira de base à la présentation des structures génériques d'applications. Le développeur d'application spécifie alors son application à l'aide d'un scripte visuel dont les composantes graphiques représentent les classes et les relations entre classes des S.G.A. .

III.5. Un exemple de "présentation"

Afin de concrétiser les propos tenus dans ce chapitre, nous utiliserons un exemple tiré de [Bod90]. Nous rappellerons tout d'abord les caractéristiques du cas que nous désirons traiter et nous terminerons par la description de la solution. Celle-ci décrira le dialogue graphique réalisé par le développeur d'application lorsqu'il désire spécifier un S.I.A.D. quantitatif. Ce dialogue est inspiré de [Bec91] et [Pra90]. Il résume parfaitement bien l'aide graphique apportée aux utilisateurs non expérimentés lorsqu'ils communiquent avec leur S.I.A.D. quantitatif.

III.5.1. Enoncé du problème

Prenons la place d'un développeur d'application qui désire réaliser l'évaluation d'un investissement par la méthode du bénéfice actualisé ou NPV (Net Present Value). Pour ce faire, il manipule un certain nombre de paramètres liés au projet dans lequel il aimerait investir [Bod90] :

- la durée de vie de l'investissement : N;
- un échéancier de revenus : $R = (R_0, R_1, ..., R_n)$;
- un échéancier de dépenses : $D = (D_0, D_1, ..., D_n)$;
- un échéancier d'investissements : $I = (I_0, I_1, ..., I_n)$;
- le coût du capital : c.

Le critère de rentabilité qu'il a choisi s'exprime de la façon suivante :

$$NPV = \sum_{j=0}^{N} \frac{(Rj - Dj - Ij)}{(1+c)^j}$$

où le Cash Flow (CF) est l'échéancier ($R_0 - D_0 - I_0, ..., R_n D_n I_n$).

Si la valeur de NPV est supérieure à 0, l'investissement qu'il désire faire dans le projet est considéré comme rentable. Il existe d'autres critères pour évaluer la rentabilité d'un investissement (Le taux d'enrichissement en capital, le taux du rendement interne, la période de récupération, etc ...) mais nous ne les utiliserons pas dans notre exemple.

III.5.2. Choix de la structure générique et des classes d'objets

Comme nous l'avons déjà vu, la première étape du développement du programme (III.3.3.2.) consiste à choisir une structure générique d'application liée au domaine que l'on désire aborder. Grâce à un outil graphique de sélection, le développeur d'application choisit facilement la structure générique qui se rapproche le plus de celle qu'il perçoit. Les représentations graphiques des structures génériques joueront déjà un rôle primordial dans cette première étape. La présentation visuelle qui se rapproche le plus de celle perçue par l'utilisateur est sans aucun doute **la structure arborescente**. L'outil graphique de sélection présentera alors les structures génériques qu'il a stockés sous forme d'arbres. Le schéma N° 3-2. montre la structure générique d'application de la méthode du bénéfice actualisé telle qu'elle est présentée à l'utilisateur.

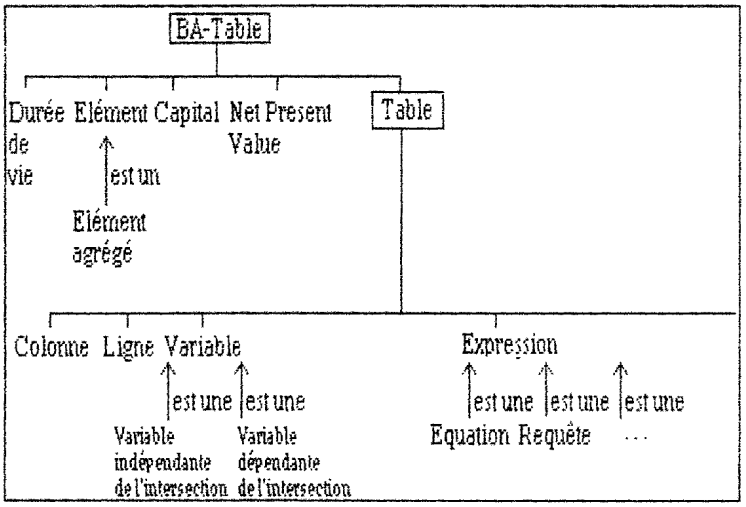


Schéma N° 3-2. : La structure générique d'application du bénéfice actualisé [Bec91].

Ce schéma nous montre les classes d'objets qui composent la structure générique liée au problème auquel on doit faire face. Une classe est représentée par son nom et une structure générique par un rectangle contenant son nom. On voit bien que la structure générique du bénéfice actualisé dépend de

la durée de vie du projet, des éléments organisationnels hiérarchisés ou non liés au projet (dépenses pour toute la société, par département, par pays, etc ...), du capital du projet, du NPV calculé et d'une autre structure générique appelée "Table". Elle représente une vue globale de l'organisation tabulaire qui va acceuillir les éléments du problème.

En comparaison, le schéma N° 3-3. représente la structure générique d'application de la méthode du taux d'enrichissement en capital.

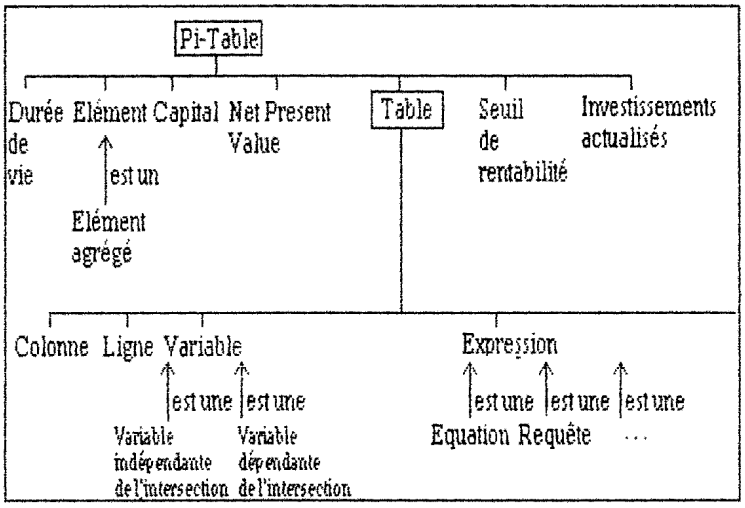


Schéma N° 3-3. : La structure générique d'application du taux d'enrichissement en capital.

Pour aider le développeur d'application à choisir la bonne structure générique, l'outil graphique de sélection offre également deux autres vues des structures génériques :

- 1) En pointant sur le nom d'une classe quelconque de la structure, on peut lire la description de cette classe (les méthodes qu'elle offre). Cette fonctionnalité peut être considérée comme une aide interactive car l'utilisateur peut se renseigner à tout moment sur le comportement local d'une classe. Le schéma N° 3-4. en est un parfait exemple.

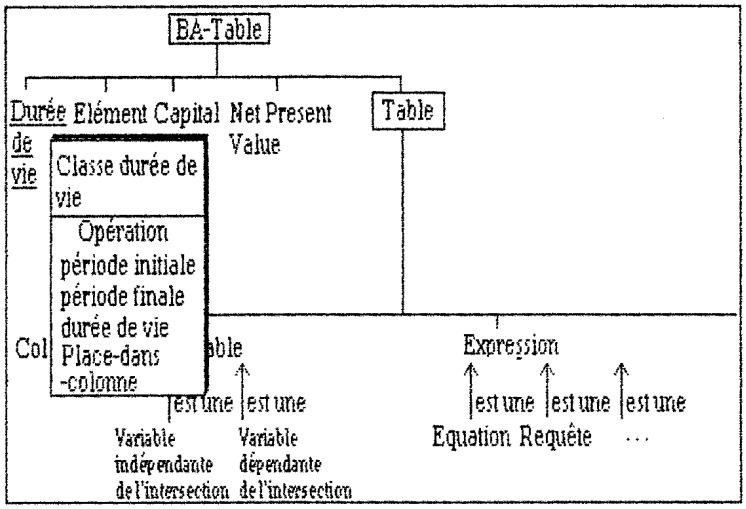


Schéma N° 3-4. : Description de la classe "Durée de vie".

