



DIRECTION GÉNÉRALE DES FINANCES PUBLIQUES
SERVICE DES SYSTÈMES D'INFORMATION
SOUS-DIRECTION ÉTUDES ET DÉVELOPPEMENT
BUREAU SI-1A DMOD
4, AVENUE MONTAIGNE
93468 NOISY-LE-GRAND CEDEX

CHARTRE DE DÉVELOPPEMENT JAVA

RÉF. : CHARTEDevJAVA

VERSION : 2.3.1 DU 14 JUIN 2024

STATUT : APPLICABLE / ~~EN VALIDATION~~ / ~~EN COURS~~ / ~~SUPPORT~~

MINISTÈRE DU BUDGET
DES COMPTES PUBLICS
ET DE LA FONCTION PUBLIQUE



Objet et domaine d'application

En conformité avec les axes stratégiques d'adoption de normes, de standards et d'utilisation de logiciels libres en vue du renforcement de la maintenabilité des applicatifs, la charte de développement Java regroupe environ 250 règles classées par catégorie et s'appuyant largement sur les critères communément admis pour l'évaluation des propriétés du code. Cette charte fixe les exigences qualité attendues avec trois niveaux de contrainte. Un outillage a été développé pour automatiser la vérification de la conformité de 80% des règles de la charte, les 20% restant étant vérifiés à l'aide de procédures manuelles. Cet outillage est complété par un mécanisme de rapport restituant une vue synthétique des résultats. La charte s'applique à tout nouveau projet développant en Java. Les projets existants intégreront ces règles au fur et à mesure des maintenances évolutives.

Document de portée	Particulière : ne concerne que ses destinataires	
	Générale : application à tous les projets DGFIP	✓

		Projet/bascule ou applicatif en		
		Production	IA / IIA / INTEX	Développement
Document d'application	Immédiate : les règles présentées s'appliquent immédiatement			✓
	Sur le flux : application immédiate sur les nouveaux développements			✓
	A planifier : renvoi d'un planning de mise en œuvre au bureau SI-1A			

Document d'importance	Vitale : sa mise en œuvre conditionne la cohérence du programme	✓
	Particulière : la mise en œuvre présente un intérêt particulier	
	Informative : solution d'un projet, présentant un intérêt général, par ex.	

PAGE DE GESTION

Historique des versions				
Versio n	Date	Site/Rubrique	Rédigé par	Commentaires / modifications apportées
0.1	28/06/2006	http://venezia.dev.impots/ plugins/owl/ owl_wrapper/295/ browse.php? sess=0&parent=56402	François LE DROFF	Document initial
0.2	03/07/2006		François LE DROFF	Avancement du document.
0.3	07/07/2006		François LE DROFF Xavier CHATELAIN	Mises en forme et avancement du document.
0.4	19/07/2006		François LE DROFF Xavier CHATELAIN	Prise en compte des remarques du groupe de travail, suite à sa réunion du 11/07/2006, et ajout de nouvelles règles (notamment Performance et J2EE).
0.5	31/07/2006		François LE DROFF Xavier CHATELAIN	Avancement du document (notamment Métriques).
0.6	22/08/2006		François LE DROFF Xavier CHATELAIN	Avancement du document (toutes règles).
0.7	25/08/2006		François LE DROFF Xavier CHATELAIN	Prise en compte des remarques de Françoise Payen et Maurizio De Cecco. Mises en forme.
0.8	31/08/2006		François LE DROFF Xavier CHATELAIN	Prise en compte des remarques du groupe de travail suite à sa réunion du 29/08.
0.8.1	08/09/2006		Xavier CHATELAIN	Prise en compte des remarques d'Eric Priol et d'Eric David.
0.8.2	14/09/2006		Xavier CHATELAIN	Prise en compte des remarques de Denis Oddoux et d'Arnaud Genre-Grandpierre.
0.8.3	15/09/2006		Françoise PAYEN	Prise en compte des remarques d'Eric David.
0.8.4	02/10/2006		Xavier CHATELAIN	Modification de la règle sur le nommage des classes abstraites.
1.0	18/10/2006		Françoise PAYEN	Mise à la norme des en têtes et du pied de page.
1.1	19/04/2007		Xavier CHATELAIN	Synchronisation avec l'outillage (automatisation du contrôle des règles), améliorations et corrections de bogues.
1.1.1	25/01/2008		Xavier CHATELAIN	Synchronisation avec la version 1.1.1 de CoCA, améliorations et corrections de

				bogues.
2.0.0	05/06/09		Nicolas DORDET	Corrections de bogues et synchronisation avec CoCA version 2.0.0.
2.0.1	10/06/09		Nicolas DORDET	Corrections de bogues et synchronisation avec CoCA version 2.0.1.
2.0.2	20/10/09		Nicolas DORDET Xavier CHATELAIN	Corrections de bogues et synchronisation avec CoCA version 2.0.2.
2.0.3	27/01/10		Nicolas DORDET Xavier CHATELAIN	Corrections de bogues et synchronisation avec CoCA version 2.0.3.
2.0.4	14/04/10		Nicolas DORDET Xavier CHATELAIN	Corrections de bogues et synchronisation avec CoCA version 2.0.4.
2.0.5	22/11/10		Nicolas DORDET	Corrections de bogues et synchronisation avec CoCA version 2.0.5.
2.0.6	21/06/11		Khalil KHOUJA Xavier CHATELAIN	Corrections de bogues et synchronisation avec CoCA version 2.0.6.
2.0.8	05/05/2014		Xavier CHATELAIN	Corrections de bogues et synchronisation avec CoCA version 2.0.8 (cf. §12.3 Notes de version).
2.0.9	10/06/2014		Xavier CHATELAIN	Amélioration et synchronisation avec CoCA version 2.0.9 (cf. §12.3 Notes de version).
2.1.0	17/07/2014		Wenceslas PETIT Xavier CHATELAIN	Amélioration et synchronisation avec CoCA version 2.1.0 (cf. §12.3.3 Notes de version).
2.1.1	01/09/2015		Xavier CHATELAIN	Remplacement d'implémentations de règles PMD et Checkstyle obsolètes avec des règles SonarQube (cf. §12.3.3 Notes de version).
2.2.0	21/04/2017		Xavier CHATELAIN	Réintroduction de règles JavaScript et remplacement d'implémentations de règles FindBugs obsolètes avec des règles SonarQube (cf. §12.3.3 Notes de version).
2.3.1	14/06/2024		Wenceslas Petit	Ajout de la liste des règles relatives au langage Java contrôlées par SonarQube avec le jeu de règles CoCA 2.3.1

Grille de lecture

Selon le profil du lecteur, la revue complète du présent document n'est pas forcément nécessaire. En effet :

- **le chef de projet informatique de la MOE** pourra limiter la lecture du présent document aux chapitres introductifs (1 à 3), et aux tableaux récapitulatifs des règles, en portant une attention particulière aux contraintes associées. A noter que l'outillage de la charte (CoCA v2) restitue un rapport détaillé sur l'évaluation de la conformité du code et permet de se prononcer sur la qualité du code livré.
- **l'architecte** devra prendre connaissance de l'ensemble du document, mais portera particulièrement son attention sur les chapitres 1, 4, 7, 8, 9, 10 et 11. Il veillera notamment à faciliter le respect de l'ensemble des règles et à la rédaction des notes d'architectures éventuellement exigées par cette charte (cf. [PROG-INTERDIT-3] [MEM-2] [MEM-3], [MEM-7], Erreur : source de la référence non trouvée, [THR-2]et [THR-4.1]. Les besoins non couverts par la charte de développement Java feront l'objet d'une demande justifiée transmise au bureau SI-1A/DMOD avant le CAI du projet ou avant (un à deux mois) le début des développements ;
- **le développeur** devra effectuer une lecture attentive de l'ensemble du document. Son environnement de développement sera configuré avec les outils adéquats pour que le non respect d'une règle lui soit immédiatement signalé, afin qu'il puisse le corriger et que le code « commité » soit exempt de toute entorse à la charte.
- **le valideur** pourra concentrer sa lecture sur le chapitre 2 et sur les tableaux récapitulatifs des règles dans les chapitres suivants. L'outillage de la charte lui permettra de vérifier la conformité de 80% des règles de la charte, les 20% restant étant vérifiés à l'aide de procédures manuelles.

SOMMAIRE

1. Présentation de la Charte de développement Java.....	8
1.1. Contexte et objectifs stratégiques sous-jacents.....	8
1.2. Les métiers ciblés.....	8
1.3. Application de la charte.....	9
1.4. Gestions des amendements et des évolutions.....	9
2. Démarche et conventions adoptées.....	10
2.1. Structure.....	10
2.2. Critères de sélection des règles.....	10
3. Règles d'organisation des sources.....	12
3.1. Finalité, critères de sélection, sources et contraintes.....	12
3.2. Organisation des paquetages (<i>packages</i>).....	12
3.3. Organisation interne des fichiers sources Java.....	12
4. Règles de nommage.....	18
4.1. La langue et la typographie.....	18
4.2. Le nom des paquetages (<i>packages</i>).....	19
4.3. Le nom des classes et fichiers.....	20
4.4. Le nom des variables, constantes et méthodes.....	22
4.5. Le nom des méthodes.....	24
4.6. Les conventions de nommage des paramètres de type dans les classes génériques.....	26
5. Règles de documentation.....	28
5.1. La documentation Javadoc.....	28
5.2. Le format des commentaires de traitements (non Javadoc).....	33
6. Règles de formatage de code.....	35
6.1. Finalité, critères de sélection, sources et contraintes.....	35
6.2. Les séparateurs du code : indentations, lignes blanches, et espaces.....	37
6.3. Le formatage et l'indentation des instructions.....	39
6.4. Format et indentations des déclarations et assignements.....	43
7. Règles de programmation : design, patterns et anti-patterns de code.....	47
7.1. Généralités.....	47
7.2. Règles d'encapsulation, d'accessibilité et d'utilisation des modificateurs.....	47
7.3. Règles de déclaration, initialisation et utilisation des variables.....	51
7.4. Règles de programmation des constructeurs.....	55
7.5. Règles générales sur les méthodes.....	58
7.6. Règles relatives aux instructions if, while et switch.....	62
7.7. Règles d'utilisation de null.....	65
7.8. Règles d'implémentation de clone().....	67
7.9. Règles d'implémentation de equals().....	68
7.10. APIs interdites.....	70
7.11. Autres règles de programmation.....	71

8. Règles métriques.....	75
8.1. Finalité, critères de sélection, sources et contrainte.....	75
8.2. Longueur des noms.....	76
8.3. Longueur et complexité des classes et interfaces.....	77
8.4. Longueur et complexité des méthodes.....	78
8.5. Code dupliqué.....	81
9. Règles de gestion des exceptions.....	82
9.1. Généralités, finalité, critères de sélection, sources et contrainte.....	82
9.2. Éviter la perte ou la corruption des exceptions ou des informations sur les erreurs.....	84
9.3. Du bon usage des exceptions.....	87
10. Règles de programmation pour la performance.....	90
10.1. Finalité, critères de sélection, et généralités sur la performance.....	90
10.2. Règles de gestion de la mémoire : fuites mémoires et ramasse-miettes.....	90
10.3. Règles de gestion des blocs critiques (synchronized) et des threads.....	94
10.4. Optimisation dans la manipulation de chaînes de caractères.....	102
10.5. Autres règles pour la performance.....	107
10.6. Micro optimisations.....	109
11. Règles J2EE.....	111
11.1. Règles JDBC.....	111
11.2. Règles JSP/Servlet.....	117
11.3. Règles EJB.....	123
12. Annexes.....	126
12.1. Liste des règles relatives au langage Java contrôlées par SonarQube dans le jeu de règles CoCA 2.3.1.....	126
12.2. Lexique.....	152
12.3. Dernières notes de version.....	155
12.4. Licence du présent document.....	157

1. Présentation de la Charte de développement Java

1.1. Contexte et objectifs stratégiques sous-jacents

La réalisation du nouveau système d'information fiscal doit répondre aux objectifs majeurs suivants :

- maîtrise du système d'information ;
- pérennité du système d'information ;
- indépendance par rapport aux technologies et aux fournisseurs.

Il s'ensuit une stratégie informatique visant, notamment, à :

- respecter les normes et standards du marché ;
- normaliser le système d'information ;
- favoriser la maintenabilité et l'exploitabilité des composants ;
- imposer le respect de la charte d'architecture au sein de la DGFIP mais aussi aux entreprises intervenant au titre de la sous-traitance.

La Charte de développement Java s'inscrit dans cette stratégie. Sa finalité est bien d'unifier, faciliter et promouvoir la qualité des développements Java tout au long du cycle de vie des applicatifs, afin de valider les livrables et faciliter la maintenance, c'est-à-dire :

- améliorer la lisibilité des programmes ;
- faciliter la création et la maintenabilité des applicatifs ;
- améliorer la fiabilité des développements et donc des applicatifs ;
- uniformiser au maximum les développements.

Elle constitue un référentiel des règles Java classées par catégorie et forme avec son outillage un instrument de vérification et de validation des livrables.

La charte n'est pas un guide de développement et ne présente pas le « comment développer en Java ». Elle s'efforce de ne retenir que les règles utiles et pertinentes, et non les habitudes de développement.

Enfin, cette citation de [Martin Fowler](#) tend à résumer le fondement de la charte :

« Any fool can write code that a computer can understand. Good programmers write code that humans can understand. »¹

1.2. Les métiers ciblés

La charte Java concerne les maîtrises d'œuvre internes ou externes (sociétés de services informatiques) et plus particulièrement les profils des personnes ciblées suivants :

- le chef de projet informatique de la MOE ;
- l'architecte ;
- le développeur ;

¹ N'importe qui peut écrire du code compréhensible par un ordinateur. Un bon programmeur écrit du code compréhensible par des humains.

- le valideur.

1.3. Application de la charte

La charte de développement Java s'applique à tout nouveau projet développant en Java.

Les projets existants intégreront ces règles au fur et à mesure des maintenances évolutives.

1.4. Gestions des amendements et des évolutions

Les anomalies et demandes d'évolution seront directement postées dans le bugzilla du projet « normes SODA » (http://venezia.dev.impots/plugins/bugzilla/?group_id=18).

2. Démarche et conventions adoptées

2.1. Structure

Le présent document regroupe les règles en grands domaines :

- règles d'organisation des sources ;
- règles de nommage ;
- règles de documentation ;
- règles de formatage de code ;
- règles métriques ;
- règles de programmation : *design*, *patterns* et *anti-patterns* ;
- règles de programmation pour la performance ;
- règles J2EE :
 - règles JDBC,
 - règles JSP/Servlets,
 - règles EJB Session et MDB.

Chaque groupe de règles listé ci-dessus respecte le plan suivant :

- finalité, critères de sélection, sources et contraintes ;
- détail des règles ;
- éventuels exemples et/ou exceptions.

2.2. Critères de sélection des règles

Les critères d'évaluation et de sélection des règles répondent à un double objectif :

- critères pour gérer les évolutions de la Charte de développement Java :
 - l'introduction d'une nouvelle règle est appréciée par rapport à une liste définie de critères caractérisant la Charte de développement Java ;
- critères pour évaluer les propriétés du code :
 - l'ajout d'une règle est apprécié par rapport à la grille de critères caractérisant le code. Une règle ne se substitue ni à l'IDE, ni à la compétence du développeur.

L'acceptation d'une règle dépend du degré de satisfaction des critères. Ces critères sont les suivants :

- la familiarité :
 - acceptation / utilisation par la communauté (interne ou externe) ;
- la complexité :
 - outillage (facilité de mise en oeuvre avec des outils, degré d'automatisation),
 - clarté (compréhension) ;
- la cohérence :

- interne des règles,
- par rapport au système d'information de la DGFIP,
- par rapport aux normes et standards en vigueur ;
- les effets positifs de la règle sur la qualité du code ou de l'architecture, les avantages de la mise en place de cette règle sur l'ingénierie logicielle et plus particulièrement en termes de :
 - bonne et rapide compréhension,
 - lisibilité du code source,
 - uniformité du code source,
 - maintenabilité, évolutivité,
 - portabilité, compatibilité, scalabilité,
 - modularité, ré-utilisabilité,
 - sécurité,
 - performance.

