CENTRE NATIONAL D'ÉTUDES SPATIALES

REFERENTIEL NORMATIF du CNES

RNC

Référence: **RNC-CNES-Q-HB-80-534**
Version 3
02 Juin 2008

MANUEL

ASSURANCE PRODUIT
REGLES POUR L'UTILISATION DU LANGAGE IDL (INTERACTIVE DATA LANGUAGE)

ACCORD du Bureau de Normalisation	**BN n° 34 du 25/06/07 – BN n°44 du 08/09/08**
APPROBATION Président du CDN Alain CUQUEL	

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : **REGLES POUR L'UTILISATION DU LANGAGE IDL (Interactive Data Language)**	
MOTS CLES : IDL, règle, langage, graphique	
NORME EQUIVALENTE : Néant	
OBSERVATIONS : Néant	
RESUME : Ce document décrit les règles applicables pour un logiciel utilisant le langage IDL.	
SITUATION DU DOCUMENT : Ce document fait partie de la collection des Manuels associés au Référentiel Normatif du CNES. Il est affilié au document « RNC-ECSS-Q-ST-80 Software product assurance ».	
NOMBRE DE PAGES : 74	**LANGUE**: Française
Progiciels utilisés / version : Word 2002	
SERVICE GESTIONNAIRE : Inspection Générale Direction de la Fonction Qualité (IGQ)	
AUTEUR(S) : **DATE : 02/06/2008**	
Repris par J-C DAMERY	

PAGES DES MODIFICATIONS

VERSION	DATE	PAGES MODIFIEES	OBSERVATIONS
Pr1	06/01/04	Toutes	Création du document par adaptation de la MP sur le langage Wave
Pr2	30/04/04	Toutes	Introduction de règles de bonne programmation. Introduction d'exemples (partiellement) dans les règles
Pr3	22/06/04	Toutes	Ajout d'exemples et contre exemple
1	09/09/04	Toutes	Ajout de règles pour prendre en compte toutes les fonctionnalités de la version 6.1
2	10/12/2006	Toutes	Mise en commun de règles et mise en forme, avec le support de T. Leydier (Virtualité Réelle). Modification du titre. Cf. FEB 48/2006 acceptée au BN n° 22 du 06/03/06. Document accepté au BN n° 34 du 25/06/07 pour introduction dans le RNC.
3	02/06/2008	Toutes	Changement de nomenclature suite à la phase de benchmarking ECSS (ancienne référence RNC-CNES-Q-80-534).

TABLE DES MATIERES

1. INTRODUCTION

Le document « REGLES POUR L'UTILISATION DU LANGAGE IDL (Interactive Data Language) » est rattaché à la norme RNC-ECSS-Q-ST-80 « Software Product Assurance ». Il décrit les règles applicables pour un logiciel utilisant le langage IDL.

2. OBJET

Le document présent décrit les règles essentielles pour l'utilisation du langage de traitement et visualisation graphique de données **IDL** et donne des recommandations d'utilisation sur quelques points particuliers (liens externes, iTools…).

Cette édition du document a été établie à partir de la version **6.1** de IDL. Il est à noter que certaines règles pourront être amenées à évoluer lors des évolutions apportées par de nouvelles versions de IDL.

Ce document n'est pas un manuel de référence du langage IDL. Il nécessite, pour être utilisé, la connaissance d'IDL.

3. DOMAINE D'APPLICATION

Ce document est applicable au développement et à la maintenance de tous les systèmes informatiques pour leurs parties réalisées en langage IDL.

Il est complété par le document « Règles communes pour l'utilisation des langages de programmation » (DA1).

Pour utiliser les règles IDL sur un projet, il faudra suivre la procédure suivante :

- Sélectionner dans les règles communes et les règles IDL, les règles applicables au projet en fonction des critères de tailorisation ; cette sélection s'effectuera avec l'outil de tailorisation.
- Adapter certaines règles au projet.

Le document est ainsi destiné à plusieurs types de lecteurs :

- le chef de projet qui doit spécifier correctement l'application IDL à développer,
- le chef de projet et/ou l'ingénieur qualité qui doit sélectionner les règles et éventuellement adapter et compléter les règles en fonction du contexte de réalisation,
- les personnes chargées de la réalisation du projet : elles doivent appliquer les règles retenues.

4. DOCUMENTS

4.1. DOCUMENTS DE REFERENCE

DR	Identification	Titre
(DR1)		USING IDL (version 6.1)
(DR2)		BUILDING IDL APPLICATION (version 6.1)
(DR3)	NF X50-130 (NF EN ISO 9000)	Système de management de la qualité – Principes essentiels et vocabulaire
(DR4)	RNC-ECSS-Q-ST-80	Software Product Assurance

4.2. DOCUMENTS APPLICABLES

DA	Identification	Titre
(DA1)	RNC-CNES-Q-HB-80-501	Règles communes pour l'utilisation des langages de programmation.

5. TERMINOLOGIE

5.1. GLOSSAIRE

Terme	Définition
Callback	Un Callback est une routine particulière qui permet de prendre en compte les événements utilisateurs (clavier, souris, ...).
Constante	IDL ne définit pas de type spécifique pour les constantes. Dans la suite de ce document, nous appelons constantes des variables avec un préfixe particulier pour lesquelles toute modification de type ou de valeur est interdite. Remarque : la routine DEFSYSV permet de définir de nouvelles variables système accessibles en lecture seule. Ce type de variable peut éventuellement être utilisé comme une constante.
IDLDE	« IDL Development Environment ». Environnement de développement de IDL incluant la gestion de projet et un système de débuggage complet.
Fichier batch	Un fichier batch contient un ensemble de commandes IDL. Toutes les commandes d'un fichier batch sont interprétées comme si elles étaient entrées sur la ligne de commande IDL.
Mode développement	Le mode développement permet au programmeur de développer et d'exécuter des routines (en utilisant les fonctionnalités de compilation/exécution de IDLDE ou avec la commande ".RUN"). Il permet aussi la création de fichiers exécutables.
Mode run-time	Ce mode désigne l'exécution de la version compilée d'un programme IDL.
Module	Une application est divisée en plusieurs modules. Chaque module est un ensemble de routines dédiées à une tâche précise. L'application est décomposée en modules lors de la conception.
Panneau	Fenêtre principale d'une IHM composée de widgets, et constituant un ensemble cohérent présenté à l'utilisateur.
Structure	Une variable organisée en structure est une variable contenant un ou plusieurs champs de types différents ("RECORD" en Fortran ou "struct" en C).
Variable simple	Ce terme "variable simple" regroupe toutes les variables de type simple (entier, réel, chaîne de caractères, entier long, réel double précision, complexe, octet) par opposition aux tableaux, aux structures et aux variables « HEAP » (pointeur, objet).
Widget	Un widget est un élément graphique qui constitue l'entité élémentaire de composition d'un panneau.

5.2. ABREVIATIONS

5.2.1. Codification des règles

Voir DA1

5.2.2. Autres abréviations ou acronymes

Voir DA1

6. ADEQUATION EVENTUELLE AUX NORMES ET STANDARDS

Sans objet

7. LES REGLES

7.1. CONCEPTION / ORGANISATION DU CODE

Org.Projet	Les codes source d'une application doivent toujours être regroupés sous forme d'un projet.
M=2;F=0;P=40;V=1	
Quelconque	

Description

Une application IDL peut être composée de quatre types d'unité de code :

- ? Un programme principal.
- ? Des routines.
- ? Des modules.
- ? Des fichiers batch.

1. Programme principal	Un programme principal IDL est une série d'instructions se terminant par l'instruction END. Si l'application contient un programme principal, elle pourra être lancée depuis la ligne de commande par l'instruction .RUN, ou en utilisant les fonctionnalités de compilation/exécution d'IDLDE. Si l'application ne contient pas de programme principal, elle contient nécessairement une routine principale (une procédure ou une fonction).
2. Routines	Chaque module de l'application est composé d'une ou plusieurs routines. Une routine est une procédure ou une fonction écrite en langage IDL. Ces routines peuvent être exécutées en mode développement (sous le prompt "idl") après les avoir interprétés à l'aide de la commande ".RUN", ou en utilisant les fonctionnalités de compilation/exécution d'IDLDE.
3. Modules	Une application est décomposée fonctionnellement en modules : cette décomposition est issue de la conception. Un module est un regroupement logique de routines mais n'est pas une notion IDL.
4. Fichier batch	Un fichier batch contient un ensemble de commandes IDL. Toutes les commandes d'un fichier batch sont interprétées comme si elles étaient entrées sur la ligne de commande IDL.

IDL permet de regrouper tous les fichiers source d'une application sous forme d'un projet, grâce à l'option de IDLDE : File->New->Project.

Justification

Le regroupement des codes source d'une application sous forme d'un projet présente de nombreux avantages :

La facilité de débuggage de l'application.
La facilité d'exportation sous forme de code source.
La facilité d'exportation sous forme de fichier exécutable.
La possibilité de sauvegarde de l'environnement de travail.

7.2. PRESENTATION DU CODE

7.3. IDENTIFICATEURS

Id.MotsClés	On ne doit pas abréger le nom des mots-clé.
M=1;F=1;P=48;V=1	
Quelconque	

Description

IDL autorise l'abréviation des mots-clé à la plus courte chaîne de caractères non-ambiguë. Il est recommandé de ne pas abuser de cette propriété.

Justification

Il est préférable d'utiliser le nom complet d'un mot-clé plutôt qu'une abréviation, car de futurs mots-clé pourraient rendre abréviation courante ambiguë.

Id.Natif	Les noms des variables doivent être différents des noms des routines natives IDL.
M=3;F=1;P=76;V=2	
Quelconque	

Description

Sans objet.

Justification

L'application de cette règle permet d'éviter un nombre important d'erreurs en phase d'intégration et de maintenance de l'application.

Id.Widget	On doit utiliser une convention de nommage pour les widgets de l'IHM.
M=1;F=0;P=88;V=1	
Quelconque	

Description

Il est préférable de nommer les widgets d'une interface homme-machine en respectant une convention du type : wNameType. Le rôle de la partie « Name » est purement descriptif ; le rôle de la partie « Type » est d'indiquer le type du widget tel que spécifié par la routine de création.

Justification

Le respect d'une telle convention de nommage améliore grandement la lisibilité, la réutilisabilité et la maintenabilité du code de l'interface homme-machine.

Exemple

```
wMainBase              = WIDGET_BASE( ... )
wExtensionBgroup       = CW_BGROUP( ... )
wExitButton                  = WIDGET_BUTTON( ... )
wCurveDraw             = WIDGET_DRAW( ... )
wNamesDroplist               = WIDGET_DROPLIST( ... )
wInfoLabel             = WIDGET_LABEL( ... )
wLengthSlider                = WIDGET_SLIDER( ... )
wHeightText                  = WIDGET_TEXT( ... )
```

7.4. DONNEES

Don.InterfaceMin	Chaque classe doit posséder au minimum 3 méthodes : définition de la classe, constructeur, destructeur.
M=2;F=1;P=23;V=1	
Quelconque	

Description

Tout objet créé avec IDL est *persistant*, ce qui signifie qu'il existe en mémoire jusqu'à ce qu'il soit détruit. La « vie » d'un objet peut être décomposée en quatre étapes : création, initialisation, utilisation, et destruction. Les étapes de création, initialisation et destruction d'un objet sont appelées les méthodes « cycle de vie » (« lifecycle ») d'un objet : elles permettent de contrôler ce qui se produit lorsqu'un objet est créé ou détruit.

? **DEFINITION D'UNE CLASSE**

Une classe doit OBLIGATOIREMENT posséder une méthode nommée CLASSE__DEFINE, où « CLASSE » est le nom de la nouvelle classe créée. Cette méthode prend la forme d'une procédure et permet de définir les données membres de la classe, ainsi que la ou les classe(s) héritées par le mécanisme INHERITS.

? **INITIALISATION**

Une classe doit OBLIGATOIREMENT posséder une méthode nommée « CLASSE ::INIT ». Le rôle de cette méthode est d'initialiser l'ensemble des données membre de la classe. Cette méthode prend la forme d'une fonction qui renvoie la valeur 1 si l'initialisation s'est bien passée, et toute autre valeur dans le cas contraire. Cette méthode est automatiquement appelée par la fonction de création d'objet OBJ_NEW.

? **DESTRUCTION**

Une classe doit OBLIGATOIREMENT posséder une méthode nommée CLASSE ::CLEANUP. Le rôle de cette méthode est de détruire l'objet ainsi que toute la mémoire allouée à cet objet (libération des pointeurs, destruction des données membre de type objet, destruction des widgets pour le cas des IHM encapsulées dans des objets…).

Justification

Sans objet.

Don.Héritage	On doit employer la technique d'héritage dès que possible.
M=2;F=0;P=41;V=0	
Quelconque	

Description

La technique objet d'héritage permet de créer rapidement des objets héritant des données membres et des méthodes d'un objet donné, et est implémentée dans IDL par le mécanisme INHERITS.
Remarque : les méthodes INIT et CLEANUP de la classe IMAGE doivent faire appel aux méthodes INIT et CLEANUP de la classe parente IDLGRIMAGE.

Justification

La technique objet d'héritage peut s'avérer particulièrement utile pour la création de nouveaux objets graphiques. Si l'on considère par exemple l'objet IDLGRIMAGE, cet objet constitue l'un des atomes graphiques d'IDL, c'est à dire qu'il peut être inséré en standard dans une hiérarchie graphique et visualisé vers un objet destination. De plus, cet objet possède déjà un nombre conséquent de méthodes permettant d'accéder en lecture ou en écriture à l'ensemble de ses propriétés.
Par contre, il n'est pas possible de passer directement en paramètre de la méthode INIT de la classe IDLGRIMAGE un nom de fichier, afin de charger automatiquement une image. Pour se faire, la technique consiste simplement à créer une nouvelle classe IMAGE héritant de la classe IDLGRIMAGE, comme illustré dans l'exemple ci-dessous.

Exemple

Illustration de la technique objet d'héritage.

```
pro image::cleanup
    ; Appel de la méthode cleanup de la classe parente :
    self->IDLGRIMAGE::CLEANUP
end

function image::init, filename, _extra = e
    ; Appel de la méthode INIT de la classe parente :
    out = self->IDLGRIMAGE::INIT(_EXTRA = e)
    IF out THEN BEGIN
        IF FILE_TEST(filename) THEN BEGIN
            ; Lecture image :
            data = READ_IMAGE(filename)

            ; Lecture OK ?
            IF data(0) NE -1 THEN BEGIN

                self->IDLGRIMAGE::SETPROPERTY, DATA = data

            END ELSE $
                out = 0

        END ELSE $
            out = 0

    END
```

```
    RETURN, out

END

pro image__define

    define = {IMAGE, INHERITS IDLGRIMAGE}

end
```

Don.TableauCreation	Un tableau doit être créé en utilisant le mot-clé `NOZERO` dans les fonctions de création de tableaux.
M=0;F=1;P=90;V=1	
Quelconque	

Description

L'utilisation du mot-clé NOZERO dans une fonction de création de tableaux (BYTARR, FLTARR, ….) permet de créer un nouveau tableau sans initialiser ses éléments à zéro.

Justification

Le comportement par défaut des différentes fonctions de création de tableaux (BYTARR, FLTARR, …) est d'allouer de la mémoire pour le tableau, et de mettre à zéro l'ensemble des éléments de ce tableau. L'utilisation du mot-clé NOZERO réduit l'opération de création d'un nouveau tableau à une simple opération d'allocation mémoire, et constitue un gain de temps substantiel.
Cela est particulièrement intéressant quand le tableau est utilisé en modification juste après sa création.

Don.Bornage	On doit utiliser les signes "<" et ">" pour borner un tableau de préférence à la boucle "`for`" suivie d'un test et d'une affectation.
M=0;F=0;P=113;V=1	
Quelconque	

Description

Cette règle permet d'affecter automatiquement une valeur maximum ou minimum à un tableau sans le parcourir à l'aide d'une boucle.

Justification

Cette règle permet d'améliorer les performances.

Exemple

```
Tab = [1,2,3,4,5]
Tab=Tab < 3 ; modifie le tableau qui vaut désormais [1,2,3,3,3]
```

Don.StuctCacher	On doit utiliser la procédure `STRUCT_HIDE` pour dissimuler des structures.
M=2;F=0;P=43;V=1	
Quelconque	

Description

La procédure STRUCT_HIDE est utilisée pour « dissimuler » certaines structures et certains objets. Ces éléments « dissimulés » ne sont alors plus affichés par la procédure HELP (HELP, /STRUCTURES ou HELP, /OBJECTS), à moins d'utiliser le mot-clé FULL.

Justification

La procédure HELP (HELP, /STRUCTURES ou HELP, /OBJECTS) affiche des informations sur toutes les structures et classes d'objets connues. Même si ce comportement est le plus souvent souhaité, les développeurs d'applications volumineuses ou de librairies souhaitent parfois préserver une partie privée dans leur application. Lors de la création d'objets, le développeur pourra par exemple placer la procédure STRUCT_HIDE dans la procédure __DEFINE qui définit la structure.

Exemple

Cet exemple montre comment une structure peut être dissimulée si un développeur souhaite rendre cette structure « invisible » par la routine HELP, /STRUCTURE, /BRIEF :

```
; Définition d'une variable contenant la structure nommée :
tmp = { bullwinkle, moose:1, squirrel:0 }

; IDL renvoie BULLWINKLE ainsi que d'autres variables système :
HELP, /STRUCTURE, /BRIEF

; Dissimulation de la structure en utilisant la procédure STRUCT_HIDE :
STRUCT_HIDE, tmp

; Cette fois, IDL renvoie les variables système mais pas la structure
BLLWINKLE :
 HELP, /STRUCTURE, /BRIEF

; Le seul moyen d'afficher le contenu de la structure tmp est de
; connaître son nom, puisque cette structure a été dissimulée par la
; routine STRUCT_HIDE :
HELP, tmp
```

Don.TailleIndét	On doit utiliser des pointeurs pour stocker des données de taille indéterminée.
M=0;F=0;P=115;V=1	
Quelconque	

Description

On utilise l'instruction PTR_NEW avec le mot-clé NO_COPY pour stocker des données de taille indéterminée.

