ACube

Règles de développement ACube

Version 1.2 du 22/02/2010

Etat : Validé

SUIVI DES MODIFICATIONS

Version	Rédaction	Description	Vérification	Date
1.0		Version initiale		01/02/06
1.1	G.Pasquereau	Intégration des règles de nommage de LISE 2.x		14/09/06
1.2	JP. Wilsch	Intégration du éléments de Lise 3.x		19/02/10

SOMMAIRE

DOCUMENTS DE REFERENCE

Version	Titre
1.4	Normes de développement Quartz
1.1	Normes de développement Jasper

1 INTRODUCTION

1.1 POURQUOI AVOIR DES NORMES

Les normes sont importantes pour les développeurs, et ce pour différentes raisons :

- la maintenance d'un logiciel représente environ 80% de son coût total,
- il est rare que la personne ayant développé le logiciel à l'origine le maintienne pendant toute sa durée de vie,
- l'utilisation des normes permet d'accroître la lisibilité du code, tout en aboutissant à une industrialisation et à une standardisation du code.

1.2 OBJECTIFS DE CE DOCUMENT

Ce document recense les normes, standards et conventions à appliquer lors des développements ACube. Ce document étant fortement lié aux technologies utilisées, certaines parties peuvent être amenées à évoluer.

2 REGLES DE DEVELOPPEMENT DE LA PARTIE SERVEUR

2.1 REGLES DE NOMMAGE

Ce chapitre décrit les règles de nommage à appliquer lors de développement de la partie serveur de l'application.

L'application de règles de nommage facilite la lecture des fichiers sources.

2.1.1 NOM DES PACKAGES

Les packages d'un projet sont répartis en deux sous ensembles :

- Une partie contenant les classes du framework Serveur,
- Une partie contenant les classes du projet.

2.1.1.1 PACKAGES DU FRAMEWORK SERVEUR

Les packages du framework serveur sont les suivants

PACKAGES	DESCRIPTION
acube.framework.action	Contient les actions de base du framework v2.x (BaseAction, Deconnecter, …)
acube.framework.exception	Contient les exceptions du framework
acube.framework.integration	Contient les classes et interfaces liées aux DAOFactory.
acube.framework.technical	Contient les Wrapper (JDBC, …), le gestionnaire de configuration, le gestionnaire de log, ...
acube.framework.utility	Contient les classes utilitaires (Gestion de dates, …)
acube.framework.vo	Contient les VO du framework (ErreurVO,…)
acube.framework.web	Contient les extensions ACube de Struts2 piour le framework v3.x

2.1.1.2 PACKAGES DU PROJET

Les packages du projet sont les suivants

PACKAGES	DESCRIPTION

acube.projet.web.action	Contient toutes les actions du projet
acube.projet.web.action.frameset	Contient toutes les actions du projet liées au menu, à l'entête, ...
acube.projet.business.service	Contient les services pour les traitements métiers
acube.projet.business.bo	Contient les objets de transfert de données
acube.projet.integration.dao	Contient les DAO
acube.projet.integration.vo	Contient les VO
acube.projet.utility	Contient les classes utilitaires (classes techniques, …)

Les petits projets (un seul item de menu de niveau 1) peuvent utiliser l'arborescence ci-dessus. Pour les plus gros projets (au moins deux items de menu de niveau 1), il faut ajouter un niveau supplémentaire en suffixant les packages du projet avec le domaine fonctionnel :

Ce niveau est obligatoire pour les packages suivants :

```
acube.projet.action.<fonction>
acube.projet.business.service.<fonction>
```

Ce niveau est facultatif pour les packages suivants :

```
acube.projet.business.bo.<fonction>
acube.projet.integration.dao.<fonction>
acube.projet.integration.vo.<fonction>
```

Le code commun à plusieurs domaines fonctionnels doit être mis dans les packages suivants

```
acube.projet.web.action
acube.projet.business.service
acube.projet.business.bo
acube.projet.integration.dao
acube.projet.integration.vo
```

2.1.2 NOM DES FICHIERS

Les fichiers JAVA ne contiennent qu'une seule classe. Ainsi, le nom du fichier et le nom de la classe sont identiques.

2.1.3 NOM DES CLASSES

Les règles de nommage des classes dans les packages sont les suivantes :

- Les noms de classes commencent par une majuscule
  - ex : **M**aClasse

- Une majuscule doit être utilisée pour séparer des mots (il ne faut pas utiliser le caractère underscore '_')
    - ex : Ma**C**lasse
- Les classes sont suffixées suivant leur nature :

CLASSES	NOM
Actions	NomClasse**Action**
Services	NomClasse**Service** (interface) NomClasse**ServiceImpl**
BO	NomClasse**BO**
Exception	NomClasse**Exception**
DAO	NomClasse**DAO** (interface) NomClasse**DAOImpl**
VO	NomClasse**VO**
Utilitaires	NomClasse

2.1.4 NOM DES METHODES

2.1.4.1 PRINCIPES

Les règles de nommage à appliquer aux méthodes sont les suivantes :

- Elles commencent par une lettre minuscule,
    - ex : **m**aMethode()
- Une majuscule est utilisée pour séparer plusieurs mots
    - ex : ma**M**ethode()
- les méthodes d'instance sont toujours accédées avec « this » à partir d'une méthode de la même classe,
    - ex : **this**.maMethodeDeClasse()

2.1.5 NOM DES VARIABLES

Les règles de nommage à appliquer aux variables, valables également pour les instances d'objet, sont les suivantes :

