Dream IT : Normes de développement

Document présentation

Description	Ce document a pour objectifs : · de fournir des règles de développement applicables à toute application écrite en Java et qui peuvent être implémentées dans des outils de contrôle comme CAST ou CheckStyle. · de fournir une liste de préconisations et bonnes pratiques utiles au développement d'applications Java. Ce document n'a pas pour but de présenter, ni d'expliquer le langage Java.

Document certification

	Name	*Fonction*	*Date livraison*
Author	RAKOTOBE Eric	Responsable Qualité	02 mars 2013

Document Version history

Version	*Date*	*Change Summary*	*Change Author*
1.0	02/03/2013	Initialisation du document	RAKOTOBE Eric

1 Introduction

Ce document a pour objectifs :
· de fournir des règles de développement applicables à toute application écrite en Java et qui peuvent être implémentées dans des outils de contrôle comme CAST ou CheckStyle.
· de fournir une liste de préconisations et bonnes pratiques utiles au développement d'applications Java.
Ce document n'a pas pour but de présenter, ni d'expliquer le langage Java.
Il faut garder à l'esprit que l'efficacité d'une norme se mesure par le nombre de personnes qui l'accepte et la supporte. Toutes les règles présentes dans ce document tolèrent des exceptions si elles sont justifiées. Toute exception à l'application de ces règles doit être documentée.
Ce document est essentiellement à destination des développeurs. Il peut également être utilisé comme base pour mesurer la qualité du code d'une application.
Les conventions d'écriture du code Java fournies par SUN sont la base de ce document. Ces conventions que l'on peut retrouver sur http://java.sun.com/docs/codeconv/ sont rappelées rapidement pour les plus importantes dans le premier chapitre.

2 Conventions d’écriture du code Java

2.1 RÈGLES DE NOMMAGE

2.1.1 Fichiers

Les fichiers sources portent l’extension .java, les fichiers compilés l’extension .class.

2.1.2 Classes et interfaces

Un nom de classe ou d’interface est composé uniquement de lettres et commence par une lettre majuscule.
Le nom représente un groupe de mots. Il doit être court mais suffisamment explicatif.
La première lettre de chaque mot est une majuscule.

```
class   FileReaderServiceException
```

2.1.3 Méthodes

Un nom de méthode est composé uniquement de lettres et commence par une lettre minuscule.
Le nom est un groupe de mots.
Une méthode représente souvent une action.
Le premier mot du groupe est donc généralement un verbe.

```
public boolean isUserAuthenticated();
```

La première lettre de chaque mot est une majuscule sauf pour le premier mot du groupe.

2.1.4 Variables

Un nom de variable est composé de lettres, éventuellement de chiffres, et commence par une lettre minuscule.
Cette règle s'applique à toutes les variables qu'elles soient de classe, d'instance ou locales.
Le nom est généralement un groupe de mots.
La première lettre de chaque mot est une majuscule sauf pour le premier mot du groupe.
L'usage du caractère dollar ($) est interdit pour les noms de variables.
L'usage du caractère trait bas (_) est à éviter.
L'utilisation de noms d'une seule lettre est réservée à des cas très particuliers comme pour les variables d'itération.
Les noms d'une seule lettre les plus utilisés sont i, j, k, l, m, n pour les entiers et c, d, e pour les caractères.

```
private String serverHostName;
```

Les constantes sont écrites en majuscules.
Les mots qui composent le nom de la constante sont séparés par un trait bas (_).

```
private static final String[] RETURNED_ATTRIBUTS = { "cn" };
```

2.2 RÈGLES SUR L'ORDRE, LE FORMATAGE ET LA PRÉSENTATION

2.2.1 Ordre

Un fichier Java ne contient qu'une seule classe ou interface publique.
Les classes privées sont définies à la fin du fichier, après cette unique classe publique.
Généralement, un fichier ne contiendra qu'une seule classe.
Les interfaces et les classes privées associées à une classe publique peuvent être placées dans le même fichier que cette classe publique.
Le regroupement de classes dans un même fichier est admissible dans certains cas particuliers. Par exemple le développement d'IHM pour lesquelles il faut étendre de nombreux évènements.
Les méthodes sont regroupées par groupes de fonctionnalités, indépendamment de leur visibilité.

