

*CAD/GL*

# **ESCADRE V4 / Graphique**

**User\_Interface**

**Edition du 05/03/97 14:48**

**E Berry**







---

# Sommaire

<b>1. Généralités</b>	<b>1-1</b>
1.1 Introduction	1-1
1.2 Documents rattachés	1-1
1.3 Philosophie de la bibliothèque	1-1
1.4 Fonctionnalités de la bibliothèque <i>UI</i>	1-2
1.5 Multi-display, mutli-thread et multi-opérateurs	1-3
1.6 Paquetages et Fichiers	1-3
1.7 Programmes d'exemple	1-5
1.8 Structure du document	1-7
1.9 Audience de la documentation	1-8
<b>2. Initialisations et gestion des événements</b>	<b>2-1</b>
2.1 Initialisations	2-1
2.2 Gestion des événements	2-2
2.2.1 Appels bloquants	2-2
2.2.2 Appels non-bloquants	2-3
2.2.3 Boucle locale	2-3
<b>3. Interfaces multi-opérateurs</b>	<b>3-1</b>
3.1 Introduction	3-1
3.2 Contraintes d'écriture des callbacks	3-1
3.3 Invalidation des callbacks	3-2
<b>4. Création et gestion de widgets</b>	<b>4-1</b>
4.1 Création générale (CREATE_WIDGET)	4-1
4.2 Gestion d'un widget	4-3
4.3 Interfaces définis en UIL	4-4
4.4 La drawing area	4-4
4.4.1 Le widget rowColumn	4-6
4.4.2 Pulldown	4-7
4.4.3 Rangée de boutons	4-9
4.4.4 Radio box et check box	4-9
4.5 Saisie de type simple	4-13
4.5.1 Prompt widget	4-14
4.5.2 Dialogues à champ unique	4-17
4.5.3 Prompts dialogues par générique	4-18
4.6 Les widgets container	4-20
4.6.1 List dialog	4-20
4.6.2 Rangée d'attributs	4-22
4.6.3 Frame	4-22
4.6.4 Terminal et text box	4-23
4.6.5 Ouverture de connexions	4-24
4.7 Sélection d'une sous-liste	4-24

<b>5. Ressources</b>	<b>5-1</b>
<b>5.1 Fonctions générales</b>	<b>5-1</b>
5.1.1 Fonctions utilitaires	5-2
5.1.2 Attachements	5-2
5.1.3 Widgets de saisie	5-3
5.1.4 Sensibilité	5-4
5.1.5 Bitmaps et pixmapes	5-4
<b>5.2 Fichier de ressources</b>	<b>5-4</b>
<b>6. Callbacks et event handlers</b>	<b>6-1</b>
<b>6.1 Généralités sur le mécanisme de callback et interfaçage avec le monde Ada</b>	<b>6-1</b>
<b>6.2 Pointeurs sur procédure</b>	<b>6-2</b>
<b>6.3 Dépôt des callbacks et event handlers</b>	<b>6-3</b>
<b>7. Parcours de la hiérarchie</b>	<b>7-1</b>
<b>7.1 Rappels sur les hiérarchies de widgets</b>	<b>7-1</b>
<b>7.2 Descendants</b>	<b>7-2</b>
<b>7.3 Ancêtres</b>	<b>7-3</b>
7.3.1 Ancêtre direct	7-3
7.3.2 Ancêtre fenetre	7-3
7.3.3 Ancêtre shell	7-4
<b>7.4 Diffusion d'une action sur la hiérarchie</b>	<b>7-4</b>
<b>7.5 Widget interne d'un widget construit par composition</b>	<b>7-5</b>
<b>7.6 Caractéristiques d'un widget</b>	<b>7-5</b>
<b>8. Divers</b>	<b>8-1</b>
<b>8.1 Manipulation du curseur</b>	<b>8-1</b>
<b>8.2 Gestion des couleurs</b>	<b>8-1</b>
<b>8.3 Aide à la mise au point de l'IHM</b>	<b>8-2</b>
8.3.1 boîte de mise au point	8-2
8.3.2 Cascade deadlock	8-2
8.3.3 Mise au point de bas niveau	8-3
<b>8.4 Gestion des erreurs</b>	<b>8-3</b>
<b>8.5 Paquetage MOTIF_TERMINAL</b>	<b>8-4</b>
<b>8.6 Le browser</b>	<b>8-7</b>
8.6.1 Introduction	8-7
8.6.2 Définition d'une hiérarchie	8-8
8.6.3 Services offerts par le composant	8-9
8.6.4 Conditions sur les fonctions définissant la hiérarchie	8-9
8.6.5 Exemple	8-15
<b>9. Exemple détaillé</b>	<b>9-1</b>
<b>10. Spécifications Ada des composants</b>	<b>10-1</b>

# 1. Généralités

## 1.1 Introduction

La couche *G* contient les services graphiques de la boîte à outils *ESCADRE*. Elle comprend deux grappes de bas niveau, *GRAPHIC\_PRIMITIVES* et *USER\_INTERFACE*, qui sont totalement indépendantes d'une application particulière. Elle renferme également trois grappes de plus hauts niveau, *GRAPHIC\_TOOLS*, *DIAGRAM* et *ISME*, bâties à partir des composants de bas niveau.

*DIAGRAM* permet la représentation d'histogrammes et de courbes. *ISME* permet, à partir de la description banalisée d'une variable, la création d'une IHM graphique pour la saisie de cette variable. *GRAPHIC\_TOOLS* permet l'affichage d'un fond sur lequel se déplacent des symboles graphiques.

Le composant de bas niveaux *GRAPHIC\_PRIMITIVES* fournit les services graphiques élémentaires comme le tracé de lignes ou le remplissage de polygones. Enfin, le composant de bas niveaux qui nous intéresse ici, *USER\_INTERFACE*, fournit les services pour la réalisation d' IHM graphique. Le composant *USER\_INTERFACE* est également référencé par l'acronyme *UI*.

## 1.2 Documents rattachés

Ce document est le manuel du programmeur du composant *USER\_INTERFACE*. Le document de référence est constitué des spécifications abondamment commentées des paquetages du composant *UI*. La documentation *Motif*, référencée en annexe, constitue le manuel du concepteur (guide de style). Ce document est le premier document à lire concernant *USER\_INTERFACE*.

Lorsqu'une fonction de *UI* est l'équivalent d'une fonction *Motif* existante, elle n'est pas décrite en détail dans ce document. Il est alors nécessaire de se reporter à la documentation *Motif*. Ce document est insuffisant pour une personne qui n'a jamais écrit d'applications *Motif*. Notamment, le jeu de widgets *Motif* n'est pas décrit ici.

## 1.3 Philosophie de la bibliothèque

La bibliothèque *UI* a pour principal objectif de faciliter la programmation d'interface *Motif*. Elle est plus spécialisée que les bibliothèques fournies de base avec *Motif* (*Xm*, *Xt* et *Mrm*) mais fournit une interface simplifiée et fortement typée. Elle n'a donc pas vocation à être une bibliothèque totalement générale mais correspond plutôt aux besoins spécifiques d'un domaine d'application. La restriction concerne plus les possibilités de présentation que les fonctionnalités. Par exemple, seules deux fontes sont fournies par *UI*, la fonte par défaut (en général proportionnelle) et une fonte fixe.

Si les *intrinsics* (bibliothèque *Xt*) fournissent la base d'une architecture extensible de composants d'interface, *UI* fixe de façon définitive les widgets disponibles. L'apparition d'une

nouvelle classe de widget, soit propre à *Motif*, soit développée pour ses propres besoins, nécessite une mise à jour de la bibliothèque *UI*. *UI* peut ainsi fournir des interfaces programmatiques supportant plusieurs classes de widgets mais dont le traitement est spécifique à la classe. Par exemple, la même fonction *SHOW\_STRING* appelée pour un widget de classe *XmTextField* retournera le texte maintenu par ce widget, et pour un widget de classe *XmScale* retournera l'image de la valeur courante maintenu par le scale. Ainsi, la bibliothèque *UI* pousse-t-elle plus loin encore que *UIL/Mrm* la séparation entre la forme et la fonction. La forme peut changer (changement de la classe d'un widget) si la fonction ne change pas, c'est-à-dire s'il n'y a pas de changement de protocole, la modification sera transparente pour le programme applicatif. Notamment, le passage d'un objet de type widget à son équivalent gadget ne nécessite pas de modification du programme applicatif, contrairement à la bibliothèque *Xm*. Il faut donc voir la bibliothèque *UI* comme une surcouche spécialisée à des bibliothèques d'IHM publiques ou commerciales. Les besoins applicatifs ayant été déterminés, cette surcouche interface les fonctions sous-jacentes nécessaires et masque au programmeur le plus de détails techniques possibles. L'objectif essentiel de *UI* est ainsi de permettre à des personnes dont l'intérêt principal n'est pas la construction d'IHM, d'écrire néanmoins une partie de leur code.

Si l'extension par héritage n'est pas possible dans *UI*, l'extension par agrégation est en revanche possible. La construction s'effectue par assemblage de widget existant. Le widget racine, qui dans tous les cas est un widget *Motif*, représente le widget<sup>1</sup> construit. De tels widget sont dénommés "widget spécifiques *UI*".

Les ressources d'un widget sont typés directement dans le langage de programmation pour refuser dès la compilation toute association invalide. Par exemple, l'équivalent dans la bibliothèque *UI* de l'appel des intrinsics *XtSetArg(arg, XmNlabelString, I)* pour un widget de classe *XmLabel* sera refusée dès la compilation. Avec les intrinsics, le programme, au mieux, s'arrête lors de l'exécution par un accès invalide à la mémoire et, au pire, fournit un résultat aléatoire. Les *intrinsics* rendent difficile un tel typage, même dans un langage de programmation adapté, car l'architecture est prévue pour être extensible. L'apparition d'une nouvelle ressource dans une classe de widget nécessite également une mise à jour de *UI* ou plus exactement de son vocabulaire.

## 1.4 Fonctionnalités de la bibliothèque *UI*

La bibliothèque *UI* contient le noyau des fonctions nécessaires à la réalisation d'interfaces utilisateur *X-Window/Motif*. Elle peut être utilisée soit directement par le programme applicatif, soit pour la réalisation de fonctions de confort ou de bibliothèques de plus haut niveau. Elle est bâtie à partir des bibliothèques *X toolkit (intrinsics)*, *Mrm (Motif Resource Manager)* et *Xm (Motif)*. Les services qu'elle rend peuvent se classer en six grands groupes :

- gestion des connexions avec le serveur et gestion des événements
- création d'interface statique en *UIL/Mrm*
- création d'interface programmatique
- manipulation des ressources

---

<sup>1</sup>Le composant d'interface construit n'est pas à proprement parler un widget au sens de *Motif*. Toutefois, dans *UI*, on appelle widget, un widget *Motif* ou un widget construit par assemblage.

- navigation de la hiérarchie des widgets
- bibliothèque élémentaire de callbacks

La bibliothèque, dans les services qu'elle offre, ne distingue pas les widgets créés de façon procédurale et les widgets créés à l'aide du langage *UIL*. Notamment, elle permet d'intégrer des composants procéduraux et des composants statiques *UIL*. Elle offre des fonctions de confort pour la manipulation des ressources les plus utilisées, notamment celles qui maintiennent l'état lié aux actions de l'utilisateur.

La partie statique de l'interface utilisateur est en général développée avec un éditeur d'interface (*BX*, *UIM/X* ou *VUIT* par exemple) puis est intégrée à l'application lors de l'exécution en chargeant à l'aide des routines *Mrm*, la description de l'interface contenue dans la base *UID*. La bibliothèque *UI* permet d'aller chercher dans un fichier *UID* une hiérarchie de widget et d'associer à des noms de procédure ou d'identifier leurs adresses correspondantes à l'exécution.

Les intrinsics incluent des fonctions nécessaires à l'écriture de l'application finale et d'autres spécifiques à l'écriture de nouveaux widgets, sans les distinguer clairement. *UI* n'offre que les fonctions nécessaires à l'écriture de l'application finale, que l'on peut classer en cinq catégories :

- gestion de la boucle des événements (*UI.MAIN\_LOOP*, *UI.HANDLE\_NEXT\_EVENT*, ...)
- création et gestion des widgets (*UI.CREATE\_WIDGET*, *UI.MANAGE\_WIDGET*, ...)
- gestion des ressources (*UI.ARG*, *UI.SET\_VALUES*, *UI.GET\_VALUE*, ...)
- gestion de la hiérarchie des widgets (*UI.SHELL\_OF*, *UI.PARENT\_OF*, *UI.NAME\_OF*, ...)
- chargement d'interface *UIL* (*UI.FETCH\_AND\_REALIZE UIL\_INTERFACE*, *UI.REGISTER*, ...)

Les fonctions de confort de la bibliothèque *Xm* sont pour la plupart incluses dans *UI*.

## 1.5 Multi-display, mutli-thread et multi-opérateurs

Un programme qui ouvre plusieurs connections avec un ou plusieurs serveurs X est un programme **mutli-display**.

Un programme qui utilise les taches *Ada* est dit **multi-tache** ou **multi-thread**. L'IHM doit être gérée dans une seule et même tache. Cette limitation est imposée par la bibliothèque *Xlib*.

Un programme multi-display qui ouvre des connections sur au moins deux serveurs X distincts est dit **multi-opérateurs**. Les programmes mutli-opérateurs imposent quelques contraintes qui sont explicités dans le chapitre 3, interfaces multi-opérateurs.

## 1.6 Paquetages et Fichiers

Le paquetage *XLIB\_VOCABULARY* ne fait pas partie de la grappe *USER\_INTERFACE* mais son utilisation est requise. C'est ce paquetage qui définit notamment le type *DISPLAY* associé à l'ouverture d'une connexion avec le serveur X. C'est

un paquetage de vocabulaire commun avec la grappe **GRAPHIC\_PRIMITIVES**. Il correspond au fichier "**X11/Xlib.h**", c'est-à-dire le fichier de vocabulaire de **X-Window**.

La grappe **UI** est constitué des paquetages **USER-INTERFACE**, **UI-VOCABULARY**, **UI-TOOLS**, **UI-BOXES**, **MOTIF-TERMINAL**, **BROWSERS**, **PRESENTATION-MANAGER-INTERFACE** et **PLAY-INTERFACE**.

Le paquetage **UI\_VOCABULARY** contient le vocabulaire des ressources des widgets Motif. C'est l'équivalent du fichier "**Xm/Xm.h**", fichier de vocabulaire de **Motif**.

Le paquetage **USER\_INTERFACE** contient les services élémentaires (creation et gestion de widgets, ...).

Le paquetage **UI\_TOOLS** contient des utilitaires de premier niveau et **UI\_BOXES** des utilitaires de plus haut niveau.

Ces paquetages sont également référencés par un mnémonique. Il est préférable pour la lisibilité de les utiliser lors d'un "**package renames**". Le tableau ci-dessous donne la signification des différents mnémoniques.

mnémonmique	paquetage
<b>UI</b>	<b>USER_INTERFACE</b>
<b>UIT</b>	<b>UI_TOOLS</b>
<b>UIB</b>	<b>UI_BOXES</b>
<b>UIV</b>	<b>UI_VOCABULARY</b>
<b>XV</b>	<b>XLIB_VOCABULARY</b>

Un sous-programme est référencé dans la documentation par <mnémonique-de-paquetage>.<nom-de-sous-programme> soit par exemple "**UI.CREATE\_WIDGET**". La visibilité des paquetages du composant **UI** est indiquée sur la figure ci-dessous. Enfin le mnémonique du paquetage donne son nom au fichier source qui le contient (**ui\_ada**, **uit\_ada**, **uib\_ada**, ...)

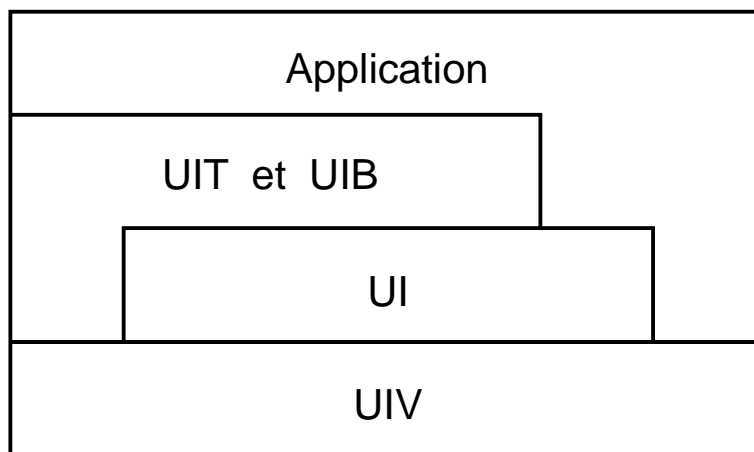


Figure 1-1 Visibilité des paquetage du composant UI

Les autres paquetages de la grappe sont plus spécialisés. Ils sont décrits sommairement ici et plus en détail dans le chapitre "Divers".

Le paquetage **MOTIF\_TERMINAL** (fichier **ui\_mt\_ada**) fournit un terminal virtuel **Motif**, c'est-à-dire un assemblage de widget motif dans lequel l'application lit et écrit exactement comme dans un terminal physique.

Le paquetage générique **BROWSERS** (fichier *ui\_brow\_ada*) permet d'offrir une interface de navigation dans une hiérarchie quelconque.

Le paquetage **PRESENTATION\_MANAGER\_INTERFACE** (fichier *ui\_pmi\_ada*) propose une boîte de dialogue permettant de modifier la base de présentations de **PRESENTATION\_MANAGER**. Le composant se charge à la fois de la création de l'interface et de la réaction aux boutons.

Le paquetage **PLAY\_INTERFACE** crée un composant d'interface, métaphore du magnéto (Play, Pause, Step, Break temporel).

Les paquetages **UI\_AUTOMATED\_TEST** et **UIA\_COMMAND** ne sont pas documentés ici bien que faisant partie de la grappe **USER\_INTERFACE**. Ces deux paquetages sont, pour l'instant, à usage privé. Le paquetage **UI\_STRING** est un paquetage privé au composant **USER\_INTERFACE** qui gère tous les messages multilingues de la grappe.

Des exemples d'utilisation des principales fonctions se trouvent dans le fichier *t\_ui.ada*. Ce fichier contient plusieurs unités de compilation *Ada* nommés *test\_\** et *utest\_\**.

Par abus de langage, dans ce document on utilisera fréquemment "composant **UI**" dans le sens stricte de "grappe **UI**".

## 1.7 Programmes d'exemple

Voici à titre d'exemple deux programmes très simples réalisant la même fonction. Le premier est écrit en *C* à partir de la bibliothèque Motif standard. Le deuxième est écrit en *Ada* à partir de **USER\_INTERFACE**.

### Exemple C

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/Text.h>

int main (argc, argv)
int argc;
char **argv;
{
    Display *Dpy ;
    XtAppContext app_context;
    Widget app_shell, main_window, text ;
    Arg args[5];
    int n;
    char *app_name = "test_user_interface" ;
    char *app_class = "Test_user_interface" ;
    int one = 1 ;

    /* Initialisation du toolkit, Ouverture d'un display et
    Creation de l'application shell */

    XtToolkitInitialize() ;
    app_context = XtCreateApplicationContext() ;

    Dpy = XtOpenDisplay(
        /* app_context */          app_context,
        /* display_string */      "",
        /* application_name */    app_name,
        /* application_class */   app_class,
        /* options */             NULL,
        /* num_options */        (Cardinal)0,
        /* argc */                &one,
        /* argv */                &app_name) ;

    if (Dpy == (Display*)NULL)
    {
        printf("Error opening display") ;
        exit (0) ;
    }

    app_shell = XtAppCreateShell(
        NULL, NULL,
        applicationShellWidgetClass,
        Dpy, (Arg*)NULL, 0) ;

    /* Creation de la fenetre principale */

    n = 0;
    XtSetArg(args[n], XmNwidth, 150); n++;
    XtSetArg(args[n], XmNheight, 125); n++;
    main_window = XtCreateWidget ("", xmMainWindowWidgetClass,
        app_shell, args, n) ;

    /* Creation d'un terminal */

    n = 0;
    XtSetArg(args[n], XmNx,10); n++;
    XtSetArg(args[n], XmNy,20); n++;
    text = XmCreateScrolledText(main_window, "", args, n) ;

    /* Gestion et affichage */

    XtManageChild(text);
    XtManageChild(main_window);

    XtRealizeWidget(app_shell);

    /* traitement des evenements */

    XtAppMainLoop(app_context);
}
```

## Exemple Ada

```

with SUPER_STANDARD ; use SUPER_STANDARD ;
with XLIB_VOCABULARY ;
with UI_VOCABULARY ;
with USER_INTERFACE ;
with UI_TOOLS ;
with UI_BOXES ;

procedure TEST_USER_INTERFACE is

  package UIV renames UI_VOCABULARY ;
  package UI renames USER_INTERFACE ;
  package UIT renames UI_TOOLS ;
  package UIB renames UI_BOXES ;

  DPY : XLIB_VOCABULARY.DISPLAY ;
  APP_SHELL, MAIN_WINDOW, TEXT : WIDGET ;

begin -- TEST_USER_INTERFACE

  --
  -- Initialisation du toolkit
  --
  UI.INITIALIZE_MOTIF_CONTEXT ;

  --
  -- Ouverture d'un display
  --
  begin
    DPY := UI.OPEN_DISPLAY ("test_user_interface", "Test_user_interface", "");
  exception
    when UI.INTERFACE_ERROR =>
      UIT.PUT_AND_CLOSE_ERROR (NULL_WIDGET, "Error opening display") ;
      return ;
  end;

  --
  -- Creation de l'application shell
  --
  APP_SHELL := UI.CREATE_APPLICATION_SHELL(DPY) ;

  --
  -- Creation de la fenetre principale
  --
  MAIN_WINDOW := UI.CREATE_WIDGET(
    name => "mainWindow",
    class => UIV.xmMainWindowWidgetClass,
    parent => APP_SHELL,
    args => (UI.ARG(UIV.XmNx,10), UI.ARG(UIV.XmNy,20))) ;

  --
  -- Creation d'un terminal
  --
  TEXT := UIB.CREATE_SCROLLED_TEXT_OUTPUT("Widget Terminal", MAIN_WINDOW) ;

  --
  -- Gestion et affichage
  --
  UI.MANAGE_CHILD (TEXT) ;
  UI.MAP_INTERFACE (MAIN_WINDOW) ;

  --
  -- traitement des evenements
  --
  UI.APP_MAIN_LOOP ;

end TEST_USER_INTERFACE ;

```

## 1.8 Structure du document

Les chapitres sont organisés par thèmes (création de widget, postage de callback, ...) et non pas en fonction des paquetages. Un chapitre regroupe donc des informations issues de l'ensemble des paquetages de la grappe *USER\_INTERFACE*. Les six premiers chapitres

reprennent les grandes catégories de services indiqués précédemment. Le chapitre sept regroupe la description de services primitifs et utilitaires d'usage plus spécifique. Enfin, le dernier chapitre contient un exemple complet et commenté et propose une démarche de développement de l'IHM.

## **1.9 Audience de la documentation**

Ce document est un manuel du programmeur, il est donc principalement destiné au programmeur de l'IHM ainsi qu'au programmeur qui fait le lien entre l'IHM et *l'API*. On y trouvera toutefois quelques détails concernant l'utilisateur lorsque cela est nécessaire à une bonne compréhension.

L'ordre de lecture n'est pas obligatoirement séquentiel. Notamment, le dernier chapitre "Exemple détaillé" peut être lu en premier lieu avec profit par les personnes qui ont une bonne expérience de *Motif* et d'*Ada*.

## 2. Initialisations et gestion des événements

### 2.1 Initialisations

Les sous-programmes suivants sont décrits dans cette section :

*UI.OPEN\_DISPLAY*

*UI.CLOSE\_DISPLAY*

*UI.GET\_DEFAULT\_DISPLAY*

*UI.SET\_DEFAULT\_DISPLAY*

*UI.INITIALIZE\_MOTIF\_CONTEXT*

*UI.CLOSE\_MOTIF\_CONTEXT*

*UI.IS\_DISPLAY\_IN\_MOTIF\_CONTEXT*

*UIT.INITIALIZE\_AND\_OPEN*

Le composant *USER\_INTERFACE* doit être explicitement initialisé par un appel à la procédure *UI.INITIALIZE\_MOTIF\_CONTEXT*. Tout programme qui propose une IHM motif basé sur *UI* doit appeler (directement ou indirectement par un sous-programme utilitaire) cette procédure. L'initialisation n'est pas effectuée automatiquement à l'élaboration du paquetage *USER\_INTERFACE* car un même programme exécutable peut proposer par un test dynamique une alternative à l'IHM graphique.

Comme pour tous les programmes reposant sur *Motif* ou *X-Window*, il est nécessaire d'ouvrir une ou plusieurs<sup>2</sup> connexions avec un serveur *X* avant de pouvoir afficher l'interface du programme. La fonction *UI.OPEN\_DISPLAY* repose directement sur la fonction *XtOpenDisplay*. La documentation intrinsèque et *Xlib* fournit une description détaillée de cette fonction. On appelle également cette connexion un *DISPLAY*. Un programme peut ouvrir plusieurs connexions avec un même serveur ou avec des serveurs distincts. Les fonctions de gestion des événements ne se limitent pas à une seule connexion mais à tous les *displays* ouverts par un programme. Par contre, la gestion de la modalité des fenêtres est limitée aux fenêtres appartenant à un même *display*. Il est de la responsabilité de l'applicatif de gérer une éventuelle modalité multi-*display*.

Le service de verrouillage d'un *display* (*UIT.LOCK\_DISPLAY*), décrit dans le chapitre interfaces multi-opérateurs, fournit un mécanisme de base pour la réalisation de la modalité multi-*display*.

Deux ou plusieurs composants qui n'ont pas de visibilité programmatique directe peuvent partager un même *display* par l'intermédiaire du paquetage *UI* qui sert de dépositaire entre les différents composants. La procédure *UI.SET\_DEFAULT\_DISPLAY* permet à un composant de déposer le *display* à partager et la procédure *UI.GET\_DEFAULT\_DISPLAY* permet aux autres composants de récupérer ce *display*. La procédure *UI.SET\_DEFAULT\_DISPLAY* ne doit être appelée qu'une seule fois. Il est donc nécessaire d'appeler avant *UI.GET\_DEFAULT\_DISPLAY*, pour s'assurer qu'aucun *display* n'a déjà été déposé. Sous certaines conditions extrêmement précises, détaillées ci-après, il est même possible

---

<sup>2</sup> Si les connexions sont sur des serveurs physiquement distincts, certaines précautions doivent être prises dans l'écriture de l'application. Le chapitre 3, interfaces multi-opérateurs fournit toutes les informations concernant ce problème.

que *UI* se charge de l'ouverture de ce display commun, ainsi que de la création d'un widget *ApplicationShell* associé. Les conditions sont les suivantes :

- une variable d'environnement (un nom logique sous *OpenVMS*) portant le nom *DISPLAY\_NAME* doit contenir le nom du serveur sur lequel on ouvre une connexion. Le display par défaut du process, défini en général par la variable d'environnement *DISPLAY* (nom logique *DECW\$DISPLAY* sous *OpenVMS*) est référençable en donnant la valeur \* ou *default*.
- la variable d'environnement *APPLICATION\_NAME* (nom logique sous *OpenVMS*) détermine le nom d'application de la connexion.
- la variable d'environnement *CLASS\_NAME* (nom logique sous *OpenVMS*) détermine le nom de classe de la connexion.

Seule la première condition est absolument obligatoire. Le display commun est alors disponible pour tous les composants qui le réclament, dès que *UI* est élaboré.

Une connexion peut être fermée par un appel à la procédure *UI.CLOSE\_DISPLAY* qui repose directement sur *XtCloseDisplay*.

Un display qui a été déposé comme display commun ne doit pas être fermé. Lorsqu'un programme se termine les displays non fermés explicitement, sont implicitement fermés par le système d'exploitation ou par le serveur.

Le sous-programme utilitaire *UIT.INITIALIZE\_AND\_OPEN* initialise *UI*, ouvre un display et dépose ce display comme le display par défaut.

### *UI* et le multi-tache

La version *XIIR5* du protocole *X* sur laquelle est bâtie *Motif* et *UI* n'est pas conçue pour fonctionner dans un programme multi-tâche. Les fonctions de bas niveau ne sont pas réentrantes et aucune exclusion mutuelle sur les structures n'est effectuée. Les tâches *Ada* ne doivent pas être utilisées pour la programmation de l'IHM.

## 2.2 Gestion des événements

Les sous-programmes suivants sont décrits dans cette section :

<i>UI.HANDLE_NEXT_EVENT</i>	<i>UI.HANDLE_PENDING_EVENTS</i>
<i>UI.APP_MAIN_LOOP</i>	<i>UI.EXIT_MAIN_LOOP</i>
<i>UI.IS_EVENT_PENDING</i>	
<i>UI.WAIT_FOR_EXPOSE</i>	<i>UIT.GET_ACTION</i>
<i>UIT.REGISTER_ACTION_TRACE</i>	<i>UIT.INVOKE_ACTION_TRACE</i>

Les programmes *Motif* ont une structure événementielle. Après l'ouverture d'une connexion et l'affichage d'une fenêtre principale, ils entrent en général dans une boucle "infinie" de traitement d'événements.

### 2.2.1 Appels bloquants

Cette boucle infinie est implantée par le sous-programme *UI.APP\_MAIN\_LOOP* équivalent de la fonction intrinsic *XtAppMainLoop*. Cette procédure est potentiellement bloquante. Il est possible d'interrompre cette boucle par un appel à la procédure

**UI.EXIT\_MAIN\_LOOP**<sup>3</sup>. Celle-ci est en général appelée dans une callback, par exemple la callback de sortie de l'application.

La procédure **UI.HANDLE\_NEXT\_EVENT** traite un seul événement (le premier de ceux qui sont en attente). Cette procédure est potentiellement bloquante. Pour éviter un blocage, il est possible de se protéger par un appel à la fonction **UI.IS\_EVENT\_PENDING** qui indique la présence d'événement(s) dans file d'attente des événements.

### 2.2.2 Appels non-bloquants

La procédure **UI.HANDLE\_PENDING\_EVENTS** traite tous les événements en attente dans la file d'attente des événements. Cette procédure n'est jamais bloquante, c'est-à-dire qu'elle rend la main immédiatement si aucun événement n'est présent dans la file d'attente.

### 2.2.3 Boucle locale

La boucle locale ne doit pas être utilisée dans les programmes multi-opérateurs. Le chapitre 3 fournit toutes les restrictions s'appliquant aux programmes multi-opérateurs.

Une boîte de dialogue **Motif** peut être modale ou non-modale (modeless). Une boîte de dialogue est modale lorsqu'elle constitue une interaction bloquante. L'utilisateur doit fermer la fenêtre (en validant ou en annulant la saisie) pour pouvoir utiliser le reste de l'IHM. Les boîtes de dialogue modales peuvent être implantées avec une boucle d'événement locale. Il est ainsi inutile d'écrire une callback. La structure d'un sous-programme réalisant une saisie modale par boucle locale se conforme au modèle suivant :

