CENTRE NATIONAL D'ÉTUDES SPATIALES

REFERENTIEL NORMATIF du CNES

RNC

Référence: RNC-CNES-Q-HB-80-501
Version 4
17 Septembre 2009

MANUEL

ASSURANCE PRODUIT

REGLES COMMUNES POUR L'UTILISATION DES LANGAGES DE PROGRAMMATION

ACCORD du Bureau de Normalisation	**BN n° 39 du 25/02/08 – BN n°44 du 08/09/08** **BN n° 54 du 16/09/09**
APPROBATION **Président du CDN** **Alain CUQUEL**	

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : **REGLES COMMUNES POUR L'UTILISATION DES LANGAGES DE PROGRAMMATION**	
MOTS CLES : Règle Commune Générique Langage Programmation	
NORME EQUIVALENTE : Néant	
OBSERVATIONS : Néant	
RESUME : Ce document présente les règles communes pour l'utilisation des langages de programmation.	
SITUATION DU DOCUMENT : Ce document fait partie de la collection des Manuels approuvés du Référentiel Normatif du CNES. Ce document est affilié au document "RNC-ECSS-ST-Q-80 Software Product Assurance".	
NOMBRE DE PAGES : 83	**Langue** : Française
PROGICIELS UTILISES / VERSION : Word 2002	
SERVICE GESTIONNAIRE : Inspection Générale Direction de la Fonction qualité (IGQ)	
AUTEUR(S) : **DATE : 17/09/09**	
J-C DAMERY	

PAGES DES MODIFICATIONS

VERSION	DATE	PAGES MODIFIEES	OBSERVATIONS
1	10/12/2006	Création	Création, avec le support de T. Leydier (Virtualité Réelle) 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.
2	10/04/2008	Page 74 § 8.1	Suite à la FEB 77/2008 acceptée au BN n° 39 du 25/02/2008, correction d'une erreur mineure dans la table récapitulative des règles.
3	02/06/2008	Toutes	Changement de nomenclature suite à la phase de benchmarking ECSS (ancienne référence « RNC-CNES-Q-80-501 »)
4	17/09/2009		Suite à la FEB 91/2009 acceptée au BN n° 54 du 16/09/09 introduisant dans le RNC le nouveau manuel RNC-CNES-Q-HB-80-535, l'outil de tailorisation présent dans le document RNC-CNES-Q-HB-80-501 est mis à jour.

TABLE DES MATIERES

1. INTRODUCTION

Le document « Règles communes pour l'utilisation des langages de programmation » est rattaché au document « RNC-ECSS-Q-ST-80 Software Product Assurance ». Il décrit les règles applicables aux langages de programmation utilisés au CNES.

2. OBJET

Le but de ce document est d'établir les règles communes aux langages de programmation. Ces règles ont été établies à partir de « l'état de l'art » et du « retour d'expérience » accumulé sur les projets. Ce document est indispensable lors de l'utilisation d'un langage de programmation dans un projet du CNES. Il est complété par des documents spécifiques à chaque langage.

3. DOMAINE D'APPLICATION

Ce document est applicable à tous les projets CNES.

Il doit être adapté et/ou complété par le responsable projet et/ou l'ingénieur qualité pour l'organisation du code, la nomenclature des identificateurs et éventuellement d'autres règles spécifiques en fonction des objectifs qualité fixés qui peuvent être déclinés à partir du document DR1.

Il n'est jamais utilisé seul, mais combiné avec un document Langage ; par exemple, sur un projet JAVA, on va combiner les règles communes et les règles décrites dans le Manuel JAVA. La combinaison ainsi qu'une sélection des règles seront réalisées en début de projet à l'aide de l'outil de tailorisation.

L'outil de tailorisation est une interface qui permet de sélectionner pour un projet donné, en fonction de critères propres au projet (maintenabilité, criticité, effort de test), les règles communes et « langage » appropriées au projet. Il est inclus dans ce document ; on peut ainsi l'activer en cliquant sur le bouton ci-dessous :

Lancer l'outil de tailorisation

Remarque : Les Manuels langages utilisées par l'outil de tailorisation doivent être dans le répertoire courant.

4. DOCUMENTS

4.1. DOCUMENTS DE REFERENCE

DR	Identification	Titre
(DR1)	RNC-ECSS-Q-ST-80	Software Product Assurance

4.2. DOCUMENTS APPLICABLES

DA	Identification	Titre
Néant		

5. TERMINOLOGIE

5.1. GLOSSAIRE

Terme	Définition
Bibliothèque	Ensemble de fonctions ou procédures ayant généralement un thème commun (en anglais library).
Fonction	Opération qui renvoie un résultat. Une fonction ne modifie généralement pas la valeur de ses paramètres.
Module	Unité de programmation qui regroupe des données et des opérations. Les modules sont généralement associés à un fichier de code.
Opération	Unité de traitement (procédure ou fonction du logiciel réalisant un traitement).
Portée	Zone de programmation dans laquelle une donnée est utilisable : lorsque la portée est locale, une donnée n'est utilisable que localement, par exemple dans une fonction ; lorsqu'elle est globale, la donnée est utilisable n'importe où dans le code.
Procédure	Opération qui ne renvoie pas de résultat et qui regroupe des instructions. Contrairement à la fonction la procédure peut modifier la valeur de ses paramètres.
Programme principal	Opération par laquelle débute l'exécution du programme.
Sous-programme	Fonction ou sous-programme.
Tâche	Flot de contrôle géré au niveau du système d'exploitation. Synonyme de processus.
Tailorisation	Sélection des règles applicables au projet avec éventuellement adaptation voire ajout de nouvelles règles (traduction du terme anglais "tailoring" - action de tailler, élagage)
Thread	Flot de contrôle plus léger que la tâche, géré au niveau du langage. Par léger on entend notamment : - espace mémoire partagé entre les threads d'un processus. - changement de contexte d'une thread à une autre plus rapide que le changement de contexte d'un processus à un autre.

5.2. ABREVIATIONS

5.2.1. Codification des règles

Chaque règle est présentée sous forme de tableau contenant 4 informations :

<identification>	<intitulé de la règle>
<Tailorisation>	
<Type de Projet>	

Ce tableau comporte les champs suivants :

- Identification de la règle sous la forme <sous-chapitre>.<code de règle> ; le sous-chapitre est standard et codé à l'aide d'un mnémonique standard. On utilise la codification suivante pour les sous-chapitres :
  - Org : Organisation du code
  - Pr : Présentation du code
  - Id : Identificateurs
  - Don : Données
  - Tr : Traitements
  - Err : Gestion des erreurs
  - Dyn : Dynamique
  - Int : Interfaces
  - QA : Qualité
  - AR : Autres règles
- Intitulé de la règle : il s'agit du libellé de la règle.
- Tailorisation : il s'agit d'un champ permettant de définir 4 paramètres quantitatifs de tailorisation. Ces paramètres sont identifiés par une lettre et présentés comme suit :
  - M=<m>;F=<f>;P=<p>;V=<v>
  - Avec
    - m : note de maintenabilité de 0 à 3 (0 = la règle a une faible influence sur la maintenabilité, 3 = la règle a une forte influence sur la maintenabilité)
    - f : note de fiabilité de 0 à 3 (0 = la règle a une faible influence sur la fiabilité, 3 = la règle a une forte influence sur la fiabilité)
    - p : priorité relative de 1 à 200 ; cette information permet de classer les règles (1 = la règle la plus importante, 200 la moins importante)
    - v : note de vérifiabilité de 0 à 2 (2 = règle vérifiable facilement (généralement de manière automatique au moyen d'un analyseur), 1 = règle vérifiable avec difficulté (il est généralement possible d'apprécier au moins partiellement le respect de la règle en combinant actions manuelles et résultats d'analyseurs), 0 = règle non vérifiable).

    Ces notes sont issues du retour d'expérience des auteurs de ce document. Elles pourront évoluer en fonction de prochains retours d'expérience.
- Type de projet : c'est un autre paramètre de tailorisation ; il définit la catégorie de projet concernée par la règle. Il est à choisir parmi les valeurs suivantes : Bord, Sol, Quelconque.

La règle est complétée par 3 paragraphes obligatoires :

- La description,
- La justification,
- Les exemples.

Exemple de règle :

id.NomDonnee	Le nom d'une donnée doit être un nom commun emprunté au langage courant ; il doit être au pluriel si la donnée est un ensemble ou un groupe.
M=3;F=0;P=43;V=0	
Quelconque	

Description

Sans Objet.

Justification

Améliore la lisibilité.

Exemple

En C++

```
LeCapteurStellaire
LesResultats
LesJoursDeLaSemaine
```

5.2.2. Autres abréviations ou acronymes

Terme	Définition
RNC	Référentiel Normatif du CNES

6. ADEQUATION EVENTUELLE AUX NORMES ET STANDARDS

Sans Objet

7. REGLES

7.1. CONCEPTION / ORGANISATION DU CODE

Org.DonneesOper	Les données et les opérations doivent être regroupées en modules afin de former des paquets cohérents, en utilisant les moyens disponibles au niveau du langage
M=3;F=0;P=41;V=0	
Quelconque	

Description

Cette règle concerne tous les moyens et tous niveaux conceptuels proposés par le langage utilisé, en ce qui concerne la modularité.

Justification

Renforce la primauté et l'antériorité de l'activité de conception sur celle du codage.
Assure la cohérence entre la conception et le code.

Exemple

En SHELL

Cette règle concerne les scripts.

En ADA

Cette règle concerne les unités, les paquetages et les bibliothèques.

En C++

Cette règle concerne les classes, fichiers et namespaces.

En JAVA

Cette règle concerne les classes, les fichiers et les paquetages.

En FORTRAN

Exemple de regroupement de données assurant la gestion d'une vanne :

```
module VANNES
   ! ========= definition d'une vanne =================
   integer, parameter :: EtatOuvert = 1, &
                         EtatFerme  = 2, &
                         EtatTransitoire = 3
   type VANNE
      integer :: ident
      integer :: etat
      real(DOUBLE) :: debit
      integer :: amont, aval
   end type VANNE
   ! ========= table globale des vannes, par ident ====
   integer, parameter :: NbMaxVannes = 1000
   type(VANNE), dimension(NbMaxVannes) :: TABLES_VANNES
   ! ========= fichier des vannes =====================
   character(LEN=*), parameter :: FichierVannes = 'vannes.dat'
   integer, parameter :: CanalVannes = 17
   ...
end module VANNES
```

Org.ModuleNom	Le nom d'un module doit véhiculer l'unité conceptuelle que le module représente
M=1;F=0;P=105;V=0	
Quelconque	

Description

Cette règle concernant toutes les sortes de modules envisageables, selon le langage concerné. Elle concerne également les fichiers associés, leur localisation, leur nom et leur extension. Elle est une conséquence logique de la règle Org.DonneesOper .
La règle devra être adaptée aux contraintes de l'environnement de production comme : le système de gestion de fichiers, l'utilisation d'un générateur de code ou les contraintes d'un compilateur.
Des règles de correspondance entre « unités de conception » et « fichier source support » devront être également définies.

Justification

Améliore la lisibilité du source.

Exemple

En ADA

Le nom des fichiers reprend le nom de l'unité de compilation Ada qu'il contient.
Dans le cas où il s'agit d'une unité séparée, le nom du fichier est préfixé par celui de l'unité "mère",
Un nom de fichier contenant une spécification (resp. un corps) de paquetage est suffixé par _s (resp. _b) ou a pour extension .ads (resp. .adb).

En SHELL :

Les scripts porteront un nom significatif rappelant le traitement associé et une extension '.sh'.

En IDL :

Utiliser obligatoirement le suffixe ".pro" pour les fichiers source IDL.
Définir un suffixe pour les fichiers batch (par exemple ".inc").

Org.Couplage	Le couplage entre modules doit être minimisé : les liens d'utilisation entre modules doivent être unidirectionnels et inférieurs à un seuil défini par le projet.
M=3;F=1;P=32;V=1	
Quelconque	

Description

Les dépendances entre modules doivent être ordonnées et limitées. Les liens circulaires sont interdits.
Les variables externes (communes à plusieurs unités de compilation) doivent être en nombre limité. Les références entre modules effectuées à l'aide d'instructions de type « utilise » ou « inclure » doivent être ordonnées et limitées.

Justification

Un couplage trop fort complique la maintenance : les modifications apportées sur un module peuvent entraîner des modifications sur tous les modules dépendants, et au mieux entraînent une recherche des régressions sur ces modules.

Exemple

En ADA

Les clauses de contexte (clause ***with***) au niveau des spécifications d'un paquetage et/ou de son corps définissent les entités dont a besoin la spécification et/ou le corps du paquetage. Ces clauses ne doivent apparaître que lorsque cela est strictement nécessaire.

En C et C++

On évitera l'utilisation d'include globaux ;on préfèrera n'inclure que les fichiers réellement utiles.
On se donnera une limite en termes de nombre de fichiers inclus, et de niveaux d'inclusions.

En JAVA

On évitera l'utilisation d'import « génériques » (utilisant le caractère *)

En FORTRAN, PVWAWE et IDL

On limitera l'utilisation des commons.

Org.Masquage	On doit éviter les liens d'utilisation de données : on doit préférer l'utilisation d'opérations d'accès en lecture et en écriture (principe de masquage de l'information et d'encapsulation des données) dès lors que ce principe n'est pas trop pénalisant pour le langage utilisé.
M=2;F=1;P=44;V=1	
Quelconque	

Description

Les seules données accessibles directement pourront être les constantes.
On pourra déroger à la règle lorsque l'on cherche une optimisation très importante en termes de temps d'exécution : l'accès direct à une donnée membre est plus rapide qu'une fonction, en particulier lorsque le langage concerné ne supporte pas les fonctions inline.

Justification

Les références à une donnée membre sont homogènes dans tout le code utilisateur puisque la notation fonctionnelle doit être nécessairement utilisée.
Le mode d'accès à une donnée membre peut être contrôlé ce qui permet de faciliter la maintenance et la mise au point : on peut par exemple tracer toutes les mises à jour d'une donnée via sa méthode d'accès en écriture.

Exemple

En C++

On déclarera les données membres « private » et on définira des opérations d'accès :

```
Changement d'implémentation d'une classe transparent pour les
utilisateurs :
// Fichier "Personne.h"
class Personne {
public: // Acces en lecture
     const Date& dateDeNaissance();
     int age();
private:
     Date dateDeNaissance_;
     int age_; // Donnee derivee de dateDeNaissance_
};
#include "Personne.I"
// Fichier "Personne.I"
inline Personne::dateDeNaissance()      { return dateDeNaissance_; }
inline Personne::age()            { return age_; }

Une seconde implémentation est définie a posteriori afin de minimiser
l'espace mémoire occupée même si c'est au détriment des performances : la
donnée membre "age_" est supprimée et le calcul de l'âge dans la méthode
"age()" se fait via la donnée "dateDeNaissance_".
```

```
Ce changement d'implémentation est transparent pour les classes
utilisatrices car l'interface de la classe Personne reste inchangée.
```

En ADA

Les manipulations possibles des variables de paquetage doivent se faire uniquement par l'intermédiaire des primitives fournies dans la spécification de paquetages. Les variables elles-mêmes sont déclarées dans les corps des paquetages et jamais dans la spécification.

En FORTRAN 90

Seules les constantes nommées doivent avoir une visibilité PUBLIC.

Org.Module	La mise en forme de chaque module doit être standardisée pour le projet .
M=1;F=0;P=104;V=1	
Quelconque	

Description

La mise en forme concerne les modules dans leur généralité : les unités de compilation et les fichiers , la déclaration des données, la déclaration des procédures, fonctions et autre services.

Justification

Une mise en forme commune facilite la maintenabilité.

Exemple

En PVWAWE

On définira une mise en forme standard pour les services et les fichiers de commande. Par exemple, chaque service doit contenir :

- une entête :
  - le nom du service,
  - la version,
  - l'auteur,
  - la date de création,
  - la description,
  - la liste des services utilisés,
  - le mode d'appel, ainsi que la description des paramètres,
  - les COMMONs utilisés,
  - la liste des variables locales,
  - l'algorithme du service.
- le corps du service :
  - l'inclusion des fichiers
  - initialisation des paramètres de retour,
  - déclaration (initialisation) des variables locales,
  - test de présence et de validité des paramètres optionnels,
  - traitement,
  - labels d'erreurs,
  - label de fin.

En C++ :

Il est recommandé de déclarer les constructeurs et le destructeur publics en tête.

Une règle de présentation utile consiste à établir a priori des catégories de méthodes et regrouper les méthodes de l'interface d'une classe suivant ces catégories (constructeur, destructeur, accès, statut,...). Chaque catégorie est introduite par un commentaire donnant son nom derrière le mot clé

public (qui peut être répété plusieurs fois en C++). Les catégories de méthode apparaissent toujours dans le même ordre dans toutes les classes.

