Les trois services du noyau sémantique indispensables a l'IHM

Jean-Daniel Fekete

Les trois services du noyau sémantique indispensables à l'IHM

Jean-Daniel Fekete
École des Mines de Nantes
4, rue Alfred KASTLER, La Chantrerie
F44070 NANTES Cedex
+33 (02) 51 85 82 08
Jean-Daniel.Fekete@emn.fr

Résumé: La construction d'interfaces Homme-Machine de qualité nécessite la coopération entre un noyau sémantique et des composantes logicielles destinées à visualiser et à interagir avec ce noyau. Jusqu'à présent, le domaine de l'Interaction Homme-Machine s'est surtout intéressé à ces composantes logicielles. Dans cet article, nous arguons que trois services indispensables aux interfaces ne peuvent être fournis convenablement que par le noyau sémantique : la notification, la prévention des erreurs et l'annulation d'opérations. Notre objectif est triple : convaincre la communauté IHM de l'importance de ces services, convaincre la communauté du Génie Logiciel de fournir ces services et décrire les mécanismes permettant d'implémenter ces services. [1]

MOT CLÉS : Génie Logiciel, Noyau Sémantique, Architecture Logicielle, Notification, Prévention des erreurs, Annulation.

1 INTRODUCTION

La branche Génie Logiciel de l'IHM a beaucoup étudié les architectures logicielles pour la construction d'applications graphiques interactives. Un modèle architectural général décrivant ces applications est l'arche [15], qui répertorie cinq composantes (voir figure 1). La majorité des travaux portant sur l'aspect génie logiciel de l'IHM, a évité le problème du noyau sémantique. Cependant, une grande partie de la complexité des applications graphiques interactives vient de la communication avec ce noyau sémantique. Le concepteur d'IHM doit réussir à extraire toutes les informations dont il a besoin de ce noyau, parfois avec énormément de mal, voire sans succès.

Dans cet article, nous voudrions convaincre les concepteurs de noyau sémantique de l'importance de ces services et décrire des mécanismes permettant de les implémenter. La mise en œuvre de ces mécanismes peut s'avérer compliquée ou impliquer un coût non négligeable, mais si ces services ne sont pas disponibles dans le noyau sémantique, ils sont impossibles à émuler convenablement.

2 LES TROIS SERVICES INDISPENSABLES

Nous pensons que toute bibliothèque ou module doit offrir les trois services suivants :

la notification : la possibilité pour un module externe d'être prévenu lorsque l'état du noyau sémantique change ;

la prévention des erreurs : la possibilité de savoir si un appel de fonction est licite dans un contexte ;

l'annulation : la possibilité de revenir à des états précédents du noyau sémantique.

Journées du GDR Programmation.
20, 21 et 22 novembre 1996.
Orléans.

1. Cet article est publié dans les actes des journées de travail d'IHM'96 a Grenoble.

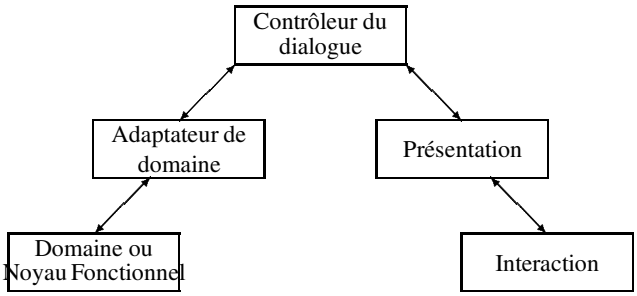

FIG. 1 – *Le modèle architectural de l'Arche*

3 NOTIFICATION

La notification est un service qui permet à un module logiciel qui le désire (l'*observateur*) d'être informé lorsqu'une structure de données est modifiée (le *sujet*). *Observateur* et *sujet* sont des termes tirés de *Design Pattern* [6] qui décrit précisément les composantes participant à la notification.

La notification nécessite trois étapes : l'enregistrement, le déclenchement et le traitement. Lors de la première étape, le programmeur enregistre une action qui sera invoquée au déclenchement et provoquera un traitement. L'action est généralement décrite par une fonction associée à un état. Lors de l'invocation, des informations contextuelles sont passées à la fonction qui précisent la cause de cette invocation.