```
-- création et affichage de la boîte modale
UI.MAP_INTERFACE(...) ;
-- boucle locale
loop
  ACTION := ... -- attente d'une action
  exit when ACTION = CANCEL ;
  -- exécuter l'action
  ...
end loop ;
-- fermer la boîte
UI.UNMAP_INTERFACE(...);
```

Ce modèle peut être implanté à partir des sous-programmes de gestion des événements disponibles dans le paquetage **USER\_INTERFACE**, mais des utilitaires réalisant cette fonction sont inclus dans le paquetage **UI\_TOOLS**. La fonction **UIT.GET\_ACTION** attend une action et retourne son rang. Cette fonction est potentiellement bloquante. Elle prend comme argument le widget fils direct du shell associé à la fenêtre modale. Le rang des actions est défini par la client data associée à la callback lors de son dépôt. Pour profiter des services de **UIT.GET\_ACTION**, il faut déposer la callback **UIT.GET\_ACTION\_CB\_PROC** sur les widgets concernés. La client-data associée au moment du dépôt détermine le rang retourné par **UIT.GET\_ACTION**. Certaines fonctions utilitaires de création de widget décrites dans le chapitre 3 utilisent ce mécanisme et se chargent du dépôt de la client-data. Le dépôt de callback et la notion de client-data sont décrits en détail dans le chapitre 5. La fonction **UIT.GET\_ACTION** est réentrante, c'est-à-dire qu'une boîte modale peut afficher une boîte modale implantée elle-aussi par **UIT.GET\_ACTION**. La ressource **XmNuserData** du widget associé à la boîte de dialogue est réservée. Elle sert de mémoire pour la sauvegarde du rang de l'action.

<sup>3</sup>La qualification "infinie" est donc un peu abusive.

Un traite-exception doit être placé dans la boucle pour gérer les erreurs de saisie de l'utilisateur. Les autres erreurs doivent entrainer la sortie de la boucle pour éviter un bouclage infinie.<sup>4</sup>

```
-- création et affichage de la boîte modale
UI.MAP_INTERFACE(...) ;
-- boucle locale
loop
  begin
    ACTION := UIT.GET_ACTION(... -- attente d'une action
    exit when ACTION = CANCEL ;
    -- executer l'action
    ...
  exception
    when INTERFACE_ERROR => PUT_AND_CLOSE_ERROR(...) ;
    when others          => exit ;
  end ;
end loop ;
-- fermer la boîte
UI.UNMAP_INTERFACE(...);
```

---

<sup>4</sup>Il est envisagé que dans une prochaine version, *UI* propose au moins deux exceptions : une pour les erreurs de saisies et une pour les erreurs de programmation.

## 3. Interfaces multi-opérateurs

### 3.1 Introduction

Un programme écrit avec la bibliothèque *UI* peut gérer les interactions de un ou plusieurs opérateurs<sup>5</sup>. Cette possibilité n'est pas spécifique à *UI*, mais elle est simplement héritée de Motif qui lui-même l'hérite de *X-Window*. Cet environnement de fenêtrage graphique a une architecture client/serveur. Gérer plusieurs utilisateurs, c'est simplement ouvrir des **connections** (display) avec plusieurs serveur X. La gestion des requêtes opérateurs ne pose pas de problème particulier car la boucle d'événement Motif (*XtMainLoop*) multiplexe les événements issus de l'ensemble des serveurs. Il n'est pas nécessaire<sup>6</sup> d'écrire une application concurrente (multi-tache ou multi-thread) car les événements sont sérialisés.

L'application doit respecter quelques règles pour garantir un temps de réponse correcte à l'ensemble des utilisateurs et pour maintenir les données de l'application dans un état cohérent:

- pas de sous-programme ou de boucle bloquant dans le corps d'une callback
- pas de traitement long (a priori) dans un corps de callback
- gestion d'un verrou pour l'accès aux ressources partagées

### 3.2 Contraintes d'écriture des callbacks

Le corps d'une callback ne doit pas appeler de sous-programme **bloquant** ou ne doit pas contenir de boucle locale **bloquante**. Un sous-programme ou une boucle est dite bloquante si dans un fonctionnement nominal du logiciel la sortie du sous-programme ou de la boucle locale est conditionnée par un événement extérieur sur lequel le logiciel n'a, à priori, aucun contrôle. Le sous-programme *TEXT\_IO.GET\_LINE* ou le sous-programme *UI.HANDLE\_NEXT\_EVENT* sont des exemples simples de sous-programmes bloquants. La boucle locale d'événement pour la gestion des interactions d'une boîte de dialogue modale, détaillée dans la section 2.2, est un exemple de boucle locale bloquante.

Cette règle d'écriture de callback, comme toute règle, comporte quelques exceptions. Lorsque parmi tous les opérateurs l'un d'eux a un rôle particulier (superviseur), la gestion des interactions de cet opérateur et seulement de cet opérateur, peut faire appel à la boucle locale d'événement. Les événements des autres opérateurs sont traités naturellement au sein de cette boucle locale. Comme le superviseur est unique et que seule la gestion de ses interactions utilise la boucle d'événement, à aucun moment, deux boucles locales ne peuvent coexister. Autre exception à la règle, lorsqu'un opérateur, superviseur ou non, réalise une saisie qui doit

---

<sup>5</sup> A la limite d'aucun opérateur mais ce cas ne présente généralement pas d'intérêt (à l'exception des programmes qui écoutent les événements clavier *KeyPress* pour espionner un utilisateur à son insu).

<sup>6</sup> C'est même interdit dans la version courante (*X11R5*) car la bibliothèque *Xlib* n'est pas multi-thread.

être modale pour l'ensemble des opérateurs. Une boucle locale ou un appel bloquant est alors possible.

Enfin une callback ne doit pas faire appel à un traitement long, car cela risque de ralentir les interactions des autres opérateurs. Une interface graphique qui ne donne pas de retour immédiat à l'opérateur n'est pas agréable à utiliser.

### 3.3 Invalidation des callbacks

Les sous-programmes suivants sont décrits dans cette section:

*UIT.LOCK\_ALL\_DISPLAY*

*UIT.UNLOCK\_ALL\_DISPLAY*

*UIT.LOCK\_DISPLAY*

*UIT.UNLOCK\_DISPLAY*

*UIT.IS\_DISPLAY\_LOCKED*

Le mécanisme de callback facilite grandement la réalisation des programmes événementielles. Il exige par contre de prendre quelques précautions notamment dans le cas des programmes multi-opérateurs. Soit par exemple une callback activée par l'action du premier opérateur dont le corps appelle directement ou indirectement *UI.HANDLE\_PENDING\_EVENTS*:

```
procedure ACTIVATE_CB (W: WIDGET; ...
...
P1;
UI.HANDLE_PENDING_EVENTS;
P2;
...
end ACTIVATE_CB;
```

La callback *ACTIVATE\_CB* peut être interrompue entre l'appel à la procédure *PI* et la procédure *P2*. En effet, la procédure *UI.HANDLE\_PENDING\_EVENTS* lit les événements en attente et peut déclencher un appel de callback. A la limite, il est même envisageable que l'on appelle à nouveau la procédure *ACTIVATE\_CB* (appel réentrant), par exemple par une action d'un deuxième opérateur. Un appel à la procédure *UI.HANDLE\_PENDING\_EVENTS* peut engendrer un changement de contexte. Il est donc nécessaire de prendre quelques précautions dans les sections critiques<sup>7</sup> du programme. On peut citer un exemple de changement de contexte inattendu. Un sous-programme qui écrit au terminal peut avoir ses sorties redirigées dans un terminal Motif et peut être interrompu.

Les sous-programmes décrits dans cette section offre un mécanisme brutale pour garantir un fonctionnement correcte du programme même dans les sections critiques. Le mécanisme est un simple verrou. Avant d'appeler un ou plusieurs services du programme, une callback active le verrou (*UIT.LOCK\_ALL\_DISPLAY*) et après le ou les appels désactive le verrou (*UIT.UNLOCK\_ALL\_DISPLAY*). Le bon fonctionnement du mécanisme requiert que tous les clients respectent le contrat suivant:

Avant d'appeler un service du programme, je m'assure que le verrou n'est pas déjà activée (*UIT.IS\_DISPLAY\_LOCKED*), et si c'est le cas, je n'effectue aucun traitement.

Un petit signal sonore peut être émis à l'aide de la procédure *BEEP\_DISPLAY* pour indiquer à l'opérateur la non-disponibilité de l'application.

Pour résumé une callback suit alors le modèle suivant:

---

<sup>7</sup> Une section critique est une séquence d'instruction qui ne doit pas être interrompue. L'insertion d'un élément dans une liste chaînée est un exemple classique d'opération critique.

```

procedure ACTIVATE_CB(W:WIDGET;...) is
begin
  if IS_LOCKED(DISPLAY_OF(W)) then
    LOCK_ALL_DISPLAY;
    ...corps de la callback...
    UNLOCK_ALL_DISPLAY;
  else
    BEEP_DISPLAY(DISPLAY_OF(W));
  end if;
end ACTIVATE_CB;

```

La séquence **UIT.IS\_DISPLAY\_LOCKED**, **UIT.LOCK\_ALL\_DISPLAY** doit être atomique, c'est-à-dire qu'elle ne doit pas être interruptible. Le corps de callback doit être protégé par un traite-exception. Dans le cas contraire, l'appel à **UNLOCK\_ALL\_DISPLAY** n'est pas garanti et le logiciel peut se trouver dans un état inconsistant. Il est conseillé de factoriser l'ensemble de ce code commun des callbacks dans un generic.

On peut remarquer que ce mécanisme brutale de protection des sections critiques ne requiert pas l'identification des zones d'interruption et des sections critiques du programme. Toutefois, le grand nombre de précautions à prendre montre que l'on a atteint les limites de la technique.

Le bon fonctionnement du mécanisme de verrou ne dépend que du respect d'un contrat, il est important que toutes les grappes susceptibles d'être assemblées respectent ce contrat. Pour que le contrat soit bien clair on l'explique ci-après plus en détail.

On considère que le code du programme est divisé en deux parties:

- un premier composant représentant l'**API** du programme et,
- un deuxième composant, client du premier, comprenant la présentation de l'interface graphique et la gestion des interactions avec l'opérateur.

L'**API** n'appelle pas explicitement de fonctions de **UI** à l'exception notable de **UI.HANDLE\_PENDING\_EVENTS**. Le verrou protège les accès au composant **API**. Dans sa classification des composants G. Booch, distingue ceux qui gèrent eux mêmes la consistance entre les accès concurrents et ceux qui laissent la responsabilité de cette gestion aux clients. Le composant **API**, ici, appartient à la deuxième catégorie, puisque la protection contre les accès concurrent est de la responsabilité des clients et non du composant.

On peut envisager quelques utilisations détournées du mécanisme de verrou, comme la gestion de la modalité multi-display. Motif n'offre pas de support direct pour gérer cette modalité multi-display. Par exemple, si un opérateur doit effectuer une saisie modale vis-à-vis des autres opérateurs (la boucle locale est alors autorisée), on peut utiliser le mécanisme de verrou dans le schéma suivant:

```

LOCK_ALL_DISPLAY;
CREATE_AND_MAP_INTERFACE(...);
loop
  ACTION := GET_ACTION(...
  case ACTION is
    ...
  end case;
end loop
UNLOCK_ALL_DISPLAY;

```

ou dans le schéma suivant:

```

LOCK_ALL_DISPLAY;
-- saisie de chaine(s) dans un terminal Motif
...(GET_STRING(...),...);
UNLOCK_ALL_DISPLAY;

```

Autre utilisation détournée du verrou, le blocage des interactions. Ce blocage peut se faire à un niveau de granularité plus fin, le display (procédures **UIT.LOCK\_DISPLAY** et

**UIT.UNLOCK\_DISPLAY**). Ces fonctions peuvent être utilisé, par exemple, lorsqu'un opérateur ayant un rôle de superviseur, doit être le seul opérateur à interagir avec l'application.

Un client peut se permettre de ne pas respecter le contrat si il sait, par sa connaissance de l'application, que cela ne posera pas de problème. En général la callback qui gère les événements **Expose** d'une **DrawingArea** ne respecte pas le contrat. Il faut garantir que la structure de donnée qui va être traversée pour redessiner la fenêtre est dans un état consistant. Pour cela les procédures qui modifient cette structure de données ne doivent pas être interruptibles. C'est en général le cas car, ces procédures sont purement calculatoires et ne réalisent aucunes entrées/sorties. Un autre cas classique de non respect du contrat est la callback gérant le bouton d'interruption (**Hold** ou **Stop**) d'une animation. En général cette callback positionne, par l'intermédiaire d'un service de l'**API**, un drapeau, qui est scruté par la procédure d'animation. Comme le contrat n'est pas respectée, il faut s'assurer que cette scrutation a lieu à un moment où les données de l'animation n'ont pas le ventre ouvert.

## 4. Création et gestion de widgets

### 4.1 Création générale (**CREATE\_WIDGET**)

Les sous-programmes suivants sont décrits dans cette section :

*UI.CREATE\_WIDGET*

*UI.CREATE\_MANAGED\_WIDGET*

*UI.CREATE\_APPLICATION\_SHELL*

*UI.CREATE\_POPUP\_MENU*

*UI.CREATE\_DRAWING\_AREA*

*UI.SET\_NEXT\_INSERT\_RANK*

*UI.DESTROY\_WIDGET*

*UI.INTERNAL\_WIDGET\_NAME*

Le mot **widget** est aujourd'hui d'usage courant; il est toutefois bon de rappeler sa définition exacte. Pour l'utilisateur, le widget est un objet d'interaction graphique. L'utilisateur peut, par exemple, interagir avec l'application en cliquant sur un bouton. Pour le programmeur d'application, le widget est un composant de base de l'interface graphique. Il construit l'interface d'un programme en assemblant ces composants graphiques prédéfinis. Enfin, pour le programmeur de widget, le widget est l'encapsulation d'une structure de données, qui décrit le composant d'interface, et de procédures qui le manipulent. Le programmeur de widgets matérialise cette structure de données en lui associant une fenêtre dans le système graphique sous-jacent.

*UI.CREATE\_WIDGET* est une fonction générale de création de widget. Elle alloue et initialise simplement la structure qui décrit le widget. Dans l'extrait de code ci-dessous, elle est utilisée pour créer la fenêtre principale d'une application.

```
--  
-- Creation de la fenetre principale  
--  
MAIN_WINDOW := CREATE_WIDGET(  
  name   => "mainWindow",  
  class  => UIV.xmMainWindowWidgetClass,  
  parent => APP_SHELL,  
  args   => (ARG(UIV.XmNx,10), ARG(UIV.XmNy,20))
```

Seuls les widgets prédéfinis par *Motif* peuvent être créés ainsi. Les widgets spécifiques à UI sont créés par des fonctions spécifiques décrites dans le reste de ce chapitre.

Certaines classes de widgets ne sont pas instanciables. Elles servent simplement de super-classes pour partager des ressources ou des procédures entre des classes enfants. Le type énuméré *UIV.ALL\_WIDGET\_CLASS* contient l'ensemble des classes, qu'elles soient instanciables ou non. Le sous-type *UIV.WIDGET\_CLASS* de *UIV.ALL\_WIDGET\_CLASS* indique les classes de widget instanciables. Le sous-type *UIV.META\_WIDGET\_CLASS*, de *UIV.ALL\_WIDGET\_CLASS*, liste les widgets non instanciables.

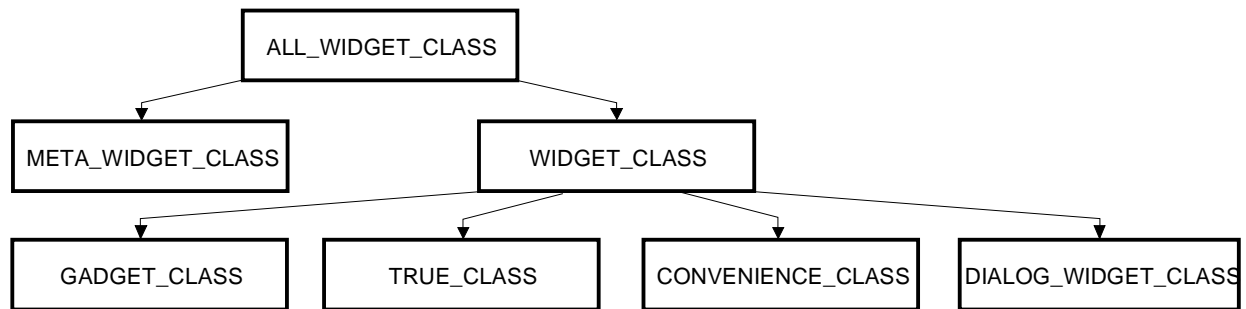


Figure 4-1 Classement des classes de widgets

Les classes de widget instanciables peuvent être classés en quatre groupes. Ils sont décrits dans le tableau ci-dessous.

<b>UIV.GADGET_CLASS</b>	classes définies nommément par Motif. Version simplifiée d'un widget appelée <u>gadget</u> .
<b>UIV.TRUE_WIDGET_CLASS</b>	classes définies nommément par Motif. Vrai widget (et non pas gadget.)
<b>UIV.CONVENIENCE_WIDGET_CLASS</b>	N'est pas en fait une nouvelle classe mais correspond aux classes de widgets dont la valeur par défaut de certaines ressources a été modifiée . Peut également correspondre à un widget include dans une scrolled window Exemples: <ul style="list-style-type: none"> <li>- le MenuBar qui est un rowColumn avec une orientation verticale à la place d'horizontale.</li> <li>- la scrolledList ou le scrolledText</li> </ul>
<b>UIV.DIALOG_WIDGET_CLASS</b>	widgets qui se chargent lors de leurs instanciations, de la création d'un widget père XmDialogShell. Exemple: XmBulletinBoardWidgetClass

Contrairement à la fonction *XtCreateWidget* des intrinsics, **UI.CREATE\_WIDGET** permet de créer les widgets Motif de confort et de dialogues.

Seuls deux classes de widget doivent obligatoirement être créées par une fonction spécifique, l'application shell (fonction **UI.CREATE\_APPLICATION\_SHELL**) et le popup menu shell (fonction **UI.CREATE\_POPUP\_MENU**).

L'extrait de code ci-dessous montre la création de l'application shell d'une application :

```

--
-- Creation de l'application shell
--
APP_SHELL := CREATE_APPLICATION_SHELL(DPY) ;

```

Dans cet autre extrait, on montre la création d'un popup menu contenant des boutons :

```

POPUP := CREATE_POPUP_MENU (
    name=> "Popup",
    parent=> DRAWING_AREA) ;

SHELL := PARENT_OF(POPUP) ;
UI.POST_CB (SHELL, (1 => CB_ARG(XmNpopupdownCallback, ...))) ;

for ... loop
    W := CREATE_WIDGET ("", xmPushButtonWidgetClass,
        POPUP, (1=>ARG(XmNlabelString, ...)),
        CALLBACKS => (1 => CB_ARG(XmNactivateCallback, ...)));
end loop ;

```

## 4.2 Gestion d'un widget

Les sous-programmes suivants sont décrites dans cette section :

<i>UI.MANAGE_CHILD</i>	<i>UI.UNMANAGE_CHILD</i>
<i>UI.MANAGE_CHILDREN</i>	<i>UI.UNMANAGE_CHILDREN</i>
<i>UI.MANAGE_ALL_DESCENDANTS</i>	
<i>UI.MAP_WIDGET</i>	<i>UI.UNMAP_WIDGET</i>
<i>UI.MAP_INTERFACE</i>	<i>UI.UNMAP_INTERFACE</i>
<i>UI.DELETE_INTERFACE</i>	
<i>UI.REALIZE_WIDGET</i>	<i>UI.UNREALIZE_WIDGET</i>

Les intrinsics distinguent l'affichage d'un widget (*XtMapWidget*) de la gestion de la géométrie (*XtManageChild*). Les widgets qui acceptent des enfants sont sous-classes de la classe "composite". Un widget est dit "géré" lorsque son père prend en compte sa géométrie. Par défaut, un widget qui est géré est également affiché. Ce défaut peut être changé en modifiant la valeur de la ressource *XmNmappedWhenManaged*. La procédure *UI.MANAGE\_CHILD* indique que le widget passé en argument doit être géré par son père. La procédure *UI.MANAGE\_CHILDREN* indique qu'une liste de widgets frères et soeurs doit être gérée par leur père. La procédure *UI.MANAGE\_ALL\_DESCENDANTS* gère de bas en haut toute une hiérarchie de widgets à partir d'une racine quelconque.

Inversement les sous-programmes *UN.UNMANAGE-CHILD* et *UI.UNMANAGE-CHILDREN* indiquent que le ou les widget(s) en question ne sont plus gérés. Cela aura évidemment une conséquence sur la disposition géométrique des widgets frères.

En général, il n'est pas nécessaire d'afficher explicitement un widget puisque l'affichage est une conséquence de la gestion d'un widget. Toutefois, lorsque cela est nécessaire, cette fonction est réalisée par la procédure *UI.MAP\_WIDGET*. Le désaffichage est réalisé par la fonction *UI.UNMAP\_WIDGET*.

Une fenêtre peut être affichée à partir d'un widget quelconque appartenant à la fenêtre. De même, on peut désafficher ou détruire une fenêtre. Ces fonctions sont réalisées respectivement par les procédures *UI.MAP\_INTERFACE*, *UI.UNMAP\_INTERFACE*, *UI.DELETE\_INTERFACE* qui prennent comme argument n'importe quel widget de la hiérarchie. Par exemple, la callback activée d'un bouton *Cancel* d'une boîte de dialogue peut appeler *UI.UNMAP\_INTERFACE* avec le widget reçu comme argument (qui est associé au bouton). Dans l'extrait de code ci-dessous, on gère en profondeur une hiérarchie de widgets et on affiche la fenêtre principale de l'application :

```
--
-- affichage
--

UI.MANAGE_ALL_DESCENDANTS (MAIN_WINDOW) ;
UI.MAP_INTERFACE (MAIN_WINDOW)
```

Rappelons que la fonction *UI.CREATE\_WIDGET* crée simplement une structure qui décrit le widget. La création de la fenêtre *X* associé au widget, c'est-à-dire la matérialisation du widget, a lieu lorsque la fonction *UI.REALIZE\_WIDGET* est appelée. Lorsqu'un widget est créé dans une hiérarchie qui est déjà réalisée, la matérialisation de ce widget est implicite.



fréquent de l'utiliser comme zone de dessin. Le dessin s'effectue en général avec la bibliothèque *Xlib*. Toutefois, toute bibliothèque graphique qui s'appuie sur *X-Window* peut convenir. C'est le cas notamment de la grappe *GRAPHIC\_PRIMITIVES* de la couche *G* d'*ESCADRE*.

A bas niveau, deux sous-programmes peuvent servir à créer un widget *drawing area*. Le premier est bien sûr la fonction *UI.CREATE\_WIDGET* en donnant au paramètre formel *CLASS* la valeur *xmDrawingAreaWidgetClass*. Toutefois, lorsque la *drawing area* sert de zone de dessin on utilisera de préférence la fonction *UI.CREATE\_DRAWING\_AREA*. Celle-ci se charge de la compression des événements "*Expose*". La callback déposée par le paramètre formel *EXPOSE* est appelé à chaque fois qu'une restauration du contenu de la *drawing area* est nécessaire. Avant de redessiner il faut, soit effacer complètement la *drawing area*, soit effacer la région X exposée et clipper le dessin par cette région. Le troisième paramètre de la callback *EXPOSE* correspond à l'identificateur de la région X concernée. Pour ne pas introduire un nouveau type de callback, *EXPOSE* est du même type que toutes les callbacks programmatiques de *USER\_INTERFACE*. Il est donc malheureusement nécessaire de faire une "*UNCHECKED-CONVERSION*" de *UI.ANY-CALLBACK-STRUCT* vers le type *XV.REGION\_X* pour récupérer l'identificateur de la région. Bien que le composant *UI* soit le créateur de la région concernée, c'est à la callback, après avoir redessiné, de détruire la région. Noter que lorsque l'on utilise la bibliothèque *GRAPHIC\_PRIMITIVES*, le sous-programme *GP.UNSET\_CLIP\_REGION* se charge de cette destruction.

La procédure *REDRAW\_VIEW\_CB* ci-dessous est extraite du composant *GRAPHIC\_TOOLS* (paquetage *GRAPHIC\_VIEW*). Cette procédure est déposée comme "Expose callback" dans sous-programme *UI.CREATE\_DRAWING\_AREA*. Elle fait appel à la procédure *REDRAW\_VIEW* qui illustre l'utilisation des primitives de troncature (clipping) du *GRAPHIC\_PRIMITIVES*.

```

procedure EXPOSE_CB (
  W: in UI.WIDGET ; CLIENT_DATA : in C_LONG ; CALL_DATA : in UI.ANY_CALLBACK_STRUCT)
is
  function TO_REGION is new UNCHECKED_CONVERSION (UI.ANY_CALLBACK_STRUCT, GP.REGION_X) ;
  REGION : constant GP.REGION_X := TO_REGION(CALL_DATA) ;
  V      : constant REF_VIEW := TO_REF_VIEW(CLIENT_DATA) ; -- retrouve les données de la
              -- vue à partir de la client data
begin
  REDRAW_VIEW (V, REGION) ;
end EXPOSE_CB ;
pragma EXPORT_PROCEDURE (EXPOSE_CB) ;
...
procedure REDRAW_VIEW (
  V : in REF_VIEW ; REG : in GP.REGION_X) is
begin
  GP.SET_CLIP_REGION(V.WIN, REG) ;
  GP.CLEAR_WINDOW (V.WIN) ;
  -- redessiner la vue ...
  GP.UNSET_CLIP_REGION(V.WIN) ;
exception
  when others => ...
end REDRAW_VIEW ;

```

A plus haut niveau, deux sous-programmes peuvent servir à créer une fenêtre complète contenant une *drawing area*. Ceux-ci faisant appel à la fonction *UI.CREATE\_DRAWING\_AREA*, les événements *Expose* sont donc compressés. Le premier, *UIT.CREATE\_DRAWING\_AREA\_WITH\_BUTTONS* ajoute une rangée de boutons figée sous la zone de dessin.

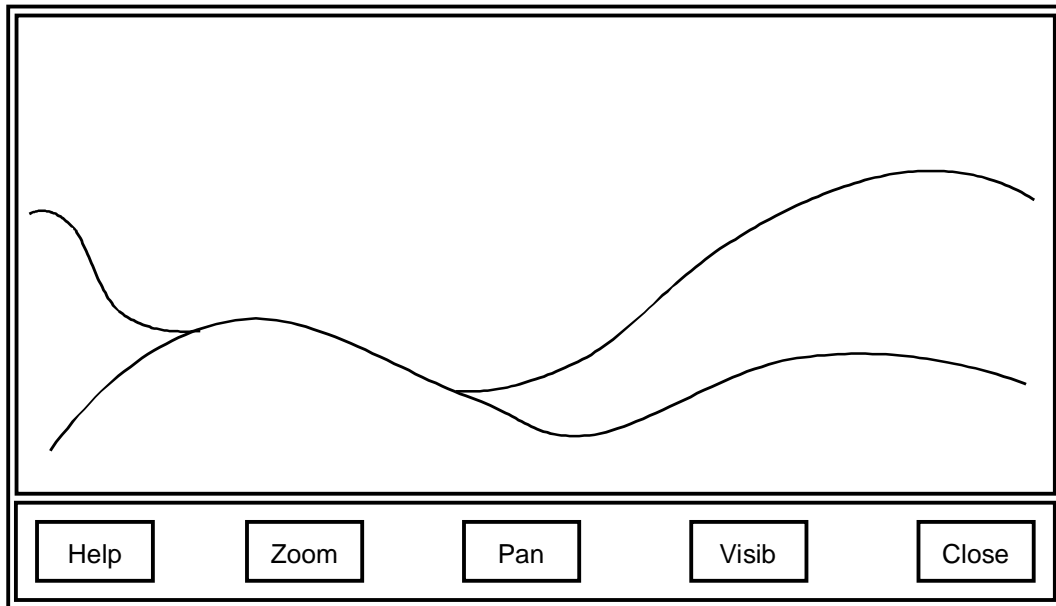


Figure 4-2 UIT.CREATE\_DRAWING\_AREA\_WITH\_BUTTONS

Le deuxième, **UIT.CREATE\_DRAWING\_AREA\_WITH\_MENU**, ajoute une barre de menu contrôlée par le programmeur au-dessus de la zone de dessin. La barre de menu doit être remplie après l'appel à cette fonction. L'extrait de code ci-dessous illustre l'utilisation de cette dernière.