2.2.2 Formatage

Un fichier ne doit pas dépasser 2000 lignes.
Un fichier plus volumineux indique que la conception doit être revue.
Lorsqu'une instruction ne tient pas sur une ligne, il est conseillé de faire un saut de ligne après une virgule, ou avant un opérateur.
Certaines instructions ne peuvent s'écrire sur une seule ligne car trop longues (surtout avec un niveau d'indentation important).
Il faut alors briser la ligne.
Il faut le faire après une virgule ou avant un opérateur.
La nouvelle ligne s'aligne avec le début de l'instruction de la ligne précédente.
Si toutes ces règles conduisent malgré tout à déborder sur la marge de droite, il faut alors utiliser une indentation de 8 caractères comme le montre les exemples qui suivent.

```
String[]

RETURNED_ATTRIBUTS = { "sn","givenName", "uid" };
System.out.println("ldap://"
+ params.getServerHostName() + ":"
+ params.getPort() + "/"
+params.getRoot());
```

L'unité standard d'indentation est de 4 espaces.
La construction de cette indentation (espaces ou tabulations) est laissée libre.
Ne pas affecter une valeur à plusieurs variables en une seule instruction.

2.2.3 Présentation

Les accolades ouvrantes ({) sont en fin de ligne d'instruction.
Les accolades fermantes (}) sont sur une ligne seule et indentées pour être au même niveau que l'instruction du bloc d'ouverture.
Seule exception les blocs nulls, les accolades sont alors sur la même ligne.

```
for     (int i = 0; i < results.length; i++)    {
        …
}

private SingletonConstructor() {}
```

Tout bloc de contrôle (if, for, while, do, switch, ...) est placé entre accolades, même s'il ne contient qu'une ligne ou s'il est vide.
Seule exception les blocs for et while qui sont vides.

```
if (isUserAuthenticated("identifiant", "motdepasse")) {
        System.out.println("Identification réussie.");
} else {
        System.out.println("Echec de l'identification.");
}

for (int i = 0; i > 0;);

while   (true);
```

Chaque ligne doit contenir une seule instruction.
Ceci vaut également pour la déclaration des variables.
Déclarer une seule variable par ligne permet de commenter simplement en fin de ligne l'utilité de cette variable.

2.3 COMMENTAIRES

Les commentaires d'implémentation (/* */, //) sont utilisés pour commenter le code.
Les commentaires de documentation (/** */) sont utilisés pour décrire les spécifications du code.
Les commentaires de documentation doivent permettre l'utilisation correcte du code pour une personne qui n'a pas accès au code source lui-même.
Les commentaires ne doivent pas être positionnés dans des boîtes formées par des astérisques ou autres caractères.
Un bloc de commentaire contient un astérisque au début de chaque ligne sauf la première.
Un tel bloc doit être aligné au même niveau que le code qu'il commente.

```
/*
*       Bloc de commentaire
*/
```

2.4 DIVERSES BONNES PRATIQUES

On accède à un membre ou une méthode statique par le nom de classe ou d'interface où il est défini, jamais via une instance.
Ne pas accéder directement aux attributs des autres classes.
Utiliser plutôt les accesseurs et mutateurs fournis pour ces attributs.
C'est l'un des principes de bases pour l'encapsulation et la sécurité des données.
Le seul cas où l'utilisation de variables de classes et d'instance publique peut être justifié est celui d'une classe qui ne contient que des variables sans aucune méthode.

3 Extension des conventions d'écriture de code Java

3.1 RÈGLES DE NOMMAGE

3.1.1 Fichiers

Les fichiers sources portent strictement le même nom (y compris au niveau de la casse des caractères) que l'unique classe Java publique ou l'unique interface Java publique qu'ils contiennent.

3.1.2 Packages

Un nom de package est composé de mots en lettres minuscules, séparés par des points.
Package pays.entreprise.nomProjet.service

3.1.3 JavaBeans

Les accesseurs d'une propriété d'un JavaBean ont un préfixe normalisé.
Pour l'accesseur il s'agit de get, pour le mutateur de set.
L'accesseur d'un attribut booléen est is.

```
public String getServerHostName() ;

public void setServerHostName(String   serverHostName) ;

public boolean isCloneable();
```

Les accesseurs d'une propriété JavaBean de type tableau ou Collection se terminent par un pluriel.
Les accesseurs indexés sont au singulier.

```
public String[] getAttributNames();
public boolean isCloneable();
public boolean isDisponible();
public String getAttributName(int i);
```

Les propriétés de JavaBean de type boolean se terminent par (able) ou (ible).

3.2 RÈGLES SUR L'ORDRE, LE FORMATAGE ET LA PRÉSENTATION

3.2.1 Ordre

Les déclarations d'un fichier java suivent un ordre défini.