Par exemple, si le système de fichiers d'un système d'exploitation offre un mécanisme de notification, une application interactive visualisant une partie du système de fichiers peut s'enregistrer comme observateur et être notifiée lorsque le système de fichiers est modifié. En se réaffichant lors de chaque notification, l'application garantit que ce qu'elle visualise est continuellement à jour. Cet exemple n'est pas pris au hasard puisque les systèmes d'exploitation comme Unix et MS-DOS n'offrent pas ce service et ne permettent donc pas à un shell iconique de maintenir une vue continuellement à jour du système de fichiers ([4]).

3.1 Problèmes

L'implémentation de la notification en dehors du noyau sémantique est très difficile à faire, voire impossible. Lorsque la modification du noyau sémantique n'est faite qu'à l'initiative du contrôleur de dialogue, celui-ci peut forcer la présentation à se remettre à jour après chaque action. Cependant, ce mode de mise à jour à l'initiative du contrôleur du dialogue implique que celui-ci sache exactement la portée de chaque action dans le noyau sémantique, ou qu'il recalcule systématiquement la totalité de l'affichage après chaque action. Le premier cas est dangereux d'un point de vue Génie Logiciel car il remet en cause la modularité du système. Le second cas est généralement trop coûteux.

Dans un nombre croissant d'applications, les modifications du noyau sémantique se font indépendamment du contrôleur du dialogue (par exemple sur un système de fichiers) et l'unique moyen de connaître ses changements d'état est d'interroger régulièrement le noyau sémantique. Cette interrogation est coûteuse. Même lorsqu'une interrogation régulière est faite, l'interprétation du changement d'état est souvent difficile et ne permet pas une mise à jour intelligente de l'interface. Par exemple, si le noyau sémantique est un système de fichiers et qu'un observateur en visualise une partie sous la forme d'un arbre, le fait de renommer un fichier modifie l'état du système de fichiers. Lorsque l'observateur constate que le système de fichiers a changé, il pourra déduire qu'un fichier a disparu et qu'un nouveau est apparu ce qui le forcera à refaire son calcul de placement d'arbre. Une notification aurait pu lui apprendre la nature de la modification et lui éviter ce travail.

Lorsque le noyau sémantique offre un service de notification, la synchronisation entre les données gérées par noyau sémantique et leurs représentations graphiques est garantie.

3.2 Utilisation de la notification dans les IHM

Le premier modèle architectural décrivant clairement le rôle de la notification dans les IHM est MVC de Smalltalk (*Model View Controler*) décrit en figure 2. Dans l'environnement Smalltalk, les objets appartenant au noyau séman-

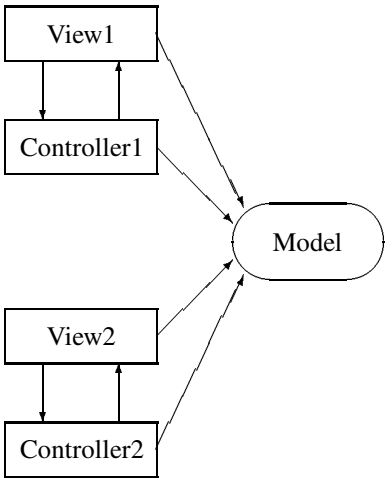

FIG. 2 – *Relations entre les composantes du modèle MVC de Smalltalk. Le Modèle ne connaît pas la nature de ses vues ni ses contrôleurs. Les vues et contrôleurs connaissent la nature du modèle qu'ils observent. Inversement, c'est le modèle qui notifie ses vues et ses contrôleurs.*