Nous avons dit précédemment que le squelette d'une S.G.A. était composé de classes d'objets et des liens qui existent entre elle. Nous avons choisi d'implémenter ces relations via les méthodes des classes. Le développeur d'application les retrouve facilement dans les descriptions de classes car les noms de ce type de méthode commencent toujours par une majuscule (Ex. Place-dans-colonne). Parce qu'on ne comprend pas forcément une relation par son nom, nous avons ajouté une autre fonctionnalité au système.

- 2) Pour aider l'utilisateur lors du choix de la structure, nous lui proposons une autre vue de celle-ci. Elle représentera toutes les relations qui existent parmi l'ensemble des classes d'objets d'une structure. Le schéma N° 3-5. donne une vue partielle des liens existant dans la structure générique d'application du bénéfice actualisé.

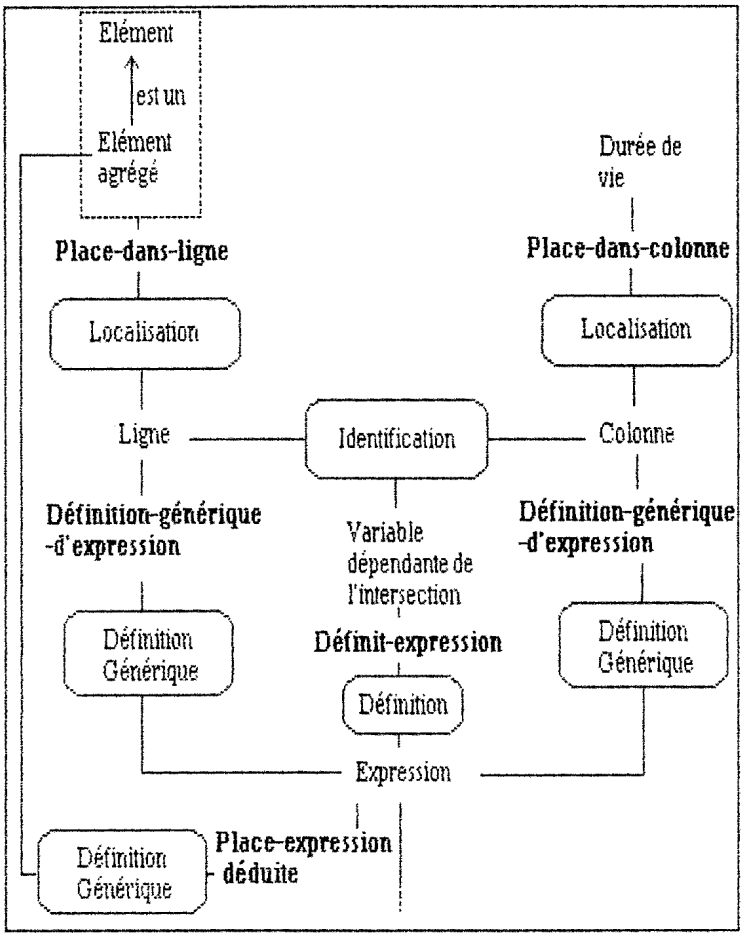


Schéma N° 3-5. : Une vue partielle des relations entre les classes de la structure générique d'application du bénéfice actualisé [Bec91].

Grâce à la vue des liens, l'utilisateur peut voir par exemple que la classe "durée de vie" est localisée par la classe "colonne" via la méthode "Place-dans-colonne". Pour accéder au schéma reprenant les liens entre classes, l'utilisateur a deux possibilités. Soit il demande le schéma complet des liens relatif à une structure (schéma N° 3-5.), soit il demande le schéma partiel des liens relatif à une méthode de la classe (schéma N° 3-6.). Dans le second cas, il pointe la méthode désirée dans la description de la classe et le système lui présente le schéma reprenant les classes affectées par cette méthode. Si je pointe sur la méthode "Place-dans-colonne" dans la description de la classe "Durée de vie" (schéma N° 3-4.), le schéma N° 3-6. apparaît.

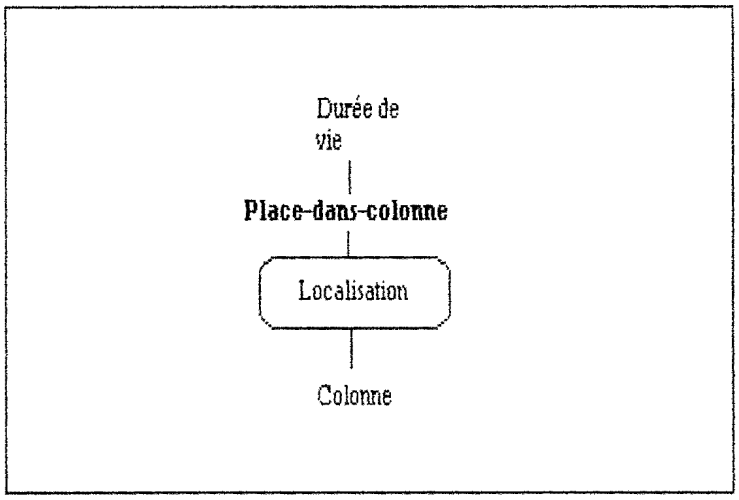


Schéma N° 3-6. : La représentation de la méthode "Place-en-colonne".

Nous avons donc sélectionné graphiquement la S.G.A. qui se rapproche le plus de la méthode de résolution que nous aimerions appliquer. La structure ainsi choisie nous donnant les classes d'objets nécessaires à la résolution du problème, la seconde étape (III.3.3.2.) du nouveau processus de développement d'une application est également achevée.

III.5.3. Spécification du comportement global

Pour réaliser cette opération, nous allons adapter le comportement local des classes d'objets de la structure au cas que nous voulons traiter. L'adaptation est réalisée **par la création d'instances de classes.** Nous utiliserons la manipulation directe pour effectuer ce traitement avec l'aide du guide de la structure.

En premier lieu, le système de sélection de structure générique passe le flambeau à un autre outil graphique qui affiche la représentation d'une instance vide de la "BA-Table" (schéma N° 3-7.). Il sait que c'est sur ce type d'instance que nous allons travailler car nous avons précédemment choisi la structure générique du bénéfice actualisé.

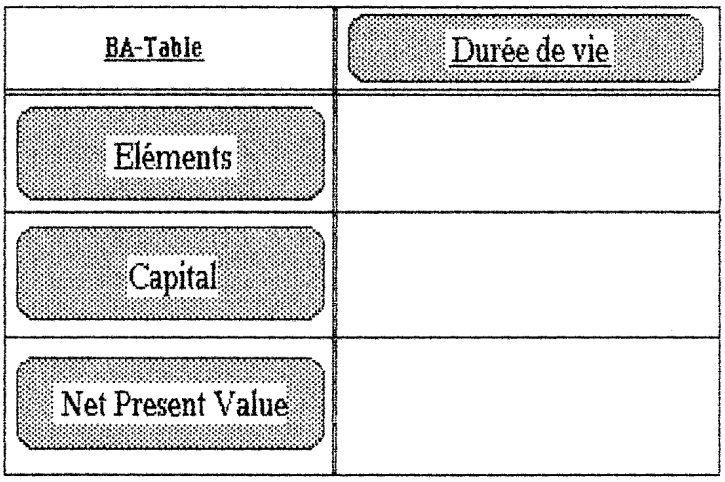


Schéma N° 3-7. : La représentation d'une instance de BA-Table.

Dans cette boîte de dialogue, chaque bouton représente une classe d'objets. Le guide aide l'utilisateur en signalant l'ordre dans lequel le développeur d'application doit activer ces boutons. Dans le schéma N° 3-7., le premier bouton à activer (son nom est souligné) est celui de la classe "Durée de vie". Une fois celui-ci activé, une nouvelle boîte de dialogue s'affiche (schéma N° 3-8.). Elle représente une instance de "Durée de vie" et l'utilisateur doit l'adapter en donnant une valeur aux paramètres de celle-ci.