Org.MultiLang	Lorsque plusieurs langages de programmation sont utilisés dans un même projet, on doit définir des règles de correspondance pour les éléments échangés entre les langages.
M=1;F=1;P=58;V=0	
Quelconque	

Description

Il est souhaitable lorsque c'est possible d'utiliser les mêmes identificateurs, dans chaque langage. Il est souhaitable de respecter aussi la casse (majuscules/minuscules). Corollairement lorsque deux langages voisins sont mélangés et que le mélange peut engendrer une confusion, on cherchera à limiter le mélange et on définira une règle pour différencier les deux langages lorsqu'ils cohabitent.

Justification

Cette règle facilite la maintenance et la lisibilité de l'application.

Exemple

En PVWAWE

Utiliser les mêmes noms des variables entre les langages de programmation C/Fortran et WAVE.

En C et C++

C et C++ n'utilisent pas les mêmes mécanismes de passage de paramètres. Si une fonction C est appelée en C++, on veillera à repérer dans l'identification de la fonction son appartenance au langage C, et vice versa.

Org.Duplication	On doit éviter la duplication de code en utilisant intelligemment les techniques disponibles au niveau du langage (passage de paramètres, utilisation d'opérations abstraites, utilisation de métalangages).
M=3;F=1;P=24;V=1	
Quelconque	

Description

Chaque langage propose des techniques pour éviter le duplication : on étudiera au cas par cas ces techniques et on choisira la plus appropriée. Le choix de la technique incombe au programmeur mais c'est souvent un choix de haut niveau, qui peut remonter jusqu'aux frontière de la conception. En outre, ce choix devra prendre en compte le fait qu'une abstraction trop importante peut nuire à la maintenabilité du programme. On privilégiera ainsi dans certains cas le paramétrage à la généricité.

Justification

La duplication doit être évitée car elle induit des surcoûts et de forts risques d'incohérences en maintenance.
Les techniques proposées par les langages ne sont pas équivalentes : le choix d'une technique inappropriée peut conduire à un code peu lisible ou peu performant.

Exemple

En C, C++ et ADA :

Certaines fonctions « courtes », peuvent générer plus d'instructions dans le passage de paramètres, l'appel de la fonction, le retour et la suppression des paramètres que pour la fonctionnalité elle-même. La directive *inline* permet de préciser au compilateur qu'il serait préférable de remplacer le

code d'appel de la fonction, par l'expansion du code de la fonction. Il peut-être également intéressant de faire appel à ce mécanisme pour une fonction plus importante et appelée une seule fois.

En C++ et JAVA :

Les templates et le polymorphisme sont deux techniques concurrentes qui permettent une programmation générique. On choisira l'une ou l'autre après analyse au cas par cas des avantages et des inconvénients.

En C++ :

Exemple de calcul d'une factorielle à l'aide de modèle de fonction :

```
// on utilise la recursivite pour iterer
template<int n>
inline int FACT () { return n * FACT<n-1>()  ;}

// on utilise la specialisation pour arreter la recursion
inline int FACT<0> () { return 1 ;}
```

Org.Principal	Le programme principal doit être limité au flot de contrôle de plus haut niveau : création des tâches, initialisations, séquencement. Il ne doit contenir ni algorithme de traitement, ni calcul.
M=0;F=1;P=94;V=0	
Quelconque	

Description

Le programme principal doit être court. Il doit synthétiser le déroulement des traitements. Il se charge d'activer un traitement éventuel d'initialisation générale, un ou plusieurs traitements nécessaires à la réalisation de l'objectif fixé et de gérer les erreurs renvoyées par les sous-programmes appelés.

Justification

La compréhension du programme est facilitée si le programme principal ne contient que le flot de contrôle du logiciel.

Exemple

En FORTAN

```
PROGRAM DEMO
     ........    declaration des variables
CALL INIT1
IF  (condition) THEN
     CALL TRAIT1
     CALL SUITE1
     ..............
ELSE
     CALL TRAIT2
ENDIF
     ...........
END
```

Org.MatérielIndep	Le code dépendant du matériel ou du système d'exploitation doit être séparé du reste du code du logiciel.
M=2;F=0;P=84;V=0	
Quelconque	

Description

Découpler au maximum l'interface matériel et système d'exploitation du logiciel à développer. Cette règle doit être appliquée quitte à diviser un module homogène dans le seul but d'en extraire les fonctionnalités non portables.

Justification

Améliore la portabilité

Exemple

Sans objet.

7.2. PRESENTATION DU CODE

Pr.Indentation	Le code doit être indenté. On devra définir et respecter une convention de représentation des structures de contrôle.
M=2;F=0;P=74;V=2	
Quelconque	

Description

Le code réalisé comporte une indentation homogène dans l'ensemble du projet. La valeur conseillée pour l'indentation est de 3 caractères. La valeur adoptée pour l'indentation pourra être conditionnée par l'outil d'édition, de présentation et d'impression du code adopté par le projet. On définira également une convention de présentation des structures de contrôle.

Justification

L'indentation augmente la lisibilité et améliore la compréhension du code.

Exemple

En IDL :

On explicite la présentation des structures de contrôle en IDL :
Exemple de présentation de WHILE

```
WHILE (index GT 3) DO BEGIN
     index = index + 1
     PRINT, “INDEX = “, index
ENDWHILE
```

Pr.Aeration	Le texte d'un programme doit être aéré. Les opérateurs et les opérandes doivent être séparés par des espaces.
M=2;F=0;P=75;V=2	
Quelconque	

Description

Les opérateurs unaires doivent être suivis ou précédés de l'opérande sans espace. Les opérateurs binaires seront entourés d’espaces de part et d’autre.

Justification

Rend la présentation du programme plus homogène et permet de distinguer les opérateurs unaires des autres.
Facilite la lisibilité.

Exemple

En C :

```
Resultat = x + y ;
```

Pr.Instruction	Il ne doit pas y avoir plus d'une instruction par ligne.
M=2;F=0;P=76;V=2	
Quelconque	

Description

Les instructions longues peuvent s'étendre sur plusieurs lignes ; elle sont coupées alors :

- avant : les mots réservés, les opérateurs, les symboles d'affectation, les parenthèses ouvrantes
- après : une virgule, un point-virgule

Justification

Rend la présentation du programme plus homogène et permet de distinguer les opérateurs unaires des autres.
Facilite la lisibilité.

Exemple

En FORTRAN 77

On utilisera le caractère '&' comme indicateur de ligne suite (en colonne 6).

En ADA

```
LE_CUMUL_DE_DEUX_IDENTIFICATEURS_LONGS
     := LA_VALEUR_DU_PREMIER_IDENTIFICATEUR
           + LA_VALEUR_DU_SECOND_IDENTIFICATEUR;
```

Pr.LongLigne	La longueur maximale d'une ligne de code source en nombre de caractères est inférieure à un seuil défini par le projet.
M=2;F=0;P=77;V=2	
Quelconque	

Description

La limite doit être établie. Elle doit tenir compte premièrement d'une éventuelle limitation du compilateur. Deuxièmement, elle doit assurer que les moyens de saisie, de visualisation, d'analyse et d'impression du projet permettent tous une manipulation et une consultation aisées du code.
On visera une limite élevée afin d'assurer au programmeur de l'aisance dans ses saisies, d'autant que les règles proposées ici conduisent à des lignes longues : nom significatif, préfixage, nommage par association, alignement des paramètres, indentation, …
Cela s'applique également aux commentaires.

Justification

Certains compilateurs ignorent les caractères situés après une certaine longueur de ligne. Au delà d'une certaine longueur, la visualisation de longues lignes est mal aisée, leur impression est tronquée. La fixation d'une longueur maximale de ligne de code par le projet, à une valeur élevée mais en de ça de ces seuils permet la compilation, et facilite la manipulation et la consultation du source.

Exemple

Sans Objet

Pr.CartStd	Un cartouche de commentaire standard défini par le projet doit être utilisé pour commenter l'entête de chaque module et la définition d'une opération.
M=2;F=1;P=50;V=1	
Quelconque	

Description

Cet en-tête présente notamment la logique essentielle du module ou de l'opération, ainsi que les aspects de programmation critiques (par exemple : pré-conditions liées à un appel, traitement des exceptions, effets de bord possible, contraintes de portabilité, conditions de synchronisation entre tâches, ...). On notera qu'un en-tête peut s'adresser à l'utilisateur du module ou au mainteneur.
Le contenu exact des en-têtes devra figurer dans les conventions initiales de chaque projet.

Justification

Cette règle permet une meilleure homogénéité, lisibilité et maintenabilité.
Ceci garantit la présence d'au moins un en-tête par fichier.

Exemple

En C

Commentaire d'entête de fichier (c ou h) :

```
////////////////////////////////////////////////////////////////////////
// PROJET: <>
// APPLICATION: <>
// AUTEURS: <>
// DATE DE CREATION: <>
// DESCRIPTION: <>
//
////////////////////////////////////////////////////////////////////////
```

Commentaire d'entête de fonction:

```
////////////////////////////////////////////////////////////////////////
// NOM DE LA FONCTION: <>
// ROLE:
// PARAMETRES EN ENTREE:
// PARAMETRES EN MISE A JOUR:
// CODE RETOUR: <>
////////////////////////////////////////////////////////////////////////
```

Pr.CartDonnée	Chaque déclaration de donnée doit être commentée.
M=2;F=0;P=78;V=1	
Quelconque	

Description

Les variables doivent être présentées et commentées, a fortiori celles ayant une importance fonctionnelle critique.

Justification

La maintenance est grandement facilitée.

Exemple

Sans Objet

Pr.CommFonc	Les commentaires doivent être fonctionnels et ne doivent pas faire double emploi avec le code.
M=3;F=0;P=42;V=0	
Quelconque	

Description

Les commentaires doivent servir exclusivement à apporter des informations supplémentaires au lecteur; ils doivent compléter les informations que le lecteur trouve dans le code même, c'est à dire dans le nom des types, des variables, des paramètres formels, des boucles, des blocs et des exits, dans l'introduction de variables temporaires ou de sous-types, dans l'usage de la qualification ou du renommage. L'information supplémentaire apportée par le commentaire doit être significative : particularité de la variable, objet du bloc, originalité de l'algorithme, …
Les commentaires ne doivent pas servir à faire de la paraphrase, ni à pallier à l'inexpressivité des noms des identificateurs, des paramètres ou des blocs fonctionnels. Il ne s'agit donc pas d'atteindre à tout prix un certain pourcentage de commentaires mais bien de n'avoir que des commentaires **utiles**.
Les commentaires indiquent le pourquoi alors que le code indique le comment.
Les commentaires peuvent même être inexistants, lorsque le code seul est totalement expressif.

Justification

Limite la double maintenance et la divergence code/commentaires.

Exemple

Sans Objet

Pr.CommIdent	Les commentaires doivent être localisés au même endroit que le code concerné et indentés au même niveau que le code concerné.
M=2;F=0;P=79;V=2	
Quelconque	

Description

Pour les instructions courtes d'affectation, on mettra plutôt le commentaire en fin de ligne.
Dans les langages comme C, C++ ou JAVA, la succession d'accolades fermantes ne permet pas de savoir à quelle accolade ouvrante elle correspond. Les accolades fermantes mal placées sont des causes fréquentes d'erreur. Les commentaires participent à la levée d'ambiguïtés.

Justification

Améliore la visibilité

Exemple

En C :

En C ou C++, Chaque accolade fermante pourra ainsi être commentée.

```
while (Condition)
{
     Traitement_1;
     if (Condition_2)
     {
          Traitement_2;
     } // fin du cas 2
} // fin du corps de la boucle de traitement
```

7.3. IDENTIFICATEURS

Id.IdentSignif	Les identificateurs doivent être significatifs.
M=3;F=1;P=25;V=0	
Quelconque	

Description

Les identificateurs seront choisis pour leur côté explicite. Les abréviations sont interdites sauf si elles appartiennent au glossaire du projet ou si elles font véritablement partie de la "culture" du projet.

Justification

Essayer d'avoir un source aussi proche du langage naturel que possible, directement compréhensible et sans ambiguïté.

Exemple

En FORTRAN 77

L'application de cette règle en FORTRAN 77 est souvent difficile (limitation à 6 caractères des noms symboliques). Il est conseillé, si les contraintes de portabilité et/ou sécurité le permettent d'utiliser les facilités de la norme FORTRAN 77 "étendue", qui permet de coder des noms sur 31 caractères.

Id.IdentRegle	Les identificateurs doivent être simples ou fabriqués par concaténation de plusieurs termes ; le principe de concaténation, l'utilisation des déterminants et l'utilisation des majuscules et minuscules doivent être communs à tous les identificateurs du projet.
M=2;F=1;P=51;V=0	
Quelconque	

Description

Les règles de nommage des identificateurs sont définies en début de projet. Elles sont personnalisées pour le projet et concernent toutes les activités. On devra différencier les identificateurs du reste des mots du langage (en particulier, les mots réservés).
En FORTRAN 77 strict (limitation à 6 caractères des noms symboliques), on pourra définir des règles qui permettent tout en restant compact d'être le plus explicite possible

Justification

Améliore la lisibilité

Exemple

En ADA

1. Les différents mots qui composent un identificateur sont séparés par un trait-bas
   `UN_BLOC_DE_TELEMESURE, LA_PILE_DE_TELECOMMANDES, ...`
2. Les variables globales sont en majuscules, les variables locales sont en minuscules et leur nom est représentatif de ce qu'elles désignent.

Id.NomDonnee	Le nom d'une donnée doit être un nom commun emprunté au langage courant ; il doit être au pluriel si la donnée est un ensemble ou un groupe.
M=3;F=0;P=43;V=0	
Quelconque	

Description

Sans Objet

Justification

Améliore la lisibilité

Exemple

En C++

```
LeCapteurStellaire
LesResultats
LesJoursDeLaSemaine
```

Id.VarSignif	Le nom d'une variable doit véhiculer sa signification
M=3;F=1;P=26;V=0	
Quelconque	

Description

Le nom d'une variable doit identifier totalement la variable. Il doit à la fois exprimer ce qu'elle est et la désigner sans ambiguïté. On pourra de plus adopter une règle de nommage concernant les formations des identificateurs de variables et les déterminants utilisés. Par exemple : un article défini ou un adjectif possessif pour une variable, une locution verbale exprimant un état vrai ou faux pour un booléen. De plus, chaque nom de variable a au moins 3 caractères sauf les indices de boucles.

Justification

Améliore la lisibilité du source et la distinction des identificateurs de variables.

Exemple

En ADA :

```
LE_STATUS_TM : UN_CODE_CORRECTEUR;
```

Id.VarType	Le nom d'une variable peut véhiculer également son type, sa nature ou sa portée.
M=2;F=1;P=41;V=1	
Quelconque	

Description

Cette règle concerne essentiellement les langages faiblement typés ou pour lesquels les contrôles statiques sont légers.

Justification

Pour ces langages, cette règle permet d'améliorer la qualité du code.

Exemple

En PVWAWE
- préfixer les COMMONs locaux à un module par **CL_**
- préfixer les COMMONs partagés par d'autres modules avec **CG_**
- préfixer le nom des constantes par **CST_**
- préfixer les types structures par **TS_**

En IDL :

Les variables seront nommées selon la règle : *Scope_Type_Desc.*

```
« Scope » représente la portée de la variable :
Variable globale : utiliser « g_ »
Variable locale : utiliser « l_ »,
Variable appartenant à un bloc commun global : utiliser « CG_ »,
Variable appartenant à un bloc common local : utiliser « CL_ »,
Données membres d'un objet : utiliser « m_ »
« Type » représente le type de la variable :
Type BYTE : utiliser « b »
Type ENTIER : utiliser « n »
Type LONG non signé : utiliser « ul »
Type LONG signé : utiliser « l »
Type FLOTTANT : utiliser « f »
Type DOUBLE : utiliser « d »
Type COMPLEXE : utiliser « c »
Type STRING : utiliser « s »
Type OBJET : utiliser « o »
Type POINTEUR : utiliser « p »
Type STRUCTURE : utiliser « st »
« Desc » est la description de la variable.
```

Id.ConstSignif	Le nom d'une constante doit véhiculer sa signification et non pas sa valeur.
M=3;F=1;P=27;V=0	
Quelconque	

Description

Le nom de chaque constante doit suivre des règles de nomenclature définies sur le projet sauf dans le cas de réutilisation. Ces règles doivent, en particulier, permettre de distinguer rapidement les constantes et les variables. Il est conseillé d'écrire le nom des constantes en MAJUSCULES et ce nom doit être représentatif de ce qu'elles désignent. Cette règle s'applique également aux constantes définies dans des types énumérés et aux macros des langages C et C++.

