Interfaces homme-machine : évaluation du modèle d'architecture logicielle PAC

Thierry Duval

Interfaces homme-machine : évaluation du modèle d'architecture logicielle PAC

Thierry Duval

Laboratoire d'informatique pour les sciences de l'ingénieur

École Centrale de Nantes

1 rue de la Noë, 44072 Nantes Cedex 04

RÉSUMÉ

L'interface utilisateur d'une application (c'est-à-dire son image externe, ou encore sa présentation) est constituée d'un ensemble d'objets interactifs. Nous essayons ici de décrire ces objets de façon à les construire plus facilement. Pour cela, nous utilisons un modèle d'architecture logicielle : le modèle PAC, et nous étudions comment le fait de structurer les applications et leurs composants sous la forme d'objets PAC peut nous permettre de construire plus facilement les interfaces de dialogue, notamment en réutilisant et en composant des objets PAC existants. Pour ceci, nous présentons des exemples d'objets et d'applications interactifs que nous avons réalisés dans le cadre de l'étude des outils pour la construction des interfaces homme-machine. Nous discutons aussi des nécessités d'effectuer des vérifications d'ordre syntaxique et sémantique lors des manipulations d'objets, et des problèmes de mise en œuvre du modèle PAC.

MOTS CLÉS : interface utilisateur, interface homme-machine, dialogue homme-machine, logiciel interactif, modèle d'architecture logicielle, méthodologie de développement, manipulation directe.

1. Introduction

On appelle « application graphique interactive » une application qui produit des dessins ou des images sur une surface de visualisation, et où les échanges entre l'utilisateur et l'application sont généralement appelés « interactions homme-machine » ou « dialogue homme-machine ». Nous nous intéressons à la partie de l'application oui est responsable de ces échanges, généralement appelée « interface utilisateur », et plus particulièrement à la façon de définir et de construire cette partie. Pour cela, nous prônons l'utilisation d'un modèle d'architecture comme cadre de pensée et cadre de réalisation. Plusieurs modèles ont été proposés : dans cet article, nous en retenons deux, nous les présentons brièvement, nous choisissons l'un d'entre eux, et nous montrons à travers de nombreux exemples une façon de définir et de construire les interfaces utilisateurs de nombreuses applications.

Avant de présenter les différents modèles proposés, nous devons préciser à quoi sert un modèle d'architecture [COU 90]. Le client privilégié est le réalisateur de l'interface utilisateur, le modèle a pour objet de lui fournir une structure générique à partir de laquelle il est possible de construire un système interactif particulier. Ainsi il comportera les descriptions des données échangées entre l'utilisateur et l'application, des étapes de transformation des données, et de l'agencement des composants qui assurent ces transformations [COU 90].

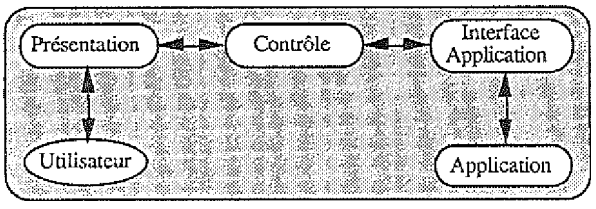

Figure 1 : **Le modèle Seeheim**

Le premier modèle proposé fut celui de Seeheim [PFA 85], proposé initialement par M. Green [GRE 81] [GRE 85], encore appelé *modèle langage* [FOL 82]. Ce modèle divise l'interface utilisateur en trois composants logiques (voir figure 1) :

- un composant *présentation,* responsable de la présentation externe du système, c'est lui qui définit l'image du système interactif, il est responsable de la gestion de l'écran, des modes d'interaction et de la rétroaction immédiate ;
- un composant *interface avec l'application,* représentant l'application du point de vue de l'interface utilisateur (IU) ; il contient les descriptions de toutes les structures de données de l'application et de toutes les procédures accessibles à l'IU ;

- un composant *contrôle du dialogue,* qui joue le rôle de médiateur entre le composant *présentation* et le composant *interface application.*

Il est facile de faire une mauvaise interprétation de ce modèle, en le prenant tel quel, ce qui aboutit généralement à une mise en œuvre trop rigide.

Un autre modèle proposé est le modèle PAC [COU 87][COU 90]. Ce modèle s'inspire du modèle Seeheim, en structurant l'architecture d'une application interactive en 3 composants : *présentation*, *abstraction* et *contrôle*, qui ont les rôles suivants :

- le composant *présentation* définit la syntaxe concrète de l'application, c'est-à-dire le comportement, en entrée comme en sortie, de l'application vis-à-vis de l'utilisateur ;
- le composant *abstraction* désigne la sémantique, c'est-à-dire les attributs et les fonctionnalités de l'application ;
- le composant *contrôle* maintient la cohérence entre le composant *présentation* et le composant *abstraction.*

Ce modèle est basé sur le principe que la frontière entre la notion d'application et la notion d'interface est floue et que par conséquent les notions de syntaxe et de sémantique ne forment pas deux blocs monolithiques mais se répartissent à différents niveaux d'abstraction. On aboutit à une organisation répartie qui gère la composition et l'interaction d'une multitude d'objets PAC.

Le modèle PAC peut être considéré comme une interprétation modulaire, ou encore répartie, du modèle Seeheim.

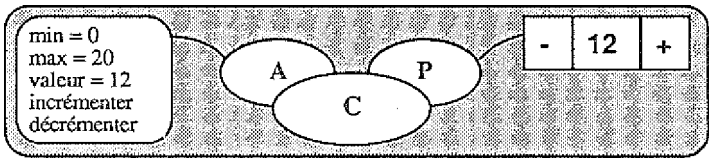

Figure 2 : **Exemple d'objet PAC : un entier « plus-moins » interactif**

Nous proposons ici d'évaluer ce modèle, c'est-à-dire de montrer qu'une interface utilisateur peut être décrite comme un assemblage d'objets interactifs, et que ces objets sont structurés de la façon proposée par le modèle PAC. Parallèlement à cette possibilité de description, nous montrons que l'utilisation systématique d'une telle structuration permet de construire des objets interactifs et donc de construire l'interface utilisateur de nombreuses applications par utilisation, réutilisation et composition de ces objets.

2. Les objets de base

Ce sont les objets que l'on peut rencontrer dans des bibliothèques de composants logiciels ou bien dans des bibliothèques graphiques, ou tout simplement qui peuvent être proposés par des langages de programmation.

2.1. Les objets simples

Nous appellerons « objets simples » les objets qui représentent des types : un objet de type « entier plus-moins », qui peut être représenté par un type intervalle (de 0 à 100, par exemple) et sur lequel il est possible de réaliser les opérations « incrémenter » et « décrémenter ».