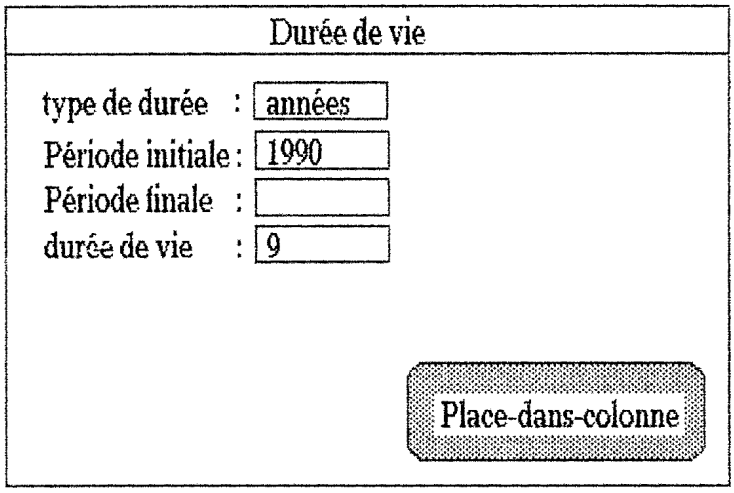


Schéma N° 3-8. : La boîte de dialogue définissant la "Durée de vie".

Le développeur ayant regroupé toutes les données concernant un projet, il a introduit le type de durée (Jours, mois, années , etc ...), la période initiale et la durée de vie du projet en unité de durée. S'il avait donné le type de durée, la période initiale et la période finale, le système aurait calculé la durée de vie.

L'instance de la classe étant définie, l'utilisateur déclenche la méthode "Place-dans-colonne" en activant le bouton portant son nom. Cette méthode transpose l'instance de classe dans la table. Elle crée en fait neuf colonnes avec leur label respectif.

La boîte de dialogue du schéma N° 3-7. réapparaît à nouveau mais c'est le bouton "Eléments" qui est activable. Il peut ainsi créer des instances de classe "Elément" ou "Elément agrégé". Pour que l'utilisateur puisse exprimer facilement les élements (agrégés ou non) du problème, il doit construire interactivement un arbre. L'interaction est basée sur l'utilisation de menus reprenant comme commandes les méthodes des classes. Ces commandes permettent de créer des instances d'éléments et de construire l'arbre représentant la hiérarchie parmi ces instances.

Le développeur d'application a terminé la création de toutes les instances de la classe "Elément" (non agrégé). Il vient de donner la nature de la dernière instance. (schéma N° 3-9.).

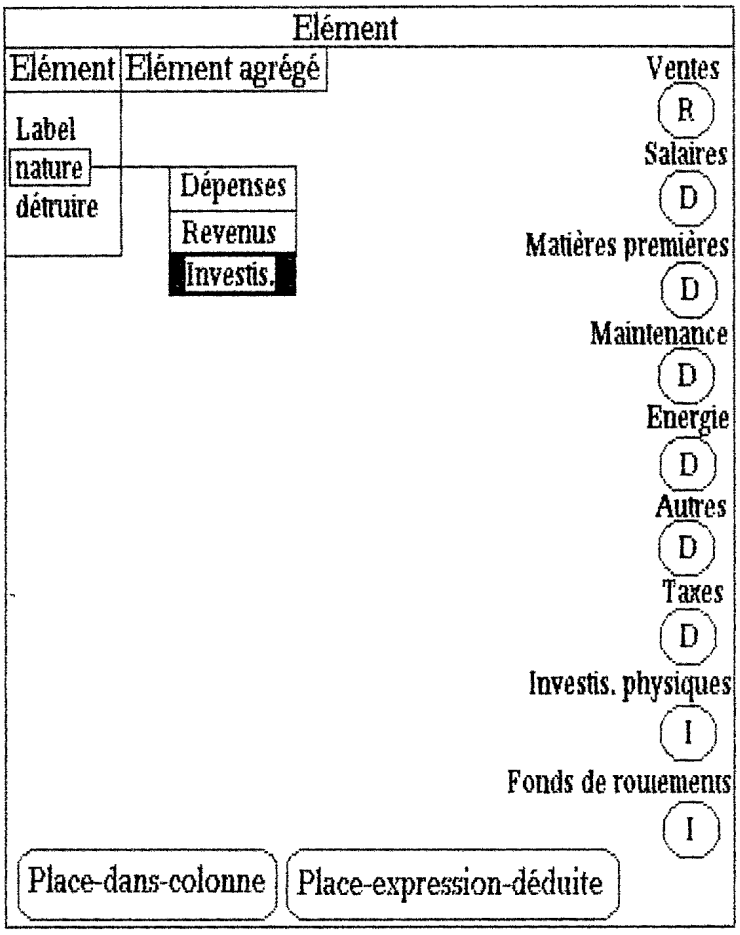


Schéma N° 3-9. : Les instances de la classe "Elément".

Il a donc donné le label et la nature (Investissements, Revenus ou Dépenses) de chaque élément. Il est temps à présent de composer interactivement la hiérarchie entre ces éléments. La nature des instances de la classe "Elément agrégé", est automatiquement déduite de la nature des éléments les composants. La hiérarchie terminée nous est donnée par le schéma N° 3-10.

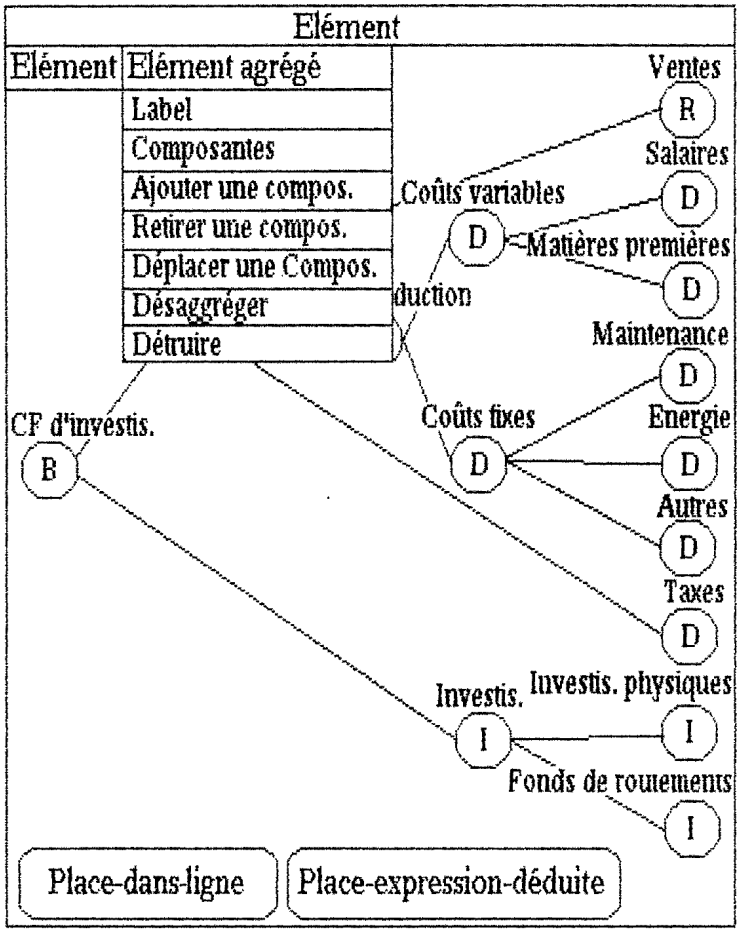


Schéma N° 3-10. : La hiérarchie des éléments de notre cas [Bec91].

Une fois la hiérarchie terminée, il faut placer ses éléments dans les lignes du tableau. Il suffit d'activer le bouton "Place-dans-ligne" pour transposer la représentation arborescente en représentation tabulaire. Pour garder une idée de la hiérarchie, les éléments agrégés sont placés en dessous des éléments qui les composent. Nous avons donc formulé une table avec des colonnes reprenant la notion du temps et des lignes représentant les éléments du problème. Il faut encore placer dans la table les expressions des évaluations déduites des éléments agrégés. La méthode "Place-expression-déduite" réalise cette opération pour le développeur d'application. Une fois ces deux opérations réalisées, la représentation d'une instance de BA-Table (schéma N° 3-7.) s'est automatiquement transformée en représentation tabulaire plus complète (schéma N° 3-11.).