Justification

En cours de programmation d'une application, on ne connaît pas toujours les dimensions des tableaux. Dans d'autres cas, la taille ou le type d'un tableau peut également être amenée à changer. On peut imaginer par exemple une application dans laquelle un utilisateur est amené à charger des images de dimensions ou de type différent.
Dans ces cas particuliers, la solution consiste à stocker le tableau contenant les données dans un pointeur appartenant à la structure d'état de l'application. On peut ainsi stocker des données de taille ou de type différent sans avoir à modifier la structure d'état de l'application.

Exemple

```
; Définition de la variable d'état de l'application.
; Initialement, le champ pArray est égal au pointeur null :
state = {buttonDown: 0B, color: 0, pArray:PTR_NEW()}

; Exemple d'initialisation du champ pArray de la structure state :
IF ptr_PTR_VALID(state.pArray) THEN $
     ; Le pointeur est valide :
     *state.pArray = image else $
     ; Le pointeur n'a pas encore été initialisé :
state.pArray = PTR_NEW(image, /NO_COPY)
```

Don.EviterDuplication	On doit éviter la duplication de la mémoire lors de l'allocation d'un nouveau pointeur.
M=1;F=1;P=50;V=1	
Quelconque	

Description

Le mot-clé NO_COPY peut être utilisé avec la fonction PTR_NEW afin d'éviter la duplication de la mémoire.

Justification

```
pointer = PTR_NEW(value)
```

Après exécution de cette instruction, le contenu du pointeur « pointer » est égal à la variable value, et la variable value reste définie. Une copie de cette variable a donc été effectuée. L'utilisation du mot-clé NO_COPY permet de transférer la variable « value » dans le pointeur « pointer », et de rendre la variable « value » indéfinie. Cette instruction est équivalente à l'instruction suivante :

```
Pointer = PTR_NEW(TEMPORARY(value))
```

Don.Typage	IDL ne permettant pas de déclaration de type, il est nécessaire de signaler en début de routine la liste des variables utilisées, avec leur mode d'utilisation.
M=3;F=1;P=78;V=1	
Quelconque	

Description

Dans le langage IDL, les variables peuvent être de trois natures (scalaires, tableaux ou structures), et de huit types : octets, entiers, entiers longs, réels, réels double précision, complexes et chaînes. Il n'existe pas de déclaration de type explicite pour les variables simples. Le typage est **implicite** à chaque affectation. Afin de pallier ce manque de typage, deux solutions peuvent être envisagées :

Déclaration virtuelle dans les commentaires : cette méthode facilite la mise à jour lorsque le programmeur décide de modifier le type d'une variable,

Nommage des variables en incluant le type : cette méthode facilite la lisibilité du programme mais semble difficile à respecter en phase de maintenance ou d'évolution du programme (en particulier pour des applications importantes, certaines variables risquent d'être oubliées).

Pour les types simples (entier, chaîne de caractères), il est possible d'adopter une troisième possibilité, en donnant une affectation à la variable.

Justification

Cette règle permet de "typer" la variable et de ne pas l'utiliser sans avoir préalablement défini son type.

Exemple

```
; DEBUT D'EN-TETE
;============================================================
; NOM                 : DIVISION
; VERSION       :
; AUTEUR        :
; DATE DE CREATION    :
; DESCRIPTION         : Réalise la division de deux valeurs
;               : retourne le résultat de type flottant
;============================================================
; LISTE DES ROUTINES UTILISEES :
; Sans objet
;============================================================
; MODE D'APPEL :
;    flottant res =
;    DIVISION (
;    flottant          val1 ;          (IN) dividende
;    flottant         ,val2 );    (IN) diviseur
;============================================================
; COMMONS UTILISES
; Sans objet
; LISTE DES VARIABLES LOCALES
; flottant resultat
; ALGORITHME
; Sans objet
;============================================================
; FIN D'EN-TETE
;============================================================
; CORPS DE LA ROUTINE
;============================================================

FUNCTION DIVISION val1,val2

; DECLARATION (INITIALISATION) DES VARIABLES LOCALES
; resultat = 0.0

; TESTS DES PRE-CONDITIONS
```

```
IF (val2 EQ 0) THEN BEGIN
resultat = CST_INFINI
GOTO LBL_FIN
ENDIF

; PARTIE TRAITEMENT
resultat = val1 / val2
GOTO LBL_FIN

LBL_FIN :

RETURN,resultat

END
```

Don.Modification	On doit ne changer ni la taille ni le type d'une variable.
M=3;F=2;P=73;V=1	
Quelconque	

Description

La seule dérogation est accordée pour la libération mémoire des tableaux qui doit toujours être accompagnée d'un commentaire.

Justification

Le langage IDL permet au programmeur de modifier le type d'une variable en l'affectant avec une valeur d'un autre type. Le changement de type d'une variable en cours de traitement est source d'erreurs et nuit fortement à la lisibilité et à la maintenabilité du programme.

Exemple

Exemple 1

```
A=2.0
B=3.0
RES = A/B est un flottant qui vaut 0.66667.

A=FIX(A)
B=FIX(B)
RES = A/B est un entier qui vaut 0 !
```

Exemple 2

```
; Construction d'un tableau de valeurs initialisées 1/nb d'éléments
; Exemple Correct
N=10
Tab = (FLTARR(N)+1.)/N est un tableau dont chaque élément vaut 1./N

; Exemple Incorrect
Tab=FLTARR(10)
Tab=1./N est une variable flottante de valeur 1./N
```

Don.VarSystème	On doit utiliser une variable système IDL pour modifier une ressource.
M=0;F=1;P=91;V=1	
Quelconque	

Description

Les variables système d'IDL représentent les paramètres par défaut d'une session IDL, et servent notamment à définir les paramètres graphiques par défaut.

Justification

La modification d'une variable système reste persistante le temps d'une session IDL, et se répercute à toutes les routines utilisant la ressource (graphique ou non) représentée par cette variable système.

Don.CommonInit	Lorsqu'un `COMMON` est utilisé pour passer des variables d'une routine à une autre, il doit toujours être initialisé par l'appelant.
M=3;F=3;P=5;V=1	
Quelconque	

Description

L'appelant doit initialiser les variables du COMMON qui sont utilisées par l'appelé. Dans le cas d'appel en cascade (P1 appelle P2 qui appelle P3), le COMMON sera initialisé par la procédure de niveau supérieur comme dans le cas de paramètres positionnels.

Justification

Cette règle permet de limiter le nombre de paramètres positionnels d'une procédure. Cette utilisation ne doit être qu'exceptionnelle.

Don.CommonReprod	Les `COMMONs` doivent être reproduits identiquement (nom et nombre de variables identiques) dans toutes les routines qui les référencent.
M=3;F=3;P=8;V=1	
Quelconque	

Description

Le moyen de garantir cette règle est d'utiliser les fichiers batch avec la commande "@". Elle permet d'assurer l'intégrité des COMMONs.

Justification

IDL autorise la redéfinition d'un COMMON avec un nombre différent de variables (idem Fortran77) et permet aussi le changement des noms qui le composent.

Exemple

Exemple 1 : Pas d'utilisation de fichiers batch

```
; Définition des commons dans un fichier
COMMON Commons_locaux, CL_var1, CL_var2

; DEBUT D'EN-TETE
;=============================================================
; NOM                 :
; VERSION       :
; AUTEUR        :
; DATE DE CREATION    :
; DESCRIPTION         :
;=============================================================
; LISTE DES ROUTINES UTILISEES :
; Sans objet
;=============================================================
; MODE D'APPEL :
;=============================================================
; COMMONS UTILISES
; commons_locaux
; LISTE DES VARIABLES LOCALES
```

```
; Sans objet
; ALGORITHME
; Sans objet
;=============================================================
; FIN D'EN-TETE
;=============================================================
; CORPS DE LA ROUTINE
;=============================================================
FUNCTION Ma_fonction val1,val2

; Utilisation des commons dans la fonction
COMMON Commons_locaux, CL_var1, CL_var2

Resultat = val1 * CL_var1

RETURN,resultat

END
```

Exemple 2 : Utilisation de fichiers batch

```
; Définition des commons dans un fichier nommé Def_Commons
COMMON Commons_locaux, CL_var1, CL_var2
; DEBUT D'EN-TETE
;=============================================================
; NOM                 :
; VERSION       :
; AUTEUR        :
; DATE DE CREATION    :
; DESCRIPTION         :
;=============================================================
; LISTE DES ROUTINES UTILISEES :
; Sans objet
;=============================================================
; MODE D'APPEL :
;=============================================================
; COMMONS UTILISES
; commons_locaux
; LISTE DES VARIABLES LOCALES
; Sans objet
; ALGORITHME
; Sans objet
;=============================================================
; FIN D'EN-TETE
;=============================================================
; CORPS DE LA ROUTINE
;=============================================================

FUNCTION Ma_fonction val1,val2

; Utilisation des commons dans la fonction
@Def_Commons

Resultat = val1 * CL_var1
```

```
RETURN,resultat

END
```

7.5. TRAITEMENTS

Tr.Indexation	L'indexation des éléments d'un tableau doit s'effectuer en utilisant les crochets [].
M=2;F=0;P=45;V=2	
Quelconque	

Description

Sans objet.

Justification

Cette règle permet d'obtenir une meilleure lisibilité du code, dans la mesure ou l'emploi des parenthèses est réservé à l'appel des routines de type fonction avec paramètres.

Tr.AccesConditionnel	On doit utiliser l'instruction `WHERE` pour l'accès conditionnel aux éléments d'un tableau.
M=0;F=1;P=93;V=2	
Quelconque	

Description

Cette règle permet d'accéder à certains éléments d'un tableau en fonction d'une certaine condition, sans avoir à utiliser de boucles et d'instructions IF.

Justification

Cette règle améliore les performances car l'instruction WHERE fonctionne avec tous les tableaux quelque soit le nombre de dimensions, et évite donc l'imbrication de boucles FOR. D'autre part, cette instruction renvoie un tableau d'indices uniques, permettant d'accéder rapidement aux éléments de ce même tableau vérifiant la condition.

Exemple

```
vec = WHERE((array GT 10) AND (array LT 125), count)

if count GT 0 then $
     print, array(vec)
```

Tr.AccesRapide	On doit utiliser un indice unique pour accéder rapidement aux éléments d'un tableau.
M=0;F=1;P=95;V=1	
Quelconque	

Description

Tous les éléments d'un tableau sont rangés en mémoire de façon contiguë. On peut accéder aux éléments d'un tableau en utilisant un indice pour chaque dimension, ou en utilisant un indice unique. L'utilisation d'un indice unique s'avère être 20% plus rapide que l'utilisation d'un indice par dimension.

Justification

Cette règle évite l'utilisation de boucles imbriquées.

Exemple

```
array = INTARR(10, 20, 30)
FOR I = 0L, N_ELEMENTS(array)-1 DO BEGIN

     ; Traitement spécifique :
     array(i) = array(i)*2


END
```

Contre Exemple

```
array = INTARR(10, 20, 30)
FOR K = 0, 29 DO BEGIN
     FOR J = 0, 19 DO BEGIN
          FOR I = 0, 9 DO BEGIN
          …
          ENDFOR
     ENDFOR
ENDFOR
```

Tr.Astérisque	Pour accéder à tous les éléments d'un tableau, on doit éviter l'utilisation de l'astérisque * à gauche du signe =.
M=0;F=1;P=96;V=2	
Quelconque	

Description

L'usage de l'astérisque à gauche du signe « = » doit se limiter aux tableaux de petite taille, c'est à dire aux tableaux dont la taille mémoire est très nettement inférieure aux capacités de la mémoire vive. Dans le cas des tableaux volumineux, il est préférable d'utiliser une boucle.
Se référer à Tr.TableauInit.

Justification

L'utilisation d'un astérisque à gauche du signe « = » implique la création par IDL d'un tableau d'indices temporaire. Plus le tableau est volumineux, plus le temps de création du tableau d'indices temporaires est important. L'utilisation d'une boucle peut donc s'avérer beaucoup plus rapide sur des tableaux volumineux.

Exemple

Si le tableau « array » contient m éléments, alors :

```
array(*) = 100
est équivalent à :
array([0, 1, 2, …, m-1]) = 100
```

Tr.OperComposés	Sur les tableaux, on doit utiliser les opérateurs composés.
M=1;F=1;P=51;V=2	
Quelconque	

Description

La liste des opérateurs composés est donnée dans le tableau ci-dessous :

##=	#=	*=	+=	-=
/=	<=	>=	AND=	EQ=
GE=	GT=	LE=	LT=	MOD=
NE=	OR=	XOR=	^=	

Ces opérateurs associent l'opération d'affectation avec d'autres opérations particulières. On peut également y ajouter les opérateurs ++ pour l'incrémentation et -- pour la décrémentation.

Justification

Une instruction qui utilise un opérateur composé utilise la mémoire de façon beaucoup plus efficace, car elle effectue l'opération sur la variable cible elle-même. Par opposition, une instruction qui utilise des opérateurs simples effectue une copie de la variable, effectue l'opération sur la copie, et réaffecte le résultat à la variable originale, ce qui implique une utilisation temporaire de mémoire supplémentaire.

Exemple

L'expression :

```
A op= expression
```

Est identique à l'instruction IDL :

```
A = TEMPORARY(A) op (expression)
```

Concrètement :

L'expression :

```
A GE= 3
```

Est identique à l'instruction IDL :

```
A = TEMPORARY(A) GE 3
```

L'expression :

```
B XOR= 200b
```

Est identique à l'instruction IDL :

```
B = TEMPORARY(B) XOR 200b
```

Tr.MatriceMultiple	On doit utiliser les opérateurs # et ## pour multiplier les opérations de multiplication sur les matrices.
M=1;F=1;P=53;V=1	
Quelconque	

Description

Utiliser l'opérateur # pour multiplier les colonnes d'un premier tableau par les lignes d'un second tableau.

Utiliser l'opérateur ## pour multiplier les lignes d'un premier tableau par les colonnes d'un second tableau.

Justification

Il est beaucoup plus rapide d'utiliser l'opérateur ## que d'effectuer la même opération par l'intermédiaire d'une boucle.

Tr.MATRIX_MULTIPLY	On doit utiliser la fonction `MATRIX_MULTIPLY` de façon adéquate.
M=1;F=1;P=55;V=1	
Quelconque	

Description

L'utilisation de la fonction MATRIX_MULTIPLY est équivalente à l'utilisation de l'opérateur #, ou à la combinaison de l'opérateur # avec une opération de transposition, comme illustré par le tableau d'équivalences ci-dessous :

# Operator	**Function**
A # B	MATRIX_MULTIPLY(*A*, *B*)
transpose(A) # B	MATRIX_MULTIPLY(*A*, *B*, /ATRANSPOSE)
A # transpose(B)	MATRIX_MULTIPLY(*A*, *B*, /BTRANSPOSE)
transpose(A) # transpose(B)	MATRIX_MULTIPLY(*A*, *B*, /ATRANSPOSE, /BTRANSPOSE)

Justification

L'utilisation de la fonction MATRIX_MULTIPLY est plus rapide et plus efficace.

Tr.TableauMultiple	On doit éviter la multiplication entre tableaux de types différents.
M=1;F=1;P=56;V=1	
Quelconque	

Description

Sans objet.

Justification

Lors de l'opération de multiplication entre des tableaux de type différent, IDL effectue au préalable une conversion du(des) tableau(x) de précision la plus faible dans le type du tableau possédant la précision la plus haute. Le résultat de cette(ces) conversion(s) est alors stocké dans une(des) variable(s) temporaire(s).La multiplication est alors effectuée avant la destruction des différentes variables temporaires produites. Ceci peut s'avérer pénalisant pour les applications avec temps d'exécution critique.

Tr.TableauDupplication	On doit éviter la duplication inutile de la mémoire en utilisant la fonction `TEMPORARY`.
M=1;F=1;P=58;V=1	
Quelconque	

Description

La fonction TEMPORARY permet d'éviter la duplication superflue de la mémoire lorsqu'une même variable apparaît de part et d'autre du signe « = ».

Justification

La fonction TEMPORARY permet d'optimiser la mémoire utilisée par une application.

Exemple

L'expression :

```
arr = 2 * arr
```

créée une variable temporaire de même dimension que le tableau arr pour stocker le résultat.

L'expression :

```
arr  = 2 * TEMPORARY(arr)
```

signifie que IDL peut utiliser l'espace mémoire de la variable arr pour stocker les variables temporaires résultantes du calcul, au lieu d'allouer de la mémoire pour un nouveau tableau.

Il est à noter que la fonction TEMPORARY rend son paramètre d'entrée (le tableau arr) indéfini. Encore une fois, utiliser moins de mémoire peut avoir des effets très sensibles sur la vitesse d'exécution d'une application, dans la mesure où cela évite la pagination.

Tr.TableauInit	On doit utiliser la fonction `REPLICATE_INPLACE` pour initialiser rapidement un tableau à une valeur donnée.
M=0;F=1;P=98;V=1	
Quelconque	

Description

L'utilisation de la fonction REPLICATE_INPLACE permet d'initialiser un tableau ou un sous-tableau à une valeur donnée.

Justification

L'utilisation de la fonction REPLICATE_INPLACE est beaucoup plus rapide que l'utilisation de l'astérisque à gauche du signe =.

Exemple

Exemple 1

```
A = FLTARR( 40, 90, 10)
; Initialisation de la variable A avec la valeur 4.5. (i.e., A[*]= 4.5 ):
REPLICATE_INPLACE, A, 4.5
```

Exemple 2

```
;Initialisation d'un sous-tableau (i.e., A[*,4,0]= 20. ):
REPLICATE_INPLACE, A, 20, 1, [0,4,0]
```

Exemple 3

```
; Initialisation d'un groupe de sous-tableaux.
```

```
    ; (i.e., A[ 0, [0, 5,89], * ] = -8 ):
    REPLICATE_INPLACE, A, -8, 3, [0,0,0], 2, [0,5,89]
```

Exemple 4

```
    ; Initialisation d'une « tranche » 2D du tableau A
    ;(i.e., A[9,*, *] = 0.):
    REPLICATE_INPLACE, A, 0., 3, [9,0,0] , 2, LINDGEN(90)
```

Tr.TableauConcaténation	On doit éviter la technique de concaténation pour augmenter la taille d'un tableau.
M=0;F=2;P=46;V=2	
Quelconque	

Description

L'expression :
array = [array, sAnotherValue]
peut s'avérer potentiellement dangereuse pour des tableaux volumineux.