1. /* En-tête de fichier normalisée */
2. Déclaration du package
3. Liste des imports
4. /** Commentaire Javadoc de description de la classe */
5. Déclaration de la classe
6. /* Commentaire de description de l'implémentation de la classe */
7. Déclaration des constantes (statiques finales)
8. Déclaration des variables de classe (statiques)
9. Déclaration des variables d'instance
10. /** Commentaire Javadoc de description du constructeur */
11. Constructeur
12. /** Commentaire Javadoc de description de la méthode */
13. Méthode
14. Déclaration de classes imbriquées

Les étapes 10/11 et 12/13 se répétant autant de fois qu'il y a de constructeurs et de méthodes.
L'en-tête normalisée contient au minimum la liste des programmeurs, la date et une brève description de l'action de la classe.

Lorsque plusieurs modificateurs sont utilisés dans une déclaration, ils suivent un ordre défini.

1. public
2. protected
3. private
4. abstract
5. static
6. final
7. transient
8. volatile
9. synchronized
10. native
11. strictfp

```
public static void main(String[] argv);
private static final String[]   RETURNED_ATTRIBUTS = { "cn" };
```

Les déclarations de constantes et variables sont classées par visibilité décroissante (public, protected, sans mot clé, puis enfin private).

3.2.2 Formatage

Une méthode (ou constructeur) ne doit pas dépasser 100 lignes.
Une ligne ne doit pas dépasser 120 caractères.
Historiquement la règle est de limiter l'affichage à 80 colonnes.
La limitation à 120 caractères permet de diminuer les contraintes dues à la plus grande verbosité de Java tout en conservant une très bonne lisibilité.

3.2.3 Présentation

Les imports sont groupés à partir du second niveau et classés par ordre alphabétique.
Dans l'exemple ci-dessous, une ligne vide sépare les imports de second niveau java.util et javax.naming.
Tous les imports sont classés par ordre alphabétique.

Ne pas utiliser de clause d'import générique (*).

```
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import javax.naming.Context;
import javax.naming.NamingEnumeration;


public static void main(String[] argv) {
        LdapParameters params = new SimpleLdapParameters();
        …
        if (connexion.isConnected()) {
                String[] resultants;
                String[] attributNames;
                …
```

Mettre les déclarations de variables en début de bloc.

3.2.4 Robustesse

Toujours avoir un bloc default dans les switch.
C'est un principe d'ouverture et qui permet de traiter les cas non prévus.

3.2.5 Performance

Mettre les déclarations des variables des boucle avant la boucle
Pour des raisons de performances, la règle de déclaration des variables de bloc ne s'applique pas pour les boucles (for, while, …).
Il faut dans ce cas mettre les déclarations avant le début du bloc.

```
String hostname;
for  (Iterator i = …serverList.iterator(); i.hasNext();){
        hostname = (String) i.next();
```

3.2.6 Equilibre

Les règles définies dans ce chapitre permettent d'offrir une vision, en terme d'équilibre, sur les différents niveaux de granularité d'une application Java.
Moins une application respectera ces règles et plus elle sera complexe et difficile à maintenir.
- Une classe ne devrait pas implémenter plus de 7 interfaces.
- Un package devrait posséder entre 3 et 75 classes.
- Une classe devrait posséder moins de 30 méthodes.
- Une classe devrait posséder moins de 4 constructeurs.
- Une classe devrait posséder moins de 30 attributs.
- Une interface devrait posséder moins de 20 méthodes.
- Une méthode devrait contenir moins de 9 règles de gestion.

Une règle de gestion est représentée par un bloc de contrôle (if, for, while, do, switch, ...)

3.3 EXCEPTIONS ET COMMENTAIRES

3.3.1 Exceptions

Réutiliser les exceptions simples définies dans le JDK.

```
//        exemple de paramètre incorrect
if (port != 389 || port != 636) {
        throw  new IllegalArgumentException ( "Le numéro de port est invalide" );
}
```

Les exceptions fonctionnelles sont dérivées de java.lang.Exception.
Il s'agit des exceptions dites contrôlées.
Il faut les prendre en compte dans un bloc try/catch adapté.
Les exceptions techniques irrémédiables sont dérivées de java.lang.RuntimeException.
Ce sont les exceptions non contrôlées.
Si au niveau d'une méthode, on ne sait pas comment traiter une exception, le mieux est de ne pas la catcher.
L'exception se propage alors au niveau supérieur qui saura peut-être la traiter.
Lorsqu'une exception est attrapée dans un bloc catch, elle ne doitvpas être « digérée ».
Une exception attrapée dans un bloc catch doit être traitée ou renvoyée.

```
//      exemple d'exception rendue muette
//      ce qu'il ne faut pas faire
try {
        …
} catch {}
```