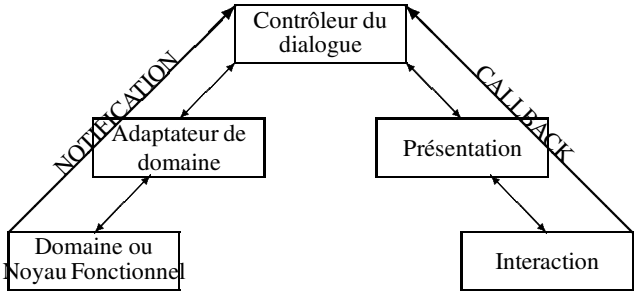

FIG. 3 – *Comparaison de la communication par Callbacks et par notification dans le modèle de l'Arche*

tique héritent de la classe *Model*. Pour visualiser un modèle, un objet de la classe *View* doit être créé et attaché au *Model*. C'est au modèle de décider quand il considère qu'il a changé. Il appelle alors la méthode *changed* qui déclenche la notification en appelant la méthode *update* sur toutes les vues attachées.

Notons qu'avec MVC, le modèle ne connaît pas la nature des vues qui lui sont attachées. Il ne sait même pas si une vue est attachée. En revanche, une vue doit connaître la nature du modèle qu'elle visualise. La notification est un service offert par le modèle à des vues qu'il ne connaît pas mais qui seront créées spécialement pour lui.

Le contrôleur de MVC ne sert qu'à la gestion de l'interaction sur la vue et la propagation de l'interaction au modèle. Il fait donc partie des composantes du modèle de l'arche qui ne nous intéressent pas ici.

Après Smalltalk, toutes les boîtes à outils ont utilisé un mécanisme de notification (notamment les *Callbacks*) mais pour gérer l'autre côté de l'arche : la communication entre la présentation et le contrôleur du dialogue (figure 3). Nous engageons le lecteur intéressé par les détails à consulter *Design Pattern*. Nous préférons nous concentrer sur l'implémentation du service et son coût.

3.3 Mécanismes

Les mécanismes de déclenchement de la notification sont parfois mis en œuvre au niveau du langage et parfois au niveau de bibliothèques. Des systèmes comme Garnet [12] et TCL [13] utilisent des mécanismes de type langage, tandis que Smalltalk et les langages à typage statique comme C, C++ ou d'autres langages à objets utilisent des mécanismes gérés en bibliothèque.

Notons que la notification est très difficile à implémenter dans des langages anciens comme Pascal [7] ou Fortran

```
1  set v 3
2  proc trace_v { var ignore op } {
3      puts "Modification de v"
4  }
5  trace variable v w trace_v
```

FIG. 4 – *Notification par variable active dans le langage TCL*

[8] qui ne permettent pas de manipuler des objets fonctionnels. C'est peut-être ce qui explique que la notification ne soit pas enseignée tôt dans les cursus universitaires, le langage Pascal servant le plus souvent de support.

3.3.1 Mécanismes de langage

Lorsque la notification est gérée par le langage, elle utilise le mécanisme de variable active ou de propagation de valeurs [11, 10]. Dans sa plus simple expression, à la demande du programmeur d'application, le fait d'accéder ou de modifier la valeur d'une variable déclenche l'appel à une fonction. Par exemple, en TCL, le programme de la figure 4 permet de tracer les modifications de la variable *v* en écriture.

Dans cet exemple, la commande `trace` en ligne 5 spécifie que lorsqu'on accédera à la variable *v* en écriture (argument *w* pour *write*), la fonction `trace_v` sera invoquée. Ici, aucun état n'est passé à cette fonction, des arguments supplémentaires auraient du être ajoutés après le nom de la fonction pour cela. Lorsqu'elle est appelée, la fonction reçoit un contexte relativement simple en argument : le nom de la variable et l'opération qui a déclenchée l'invocation. Dans cet exemple, la fonction est définie en ligne 2 et se contente d'afficher un message.

Ainsi, tout programme écrit dans un langage gérant la notification permet l'observation de la valeur des variables visibles (les règles de visibilité variant d'un langage à un autre). Cette facilité se paye à l'exécution soit par une consommation de place importante s'il faut stocker une liste de fonctions pour chaque variable active, en tout cas par une vitesse réduite puisqu'il faut au moins tester si une notification doit être déclenchée lors de l'accès et de la modification de chaque variable.