Justification

L'utilisation de la technique de concaténation de tableaux pour augmenter la taille d'un tableau peut provoquer des problèmes de fragmentation de la mémoire. Si la dimension finale du tableau est connue par avance, il est préférable d'utiliser la fonction REPLICATE (cf. Exemple 1), qui est par ailleurs plus rapide. Si la dimension finale du tableau n'est pas connue, une solution consiste à allouer une dimension prédéfinie au tableau, puis couper ce même tableau à la fin de l'opération (cf. Exemple 2).

Exemple

Exemple 1

```
Array = replicate(sValue, numValues)
```

Exemple 2

```
; Déclaration d'un tableau de dimension prédéfinie :
array = fltarr(1000)
; Compteur pour la coupure :
count = 0L
; Variable pour la lecture du fichier :
dummy = 0.0
; Lecture fichier :
WHILE NOT EOF(lun) DO BEGIN
     READF, lun, dummy
     array[count] = dummy
     count = count + 1
     IF count  GE n_elements(array) THEN $
array=[array,array]
ENDWHILE
; Coupure :
array = array[0:count]
```

Tr.MatriceCreuse	On doit utiliser la fonction `SPRSIN` pour stocker efficacement les matrices creuses.
M=0;F=1;P=100;V=1	
Quelconque	

Description

Une matrice creuse peut être définie comme un tableau contenant un grand nombre de valeurs nulles. La fonction SPRSIN permet de retenir dans un tableau les éléments de valeur supérieure ou égale à un certain seuil, et mettre toutes les autres à 0. Les éléments conservés sont alors représentés par la fonction

SPRSIN sous la forme d'une structure contenant deux vecteurs : l'un contenant les valeurs retenues du tableau, l'autre contenant des indices vers le premier vecteur.
Remarque : La fonction FULSTR permet de rendre à un tableau son état initial (c'est à dire son état avant l'utilisation de la fonction SPRSIN).

Justification

La fonction SPRSIN permet de réduire l'espace mémoire nécessaire au stockage d'un tableau. Cette fonction peut s'employer conjointement à des routines de compression de données (compression en ondelettes par exemple).

Exemple

```
; Transformation en ondelettes d'une image :
transform = WTN(image, 4)
; Compression avec la fonction SPRSIN :
reducedTransform = SPRSIN(transform)
; Restitution du tableau original :
original = FULSTR(reducedTransform)
```

Tr.Libération	Un tableau déclaré au niveau d'un programme principal doit être libéré après exécution.
M=1;F=3;P=80;V=1	
Quelconque	

Description

Les tableaux déclarés au sein d'une routine sont locaux à cette routine, et sont automatiquement détruits en sortie de routine. Par contre un tableau déclaré au niveau du programme principal existe tant qu'il n'est pas explicitement détruit.
La destruction peut être effectuée en affectant une valeur scalaire à ce tableau.

Justification

Optimisation de l'utilisation de la mémoire.

Exemple

```
; Création du tableau :
array = INTARR(10000, 20000)
; Libération mémoire :
Array = 0
```

Tr.StructAffect	On doit utiliser la procédure `STRUCT_ASSIGN` pour affecter à une structure une structure de définition différente.
M=1;F=1;P=60;V=1	
Quelconque	

Description

L'opérateur « = » ne peut affecter à une structure une structure de définition différente. Par exemple, considérons la structure SRC avec la définition suivante :

```
source = { SRC, A:FINDGEN(4), B:12 }
```

On décide de créer une seconde instance de la même structure mais avec une définition légèrement différente, par exemple :

```
dest = { SRC, A:INDGEN(2), C:20 }
```

L'exécution de la dernière instruction produit le message d'erreur suivant :

```
% Conflicting data structures: <INT       Array[2]>,SRC.
% Execution halted at:  $MAIN$
```

Depuis la version 5.1 d'IDL, il existe un mécanisme permettant de résoudre ce problème. La procédure STRUCT_ASSIGN effectue une copie champ à champ d'une structure vers une autre structure. Les champs sont copiés en suivant les règles suivantes :
Tous les champs présents dans la structure destination et absents de la structure source sont mis à zéro (0, chaîne de caractères vide, pointeur « null »ou référence objet en fonction du type du champ).
Tous les champs trouvés à la fois dans la structure source et la structure destination sont copiés un par un. Si nécessaire, une conversion de type est réalisée. Si un champ de la structure source contient moins d'éléments que le champ correspondant de la structure destination, alors les éléments supplémentaires dans le champ de la structure destination sont mis à zéro. Si un champ de la structure source contient plus d'éléments que le champ correspondant dans la structure destination, les éléments supplémentaires sont ignorés.

Justification

Le mécanisme STRUCT_ASSIGN permet de simplifier le code et de le rendre plus efficace.

Exemple

En utilisant la procédure STRUCT_ASSIGN, il devient possible d'effectuer une affectation qui échouait avec l'opérateur « = ».

```
source = { src, a:FINDGEN(4), b:12 }
dest = { dest, a:INDGEN(2), c:20 }
STRUCT_ASSIGN, source, dest, /VERBOSE
```

IDL affiche :

```
% STRUCT_ASSIGN: SRC tag A is longer than destination.
  The end will be clipped.
% STRUCT_ASSIGN: Destination lacks SRC tag B. Not copied.
```

En vérifiant la variable dest, on se rend compte qu'elle possède la définition de la structure dest et les données de la structure source :

```
HELP, dest, /STRUCTURE
```

IDL affiche :

** Structure DEST, 2 tags, length=6:

A	INT	Array[2]
C	INT	0

Tr.TableauPointeur	Le déréférencement d'un tableau de pointeurs doit s'effectuer en déréférençant chacun des pointeurs du tableau.
M=1;F=1;P=61;V=1	
Quelconque	

Description

L'opérateur « * » de déréférencement de pointeur requiert une variable SCALAIRE de type pointeur. Cela implique qu'avec un tableau de pointeurs, il est nécessaire d'indiquer l'élément du tableau à déréférencer.

Justification

Sans objet.

Exemple

```
ptarr = PTRARR(3, /ALLOCATE_HEAP)
FOR i = 0, 2 DO *ptarr[i] = I
PRINT, *ptarr
; IDL produit un message d'erreur :
% Expression must be a scalar in this context: PTARR.
% Execution halted at: $MAIN$
; Pour afficher le contenu du tableau de pointeurs, utiliser :
FOR i = 0, N_ELEMENTS(ptarr)-1 DO PRINT, *ptarr[I]
```

Tr.PointeurPointeur	On doit utiliser le parenthésage pour déréférencer des pointeurs de pointeurs.
M=0;F=1;P=101;V=1	
Quelconque	

Description

On peut créer un pointeur de pointeur de la façon suivante :
struct = {data:'10.0', pointer:ptr_new(20.0)}
ptstruct = PTR_NEW(struct)

Pour déréférencer ce pointeur de pointeur, il faut nécessairement utiliser les parenthèses : le pointeur vers la structure est déréférencé, puis le pointeur à l'intérieur de la structure est déréférencé.
PRINT, *(*pstruct).pointer

Justification

Sans objet.

Tr.RammasseMiette	On doit éviter l'utilisation du mécanisme « garbage collector » d'IDL.
M=1;F=2;P=25;V=2	
Quelconque	

Description

L'utilisation de la procédure HEAP_GC pour libérer l'espace mémoire des pointeurs non référencés est déconseillé.

Justification

Le mécanisme « garbage collector » d'IDL est très coûteux en terme d'opérations informatiques, et son utilisation dénote une mauvaise utilisation des pointeurs. Les applications devraient être écrites de telle sorte qu'aucune référence à un pointeur ou à un objet ne soit perdue, et ce afin d'éviter l'utilisation du mécanisme « garbage collector » d'IDL. Une solution consiste à s'assurer de l'utilisation paritaire des routines PTR_NEW/PTR_FREE et OBJ_NEW/OBJ_DESTROY.

Tr.Conversion	On doit éviter les conversions de type superflues.
M=2;F=1;P=26;V=1	
Quelconque	

Description

Sans objet.

Justification

La conversion de type automatique est un avantage d'IDL particulièrement appréciable dans un environnement interactif. Cependant, et pour des raisons de performance, il est préférable d'éviter les conversions de type superflues. Cela revient en fait à respecter l'interface des routines IDL en utilisant les types de données appropriés.

Tr.AccesHorsPortee	On doit utiliser les routines `SCOPE_VARFETCH` et `SCOPE_LEVEL` pour accéder aux variables situées en dehors de la portée de la routine courante.
M=0;F=0;P=116;V=1	
Quelconque	

Description

Depuis la version 6.1, IDL permet aux routines d'accéder en lecture et en écriture à des variables en dehors de leur portée.

Justification

Certaines applications requièrent cette fonctionnalité. On peut notamment citer les applications possédant une interface graphique ayant besoin d'accéder à certaines variables directement, sans avoir à passer des variables en paramètres. Par exemple, chaque iTool possède une option permettant d'importer une variable IDL pour la visualiser (File->Import->From an IDL variable). Cette variable n'a jamais été passée en paramètre au iTool qui cependant peut y accéder.

La procédure SCOPE_VARFETCH est utilisée pour accéder à ces variables. La procédure SCOPE_LEVEL permet de déterminer la portée d'une variable. Se référer à l'aide en ligne pour de plus amples informations relatives à ces routines.

Exemple

Extraction des variables déclarées au niveau du prompt IDL, puis recréation automatique de ces variables au sein d'une routine.

Pour tester cet exemple, entrer plusieurs variables au niveau du prompt IDL :

```
x = 3
tab = INDGEN(100)
```

Exécuter la routine getVariablesFromMainLevel. Cette routine extrait l'ensemble des variables déclarées au niveau du programme principal (donc x et tab) et les recrée au sein de la routine getVariablesFromMainLevel.

```
pro getVariablesFromMainLevel
   ; Extraction des noms de variables déclarées au niveau du prompt IDL :
    variables = SCOPE_VARNAME(LEVEL = 1)
  ; Boucle sur les résultats :
  for i = 0, N_ELEMENTS(variables)-1 do begin
  ; Extraction des variables déclarées au niveau du programme principal:
     var = SCOPE_VARFETCH(variables(i), LEVEL = 1)
     ; Recréation dans la portée
     ; de la routine getVariablesFromMainLevel de ces variables :
    (SCOPE_VARFETCH(variables(i), LEVEL=2)) = var

  end
  help, x, tab
end
```

Tr.CommonParam	On ne doit jamais passer en paramètre positionnel, une variable définie dans un `COMMON`.
M=3;F=1;P=81;V=1	
Quelconque	

Description

Dans le cas contraire, la même variable est référencée par deux noms différents dans la même routine.

Justification

Les COMMONs sont un moyen de passer des paramètres à une routine. Il est donc dangereux et inutile de les passer en même temps comme des paramètres positionnels d'une routine sous peine de manipuler la même variable sous deux noms différents.

Tr.ParamNombre	On doit limiter le nombre de paramètres positionnels.
M=0;F=1;P=103;V=1	
Quelconque	

Description

Chaque projet définit le nombre maximum de paramètres positionnels. Le nombre de paramètres peut être diminué par la définition de structures par exemple.
Cette règle complète la règle commune Don.Localité pour obtenir un équilibre entre les COMMONs et les paramètres positionnels.

Justification

Un trop grand nombre de paramètres positionnels nuit à la lisibilité du code et alourdit la phase d'intégration et de maintenance.

Tr.ParamRespect	On doit respecter le nombre de paramètres positionnels lors de l'appel d'une routine.
M=3;F=3;P=71;V=2	
Quelconque	

Description

La fonction " N_PARAMS " permet de vérifier que tous les paramètres positionnels sont effectivement renseignés, et éventuellement si des paramètres supplémentaires ont été passés. La fonction N_ELEMENTS permet de tester la présence d'un paramètre positionnel ou d'un mot-clé en renvoyant le nombre d'éléments de ce paramètre.

Justification

Cette règle permet d'assurer la robustesse du code. IDL n'effectue aucun contrôle de cohérence entre la déclaration d'une procédure et son appel. Si le nombre de paramètres obligatoires n'est pas respecté, le déroulement de la procédure peut être totalement incohérent.

Exemple

Exemple 1 : Utilisation de N_PARAMS

```
PRO Ma_fonction, var1, var2

IF (N_PARAMS() NE 2) THEN BEGIN
        PRINT, 'ERREUR : nombre de paramètres incorrect'
endif
...
END
```

Exemple 2 : Utilisation de N_ELEMENTS

```
PRO Ma_fonction, array

IF (N_ELEMENTS(array) GT 0) THEN BEGIN

     ; Traitement spécifique.
```

```
ENDIF

END
```

Tr.TestMotsClés	On doit vérifier systématiquement la présence des mots-clé.
M=2;F=2;P=16;V=1	
Quelconque	

Description

Utiliser systématiquement les routines N_ELEMENTS, KEYWORD_SET, ou ARG_PRESENT pour vérifier la présence des mots-clé.
N_ELEMENTS : cette fonction renvoie la valeur 0 si un mot-clé est indéfini.
KEYWORD_SET : cette fonction renvoie la valeur 0 si la valeur de son argument est 0 ou si son argument est indéfini.
ARG_PRESENT : il faut être prudent avec l'utilisation de cette fonction. Cette fonction n'indique pas si un mot-clé est présent ou non. Si le mot-clé est absent, elle renvoie la valeur 0. Si mot-clé est présent, elle renvoie 1 si le mot-clé est passé par référence, et 0 si le mot-clé est passé par valeur.

Justification

Le test des mots-clé permet d'éviter des comportements aléatoires dans les applications.

Tr.ScalaireTableau	Lorsqu'une expression fait intervenir des scalaires et des tableaux, on doit effectuer les opérations scalaires avant les opérations sur les tableaux.
M=1;F=1;P=63;V=1	
Quelconque	

Description

Sans objet.

Justification

La vitesse de calcul des expressions arithmétiques peut être améliorée par un facteur de 2 ou plus en effectuant les opérations scalaires avant les opérations sur les tableaux.

Exemple

L'expression :

```
answer = array * (scalar1 / scalar2)
```

est beaucoup plus rapide que l'expression équivalente :

```
answer = array * scalar1 / scalar2
```

Tr.CalculMatriciel	On doit intégrer les tableaux directement dans les expressions sans passer par des boucles.
M=2;F=1;P=28;V=1	
Quelconque	

Description

IDL est un langage orienté tableau. Il est donc tout à fait possible d'intégrer des tableaux dans des expressions.

Justification

L'intégration directe des tableaux dans les expressions est beaucoup plus rapide que l'utilisation de boucles. Se référer aux règles concernant les tableaux.

Tr.End	On doit utiliser les instructions `END` appropriées dans les blocs d'instructions.
M=2;F=1;P=30;V=2	
Quelconque	

Description

IDL permet la définition de blocs d'instructions BEGIN … END. L'instruction END utilisée doit correspondre au type d'instruction située avant le début du bloc :

```
IF … ENDIF
ELSE … ENDELSE
FOR … ENDFOR
CASE … ENDCASE
REPEAT … ENDREP
WHILE … ENDWHILE
SWITCH … ENDSWITCH
```

Justification

Le respect de cette règle, utilisée conjointement avec l'indentation, améliore grandement la lisibilité et la maintenabilité d'un code.

Tr.LimiteIf	On doit limiter l'utilisation de l'instruction `IF` avec les tableaux.
M=1;F=1;P=65;V=1	
Quelconque	

Description

Lorsqu'une instruction IF apparaît au milieu d'une boucle, et que chaque élément d'un tableau apparaît dans une expression conditionnelle, la boucle peut très souvent être remplacée par des expressions logiques sur les tableaux.

Justification

Les programmes utilisant des expressions matricielles fonctionnement plus rapidement que les programmes utilisant des scalaires, des boucles et des instructions IF.

Exemple

Exemple 1

```
Dans ce premier exemple, on ajoute tous les éléments positifs du tableau
B au tableau A.
; Manière rapide: on masque les éléments négatifs en utilisant des
opérateurs "tableau" :
A = A + (B GT 0) * B
; Manière encore plus rapide :
A = A + (B > 0)
Contre Exemple 1
; Manière lente : utilisation d'une boucle :
FOR I = 0, (N-1) DO IF B[I] GT 0 THEN A[I] = A[I] + B[I]
```

Exemple 2

Dans ce deuxième exemple, chaque élément du tableau C est défini par la racine carrée de A si A[I] est positif; sinon, C[I] est défini par l'opposé de la racine carrée de la valeur absolue de A[I].

```
; L'utilisation d'une expression matricielle est beaucoup plus rapide :
C = ((A GT 0) * 2-1) * SQRT(ABS(A))
```

L'expression (A GT 0) prend la valeur 1 si A[I] est positif, sinon la valeur 0. L'expression (A GT 0)*2-1 est égale à +1 si A[I] est positif et est égale à –1 si A[I] est négatif. Le même résultat peut être obtenu en utilisant la fonction WHERE pour déterminer les indices des éléments négatifs du tableau A, et effectuer une négation des éléments correspondants du résultat :

```
; Obtention des indices des éléments négatifs :
negs = WHERE(A LT 0, count)
; Calcul de la racine carrée de la valeur absolue :
C = SQRT(ABS(A))
; Négation des éléments du tableau C correspondant aux valeurs négatives
du tableau A :
```

```
IF count GT 0 THEN $
     C[negs] = -C[negs]
```

Contre Exemple 2

```
; L'utilisation d'une instruction IF est lente :
FOR i = 0, (N-1) DO IF A[I] LE 0 THEN $
     C[I] = SQRT(-A[I]) ELSE C[I] = SQRT(A[I])
```

Tr.CaseSwitch	On doit utiliser l'instruction `CASE` ou l'instruction `SWITCH` de façon appropriée.
M=1;F=0;P=105;V=0	
Quelconque	

Description

Les instructions CASE et SWITCH ont un fonctionnement similaire, mais diffèrent sur le point suivant :
avec l'instruction CASE, le code s'arrête après l'exécution de l'instruction appropriée au cas;
par contre, avec l'instruction SWITCH, le code exécute l'instruction appropriée au cas et passe à l'instruction suivante:

```
CASE                    SWITCH
X=2                     x=2
CASE x OF               SWITCH x OF
   1: PRINT,               1: PRINT,
'one'                   'one'
   2: PRINT,               2: PRINT,
'two'                   'two'
   3: PRINT,               3: PRINT,
'three'                 'three'
   4: PRINT,               4: PRINT,
'four'                  'four'
ENDCASE                 ENDSWITCH


IDL Prints:             IDL Prints:
Two                     two
                        three
                        four
```

En raison de cette différence, l'instruction BREAK est plus souvent utilisée avec l'instruction SWITCH qu'avec l'instruction CASE.
Par exemple, on peut ajouter une instruction BREAK dans l'exemple SWITCH pour obtenir le même comportement que l'exemple CASE :

```
x = 2
SWITCH x OF
     1 : PRINT , 'ONE'
     2 : BEGIN
           PRINT, 'two'
           BREAK
     END
```

```
    3 : PRINT, 'three'
    4 : PRINT, 'four'
ENDSWITCH

IDL affiche :
two
```

Dans un bloc CASE, si un cas particulier n'est pas traité et que le cas ELSE n'est pas implémenté, IDL affiche un message d'erreur et l'exécution est interrompue. Dans le même cas, l'instruction SWITCH n'échoue pas : l'exécution se poursuit immédiatement après le bloc SWITCH.