Dans une clause throws, il ne faut pas déclarer une exception non contrôlée.
Dans une clause throws, il ne faut pas déclarer une exception dérivée d'une autre exception déjà déclarée dans cette clause.
Une exception doit être immuable, c'est-à-dire que tous ses membres doivent être déclarés final.
Il est recommandé de ne pas utiliser System.out, System.err et Throwable.printStackTrace() dans les blocs catch.
Il existe des frameworks performants de gestion de la journalisation dédiés à cette problématique.

Une étude est en cours sur le meilleur framework à utiliser.
Aucune solution n'est pour le moment imposée.
Le framework log4j est l'un des plus utilisé dans ce domaine et est un exemple des frameworks qui peuvent être exploités.

3.3.2 Commentaires

Les commentaires JavaDoc sont utilisés pour l'ensemble d'un projet, pour tous les niveaux d'accès (public, package, protected, private) et pour tous les types de déclarations (classes, interfaces, méthodes, champs, constructeurs, …)
Le champ Javadoc @author doit être renseigné pour toutes classesou interfaces.

```
/**
* @author Prénom Nom
*/
```

Dans le cas où ce champ doit être changé, il faut y indiquer le nom de la ou des personnes qui peuvent être facilement contactés et qui ont le plus de connaissances sur le code, généralement les auteurs eux-mêmes.
Il est admis qu'un code est suffisamment documenté lorsque les lignes de commentaires représentent au minimum 5 à 10% du nombre total de lignes du fichier.
Il ne faut pas laisser de code en commentaire, sauf dans un but explicatif.
Un code en commentaire non-explicatif s'apparente à du code mort et doit donc être évité.

```
// exemple de commentaires à supprimer
params.setPort(port)          ;
params.setServerHostName(serveurHostName)          ;
// params.setPort(389) ; // valeurs de tests
// params.setPort("serveurtest.test.fr") ;
```

Les marques laissées dans le code sont normalisées.
Certains motifs laissés dans les commentaires on une forme prédéfinis qui permettent de retrouver rapidement et de synthétiser certains développements.

Code qui doit être amélioré :

```
//        FIXME: réparation à appliquer
```

Ajout d'une fonctionnalité :

```
//        TODO: description de la fonctionnalité à ajouter
```

Correction du bug numéroté 10 dans le bugtracker du projet :

```
//        BUGFIX#10
```

Correction du bug numéroté 10 dans le bugtracker du projet :

```
//        BUGFIX#10
```

Désigne un code bugué mais qui fonctionne :

```
//        XXX: description du problème
```

4 Astuces diverses et bonnes pratiques

Toujours affecter la visibilité la plus restrictive à une méthode et à un attribut.
Dans les comparaisons, placer les constantes à gauche.
Par exemple :

```
if ( 1 == variable) {
        …
}

if ("Chaîne".equals(variable))       {
        …
}
```

Ceci permet d'éviter que :

```
if (variable.equals("Chaîne")) {
        …
}
```

ne renvoie une NullPointerException si variable == null.

Ne pas utiliser les types float et double pour des calculs arithmétiques, mais plutôt l'objet BigDecimal.
Les types floats et double n'offrent pas une représentation exacte des nombres réels, un exemple est la représentation du nombre 0,8.
Dans certains domaines, comme les calculs financiers, cette approximation ne peut pas être tolérée, il faut donc utiliser le type BigDecimal.
Ne pas utiliser les classes java.util.Vector, java.util.Hashtable et java.util.Enumeration, mais plutôt java.util.List, java.util.HashMap et java.util.Iteration.

Attention tout de même, Vector est synchronized et peut donc être préféré dans certain cas bien précis.

Il est recommandé de ne pas écrire de SQL dans les classes.
Il existe des frameworks très performants qui permettent de s'occuper de toute la couche d'accès aux bases de données.
Ils fournissent une très bonne abstraction qui permet de se passer, ou tout du moins de minimiser,
l'écrire de SQL dans une classe Java.
Une étude est en cours et aucun framework n'est pour le moment imposé.
Le framework Hibernate et le pattern DAO sont des solutions reconnues dans ce domaine et des exemples des réponses qui peuvent être exploitées.