- Elles commencent par une lettre minuscule,
    - ex : **m**aVariable
- Une majuscule est utilisée pour séparer plusieurs mots
    - ex : ma**V**ariable

- les attributs de classe sont toujours accédés avec « this »
  - ex : **this**.monAttributDeClasse
- Les arguments des fonctions sont préfixés par « arg »
  - ex : **arg**MonArgument

> ✔ **Les règles de nommage sont indépendantes de la portée de la variable.**

2.1.6 NOM DES CONSTANTES

Les règles de nommage à appliquer aux constantes sont les suivantes :

- Elles sont toujours en lettre majuscule,
  - ex : **CONSTANTE**
- Un caractère « _ » est utiliser pour séparer plusieurs mots
  - ex : MA_CONSTANTE_PI

2.1.7 NOM DES LIVRABLES

Le formalisme des **livrables** est le suivant :

- Livrables Client :

livrable-<nomprojet>-statique-<X.Y>.zip

- Livrables Serveur :

livrable-<nomprojet>-dynamique-<X.Y>.war

- Livrables SGBD :

livrable-<nomprojet>-sgbd-<X.Y>.zip

avec <X.Y> au format suivant pour les équipes de développements:

X: version majeure

Y: version mineure

2.1.8 TABLEAU DE SYNTHESE

Elément	Règle de nommage	Exemple
Package	Le nom d'un package est toujours écrit en lettres minuscules ASCII.	`acube.projet.web.action` `acube.projet.integration`
Classe	Le nom de la classe doit être significatif, écrit en minuscules avec la première	`class Authentification` `class GererRetourErreur`

	lettre de chaque mot en majuscules. Evitez l'utilisation d'acronymes ou d'abréviations à moins qu'ils ne soient plus répandus que la forme longue du mot qu'ils désignent.	`class LaissezPasser`

Interface	La dénomination des interfaces suit la même norme que celle concernant les classes.
	
	Dans le cas d'interface utilisée comme flag, elle doit se terminer par '`able`' ou '`ible`' selon le cas.
		`interface PasseportUrgenceDAO`
		
		`interface Serialisable`
		
		`interface Lisible`

Méthode	Les noms de méthode sont explicites de leur fonction. Elles sont nommées selon leur type :
	• Verbe à l'infinitif en minuscules suivi d'un nom sur lequel il s'applique, lorsqu'il s'agit d'une action sur l'instance ou la classe (le nom est optionnel si le contexte est évident).
	• Nom de l'objet suivi de l'action lorsqu'il s'agit d'un événement.
	Le nom de la méthode doit être écrit en minuscules avec la première lettre de chaque mot en majuscules, excepté pour la première lettre du nom de la méthode.
		`run();`
		`setNom();`
		`getNom();`
		`checkAttribute();`
		`init();`

Variable	Le nom d'une variable doit être écrit en minuscules avec la première lettre de chaque mot en majuscule, excepté pour la première lettre du nom de la variable. Le nom d'une variable ne doit jamais commencer par le signe « _ » ou « $ », bien que cela soit autorisé par le langage.
	
	Le nom d'une variable doit être court et significatif. Les variables dont le nom est composé d'un unique caractère doivent être évitées, excepté pour les variables temporaires. Nous prendrons généralement pour ces variables les noms i, j, k, m ou n pour les entiers, et c, d ou pour les variables chaînes de caractères.
		`int i;`
		`char c;`
		`float maLargeur;`

Constante	Le nom des constantes (déclaration	`static final int MIN_WIDTH = 4;`

	`static final`) est toujours écrit en lettres majuscules. Lorsque différents mots sont utilisés pour le nom d'une constante, il est alors nécessaire de séparer chacun de ses mots par un « underscore ».	`static final int MAX_WIDTH = 999;` `static final int GET_THE_CPU = 1;`
Accesseur	Les méthodes d'affectation (`set`) et d'obtention (`get`) d'un attribut doivent respecter la règle de dénomination suivante `<set_or_get_prefixe><Attribut>`	`setNom();` `getNom();`
Prédicat	Une méthode renvoyant l'état d'un objet sous forme d'un booléen (que ce soit en interrogeant un attribut ou par calcul) sera préfixée par `is`.	

2.2 TAILLE LIMITE DES ELEMENTS

Afin de conserver une bonne lisibilité du code source, la taille des éléments est limitée.

2.2.1 TAILLE DES FICHIERS

Un fichier (classe java ou autre) doit contenir moins de **2000 lignes**.

2.2.2 TAILLE DES METHODES

Une méthode doit contenir moins de **150 lignes**.

2.2.3 TAILLE DES LIGNES

Dans un fichier, une ligne doit contenir moins de **80 caractères**.

2.2.4 NOMBRE D'ARGUMENT D'UNE METHODE

Une méthode doit accepter au plus **7 paramètres**.

2.3 ORGANISATION DU CODE

Afin de conserver une bonne lisibilité du code source, le code est constitué d'un ensemble de sections séparées par des lignes vides.

2.3.1 PRESENTATION DU CODE

2.3.1.1 CARACTERES

Un source doit être saisi dans un **format texte ASCII**, les caractères accentués sont acceptés dans les commentaires.

2.3.1.2 LIGNES VIDES

L'utilisation de lignes vides permet d'augmenter la lisibilité du code par regroupement de code apparenté.