2.2. Les objets de visualisation

Nous appellerons « objets de visualisation » les objets dont l'essence même est de produire un résultat visuel, qui peut être la représentation d'un type de données abstrait. Exemple : les dispositifs de sortie de GKS, et parmi eux le tracé de segments (POLYLINE), dont le résultat est la représentation d'une liste de points par un ensemble de segments de droites reliant ces points.

2.3. Les objets d'entrée

Nous appellerons « objets d'entrée » les objets dont l'essence même est de capter des événements en provenance de l'utilisateur, en général à l'aide de dispositifs de pointage (souris, tablettes à digitaliser, crayons optiques...), qu'ils interprètent de façon à produire un objet simple.

Exemple : les dispositifs d'entrée de GKS, et parmi eux le releveur de coordonnées (LOCATOR), qui renvoie un objet de type « point » à partir d'un événement graphique de base.

Remarque. — Les objets d'entrée ont généralement plusieurs modes de fonctionnement (requête, événement, échantillonnage). Ces modes, que nous pourrons retrouver chez des objets d'autres types, n'influeront pas sur leur structure, c'est-à-dire sur la façon de les construire.

2.4. Les objets interactifs simples

Nous appellerons « objets interactifs simples », des objets d'entrée dont la fonction est d'être désignés par l'utilisateur : ils renvoient un message à l'objet qui les utilise, indiquant qu'ils ont été désignés. Exemple : un bouton.

3. Les objets illustrés

Ils peuvent être obtenus en composant des objets simples avec des visualisations appropriées, et en assurant la cohérence entre les objets simples et leur visualisation.

3.1. illustration d'un entier plus-moins

Nous pouvons, par exemple, composer un entier plus-moins avec une représentation de type « chaîne de caractères » (TEXT si l'on se réfère à GKS). Le résultat souhaité est qu'à tout instant la valeur d'un entier plus-moins illustré soit affichée dans une zone quelconque. Pour ce faire, il faut créer une sorte de ciment entre l'entier plus-moins et sa visualisation : chaque appel à la primitive « incrémenter » doit être suivi d'un appel à la primitive « afficher » de la visualisation. Une façon de procéder est de définir un nouvel objet, dont l'interface (au sens primitives exportées) est une réplique de celle de l'objet entier plus-moins, chaque primitive faisant tout d'abord appel à la primitive correspondante de l'objet entier plus-moins, puis ensuite à la primitive « afficher » de la visualisation (cf. figure 3).

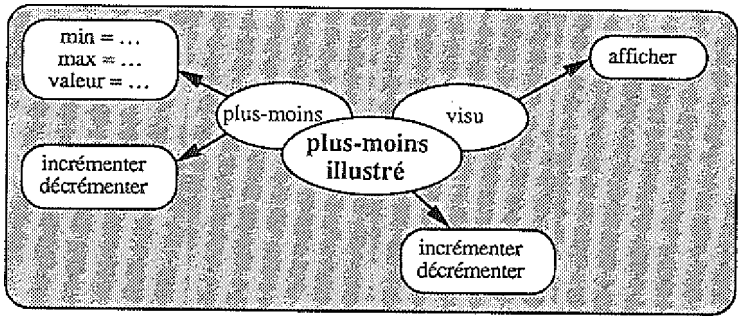

Figure 3 : **Le type entier plus-moins illustré**

On obtient en quelque sorte un objet PAC, dont l'abstraction est constituée par l'entier plus-moins et dont la présentation est constituée par la visualisation.

3.2. Illustration d'un entier

Nous voulons ici disposer d'une illustration du type entier sous la forme d'un rectangle de longueur proportionnelle à sa valeur. Nous allons donc pour cela composer un entier avec un objet de visualisation déplaçable (nous voulons pouvoir déplacer la visualisation de notre entier), cf. figure 4.

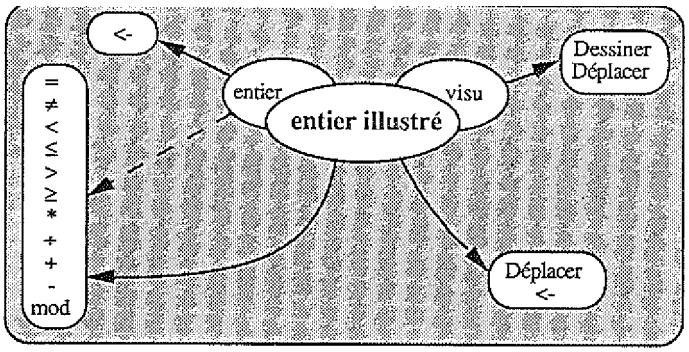

Figure 4: **Le type entier illustré**

Le maintien de la cohérence entre nos deux composants s'effectue de la façon suivante : l'instruction d'affectation du type entier est surchargée, elle consiste maintenant en un appel à l'instruction d'affectation de l'entier, suivi d'un appel à la primitive « dessiner » de la visualisation. Toutes les autres primitives (opérations et comparaisons sur les entiers, dessin de la visualisation) peuvent être réutilisées telles quelles.

3.3. Illustration d'une pile

Nous voulons ici disposer d'une illustration d'une structure de pile d'objets, à condition que ces objets disposent eux aussi d'une illustration et qu'il soit possible de déplacer cette illustration, car nous désirons obtenir des effets visuels au niveau des opérations « empiler » et « dépiler » : les objets manipulés passeront en un point situé sur le dessus de la pile. L'illustration d'entier réalisée précédemment convient parfaitement à ce type de manipulations. Nous allons donc pour cela composer une pile avec un objet de visualisation, cf. figure 5.

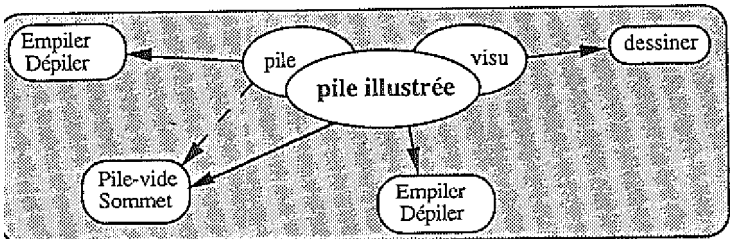

Figure 5 : **Le type pile illustrée**

Le maintien de la cohérence entre nos deux composants s'effectue de la façon suivante : les primitives de manipulation du type pile « empiler » et « dépiler » sont surchargées, elles consistent maintenant en un appel à l'ancienne primitive, en un appel à la primitive « dessiner » de la visualisation, et en une demande de déplacement à l'objet concerné. Les autres primitives, « pile-vide » et « sommet », peuvent être réutilisées telles quelles : elles n'ont pas trait à la représentation graphique de la pile.

3.4. Visualisation des objets d'une application

Supposons maintenant qu'une application manipule des objets simples, et qu'elle puisse disposer par la suite des mêmes objets illustrés. Pour pouvoir utiliser ces objets, il va falloir soit changer « manuellement » les types des objets manipulés, soit utiliser la généricité (si cela est possible). Pour une application résolvant le problème des tours de Hanoï, et qui utilise pour cela des piles d'entiers, il suffit donc d'utiliser les types illustrés que nous venons de décrire pour disposer d'une visualisation des objets de ce type.