4.1 CLASSES STATIQUES

Une classe qui n'a que des méthodes statiques ne doit pas avoir de constructeur public.
Il faut donc déclarer obligatoirement un constructeur privé ou protected.

4.2 HÉRITAGE

Une classe ne doit pas définir une méthode privée portant la même signature (nom et arguments) qu'une méthode privée définie sur une classe mère.
Eviter de réécrire des attributs dans une sous-classe.
Généralement cela met en lumière un problème de conception.
Ce n'est pas de la bonne programmation orientée objet.
Une classe ne devrait pas connaître ses sous-classes.
Si on redéfinit la méthode clone() pour une classe, il faut toujours appeler la méthode clone() de la classe mère par super.clone().
Si on redéfinit la méthode finalize(), la dernière instruction de la méthode redéfinie sera un appel à super.finalize().
Toute classe qui redéfinit equals doit également redéfinir hashcode (cf Javadoc de la classe Object).

La règle à respecter est :
(a.equals(b)) => (a.hashCode() == b.hasCode())
L'inverse n'est pas obligatoire.

4.3 PORTABILITÉ

L'import des packages sun.* est interdit.
Ces packages sont spécifiques pour chaque plate-forme et donc non portables.
Ne pas utiliser java.lang.runtime.Exec().
Cette méthode permet de lancer une commande système.
La réussite de son exécution dépend uniquement de l'environnement.
Ce code n'est donc pas portable.
Ne pas utiliser System.exit().
L'utilisation de cette méthode va, dans une majorité de cas, à l'encontre du modèle de gestion des exceptions.

4.4 GESTION DE LA MÉMOIRE

Libérer les ressources dans le bloc finally.
Remettre à null les références aux objets volumineux comme les tableaux.
Un objet sans référence va être nettoyé par le ramasse-miette ce qui va donc libérer de la mémoire.
Ne pas appeler System.gc explicitement dans le code.
C'est un mécanisme automatique de la JVM.
De plus l'utilisation de cette méthode consomme une grande quantité de ressources.

```java
/*
* @(#)Blah.java 1.82 18/03/03
*/
package fr.unice.esinsa.blah;
import java.blah.blahdy.BlahBlah;

/**
*       Description de la classe.
*       @version 1.82 18 mars 2003
*       @author Nom Prenom
*       @invar Invariant de classe
*/

public class Blah extends UneClasse {

        /* Un commentaire specifique a l'implementation. */
        /** Documentation de la variable de classe variableDeClasse1. */
        public static int variableDeClasse1;

        /**
        * Documentation pour variableDeClasse2 qui se trouve etre plus
        * long qu'une ligne de 70 caracteres.
        */
        private static Object variableDeClasse2;

        /** Commentaire pour le champ variableDInstance1. */
        public Object variableDInstance1;

        /** Commentaire pour le champ variableDInstance2. */
        protected int variableDInstance2;

        /** Commentaire pour le champ variableDInstance3. */
        private Object[] variableDInstance3;
```

```
	/**
	* Documentation du constructeur Blah() par defaut.
	* @cons Consequent de la construction.
	*/
	public Blah() {
		//	... implementation
	}

	/**
	* Documentation de la methode faitQuelqueChose.
	* @ante Antecendant global de la methode
	* @throws Exception et pourquoi
	* @return Detaille la valeur de retour
	* @cons Consequent global de la methode
	* @compl Complexite algorithmique du traitement
	*/
	public Object faitQuelqueChose() throws Exception {
		//	... implementation
	}

	/**
	*	Documentation de la methode faitQuelqueChoseDAutre.
	*	@ante Antecendant global de la methode
	*	@param unParametre Description du parametre
	*	@param unEntier Description du parametre
	*	@cons Consequent global de la methode
	*	@see variableDeClasse1
	*/
	public void faitQuelqueChoseDAutre(Object unParametre, int unEntier) {
		//	... implementation
	}
}
```

6 Derniers conseils

Ne pas réinventer la roue.
Vérifiez que ce que vous vous apprêtez à coder n'existe pas déjà dans les API à votre disposition.
Exploiter les services fournis par l'environnement d'exécution.
Dans le cas d'une application J2EE, qui sera exécutée dans un container, profiter de la gestion de transaction JTA incluses dans les EJB (cf spec des EJB), utiliser les pools de connexion (maintenant présents dans la plupart des serveurs d'application), etc.
Eviter le copier-coller.
Si le besoin se fait sentir de dupliquer du code, généralement cela met en lumière un problème de conception.