- Deux lignes vides doivent être laissées dans les cas suivants :
  - entre les différentes sections d'un même fichier source ;
- Une ligne vide doit être laissée dans les cas suivants :
  - entre les méthodes
  - avant un commentaire de type de « bloc », c'est-à-dire sur plusieurs lignes, ou « simple », c'est-à-dire sur une seule ligne (Cf. chapitre 2.4)

2.3.1.3 CARACTERE ESPACE

Les espaces doivent être utilisés dans les cas suivants :

- Un espace doit être laissé entre **un mot-clé et une parenthèse** comme ci-dessous :

```
while (true){
…
}
```

exception faite de cette règle **pour les noms de méthodes** qui sont collés à la parenthèse suivante.

- Un espace doit être laissé après chaque virgule dans une liste d'arguments ;
- Un espace doit être laissé entre les opérateurs binaires (excepté le « . ») et leur opérande. Par contre il ne faut pas laisser d'espace entre les instructions d'incrémentation et de décrémentation :

```
a += c + d;
while (d++ = s++) {
n++;
}
System.out.println("la taille est :" + n);
```

- Un espace doit être laissé entre les différentes expressions dans une instruction `for` :

```
for (expr1 ; expr2 ; expr3)
```

- Les instructions de `cast` doivent également être suivies d'un espace :

```
maMethod((byte) aNum, (Object) x) ;
maMethode((int) (cp + 5), ((int) (i + 3)) + 1);
```

2.3.1.4 INDENTATION

L'unité d'indentation est de quatre espaces. L'éditeur de l'environnement de développement doit être paramétré dans ce sens.

2.3.1.5 RETOUR À LA LIGNE

Lorsqu'une expression ne peut être écrite sur une seule ligne, écrivez-la sur différentes lignes en respectant les principes suivants :

- retour à la ligne après une virgule :

```
maMethod(longExpression1, longExpression2, longExpression3,
longExpression4, longExpression5) ;
```

- retour à la ligne avant un opérateur :

```
longName1 = longName2 * (longName3 + longName4 – longName5)
            + 4 * longname6;
```

- préférer les retours à la ligne de haut niveau par rapport à ceux d'un niveau inférieur

```
longName1 = longName2 * (longName3 + longName4 – longName5)
            + 4 * longname6; // OK
longName1 = longName2 * (longName3 + longName4
                          – longName5) + 4 * longname6; // A éviter
```

- aligner la nouvelle ligne avec le début de l'expression de même niveau de la ligne précédente
- si les règles ci-dessus peuvent amener à confondre des parties de code, ou bien à ce qu'une partie de code soit écrasée contre la marge de droite, faire alors une simple indentation de 4 espaces.

```
private static synchronized longMethodName(int unArg,
    Object unAutreArg, String encoreUnAutreArg,
    Objet etEncore UnAutreArg) {
```

2.3.2 LES FICHIERS SOURCES JAVA

On doit retrouver dans un fichier source JAVA les éléments suivants ainsi ordonnés :

1. Package et Import
2. Commentaires de début
3. Déclaration de classe et d'interface

2.3.2.1 PACKAGE ET IMPORT

La première ligne non commentée d'un fichier JAVA est le package. Ensuite, nous devons avoir les Imports. Par exemple :

```
package acube.projet.rules;

import org.apache.log4j.Logger;

import acube.framework.exception.DAOException;
import acube.framework.vo.ErreurVO;

import acube.projet.passeport.business.DocumentSecurisePasseportDelegate;
```

Prendre l'import explicite de classe et non pas l'import de l'ensemble du package pour tous les imports de bibliothèques externes et classes internes (`import org.apache.log4j.Category; import . acube.framework.exception;`).

Généralement, nous ferons trois séries d'imports, chacune séparée de la précédente par une ligne vide :

- classes du JDK (java.*, javax.*, ...) ;
- classes de bibliothèques externes (log4j, jakarta-regexp, ...) ;
- classes développées en interne (acube.framework.*, ...)

Dans chaque série, classer les imports par ordre alphabétique.

2.3.2.2 DECLARATION DE CLASSE ET D'INTERFACE

Le tableau suivant décrit les différents éléments de la déclaration d'une classe ou d'une interface que l'on doit retrouver dans l'ordre indiqué ci-dessous :

Elément de la déclaration de la classe ou de l'interface	**Description**
Commentaire de classe ou d'interface (`/**. . . */`)	Voir le chapitre 2.4 sur les commentaires
`class` ou `interface`	
Commentaire d'implémentation de la classe ou de l'interface, seulement si nécessaire (`/*. . . */`)	Ce commentaire contient toutes les informations qui ne doivent pas figurer dans le commentaire de classe ou d'interface
Attributs de classe (`static`)	D'abord apparaissent les attributs de classe `public`, puis les variables `protected`, celles de niveau `package`, puis les `private`.
Attributs d'instance	D'abord apparaissent les attributs `public`, puis les variables `protected`, celles de niveau package, puis les `private`.
Constructeurs	
Méthodes	Les méthodes doivent plutôt être regroupées en fonction de leur finalité plutôt qu'en fonction de leur signature ou de leur accessibilité. Ainsi, une méthode privée d'une classe peut être entre deux méthodes publiques d'instance. Le but est de rendre la lecture et la compréhension du code plus simple.

2.4 LES COMMENTAIRES

2.4.1 BALISE JAVADOC

Les commentaires JavaDoc, dits de documentation décrivent les classes, les interfaces, les constructeurs les méthodes ainsi que les différentes variables membres. Chaque commentaire est inclus dans le délimiteur suivant : /** . . . */, avec un commentaire par classe, interface ou variable membre.

Ce type de commentaire doit apparaître juste avant la déclaration.

```
/**
```

```
 * Cette classe Exemple permet de …
 */
public class Exemple { . . .
```

La première ligne d'un commentaire de documentation de déclaration d'une classe ou bien d'une interface n'est jamais indentée car la déclaration de la classe ou bien de l'interface ne l'est pas également. Tous les autres types de commentaires de documentation sont indentés de manière normale.