4. Les objets interactifs composés

Ils peuvent être obtenus en composant des objets illustrés et des objets d'entrée, ou bien en composant (à un niveau inférieur d'abstraction) des objets simples, des objets de visualisation, et des objets d'entrée (ce second type d'objets interactifs sera étudié avec les objets composés).

4.1. Un entier plus-moins interactif

Si nous désirons maintenant agir directement sur un objet entier plus-moins de façon à modifier sa valeur, à l'aide d'un dispositif de désignation graphique par exemple, nous pouvons composer un objet illustré avec plusieurs objets d'entrée. Nous pouvons ici associer à notre précédent entier plus-moins illustré deux objets interactifs simples : un bouton « + » et un bouton « - ») comme illustré figure 6. Il suffit ensuite d'assurer la cohérence entre ces objets.

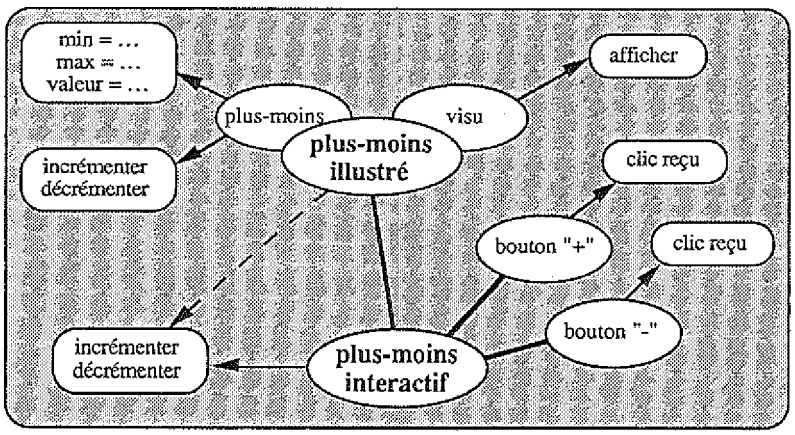

Figure 6: **Le type entier plus-moins interactif**

Au niveau des manipulations logicielles, notre objet interactif propose exactement les deux mêmes primitives que l'objet illustré. Par contre, il doit maintenant tenir compte des événements susceptibles de lui arriver en provenance des deux boutons sur lesquels l'utilisateur peut agir. En fonction de la provenance de l'événement, il n'a plus qu'à faire exécuter la primitive adéquate.
Notons bien qu'ici il n'est pas question d'empêcher un utilisateur d'essayer d'incrémenter un entier ayant déjà une valeur maximale (par exemple) : il faudrait pour ceci demander des renseignements au niveau des données sémantiques de l'entier plus-moins, de façon à répercuter ces informations sur la présentation des dispositifs d'entrée.

4.2. Un objet graphique interactif

Si nous voulons maintenant déplacer un entier illustré en le manipulant de façon directe, nous pouvons ici encore composer un entier illustré avec des dispositifs d'entrée : un sélecteur (PICK) qui signale à l'entier qu'il a été désigné pour une manipulation, et un releveur (LOCATOR) que l'entier consulte afin de savoir les déplacements que l'on veut lui faire subir.

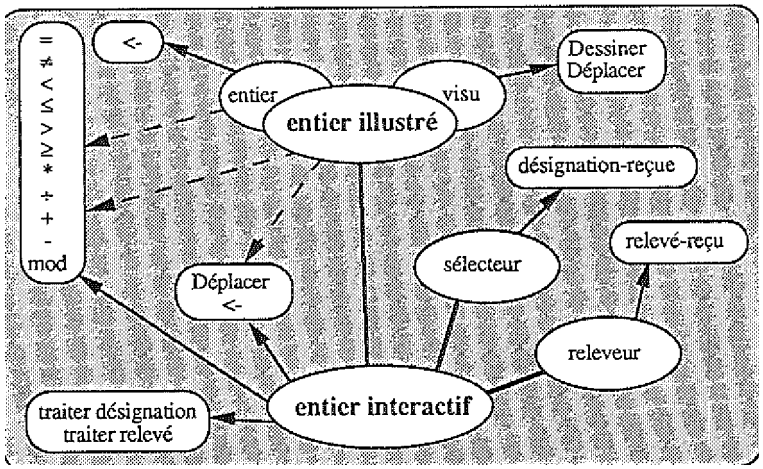

Figure 7 : **Le type entier interactif**

Comme c'était le cas avec l'entier plus-moins, au niveau des manipulations logicielles, notre objet interactif propose les mêmes primitives que l'objet illustré. Par contre, il doit maintenant tenir compte des événements susceptibles de lui arriver en provenance du sélecteur, qui le mettent en mode « actif ». C'est-à-dire qu'une fois désigné, l'objet assume la responsabilité de ses déplacements en effectuant des relevés de coordonnées, ceci afin de déterminer les déplacements que l'on souhaite lui voir effectuer, déplacements qu'il propose à l'entier illustré, ce dernier s'occupant à son tour de répercuter les déplacements à sa partie visualisation. Voir figure 7.
Notons qu'ici nous supposons qu'un entier interactif ne peut que se déplacer, et qu'il assume totalement la manipulation que l'on effectue sur lui : notamment il ne demande pas à l'application qui l'utilise si un déplacement donné est valide ou non. Il serait possible d'une part d'étendre le nombre de manipulations à effectuer sur un tel entier, et d'autre part de lui faire effectuer des demandes d'autorisation de manipulation à l'application qui l'utilise.

5. Les objets composés

Ce sont les objets que l'on rencontre le plus souvent, ils sont obtenus en composant des objets qui peuvent être de base, illustrés, interactifs ou composés.

5.1. Un releveur de coordonnées en trois dimensions

La composition d'objets va nous permettre d'obtenir des objets plus spécifiques, qui seront par exemple aptes à gérer des interactions complexes. Il est ainsi possible de créer un releveur de coordonnées en trois dimensions à partir de deux objets releveurs de coordonnées en deux dimensions : un releveur XY et un releveur YZ (cf. figure 8). Plusieurs possibilités se présentent :

- mettre les deux dispositifs en séquence (releveur XY puis releveur YZ, ou bien le contraire), en initialisant le Y du second avec la valeur Y relevée à l'aide du premier ;
- mettre les deux dispositifs en parallèle, et dès que l'on obtient un relevé sur l'un des deux dispositifs, on contraint l'autre au niveau du Y ;
- même fonctionnement que précédemment mais cohérence permanent& au niveau du Y entre les deux releveurs (solution la plus agréable pour un utilisateur du releveur en trois dimensions).