```

declare
WINDOW, MENU_BAR, DRAWING_AREA : WIDGET := NULL_WIDGET ;
DPY : constant DISPLAY := UI.GET_DEFAULT_DISPLAY ;
COLOR_BACKGROUND_X : constant PIXEL_X
:= GET_VALUE(UI.APPLICATION_SHELL_OF(DPY), XmNbbackground);
function EXPOSE_CB_PROC is new CB_PROC("EXPOSE_CB", EXPOSE_CB) ; -- defini ci-dessus
EXPOSE_PROC : constant CALLBACK := UI.CB(EXPOSE_CB_PROC, C_LONG'(0)) ;
CASCADE_NAMES : constant UN.NAME_LIST(1..) := ... ;
function MENU_LABELS (I : Positive) return UN.NAME_LIST ... ;
MENU_PROC : constant CB_NAMED_ADDRESS := ...
begin
UIT.CREATE_DRAWING_AREA_WITH_MENU(
    w      => WINDOW,          -- mode out, reference du dialogue cree
    bar    => MENU_BAR,        -- mode out, menu bar a remplir
    win    => DRAWING_AREA,    -- mode out, drawing area pour dessiner
    dpy    => DPY,             -- mode in, cree sur ce display
    name   => "diagram_view",  -- mode in, nom du widget shell cree
    back   => COLOR_BACKGROUND_X, -- couleur de fond de la drawing area
    x      => 1,               -- position de
    y      => 1,               -- la fenetre
    width  => 500,             -- largeur et
    height => 300,             -- hauteur de la drawing area
    expose => EXPOSE_PROC,     -- callback pour les expose
    mapped => True,            -- fenetre affichee
    shellargs => (UI.ARG(UIC.XMNTITLE, "Diagram"), -- arguments du shell
                 UI.ARG(UIC.XMNICONNAME, "Diag"));

    --
    -- remplissage de la barre de menu
    --
    for I in CASCADE_NAMES'Range loop
        MENU := CREATE_SIMPLE_PULLDOWN(CASCADE_NAMES(I), MENU_BAR,
                                       MENU_LABELS(I), MENU_PROC);
    end loop ;
end ;

```

## 4.4.1 Le widget rowColumn

### 3.7.1 Fonctions SIMPLE

Le widget RowColumn est un widget caméléon qui peut se conjuguer à plusieurs temps. Il peut prendre l'aspect, entre autre, d'une barre de menu, d'un menu déroulant, d'une radio box ou d'une check box. La fonction **UIT.CREATE\_SIMPLE\_ROWCOLUMN** est une fonction

générale de création de rowColumn contenant des boutons quelconques. Il existe également des fonctions dédiées à chaque type de rowColumn. Celles-ci existent sous deux formes, la version "SIMPLE" et la version non-"SIMPLE", soit par exemple pour une check box **UIT.CREATE\_SIMPLE\_CHECK\_BOX** et **UIT.CREATE\_CHECK\_BOX**. Toutes ces fonctions sont décrites individuellement dans les sections qui suivent.

Les fonctions "SIMPLE" ont la caractéristique commune suivante. Pour ces fonctions, une seule et même procédure est déposée comme callback de réaction à l'activation des boutons du rowColumn. A l'appel de ces fonctions on ne précise pas explicitement la client-data. A la création de chacun des boutons, la même callback est déposée, avec une client-data qui est le rang du bouton dans la liste. Dans l'extrait de code ci-dessous, on montre la création d'un pull-down menu avec une fonction "SIMPLE" et la callback associée. Certains détails de cette exemple, notamment le "pragma **EXPORT\_PROCEDURE**" ne sont compréhensibles qu'après la lecture du chapitre "Callbacks et event handlers".

```
-- declaration en avant de la callback pour le dépôt lors de la création de l'IHM
procedure FILE_MENU_CB (W: WIDGET; CLIENT_DATA: C_LONG; CALL_DATA: ANY_CALLBACK_STRUCT) ;
pragma EXPORT_PROCEDURE (FILE_MENU_CB) ;
function FILE_MENU_CB_PROC                                -- cette fonction retourne une
référence
    is new CB_PROC("FILE_MENU_CB", FILE_MENU_CB) ;    -- typée sur la procedure FILE_MENU_CB

-- procedure de creation de l'IHM
procedure CREATE_IHM (...) is
...
    FILE_LABELS : constant UN.NAME_LIST := (1=> UN.DEFINE("Open"),
                                             2=> UN.DEFINE("Save"),
                                             3=> UN.DEFINE("Close")) ;

begin
...
    PULLDOWN_MENU := CREATE_SIMPLE_PULLDOWN (
        name=>    "File"
        parent=>  MENU_BAR,
        labels=>  FILE_LABELS,
        cb=>      FILE_MENU_CB_PROC) ; -- partagée entre tous les
-- choix
...
end ;

-- callback de reaction aux boutons
procedure FILE_MENU_CB (W: WIDGET; CLIENT_DATA: C_LONG; CALL_DATA: ANY_CALLBACK_STRUCT) is
begin
    case CLIENT_DATA is
        when 1 => -- bouton 'File/Open'                -- ce sont ces valeurs (1, 2 et 3)
            API.OPEN (... ) ;
        when 2 => -- bouton 'File/Save'                -- qui sont calculées et
            API.SAVE (... ) ;
        when 3 => -- bouton 'File/Close'              -- déposées par la procédure
            API.CLOSE (... ) ;
        when others =>
            null ;
    end case ;
exception -- aucune exception ne doit être levé dans cette partie
    when INTERFACE_ERROR => PUT_AND_CLOSE_ERROR (... ) ;
    when others          => ... ;
end FILE_MENU_CB ;
```

Les fonctions non-"SIMPLE" laissent libre choix de la client-data. Elles prennent comme paramètre un tableau de callbacks qui doit être en correspondance (taille et index) avec le paramètre formel **LABELS**. Elles permettent une utilisation normale de la client-data.

## 4.4.2 Pull-down

Un menu pull-down est constitué d'un bouton cascade d'un rowColumn de type pull-down et de boutons. La figure ci-dessous déroule un menu pull-down "File" d'une application.

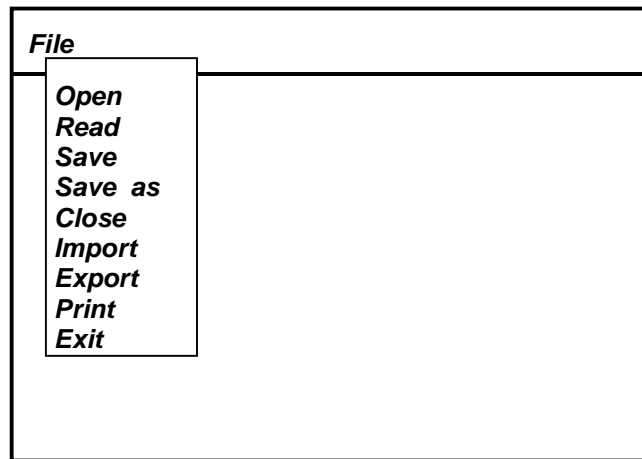
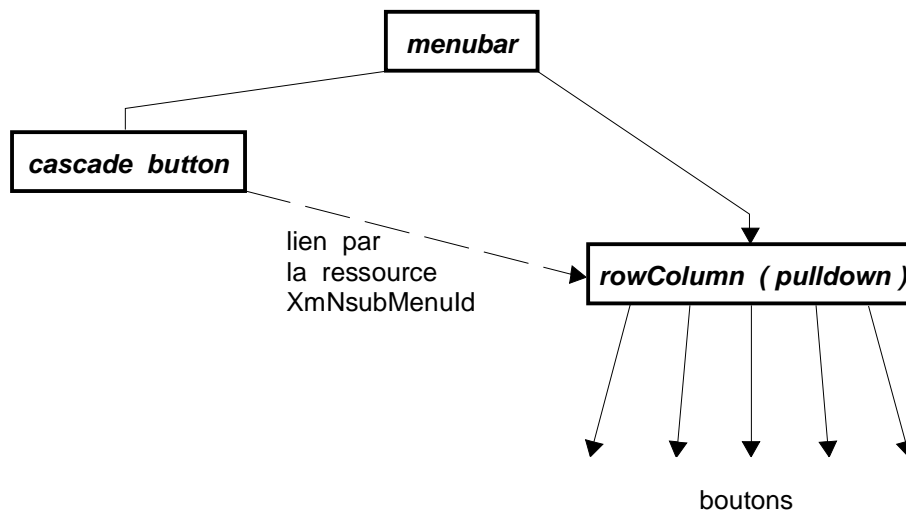


Figure 4-3 Exemple de pulldown menu

La hiérarchie des widgets est la suivante<sup>8</sup>:



—————▶ lien de filiation      - - - - -▶ lien par ressource

Figure 4-4 Hiérarchie de widget d'un pulldown menu

Le tableau ci-dessous indique, pour les sous-programmes créant tout ou partie d'un pulldown menu, leur fonction exacte.

fonction	description
<i>UIT.CREATE_SIMPLE_PULLDOWN</i>	créé un cascade, le pulldown et ses boutons
<i>UIT.CREATE_PULLDOWN</i> <i>UIT.CREATE_SIMPLE_PULLDOWN</i>	créé un cascade et le pulldown
<i>UIT.CREATE_SIMPLE_PULLDOWN_BUTTONS</i> <i>UIT.CREATE_PULLDOWN_BUTTONS</i>	créé les boutons d'un pulldown
<i>UIB.CREATE_CASCADE_MENU</i>	créé le pulldown et les boutons

<sup>8</sup>Les puristes remarqueront que le widget shell père du rowColumn (pulldown) n'a pas été représenté car celui-ci est créé implicitement.

### 4.4.3 Rangée de boutons

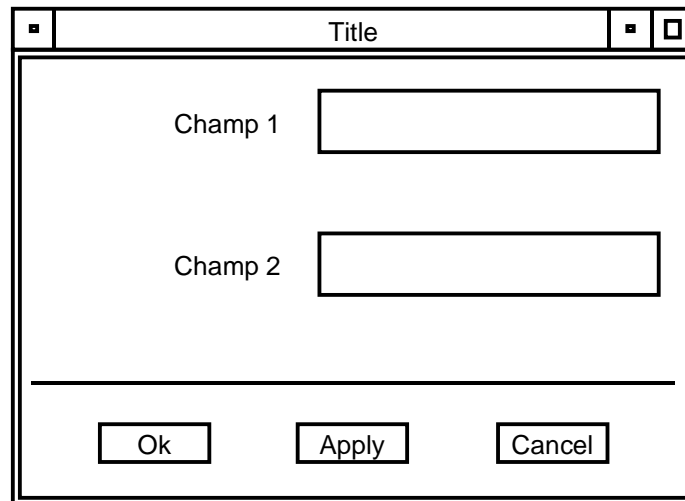


Figure 4-5 Fonction *CREATE\_SIMPLE\_BUTTONS*

Les fonctions *UIT.CREATE\_SIMPLE\_BUTTONS* et *UIT.CREATE\_BUTTONS* permettent de créer une rangée de boutons régulièrement espacés. Lorsque la fenêtre qui les contient est retaillée, ceux-ci sont redispesés pour maintenir un espacement constant. La figure ci-dessous illustre l'aspect visuel des composants d'interfaces créés par ces deux sous-programmes. La rangée de boutons "Ok, Apply, Cancel" de la boîte de dialogue de cette figure est créée par la fonction *UIT.CREATE\_SIMPLE\_BUTTONS*.

### 4.4.4 Radio box et check box

Le widget radio box est un widget *rowColumn* configuré pour contenir des toggle boutons dont un seul sera selectionné à un instant donné. Ce widget est créé par la fonction utilitaire *UIT.CREATE\_RADIO\_BOX* ou par la fonction *UI.CREATE\_WIDGET* avec la classe *xmRadioBoxWidgetClass* ou *xmSimpleRadioBoxWidgetClass*. La figure ci-dessous montre l'aspect visuel d'une radio box.

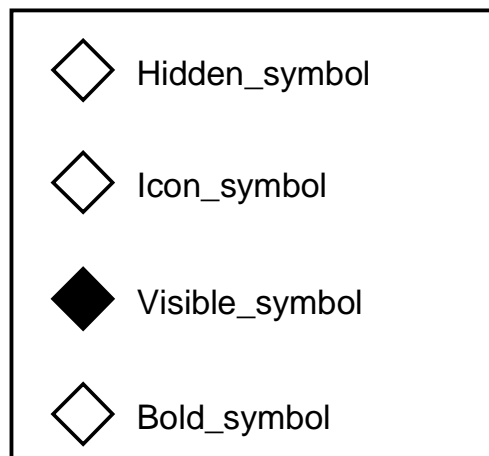


Figure 4-6 Radio box

Le widget check box est similaire au widget radio box. Plusieurs toggle buttons peuvent être sélectionnés. Ce widget est crée par la fonction utilitaire **UIT.CREATE-SIMPLE-CHECK-BOX** ou par la fonction **UI.CREATE\_WIDGET** avec la classe **xmCheckBoxWidgetClass** ou **xmSimpleCheckBoxWidgetClass**. La figure ci-dessous montre l'aspect visuel d'une check box.

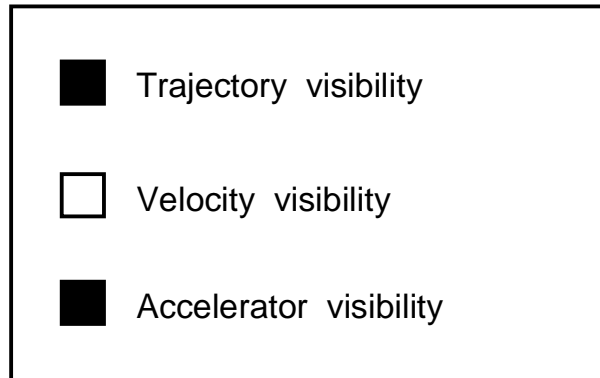


Figure 4-7 Check Box

La version non-"SIMPLE" de ces deux fonctions ajoute deux boutons "All" et "None". Le bouton "All" sélectionne l'ensemble des boutons. Le bouton "None" désélectionne l'ensemble des boutons (voir figure ci-dessous).

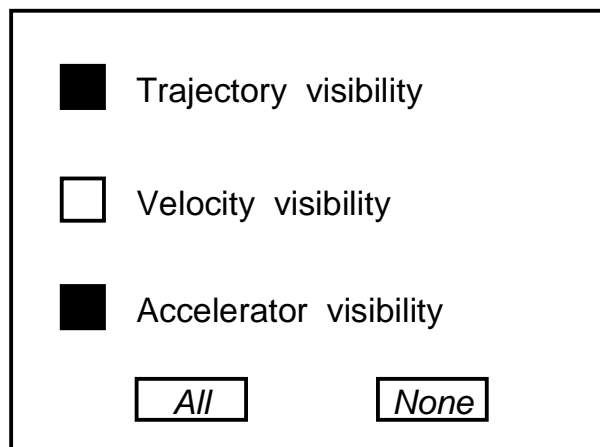


Figure 4-8 Check Box avec boutons "All" et "None"

Dans ce cas, le widget retourné par la fonction est toujours le rowColumn contenant les toggle buttons. Le widget qui contient l'ensemble du composant d'interface est le père de ce widget rowColumn (obtenu par la fonction **UI.PARENT\_OF**). C'est bien sûr ce widget père qui doit être utilisé lorsque l'on définit des attachements sur ce composant d'interface.

Enfin, le paquetage **UI\_BOXES** offre un radio box étendu qui permet d'associer un champ de saisie quelconque à chacun des toggle buttons. La famille de fonctions associées à ce widget sont décrites dans le tableau ci-dessous.

<b>UIB.CREATE_RADIO_BOX</b>	créé un radio box étendu (radio box simple + champs facultatifs associés)
<b>UIB.SET_RADIO_BOX_EQUAL_WIDTH</b>	aligne les champs
<b>UIB.SHOW_RADIO_BOX_SELECTED_RANK</b>	retourne le rang du bouton sélectionné
<b>UIB.RADIO_BOX_FIELDS_OF</b>	retourne la liste des champs
<b>UIB.RADIO_BOX_LABELS_OF</b>	retourne la liste des labels

<b><i>UIB.RADIO_BOX_TOGGLES_OF</i></b>	retourne la liste de toggles
<b><i>UIB.RADIO_BOX_FIELD_OF</i></b>	retourne un champ d'un rang quelconque

La figure ci-dessous montre l'aspect visuel de la radio box étendue définie par **UIB**. Sur cette figure, un seul champ a été créé.

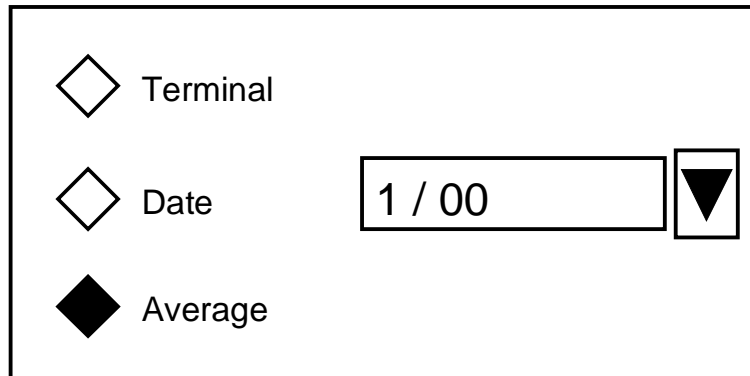


Figure 4-9 *UIB.CREATE\_RADIO\_BOX*

### 3.14.1 Option Menu et Browse Menu

L'option menu est un widget très utilisé qui fait partie du jeu de widgets *Motif*. Il peut être créé par la fonction utilitaire **UIT.CREATE\_OPTION\_MENU** ou par la fonction **UI.CREATE\_WIDGET** avec la classe **UIV.xmOptionMenuWidgetClass** ou **xmSimpleOptionMenuWidgetClass**. La fonction utilitaire répartit les choix sur plusieurs colonnes si nécessaire.

Le widget browse menu est similaire à un option menu. Ce widget ne fait pas partie du jeu de widgets *Motif*. Il présente les choix sous une forme hiérarchique. La hiérarchie est calculée dynamiquement à partir de la liste séquentielle des choix. Cette liste doit être triée, par exemple à l'aide de la fonction **UIT.SORT**. Le browse menu est créé par la fonction **UIT.CREATE\_BROWSE\_MENU** en donnant la valeur **True** au paramètre **LABEL** et **False** au paramètre **POPUP**. Cette fonction fait appel aux fonctions **UIT.CREATE\_CASCADE\_OPTION\_MENU** et **UIT.CREATE\_PULLDOWN\_MENU** pour implanter le browse menu.

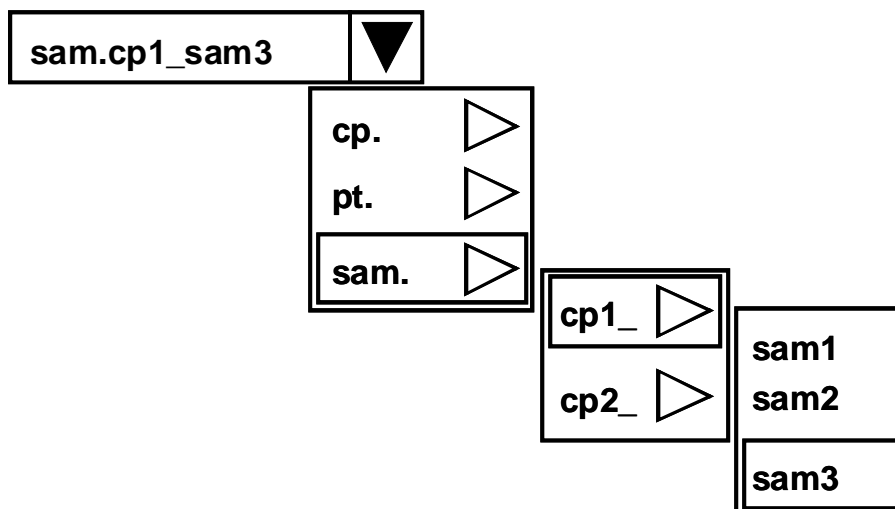


Figure 4-10 *Browse menu*

La figure ci-dessous montre l'aspect visuel d'un browse menu. Celui-ci n'est rien d'autre qu'un option menu dont les choix sont présentés hiérarchiquement.

La fonction de création d'un browse menu, peut également servir à créer l'aide d'un champ de saisie (paramètre **LABEL** à **False**) ou un popup menu hiérarchique (paramètre **POPUP** à **True**). La callback déposée sous le paramètre formel **CB** est appelée lorsqu'un **pushButton** est activé.

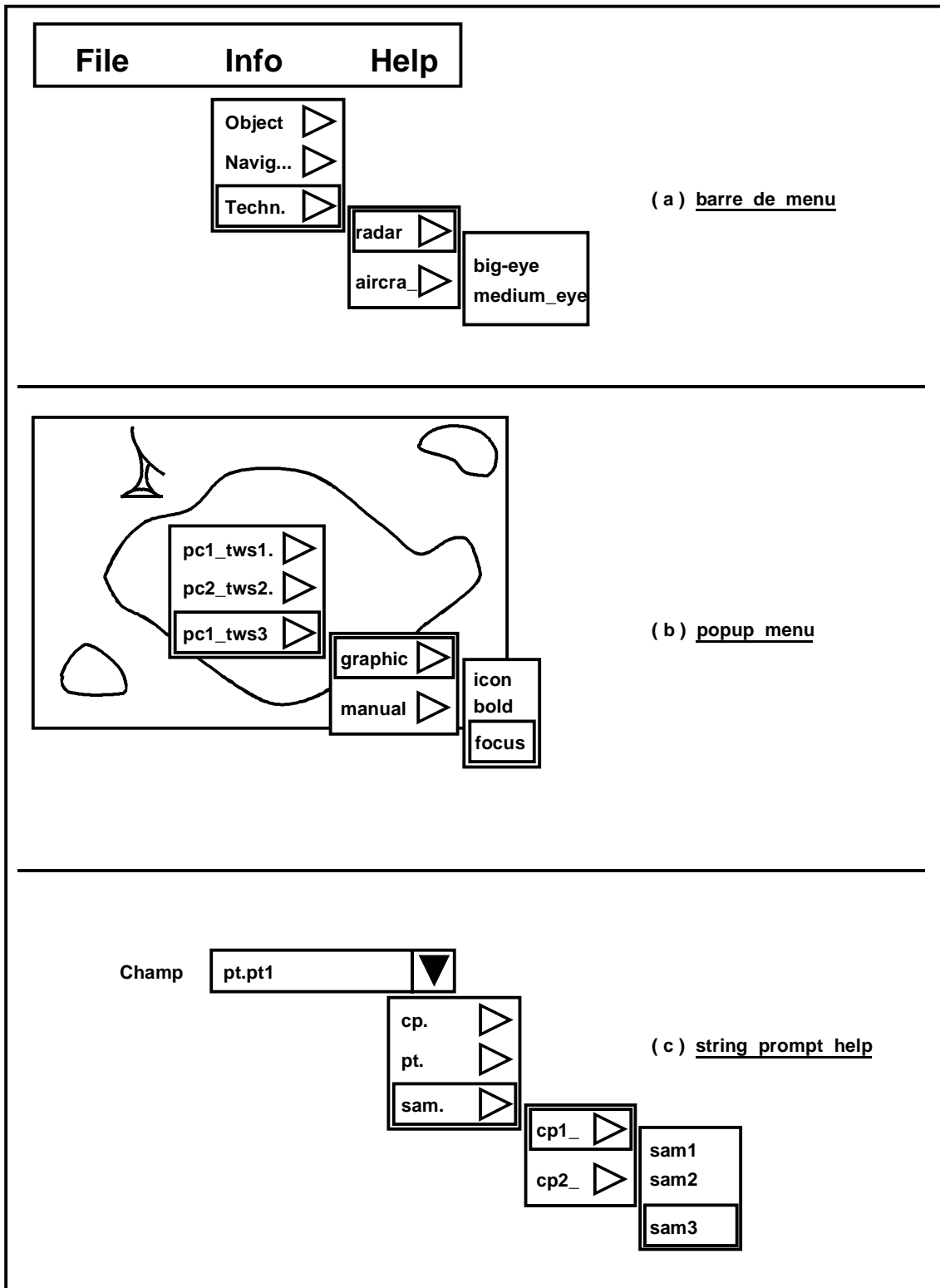


Figure 4-11 Trois variations d'un browse menu

La fonction **UIT.BROWSE\_MENU\_NAME\_OF** retourne le nom complet d'un choix quelconque. La fonction **UIT.BROWSE\_MENU\_LEVEL\_OF** retourne la profondeur

hiérarchique d'un choix quelconque. La fonction **UIT.BROWSE\_MENU\_OF** permet d'identifier le menu à partir de l'un de ces choix.

Les choix d'un browse menu peuvent être gérés de façon dynamique (paramètre **DYNAMIC** à **True**) ou de façon statique (paramètre **DYNAMIC** à **False**). La callback déposée sous le paramètre formel **CASCADE\_CB** est appelée lorsqu'un browse menu dynamique doit être étendu. Celle-ci fait appel à la fonction **UIT.APPEND\_BROWSE\_MENU** pour étendre le menu. Lorsqu'on ne connaît pas le texte de l'aide à la création du widget browse menu, on peut appeler la fonction **UIT.CREATE\_EMPTY\_BROWSE\_MENU**. On devra de toute façon initialiser l'aide par un appel à **UIT.SET\_PROMPT\_LABELS** ou à **UIT.APPEND\_BROWSE\_MENU** avant l'affichage de la boîte. Les composants internes d'un browse menu, dont certains sont facultatifs, sont retournés par les fonctions **UIT.LABEL\_CASCADE\_OF** (label du browse menu), **UIT.DRAW\_CASCADE\_OF** (flèche inversée) et **UIT.BROWSE\_MENU\_OF** (le pull-down menu).

## 4.5 Saisie de type simple

**Motif** ne fournit aucun widget ou utilitaire pour construire des boîtes de saisies de données atomiques. Le composant **UI** propose 3 utilitaires pour ce besoin:

- Les prompts widgets (**UIT.\*PROMPT\***, **UIB.\*PROMPT\***)
- Les prompts dialogues à champ unique (**UIT.GET\_\***)
- Le dialogue générique (**UIB.UTIL\_IO\_MOTIF**)

Les prompts widgets peuvent servir à construire tout ou partie d'une boîte de saisie contenant une liste de champs élémentaires. Tous les types prédéfinis du langage **Ada** sont supportés.

Les "prompts dialogues" créent des boîtes pour la saisie d'un champ unique. Elles utilisent la technique de la boucle locale, ce qui implique qu'elles sont réservées aux interactions modales.

Le dialogue générique offre un sous-ensemble des fonctionnalités des prompts widgets, mais masque les détails de construction de la boîte **Motif**.

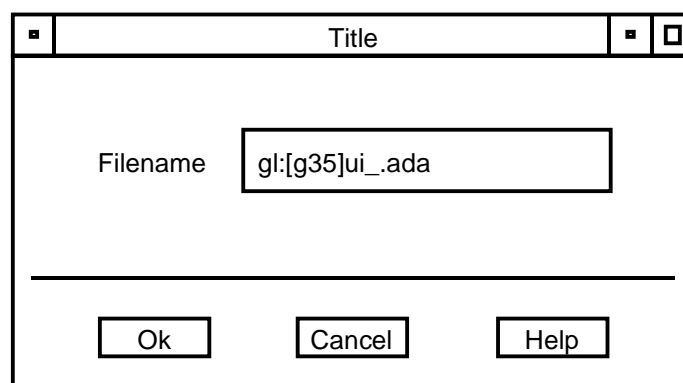


Figure 4-12 Boîte de dialogue **UIT.GET\_STRING**

La figure ci-dessous montre l'aspect de la boîte de dialogue créée par la fonction **UIT.GET\_STRING**. Dans la même famille, la fonction **UIT.GET\_FILE** est d'un usage fréquent. Elle permet la sélection d'un fichier à l'aide d'une **SelectionBox**. Ces dialogues sont modaux.

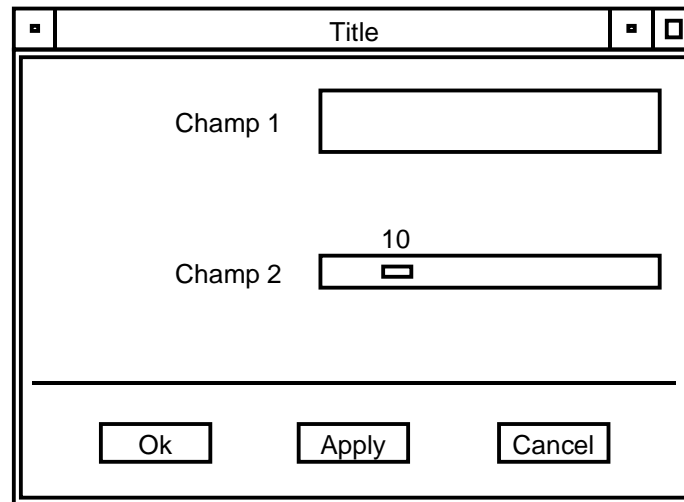


Figure 4-13 Prompt dialogue

La figure ci-dessous montre l'aspect d'une boîte de dialogue construite à partir des fonctions de création de champs de saisie. La même boîte pourrait également être construite à partir du composant générique *UTIL\_IO\_MOTIF*.

Le décodage des champs s'effectue à l'aide des fonctions *UI\*.SHOW\_\** détaillées dans la section "Fonctions utilitaires.widgets de saisie" du chapitre "ressources". Chacun de ces constructeurs de dialogues est décrit plus en détail dans l'une des sections suivantes.

#### 4.5.1 Prompt widget

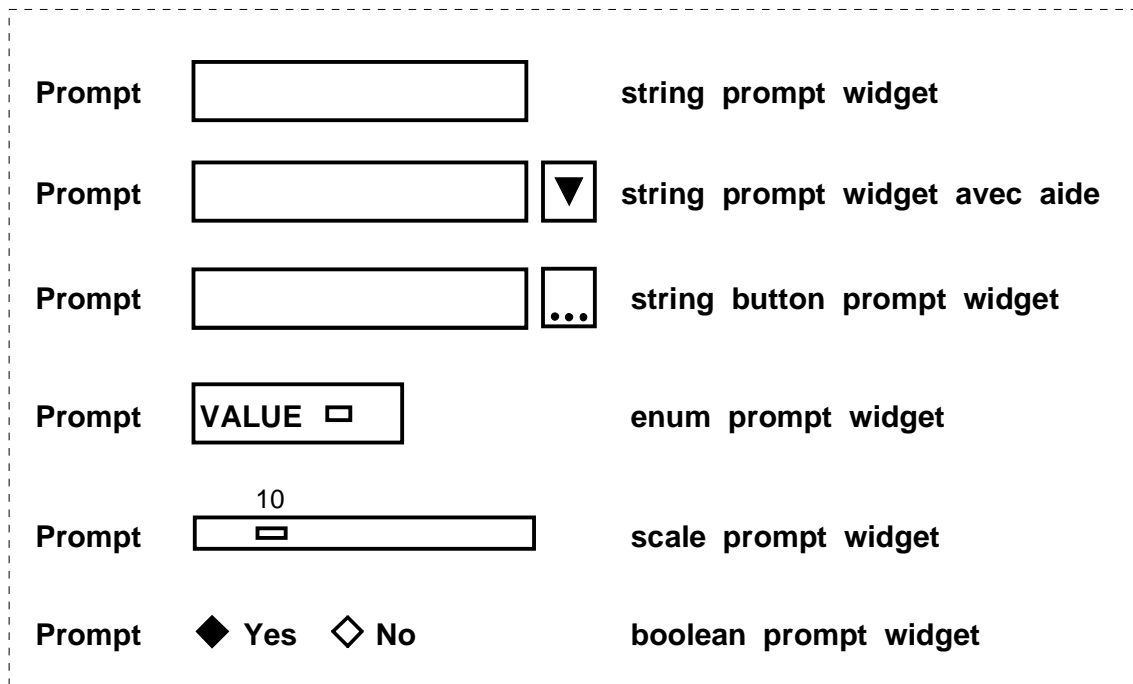


Figure 4-14 Les différents types de prompt widget

La figure ci-dessous montre les différents types de champs qui peuvent être créés.

Deux niveaux d'accès sont disponibles pour les prompts widgets. Celui de plus bas niveau est présent dans *UIT* et celui de haut niveau l'est dans *UIB*.

Dans l'interface de haut niveau, on décrit, par une variable de type *UIB.FIELD\_LIST*, l'ensemble des champs de la boîte de dialogue ou d'une partie d'une boîte. Le sous-programme

**UIT.CREATE\_PROMPT\_DIALOG** crée à partir de cette description une boîte de dialogue à laquelle on peut ajouter une rangée de boutons quelconques. Le sous-programme **UIT.CREATE\_FIELDS** crée la liste des champs dans un container. Celui-ci est à intégrer dans une fenêtre pour constituer une boîte complète. L'interaction peut être modale ou non; elle est de toute façon à la charge du programmeur. Dans le cas d'une interaction modale implantée sous la forme d'une boucle locale, il faut respecter les indications et contraintes données dans la section "gestion des événements"."boucle locale" du chapitre "Initialisations et gestion des événements".

Dans l'interface de bas niveau, on crée les champs un par un à partir des fonctions suivantes :

- **UIT.CREATE\_STRING\_PROMPT\_WIDGET**
- **UIT.CREATE\_BOOLEAN\_PROMPT\_WIDGET**
- **UIT.CREATE\_SCALE\_PROMPT\_WIDGET**
- **UIT.CREATE\_ENUM\_PROMPT\_WIDGET**

Un champ entier peut être implanté soit par un scale (**SCALE\_PROMPT\_WIDGET**) soit par chaîne (**STRING\_PROMPT\_WIDGET**). Toutes ces fonctions sont implantées à partir de **UIT.CREATE\_PROMPT\_WIDGET**. Cette fonction générale de création de prompts widgets peut servir à créer d'autres types de prompts widgets comme le toggleButton. Les champs doivent être inclus dans un rowColumn. L'alignement des prompts et des champs est réalisé par l'un des sous-programmes **UIT.SET-PROMPT-EQUAL-WIDTH** ou **UIT.EQUAL-WIDTH** décrit dans le tableau ci-dessous. Optionnellement, les champs string (**STRING\_PROMPT\_WIDGET**) peuvent proposer une aide sous forme linéaire ou hiérarchique.

<b>UIT.SET_PROMPT_EQUAL_WIDTH (WIDGET)</b>	Aligne les prompts de champs contenus dans un même rowColumn. On passe le rowColumn en paramètre.
<b>UIT.SET_PROMPT_EQUAL_WIDTH (WIDGET_LIST)</b>	Aligne les prompts de champs pouvant ne pas appartenir au même rowColumn. On passe la liste des prompts à aligner.
<b>UIT.EQUAL_WIDTH</b>	Comme la procédure précédente mais retourne la taille en pixels du plus grand des prompts

La figure ci-dessous montre les trois types de strings prompts widgets.

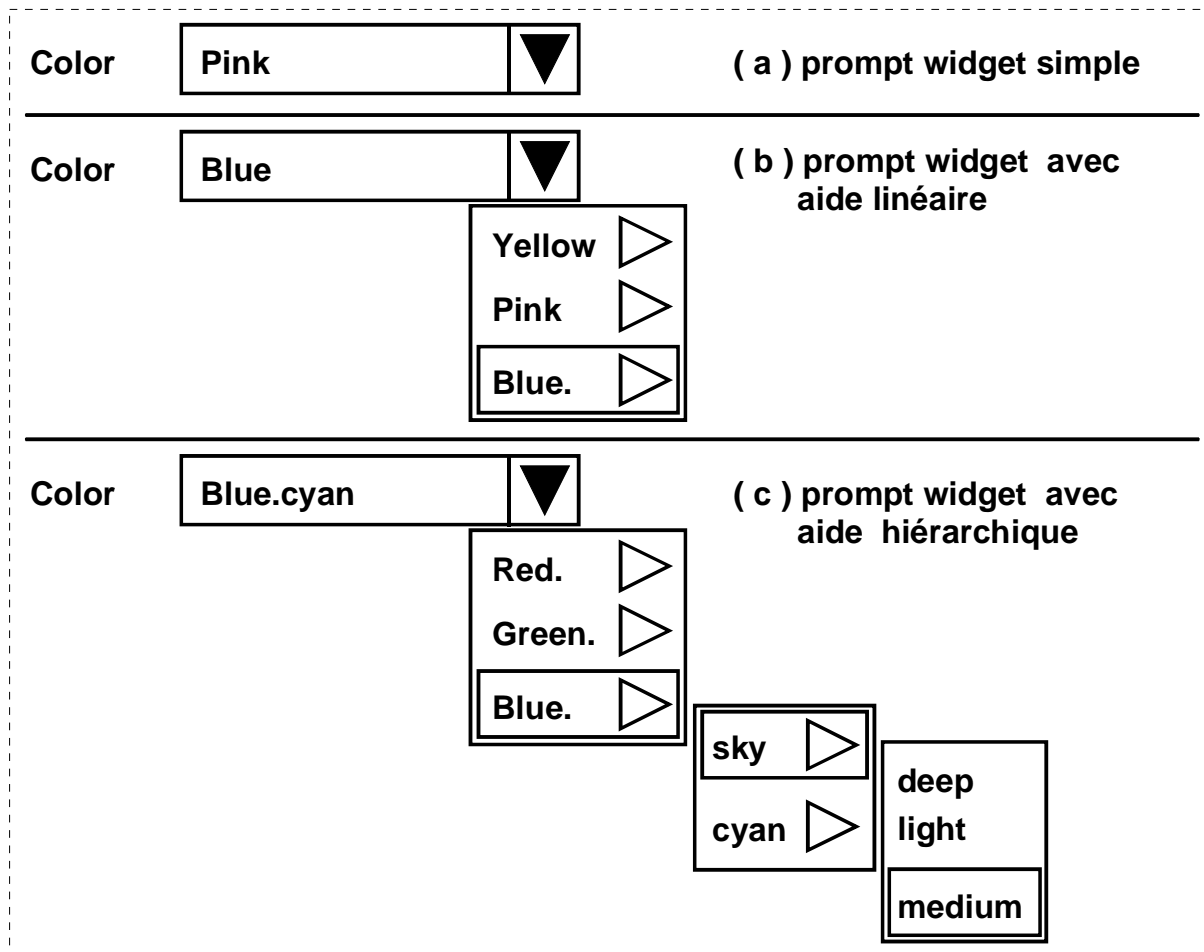


Figure 4-15 Les trois types de `STRING_PROMPT_WIDGET`

Le champ string est sans aide si le paramètre `LABELS` de type `UN.NAME_LIST` est un tableau de longueur nulle. Sinon, et avec les autres paramètres à leur valeur par défaut, une aide linéaire est ajoutée. Si le paramètre formel `BROWSE` vaut `True`, l'aide est présentée sous forme hiérarchique. La représentation hiérarchique est déduite des règles énoncées dans le paquetage `NAMES_BROWSER`. Les parties communes des chaînes séparées par des caractères spécifiques, dont la liste est donnée dans le paramètre `SEPARATORS`, sont groupées récursivement sous un même ancêtre. L'aide hiérarchique peut être dynamique, en donnant la valeur `True` au paramètre formel `DYNAMIC`. Dans ce cas, les sous-menus sont créés dynamiquement en fonction du parcours de l'utilisateur dans la hiérarchie.

#### Modification de l'aide et calcul du contenu du champ à partir de l'aide

L'aide, qu'elle soit linéaire ou hiérarchique, peut être modifiée en appelant la procédure `UIT.SET_STRING_PROMPT_LABELS`. Cette procédure permet également de personnaliser le calcul du champ à partir de l'aide. Le paramètre `LAB_CB` est une callback qui effectue ce calcul. Deux callbacks prédéfinies sont fournies, `UIT.COPY_LABEL_CB` et `UIT.APPEND_LABEL_CB`. `UIT.COPY_LABEL_CB` remplace le contenu du champ par le texte de l'aide. `UIT.APPEND_LABEL_CB` concatène avec une "," le texte de l'aide au contenu du champ. Ces deux callbacks fonctionnent aussi bien pour les aides linéaires que

pour les aides hiérarchiques. Lorsque l'on dépose ses propres callbacks pour les aides hiérarchiques, on peut utiliser les fonctions `UIT.BROWSE_MENU_LEVEL_OF` et

**UIT.BROWSE\_MENU\_NAME\_OF**. **UIT.BROWSE\_MENU\_LEVEL\_OF** retourne le niveau hiérarchique d'un choix et **UIT.BROWSE\_MENU\_NAME\_OF** retourne le texte complet. Les composants internes d'un string prompt widget sont retournés par la fonction **UIT.STRING\_PROMPT\_TEXT\_OF** (retourne le widget texte) et par la fonction **UIT.STRING\_PROMPT\_PULLDOWN\_OF** (retourne le pulldown menu de l'aide).

Lorsque l'aide linéaire ou hiérarchique ne suffit pas il est possible d'utiliser la fonction **UIT.CREATE\_STRING\_BUTTON\_PROMPT\_WIDGET**. Un bouton est ajouté immédiatement après le champ texte. Celui-ci peut par exemple déclencher l'affichage d'une boîte d'aide. Ce champ **STRING\_BUTTON\_PROMPT\_WIDGET** peut également servir de base pour saisir des champs non scalaires.

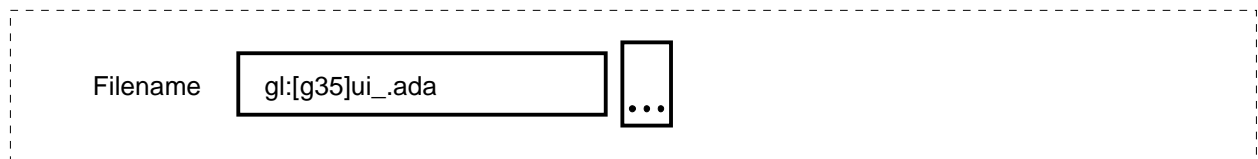


Figure 4-16 fonction **CREATE\_BUTTON\_PROMPT\_WIDGET**

La figure ci-dessous montre l'utilisation d'un "button\_string\_prompt\_widget" pour la saisie d'un nom de fichier. Le bouton qui suit le champ texte, de label '...', peut par exemple appeler une boîte **fileSelectionBox** à l'aide de l'utilitaire **UIT.GET\_FILE** et recopier le choix dans le champs.

Le sous-programme **UIT.CREATE\_STRING\_BUTTON\_PROMPT\_WIDGET** se charge uniquement de l'aspect graphique. Le calcul et l'affectation du champ texte est à la charge du programmeur.

## 4.5.2 Dialogues à champ unique

Dans le cas particulier où la saisie ne concerne qu'un seul champ on peut utiliser les fonctions suivantes:

- **UIT.GET\_STRING** pour la saisie d'une chaîne de caractères.
- **UIT.GET\_BOOLEAN** pour la saisie d'un choix Oui/Non.
- **UIT.GET\_FILE** pour la saisie d'un fichier par l'intermédiaire d'une **fileSelectionBox**.
- **UIT.GET\_INDEX** pour la saisie d'un élément dans une liste par l'intermédiaire d'un **optionMenu**.

Toutes ces fonctions sont implantées avec une boucle locale et sont donc réservées à des interactions modales. Bien qu'elles n'aient par de rapport direct avec l'objet de cette section, on peut également citer les fonctions suivantes qui ont une structure proche des précédentes:

- **UIT.CREATE\_WORK\_IN\_PROGRESS** pour construire une dialogue indiquant qu'un traitement est en cours.
- **UIT.GET\_ACKNOWLEDGEMENT** pour attendre un acquiescement de l'utilisateur.
- **UIT.PUT\_AND\_CLOSE\_ERROR** pour afficher un message et la pile d'erreur et attendre un acquiescement.
- **UI.PUT\_MESSAGE** pour créer une boîte de dialogue affichant un message quelconque.

### 4.5.3 Prompts dialogues par générique

Le paquetage générique *UIB.UTIL\_IO\_MOTIF* offre un service similaire à la fonction *UIB.CREATE\_PROMPT\_DIALOG* décrite ci-dessus. Toutefois, le composant se charge également de la gestion du dialogue qui est nécessairement modale. L'aspect graphique des boîtes créées par cet utilitaire est exactement identique à celui de l'utilitaire *UIT.CREATE\_PROMPT\_DIALOG*. En effet, les deux utilitaires font appel aux mêmes fonctions de bas niveau. Le paquetage *UIB.UTIL\_IO\_MOTIF* est générique vis-à-vis de la liste des boutons de la boîte de dialogue et vis-à-vis d'une fonction qui réagit à l'activation de l'un de ces boutons. Cette fonction reçoit en argument le rang du bouton activé et doit retourner True pour une saisie terminée. La fonction retourne False si l'on souhaite continuer la saisie. C'est le cas, par exemple, lors d'une erreur de saisie ou lorsque la boîte a un bouton "Apply". Le composant offre des fonctions de création de champs et une fonction qui affiche le dialogue et attend la saisie. La spécification de ce paquetage est donnée ci-dessous.