3. Règles d'organisation des sources

3.1. Finalité, critères de sélection, sources et contraintes

Les règles d'organisation décrites ci-dessous sont depuis longtemps adoptées par la majeure partie des développeurs Java, elles sont cependant arbitraires.

Ces conventions ont pour but d'améliorer la lisibilité et l'uniformité du code. Elles ont pour finalité de garantir aux développeurs une plus rapide et une meilleure compréhension du code source.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[ORG-1]	Le cartouche	Checkstyle	Regexp Header	Mineure
[ORG-2]	Organisation des classes et interfaces	SonarQube	S1213	Mineure
[ORG-3]	Organisation des enum			Mineure
[ORG-4]	L'ordre des modificateurs	SonarQube	ModifiersOrder Check	Mineure

3.2. Organisation des paquetages (packages)

Les paquetages permettent de grouper les classes sous une forme hiérarchisée.

3.3. Organisation interne des fichiers sources Java

Les différents éléments qui composent la définition de la classe ou de l'interface **doivent** être indiqués dans l'ordre suivant :

3.3.1. [ORG-1] Le cartouche

On entend par cartouche la documentation du début de fichier.

Chaque fichier source **doit** commencer par un commentaire multi-lignes contenant au minimum le *copyright* et la date de création du fichier , et éventuellement tout autre description ou commentaire jugé utile.

Exemple :

```
/*
 * Copyright © [année de création-année courante] DGFIP - Tous droits réservés
 *
 */
```

A ce cartouche vient s'ajouter une convention de documentation pour les classes et interfaces. Cet en-tête est construit généralement à l'aide des environnements de développement intégré pour Java .

3.3.2. [ORG-2] Organisation des classes et interfaces

Pour chaque classe/interface, on **doit** trouver dans l'ordre :

- le cartouche défini ci-dessus (cf. [ORG-1]) ;
- la déclaration du paquetage ;
- la déclaration des imports statiques du plus générique au plus spécifique ;
- la déclaration des imports (classiques), du plus générique au plus spécifique ;
- l'en-tête Javadoc normalisé précédant la déclaration de la classe ou de l'interface ;
- le cas échéant, les types énumérés de classes (déclarées avec le mot clé **static**) qui doivent être triées selon l'ordre des modificateurs spécifiés dans [ORG-4] ;
- le cas échéant, les variables de classes (déclarées avec le mot clé **static**) qui doivent être triées selon l'ordre des modificateurs spécifiés dans [ORG-4] ;
- le cas échéant, les blocs d'initialisation de classe (déclarées avec le mot clé **static**). On notera que les blocs d'initialisation d'instance non statiques sont interdits ;
- le cas échéant, les types énumérés (**enum** depuis Java5) d'instances qui doivent être triées selon l'ordre des modificateurs spécifiés dans [ORG-4] ;
- le cas échéant, les variables d'instances qui doivent être triées selon l'ordre des modificateurs spécifiés dans [ORG-4] ;
- le cas échéant, les initialisateurs qui doivent être triés selon l'ordre des modificateurs spécifiés dans [ORG-4] ;
- le cas échéant, le ou les constructeurs ;
- le cas échéant, les méthodes : elles seront regroupées par fonctionnalités et/ou selon leur accessibilité.

Exemple :

```
/**
 * [ORG-2]
 * Ici en-tête Javadoc normalisé [JDOC-5]
 */
public final class Carte
{
    /**
     * Ci-dessous, types énumérés de classes (déclarées avec le mot clé static) qui
     * doivent être triées selon l'ordre des modificateurs spécifiés dans
     * [ORG-4]
     */
    public static enum Rang
    {
        DEUX, TROIS, QUATRE, CING, SIX, SEPT, HUIT, NEUF, DIX, VALET, REINE, ROI,
    }

    public static enum Suite
    {
        TREFLES, CARREAUX, COEURS, PIQUES
    }

    /**
     * Ci-dessous, les variables de classes (déclarées avec le mot clé static) qui
```

```

    * doivent être triées selon l'ordre des modificateurs spécifiés dans
    * [ORG-4]
    */
private static final List<Carte> pioche = new ArrayList<Carte>();

/**
 * Ci-dessous, un bloc d'initialisation de classe (déclaré avec le mot clé
 * static)
 */
static
{
    for (Suite suite : Suite.values())
    {
        for (Rang rang : Rang.values())
        {
            pioche.add(new Carte(rang, suite));
        }
    }
}

/**
 * Ci-dessous, les variables d'instances qui doivent être triées selon l'ordre
 * des modificateurs spécifiés dans [ORG-4]
 */
private final Rang rang;

private final Suite suite;

/* Ci-dessous le ou les constructeurs */
private Carte(final Rang rng, final Suite ste)
{
    this.rang = rng;
    this.suite = ste;
}

/**
 * Ci-dessous les méthodes, triées selon l'ordre des modificateurs défini par
 * [ORG-4]
 */
public Rang getRang()
{
    return rang;
}

public Suite getSuite()
{
    return suite;
}

public String toString()
{
    return rang + " of " + suite;
}

public static ArrayList<Carte> newPioche()
{
    return new ArrayList<Carte>(pioche); // Retourne une copie de pioche
}

public static List<Carte> getPioche()
{
    return pioche;
}
```

```
}
```

3.3.3. [ORG-3] Organisation des types énumérés (enum, depuis Java 5)

Pour chaque type énuméré, on **doit** trouver dans l'ordre :

- le cartouche défini ci-dessus (cf. [ORG-1]) ;
- la déclaration du paquetage ;
- la déclaration des imports, du plus générique au plus spécifique ;
- l'en-tête Javadoc normalisé précédant la déclaration du type énuméré ;
- les éléments du type énuméré, un par ligne, séparé par une virgule en fin de ligne ;
- le reste est identique à l'organisation des classes.

Exemple 1 :

```
public enum Planete
{
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6),
    MARS(6.421e+23, 3.3972e6),
    JUPITER(1.9e+27, 7.1492e7),
    SATURN(5.688e+26, 6.0268e7),
    URANUS(8.686e+25, 2.5559e7),
    NEPTUNE(1.024e+26, 2.4746e7);

    private final double masse;    // en kilogrammes

    private final double rayon;    // en metres

    Planete(double masse, double rayon)
    {
        this.masse = masse;
        this.rayon = rayon;
    }

    public double getMasse()
    {
        return masse;
    }

    public double getRayon()
    {
        return rayon;
    }

    /**
     * Constante universelle de gravitation (m3 kg-1 s-2)
     */
    public static final double G = 6.67300E-11;

    public double getGravite()
    {
        return G * masse / (rayon * rayon);
    }

    public double getPoids(double autreMasse)
    {

```

```
        return autreMasse * getGravite();  
    }  
}
```

Exemple 2 :

```
public enum Operation  
{  
    PLUS  
    {  
        double eval(double x, double y)  
        {  
            return x + y;  
        }  
    },  
    MINUS  
    {  
        double eval(double x, double y)  
        {  
            return x - y;  
        }  
    },  
    TIMES  
    {  
        double eval(double x, double y)  
        {  
            return x * y;  
        }  
    },  
    DIVIDE  
    {  
        double eval(double x, double y)  
        {  
            return x / y;  
        }  
    };  
  
    /* Do arithmetic op represented by this constant */  
    abstract double eval(double x, double y);  
}
```

3.3.4. [ORG-4] L'ordre des modificateurs

L'ordre d'apparition des modificateurs, pour chacun des types de déclarations listés en [ORG-2] (les types énumérés de classes, les variables de classes, les types énumérés d'instances et les variables d'instances), doit se conformer à la liste suivante :

1. Annotations
2. public;
3. protected;
4. private;
5. abstract;
6. static;
7. final;

8. `transient`;
9. `volatile`;
10. `synchronized`;
11. `native`;
12. `strictfp`.

Pour un exemple de code source, se référer à [ORG-2].

4. Règles de nommage

4.1. La langue et la typographie

4.1.1. Finalité, critères de sélection, sources et contraintes

Ces conventions de langue et typographie sont pour la plupart déjà ainsi décrites dans la majorité des guides projets de la DGFIP. Elles ont pour but d'améliorer la portabilité, la compatibilité, la lisibilité, et la compréhension du code source.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Criticité
[NOM-1]	Langue française			
[NOM-2]	Typographie	SonarQube	S00100 S00101 S00114 S00116 S00117 S00119 S00120	Mineure

4.1.2. [NOM-1] Langue française

L'utilisation de la langue française est **obligatoire** à la fois pour le code et à fortiori pour la documentation. Cela permet une meilleure interprétation des noms de données.

Pour des raisons de compatibilité et de portabilité, les caractères accentués sont interdits dans les noms de fichiers et dans le code. Ils sont par contre autorisés dans les commentaires de code et dans la documentation Javadoc.

Les mots anglais utilisés couramment dans le langage informatique sont admis (comme par exemple **Factory**, **Servlet** ou **Bean**).

- **Exception** : certains projets faisant l'objet d'une autorisation spéciale et/ou d'une distribution internationale (*open source* par exemple) pourront utiliser la langue anglaise.
- **Généralités** : les abréviations sont à utiliser uniquement si elles sont usuelles et reconnues de tous. Lorsqu'une abréviation est choisie, n'employer que celle-ci. De même, on évitera les acronymes à moins que ceux-ci soient très courants (ex. URL).

Et enfin, on veillera à toujours choisir des noms significatifs et non ambigus quant au concept qu'ils désignent.

4.1.3. [NOM-2] Typographie

Les caractères spéciaux sont **interdits**, que ce soit dans les noms de packages/répertoires, dans les noms de fichiers sources (java, properties, xml et autres fichiers de configuration) ainsi que dans les noms de classes, types énumérés, interfaces, variables et méthodes Java.

On entend par caractères spéciaux : tous les caractères qui ne sont pas alphanumériques.

Exceptions :

- On autorisera le caractère *underscore* '_' dans le nommage des constantes .
- On autorisera dans les commentaires de traitement les caractères accentués, le C cédille (« ç ») et tous les autres caractères utilisés en français afin d'en faciliter la lisibilité. L'ensemble des caractères autorisés pour les commentaires de traitement n'est pas remonté par l'expression régulière suivante :
`[^0-9>A-Za-zèùêêçà<\-\\?\\!;\\.\\:\\(\\)\\'@]`

4.2. Le nom des paquetages (packages)

4.2.1. Finalité, critères de sélection, sources et contraintes

La finalité de cette règle est l'uniformité et plus particulièrement ici, la maintenance et l'évolution des projets.

Le nom de paquetage de plus haut niveau est : **fr.gouv.finances.application**

L'application du nom de paquetage de plus haut niveau est **obligatoire** pour les nouveaux projets ou en maintenance évolutive. Les autres projets intégreront cette évolution dans leur programmation annuelle.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Criticité
[NOM-3]	Norme ISO Standard 3166	SonarQube	S120	Majeure
[NOM-3.1]	Toute classe doit définir explicitement son paquetage	SonarQube	S1220	Majeure
[NOM-4]	Convention de nommage des paquetages	SonarQube	S120	Majeure

4.2.2. [NOM-3] La norme ISO Standard 3166

Le nommage des paquetages **doit** utiliser les conventions d'URL inversée définie dans la norme ISO Standard 3166 (standard Java 1.2). L'ensemble des lettres sera en minuscule.

4.2.3. [NOM-3.1] Toute classe doit définir explicitement son paquetage

Les classes utilisant le paquetage par défaut sont **interdites**.

Exemple :	Contre-exemple
<code>package fr.gouv.finances.example.MaClasse</code>	<code>public class MaClasse</code>

4.2.4. [NOM-4] Convention de nommage des paquetages

Les paquetages ne **doivent** comporter que des caractères alphanumériques.

Pour les nouveaux projets, le paquetage de plus haut niveau de l'applicatif **doit** être `fr.gouv.finances`.

Les noms des paquetages sont donc des instances de l'expression régulière suivante :

```
^fr.gouv.finances.+(\.[a-z][a-z0-9]*)*$
```

Exemple :

```
fr.gouv.finances.<application>.
```

4.3. Le nom des classes et fichiers

4.3.1. Finalité, critères de sélection, sources et contraintes

La finalité de cette règle de nommage de fichiers et de classes est à nouveau l'uniformité et la lisibilité du code, plus particulièrement ici en vue de la maintenance.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Critère
[NOM-5]	Nommage des classes et fichiers	SonarQube	S00101	Mineure
[NOM-6]	Suffixes techniques courants des classes			
[NOM-7]	Suffixes des classes Exception	SonarQube	S2166	Mineure
[NOM-8]	Préfixes des classes abstraites	SonarQube	S00118	Mineure
[NOM-9]	Nommage des interfaces	SonarQube	S00114	Mineure
[NOM-10]	Nommage des classes d'implémentation	SonarQube	S2176	Mineure
[NOM-10-BIS]	Nommage des classes d'extension	SonarQube	S2176	Mineure

4.3.2. [NOM-5] Règles de nommage des classes et fichiers

Chaque fichier source **doit** contenir une et une seule classe publique ou interface, dont le nom **doit** débiter par une majuscule et être suivi de minuscules et majuscules.

Ainsi, pour une meilleure lisibilité des noms, les caractères minuscules et majuscules sont alternés : la première lettre de chaque champ sémantique le composant doit être en majuscule, et le reste en minuscules. L'usage de caractères non alphanumériques est interdit. L'usage de chiffres est toléré quand il s'agit d'un besoin métier. Aucun chiffre ne pourra cependant être utilisé comme préfixe, mais plutôt comme suffixe. A moins que la classe désigne une abréviation très courante (ex. URL), on évitera les acronymes.

Il en sera de même pour le nommage des fichiers de configuration et autres fichiers de déploiement non assujettis à une norme tierce, inhérente à la technologie ou au standard employé (fichier de configuration spécifique au serveur d'applications par exemple).

On notera que le caractère *underscore* '_' est interdit dans les noms de fichiers, tout comme les caractères accentués et les caractères spéciaux (cf. [NOM-2]). Les noms des classes sont ainsi des instances de l'expression régulière suivante : `^[A-Z][a-zA-Z0-9]*$`

Exception : les noms de fichiers JSP commencent par une minuscule, les majuscules servant toujours à délimiter les substantifs .

4.3.3. [NOM-6] Les suffixes techniques courants

Il est recommandé de suffixer le nom d'une classe, si le suffixe peut indiquer le modèle de conception utilisé. Les mots anglais sont d'ailleurs ici souvent utilisés.

Remarque : pour des questions de lisibilité, il est toujours préférable de suffixer un nom plutôt que de le préfixer.

Exemples :

```
MenuServlet  
DocumentFactory  
ActualiteBean
```

4.3.4. [NOM-7] Le nom des classes exceptions

Le nom d'une classe d'exception **doit** se terminer par le suffixe `Exception`. A l'inverse, il **ne faut pas** préfixer ou suffixer par `Exception` le nom d'une classe qui n'est pas une exception.

4.3.5. [NOM-8] Le nom des classes abstraites

Les classes abstraites devraient être facilement repérables. Il est recommandé de les préfixer par `Abstract`.

4.3.6. [NOM-9] Le nom des interfaces

Le nom d'une interface doit respecter la même règle de nommage [NOM-5] que les autres classes. Mais le nom d'une interface **doit** également décrire un comportement. Le nom peut être un adjectif caractérisant le comportement attendu d'une implémentation de cette interface. Il est déconseillé de préfixer le nom de l'interface par `I`.

Au final, le nom des interfaces doit être une instance de l'expression régulière suivante :

```
^(?!I)[A-Z][a-zA-Z0-9]*$
```

Exemple :

```
public interface List
```

4.3.7. [NOM-10] Le nom des classes d'implémentation

Le nom d'une classe pourra se terminer par `Impl` s'il n'existe qu'une interface du même nom. Mais comme derrière une interface existent la plupart du temps plusieurs implémentations, il est recommandé de choisir un nom significatif renvoyant à la fois à l'interface et au caractère spécifique de l'implémentation.

Exemples :

```
ArrayList
```

LinkedList

Il est recommandé de ne pas nommer une classe de la même façon que l'interface qu'elle implémente, même si celles-ci se trouvent dans des *packages* différents.

4.3.8. [NOM-10-BIS] Le nom des classes d'extension

Il est recommandé de ne pas nommer une classe de la même façon que la classe qu'elle étend, même si celles-ci se trouvent dans des *packages* différents.