Quel que soit le type de commentaire de documentation, les lignes sont indentées d'un espace par rapport à la première et à la dernière ligne du commentaire de manière à ce que toutes les astérisques soient alignées au même niveau.

Par défaut, JavaDoc prend en compte les éléments suivants : les classes, les interfaces, les méthodes et les champs publics et protected

Il est possible d'utiliser des tags HTML pour formater le texte : il ne faut pas utiliser de tags HTML de structure tel que Hn, HR ... qui sont utilisés par JavaDoc pour formater la documentation.

Enfin il est possible d'utiliser des tags prédéfinis par JavaDoc pour fournir des informations plus précises sur des composants particuliers (auteur, parametres, valeur de retour). Ces tags commencent tous par @.

On distingue 2 types de tags :

- les tags standards, dont la syntaxe est : @tag

- les tags qui seront remplacés par une valeur, dont la syntaxe est : {@tag}

Pour pouvoir être interprétés les tags standards doivent obligatoirement commencer en début de ligne.

Tag JavaDoc	Rôle	Elément concerné	Version du JDK
Tags JavaDoc standards			
@author	permet de préciser l'auteur de l'élément **= nom de la société pour une SSII et nom de la personne pour un agent du MAE**	Classe et interface	1.0
@deprecated	permet de préciser qu'un élément est déprécié	Package, Classe, Interface, Méthode, Attribut	1.1
{@docRoot}	représente le chemin relatif du répertoire principal de génération de la documentation		1.3
@exception	permet de préciser une exception qui peut être levée par l'élément	Méthode	1.0
{@link}	permet d'insérer un lien vers un élément de la documentation	Package, Classe, Interface, Méthode, Attribut	1.2
@param	permet de préciser un paramètre de l'élément	Constructeur et méthode.	1.0
@return	permet de préciser la valeur de retour d'un élément	Méthode	1.0

@see	permet de préciser un élément en relation avec l'élément documenté	Package, Classe, Interface, Méthode, Attribut	1.0
@throws	identique à @exception	Méthode	1.2
Voir aussi http://java.sun.com/j2se/javadoc/proposed-tags.html, http://java.sun.com/j2se/javadoc/writingdoccomments/index.html#annotations			

- Ne pas utiliser de caractères spéciaux dans les commentaires
- Si de l'information sur une classe, une interface, une méthode ou une variable doit être donnée et que cette information ne rentre pas dans la catégorie des informations que l'on retrouve habituellement dans des commentaires de documentation, il faut alors utiliser un commentaire d'implémentation de type « block » ou « single-line » juste après la déclaration correspondante et ne pas inclure ces informations dans un commentaire de documentation.
- Aucun commentaire de documentation ne doit se retrouver à l'intérieur d'une méthode ou d'un constructeur car Java associe ce type de commentaire avec la première ligne suivant le commentaire.

2.4.2 ENTETE DE CLASSE

L'entête de classe est un commentaire javadoc permettant de décrire l'utilité de cette classe.

Cet entête comprend :

- Le nom du projet,
- La date de réalisation de cette classe,
- l'auteur de la classe,
- une description de la classe.

```
/**
 * Projet PHILEAS
 * @date   03/2005
 * @author Michel MARTIN
 *
 * La classe abstraite BaseAction est la classe mère de toutes les
 * classes du package action.
 * Elle définit les méthodes "executeAtStart" et "executeAtEnd" permettant de
 * rendre transparent la gestion des logger, des listes de beans,
 * du chargement des feuilles de styles et de la validité de la session courante
*/
public abstract class BaseAction extends Action {
```

Exemple d'entête de classe

2.4.3 ENTETE DE METHODE

L'entête de méthode est un commentaire javadoc permettant de décrire le fonctionnement de la méthode.

Cet entête comprend :

- une description de la méthode,
- la liste détaillée des paramètres de la méthode,
- une description de l'objet retourné par la méthode,
- La liste des exceptions renvoyées par la méthode.

```
/**
 * Méthode appelée au début de l'action, pour effectuer les différentes
 * initialisations
 *
 * @param mapping ActionMapping
 * @param request HttpServletRequest
 * @param response HttpServletResponse
 * @param activite String feuille xsl à utiliser pour cette action
 * @return HttpSession session http valide de l'utilisateur connecté
 * @throws AuthentificationException exception d'authentification
 *         si l'utilisateur n'a pas de session valide
 */
 protected HttpSession executeAtStart(ActionMapping mapping,
                                      HttpServletRequest request,
                                      HttpServletResponse response,
                                      String activite)
     throws AuthentificationException {
 …
}
```

Exemple d'entête de méthode

2.4.4 ENTETE D'ATTRIBUT

Les commentaires des attributs sont des commentaires javadoc. Ils permettent d'expliquer le rôle de l'attribut.

```
    /** <code>_xslFile</code> feuille xsl à utiliser pour cette action */
    protected String xslFile ;
```

Exemple d'entête de variable

2.4.5 DANS UNE METHODE

Les commentaires présents dans les méthodes doivent permettre de comprendre pourquoi les choses ont été ainsi faites et non pas seulement comment.

Ces commentaires ne sont pas des commentaires javadoc, mais n'en sont pas moins indispensables.

Ils doivent permettre aux développeurs de comprendre l'enchaînement du code.

Ces commentaires sont les commentaires dits d'implémentation, délimités par /* …*/.