```
generic
  DEFAULT_TITLE : in String := "" ;
  CHOICES       : UN.NAME_LIST ;
  with function DISPATCH (CHOICE : Positive) return Boolean;

package UTIL_IO_MOTIF is

  function CREATE_STRING (          -- saisie d'un champ string avec
    PROMPT : String ;              -- ce prompt
    VALUE  : String := "" ;        -- cette valeur initiale
    SIZE   : Positive := 24)       -- cette taille
    return WIDGET ;

  function CREATE_ENUM (           -- saisie d'un champ enum avec
    PROMPT : String ;              -- ce prompt
    ITEMS  : UN.NAME_LIST ;        -- parmi cette liste
    VALUE  : Positive := 1)        -- cette valeur initiale
    return WIDGET ;

  function CREATE_BOOLEAN (        -- saisie d'un champ boolean avec
    PROMPT : String ;              -- ce prompt
    VALUE  : Boolean := True)      -- cette valeur initiale
    return WIDGET ;

  function CREATE_SCALE (          -- saisie d'un champ entier avec
    PROMPT : String ;              -- ce prompt
    VALUE  : Integer := 0 ;        -- cette valeur initiale
    FROM   : Integer := 0 ;        -- entre celui-ci et
    TO     : Integer := 100 ;     -- celui-la
    SIZE   : Positive := 200)      -- cette taille
    return WIDGET ;

  procedure CREATE_SEPARATOR ;     -- cree un separateur

  function IS_COMPLETED            -- Indique si la boîte a ete
    return Boolean ;               -- terminée

  procedure GET (                  -- ouverture de la boîte et saisie
    TITLE : in String
      := DEFAULT_TITLE) ;
  -- La boîte est terminée lorsque cette procédure est appelée
  -- La boîte est détruite après la saisie, c'est-à-dire à la fin
  -- de cette procédure

end UTIL_IO_MOTIF ;
```

Voici un exemple d'utilisation de cet utilitaire. Il est extrait du scénario 4 du programme d'exemple d'*ESCADRE*.

Les parties les plus significatives vis-à-vis du composant générique décrit ici sont soulignées.

```
procedure CP_CREATE_MOTIF (PATH: in String) is
```

```

use STANDARD_PRESENTATION;
package SS renames SIMU_SCENARIO;
package UI renames USER_INTERFACE ;
package UIT renames UI_TOOLS;
TNOS      : constant UN.NAME_LIST:= CP.TNO_NAMES;
POS_X_WGT,
POS_Y_WGT,
POS_Z_WGT : WIDGET ;

NAME_WGT, TEAM_WGT, TNO_WGT : UI.WIDGET ;
type CHOICES is (OK_CHOICE, APPLY_CHOICE, CANCEL_CHOICE) ;

function SHOW (
  W      : in UI.WIDGET;
  LOOK   : in PM.PRESENTATION;
  FROM   : in Float:= Float'First;
  TO     : in Float:= Float'Last)
return Float
is
  ATT : constant PM.PRESENTATION_ATT := PM.ATT_OF(LOOK);
begin
  return UIT.SHOW_FLOAT(W, "", ATT.AFT, ATT.EXP, ATT.UNIT, ATT.DB, FROM, TO);
end SHOW;

function DISPATCH (CHOICE : Positive) return Boolean
is
procedure APPLY is
  NAME: constant String := UI.SHOW_STRING(NAME_WGT);
  TEAM: constant ENU_TEAM:= SS.TEAM_VALUE(UI.SHOW_STRING(TEAM_WGT));
  TNO : constant String := UI.SHOW_STRING(TNO_WGT) ;
  INIT: CP.REC_INIT;
begin
  INIT.POS(1):= SHOW(POS_X_WGT, POS_LOOK);
  INIT.POS(2):= SHOW(POS_Y_WGT, POS_LOOK);
  INIT.POS(3):= SHOW(POS_Z_WGT, ALT_LOOK, 0.0);
  CP.CREATE_AND_START (NAME, TEAM, TNO, INIT);
exception
  when ESCADRE_ERROR =>
    if ERROR_MANAGER.IS_LAST_REASON(OBJECT_ERROR) then
      WARNING ("Bad Object Name");
    else
      WARNING ("ESCADRE error");
    end if ;
  when others =>
    WARNING ("Unknow error");
end APPLY ;
begin
case CHOICE is
  when 1 => APPLY ; return True ;
  when 2 => APPLY ; return False ;
  when 3 => null ; return True ;
  when others => return False ;
end case ;
end DISPATCH ;

package IOM is new UI_BOXES.UTIL_IO_MOTIF (
  DEFAULT_TITLE => PATH,
  CHOICES       => (UIT.UI_C_OK,
                   UIT.UI_C_APPLY,
                   UIT.UI_C_CANCEL),
  DISPATCH     => DISPATCH) ;
begin
  NAME_WGT := IOM.CREATE_STRING ("Name") ;
  TEAM_WGT := IOM.CREATE_ENUM ("Team" , SS.TEAM_NAME_LIST);
  TNO_WGT  := IOM.CREATE_ENUM ("Technology", TNOS) ;
  POS_X_WGT := IOM.CREATE_STRING (PM.TITLE_IMAGE(POS_LOOK, "Pos_X"));
  POS_Y_WGT := IOM.CREATE_STRING (PM.TITLE_IMAGE(POS_LOOK, "Pos_Y"));
  POS_Z_WGT := IOM.CREATE_STRING (PM.TITLE_IMAGE(ALT_LOOK, "Altitude"));
  IOM.GET ;
exception
  when others => ...
end CP_CREATE_MOTIF;

```

La figure ci-dessous montre l'aspect graphique de la boîte créé par ce sous-programme.

Figure 4-17 Création d'un dialogue à l'aide de *UTIL\_IO\_MOTIF*

## 4.6 Les widgets container

Un container est un widget qui en contient d'autre(s). Il doit être rempli après avoir été créé.

### 4.6.1 List dialog

Les sous-programmes suivants sont décrits dans cette section :

*UIT.CREATE\_LIST\_DIALOG*

*UIT.UPDATE\_LIST\_DIALOG*

*UIT.REGISTER\_LIST\_DIALOG\_BUTTONS*

Le dialogue liste ("list dialog") permet la consultation ou la modification d'attributs définis pour une liste d'éléments. Le composant est créé par la procédure *UIT.CREATE\_LIST\_DIALOG*. Celle-ci se charge de la création du widget liste, d'une rangée de boutons et d'un container pour les attributs. Ce container, appelé attribut, doit ensuite être rempli par exemple à l'aide de la procédure *UIB.CREATE\_FIELDS*. Le composant est capable de gérer la sensibilité des boutons en fonction de l'état de la sélection de la liste. Le type énuméré *UIT.LIST\_BUTTON\_SENSITIVITY* décrit les différents types de sensibilité possibles. Le tableau ci-dessous indique pour chacune des valeurs de ce type énuméré, l'état de la sélection de la liste pour que le bouton soit sensible.

<i>B_NONE</i>	bouton sensible si aucun élément de la liste est
---------------	--

	selectionné
<b><i>B_AT_LEAST_ONE</i></b>	bouton sensible si au moins un élément de la liste est selectionné
<b><i>B_ONLY_ONE</i></b>	bouton sensible si un et un seul élément de la liste est selectionné
<b><i>B_MORE_THAN_ONE</i></b>	bouton sensible si plus d'un élément de la liste est selectionné
<b><i>B_ALL</i></b>	bouton sensible si tous les éléments de la liste sont selectionnes
<b><i>B_NOT_ALL</i></b>	bouton sensible si au moins un élément n'est pas selectionné
<b><i>B_ALWAYS</i></b>	bouton toujours sensible indépendemment de la sélection de la liste

Dans l'exemple ci-dessous, qui correspond à la figure qui suit, le bouton "Show", qui montre les attributs d'un élément est sensible lorsqu'un élément et un seul est sélectionné (***B\_ONLY\_ONE***).

```

declare
  ...
  BUTTONS : constant LIST_BUTTONS := (           -- boutons et sensibilité associé
    1 => (UN.DEFINE("Apply"), B_AT_LEAST_ONE),
    2 => (UN.DEFINE("Show"), B_ONLY_ONE),
    3 => (UN.DEFINE("Cancel"), B_ALWAYS) ;
begin
  UIT.CREATE_LIST_DIALOG (
    main_window => MAIN_WINDOW,                -- mode out, identifie le dialogue
    list        => LIST,                        -- mode out, composant interne list
    attributs   => ATTRIBUTS,                  -- mode out, composant interne attribut
    title       => "Valuator",                 -- titre de la fenêtre
    list_title  => "List of valuator",         -- label de la liste
    att_title   => "",                          -- label de l'attribut
    name        => INTERFACE_NAME,            -- nom interne de la main window
    parent      => PARENT,                     -- widget père
    items       => VLTRS,                       -- éléments initiaux de la liste
    buttons     => BUTTONS,                    -- boutons
    cb          => CB,                          -- callback
    filter      => True) ;                     -- présence du champ filter
end ;

```

Optionnellement, on peut ajouter un champ "filter" pour filtrer les éléments de la liste. On peut également associer une action par défaut au double-clic sur la liste. La procédure ***UIT.UPDATE\_LIST\_DIALOG*** doit être appelé à chaque fois qu'une opération susceptible de changer la sélection est effectuée. La procédure ***UIT.REGISTER-LIST-DIALOG-BUTTONS*** permet de profiter de la gestion automatique de la sensibilité pour des boutons créés indépendamment dans la partie attributs.

La figure ci-dessous montre le résultat de l'utilisation de ce sous-programme. La partie droite du dialogue ("Visibility") appelé attribut est ajoutée postérieurement et indépendamment par le programmeur.

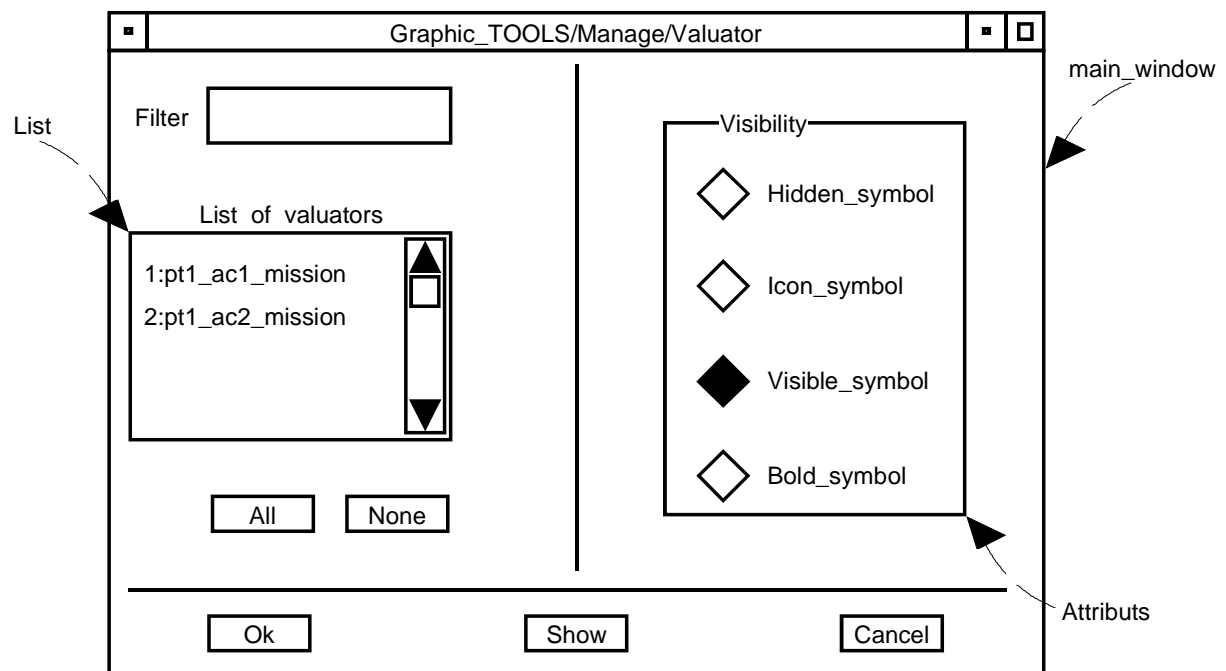


Figure 4-18 Liste dialogue

## 4.6.2 Rangée d'attributs

Le sous-programme suivant est décrit dans cette section :

### *UIB.CREATE\_ATTRIBUTS*

La procédure *UIB.CREATE\_ATTRIBUTS* crée une rangée de containers horizontaux ou verticaux précédés d'un header.

## 4.6.3 Frame

Le sous-programme suivant est décrit dans cette section :

### *UIB.CREATE\_FRAME*

Le container frame permet d'ajouter un cadre labelisé autour d'un widget quelconque. La procédure *UIB.CREATE\_FRAME* crée un cadre, un label et un widget form. La partie droite de la figure ci-dessus fait appel à ce programme pour la création du cadre de label "Visibility". Ci-dessous se trouve l'extrait de code qui crée cette partie de la boîte.