4.4. Le nom des variables, constantes et méthodes

4.4.1. Finalité, critères de sélection, sources et contraintes

Les sources, références et finalités de ces règles de nommage sont identiques à celles précisées pour le nommage des fichiers Java, à savoir l'uniformité et la lisibilité du code.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outillage	Criticité
[NOM-11]	Nommage des variables	SonarQube SonarQube Checkstyle	S00116 S00117 Static Variable Name	Mineure
[NOM-12]	Nommage des constantes	SonarQube	S00115	Mineure
[NOM-13]	Interdiction d'utiliser <code>enum</code>	SonarQube	S1190	Majeure
[NOM-14]	Interdiction d'utiliser <code>assert</code>	SonarQube	S1190	Majeure
[NOM-15]	Éviter les noms de champs identiques aux noms des classes mères	SonarQube	S1700	Mineure
[NOM-16]	Éviter le nom de champs identiques aux noms de méthodes	SonarQube		Mineure

4.4.2. [NOM-11] Le nom des variables

La première lettre est **obligatoirement** une minuscule.

Pour le reste du mot on suivra à nouveau la règle de nommage des classes : on combinera ensuite les caractères minuscules et majuscules. La première lettre de chaque champ sémantique le composant doit être en majuscule, et le reste en minuscules. L'usage de caractères non alphabétiques est interdit. L'usage de chiffres est toléré

quand il s'agit d'un besoin métier. Aucun chiffre ne pourra cependant être utilisé comme préfixe, mais plutôt comme suffixe et le cas échéant au sein du nom. Les noms des variables sont ainsi des instances de l'expression régulière suivante : `^[a-z][a-zA-Z0-9]*$`

Il faudra également veiller à éviter la notation hongroise. En effet, le langage Java est fortement typé, imposant par sa nature même au programmeur de porter la plus grande attention aux types de données utilisés. Il est donc inutile (et même néfaste) d'utiliser la « notation hongroise », où chaque nom de variable est préfixé par son type. Cette notation a pu donner des aberrations de notation comme `pszName` (exemple tiré de l'API Windows pour dénoter un pointeur sur une chaîne de caractères C, terminée par `\0`).

Si on sait qu'une variable est déclarée comme `String` `nom`, l'information portée par le type est auto suffisante. Il est inutile d'appeler cette variable `strName` ou `sName`.

Exemples :

```
String nom;  
Date dateDeNaissance;
```

4.4.3. [NOM-12] Le nom des constantes

Les constantes **doivent** être toujours en majuscules, les mots sont séparés par un *underscore* '_'. L'usage de chiffres est toléré quand il s'agit d'un besoin métier. Aucun chiffre ne pourra cependant être utilisé comme préfixe, mais plutôt comme suffixe et le cas échéant au sein du nom. Les noms des constantes sont ainsi des instances de l'expression régulière suivante : `^[A-Z][_A-Z0-9]*$`

Exemples :

```
static final int VAL_MIN = 0;  
static final int VAL_MAX = 9;
```

4.4.4. [NOM-13] Interdiction d'utiliser l'identifiant `enum`

Pour éviter les confusions et augmenter la lisibilité du code, le nom `enum` est **interdit** comme identifiant.

4.4.5. [NOM-14] Interdiction d'utiliser l'identifiant `assert`

Pour éviter les confusions et augmenter la lisibilité du code, le nom `assert` est **interdit** comme identifiant.

4.4.6. [NOM-15] Éviter les nom de champs identiques aux noms des classes mères

Il est **interdit** (car déconcertant) d'avoir un nom de champ identique à sa classe mère. C'est souvent symptomatique d'un manque de précision du nom du type et/ou du nom du champ.

4.4.7. [NOM-16] Éviter le nom de champs identiques aux noms de méthodes

Il est **interdit** (car déconcertant) d'avoir un nom de champ identique à un nom de méthode. C'est souvent symptomatique d'un manque de précision du nom du type et/ou du nom de la méthode et d'un non respect des règles de nommage des méthodes .

Contre-exemple des règles [NOM-15] et [NOM-16] et [NOM-18] :

```
public class Compteur
{
    private int compteur = 0;    // nok règles [NOM-15] et [NOM-16]

    public Compteur()
    {}    // ok car c'est le constructeur

    public void compteur()    // nok règles [NOM-16] et [NOM-18] : faute de frappe
ou mauvais nom ?
    {
        /* fait le compte */
    }
}
```

4.5. Le nom des méthodes

4.5.1. Finalité, critères de sélection, sources et contraintes

Les sources, références et finalités de ces règles de nommage sont identiques aux précédentes à savoir l'uniformité et la lisibilité du code.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[NOM-17]	Règles générales de nommage des méthodes	PMD SonarQube	BooleanGetMethodName S00100	Mineure
[NOM-18]	Éviter les noms de méthodes identiques aux noms des classes mères	SonarQube	S1223	Mineure
[NOM-19]	Détection des noms de méthodes suspects	SonarQube FindBugs	S1221 S1201 NM_VERY_CONFUSIN G_INTENTIONAL NM_CONFUSING	Majeure

4.5.2. [NOM-17] Règles générales de nommage des méthodes

Leur nom **doit** contenir un verbe, et la première lettre doit obligatoirement être une minuscule.

Pour le reste du mot on suivra à nouveau la règle de nommage des classes : on combinera les caractères minuscules et majuscules. La première lettre de chaque champ sémantique le composant doit être en majuscule, et le reste en minuscules. L'usage caractères non alphabétiques est interdit. L'usage de chiffres est toléré quand il s'agit d'un besoin métier. Aucun chiffre ne pourra cependant être utilisé comme préfixe, mais plutôt comme suffixe. Les noms des méthodes sont ainsi des instances de l'expression régulière suivante : `^[a-z][a-zA-Z]*[0-9]*$`

Il serait préférable que chaque méthode ne fasse qu'une seule chose, et le nom de chacune reflète cela de manière exacte. Si une méthode fait plus d'une chose, s'assurer que cela est reflété dans son nom.

Exemple :

```
public float calculerMontant()
```

De plus :

- [NOM-17.1] Les méthodes pour obtenir la valeur d'une variable d'instance **doivent** commencer par `get`, suivi du nom du champ ;
- [NOM-17.2] Les méthodes pour mettre à jour la valeur d'une variable d'instance **doivent** commencer par `set`, suivi du nom du champ ;
- [NOM-17.3] Les méthodes pour créer des objets (*factory*) **devraient** commencer par `new` ou `create` ;
- [NOM-17.4] Les méthodes de conversion **devraient** commencer par `to`, suivi du nom de la classe renvoyée à la suite de la conversion ;
- [NOM-17.5] Les méthodes de test sur une variable d'instance **doivent** commencer par `is` ou `est`.

4.5.3. [NOM-18] Éviter les noms de méthodes identiques aux noms des classes mères

Il est **interdit** (car déconcertant) d'avoir un nom de méthodes identique à sa classe mère et donc au constructeur. C'est souvent symptomatique du manque de précision du nom du type et/ou du nom de méthode et (ou d'une faute de frappe).

4.5.4. [NOM-19] Détection des noms de méthodes suspects

Les noms de méthodes trop proches de l'API de `java.lang.Object` tel que `hashCode()` ou `equals(Object)` sont **interdits** et seront considérés comme des fautes de frappes.

Contre-exemple :

```
public int hashCode()  
{  
    /* oops, ce devrait être hashCode() */  
}  
  
public boolean equals(String s)  
{  
    /* oops, ce devrait être boolean equals(Object) */  
}
```

Il est également **interdit** de nommer des méthodes, présentes dans des classes différentes, de façon identique même si la casse est différente. La similitude porte à confusion, notamment lors de la surcharge.

Contre-exemple :

```
Public class Bar  
{  
    public int maMethode()  
    {  
        ...  
    }  
}
```

```

}

Public class Foo
{
    public boolean maMethode()
    {
        ...
    }
}

```

4.6. Les conventions de nommage des paramètres de type dans les classes génériques

4.6.1. Finalité, critères de sélection, sources et contraintes

Ces conventions permettent de distinguer une variable type d'une classe ordinaire (ou d'une interface).

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[NOM-20]	Les paramètres de type dans les classes génériques se composent d'une simple lettre majuscule.	SonarQube	S00119	Mineure

4.6.2. [NOM-20] Les paramètres de type dans les classes génériques se composent d'une simple lettre majuscule

Par convention, les paramètres de type **doivent** être composés d'une simple lettre majuscule. Ceci a pour but de bien contraster avec les conventions de nommage des variables décrites ci-dessus.

Les noms de paramètres de types les plus couramment utilisés sont :

- E : pour définir les éléments utilisés de façon extensive dans l'API **Collection** ;
- K : pour définir les clefs ;
- N : pour définir les nombres ;
- T : pour définir les types ;
- S, U, V, etc. : pour les 2^{ème}, 3^{ème}, 4^{ème} types ;
- V : pour les valeurs.

Mais il est possible d'utiliser tout autre lettre qui pourrait faciliter la lecture :

Exemple de « chaîne alimentaire » « générifiée » :

```

public interface Nourriture
{
    int getIndexGlycemique();
}

public interface Animal<N extends Nourriture>
{
    public void mange(N nourriture);
}

```

```
public class Antilope implements Nourriture, Animal<Herbe>
{
    public int getIndexGlycemique()
    {
        return 5;
    }

    public void mange(Herbe nourriture)
    {
        /* TODO à implémenter */
    }
}

public class Lion implements Animal<Antilope>
{
    public void mange(Antilope nourriture)
    {
        /* TODO à implémenter */
    }
}
```

5. Règles de documentation

On rappelle que la documentations (Javadoc) et commentaires (de traitement) doivent être rédigés en français (cf. [NOM-1]).

5.1. La documentation Javadoc

Les commentaires de documentation devront utiliser la syntaxe de l'outil Javadoc afin de produire une documentation standardisée des classes et interfaces au format HTML.

5.1.1. Finalité, critères de sélection, sources et contraintes

La finalité des règles d'utilisation de la Javadoc est de faciliter la lisibilité et la maintenance par une documentation uniforme et adéquate du code.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[JDOC-1]	La Javadoc est valide	CheckStyle	TodoComment	Mineure
[JDOC-2]	Convention d'écriture et de style de la Javadoc			
[JDOC-3.1]	Fermer les tags HTML de mise en forme (PRE, TT, EM, etc.) utilisés dans les commentaires Javadoc.			
[JDOC-3.2]	Interdiction d'utiliser des tags HTML de structure dans les commentaires Javadoc.			
[JDOC-4.1]	Chaque paquetage doit contenir un fichier package-info.java	CheckStyle	Javadoc Package	Mineure
[JDOC-4.2]	Javadoc des paquetages : les fichiers overview.html			
[JDOC-5]	Règles de documentation des entêtes Javadoc des classes, interfaces, types énumérés et annotations	SonarQube	Undocumented Api	Mineure
[JDOC-6.1]	Javadoc minimale des méthodes public	SonarQube	Undocumented Api	Mineure
[JDOC-6.2]	Javadoc recommandée pour les méthodes « complexes »			
[JDOC-6.3]	Tags requis pour la Javadoc des méthodes et constructeurs	SonarQube	Undocumented Api	Mineure
[JDOC-7.1]	Documentation des exceptions vérifiées	SonarQube	Undocumented Api	Mineure
[JDOC-7.2]	Documentation des exceptions non-vérifiées			

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[JDOC-8]	Différenciation des tags Javadoc personnalisés et des annotations			
[JDOC-9]	Le format et l'indentation des commentaires Javadoc			

5.1.2. [JDOC-1] La validation de la Javadoc

Les commentaires Javadoc **doivent** être valides.

Javadoc définit un certain nombre de tags qu'il convient d'utiliser à bon escient. On trouve ainsi, dans l'ordre : [@param](#) [@return](#) [@throws](#) [@exception](#) [@see](#) [@since](#) [@serial](#) [@serialField](#) [@serialData](#) [@deprecated](#) [{@link}](#).

Ces tags permettent de définir des caractéristiques normalisées. Ils sont connus des IDEs. Le squelette Javadoc correspondant peut être automatiquement construit par l'IDE. Cependant il est indispensable au développeur de connaître la sémantique de ces tags Javadoc standards et de les documenter. Les commentaires générés par les formateurs (par exemple DOCUMENTEZ_MOI) ne sont pas considérés comme valides et devront être substitués par du commentaire textuel.

5.1.3. [JDOC-2] Convention d'écriture et de style

[JDOC-2.1] De façon générale il est recommandé de choisir des formulations brèves, tout en évitant les abréviations acronymes sauf si elles sont connues de tous. Ainsi la description d'une méthode commencera par un verbe.

La documentation d'une méthode ne doit pas se borner à reprendre le nom de la méthode, elle doit apporter un plus, même si le code semble auto-documenté.

[JDOC-2.2] Il est recommandé d'utiliser la balise Javadoc `<code>` pour les mots clefs réservés et les noms lorsqu'ils sont cités dans les descriptions. Cela comprend :

- les mots clefs Java ;
- les noms de paquetage ;
- les noms de classes ;
- les noms de méthodes ;
- les noms d'interfaces ;
- les noms de champs, de variables ;
- les noms d'arguments ;
- les exemples de codes.

5.1.4. [JDOC-3] Règles d'utilisation des balises HTML dans les commentaires Javadoc

[JDOC-3.1] Si des tags HTML de mise en forme (PRE, TT, EM, etc.) sont utilisés dans les commentaires Javadoc, il **faut** les fermer.

[JDOC-3.2] Il est **interdit** d'utiliser des tags HTML de structure tels que H<n=1,2,...>, HR, TABLE car ils sont utilisés par Javadoc pour formater la documentation.

5.1.5. **[JDOC-4]** Documentation Javadoc des paquetages

[JDOC-4.1] Chaque paquetage **doit** contenir un fichier package-info.java

A noter que les fichiers Javadoc **package.html**, qui fournissent une description du paquetage au format HTML, sont encore tolérés pour les projets existants. Les nouveaux projets doivent privilégier le fichier package-info.java, introduit depuis la version 5 de Java.

[JDOC-4.2] Il est également **recommandé** d'éditer les fichiers **overview.html** permettant de fournir un résumé de plusieurs paquetages au format HTML. Ce fichier doit être placé dans le répertoire qui inclut les paquetages décrits.

5.1.6. **[JDOC-5]** Règles de documentation des entêtes Javadoc des classes, interfaces, types énumérés et annotations

[JDOC-5.0] Toute classe, interface, type énuméré ou annotations possède **obligatoirement** un en-tête de documentation au format Javadoc. Cet en-tête sera construit généralement à l'aide des Environnements de Développement Intégré pour Java. L'en-tête doit fournir au développeur qui utilise cette classe les informations suivantes :

Les rubriques avec une * **doivent** obligatoirement apparaître dans l'en-tête, les autres sont simplement recommandées (l'ordre est à respecter).

Rubrique	Réf.	Utilisation	Tag à utiliser
* Description / rôle	[JDOC-5.1]	Indiquer ici la raison de cette classe / interface, ce qu'elle fait de manière générale. Pour illustrer son rôle, les principaux services de la classe / interface peuvent être cités. De même, des exemples peuvent être fournis.	aucun
Bogues ou défauts connus	[JDOC-5.2]	Indiquer ici les dysfonctionnements connus et encore non corrigés de la classe / interface.	aucun
	[JDOC-5.3]	Sous-règle supprimée depuis la version 2.0.5.	
	[JDOC-5.4]	Sous-règle supprimée depuis la version 2.0.2.	
	[JDOC-5.5]	Sous-règle supprimée depuis la version 2.0.5.	
Date	[JDOC-5.6]		Date
Voir aussi	[JDOC-5.7]	Indiquer ici les éléments (principalement les classes / interfaces) dont la connaissance peut être utile pour la bonne compréhension de la classe / interface.	@see
Depuis	[JDOC-5.8]	Indiquer ici depuis quelle version de l'appliquatif cette classe / interface est disponible.	@since

Obsolète	[JDOC-5.9]	Indiquer ici que la classe / interface est devenue obsolète et qu'il est préférable d'utiliser une autre classe / interface.	<code>@deprecated</code>
----------	-------------------	--	--------------------------

Exemple d'en-tête Javadoc d'une classe :

```
/**
 * La classe <code>Auteur</code> caractérise un écrivain ayant écrit au moins un
 livre (roman, nouvelle, poème, ...).
 * Il est possible d'obtenir depuis un auteur, sa bibliographie, les livres publiés
 (choix possible par éditeur, par
 * année). Bogue connus : la méthode compterLivres soulève une exception dans le
 cas où l'auteur n'a pas écrit de livre
 * au lieu de renvoyer simplement 0
 * $Date: 2006/08/09 09:25:07 $
 *
 * @see Livre
 * @see Livre#obtenirAuteur()
 * @see Editeur
 * @since GestBibli v1.0
 */
public class Auteur
{...
```

Exemple d'en-tête Javadoc d'une annotation :

```
/**
 * Décrit les demandes de travaux qui ont
 * conduit à la présence de l'élément annoté
 * $Date: 2006/08/09 09:25:07 $
 *
 */
public @interface DemandeDeTravaux
{
    int id();

    String synopsis();

    String engineer() default "[non assigné]";

    String date() default "[non réalisé]";
}
```

5.1.7. [JDOC-6] Les règles de documentation Javadoc des méthodes, champs et constructeurs

[JDOC-6.1] Un commentaire Javadoc **doit** être à minima défini pour chaque méthode, champ ou constructeur déclarés `public`, `package` ou `protected`.