Les commentaires d'implémentation permettent de commenter le code ou une implémentation particulière. Les commentaires de documentation décrivent les spécificités du code, et sont destinés à être lues par des développeurs qui ne disposent pas forcément du code source.

2.4.5.1 COMMENTAIRES SUR UNE SEULE LIGNE

```
        // vérification de l'encodage
        updateEncoding(request, getLocale(request));
```

Exemple de commentaire dans une méthode

2.4.5.2 COMMENTAIRE SUR PLUSIEURS LIGNES

```
       /* chargement du contenu du fichier de configuration de strutscx
        * NOTE: le contenu de ce document est stocké dans le contexte de
        * la servlet, sous le nom StrutsCXConstants.PROPERTIES
        */
        strutscx_config = (Document) servlet.getServletContext().
                                    getAttribute(StrutsCXConstants.PROPERTIES);
```

Exemple de commentaire dans une méthode

2.4.6 FORMAT DE COMMENTAIRES D'IMPLEMENTATION

Il existe quatre styles de commentaires d'implémentation : « block », « single-line », « trailing » et « end-of-line ».

2.4.6.1 COMMENTAIRES D'IMPLEMENTATION DE TYPE « BLOCK »

Les commentaires de type « block » doivent être indentés au même niveau que la partie du code que ces commentaires décrivent. Un commentaire de type « block » doit toujours être précédé d'une ligne vide afin de le séparer du reste du code.

```
/*
 * Voici un commentaire de type « block »
 */
```

2.4.6.2 COMMENTAIRES D'IMPLEMENTATION DE TYPE « SINGLE-LINE »

Il s'agit de commentaires assez courts indentés au même niveau que la ligne suivante de code. Si le commentaire est trop long pour être écrit sur une seule ligne, il convient alors d'utiliser un commentaire de type « block ».

```
If (condition) {

    /* Insérez ici le commentaire. */
    . . .
}
```

2.4.6.3 COMMENTAIRES D'IMPLEMENTATION DE TYPE « TRAILING »

Des commentaires très courts peuvent apparaître sur la même ligne que le code qu'ils décrivent, mais de manière assez éloignée du code de manière à ne pas confondre le commentaire avec d'autres instructions.

Si plus d'un commentaire très court doivent apparaître dans une même partie du code, ils doivent tous être indentés au même niveau.

```
If (a == 2) {
    return TRUE ;      /* premier commentaire */
} else {
    return isPrime(a) ;       /* deuxième commentaire */
}
```

2.4.6.4 COMMENTAIRES D'IMPLÉMENTATION DE TYPE « END-OF-LINE »

Le symbole « // » peut être utilisé pour commenter une ligne. Il ne doit pas être utilisée sur plusieurs lignes consécutives, sauf lorsque l'on souhaite commenter plusieurs lignes de code.

```
if (condition) {

    // Insérez ici le commentaire.
...
} else {
    …   // Expliquez pouquoi.
}
```

Les commentaires doivent donner une vue globale du code et apporter une information supplémentaire non lisible directement dans le code lui-même.

Les commentaires doivent contenir de l'information appropriée à la lecture et à la compréhension du programme. Ainsi, des informations telles que la manière dont le package est construit ou bien le répertoire où ce dernier se trouve ne devraient pas apparaître dans un commentaire. Éviter de faire apparaître dans les commentaires des choses évidentes clairement lisibles dans le code.

2.5 LES INSTRUCTIONS

2.5.1 INSTRUCTIONS SIMPLES

Chaque ligne doit contenir une seule instruction. De même, lors des déclarations de variables, on ne fera figurer qu'une seule déclaration par ligne.

```
argv++;         //correct
argc++;         //correct
argv++; argc--;//incorrect

int     pos;    // position dans le tableau
int     taille = 3 ;    // taille du tableau
```

Ne pas écrire :

```
int pos, taille = 3;
```

2.5.2 Instructions imbriquées

- Les instructions imbriquées sont celles contenant une liste d'autres instructions incluses dans des accolades.

Les instructions imbriquées doivent être indentées d'un cran de plus par rapport à l'instruction imbriquée de niveau supérieur.

- L'accolade ouvrante doit se situer à la fin de la ligne débutant une nouvelle instruction composée. L'accolade fermante doit se trouver seule sur une ligne et être indentée au même niveau que la ligne contenant l'accolade ouvrante lui correspondant.
- Les accolades sont utilisées autour des instructions, même lorsqu'il s'agit d'instructions simples, lorsque ces dernières appartiennent à une structure de contrôle (`if-then-else` par exemple). Ceci rend plus simple l'ajout de nouvelles instructions.

2.5.3 Instruction "return"

Une instruction `return` renvoyant une valeur ne doit pas utiliser de parenthèses, à moins que cela ne permette de mieux faire ressortir la valeur retournée.

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

2.5.4 Les Blocs

Afin de rendre le code plus lisible, les blocs de type **if-else**, **do-while**, **for**, **try-catch**, **switch-case** … doivent respecter les règles suivantes :

- Les instructions à l'intérieur d'un bloc doivent systématiquement être entourées par des accolades, même dans le cas d'une instruction unique,
- L'accolade ouvrante doit être sur la même ligne que le mot clé du bloc,
- l'accolade fermante doit être sur une ligne seule, ou suivi d'un autre mot clé de boucle, et être indenté au même niveau que le bloc.

```
// Exemple de boucle if
if (condition) {
    instruction1;
} else {
    instruction2;
}
```

```
// Exemple de bloc switch
switch (i) {
        case 1:
                instruction1;
                /* commentaire */
        case 2:
```

```
                instruction2;
                break;
        case 3:
                instruction3;
                break;
        default:
                instruction4;
                break;
}
```

Pour les cas ne contenant pas de break, ajouter un commentaire.