Les langages récents, comme Eiffel [9], Ada-95 [1] ou même C++ [5] n'ont pas de mécanisme de notification prédéfini, peut-être pour des problèmes de performances mais surtout parce que l'observabilité et la visibilité des variables est en contradiction avec l'encapsulation.

3.3.2 Mécanismes en bibliothèque

Lorsque la notification est gérée en bibliothèque, chaque module offrant la notification dispose d'une fonction permettant d'enregistrer un observateur (`AddDependent` en Smalltalk, *AddCallback* avec la X Toolkit).

La principale différence entre les variables actives et la notification gérée en bibliothèque concerne le déclenchement de la notification. Elle est implicite dans le premier cas et explicite dans le deuxième. Lorsque le sujet est simple, par exemple un nombre ou une chaîne de caractères, chacune de ses modifications doit déclencher une notification. Lorsque la structure observée est plus complexe, certaines opérations primitives modifient plusieurs champs. La notification doit alors être déclenchée à la fin de l'opération et non lors de la modification de chacun des champs. Suivant les cas, les observateurs peuvent être intéressés par les aspects structurels de la notification (quels sont les champs qui ont changé ?) ou par l'aspect fonctionnel (pourquoi le noyau sémantique a-t-il changé ?). La notification doit autant que possible lui donner ces deux informations. Enfin, lorsque le noyau sémantique n'est que partiellement observé, il peut être important de filtrer les notifications reçues par un observateur. Par exemple, la visualisation de l'arborescence d'un système de fichiers ne porte que sur une partie de l'arbre. Seule une notification sur cette partie est importante au programme de visualisation. Recevoir une notification sur toutes les opérations serait certainement inutile et coûteux.

3.4 Notification synchrone et asynchrone

Nous avons jusqu'à présent décrit des mécanismes de notification synchrones, c'est-à-dire dont le traitement suit immédiatement le déclenchement. Lorsque la notification est asynchrone, le traitement est différé. La notification synchrone n'est pas indispensable pour l'IHM car le réaffichage est généralement beaucoup moins fréquent que la modification du noyau sémantique. Seules certaines phases de manipulation directe requièrent un réaffichage synchrone.

De plus, une notification peut être déclenchée à un moment très inopportun pour l'interface, par exemple pendant un réaffichage ou une action modale. Sa prise en compte doit alors être retardée. Pour ces deux raisons, les applications graphiques interactives préfèrent gérer une queue de notifications de la même façon qu'ils traitent une queue d'événements (les deux pourraient d'ailleurs être unifiés). Suivant les cas, une queue différente peut être gérée pour chaque source de notification ou une queue globale peut contenir tous les événements. Lorsqu'un seul fil d'exécution (*thread*) existe, nous pensons souhaitable qu'une seule queue unifiée serve pour les notifications et les événements (nous ne connaissons que la boîte à outils Xtv [3, 2] qui prenne ce parti). Dans le cas contraire, la boucle principale de gestion des événements et des notifications asynchrones doit connaître la liste de toutes le queues et gérer leur interclassement. De plus, l'analyse des événements dans leur chronologie est très riche en informations. Par exemple, le fait de savoir qu'un réaffichage complet suivra une séquence de notifications permet parfois d'ignorer les notifications.

3.5 Choix d'implémentation

Lors de la conception d'un noyau fonctionnel, il est donc essentiel de prévoir le service de notification. Plusieurs mécanismes sont utilisables et chaque langage ou domaine peut nécessiter un mécanisme plutôt qu'un autre. Nous montrons ici que la gestion de la notification n'est pas très compliquée à implémenter.

Polymorphisme Si le langage de programmation le permet, une structure de donnée peut être dynamiquement remplacée par une autre lorsqu'elle a besoin d'être observée. Par exemple, dans un langage à objets, si un objet doit être observé, il peut être remplacé soit par un objet de classe dérivée, soit par un représentant, qui gérera la notification. Ces deux mécanismes sont très utilisés en Smalltalk. Si la notification n'est pas demandée, l'objet utilisera des méthodes rapides et, dès que la notification sera demandée, un nouvel objet sera créé qui, suivant les cas, copiera les champs du premier ou pointera vers l'objet original. Par la suite, les méthodes de l'objet créé géreront la notification. Ce mécanisme repose sur le polymorphisme de sous-typage qui rend l'appel de fonction un peu plus coûteux. Si ce coût est acceptable ou déjà payé parce que l'objet doit pouvoir être spécialisé, la notification devient gratuite pour les objets qui n'ont pas d'observateurs.