Cette règle devra être scrupuleusement respectée, et particulièrement dans le cadre de classes participant à un héritage. La classe mère en particulier devra alors contenir la documentation précise des contextes d'utilisation et des effets induits par la surcharge de chacune de ses méthodes (`public`, `package` et `protected`).

[JDOC-6.2] Il est également **recommandé** de documenter toutes les méthodes « métier » « complexes » (dont le nombre de lignes est supérieur à 30 ou dont la complexité cyclomatique est supérieure à 8), même si elles ne sont que `package` ou `private`.

[JDOC-6.3] Les tags suivants **sont requis** à minima dans la documentation Javadoc de toutes les méthodes et constructeurs :

1. **@return** est requis (sauf pour les constructeurs et méthodes sans valeur de retour (**void**)) ;
2. **@param** est requis (sauf s'il n'y a pas de paramètre) ;
3. **@exception** est requis s'il y a des exceptions propagées par la méthode.

5.1.8. [JDOC-7] Les Règles de documentation Javadoc des exceptions (vérifiées et non-vérifiées)

Cette section précise les règles de documentation avec le tag Javadoc **@throws** et non pas les règles de développement des exceptions qui font l'objet d'un autre chapitre.

Le but du tag **@throws** est d'indiquer quelles exceptions le programmeur :

- doit attraper (**catch**) pour les exceptions vérifiées (*checked*) ;
- peut attraper (**catch**) pour les exceptions non-vérifiées (*unchecked*).

[JDOC-7.1] Il est demandé de **commenter** avec le tag **@throws** toutes les exceptions vérifiées (*checked*) déclarées dans la clause **throws**.

[JDOC-7.2] Il est également **recommandé** de documenter toutes les exceptions non-vérifiées (*unchecked*) que l'appelant pourrait raisonnablement attraper (**catch**). Ces exceptions non-vérifiées n'apparaîtront pas dans la signature de la méthode (dans la clause **throws**).

La documentation des exceptions non-vérifiées est encouragée mais est laissée à la discrétion du concepteur de l'API. En documentant une exception non-vérifiée, on permet à l'appelant de « traduire » une exception non-vérifiée dépendante de l'implémentation en une autre exception plus appropriée pour l'abstraction exportée de l'appelant.

5.1.9. [JDOC-8] Différenciation des tags Javadoc personnalisés et des annotations

Depuis Java 5, il est possible d'enrichir le code avec des annotations et des tags personnalisés. Il convient de ne pas confondre annotations et tags :

- les annotations n'affectent pas directement la sémantique de la programmation mais affectent la façon dont les programmes sont traités par les outils et les bibliothèques, qui à leur tour peuvent affecter la sémantique d'un programme au moment de son exécution ;
- les tags sont conçus pour ajouter structure ou contenu à la documentation (on peut en effet mettre à profit les options Javadoc **-tag** ou **-taglet** pour créer des tags personnalisés). Ces tags ne doivent jamais avoir un effet sur la sémantique d'un programme.

Pour ne pas les confondre, il est **demandé** d'utiliser une majuscule pour une annotation personnalisée et une minuscule pour un tag.

Par exemple l'annotation **@Deprecated** sera utilisée pour alerter le compilateur et le tag **@deprecated** pour le commentaire texte.

5.1.10. [JDOC-9] Le format et l'indentation des commentaires Javadoc

La première ligne de commentaire ne **doit** contenir que `/**`

Les lignes de commentaires suivantes **doivent** obligatoirement commencer par un espace et une étoile. Toutes les premières étoiles doivent être alignées.

La dernière ligne de commentaires ne **doit** contenir que `*/` précédé d'un espace.

Exemple :

```
/**
 * Description de la methode
 */
public void maMethode()
```

5.2. Le format des commentaires de traitements (non Javadoc)

5.2.1. Finalité, critères de sélection, sources et contraintes

La finalité des règles relatives aux commentaires de traitement est de faciliter la lisibilité et la maintenance par une documentation uniforme et adéquate du code.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[COM-1]	Règles d'écriture des commentaires de traitement			Mineure
[COM-2]	Le format des commentaires de traitement			Mineure

5.2.2. [COM-1] Règles d'écriture des commentaires de traitement

Ces commentaires doivent évidemment ajouter du sens et des précisions au code : ils ne doivent pas reprendre ce que le code exprime mais expliquer clairement son rôle.

- Les commentaires doivent offrir une autre vue sur le code et donner des informations non disponibles à sa simple lecture ;
- Les commentaires doivent contenir des informations facilitant la lecture et la compréhension du code ;
- Les commentaires doivent décrire "pourquoi" quelque chose est fait et pas uniquement "comment" ;
- La formulation des commentaires doit être simple et concise (style notes) ;
- Les commentaires ne doivent pas être une reformulation ou une duplication en langue française du code écrit.

Les méthodes (et particulièrement les méthodes dites « métier ») les plus complexes (dont le nombre de lignes est supérieur à 30, ou dont la complexité cyclomatique est supérieure à 8) **devraient** avoir un taux de commentaire important à l'intérieur de leur corps pour expliquer leur logique (taux de commentaire > 30 %).

5.2.3. [COM-2] Le format des commentaires de traitements

Il existe plusieurs styles de commentaires de traitements :

- les commentaires sur une ligne ;
- les commentaires sur une portion de ligne ;
- les commentaires multi-lignes.

[COM-2.0] Il est **demandé** de mettre un espace après le délimiteur de début de commentaire et avant le délimiteur de fin de commentaire lorsqu'il y en un, afin d'améliorer sa lisibilité.

Les commentaires ne doivent pas être inclus dans des "boîtes" réalisées avec des astérisques (*) ou autres caractères. Cette mise en forme est trop sophistiquée et de ce fait trop difficile à maintenir.

5.2.3.1. [COM-2.1] Le format des commentaires de traitements d'une ligne et de plusieurs lignes

Les commentaires mono-ligne sont définis entre les caractères /* et */ sur une même ligne.

Exemple :

```
if (i < 10)
{
    /* commentaires utiles au code */
    ...
}
```

Ce type de commentaires **doit** être précédé d'une ligne blanche (l'accolade pourra être considérée comme une ligne blanche dans ce cas précis) et doit suivre le niveau d'indentation courant.

Les commentaires multi-lignes sont définis entre les caractères /* et */ sur plusieurs lignes.

Exemple :

```
/*
 * Commentaires sur plusieurs lignes
 * utiles au code
 */
```

Ce type de commentaires **doit** être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

5.2.3.2. [COM-2.2] Le format des commentaires de traitements de fin de ligne

Ce type de commentaire peut délimiter un commentaire sur une ligne complète ou une fin de ligne.

Exemple :

```
i++;    // commentaires utiles au code
```

Ce type de commentaire peut apparaître sur la ligne de code qu'elle commente mais il **faut** inclure une unité d'indentation (constituée de 4 espaces) qui permette de séparer le code et le commentaire.

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner.

Exemple :

```
i++;    // commentaire utile au code
j++;    // second commentaire utile au code
```

6. Règles de formatage de code

6.1. Finalité, critères de sélection, sources et contraintes

La finalité des règles relatives au formatage de code est de faciliter la lisibilité et donc la maintenabilité du code en participant à l'uniformité de code.

On retrouvera ainsi dans ce chapitre toutes les règles d'usage des séparateurs et règles de pur formatage tels que les retours à la ligne, les lignes blanches, les espaces, les indentations.

Elles tendent ensemble à rendre le code moins « dense », plus uniforme et donc plus lisible.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[FORM-1]	L'indentation et accolades	SonarQube	IndentationCheck RightCurlyBraceStartLineCheck LeftCurlyBraceStartLineCheck S00105	Mineure
[FORM-2]	Les lignes blanches			Mineure
[FORM-3]	Les espaces	CheckStyle	Whitespace After Whitespace Around No Whitespace After No Whitespace Before	Mineure
[FORM-4]	La coupure de ligne longue			Mineure
[FORM-5]	Une instruction par ligne	SonarQube	S00122	Mineure
[FORM-6]	Les instructions composées			Mineure
[FORM-7]	L'instruction <code>return</code>			Mineure
[FORM-]	L'instruction <code>if</code> : indentations	SonarQube	S00121	Mineure

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
8.1]				
[FORM-8.2]	L'instruction <code>if</code> : l'usage des parenthèses			Informati ve
[FORM-9]	L'instruction <code>for</code>	Checkstyle	Whitespace After (tokens = SEMI)	Mineure
[FORM-10]	Formatage de la boucle <code>for</code> étendue	Checkstyle	Whitespace Around	Mineure
[FORM-11]	L'instruction <code>while</code>	SonarQube	LeftCurlyBraceStart LineCheck RightCurlyBraceSta rtLineCheck	Mineure
[FORM-1.2]	L'instruction <code>do-while</code>	SonarQube	LeftCurlyBraceStart LineCheck RightCurlyBraceSta rtLineCheck	Mineure
[FORM-13.1]	L'instruction <code>switch</code> : indentations	SonarQube	LeftCurlyBraceStart LineCheck RightCurlyBraceSta rtLineCheck	Mineure
[FORM-13.2]	Le cas <code>default</code> doit être le dernier de l'instruction <code>switch</code>	SonarQube	SwitchLastCaseIsD efaultCheck	Mineure
[FORM-13.3]	Règle remplacée par [PROG-SWITCH-2] depuis la version 2.0.0			
[FORM-13.4]	Les labels autres que <code>case</code> sont interdits dans l'instruction <code>switch</code>	SonarQube	S1219	Mineure
[FORM-14]	Les instructions <code>try-catch</code>	SonarQube	LeftCurlyBraceStart LineCheck RightCurlyBraceSta rtLineCheck	Mineure
[FORM-15]	Format de déclaration des imports	SonarQube	S2208	Majeure
[FORM-16]	Format de déclaration des imports statiques	CheckStyle	AvoidStarImport	Mineure
[FORM-17]	Format de déclaration des méthodes retournant un tableau	SonarQube	S1197	Mineure
[FORM-18]	Format de déclaration des variables	SonarQube	S1659	Mineure

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Criticité
[FORM-19]	Format d'assignement des variables	SonarQube	AssignmentInSubExpressionCheck	Mineure
[FORM-20]	Format des structures conditionnelles	SonarQube	S1774	Mineure

6.2. Les séparateurs du code : indentations, lignes blanches, et espaces

6.2.1. [FORM-1] Indentation et accolades

[FORM-1.1-INDENT] L'unité d'indentation doit être constituée de 4 espaces.

[FORM-1.1-TAB] Il n'est **pas autorisé** d'utiliser les tabulations pour l'indentation.

[FORM-1.2] Deux formats sont acceptés dans l'utilisation des accolades, une fois l'un ou l'autre format choisi, il **doit** être adopté sur tout le code source du projet.

Format préféré :

- L'accolade ouvrante qui définit le début du bloc de code **doit** se situer sur une nouvelle ligne, au même niveau d'indentation que la déclaration.
- L'accolade fermante **doit** être sur une ligne séparée dont le niveau d'indentation correspond à celui de la déclaration. Une exception tolérée concerne un bloc de code vide : dans ce cas les deux accolades peuvent être sur la même ligne.

Format Sun : il est identique au format précédant et ne diffère que par le placement de l'accolade à la fin de la ligne qui contient la déclaration.

Exemple (mode Sun, toléré):	Exemple (préféré) :
<pre> MaClasse(String nom, String prenom) { this.nom = nom; this.prenom = prenom; } void faitQuelqueChose() { if (condition) { faitCeci(); } else { faitCela(); } } </pre>	<pre> MaClasse(String nom, String prenom) { this.nom = nom; this.prenom = prenom; } void faitQuelqueChose() { if (condition) { faitCeci(); } else { faitCela(); } } </pre>

6.2.2. [FORM-2] Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

[FORM-2.1] Deux lignes blanches **doivent** toujours séparer deux sections d'un fichier source et les définitions des classes et des interfaces.

Une ligne blanche **doit** toujours être utilisée dans les cas suivants :

- **[FORM-2.2]** avant la déclaration d'une méthode. En cas de Javadoc au-dessus de la déclaration de la méthode, la ligne blanche doit se trouver au-dessus de la Javadoc ;
- **[FORM-2.3]** entre un groupe (au moins deux) déclarations de variables locales et la première ligne de code qui les suit ;
- **[FORM-2.4]** avant un commentaire multi ou mono ligne ;
- **[FORM-2.5]** avant chaque section logique dans le code d'une méthode.

6.2.3. **[FORM-3]** Les espaces

Un espace vide **doit** toujours être utilisé dans les cas suivants :

- **[FORM-3.1]** entre un mot clé et une parenthèse ;
- **[FORM-3.2]** après chaque virgule dans une liste d'argument ;

Exemple:

```
while (i < 10)
```

- **[FORM-3.3]** tous les opérateurs binaires, opérateurs d'affectation (=, +=, -=, *=, /=, %=), opérateurs relationnels (>, >=, <, <=, ==, !=) et opérateurs logiques (&&, ||, !) **doivent** avoir un blanc qui les précède et les suit ;

Exemple :

```
a = (b + c) * d;           // opérateur d'affectation
x <= n;                   // opérateurs binaires
if (score != SCORE_MAX)   // opérateur relationnel
if ((a == true) || (b == true)) // opérateurs logiques
```

- **[FORM-3.4]** les conversions de type explicites (cast) **doivent** être suivi d'un et un seul espace ;

Exemple :

```
i = ((int) (valeur + 10));
```

- **[FORM-3.5]** Il **ne faut pas** mettre d'espace entre un nom de méthode et sa parenthèse ouvrante.
- **[FORM-3.6]** Il **ne faut pas** non plus mettre de blanc avant et après les opérateurs unaires, tels que les opérateurs d'incrément '++' et de décrétement '--'.

Exemple :

```
i = getTaille();
```

```
i++;
```

Exemple :

```
for (int index = 0; index < list.size(); ++index)
{
    /* code */
}
```

6.2.4. [FORM-4] La coupure de ligne longue

Il arrive parfois qu'une ligne de code soit très longue (cf. [METR-CLASS-2]: plus de 140 caractères).

Dans ce cas, il est alors recommandé de couper cette ligne en plusieurs, tout en respectant quelques règles :

- [FORM-4.1] La coupure d'une expression
 - couper la ligne avant un opérateur ou une instruction ;
 - ajouter une tabulation à partir du premier élément de l'expression.

Exemple :

```
maMethode(parametre1, parametre2, parametre3)
    throws monException
```

- [FORM-4.2] La coupure d'une expression au niveau des paramètres
 - couper la ligne après une virgule ;
 - aligner le paramètre coupé sur le premier paramètre de l'expression.

Exemple :

```
maMethode(parametre1, parametre2, parametre3
          parametre4, parametre5);
```

6.3. Le formatage et l'indentation des instructions

6.3.1. [FORM-5] Une instruction par ligne

Même s'il est possible de mettre plusieurs instructions sur une ligne, chaque ligne ne **doit** avoir qu'une seule instruction.

Contre-exemple :

```
i = getTaille();
i++; j++;
```

Exemple :

```
i = getTaille();
i++;
j++;
```

6.3.2. [FORM-6] Les instructions composées

Elles correspondent à des instructions qui utilisent des blocs de code.