Justification

La décision d'utiliser l'instruction CASE ou l'instruction SWITCH revient à décider du comportement qui correspondra le mieux à la logique du code.

Tr.CasElse	Le cas `ELSE` est obligatoire dans un choix multiple.
M=1;F=1;P=66;V=2	
Quelconque	

Description

Le cas "ELSE" (autre cas) d'une instruction de choix multiple est obligatoire. Si aucun traitement n'est prévu, il faut produire un message ou écrire un commentaire approprié ou sortir en erreur.
Remarque : Comme indiqué dans la règle Tr.CaseSwitch, la non-utilisation du cas ELSE avec l'instruction SWITCH ne provoque pas de message d'erreur ou d'interruption dans l'exécution du code si le cas n'est pas trouvé. L'exécution est simplement poursuivie après le bloc SWITCH … ENDSWITCH.

Justification

Le traitement des autres cas d'une instruction de choix multiple permet de se prémunir contre les oublis et de traiter le cas par défaut. Cela facilite grandement la maintenance. D'autre part, l'absence de cas ELSE dans l'instruction CASE peut provoquer un message d'erreur et une interruption dans l'exécution du code, si le cas n'est pas trouvé.

Tr.FusionBoucles	On doit fusionner les boucles dès que possible.
M=0;F=1;P=106;V=1	
Quelconque	

Description

Sans objet.

Justification

Lorsque deux boucles opèrent sur les mêmes éléments, il peut être possible de combiner ces boucles et d'effectuer les deux opérations en une seule boucle.

Tr.Vectorisation	On doit éliminer les boucles par vectorisation.
M=2;F=1;P=31;V=1	
Quelconque	

Description

La vectorisation fait référence au procédé d'élimination des boucles dans un programme, et à l'utilisation directe des opérateurs logiques et arithmétiques sur les tableaux, sans avoir à utiliser de boucles.

Justification

La vectorisation est utile dans tous les langages interprétés, et permet d'éviter la surcharge des instructions dans la mémoire de l'interpréteur. La vectorisation est une technique générale de programmation avec IDL et devrait être utilisée dès que possible. IDL facilite la vectorisation dans la mesure où il inclut un grand nombre d'opérateurs qui fonctionnent aussi bien sur des tableaux que sur des vecteurs.

Exemple

Calcul de la somme des éléments d'un tableau en utilisant la fonction TOTAL.

```
TotalArray = TOTAL(array)
```

Calcul de la somme des éléments d'un tableau en utilisant une boucle (à éviter)

```
TotalArray = 0.0
FOR I = 0, N_ELEMENTS(array)-1 DO $
     totalArray += array[I]
```

Tr.AppelIndirect	On ne doit pas utiliser la routine `EXECUTE` pour le développement d'applications à utiliser avec IDL Virtual Machine.
M=0;F=0;P=118;V=2	
Quelconque	

Description

IDL inclut trois routines dédiées à l'appel indirect de routines : CALL_FUNCTION, CALL_PROCEDURE et EXECUTE., mais IDL Virtual Machine ne supporte pas la routine EXECUTE.
Remarque : Les routines CALL_FUNCTION et CALL_PROCEDURE sont moins flexibles que la routine EXECUTE, mais sont également beaucoup plus rapides.

Justification

Sans objet.

Exemple

Cet exemple, extrait de la fonction SVDFIT, appelle une fonction dont le nom est passé en mot-clé sous forme d'une chaîne de caractères. Si le mot-clé est omis, la fonction POLY est appelée.

```
; Déclaration de la fonction :
FUNCTION SVDFIT,..., FUNCT = funct
...
; Utilisation du nom POLY par défaut si funct n'est pas spécifié :
```

```
    IF N_ELEMENTS(FUNCT) EQ 0 THEN FUNCT = 'POLY'

; Constitution d'une chaine de caractères de la forme "a = funct(x,m)" et
exécution.
          Z = EXECUTE('A = '+FUNCT+'(X,M)')

          ...
```

L'exemple ci-dessus pourrait être rendu beaucoup plus efficace en remplaçant l'appel à la routine EXECUTE avec la ligne suivante :

```
A = CALL_FUNCTION(FUNCT, X, M)
```

Tr.AppelsSysteme	On doit favoriser l'utilisation des fonctions et procédures système.
M=2;F=1;P=33;V=0	
Quelconque	

Description

IDL fournit un grand nombre de fonctions et procédures préconstruites pour effectuer les opérations courantes. Ces routines ont été soigneusement optimisées.

Justification

Il n'est pas nécessaire de « réinventer la roue ». IDL inclut une aide en ligne sophistiquée permettant de déterminer si un algorithme spécifique existe déjà dans IDL. D'autre part, le web inclut un nombre de sites très intéressants proposant de très nombreuses routines, quelque soit le domaine d'application.

Tr.Recursivite	On doit utiliser le mécanisme `FORWARD_FUNCTION` pour la création de fonctions récursives.
M=0;F=0;P=119;V=1	
Quelconque	

Description

La difficulté de création des fonctions récursives réside dans le fait que ces fonctions s'auto-référencent, et que cela peut prêter à confusion lors de l'exécution du code.

Justification

L'utilisation du mécanisme FORWARD_FUNCTION empêche IDL d'interpréter l'appel récurrent à la fonction, dans le corps même de la fonction, comme une référence à une variable.

Exemple

Un exemple célèbre de fonction récursive est celui qui calcule la valeur factorielle du nombre N. Par exemple, N*(N-1)*(N-2)*…*1. Voici la manière dont cet exemple pourrait être écrit avec IDL :

```
FUNCTION FACTORIAL, number
FORWARD_FUNCTION FACTORIAL
IF number LE 1 THEN $
RETURN, 1L
```

```
        RETURN, LONG(number) * FACTORIAL(number - 1)
    END
```

Cette fonction peut être appelée de la façon suivante :

```
    PRINT, FACTORIAL(5)
```

7.6. GESTION DES ERREURS

Err.IHM	Pour gérer les erreurs dans une IHM, on doit utiliser un `CATCH` conjointement à la routine HELP, /LAST_MESSAGE.
M=2;F=2;P=18;V=1	
Quelconque	

Description

La première idée pour gérer efficacement les erreurs dans une IHM est l'utilisation d'un CATCH. Le problème d'un CATCH est que, dans le contexte du code d'une IHM, le CATCH n'indique pas clairement quelle ligne de code a généré l'erreur. On propose dans l'exemple ci-dessous une méthode permettant d'obtenir un tel comportement. La technique consiste simplement à utiliser le mot-clé LAST_MESSAGE avec la routine HELP.

Justification

Cette règle facilite grandement la maintenabilité d'une application de type IHM.

Exemple

Gestion efficace des erreurs dans une IHM.

Le code source de la routine ERROR_MESSAGE est donné dans l'annexe 3 de ce document.

```
CATCH, theError
IF theError NE 0 THEN BEGIN
CATCH, /Cancel
          ok = Error_Message(Traceback=1)
          IF N_Elements(info) NE 0 THEN $
WIDGET_CONTROL, event.top, SET_VALUE = info, /NO_COPY
          RETURN
ENDIF
```

7.7. DYNAMIQUE

7.8. INTERFACES

Int.UniteLogique	Le numéro d'unité logique d'un fichier doit être obtenu par une fonction IDL.
M=3;F=1;P=83;V=1	
Quelconque	

Description

Un fichier IDL peut être ouvert en lecture, écriture ou mise à jour par l'intermédiaire des routines respectives OPENR , OPENW, OPENU et de 2 manières:

? Par spécification « manuelle » d'une unité logique :

Dans ce cas, le développeur décide lui-même d'une unité logique à attribuer. L'unité logique doit alors être comprise entre 1 et 99.

? Par spécification « automatique » d'une unité logique :

Dans ce cas, le développeur « demande » à IDL de choisir une unité logique. L'unité logique sera alors comprise entre 100 et 128.

Il est demandé d'utiliser cette seconde possibilité.
Remarque : les unités logiques 0, -1 et –2 sont des unités logiques réservées.

Justification

Cette règle évite de nombreuses erreurs lors de l'ouverture de fichiers : utilisation d'une unité logique inexistante, déjà occupée ou réservée.

Exemple

```
OPENR, unit, nom_fichier, /GET_LUN
READU, unit, var
FREE_LUN, unit
```

Int.Fichiers	On doit utiliser les fonctions IDL de manipulation de fichiers.
M=2;F=2;P=20;V=2	
Quelconque	

Description

IDL inclut un nombre important de routines dédiées à la manipulation de fichiers pour des opérations telles que la suppression ou la copie de fichiers (routines FILE_*).

Justification

L'intérêt de l'utilisation de ces routines est qu'elles prennent en compte de façon transparente le système d'exploitation. Le développeur n'a donc plus besoin d'écrire des routines dépendantes du système d'exploitation ou d'utiliser la routine « SPAWN » pour ces opérations spécifiques sur les fichiers.

Exemple

Consulter l'aide en ligne pour le fonctionnement des routines FILE_COPY, FILE_DELETE, FILE_MOVE, FILE_MKDIR, etc.…

Int.Portable	On doit utiliser la technique « External Data Representation » pour créer des fichiers binaires portables.
M=3;F=1;P=85;V=2	
Quelconque	

Description

Normalement, les fichiers binaires ne sont pas portables entre machines d'architecture différentes. Cependant, il est possible de créer des fichiers binaires portables en spécifiant le mot-clé XDR avec les procédures OPEN. XDR (eXternal Data Representation) est un standard permettant d'écrire des données binaires suivant une représentation unique. Toutes les machines supportant XDR ont la possibilité de convertir des données entre ce standard et leur propre représentation interne.

Justification

L'utilisation du mot-clé « XDR » avec les procédures OPEN permet de rendre les applications plus portables.

Int.AccesRapide	On doit utiliser la routine ASSOC pour accéder rapidement aux éléments d'un fichier contenant des structures répétitives.
M=1;F=1;P=68;V=1	
Quelconque	

Description

Les données binaires stockées dans des fichiers consistent souvent en des séries répétitives de tableaux ou de structures. Un exemple commun est un fichier contenant une série d'images. Les variables associées d'IDL sont une façon efficace d'accéder à de telles données.
Une variable associée est une variable qui calque la structure d'un tableau IDL ou d'une structure IDL sur le contenu d'un fichier. Le fichier est alors considéré comme un tableau contenant ces unités répétitives. De telles variables ne conservent pas les données en mémoire comme une variable classique. Au lieu de cela, lorsqu'une variable associée est indicée, IDL effectue l'opération d'entrée-sortie requise pour accéder à ces données à l'intérieur du fichier.

Justification

Lorsque l'utilisation des variables associées est appropriée (le fichier d'intérêt contient des structures répétitives), les variables associées présentent les avantages suivants par rapport à la routine READU:

- L'opération d'entrée-sortie est effectuée sur le fichier dès que la variable associée est indicée : il devient ainsi possible d'effectuer une opération d'entrée-sortie sur un fichier en utilisant directement une *expression*.
- Il n'est pas nécessaire de déclarer le nombre maximum de tableaux ou de structures contenus dans le fichier.
- Les variables associées offrent un accès transparent vers les données. Un accès direct à n'importe quel élément du fichier est simple et rapide. Il n'est pas nécessaire de calculer des offsets dans le fichier, ou de positionner le pointeur fichier à un endroit spécifique avant d'effectuer l'opération d'entrée-sortie.
- Cette technique permet d'accéder au contenu de fichiers très volumineux : la limitation principale est la taille du fichier sur le disque dur, et non pas la quantité de mémoire disponible.

Int.GUIBuilder	On doit utiliser le GUIBUILDER pour la création rapide d'une maquette de l'IHM.
M=3;F=1;P=86;V=1	
Quelconque	

Description

Pour créer la maquette d'une interface homme-machine, sélectionne File->New->GUI depuis l'environnement de développement IDL. Se référer à la documentation IDL « IDL GUIBuilder Tools » pour une description complète des outils disponibles avec le GUIBUILDER.
Remarque : Lors de l'utilisation du GUIBUILDER, privilégier le positionnement relatif des widgets. On pourra par exemple créer une base « colonne » qui servira à contenir un ensemble de boutons. Se référer à la règle « Int.Positionnement » pour plus d'information.

Justification

La création d'une maquette avec le GUIBUILDER permet de définir rapidement le style et l'ergonomie d'une IHM sans avoir à écrire de code. Tous les widgets peuvent être positionnés visuellement, de façon

similaire à d'autres environnements de développement tel que Microsoft Visual Studio. D'autre part, le GUIBUILDER permet de générer automatiquement le code associé à la maquette créée.

Int.Positionnement	On doit privilégier le positionnement relatif des widgets d'une IHM.
M=3;F=0;P=12;V=1	
Quelconque	

Description

Les widgets peuvent être positionnés de façon relative ou absolue dans une interface homme-machine. La façon absolue implique l'utilisation des mots-clé XSIZE, YSIZE, SCR_XSIZE, SCR_YSIZE, XOFFSET, ou YOFFSET dans les fonctions de création des widgets. La façon relative implique l'intégration des différents widgets de l'interface homme-machine dans des bases « colonne » ou « ligne ».
Remarque 1 : Essayer de positionner les widgets text, label et list à des endroits où leur dimension absolue peut varier sans que toute l'application en soit affectée. En effet, les polices de caractère utilisées sur les différents systèmes peuvent modifier conséquemment l'apparence physique de ces widgets.
Remarque 2 : L'utilisation du mot-clé GRID_LAYOUT avec la fonction de création WIDGET_BASE permet de superposer sur la base une grille régulière dans laquelle viendront se positionner les différents widgets. Ce mot-clé permet de garantir que tous les widgets de la base posséderont les mêmes dimensions.

Justification

L'utilisation du positionnement relatif des widgets dans une interface homme-machine permet d'améliorer la portabilité de toute l'application. Le positionnement absolu peut produire des interfaces homme-machine incohérentes lors du portage vers une autre plate-forme. En effet, cette autre plate-forme peut avoir des paramètres différents en termes de police de caractères, dimension des polices de caractères, épaisseur des bords, espace entre les widgets, etc. …

Int.Regrouprement	On doit établir des liens entre les différentes parties d'une IHM.
M=2;F=2;P=21;V=0	
Quelconque	

Description

Le mot-clé GROUP_LEADER peut être utilisé avec la fonction de création WIDGET_BASE afin de définir des relations entre les différentes parties d'une IHM.

Justification

Le regroupement des widgets par l'intermédiaire du mot-clé GROUP_LEADER de la fonction de création WIDGET_BASE permet de regrouper de façon appropriée des widgets pour des opérations d'icônification, de positionnement et de destruction.

Int.DynamiqueIHM	On doit regrouper la partie dynamique associée à un panneau dans un même module.
M=3;F=0;P=13;V=1	
Quelconque	

Description

Sans objet.

Justification

Cette règle facilite la modification de la partie dynamique de l'IHM (enchaînement des panneaux) et permet de pouvoir utiliser un même panneau (en fonction de sa conception) dans plusieurs applications.

Int.EtatIHM	On doit utiliser une structure d'état dans une valeur utilisateur pour mémoriser l'état d'une application.
M=3;F=2;P=75;V=1	
Quelconque	

Description

Plusieurs techniques sont envisageables dans une IHM afin de préserver l'état d'une application et d'échanger des données entre routines. La première technique la plus évidente consiste à employer des blocs communs. Cependant, l'utilisation de blocs communs perturbe l'exécution concurrentielle de plusieurs instances d'une même IHM. Une autre solution consiste à utiliser la valeur utilisateur d'un des widgets de l'IHM pour stocker la structure d'état de l'application (voir exemple 1). Pour simplifier le code, on peut également stocker un pointeur vers la structure d'état dans la valeur utilisateur d'un des widgets (voir exemple 2).

Justification

En utilisant cette technique, plusieurs instances d'une même IHM peuvent être exécutées de façon concurrentielle. Comme une valeur utilisateur peut être de n'importe quel type, une structure peut être utilisée pour stocker n'importe quel nombre de variables d'état.