```
CREATE_FRAME (
  name   => "",                -- nom interne du widget frame
  parent => ATT,               -- widget pere,
  title  => "Visibility",      -- label du cadre
  frame  => VISIBILITY_FRAME,  -- mode out, widget frame créée
  form   => VISIBILITY_FORM) ; -- mode out, form à remplir

CREATE_RADIO_BOX(
  name => "",
  parent => VISIBILITY_FORM,
  ...) ;

MANAGE_CHILD (APP) ;
MANAGE_CHILD (VISIBILITY_FRAME) ;
MANAGE_CHILD (VISIBILITY_FORM) ;
```

#### 4.6.4 Terminal et text box

Il n'existe pas dans Motif de widget prédéfini de type terminal. Les fonctions principales d'un tel widget seraient l'écriture en concaténation et la saisie de commandes. Un widget utilitaire "terminal" est défini dans le paquetage *UI\_BOXES*. Celui-ci est construit à partir d'un simple widget Text. La fonction *UIB.CREATE\_TERMINAL* crée un terminal constitué d'une zone de sortie de texte et d'une zone de saisie de commande. Les deux zones peuvent être créées indépendamment par les fonctions *UIB.CREATE\_SCROLLED\_TEXT\_OUTPUT* et *UIB.CREATE\_COMMAND*. Elles sont nécessaires lorsque les deux zones ne sont pas adjacentes géométriquement. Les fonctions *UIB.PUT*, *UIB.NEW\_LINE* et *UIB.PUT\_LINE* fournissent les services habituels d'écriture dans un terminal. La fonction *UIB.GET\_COMMAND* saisit une commande. Cette saisie est nécessairement modale et est implantée à l'aide d'une boucle locale. La partie commande d'un terminal peut être temporairement masquée par un appel à la procédure *UIB.HIDE\_COMMAND* et restaurée par un appel à *UIB.SHOW\_COMMAND*.

##### Scrolling dans le terminal et performance

Par défaut, les écritures dans le terminal ne sont pas bufferisées. Les performances d'écriture sont donc médiocres lorsque l'on écrit par rafales. La procédure *UIB.SET\_TERMINAL\_SCROLLING* permet un passage temporaire en mode bufferisé. Cette procédure permet également la paramétrisation d'un nombre de lignes mémorisé dans le terminal. La fonction *UIB.SHOW\_TERMINAL\_SCROLLING* retourne la valeur des paramètres courants, et permet de modifier temporairement les paramètres et ensuite de restaurer les anciennes valeurs.

L'extrait de code ci-dessous montre une utilisation possible du "batch" scrolling :

```

procedure SET_BATCH_SCROLLING is
  use UI_BOXES;
begin
  UI_BOXES.SET_TERMINAL_SCROLLING (
    TEXT_OUTPUT,
    (MODE           => TS_BATCH_SCROLLING,
      RECORDED_LINES => 1000,
      BATCHED_LINES  => 50));
end SET_BATCH_SCROLLING ;

procedure SET_IMMEDIATE_SCROLLING is
  use UI_BOXES;
begin
  UI_BOXES.SET_TERMINAL_SCROLLING (
    TEXT_OUTPUT,
    (MODE           => TS_IMMEDIATE,
      RECORDED_LINES => 1000));
end SET_IMMEDIATE_SCROLLING ;

procedure ... is
begin
  ...
  SET_BATCH_SCROLLING ;
  begin
    -- lecture/écriture sur terminal (primitives spécifiques à UTIL_IO)
  exception
    when TERMINAL_END_OF_FILE => null ;
  end ;
  SET_IMMEDIATE_SCROLLING ;
  ...
end ;

```

Optionnellement, le terminal peut offrir des fonctions prédéfinies (*Search*, *Clear*, ...) par l'intermédiaire d'un popup menu. La figure ci-dessous illustre ces possibilités.

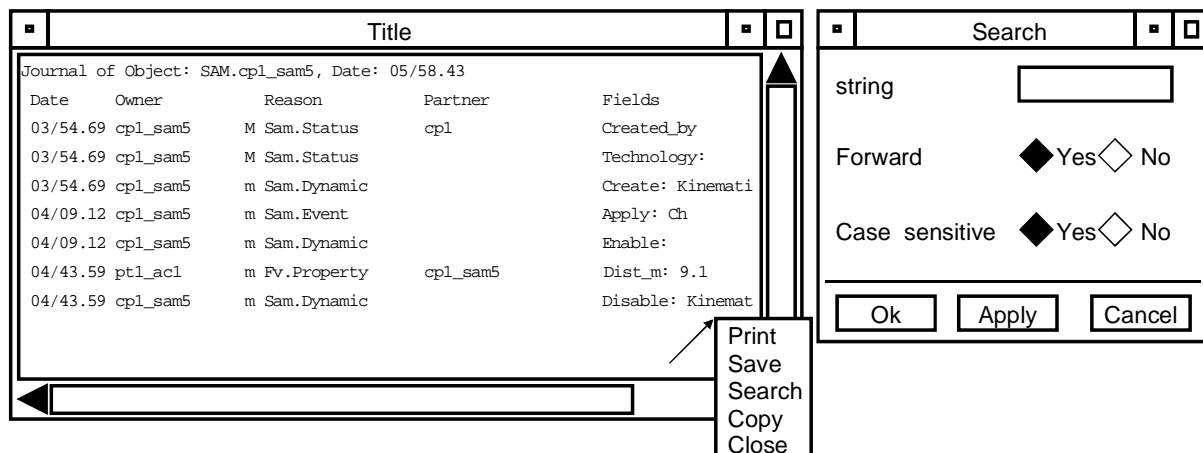


Figure 4-19 Boite texte avec son popup et sa boite Search

Lorsqu'aucune saisie n'est nécessaire et qu'une boîte indépendante convient, la fonction **UIB.CREATE\_TEXT\_BOX** crée une boîte contenant une zone de sortie textuelle. Les sous-programmes **UIB.OPEN-ALTERNATE-OUTPUT**, **UIB.MAP-AND-CLOSE-ALTERNATE-OUTPUT** et **UIB.CLOSE\_ALTERNATE\_OUTPUT** simplifient la redirection dans cette boîte des sorties standards de **UTIL\_IO**. Comme le terminal, la boîte texte propose également des fonctions prédéfinies par l'intermédiaire d'un popup menu. La procédure **UIB.MAP\_SEARCH\_TEXT**, qui implante la fonction de recherche de chaîne dans un terminal ou une boîte texte, offre également la même fonction comme service indépendant pour tout widget texte.

```

declare
  DATA : TEXT_IO.FILE_TYPE ;
begin
  TEXT_IO.OPEN (DATA, TEXT_IO.IN_FILE, FILENAME);
  CREATE_TEXT_BOX (
    w      => TEXT,          -- mode out, boîte créé
    shell  => APP_SHELL,
    data   => DATA,        -- mode in out,
    wrap   => ...,
    row    => ...,
    column => 80,
    title  => "",
    save   => ...,         -- sauve sous ce nom
    copy   => False,      -- copie du contenu de la boîte dans le canal maintenu par UTIL_IO
    edit   => False);
  MANAGE_CHILD (TEXT);
  MAP_INTERFACE (TEXT);
end ;

```

#### 4.6.5 Ouverture de connexions

La fonction **UIT.CREATE\_DISPLAY\_OPEN** crée une interface pour ouvrir des connexions avec des serveurs **X11**. Elle est d'usage suffisamment peu général pour ne pas être décrite en détail ici. Le programme d'exemple **TEST\_GRAPHIC\_TOOLS** l'utilise dans sa phase d'initialisation (bouton "Channel..." de la boîte "View Selector" appelé à l'activation du bouton "Initialize"). Les fonctions **UIT.SHOW-SERVER-NAME** et **UIT.SHOW-CHANNEL-NAME** permettent le décodage des champs de cette boîte.

### 4.7 Sélection d'une sous-liste

Les sous-programmes suivants sont décrits dans cette section :

**UIB.CREATE\_SUBLIST\_SELECT**

**UIB.SHOW\_SUBLIST**

**UIB.UPDATE\_SUBLIST**

Ce widget utilitaire permet de séparer en deux groupes une liste de noms d'éléments. En général, cette séparation correspond à la sélection d'un sous-ensemble d'un ensemble d'élément. A chaque groupe est associé un widget liste Motif. Des boutons permettent de faire passer un élément d'une liste à l'autre, c'est-à-dire d'un groupe à l'autre. Ce widget utilitaire est créé par la fonction **UIB.CREATE\_SUBLIST\_SELECT**. La fonction **UIB.SHOW\_SUBLIST** retourne le sous-ensemble des éléments sélectionnés. Le widget étant affiché, la fonction **UIB.UPDATE\_SUBLIST** permet de modifier la liste des éléments et le sous-ensemble des éléments sélectionnés.

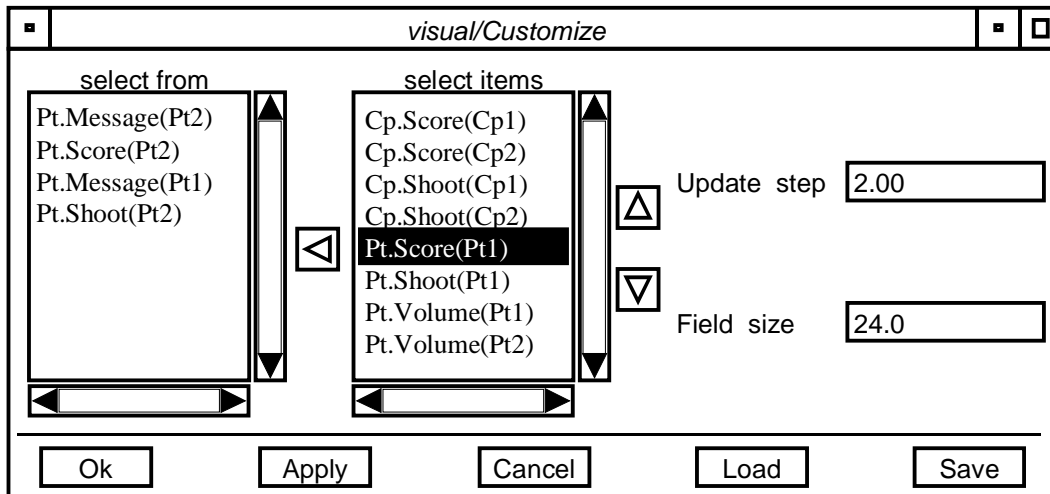


Figure 4-20 Sélection d'une sous-liste



## 5. Ressources

### 5.1 Fonctions générales

Les sous-programmes suivants sont décrits dans cette section :

<i>UI.GET_VALUE</i>	<i>UI.SET_VALUE</i>
<i>UI.SET_VALUES</i>	<i>UI.ARG</i>
<i>UI.GET_FIXED_FONT_LIST</i>	<i>UI.CREATE</i>
<i>UI.IMAGE</i>	<i>UI.XM_STRING_WIDTH</i>
<i>UI.FREE</i>	

Toutes les ressources d'un widget peuvent être manipulées à partir des fonctions *UI.GET\_VALUE* et *UI.SET\_VALUE*. Ces deux fonctions sont équivalentes aux fonctions *XtGetValues* et *XtSetValues* des intrinsèques. L'interface définie par *UI* est plus typée que l'interface des intrinsèques. La fonction *UI.GET\_VALUE* est surchargée pour tous les types possibles des ressources. La fonction *UI.SET\_VALUE* et son pluriel *UI.SET\_VALUES* prennent des arguments nommés (*UI.NAMED\_ARG*) construits par les fonctions surchargées *UI.ARG*. Ces fonctions *UI.ARG* sont surchargées pour tous les types de ressources. Elles construisent un descripteur banalisé de la ressource et de sa valeur associée. Pour les types composites (*String*, *XM\_STRING*, ...) elles réalisent une allocation de mémoire. Par défaut, la libération a lieu automatiquement. Si un partage est souhaitable, les fonctions *UI.ARG* en question ont un argument *SHARED*, auquel il faut donner la valeur True. Dans ce cas la désallocation ne sera pas effectuée après l'utilisation de l'argument. Pour éviter une allocation de mémoire, quelques services redondants court-circuitent *UI.ARG*: *UI.SET\_STRING\_VALUE* et *UI.SET\_XM\_STRING\_LIST\_VALUE*. Les arguments dont le partage est contrôlable par le programmeur doivent être utilisés avec grande précaution. L'extrait de code ci-dessous conduit à un programme dont le comportement est non prévisible car il référence deux fois un argument non-partageable.

```
declare
  ARG : constant NAMED_ARG := ARG(XmNtitle, "bonjour", SHARED=> False) ;
begin
  SET_VALUE(W1, ARG) ;-- est correct, entraîne la libération de l'argument car
                      -- non-partageable
  SET_VALUE(W2, ARG) ;-- NON correct, car référence un argument qui vient d'être libéré
end ;
```

Les extraits de code suivants, qui illustrent les fonctions de manipulations de ressources sont par contre parfaitement corrects:

```
-- retourne la couleur de fond associée au widget W.
... := UI.GET_VALUE (W, UIV.XmNbackground) ;

-- place en x=10 et y=20 le widget W.
UI.SET_VALUES (W, (UI.ARG(XmNx, 10),
                  UI.ARG(XmNy, 20)));

-- donne le titre et le nom d'icône à une fenêtre
UI.SET_VALUES(W, (UI.ARG(XmNtitle, "Diagram"),
                 UI.ARG(XmNiconName, "Diag")));
```

Le paramètre formel **SHARED** des fonctions **UI.ARG** a pour valeur par défaut **False**. Les deux appels à **UI.ARG** utilisés pour positionner le titre et le nom d'icône font appel cette propriété.

Dans le tableau ci-après, on liste pour quelques types de ressources, des ressources associées et les surcharges des fonctions **UI.ARG** et **UI.GET\_VALUE** correspondantes.

Type de la ressources	Exemple(s) de ressources	de <b>UI.ARG</b>	<b>UI.GET_VALUE</b>
<b>WIDGET_LIST_RESOURCE</b>	<b>XmNchildren</b>	<pre>function ARG (   NAME:   WIDGET_LIST_RESOURCE;   VALUE: WIDGET_LIST;   SHARED: Boolean := False)   return NAMED_ARG ;</pre>	<pre>function GET_VALUE (   W: WIDGET;   NAME: WIDGET_LIST_RESOURCE)   return WIDGET_LIST ;</pre>
<b>POS_RESOURCE</b>	<b>XmNx</b> <b>XmNy</b>	<pre>function ARG (   NAME: POS_RESOURCE;   VALUE: POSITION)   return NAMED_ARG;</pre>	<pre>function GET_VALUE (   W: WIDGET;   NAME: POS_RESOURCE)   return POSITION ;</pre>
<b>ORIENTATION_RESOURCE</b>	<b>XmNorientation</b>	<pre>function ARG (   NAME:   ORIENTATION_RESOURCE;   VALUE: Orientation)   return NAMED_ARG;</pre>	<pre>function GET_VALUE (   W: WIDGET;   NAME: ORIENTATION_RESOURCE)   return Orientation;</pre>

Ce tableau n'est bien sur pas exhaustif.

### 5.1.1 Fonctions utilitaires

### 5.1.2 Attachements

Le container formWidgetClass permet de définir des attachements sur ses enfants. Ces attachements sont spécifiés par des ressources sur les widgets enfants. Quelques fonctions utilitaires sont fournies dans le paquetage **UI\_TOOLS**. Un coté d'un fils peut être attaché à un bord de la forme (fonctions **UIT.TOP\_FORM**, **UIT.LEFT\_FORM**, **UIT.RIGHT\_FORM** et **UIT.BOTTOM\_FORM**). Un enfant peut être attaché à un autre enfant (fonctions **UIT.TOP-WIDGET**, **UIT.LEFT-WIDGET**, **UIT.RIGHT-WIDGET** et **UIT.BOTTOM-WIDGET**). Enfin, un enfant peut être accroché à une certaine position (proportion) de la forme (fonctions **UIT.TOP-POSITION**, **UIT.LEFT-POSITION**, **UIT.RIGHT-POSITION** et **UIT.BOTTOM-POSITION**). Les fonctions **UIT.LEFT\_RIGHT\_TOP\_FORM** et **UIT.LEFT\_RIGHT\_BOTTOM\_FORM** combinent l'action de plusieurs ces fonctions élémentaires.

Le tableau ci-dessous résume les fonctions disponibles. Horizontalement, les entrées du tableau contiennent les quatres cotés sur lesquels on peut réaliser l'attachement. Verticalement, les entrées correspondent aux trois types d'attachement possible :

**Form** attachement absolu à un bord de la forme

**Widget** attachement absolu à un bord d'un widget frère

**Position** attachement relatif à un bord de la forme.

<b>Form</b>	<b>TOP_FORM</b>	<b>TOP_FORM</b>	<b>LEFT_FORM</b>	<b>RIGHT_FORM</b>
<b>Widget</b>	<b>TOP_WIDGET</b>	<b>BOTTOM_WIDGET</b>	<b>LEFT_WIDGET</b>	<b>RIGHT_WIDGET</b>
<b>Position</b>	<b>TOP_POSITION</b>	<b>BOTTOM_POSITIO</b> <b>N</b>	<b>LEFT_POSITIO</b> <b>N</b>	<b>RIGHT_POSITION</b>

### 5.1.3 Widgets de saisie

Si certaines ressources contrôlent l'apparence d'un widget, d'autres sont une mémoire du contenu du widget. L'interrogation de ces ressources peut se faire à l'aide de la fonction *UI.GET\_VALUE*. Il est toutefois préférable d'utiliser les fonctions *SHOW\_\** qui offrent en plus un service de présentation. Dans certains cas, une même fonction *SHOW\_\** peut accepter en entrée des widgets de classe différente. Par exemple, la fonction *UI.SHOW\_INTEGER* fonctionne aussi bien pour les widgets *textField* que *scale*. La description de la fonction au sein du paquetage précise la liste des classes acceptées en entrée. Le paquetage *USER\_INTERFACE* fournit les services pour les widgets prédéfinis de *Motif* (fonctions *UI.SHOW\_STRING*, *UI.SHOW\_INTEGER* et *UI.SHOW\_BOOLEAN*). Le paquetage *UI\_TOOLS* fournit des services qui supportent également les widgets utilitaires du composant *UI*. Ces fonctions ont des signatures identiques à celles de *UTIL-IO* (fonctions *UIT.SHOW\_BOOLEAN*, *UIT.SHOW\_INTEGER*, *UIT.SHOW\_FLOAT* et *UIT.SHOW\_VIRTUAL\_TIME*).

Toutes les fonctions *SHOW\_\** décrites ci-dessus ont une fonction *SET\_\** associées. Celles-ci servent au positionnement de la ressource et sont des services équivalents à *UI.SET-VALUE* (fonctions *UI.SET-INTEGER*, *UI.SET-BOOLEAN*, *UIT.SET-BOOLEAN*, *UIT.SET-FLOAT* et *UIT.SET-VIRTUAL-TIME*).

Enfin, la fonction *UI.ERASE* réinitialise les widgets en question à leur valeur par défaut *Motif*.

Les widgets *XmList*, *XmCheckBox*, *XmRadioBox* et *XmOptionMenu* gèrent une liste d'élément. Le domaine est manipulé par les fonctions :

<i>UI.ADD_ELT</i> ,	<i>UI.REMOVE_ELT</i> ,
<i>UI.REMOVE_ELTS</i>	<i>UI.REMOVE_ALL_ELTS</i>
<i>UI.REMOVE_STRING_ELT</i>	<i>UI.REMOVE_SELECTED</i> ,
<i>UI.SHOW_ELT_COUNT</i>	<i>UI.SHOW_STRING_ELT</i> .

La gestion de la sélection des éléments s'effectue par les fonctions:

<i>UI.DESELECT_ALL</i>	<i>UI.SET_SELECTED_STRING</i> ,
<i>UI.SET_SELECTED_RANK</i>	<i>UI.SHOW_SELECTED_LENGTH</i>
<i>UI.IS_RANK_SELECTED</i> ,	<i>UI.SHOW_SELECTED_STRING</i>
<i>UI.SHOW_SELECTED_RANK</i> ,	<i>UI.SHOW_MULTIPLE_SELECTED_RANK</i>
<i>UI.SHOW_MULTIPLE_SELECTED_ELT</i> ,	<i>UI.DESELECT_RANK</i>
<i>UI.DESELECT_ELT</i> ,	<i>UI.SELECT_ALL</i>

Ces fonctions sont décrites en détail dans le paquetage *USER\_INTERFACE*

Certaines fonctions sont spécifiques à une classe de widgets. La fonction *UI.CHECK\_BOX\_SHOW* retourne pour un widget *check box* l'état de la des sélections des boutons internes.

Les fonctions suivantes sont spécifiques au widget texte :

<i>UI.TEXT_FLUSH</i>	<i>UI.TEXT_SCROLL</i>
<i>UI.TEXT_REMOVE_LINES</i>	<i>UI.ADD_STRING</i>
<i>UI.ADD_STRING</i>	<i>UI.SHOW_PRIMARY_SELECTION</i>
<i>UI.SET_PRIMARY_SELECTION</i>	<i>UI.CLEAR_PRIMARY_SELECTION</i>

### 5.1.4 Sensibilité

Un widget de saisie peut être "gélé" afin d'empêcher l'utilisateur de manipuler le widget. On dit que le widget a été rendu "insensible". La sensibilité d'un widget est contrôlée par les ressources *XmNsensitive* et *XmNancestorSensitive*. Il est préférable de pas manipuler ces deux ressources directement par les fonctions *UI.SET\_VALUE* et *UI.GET\_VALUE*. La fonction *UI.SET\_WIDGET\_SENSITIVE* rend sensible ou insensible un widget et ses descendants s'il en a. La fonction fonction *UI.IS\_SENSITIVE* permet d'interroger la sensibilité d'un widget. Les fonctions utilitaires *UIT.SET\_SENSITIVE* et *UIT.SET\_UNSENSITIVE* offrent un service identique à *UI.SET\_WIDGET\_SENSITIVE* mais évitent la traversée complète de la hiérarchie qui modifie intempestivement certaines classes de widgets. Notamment, pour une raison inconnue, la modification de la sensibilité d'une drawingArea génère un événement expose. La fonction *UIT.TOGGLE\_SENSITIVITY* inverse la sensibilité d'un ou plusieurs widget(s).

### 5.1.5 Bitmaps et pixmaps

Les widgets labels et drawnButton peuvent prendre à la place d'un label un bitmap ou un pixmap. Un bitmap est limité à deux couleurs. Un pixmap est multi-couleurs. Les bitmaps et les pixmaps sont en général stockés dans des fichiers. Les bitmaps supportent le format *XBM* et le format *UID*. Les pixmaps supportent uniquement le format *UID*. La fonction *UI.FETCH\_BITMAP* lit un bitmap dans un fichier *UIL* défini par xbitmapfile. La fonction *UI.READ\_BITMAP\_FILE* lit un bitmap dans un fichier *XBM*. La fonction *UIT.READ\_BITMAP\_FILE* offre le même service mais avec une interface simplifiée. La fonction *UI.FETCH\_ICON* lit un pixmap dans un fichier *UID*. Enfin, il est possible de construire un pixmap à partir d'un bitmap à l'aide de la fonction *UI.CREATE\_PIXMAP\_FROM\_BITMAP*.

## 5.2 Fichier de ressources

*X-Window* et *Motif* fournissent un mécanisme de personnalisation de l'interface basé sur les fichiers de ressources. Des fichiers de ressources sont chargés automatiquement lors de l'ouverture de la connexion avec le serveur *X*. Toutefois, il est possible de charger explicitement un fichier de ressources à l'aide de la fonction *UI.LOAD\_RESOURCE\_FILE*. Il est possible, par exemple, d'utiliser un fichier de ressources pour enregistrer une personnalisation d'un utilisateur. Le fichier peut être construit à l'aide des fonctions *UI.CREATE*, *UI.PUT* et *UI.CLOSE*. Les ressources peuvent être des ressources standards ou des ressources spécifiques à l'application. La fonction *UI.GET-APPLICATION-RESOURCE* permet l'interrogation de ces dernières. A chaque connexion est associée une base de ressources. Lorsqu'un fichier de ressource est lu, ces définitions de ressources sont enregistrées dans une base de ressource. Lors de la création d'un widget, les ressources qui ne sont pas définies explicitement, sont recherchées dans la base de données. La fonction *UI.SET\_DEFAULT\_VALUE* permet de remplir la base de données des ressources. Elle permet, notamment, d'offrir un défaut dynamique.

## 6. Callbacks et event handlers

Les sous-programmes suivants sont décrits :

*UI.SHOW\_CB*

*UI.POST\_CB*

*UI.POST\_CB*

*UI.UNPOST\_CB*

*UI.UNPOST\_ALL\_CB*

*UI.NOT\_YET\_IMPLEMENTED\_CB*

*UI.CALL\_CALLBACKS*

*UI.INVOKE\_CALLBACK*

*UI.EVENT\_HANDLER*

*UI.EVENT\_HANDLER*

*UI.POST\_EVENT\_HANDLER*

*UI.POST\_RAW\_EVENT\_HANDLER*

*UI.UNPOST\_EVENT\_HANDLER*

*UI.UNPOST\_RAW\_EVENT\_HANDLER*

### 6.1 Généralités sur le mécanisme de callback et interfaçage avec le monde Ada

La logique de contrôle d'un programme *Motif* n'est pas séquentielle. A bas niveau elle est de nature événementielle. Le composant 'translation manager' des intrinsics se charge de la traduction d'un événement ou d'une suite d'événements en action de haut niveau. Certaines actions déclenchent des callbacks. La présence de ce composant dans les intrinsics permet de ne pas avoir de dispatcher d'événement codé en dur dans son application. L'application déclare son intérêt sur certaines action en déposant des callbacks (on dit aussi en postant des callbacks). La callback la plus utilisée est l'activateCallback. Celle-ci est postée pour un widget sous-classe de pushButton et est déclenchée lorsque l'utilisateur appuie et relâche le bouton associé au widget.

Il n'existe pas de pointeur sur procédure en *Ada 83*. Toutefois, tous les compilateurs fournissent un moyen pour récupérer l'adresse d'une procédure. En général, le compilateur exige que cette procédure soit statique. Malheureusement, les contraintes ne sont pas les mêmes pour tout les compilateurs. Le mécanisme de dépôt de callback proposé par *UI* respecte l'union des contraintes des compilateurs *DECAda*, *Verdix* et *Alsys*.

Le tableau ci-dessous indique pour chaque compilateur le ou les pragma(s) à associer aux procédures déposées comme callbacks.

compilateur	pragma
<i>DECAda</i>	<i>pragma EXPORT_PROCEDURE</i>
<i>Verdix</i>	aucun pragma n'est nécessaire
<i>Alsys</i>	<i>pragma EXTERNAL_NAME</i> <i>pragma CALL_IN</i>

## 6.2 Pointeurs sur procédure

A chaque signature de callback est associée un type privé "pointeur sur procédure ayant cette signature". Les constructeurs de ces types privés sont génériques vis-à-vis de la procédure callback. La généricité permet la vérification à la compilation de la signature de la procédure déposée et permettra la compatibilité du code applicatif avec *Ada 95*. Un deuxième constructeur est fourni mais celui-ci n'offre aucune sécurité. Il doit être utilisé uniquement pour implanter dans les utilitaires ses propres constructeurs par générique.

3 signatures distinctes sont reconnues par *UI* :

- signature d'une callback programmatique (type *CB\_NAMED\_ADDRESS*)
- signature d'une callback *UIL* (type *UIL\_NAMED\_ADDRESS*)
- signature d'un event handler (type *EH\_NAMED\_ADDRESS*)

Les callbacks programmatiques sont les callbacks postées par programme. Les callbacks *UIL* sont les callbacks référencées dans un fichier *UIL*. Les events handlers de *UI* correspondent aux events handlers des intrinsics et sont postés par programme.

Les noms des constructeurs génériques sont donnés dans le tableau ci-dessous :

signature	constructeur
<i>CB_NAMED_ADDRESS</i>	<i>CB_PROC</i>
<i>UIL_NAMED_ADDRESS</i>	<i>UIL_PROC</i>
<i>EH_NAMED_ADDRESS</i>	<i>EH_PROC</i>

Une callback programmatique (type *CB\_NAMED\_ADDRESS*) doit nécessairement avoir la signature suivante :