Justification

Améliore la lisibilité.

Exemple

En ADA :

```
TAILLE_DU_BUFFER : constant := 100;    --  plutot que CENT;
```

Id.ClasseType	S'il n'est pas imposé par le langage, le nom d'un type ou d'une classe doit être un terme général qui désigne un ensemble ou une catégorie de données
M=3;F=0;P=40;V=0	
Quelconque	

Description

Sans Objet

Justification

Améliore la lisibilité du source et la distinction des identificateurs de types.

Exemple

En ADA

```
type UN_CODE_CORRECTEUR is array (1 .. NB_DE_BITS) of BOOLEAN;
```

En C

```
typedef struct
{
     int positionX;
     int positionY;
} tPosition;
```

Id.Pointeur	Si le langage supporte les concepts de pointeur ou de référence, le nom d'un pointeur ou d'une référence doit véhiculer la sémantique de l'objet qu'il désigne (objet pointé ou référencé).
M=3;F=3;P=4;V=0	
Quelconque	

Description

Sans Objet

Justification

Améliore la lisibilité et la distinction des identificateurs de pointeurs.

Exemple

En ADA :

```
type UN_PTR_DE_MAILLON is access UN_MAILLON ;
PTR_COURANT : UN_PTR_DE_MAILLON;
```

Id.Procedure	Les noms de procédures doivent être des verbes ou des groupes verbaux à l'infinitif indiquant l'action à accomplir.
M=3;F=1;P=80;V=0	
Quelconque	

Description

Les verbes doivent être actifs. Cette règle concerne également les macros des langages C et C++. Lors du masquage des données, les méthodes d'accès en écriture auront un préfixe standard.

Justification

Améliore la lisibilité

Exemple

En C++ :

Définition d'un type Complexe et des méthodes d'accès à la partie réelle et la partie imaginaire du nombre complexe :

```
class Complexe {
     …
     {}
public: // Acces
     int virtual obtenirPartieReelle(void)
           // Partie reelle du nombre complexe
           {
              return partieReelle_;
           }

     int virtual obtenirPartieImaginaire(void)
           // Partie imaginaire du nombre complexe
           {
              return partieImaginaire_;
           }
…
};
```

Id.Tache	Les noms de tâches doivent être composés à l'aide des procédures associées et des évènements utilisés pour déclencher ou séquencer la tâche.
M=3;F=2;P=19;V=0	
Quelconque	

Description

Les procédures associées sont les opérations appelées par la tâche : lorsque la tâche possède une cohésion fonctionnelle forte, une seule opération est appelée par cette tâche.

Justification

Améliore la lisibilité.

Exemple

En C :

`void GetAngleStellaire ()` est la procédure appelée cycliquement toutes les secondes pour acquérir l'angle avec une étoile donnée ; on appellera la tâche associée :

```
GetAngleStellaire_1s
```

En ADA :

```
task LE_TAMPON is
      entry PRENDRE (L_ELEMENT : out UN_ELEMENT);
end LE_TAMPON;
```

Id.Fonction	Les fonctions doivent être nommées à l'aide d'un substantif représentant la valeur fournie par cette fonction. Dans le cas d'une fonction retournant une valeur booléenne, on doit utiliser une locution verbale exprimant un état vrai ou faux.
M=3;F=1;P=29;V=0	
Quelconque	

Description

Cette règle concerne également les macros des langages C et C++. Lors du masquage des données, les méthodes d'accès en lecture auront un préfixe standard.

Justification

Améliore la lisibilité.

Exemple

En ADA :

```
function RACINE_CARREE (DE : in UN_REEL) return UN_REEL;
function EXISTE_DEJA  (LA_PISTE : in UNE_PISTE) return BOOLEAN ;
```

Id.NomParFormel	Le nom d'un paramètre formel doit véhiculer le lien entre le paramètre et l'opération concernée.
M=3;F=1;P=80;V=0	
Quelconque	

Description

Sans objet

Justification

Permet une meilleure lisibilité. La facilité de lecture doit primer sur celle de l'écriture.
On obtient ainsi une forme sémantiquement plus explicite.

Exemple

En ADA

```
procedure CREER  (     AVEC_LA_CHAINE    : in  STRING ;
                       LE_MOT            : out UN_MOT);
```

7.4. DONNEES

Don.Declaration	Toute donnée utilisée doit être explicitement déclarée.
M=3;F=3;P=1;V=2	
Quelconque	

Description

Cette règle concerne les langages permissifs qui permettent l'omission des déclarations.
Les directives de déclaration seront explicitées (public, privées, static, etc.).
De plus, toute donnée déclarée doit être utilisée.

Justification

Améliore la maintenabilité et la fiabilité.

Exemple

En FORTRAN 90

L'instruction IMPLICIT NONE est obligatoire.

En C++

On n'utilisera pas de directives de déclaration par défaut.

Don.Separee	Chaque donnée doit faire l'objet d'une déclaration séparée.
M=2;F=1;P=45;V=2	
Quelconque	

Description

On utilisera une ligne pour chaque déclaration.

Justification

Chaque déclaration peut ainsi être commentée.

Exemple

En C ou C++

Incorrect

```
float vIni, vFin, vMoyenne;   // Variables de calcul de la vitesse.
```

Correct

```
float vIni;        // Vitesse en debut d acceleration.
float vFin;        // Vitesse en fin d acceleration.
float vMoyenne;    // Vitesse moyenne.
```

Don.Typage	Les données doivent être systématiquement et explicitement typées.
M=3;F=3;P=2 ;V=2	
Quelconque	

Description

Les types doivent correspondre aux domaines de variation des données. On choisira les domaines de définition les plus « restreints » possibles, en accord avec la sémantique de la donnée.
On doit préciser explicitement toutes les directives d'allocation.

Justification

L'absence de typage explicite peut révéler une anomalie de programmation.
L'attribution d'un type par défaut peut entraîner des erreurs et pose des problèmes de portabilité.

Exemple

En ADA

```
integer NBNOEUD := 10
type(NOEUD), dimension(:), allocatable :: TABNOEUD
integer, dimension(:,:), allocatable, target :: CONNECTIVITE
```

En FORTRAN 90

Utiliser la forme attribuée des déclarations.

Don.TypeAnonyme	On n'utilise pas de type anonyme.
M=3;F=2;P=32;V=1	
Quelconque	

Description

Un type anonyme est un type implicitement déclaré au travers de la déclaration d'une donnée, mais non déclaré tel quel en tant que type.
Les déclarations de données faites à partir de type anonymes sémantiquement équivalents ne sont pas autorisées.
En C, on évitera les "compound literals" et dont la portée dans une fonction est limitée au bloc d'instructions englobant, et les tags.

Justification

Supprime les problèmes d'incompatibilité de types.
Favorise l'évolutivité et la réutilisation du type.

Exemple

En ADA

Remplacer :

```
L_ECHIQUIER : array (1 .. 8, 1 .. 8) of UNE_CASE;
L_AUTRE_ECHIQUIER : array (1 .. 8, 1 .. 8) of UNE_CASE;
```

par :

```
UN_ECHIQUIER is array (1 .. 8, 1 .. 8) of UNE_CASE;
L_ECHIQUIER : UN_ECHIQUIER;
L_AUTRE_ECHIQUIER : UN_ECHIQUIER;
```

Don.Localite M=3;F=1;P=30;V=1 Quelconque	On doit privilégier les déclarations de données locales aux déclarations plus globales : les données locales à un module doivent être préférées aux données globales, les paramètres formels aux données globales, les données locales d'une opération seront préférées aux données de niveau module, les données locales d'un bloc d'instruction doivent être préférées aux données locales d'une opération.

Description

Cette règle est très générale et doit être déclinée selon les contextes et les langages.

Justification

La lisibilité est meilleure si les variables ont une portée limitée (on sait que les variables ne sont pas pertinentes en dehors de leur portée).
L'utilisation de variables plus globales est toujours plus coûteuse en terme d'occupation mémoire et de temps d'accès.
L'utilisation de variables plus globales rend le code peu générique et difficile à maintenir ou réutiliser.
L'utilisation de variables plus globales rend le code moins fiable.
Le compilateur peut le cas échéant éviter des allocations ou du code inutiles : une donnée locale non affectée n'est pas allouée ; une donnée locale non réemployée n'est pas calculée (le code qui l'affecte n'est pas généré). Cela sera particulièrement efficace lors de l'utilisation de la compilation conditionnelle.
Il s'agit de limiter au maximum la portée des variables.

Exemple

En FORTRAN ou IDL
On évitera le mécanisme de COMMON au profit de paramètres
En SHELL ou PERL
On évitera les variables d'environnement
En SHELL
Il est recommandé de définir les variables locales d'une fonction à l'aide de typeset ou à l'aide de l'attribut local
En C++ ou JAVA
On évitera les données statiques.
En C++
Exemple de déclaration au fil de l'eau :

```
void f4 (int &x, int &y, int z  []) {

     // PreTraitement
     f (x);
     g (y);

     // Elaboration de local 1
     int local1 = x+y ;

}
```

Don.Invariant	Des constantes doivent être définies pour les entités dont la valeur est un invariant.
M=2;F=1;P=52;V=1	
Quelconque	

Description

Dans le cas où un invariant n'est utilisé qu'une seule fois (pour une sémantique donnée), la définition d'une constante peut être cependant discutée.

Justification

Cette règle permet de garantir les invariants et donc la fiabilité. De plus, il n'y a aucun impact sur le code en cas de modification de la valeur de la constante (localisation de la modification et unicité), ce qui facilite l'adaptabilité et l'évolutivité.

Exemple

En ADA

```
package PACKAGE_EXEMPLE is
     LONGUEUR_MAX_DE_LIGNE : constant := 255 ;
     type UNE_LONGUEUR_DE_LIGNE is range 0.. LONGUEUR_MAX_DE_LIGNE ;
     MA_LONGUEUR_DE_CARTE : constant UNE_LONGUEUR_DE_LIGNE := 80 ;
     ...
```

Don.Enumeration	On doit privilégier l'utilisation de constantes ou de symboles (de types énumératifs si le langage le permet) aux données numériques entières. L'utilisation de données numériques entières doit se limiter essentiellement aux calculs ou aux comptages simples.
M=2;F=2;P=36;V=1	
Quelconque	

Description

Toutes les constantes (y compris les dimensions d'un tableau) doivent être nommées par des symboles. Les constantes littérales sont interdites, sauf cas très particuliers comme les pas d'incréments 1 et -1. Si le langage le permet, les constantes doivent être typées. Si le langage propose plusieurs mécanismes pour implanter les constantes, on choisira le plus adapté au contexte.

Justification

Cette technique garantit cohérence, évolutivité et réutilisabilité du code.

Exemple

En ADA

Remplacer :

```
type UN_INSTRUMENT is range 1 .. 4;
            -- 1 correspond à CAMERA
            -- 2 correspond à ALTIMETRE
            -- 3 correspond à INTERFEROMETRE
            -- 4 correspond à LASER
```

par

```
type UN_INSTRUMENT is (CAMERA, ALTIMETRE, INTERFEROMETRE, LASER);
```

En C et C++ :

La directive *#define* ne fait pas partie du langage C mais est une commande pour son pré processeur *cpp* : *#define* permet de substituer dans le code source une constante littérale par sa valeur. Le

compilateur travaille sur une version du source après processing et ne connaît donc pas la constante littérale d'origine. La déclaration d'énumérations augmente les possibilités de contrôle.
Exemple incorrect :

```
#define BLANC 0
#define NOIR 1
#define ROUGE 2
#define VERT 3
#define BLEU 4
int uneCouleur ;
uneCouleur = ROUGE ; // correct à la compilation
```

Exemple correct :

```
typedef enum { blanc, noir } tCouleur1 ;
typedef enum { blanc, noir, rouge, vert, bleu } tCouleur2 ;
tCouleur1 uneCouleur = rouge ; // refus justifie à la compiltation
```

Don.Structure	Lorsqu'un objet conceptuel nécessite d'être implanté sous forme de plusieurs données, on doit grouper ces données dans une entité structurante (classe, structure, enregistrement, type) selon les possibilités offertes par le langage.
M=3;F=2;P=18;V=0	
Quelconque	

Description

Sans Objet

Justification

Cela assure une meilleure cohésion des unités de code

Exemple

Sans Objet

Don.Homonymie	L'homonymie doit être évitée sauf dans le cas de surcharge ou de redéfinition explicite.
M=2;F=1;P=46;V=1	
Quelconque	

Description

Une variable locale à un sous-programme ne doit pas avoir le même nom qu'une variable globale à l'unité de compilation ou qu'une variable externe.

Justification

Améliore la lisibilité.
Evite les conflits de visibilité mettant en œuvre des règles parfois complexes.

Exemple

En C

L'utilisation de règles de nommage permettant de distinguer variables locales et variables statiques permet d'éviter ce type d'erreur, souvent difficile à détecter.

Don.Initialisation	Les variables doivent être initialisées avant leur première utilisation.
M=2;F=3;P=11;V=2	
Quelconque	

Description

Toute variable doit être initialisée, soit à la déclaration, soit avant sa première utilisation. Il est recommandé, si possible, d'initialiser les variables lors de leur déclaration : ceci concerne en particulier toutes les variables de type simple (integer, float, char, ...), les pointeurs et références, les variables locales et les variables d'environnement utilisées par le programme et les scripts.
Cette initialisation devrait être faite au moment de leur déclaration dans la mesure où l'on a les moyens d'initialiser la variable avec une valeur significative. A noter que certains langages peuvent imposer ou vérifier l'initialisation des variables, en particulier les variables locales.

Justification

Permet d'éviter les effets de bord et d'éventuels problèmes de portabilité. Ne pas initialiser une variable revient implicitement à utiliser l'initialisation de la mémoire faite par le système d'exploitation qui peut être différente d'un calculateur à l'autre.

Exemple

En FORTRAN

Lorsqu'un COMMON est utilisé pour passer des variables d'un service à un autre, il doit toujours être initialisé par l'appelant.

En C

```
const int MAX_CHAINE = 80;
int Nombre_avion = 0;
char Prenom[MAX_CHAINE]="";
const int TAILLE = 10;
int Tab[TAILLE]={1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10};
```

Don.PointeurNonAff	Si le langage supporte le concept de pointeur, lorsqu'un pointeur n'est pas associé à un objet précis lors de la déclaration, on doit préciser par un commentaire l'objet qui lui sera associé et, si le langage le permet, l'initialiser à null.
M=2;F=3;P=13;V=1	
Quelconque	

Description

Le but de cette règle est de documenter l'utilisation des pointeurs et des références dont la dynamique est compliquée.

Justification

Une des causes d'erreurs la plus fréquente dans l'utilisation des pointeurs ou des références est l'utilisation d'une référence nulle.

Exemple

En JAVA :

```
Point P1 ;  // Premiere extremite du segment
            // Sera affectee des que le segment sera cree
Point P2 ;  // Deuxieme  extremite du segment
            // Sera affecte des que le segment est cree
```

```
Segment S = new Segment () ;
…
P1=S.First () ;
P2=S.Last () ;
```

Don.LocalUnique	Chaque donnée locale doit avoir une utilisation unique.
M=2;F=0;P=80;V=0	
Quelconque	

Description

On doit éviter notamment la définition de données générales réutilisées à plusieurs endroits du code.

Justification

Le code est plus cohérent.
Cela diminue le risque d'effets de bord dus à une initialisation antérieure de la variable.
Multiplier les données locales n'est pas pénalisant en termes de performances : les compilateurs récents savent gérer efficacement les ressources associées.

Exemple

Sans Objet

Don.Utilisee	Toute donnée définie doit être utilisée ; une donnée qui n'est plus utilisée doit être supprimée.
M=2;F=0;P=81;V=2	
Quelconque	

Description

Il faut en particulier penser à supprimer les données locales créées pour un besoin ponctuel, lorsque ce besoin disparaît. Cela est facilité si l'on respecte la règle Don.TypeAnonyme .

Justification

Une variable déclarée et non utilisée correspond à un code inutile qui nuit à la lisibilité et pollue le programme

Exemple

Sans Objet

Don.TablePrincipe	On doit définir le principe de traitement des tableaux à double entrée (ligne x colonne ou colonne x ligne).
M=2;F=2;P=37;V=0	
Quelconque	

Description

On doit définir les principes d'utilisation des tableaux à double entrée; comment les déclarer, quels indices correspondent aux lignes et aux colonnes.