Si le polymorphisme de sous-typage n'est pas utilisable, alors chaque fonction du noyau sémantique faisant un effet de bord doit déclencher explicitement une notification.

Communication Lorsqu'une notification est déclenchée, un contexte doit être passé à l'observateur. La richesse de ce contexte dépend de la nature du sujet et varie entre deux extrêmes, ce sont les modèles *pousse* et *tire* de *Design Patterns* (*push/pull*). Dans le modèle *pousse*, le sujet passe en paramètre tout son état et l'observateur n'a donc rien à demander de plus au sujet pour se mettre à jour. Dans le modèle *tire*, le sujet ne passe rien (sauf lui-même) à l'observateur, qui doit alors interroger le sujet pour trouver ce qui a été modifié et a déclenché la notification.

En général, plus le sujet est complexe et plus les informations passées doivent être nombreuses.

3.6 Synthèse

Puisque le service de notification doit être implémenté au niveau du noyau sémantique, les concepteurs de langages, de systèmes, de bibliothèques et de paquetages doivent mettre en œuvre un des mécanisme décrit dans cette section.

4 PRÉVENTION DES ERREURS

Le second service indispensable aux IHM doit permettre de savoir si l'appel à une fonction du noyau sémantique provoquera une erreur, et plus généralement la liste des fonctions du noyau sémantique qui peuvent être invoquées sans déclencher d'erreur. Le premier cas permet de suivre une des huit règles d'or de la conception du dialogue de Ben Shneiderman [14] :

> *Offrir une gestion simple des erreurs.* Autant que possible, concevoir le système de manière à ce que l'utilisateur ne puisse pas faire d'erreur sérieuse. Si une erreur est faite, le système doit la détecter et offrir un mécanisme simple et compréhensible pour la gérer. [...] Une commande erronée doit laisser le système inchangé, ou le système doit donner des instructions pour revenir à l'état précédent.

Dans sa généralisation, le service de prévention des erreurs est indispensable pour offrir le retour d'information nécessaire à la manipulation directe. Par exemple, dans un shell iconique comme celui du Macintosh, le fait de prendre un icone i_1 et de le lâcher sur un icone i_2 déclenche la gestion du fichier visualisé par i_1 par le fichier visualisé par i_2 (le sens de gérer dépend de i_2, en général, il s'agit de faire éditer le document i_1 par l'application i_2). Pendant la manipulation directe, seuls les icones capables de gérer i_1 passent en inverse vidéo lorsque le pointeur passe au-dessus, signalant à l'utilisateur les contextes valides pour terminer la manipulation. On imagine mal comment cette information pourrait être déduite sur un système comme Unix. Sur le Macintosh, les documents sont typés et indiquent au shell iconique la liste des types de documents qu'ils sont capables de gérer : le noyau sémantique fournit donc le service de prévention des erreurs sous la forme d'une précondition pour la fonction *gérer(i_1, i_2)*.

Même en dehors de la manipulation directe, ce service est important. Par exemple, lorsque des objets graphiques sont sélectionnés, les items de menu agissant sur la sélection doivent être désactivés si leur activation produit une erreur.

Dans le modèle de l'arche, c'est l'adaptateur du domaine et le contrôleur du dialogue qui sont en charge de simuler ce service lorsqu'il n'est pas offert par le noyau sémantique. Certaines conditions peuvent être effectivement testées en dehors du noyau sémantique, mais une grande partie ne le peut pas (comme dans l'exemple du shell iconique) ou à un coût prohibitif.