Les instructions incluses dans ce bloc sont encadrées par des accolades et **doivent** être indentées.

Un bloc de code **doit** être défini pour chaque traitement même si le traitement ne contient qu'une seule instruction. Cela facilite l'ajout d'instructions et évite des erreurs de programmation.

Exemple d'accolades en mode Sun (toléré) :	Exemple d'accolades (préféré) :
<pre>if (condition) { traitements; } else { traitements; }</pre>	<pre>if (condition) { traitements; } else { traitements; }</pre>

6.3.3. [FORM-7] L'instruction return

Elle ne **doit** pas utiliser de parenthèses, sauf si celles-ci facilitent la compréhension.

Exemples :

```
return valeur;
return (personne.isHomme()? 'M' : 'F');
```

6.3.4. [FORM-8] L'instruction if

- [FORM-8.1] Elle **doit** avoir une des formes suivantes :

```
if (condition)
{
    traitements;
}

if (condition)
{
    traitements;
}
else
{
    traitements;
}

if (condition)
{
    traitements;
}
else
    if (condition)
    {
        traitements;
    }
else
```



```
{
    traitements;
}
```

Même si cette forme est syntaxiquement correcte, il est **interdit** d'utiliser l'instruction `if`, `else` ou `elseif` sans accolades :

Contre-exemple :

```
if (i == 10)
    i = 0;    // il manque les accolades
else
    i = 1;    // il manque ici aussi les accolades
```

- **[FORM-8.2]** L'usage des parenthèses :

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes liés à la priorité des opérateurs.

Contre-exemple :

```
if (i == j && m == n)    // à éviter
```

Exemple correct :

```
if ((i == j) && (m == n))    // à utiliser
```

De même si la condition dans un opérateur ternaire `?` : contient un opérateur binaire, cette condition devrait être mise entre parenthèses.

Exemple :

```
( i >= 0 ) ? i : -i;
```

6.3.5. **[FORM-9]** L'instruction `for`

Elle **doit** avoir la forme suivante, chaque expression dans une boucle `for` **doit** être séparée par un espace :

```
for (initialisation; condition; mise à jour)
{
    /* traitements */
}
```

6.3.6. **[FORM-10]** Formatage de la nouvelle boucle `for` étendue

Java 5 introduit une manière supplémentaire d'utiliser le mot clé `for`, qui s'avère être beaucoup plus légère et facile à mettre en oeuvre. Elle **doit** avoir la forme suivante :

```
for (TypeDeLInstance nomDeLInstanceIteree : instanceIterable)
{
    /* traitements */
}
```

On rappelle que `instancelIterable` doit être un tableau ou une instance de la nouvelle interface `java.lang.Iterable`. L'interface `java.util.Collection` étend maintenant `Iterable`.

6.3.7. [FORM-11] L'instruction while

Elle **doit** avoir la forme suivante :

```
while (condition)
{
    /* traitements */
}
```

6.3.8. [FORM-12] L'instruction do-while

Elle **doit** avoir la forme suivante :

```
do
{
    /* traitements */
}
while (condition);
```

6.3.9. [FORM-13] L'instruction switch

- [FORM-13.1] Elle **doit** avoir la forme suivante :

```
switch (condition)
{
    case ABC:
        /* traitements */
        break;
    case DEF:
        /* traitements */
        break;
    case XYZ:
        /* traitements */
        break;
    default:
        /* traitements */
        break;
}
```

- [FORM-13.2] Le cas default **doit** être le dernier de l'instruction switch

Contre-exemple :

```
void bar(int a)
{
    switch (a)
    {
        case 1:
            /* traitements */
            break;
        default:
            /* traitements */
            break;
        case 2:
```

```
        /* traitements */
        break;
    }
}
```

Le cas default est obligatoire, se référer à la règle [PROG-SWITCH-3].

- **[FORM-13.3]** Il est préférable de terminer les traitements de chaque cas avec une instruction `break`. Si ce n'est pas le cas, il est recommandé d'ajouter un commentaire `/* falls through */` là où le `break` aurait figuré. Même si elle est redondante, une instruction `break` devrait être incluse en fin de traitement du cas **default**.
- **[FORM-13.4]** Les labels autres que `case` sont interdits dans l'instruction `switch`.
Utiliser un label autre que le mot-clé `case` à l'intérieur d'un `switch` est syntaxiquement correct, mais peut confondre le lecteur, et donc **interdit**.

Contre-exemple :

```
void bar(int a)
{
    switch (a)
    {
        case 1:
            /* traitements */
            break;
        monlabel: // syntaxiquement correct, mais provoque la confusion !
            break;
        default:
            break;
    }
}
```

6.3.10.[FORM-14] Les instructions try-catch

Elles **doivent** avoir la forme suivante :

```
try
{
    /* traitements */
}
catch (Exception1 e1)
{
    /* traitements; */
}
catch (Exception2 e2)
{
    /* traitements */
}
finally
{
    /* traitements */
}
```

6.4. *Format et indentations des déclarations et assignements*

6.4.1. [FORM-15] Format de déclaration des imports

Le bloc import **doit** contenir tous les types et seulement les types utilisés par la classe du fichier. Par conséquent, la présence de type pleinement qualifié dans le corps de la classe **n'est pas autorisée**. L'étoile ne doit donc pas être utilisée dans la déclaration d'imports de librairie.

Contre-exemple :	Exemple correct :
<code>import java.io.*;</code>	<code>import java.io.file;</code>

On veillera évidemment également à ne rien importer du paquetage `java.lang` (automatiquement importé) et ne pas importer un type contenu dans le même paquetage.

6.4.2. [FORM-16] Format de déclaration des imports statiques

Depuis Java 5, la fonctionnalité d'import statique de paquetage permet de réduire le code à écrire concernant les membres de classe. Cette fonctionnalité est développée dans la JSR 201. L'utilisation de l'importation statique s'applique à tous les membres de classe : constantes et méthodes `static` de la classe/interface importée.

Contre-exemple (en masse) :	Exemple correct (de façon fine) :
<code>import static java.lang.Math.*;</code>	<code>import static java.lang.Math.PI;</code>

Une fois ces membres de classe importés ils peuvent être utilisés sans qualification :

```
double r = cos(PI * theta);

// remplace :
// double r = Math.cos(Math.PI * theta);
```

L'utilisation systématique de cette fonctionnalité d'import statique peut rendre le code illisible et difficilement maintenable, polluant son espace de nommage (*namespace*) avec tous les membres de classe importés. Les lecteurs du code (même l'auteur, quelques mois après l'avoir écrit) ne sauront plus de quelles classes un membre statique est originaire.

Importer tous les membres statiques d'une classe peut être particulièrement dommageable à la lisibilité, il est donc **fortement recommandé** d'importer chaque membre individuellement et d'**éviter le caractère étoile**, comme pour l'import classique.

6.4.3. [FORM-17] Format de déclaration des méthodes retournant un tableau

Java propose deux syntaxes pour déclarer une méthode qui retourne un tableau. La deuxième syntaxe est **exigée** :

Contre-exemple :	Exemple :
<code>public int notes() [] {</code>	<code>public int[] notes() {</code>

6.4.4. [FORM-18] Format de déclaration des variables

Il ne **doit** y avoir qu'une seule déclaration d'entité par ligne.

Contre-exemple :	Exemple :
<code>String nom, prenom;</code>	<code>String nom; String prenom;</code>

Il **faut** a fortiori éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

Contre-exemple :
<code>int age, notes[]; // ce type de déclaration est à proscrire</code>

Il est **demandé** d'aligner le type, l'identifiant de l'objet et les commentaires si plusieurs déclarations se suivent pour retrouver plus facilement les divers éléments.

Exemple :
<pre>String nom; // nom de l'eleve String prenom; // prenom de l'eleve int age; // age de l'eleve int notes[]; // notes de l'eleve</pre>

Il est également **recommandé** d'initialiser les variables au moment de leur déclaration si la valeur d'initialisation est significative dans l'algorithme. Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc (un bloc est un morceau de code entouré par des accolades). La seule exception concerne la déclaration de la variable utilisée comme index dans une boucle.

6.4.5. [FORM-19] Format d'assignement des variables

Il est **interdit** d'assigner la même valeur à plusieurs variables sur la même ligne.

Contre-exemple :
<code>i = j = k; // cette forme n'est pas autorisée</code>

Il **ne faut pas** utiliser l'opérateur d'assignement imbriqué.

Contre-exemples :	Exemples corrects :
<code>valeur = (i = j + k) + m;</code>	<code>i = j + k; valeur = i + m;</code>
<code>valeur = i++;</code>	<code>valeur = i; i++;</code>

6.4.6. [FORM-20] Format des structures conditionnelles

Il est **préconisé** de ne pas utiliser d'opérateur ternaire pour faire des conditions uni-ligne. Cette pratique raccourcit le code mais peut vite devenir illisible lorsque le conditionnement se complique.

Contre-exemples :	Exemples corrects :
-------------------	---------------------

<pre>max = (a > b) ? a : b ;</pre>	<pre>If (a > b) { max = a; } else { max = b; }</pre>
<pre>String b (a == null a.lenght < 1) ? null : a.substring(1);</pre>	<pre>String b; if (a == null a.lenght < 1) { b = null; } else { b = a.substring(1); }</pre>

7. Règles de programmation : *design, patterns* et *anti-patterns* de code

7.1. Généralités

Ce chapitre couvre l'ensemble des règles de programmation relatives au *design* de code, à ses *patterns* et *anti-patterns* pouvant être exprimés sous forme de règles.

Il débute par un ensemble de recommandations et de règles autour de Java 7. Ces règles pourraient être élargies et exprimées ainsi : « Vous devez connaître les nouvelles fonctionnalités et API du langage ».

Suivent ensuite un ensemble des règles visant à améliorer l'encapsulation, la pertinence de l'utilisation des modificateurs et des niveaux d'accessibilité.

Le chapitre se poursuit sur des sous-ensembles de règles simples mais primordiales pour la lisibilité et la qualité du code source, autour de :

- la déclaration de variables ;
- la programmation des constructeurs et des méthodes ;
- la programmation des instructions `if`, `while` et `switch` ;
- la programmation de `clone` .

La chapitre se termine par une liste de règles plus générales encore et par une liste d'API interdites et/ou déconseillées.

7.2. Règles d'encapsulation, d'accessibilité et d'utilisation des modificateurs

7.2.1. Finalité, critères de sélection, sources et contraintes

L'ensemble des remarques, règles et recommandations introduites dans ce sous chapitre ont pour objectif :

- d'améliorer la pertinence des niveaux d'accessibilité ;
- de permettre une meilleure encapsulation ;
- de garantir une bonne utilisation des modificateurs, pour une meilleure conception objet.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[ENCAPS-0]	Il faut rendre chaque classe et chaque membre aussi inaccessibles que possible.	SonarQube	S2156 S2386	Critique
[ENCAPS-0.1]	Les classes et membres de classe ne doivent être <code>public</code> que s'ils sont utilisés à l'extérieur de leur paquetage.			
[ENCAPS-0.2]	Les (inner) classes et les membres de classe ne doivent être <code>package</code> que s'ils sont utilisés à l'extérieur de leur classe d'implémentation.			
[ENCAPS-0.3]	Les classes et les membres de classe ne doivent être <code>protected</code> que s'ils sont prévus pour un héritage.			

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[ENCAPS-0.4]	Les (inner) classes et les membres de classe devront rester <code>private</code> s'ils ne sont utilisés que dans leur classe d'implémentation.			
[ENCAPS-0.5]	Les variables d'instance sont par défaut <code>private</code>			
[ENCAPS-1]	Une classe abstraite doit contenir au moins une méthode abstraite.	SonarQube	S1694	Majeur
[ENCAPS-2]	Limiter les imports statiques.	PMD	AvoidStaticImport	Majeur
[ENCAPS-3]	Les interfaces conteneurs de constantes sont déconseillées.	SonarQube	S1214	Majeur
[ENCAPS-4]	Règles d'utilisation du modificateur <code>static</code> : une classe « utilitaire » ne doit pas avoir de constructeur public.	SonarQube	S1118	Majeur
[ENCAPS-5]	Règles d'appel des membres <code>static</code>			Bloquant

7.2.2. De l'utilisation du modificateur `public`

Il ne faut pas déclarer de variables d'instances ou de classes `public` sans raison valable et sans volonté de vouloir intentionnellement exposer une API.

La caractéristique la plus importante qui distingue un module de bonne facture d'un module de qualité médiocre réside dans sa capacité à cacher ces données et implémentations internes et à les séparer de façon propre de son API.

[ENCAPS-0] Il **fa**ut rendre chaque classe et chaque membre aussi inaccessibles que possible.

Cette règle/bonne pratique est décomposée en sous-règles, à travers [ENCAPS-0.1] [ENCAPS-0.2] et [ENCAPS-0.3].

- [ENCAPS-0.1] Les classes et membres de classe **ne doivent** être `public` **que** s'ils sont utilisés à l'extérieur de leur paquetage.

7.2.3. De l'utilisation du modificateur `package`

Lorsque l'on ne précise rien, un membre est dit *package* (ou *friendly*), accessible par toutes les classes du même paquetage (classe courante, sous-classes du même paquetage, non sous-classes du même paquetage) mais par aucune classe d'un paquetage différent, même par une classe fille.

Il est important de maîtriser cette notion et de ne pas laisser champs, méthodes et classes en accès *package* par défaut (ou négligence), tout comme l'accès `public` est motivé par la volonté d'exposer une API *package*. L'accès *package* doit venir d'une réelle volonté d'architecture logicielle (comme la séparation des responsabilités en *package*, et la non exposition d'une API en dehors d'un *framework*).

- [ENCAPS-0.2] Les (inner) classes et les membres de classe **ne doivent** être *package* **que** s'ils sont utilisés à l'extérieur de leur classe d'implémentation.

7.2.4. De l'utilisation du modificateur `protected`

Un élément `protected` (protégé) est accessible uniquement aux classes d'un paquetage et à ses classes filles.

Il est important de maîtriser cette notion et de ne pas laisser champs, méthodes et classes en accès `protected` par hasard ou négligence. Ce niveau d'accessibilité doit être motivé par la volonté d'exposer les classes et membres de classes en vue d'un héritage.

- **[ENCAPS-0.3]** Les classes et les membres de classe ne doivent être `protected` que s'ils sont prévus pour un héritage.

7.2.5. De l'utilisation du modificateur `private`

- **[ENCAPS-0.4]** Les (inner) classes et les membres de classe **doivent** rester `private` s'ils ne sont utilisés que dans leur classe d'implémentation.
- **[ENCAPS-0.5]** De plus, on veillera à ce que les variables d'instance **soient** `private`, par défaut et éventuellement `protected` (cf. [ENCAPS-0.3]). On préférera ainsi la création de méthodes `public` permettant d'assurer une protection lors de l'accès à la variable et éventuellement un contrôle lors de la mise à jour de leur valeur. Ainsi, une classe **ne doit pas** avoir de champs `public` à moins qu'il soit `final` ou `static final`.

7.2.6. Favoriser les classes immuables

Les objets immuables ont de nombreux avantages et cette charte encourage leur utilisation :

Ils sont simples, *threadsafe*, ne requièrent aucune synchronisation, sont partageables de façon transparente. Leur seul désavantage est qu'ils demandent une instance séparée par valeur distincte.

Bien que ce ne soit pas transposable en règle, il est recommandé ici que toute classe soit immuable, à moins d'avoir une bonne raison pour qu'elle ne le soit pas. Il faudrait alors encore minimiser sa mutabilité autant que possible (cf. [VAR-3] et cf. [CONSTRUCT-7]).

Comment garantir la caractère immuable d'une classe :

- ne pas fournir de méthode qui modifie l'objet (appelée *mutator*) ;
- s'assurer qu'aucune méthode ne peut-être surchargée ;
- rendre tous les champs `final` ;
- s'assurer de l'accès exclusif à tous composants mutables.

7.2.7. Concevoir et documenter l'héritage, sinon le prohiber

L'héritage de classe est une façon puissante de parvenir à la réutilisation du code, mais n'est pas toujours le meilleur outil pour y réussir. Utilisé de façon inappropriée, l'héritage de classe peut fragiliser le code. L'héritage sera plus sécurisé si la classe mère est spécifiquement conçue et documentée pour une extension.