Justification

Sans règle précise, il peut y avoir confusion entre deux développeurs : l'un voit le tableau tel quel ; l'autre voit le tableau transposé.
Dans la plupart des langages, le mode d'adressage des éléments des tableaux conduit à une dissymétrie des performances selon que l'on parcourt le tableau en ligne ou en colonne

Exemple

En FORTRAN

Traiter de préférence les tableaux par colonnes et non par lignes
Faire (traitement d'une colonne dans la boucle la plus interne) :

```
DO   J = 1,N
 DO   I = 1,N
  A(I,J) = B(I,J) * 5.0
 END DO
END DO
```

plutôt que (traitement d'une ligne dans la boucle la plus interne) :

```
DO  I = 1,N
 DO  J = 1,N
  A(I,J) = B(I,J) * 5.0
 END DO
END DO
```

En PVWAWE

Effectuer les boucles sur les indices d'un tableau en commençant par les colonnes puis par les lignes.

En IDL

Dans le cas des tableaux à plusieurs dimensions, effectuer les boucles sur les indices en parcourant les premiers indices dans les boucles les plus internes

Don.TableOper	Les opérations globales sur les tableaux (initialisation, copie, duplication, comparaison) doivent être effectuées à l'aide de primitives standard fournies par le langage lorsque celles ci existent.
M=2;F=2;P=38;V=1	
Quelconque	

Description

Sans Objet

Justification

Le code est plus lisible.
Le code est plus performant.

Exemple

En C et C++ :

On utilisera les fonctions memset, memcpy, etc.

En IDL :

La fonction ARRAY_EQUAL permet de comparer rapidement le contenu de 2 tableaux sans avoir à utiliser de boucles FOR ou d'instruction WHERE.

Don.ChaineOper	Les opérations globales sur les chaînes de caractères (initialisation, copie, duplication, comparaison, recherche, modification) doivent être effectuées à l'aide de primitives standards fournies par le langage lorsque celles ci existent.
M=2;F=2;P=39;V=1	
Quelconque	

Description

Sans Objet

Justification

Le code est plus lisible.
Le code est plus performant.

Exemple

En FORTRAN77 :
On utilisera les fonctions LEN et INDEX.
En C :
On utilisera les fonctions de l'interface <string.h> (strcpy, strcmp, strcat, etc.).
En C++ :
On utilisera le type String de la STL.
En PERL :
Les comparaisons de chaînes de caractères nécessitent l'utilisation d'opérateurs alphabétiques dédiés (eq, lt, gt, le, ge) au lieu des opérateurs numériques standards (==, <, >, <=, >=). Le fait d'utiliser les opérateurs de comparaison numérique sur des chaînes de caractères ne provoque pas d'erreur syntaxique (seulement un avertissement en mode warning) mais ne renvoie pas une valeur correcte…

Don.AllocDynBord	L'allocation dynamique de mémoire est interdite.
M=0;F=3;P=30;V=1	
Bord	

Description

Toutes les instructions qui conduisent à l'allocation ou la désallocation dynamique de la mémoire sont interdites.

Justification

L'allocation puis la désallocation successive peut conduire à une fragmentation trop importante de la mémoire et pour des problèmes de charge CPU il est peu envisageable d'embarquer un processus de défragmentation en continu de la mémoire.

Exemple :

En C :
Il est interdit d'utiliser des mécanismes d'allocation dynamique de mémoire qui font appel à malloc/free (bibliothèque standard) dans les applications temps-réel embarquées

Don.AllocDynSol	Si le langage supporte le concept, l'allocation dynamique de mémoire doit être utilisée avec précaution et opportunité.
M=0;F=2;P=61;V=0	
Sol	

Description

Le projet peut choisir d'interdire l'allocation dynamique de la mémoire, ou de la limiter à certaines unités de compilation, ceci afin de maîtriser l'utilisation de la mémoire.

Justification

L'allocation dynamique exige une analyse de la dynamique de l'application, et peut conduire à des problèmes de morcellement de la mémoire qui peuvent faire chuter les performances

Exemple

Sans Objet

Don.AllocEchec	Si le langage supporte le concept d'allocation dynamique, on doit prévoir systématiquement l'échec possible d'une requête d'allocation mémoire.
M=0;F=2;P=62;V=1	
Sol	

Description

Une requête d'allocation dynamique de mémoire peut échouer du fait d'une place mémoire disponible insuffisante.
Dans tous les cas, un traitement doit être prévu en cas d'échec.

Justification

Une erreur d'allocation mémoire est une erreur grave.
Il est généralement très difficile de remonter à la cause (l'échec de la requête d'allocation) à partir de l'un de ses effets.

Exemple

En C++

Pour prévenir un risque d'échec d'allocation, on pourra envisager les possibilités suivantes :
Définition d'une fonction de traitement global d'erreur, que l'on positionne comme fonction appelée implicitement à l'échec de *new* grâce à la primitive de gestion d'erreur "*set_new_handler*".
Redéfinition de l'opérateur *new* pour une classe donnée : cette technique plus compliquée permet un traitement adapté pour chaque classe.
Utiliser l'exception de la bibliothèque standard «bad_alloc ».
Tester la valeur de retour d'un appel à new et prévoir un traitement ad hoc pour une valeur de retour nulle qui correspond à une erreur d'allocation. On utilisera dans ce cas l'opérateur *new* avec la directive (*nothrow*) afin d'éviter le lancement de l'exception.

Don.AllocLiberation	Toute mémoire allouée doit être libérée au même niveau conceptuel.
M=1;F=1;P=53;V=0	
Sol	

Description

Toute allocation d'une zone mémoire doit faire l'objet d'une désallocation explicite dès que possible et au même niveau conceptuel : opération, service, module, classe. A noter que cette règle est sans objet pour les langages à libération automatique de mémoire comme JAVA.

Justification

Une désallocation systématique économise les ressources mémoire.
C'est au niveau conceptuel où l'on a alloué la mémoire qu'il est le plus simple de libérer cette mémoire.

Exemple

En C

Si un module propose une fonction d'allocation mémoire, il devra aussi proposer la fonction de libération.

En C++

Si les constructeurs allouent de la mémoire, le destructeur la libère.

Don.AllocErreur	Une erreur survenant au cours d'un traitement ne doit pas entraîner une non libération de mémoire.
M=0;F=2;P=63;V=0	
Sol	

Description

Un déroulement de code conduisant à une exception risque de sauter le code de libération des ressources.

Justification

Les ressources allouées doivent être libérées quel que soit le déroulement du code.

Exemple

En C++

Exemple de fonction interrompue par une exception qui conduit à une non libération

```
 void Exception1 (void) {
      try {
            tA * pA ;

            // Allocation locale
            pA = new tA (0) ;

            // Traitement interrompu par une exception
            // ...

            // Liberation de la ressource
            delete pA ;
          }
      catch (MonException e) {
            // ... Traitement de l'exception
      }
```

```
}
```

Exemple de fonction lançant elle même une exception sans libérer les ressources

```
void Exception2 (void) throw (MonException) {
     tA * pA ;
     // Allocation locale
     pA = new tA (0) ;
     // Traitement lancant une exception
     if (true) throw MonException (1);
     // Liberation de la ressource
     delete pA ;
}
```

7.5. TRAITEMENTS

Tr.TestEgalite	L'usage de test d'égalité ou de différence doit être remplacée par des inégalités dès que cela est possible.
M=0;F=3;P=45;V=1	
Quelconque	

Description

Les tests d'égalité ou de différence sont délicats à gérer dans le cas de parcours d'intervalle.

Justification

Amélioration de la robustesse

Exemple

En C :

Remplacer

```
for (int i=0 ; i != MAX ; i++)
```

par

```
for (int i=0 ; i < MAX ; i++)
```

Tr.ComparaisonStrict	La comparaison stricte (égalité, différence) entre nombres flottants (réels, complexes) doit être remplacée par une inégalité.
M=0;F=3;P=46;V=1	
Quelconque	

Description

On ne testera jamais l'égalité entre réels par l'opérateur d'égalité, mais on utilisera un encadrement de leur différence.

Justification

L'égalité stricte de deux opérandes de types réels n'a pas de sens.

Exemple

En ADA :

Remplacer :

```
if MON_REEL = TON_REEL then
```

par :

```
if (abs(MON_REEL - TON_REEL) < EPSILON) then
```

où EPSILON représente la précision machine.

Tr.ModifConst	On ne doit pas modifier la valeur d'une constante.
M=3;F=3;P=3;V=2	
Quelconque	

Description

Cette règle concerne les langages pour lesquels le concept de constante n'est pas défini.
En C et C++, on évitera tout mécanisme de casting qui pourrait modifier une valeur de constante.

Justification

Les constantes représente des invariants qu'il faut respecter

Exemple

En C ou C++

Il faut éviter ce code :

```
const double pi=3.1415926 ;
const double * ptr1 = & pi ;
double * ptr2 = (double *) (ptr1) ;
*ptr2 = 3 ;
```

Tr.ControleRacc	Si le langage supporte le concept, on doit utiliser les formes de contrôle raccourcies chaque fois que le cas s'y prête.
M=2;F=2;P=35;V=1	
Quelconque	

Description

Les formes de contrôle raccourcies sont spécifiques aux langages et correspondent à des cas particuliers courants de forme de contrôles : forme itératives, formes décisionnelles, etc.

Justification

Améliore la lisibilité.
Améliore la rapidité d'exécution du code.

Exemple

En C :

Préférer l'utilisation de l'instruction `for` à celle d'un `while`, lorsque cela est possible.
Préférer l'utilisation d'un switch à une série de `if` – `else if` si les conditions concernent l'énumération des valeurs d'une expression entière.

En ADA :

Il est préférable d'écrire

```
if Y /= 0 and then (X / Y) = 10 then .    -- OK
```

que d'écrire

```
if Y /= 0 and (X / Y) = 10 then ...       -- CONSTRAINT_ERROR
POSSIBLE
```

Tr.Choix	On doit utiliser une instruction de choix à la place d'une simple instruction conditionnelle dès qu'il y a plus d'une alternative.
M=2;F=0;P=70;V=1	
Quelconque	

Description

Sans Objet

Justification

Dans le cas d'un choix multiple, le compilateur construit une table des adresses de branchement de chaque cas (ce qui est possible si les valeurs à tester sont numériquement consécutives), de manière à ce que le temps d'accès à chaque cas soit invariant.
Améliore la lisibilité et l'auto-description du code.

Exemple

En ADA :

```
type UNE_REPONSE is (OUI, NON, PEUT_ETRE) ;
LA_REPONSE_DE_L_OPERATEUR := REPONSE_OPERATEUR ( DE_L_OPERATEUR =>
OPERATEUR_ACTIF) ;
case LA_REPONSE_DE_L_OPERATEUR is
     when OUI          => TRAITER;
     when NON          => NE_PAS_TRAITER ;
     when PEUT_ETRE    => DECIDER ;
end case ;
```

En FORTRAN 77 :

On évitera le goto calculé, moins lisible, et on préfèrera des if elseif imbriqués.

En C et C++ :

Dès qu'il y a plus d'une alternative possible, préférer l'utilisation d'une instruction `switch/case` à celle d'une instruction **`if/else if/else`**

Tr.OrdreChoix	Lors de l'utilisation d'une instruction de choix, tous les cas possibles doivent être traités, de préférence de façon explicite et dans l'ordre "logique" des cas.
M=3;F=2;P=17;V=1	
Quelconque	

Description

Cela signifie entre autre que l'on doit renoncer aux traitements par défaut.

Justification

Améliore la maintenabilité et la fiabilité du logiciel.

Exemple

En ADA :

Il est préférable d'écrire :

```
type UNE_REPONSE is (OUI, NON, PEUT_ETRE) ;
LA_REPONSE_DE_L_OPERATEUR := REPONSE_OPERATEUR ( DE_L_OPERATEUR =>
OPERATEUR_ACTIF) ;
case LA_REPONSE_DE_L_OPERATEUR is
     when PEUT_ETRE    => DECIDER ;
     when OUI          => TRAITER;
```

```
     when NON          => NE_PAS_TRAITER ;
end case ;
```

Que d'écrire :

```
type UNE_REPONSE is (OUI, NON, PEUT_ETRE) ;
LA_REPONSE_DE_L_OPERATEUR := REPONSE_OPERATEUR ( DE_L_OPERATEUR =>
OPERATEUR_ACTIF) ;
case LA_REPONSE_DE_L_OPERATEUR is
     when OUI          => TRAITER;
     when OTHERS       => NE_PAS_TRAITER ;
end case ;
```

Tr.Goto	L'instruction de branchement inconditionnel (goto) doit n'être utilisée que dans des cas très limités et spécifiques.
M=3;F=3;P=6;V=2	
Quelconque	

Description

Le goto doit être utilisé uniquement pour le traitement d'erreur. Si le langage supporte le traitement d'exceptions comme ADA, JAVA ou C++, le goto sera banni.
Il est interdit pour faire un branchement vers l'arrière, ou dans une instruction structurée comme une boucle.

Justification

L'instruction **goto** conduit souvent à un programme déstructuré, ce qui augmente sa complexité et le risque d'erreurs.

Exemple

En C

L'utilisation est tolérée pour le traitement d'erreur :

```
while(Condition_1)
{
     Traitement_1;
     if (Condition_2)
     {
          goto Erreur
     }
     Traitement_2;
     if (Condition_3)
     {
          goto Erreur
     }
     ...
}
goto Fin;

Erreur : Traitement_Erreur;

Fin : ...
```

En revanche, le saut long (set jump, long jump) est interdit.

Tr.BoucleSortie	Une boucle doit posséder une sortie nominale unique.
M=3;F=2;P=16;V=2	
Quelconque	

Description

Un algorithme de boucle bien structuré ne doit pas nécessiter plusieurs sorties possibles. C'est la condition qui doit éventuellement tester les différentes possibilités d'interruption de la boucle.
L'utilisation de l'instruction de sortie inconditionnelle peut être tolérée si le respect de la règle aboutit à une programmation beaucoup plus complexe de la boucle.

Justification

La multiplicité des sorties de boucles déstructure le programme et nuit à sa compréhension.
L'instruction de sortie inconditionnelle dans une boucle déstructure le programme et augmente sa complexité.

Exemple

En C

```
// incorrect
  Indice = 0;
  while (Indice < MAX)
  {
       if (Lettre[Indice] == CLE)
       {
            break;
       }
       Indice ++;
       Traitement;
  }
// correct
  Indice = 0;
  while ((Indice < MAX ) && (Lettre[indice] != CLE))
  {
       Indice ++;
       // traitement
  } // fin de boucle sur variable
```

Tr.ModifCondSortie	On ne doit pas modifier la condition de sortie de la boucle, dans le traitement de la boucle.
M=3;F=3;P=8;V=1	
Quelconque	

Description

Le test de sortie de boucle doit être une comparaison de la valeur du paramètre de boucle avec une valeur connue à l'entrée de la boucle et indépendante du traitement du corps de boucle.

Justification

Améliore la lisibilité

Exemple

En C

```
// incorrect
 for (I = 0; I == Max ; I++)
 {
      ...
      Max = Fonc_1();
      ...
 }
```

Tr.ModifCompteur	On ne doit pas modifier le compteur de boucle dans le traitement de la boucle.
M=3;F=3;P=7;V=2	
Quelconque	

Description

La valeur du paramètre de boucle ne doit pas être modifiée par le traitement du corps de la boucle, sauf pour les instructions itératives qui ne modifient pas implicitement le compteur de boucle. Dans ce dernier cas, la modification du compteur de boucle ne sera effectuée qu'une seule fois, en fin de corps de boucle.

Justification

Améliore la lisibilité

Exemple

En C

```
// Incorrect
for (I = 0; I <= max; I++)
{
     ...
     I = Fonc_1();
     ...
}
// Correct
while (I <= max)
{
     ...
     I++ ;
}
```

Tr.RecursifSol	On ne doit utiliser d'opération récursive que si elle est conceptuellement plus simple que l'opération itérative équivalente.
M=0;F=2;P=64;V=1	
Sol	

Description

A tout problème récursif correspond une solution itérative. Cependant, certains types de données se prêtent particulièrement bien à des algorithmes récursifs. Dans ces cas, on privilégiera ce type de solution. Cependant, avant de retenir une solution récursive on s'assurera dès la phase de conception de définir les mécanismes de sortie de récursivité. On pourra par exemple estimer si la profondeur maximum d'appel que l'on risque d'atteindre à l'exécution est admissible. Si c'est le cas on utilisera ce maximum comme limite de type d'un paramètre de contrôle de la profondeur afin de traiter correctement l'exception levée par un test dans le cas d'un dépassement.

Justification

L'utilisation de la récursivité est plus difficile à tester et à comprendre : de ce fait, son utilisation doit être limitée.

Exemple

Sans Objet

Tr.RecursifBord	La récursivité est interdite.
M=0;F=3;P=64;V=2	
Bord	

Description

Cette règle concerne la récursivité directe et la récursivité indirecte ou croisée.

Justification

La récursivité peut engendrer un comportement non déterministe et dangereux lorsque la profondeur (i.e. le nombre d'appels successifs) n'est pas connue a priori. Il est alors difficile d'estimer la taille de la pile d'exécution nécessaire au déroulement d'un algorithme récursif.