Exemple

Exemple 1 :Utilisation d'une structure d'état dans une valeur utilisateur, pour mémoriser l'état d'une application.

```
; Gestionnaire d'événements :
PRO my_widget_event, event

     ; Récupération de la structure d'état :
WIDGET_CONTROL, event.TOP, GET_UVALUE=state, /NO_COPY

; Code du gestionnaire d'événements :
; …

; Cette ligne de code est nécessaire en raison de l'utilisation de
NO_COPY :
WIDGET_CONTROL, event.TOP, SET_UVALUE=state, /NO_COPY
END
```

```
; Définition de l'interface :
PRO my_widget

; Création des widgets :
wBase = WIDGET_BASE(/COLUMN)
wDraw = WIDGET_DRAW(wBAse, XSIZE=300, YSIZE=300)

; Réalisation de l'interface :
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

; Création d'une structure d'état et stockage dans la valeur utilisateur
; du top level base :

state = {wDraw:wDraw, idxDraw:idxDraw}
WIDGET_CONTROL, wBase, SET_UVALUE=state

; Appel à XMANAGER pour la gestion des widgets :
XMANANAGER, 'my_widget', wBase
END
```

Exemple 2 : Utilisation d'un pointeur vers une structure d'état dans une valeur utilisateur, pour mémoriser l'état d'une application.

```
; Gestionnaire d'évènements :
PRO my_widget_event, event

     ; Récupération du pointeur vers la structure d'état :
WIDGET_CONTROL, event.TOP, GET_UVALUE=pstate

     if PTR_VALID(pstate) then BEGIN

; Code du gestionnaire d'évènements :
; …

     END

END

; Définition de l'interface :
PRO my_widget

; Création des widgets :
wBase = WIDGET_BASE(/COLUMN)
wDraw = WIDGET_DRAW(wBAse, XSIZE=300, YSIZE=300)

; Réalisation de l'interface :
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

; Création d'une structure d'état et stockage dans la valeur utilisateur
; du top level base :
```

```
state = {wDraw:wDraw, idxDraw:idxDraw}
WIDGET_CONTROL, wBase, SET_UVALUE=PTR_NEW(state, /NO_COPY)

; Appel à XMANAGER pour la gestion des widgets :
XMANANAGER, 'my_widget', wBase
END
```

Int.ObjetIHM	On doit encapsuler l'IHM dans un objet pour obtenir une meilleure flexibilité.
M=3;F=0;P=15;V=1	
Quelconque	

Description

On propose ici d'utiliser la technologie objet dans la création d'IHM.

Justification

L'encapsulation d'une IHM dans un objet permet le traitement de cette IHM comme un objet à part entière. Une application extérieure peut alors créer une instance de cet objet, et utiliser les méthodes spécifiques de cet objet pour interagir avec l'IHM. Les intérêts de cette technologie sont multiples :

? Simplicité

L'utilisateur n'a pas besoin de connaître la programmation widget pour interagir avec l'IHM. Il doit simplement utiliser les méthodes de l'interface de l'objet.

? Préservation de la partie fonctionnelle de l'IHM

La création de l'IHM est encapsulée dans une méthode spécifique de l'objet.

? Maintenabilité

L'interaction avec l'IHM s'effectue en ajoutant de nouvelles méthodes dans l'interface de l'objet.

? Réutilisabilité

L'objet décrivant l'IHM peut très facilement être réutilisé dans d'autres applications.

Exemple

Création d'un IHM encapsulée dans un objet.

Dans cet exemple, 2 classes sont proposées. La classe « IMAGE », qui hérite de la classe « IDLGRIMAGE » permet de lire une image quel que soit son format. La classe « IMAGEWIN » qui hérite de la classe « IMAGE » encapsule l'IHM permettant de visualiser l'image sélectionnée.
Pour afficher une image donnée dans l'IHM, il suffit d'entrer la ligne de code suivante :

```
OImageWin = OBJ_NEW("IMAGEWIN",
"c:\RSI\IDL61\EXAMPLES\DATA\glowing_gas.jpg")
```

Pour détruire l'IHM :

```
OBJ_DESTROY, oImageWin
```

Pour les codes associés aux classe « IMAGE » et « IMAGEWIN », consulter l'0 Exemple d'encapsulation d'une IHM dans un objet .

Int.CompoundWidgetUse	On doit utiliser les « compound widgets » dès que possible.
M=2;F=1;P=35;V=0	
Quelconque	

Description

Un « compound widget » est un assemblage de widgets qui se comporte exactement comme un autre widget standard, à la différence qu'il est entièrement écrit en langage IDL. Chacun de ces « compound widget » est dédié à une tâche bien spécifique : on peut citer par exemple CW_BGROUP et CW_PDMENU qui permettent de créer rapidement des menus. Ces fonctions sont plus faciles à utiliser que d'avoir à réécrire le code permettant de gérer ces mêmes menus.
Une liste complète des « compound widgets » disponibles est fournie dans l'aide en ligne à la rubrique « Compound widgets ».

Justification

L'utilisation de « compound widget » permet d'obtenir une meilleure lisibilité, une meilleure fiabilité, et une meilleure efficacité du code.

Int.CompoundWidgetCreat	On doit créer des compound widget dès que possible.
M=2;F=1;P=36;V=0	
Quelconque	

Description

Tout nouveau compound widget créé prend la forme d'une fonction renvoyant l'identifiant du widget base principal décrivant le compound widget. Les caractéristiques principales de tout nouveau compound widget créé sont les suivantes :

- Un compound widget possède une valeur, comme tout autre widget, accessible en lecture/écriture par l'intermédiaire de la routine WIDGET_CONTROL et des mots-clé GET_VALUE et SET_VALUE.
- Un compound widget possède une valeur utilisateur, accessible en lecture/écriture par l'intermédiaire de la routine WIDGET_CONTROL et des mots-clé GET_UVALUE et SET_UVALUE.
- Un compound widget peut générer un événement particulier, comme le font l'ensemble des widgets standard d'IDL.

Justification

La création de nouveaux compound widgets joue en faveur de la réutilisabilité du code, mais n'a de sens que dans le cadre d'une programmation « classique » des IHM. En effet, comme indiqué dans la règle « Int.ObjetIHM », une IHM peut être entièrement encapsulée dans un objet réutilisable par définition.

Exemple

Consulter l'0Annexe B - EXEMPLE DE CREATION D'UN COMPOUND WIDGET de ce document qui fournit le code source complet commenté d'un modèle de compound widget.

Int.PartageDonnées	On doit utiliser la technique de partage de données, si plusieurs objets graphiques sont associés aux mêmes données.
M=1;F=0;P=108;V=0	
Quelconque	

Description

Il est possible avec IDL de créer des objets graphiques partageant les mêmes données. Ce partage est effectué en utilisant le mot-clé SHARE_DATA dans les fonctions de création des objets graphiques. Par exemple, N objets peuvent partager des données avec un objet image global.

Justification

On peut imaginer le cas où l'on souhaite représenter des données identiques avec des paramètres de visualisation différents (la palette de couleurs par exemple). Pour charger de nouvelles données, il suffit de les charger dans l'objet graphique global, sans avoir à les charger dans les objets graphiques individuels.

Exemple

Création de quatre objets image partageant les mêmes données. Pour modifier les données représentées par ces images, il suffit de modifier les données associées à l'objet image global.

```
pro test

    ; Création de l'objet image global :
    oPalette1 = OBJ_NEW("IDLGRPALETTE")
    oPalette1->LOADCT, 1
    oGlobalImage =
           OBJ_NEW("IDLGRIMAGE", BINDGEN(256,256), LOCAT = [0, 0], $
    PALETTE = oPalette1)

    ; Création des autres objets images, partageant les
    ; données avec l'objet image global :
    oPalette2 = OBJ_NEW("IDLGRPALETTE")
    oPalette2->LOADCT, 2
    oImage2 = OBJ_NEW("IDLGRIMAGE", SHARE_DATA = oGlobalImage, $
    LOCATION = [256, 0], PALETTE = oPalette2)

    oPalette3 = OBJ_NEW("IDLGRPALETTE")
    oPalette3->LOADCT, 3
    oImage3 = OBJ_NEW("IDLGRIMAGE", SHARE_DATA = oGlobalImage, $
    LOCATION = [0, 256], PALETTE = oPalette3)

    oPalette4 = OBJ_NEW("IDLGRPALETTE")
    oPalette4->LOADCT, 4
    oImage4 = OBJ_NEW("IDLGRIMAGE", SHARE_DATA = oGlobalImage, $
    LOCATION = [256, 256], PALETTE = oPalette4)

    ; Création de la hiérarchie des objets graphiques :
    oModel = OBJ_NEW("IDLGRMODEL")
    oModel->ADD, oGlobalImage
    oModel->ADD, oImage2
    oModel->ADD, oImage3
    oModel->ADD, oImage4
```

```
        ; Création de la vue :
        oView = OBJ_NEW("IDLGRVIEW", DIMENSIONS = [256*2, 256*2], $
        VIEWPLANE_RECT = [0, 0, 256*2, 256*2])
        oView->ADD, oModel

        ; Création d'un objet destination :
        oWin = OBJ_NEW(
          "IDLGRWINDOW", DIMENSIONS = [256*2, 256*2], GRAPHICS_TREE = oView)
        oWin->DRAW, oView

        ; Modification des données de l'objet image global,
        ; la répercussion sur les autres objets images est automatique.
        wait, 1
        oGlobalImage->SETPROPERTY, DATA = SHIFT(DIST(256, 256), 128, 128)
        oWin->DRAW, oView
        wait, 1

        ; Libération mémoire :
        OBJ_DESTROY, [oGlobalImage, oImage2, oImage3, oImage4, $
                      oPalette1, oPalette2, oPalette3, oPalette4, $
                      oModel, oView, oWin]

    end
```

Int.AccelGraphique	On doit utiliser le mot-clé `RENDERER` pour accélérer la visualisation des objets graphiques.
M=1;F=1;P=70;V=2	
Quelconque	

Description

Tous les objets graphiques destination possèdent un mot-clé RENDERER permettant de passer d'un rendu graphique *software* à un rendu graphique *hardware* et vice-versa.
Remarque : le mode software ou hardware peut être également activé en passant par les préférences d'IDL :
File->Preferences->Graphics->Default object graphic renderer

Justification

Par défaut, IDL tente d'utiliser un rendu graphique hardware lorsque la machine le permet. Pour des raisons de performance, on peut donner le conseil suivant : passer en mode software pour la visualisation d'objets 2D type image, et en mode hardware pour le reste.

Int.AccelGraphInst	On doit utiliser l'instanciation pour accélérer la visualisation des objets graphiques.
M=0;F=1;P=110;V=1	
Quelconque	

Description

Lorsque seule une partie d'une scène graphique est amenée à changer, il peut être plus efficace de remettre à jour la partie dynamique de la scène, et de laisser le reste inchangé. Cette technique s'appelle l'instantiation. La technique d'instantiation est implémentée par l'intermédiaire des mots-clé CREATE_INSTANCE et DRAW_INSTANCE de la méthode DRAW des objets destination.

Justification

L'objectif est de rendre plus fluide la visualisation dynamique d'objets graphiques.

Exemple

Consulter la rubrique « Instancing » dans l'aide en ligne d'IDL.

Int.AccelGraphRetain	On doit utiliser le mot-clé `RETAIN` de la classe `IDLGRWINDOW` pour rafraîchir efficacement les fenêtres.
M=0;F=1;P=111;V=2	
Quelconque	

Description

Lors de la visualisation, une fenêtre objet peut être partiellement ou complètement dissimulée par une autre application. Lorsque cette fenêtre est repositionnée en premier plan, son contenu doit être rafraîchi. La manière dont ce rafraîchissement est effectué dépend de la valeur du mot-clé RETAIN utilisé avec la classe IDLGRWINDOW.

Justification

L'utilisation du mot-clé RETAIN est une question de choix :

- RETAIN = 0 : Aucun rafraîchissement n'est effectué. Cependant, si l'objet fenêtre est un élément d'une IHM, un événement peut être généré pour signaler qu'un rafraîchissement est nécessaire (grâce à l'utilisation du mot-clé EXPOSE_EVENTS avec la fonction de création WIDGET_DRAW).
- RETAIN = 1 Le rafraîchissement est effectué directement par le système d'exploitation, si le système d'exploitation le peut.
- RETAIN = 2 Le rafraîchissement est effectué par IDL. Ce mode peut être très coûteux en performance, particulièrement pour l'affichage dynamique de données (utiliser alors l'instantiation, voir la règle Int.AccelGraphInst).

7.9. QUALITE

QA.IDL_PATH	La variable d'environnement "`IDL_PATH`" doit être positionnée en dehors de l'application et ne doit pas être modifiée par les routines.
M=2;F=1;P=38;V=1	
Quelconque	

Description

Cette variable d'environnement indique le (ou les répertoires) dans lequel se trouvent les routines utilisables par IDL (.pro et sav.). Les répertoires sont scrutés dans l'ordre de leur apparition dans "IDL_PATH". Cette variable est comparable à la variable d'environnement "PATH" utilisée sous UNIX. La variable IDL_PATH peut être modifiée de plusieurs façons :

? De façon temporaire
Par modification directe de la variable système !path. Lors de l'ouverture d'une nouvelle session IDL, la variable système !path retrouvera sa valeur par défaut.
? De façon définitive
- o Dans un fichier batch :
Exemple : path = getenv("TEMP")+PathSep()+!path
- o En utilisant l'option File->Preferences->Path de IDLDE

Justification

L'application de cette règle permet de mieux cerner la localisation de l'exécutable, augmente la fiabilité et facilite la mise au point.

7.10. AUTRES REGLES

Sans objet

8. AUTRES ASPECTS SPECIFIQUES AU LANGAGE

8.1. LIENS EXTERNES

L'objet de ce paragraphe est de présenter rapidement les différents mécanismes de liens externes proposés par IDL afin de choisir au mieux le mécanisme le plus approprié au logiciel à développer.

Les différentes techniques de lien externe présentées sont valables pour FORTRAN et C/C++ , quelque soit le système d'exploitation, mais ne font pas référence à JAVA (l'appel de librairies JAVA depuis IDL se faisant par l'intermédiaire d'une technologie IDL spécifique : le JAVA bridge).

Remarque: Tout ce qui concerne la programmation des liens externes est décrit dans le fichier « edg.pdf » de la distribution IDL.

IDL est un langage ouvert à d'autres langages de programmation type C/C++ ou FORTAN, et offre les possibilités suivantes :

? L'envoi de commandes au système d'exploitation par l'intermédiaire de la procédure SPAWN.
? L'appel de routines compilées dans d'autres langages de programmation, par l'intermédiaire du mécanisme CALL_EXTERNAL.
? L'insertion au cœur d'IDL de nouvelles routines natives compilées dans une librairie partagée, par l'intermédiaire du mécanisme LINKIMAGE.
? L'utilisation IDL en tant que librairie partagée au sein d'un environnement client, type Visual C++.

Pourquoi utiliser des techniques de lien externe ? Les raisons suivantes peuvent être citées :

? Disposer d'une librairie déjà développée avec un langage qui n'est pas IDL: la traduction en IDL et la correction du code peut demander beaucoup de temps.
? Besoin de contrôler des instruments : pour des raisons de sécurité, IDL ne permet pas l'accès bas-niveau au hardware.
? Optimisation: normalement le code IDL est déjà optimisé, spécialement s'il agit de traitement de tableaux, mais dans certains cas (algorithmes itératifs en particulier) le code externe peut s'exécuter plus rapidement.

Quand utiliser CALL_EXTERNAL et quand utiliser LINKIMAGE ?

? CALL_EXTERNAL
  - o Bien souvent, les développeurs doivent utiliser des librairies externes dédiées à des tâches spécifiques, mais ne possèdent pas le code source. Dans ce cas, cette librairie externe ne peut être utilisée dans IDL que par l'intermédiaire du mécanisme CALL_EXTERNAL, sous réserve de connaître l'interface de cette librairie.
  - o Si une librairie partagée n'est utilisée dans IDL que de façon ponctuelle, l'utilisation du mécanisme CALL_EXTERNAL est suffisante.
  - o **LIMITATIONS :**
    - ✍ Le mécanisme CALL_EXTERNAL ne peut renvoyer dans IDL que des variables de type scalaire.
    - ✍ Il n'est pas possible de passer des mots-clé à la fonction de la librairie partagée.
? LINKIMAGE
  - o Si une librairie partagée est utilisée de façon répétée dans IDL, il peut être intéressant d'utiliser le mécanisme LINKIMAGE. LINKIMAGE permet effectivement d'ajouter de nouvelles routines au sein d'IDL . Cela suppose toutefois de disposer du code source de

cette librairie, et de pouvoir modifier ce code en y incluant les fonctions de l'API C de IDL qui vont permettre d'ajouter ainsi de nouvelles routines au sein d'IDL.

- Il est possible de passer des mots-clé à la fonction de la librairie partagée.
- **LIMITATIONS :**
  - La connaissance de l'API C d'IDL est absolument nécessaire : en effet, la création d'une librairie partagée (C/C++ ou FORTRAN) en vue d'une utilisation avec le mécanisme LINKIMAGE implique obligatoirement une utilisation de l'API C d'IDL.

8.2. LES ITOOLS

L'objet de ce chapitre n'est pas d'énumérer des règles de programmation relatifs aux iTools, mais de présenter quelques techniques de programmation propres aux iTools.

Référencement programmatique du iTool courant

Pour changer les propriétés d'un iTool depuis la ligne de commande IDL (ou depuis une routine IDL), il est nécessaire de récupérer au préalable une référence objet vers cet iTool. L'utilisation de la routine ITGETCURRENT (disponible depuis la version 6.1 d'IDL) et du mot-clé TOOL permet d'atteindre cet objectif.