La classe doit documenter précisément les effets induits par la surcharge de chacune de ses méthodes. La documentation Javadoc doit indiquer pour chaque méthode `public` ou `protected` quelles méthodes surchargeables elle invoque, dans quel ordre et quels sont les effets notables de chaque invocation (cf. [JDOC-6.1]).

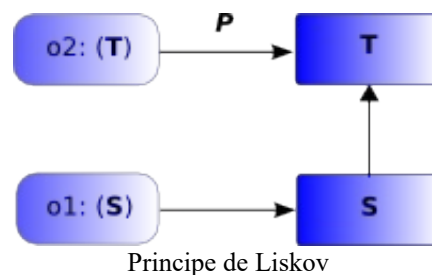
Il existe d'autres restrictions inhérentes à l'héritage, et les plus notables sont exprimées dans les règles sur les constructeurs : [CONSTRUCT-4] et [CONSTRUCT-5]. En effet, appeler une ou des méthodes surchargeables directement ou indirectement dans un constructeur peut s'avérer particulièrement dangereux, fragilisant.

7.2.7.1. Principe de Liskov

On veillera également à respecter le principe de Liskov : l'héritage est la faculté d'une sous-classe ou d'un sous-type d'hériter des propriétés de son parent et de les raffiner.

Les opérations applicables à une classe doivent s'appliquer à toutes ses sous-classes. Toute opération définie sur une classe devrait être applicable avec la même sémantique à toutes les sous-classes. En d'autres termes l'héritage ne peut modifier que l'implémentation des opérations.

Ainsi, si pour chaque objet *o1* de type *S*, il y a un objet *o2* de type *T* tel que pour tous les programmes *P* définis avec *T*, le comportement de *P* est inchangé quand *o1* est substitué avec *o2*, alors *S* est un sous type de *T* (cf. illustration ci-dessous).



7.2.8. [ENCAPS-1] Règles d'utilisation du modificateur **abstract**

Une classe abstraite suggère une implémentation incomplète qui doit être complétée par des sous classes implémentant les méthodes abstraites.

Si une classe est prévue pour être utilisée comme classe de base uniquement (et ne doit pas être instanciée directement), on préférera l'utilisation d'un constructeur **protected** (qui empêchera toute instanciation directe) à la spécification **abstract** de la classe. C'est pourquoi cette charte **interdit** qu'une classe abstraite ne contienne aucune méthode abstraite.

7.2.9. [ENCAPS-2] Limiter les imports statiques

Comme introduit avec de la règle de formatage des imports statiques (cf. [FORM-16]), l'utilisation trop fréquente de l'import statique peut rendre le code illisible et difficilement maintenable, polluant son espace de nommage (*namespace*).

Il est donc recommandé d'utiliser les imports statiques de façon parcimonieuse : il est recommandé d'utiliser cette fonctionnalité lorsque vous êtes tentés :

- de déclarer des copies locales de constantes ;
- d'utiliser abusivement l'héritage et de mettre des membres statiques dans une interface, pour ensuite hériter de cette interface. Il s'agit d'un *anti-pattern* appelé « Constant Interface Anti-pattern » (cf. [ENCAPS-3]).

C'est pourquoi il est fortement recommandé de limiter au nombre de 4 les imports statiques.

7.2.10. [ENCAPS-3] Éviter les « interfaces constantes »

Comme introduit ci-dessus, un des *anti-pattern* fréquemment retrouvé avant l'apparition des imports statiques est l'« interface constante » (« Constant Interface Anti-pattern ») : c'est à dire la création d'une interface non pas pour décrire un modèle de comportement mais comme conteneur de constantes. Un tel usage des interfaces est

déconseillé. En effet, cet usage entre en conflit avec l'orientation objet et expose souvent des détails d'implémentation qui se retrouvent publiés dans l'API.

Contre-exemple :

```
public interface ConstantInterfaceAntiPattern
{
    public static final double PI = Math.PI;

    public static final int JANVIER = Calendar.JANUARY;
    public static final int FEVRIER = Calendar.FEBRUARY;
    ...
}
```

Ce contre-exemple pourrait être facilement corrigé grâce à l'utilisation d'imports statiques de bibliothèques Java ou tierces, ou d'une classe utilitaire dont voici un exemple :

Exemple :

```
public class PhysicalConstants
{
    private PhysicalConstants()
    {} /* Préviens l'instanciation */

    public static final double AVOGADROS_NUMBER = 6.02214199e23;

    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Cet *anti-pattern* trouvera également souvent son « antidote » dans l'utilisation de types énumérés.

7.2.11.[ENCAPS-4] Règles d'utilisation du modificateur `static`

Une classe qui ne contient que des méthodes statiques (souvent dénommée utilitaire) **ne doit pas** avoir de constructeur public. La classe utilitaire doit posséder un constructeur privé (ou `protected`) pour empêcher son instanciation.

On peut également noter que si une classe utilitaire est parfois candidate à un *refactoring* sous forme de *singleton*, il faudra alors veiller à respecter les règles sur les *singletons* (cf.[THR-3]).

7.2.12. [ENCAPS-5] Règles d'appel des membres `static`

Pour plus de lisibilité et de clarté dans le code, il n'est pas autorisé d'utiliser des variables ou des méthodes statiques de classes à partir d'un objet instancié : **il ne faut pas** utiliser `maClasseInstanciée.methode()` mais `MaClasse.methode()`.

7.3. Règles de déclaration, initialisation et utilisation des variables

7.3.1. Généralités, finalités, critères de sélection, sources et contraintes

Ce sous chapitre précise un ensemble de règles relatives à la déclaration, l'initialisation et l'utilisation des variables. Toutes ces règles tendent vers une amélioration de la lisibilité et de la maintenabilité du code, et à la réduction des erreurs. On rappellera à ce sujet que :

- **minimiser** la portée des variables (et le fait de les déclarer quand elles sont utilisées pour la première fois et de les initialiser explicitement) et le nombre des variables (cf. [METR-CLASS-5]) sont des bonnes pratiques qui participent à cet aspect de qualité du code ;
- les règles de formatage et de nommage des variables introduites auparavant (cf. [FORM-18], [FORM-19], [NOM-11], [NOM-12], [NOM-13] [NOM-14] [NOM-15] et [NOM-16]) s'inscrivent également dans cette recherche de qualité autour de l'utilisation des variables.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[VAR-1]	La déclaration de variable “masquante” est interdite.			Majeure
[VAR-2]	Les variables locales et d'instance private non utilisées doivent être supprimées.	SonarQube	S1068 S1481 S1488	Mineure
[VAR-3]	Rendre les champs invariants en champs immuable (final).	PMD	ImmutableField	Majeure
[VAR-4]	Il est fortement déconseillé d'utiliser des constantes numériques en dur dans le code, celles ci devraient faire l'objet de déclarations de constantes.	SonarQube	S109	Mineure
[VAR-5]	Les transtypages impossibles, inutiles ou risqués sont interdits.	SonarQube	S2185 S2184	Critique
[VAR-6]	Éviter les types float et double pour les calculs exacts/monétaires. Utiliser int , long ou BigDecimal .	SonarQube	S1244	Critique

7.3.2. [VAR-1] La déclaration de variable “masquante” est interdite

Il est **proscrit** toute déclaration de variable masquant une variable définie dans un bloc parent, afin de ne pas complexifier inutilement le code. Cette règle rejoint les règles de nommage [NOM-15] et [NOM-16].

Contre-exemple :

```
int taille;
...
void maMethode()
{
```

```
int taille;
}
```

7.3.3. **[VAR-2]** Les variables locales et les variables d'instances **private** non utilisées **doivent** être supprimées

Contre-exemple :	Exemple corrigé :
<pre>public class CodeInutile { private static int FOO = 2; // Non utilisé public int addOne() { private int j = 5; // Non utilisé private int j = 6; // Non nécessaire int k = 5; // Non utilisé return j++; } }</pre>	<pre>public class CodeInutile { private int j = 6; public int addOne() { return j++; } }</pre>

7.3.4. **[VAR-3]** Rendre les champs invariants en champs immuable (**final**)

Il est **recommandé** de transformer en champs immuables (en ajoutant le modificateur **final**) tous les champs **private** qui ne changent jamais. Cela facilite alors une conversion éventuelle de classe vers une classe immuable (cf. § 7.2.6).

7.3.5. **[VAR-4]** L'utilisation de constantes

Il est **fortement déconseillé** d'utiliser des constantes numériques « en dur » dans le code, celles-ci devraient faire l'objet de déclarations de constantes, avec des noms explicites. Une exception concerne les valeurs -1, 0, 1 et 2 dans les boucles **for**.

On rappelle à ce sujet que ces constantes pourront éventuellement faire l'objet de typage à travers l'utilisation des types énumérés (cf. § Erreur : source de la référence non trouvée), mais ne devraient pas faire l'objet d'interface dédiées (cf. [ENCAPS-3]).

7.3.6. **[VAR-5]** Les transtypages impossibles, inutiles ou risqués sont interdits

Contrairement au C++, les transtypages avec Java ne sont pas effectués lors de la compilation. Ils ont un coût lors de l'exécution et quand ils sont inutiles, ils réduisent la lisibilité du code. C'est pourquoi les transtypages (cast) inutiles sont **interdits** (d'autant plus qu'ils sont repérables facilement et systématiquement par tous les IDE récents).

On notera que depuis Java 5, il est devenu extrêmement rare de nécessiter de l'usage de transtypage.

Par ailleurs, certains transtypages amènent inéluctablement à un **ClassCastException**. Ils sont évidemment à proscrire. Il est parfois aussi risqué de transtyper un **int**. Les cas suivants sont prohibés :

- un **int** est transtypé en **double** puis passé à **Math.ceil** ;
- un **int** est transtypé en **float** puis passé à **Math.round** ;

- le résultat de la division d'un **integer** est transtypé en **double** ou **float** ;
- le résultat d'une multiplication d'**integer** est transtypé en **long**.

Contre-exemple : division	Exemple corrigé :
<pre>int x = 2; int y = 5; /* mauvais résultat : 0.0 */ double value1 = x/y;</pre>	<pre>int x = 2; int y = 5; /* bon résultat : 0.4 */ double value2 = x/(double)y;</pre>

Contre-exemple : multiplication	Exemple corrigé :
<pre>long convertDaysToMilliseconds(int days) { return 1000*3600*24*days; }</pre>	<pre>Long convertDaysToMilliseconds(int days) { return 1000L*3600*24*days; }</pre>

7.3.7. [VAR-6] Éviter float et double si un résultat exact (monétaire) est requis

Les types **float** et **double** ont été conçus pour les calculs scientifiques et l'ingénierie. Ils réalisent de l'arithmétique binaire à virgule flottante qui a été conçue pour fournir des approximations efficaces et rapides sur une large échelle. Ils ne fournissent pas cependant de résultats exacts et ne **devraient** pas être utilisés quand ils ne sont pas requis.

Les types **float** et **double** sont particulièrement **inadéquats** pour le calcul monétaire. Il est en effet impossible de représenter de manière exacte 0.1 (ou tout autre puissance négative de 10) comme **float** ou **double**.

Supposez que vous ayez un euro en poche, vous achetez 9 bonbons à 10 cents, que vous reste-t-il en poche ?

```
System.out.println(1.00 - 9*.10);
/* surprise ! : il s'affiche : 0.09999999999999998 */
```

En utilisant **BigDecimal**, il me restera un peu plus d'argent :

```
final BigDecimal DIX_CENTS = new BigDecimal(".10");
int bonbonsAchetes = 0;
BigDecimal argentDePoche = new BigDecimal("1.00");
for (int i = 1; i <= 9; i++)
{
    bonbonsAchetes++;
    argentDePoche = argentDePoche.subtract(DIX_CENTS);
}
System.out.println(bonbonsAchetes + " bonbons achetes.");
/* il s'affiche : 9 bonbons achetes. */
System.out.println("Argent de poche restant : " + argentDePoche);
/* il s'affiche : Argent de poche restant : 0.10 */
```

Utiliser ainsi **BigDecimal** si le système doit contrôler les décimales et que le sur-coût de gestion de l'objet n'est pas trop important. **BigDecimal** apporte notamment l'avantage d'un contrôle total de l'arrondi (8 modes d'arrondi sont disponibles). Cette fonctionnalité est particulièrement intéressante quand l'arrondi est contractuellement défini.

Si la performance est primordiale, que les décimales peuvent être gérées par l'applicatif, et que les quantités ne sont pas trop grandes, utiliser **int** ou **long** :

```
final int DIX_CENTS = 10;
int bonbonsAchetes = 0;
int argentDePoche = 100; // 100 cents = 1 euro
```

```

for (int i = 1; i <= 9; i++)
{
    bonbonsAchetes++;
    argentDePoche = argentDePoche -DIX_CENTS;
}
System.out.println(bonbonsAchetes + " bonbons achetes.");
/* il s'affiche : 9 bonbons achetes. */
System.out.println("Argent de poche restant (cents) : " + argentDePoche);
/* il s'affiche : Argent de poche restant (cents) : 10 */

```

Il est ainsi possible d'utiliser `int` jusqu'à 9 chiffres (inclus) significatifs (*digit*), `long` jusqu'à 18 (inclus), mais au-delà `BigDecimal` **doit** être utilisé.

Attention à l'utilisation de `BigDecimal` : il est immuable (cf. [PROG-AUTR-7]) et le constructeur recommandé est `BigDecimal(String)` et non `BigDecimal(double)` (cf. [PROG-AUTR-6]).

7.4. Règles de programmation des constructeurs

7.4.1. Finalité, critères de sélection, sources et contraintes

Ce sous-chapitre précise un ensemble de règles relatives à la programmation des constructeurs. Toutes ces règles tendent vers une amélioration de la lisibilité et de la maintenabilité du code et à la réduction des erreurs. On rappellera à ce sujet que les règles de formatage (cf. [FORM-17]) et de documentation (cf. [JDOC-6]) s'inscrivent également dans cette recherche de lisibilité et d'uniformité autour de la programmation des constructeurs.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[CONSTRUCT-1]	Définir explicitement au moins un constructeur	SonarQube	S1258	Majeure
[CONSTRUCT-2]	Initialiser les variables d'instances dans les constructeurs	FindBugs	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	Majeure
[CONSTRUCT-3]	Appeler explicitement le constructeur hérité	PMD	CallSuperInConstructor	Majeure
[CONSTRUCT-4]	Les constructeurs ne doivent pas faire appel à des méthodes surchargeables directement ou indirectement.	SonarQube	S1699	Majeure
[CONSTRUCT-5]	Pas de comportement dans les constructeurs			
[CONSTRUCT-6]	Les blocs d'initialisation d'instance non statiques sont interdits.	SonarQube	S1171	Bloquante
[CONSTRUCT-7]	Une classe qui ne possède que des	SonarQube	S2974	Majeure

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[CONSTRUCT-8]	constructeurs private devrait être final . Une classe qui ne possède que des constructeurs private et aucune méthode statique est interdite car inutilisable.	PMD	MissingStaticMethodInNonInstantiatableClass	Bloquante

7.4.2. [CONSTRUCT-1] Définir au moins un constructeur

Il est **fortement recommandé** de définir explicitement au moins un constructeur pour chaque classe Java, plutôt que de s'appuyer sur le constructeur fourni par défaut par le compilateur (le constructeur sans paramètre). Il est même parfois préférable de baisser le niveau d'accessibilité du constructeur pour prévenir toute instantiation inopportune.

7.4.3. [CONSTRUCT-2] Initialiser les variables d'instances dans le constructeur

Il est préférable de toujours initialiser les variables d'instances dans les constructeurs, soit avec les valeurs fournies en paramètres du constructeur, soit avec des valeurs par défaut.

Exemple :

```
public class Personne
{
    protected String nom;

    protected String prenom;

    protected int age;

    Personne(String nom, String prenom, int age)
    {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
}
```

Il est possible d'appeler un constructeur dans un autre constructeur pour faciliter l'écriture.

7.4.4. [CONSTRUCT-3] Appeler explicitement le constructeur hérité

Il est recommandé de toujours appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille, grâce à l'utilisation du mot clé **super**.

Exemple :