BA-Table	1990	1991	1992	1993	1994	1995	1996	1997	1998	Total
Ventes										
Salaires										
Matières premières										
Coûts variables										
Maintenance										
Energie										
Autres										
Coûts fixes										
Coûts de production										
Résultats avant provisions										
Taxes										
Cash flow d'exploitation										
Investis. physiques										
Fonds de roulements										
Investis.										
CF d'investis.										
Capital										
Net Present Value										

Schéma N° 3-11. : Une instance avancée de la classe BA-Table.

Dans cette présentation, nous ne voyons pas les expressions des évaluations déduites. Pour ce faire, il suffit de pointer l'élément agrégé que l'on désire analyser. Le résultat de cette opération nous est présenté dans le schéma N° 3-12.

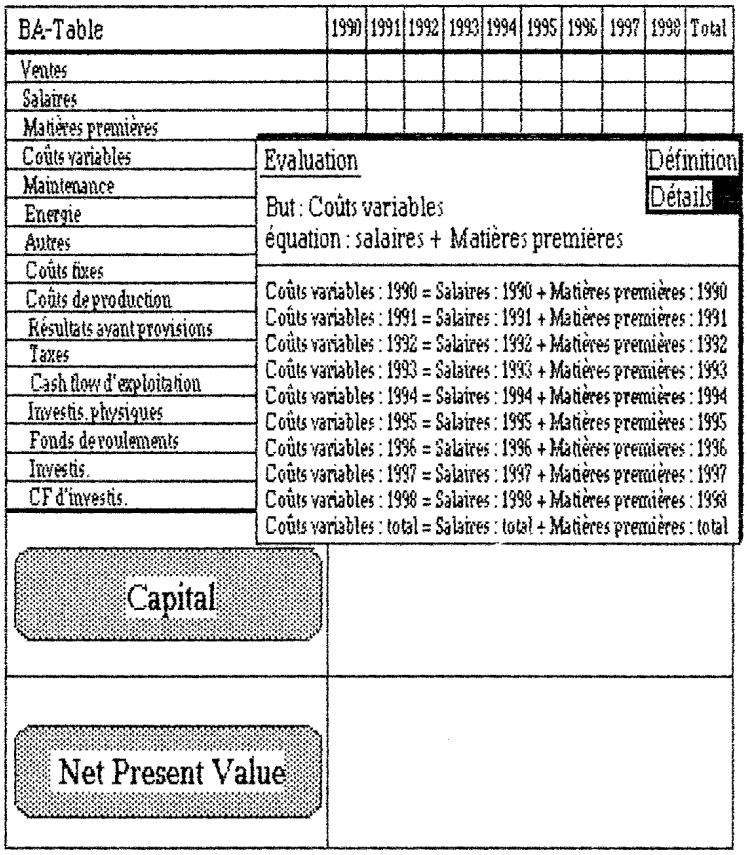


Schéma N° 3-12. : Détail de l'évaluation des coûts variables [Bec91].

Le développeur d'application voit ainsi les résultats de la méthode "Place-expression-déduite" sur l'élément agrégé "Coûts variables". Il a également, par la méthode "création de colonne" associée à la structure générique "Table", créé pour des raisons pratiques une colonne supplémentaire intitulée "total".

Le guide rappelle alors qu'il reste encore deux classes à instancier. Il s'agit de la classe "Capital" et la classe "Net Present Value".

Pour calculer le capital dans un problème de ce type, il existe plusieurs méthodes différentes. Selon leurs complexités, le développeur utilisera soit des expressions pour évaluer le capital, soit une structure générique plus complexe qui le guidera dans sa tâche. Nous ne détaillerons pas cette opération car elle n'apporte aucun concept supplémentaire à notre exposé.

L'utilisateur terminera par la création d'une instance de la classe "Net Present Value". Nous exprimerons cette instance par une évaluation (schéma N° 3-13.).

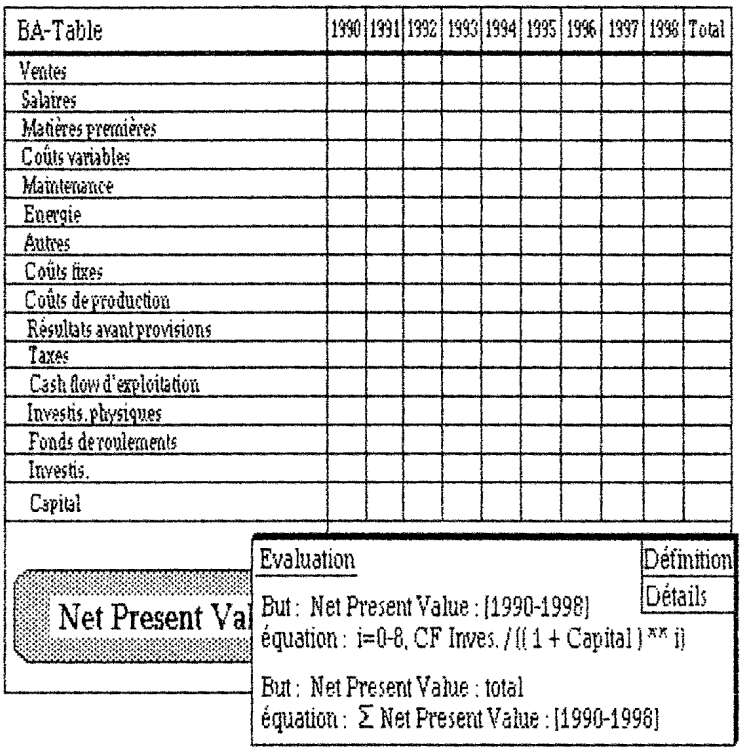


Schéma N° 3-13. : L'expression d'une instance de la classe "Net PresentValue".

On peut dire que la spécification de programme appliquant la méthode du bénéfice actualisé (ou NPV) au projet à analyser est terminée. Nous avons donc réalisé la troisième (adapter le comportement des classes d'objets) et la quatrième étape (spécifier l'application) du nouveau processus de développement de programmes (III.3.3.2.).

III.5.4. L'exécution du programme

L'exécution du nouveau programme va servir de test pour voir s'il réagit comme nous lui avons demandé de réagir. Pour passer à cette étape, le système nous demande de donner des valeurs aux éléments non agrégés. En fonction de la complexité des calculs de ces valeurs, on peut les encoder directement dans le tableau, les rechercher via une requête dans la base de données ou utiliser une structure générique d'application représentant une méthode quelconque de calcul prévisionnelle.

Les résultats de l'exécution du nouveau programme peuvent être présentés dans la strucutre tabulaire ou résumés dans par les graphiques classiques que l'on retrouve couramment en gestion (Bâtonnets, camemberts, etc ...) [Dic86]. Ces graphiques seront produits automatiquement à partir des valeurs des résultats du tableau.

Les utilisateurs ont également la possibilité de stocker les résultats d'une exécution dans des bases de données (Software Information Base [Nie90]) spécialement prévues à cet effet. Cette fonctionnalité est très intéressante quand on sait que les gestionnaires financiers réalisent un certain nombre de simulations avant d'arrêter leur choix. Les résultats de ces simulations étant intelligemment stockés, le système peut présenter graphiquement les résultats globaux de toutes les simulations. L'aide à la prise de décision est donc parfaitement réalisée par le système et l'utilisation de représentations visuelles dans tout le processus de développement y est pour beaucoup.

III.5.5. Faire modifier une structure générique d'application

Comme nous l'avons déjà dit, la création des structures génériques d'application est un processus itératif. Le concepteur d'application ne trouve pas toujours de bons invariants qui soient assez génériques. Le développeur d'application peut alors l'aider dans cette tâche en lui faisant part de l'expérience qu'il vient d'avoir avec une structure. Son avis sera donc très important et il contribuera ainsi à la qualité des structures. D'autres gestionnaires financiers pourront en profiter.

Pour réaliser cette opération, il est important de créer un canal de communication efficace entre le développeur d'application et le concepteur de celle-ci. Les systèmes informatiques et les moyens de communications actuels seront sûrement de bons catalyseurs lors de cette communication.

III.6. Conclusion

J'espère que ce troisième chapitre vous a convaincus de l'efficacité du support offert par des représentations graphiques lors de la spécification d'un programme. Dans la description détaillée du dialogue, nous n'avons pas abordé le difficile problème de l'implémentation. Nous profitons de cette conclusion pour en parler un peu.