Exemple : Obtention d'un identifiant et d'une référence objet vers le iTool actif.

```
idTool = ITGETCURRENT(TOOL = oTool)
```

Remarque

Un iTool peut être rendu actif de 3 manières :

- Un iTool vient juste d'être créé et est nécessairement le iTool actif.
- Un iTool a été sélectionné à la souris.
- Un iTool a été sélectionné en utilisant la routine ITCURRENT : ITCURRENT, idTool

Extension des messages d'erreur produits par les iTools

Bien souvent, durant la programmation d'une application iTool, le développeur se rendra compte que les messages d'erreur produits ne sont pas très souvent explicites. On propose ici une méthode permettant de remonter efficacement à la source de l'erreur :

- Editer le fichier dlitimessaging__define.pro
- Rechercher les lignes de code suivantes :

```
;; fill in a prompt object and send it to the UI
oMsg = obj_new("IDLitError", description=strMessage, $
               message=title, severity=severity)
iStatus = self.__oTool->SendMessageToUI(oMsg)
obj_destroy, oMsg
```

- Aussitôt après ces lignes de code, insérer le code suivant :

```
IF severity GT 1 THEN BEGIN
     ; Obtention de la pile d'appel et du nom de la routine appelante.
      Help, Calls=callStack
      callingRoutine = (StrSplit(StrCompress(callStack[1])," ", /Extract))[0]
      Help, /Last_Message, Output=traceback
      Print,''
      Print, 'Traceback Report from ' + StrUpCase(callingRoutine) + ':'
      Print, ''
      FOR j=0,N_Elements(traceback)-1 DO Print, " " + traceback[j]
ENDIF
```

Désormais, lorsqu'une erreur se produit, la liste des routines appelées avant que l'erreur ne se produise apparaîtra clairement. Par exemple :

```
% Unable to invoke method on NULL object reference: <OBJREF
(<NullObject>)$GT.
% Execution halted at:  IDLITVISPLOT::SETPROPERTY  650
H:\RSI\IDL60\lib\itools\components\idlitvisplot__define.pro
%                      IDLITOPSETPROPERTY::_DOSETPROPERTY   82
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitopsetproperty__define.pro
%                      IDLITOPSETPROPERTY::_EXECUTEONTARGET  187
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitopsetproperty__define.pro
%                      IDLITOPSETPROPERTY::DOSETPROPERTYWITH_EXTRA  289
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitopsetproperty__define.pro
%                      IDLITSRVCREATEVISUALIZATION::_APPLYPROPERTIES  377
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsrvcreatevisualization__define.pro
%                      IDLITSRVCREATEVISUALIZATION::_CREATE  307
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsrvcreatevisualization__define.pro
%                      IDLITSRVCREATEVISUALIZATION::CREATEVISUALIZATION  736
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsrvcreatevisualization__define.pro
%                      IDLITSYSTEM::CREATEVISUALIZATION 2041
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsystem__define.pro
%                      IDLITSYSTEM::CREATETOOL 1985
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsystem__define.pro
%                      IDLITSYS_CREATETOOL  130
H:\RSI\IDL60\LIB\ITOOLS\framework\idlitsys_createtool.pro
%                      IPLOT             316 H:\RSI\IDL60\LIB\ITOOLS\iplot.pro
%                      $MAIN$
```

- **Compilation d'une application incluant les iTools**

La librairie iTools est une librairie objet, et la procédure RESOLVE_ALL ne sait pas résoudre le code source associé à cette librairie. Depuis la version 6.1 d'IDL, la procédure ITRESOLVE permet de résoudre l'intégralité du code situé dans le répertoire iTools.

Exemple : Création d'un fichier exécutable à partir d'une application utilisant les itools.

```
; Compilation du code principal :
.COMPILE mytool

; Compilation de la librairie iTools :
ITRESOLVE

; Création d'un fichier exécutable .SAV :
SAVE, FILE='mytool.sav', /ROUTINES, /COMPRESS
```

Remarque

Un fichier exécutable .SAV créé de cette manière fonctionnera également avec IDL Virtual Machine.

8.3. LIVRAISON D'UNE APPLICATION

La création d'une application IDL exécutable passe par plusieurs étapes :

- **Résolution des symboles**

La routine RESOLVE_ALL résout de façon itérative (en compilant) l'ensemble des routines non compilées appelées par des routines déjà compilées. Le processus s'arrête dès que l'ensemble des routines non résolues a été compilé. Si une routine non résolue ne se trouve pas dans l'un des répertoires spécifiés par la variable système !PATH, alors la routine RESOLVE_ALL produit un message d'erreur, et le processus est interrompu.

Remarque 1

La routine RESOLVE_ALL ne résout pas les procédures ou les fonctions qui sont appelées par l'intermédiaire de guillemets tel qu'avec les routines CALL_PROCEDURE, CALL_FUNCTION, et EXECUTE. De façon similaire, la routine RESOLVE_ALL ne résout pas automatiquement les routines de gestion d'évènements des IHM. C'est la raison pour laquelle on conseille d'inclure la partie statique et la partie dynamique d'une IHM dans le même fichier.

Remarque 2 : cas des applications contenant des objets graphiques

Normalement, pour créer une application exécutable avec IDL, il suffit de compiler l'ensemble des codes source de l'application, d'utiliser la routine RESOLVE_ALL, puis d'employer la routine « SAVE » pour sauvegarder la version exécutable de l'application.

Cette approche n'est pas suffisante pour des applications IDL contenant des objets (graphiques ou pas) : en effet, la routine RESOLVE_ALL ne sait pas tout résoudre, et en particulier les références à des objets.

Afin de pallier cette limitation, on peut utiliser la technique décrite dans l'exemple ci-dessous.

Exemple : Création d'un exécutable IDL à partir d'une application contenant des objets.

```
.Compile myNeatProgram     ; Compilation de la routine
; principale de l'application.
Resolve_All               ; Résolution des symboles "standard".
.Compile trackball__define ; Compilation de l'objet "trackball".
.Compile vcolorbar__define ; Compilation séparée de l'objet "vcolorbar".
Save, /Routines, File='myNeatProgram.sav' ; Exportation du code
; exécutable dans un fichier .SAV.
```

? Création d'un programme IDL exécutable

Les fichiers exécutables .SAV peuvent contenir du code ou des données, mais pas les deux simultanément. Avant la création d'un fichier exécutable .SAV, l'ensemble des routines de l'application doit être compilé et résolu.

Les avantages principaux de la création de fichiers exécutables .SAV sont les suivants :

? Seule l'utilisation de « IDL Virtual Machine » est nécessaire pour la distribution et l'utilisation ultérieure d'un fichier .SAV.
? Un fichier .SAV est la forme la plus compacte d'un code source IDL.
? Un fichier .SAV dissimule l'implémentation du code source.

La création d'un exécutable se fait en utilisant la procédure SAVE. Pour compresser le fichier exécutable produit, utiliser le mot-clé COMPRESS.

Remarque 1

La fonction IDL « EXECUTE » est désactivée avec IDL Virtual Machine. Utiliser plutôt les fonctions CALL_FUNCTION et CALL_PROCEDURE en vue d'une utilisation du programme exécutable avec « IDL Virtual Machine ».

Remarque 2

Les applications CALLABLE IDL ne fonctionnent pas avec IDL Virtual Machine.

Remarque 3

Si la routine principale de l'application porte le même nom que le fichier .SAV, alors cette routine sera exécutée automatiquement par IDL Virtual Machine en double-cliquant sur le fichier .SAV.

Restauration d'un programme exécutable

Restauration globale

La procédure «RESTORE » d'IDL permet de restaurer un fichier exécutable IDL dans son intégralité.

Restauration sélective

Depuis la version 6.1 d'IDL, l'objet IDL_SaveFile permet de restaurer le contenu d'un fichier exécutable IDL de façon sélective. En utilisant cet objet, on peut obtenir des informations concernant :

- Le créateur du fichier exécutable.
- La machine sur laquelle a été créé le fichier exécutable.
- Le système d'exploitation sur lequel a été créé le fichier exécutable.
- Le nombre d'éléments du fichier exécutable.
- Les types d'éléments du fichier exécutable (variables, blocs communs, routines, …).

Exemple : Restauration sélective du contenu d'un fichier exécutable .SAV.

```
; Sélection du fichier .SAV à restaurer :
savefile = FILEPATH('cduskcD1400.sav', $
   SUBDIRECTORY=['examples', 'data'])

; Création d'un objet IDL_SaveFile :
sObj = OBJ_NEW('IDL_Savefile', savefile)

; Interrogation du fichier IDL_SaveFile afin de déterminer le nombre de
variables régulières à l'intérieure :
sContents = sObj->CONTENTS()
PRINT, sContents.N_VAR

; IDL affiche :
; 3

; Récupération des noms des variables du fichier .SAV :
sNames = sObj->NAMES()
PRINT, sNames

; IDL affiche :
; DENSITY MASSFLUX VELOCITY

; Détermination de la taille de la variable DENSITY :
sDensitySize = sObj->SIZE('density')
PRINT, sDensitySize

; IDL affiche :
; 3          30          30          15           1       13500

; Restauration de la variable DENSITY :
sObj->RESTORE, 'density'
```

```
; Visualisation de la variable dans iVolume :
IVOLUME, density
```

9. SYNTHESE

9.1. TABLE RECAPITULATIVE DES REGLES

Les règles sont récapitulées ici, classées par ordre alphabétique.

Id. Règle	Intitulé	Page
Don.Bornage	On doit utiliser les signes "<" et ">" pour borner un tableau de préférence à la boucle "for" suivie d'un test et d'une affectation.	15
Don.Bornage	Les codes source d'une application doivent toujours être regroupés sous forme d'un projet.	11
Don.CommonInit	Lorsqu'un `COMMON` est utilisé pour passer des variables d'une routine à une autre, il doit toujours être initialisé par l'appelant.	20
Don.CommonReprod	Les `COMMONs` doivent être reproduits identiquement (nom et nombre de variables identiques) dans toutes les routines qui les référencent.	20
Don.EviterDuplication	On doit éviter la duplication de la mémoire lors de l'allocation d'un nouveau pointeur.	17
Don.Héritage	On doit employer la technique d'héritage dès que possible.	13
Don.InterfaceMin	Chaque classe doit posséder au minimum 3 méthodes : définition de la classe, constructeur, destructeur.	13
Don.Modification	On doit ne changer ni la taille ni le type d'une variable.	19
Don.StuctCacher	On doit utiliser la procédure `STRUCT_HIDE` pour dissimuler des structures.	15
Don.TableauCreation	Un tableau doit être créé en utilisant le mot-clé `NOZERO` dans les fonctions de création de tableaux.	15
Don.TailleIndét	On doit utiliser des pointeurs pour stocker des données de taille indéterminée.	16
Don.Typage	IDL ne permettant pas de déclaration de type, il est nécessaire de signaler en début de routine la liste des variables utilisées, avec leur mode d'utilisation.	17
Don.VarSystème	On doit utiliser une variable système IDL pour modifier une ressource.	19
Err.IHM	Pour gérer les erreurs dans une IHM, on doit utiliser un `CATCH` conjointement à la routine HELP, /LAST_MESSAGE.	40
Id.MotsClés	On ne doit pas abréger le nom des mots-clé.	12
Id.Natif	Les noms des variables doivent être différents des noms des routines natives IDL.	12
Id.Widget	On doit utiliser une convention de nommage pour les widgets de l'IHM.	12
Int.AccelGraphInst	On doit utiliser l'instanciation pour accélérer la visualisation des objets graphiques.	49
Int.AccelGraphique	On doit utiliser le mot-clé `RENDERER` pour accélérer la visualisation des objets graphiques.	49
Int.AccelGraphRetain	On doit utiliser le mot-clé `RETAIN` de la classe `IDLGRWINDOW` pour rafraîchir efficacement les fenêtres.	49
Int.AccesRapide	On doit utiliser la routine ASSOC pour accéder rapidement aux éléments d'un fichier contenant des structures répétitives.	42
Int.CompoundWidgetCreat	On doit créer des compound widget dès que possible.	47
Int.CompoundWidgetUse	On doit utiliser les « compound widgets » dès que possible.	46
Int.DynamiqueIHM	On doit regrouper la partie dynamique associée à un panneau dans un même module.	43
Int.EtatIHM	On doit utiliser une structure d'état dans une valeur utilisateur pour mémoriser l'état d'une application.	44
Int.Fichiers	On doit utiliser les fonctions IDL de manipulation de fichiers.	41

Id. Règle	Intitulé	Page
Tr.Recursivite	On doit utiliser le mécanisme `FORWARD_FUNCTION` pour la création de fonctions récursives.	39
Tr.ScalaireTableau	Lorsqu'une expression fait intervenir des scalaires et des tableaux, on doit effectuer les opérations scalaires avant les opérations sur les tableaux.	34
Tr.StructAffect	On doit utiliser la procédure `STRUCT_ASSIGN` pour affecter à une structure une structure de définition différente.	28
Tr.TableauConcaténation	On doit éviter la technique de concaténation pour augmenter la taille d'un tableau.	27
Tr.TableauDupplication	On doit éviter la duplication inutile de la mémoire en utilisant la fonction `TEMPORARY`.	25
Tr.TableauInit	On doit utiliser la fonction `REPLICATE_INPLACE` pour initialiser rapidement un tableau à une valeur donnée.	26
Tr.TableauMultiple	On doit éviter la multiplication entre tableaux de types différents.	25
Tr.TableauPointeur	Le déréférencement d'un tableau de pointeurs doit s'effectuer en déréférençant chacun des pointeurs du tableau.	29
Tr.TestMotsClés	On doit vérifier systématiquement la présence des mots-clé.	33
Tr.Vectorisation	On doit éliminer les boucles par vectorisation.	38

9.2. TRAÇABILITE VIS A VIS DU DOCUMENT « COMMUN »

Cette table a été créée lors de la mise en commun des règles entre langages. Elle donne pour chaque règle :

? L'ancienne identification de la règle (Version **1**)

? La nature de la modification : Aucune, Renommée (on a changé l'identification de la règle), Communalisé (la règle a été déplacée vers les document commun), Supprimée.

? La nouvelle identification dans le cas où la règle n'a pas été supprimée, une raison de la suppression sinon.

Identification Version **1**	Nature	Nouvelle identification
BlocCommun.EVITER	Commonalisée	Don.Localite
BlocCommun.INITIALISATION	Renommée	Don.CommonInit
BlocCommun.NOMMAGE	Commonalisée	Id.IdentRegle;Id.ClasseType
BlocCommun.PARAMETRE-POSITIONNEL-1	Commonalisée	Don.Localite
BlocCommun.PARAMETRE-POSITIONNEL-2	Renommée	Tr.CommonParam
BlocCommun.PARTAGE	Commonalisée	Org.Couplage
BlocCommun.REPRODUCTION	Renommée	Don.CommonReprod
CodeSource.PROJET	Renommée	Org.Projet
Constante.DEFINITION	Commonalisée	Don.Invariant;Don.Enumeration
Constante.PRESERVATION	Commonalisée	Tr.ModifConst
E_S.ACCESRAPIDE	Renommée	Int.AccesRapide
E_S.FERMETURE	Commonalisée	Int.FichierFermeture
E_S.MANIPULATION	Renommée	Int.Fichiers
E_S.UNITELOGIQUE	Renommée	Int.UniteLogique
E_S.XDR	Renommée	Int.Portable
Erreurs.CATCH	Commonalisée	Err.Mecanisme
Erreurs.ON_ERROR-ON_IOERROR	Commonalisée	Err.Mecanisme
Expression.CODAGE-MATRICIEL	Renommée	Tr.CalculMatriciel
Expression.FACTORISATION	Commonalisée	Qa.Factorisation

Identification Version **1**	Nature	Nouvelle identification
Expression.IDENTITE	Commonalisée	Qa.Algèbre
Expression.INVARIANT	Commonalisée	Don.Invariant
Expression.ORDONNANCEMENT-OPTIMAL	Renommée	Tr.ScalaireTableau
Expression.PARENTHESAGE	Commonalisée	Tr.Parenthèses
Expression.PRESENTATION	Commonalisée	Pr.Instruction
Expression.REGROUPEMENT	Commonalisée	Qa.Correlation
IHM.COMPOUNDWIDGETS-CREATION	Renommée	Int.CompoundWidgetCreat
IHM.COMPOUNDWIDGETS-UTILISATION	Renommée	Int.CompoundWidgetUse
IHM.CONVENTION-NOMMAGE	Renommée	Id.Widget
IHM.DYNAMIQUE	Renommée	Int.DynamiqueIHM
IHM.ETAT	Renommée	Int.EtatIHM
IHM.GESTION-ERREURS	Renommée	Err.IHM
IHM.GUIBUILDER	Renommée	Int.GUIBuilder
IHM.OBJET	Renommée	Int.ObjetIHM
IHM.POSITIONNEMENT-RELATIF	Renommée	Int.Positionnement
IHM.REGROUPEMENT-WIDGETS	Renommée	Int.Regrouprement
Instruction.CASE/SWITCH-CLASSIFICATION	Commonalisée	Qa.Branches
Instruction.CASE/SWITCH-UTILISATION	Renommée	Tr.CaseSwitch
Instruction.CAS-ELSE	Renommée	Tr.CasElse
Instruction.COMMENTAIRE	Commonalisée	Pr.CommFonc
Instruction.FIN-APPROPRIEE	Renommée	Tr.End
Instruction.FOR-BREAK	Commonalisée	Tr.BoucleSortie
Instruction.FOR-CONSERVATION	Commonalisée	Tr.ModifCompteur
Instruction.FOR-FUSION	Renommée	Tr.FusionBoucles
Instruction.FOR-VECTORISATION	Renommée	Tr.Vectorisation
Instruction.GOTO	Commonalisée	Tr.Goto
Instruction.IF-LIMITATION	Renommée	Tr.LimiteIf
MotClé.NOMMAGE	Renommée	Id.MotsClés
MotClé.TEST-PRESENCE	Renommée	Tr.TestMotsClés
Nommage.SUFFIXE	Commonalisée	Org.ModuleNom
ObjetGraphique.ACCELERATION1	Renommée	Int.AccelGraphique
ObjetGraphique.ACCELERATION2	Renommée	Int.AccelGraphInst
ObjetGraphique.ACCELERATION3	Renommée	Int.AccelGraphRetain
ObjetGraphique.PARTAGE-DONNEES	Renommée	Int.PartageDonnées
Objets.HERITAGE	Renommée	Don.Héritage
Objets.INTERFACE-MINIMALE	Renommée	Don.InterfaceMin
ParamètrePositionnel.LIMITER-NOMBRE	Renommée	Tr.ParamNombre
ParamètrePositionnel.NATURE	Commonalisée	Tr.ModifParSortie
ParamètrePositionnel.NOMBRE	Renommée	Tr.ParamRespect
Pointeur.DEREFERENCEMENT1	Renommée	Tr.TableauPointeur
Pointeur.DEREFERENCEMENT2	Renommée	Tr.PointeurPointeur
Pointeur.DUPLICATION-MEMOIRE	Renommée	Don.EviterDuplication
Pointeur.GARBAGE-COLLECTOR	Renommée	Tr.RammasseMiette
Pointeur.LIBERATION-MEMOIRE	Commonalisée	Don.AllocLiberation
Pointeur.STOCKAGE-DONNEES	Renommée	Don.TailleIndét
Présentation.FICHIER-BATCH	Commonalisée	Org.Module
Présentation.MODULE	Commonalisée	Org.Module
Présentation.ROUTINE	Commonalisée	Pr.CartStd;Org.Module
Présentation.STRUCTURES-CONTROLE	Commonalisée	Pr.Instruction;Org.Module