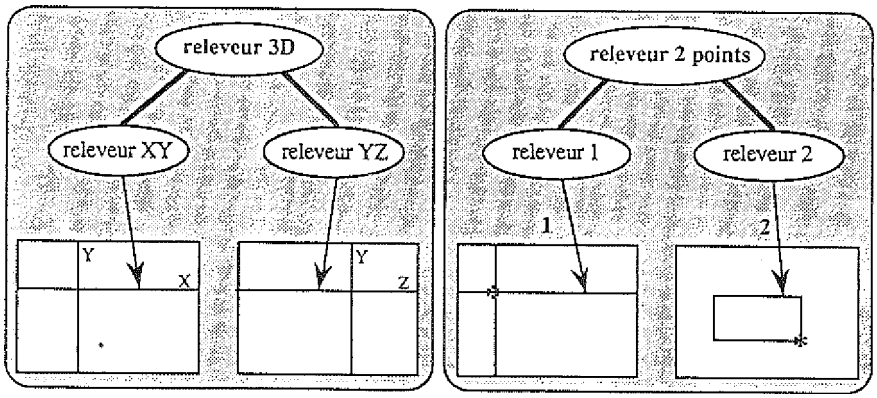

Figure 8 : **Un releveur 3D et un releveur « deux points »**

5.2. Un releveur de couples de coordonnées, avec écho adapté

On peut aussi créer un releveur servant à l'acquisition de données pour la création d'objets graphiques comme des cercles, des rectangles… Par exemple, pour obtenir un dispositif permettant de relever deux points (centre et point sur le cercle, ou deux coins opposés d'un rectangle...), il suffit de créer un objet utilisant deux releveurs à deux dimensions (ou bien de réutiliser deux fois le même, car ici on ne pourra travailler que de façon séquentielle), en utilisant la position relevée à l'aide du premier dispositif pour initialiser le second dispositif, avec un « écho » adapté à l'objet pour lequel on veut utiliser les points.
Ici, plusieurs structures sont possibles pour le releveur « deux points » : il est possible de l'initialiser avec une indication permettant de déterminer « l'écho » adéquat, ou bien, après l'acquisition du premier point, le releveur demande à l'objet qui l'utilise quel « écho » il doit prendre, et dans ce cas il a besoin de pouvoir communiquer avec un responsable.

5.3. Un menu « intelligent »

Il est possible d'apporter un peu de sécurité au niveau du menu : un article de menu peut ne pas être valide à un moment donné, aussi il ne doit pas être sélectionnable, mais il est bon qu'il soit néanmoins affiché, d'une façon différente des autres articles. De même, lorsqu'un article vient d'être désigné, ou s'il représente un état de l'objet utilisant le menu, il est bon qu'il soit mis en valeur. Après chaque « action » de l'objet, celui-ci est susceptible de réévaluer la validité des articles du menu, aussi est-il nécessaire de mettre à jour le menu (si l'on a encore besoin de lui).

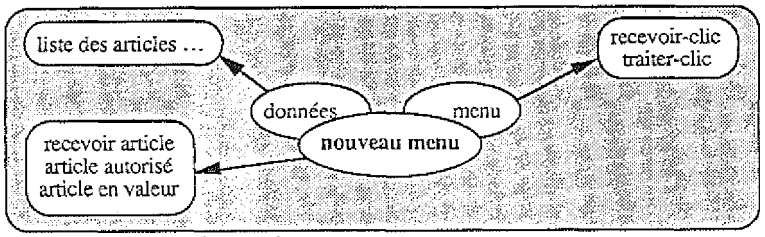

Figure 9 : Le nouveau type menu

Il faut créer un objet de type « nouveau menu » (voir figure 9), qui sera capable de prendre en compte tout ceci, c'est-à-dire qui possédera, dans une partie contrôle, des primitives pouvant renseigner le menu de base sur la façon d'afficher un article, et une primitive permettant de recevoir l'article désigné à l'aide du menu de base. Pour ce faire, nous avons plusieurs possibilités :

- il est possible de communiquer au menu la liste des articles, la liste des articles valides, la liste des articles à mettre en valeur ;
- il est aussi possible de communiquer au menu seulement la liste des articles, il doit alors, pour chaque article, demander à l'objet qui l'utilise s'il est valide et s'il doit être mis en valeur ;
- il est encore possible de donner au menu (dans sa partie abstraction) les éléments lui permettant de déterminer lui-même si un article est valide et s'il doit être mis en valeur. Dans ce dernier cas, le contrôle est effectivement bien réparti, mais le menu est très lié à l'objet qui l'utilise, ce qui n'est pas toujours souhaitable, il est souvent préférable de créer un « super menu » (voir figure 10) composé d'un menu de base et des éléments de décision.

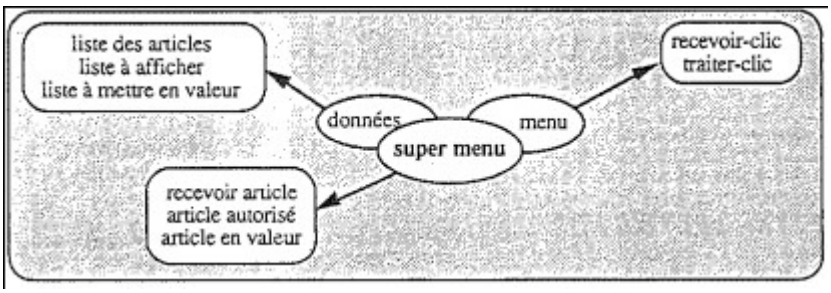

Figure 10 : Un type de menu amélioré avec un contrôle local

5.4. Des objets graphiques interactifs

Nous allons maintenant étudier des objets fortement interactifs : nous souhaitons disposer d'objets graphiques (cercles, carrés...) sur lesquels il soit possible d'effectuer des manipulations directes de façon à les déplacer, modifier, supprimer…

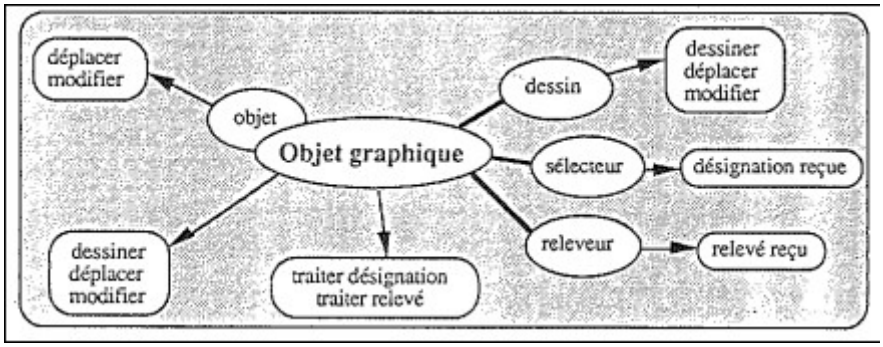

Figure 11 : Le type objet graphique

La classe objet graphique, réalisée afin de pouvoir manipuler un objet (figure 11), propose des primitives qui permettent de le dessiner, modifier et déplacer (partie contrôle de notre objet). Ces primitives font les appels adéquats aux primitives de l'objet (partie sémantique ou abstraction de notre objet) ainsi qu'aux primitives de la partie graphique (partie visualisation de la présentation).
En plus des manipulations logicielles, cette classe va aussi être capable de gérer des manipulations directes de la part d'un utilisateur, ceci grâce à des objets d'entrée sélecteur et releveur (partie acquisition de la présentation).
La partie de fonctionnement la plus intéressante est la partie manipulation directe :