```
procedure (  
  W           : in WIDGET ;  
  CLIENT_DATA : in C_LONG ;  
  CALL_DATA  : in ANY_CALLBACK_STRUCT) ;
```

Le choix du type de la client data (*C\_LONG*) exige un petit commentaire. En choisissant un type atomique on laisse l'entière responsabilité de la gestion de la client-data à l'application. Soit elle se contente d'un entier, soit elle y met un pointeur (*UNCHECKED\_CONVERSION*) dont elle gère l'allocation et la désallocation.

Une callback *UIL* (type *UIL\_NAMED\_ADDRESS*) doit nécessairement avoir la signature suivante :

```
procedure (  
  W           : in WIDGET ;  
  CLIENT_DATA : in UIL_CLIENT_DATA ;  
  CALL_DATA  : in ANY_CALLBACK_STRUCT) ;
```

Le type *UIL\_CLIENT\_DATA* est un type privé qui n'a pas de constructeur (l'invocation de la callback est réalisé par le *Mrm* de *Motif*) et qui peut être transformé en *C\_LONG* par la fonction *TO\_C\_LONG*. Ce mécanisme apparemment complexe est nécessaire pour garantir la portabilité du source applicatif sur les différentes plate-formes supportées.

Une event handler (type *EH\_NAMED\_ADDRESS*) doit nécessairement avoir la signature suivante :

```
procedure (  
  W           : in WIDGET ;  
  CLIENT_DATA : in C_LONG ;  
  EVNT       : in C_EVENT) ;
```

## 6.3 Dépôt des callbacks et event handlers

Les callbacks programmatiques peuvent être déposées à la création du widget ou plus tard, à l'aide des sous-programmes surchargés *UI.POST\_CB*. Le lien entre le nom des callbacks *UIL* et leur adresse est donné lors du chargement de l'interface (fonction *UI.FETCH\_AND\_REALIZE\_UIL\_INTERFACE*) ou ultérieurement (procédures *UI.REGISTER*). Les events handlers sont postés par les sous-programmes surchargés *UI.POST\_EVENT\_HANDLER* et *UI.POST\_RAW\_EVENT\_HANDLER*. La distinction entre un event handler et un raw event handler est définie par les intrinsics. Un raw event handler déposé sur un widget ne sélectionne pas de nouveau événement sur la fenêtre *X* associée au widget (*XSelectInput*).

Les sous-programmes de postage des callbacks programmatiques et des events handlers existent sous la forme de fonctions et de procédures. Lorsque l'on souhaite pouvoir retirer une procédure déposée, il est nécessaire d'utiliser la fonction. Celle-ci retourne une clé qui permet de retirer le dépôt (procédures *UI.UNPOST-CB*, *UI.UNPOST-EVENT-HANDLER* et *UI.UNPOST-RAW-EVENT-HANDLER*)

L'exemple ci-dessous illustre l'utilisation des sous-programmes de postage.

```
POST_CB (DRAWING_AREA, XmNexposeCallback,
        CB(GET_ACTION_CB_PROC, C_LONG(1))) ;
```

Les constructeurs non génériques ne sont pas explicités en détail car on ne souhaite pas promouvoir leur utilisation. Ils sont toutefois rappelés succinctement dans le tableau ci-dessous.

signature	Constructeur
<i>CB_NAMED_ADDRESS</i>	<i>PROC</i>
<i>UIL_NAMED_ADDRESS</i>	<i>IDENT</i>
<i>EH_NAMED_ADDRESS</i>	<i>EVENT_HANDLER</i>



## 7. Parcours de la hiérarchie

Les sous-programmes suivants sont décrits dans les sections qui suivent :

- descendants

*UI.CHILD\_OF*

*UI.POPUP\_CHILDREN\_OF*

*UIT.SIMPLE\_DESCENDANTS\_OF*

- ancêtres

*UI.PARENT\_OF*

*UI.MAIN\_WINDOW\_OF*

*UI.APPLICATION\_WINDOW\_OF*

*UI.TOPLEVEL\_SHELL\_OF*

*UI.MAIN\_WINDOWS\_OF*

*UI.APPLICATION\_SHELLS\_OF*

- diffusion d'une action

*UI.DIFFUSE\_ACTION*

- caractéristique d'un widget

*UI.IS\_CLASS*

*UI.IS\_MANAGED*

*UI.IS\_VISIBLE*

- sélection selon un critère

*UIT.MANAGED\_OF*

*UIT.SELECTED\_FROM\_CLASS*

*UIT."- "*

- widget interne d'un widget construit par composition

*UIT.LABEL\_CASCADE\_OF*

*UIT.BROWSE\_MENU\_OF*

*UIT.STRING\_PROMPT\_PULLDOWN\_OF*

*UI.CHILDREN\_OF*

*UIT.CHILDREN\_OF*

*UIT.SIBLING\_OF UI.DEPTH\_OF*

*UIT.PARENTS\_OF*

*UI.TOP\_WINDOW\_OF*

*UI.SHELL\_OF*

*UI.APPLICATION\_SHELL\_OF*

*UI.TOP\_WINDOWS\_OF*

*UIT.SIBLINGS\_OF*

*UI.IS\_SUBCLASS*

*UI.IS\_WIDGET*

*UIT.SENSITIVE\_OF*

*UIT.IS\_INTRO*

*UIT.DRAWN\_CASCADE\_OF*

*UIT.STRING\_PROMPT\_TEXT\_OF*

### 7.1 Rappels sur les hiérarchies de widgets

Un widget peut avoir deux types d'enfants, des enfants standards ou des enfants popups. Un enfant standard est géométriquement à l'intérieur de son père. Un enfant popup n'a pas cette contrainte. Un popup menu est un exemple d'enfant popup. Seuls les widgets héritant de composite peuvent avoir des enfants standards. Ces classes de widgets acceptent et sont capables de gérer des widgets enfants. Ces classes s'opposent aux sous-classes de primitif dont

les instances ne peuvent avoir d'enfants. Par contre, tout widget est susceptible d'avoir des enfants popups. Par exemple, un widget de classe *XmText*, bien que sous-classe de primitif, peut dérouler un popup menu.

La figure ci-dessous donne le début de la hiérarchie d'une application constituée d'une fenêtre principale, d'une fenêtre secondaire et d'une boîte de dialogue. Une boîte de dialogue est en général transitoire et modale alors qu'une fenêtre secondaire est permanente et icônifiable.

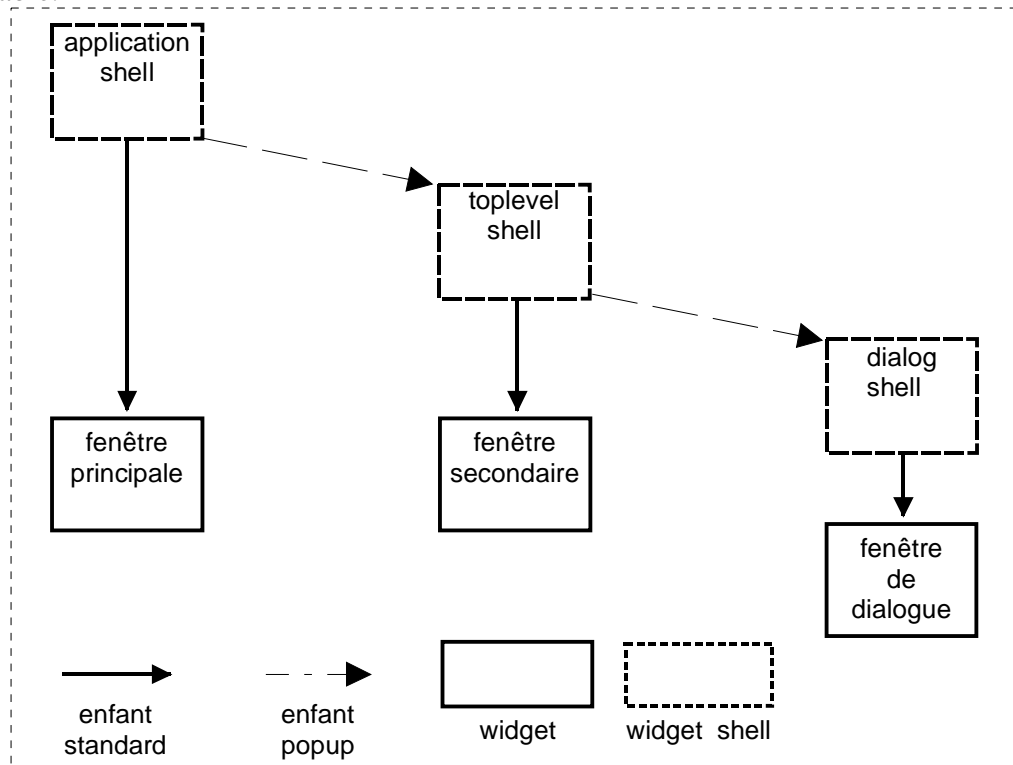


Figure 7-1 Hiérarchie des fenêtres d'une application

Les widgets shells ont un rôle bien définis. Ils servent de coquille entre le monde extérieur (les autres clients du serveur dont le window manager) et les fenêtres de l'application considérée. Ils encapsulent notamment tous les échanges avec le window manager. Toutes les fenêtres indépendantes d'une application, qu'elles aient ou non des décorations, sont filles standards d'un widget shell. La fenêtre principale d'une application, lorsqu'elle existe, est fille standard d'un widget shell de classe *applicationShell*. Les fenêtres secondaires sont filles standards de shells de classe *oplevelShell* eux-mêmes fils popups de l'*applicationShell* ou d'un autre *oplevelShell*. Les fenêtres de dialogue, sont filles de widget de classe *dialogShell* eux mêmes fils popups de l'*applicationShell*, d'un *oplevelShell* ou d'un dialog shell. Enfin, une fenêtre popup est fille standard d'un *popupShell* lui-même fils popup d'un widget quelconque. La figure "hiérarchie des fenêtres d'une application" illustre cette décomposition hiérarchique. Chaque display dispose de son propre *applicationShell*. Un *applicationShell* peut ne pas avoir de fils standards. C'est le cas lorsqu'une application n'a pas de fenêtre ayant un rôle plus important que les autres.

## 7.2 Descendants

La fonction *UI.CHILDREN\_OF* retourne les enfants standards d'un widget composite. La fonction *UI.POPUP\_CHILDREN\_OF* retourne les enfants popups d'un widget. La fonction *UI.CHILD\_OF* retourne l'enfant unique pour les parents qui n'acceptent qu'un seul

enfant standard. C'est le cas par exemple des widgets shells. La fonction *UIT.CHILDREN\_OF* retourne la concaténation des enfants d'un ensemble quelconque de widgets. La fonction *UIT.SIMPLE\_DESCENDANTS\_OF* retourne les descendants standards d'un widget composite. La fonction *UIT.SIBLING\_OF* retourne les frères d'un widget. La fonction *UI.DEPTH\_OF* retourne la profondeur d'un widget vis-à-vis d'un widget ancêtre.

## 7.3 Ancêtres

### 7.3.1 Ancêtre direct

La fonction *UI.PARENT\_OF* retourne le père d'un widget. La fonction *UIT.PARENTS\_OF* retourne les ancêtres d'un widget.

### 7.3.2 Ancêtre fenêtre

La fonction *WINDOW\_OF* retourne la fenêtre principale, secondaire ou de dialogue ancêtre d'un widget. La fonction *UI.TOP\_WINDOW\_OF* retourne une fenêtre principale ou secondaire ancêtre d'un widget. La fonction *UI.APPLICATION\_WINDOW\_OF* retourne la fenêtre principale ancêtre d'un widget.

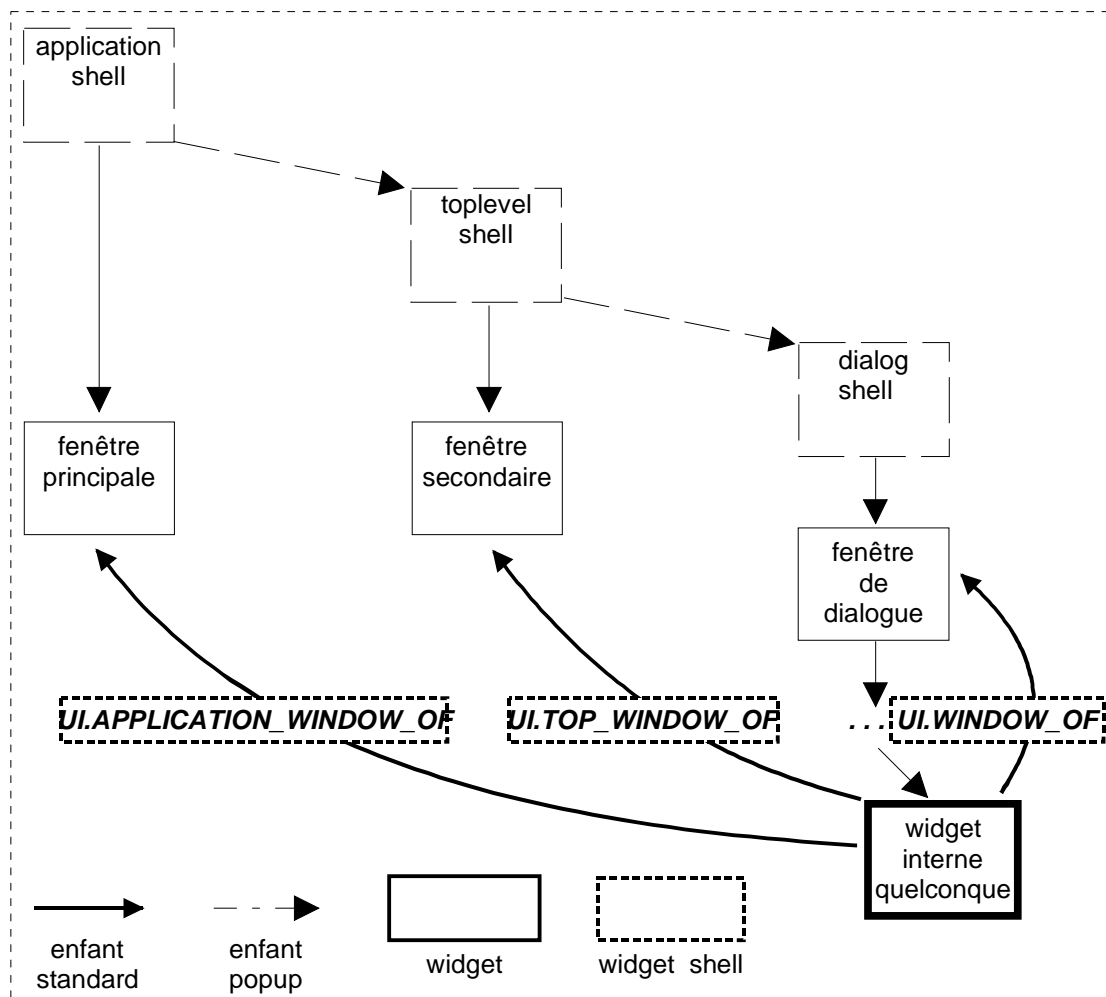


Figure 7-2 Les trois fenêtres ancêtres et leur sélecteur

### 7.3.3 Ancêtre shell

La fonction *UI.SHELL\_OF* retourne le premier ancêtre shell distinct d'un popup shell. La fonction *UI.TOPLEVEL\_SHELL\_OF* retourne l'ancêtre shell sous-classe de *oplevelShell* donc père d'une fenêtre principale ou secondaire. La fonction *UI.APPLICATION\_SHELL\_OF* retourne l'*applicationShell* ancêtre d'un widget ou l'*applicationShell* associé à un display.

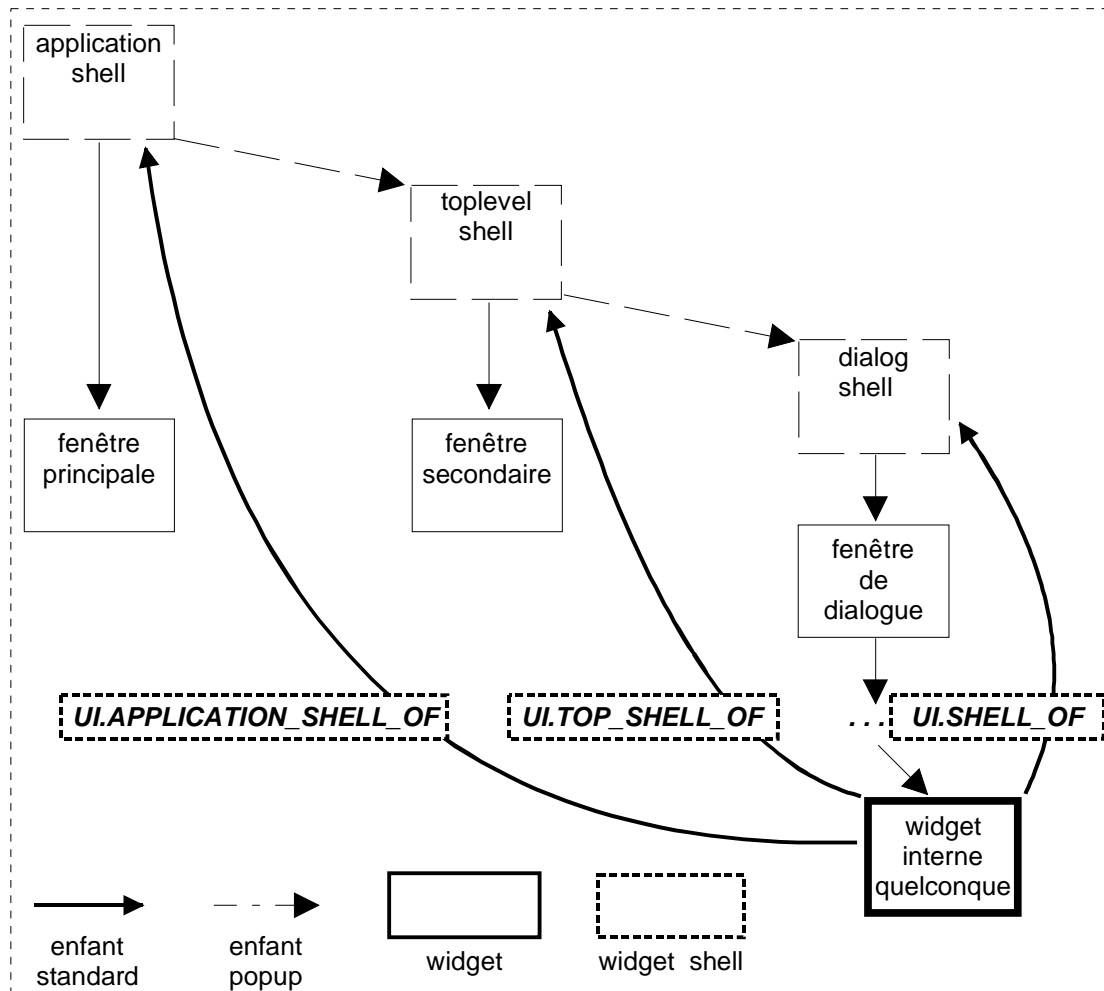


Figure 7-3 Les trois shells et leurs sélecteurs

Les fonctions *UI.WINDOW\_OF*, *UI.TOP\_WINDOW\_OF* et *UI.APPLICATION\_SHELL\_OF* existent dans une version plurielle. Elles retournent l'ensemble des fenêtres vérifiant le critère de la fonction au singulier.

La fonction *UIT.SIBLINGS\_OF* retourne les frères, oncles, grands-oncles, ... jusqu'à un widget racine.

## 7.4 Diffusion d'une action sur la hiérarchie

Les fonctions précédentes permettent de parcourir de façon active une hiérarchie de widgets. La procédure générique *UI.DIFFUSE\_ACTION* est un itérateur passif qui permet d'appliquer une action sur les descendants d'un widget. L'action peut, au choix, être appliquée de bas en haut ou de haut en bas. La traversée peut-être interrompue si nécessaire après l'exécution d'une action.

## 7.5 Widget interne d'un widget construit par composition

Certains widgets pré-existants dans *Motif* ou spécifiques à *UI* sont construits en assemblant des widgets. La hiérarchie interne de ces widgets internes n'est en général pas publique. Il n'est pas souhaitable que les programmeurs aient connaissance de cette hiérarchie car celle-ci peut évoluer. Il existe des fonctions qui retournent les composants internes significatifs d'un widget construit par assemblage. Elles peuvent se présenter soit sous la forme d'une fonction unique appelée *UI.CHILD\_OF* ou sous la forme d'une famille de fonctions. Dans le premier cas, on précise à l'appel le composant interne requis, et, dans le deuxième cas, il existe une fonction par composant interne.

Pour les widgets pré-existants dans *Motif*, ces fonctions sont décrites complètement ici. Pour les widgets spécifiques à *UI*, elles sont simplement listées pour mémoire. Elles sont décrites en détail là où le widget en question est introduit.

Le tableau ci-dessous précise l'ensemble des surcharges de la fonction *UI.CHILD\_OF* et les classes de widgets que ces fonctions acceptent. La colonne "composants internes" correspond à un ou plusieurs types énumérés qui listent l'ensemble des composants internes.

Classe	Composants internes
<i>XmCommand</i>	<i>DialogTypeSelection</i>
<i>XmFileSelectionBox</i>	<i>DefaultButtonType</i> et <i>ChildType</i>
<i>XmMessageBox</i>	<i>DefaultButtonType</i>
<i>XmSelectionBox</i>	<i>DefaultButtonType</i> et <i>ChildType</i>
<i>XmOptionMenu</i>	<i>OptionMenuChildType</i>

Le tableau ci-dessous donne le nom des fonctions qui retournent les composants internes de certains widgets.

Classe	Composants internes
<i>BrowseMenu</i>	<i>UIT.LABEL_CASCADE_OF</i> <i>UIT.DRAWN_CASCADE_OF</i> <i>UIT.BROWSE_MENU_OF</i>
<i>PromptWidget</i>	<i>UIT.PROMPT_OF</i>
<i>StringPromptWidget</i>	<i>UIT.STRING_PROMPT_TEXT_OF</i> <i>UIT.STRING_PROMPT_PULLDOWN_OF</i>
<i>ExtendedRadioBox</i>	<i>UIB.RADIO_BOX_LABEL_OF</i> <i>UIB.RADIO_BOX_TOGGLE[S]_OF</i> <i>IB.RADIO_BOX_FIELD[S]_OF</i>
<i>Terminal</i>	<i>UIB.COMMAND_PROMPT_OF</i> <i>UIB.COMMAND_TEXT_OF</i>

Ces fonctions sont décrites dans la section où le widget en question est introduit.

## 7.6 Caractéristiques d'un widget

Les actions exécutées lors de la traversée active ou passive d'une hiérarchie ont fréquemment besoin des caractéristiques d'un widget. On ne duplique pas ici la description des fonctions de manipulation de ressources décrites dans le chapitre "Ressources". On se limite aux fonctions qui retournent un attribut de widget ne correspondant pas à une ressource. Le

tableau ci-dessous donne un aperçu de chacune de ces fonctions. Elles sont détaillées dans le paquetage où elles apparaissent.

<b>ULIS_CLASS</b>	indique si un widget est d'une classe donnée
<b>ULIS_SUBCLASS</b>	indique si un widget est sous-classe d'une classe donnée
<b>ULIS_WIDGET</b>	indique si un widget n'est pas un gadget
<b>ULIS_VISIBLE</b>	indique si un widget est visible, c'est-à-dire s'il n'est pas clippé par son père ou s'il n'est pas recouvert par un frère
<b>UIT.IS_INTO UIT."-"</b>	deux opérations ensemblistes sur une liste de widgets
<b>UIT.MANAGED_OF</b>	retourne les widgets "managés" d'une liste de widgets
<b>UIT.SENSITIVE_OF</b>	retourne les widgets "sensibles" d'une liste de widgets
<b>UIT.SELECTED_FROM_CLASS</b>	retourne les widgets d'une classe donnée d'une liste de widgets

Une bonne maîtrise des primitives de parcours de hiérarchie et des primitives de caractérisation de widgets est importante. Elles permettent d'éviter les variables globales et il est plus facile d'écrire des callbacks réutilisables (indépendantes d'une IHM particulière).

L'extrait de code ci-dessous montre l'écriture, à partir de ces primitives, d'une fonction qui recherche l'ancêtre d'un widget d'une classe donnée :