Exemple

Sans Objet

Tr.FonctionSortie	Une fonction ne doit contenir qu'une instruction de sortie.
M=3;F=2;P=15;V=2	
Quelconque	

Description

Dans le cas d'une fonction, la sortie se fait par une instruction return qui doit être accompagnée d'une valeur nominale significative. Des points de sortie multiples peuvent être tolérés dans le cas particulier des traitements d'erreur (instruction return associée au renvoi d'une valeur de code d'erreur).

Justification

Cette règle améliore la maintenabilité du sous-programme dans le cas où des traitements devraient être ajoutés avant l'instruction de sortie.
Une seule sortie nominale et une seule sortie en erreur diminuent la complexité du sous-programme et l'effort de test associé.

Exemple

En C :

```
// fonction retournant un entier
int Fonction_1 (void)
{
     int Res ;
     if feof (F_Desc)
     {
          Res = 0;
     }
```

```
        else
        {
              Res = 1;
        }
        return (Res);
  }
```

Tr.ProgDefensive	On doit favoriser la programmation défensive en réalisant des pré-conditions et des post-conditions.
M=1;F=3;P=20;V=1	
Quelconque	

Description

La programmation défensive consiste à rajouter des assertions dans le code dont l'objectif est la vérification d'invariants : en entrée, ce sont des pré-conditions ; en sortie, ce sont des post-conditions. En cas de non vérification d'une assertion, une erreur est signalée ou exception est levée.
Afin d'éviter la dégradation des performances, on peut rendre « optionnelle » les assertions par des techniques de compilation conditionnelle.

Justification

Les contraintes d'utilisation de la fonction sont spécifiées formellement (pré condition). Les post-conditions donnent des garanties à l'utilisateur sur les traitements effectués.
La mise au point de l'application est facilitée

Exemple

En SHELL

Les programmes doivent vérifier la validité de tous leurs arguments avant de commencer leur traitement et contrôler les entrées de l'utilisateur (au clavier)

En C++

```
class TableUnsigned{
     // Table d entiers positifs.
public: // Constructeur
     Table(unsigned min, unsigned max);
           // Creation d une table de bornes min et max.
public: // Acces
     int& operator [ ](int index)
           // Acces en lecture et ecriture
     {
           precondition( index > min() && index < max() );
                 // Test de la precondition
           int& retour = accesLectureEcriture(index);
                 // Appel de la fonction de delegation.
           postcondition( retour >= 0 );
                 // Test de la postcondition
           return retour;
     }
private: // fonction de delegation de l operateur [].
     int& accesLectureEcriture(int index);
}
```

Tr.Residus	Il ne doit pas y avoir de résidus de programmation en commentaire dans le code : une instruction qui n'est plus utilisée doit être supprimée.
M=2;F=0;P=71;V=2	
Quelconque	

Description

Les résidus sont souvent liés à des portions de code mort apparaissant après modification de code. Il peut y avoir cependant pour des raisons de robustesse du code non atteignable : il doit être alors commenté.

Justification

Le code mort alourdit et nuit à la lisibilité
Le code mort peut entraîner un effort de test inutile

Exemple

En FORTRAN

Toute étiquette doit être utilisée. Une étiquette qui n'est plus référencée doit être supprimée.

Tr.Parenthèses	On doit systématiquement parenthéser les expressions.
M=1;F=2;P=42;V=2	
Quelconque	

Description

On ajoute des parenthèses syntaxiquement redondantes pour améliorer la lisibilité.

Justification

Améliore la lisibilité, facilite la portabilité du code.

Exemple

En C

On remplace :

```
pressionTotale = forceA / SurfaceA + forceB / SurfaceB ;
```

Par :

```
pressionTotale = (forceA / SurfaceA) + (forceB / SurfaceB) ;
```

Tr.CalculStatique	Dans le cas d'un langage compilé, on doit privilégier les calculs sur des expressions statiques qui sont faits à la compilation, en précision maximale, plutôt que les expressions calculées dynamiquement.
M=0;F=1;P=100;V=1	
Quelconque	

Description

Ce point est d'autant plus important lorsque la machine cible est beaucoup moins performante que la machine de développement.

Justification

Portabilité, performance

Exemple

En ADA :

Dans ce premier cas, toutes les données sont constantes : les valeurs peuvent être alors calculées un fois pour toutes, à la compilation, avec la précision de la machine de développement.

```
PI : constant := 3.1415926536;
PI_SUR_2 : constant := PI/2.0 ;
DE_DEGRE_VERS_RADIAN : constant := PI_SUR_2 / 90.0 ;
DE_RADIAN_VERS_DEGRE : constant := 1.0 / DE_DEGRE_VERS_RADIAN;
```

Dans ce second cas, les données sont variables : le compilateur va donc générer un code d'initialisation, qui sera exécuté sur la machine, avec la précision de cette dernière.

```
PI : real := 3.1415926536;
PI_SUR_2 : real := PI/2.0 ;
DE_DEGRE_VERS_RADIAN : real := PI_SUR_2 / 90.0 ;
DE_RADIAN_VERS_DEGRE : real := 1.0 / DE_DEGRE_VERS_RADIAN;
```

Tr.Booleen	Une expression conditionnelle complexe doit être remplacée par un unique booléen exprimant un état.
M=2;F=0;P=72;V=1	
Quelconque	

Description

Sans objet.

Justification

Améliore la compréhension et la lisibilité du code.

Exemple

En C

On remplace :

```
if (forceA >= Seuil1 &&  abs(forceB) < Seuil2) …
```

Par :

```
bool contrainteA = forceA >= Seuil1 ;
bool contrainteB = abs(forceB) < Seuil2 ;
bool conditionAB = contrainteA && contrainteB ;
if (conditionAB) …
```

Tr.DoubleNeg	Dans les expressions booléennes on doit éviter les doubles négations.
M=2;F=1;P=47;V=2	
Quelconque	

Description

Sans objet.

Justification

Les doubles négations rendent la compréhension difficile.

Exemple

En ADA

Il vaut mieux écrire
```
 if  EXISTE then ....         -- COMPREHENSIBLE
```
plutôt que
```
 if not N_EXISTE_PAS then ....      -- LOURD
```

Tr.MelangeType	On ne doit pas mélanger des données de types différents dans une même expression.
M=3;F=3;P=9;V=2	
Quelconque	

Description

Le type d'une expression arithmétique est généralement déterminé par le compilateur en fonction du type des opérandes suivant des règles qui peuvent parfois échapper au développeur.
Les exceptions à cette règle sont les suivantes :

- l'exponentiation par un entier(qui n'est pas stricto sensu une exception puisque les règles de coercition ne demandent pas dans ce cas une conversion),
- la multiplication par un scalaire littéral de type entier et de faible valeur.

Justification

Améliore la lisibilité
Permet la maîtrise de l'évaluation des expressions

Exemple

En FORTRAN

Cet exemple montre comment le mélange de données de type différent peut conduire à une évaluation de précision moindre à celle éventuellement souhaitée

```
      REAL OPER1,OPER2
      DOUBLE PRECISION RESUL,OPER3
            ..............
      RESUL = OPER1 + OPER2 + OPER3
```

En FORTRAN 77, l'instruction précédente est équivalent en fait à la séquence suivante :

```
            REAL TEMP
            TEMP = OPER1 + OPER2
            RESUL = DBLE(TEMP) + OPER3
```

Pour obtenir la précision maximale, il aurait fallu écrire :

```
            RESUL = DBLE(OPER1) + DBLE(OPER2) + OPER3
```

Tr.ComparConst	Dans une comparaison avec une constante, la variable doit être toujours à gauche de l'opérateur de comparaison.
M=1;F=1;P=59;V=1	
Quelconque	

Description

L'expression de la comparaison d'une variable avec une constante peut être écrite de deux manières différentes suivant que ce soit la variable qui soit comparée à la constante ou l'inverse. On choisira toujours l'écriture traduisant la comparaison de la variable à la constante.

Justification

Une telle écriture de comparaison améliore la lisibilité du programme.

Exemple

En C ou C++

```
// incorrect
#define MAX_PARAM
if (MAX_PARAM >= Nb_Param)
{
     // traitement nominal
}
// correct
#define MAX_PARAM
if (Nb_Param <= MAX_PARAM)
{
     // traitement nominal
}
```

Tr.OrdreParFormel	L'ordre de déclaration des paramètres formels doit être standardisé.
M=1;F=0;P=116;V=1	
Quelconque	

Description

L'ordre sera défini pour le projet. On n'utilisera pas les modes de passage par défaut (par exemple in en ADA).

Justification

Lisibilité accrue par clarification de la sémantique.

Exemple

En ADA

Les paramètres sont cités selon l'ordre (***in*** puis ***in out*** et enfin ***out***).

En FORTRAN

La liste des paramètres doit être dans l'ordre suivant :
- nom de sous-programme,
- entrée,
- entrée/sortie,
- sorties,
- code retour

Tr.ParamOptionnel	On ne doit pas utiliser les paramètres optionnels lors de la définition d'une opération.
M=1;F=0;P=115;V=2	
Quelconque	

Description

On renoncera à l'utilisation de paramètres optionnels lorsque ceux ci sont possibles ; certains langages évolués comme JAVA ont d'ores et déjà abandonner ce mécanisme jugé trop risqué.

Justification

L'utilisation de paramètres optionnels masque l'interface réelle des opérations et peut conduire à des surprises concernant leur comportement.

Exemple

En C++

On remplacera :

```
void Calcul (double x, double epsilon=0.00001) { …
```

Par :

```
void Calcul (double x) { Calcul (x, 0.00001) ; }
void Calcul (double x, double epsilon) { …
```

Tr.ModifParSortie	Une opération ne doit pas modifier les paramètres en entrée.
M=2;F=1;P=48;V=2	
Quelconque	

Description

C'est vrai en particulier pour les éléments en entrée de type non scalaire (comme les tableaux, les structures, les instances) qui sont passés par adresse ou par référence.

Justification

Déclarer constant un paramètre d'entrée contribue à la fiabilité de l'application puisque la constance est vérifiée par le compilateur. C'est également un commentaire formel pour les clients de la fonction qui leur assure la non-modification des objets passés en paramètre.

Exemple

En C ou C++

Dans le cas d'un passage par adresse d'un argument d'entrée, il sera obligatoirement protégé par le qualificateur `const`

```
int Cherche_Ind (const int * Tab, int Dim, int Val)
{
   …
}
```

Tr.ModifVarGlobal	Une fonction ne doit pas modifier la valeur d'une variable globale, ni ne doit comporter de paramètres en sortie.
M=2;F=3;P=12;V=2	
Quelconque	

Description

Un sous-programme qui n'a qu'un seul paramètre en sortie doit être une fonction sauf si ce paramètre n'est qu'un sous-produit du traitement (auquel cas on utilise une procédure).

Justification

Améliore la lisibilité en mettant mieux en valeur l'objet de l'action déclenchée.
Elimination des effets de bords.
Améliore la fiabilité et la portabilité.

Exemple

En ADA

Remplacer :

```
EXTRAPOLER  (L_ORBITE  => L_ORBITE_COURANTE,
                          A_LA_DATE => LA_DATE_COURANTE);
```

Par :

```
L_ORBITE_COURANTE := EXTRAPOLATION  (DE => L_ORBITE_COURANTE,
                                     A_LA_DATE => LA_DATE_COURANTE);
```

Tr.ParSortie	Tous les paramètres en sortie d'une procédure doivent avoir reçu une valeur avant la première condition du traitement, si besoin par une initialisation par défaut. Il en est de même pour toute variable utilisée pour retourner la valeur d'une fonction.
M=;F=;P=25;V=1	
Quelconque	

Description

On notera cependant que ce qui est important ce n'est pas tant de donner des valeurs, mais surtout que le sous-programme accomplisse ce qu'il doit faire (ou alors lève une exception).
Cette règle n'est pas applicable si un des paramètres est un code de retour ; certains autres paramètres ***out*** peuvent alors ne pas être initialisés s'ils ne sont pas significatifs (ce style de programmation est en général à éviter, mais ce n'est pas toujours possible, notamment en cas d'interfaçages avec d'autres langages).

Justification

Evite des résultats aléatoires.

Exemple

En C :

Exemple correct :

```
void setAlpha (int * alpha) {
     *alpha=0 ; // valeur par defaut
     if (…) ///
}
```

Exemple incorrect :

```
void setAlpha (int * alpha) {
     if (…) ///
}
```

7.6. GESTION DES ERREURS

Err.Mecanisme	La gestion d'erreur doit être effectuée en utilisant les moyens mis en œuvre au niveau du langage (exceptions ou autres). Si le langage propose plusieurs mécanismes possibles, on choisira celui qui respecte le mieux les autres règles sur la gestion des erreurs. Si le langage ne propose pas de mécanismes spécifiques à la gestion d'erreur, on devra réaliser un module dédié à la gestion d'erreurs.
M=3;F=3;P=10;V=0	
Quelconque	

Description

Pour les langages supportant le mécanisme d'exception, ce mécanisme sera utilisé. Pour les autres, on utilisera le code retour des fonctions ou des services, en définissant des règles concernant cette valeur de retour et en imposant l'utilisation de ces valeurs de retour par les appelants.

Justification

Lorsque le mécanisme d'exception est disponible, il permet de traiter clairement et efficacement les erreurs survenues lors de l'exécution. Les rôles de l'appelant et de l'appelé sont bien définis.

Exemple

En ADA

La politique de gestion des pannes utilise le mécanisme d'exception du langage Ada. La technique de code de retour associé aux sous programmes, celle d'associer un marqueur de validité aux variables, et celle de centralisation des erreurs sont interdites.

En IDL

On utilisera le mécanisme CATCH , plutôt que le mécanisme ONERROR

En Java et C++

Il ne faut pas utiliser le retour fonctionnel pour la gestion des erreurs. On utilisera le mécanisme d'exception.

En SHELL

Un programme se terminant correctement doit toujours renvoyer explicitement la valeur 0, un programme doit toujours renvoyer explicitement un code d'erreur en cas d'incident, le projet peut définir des familles d'erreurs, et les codes de terminaison associés, dès lors que le programme shell peut échouer pour plusieurs raisons différentes

Err.TraitementDiff	Les traitements d'erreur doivent être différenciés selon les pannes.
M=0;F=2;P=65;V=0	
Quelconque	

Description

Le traitement des pannes est différencié selon les pannes. Il correspond à la logique applicative du projet et répond à son objectif de robustesse. On différenciera notamment les pannes prévues et les pannes imprévues, les cas où l'on sait résorber la panne et les cas où on ne sait pas. La différenciation des pannes s'effectue en créant soient des « familles » d'exceptions, soit en codifiant les numéros d'erreurs.

Justification

Améliore la fiabilité

Exemple

En C++ et JAVA

On définira des hiérarchie d'exception en utilisant le mécanisme d'héritage et en filtrant les erreurs par une organisation astucieuse des blocs catch.

En C

On utilisera un entier pour coder les erreurs avec les règles suivantes :

```
errno < 1024 => erreur système
1024 <= errno < 2048 => erreur d'entrée sortie niveau applicatif
Etc.
```

Err.Impression	L'opportunité de signaler un message d'erreur doit être étudiée.
M=0;F=1;P=99;V=1	
Quelconque	

Description

Signaler un message d'erreur peut être effectué par le biais d'une trace, d'une impression, de la création d'un fichier d'erreur, de l'utilisation d'une fenêtre ou d'une console d'erreur ou encore de l'utilisation d'un périphérique d'erreur dédié

Justification

Facilite la mise au point.

Exemple

En SHELL et PERL

Les erreurs seront enregistrées dans un fichier journal.

Err.Nom	Un traitement d'erreur ou une exception doit porter un nom exprimant la raison pour laquelle le service demandé ne peut être rendu.
M=0;F=1;P=95;V=0	
Quelconque	

Description

Cette règle concerne en premier lieu les langages à exception. Pour les autres langages, on pourra définir des symboles significatifs pour chaque « code d'erreur ».

Justification

Améliore la lisibilité.

Exemple

En ADA :

```
package DES_LISTES is
type UNE_LISTE is limited private ;
    LISTE_SATUREE : exception;
    -- Levée lorsque la liste est saturée à la création ou à l'insertion.
end DES_LISTES ;
```

En SHELL

Les scripts utiliseront les codes de fin anormale suivants (la valeur numérique est indiquée entre parenthèses) :

```
BAD_ARGS (1)     Erreur dans le nombre des arguments d'une fonction
NO_FILE (2)      Erreur d'accès à un fichier
UNKNOWN (3)      Toutes autres erreurs.
```

Err.FinOperation	Le traitement d'erreur doit être localisé en fin d'opération.
M=1;F=1;P=60;V=1	
Quelconque	

Description

Dans une majorité de cas, la levée d'une exception provoque l'arrêt de l'opération. Le traitement de toutes les exceptions pouvant être levées doit alors être réalisé en fin d'opération plutôt que par des blocs imbriqués gérant chacun d'eux leurs exceptions propres.