L'implémentation d'un tel système ne peut être facilement réalisée qu'avec l'aide d'outils de conception bien adaptés. De plus, on n'atteindra une certaine efficacité que si ces outils se basent sur des concepts fort proches de ceux manipulés par notre S.I.A.D.Q. (arbres, tableaux, équations, etc ...). On pense alors aux S.I.A.D. classiques (tableurs, systèmes de gestion de bases de données, langages de quatrième génération, générateurs de graphiques, etc ...). Mais ces types d'outils n'ont pas vraiment été développés pour combler toutes nos attentes. Alors pourquoi ne pas employer le système Labyrinth dont nous avons dit tant de bien dans le second chapitre ?

Nous savons déjà que le système Labyrinth est un logiciel graphique qui peut en construire d'autres du même type. Il s'adapte parfaitement bien à l'environnement dans lequel on le plonge grâce à ses librairies de cellules. Dans le modèle que nous utilisons, nous retrouvons un certain nombre de concepts de haut niveau (structures génériques d'applications, classes, structures arborescentes, structures tabulaires, équations, etc ...). Il faudra les retrouver d'une façon ou d'une autre dans la librairie de cellules avec laquelle Laby travaillera. Malheureusement, il est encore impossible de créer de telles cellules à l'heure actuelle car les primitives existantes ne concernent que le domaine de la géométrie. Mais il n'est pas interdit d'imaginer ce que pourrait être la solution.

La cellule composée de plus haut niveau sera la **"structure générique d'application"**. Cette cellule possédera un certain nombre de ports qui la caractériseront. Nous y associerons principalement un port "nom" de type chaîne de caractères qui identifiera la structure générique, un port "arbre" qui est du type "cellule composée", un port de connection présenté dans le paragraphe suivant, des ports de sorties pour distribuer des ordres, etc ... . La

valeur du second port est l'arbre représentant **le squelette** de la structure (III.3.3.1.). **Le guide** de celle-ci sera implémenté dans la structure grâce à des cellules invisibles représentant des contraintes (temporelles, événementielles, séquentielles, etc ...). Ces contraines seront constamment maintenues par l'évaluateur d'événements et distribuées par lui via les ports de sorties. Quand à **la présentation** de cette cellule composée, elle dépendra uniquement des effets graphiques de ses composantes. Revenons à présent à la cellule "arbre" qui décrit le squelette d'une structure.

Cette cellule est également une cellule visible composée. Elle possédera un port "composante" de type liste de cellules qui reprendra toutes les composantes formant la structure générique. Celles-ci seront principalement des cellules composées de deux types : le type "classe" que nous présenterons après ou le type "structure générique d'application" que nous avons déjà analysé. Le rôle de l'arbre est de retrouver et présenter la hiérarchie existante entre ses noeuds et ses feuilles. Chaque composante connaissant son père et ses éventuels fils (via ses ports de connections), il suffit de représenter graphiquement (par des lignes) ces liens de parenté pour construire l'arbre.

La seconde cellule composée de haut niveau est la cellule "classe". Cette cellule doit être une abstraction du domaine tel qu'il est perçu par le programmeur. La structure générique étant un peu le mode d'emploi, la classe est la véritable brique de base du modèle. Comme toute cellule, la classe possède des ports. Le port "nom" identifiera la classe et le port "connection" les liens de parenté. Le port "opération" de type "liste de cellules" donne la liste des cellules visibles ou invisibles réalisant le travail des méthodes de la classe. Une méthode bien particulière ne sera ainsi représentée qu'une seule et unique fois par le système. Le principe d'héritage entre classe sera aussi implémenté efficacement car le système connait les relations de parenté entre les composantes ainsi que les méthodes de chacune de celles-ci. L'effet visuel de la cellule "classe" est de deux types selon la demande : L'affichage du nom de la classe ou l'affichage du nom et des noms des méthodes de celle-ci (schéma N° 3-4.).

Une cellule "méthode" peut être une primitive ou une cellule composée. Elle a un port "nom", des ports d'entrées qui fournissent à la cellule les paramètres d'exécution, un port de sorties de type "liste de valeurs" pour distribuer les résultats des éventuels calculs, etc ... . Pour les méthodes représentant un lien entre plusieurs classes, le système peut déduire du port de sorties d'une cellule "méthode" les cellules "classe" qui sont touchées par les effets de celle-ci. La représentation graphique d'une cellule "methode" varient en fonction du type de méthode. La représentation des méthodes simples se fait par l'affichage du nom de celles-ci. Pour les méthodes représentant un lien entre deux classes (leurs noms commencent toujours par une majuscule), on peut les représenter par leurs noms ou par un petit graphe reprenant leurs noms, le nom des classes sur lesquelles elles travaillent et le type de relation (génération, identification, etc ...) qu'elles engendrent (schéma N° 3-6.) C'est la composition parfaite de tous les effets locaux de ces cellules "méthode" qui produira l'effet global escompté. Le compositeur du système est alors l'utilisateur et le chef d'orchestre l'évaluateur d'événements.

La description effectuée ci-dessus est bien sûr incomplète mais elle donne une idée du contenu des cellules utilisées par Laby pour créer notre S.I.A.D.Q.. Pour pouvoir réellement proposer une solution définitive, il faudrait connaître toutes les fonctionnalités de Laby lorsqu'il sera achevé ainsi que toutes les primitives de bases qui seront livrées avec lui.

Conclusion

Je ne reviendrai plus sur le contenu proprement dit des précédents chapitres. Trois petites conclusions s'en sont chargées pour moi (I.4., II.6. et III.6.). J'aimerais plutôt exprimer ce que m'a apporté l'aventure que je viens de vivre.

Personnellement, je crois bien être arrivé à Ithaque [1]. Comme prévu, le chemin fut long, riche en péripéties et en expériences. Nombreux furent les matins d'été, où j'ai pénétré dans des ports vus pour la première fois. J'ai bien fait escale à des comptoirs phéniciens où j'ai acquis de belles marchandises. Et enfin, j'ai visité de nombreuses cités égyptiennes où je me suis instruit avidement auprès de leurs sages.

Le voyage terminé, je peux enfin prendre un peu de recul pour encore mieux l'apprécier. Je pense alors au navigateur qui a guidé mon bateau jusqu'à bon port, aux sages proches d'un palais mythique qui m'ont appris tant de choses et à toutes les personnes qui m'ont aidé à découvrir ce qu'est véritablement Ithaque. Pour ma part, cette complicité, les souvenirs de ports vus pour la première fois et les marchandises acquises, resteront à tout jamais gravés dans ma mémoire. Ma plus grande récompense serait qu'il en soit de même pour les personnes que j'ai croisées durant mon périple.

Seul l'avenir nous l'apprendra ...

[1] Assurez-vous d'avoir bien lu le poème de C. Kavafis qui se trouve tout au début de cet ouvrage.

BIBLIOGRAPHIE

[Ado - Bie]

[Ado87a] "PostScript Language : Tutorial and Cookbook" by Adobe Systems Incorporated; Adisson-Wesley Publishing Company, March 1987.

[Ado87b] "PostScript Language : Reference Manual" by Adobe Systems Incorporated; Adisson-Wesley Publishing Company, August 1987.

[Alb84] "GRASE - A Graphical Syntax Directed Editor for Structured Programming" by Albizuri-Romero M. E.; ACM SIGPLAN Notices, Vol. 19, n° 2, pp. 28-37, February 1984.

[Amb89] "Influence of Visual Technology on the Evolution of Language Environments" by A. L. Ambler and M. M. Burnett; Computer IEEE, October 1989.

[Bae86] "Design Principles for the Enhanced Presentation of Computer Program Source Text" by R. Baecker and A. Marcus; CHI'86 Proceedings, April 1986.

[Bar82] "The Handbook of Artificial Intelligence" by Barr A. and Feigenbaum E. A.; William Kaufmann Inc., Los Altos, California, Vol. N°2., Chap. 10, 1982.

[Bar84] "User Interfaces for Problem Solving Support" by G. R. Barber; in *Human Factors and Interactive Computer Systems,* Ed. Y. Vassiliou, Ablex Publishing Corp., 1984.