Penser également à toujours indiquer un cas par défaut.

```
// Exemple de boucle for
for (initialisation; condition; mise à jour) {
    instructions ;
}
```

Une instruction `for` sans instructions, c'est-à-dire une boucle où l'ensemble des traitements sont réalisés dans l'initialisation, la condition ou la clause de mise à jour doit avoir la forme suivante :

```
for (initialisation; condition; mise à jour);
```

```
// Exemple de boucle while
while (initialisation) {
    instructions ;
}
```

Une instruction `while` sans instructions doit avoir la forme suivante :

```
while (condition);
```

```
// Exemple de boucle do-while
do {
    instructions ;
} while (condition) ;
```

```
// Exemple de bloc try-catch
try {
    instructions ;
} catch (ExceptionClass e) {
    instructions;
}
```

```
try {
    instructions ;
} catch (ExceptionClass e) {
    instructions;
} finally {
    instructions;
}
```

Dans le deuxième cas, le bloc finally est toujours exécuté, quel que soit le résultat de l'exécution du bloc try.

2.6 BONNES PRATIQUES

2.6.1 ACCÈS AUX VARIABLES DE CLASSES ET D'INSTANCE

- Ne rendez pas visible une variable de classe ou d'instance publique sans une bonne raison. Souvent, les variables d'instance ne nécessitent pas d'être écrites ou lues (autrement que via les getter et les setter).
- Un des cas où l'utilisation de variables publiques d'instance se justifie est celui où la classe est essentiellement utilisée comme structure de données, sans avoir de réel comportement.
- L'accès aux attributs d'instance et de classe doit se faire par des accesseurs (fonctions `get` et `set`) et non directement. Ceci est fait pour des raisons évidentes d'encapsulation de code.
- Afin de limiter les points d'entrée aux attributs, les méthodes internes à la classe doivent aussi passer par ces accesseurs.

2.6.2 RÉFÉRENCE AUX VARIABLES DE CLASSES ET AUX MÉTHODES

Eviter l'utilisation d'un objet pour accéder à une variable (statique) de classe ou à une méthode. Utiliser plutôt le nom de la classe.

```
classMethod();                  //OK
AClass.classMethod();           //OK
anObject.classMethod();                 //A EVITER!
```

2.6.3 CONSTANTES

Les constantes numériques ne doivent pas apparaître en dur dans le code, excepté pour –1, 0 et 1 qui peuvent par exemple apparaître comme compteur dans une boucle `for`. Il est préférable de déclarer des constantes, en leur donnant un nom représentatif : public static final int MAX_ = 12 ;

2.6.4 VALORISATION DES VARIABLES

Evitez de valoriser différentes variables avec la même valeur dans une seule instruction. Cela rend le code difficile à lire.

```
fooBar.fChar = barFoo.lchar = 'c'; // A EVITER !
```

N'utilisez pas l'opérateur d'affectation à un endroit où il peut facilement être confondu avec égal.

2.6.5 PARENTHÈSES

Pensez à utiliser des parenthèses dans les expressions utilisant différents opérateur de manière à éviter tout problème de priorité d'opérateur.

```
if (a == b && c == d)          // A EVITER !
if ((a == b) && (c == d))      // OK
```

2.6.6 REUTILISATION DU CODE

Pour des raisons de lisibilité, de performance et de maintenabilité :

Eviter les instructions inutiles. Penser à factoriser le code.

Une règle de gestion ne doit être définie que par une seule méthode d'un seul objet. Il est toutefois possible qu'une même méthode implémente différentes règles de gestion.

2.6.7 CLASSE STRING ET STRINGBUFFER

- Pour la comparaison de String, utiliser de préférence :

equals() et equalsIgnoreCase() qui retournent un booléen, à compareTo() et compareToEqualsIgnoreCase() qui retournent un entier.

- Pour la concaténation de chaînes de caractères (par exemple pour générer une requête SQL), utiliser un objet StringBuffer à la place d'un objet String pour des questions de performance :

```
StringBuffer sReqSql = new StringBuffer( "SELECT utilisateur_nom");
sSqlReq.append( " FROM Table");
sSqlReq.append( " WHERE utilisateur_id=" + m_nId);
```

2.6.8 REGLES DE DEVELOPPEMENT LIEES A L'UTILISATION D'OUTILS OPEN SOURCE

L'utilisation des outils Open Source Quartz et Japser Reports, coté serveur, dans les applications ACube, suivent des normes de développement précises, décrites dans les documents de référence :

- Normes de développement Quartz
- Normes de développement Jasper

3 REGLES DE DEVELOPPEMENT DE LA PARTIE CLIENT RICHE

3.1 REGLES DE NOMMAGE

Ce chapitre décrit les règles de nommage à appliquer lors de développement de la partie client de l'application.

3.1.1 NOM DES FICHIERS

L'arborescence du client riche est la suivante :

REPERTOIRES	DESCRIPTION
css	Contient les feuilles de style du projet
divers	Contient les autres éléments (tutorial, svg, …)
flux	contient tous les éléments dynamiques bouchonnés (flux XML, PDF, XSL, DOC, …)
flux/protected	contient tous les éléments dynamiques bouchonnés soumis à droits d'accès (en version non bouchonnée)
frameworkErgo	Contient le framework ergonomique
html	contient les pages HTML du projet
images	contient les images du projet
jsclient	contient les fonctions javascripts du projet
xml	contient les XML statiques du projet

Un sous répertoire par fonction doit être créé dans les répertoires flux, flux/protected, html, jsclient et xml, et pour un écran donné, les fichiers HTML, JS et XML doivent être nommés de façon identiques.