- le sélecteur reçoit un événement de désignation, il le signale au composant contrôle ;
- le composant contrôle doit effectuer une demande à l'objet qui l'utilise afin de savoir quelle est l'action que l'on attend de lui ;
- le composant contrôle donne la main à la primitive sachant gérer l'interaction souhaitée ; si l'action attendue est une suppression, l'objet n'a plus qu'à effectuer la même séquence d'actions que lorsque la demande de suppression lui parvient par voie logicielle, par contre si l'action attendue est une modification ou un déplacement, l'interaction est un peu plus complexe : cette primitive gère donc les événements en provenance du releveur jusqu'à ce qu'elle reçoive un événement d'acquittement (de la part du sélecteur), elle calcule les déplacements à effectuer et appelle la primitive adéquate de l'objet permettant sa mise à jour, avant d'appeler à son tour la primitive adéquate de la visualisation. Quand cette primitive reçoit un événement d'acquittement de la part du releveur, l'interaction est terminée.

Il est possible de faire un certain nombre de vérifications au cours des manipulations des objets graphiques. En effet, un objet graphique peut être autorisé à évoluer seulement dans une certaine zone, ou bien, suivant la zone dans laquelle il se trouve, il peut être limité à une certaine taille. Les déplacements et modifications d'un objet graphique doivent donc parfois être soumis à des vérifications. Ici encore il existe plusieurs façons de réaliser ces vérifications, la plus souhaitable semblant être de demander à l'objet qui utilise un objet graphique, en cours d'interaction, si l'état dans lequel on veut arriver (nouvelle taille ou nouvelle position) est bien un état valide.

5.5. Une application manipulant des objets graphiques

Considérons maintenant une application qui permet de manipuler des objets graphiques tels qu'on en a défini précédemment. Cette application est composée uniquement d'objets structurés selon le modèle PAC : un menu, des cercles, des carrés (ces deux derniers types d'objets dérivant du type objet graphique).

Il est bon de noter que dans notre exemple, les objets graphiques vont pouvoir évoluer librement (pas de contrainte sur leur évolution), et que par contre le menu va être soumis à des contraintes de la part de l'application.

Une manipulation d'objet graphique va pouvoir se dérouler de la façon suivante (voir figure 12) :

1. l'utilisateur désigne un objet graphique, événement renvoyé à l'objet concerné,
2. l'objet demande à l'application quelle est la commande courante, de façon à effectuer l'interaction correspondante,
3. l'objet peut donc effectuer l'interaction en question,
4. l'interaction terminée, l'objet signale à l'application qu'un objet modifié est à prendre en compte,
5. l'application effectue les opérations adéquates, en particulier elle vérifie quelles sont les commandes autorisées à ce moment,
6. l'application n'a plus qu'à demander au menu de se mettre à jour de façon à refléter son nouvel état (celui de l'application),
7. le menu demande alors à l'application, pour chaque article, s'il est valide et s'il doit être mis en évidence,
8. le menu peut ensuite afficher cet article correctement.

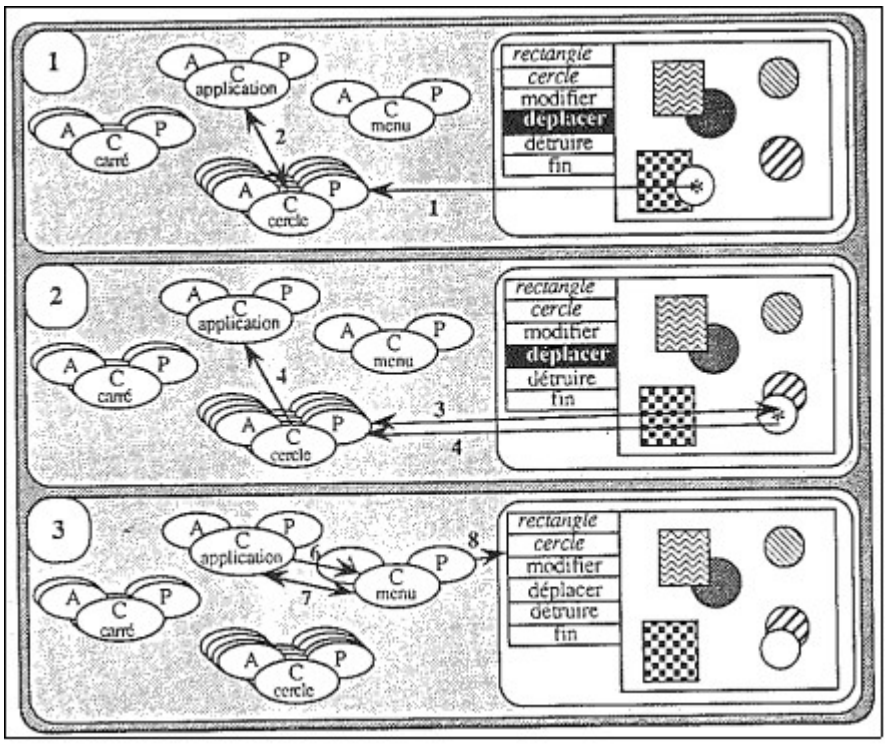

Figure 12 : Un exemple d'interaction

L'exemple d'interaction de déplacement présentée sur la figure 12 possède un nombre d'objets maximum qui est six, ce qui explique que les commandes de création (« rectangle » et « cercle ») ne soient pas proposées et que la commande de déplacement (« déplacer ») soit mise en valeur.

L'application, quant à elle, propose des primitives permettant de savoir si une commande est valide à un moment donné et permettant d'obtenir la « commande courante ».

Remarquons ici que la façon dont on a structuré nos objets n'impose rien quant au mode de fonctionnement de notre application de dessin. Nous avons choisi de fonctionner en mode événement, mais il est possible de travailler en mode requête ou échantillonnage.

6. Relations entre objets d'une même application

Il pourra exister une certaine hiérarchie entre les objets d'une application : certains objets en créent d'autres pour leurs besoins de fonctionnement, d'autres ne font qu'utiliser des objets créés ailleurs.