[Bec90] "Incremental Reasoning Process through Abstraction levels" by K. Becker, T. Petitjean and F. Bodart; Facultés Universitaires Notre-Dame de la Paix, Namur, 1990.

[Bec91] "Reusable Object-Oriented Specification for Decision Support Systems" by Becker K. and Bodart F. ; Facultés Universitaires Notre-Dame de la Paix, Namur, 1991.

[Bie85] "Automatic Programming : A Tutorial on Formal Methodologies" by Bierman A. W.; Journal of Symbolic Computation, Vol. 1, pp. 119-142, 1985.

[Bie - Bro]

[Bie76] "Constructing Programs from example Computations" by Bierman A. W. and Krischnaswamy; IEEE Transactions on Software Engineering, Vol. se-2, n° 3, pp. 141-153, September 1976.

[Bij84] "Saying what you want with Words and Pictures" by A. Bijl and P. Szalapaj; Human-Computer Interaction - INTERACT '84 (IFIP, 1985).

[Blag87] "Conversing with management information systems in natural language" by R. W. Blanning; CACM, Vol. 27, n° 3, March 1987.

[Blan89] "Using Expert Systems to Construct Formal Specifications" by M. R. Blackburn; IEEE Expert, Spring 1989.

[Böc85] "Making the Invisible Visible : Tools for Explorating Programming" by H. D. Böcker and Nieper H.; Proceedings of the first Pan Pacific Computer Conference, The Australian Computer Society, Melbourne, Australia, September 1985.

[Böc86] "The Enhancement of Understanding through Visual Representations" by H. D. Böcker, G. Fischer and H. Nieper; CHI'86 Proceedings, pp. 44-50, August 1986.

[Bod89] "Conception assistée des systèmes d'informations : Méthode -Modèles - Outils" par F. Bodart and Y. Pigneur; Masson, 1989.

[Bod90] "Evaluation de la rentabilité d'un investissement. Un exemple simplifié de Quantitatif Decison Support System." par F. Bodart, Facultés Universitaires Notre-Dame de la Paix, Namur, novembre 1990.

[Bor86] "Defining Constraints Graphically" by A. Borning; CHI'86 Proceedings, April 1986.

[Broc86] "High-Tech Hieroglyphics Make Programming Child's Play" by Brock R. G.; Data Management, Vol. 24, n° 1, pp. 34-36, 50, January 1986.

[Bro84] "A System for Algorithm Animation" by M. H. Brown; ACM Computer Graphics, Vol. 18, N° 3, pp. 177-186, July 1984.

[Bro85a] "Program Visualization : Graphical Support for Software Development" by G. P. Brown, R.T. Carling, C. F. Herot, D. A. Kramlich and P. Souza; Computer IEEE, August 1985.

[Bro - Cze]

[Bro85b] "Techniques for Algorithm Animation" by M. H. Brown and R. Sedgewick; IEEE Software, Vol. 2, n°1, pp.28-39, January 1985.

[Bro87] "An experimental study of people creating spreadsheets" by Brown P. and Gould P.; ACM TOIS, Vol. 5., n° 3, pp. 258-272, July 1987.

[Car84] "Window-Based Computer Dialogues" by S. K. Card, M. Pavel and J. E.Farrell; Human-Computer Interaction - INTERACT '84 (IFIP, 1985).

[Car85] "Squeak : A Language for Communicating with Mice" by Cardelli L. and Pike R.; Proceedings of ACM SIGGRAPH'85, pp. 199-204, July 1985.

[Cha79] "A Generalized Zooming Technique for Pictorial Database System" by Chang S. K., Lin B. S. and Walser R.; Proceedings of the National Computer Conference, pp. 147-156, June 1979.

[Cha86] "Visual Languages" by S. K. Chang; Ed. Plenum Publishing Corporation, New York, 1986.

[Cha87] "Visual Languages : A tutorial and survey" by S. K. Chang; IEEE Software, Vol. 4, n°1, pp.29-39, January 1987.

[Cha89] "A visual language compiler" by S. K. Chang, M. J. Tauber, and J. Y. Yu; IEEE transaction on Software Engineering, Vol. 15, N° 5, May 1989.

[Coo90] "Full Circle", by R. Cook; Byte, August 1990.

[Cox88] "Using a pictorial representation to combine dataflow and object-orientation in a language independent programming mecanism" by Cox P. T. and Pietrzykowski; Proceedings International Computer Science Conference, pp. 695-704, 1988.

[Cun86] "Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs ? A Comparison of FPL and Pascal" by N. Cunnif, R. P. Taylor and J. B. Black; CHI'86 Proceedings, April 1986.

[Cze90] "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model" by B. Czejdo, R. Elmasri and M. Rusinkiewicz; Computer IEEE, March 1990.

[Dav - Fit]

[Dav82] "Data Flow Program Graphs" by Davis A. L. and Keller R. M.; IEEE Computer, Vol. 15, n° 2, pp. 26-41, February 1982.

[Den80] "A Business Language" by N. J. Denil; IBM J. Res. Develop., Vol. 24, N°6, November 1980.

[Der85] "Information Support Systems for Problem Solving" by D. Dery and T. J. Mock; Decision Support systems, 1, North-Holland, 1985.

[Des89] "Graphical Specification of User Interfaces with Behavior Abstraction" by J. F. Desoi, W. M. Lively and S. V. Sheppard; CHI'89 Proceedings, May 1989.

[Dia80] "Pascal / HSD : a Graphical Programming System" by Diaz-Herrera J. L. and Flude R. C.; Proceedings of IEEE Compsac'80, pp. 723-728, 1980.

[Dic86] "Understanding the Effectiveness of Computer Graphics for Decision Support : A Cumulative Experimental Approach" by G. W. Dickson, G. Desanctis and J. Mcbride; Communications of the ACM, Vol. 29, N° 1, January 1986.

[Ede88] "The Tinkertoy graphical programming environment" by M. Edel; IEEE transaction on Software Engineering, Vol. 14, n° 8, August 1988.

[Er88] "D.S.S. : A Summary, Problems, and Future Trends" by M. C. ER; Decision Support System 4 (1988), North Holland.

[Esp84] "ESPRIT : towards the information society in Europe" in University Computing N° 6, 1984.

[Fik85] "The Role of Frame-Based Representation in Reasoning" by R. Fikes and T. Kehler; Communications of the ACM, Vol. 28, N° 9, September 1985.

[Fin84] "Programming by Rehearseal" by Finzer W and Gould L.; Byte, Vol. 9, n°6, pp.187-210, June 1984.

[Fit79] "When do diagrams make good computer languages ? " by M. Fitter and T. R. G. Green; Int. J. Man-Machine Studies, Vol. 11, 1979.

[For - Har]

[For85] "D. S. S. and Expert systems : A comparaison" by F. N. Ford; Information and Management, North Holland, Elsevier Scientific Publishers, 1985.

[Ger90] "An Approach to Dialog Management for Presentation and Manipulation of Composite Models in D.S.S." by J. Gerlach and F-Y Kuo; Decision Support System 6 (1990), North Holland.

[Gib90] "Class Management for Software Communities" by S. Gibbs, D. Tsichritsis, E. Casois, O. Nierstraz, and X. Pintado; CACM, Vol. 33, N° 9, September 1990.

[Gli84] "PICT : An Interactive Graphical Programming Environment", by E. P. Glinert and S. L. Tanimoto; Computer IEEE, November 1984.

[Gli90a] "Visual Programming Environment : Paradigms and Systems" by E. P. Glinert; Tome I, IEEE Computer Society Press Tutorial, December 1990.

[Gli90b] "Visual Programming Environment : Applications ans Issues" by E. P. Glinert; Tome II, IEEE Computer Society Press Tutorial, December 1990.

[Gol85] "ISIS : Interface for a Semantic Information System" by K. J. Goldman, S. A. Goldman, P. C. Kanellakis and S. B. Zdonik; Proceedings of ACM SIGMOD International Conference on the Management of Data, May 1985.

[Gra85] "Visual Programming : Guest Editors' Introduction", by R. B. Grafton and T. Ichikawa; Computer IEEE, Vol. 18, N° 8, 1985.