```
public class Employe extends Personne
{
    private int matricule;

    public Employe(String nom, String prenom, int age, int matricule)
    {
```



```
        super(nom, prenom, age);
        this.matricule = matricule;
    }
}
```

7.4.5. [CONSTRUCT-4] Pas d'appel à des méthodes surchargeables dans les constructeurs

Il est recommandé que Les constructeurs ne fassent pas appel à des méthodes surchargeables directement ou indirectement.

Si cette règle est violée, il y a de forte probabilité que des problèmes surviennent dans le cycle de vie du projet, tout particulièrement dans les futures implémentations des classes filles, et ceci malgré la documentation éventuelle (cf. 7.2.7).

On peut en effet se retrouver alors dans des situations complexes où notamment les sous-classes seront incapables de construire leur classe mère ou forcées de répliquer le processus de construction.

Contre-exemple :

```
public class Senior
{
    public Senior()
    {
        toString();    // nok règle
    }

    public String toString()
    {
        return "Je suis un Senior";
    }
}

public class Junior extends Senior
{
    private String name;

    public Junior()
    {
        super();    // va automatiquement entrainer une NullPointerException
        name = "Je suis un Junior";
    }

    public String toString()
    {
        return name.toUpperCase();
    }
}
```

7.4.6. [CONSTRUCT-5] Pas de comportement dans les constructeurs

On rajoutera même à ces règles précédentes qu'il est **fortement déconseillé** d'introduire tout comportement dans les constructeurs, et tout particulièrement de prise de ressource.

Si l'initialisation d'un objet est particulièrement complexe, elle doit être traitée à part, en dehors de la construction de l'objet, et doit faire l'objet d'une méthode dédiée d'« initialisation » et d'appels explicites de la part de ses clients. Il en va de même que pour la « terminaison » de l'objet (cf. Méthode de « terminaison »).

7.4.7. [CONSTRUCT-6] Les blocs d'initialisation d'instance non **static** sont interdits

Bien que Java offre cette syntaxe pour initialiser les variables non **static** pour chaque objet, les initialisations d'instance non **static** sont **interdites**, car rarement utilisées et peu claires.

Contre-exemple :

```
public class ConstructSix
{
    /* Ce bloc sera executé avant n'importe quel appel au constructeur */
    {
        /* traitements */
    }
}
```

7.4.8. [CONSTRUCT-7] Une classe qui ne possède que des constructeurs **private** devrait être final

Pour plus d'explication, cf. &7.2.6.

7.4.9. [CONSTRUCT-8] Une classe qui ne possède que des constructeurs **private** et aucune méthode statique est interdite car inutilisable

7.5. Règles générales sur les méthodes

7.5.1. Finalité, critères de sélection, sources et contraintes

Ce sous chapitre précise un ensemble de règles relatives à la programmation des méthodes. Toutes ces règles tendent vers une amélioration de la lisibilité et de la maintenabilité du code et à la réduction des erreurs. On rappellera à ce sujet que les règles de formatage (cf. [FORM-17]), de nommage (cf. [FORM-17], [NOM-17], [NOM-18] et [NOM-19], de documentation (cf. [JDOC-5]) et de métriques (cf. [METR-METH-1], [METR-METH-2], [METR-METH-3], [METR-METH-4] et [METR-METH-5]) associées aux méthodes s'inscrivent également dans ce cadre.

7.5.2. Généralités sur la conception des méthodes

Si les signatures des méthodes sont conçues avec grande attention, elle rendront une API plus facile à apprendre, à utiliser et moins sujette au bogues. Il faudra résister à la tentation d'écrire un nombre trop important de fonctions utilitaires : ne le faire que si elles sont fréquemment utilisées. Il faudra veiller à réduire ainsi au maximum la taille (cf. [METR-CLASS-6]) et la complexité d'une API (cf. [METR-METH-2]), et à chasser le code mort **private** (cf. [METR-METH-3]) mais aussi **public**.

Il **faudra** également favoriser l'utilisation d'interfaces plutôt que de classes dans le typage des paramètres.

Et tout comme pour les constructeurs, il faudra également programmer les méthodes en gardant à l'esprit que les clients des classes feront de leur mieux pour détruire les invariants. Pour cela, il est parfois essentiel de ne renvoyer que des copies défensives des variables d'instances mutables des classes.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outillage	Criticité
[METH-1]	Les méthodes vides et non commentées sont interdites.	SonarQube	S1186	Mineure

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METH-2]	Les méthodes surchargeantes inutiles sont interdites.	SonarQube	S1185	Majeure
[METH-3]	Les méthodes privées non utilisées doivent être supprimées.	SonarQube	UnusedPrivateMethod	Mineure
[METH-4]	Les paramètres non utilisés doivent être supprimés.	SonarQube	S1172	Majeure
[METH-5.0]	Éviter de réassigner des paramètres	SonarQube	S1226	Majeure
[METH-6]	Éviter les instructions <code>return</code> dans les méthodes <code>void</code>	PMD	UnnecessaryReturn	Majeure
[METH-7]	Retourner un tableau de taille zéro plutôt que <code>null</code>	Squid	S1168	Majeure
[METH-8]	Trois <code>return</code> maximum par méthode	SonarQube	S1142	Majeure
[METH-9]	Simplifier les retours booléens	SonarQube	S1126	Mineure
[METH-10]	Utiliser des paramètres dont la signature du type est identique lors de la surcharge de méthodes	FindBugs	NM_WRONG_PACKAGE_INTENTIONAL NM_WRONG_PACKAGE	Bloquante
[METH-11]	Les méthodes destinées à la sérialisation doivent être privées	SonarQube	S2061	Critique

7.5.3. [METH-1] Les méthodes vides et non commentées sont interdites

Commenter explicitement les méthodes vides facilite la distinction entre les méthodes intentionnellement et non intentionnellement vides.

7.5.4. [METH-2] Les méthodes surchargeantes inutiles sont interdites

Les méthodes surchargeantes se contentant d'appeler la méthode définie dans la classe mère sont interdites car inutiles.

7.5.5. [METH-3] Les méthodes privées non utilisées doivent être supprimées

7.5.6. [METH-4] Les paramètres non utilisés doivent être supprimés.

7.5.7. [METH-5.0] Éviter de réassigner des paramètres

Réassigner des paramètres est **fortement déconseillé**, il s'agit d'une pratique discutable.

Par ailleurs, il est conseillé de déclarer **final** les paramètres qui ne sont pas réassignés.

7.5.8. Règles de retour (return)

7.5.8.1. [METH-6] Éviter les instructions return dans les méthodes void

7.5.8.2. [METH-7] Retourner un tableau de taille zéro plutôt que null

Pour toute méthode renvoyant un tableau, il est recommandé de retourner un tableau de taille zéro plutôt que `null` et de documenter ce cas limite dans la Javadoc. De plus, on préférera l'utilisation d'exception au renvoi de `null` dans le cas d'erreur de traitement.

7.5.8.3. [METH-8] Trois return maximum

Il est préférable de minimiser de façon systématique le nombre d'instructions `return` dans un bloc de code en les limitant à trois. Cela permet de diminuer la complexité de la méthode, en ne cassant pas le flux d'exécution à chaque instruction `return` et donc en rendant la logique du code plus facile à comprendre.

7.5.8.4. [METH-9] Simplifier les retours booléens.

Il est **recommandé** d'éviter les instructions `if...then...else` inutiles pour retourner un booléen.

Exemple :

```
public boolean isLeMemeAge(Personne unAutre)
{
    if (unAutre.age == this.age)
    {
        return true;
    }
    else
    {
        return false;
    }
    /*
    * aurait du s'écrire :
    * return (unAutre.age == this.age);
    */
}
```

7.5.9. [METH-10] Utiliser des paramètres dont la signature du type est identique lors de la surcharge de méthodes.

Une méthode ne surcharge pas correctement une méthode similaire d'une superclasse si le paramètre utilisé provient d'un mauvais *package*.

Contre-exemple :

```
import alpha.Foo;
public class A
```

```
{
    public int f(Foo x) { return 17; }
}
----
import beta.Foo;
public class B extends A
{
    public int f(Foo x) { return 42; }
}
```

7.5.10. **[METH-11]** Les méthodes destinées à la sérialisation **doivent** être privées.

Les méthodes **doivent** être privées pour que la sérialisation puisse fonctionner. En effet, lorsqu'une classe implémente l'interface **Serializable** et définit une méthode de sérialisation/désérialisation ; si cette méthode n'est pas déclarée **private**, elle sera ignorée par l'API de sérialisation/désérialisation.

7.6. Règles relatives aux instructions *if*, *while* et *switch*

7.6.1. Finalité, critères de sélection, sources et contraintes

L'ensemble des règles et recommandations introduites dans ce sous chapitre ont pour objectif d'améliorer la lisibilité et la bonne et rapide compréhension du code (suppression du code mort, simplification des instructions).

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Criticité
[PROG-IF-1]	Ne pas utiliser d'instruction <code>if</code> si celle-ci retourne toujours vrai ou toujours faux.	SonarQube	S1145	Majeure
[PROG-IF-2]	Les blocs <code>if</code> vides doivent être supprimés.	SonarQube	S00108	Critique
[PROG-IF-3]	Regrouper les instructions <code>if</code> si possible	SonarQube	S1066	Mineure
[PROG-IF-4]	Éviter les assignements au sein d'autres instructions	SonarQube	AssignmentInSubExpressionCheck	Mineure
[PROG-WHILE-1]	Les blocs <code>while</code> vides doivent être supprimés.	SonarQube	S00108	Critique
[PROG-SWITCH-1]	Les blocs <code>switch</code> vides doivent être supprimés.	SonarQube	S00108	Critique
[PROG-SWITCH-2]	Le traitement associé à un ou plusieurs <code>case</code> doit se terminer par <code>break</code> ou <code>return</code> .	SonarQube FindBugs	S128 SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH_THROW	Critique
[PROG-SWITCH-3]	Une expression <code>switch</code> doit contenir un <code>case default</code> .	SonarQube	SwitchLastCaseIsDefaultCheck	Majeure
[PROG-SWITCH-4]	Ne pas utiliser le même code dans deux <code>case</code> différents.	SonarQube	S1871	Majeure

7.6.2. [PROG-IF-1] Ne pas utiliser d'instruction `if` si celle-ci retourne toujours vrai ou toujours faux

Contre-exemple :

```
public void fermer()
{
    if (true)
    {
        ...
    }
}
```

7.6.3. [PROG-IF-2] Les blocs `if` vides doivent être supprimés

Contre-exemple :

```
void bar(int x)
{
    if (x == 0)
    {
        /* vide ! */
    }
}
```

7.6.4. [PROG-IF-3] Regrouper les instructions `if` si possible

Deux instructions `if` **peuvent** parfois être regroupées à travers l'utilisation d'un opérateur booléen.

Contre-exemple :

```
void bar()
{
    if (x)
    {
        if (y)
        {
            /* traitements */
        }
    }
}
```

Exemple correct :

```
void bar()
{
    if (x && y)
    {
        /* traitements */
    }
}
```

7.6.5. [PROG-IF-4] Éviter les assignements au sein d'autres instructions

Éviter les assignements au sein d'autres instructions, cela rend le code plus difficile à lire et plus compliqué.

Initialement dédiée aux instructions `if`, cette règle est généralisée depuis la version 2.1.0 du présent document à toutes les instructions.

Contre-exemple au sein d'une instruction `if` :

```
public class ProgIfQuatre
{
    public void bar()
    {
        int x = 2;
        if ((x = getX()) == 3)
        {
            System.out.println("3 !");
        }
    }

    private int getX()
    {
        return 3;
    }
}
```

7.6.6. [PROG-WHILE-1] Les blocs `while` vides doivent être supprimés

Les blocs `while` vides **doivent** être supprimés. S'il s'agit d'une boucle d'attente, il faudra considérer l'utilisation de `Thread.sleep()`.

Contre-exemple :

```
void bar(int a, int b)
{
    while (a == b)
    {
        /* vide ! */
    }
}
```

7.6.7. [PROG-SWITCH-1] Les blocs `switch` vides doivent être supprimés

Contre-exemple :

```
public void bar()
{
    int x = 2;
    switch (x)
    {
        /* il y a peut-être déjà eu du code ici... mais il a été commenté... */
    }
}
```

7.6.8. [PROG-SWITCH-2] Le traitement associé à un ou plusieurs `case` doit se terminer par `break` ou `return`

En omettant de terminer un `case` d'une expression `switch` par `break` ou `return`, il y a un risque de tomber dans le `case` suivant.

Remarque : plusieurs `case` peuvent se succéder sans traitement, il n'est pas nécessaire dans ce cas de terminer chaque `case` par `break` ou `return`, comme indiqué dans l'exemple ci-dessous.

7.6.9. [PROG-SWITCH-3] Une expression `switch` doit contenir un `case default`

La section `default` traite toutes les valeurs non traitées par les `case`. Il est donc nécessaire de toujours utiliser cette section pour traiter les cas non explicitement abordés par les autres sections.

Cas particulier : si l'argument du `switch` est une `enum` et toutes les valeurs de l'`enum` ont un `case` correspondant, alors le `case default` est inutile.

7.6.10. [PROG-SWITCH-4] Ne pas utiliser le même code dans deux `case` différents

L'utilisation d'un même code pour deux `case` au sein d'un `switch` peut-être dû à une erreur de codage. Pour éviter ce risque, il est préconisé de regrouper les `case` ayant le même traitement comme indiqué dans l'exemple qui suit.

Exemple :


```

class SwitchDemo
{
    public static void main(String[] args)
    {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month)
        {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0))
                {
                    numDays = 29;
                }
                else
                {
                    numDays = 28;
                }
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}

```

7.7. Règles d'utilisation de null

7.7.1. Finalité, critères de sélection, sources et contraintes

L'ensemble des règles et recommandations introduites dans ce sous chapitre ont pour objectif d'améliorer la lisibilité et la bonne et rapide compréhension du code (suppression du code mort, simplification des instructions).

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[PROG-NULL-1]	La nullité doit être testée avant l'appel à la méthode equals()	SonarQube	S2259	Majeure
[PROG-	equals(null) interdit	SonarQube	S2159	Critique

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
NULL-2]				
[PROG-NULL-3]	Ne pas tester la nullité avant un appel à instanceof	PMD	SimplifyConditional	Critique
[PROG-NULL-4]	Règle supprimée depuis la version 2.1.0			

7.7.2. [PROG-NULL-1] La nullité doit être testée avant l'appel à la méthode equals()

Si la variable testée avec la méthode `equals()` est null, une `NullPointerException` est générée. Pour cette raison, la nullité de la variable **doit** être testée avant l'appel à la méthode `equals()`.

Contre-exemple :

```
public void bar()
{
    /* tester la nullité après ne sert à rien ! */
    if (a.equals("hi") && a != null)
    {
        /* traitements */
    }
}
```

7.7.3. [PROG-NULL-2] equals(null) interdit

Utiliser la méthode `equals()` pour comparer quelque chose à null n'a pas de sens, car en cas de réussite du test une `NullPointerException` sera générée. Cette pratique est donc **interdite**.

Contre-exemple :

```
void foo()
{
    String x = "foo";
    if (x.equals(null))
    {
        /* non-sens */
        faisQqc();
    }
}
```

7.7.4. [PROG-NULL-3] Ne pas tester la nullité avant un appel instanceof

Tester la nullité avant un appel à `instanceof` est inutile, car `instanceof` retourne `false` si l'argument est null.

Contre-exemple :

```
void bar(Object x)
{
    if (x != null && x instanceof Bar)
    {
```

```

        /* supprimer la vérification "x != null", elle ne sert à rien */
    }
}

```

7.8. Règles d'implémentation de clone()

7.8.1. Finalité, critères de sélection, sources et contrainte

Les règles suivantes ont pour objectif d'empêcher des implémentations litigieuses de la méthode clone().

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outillage	Criticité
[PROG-CLONE-1]	Si implémenté, Object.clone() doit invoquer super.clone()	SonarQube	S1182	Bloquante
[PROG-CLONE-2]	clone() doit déclarer CloneNotSupportedException	SonarQube	S1182	Bloquante
[PROG-CLONE-3]	Toute classe redéfinissant clone() doit implémenter l'interface Cloneable	SonarQube	S1182 S2157	Bloquante

7.8.2. [PROG-CLONE-1] Si implémenté, Object.clone() doit invoquer super.clone()

Comme l'accessibilité de clone est **protected**, l'invocation ne peut figurer qu'à l'intérieur de la classe de l'objet à dupliquer ; sinon, le message d'erreur « clone() has protected access in java.lang.Object » est émis par le compilateur. Afin de pouvoir dupliquer un objet utilisé dans une classe quelconque, il faut redéfinir la méthode clone dans sa classe, en **invoquant** super.clone() et en la rendant **public**.

Exemple :