Identification Version **1**	Nature	Nouvelle identification
Routines.ACCES	Renommée	QA.IDL_PATH
Routines.APPEL-INDIRECT	Renommée	Tr.AppelIndirect
Routines.COMPTERENDU	Commonalisée	Qa.TestRetour;Err.Mecanisme
Routines.ITERATIVITE	Commonalisée	Tr.RecursifSol
Routines.MODIFICATION-PARAMETRE	Commonalisée	Tr.ModifVarGlobal
Routines.MULTITHREADING	Commonalisée	Dyn.OS
Routines.NOMMAGE	Commonalisée	Id.Procedure;Id.Fonction
Routines.RECURSIVITE	Renommée	Tr.Recursivite
Routines.SORTIE	Commonalisée	Tr.FonctionSortie
Routines.SYSTEME	Renommée	Tr.AppelsSysteme
Routines.UNICITE-NOM	Commonalisée	Don.Homonymie
Routines.VARIABLESHEAP	Commonalisée	Don.AllocLiberation
Structure.AFFECTATION	Renommée	Tr.StructAffect
Structure.DISSIMULATION	Renommée	Don.StuctCacher
Structure.MINIMISATION	Commonalisée	Don.Structure
Tableau. ACCES-CONDITIONNEL	Renommée	Tr.AccesConditionnel
Tableau. ACCES-RAPIDE	Renommée	Tr.AccesRapide
Tableau. ASTERISQUE	Renommée	Tr.Astérisque
Tableau. PARCOURS	Commonalisée	Don.TablePrincipe
Tableau.AUGMENTATION-TAILLE	Renommée	Tr.TableauConcaténation
Tableau.BORNAGE	Renommée	Don.Bornage
Tableau.CREATION-RAPIDE	Renommée	Don.TableauCreation
Tableau.EGALITE	Commonalisée	Don.TableOper
Tableau.EVITER-DUPLICATION	Renommée	Tr.TableauDupplication
Tableau.INDEXATION	Renommée	Tr.Indexation
Tableau.INITIALISATION-RAPIDE	Renommée	Tr.TableauInit
Tableau.LIBERATION-MEMOIRE	Renommée	Tr.Libération
Tableau.MATRICE-CREUSE	Renommée	Tr.MatriceCreuse
Tableau.MATRIX_MULTIPLY	Renommée	Tr.MATRIX_MULTIPLY
Tableau.MULTIPLICATION	Renommée	Tr.TableauMultiple
Tableau.MULT-MATRICE#	Renommée	Tr.MatriceMultiple
Tableau.MULT-MATRICE##	Renommée	Tr.MatriceMultiple
Tableau.OPERATEURS-COMPOSES	Renommée	Tr.OperComposés
Variable.ACCES-ENDEHORS-PORTEE-ROUTINE	Renommée	Tr.AccesHorsPortee
Variable.CONVERSION	Renommée	Tr.Conversion
Variable.DECLARATION	Renommée	Don.Typage
Variable.EGALITE	Commonalisée	Tr.ComparaisonStrict
Variable.MODIFICATION	Renommée	Don.Modification
Variable.NOMMAGE1	Commonalisée	Id.VarSignif;Id.VarType
Variable.NOMMAGE2	Renommée	Id.Natif
VariableSystème.CONSERVATION	Commonalisée	Tr.ModifParSortie;Tr.ModifVarGlobal
VariableSystème.RESSOURCE	Renommée	Don.VarSystème

ANNEXE A - EXEMPLE D'ENCAPSULATION D'UNE IHM DANS UN OBJET

CODE ASSOCIE A LA CLASSE IMAGE :

```
; Destructeur :
pro image::cleanup
    ; Appel de la méthode cleanup de la classe parente :
    self->IDLGRIMAGE::CLEANUP
end

; Dimensions de l'image chargée :
function image::getDimensions
    ; Appel de la méthode GETPROPERTY de la classe parente :
    self->IDLGRIMAGE::GETPROPERTY, DIMENSIONS = dims
    return, dims
end

; Constructeur :
function image::init, filename
    ; Résultat de l'initialisation :
    out = 1
    ; Vérification des paramètres de position :
    if n_params() eq 1 then begin
       ; Test de l'existence du fichier :
       if file_test(filename) then begin
         ; Lecture image :
         result = read_image(filename)
         if result(0) ne -1 then begin
          ; Appel de la méthode init de la classe parente :
          out = self->IDLGRIMAGE::INIT(result)
          if out then $
              self.nomFichierImage = filename
         end else $
          out = 0
       end else $
         out = 0
    end else $
       out = 0
    return, out
end

; Définition de la classe :
pro image__define
    define = {IMAGE, $
       ; Héritage :
        inherits IDLGRIMAGE, $
        ; Nom de l'image :
        nomFichierImage:'' $
        }
end
```

CODE ASSOCIE A LA CLASSE IMAGEWIN :

```
; Une méthode d'un objet ne peut pas être appelée directement comme un gestionnaire
; d'évènements. C'est la raison pour laquelle la routine XMANAGER fait appel à un
; gestionnaire d'évènements nommé "widgetEvents". La routine "widgetEvents" appelle
; alors une routine spécifique nommée "handleEvents" pour traiter les
; évènements :
pro widgetEvents, event
    ; Obtention de l'objet référencant l'IHM à partir de la valeur utilisateur de
    ; la top-level base :
    widget_control, event.top, GET_UVALUE = self
    ; Appel à la méthode handleevents pour traiter les évènements :
    self->handleEvents, event
end

; Gestionnaire d'évènements :
pro imagewin::handleEvents, event
    ; Un seul évènement est généré : l'évènement produit par le bouton "quit" du
    ; menu principal :
    obj_destroy, self
end

; Routine de création des widgets :
pro imagewin::createWidgets
    ; Création de la top level base :
    self.wMainBase = widget_base(TITLE = "Encapsulation de widgets dans un objet",
    MBAR = mbar)
    ; Menu principal :
    wFileButton = widget_button(mbar, VALUE = "Fichier")
    wQuitButton = widget_button(wFileButton, VALUE = "Quitter", UVALUE = "Quitter")
    ; Création d'un widget draw en graphique objet :
    self.wMainDraw = widget_draw(self.wMainBase, XSIZE = 400, YSIZE = 400,
    GRAPHICS_LEVEL = 2)
    ; Réalisation des widgets :
    widget_control, self.wMainBase, /REALIZE
    ; Stocakge de la référence SELF dans la valeur utilisateur de wMainBase :
    widget_control, self.wMainBase, SET_UVALUE = self
    ; Appel à XMANAGER :
    xmanager, "imagewin::createWidgets", self.wMainBase, EVENT_HANDLER =
    "widgetEvents", /NO_BLOCK
end

; Routine de création des objets graphiques :
pro imagewin::createObjects
    omodel = obj_new("idlgrmodel")
    omodel->add, self
    self.oview = obj_new("idlgrview")
    self.oview->add, omodel
end

; Destructeur :
pro imagewin::cleanup
    ; Appel du destructeur de classe parente :
    self->IMAGE::CLEANUP
```

```
    ; Destruction des objets :
    if obj_valid(self.oview) then $
       obj_destroy, self.oview
    if obj_valid(self.oWin) then $
        obj_destroy, self.oWin
    ; Destruction de l'IHM :
    widget_control, self.wMainBase, /DESTROY
end

; Constructeur :
function imagewin::init, name
    ; Résultat de l'initialisation :
    out = 1
    ; Appel de la méthode INIT de la classe parente :
    out = self->IMAGE::INIT(name)
    if out then begin
       ; Création des widgets de l'IHM :
       self->createWidgets
       ; Création des objets graphiques :
       self->createObjects
       ; Dimensions de l'image chargée :
       dims = self->IMAGE::GETDIMENSIONS()
       ; Dimensionnement correct du widget draw :
       widget_control, self.wMainDraw, XSIZE = dims(0), YSIZE = dims(1)
       ; Dimensionnement correct de la vue :
       self.oview->setProperty, VIEWPLANE_RECT = [0, 0, dims(0), dims(1)]
       ; Identifiant associé au widget draw :
       widget_control, self.wMainDraw, GET_VALUE = oWin
       ; Initialisation des données membres :
        self.oWin = oWin
       ; Affichage image :
       self.oWin->draw, self.oview
    end
    return, out
end

; Définition de la classe IMAGEWIN :
pro imagewin__define
    define = {IMAGEWIN, $
    ; Référence vers le top level base :
    wMainBase: 0L, $
    ; Référence vers le widget draw :
    wMainDraw: 0L, $
    ; Référence vers les objets :
    oWin: obj_new(), oview: obj_new(), $
    ; Héritage :
    INHERITS IMAGE}
End
```

ANNEXE B - EXEMPLE DE CREATION D'UN COMPOUND WIDGET

```
; Cette routine sera utilisée par WIDGET_CONTROL pour modifier la valeur du
compound widget.
```

```
PRO tmpl_set_value, id, value
    COMPILE_OPT hidden
    ; Retour à l'appelant en cas d'erreur :
    ON_ERROR, 2
    ; Récupération de la structure d'état dans le pointeur pstate :
    stash = WIDGET_INFO(id, /CHILD)
    WIDGET_CONTROL, stash, GET_UVALUE = pstate
    ;
    ; Stockage de la valeur d'intérêt value dans le pointeur pstate : A COMPLETER
    ; (*pstate).... = value
END

; Cette routine sera utilisée par WIDGET_CONTROL pour obtenir la valeur du compound
widget.
FUNCTION tmpl_get_value, id
    COMPILE_OPT hidden
    ; Retour à l'appelant en cas d'erreur :
    ON_ERROR, 2
    ; Récupération de la structure d'état :
    stash = WIDGET_INFO(id, /CHILD)
    WIDGET_CONTROL, stash, GET_UVALUE = pstate
    ;
    ; Obtention de la valeur d'intérêt. Cette valeur est stockée dans une variable
     ; value : A COMPLETER
    ;

    ; Renvoi de la valeur :
    return, value
END

; Gestionnaire d'évènements du compound widget :
FUNCTION tmpl_event, ev
    ; Cette routine gère tous les évènements qui se produisent
    ; au sein du compound widget.
    ; Si l'évènement a besoin d'être traité à un niveau supérieur
    ; (par exemple, traité également par le programme appelant), cette
    ; routine doit renvoyer un nouvel évènement.

    ; Si l'évènement ne nécessite pas d'être traité à un niveau supérieur,
    ; elle peut par exemple renvoyer la valeur 0, ou mieux, être implémentée
    ; sous forme d'une procédure (ne pas oublier alors d'utiliser le
    ; mot-clé adéquat, EVENT_PROC ou EVENT_FUNC, dans la
    ; définition de la base principale du compound widget).

    COMPILE_OPT hidden
    parent = ev.handler
    ; Récupération de la structure d'état :
    stash = WIDGET_INFO(parent, /CHILD)
    WIDGET_CONTROL, stash, GET_UVALUE = pstate

    ;
    ; Traitement des évènements générés par le compound widget :
    ;
```

```
    ; Renvoi éventuel d'un nouvel évènement pour un traitement à un niveau
     ; supérieur.
    ; Cet évènement peut également prendre la forme d'une structure nommée.
     RETURN, { ID:parent, TOP:ev.top, HANDLER:0L }
END

; Définition du compound widget.
; Comme tout autre widget, un compound widget possède un parent, représenté ici
; par la variable parent. Il doit également être possible de définir une valeur
; utilisateur et un nom à ce compound widget (mots-clé UVALUE et
; UNAME), lors de sa création. Tout autre mot-clé, tel que le paramètre
; TAB_MODE peut également être spécifié durant la création d'un compound widget.
FUNCTION cw_tmpl, parent, $
    UVALUE = uval, $
    UNAME = uname, $
    TAB_MODE = tab_mode
    ; Vérification des paramètres positionnels :
    IF (N_PARAMS() EQ 0) THEN $
        MESSAGE, 'Must specify a parent for Cw_Tmpl'
    ; Retour à l'appelant en cas d'erreur :
    ON_ERROR, 2
    ; Valeurs par défaut des mots-clé :
    IF NOT (KEYWORD_SET(uval))  THEN $
        uval = 0
    IF NOT (KEYWORD_SET(uname))  THEN $
        uname = 'CW_TMPL_UNAME'
    ; Création d'une structure d'état pour la conservation de l'état
    ; du compound widget :
    state = { id:0 }
    ; Définition du widget base permettant d'identifier le compound
    ; widget.
    ; L'identifant de ce widget base est renvoyé à l'utilisateur, après
    ; la création du compound widget.
     ; Si ce widget base est ultérieurement
     ; caché (mot-clé MAP de WIDGET_CONTROL)
    ; ou désensibilisé (mot-clé SENSITIVE de WIDGET_CONTROL),
    ; l'ensemble des composants constituant ce compound widget le sera également.
    ; Le gestionnaire d'évènements du compound widget est également défini lors
    ; de la création du widget base principal.
    ; Les routines du compound widget qui seront appelées lors de l'utilisation
    ; de WIDGET_CONTROL, ..., GET_VALUE et WIDGET_CONTROL, ..., SET_VALUE, sont
    ; également définies lors de la création du
    ; widget base principal, par l'intermédiaire
    ; des mots-clé FUNC_GET_VALUE et PRO_SET_VALUE.
    wMainBase = WIDGET_BASE(parent, UVALUE = uval, UNAME = uname, $
    EVENT_FUNC = "tmpl_event", $
    FUNC_GET_VALUE = "tmpl_get_value", $
    PRO_SET_VALUE = "tmpl_set_value")
    if ( N_ELEMENTS(tab_mode) ne 0 ) then $
        WIDGET_CONTROL, wMainBase, TAB_MODE = tab_mode
    ; Définition des sous-composants du compound widget :
    state.id = WIDGET_LABEL(wMainBase, VALUE = "Compound Widget Template")
    ; Sauvegarde de la structure d'état du compound widget dans la valeur
    ; utilisateur du premier "descendant" de wMainBase.
```

```
    ; La valeur utilisateur de wMainBase ne peut pas être utilisée directement,
    ; car celle-ci est réservée : elle doit être effectivement possible de
    ; pouvoir associer une valeur utilisateur à un compound widget, comme à tout
    ; autre widget.
    WIDGET_CONTROL, WIDGET_INFO(wMainBase, /CHILD),
    SET_UVALUE = PTR_NEW(state,/NO_COPY)
    ; Renvoi de l'identifiant de wMainBase décrivant le compound widget:
    RETURN, wMainBase
END
```

ANNEXE C - GESTION EFFICACE DES ERREURS DANS UNE IHM AVEC IDL

CODE ASSOCIE A LA ROUTINE ERROR_MESSAGE :

```
FUNCTION ERROR_MESSAGE, theMessage, Error=error, Informational=information, $
Traceback=traceback, NoName=noname, Title=title, _Extra=extra
On_Error, 2
; Check for presence and type of message.
IF N_Elements(theMessage) EQ 0 THEN theMessage = !Error_State.Msg
s = Size(theMessage)
messageType = s[s[0]+1]
IF messageType NE 7 THEN $
Message, "The message parameter must be a string.", _Extra=extra

; Get the call stack and the calling routine's name.
Help, Calls=callStack
IF Float(!Version.Release) GE 5.2 THEN $
callingRoutine = (StrSplit(StrCompress(callStack[1])," ", /Extract))[0] ELSE $
callingRoutine = (Str_Sep(StrCompress(callStack[1])," "))[0]

; Are widgets supported?
widgetsSupported = ((!D.Flags AND 65536L) NE 0)
IF widgetsSupported THEN BEGIN

; If this is an error produced with the MESSAGE command, it is a trapped
; error and will have the name "IDL_M_USER_ERR".
IF !ERROR_STATE.NAME EQ "IDL_M_USER_ERR" THEN BEGIN
IF N_Elements(title) EQ 0 THEN title = 'Trapped Error'

; If the message has the name of the calling routine in it,
; it should be stripped out. Can you find a colon in the string?
; Is the calling routine an object method? If so, special processing
; is required. Object methods will have two colons together.
doublecolon = StrPos(theMessage, "::")
IF doublecolon NE -1 THEN BEGIN
prefix = StrMid(theMessage, 0, doublecolon+2)
submessage = StrMid(theMessage, doublecolon+2)
colon = StrPos(submessage, ":")
IF colon NE -1 THEN BEGIN
; Extract the text up to the colon. Is this the same as
; the callingRoutine? If so, strip it.
IF StrMid(theMessage, 0, colon+StrLen(prefix)) EQ callingRoutine THEN $
theMessage = StrMid(theMessage, colon+1+StrLen(prefix))
```

```
ENDIF
ENDIF ELSE BEGIN
colon = StrPos(theMessage, ":")
IF colon NE -1 THEN BEGIN
; Extract the text up to the colon. Is this the same as
; the callingRoutine? If so, strip it.
IF StrMid(theMessage, 0, colon) EQ callingRoutine THEN $
theMessage = StrMid(theMessage, colon+1)
ENDIF
ENDELSE
; Add the calling routine's name, unless NONAME is set.
IF Keyword_Set(noname) THEN BEGIN
answer = Dialog_Message(theMessage, Title=title, _Extra=extra, $
Error=error, Information=information)
ENDIF ELSE BEGIN
answer = Dialog_Message(StrUpCase(callingRoutine) + ": " + $
theMessage, Title=title, _Extra=extra, $
Error=error, Information=information)
ENDELSE
ENDIF ELSE BEGIN
; Otherwise, this is an IDL system error.
IF N_Elements(title) EQ 0 THEN title = 'System Error'
IF StrUpCase(callingRoutine) EQ "$MAIN$" THEN $
answer = Dialog_Message(theMessage, _Extra=extra, Title=title, $
Error=error, Information=information) ELSE $
IF Keyword_Set(noname) THEN BEGIN
answer = Dialog_Message(theMessage, _Extra=extra, Title=title, $
Error=error, Information=information)
ENDIF ELSE BEGIN
answer = Dialog_Message(StrUpCase(callingRoutine) + "--> " + $
theMessage, _Extra=extra, Title=title, $
Error=error, Information=information)
ENDELSE
ENDELSE
ENDIF ELSE BEGIN
Message, theMessage, /Continue, /NoPrint, /NoName, /NoPrefix, _Extra=extra
Print, '%' + callingRoutine + ': ' + theMessage
answer = 'OK'
ENDELSE
; Provide traceback information if requested.
IF Keyword_Set(traceback) THEN BEGIN
Help, /Last_Message, Output=traceback
Print,''
Print, 'Traceback Report from ' + StrUpCase(callingRoutine) + ':'
Print, ''
FOR j=0,N_Elements(traceback)-1 DO Print, "     " + traceback[j]
ENDIF
RETURN, answer
END ; ------------------------------------------------------------------------------
```