4.1 Mécanismes

Les langages modernes utilisent des *exceptions* pour signaler les erreurs. Les exceptions permettent de rattraper les erreurs, pas de tester si elles vont se produire. Des langages comme Eiffel permettent de spécifier des préconditions, qui pourraient être utilisées mais qui ne sont pas accessibles comme prédicat dans le langage, elles sont destinées au compilateur pour des tests statiques ou dynamiques.

Les réseaux de Pétri maintiennent explicitement l'information des transitions possibles. Hélas, aucun système opérationnel ne les utilise.

À défaut de mécanismes gérés par le langage, nous proposons d'associer à chaque fonction ou procédure f du noyau sémantique provoquant un effet de bord, un test de validité. Ce test est une fonction T_f qui prend les mêmes arguments que f et retourne une valeur dans l'ensemble { accepte, refuse, ne-sait-pas }. La fonction de test ne doit jamais provoquer d'erreur ni modifier l'état du noyau sémantique. Avant d'appliquer la fonction f du noyau sémantique, le contrôleur du dialogue peut appeler la fonction T_f et éviter de déclencher une erreur lorsque le test retourne la valeur *refuse*. La valeur *ne-sait-pas* est nécessaire car certaines opérations ne sont pas testables dans un délai raisonnable.

5 ANNULATION D'OPÉRATIONS

Le dernier service indispensable aux applications interactives et qui ne peut être convenablement émulé est la gestion de l'annulation des opérations. Shneiderman cite la règle suivante :

> *Permettre une annulation facile des actions.* Autant que possible, les actions doivent être réversibles. Cette caractéristique diminue l'inquiétude car l'utilisateur sait que les opérations erronées peuvent être annulées. Il est donc encouragé à explorer des opérations qui ne lui sont pas familières. L'annulation peut porter sur une seule action, la saisie de données ou un groupe entier d'actions.

La gestion de l'annulation peut parfois être simulée par l'adaptateur de domaine. Par exemple, dans un système de fichiers ne gérant pas de version, la destruction d'un fichier est irrémédiable. Cependant, un shell iconique peut simuler la destruction d'un fichier par son déplacement dans un répertoire spécial. Ainsi, la destruction peut être annulée en replaçant le fichier à sa place originale. Cependant, si une application quelconque détruit le fichier, l'annulation ne pourra pas fonctionner[2].

5.1 Mécanismes

Dans les IHM, deux modèles existent : l'annulation à un niveau et à plusieurs niveaux. Dans le premier cas, seule la dernière commande de l'utilisateur est annulée tandis que dans le second, l'utilisateur peut annuler et refaire plus d'une commande, en général un nombre limité par les ressources informatiques disponibles. Du point de vue du noyau

2. Bizarrement, le service de version de fichier n'existe pas dans les systèmes les plus populaires (Unix, MS-DOS, MacOS) alors qu'il existait sur des systèmes plus anciens (VMS, Multics, TOPS-20).

fonctionnel, il est impossible de prévoir la granularité des commandes proposées à l'utilisateur et un mécanisme d'annulation général doit être mis en œuvre.

Le mécanisme le plus connu pour gérer l'annulation est celui utilisé par les bases de données pour le *Rollback*. L'interface de ce mécanisme offre la possibilité de nommer un état de la base de données (parfois appelé *signet*), puis, plus tard, de défaire toutes les modifications effectuées à la base de données pour revenir à cet état.

Pour l'IHM, un tel mécanisme convient parfaitement pour implémenter l'annulation. Si la réexécution est proposée, alors le contrôleur de dialogue doit enregistrer toutes les commandes effectuées par l'utilisateur afin de pouvoir les rejouer. Ce mécanisme, décrit dans *Design Pattern* sous le nom de *Command* fonctionne parfaitement une fois que l'annulation est fournie par le noyau sémantique.

Pour utiliser le mécanisme, trois fonctions sont nécessaires : la création d'un signet, sa destruction et l'annulation des opérations jusqu'au signet. Cette dernière opération rend aussi le signet invalide. Avant chaque commande interactive, le contrôleur du dialogue doit créer un signet permettant ensuite de l'annuler.