Justification

Cette règle améliore la lisibilité du code. En effet, l'algorithme nominal n'est pas pollué par le traitement des erreurs et les traitements communs à plusieurs exceptions peuvent être factorisés en fin d'opération.

Exemple

Sans Objet

Err.Operation	Le traitement d'une erreur doit être effectué au niveau de l'opération qui peut traiter cette erreur.
M=1;F=3;P=21;V=0	
Quelconque	

Description

Dans les langages à exception, une exception ne doit pas être récupérée par une fonction qui n'a pas les moyens de la traiter.

Justification

Vouloir traiter une erreur trop tôt alourdit le code pour rien
En cas de traitement trop précoce, le programmeur risque d'oublier de propager l'erreur au niveau au-dessus, où il est peut-être possible de la traiter.

Exemple

En C++ ou JAVA

Exemple incorrect

```
// methode1 peut envoyer l exception MonException.
void methode1() throws MonException {
    if (...) {
        throw new MonException();
    }
}
// methode2 appelle methode1 mais ne sait pas traiter
// MonException.
void methode2() {
    try {
        methode1();
    } catch (MonException e) {
        // Simple trace, pas de traitement de l erreur.
    }
}
```

Exemple correct

```
void methode1() throws MonException {
    if (...) {
        throw new MonException();
    }
}

// On declare que methode2 peut renvoyer l exception.
void methode2() throws MonException {
    // Si methode1 leve l exception MonException
    // elle est propagee a l appelant de methode2.
    methode1();
}
```

Err.IntegriteDonnee	Le déclenchement d'une erreur ne doit pas modifier l'intégrité d'une donnée.
M=1;F=3;P=22;V=0	
Quelconque	

Description

Dans le cas des langages objets, pour les objets à construction non triviale, le lancement d'une exception lors de la modification de l'objet peut conduire à une non-intégrité.

Justification

Une donnée non intègre conduit à des problèmes graves ou à des fonctionnements non prédictibles.

Exemple

En C++ :

Un exemple ou une allocation qui échoue lors d'une opération d'assignation rend la donnée non intègre

```
class  Chaine {
private:
     char * chaine ;
public:

     Chaine (const char *s) ;
     Chaine (Chaine & s) ;
     ~Chaine () ;

     // Operateur d assignation
     Chaine & operator= (const Chaine & s) ;
};

Chaine::Chaine (const char * s) {
     int size = strlen (s) + 1 ;
     chaine = new char [size] ;
     strcpy (chaine,s);
}

Chaine & Chaine::operator= (const Chaine & s) {
     if (this!=&s) {
          delete [] chaine ;
          int size = strlen (s.chaine) + 1 ;
          chaine = new char [size] ;
          // si l allocation echoue,
```

```
            // chaine pointe sur une zone qui vient d etre
            // liberee (et donc succeptible d etre reutilisee plus tard)
            strcpy (chaine,s.chaine);
      }
      return *this ;
}
```

Si on veut éviter ce problème, il faut placer chaque appel à new dans un bloc try catch ; en cas d'erreur, chaine doit être affecté à 0 ; il reste alors à vérifier que chaine est différent de 0 avant toute opération licite, y compris la destruction.

Err.ToutesTraitées	Toutes les erreurs doivent être traitées, aucune erreur ne doit être masquée ou ignorée : le déclenchement d'une erreur ne doit jamais interrompre le programme brutalement.
M=0;F=3;P=42;V=0	
Quelconque	

Description

Pour améliorer la robustesse de l'application celle-ci doit prévoir la réception de tous les signaux (ou interruption) et définir une stratégie de traitement au cas par cas en fonction du besoin et de la nature du signal.

Justification

Une exception non traitée entraîne l'arrêt brutal du programme, ce qui n'est jamais souhaitable.

Exemple

En général : cas des exceptions numériques

Dans le cas particulier du processeur arithmétique, l'ensemble des exceptions numériques doit être passé en revue. Certaines exceptions peuvent être masquées (en général les exceptions d'arrondi et d'underflow) avec justification à l'appui. Les exceptions non masquées devront êtres associées à un traitement logiciel dédié.

En C++

Une exception non traitée est remontée au plus haut niveau, et donne lieu au déclenchement des handlers d'exception non traitée. Il existe deux handler : terminate qui termine obligatoire l'exécution en cours, et unexpected qui permet éventuellement de ré-aiguiller l'exception courante (qui est non traitée) vers une exception traitée. Il est possible de redéfinir ces handler par des fonctions personnalisées : ce moyen peut être utile pour la recherche d'une robustesse ultime ou pour traiter à haut niveau des exceptions très générales. Elle ne doit pas se substituer au principe de traitement local des exceptions. Le fonctionnement précis et les liens entre les deux handler terminate et unexpected dépendent généralement du contexte applicatif : il conviendra donc lors de leur utilisation d'analyser précisément leurs comportements respectifs.

Exemple de redéfinition de terminate

```
void MaFin () {
      cerr << "Exception non traitée\n" ;
      exit (-1) ;
}

void Fin () throw (char *) {
      // Modification du handler de terminaison
      terminate_handler previousTerminate = set_terminate (MaFin) ;

      // Code
      if (…) throw "Exception" ;
```

```
     // Restitution du handler standard
     set_terminate (previousTerminate);
}

Exemple de redéfinition de unexpected
void MaFin2 () throw (int) {
     cerr << "Exception non attendue\n" ;
     throw (1);
}

void Fin2C () throw (char *) {
     // Modification du handler de terminaison
     unexpected_handler previousUnexpected = set_unexpected (MaFin2) ;

     // Code
     if (…) throw "Exception" ;

     // Restitution du handler standard
     set_unexpected (previousUnexpected);
}

void Fin2 () {
     try {
          Fin2C ();
     }
     catch (int e) {
          cerr << "Interception " << e << endl ;
     }
}
```

Err.Canal	Les messages d'erreur doivent être transmis à l'utilisateur par le biais du canal d'entrée sortie dédié, lorsque celui ci existe au niveau langage. S'il n'existe pas, un canal dédié doit être créé pour cela.
M=1;F=0 ;P=114;V=1	
Quelconque	

Description

Le canal peut être un moyen de communication quelconque : un canal logique, un fichier, une console, etc.

Justification

Améliore la cohérence dans le signalement des erreurs
Améliore la fiabilité

Exemple

En SHELL

On utilise Le descripteur d'erreur 2 qui correspond au fichier d'erreurs standard. L'utilisateur peut ainsi demander la redirection des affichages normaux dans un fichier, tout en conservant à l'écran l'affichage des erreurs.

```
if un_test_qui_doit_reussir
then
   # Opérations quelconques
else
   print 'Le test_qui_doit_reussir a échoué !' >&2
```

```
fi
```

7.7. DYNAMIQUE

Dyn.OS	On doit analyser avec précision les mécanismes de gestion des tâches et threads proposés par le système d'exploitation et/ou le noyau temps réel. On doit argumenter avec précision les décisions concernant l'utilisation ou la non utilisation de chaque mécanisme.
M=1;F=1;P=54;V=0	
Quelconque	

Description

L'environnement de programmation propose généralement des classes et des services spécialisés qui permettent de gérer le multi-tasking et le multi-threading. En particulier des classes pour gérer l'exclusion mutuelle, des services pour inhiber la préemption, etc.
Des options de compilation permettent également de personnaliser le code généré : par exemple pour vérifier ou pour assurer le non partage d'une variable par différents threads.

Justification

La programmation multi-thread est très liée au contexte.

Exemple

En IDL

Sur les machines multiprocesseurs, IDL autorise le multi-threading qui permet d'augmenter la vitesse des calculs en exploitant simultanément les processeurs disponibles. IDL évalue automatiquement les calculs réalisés par les différentes routines et décide de celles pouvant bénéficier du multi-threading en fonction des paramètres suivants :

Nombre d'éléments concernés,
Disponibilité des processeurs,
Disponibilité d'une version multi-threadée de la routine utilisée.

Seul un certain nombre d'instructions IDL possède une version multi-threadée, et peuvent donc bénéficier du multi-threading. Pour obtenir cette liste, se référer à l'aide en ligne de IDL à la rubrique « Services that use the thread pool ».

Dyn.AttenteActive	Aucune tâche ou thread ne doit comporter d'attente active.
M=0;F=1;P=96;V=0	
Quelconque	

Description

On qualifie ici de boucle active une boucle permanente autour d'une activité de scrutation, ou de traitement, qui n'est jamais suspendue par une mise en sommeil ou par une attente.
Une tâche comportant une boucle active est interdite.

Justification

Une tâche comportant une boucle active est active en permanence, et risque de monopoliser la CPU au détriment des autres tâches et ainsi de bloquer l'application.
Le bon fonctionnement d'un programme qui ne respecte pas cette règle dépend alors du comportement du calculateur et de son système d'exploitation face à la gestion des tâches. Il est conditionné par le nombre de processeurs, la gestion des priorités, le partage du temps réalisé mis en place.

La fiabilité d'un tel programme est incertaine, tout au moins liée à sa machine d'exécution.

Exemple

Sans Objet

Dyn.Abort	On ne doit jamais terminer le programme brutalement par une instruction d'arrêt de tâche ou de thread (comme exit ou abort).
M=0;F=1;P=91;V=1	
Quelconque	

Description

Lorsque une instruction d'interruption est effectivement exécutée, cela provoque un arrêt brutal de la tâche. Les ressources qu'elle utilisait peuvent se retrouver dans un état incohérent ; les tâches en dépendant peuvent être avortées à leur tour brutalement.

Justification

Améliore la fiabilité des programmes, en particulier en ce qui concerne la cohérence des données et des traitements

Exemple

En ADA

L'exécution effective de l'instruction ***abort*** n'a pas lieu au moment où l'instruction est rencontrée mais est différée jusqu'au moment où elle rencontre un "check-point" tel que le début ou la fin d'une élaboration, l'instruction select, ***delay***, ... Ce différé n'est pas maîtrisable et peut conduire à un comportement non prévu.

Dyn.PrioRelatives	On ne doit pas utiliser les priorités absolues des tâches et des threads, mais plutôt les priorités relatives.
M=1;F=1;P=55;V=1	
Quelconque	

Description

La conception d'une architecture Temps Réel peut et doit être effectuée sans préjuger de l'algorithme d'ordonnancement des tâches. Le respect de cette règle garantit alors la portabilité de l'application.

Justification

Améliore l'efficacité des programmes multitâches
Améliore la portabilité des programmes multitâches

Exemple

En ADA

On n'utilise pas de pragma ***priority*** pour gérer la synchronisation des tâches.
Dans certaines applications à caractère Temps Réel, le pragma ***priority*** peut être autorisé afin de permettre une optimisation des ressources du calculateur.

Dyn.Ressources	Les ressources allouées dans un thread doivent être libérées dans le même thread.
M=0;F=3;P=38;V=0	
Quelconque	

Description

Dans le cadre d'un support au multi-threading, bons nombres de compilateurs vérifient ce point.
Dérogation : si on écrit un thread pour implémenter une fabrique d'instances : ce thread est alors uniquement chargé de la constructions des instances, qui sont détruites par les d'autres threads.

Justification

Améliore la fiabilité des applications multitâches

Exemple

En C++

Dans le cas d'une programmation sous *Windows* , l'allocation d'un objet COM/OLE par un thread doit être libéré par le thread.

Dyn.SectionCrtique	La création et l'initialisation de tâches ou de threads doivent être encapsulées ; elles doivent être effectuées dans une section critique, sans possibilité d'interruption.
M=0;F=3;P=37;V=0	
Quelconque	

Description

Sans objet

Justification

Aucun évènement ne doit venir perturber la création et l'initialisation des tâches.

Exemple

En JAVA

On appellera la méthode start() à l'intérieur de la classe.
Exemple incorrect

```
// Implementation  de l interface Runnable.
class Affichage implements Runnable {
    ...
    public void run() {
        while (true) {
           // Dessiner.
           ...
           repaint();
       }
    }
}

// Dans une autre classe, creation de l interface.
Affichage dessin = new Affichage("Arbre de Noël");

// Creation de l objet Thread.
Thread myThread = new Thread(dessin);   // pas d encapsulation
```

```
// Activation du thread.
myThread.start();                    // pas d encapsulation
...
```

Exemple correct

```
// Implementation de l interface Runnable.
class Animation implements Runnable {

    // Attribut private utilise pour stocker l identifiant
    // du thread.
    private Thread myThread;

    // Creation de l objet Thread et activation du thread.
    // Initialisations dans le constructeur.
    Animation(String name) {          // On ne specifie pas la portee
                                      // du constructeur.
        myThread = new Thread(this);  // Creation de l objet Thread
        myThread.start();             // activation du thread.
    }
    ...
}
// Creation de l animation.
// De l exterieur la façon dont l objet Animation est mis en oeuvre
// n apparait pas.
Animation coucou = new Animation("Coucou");
```

Dyn.Partage	On doit analyser avec soin les variables partagées entre threads.
M=1;F=3;P=23;V=0	
Quelconque	

Description

On analysera notamment les ressources (essentiellement les variables) partagées entre le thread principal et les threads secondaires, ainsi que les ressources partagées entre threads secondaires.
Un thread est généralement implémenté par une fonction dont le mode de lancement est asynchrone : cette règle signifie ainsi que l'on pourra manipuler dans un thread des données locales à cette fonction ou aux fonctions appelées par cette fonction.
Lors de manipulations éventuelles de variables partagées entre threads, on utilisera le mot clé volatile pour inhiber des optimisations du compilateur liées à la reconnaissance de sous expressions : attention cependant, cet attribut ne garanti par l'intégrité des données. Il faudra compléter cette déclaration par l'utilisation de sémaphores ou de *mutex*.

Justification

La non-synchronisation entre threads peut conduire à des résultats curieux ou des comportements non prévisibles concernant les ressources partagées. Cette non-synchronisation peut être très critique lors de l'allocation ou de la désallocation de ces ressources.

Exemple

Sans Objet

7.8. INTERFACES

Int.ExistenceFichier	On doit toujours vérifier l'existence ou la non existence d'un fichier avant de l'ouvrir ou de le créer ; on doit prévoir les actions à effectuer en cas d'échec.
M=0;F=2;P=66;V=1	
Quelconque	

Description

Plusieurs raisons peuvent empêcher l'accès en lecture ou écriture à un fichier.

Justification

Améliore la fiabilité.

Exemple

Sans Objet

Int.CheminFichier	Le chemin d'accès à un fichier quelconque doit être paramétré.
M=2;F=0 ;P=113;V=1	
Quelconque	

Description

Le chemin d'accès à un fichier peut être placé dans une variable d'environnement mais d'autres moyens de paramétrage peuvent être envisagés (fichiers de paramètres, …).

Justification

Facilite l'évolutivité.

Exemple

En C

```
#include   <stdlib.h>
char *Nom_Repertoire;
Nom_Repertoire = getenv ("REP_FICH_CONF"); // recuperation du chemin
// complet vers le repertoire contenant
// les fichiers de configuration
```

Int.CheminAbsolu	Les chemins d'accès ne doivent faire aucune hypothèse sur le répertoire courant.
M=2;F=1;P=56;V=0	
Quelconque	

Description

Le répertoire courant est une information volatile.

Justification

Améliore la fiabilité et la portabilité.

Exemple

En C

Les chemins vers des includes seront indépendant du répertoire de compilation ou de localisation des fichiers

En SHELL

Le répertoire courant ('.') ne doit jamais être dans le chemin de recherche utilisé par un programme SHELL

Int.Environement	Les éléments liés à l'installation d'un programme doivent être désignés par le biais de variables d'environnement spécifiques
M=2;F=0;P=83;V=0	
Quelconque	

Description

Sans objet

Justification

Améliore la portabilité.

Exemple

En WAWE

La variable d'environnement " WAVE_PATH " doit être positionnée en dehors de l'application et ne doit pas être modifiée dans des services. Cette variable d'environnement indique le (ou les répertoires) dans lequel se trouvent les modules et les services utilisables par WAVE. Les répertoires sont scrutés dans l'ordre de leur apparition dans " WAVE_PATH ". Cette variable est comparable à la variable d'environnement " PATH " utilisée sous UNIX.

Int.Temporaire	Tous les fichiers temporaires crées par l'application doivent être localisés dans des espaces dédiés et détruits au plus tard en fin d'exécution.
M=0;F=1;P=92;V=0	
Quelconque	

Description

L'exécution du programme ne doit pas polluer l'espace disque.
Il est recommandé de détruire les fichiers temporaires le plus tôt possible, surtout s'ils sont de taille importante.