6.1. Relations entre objets employés et objets employeurs

La composition d'objets crée une sorte de structure arborescente. Chaque objet qui crée un objet peut être considéré comme son « père ». Au niveau des communication, un objet peut avoir besoin de communiquer — à son père — qu'il a reçu un certain événement, il peut aussi lui poser des questions (au sujet du contexte dans lequel il se trouve, ou encore à propos de ce qu'il est autorisé à faire...), ce qui nécessite un échange d'informations du fils vers le père, puis du père vers le fils.

Indépendamment de cette relation de création qu'est la relation père-fils, il peut exister une relation d'appartenance entre deux objets, lorsque l'un des deux est un sous-objet de l'autre (un entier dans une pile, par exemple). Quand un événement survient au niveau du sous-objet, celui-ci peut avoir besoin d'effectuer une demande d'information non plus à son père, mais à l'objet dont il fait partie, ceci pour répartir les responsabilités à travers les différents composants d'une application. Il semble donc souhaitable qu'un objet possède une notion non pas de père, mais plutôt de responsable, afin d'obtenir les renseignements dont il a besoin, ce responsable pouvant à son tour effectuer des demandes à son propre responsable...

6.2. Exemple : une application à manipulation directe

Nous pouvons mettre la nécessité de la notion de responsable en évidence à l'aide de manipulations interactives dans le cadre de la résolution du problème des tours de Hanoï (qui seront en fait ici des piles d'entiers) en effectuant des manipulations directes (sur les entiers).
Il faut définir un certain nombre de règles afin de manipuler correctement nos entiers (nous allons en fait exprimer des contraintes s'exerçant sur les déplacements des objets interactifs entiers) :

- un entier ne pourra pas être empilé sur un entier qui lui est inférieur ;
- seuls les entiers se trouvant sur un sommet de pile seront manipulables ;
- lors d'une manipulation, un entier se trouvant à l'intérieur d'une pile ne pourra se déplacer que verticalement, alors qu'un entier ne se trouvant pas dans une pile ne pourra se déplacer qu'horizontalement (on supposera que les trois piles possèdent leur sommet physique sur une même horizontale : leur sommet sera leur « point d'entrée ») ;
- un objet abandonné en fin de manipulation en-dehors des trois piles, ou dans une pile sur laquelle on ne peut pas l'empiler, se verra ramené dans sa pile d'origine.

Nous proposons le fonctionnement suivant (voir figure 13) :

1. l'utilisateur désigne l'entier qu'il veut manipuler, événement envoyé à l'entier concerné,
2. ce dernier demande à son responsable (une pile) s'il est autorisé à être déplacé,
3. les coordonnées proposées pour l'entier sont communiquées à la présentation de ce dernier,
4. l'entier calcule le déplacement auquel il doit être soumis et demande à son responsable quel est le déplacement qu'il doit réellement effectuer,
5. si le responsable est une pile, elle va elle aussi demander l'approbation de son propre responsable (l'application),
6. l'entier effectue son déplacement (si finalement déplacement il y a),
7. lorsque finalement l'utilisateur termine l'interaction, cet événement est transmis à l'objet manipulé, qui peut alors calculer le dernier déplacement à effectuer,
8. déplacement qu'il soumet à son responsable,
9. si le responsable est une pile, le résultat calculé est lui aussi soumis au responsable de cette pile (c'est toujours l'application),
10. l'entier effectue alors son dernier déplacement,
11. l'application récupère la main, elle détermine dans quelle pile il faut transférer l'entier.