```
function SEARCH_ANCESTOR ( -- Recherche le premier widget
  W      : WIDGET;          -- ancetre de celui-ci,
  CLASS: WIDGET_CLASS)    -- de cette classe
return WIDGET
is
  S, R : WIDGET := NULL_WIDGET ;
begin
  S := W ;
  loop
    exit when S = NULL_WIDGET ;
    if IS_CLASS(S, CLASS) then
      R := S ;
      exit ;
    end if ;
    S := PARENT_OF(S) ;
  end loop ;
  return R ;
end SEARCH_ANCESTOR ;
```

## 8. Divers

### 8.1 Manipulation du curseur

Les fonctions suivantes sont décrites dans cette section :

<i>UI.ROOT_WINDOW_OF</i>	<i>UI.TRANSLATE_COORDINATE</i>
<i>UI.SHOW_POINTER</i>	<i>UI.CREATE_CURSOR</i>
<i>UI.DEFINE_CURSOR</i>	<i>UI.UNDEFINE_CURSOR</i>
<i>UI.DESTROY_CURSOR</i>	<i>UI.SET_FOCUS</i>

Les fonctions de manipulation du curseur associé à la souris font partie de la bibliothèque *Xlib*. Elles sont toutefois interfacées par *UI* car la partie d'une application qui gère l'IHM a en général besoin de changer la forme du curseur. L'utilisation la plus fréquente consiste à donner au curseur la forme d'une montre pour indiquer à l'utilisateur qu'une partie de l'IHM n'est pas prête à recevoir ses ordres.

La fonction *UI.CREATE\_CURSOR* permet de définir une forme de curseur parmi les formes standards de *X-Window*. La fonction *UI.DEFINE\_CURSOR* permet d'associer cette forme à un widget de l'IHM.

L'association se termine par un appel à la procédure *UI.UNDEFINE\_CURSOR*. Le curseur retrouve alors sa forme précédente. Enfin, la libération de la ressource créée par la fonction *UI.CREATE\_CURSOR* se fait par le fonction *UI.DESTROY\_CURSOR*.

La fonction *UI.SHOW\_POINTER* indique dans quel widget se trouve le curseur. La fonction *UI.TRANSLATE-COORDINATE* est identique à la fonction *XtTranslateCoordinate*. La fonction *UI.ROOT-WINDOW-X-OF* retourne la fenêtre racine associée à un display.

### 8.2 Gestion des couleurs

Les fonctions suivantes sont décrites dans cette section :

<i>UI.ALLOC_COLOR_CELLS</i>	<i>UI.STORE_COLOR</i>
<i>UI.FREE_COLORS</i>	

La gestion de la couleur est en général traitée au niveau du composant "*PRIMITIVES GRAPHIQUES*", quel qu'il soit. Toutefois, certains programmeurs ont un besoin impérieux d'offrir un widget de saisie de couleurs interactif. Pour ces programmeurs, *UI* interface les fonctions *X-Window* nécessaires à la réalisation de ce besoin.

La fonction *UI.ALLOC\_COLOR\_CELLS* alloue des cellules dans la colormap par défaut d'un display. Cette procédure est identique à *XAllocColorCells*. Le nombre de plans alloués est déterminé par la taille du tableau de nom formel *PLANE\_MASKS*. Les cellules alloués sont privées à ce client X et la fonction *UI.STORE\_COLOR* permet de changer le

contenu de la cellule. Les cellules peuvent être restituées par un appel à la procédure **UI.FREE\_COLORS**.

## 8.3 Aide à la mise au point de l'IHM

### 8.3.1 boîte de mise au point

La mise au point d'une IHM peut parfois s'avérer assez difficile. Les intrinsics effectuent peu de tests quant à la validité d'une requête. Il est fréquent qu'un programme non conforme à la spécification écrite des intrinsics fonctionne parfaitement, du moins pendant un certains temps ... Le typage de **UI** offre déjà une première protection contre les erreurs élémentaires. Sinon, une boîte de "debugging" est disponible pour toute application qui utilise **UI**. Celle-ci s'active en donnant la valeur **True** à la variable d'environnement **UI\_DEBUG** (nom logique sous **OpenVMS**). Elle permet de naviguer dans la hiérarchie des widgets et d'inspecter certains de leurs attributs ou de leurs ressources. Quelques opérations, **manage/unmanage** par exemple, peuvent être appliquées sur les widgets de l'interface. **UI** offre quelques services pour les besoins de cette boîte de "debugging", mais ceux-ci ne sont pas d'usage général. Ils ne sont d'ailleurs pas documentés ici (fonctions **UIB.CREATE\_UI\_TOOLS\_INTERFACE**, **UIT.SET\_TRACE\_CALLBACKS**, **UIT.SET\_TRACE\_DESTINATION** et **UIT.SHOW\_TRACE\_STRING**).

L'interface de mise au point contient les menus suivants :

File:	opérations sur le dialogue (Separator, Version, Close)
Trace:	trace sur les callbacks, les actions ou les événements (Callback, Actions, Events)
Hierarchy:	traversée de la hiérarchie (AppShells, Shells, Children, PopupChildren, Descendant, Parents)
Widget:	opérations sur les widgets (Set, Show, Manage, Unmanage)

La boîte contient une zone de texte, dans laquelle les opérations écrivent leurs messages. Cette boîte contient un popup menu avec les choix **Clear**, **Print**, **Save**, **Search** et **Customize**. L'action **File/Separator** ajoute un séparateur à la fin de cette zone de texte. Les choix du menu trace permettent de tracer les appels de callbacks, les actions (au sens des intrinsics) et les événements (au sens de **X-Window**). Pour naviguer dans la hiérarchie, commencer par activer le choix **Hierarchy/AppShell**. Choisir un widget dans la liste des applications shells affichés en double cliquant sur son widget id. Ce widget devient alors le widget courant. Il est alors possible de "descendre" dans la hiérarchie à l'aide des opérations **Hierarchy/Children**, **Hierarchy/PopupChildren** et **Hierarchy/Descendant**. Il est également possible de "remonter" dans la hiérarchie à l'aide de l'opération **Hierarchy/Parents**. Le choix **Widget/Set\_current** permet de choisir le widget courant en indiquant son adresse en hexa. Il va de soi que c'est une opération réservée pour le debugging de bas niveau. Le choix **Widget/Show\_current** permet d'afficher, pour le widget courant, ses attributs et quelques unes de ses ressources. Certains attributs ou ressources dépendent de la classe du widget. Enfin, les choix **Widget/Manage** et **Widget/Unmanage** permettent de gérer ou de ne pas gérer le widget courant.

### 8.3.2 Cascade deadlock

La fonction **UIT.UNGRAB\_CASCADE** permet de debugger les callbacks qui sont activées sur le déroulement d'un pulldown menu (cascading callback). En effet, si l'on pose un point d'arrêt sur une telle callback, il se produit un deadlock (blocage infini) du serveur **X**. Lorsqu'un menu pulldown est déroulé, les intrinsics réservent (grab) le curseur à ce client **X**. Lorsque le debugger active le point d'arrêt, il n'est plus possible de donner le focus au

debugger car celui-ci est attribué a un autre client lui-même bloqué par le debugger... La solution la plus simple à ce problème est d'ordre matériel : brancher un terminal sur la station et y exécuter le programme. Une autre solution est purement logicielle. Il faut inclure au début du code de la callback les instructions suivantes :

```
-- Eviter le deadlock lors du debugging
if UI.CASCADE_DEBUG then
  UNGRAB_CASCADE(DISPLAY_OF(W));
end if ;
```

Enfin, avant l'exécution, donner à la variable d'environnement **UI\_CASCADE\_DEBUG** (nom logique sous *OpenVMS*) la valeur *True*.

### 8.3.3 Mise au point de bas niveau

Le debugger de certains compilateurs coexiste mal avec les callbacks et avec les modules écrits en langage *C*. L'affichage de traces est commandé par la valeur de la variable d'environnement **UI\_BODY\_DEBUG** (nom logique sous *OpenVMS*). Sa valeur doit être assignée à un nombre entre 0 et 3 correspondant au seuil de trace désiré. Le tableau ci-dessous donne pour chaque seuil, les traces correspondantes affichées.

nom symbolique	val.	description
<b>UI_TRACE_NO</b>	0	aucune trace affichée
<b>UI_TRACE_INIT</b>	1	trace des initialisations
<b>UI_TRACE_CALLBACK</b>	2	appels des callbacks
<b>UI_TRACE_ALL</b>	3	toutes traces affichées

Les modules *Ada* proposent également un système de trace équivalent. Les traces sont toutefois beaucoup plus succinctes, car, en général, les debuggers *Ada* sont suffisants. La variable d'environnement (nom logique sous *OpenVMS*) doit porter le nom *nom-de-paquetage\_DEBUG* et avoir la valeur 0 (trace inactive) ou 1 (trace active).

## 8.4 Gestion des erreurs

La section précédente décrivait exclusivement les services offerts pour la phase de mise au point du cycle de vie du logiciel. Cette section, au contraire, décrit les services de gestion des erreurs nécessaire à la robustesse du programme.

La gestion des erreurs se repose sur le composant **ERROR\_MANAGER** de la couche **K**. Le composant **UI** avant de propager une exception dépose les raisons de l'exception à l'aide du service **ERROR\_MANAGER.OPEN\_ERROR** ou **ERROR\_MANAGER.ADD\_ERROR**. Les exceptions levées explicitement par le composant **UI** ont toujours, sauf indication contraire dans la documentation de référence, le nom **UI.INTERFACE\_ERROR**. Dans un traite-exception de l'application il est possible d'accéder à la raison de l'erreur par le service **UI.SHOW\_AND\_CLOSE\_ERROR**, qui renomme **ERROR\_MANAGER.SHOW\_AND\_CLOSE\_ERROR**.

Enfin, le service **UIT.PUT\_AND\_CLOSE\_ERROR** crée et affiche une boîte avec ce même message d'erreur<sup>9</sup>.

<sup>9</sup>La solution proposée a toutefois un inconvénient majeur. Il n'est pas possible dans un même traite-exception de distinguer une erreur de l'utilisateur d'une erreur de programmation. Une solution serait d'introduire une exception ou une raison **DIALOG\_ERROR**.

## 8.5 Paquetage **MOTIF\_TERMINAL**

Le paquetage *UI\_BOXES* contient les primitives pour construire un terminal à partir de widgets Motif. La section "Terminal et text box" du chapitre "Création et gestion de widgets" décrit ces primitives. Le paquetage *TERMINAL\_IO* de la couche *K* permet de substituer au terminal physique sur lequel opèrent par défaut les fonctions *UTIL\_IO.PUT\** et *UTIL\_IO.GET\_\**, un terminal virtuel quelconque.

Le paquetage *MOTIF\_TERMINAL*, à partir des services des deux paquetages cités ci-dessus, offre un terminal Motif. Le code applicatif, en général existant, qui lit et écrit sur un terminal peut ainsi être réutilisé sans changement, dans une application à interface Motif. Par contre, on perd dans ce cas l'avantage des interfaces *Motif*, c'est-à-dire l'aspect événementielle. Le terminal virtuel doit être "activé" lorsqu'un sous-programme applicatif fonctionnant sous Motif réalise des lectures/écritures sur le terminal d'*UTIL\_IO* par les fonctions *PUT* et *GET*.

Le terminal virtuel s'utilise en général selon la séquence suivante :

- activer le terminal virtuel
- réaliser des écritures/lectures sur le terminal
- désactiver le terminal virtuel

Il est possible d'"empiler" plusieurs terminaux virtuels. Naturellement, la désactivation (désempilement) doit avoir lieu dans l'ordre inverse de l'activation (empilement). L'empilement des terminaux peut, par exemple, servir dans une callback d'action à saisir quelques paramètres auprès de l'utilisateur.

Cette solution est toutefois incomplète car elle ne garantit pas la modalité des lectures sur le terminal. En effet, rien n'empêche l'utilisateur d'aller cliquer sur des menus ou des boutons de l'IHM alors que le terminal virtuel est actif. Dans le cas où le code applicatif réalise des lectures (avec ou sans écritures), la séquence ci-dessus doit être modifiée comme suit :

- activer le terminal virtuel
- commencer une interaction modale
- réaliser des lectures/écritures sur le terminal
- terminer l'interaction modale
- désactiver l'interaction modale

L'activation et le commencement de l'interaction modale sont séparés car un code applicatif qui ne réalise que des écritures n'a pas le problème de la modalité.

Le spécification du paquetage générique *MOTIF\_TERMINAL* est la suivante :

---

```

with USER_INTERFACE ; use USER_INTERFACE ;
package MOTIF_TERMINAL is
  type MEMORY is private ;
  generic
    TERMINAL, OUTPUT, COMMAND : in out WIDGET ;
  package TERM is
    procedure SET_TERMINAL (    -- Active ce terminal et memorise le
      MEM : out MEMORY) ;      -- terminal precedemment actif
    procedure RESTORE_TERMINAL (-- Desactive le terminal actif, et reactive
      MEM : in MEMORY) ;      -- le precedent
    procedure SET_MODAL ;      -- Commence une operation modale dans ce
      -- terminal c'est-a-dire un groupe de lectures
      -- La fonction peut modifier la sensibilite des widgets
    procedure CANCEL_MODAL ;   -- Termine l'operation modale
    function IS_ACTIVE         -- Indique que la memoire de ce terminal
      return Boolean ;         -- est active (saisies de commandes en cours)
    function IS_NULL           -- Indique que la memoire de ce terminal
      return Boolean ;         -- est vide (est different de not IS_ACTIVE)
  end TERM ;
private
  ...
end MOTIF_TERMINAL ;

```

Le paquetage doit être instancié avec les widgets *TERMINAL*, *OUTPUT* et *COMMAND* construits par la procédure *UIB.CREATE\_TERMINAL*. Une variable du type privé *MEMORY* est retournée à chaque activation d'un terminal virtuel. Elle sert de mémoire et permet l'implantation de l'empilement. L'activation est réalisée par la procédure *SET\_TERMINAL* et la désactivation par la procédure *RESTORE\_TERMINAL*. On déclare le commencement d'une interaction modale par un appel à la procédure *SET\_MODAL*, et la terminaison par un appel à *CANCEL\_MODAL*. Ces deux sous-programmes utilisent l'utilitaire de gestion de la modalité multi-display décrit ci-dessus dans ce chapitre. Visuellement, la modalité se traduit par l'insensibilisation du reste de l'IHM.

L'exemple ci-dessous montre les extraits significatifs d'une application qui utiliserait un tel terminal pour saisir occasionnellement des commandes textuelles.

```
with MOTIF_TERMINAL ;

package ... is

    TERMINAL, TEXT_OUTPUT, COMMAND : WIDGET ;
    TERMINAL_MEMORY : MOTIF_TERMINAL.MEMORY ;

package MOTIF_TERM is new MOTIF_TERMINAL.TERM (TERMINAL, TEXT_OUTPUT, COMMAND) ;

-- Les deux procedures BEGIN_TEXTUAL_DIALOG et END_TEXTUAL_DIALOG
-- encadre les E/S terminal. Elles ont pour fonction de faire apparaitre ou
-- disparaitre la zone de commande du terminal et de gérer la modalité

-----
procedure BEGIN_TEXTUAL_DIALOG is
begin
    UI_TOOLS.SHOW_COMMAND (COMMAND) ;
    MOTIF_TERM.SET_MODAL ;
end BEGIN_TEXTUAL_DIALOG ;

-----
procedure END_TEXTUAL_DIALOG is
begin
    MOTIF_TERM.CANCEL_MODAL ;
    UI_TOOLS.HIDE_COMMAND (COMMAND) ;
end END_TEXTUAL_DIALOG ;
```

```

-----
procedure CREATE_IHM is
begin -- MAIN_MENU

    ...

    --
    -- Creation de l'IHM du terminal
    --
    TEXT_OUTPUT := CREATE_SCROLLED_TEXT_OUTPUT("Output", ... , POPUP => True) ;
    COMMAND     := CREATE_COMMAND ("PlayCommand", ... , TEXT => TEXT_OUTPUT) ;
    TERMINAL    := PARENT_OF(COMMAND) ;

    --
    -- Association d'un terminal logique a l'IHM terminal
    --
    MOTIF_TERM.SET_TERMINAL (TERMINAL_MEMORY) ;

    ...

end MAIN_MENU ;

-----
procedure EXIT_APP is -- Appelée systématiquement lors de la sortie
begin                -- de l'application

    ...

    MOTIF_TERM.RESTORE_TERMINAL (TERMINAL_MEMORY) ;

    ...

end EXIT_APP;

-----
procedure INITIALIZE is-- Appelée lors de l'initialisation de l'application
begin

    ...

    BEGIN_TEXTUAL_DIALOG;
    GET_... ;
    ...
    END_TEXTUAL_DIALOG;

    ...

end INITIALIZE ;

end ... ;

```

Cet exemple montre clairement l'indépendance totale entre l'IHM du terminal et la fonction de "terminal virtuel" offert par *MOTIF-TERMINAL*.

Les autres fonctions du paquetage *MOTIF\_TERMINAL* sont d'usage moins fréquent et sont décrites plus succinctement. La fonction *ULIS\_ACTIVE* indique si le terminal associé à cette instanciation est actuellement activé. La fonction *ULIS\_NULL* indique si une mémoire est vide.

## 8.6 Le browser

### 8.6.1 Introduction

Le composant *BROWSER* permet de visualiser graphiquement une hiérarchie dynamique de noms. Le composant est introduit à partir d'un exemple d'utilisation. La présentation est ainsi plus concrète et ne nuit pas trop à la généralité.

La figure ci-dessous illustre l'utilisation du *BROWSER* pour la représentation de la hiérarchie des objets *ESCADRE*. Comme le composant est capable de gérer une hiérarchie dynamique, les objets *ESCADRE* sont insérés et retirés de cette représentation au fur et à mesure du déroulement d'une simulation. Dans cet exemple, il est important de distinguer ce

qui offert par le composant **BROWSER** de ce qui est ajouté par l'applicatif. La représentation hiérarchique est fournie par le composant **BROWSER**. Celle-ci a lieu dans une simple *XmlList Motif*. Le reste de l'IHM (fenêtre, rangée de boutons, popup menu) doit être ajouté par l'applicatif. Certaines réactions des boutons de cette partie de l'IHM sont directement associées à des services programmatiques du composant **BROWSER**. Ce choix permet une grande liberté dans le "look&feel" de l'IHM. Ce choix permet également d'ajouter des composants d'IHM spécifiques à l'application. La présence d'un caractère "+" (ce caractère est paramétable) en début de ligne indique que l'objet en question a des descendants mais que ceux-ci ne sont pas visualisés à ce moment là. L'affichage des objets est contrôlé par l'utilisateur et non pas par le programmeur. L'action qui consiste à masquer les descendants d'un objet s'appelle "collapse". L'action inverse, s'appelle "expand". Un expand peut être surfacique -limité aux enfants- ("Children") ou profond ("Descendant").

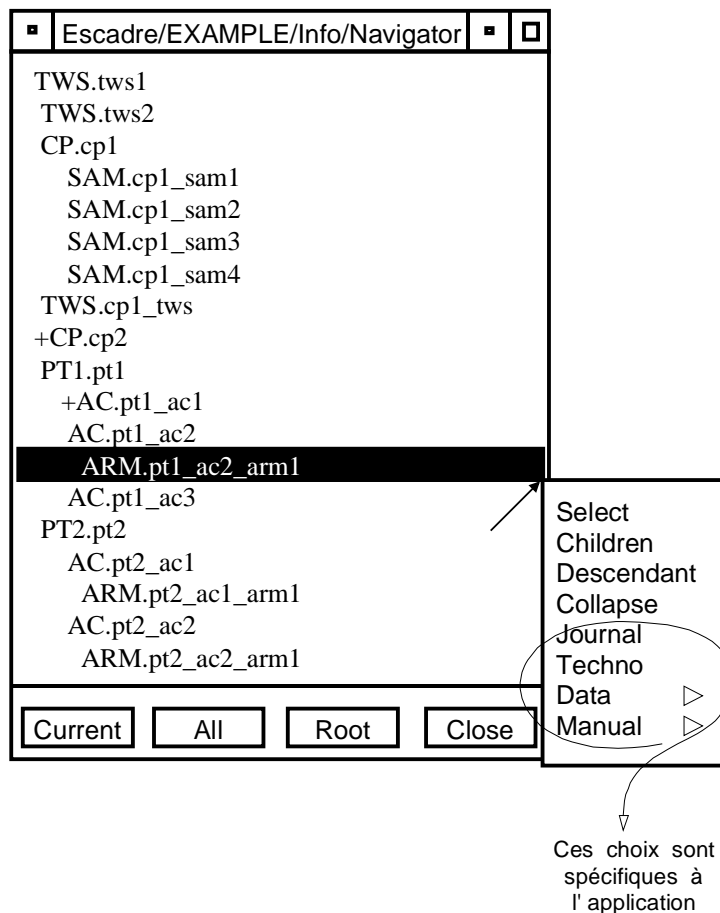


Figure 8-1 Exemple d'utilisation du **BROWSER**

Le composant **BROWSER** désigne par *élément (item)* un objet de la hiérarchie.

## 8.6.2 Définition d'une hiérarchie

On précise dans cette section les conditions qui doivent être respectées par l'applicatif pour que la hiérarchie soit "acceptable" par le composant **BROWSER**. Le composant **BROWSER** ne visualise pas un arbre mais une forêt. La hiérarchie est connue du composant **BROWSER** par l'intermédiaire de fonctions applicatives **CHILDREN\_OF**, **ROOTS** et **PARENT\_OF**. La fonction **ROOTS** retourne les racines de la forêt. La fonction **CHILDREN\_OF** retourne les enfants d'un objet quelconque de la hiérarchie. La fonction **PARENT\_OF** retourne l'objet père d'un objet distinct d'un objet racine. Cette dernière fonction est redondante vis-à-vis des deux autres. Les conditions requises sont les suivantes :

- 1. Un élément ne doit pas être l'enfant de deux pères différents
- 2. Un élément ne peut être à la fois racine et enfant d'un élément
- 3. Le père d'un enfant quelconque d'un élément est cet élément
- 4. Un élément racine n'a pas de père

Ces conditions semblent "évidentes" mais ce sont les axiomes définissant une hiérarchie.

Si la condition (1) n'est pas respectée, on est en présence non pas d'une hiérarchie mais d'un graphe orienté.

### 8.6.3 Services offerts par le composant

Le composant offre les services de visualisation d'une hiérarchie. Les deux services programmatiques *EXPAND* et *COLLAPSE*, une fois associés à un composant de l'IHM, permettent à l'utilisateur de naviguer au sein de cette hiérarchie. Les services *ACTIVATE* et *DEACTIVATE* permettent de "geler" temporairement la gestion géométrique et la visualisation de la hiérarchie. On peut faire appel à ces deux fonctions lorsque la boîte de dialogue qui contient le *BROWSER* devient affichée ou n'est plus affichée. On peut se référer à la spécification du paquetage qui est donnée ci-après pour avoir une liste exhaustive des services programmatiques.

Le composant fait appel *ERROR\_MANAGER* pour la gestion de ses erreurs.

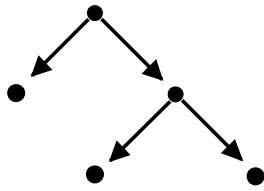
#### Limitations du composant

Le composant n'offre pas de fonction *REPARANT* permettant de déplacer une branche. Il est par contre possible d'obtenir un service équivalent à partir des services *DELETE* et *CREATE*. Cela consiste à retirer de la hiérarchie l'élément à déplacer par un appel à *DELETE* (ses descendants sont alors également retirés), puis de le ré-insérer avec un nouveau père. La fonction *DELETE* accepte les éléments qui ont des descendants.

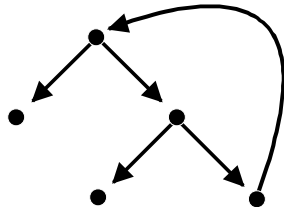
Le composant se protège contre quelques erreurs possibles du client. Certaines opérations (*EXPAND*, *COLLAPSE*) exigent que l'élément appartienne à la hiérarchie. Si tel n'est pas le cas, lors de l'invocation d'une de ces opérations, le composant lève *BROWSER\_ERROR* avec un message de raison *BR\_E\_NOITEM*.

### 8.6.4 Conditions sur les fonctions définissant la hiérarchie

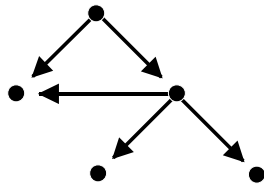
Ces conditions sont essentiellement la traduction "informatique" des conditions énoncées dans la section "Définition de la hiérarchie".



a. hiérarchie sans cycle et sans circuit



b. circuit



c. cycle

Figure 8-2 Définition d'un circuit et d'un cycle

La hiérarchie est connue du composant par l'intermédiaire de trois fonctions paramètres du générique. Ces fonctions ont pour nom formel **CHILDREN\_OF**, **ROOTS** et **PARENT\_OF**. La fonction **PARENT\_OF** est redondante vis-à-vis des deux autres et doit être en conformité avec celles-ci. Cette conformité est détaillée ci-après.

#### Conditions statiques élémentaires

Les éléments retournés par les fonctions **ROOTS** et **CHILDREN\_OF** doivent être ordonnés selon la fonction "<", également paramètre formel du générique.

La fonction **PARENT\_OF** doit retourner **NULL\_ITEM** pour tous les éléments racines et seulement ceux-là.

L'intersection du résultat de **ROOTS** et du résultat de **CHILDREN\_OF** appliqué à un élément quelconque doit être vide.

```
declare
  function "and" (L1,L2 : ITEM_LIST)    -- retourne les éléments communs à ces deux listes
    return ITEM_LIST ;                 -- (intersection)
  I : ITEM := ... ;                    -- un élément
  L : constant ITEM_LIST := ROOTS and CHILDREN_OF(I);
begin
  if L'Length > 0 then
    -- ne doit pas arriver
  end if ;
end ;
```

Toutefois on peut remarquer que ce cas d'erreur est similaire à la présence d'un cycle dans l'une des branches et on peut se reporter aux conclusions ci-dessous concernant les cycles.

#### Cohérence de **PARENT\_OF** avec **CHILDREN\_OF** et **ROOTS**

La fonction *PARENT\_OF* doit être conforme avec les deux autres fonctions. Si on applique à un élément quelconque la fonction *CHILDREN\_OF* et que l'on applique la fonction *PARENT\_OF* à l'un des enfants retournés, on doit retrouver l'élément initial :

```
declare
  I : ITEM := ... ; -- un élément
  CHILDREN : constant ITEM_LIST := CHILDREN_OF(I) ;
begin
  for C in CHILDREN'Range loop
    if I /= PARENT_OF(CHILDREN(C)) then
      raise BROWSER_ERROR ;
    end if ;
  end loop ;
end ;
```

Un même élément ne peut être enfant de deux pères distincts, c'est-à-dire en termes "informatiques" :

```
declare
  P1 : ITEM ; -- un élément
  P2 : ITEM ; -- un autre élément
begin
  if P1 /= P2 then -- Si P1 et P2 sont deux éléments distincts
    declare
      LIST : constant ITEM_LIST := CHILDREN_OF(P1) and CHILDREN_OF(P2) ;
    begin
      if LIST'Length /= 0 then
        -- ne doit pas arriver
      end if ;
    end ;
  end if ;
end ;
```

### *CHILDREN\_OF* et *ROOTS* ne doivent pas définir un graphe avec circuit

Les fonctions *CHILDREN\_OF* et *ROOTS* qui définissent la hiérarchie initiale pourraient également définir un graphe.

Les notions de cycle et de circuit qui caractérisent un graphe sont illustrées par la figure ci-dessus.

Pour se prémunir contre un éventuel circuit le composant *BROWSER* vérifie que chaque enfant retourné par *CHILDREN\_OF* n'appartient pas à la liste de ses ancêtres. Si tel n'est pas le cas le composant propage *BROWSER\_ERROR* avec la raison *BR\_E\_CIRCUIT*.

Le composant *BROWSER* tolère la présence de cycle dans la hiérarchie (qui n'en est alors plus une !). Cela signifie que l'on tolère que la fonction *CHILDREN\_OF* retourne un élément qui soit un colatéral (dans une branche distincte). Dans ce cas, à la visualisation, une même sous-branche est visible plusieurs fois.

### Conditions lorsque la hiérarchie est dynamique

L'insertion et la suppression d'élément(s) dans la hiérarchie est annoncée au composant *BROWSER* en appelant respectivement les fonctions *CREATE* et *DELETE*. Si les fonctions *CHILDREN\_OF*, *ROOTS* ou *PARENT\_OF* retournent des éléments dont l'apparition ou la disparition n'ont pas été annoncés au composant *BROWSER*, le comportement du composant est alors imprévisible. Il est probable que l'exécution d'une opération génère alors une erreur *BROWSER\_ERROR* avec la raison *BR\_E\_INTERNAL*. Se prémunir contre ce type d'erreur poserait des problèmes de temps de réponse.

Les ajouts et suppressions d'éléments dans la hiérarchie doivent être synchronisés avec les retours des fonctions *CHILDREN\_OF*, *PARENT\_OF* et *ROOTS*. Lors de l'appel à la fonction *DELETE* (suppression d'un élément de la hiérarchie), l'élément effacé doit encore être retourné par *CHILDREN\_OF* et doit accepter un appel à *PARENT\_OF*. L'appel à

**DELETE** correspond donc à une annonce de destruction. La destruction d'un élément se fait en deux étapes, dont l'ordre est important :

1. retirer l'élément de la hiérarchie (par un appel à **DELETE**)
2. détruite effectivement l'élément côté client

Un élément ajouté ou supprimé peut avoir un ou plusieurs enfant(s).