ANNEXE D - TRUCS ET TECHNIQUES DE PROGRAMATION

D1 - Technique d'extraction des champs d'une structure par l'intermédiaure d'une variable

L'extraction des champs d'une structure par l'intermédiaire d'une variable peut se faire en utilisant les fonctions WHERE et TAG_NAMES d'IDL, comme illustré dans l'exemple ci-dessous :

```
struct = {name: 'john', age:18}
field = 'age'
index = Where(Tag_Names(struct) EQ StrUpCase(field), count)
IF count NE 0 THEN $
PRINT, struct.(index[0]) ELSE Print, 'Field not found.'
```

D2 - Technique d'extraction de tous les champs d'une structure de façon récursive

Dans cet exemple, la fonction GET_TAGS permet d'extraire tous les champs d'une structure de façon récursive, et constitue en fait une extension de la fonction IDL « TAG_NAMES ». L'exemple ci-dessous peut être entièrement copié-collé puis exécuté dans IDL :

```
function All_Tags, structure, rootname
    IF N_ELEMENTS(rootname) EQ 0 THEN rootname = '.' else $
        rootname = STRUPCASE(rootname) + '.'
    names = TAG_NAMES(structure)
    retValue = PTR_NEW(rootname + names)
    FOR j = 0, N_ELEMENTS(names)-1 DO BEGIN
        ok = EXECUTE('s = SIZE(structure.' + names[j] + ')')
        IF s[s[0]+1] eq 8 THEN BEGIN
         newrootname = rootname + names[j]
         theseNames = CALL_FUNCTION('All_Tags', $
            structure.(j), newrootname)
         retValue = [[retValue],[theseNames]]
        ENDIF
    ENDFOR
    RETURN, retValue
END”
;---------------------------------------------------------------------
function Get_Tags, structure, rootname
    ON_ERROR, 1
    CASE N_PARAMS() OF
       0: BEGIN
          MESSAGE, 'Structure argument is required.'
       ENDCASE
       1: BEGIN
          rootname = ''
          s = size(structure)
          IF s[s[0]+1] ne 8 THEN $
            MESSAGE, 'Structure argument is required.'
       ENDCASE
       2: BEGIN
          s = SIZE(structure)
          IF s[s[0]+1] ne 8 THEN $
             MESSAGE, 'Structure argument is required.'
          s = SIZE(rootname)
          IF s[s[0]+1] ne 7 THEN $
             MESSAGE, 'Root Name parameter must be a STRING'
       ENDCASE
```

```
    ENDCASE
    tags = All_Tags(structure, rootname)
    retval = [*tags[0,0]]
    PTR_FREE, tags[0,0]
    s = SIZE(tags)
    FOR j=1,s[2]-1 DO BEGIN
        retval = [retval, *tags[0,j]]
        PTR_FREE, tags[0,j]
    ENDFOR
    PTR_FREE, tags
RETURN, retval
END

; Programme principal
d = {dog:'spot', cat:'fuzzy'}
c = {spots:4, animals:d}
b = {fast:c, slow:-1}
a = {cars:b, pipeds:c, others:'man'}
tags = Get_Tags(a)
s = Size(tags)
FOR j=0,s[1]-1 DO PRINT, tags[j]
END
```

D3 - Technique de concaténation de structures anonymes

La concaténation de structures anonymes de même type produit le message d ‘erreur % Conflicting data structure, *alors que la concaténation de structures nommées de même type fonctionne parfaitement. On indique ici la méthode à suivre pour la concaténation de structures anonymes.*

IDL donne aux structures anonymes des noms « invisibles ». On peut le constater grâce aux lignes de code suivantes :

```
a = {Name:'Larry', Age:46}
b = {Name:'JoeBob', Age:39}
HELP, a, b, /Structure
** Structure [f19348], 2 tags, length=12, refs=1:
NAME            STRING    'Larry'
AGE             INT              46
** Structure [f19c98], 2 tags, length=12, refs=1:
NAME            STRING    JoeBob
AGE             INT              39
```

Lorsque l’on essaie de concaténer les structures a et b, IDL produit le message d’erreur suivant : % Conflicting data structure. *Cette erreur vient du fait que la première structure possède le nom « invisible »* f19348 et que la seconde structure possède le nom « invisible » f19348. IDL refuse la concaténation car les deux entités sont effectivement différentes.

La solution pour la concaténation de structures anonymes est la suivante :

```
a = {Name:'Larry', Age:46}
b = a
b.Name = 'JoeBob'
b.Age = 39
c = [a, b]
```

Cela fonctionne car désormais, les structures a et b possèdent le même nom « invisible » :

```
HELP, a, b, /STRUCTURE
** Structure [f19e98], 2 tags, length=12, refs=3:
NAME            STRING    'Larry'
AGE             INT             46
** Structure [f19e98], 2 tags, length=12, refs=3:
NAME            STRING    'JoeBob'
AGE             INT             39
```

D4 - Technique d'affichage de splash screen pendant la période d'initialisation d'une IHM

Souvent, l'affichage d'un « splash screen » est souhaité durant la période d'initialisation plus ou moins longue qui précède le démarrage effectif d'une application. On donne ci-dessous la technique à utiliser pour créer un « splash screen », ainsi qu'un exemple d'utilisation. Cet exemple peut être entièrement copié-collé puis exécuté dans IDL :

```
function show_splash_screen, image, title=title, true=true, order=order
    compile_opt idl2
    on_error, 2
    true_local = n_elements(true) eq 0 ? 0 : true
    sz = size(image, /STRUCTURE)
    if (true_local eq 0 and sz.n_dimensions ne 2) then $
        message, "TRUE keyword must be set to 1, 2, 3 for 24-bit images"
    if (true_local ne 0 and sz.n_dimensions ne 3) then $
        message, "TRUE keyword must be set to 0 for 8-bit image"
    if n_elements(delay) eq 0 then $
        delay = 1
    xind = (true_local ne 1) ? 0 : 1
    yind = ((true_local eq 0) or (true_local eq 3)) ? 1 : 2
    device, GET_SCREEN_SIZE = screen_size
    xoffset = (screen_size[0] - sz.dimensions[xind])/2
    yoffset = (screen_size[1] - sz.dimensions[yind])/2
    wMainBase = widget_base(TLB_FRAME_ATTR = 4, /COLUMN, TITLE = title, $
    XPAD = 0, YPAD = 0, XOFFSET = xoffset, YOFFSET = yoffset)
     wMainDraw = widget_draw(wMainBase, XSIZE = sz.dimensions[xind], YSIZE =
     sz.dimensions[yind])
    widget_control, wMainBase, /REALIZE
    widget_control, wMainDraw, GET_VALUE = win_id
    wset, win_id
    tv, image, TRUE = true_local, ORDER = keyword_set(order)
    return, tlb
end

pro testSplash
     ; Affichage du « splash screen » :
     filename = FILEPATH('rsi.png', SUBDIR = 'training')
     image = READ_IMAGE(filename)
     id = show_splash_screen(image, true = 1)
     ; Initialisation de l'application :
     res = initApplication()
     ; Destruction du « splash screen » après initialisation :
     WIDGET_CONTROL, id,  /DESTROY
End
```

D7 - Technique de création de barres de menu spécifiques à une application

La création de barres de menu spécifiques permet d'améliorer grandement l'ergonomie d'une application. De plus, comme illustré par l'exemple ci-dessous, la création d'un menu spécifique avec plusieurs niveaux hiérarchiques est très simple à implémenter avec IDL. Cette création se fait par l'intermédiaire du mot-clé MBAR dans la fonction de création WIDGET_BASE().

```
pro testMBAR
     ; Création de la base principale :
     wMainBase = WIDGET_BASE(/ROW, MBAR = mbar, XSIZE = 200, YSIZE = 200)
     ; Création d'un menu principal "Fichier" :
     wFichierButton = WIDGET_BUTTON(mbar, VALUE = "Fichier")
     ; Les 2 options "Ouvrir" et "Quitter" appartiennent au menu principal "Fichier"
     wOuvrirButton = WIDGET_BUTTON(wFichierButton, VALUE = "Ouvrir")
     wQuitterButton = WIDGET_BUTTON(wFichierButton, VALUE = "Quitter")
     ; Création d'un menu principal "Options" :
     wOptionsButton = WIDGET_BUTTON(mbar, VALUE = "Options")
     ; L'option "Options 1" est une option de niveau hiérarchique 1 :
     wOptions1Button = WIDGET_BUTTON(wOptionsButton, VALUE = "Options 1", /MENU)
     ; Toutes les options suivantes sont des options de niveau hiérarchique 3 :
     wOptions11Button = WIDGET_BUTTON(wOptions1Button, VALUE = "Options 11")
     wOptions12Button = WIDGET_BUTTON(wOptions1Button, VALUE = "Options 12")
     wOptions13Button = WIDGET_BUTTON(wOptions1Button, VALUE = "Options 13")
     WIDGET_CONTROL, wMainBase, /REALIZE
End
```

D5 - Technique d'intégration d'une fonctionnalité de navigation entre les widgets d'une IHM

IDL autorise désormais la navigation entre les différents widgets d'une IHM, en utilisant la touche « Tab ». Cette fonctionnalité permet à un utilisateur de se déplacer rapidement entre les widgets de l'IHM, sans avoir à utiliser la souris. Cette fonctionnalité peut être activée en utilisant le mot-clé TAB_MODE dans les fonctions de création suivantes :

```
widget_base, widget_button, widget_combobox, widget_droplist, widget_list,
widget_slider, widget_tab, widget_table, widget_text et widget_tree.
```

L'ordre de navigation entre les différents widgets s'effectue en respectant la hiérarchie définie durant la création de l'IHM.

Se référer à la rubrique « Enhancing Widget Application Usability » de l'aide en ligne d'IDL pour obtenir de plus amples informations sur cette nouvelle fonctionnalité.

D6 - Technique d'intégration d'accélérateurs dans une IHM

Les accélérateurs clavier permettent à l'utilisateur d'activer certaines fonctionnalités d'une IHM sans avoir à utiliser la souris. Des accélérateurs peuvent être définis pour des éléments de menu et différents types de WIDGET_BUTTON sous Windows. Cependant, UNIX ne supporte les accélérateurs clavier que pour des éléments de type menu. L'association d'un accélérateur clavier pour un WIDGET_BUTTON se fait par l'intermédiaire du mot-clé ACCELERATOR. Un exemple est donné ci-dessous :

```
bRun = WIDGET_BUTTON( base, VALUE = "Run", ACCELERATOR= "F5" )
bPause = WIDGET_BUTTON( ( base, VALUE = "Pause", $
   ACCELERATOR = "Ctrl+F5" )
bResume = WIDGET_BUTTON ( base, VALUE = "Resume", $
   ACCELERATOR = "Ctrl+Shift+F5" )
```

D8 - Technique d'extraction des données membres d'un objet, en utilisant un nom de champ

Pour accéder à la valeur d'une donnée membre d'un objet, le développeur doit créer une méthode spécifique qui renvoie la valeur de la donnée membre. On propose ici une technique permettant d'accéder à toutes les données membre d'un objet sans avoir à écrire de méthodes spécifiques. Cette technique peut s'avérer particulièrement intéressante pour connaître la définition précise de toutes les classes graphiques d'IDL.

Exemple 1 : Détermination rapide de toutes les données membres d'une classe :

```
IDL> test = {IDLGRIMAGE}
IDL> HELP, test, /FULL, /STRUCTURE
```

IDL affiche :

```
** Structure IDLGRIMAGE, 47 tags, length=400, data length=372:
   IDLITCOMPONENT_TOP
                  LONG64                          0
   IDLITCOMPONENTVERSION
                  INT              0
   DESCRIPTION    STRING    ''
   NAME           STRING    ''
   ICON           STRING    ''
   IDENTIFIER     STRING    ''
   HELP           STRING    ''
   UVALUE         POINTER   <NullPointer>
   _PARENT        OBJREF    <NullObject>
   PROPERTYDESCRIPTORS
                  OBJREF    <NullObject>
   _FLAGS         LONG                 0
   IDLITCOMPONENT_BOTTOM
                  LONG64                          0
   IDLGRCOMPONENT_TOP
                  LONG64                          0
   IDLGRCOMPONENTVERSION
                  INT              0
   HIDE           LONG                 0
   PARENT         OBJREF    <NullObject>
   IDLGRCOMPONENT_BOTTOM
                  LONG64                          0
   IDLGRGRAPHIC_TOP
                  LONG64                          0
   IDLGRGRAPHICVERSION
                  INT              0
   ALPHACHANNEL   FLOAT          0.000000
   CLIP_PLANES    POINTER   <NullPointer>
   COLOR          POINTER   <NullPointer>
   DEPTH_TEST_DISABLE
                  LONG                 0
   DEPTH_TEST_FUNCTION
                  LONG                 0
   DEPTH_WRITE_DISABLE
                  LONG                 0
   GRAPHICFLAGS   LONG                 0
   PALETTE        OBJREF    <NullObject>
   XCOORD_CONV    DOUBLE    Array[2]
   YCOORD_CONV    DOUBLE    Array[2]
```

```
   ZCOORD_CONV     DOUBLE    Array[2]
   XRANGE          DOUBLE    Array[2]
   YRANGE          DOUBLE    Array[2]
   ZRANGE          DOUBLE    Array[2]
   GRAPHIC_DATA_OBJECT
                   POINTER   <NullPointer>
   IDLGRGRAPHIC_BOTTOM
                   LONG64                        0
   IDLGRIMAGE_TOP  LONG64                        0
   IDLGRIMAGEVERSION
                   INT             0
   CHANNEL         LONG                  0
   DATA            POINTER   <NullPointer>
   DIMENSIONS      DOUBLE    Array[2]
   SUB_RECT        FLOAT     Array[4]
   IMAGEFLAGS      LONG                  0
   LOCATION        DOUBLE    Array[3]
   INTERLEAVE      LONG                  0
   INTERPOLATE     LONG                  0
   BLEND_FUNCTIONS LONG      Array[2]
   IDLGRIMAGE_BOTTOM
                   LONG64                        0
```

Exemple 2 : Routine d'extraction d'une donnée membre d'un objet en utilisant un nom de champ.

```
FUNCTION OBJECT::EXTRACT, field
   ; On s'assure d'abord que la variable field est du type string :
   s = Size(field)
   IF s[s[0]+1] NE 7 THEN Message, "Field variable must be a sting."
   ; Obtention du nom de la classe :
   thisClass = Obj_Class(self)
   ; Création d'une structure locale de ce type :
   ok = Execute('thisStruct = {' + thisClass + '}')
   ; Extraction des noms de champs de la structure locale :
   structFields = Tag_Names(thisStruct)
   index = WHERE(structFields EQ StrUpCase(field), count)
   ; Renvoide la valeur du champ d'intérêt, si ce champ est trouvé :
   IF count EQ 1 THEN BEGIN
      RETURN, self.(index[0])
   ENDIF ELSE BEGIN
      Message, 'Can not find field "' + field + $
          '" in structure.', /Informational
      RETURN, -1
   ENDELSE
END
```

Cette méthode doit être ajoutée au code source de la classe d'intérêt, en remplaçant OBJET par le nom de la classe d'intérêt. En reprenant la classe IMAGE définie au point 11.1.3, on ajouterait à cette classe un méthode IMAGE ::EXTRACT. Cette nouvelle méthode pourrait alors être utilisée de la manière suivante :

```
oImage = obj_new("IMAGE", "C:\RSI\IDL61\EXAMPLES\DATA\people.jpg")
pImage = oImage->extract("DATA")
help, *pimage
<PtrHeapVar25>  BYTE      = Array[3, 256, 256]
```

D9 - Utilisation de la classe IDLGRCLIPBOARD pour transfert dans le clipboard de Windows

Il peut être parfois intéressant de pouvoir copier le contenu de fenêtres graphiques IDL dans le clipboard de Windows, pour pouvoir le réutiliser dans des documents Latex, Word ou PowerPoint. La classe « IDLGRCLIPBOARD » implémente cette fonctionnalité, dont un exemple est donné ci-dessous :

```
pro view_contour
    compile_opt idl2
    ; Création d'un atome graphique de type contour :
    oContour = OBJ_NEW("IDLGRCONTOUR", DIST(30), N_LEVELS = 10,
     /PLANAR, GEOMZ = 0, $
    COLOR = [0, 0, 255])
    ; Création d'un modèle :
    oModel = OBJ_NEW("IDLGRMODEL")
    ; Création d'une vue :
    oView = OBJ_NEW("IDLGRVIEW", VIEWPLANE_RECT = [-5, -5, 40, 40])
    ; Création de la hiérarchie :
    oView->ADD, oModel
    oModel->ADD, oContour
    ; Création de l'objet destination :
    oWindow = OBJ_NEW("IDLGRWINDOW", RETAIN = 2)
    ; Visualisation de la hiérarchie :
    oWindow->DRAW, oView
    ; Transfert des données vers le clipboard de Windows :
    oClipBoard = OBJ_NEW("IDLGRCLIPBOARD")
    oClipBoard->DRAW, oView
end
```

Après exécution de cet exemple, il est possible d'ouvrir une application Windows quelconque puis d'effectuer un copier coller pour récupérer le contenu de la fenêtre IDL en question.

ANNEXE E - REFERENCES DE SITES INTERNET

WWW.RSINC.COM

WWW.DFANNING.COM

www.mpimet.mpg.de/en/misc/software/idl/html/lib_index.php

REFERENTIEL NORMATIF REALISE PAR :

Centre National d'Etudes Spatiales

Inspection Générale Direction de la Fonction Qualité

18 Avenue Edouard Belin

31401 TOULOUSE CEDEX 9

Tél. : 05 61 27 31 31 - Fax : 05 61 28 28 49

CENTRE NATIONAL D'ETUDES SPATIALES

Siège social : 2 pl. Maurice Quentin 75039 Paris cedex 01 / Tel. (33) 01 44 76 75 00 / Fax : 01 44 46 76 76
RCS Paris B 775 665 912 / Siret : 775 665 912 00082 / Code APE 731Z