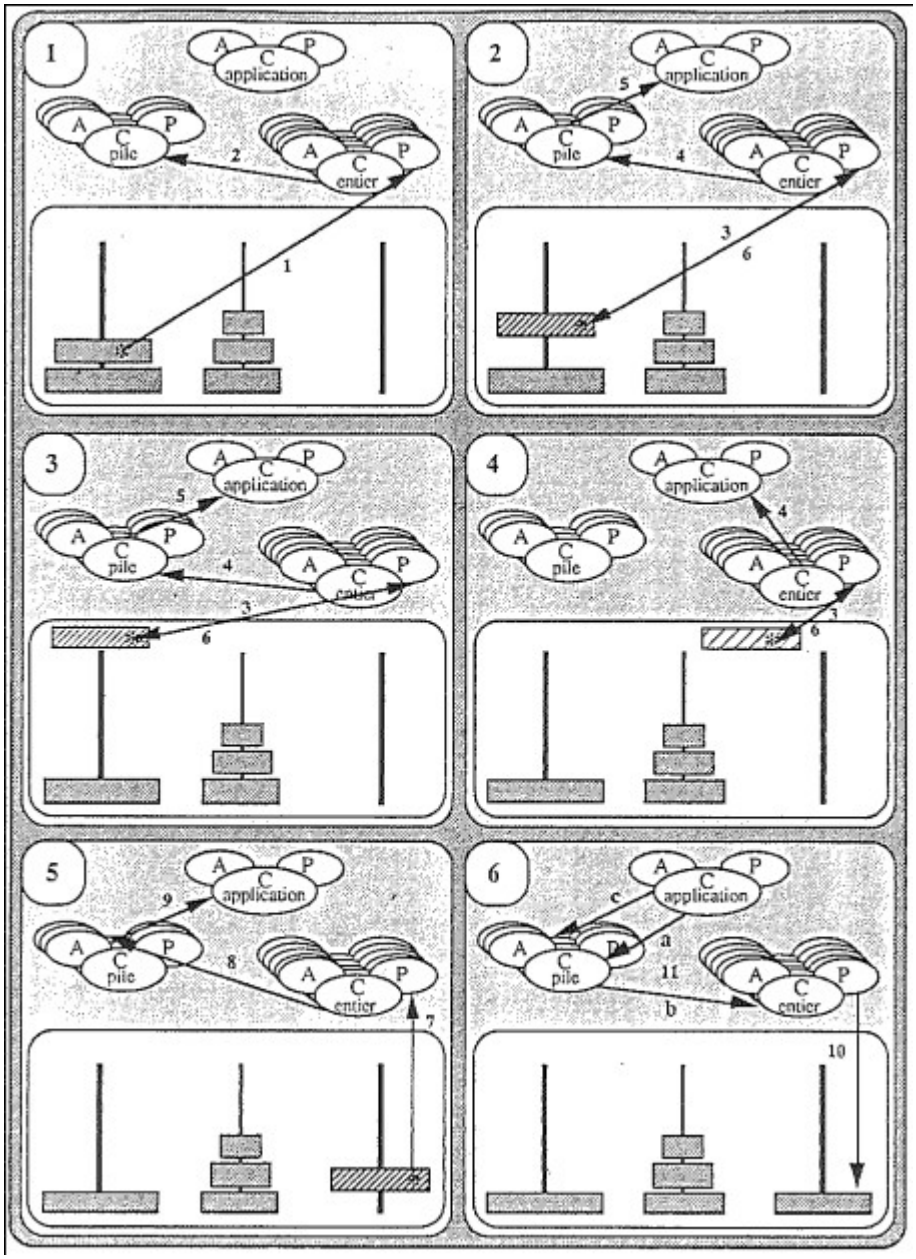

Figure 13 : Exemple de manipulation directe d'entier

Pour obtenir ce type de fonctionnement, nous utilisons un nouveau type d'objet : le type pile interactive, qui dérive de la classe pile illustrée. Nous utilisons aussi le type entier interactif décrit au paragraphe 4.2., auquel nous avons ajouté la notion de responsable nécessaire à nos manipulations.
Lorsqu'un entier se trouve sur un sommet physique de pile, il se trouve dans une position à partir de laquelle il peut évoluer soit horizontalement pour quitter la pile, soit verticalement pour entrer véritablement dans la pile, il faut alors qu'une pile puisse communiquer avec l'application dont elle dépend, de façon à ne pas empêcher un déplacement horizontal pour un entier en sommet physique de pile (c'est en effet l'application, et non pas la pile, qui peut autoriser un éventuel déplacement horizontal). Il faut aussi ajouter la notion de responsable à la classe pile interactive, ainsi que des primitives permettant de savoir si un déplacement donné est autorisé, s'il est possible de déplacer un entier... (voir figure 14).
Comme nos objets vont maintenant devoir faire appel à l'application afin d'obtenir certains renseignements, on ne peut plus considérer cette dernière comme un simple programme : il faut plutôt la considérer comme un module comportant plusieurs primitives travaillant dans un même environnement (les objets manipulés par l'application ont besoin de communiquer avec elle afin d'obtenir des précisions quant au comportement qu'ils doivent avoir).
Dans un souci d'homogénéité, nous considérerons notre application comme un objet possédant des primitives qui permettent de savoir si un déplacement d'entier est autorisé et de confirmer une proposition de déplacement d'entier faite par une pile.

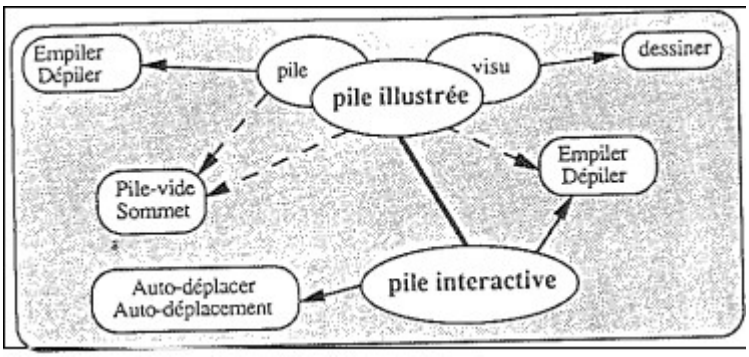

Figure 13 : Le type pile interactive

De façon à réaliser une telle interaction, il y a plusieurs choix possibles : on peut ne faire aucun appel à l'application en cours d'interaction, ou bien faire quelques appels afin de réaliser des vérifications d'ordre syntaxique, ou enfin faire des appels afin de réaliser des vérifications syntaxiques et sémantiques.

6.3. Pourquoi faire des vérifications ?

Ne faire aucune vérification conduit à la situation suivante : au cours de l'interaction, l'utilisateur peut manipuler les objets comme il le désire, il peut en particulier les amener en un point quelconque de la zone de travail, en des situations qui ne sont pas des situations « fonctionnelles », c'est-à-dire qui ne représentent rien pour l'application. Auquel cas, quand elle retrouve le contrôle, l'application n'a plus qu'à annuler tous les efforts de l'utilisateur pour se replacer dans l'état d'avant la manipulation. De plus, en cours d'interaction, on ne peut avoir aucune information nous indiquant si ce que l'on est en train de faire est correct ou non, et en particulier si la situation finale est bien « fonctionnelle ».

Faire des appels de vérifications syntaxiques présente des avantages : on ne peut pas s'écarter d'une certaine ligne de conduite. C'est ce que nous avons réalisé ici, en nous assurant qu'on ne pouvait pas déplacer nos entiers n'importe où (ils doivent rester dans une des trois piles ou bien dans un espace de jonction entre ces trois piles), et en indiquant à l'utilisateur, en changeant la couleur de l'objet manipulé, dans quel type d'endroit il se trouve (dans la jonction et dans une pile, seulement dans une pile, ou bien seulement dans la jonction). Ainsi, les risques d'erreur se trouvent limités : l'utilisateur peut encore placer un objet sur un plus petit que lui, ou bien lâcher l'objet dans la jonction, mais dans ce dernier cas, c'est qu'il n'a pas tenu compte des indications fournies.

La dernière chose que l'on puisse faire est d'empêcher l'utilisateur, à l'aide de vérifications sémantiques, de *tenter* de poser un objet sur un objet plus petit que lui. Pour réaliser ceci, il faut, en plus des vérifications présentées ci-dessus, vérifier, quand un objet se trouve à la fois dans une pile et dans une jonction, qu'il est effectivement plus petit que l'objet se trouvant au sommet de la pile en question. Sinon l'application ne doit pas lui donner l'accès à la pile.
Ce troisième type de vérifications est un peu plus complexe que le deuxième, et peut faire chuter les performances de l'application. C'est pourquoi il est parfois nécessaire de se limiter, au niveau de la réalisation, à des vérifications d'ordre syntaxique. Par contre, l'étude des vérifications sémantiques est très intéressante car elle seule peut nous amener à réaliser du logiciel limitant réellement le risque d'erreur de manipulation.

7. Les réalisations

Nous avons réalisé plusieurs applications en essayant de suivre le modèle d'architecture PAC, et en utilisant différents langages de programmation.

7.1. ADA et GKS : une première application de dessin

Tout d'abord, nous avons réalisé une première application de dessin (type MacDraw) en utilisant le langage ADA et le logiciel graphique GKS [DUV 90a]. Dans cet environnement, un objet PAC se présente sous la forme de trois paquetages génériques qui correspondent aux parties présentation, abstraction et contrôle de l'objet, les paquetages présentation et abstraction étant instanciés à l'intérieur de la partie privée du paquetage contrôle, et ce dernier étant instancié à l'intérieur du paquetage représentant l'objet qui va l'utiliser. Une telle façon de faire est très sécurisante, en ce sens qu'il est impossible de faire communiquer des objets ou des composants présentation et abstraction d'objets autrement que par les composants contrôle adéquats.

Un premier problème de communication se pose cependant : il n'y a pas d'accès direct à la primitive du composant présentation qui doit recevoir les événements graphiques, il faut donc ajouter au composant contrôle une primitive servant de relais pour communiquer les événements au composant présentation.

L'autre problème de communication est qu'il semble difficile de modifier le responsable d'un objet en cours d'exécution : en effet, le moyen de communication avec le responsable utilisé ici est de passer la primitive permettant d'appeler le propriétaire parmi les paramètres génériques du paquetage contrôle de l'objet PAC. Ce schéma est effectivement peu apte à gérer le changement de responsable.