[Gys90] "A Graph-Oriented Object Model for Database End-User Interfaces" by M. Gyssens, J. Paredaens and D. Van Gucht; Proceedings of 1990 ACM International Conference on Management of Data.

[Hal84] "Programming by Example" by Halbert D. C.; Ph. D. dissertation, Computer Science Division, University of California, Berkeley, 1984.

[Ham81] "Database Description with SDM : a Semantic Data Model" by Hammer M. et McLeod D.; ACM Transaction on Databases Systems, Vol. 6, n°3, pp. 351-386, September 1981.

[Har88] "On Visual Formalisms" by D. Harel; Communications of the ACM, Vol. 31, N° 5, May 1988.

[Haw - Kar]

[Haw84] "Berlaymont means 'paper mountain' : a view from inside ESPRIT" by J. Hawgood; University Computing N° 6, 1984.

[Her80] "Spatial Management of data" by Herot C. F.; ACM Transactions on Database Systems, Vol. 5., n°4, pp. 493-514, December 1984.

[Her84] "Graphical User Interfaces" by C. F. Herot; in *Human Factors and Interactive Computer Systems,* Ed. Y. Vassiliou, Ablex Publishing Corp., 1984.

[Hir86] "HI-VISUAL : A Language Supporting Visual Interaction in Programming" by Hirakawa and al.; in Visual Languages, Ed. S. K. Chang et al., Plenul Press, pp. 233-259, 1986.

[Ich90] "Iconic Programming : where to go ?" by T. Ichikawa and M. Hirokawa; IEEE Software, November 1990.

[Jac85] "A State Transition Diagram Language for Visual Programming" by R. J. K. Jacob; Computer IEEE, August 1985.

[Jac85b] "An Executable Specification Technique for Describing Human-Computer Interaction" by Jacob R. J. K.; in Advances in Human-Computer Interaction, Vol. 1, Ed. by Rex Hartson, Ablex Publishing Corp., pp. 211-242, 1985.

[Jar88] "Graphics and Managerial Decision Making : Research Based Guidelines" by S. L. Jarvenpaa and G. W. Dickson; Communication of the ACM, Vol. 31, N° 6, June 1988.

[Kan85] "Experiences on user participation in the development of a conceptual schema by using a concept structure interface" by H. Kangassalo and P. Aalto; Human-Computer Interaction - INTERACT '84 (IFIP, 1985).

[Kap89] "An object-based visual scripting environment" by Kappel J., Nierstrasz O., Gibbs S., Junod B., Stadelmann M. and Tsichritzis D.; p. 123-142 in Object oriented development, Geneva, Geneva University, Ed. Tsichritzis, 1989.

[Kar88] "An Icon-Based Design Method for Prolog" by G. M. Karam; IEEE Software, July 1988.

[Kas - Lam]

[Kas85] "Graphical Support for Dialogue Transparency" by J. Kaster and H. Widdel; Human-computer Interaction - INTERACT '84 (IFIP, 1985).

[Kat89a] "The Ithaca Graphical Design Tools" by M. Katevenis, ITHACA.FORTH.89.E31.3b, Interim Report, FOndation of Research and Technology - Hellas, 1989.

[Kat89b] "Laby User's Manual Version 2.6 (Draft of realease 1)" by M. Katevenis, T. Sorilos, C. Georgis and P.Kalogerakis; ITHACA.FORTH.89.E3.1.#4, FOndation of Research and Technology - Hellas, 1989.

[Kat90a] "Laby User's Manual Version 2.8" by M. Katevenis, T. Sorilos, C. Georgis and P.Kalogerakis; ITHACA.FORTH.90.E.3.3.#6, FOndation of Research and Technology - Hellas, May 1990.

[Kat90b] "Laby User's Manual Version 2.10" by M. Katevenis, T. Sorilos, C. Georgis and P.Kalogerakis; ITHACA.FORTH.90.E.3.3.#7, FOndation of Research and Technology - Hellas, December 1990.

[Kee87] "D.S.S. : The Next Decade" by P. G. W. Keen; Decision Support System 3 (1987), North Holland.

[Ker86] "The C Programming Language" by B. W. Kernighan and M. Ritchie; Bell Laboratories, 1986.

[Kid84] "Human Factors Problems in the Design and use of Expert System" by A. Kidd; Fundamentals of Human-Computer Interaction, Academic Press London, 1984.

[Kra88] "Animation of requirements specifications" by J. Kramer and K. NG; Software Practice and Experience, vol. 18(8), August 1988.

[Kra89] "Graphical Configuration Programming" by J. Kramer, J. Magee and Keng Ng; Computer IEEE, October 1989.

[Lam84] "Hints for Computer System Design" by Lampson B. W.; IEEE Software, Vol. 1, n°1, pp. 11-28, January 1984.

[Lef - Mcl]

[Lef79] "A Status Report on the Activities of the CODASYL End User Facilities Committee (EUFC) by Lefkovits H. C. et al.; SIGMOD Record, Vol. 10, n° 2 and 3, August 1979.

[Lei86] "Approaches to End-User Computing : Service May Spell Success" by Leitheiser R. L. and Wetherbe J. C.; Journal of Information Systems Management, pp. 9-14, 1986.

[Lev86] "Visual Programming : A LISP editor that lets you create LISP programs visually." by R. Levien; Byte, February 1986.

[Lia87] "User interface design dor Desicion Support Systems : A self adaptation approach" by T. P. Liany; Information and Management, North Holland, Elsevier Scientific Publishers, 1987.

[Liu89] "Iconic Communication : Image Realism and Meaning" by F-S. Liu and J-W. Tai; IEEE Data Engineering, Vol. 12 n° 4., December 1989.

[Liv85] "On the Modelling of Human-Computer Interaction as the Interface between the user's work Activity and the Information System" by J. Livari and E.Koskela; Human-Computer Interaction - INTERACT '84 (IFIP, 1985).

[Lod83] "Iconic Interfacing" by Lodding K. N.; IEEE Computer Graphics and Applications Vol. 3, n° 2, pp. 11-20, March / April 1983.

[Lon85] "Animating Programs Using Smalltalk" by R. L. London and R. A. Duisberg; Computer IEEE, August 1985.

[Mau89] "Inducing Programs in a Direct-Manipulation Environment" by D. L. Maulsby and I. Witten; CHI'89 Proceedings, May 1989.

[Mcd84] "A Multi Media Approach to the User Interface" by H. C. Mcdonald; in *Human Factors and Interactive Computer Systems,* Ed. Y. Vassiliou, Ablex Publishing Corp., 1984.

[Mcl86] "Show and Tell User's Manual" by McLain P. and Kimura T. D.; Washington Univresity, Technical Report WUCS-86-4, March 1986.

[Mey - Obe]

[Mey91] "VISTA User's Guide" by V. de Mey; ITHACA.CUI.90.E4.#2, University of Geneva, Janvier 1991.

[Mia83] "Program Identation and Comprehensibility" by Miara R. J., Mussselman J. A., Navarro J. A. and Schneiderman B.; Communication of the ACM, Vol. 26, n°11, pp. 861-867, November 1983.

[Mors79] "Some Principles for the Effective Display of Data" by A. Morse; Computer Graphics, 1979.

[Mora81] "An Applied Psychology of the User : Guest Editor's Introduction..." by T.P. Moran; Computing Survey, Vol. 13, N° 1, March 1981.

[Mori85] "Visualizing Program Designs Through PegaSys", by M. Moriconi and F. Hare; Computer IEEE, August 1985.

[Mye83] "INCENSE : A System for Displaying Data Structures" by B. A. Myers; ACM Computer Graphics, Vol. 17, N° 3, pp. 115-125, July 1983.

[Mye86a] "Introduction to Expert Systems" by W. Myers; IEEE Expert, Spring 1986.

[Mye86b] "Visual Programming, Programming by Example, and Program Visualization : A Taxonomy" by B. A. Myers; CHI'86 Proceedings, pp. 59-66, April 1986.

[Mye86c] "Creating Highly-Interactive and Graphical User Interfaces by Demonstration" by Myers B. A. and Buxton W.; SIGGRAPH'86 Proceedings, pp. 249-258, August 1986.