Exemple de nommage des fichiers pour la partie client :

```
Client
|
+---flux
|   +---protected
|   |   +---fonction1
|   |   |   +---ecran1
|   |   |   |       action1.xml
|   |   |   |       action2.xls
|   |   |   |
|   |   |   |---ecran2
|   |   |   |       action3.xml
|   |   |   |       action4.pdf
|   |   |   |
+---html
|   +---fonction1
|   |       ecran1.html
|   |       ecran2.html
|
+---jsclient
|   +---fonction1
|   |       ecran1.js
|   |       ecran2.js
|
\---xml
    +---fonction1
    |       ecran1.xml
    |       ecran2.xml
```

3.1.2 NOM DES FONCTIONS

Les règles de nommage à appliquer aux fonctions javascripts sont les suivantes :

- Elle commence par le nom du fichier javascript suivi du caractère '_'
  - ex : **situationLocale_**preparePage()
- Elles commencent par une lettre minuscule :
  - ex : **s**ituationLocale_preparePage()
- Une majuscule est utilisée pour séparer plusieurs mots
  - ex : situation**L**ocale_prepare**P**age()

3.1.3 NOM DES BALISES

Les balises HTML doivent être écrites en minuscule..

3.2 ORGANISATION DU CODE

3.2.1 PRESENTATION DU CODE

Le code JavaScript doit être placé dans des fichiers .js séparés.

3.2.1.1 CARACTERES

Un source doit être saisi dans un format texte ASCII.

Les caractères accentués sont acceptés dans les commentaires pour les fichiers .js.

Pour accepter les caractères accentués dans les fichiers .xml et .html sans avoir à utiliser les caractères spéciaux lors de l'encodage, il faut indiquer que le codage est à réaliser avec un jeu de caractères à la norme iso-8859-1 (norme qui prend en charge les accents), par la ligne suivante :

- dans les fichiers .html :

```
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
```

qui signale que le document à traiter est un document texte et que le codage est réalisé avec un jeu de caractères à la norme iso-8859-1

- dans les fichiers .xml :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

3.2.1.2 LIGNES VIDES

L'utilisation des lignes vides s'applique aux fichiers .html et .js. Elle suit les règles décrites pour JAVA au chapitre 2.3.1.2.

3.2.1.3 CARACTÈRE ESPACE

L'utilisation des espaces dans les fichiers .js suit les règles décrites pour JAVA au chapitre 2.3.1.3.

Pour les balises html qui n'ont pas de balise de fermeture : c'est le cas de la balise
 pour les sauts de ligne, et de la balise <img> permettant l'insertion d'une image, nous vous conseillons d'utiliser la norme xml qui consiste à terminer la ligne par un espace suivi de />.

```
<img vos_attributs />
```

Pour les commentaires html, veillez à bien placer un espace avant la marque finale -->, cf chapitre 3.5

3.2.1.4 INDENTATION

L'unité d'indentation pour les fichiers .html, .js, .xml suit la règle décrite pour JAVA au chapitre 2.3.1.4.

N.B : à l'exception des fichiers xml contenant les flux informatifs bouchonnés des tableaux (sous /flux) qui ne doivent pas être indentés.

3.2.1.5 RETOUR À LA LIGNE

Pour les fichiers .js, les règles de retour à la ligne suivent celles décrites pour JAVA au chapitre 2.3.1.5.

Pour les fichiers .html, nous conseillons pour les balises allant par paires :

- d'écrire les balises d'ouverture et de fermeture en allant toujours à la ligne (votre page sera bien plus claire et plus facile à corriger par la suite).
- d'aligner la ligne d'une balise de fermeture à la ligne de sa balise d'ouverture

```
<nom_de_la_balise>
</nom_de_la_balise>

<body>
    <table>
        <p>
        </p>
    </table>
</body>
```

3.3 LES COMMENTAIRES

3.3.1 BALISE JSDOC

Les balises JsDoc utilisées sont les suivantes :

On distingue 2 types de tags :

- les tags standards, dont la syntaxe est : @tag

- les tags qui seront remplacés par une valeur, dont la syntaxe est : {@tag}

Tag	Rôle	Elément concerné	JSDoc
@fileoverview	permet de générer un Résumé pour le fichier .js	Fichier source .js	1.9.6
@author	permet de préciser l'auteur	Fichier .js, Classe, Méthode	1.9.6
@class	permet de générer une description pour la classe	Classe	1.9.6
@constructor	permet de générer une description du constructeurs	Classe	1.9.6
@param	permet de préciser un paramètre de l'élément Ex : La syntaxe utilisée pour la JSDoc du framework JavaScript ACUBE est : **@param {<type>}** <nomParam> <descriptif>	Constructeur et méthode.	1.9.6
@extends	permet d'indiquer une classe en sous-classe d'une autre classe.	Classe	1.9.6
@private	permet de qualifier l'accessibilité d'un élément 'privé' Attention : pour faire apparaître les éléments qualifiés '@private' dans la JSDoc Html, ne pas oublier l'**option '--private' en appel du script**	Classe, Attribut, Méthode	1.9.6