```

Spécification programmatique
-----
-- ++
-- BIBLIOTHEQUE      : CAD/GL/GRAPHIC/BROWSERS_.ADA
--
-- NOM DE l'UNITE    : BROWSERS  Specification
--
-- DATE DE MISE A JOUR : 12-JAN-1993
--
-- ANALYSTE          : E. BERRY
--
-- DESCRIPTION :
--
--   Le composant offre les services minimum d'affichage et de visualisation
--   d'une hierarchie statique ou dynamique. La visualisation a lieu dans une
--   XmlList Motif. La hierarchie est represente par une indentation des elements
--   de la liste.
--
--   Le composant doit etre instancie pour une hierarchie. Ensuite il est
--   possible de creer plusieurs vues de cette hierarchie. Chaque vue est
--   represente par une variable de type BROWSER. Certaines operations agissent
--   sur un browser, c'est-a-dire sur une vue de la hierarchie (EXPAND par
--   exemple). D'autres agissent sur la hierarchie elle-meme (CREATE par exemple).
--
--   La hierarchie est connue du composant BROWSER par l'intermediaire de
--   de sous-programmes formels du paquetage generique. Ainsi la hierarchie est
--   parcouru a l'aide des fonctions ROOTS, CHILDREN_OF et PARENT_OF.
--   Les elements retournes par ROOTS et CHILDREN_OF doivent etre classes selon
--   "<". Il n'est pas exige que la fonction NAME_OF retourne un nom unique.
--   Toutefois pour eviter au composant un parcours permanent de la hierarchie,
--   il doit etre prevenu de toute disparition ou insertion d'element dans la
--   hierarchie par un appel au sous-programme DELETE ou CREATE.
--
--   Le composant utilise ERROR_MANAGER pour la gestion de ces erreurs et
--   leve l'exception BROWSER_ERROR.
--
--   Le composant se protege contre quelques erreurs possibles du client.
--   Certaines operations (EXPAND, COLLAPSE) exigent que l'element appartientent
--   a la hierarchie. Si tel n'est pas le cas, lors de l'invocation d'une de
--   ces operations, le composant propage BROWSER_ERROR (BR_E_NOITEM).
--
--   Les fonctions CHILDREN_OF et ROOTS permettent de definir la hierarchie
--   courante mais une telle definition pourrait egalement servir a definir un
--   graphe. Pour se proteger contre d'eventuel circuit, le composant verifie que
--   chaque enfant retourne par CHILDREN_OF n'appartient pas a la liste de ces
--   ancetres. Si tel n'est pas le cas le composant propage BROWSER_ERROR
--   (BR_E_CIRCUIT). (Un autre choix etait egalement envisageable: supprimer
--   le fils de la liste). Les cycles sont par contre toleres. On est en presence
--   d'un cycle si la fonction CHILDREN_OF retourne un
--   element qui est un colateral (dans une branche distincte) aucune erreur
--   n'est propage. La consequence a la visualisation sera qu'une meme
--   sous-branche sera visible plusieurs fois.
--
--   Par contre le composant n'est pas protoge contre certaines erreurs. Si
--   apres l'instanciation, les fonctions CHILDREN_OF, ROOTS ou PARENT_OF
--   retourne des elements qui n'ont pas ete insere dans la hierarchie par
--   la procedure CREATE le comportement du composant est
--   imprevisible mais generera probablement BROWSER_ERROR (BR_E_INTERNAL).
--   On considere qu'une telle erreur de la part du client est trop grossiere et
--   qu'il serait trop couteux de s'en proteger.
--
--   Le composant n'offre pas de fonction REPARENT permettant de deplacer une
--   branche. Il est necessaire pour cela de retirer l'element de la hierarchie
--   (il peut avoir des descendants), puis de le reinserer avec un nouveau pere
--   (la fonction CHILDREN_OF doit pouvoir retourner ses enfants).
--
--   Les retours des fonctions CHILDREN_OF, PARENT_OF et ROOTS et les appels
--   a DELETE et CREATE doivent etre coherent. Lors de l'appel a la fonction
--   DELETE, l'element efface doit encore etre retourne par CHILDREN_OF et doit
--   accepter PARENT_OF.
--
-- MODIFICATIONS :
-- EXTENSIONS:
--
--   . commande filter (c'est a dire un SEARCH)
--
--   . fonction LENGTH (B : BROWSER) retournant le nombre d'element VISIBLE
--     dans la hierarchie.
--
-- IMPORTATION :
--
-- with SUPER_STANDARD ; use SUPER_STANDARD ;
-- with USER_INTERFACE ; use USER_INTERFACE ;
-----

```

```

generic
  type ITEM is range <> ;
  NULL_ITEM : in ITEM ;
  with function "<"(LEFT,RIGHT: ITEM) return Boolean is <> ;
  type ITEM_LIST is array (Positive range <>) of ITEM ;
  with function NAME_OF (I : ITEM) return String ;
  with function ROOTS return ITEM_LIST;
  with function CHILDREN_OF (I : ITEM) return ITEM_LIST;
  with function PARENT_OF (I : ITEM) return ITEM ;
  INDENTATION : in String := 2*' ' ;
  DESCENDANT : in String := ">" ;
  NO_DESCENDANT : in String := " " ;

-----
package BROWSERS is

  type BROWSER is limited private ; -- une vue de la hierarchie

  BROWSER_ERROR : exception ;

  ----- gestion de la hierarchie -----

  -- Si la hierarchie manipulee n'est pas statique (c'est a dire si des
  -- elements sont susceptibles d'apparaître et de disparaître), il est
  -- necessaire de prevenir le composant par l'intermediaire des fonctions
  -- DELETE et CREATE.

  procedure DELETE ( -- Indique que cet
    I : in ITEM) ; -- element est supprime de la hierarchie

  procedure CREATE ( -- Indique que cet
    I : in ITEM) ; -- element est ajoute a la hierarchie

  ----- gestion de la hierarchie visible -----

  -- Tous les elements de la hierarchie ne sont pas systematiquement visible.
  -- Toutes sous-branches peut etre masque et represente simplement
  -- par sa racine precede d'un signe distinctif (indiquant la presence
  -- d'enfant non visualise).

  procedure EXPAND ( -- Rend visible (si il ne le sont pas deja)
    B : in out BROWSER ; -- pour ce browser
    I : in ITEM ; -- les descendants de cet item jusqu'a ce
    DEPTH : in Positive:= 1); -- niveaux (1=fils)

  procedure COLLAPSE ( -- Rend invisible les descendants
    B : in out BROWSER ; -- pour ce browser
    I : in ITEM) ; -- de cet element

  procedure COLLAPSE_OR_EXPAND ( -- Realise la fonction EXPAND ou COLLAPSE
    B : in out BROWSER ; -- pour ce browser en fonction de la
    I : in ITEM) ; -- visibilite de cet element

  procedure FOCUS ( -- Rend visible
    B : in out BROWSER ; -- pour ce browser
    I : in ITEM) ; -- cet element et ces ancetres

  ----- gestion de la selection -----

  -- Le composant peut etre interroge pour connaitre l'element selectionne.
  -- Il est egalement possible de deposer une callback pour etre prevenue
  -- de toute selection/deselection.

  function SHOW_SELECTED ( -- Retourne l'element selectionne
    B : in BROWSER) -- ou NULL_ITEM pour ce browser
  return ITEM ;

  procedure SET_SELECTED ( -- Selectionne
    B : in BROWSER ; -- pour ce browser
    I : in ITEM) ; -- cet element

  procedure UNSELECT ( -- Deselectionne tout element selectionne
    B : in BROWSER) ; -- pour ce browser

  procedure POST_SELECT_CB ( -- Ajoute cette procedure
    B : in BROWSER ; -- pour ce browser
    CB : in CB_NAMED_ADDRESS) ;-- a la callback liste de selection

  procedure UNPOST_SELECT_CB ( -- Retire cette procedure de
    B : in BROWSER ; -- pour ce browser
    CB : in CB_NAMED_ADDRESS) ;-- la callback liste de selection

  ----- Constructeur de widget -----

```

```

procedure CREATE (
  B      : in out BROWSER ;
  NAME   : in      String ;
  PARENT : in      WIDGET ;
  ARGS   : in      NAMED_ARG_LIST := NULL_NAMED_ARG_LIST) ;

function IS_NULL (
  B      : BROWSER)
return Boolean ;

procedure MANAGE (
  B      : in out BROWSER) ;

procedure UNMANAGE (
  B      : in out BROWSER) ;

procedure DELETE (
  B      : in out BROWSER);

----- personalisation -----

-- La personnalisation du comportement du composant peut etre effectuee soit
-- par des services programmatiques soit par l'intermediaire d'une boîte de
-- dialogue.

procedure MAP_CUSTOMIZE (
  B      : in out BROWSER ;
  NAME   : in      String ;
  PARENT : in      WIDGET ) ;

type INSERT_VISIBILITY is (
  IV_ALWAYS,
  IV_NETHER,
  IV_BROTHER) ;

procedure SET_INSERT_VISIBILITY (
  INSVIS : in INSERT_VISIBILITY) ;

-----
private

  type BROWSER is new Natural ;

end BROWSERS ;
-----

```

### 8.6.5 Exemple

A titre d'exemple, un browser de fichier a été écrit à partir du composant **BROWSER** et des primitives de parcours de directory de **PORTABLE\_FILES**. On ne donne que quelques extraits de code significatifs mais le code complet est consultable en ligne<sup>10</sup>.

Indépendamment de la syntaxe d'un fichier (spécifique en général à un système d'exploitation), les concepts sous-jacents (directory, path, ...) sont les mêmes. Ce browser de fichier est écrit à partir de services qui donnent une vision unifiés d'une spécification de fichier.

L'IHM du programme n'est pas le thème central ici, aussi n'est-il pas reproduit ici. Si nécessaire, celui-ci est consultable en ligne.

Les fonctions de définition de la hiérarchie nécessaires à l'instanciation du générique **BROWSERS** se trouvent dans le paquetage **UTEST\_BROWSER\_API** dont la spécification est donnée ci-dessous

<sup>10</sup>couche G35, fichier t\_browser.ada

```
package UTEST_BROWSER_API is

  type ITEM is new Integer ;
                                -- represente un element de la hierarchie
                                -- ici un fichier (au sens general)

  NULL_ITEM : constant ITEM := 0 ;

  type ITEM_LIST is array (Positive range <>) of ITEM ;

  function ITEM_ID_OF (          -- Construit pour
    NAME : String)              -- le fichier portant ce nom
    return ITEM ;                -- son element associe

  function "<" (                  -- Indique si
    LEFT,                        -- ce fichier precede
    RIGHT: ITEM)                 -- ce fichier
    return Boolean ;

  function CHILDREN_OF (        -- Retourne les enfants de ce
    I : ITEM)                   -- fichier
    return ITEM_LIST ;

  function ROOTS                -- Retourne les fichiers racines
    return ITEM_LIST ;

  function PARENT_OF (          -- Retourne le fichier pere (directory)
    I : ITEM)                   -- de ce fichier
    return ITEM ;

  function ITEM_NAME_OF (      -- Retourne le nom (tail) de
    I : ITEM) return String ;  -- ce fichier

  procedure SET_ROOT (          -- Positionne la racine a partir de la
    ROOT : String) ;           -- quelle on browse

  procedure SET_FILTER (        -- Positionne un filtre sur les
    FILTER : String             -- noms de fichiers
      := FILE_PARSER.ANY_TAIL) ; -- visualise

  function GET_ROOT             -- Retourne la racine
    return String ;            -- courante

  function GET_FILTER           -- Retourne le filtre
    return String ;            -- courant

  package FILE_BROWSER is new BROWSERS (
    ITEM          => ITEM,
    NULL_ITEM     => NULL_ITEM,
    "<"           => "<",
    ITEM_LIST     => ITEM_LIST,
    NAME_OF       => ITEM_NAME_OF,
    ROOTS         => ROOTS,
    CHILDREN_OF   => CHILDREN_OF,
    PARENT_OF     => PARENT_OF,
    INDENTATION   => 3*' ',
    DESCENDANT    => ">",
    NO_DESCENDANT => " ") ;

  BROWSER : FILE_BROWSER.BROWSER ;

  function SHOW_SELECTED_NAME    -- Retourne le nom du fichier
    return String ;              -- selectionne dans le browser

  -- Test de certaines erreurs

  TEST_BR_E_PARENT : Boolean := False ;
  TEST_BR_E_CIRCUIT : Boolean := False ;
  TEST_BR_E_CYCLE : Boolean := False ;

end UTEST_BROWSER_API ;
```

Des extraits du corps de ce paquetage sont également donné ci-dessous. On a surtout insisté sur la fonction *CHILDREN\_OF*, en conservant uniquement les étapes importantes :

```

package body UTEST_BROWSER_API is
    ...
    -----
    function ITEM_ID_OF (NAME : String) return ITEM is
        N : UN.NAME := UN.CREATE(NAME, SHARED => True) ;
        function UNCHECKED_TO_ITEM is new UNCHECKED_CONVERSION (
            SOURCE => UN.NAME,
            TARGET => ITEM) ;
    begin
        return UNCHECKED_TO_ITEM(N);
    end ITEM_ID_OF ;
    -----
    function NAME_OF (I : ITEM) return String is
        function UNCHECKED_TO_NAME is new UNCHECKED_CONVERSION (
            SOURCE => ITEM,
            TARGET => UN.NAME) ;
    begin
        return UN.IMAGE(UNCHECKED_TO_NAME(I)) ;
    end NAME_OF ;
    -----
    function CHILDREN_OF (I : ITEM) return ITEM_LIST is
        FILE : constant String := NAME_OF(I) ;
    begin
        if IS_DIR(FILE) then
            declare
                LIST : constant PORTABLE_FILES.FILE_LIST
                    := PORTABLE_FILES.FIND_FILE (
                        TO_DIR(FILE),
                        NAME => UN.IMAGE(FILTER)) ;
                ITEMS : ITEM_LIST(1..LIST.LENGTH) ;
            begin
                for RANK in ITEMS'Range loop
                    ITEMS(RANK) := ITEM_ID_OF(PORTABLE_FILES.SHOW_FILE(LIST, RANK)) ;
                end loop ;
                return ITEMS ;
            end ;
        else
            return (1..0=> NULL_ITEM);
        end if ;
    end CHILDREN_OF ;
    -----
    function ROOTS return ITEM_LIST is
    begin
        return (1=>ITEM_ID_OF(TO_PATH(UN.IMAGE(ROOT)))) ;
    end ROOTS ;
    -----
    function PARENT_OF (I : ITEM)
        return ITEM
    is
        R : constant ITEM_LIST := ROOTS ;
        FILE : constant String := NAME_OF(I) ;
    begin
        return ITEM_ID_OF(TO_PATH(DIR_OF(FILE))) ;
    end PARENT_OF ;
    -----
    function SHOW_SELECTED_NAME
        return String
    is
        begin -- SHOW_SELECTED_NAME
            return NAME_OF(FILE_BROWSER.SHOW_SELECTED(BROWSER)) ;
        end SHOW_SELECTED_NAME ;
    ...
end UTEST_BROWSER_API ;
    -----

```

On remarquera les *UNCHECKED\_CONVERSION* pour passer du nom de fichier codé comme un *UN.NAME* au type élément tel qu'il est vue par le *BROWSER*. Le composant *BROWSER* aurait pu être moins contraignant, en choisissant par un exemple un modèle de type "private", mais cela aurait compliqué l'implantation.



## 9. Exemple détaillé

Dans ce chapitre, on donne un exemple complet, compilable et exécutable d'un programme qui utilise les services du composant *UI*. Ce chapitre doit permettre à une personne qui a déjà une bonne connaissance de Motif d'avoir une vue d'ensemble rapide et concrète du composant *UI*. Pour ces personnes, ce chapitre peut être lu en premier. Pour les personnes moins expérimentées, il constitue un exemple concis et exhaustif d'utilisation du composant *UI*.

Le programme présenté est un petit éditeur<sup>11</sup>. Il est important dans cet exemple de distinguer ce qui est imposé par *UI* de ce qui est un choix méthodologique de l'applicatif. Néanmoins, cet exemple a été développé dans le but de fournir un modèle conseillé d'utilisation de *UI*. Le choix le plus important est la séparation de l'IHM et de la "science" du programme. L'IHM doit être vue comme une couche de présentation ajoutée à un programme qui lui contient toute la "science". Il doit être possible de changer l'IHM sans avoir à intervenir dans la partie du programme qui contient la connaissance. Cette partie est disponible par l'intermédiaire d'une *API* (Application Programming Interface). L'IHM est décomposé en deux parties. La première se charge des créations des boîtes de dialogue et des fenêtres (paquetage *UTEST\_EDITOR\_IHM*). La deuxième, qui contient les callbacks de réaction aux actions de l'utilisateur, fait le lien entre l'*API* et l'IHM (paquetage *UTEST\_EDITOR\_CB*)

<insérer fichier g\_editor.eps>

Figure 9-1 Capture d'écran du programme EDITOR

L'application permet d'éditer un fichier et d'en avoir deux vues distinctes (menu Buffer2).

Le paquetage *UTEST\_EDITOR\_IHM* contient une fonction de création de la fenêtre principale de l'application, des identificateurs des menus, des opérations qui manipulent l'IHM et des fonctions de création de boîtes utilitaires secondaires.

---

<sup>11</sup>Le code est consultable en ligne : couche G35, fichier t\_editor.ad

```

-----
with TEXT_IO          ;
with SUPER_STANDARD  ; use SUPER_STANDARD ;
with USER_INTERFACE ;

package UTEST_EDITOR_IHM is

  package UI  renames USER_INTERFACE ;

  ----- boîte principale -----

  procedure MAP_MAIN_WINDOW (          -- Cree et affiche la fenetre
    PARENT      : UI.WIDGET ;          -- principale avec ce pere
    FILE_CB     : UI.CB_NAMED_ADDRESS;
    BUFFER2_CB  : UI.CB_NAMED_ADDRESS);

  --
  -- Menu pulldown 'File'
  --

  type FILE_CHOICE is new Integer range 1 .. 4 ;
  C_OPEN   : constant FILE_CHOICE := FILE_CHOICE'First ;
  C_SAVE   : constant FILE_CHOICE := C_OPEN   + 1 ;
  C_SAVEAS : constant FILE_CHOICE := C_SAVE   + 1 ;
  C_EXIT   : constant FILE_CHOICE := C_SAVEAS + 1 ;

  FILE_LABELS : array (FILE_CHOICE) of UN.NAME := (
    C_OPEN  => UN.DEFINE("Open"),
    C_SAVE  => UN.DEFINE("Save"),
    C_SAVEAS => UN.DEFINE("Save as..."),
    C_EXIT  => UN.DEFINE("Exit"));

  --
  -- Menu pulldown 'Buffer2'
  --

  type BUFFER2_CHOICE is new Integer range 1 .. 2 ;
  C_SHOW : constant BUFFER2_CHOICE := BUFFER2_CHOICE'First ;
  C_HIDE : constant BUFFER2_CHOICE := C_SHOW + 1 ;

  BUFFER2_LABELS : array (BUFFER2_CHOICE) of UN.NAME := (
    C_SHOW => UN.CREATE("Show"),
    C_HIDE => UN.CREATE("Hide"));

  ----- Texte -----

  procedure LOAD_TEXT (          -- Charge dans la fenetre
    DATA : in out TEXT_IO.FILE_TYPE) ; -- ce fichier

  procedure SAVE_TEXT (
    FILENAME : String) ;

  ----- Buffer2 -----

  procedure SHOW_BUFFER2 ;          -- Rend visible le deuxieme buffer
  procedure HIDE_BUFFER2 ;         -- Cache le deuxieme buffer

  -----

  procedure PUT_MSG (            -- Affiche un message dans la zone
    MSG : String) ;              -- des messages

  procedure EXIT_MAIN_LOOP ;

  ----- boîtes utilitaires -----

  function GET_FILE (            -- Ouvre une 'file selection box'
    TITLE : String ;            -- avec ce titre et
    MASK  : String)              -- ce masque et retourne
    return String ;              -- fichier selectionne

end UTEST_EDITOR_IHM ;
-----

```

Le paquetage *UTEST\_EDITOR\_CB* contient les callbacks. Les deux pulldown menus du programme *EDITOR*, *File* et *Buffer2*, ont chacun leur callback. La spécification de ce paquetage fournit des références typées sur ces deux callbacks.

```

with SUPER_STANDARD ; use SUPER_STANDARD ;
with USER_INTERFACE ; use USER_INTERFACE ;

-----
package UTEST_EDITOR_CB is

  procedure EDITOR_FILE_CB (W: WIDGET; C: C_LONG ; CD: ANY_CALLBACK_STRUCT);
  pragma EXPORT_PROCEDURE (EDITOR_FILE_CB);
  function EDITOR_FILE_CB_PROC is new CB_PROC (
    "EDITOR_FILE_CB", EDITOR_FILE_CB) ;

  procedure EDITOR_BUFFER2_CB (W: WIDGET; C: C_LONG ; CD: ANY_CALLBACK_STRUCT);
  pragma EXPORT_PROCEDURE (EDITOR_BUFFER2_CB);
  function EDITOR_BUFFER2_CB_PROC is new CB_PROC (
    "EDITOR_BUFFER2_CB", EDITOR_BUFFER2_CB) ;

end UTEST_EDITOR_CB ;
-----

```

Le programme principal, *TEST\_EDITOR*, se charge de l'initialisation, de l'affichage de la fenêtre principale et de la boucle principale d'événements.

```

with SUPER_STANDARD ; use SUPER_STANDARD ;

with XLIB_VOCABULARY ;
with UI_VOCABULARY ;
with USER_INTERFACE ;
with UI_TOOLS ;
with UI_BOXES ;

with UTEST_EDITOR_IHM;
with UTEST_EDITOR_CB ;

-----
procedure TEST_EDITOR is

  package XV renames XLIB_VOCABULARY ;
  package UI renames USER_INTERFACE ;
  package UIT renames UI_TOOLS ;

  package ED_IHM renames UTEST_EDITOR_IHM ;
  package ED_CB renames UTEST_EDITOR_CB ;

  DPY : XV.DISPLAY ;
  APP_SHELL : UI.WIDGET ;

begin

  --
  -- Initialisation et ouverture du display
  --
  DPY := UIT.INITIALIZE_AND_OPEN("editor") ;

  --
  -- Creation d'un widget shell racine de l'application
  --
  APP_SHELL := UI.CREATE_APPLICATION_SHELL(DPY) ;

  --
  -- Affichage de la fenetre principale
  --
  ED_IHM.MAP_MAIN_WINDOW (APP_SHELL,
    ED_CB.EDITOR_FILE_CB_PROC,
    ED_CB.EDITOR_BUFFER2_CB_PROC) ;

  --
  -- traitement des evenements
  --
  UI.APP_MAIN_LOOP ;

end TEST_EDITOR ;
-----

```

Le corps du paquetage *UTEST\_EDITOR\_IHM* contient l'aspect plus technique de la création de l'IHM.



```

with XLIB_VOCABULARY ;
with UI_VOCABULARY ; use UI_VOCABULARY ;
with UI_TOOLS ;
with UI_BOXES ;

-----
package body UTEST_EDITOR_IHM is

package XV renames XLIB_VOCABULARY ;
package UIV renames UI_VOCABULARY ;
package UIT renames UI_TOOLS ;
package UIB renames UI_BOXES;

----- boîte principale -----

TITLE_PATH : constant String := "editor" ;

MAIN_WINDOW,          -- l'indentation rend compte
MENU_BAR,             -- de la hiéararchie des widgets
FILE_MENU,
BUFFER2_MENU,
PANED,
WINDOW1,
TEXT1,
WINDOW2,
TEXT2,
MESSAGE : UI.WIDGET ;

-----
procedure MAP_MAIN_WINDOW (
PARENT      : UI.WIDGET ;
FILE_CB     : UI.CB_NAMED_ADDRESS ;
BUFFER2_CB  : UI.CB_NAMED_ADDRESS)
is
FILE : TEXT_IO.FILE_TYPE ;
use UI ;
begin -- MAP_MAIN_WINDOW

--
-- Creation de la fenetre principale
--
MAIN_WINDOW := UI.CREATE_WIDGET("", xmFormWidgetClass, PARENT) ;

--
-- Creation d'une barre de menu
--
MENU_BAR := UI.CREATE_WIDGET("", xmMenuBarWidgetClass, MAIN_WINDOW,
ARGS => UIT.LEFT_RIGHT_TOP_FORM (0)) ;

--
-- Creation des menus
--
FILE_MENU := UIT.CREATE_SIMPLE_PULLDOWN("File", MENU_BAR,
UN.NAME_LIST(FILE_LABELS),
FILE_CB) ;
BUFFER2_MENU := UIT.CREATE_SIMPLE_PULLDOWN("Buffer2", MENU_BAR,
UN.NAME_LIST(BUFFER2_LABELS),
BUFFER2_CB) ;

--
-- Creations des windows de l'editeur
--
PANED := UI.CREATE_WIDGET("", xmPanedWindowWidgetClass, MAIN_WINDOW,
ARGS => UIT.LEFT_RIGHT_BOTTOM_FORM(0) +
UIT.TOP_WIDGET(MENU_BAR, 0)) ;

UIB.CREATE_TEXT (WINDOW1, TEXT1, PARENT=> PANED, DATA=> FILE) ;

UIB.CREATE_TEXT (WINDOW2, TEXT2, PARENT=> PANED, DATA=> FILE) ;

MESSAGE := UIB.CREATE_SCROLLED_TEXT_OUTPUT("msg", PANED, 3, 80,
SIZE => Positive'Last);

--
-- Ressources
--
UI.SET_VALUE (TEXT2,
UI.ARG(XmNsource, Integer'(GET_VALUE(TEXT1, XmNsource)))) ;
UI.SET_VALUE (TEXT1,
ARG(XmNcursorPositionVisible, True));

```

```
    UI.SET_VALUE (TEXT2,
    ARG(XmNcursorPositionVisible, True));

    --
    -- Gestion et affichage
    --
    UI.MANAGE_ALL_DESCENDANTS(MAIN_WINDOW) ;
    UI.MAP_INTERFACE(MAIN_WINDOW) ;
    UI.UNMANAGE_CHILD (WINDOW2) ;

end MAP_MAIN_WINDOW ;

-----
procedure LOAD_TEXT (          -- Charge dans la fenetre
    DATA : in out TEXT_IO.FILE_TYPE) is -- ce fichier
begin -- LOAD_TEXT
    UI.ERASE (TEXT1) ;
    UIB.LOAD_TEXT (TEXT1, DATA, EDIT=> True) ;
end LOAD_TEXT ;

-----

procedure SAVE_TEXT (FILENAME : String) is
begin
    UIB.SAVE_TEXT (TEXT1, FILENAME) ;
end SAVE_TEXT ;

-----

procedure EXIT_MAIN_LOOP is
begin
    UI.EXIT_MAIN_LOOP ;
end ;

-----

procedure PUT_MSG (
    MSG : String)
is
begin -- PUT_MSG
    UIB.PUT_LINE(MESSAGE, MSG) ;
end PUT_MSG ;

-----

procedure SHOW_BUFFER2 is
begin -- SHOW_BUFFER2
    UI.MANAGE_CHILD(WINDOW2);
end SHOW_BUFFER2 ;

-----

procedure HIDE_BUFFER2 is
begin -- HIDE_BUFFER2
    UI.UNMANAGE_CHILD(WINDOW2);
end HIDE_BUFFER2 ;

----- boîtes utilitaires -----

-----
function GET_FILE (          -- Ouvre une 'file selection box'
    TITLE : String ;         -- avec ce titre et
    MASK : String)           -- ce masque
    return String is        -- fichier selectionne
begin -- GET_FILE
    return UIT.GET_FILE (MAIN_WINDOW, TITLE_PATH/TITLE, "", MASK) ;
end GET_FILE ;

end UTEST_EDITOR_IHM ;
=====
```

Enfin, le corps du paquetage *UTEST\_EDITOR\_CB* contient le corps des callbacks.

```

with TEXT_IO ;

with UTEST_EDITOR_IHM ;

-----
package body UTEST_EDITOR_CB is

package ED_IHM renames UTEST_EDITOR_IHM ;

FILENAME : UN.NAME ;

-----
procedure OPEN_FILE is
NAME : constant String := ED_IHM.GET_FILE ("Open File", "*.txt") ;
FILE : TEXT_IO.FILE_TYPE ;
begin
TEXT_IO.OPEN (FILE, TEXT_IO.IN_FILE, NAME) ;
ED_IHM.LOAD_TEXT(FILE) ;
TEXT_IO.CLOSE(FILE) ;
FILENAME := UN.CREATE(DROP_VERSION(NAME)) ;
exception
when others =>
ED_IHM.PUT_MSG ("Error opening file " & NAME) ;
end OPEN_FILE ;

-----
procedure SAVE_FILE (AS : Boolean) is
begin
ED_IHM.SAVE_TEXT (UN.IMAGE(FILENAME)) ;
end SAVE_FILE ;

-----
procedure EDITOR_FILE_CB (W: WIDGET; C: C_LONG ; CD: ANY_CALLBACK_STRUCT) is
begin -- EDITOR_FILE_CB
case ED_IHM.FILE_CHOICE(C) is
when ED_IHM.C_OPEN => OPEN_FILE ;
when ED_IHM.C_SAVE => SAVE_FILE (AS=> False) ;
when ED_IHM.C_SAVEAS => SAVE_FILE (AS=> True) ;
when ED_IHM.C_EXIT => ED_IHM.EXIT_MAIN_LOOP ;
end case;
end EDITOR_FILE_CB ;

-----
procedure EDITOR_BUFFER2_CB (W: WIDGET; C: C_LONG ; CD: ANY_CALLBACK_STRUCT) is
begin -- EDITOR_BUFFER2_CB
case ED_IHM.BUFFER2_CHOICE(C) is
when ED_IHM.C_SHOW => ED_IHM.SHOW_BUFFER2 ;
when ED_IHM.C_HIDE => ED_IHM.HIDE_BUFFER2 ;
end case;
end EDITOR_BUFFER2_CB ;

end UTEST_EDITOR_CB ;
-----

```

Il va de soi que dans un programme plus important, toute l'IHM ne serait pas regroupé dans un paquetage unique. Les principes de décomposition appliqués au reste du programme s'appliquent également à l'IHM.



## 10. Spécifications Ada des composants

<i>Paquetage</i> <b>UI_VOCABULARY</b>	<i>uiv_.ada</i>
<i>Paquetage</i> <b>USER_INTERFACE</b>	<i>ui_.ada</i>
<i>Paquetage</i> <b>UI_TOOLS</b>	<i>uit_.ada</i>
<i>Paquetage</i> <b>UI_BOXES</b>	<i>uib_.ada</i>
<i>Paquetage</i> <b>BROWSERS</b>	<i>ui_brow_.ada</i>
<i>Paquetage</i> <b>MOTIF_TERMINAL</b>	<i>ui_mt_.ada</i>
<i>Paquetage</i> <b>PRESENTATION_MANAGER_INTERFACE</b>	<i>ui_pmi_.ada</i>