Justification

L'espace disque est fini ; il est vital de le conserver

Exemple

Sans Objet

Int.FichierFermeture	Tout fichier ouvert, doit être fermé au même niveau algorithmique : module, classe, opération.
M=0;F=1;P=93;V=0	
Quelconque	

Description

Sans Objet

Justification

Cette règle permet de regrouper dans un même service les opérations d'ouverture et de fermeture de fichiers afin de s'assurer que la fermeture du fichier est bien réalisée.
La fermeture du fichier est importante car elle permet de libérer les unités logiques et donc d'ouvrir d'autres fichiers (le nombre d'unités logiques disponibles est limité).

Exemple

En PVWAWE

Les fichiers sont fermés soit avec la fonction “ CLOSE ”, soit avec la fonction “ FREE_LUN ” suivant le mode d’ouverture.
Les traitements associés aux fichiers (lecture, écriture) peuvent être effectués dans d'autres services appelés. Seules les opérations OPEN et CLOSE doivent être effectuées dans le même service.

Int.GrouperES	On doit regrouper les instructions d'entrée/sortie de même type.
M=1;F=0;P=112;V=0	
Quelconque	

Description

Il vaut mieux des instructions d'I/O longues et peu nombreuses que beaucoup de petites instructions d'I/O courtes.

Justification

Le surplus lié aux appels aux fonctions du noyau du système d'exploitation est pénalisant.

Exemple

En C

```
// incorrect
printf(" x = %f", Var_X);
printf(" y = %f", Var_Y);

// correct
printf("x = %f y = %f", Var_X, Var_Y);
```

7.9. QUALITE

Qa.Ressources	On doit rendre le logiciel indépendant des détails d'interface utilisateur par l'utilisation séparée de ressources graphiques
M=1;F=0;P=111;V=0	
Quelconque	

Description

On ne doit pas présumer des capacités graphiques des machines cibles. Dans la construction d'une interface graphique on ne fixera jamais les valeurs d'attributs graphiques de la plate-forme matérielle.

Justification

Améliore la portabilité

Exemple

En Général :

Dans la construction d'une interface graphique on ne doit jamais fixer en dur la police de caractères utilisée ni la taille des caractères.

En JAVA :

Pour déterminer la liste des polices de caractères disponibles, utiliser :

```
java.awt.Toolkit.getFontList()
```

Pour déterminer la taille d'une police de caractères, utiliser :

```
Font.getFontMetrics()
Graphics.getFontMetrics()
```

Par exemple :

```
String fontCourier = ... // Pas en dur.
titleFont = new java.awt.Font(fontCourier, Font.BOLD, 12);
titleFontMetrics = getFontMetrics(titleFont);
```

En PVWAWE :

Affecter les polices de caractères et les couleurs dans un fichier de ressources.
Utiliser des constantes pour définir les dimensions des widgets en pixels (taille, position,...).
Utiliser des constantes pour l'affectation des ressources.
Utiliser des constantes pour repérer la position des items d'un menu.

Qa.PortType	On doit veiller à la portabilité des types de base.
M=1;F=0;P=110;V=1	
Quelconque	

Description

Les types de base (numériques, caractères) sont généralement dépendants de la machine ou de l'environnement d'exécution.

Justification

Améliore la portabilité

Exemple

En C :

Les types de base (int, float) ne seront pas utilisés tels quels . En effet , la taille physique du type entier, int, dépend de la machine cible. En général elle correspond à la taille la plus naturelle sur la machine cible

En ADA :

On définira des sous types de INTEGER.
De plus, un type distinct sera défini pour chaque groupe d'entité quantifiable, avec des contraintes applicatives appropriées. Le type est alors implémenté correctement quelle que soit la machine. Si l'on souhaite que le type soit représenté de manière identique (16, 32, 64, ... bits) sur toutes machines, il faut alors ajouter une clause de représentation.

```
--  Premier exemple non autorise
procedure COMPTER_AVIONS is
NB_D_AVIONS : INTEGER := 10 ;
begin
     ...
end COMPTER_AVIONS ;


--  Deuxieme exemple autorise : on prend la peine de définir un
--  type applicatif uniquement pour compter les avions
procedure COMPTER_AVIONS is
NB_MAX_D_OBJETS : constant := 100;
type UN_COMPTEUR_D_AVION is range 0 .. NB_MAX_D_OBJETS ;
NB_D_AVIONS : UN_COMPTEUR_D_AVION := 10 ;
begin
     ...
end COMPTER_AVIONS ;
```

Qa.RepérerPort	On doit repérer les éléments non standard ou non portable utilisés, et éventuellement adapter le fonctionnement du programme.
M=1;F=0;P=108;V=0	
Quelconque	

Description

Les programmes qui doivent être exécutés sur plusieurs cibles doivent détecter et s'adapter aux cibles.

Justification

Améliore la portabilité

Exemple

En SHELL

De très nombreux aspects du fonctionnement d'un script peuvent être altérés selon le système d'exploitation sur lequel le programme s'exécute. Si un script doit être portable, il est impératif de factoriser au maximum ces dépendances et de les isoler dans un bloc d'initialisation particulier. Il peut être judicieux de créer un fichier d'initialisation/configuration spécial contenant ces dépendances, fichier qui concernera tout le projet et sera lu par chaque script.
Voici comment un script peut commencer :

```
version=`uname -r | cut -d. -f1`
case $version in
   5) # Initialisations Solaris 2.x
      …
   4) # Initialisations SunOS
```

```
        …
    *) echo Type de système `uname -a` inconnu
       exit 1
       …
esac
```

Qa.TestRetour	On doit tester systématiquement le retour des fonctions, en particulier, le retour des fonctions systèmes.
M=0;F=2;P=67;V=0	
Quelconque	

Description

Un appel de fonction ne doit jamais apparaître comme une instruction indépendante. On ne doit jamais utiliser une fonction seulement pour ses effets de bord.

Justification

Le rôle d'une fonction est de fournir une valeur lors de l'évaluation d'une expression. Le programmeur doit utiliser cette valeur de retour de la fonction : si tel n'est pas le cas, le programmeur devra utiliser une procédure et non une fonction.

Exemple

En C :

```
// correct
Etat = Control_Etat (Var_X, Var_Y, Var_Z);
(void) printf ("Etat =%d", Etat); // Tolere pour ce genre de fonctions

// incorrect
(void) Control_Etat (Var_X, Var_Y, Var_Z);
// ou
Control_Etat (Var_X, Var_Y, Var_Z);
```

Qa.Branches	Dans les instructions conditionnelles, les branches les plus fréquentes et les plus simples doivent être traitées avant les autres si l'on souhaite améliorer les performances.
M=1;F=0;P=107;V=0	
Bord	

Description

Les instructions de type choix multiples vérifient les cas en fonction de leur ordre d'apparition dans le bloc. Il est donc préférable de positionner les cas fréquents et simples avant les cas rares et complexes. Si la portion de code concernée n'est pas critique du point de vue temps d'exécution, on préfèrera utiliser l'ordre logique des cas (cf. Tr.Choix)

Justification

Améliore le temps d'exécution

Exemple

Sans Objet

Qa.Performances	Une recherche d'efficacité doit passer par l'étude des possibilités offertes par l'environnement de développement (compilateur, outils d'analyse de performances, etc.)
M=1;F=0;P=106;V=0	
Quelconque	

Description

Les " *profilers* " permettent de tracer l'exécution d'une application. On peut ensuite utiliser des outils d'analyse pour examiner les parties de l'application qui sont les plus gourmandes en ressources (mémoire ou temps d'exécution). Certains logiciels permettent également d'effectuer ces analyses de manière automatique en assurant le suivi des processus distribués.
Dans la majorité des cas, 90% du temps d'exécution est consommé par 10% seulement d'un programme, et ceci le plus souvent en des endroits où l'on ne s'y attend pas a priori. C'est donc ici qu'il faut concentrer ses efforts d'optimisation.

Justification

Améliore l'efficacité

Exemple

En C++ :

Les compilateurs proposent généralement des options spécifiques destinées à la gestion des fonctions. En particulier :
gestion des appels (*fast call*, utilisation de registres au lieu de la pile pour les fonctions non récursives, appels courts, etc.),
gestion des inline : inhibition des expansions sur option, sur condition ; recherche des fonctions à mettre inline automatiquement, etc.,
mode de représentation des pointeurs vers les fonctions membres virtuelles : définition a priori de pointeurs, prise en compte de l'héritage multiple, etc..

En JAVA :

L'option prof de l'interpréteur crée un fichier profile dans le répertoire courant que l'on peut ensuite exploiter.

Qa.Pile	On doit étudier avec précision la consommation de pile vis à vis de la quantité disponible. En particulier : les données locales, les paramètres, la profondeur de l'arbre d'appel.
M=0;F=3;P=42;V=1	
Bord	

Description

Lorsque la taille allouée à la pile est critique, on pourra préférer travailler par effet de bord sur des variables globales plutôt que d'utiliser des données locales, ou des passages de paramètres qui vont entraîner une consommation de pile. Ce choix doit être guidé par une analyse précise de la profondeur de l'arbre d'appel, sur des scénarios de type « pire cas ».
En C, C++ et ADA, on pourra également utiliser un mode de passage par adresse ou référence pour les données dont l'occupation mémoire est importante, ce qui permet d'économiser en taille de pile (la taille de l'adresse de la donnée étant inférieure à la taille de la donnée elle-même).

Justification

Améliore l'efficacité : un bon dimensionnement de pile permet d'optimiser les ressources en mémoire vive
Améliore la fiabilité en évitant un débordement de pile

Exemple

Sans Objet

Qa.Interruptions	Le traitement logiciel dédié à la prise en compte des interruptions matérielles doit être le plus bref possible.
M=0;F=1;P=93;V=0	
Quelconque	

Description

Ce traitement logiciel est appelé *handler d'interruption*. A chaque interruption est associé un *handler* ou encore *traitement sous IT*. La prise en compte d'une interruption bloque la prise en compte d'autres interruptions (hors considération de réentrance) qui peuvent être des événements importants à traiter avec une forte réactivité. Ainsi afin de ne pas monopoliser le processeur dans un état « sourd », les temps de traitement sous interruption doivent être réduits au minimum, les tâches importantes devant être déportées dans l'application hors interruption.

Justification

Améliore le temps de réponse pour une application temps réel.

Exemple

Sans Objet

Qa.Factorisation	Les sous-expressions coûteuses en temps d'exécution doivent n'être évaluées qu'une seule fois.
M=0;F=1;P=97;V=0	
Quelconque	

Description

Les expressions arithmétiques doivent être factorisées au maximum ; il faut également sortir les invariants des boucles.

Justification

Améliorer l'efficacité

Exemple

En ADA

```
Y = 3*X*X + 2*X => 5 opérations
Y = X * (3*X + 2) => 4 opérations
```

Qa.Algèbre	On doit exploiter les identités algébriques qui peuvent accélérer les calculs.
M=0;F=1;P=98;V=0	
Quelconque	

Description

Les identités algébriques peuvent être employées de façon judicieuse pour alléger certains calculs.

Justification

Améliorer l'efficacité

Exemple

En Général :

Dans la recherche du point le plus près du point (X0, Y0), rechercher le point qui minimise l'expression (X1-X0)^2 + (Y1-Y0)^2 est suffisant ; calculer la racine carrée n'est pas nécessaire.

Qa.Correlation	On doit calculer les quantités corrélées simultanément.
M=1;F=1;P=57;V=0	
Quelconque	

Description

Regrouper les calculs relatifs à une même problématique.

Justification

Améliorer l'efficacité

Exemple

En IDL :

IDL inclut des routines permettant de calculer simultanément des quantités corrélées.
On calcule la valeur maximale du tableau « array », et on en profite pour calculer simultanément la valeur minimale.

```
MaxValue = MAX(array, MIN = minValue)
```

Qa.ReutValide	On ne doit réutiliser que des services validés.
M=3;F=2;P=14;V=0	
Quelconque	

Description

Il ne faut réutiliser que des composants standards, validés et à jour.

Justification

Améliore la portabilité

Exemple

En JAVA :

Utiliser les packages JFCs - *Java Fundation Class* - de Sun, qui contiennent entre autres les classes d'interface graphique Swing et une implémentation de l'API Java 2D.
Dans le cas de contraintes projets spécifiques, les bibliothèques peuvent être réécrites
Exemple : Cela peut être le cas de la bibliothèque des fonctions mathématiques de base autres que celles fournies avec le processeur arithmétique de la cible. Cela permet à l'applicatif de maîtriser parfaitement :
le comportement numérique (qui est alors indépendant de la machine, hôte ou cible)
le nombre de niveaux d'imbrication dans l'arbre d'appel (et donc minimiser la taille de la pile d'exécution)
les performances en termes de temps de calcul.

En PERL :

Pour des développements nécessitant la possibilité actuelle ou future de s'exécuter sur des systèmes d'exploitation différents, il est recommandé de ne pas effectuer d'appels systèmes directs mais d'utiliser dans la mesure du possible les primitives Perl en général plus portables.

```
# Ne pas ecrire :
print `whoami`;
# mais :
print getlogin;
```

Qa.OptionsCompil	Dans le cas d'un langage compilé, on doit utiliser les options de compilation permettant de mettre en évidence le maximum de warning de compilation ; on doit justifier chaque non résolution de warning.
M=1;F=2;P=43;V=1	
Quelconque	

Description

Par défaut les compilateurs ne donnent pas le maximum de warning de compilation.

Justification

Améliore la robustesse

Exemple

En C

Exemple : Utiliser l'option –Wall du compilateur `gcc` permet en particulier de repérer des problèmes difficiles à déboguer : conversions implicites entre valeurs signées et non signées ou utilisation illicite de l'affectation dans un test.

En PERL

Perl fournit au programmeur un moyen de l'avertir lorsqu'il effectue dans son code des opérations non recommandées (voire interdites…). Il existe pour activer ce contrôle deux moyens : le premier est l'utilisation de l'option "`-w`" passée à l'interpréteur Perl, la deuxième solution est l'utilisation du pragma "`use warnings`".

Qa.Instrumentation	L'instrumentation du code (code, assertions) doit être réalisée à l'aide d'opérations dédiées. Si le langage ne propose pas ces opérations, on devra réaliser un module spécifique les concernant.
M=2;F=1;P=49;V=1	
Quelconque	

Description

L'instrumentation du code permet l'application de la règle Tr.ProgDefensive .
Dans le cas particulier des logiciels embarqués, l'instrumentation sera effectuée avec un soin tout particulier.

Justification

Améliore la robustesse

Exemple

En C embarqué

Sur la filière Myriade, on utilise les macros définies dans le fichier « lice.h ».

8. SYNTHESE

8.1. TABLE RECAPITULATIVE DES REGLES

Les règles sont récapitulées ici, classées par ordre alphabétique.

Id. Règle	Intitulé	Page
Don.AllocDynBord	L'allocation dynamique de mémoire est interdite.	35
Don.AllocDynSol	Si le langage supporte le concept, l'allocation dynamique de mémoire doit être utilisée avec précaution et opportunité.	36
Don.AllocEchec	Si le langage supporte le concept d'allocation dynamique, on doit prévoir systématiquement l'échec possible d'une requête d'allocation mémoire.	36
Don.AllocErreur	Une erreur survenant au cours d'un traitement ne doit pas entraîner une non libération de mémoire.	37
Don.AllocLiberation	Toute mémoire allouée doit être libérée au même niveau conceptuel.	37
Don.ChaineOper	Les opérations globales sur les chaînes de caractères (initialisation, copie, duplication, comparaison, recherche, modification) doivent être effectuées à l'aide de primitives standards fournies par le langage lorsque celles ci existent.	35
Don.Declaration	Toute donnée utilisée doit être explicitement déclarée	27
Don.Enumeration	On doit privilégier l'utilisation de constantes ou de symboles (de types énumératifs si le langage le permet) aux données numériques entières. L'utilisation de données numériques entières doit se limiter essentiellement aux calculs ou aux comptages simples.	30
Don.Homonymie	L'homonymie doit être évitée sauf dans le cas de surcharge ou de redéfinition explicite.	31
Don.Initialisation	Les variables doivent être initialisées avant leur première utilisation.	32
Don.Invariant	Des constantes doivent être définies pour les entités dont la valeur est un invariant.	30
Don.Localite	On doit privilégier les déclarations de données locales aux déclarations plus globales : les données locales à un module doivent être préférées aux données globales, les paramètres formels aux données globales, les données locales d'une opération seront préférées aux données de niveau module, les données locales d'un bloc d'instruction doivent être préférées aux données locales d'une opération.	29
Don.LocalUnique	Chaque donnée locale doit avoir une utilisation unique.	33
Don.PointeurNonAff	Si le langage supporte le concept de pointeur, lorsqu'un pointeur n'est pas associé à un objet précis lors de la déclaration, on doit préciser par un commentaire l'objet qui lui sera associé et, si le langage le permet, l'initialiser à null.	32
Don.Separee	Chaque donnée doit faire l'objet d'une déclaration séparée.	27
Don.Structure	Lorsqu'un objet conceptuel nécessite d'être implanté sous forme de plusieurs données, on doit grouper ces données dans une entité structurante (classe, structure, enregistrement, type) selon les possibilités offertes par le langage.	31
Don.TableOper	Les opérations globales sur les tableaux (initialisation, copie, duplication, comparaison) doivent être effectuées à l'aide de primitives standard fournies par le langage lorsque celles ci existent.	34
Don.TablePrincipe	On doit définir le principe de traitement des tableaux à double entrée (ligne x colonne ou colonne x ligne).	33
Don.Typage	Les données doivent être systématiquement et explicitement typées.	28
Don.TypeAnonyme	On n'utilise pas de type anonyme.	28
Don.Utilisee	Toute donnée définie doit être utilisée ; une donnée qui n'est plus utilisée doit être supprimée.	33

8.2. TRAÇABILITE « COMMUNE »

Cette table donne la correspondance entre les règles de ce document et les règles des Manuels langages. Elle comporte autant de lignes que de règles de ce document, et autant de colonnes qu'il y a de Manuels langages. Chaque cellule est vide si la règle commune n'était pas citée dans le Manuel langage ; sinon, elle contient la liste des règles du Manuel langage couverte par la règle commune.