	perl.		
@type	permet d'indiquer le type du retour de la fonction ou de l'attribut	Attribut et retours	1.9.6
{@link}	permet d'inserer un lien vers un élément de la documentation : Ex : Classe ⇨ **{@link <NomClasse>** texte**}**, Attribut ⇨ **{@link #<nomAttribut>** texte**}**, Méthode ⇨ {@link #<nomMethode> texte} Element d'une autre classe ⇨ **{@link <NomClasse>#<**nomAttribut | nomMethode>**}**	Fichier .js, Classe, Méthode, Attribut	1.9.6
@see	permet de préciser un élément en relation avec l'élément documenté : Ex : 'Voir' Classe **@see** ElementListe 'Voir' Elément attribut ou méthode **@see #**ecrireBoiteHtml 'Voir' Elément attribut ou méthode d'une autre classe **@see ComposantEntete#**setDivBind	Fichier .js, Classe, Attribut, Méthode	1.9.6
@return	permet de préciser la valeur de retour d'un élément	Méthode	1.9.6
Voir aussi la référence des tags sur : http://jsdoc.sourceforge.net/#tagref.			

Il est aussi possible d'utiliser des tags HTML pour formater le texte : il ne faut pas utiliser de tags HTML de structure tel que Hn, HR ... qui sont utilisés par JSDoc pour formater la documentation.

3.3.2 ENTETE DE FICHIER

L'entête de fichier est un commentaire jsdoc permettant de documenter les fichiers javascripts.

Cette entête comprend les éléments suivants :

- une description fonctionnelle le l'écran,
- la liste des contraintes pour chacun des champs,
- la liste des contrôles de surface,
- les flux statiques,
- les flux dynamiques,
- la navigation possible depuis cette page,
- le nom du projet,
- l'auteur du fichier.

```
/**
 * @fileoverview
 * <pre>
 * Cet écran permet de définir ou de consulter, selon la période en cours,
 * les dates du calendrier de la campagne boursière.
 *
 * <b><u>Contraintes:</u></b>
 * ------------------------------------------------------------------------
 * Champs concerné                  | Description de la contrainte
 * ---------------------------------|--------------------------------------
 * Lancement de la campagne           Ce champs possède au plus 10 caratères
 * Limite de réception des dossiers   Ce champs possède au plus 10 caratères
 * Réunion le                         Ce champs possède au plus 10 caratères
 *
 * <b><u>Contrôles de surfaces:</u></b>
 * ------------------------------------------------------------------------
 * Donnée concernée                 | Obligatoire | Description
 * ---------------------------------|-------------|-------------------------
 * Lancement de la campagne              N          Date au format jj/mm/aaaa
 * Limite de réception des dossiers      N          Date au format jj/mm/aaaa
 * Réunion le                            N          Date au format jj/mm/aaaa
 *
 * <b><u>Flux Statiques:</u></b>
 * - /xml/gc/calendrier.xml
 *
 * <b><u>Flux dynamiques:</u></b>
 * - /flux/protected/gc/calendrier/editerCalendrier.xml
 * - /flux/protected/gc/calendrier/modifierCalendrier.xml
 * - /flux/protected/gc/calendrier/validerDateLancementCampagne.xml
 *
 * <b><u>Navigation à partir de cet écran:</u></b>
 * - /html/gc/gestionCampagne.html
 * </pre>
 * Projet SCOLA (10/2005 -> 05/2006)
 * @author Michel MARTIN
 */
```

exemple d'entête de fichier javascript

3.3.3 ENTETE DE FONCTION

L'entête de fonction est un commentaire jsdoc permettant de décrire l'utilisation de la fonction

Cette entête comprend :

- une description de la fonction,
- la liste détaillée des paramètres de la fonction,
- une description de l'objet retourné par la fonction,

```
/**
 * Fonction retournant la valeur du paramètre "name" présent dans la chaine "txt"
 * @param {String} txt La chaine de caractères dans laquelle il faut extraire des
 *                     paramètres particuliers.
 * @param {String} name Le nom du paramètre auquel on extrait sa valeur.
 * @param {String} sep Le séparateur dans la chaine de caractères.
 * @return La valeur du paramètre "name" contenu dans la chaine "txt" et séparé
 *          par le séparateur "sep".
 * @type String
 */
function getValue(txt,name,sep)
{
  var rg = new RegExp(name+"=([^"+sep+"]*)");
  var r = rg.exec(txt);
  return (r?unescape(r[1]): null);
}
```

exemple d'entête de fonction javascript

3.3.4 ENTETE DE VARIABLE

Les entêtes de variables permettent de décrire fonctionnellement une variable. Il ne s'agit pas d'un commentaire jsdoc.

```
// periode de la campagne boursière courante
var periodeCampagne = null;
```

exemple d'entête de variable javascript

3.3.5 DANS UNE METHODE

Les commentaires présents dans les méthodes doivent permettre de comprendre pourquoi les choses ont été ainsi faites et non pas seulement comment.

Ces commentaires ne sont pas des commentaires jsdoc mais n'en sont pas moins indispensables.

Ils doivent permettre aux développeurs de comprendre l'enchaînement du code.

```
// En dehors de la période "clôture", la fiche est en modification
var defaultStatut = STATUT_ENABLED;
if (periodeCampagne == 'cloture') {
        // mode consultation
        defaultStatut = STATUT_DISABLED;
}
```

Exemple de commentaire dans une méthode

3.4 FORMAT DE COMMENTAIRES D'IMPLEMENTATION

Pour les fichiers .js, le format des commentaires d'implémentation suit les règles décrites pour JAVA au chapitre 2.4.6.

3.5 COMMENTAIRES HTML

Pour les fichiers .html, le format des commentaires est le suivant :

```
<!-- Ecrire ici son commentaire -->
```

Veillez à bien placer un espace avant la marque finale `-->`