Pour un noyau sémantique non réparti, l'implémentation de l'annulation repose sur la maintenance d'un historique des opérations. Chaque opération primitive modifiant le noyau sémantique doit allouer un bloc de mémoire et stocker dedans tous les éléments nécessaires à annuler l'opération, ainsi qu'un identificateur de fonction d'annulation. Une fonction d'annulation est associée à chacune des fonctions primitives. Elle reçoit en paramètre le bloc de mémoire et restaure l'état du noyau sémantique. Chaque bloc créé est rajouté à une pile. Un signet est simplement un index dans cette pile. L'annulation jusqu'à un signet revient à dépiler chaque bloc, le passer à sa fonction d'annulation et le détruire, jusqu'à arriver à l'index du signet. La destruction du premier signet valide de la pile permet de récupérer toutes les ressources du début de la pile jusqu'au signet suivant.

Le fait de sauvegarder l'état modifié peut être coûteux en temps et en place. Notons que tant qu'aucun signet n'est créé, l'historique n'a pas besoin non plus d'être créé. Du point de vue de l'IHM, plus l'utilisateur peut revenir en arrière, meilleure est l'interface. Néanmoins, au delà de quelques niveaux d'annulation, le retour en arrière devient de l'ordre du confort et plus de la sécurité. Un moyen d'éviter que ce confort ne gaspille trop de ressources est de limiter le nombre de signets. Cette limitation peut être faite au niveau de l'interface en détruisant les premiers signets au fur et à mesure que de nouveaux sont créés. Nous avons testé plusieurs stratégies et préférons limiter le nombre de signets en fonction des ressources consommées. Pour cela, une fonction permettant de connaître la quantité de ressources associée à un signet doit être rajoutée au noyau sémantique.

6 CONCLUSION

Nous avons décrit dans cet article trois services qui manquent à la plupart des noyaux sémantiques bien qu'ils soient indispensables à la construction d'IHM de bonne qualité. Notre objectif est triple :

- pousser la communauté des IHM à exiger ces services de la part des clients pour lesquels ils travaillent ;
- convaincre la communauté des langages et systèmes que ces services sont indispensables ;
- donner quelques éléments leur permettant d'implémenter ces services.

Nous espérons recevoir, de la part des participants aux journées IHM'96, des commentaires et des compléments afin de réaliser ces trois objectifs.

Références

[1] Ada9X Mapping/Revision Team. Programming Language Ada–Language and Standard Libraries. Draft, Version 4.0, September 1993.

[2] Michel Beaudouin-Lafon, Yves Berteaud, and Stéphane Chatty. Creating direct manipulation applications with xtv. In *EX'90 European X Window System Conference*, April 1990.

[3] Michel Beaudouin-Lafon, Yves Berteaud, Stéphane Chatty, Jean-Daniel Fekete, and Thomas Baudel. The X television – C++ library for direct manipulation interfaces. Technical report, LRI, Université de Paris-Sud, France, February 1991. version 2.0.

[4] Michel Beaudouin-Lafon and Solange Karsenty. Iconic shells for multitasking workstations. *ACM Symposium on Personal and Small Computers*, pages pp 187–196, May 1988.

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, USA, 1990.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, USA, 1994.

[7] International Organization for Standardization. *Specification for Computer Programming Language Pascal, ISO 7185-1982*, 1982.

[8] Harry Katzan, Jr. *Fortran 77*. Computer Science Series. Van Nostrand Reinhold, New York, NY, USA, 1978.

[9] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, February 1987.

[10] J. Moloney, A. Borning, and B. Freeman-Benson. Constraint technology for user-interface construction in ThingLab II. *ACM SIGPLAN Notices*, 24(10):381–388, October 1989.

[11] Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, UI Builders, pages 211–220, 1991.

[12] Brad A. Myers, Dario Giuse, and Roger Dannenberg et al. GARNET comprehensive support for graphical, highly interactive user interfaces. *COMPUTER magazine*, November 1990.

[13] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[14] Ben Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, pages 57–69, August 1983.

[15] User Interface Developer's Workshop. The Arch model: Seeheim revisited. Presented at ACM SIGCHI, April 1991.