Règle / Langage (Version)	ADA (5)	ADA BORD (3)	C (6)	C BORD (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Don.AllocDynBord		MEM (1)		EMBED-malloc								
Don.AllocDynSol	Mémoire.Allocation											
Don.AllocEchec							Excep.Allocation					
Don.AllocErreur							Excep.Libération					
Don.AllocLiberation			CO.DO-LiberDyn								MEM(1)	Pointeur.LIBERATION-MEMOIRE;Routines.VARIABLESHEAP
Don.ChaineOper										Prog_CompStr_1		
Don.Declaration			CO.DO-Vis	CO.DO-Vis	CO.DO(1);CO.DO(4)	DECL(1)				Prog_DeclVar_1		
Don.Enumeration	Types.Enumérés	DECL (7)		CO.DO-Litt	CO.DO(1);CO.DO(2)	DON(1)	Const.Define;Enum.Semantique	MAINT-Const				Constante.DEFINITION
Don.Homonymie	Identificateurs.Homonymies	IDENT (10)	CO.DO-VarRedef			NOM(2)						Routines.UNICITE-NOM
Don.Initialisation	Variables.Initialisation	DECL (9)	CO.DO-IniVar	CO.DO-IniVar	CO.DO(5)	DON(7)	Donnees.InitLocale	VARLOC-Initialisation	ENV-5		COMMON(3)	
Don.Invariant	Constantes.Définition	DECL (10)		CO.DO-Litt		DON(3)	Const.Litéraux					Expression.INVARIANT;Constante.DEFINITION
Don.Localite	Variables.Bloc	DIVERS(6)				DECL(10);DECL(14)	Donnees.Locales;Donnees.Proximité	VARLOC-Proximité;OPTIM-AccesVars	VAR-5;ENV-1		COMMON(4)	BlocCommun.EVITER;;BlocCommun.PARAMETRE-POSITIONNEL-1
Don.LocalUnique								VARLOC-Utilisation				
Don.PointeurNonAff						DECL(13)						
Don.Separee							Donnees.DeclSéparée	VARLOC-Ligne				
Don.Structure			CO.DO-Litt;CO.DO-TypConst			DON(5)						Structure.MINIMISATION
Don.TableOper	Instructions.CopieTableaux											Tableau.EGALITE
Don.TablePrincipe					CO.DO(9)						TABLEAU(5)	Tableau.PARCOURS
Don.Typage			CO.DO-TypVar	EMBED-types-util		DECL(2)						
Don.TypeAnonyme	Types.Déclaration	DECL (1)	CO.TY-Def;CO.TY-CompLit									
Don.Utilisee					CO.DO(6)	DON(9)				Prog_InitVar_1		
Dyn.Abort	Tâches.Abort	TACHE (8)			CD.SG(3)							
Dyn.AttenteActive	Tâches.BoucleActive	TACHE (6)										

Règle / Langage (Version)	ADA (5)	ADA BORD (3)	C (6)	C BORD (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Dyn.OS		NOYAU (5)					Thread.Configuration					Routines.MULTITHREADING
Dyn.Partage							Thread.Partage					
Dyn.PrioRelatives	Tâches.Priority	TACHE (3)										
Dyn.Ressources							Thread.Ressources					
Dyn.SectionCrtique		NOYAU (4)						THREAD-Encapsuler				
Err.Canal									E/S-5			
Err.FinOperation	Exceptions.Regroupement											
Err.Impression	Exceptions.Panne	EXCEPT (1)								Prog_TraceErr_1		
Err.IntegriteDonnee							Excep.Intégrité					
Err.Mecanisme	Exceptions.Panne	EXCEPT (5)	CD.TR-GestErr			EXEP(1)	Excep.Stratégie	METH-Retour	ERR-1;ERR-2;ERR-3	Prog_CodeErr_1	ERREUR(1);ERREUR(2)	Routines.COMPTERENDU;Erreurs.ON_ERROR-ON_IOERROR;Erreurs.CATCH
Err.Nom	Identificateurs.Exceptions	IDENT (9)										
Err.Operation		INSTR (5) ;EXCEPT (3)					Excep.Trait	EXCEP-CatchUtil				
Err.ToutesTraitées				EMBED-Interruptions			Excep.Terminate					
Err.TraitementDiff	Exceptions.Panne	EXCEPT (1)										
Id.ClasseType	Identificateurs.Types	IDENT (2)	CO.TY-NomTyp					NOM-Classe			STRUCT(8)	BlocCommun.NOMMAGE
Id.ConstSignif	Identificateurs.Constantes	DECL (11)	CO.DO-NomConst;CD.PP-NomMacro					NOM-Constante				
Id.Fonction	Identificateurs.Fonctions	IDENT (6)	CP.SG-NomFonc					NOM-AccèsAttribut	FICHIER-3	Nom_IdFonc_1		Routines.NOMMAGE
Id.IdentRegle	Identificateurs.Underscore	IDENT (2) ; IDENT (3) ;IDENT (12) ; FICHIER(2)			CO.PR(6)	NOM(4)	Nom.Generalités	STYLE-Langue; NOM-Défaut		Nom_idCons_1		BlocCommun.NOMMAGE
Id.IdentSignif	Identificateurs.Nommage; Identificateurs.Significativité	IDENT (1)			CO.PR(6)		Nom.Generalités	NOM-Explicite		Nom_SignifId_1	STRUCT(7)	
Id.NomDonnee												
Id.NomParFormel	Identificateurs.ParamFormels	IDENT (13)										
Id.Pointeur	Identificateurs.Pointeurs	IDENT (8)										
Id.Procedure	Identificateurs.Procédures	IDENT (5)	CP.SG-NomFonc;CD.PP-NomMacro				Nom.DonnéePrivée	NOM-AccèsAttribut	FICHIER-3			Routines.NOMMAGE
Id.Tache	Identificateurs.Procédures	IDENT (5)										
Id.VarSignif	Identificateurs.Variables	IDENT (4)	CO.DO-NomVar						VAR-2	Nom_IdVar_1		Variable.NOMMAGE1
Id.VarType	Identificateurs.Variables		CO.DO-NomVar							Nom_IdVar_1	STRUCT(8);STRUCT(9)	Variable.NOMMAGE1
Int.CheminAbsolu									ENV-3			
Int.CheminFichier	Fichier.CheminAccès		CD.IO-ParamFic									

Règle / Langage (Version)	ADA (5)	ADA BORD (3)	C (6)	C BORD (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Int.Environement								PORTAB-ConstPlat	ENV-4		STRUCT(11)	
Int.ExistenceFichier												
Int.FichierFermeture											E/S(3)	E_S.FERMETURE
Int.GrouperES			CO.IO-ESGroupe		CO.IO(3)							
Int.Temporaire									E/S-1;E/S-2			
Org.Couplage	Paquets.Couplage		CO.DO-NbVarGlob			MOD(2)					COMMON(2)	BlocCommun.PARTAGE
Org.DonneesOper	Paquets.Concept	PAQUET (1)				DECL(5)		ORGANI-Package	FICHIER-5			
Org.Duplication			CD.PP-FoncInline	EMBED-Inline			Fonction.Inline;Meta.Techniques;Meta.Codage					
Org.Masquage	Paquets.Spécification	PAQUET (3);RISQ (3)				MOD(4)	Encap.DonnéesMembres	CLASSE-ProtégerDon				
Org.MatérielIndep			CD.DV-SeparPort			PORT(2)		PORTAB-InOutErr; PORTAB-GUICapa, PORTAB-Limit		Prog_SysSpec_1		
Org.Module					CD.SG(1)	PRES(4)	Orga.Ordre;Orga.PresFonc			Pres_OrgMod_1; Pres_OrgScript_1; Pres_OrgFonc_1	STRUCT(2);STRUCT(3);STRUCT(4);STRUCT(6)	Présentation.ROUTINE; Présentation.STRUCTURES-CONTROLE; Présentation.FICHIER-BATCH;Présentation.MODULE
Org.ModuleNom	Identificateurs.Paquetages;Fichier.Nommage	IDENT (7) ; FICHIER(2)	CP.SG-RoleFic			NOM(1);MOD(1)	Nom.Fichiers		FICHIER-1	Nom_IdMod_1; Nom_IdScript_1	STRUCT(1)	Nommage.SUFFIXE
Org.MultiLang											COMM(5)	
Org.Principal					CD.SG(5)	PROG(1)						
Pr.Aeration	Présentation.Aération	DIVERS(10)	CO.EX-OpUnaire;CO.EX-OpBinaire		CO.PR(5)			DOC-MiseEnPage		Pres_Space_1;Pres_NoSpace_1;Pres_LignSep_1	STRUCT(6)	
Pr.CartDonnée			CO.PR-CommVar					DOC-MiseEnPage			STRUCT(6)	
Pr.CartStd	Présentation.Entête	PRES (7)	CP.PR-Cart		CO.PR(1);CO.PR(2)		Organ.Entête	DOC-MiseEnPage	COMT-1;COMT-2	Pres_EnteteFonc_1	STRUCT(6)	Présentation.ROUTINE
Pr.CommFonc	Commentaire.Autodoc;Commentaire.Interprétation;Types.Commentaire	COMM (1)	CO.PR-CommFonc		CO.PR(11)							Instruction.COMMENTAIRE
Pr.CommIdent	Commentaire.Indentation	COMM (2)	CO.PR-CommInd;CO.PR-CommFBlo		CO.PR(10)			DOC-MiseEnPage				
Pr.Identation	Présentation.Indentation	PRES(1)	CO.PR-Indent						FICHIER-6	Pres_Indent_1;Pres_Accolad_1;Pres_AlignCode_1;Pres_PosElse_1	STRUCT(6)	
Pr.Instruction	Présentation.InstrSimple		CO.PR-MultInstr;CO.TR-InstrLim		CO.PR(4);CO.PR(7)	PRES(3)				Pres_LongLign_1		Expression.PRESENTATION; Présentation.STRUCTURES-CONTROLE
Pr.LongLigne	Présentation.LgLigne;Présentation.Troncation	PRES (3);PRES (4)	CO.PR-LigneLim							Pres_LongLign_1; Pres_ComLong_1		

Règle / Langage (Version)	ADA (5)	ADA BORD (3)	C (6)	C BORD (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Qa.Algèbre												Expression.IDENTITE
Qa.Branches												Instruction.CASE/SWITCH-CLASSIFICATION
Qa.Correlation												Expression.REGROUPEMENT
Qa.Factorisation								OPTIM-SousExpr; OPTIM-InvBoucle				Expression.FACTORISATION
Qa.Instrumentation Qa.Interruptions				EMBED-Trait_interrup								
Qa.OptionsCompil				EMBED-OptionComp						Prog_UtilWarn_1		
Qa.Performances							Fonction.Conf	OPTIM-9010				
Qa.Pile				EMBED-Pile;EMBED-NbArg;EMBED-TaillePile;EMBED-VarAuto								
Qa.PortType	Types.Prédéfinis	DECL (3)		EMBED-Types_int_float			Type.RedefTypeBase					
Qa.RepérerPort												
Qa.Ressources								PORTAB-GUIFonts			IHM(4);IHM(8);IHM(9);IHM(10)	
Qa.ReutValide			CP.DV-Reutilisation	EMBED-Biblio				PORTAB-DependAP		Prog_PortAppelSys_1		
Qa.TestRetour		NOYAU (2)	CD.TR-RetUtil								STRUCT(16)	Routines.COMPTERENDU
Tr.Booleen	Expressions.ConditionCplxe											
Tr.BoucleSortie	Instructions.Exit		CO.TR-BreakBoucle		CO.TR(9)	FLC(5);FLC(7)			CTRL-3			Instruction.FOR-BREAK
Tr.CalculStatique	Expressions.Statiques	DIVERS (2)										
Tr.Choix	Instructions.ChoixMultiples	INSTR (1)		EMBED-switch_case	CO.TR(5)	FLC(3)						
Tr.ComparaisonStrict	Instructions.EgalitéFlottant		CO.DO-CompFloat		CO.EX(2)	EXP(2)						Variable.EGALITE
Tr.ComparConst			CO.PR-CompConst									
Tr.ControleRacc.	Expressions.CtrlRaccourcis			EMBED-for								
Tr.DoubleNeg	Expressions.DbleNégations											
Tr.FonctionSortie	SousProgr.Return	SPROG(9)	CD.TR-Sortie1		CD.SG(3)						STRUCT(15)	Routines.SORTIE
Tr.Goto	Instructions.Goto	INSTR (3)	CO.TR-Goto		CO.TR(6);CO.TR(7);CO.TR(5)	FLC(9)	Contrôle.Goto				STRUCT(18)	Instruction.GOTO
Tr.MelangeType			CO.TY-Conv		CO.TY(4)	DON(10)						
Tr.ModifCompteur			CO.TR-IndFor		CO.TR(9)	FLC(6)		CONTR-ParamFor	CTRL-2	Prog_ModForeach_1		Instruction.FOR-CONSERVATION
Tr.ModifCondSortie			CO.TR-CondFor		CO.TR(9)	FLC(6)		CONTR-ConditionFor				
Tr.ModifConst											TYPE(3)	Constante.PRESERVATION

Règle / Langage (Version)	ADA (5)	ADA BORD (3)	C (6)	C BORD (2)	FORTRAN 77 (4)	FORTRAN 90 (2)	C++ (4)	JAVA (4)	SHELL (3)	PERL (1)	PVWAWE (2)	IDL (1)
Tr.ModifParSortie		SPROG(4)		EMBED-ArgValeur ;EMBED-ArgAdresse	CO.PA(1);CO.PA(6)		Fonction.ReferConst				STRUCT(17)	VariableSystème.CONSERVATION;ParamètrePositionnel.NATURE
Tr.ModifVarGlobal	SousProgr.Fonction;SousProgr.VarGlobale	DIVERS(5)			CO.PA(8)	PAS(9)			COMT-4		STRUCT(17)	VariableSystème.CONSERVATION;Routines.MODIFICATION-PARAMETRE
Tr.OrdreChoix	Instructions.ChoixEnumération;Instructions.ChoixOthers	INSTR (2)	!CO.TR-CasDefaut			FLC(4);FLC(8)		!CONTR-Default	CTRL-1			
Tr.OrdreParFormel	SousProgr.OrdreParam	SPROG (1)		EMBED-ArgValeur	CO.PA(3)	PAS(8)					PARAM(1)	
Tr.ParamOptionnel	!SousProgr.ParamParDéfaut	!SPROG (5)				SPRO(5)					!PARAM(3);!PARAM(4);PARAM(5)	
Tr.Parenthèses	Expressions.OrdrePriorité	EXPR (1)			CO.EX(1)	EXP(1)					INSTR(1)	Expression.PARENTHESAGE
Tr.ParSortie	SousProgr.ValeurSortie	SPROG(6)										
Tr.ProgDefensive	SousProgr.TestsDéfensifs						Fonction.PrePostCond		ARGS-3;CTRL-8		STRUCT(16)	
Tr.RecursifBord		RISQ (4)										
Tr.RecursifSol	SousProgr.Récursivité					SPRO(4)						Routines.ITERATIVITE
Tr.Residus					CD.SG(4)					Pres_NettDebug_1		
Tr.TestEgalite		DIVERS(8)										

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