7.2. LISP : une seconde application de dessin

Nous avons réalisé une seconde application de dessin au trait, avec un menu de commandes qui ne proposait que les commandes valides [DUV 90b]. Cette seconde application a été implémentée en LISP avec une couche orientée objet. Elle est peu différente de la première, si ce n'est au niveau de la réalisation où il faut faire très attention à bien respecter la structure PAC (le LISP étant assez permissif). C'est ici que nous nous sommes rendus compte qu'il fallait donner à un objet PAC une notion de responsable, afin de pouvoir obtenir des renseignements qu'un objet ne peut pas connaître à son niveau d'abstraction. Au niveau de la réalisation, ce responsable est ici un objet LISP qu'il est possible

7.3. LISP : visualisation d'objets simples

Toujours en LISP, nous avons étudié les possibilités d'extensions de classes d'objets simples (sans visualisation) à des classes d'objets avec visualisation, dans un but d'illustration des objets manipulés par une application [DUV 91]. Ici aussi, quelques problèmes sont apparus au niveau des réalisation.

Si l'on considère qu'un objet « avec visualisation » est formé par la composition d'un objet sans visualisation (pour la sémantique) et d'un objet graphique (pour la présentation), on a alors le problème suivant : les primitives de l'abstraction qui n'ont pas besoin d'être redéfinies doivent l'être quand même (elles consistent alors en un simple appel aux primitives correspondantes de l'objet composant l'abstraction) de façon à pouvoir être exportées, ce qui semble un peu lourd, mais correct. Si l'on considère qu'un objet avec visualisation « hérite » de l'objet sans visualisation et d'un objet graphique, les primitives de l'abstraction peuvent être exportées sans modification ; par contre, on perd l'avantage de la sécurité au niveau des manipulations sur les objets sans visualisation et sur les objets graphiques car l'héritage implique que l'on ait accès à tous les attributs et primitives des classes dont on hérite, ce qui fait perdre les propriétés d'encapsulation des classes d'objets précédemment définies. La première solution, c'est-à-dire la composition d'objets, semble la meilleure.

7.4. LISP : de l'illustration vers la manipulation directe

Enfin, toujours en LISP, nous avons étudié les possibilités d'extension des classes d'objets avec visualisation à des classes d'objets manipulables, dans un souci de réutilisation logicielle [DUV 91].

Ici encore, des problèmes de réalisation se sont posés : comment obtenir un objet manipulable à partir d'un objet possédant une visualisation ? Faut-il le faire hériter de l'objet avec visualisation, ou bien est-il préférable d'utiliser un objet avec visualisation à l'intérieur d'un objet manipulable ? Il semblerait qu'aucune de ces deux solutions ne soit convenable. En effet, pour un objet manipulable, il va être nécessaire d'avoir une connaissance directe de ses attributs de présentation. Utiliser un objet avec visualisation ne sera pas forcément d'un grand secours car on n'aura pas accès aux particularités graphiques de cet objet : on ne pourra pas toujours s'accommoder d'une telle restriction. Faire hériter l'objet manipulable de la classe avec visualisation n'offre pas non plus une solution viable puisque l'on ne peut toujours pas accéder aux ressources purement graphiques de l'objet avec visualisation (accès uniquement aux primitives exportées par l'objet graphique), sauf bien sûr si ce dernier hérite de l'objet graphique servant à le visualiser, de l'objet graphique servant à la visualisation, et étendu de façon à traiter la manipulation. Un objet manipulable peut alors être obtenu en composant un objet sans visualisation (pour l'abstraction) avec un objet de manipulation (pour la présentation), et en créant un nouveau composant de liaison (pour le contrôle), ce dernier pouvant s'inspirer du contrôle de l'objet avec visualisation. C'est un processus d'héritage qui ensemble beaucoup à ce que l'on peut retrouver dans l'utilisation du modèle MVC de Smalltalk [DUG 90].

8. Conclusion

Les différents exemples que nous avons étudiés nous ont permis de constater que de nombreux objets fréquemment utilisés dans les applications graphiques interactives peuvent être structurés selon le modèle PAC. Ces objets peuvent alors être utilisés tels quels dans d'autres applications (notamment les objets illustrés) ou bien être composés pour créer de nouveaux objets.

Cependant, il n'est pas toujours aisé de donner une architecture PAC à un objet, surtout quand celui-ci doit réutiliser des objets existants. Par contre, PAC offre une méthodologie souvent utile pour donner une structure claire aux objets et applications manipulés. Ceci est plus

particulièrement vrai dans les environnements dits « orientés objets », même si le maintien de la structure PAC n'est pas toujours facile à concilier avec le mécanisme d'héritage et les conseils de programmation objet [MEY 90]. Il faut donc davantage considérer ce modèle comme un cadre de pensée que comme un cadre de réalisation, les langages de programmation imposant parfois certaines contraintes architecturales.

Bibliographie

[BOO 87] G. BOOCH, *Software Components With ADA,* Benjamin/Cummings Publishing Company Inc, 1987.

[CAP 90] P. CAPIRCIO, *Programmation objet, développement d'applications en SMALLTALK,* Armand-Colin Editeur, 1990.

[COU 87] J. COUTAZ, « Méthodes et outils pour l'implémentation des interfaces modernes », Cours INRIA 1987.

[COU 88] J. COUTAZ, « Interfaces homme-ordinateur », thèse de doctorat d'Etat, université Joseph Fourier de Grenoble, décembre 1988.

[COU 90] J. Coutaz, *Interfaces homme-ordinateur, conception et réalisation ;* Dunod Informatique, Bordas 1990 (d'après [Coutaz 88]).

[DUG 90] P. DUGERDIL, *SMALLTALK-80, Programmation par Objets,* Presses Polytechniques et Universitaires Romandes, 1990.

[DUV 90b] T. DUVAL, « Evaluation du modèle PAC, l'apport d'un langage de classes », rapport de recherche ENSM-LISI-R15, juillet 1990.

[DUV 91] T. DUVAL, « Visualisation et manipulation des objets d'une application », rapport de recherche ENSM-LIST-R20, mars 1991.

[FOL 82] J.D. FOLEY, A. Van DAM, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

[GRE 81] M. GREEN, « A methodology for the Specification of Graphical User Interface », *Computer Graphics,* août 1981.

[GRE 85] M. GREEN, « The University of Alberta User Interface Management System », *ACM,* 1985.

[MAS 89] G. MASINI, A. NAPOLI, D. COLNET, D. LEONARD, *Les langages à objets,* InterEditions 1987.

[MEV 87] A. MEVEL, T. GUÉGEN, *SMALLTALK-80,* Eyrolles, 1987.

[MEY 90] B. MEYER, Conception et programmation par objets, InterEditions, 1990.

[PFA 85] G.E. PFAFF, *User Interface Management System,* (Seeheim. 1983), Springer-Verlag, 1988