[Nas73] "Flowchart Techniques for Structured Programming" by Nassi I. and Schneiderman B.; ACM SIGPLAN Notices, Vol. 8, n°8, pp. 11-26, August 1973.

[Ng79] "A Graphical Editor for Programming Using Structured Charts" by NG N.; IEEE Compcon'79, pp. 238-243, February 1979.

[Nie90] "Visual Scripting : Towards Interactive Construction of Object-Oriented Applications", by O. Nierstrasz, L. Dami, V. de Mey, M. Stadelmann, D. Tsichritzis, J. Vitek; Université de Genève, May 1990.

[Obe90] "Naturel Selection", by K. K. Obermeier; Byte, August 1990.

[Opp - Rod]

[Opp80] "Prettyprinting" by Oppen D. D.; ACM Transactions on Programming Languages ans Systems, Vol. 2, n°4, pp. 465-483, October 1980.

[Pag83] "A diagramatic notation for abstract syntax and abstract structured objects" by F. G. Pagan; IEEE Transaction on Software Engineering, Vol. se-9, N° 3, May 1983.

[Pon83] "PIGS : A System for Programming with Interactive Graphical Support" by Pong M. C. and Ng N.; Software - Practice and Experience, Vol. 13, n° 9, pp. 847-855, September 1983.

[Pra90] "Model Visualization : Graphical Suppport for DSS Problem Structuring and Knowledge Organization", by W. E. Pracht; Decision Support Systems, 1990.

[Pro89] "ITHACA : an integrated toolkit for highly advanced computer application" by Pröfrock A., Tsichritzis D., Müller G., Ader M.; in Object oriented development, Tsichritzis D. Ed., Geneva, Geneva University, pp. 321-344, 1989.

[Rae84] "Programming in Pictures" by G. Raeder; PH. D. thesis, Departement of Computer Science, University of Southern California, November 1984.

[Rae85] "A Survey of Current Graphical Programming Techniques", by G. Raeder; Computer IEEE, Vol. 18, N° 8, 1985.

[Rei85] "PECAN : Program Development Systems that support Multiple Views" by S. P. Reiss; IEEE Trans. on Software Engineering, Vol. se-11, N° 3, pp. 276-285, March 1985.

[Rei86] "GARDEN Tools : Support for Graphical Programming" by S. P. Reiss; in Advanced Programming Environment, Springer-Verlag, 1986, N.Y.

[Rei87] "Working in the GARDEN environment for conceptual programming" by S. P. Reiss; IEEE Software, Vol. 6, n° 6., pp. 16-27, November 1987.

[Rod79] "Direct Flowgraphs : The basis of a specification and construction method for real-time systems" by J. E. Rodriguez and S.J. Greenspan; The Journal of System and Software, 1, Elsevier North Holland Inc., 1979.

[Rous - Shu]

[Rous85] "Direct Spatial Search on Pictorial Databases Using Packed R-trees" by N. Roussopoulos and D. Leifker; Proceedings of ACM SIGMOD International Conference on the Management of Data, May 1985.

[Rouk90] "ProGraph", by J. Roussak; Personnal Computer World, June 1990.

[Row85] "Fill-in-the-form' Programming" by Rowe L. A.; Proceedings of the VLDB' 85, pp. 394-404, August 1985.

[Rub85] "ThinkPad : A graphical System for Programming by Demonstration" by Rubin R. V., Golin E. J. and Reiss S. P.; IEEE Software, Vol. 2, n° 2, pp. 73-79, March 1985.

[Rya90] "Scripts Unbounded", by B. Ryan; Byte, August 1990.

[Sche88] "X Window System : C Library and Protocol Reference" by R. W. Scheifler, J. Gettys and R. Newman; DIGITAL Press, 1988.

[Schn83] "Direct Manipulation : A Step Beyond Programming Languages" by B. Schneiderman; Computer IEEE, August 1983.

[Shu82] "Specification of forms processing and business procedures for office automation" by N. C. Shu, V. Y. Lam, F. C. Tang, and C. L. Chang; IEEE Transaction on Software Engineering, Vol. se-8, N° 5, September 1982.

[Shu85a] "FORMAL : A Forms-Oriented, Visual-Directed Application Development System", by N.C. Shu, Computer IEEE, August 1985.

[Shu85b] "Visual Programming Languages : A Dimensional Analys" by Shu N. C.; Proceedings of the IEEE International Journal on Policy Analysis and Information System, pp. 326-344, August 1985.

[Shu88a] "Visual Programming", by N.C. Shu; Van Nostrand Reinhold Company, New York, 1988.

[Shu88b] "A visual programming language designed for automatic programming" by Shu N. C.; IEEE Proceedings 21st Hawaii International Conférence on System Science, Vol. 2 : Software Track, pp. 662-671, 1988.

[Shu89] "Visual programming : Perspectives and approaches", by N.C. Shu; IBM Systems Journal, Vol 28 N° 4, 1989.

[Smi - Wan]

[Smi77] "Priciples of Iconic Programming" by Smith D. C.; Published by Birkhauser Verlag, pp. 68-153, 1977.

[Smi82] "Designing the Star User Interface" by Smith D. C., Irby C. and Kimball R.; Byte, Vol. 7, n° 4, pp. 242-282, April 1982.

[Str88] "The C++ Programming Language" by B. Stroustrup; Bell Laboratories, 1988.

[Taz90] "End-User Programming", by J. M. Tazelaar; Byte, August 1990.

[Teit81] "The Interlisp Programming Environment" by Teitelman W. and Masinter L.; IEEE Computer, Vol. 14, n° 4., pp. 25-34, April 1981.

[Teit85] "A Tour through Cedar" by Teitelman W.; IEEE Transactions on Software Engineering, Vol. se-11, n° 3., pp. 285-302, March 1985.

[Tri88] "A Survey of Graphical Notations for Program Design : An Update" by Tripp L. L. ACM SIGSOST software Engineering Notes, Vol. 13, n° 4, pp. 39-44, 1988.

[Tur84] "Using Restricted Natural Language for Data Retrieval : A Plan for Field Evaluation" by J. A. Turner, M. Jarke, E. A. Stohr, Y. Vassiliou and N. White; in *Human Factors and Interactive Computer Systems,* Ed. Y. Vassiliou, Ablex Publishing Corp., 1984.

[Tsi89] "ObjectWorld" by D. Tsichritzis; in Object oriented development, Geneva University, Ed. Tsichritzis, 1989.

[Vas89] "Query Languages - A Taxonomy" by Y. Vassiliou and M. Jarke; in *Human Factors and Interactive Computer Systems,* Ed. Y. Vassiliou, Ablex Publishing Corp., 1984.

[Vis88] "User Interface Design : differing requirements of novice, occasional and expert users" by M. Visvalingam; University Computing, N° 10, 1988.

[Wan73] "The Chinesse Language" by Wang W. S-Y.; Scientific American, Vol. 228, n° 2, pp. 50-63, February 1973.

[Was - Zlo]

[Was85] "Extending State Transition Diagrams for the specification of Human-Computer Interaction" by Wasserman A. I.; IEEE Transactions on Software Engineering, Vol. se-11, n° 8, pp. 699-713, August 1985.

[Was86b] "Building Reliable Interactive Information Systems" by Wasserman A. I., Pircher P. and Shewmake D. T.; IEEE Transactions on Software Engineering, Vol. se-12, n° 1., pp. 147-156, January 1986.

[Was90] "The O. O. Structured Design Notation for Software Design Representation" by A. I. Wasserman, P. A. Pircher and R. J. Muller; Computer IEEE, March 1990.

[Wir90] "Surveying current research in object-oriented design" by Wirfs-Brock R. J. and Johnson R. E.; Communication of the ACM, Vol. 32, n° 9., pp. 104-124, September 1990.

[Yao84] " FORMANAGER : An Office Forms Management System" by Yao S. B., Hevner A. R., Shi Z. and Luo D.; ACM Transactions of Office Information Systems, Vol. 2, n° 3, pp. 235-262, July 1984.

[You87] "The Use of Scenarios in Human-Computer Interaction Research " by R. M. Young and P. Barnard; CHI + GI 1987 Proceedings.

[Zlo80] "A language for office and business automation" by Zloof M. M.; AFIPS Office Automation Conference Digest, pp. 249-260, 1980.