```

class ProgClone implements Cloneable
{
    public Object clone() throws CloneNotSupportedException
    {
        return (ProgClone) super.clone();
    }
}

```

7.8.3. [PROG-CLONE-2] clone() doit déclarer CloneNotSupportedException

Pour dupliquer un objet, la méthode clone commence par tester si la classe de l'objet réalise l'interface Cloneable ; si ce n'est pas le cas, elle déclenche l'exception CloneNotSupportedException.

Pour cela, la méthode clone() **doit** contenir throws CloneNotSupportedException dans sa déclaration.

7.8.4. [PROG-CLONE-3] Toute classe redéfinissant clone() doit implémenter l'interface Cloneable

Pour illustration, se référer à l'exemple de [PROG-CLONE-1] .

A l'inverse, une classe qui ne redéfinit pas clone() ne doit pas implémenter l'interface Cloneable .

7.9. Règles d'implémentation de equals()

7.9.1. Finalité, critères de sélection, sources et contraintes

Les règles suivantes ont pour objectif d'empêcher des implémentations litigieuses de la méthode equals().

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[PROG-EQUAL-COMP]	Comparer les objets avec equals()	SonarSquid	S1698	Critique
[PROG-EQUAL-SUR]	Surcharger equals() et hashCode(), ou aucune des deux	SonarQube	S1206	Bloquante
[PROG-EQUAL-DEF]	Règles sur la redéfinition de equals()	SonarQube FindBugs	S1201 S1210 S2162 EQ_ALWAYS_FALSE S1872	Bloquante
[PROG-EQUAL-UTIL]	Règles sur l'utilisation de equals()	SonarQube	S2159 S1698	Critique

7.9.2. [PROG-EQUAL-COMP] Comparer les objets avec equals()

Utiliser la méthode equals() pour comparer deux objets, et non l'opérateur ==

Contre-exemple :

```
public class ProgAutrDeux
{
    boolean bar(String a, String b)
    {
        return a == b;
    }
}
```

7.9.3. [PROG-EQUAL-SUR] Surcharger equals() et hashCode(), ou aucune des deux

Surcharger les deux méthodes public boolean Object.equals(Object other) et public int Object.hashCode(), ou aucune des deux.

En effet, les deux méthodes sont fortement liées, sachant que hashCode() calcule le code de hachage d'un quelconque objet Java et equals() vérifie l'égalité entre les codes de hachage de deux objets. Si equals() est redéfinie pour comparer des objets personnalisés, alors hashCode() doit l'être également.

Il faudra veiller lors de la surcharge de ces méthodes à respecter les points suivants :

- lorsqu'une classe définit equals(), s'assurer qu'elle n'utilise pas Object.hashCode() ;

- lorsqu'une classe définit `hashCode()`, s'assurer qu'elle n'utilise pas `Object.equals()` ;
- lorsqu'une classe hérite de `equals()`, s'assurer qu'elle n'utilise pas `Object.hashCode()`.

7.9.4. [PROG-EQUAL-DEF] Règles sur la redéfinition de `equals()`

Rappel sur la méthode `equals` :

- elle est **réflexive** : pour toute référence de valeur non nulle `x`, `x.equals(x)` retourne vrai ;
- elle est **symétrique** : pour toute référence valeur non nulle `x` et `y`, `x.equals(y)` retourne vrai si et seulement si `y.equals(x)` retourne vrai ;
- elle est **transitive** : pour toute référence valeur non nulle `x`, `y` et `z`, si `x.equals(y)` retourne vrai et `y.equals(z)` retourne vrai alors `x.equals(z)` retourne vrai ;
- elle est **cohérente** : pour toute référence valeur non nulle `x` et `y`, plusieurs invocations de `x.equals(y)` retournent la même valeur si aucune modification de `x` et `y` n'ont eu lieu ;
- pour toute référence de valeur non nulle `x`, `x.equals(null)` retourne toujours faux.

Exemple :

```
public class ProgEqualDef
{
    private int num;
    private String data;

    public boolean equals(Object obj)
    {
        if(this == obj)
            return true;
        if((obj == null) || (obj.getClass() != this.getClass()))
            return false;

        ProgEqualDef ProgEqualDef = (ProgEqualDef) obj;
        return num == ProgEqualDef.num &&
            (data == ProgEqualDef.data || (data != null &&
data.equals(ProgEqualDef.data)));
    }

    public int hashCode()
    {
        int hash = 7;
        hash = 31 * hash + num;
        hash = 31 * hash + (null == data ? 0 : data.hashCode());
        return hash;
    }
}
```

7.9.5. [PROG-EQUAL-UTIL] Règles sur l'utilisation de `equals()`

Lors de l'appel à la méthode `equals()`, veiller à bien comparer les mêmes types.

Il faut donc s'assurer de ne pas comparer :

- une classe et une interface sans relation ;

- deux interfaces de types différents ;
- deux classes de types différents.

En effet, le contrat établi par la méthode `equals` définie par `java.lang.Object.equals(Object)` stipule que si on compare des objets de différentes classes, la comparaison retournera toujours faux au *runtime*.

7.10. APIs interdites

7.10.1. Finalité, critères de sélection, sources et contraintes

Les règles suivantes interdisent certaines API, dont l'utilisation enfreindrait les principes de base d'une architecture multi-couches et multi-plateformes.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[PROG-INTERDIT-1]	Le remplacement des méthodes dépréciées (<i>deprecated</i>) doit être étudié.	SonarQube	CallToDeprecatedMethod	Majeure
[PROG-INTERDIT-2]	Interdiction d'utiliser <code>getDeclaredConstructors()</code> , <code>getDeclaredConstructor(Class[])</code> , <code>setAccessible()</code> et <code>PrivilegedAction</code> .	PMD	AvoidAccessibilityAssertion	Bloquante
[PROG-INTERDIT-3]	L'utilisation de code natif est par défaut interdit.	PMD	AvoidUsingNativeCode	Critique
[PROG-INTERDIT-4]	L'utilisation de <code>System.exit()</code> , <code>System.in.*</code> , <code>System.out.*</code> , <code>System.err.*</code> , <code>System.getenv()</code> , <code>System.runFinalizerOnExit()</code> ou <code>Runtime.runFinalizerOnExit()</code> est interdite.	SonarQube	S106 S1147 S1148 S2151	Critique
[PROG-INTERDIT-5]	Il est déconseillé d'utiliser la méthode <code>File.deleteOnExit()</code>	SonarQube	CallToFileDeleteOnExitMethod	Majeure

7.10.2. **[PROG-INTERDIT-1]** Le remplacement des méthodes dépréciées (*deprecated*) doit être étudié

Afin d'anticiper et de diminuer les coûts de migrations, l'utilisation de méthodes dépréciées est déconseillé.

7.10.3. **[PROG-INTERDIT-2]** Interdiction d'utiliser `getDeclaredConstructors()`, `getDeclaredConstructor(Class[])`, `setAccessible()` et `PrivilegedAction`

Les méthodes `getDeclaredConstructors()`, `getDeclaredConstructor(Class[])` et `setAccessible()`, ainsi que l'interface `PrivilegedAction`, permettent de modifier à chaud le niveau de visibilité de variables et méthodes, même privées. Pour cette raison, elles sont interdites.

7.10.4. [PROG-INTERDIT-3] L'utilisation de code natif est par défaut interdit

Le langage Java et son API standard sont suffisamment riches pour écrire des applicatifs complets. Il ne **devrait** donc pas être nécessaire d'appeler du code non-Java, et ce même à travers la *Java Native Interface* (JNI). Dans des cas très particuliers, cette règle pourra faire l'objet d'une demande de dérogation à travers une note d'architecture et celle-ci sera commentée dans le code source.

7.10.5. [PROG-INTERDIT-4] Restrictions dans l'utilisation de l'API System

- L'utilisation des API `System.in.*`, `System.out.*`, `System.err.*`, `System.getenv()` est interdite.
- L'utilisation de `System.exit` est tolérée pour les applications *standalone* mais est interdite pour les applicatifs Web et J2EE (gérés par un conteneur). En effet, la responsabilité d'arrêter une machine virtuelle Java revient au serveur Web ou au serveur d'applications, et **non** aux applicatifs hébergés par ces derniers.
- A la place de `System.out.*`, `System.err.*` et `PrintStackTrace()`, il faudra utiliser un *logger* (dans la matrice technologique ^[2b], il s'agira de Log4J).
- `System.getenv()` a été déprécié des versions 1.1 à 1.4 de Java. Malgré son retour en Java 5, son utilisation reste **interdite**, car entièrement dépendante du système (et donc contradictoire avec le principe d'indépendance de la plate-forme).
- Les appels aux méthodes `System.gc()`, `Runtime.getRuntime().gc()`, `System.runFinalization()`, `System.runFinalizerOnExit()` ou `Runtime.runFinalizerOnExit()` sont interdits (cf. [MEM-4]).

7.10.6. [PROG-INTERDIT-5] Il est déconseillé d'utiliser la méthode `File.deleteOnExit()`

La méthode `File.deleteOnExit()` supprime un fichier ou un répertoire lorsque la JVM se termine normalement. Cette méthode n'est pas recommandée car dans les cas de sortie anormale de la JVM (kill, crash), les fichiers ne sont pas supprimés et peuvent donc s'accumuler de façon dangereuse (saturation de l'espace disque ou sensibilité des données) sur le système.

7.11. Autres règles de programmation

7.11.1. Finalité, critères de sélection, sources et contraintes

Les règles de programmation suivantes ont été regroupées ensemble car n'appartiennent pour l'instant à aucune catégorie particulière.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outils	Criticité
[PROG-AUTRE-1]	Déplacé dans [PROG-EQUAL-COMP] depuis la 2.0.0			
[PROG-AUTRE-2]	Déplacé dans [PROG-EQUAL-SUP]			

Réf.	Descriptif	Contrôle automatisé		
		Outils	Règle	Criticité
	depuis la 2.0.0			
[PROG-AUTR-3]	Les classes serialisables doivent fournir un <code>serialVersionUID</code>	SonarQube	S2057	Critique
[PROG-AUTR-4]	Les instructions tautologiques sont interdites	SonarQube	S1656	Critique
[PROG-AUTR-5]	Locale obligatoire pour l'instanciation de <code>SimpleDateFormat</code> et <code>String.toLowerCase()/toUpperCase()</code>	PMD	SimpleDateFormatNeedsLocale	Bloquante
[PROG-AUTR-6]	Utiliser un <code>String</code> pour créer une instance de <code>BigDecimal</code> à partir d'un décimal	SonarQube	S2111	Critique
[PROG-AUTR-7]	Opérations inutiles sur les objets immuables	PMD	UselessOperationOnImmutable	Critique
[PROG-AUTR-8]	Dans les stockages et les échanges sérialisés de dates et d'objets calendaires, un et un seul <code>TimeZone</code> doit être utilisé (GMT est conseillé).			Bloquante
[PROG-AUTR-STATIC-DATE]	Dans un environnement <i>multithreads</i> , il est interdit d'instancier le type <code>Date</code> en statique.	SonarQube		Critique

7.11.1. [PROG-AUTR-3] Les classes serialisables **doivent** fournir un `serialVersionUID`

Chaque classe implémentant `Serializable` est identifiée par un *serial Version Unique Identifier*. S'il n'est pas explicitement spécifié par la déclaration d'un champ `private static final long` nommé `serialVersionUID`, le système en générera automatiquement un de façon déterministe, à partir du nom de la classe, du nom des interfaces qu'elle implémente, et de ses membres publics et protégés. Si ces valeurs changent par la suite, par exemple en rajoutant une méthode, le `serialVersionUID` automatiquement généré change, et la compatibilité avec les autres instances n'est plus assurée.

Ce `serialVersionUID`, peut être considéré comme un numéro de série, obligatoire garanti d'intégrité.

7.11.2. [PROG-AUTR-4] Les instructions tautologiques sont interdites

Éviter les instructions tautologiques, elles sont inutiles!

Contre-exemple :	Contre-exemple :
<pre>public class ProgAutrQuatre { public void bar() { int x = 2; } }</pre>	<pre>public class ProgAutrQuatre { public void foo() { int x,y; } }</pre>

<pre> x = x; // tautologique! } }</pre>	<pre> x = x = 17; // tautologique! } }</pre>
--	--

7.11.3. **[PROG-AUTR-5]** L'objet locale doit être fourni pour l'instanciation de `SimpleDateFormat` et `String.toLowerCase()/toUpperCase()`

S'assurer lors de la création d'une instance de `SimpleDateFormat` ou de `String.toLowerCase()/toUpperCase()` que la locale est spécifiée et fournie.

7.11.4. **[PROG-AUTR-6]** Utiliser un `String` pour créer une instance de `BigDecimal` à partir d'un décimal

On pourrait penser que `new BigDecimal(.1)` est exactement égal à « 0,1 », alors qu'il retourne en réalité « .100000000000000005511151231257827021181583404541015625 ». Ceci est dû au fait que « 0,1 » ne peut être représenté comme un `double` (c'est-à-dire, une fraction binaire d'une longueur finie). Le constructeur prenant un `String` est lui par contre complètement prévisible : `new BigDecimal("1")` est exactement égal à « 0,1 ». Il est donc demandé d'**utiliser** un `String` pour créer une instance décimale de `BigDecimal`.

Contre-exemple :	Exemple correct :
<pre> import java.math.BigDecimal; public class ProgAutrSix { public static void main(String[] args) { BigDecimal bd = new BigDecimal(1.123); } }</pre>	<pre> import java.math.BigDecimal; public class ProgAutrSix { public static void main(String[] args) { BigDecimal bd = new BigDecimal("1.123"); BigDecimal bd = new BigDecimal(12); } }</pre>

7.11.5. **[PROG-AUTR-7]** Opérations inutiles sur les objets immuables

Une opération sur un objet immuable (`BigDecimal` ou `BigInteger`) ne change pas l'objet lui-même, le résultat de l'opération est un nouvel objet. En conséquence, ignorer le résultat de l'opération est considéré comme une **erreur**.

Contre-exemple :	Exemple correct :
------------------	-------------------

<pre>import java.math.*; class ProgAutrSept { void methode1() { BigDecimal bd = new BigDecimal(10); bd.add(new BigDecimal(5)); } }</pre>	<pre>import java.math.*; class ProgAutrSept { void methode1() { BigDecimal bd = new BigDecimal(10); bd = bd.add(new BigDecimal(5)); } }</pre>
---	--

7.11.6. [PROG-AUTR-8] Règles d'utilisation des objets TimeZone

Dans les stockages et les échanges sérialisés de dates et d'objets calendaires, un et un seul **TimeZone** doit être utilisé (GMT est obligatoire).

7.11.7. [PROG-AUTR-STATIC-DATE] L'instanciation statique du type DateFormat en interdite.

Les objets DateFormat n'étant pas synchronisés, il est **interdit** de créer ou d'appeler des instances statiques de type DateFormat dans un environnement *multithreads*.

8. Règles métriques

8.1. Finalité, critères de sélection, sources et contrainte

Les métriques appliquées au code Java fournissent des indicateurs de qualité des développements.

S'agissant d'indicateurs, la plupart des règles ne seront que recommandées, voir fortement recommandées. Il est cependant rappelé qu'en-dessous d'un certain seuil global de qualité, un livrable peut être déclaré non conforme ou avec réserves.

Les métriques ont été sélectionnées quand elles permettaient de mesurer et vérifier la lisibilité et clarté du code source, ainsi que sa maintenabilité, évolutivité, modularité et réutilisabilité.

Bien que couramment utilisées, les métriques suivantes n'ont pas été retenues pour la charte :

- Nombre de classes et d'interfaces : caractère informatif, variable suivant la taille du projet.
- Nombre d'enfants (héritages directs) : caractère informatif, variable selon l'architecture.
- Nombre de méthodes surchargées : caractère informatif, variable selon l'architecture.
- Index de spécialisation ((Nombre de méthodes surchargées)* (degré d'héritage) / (nombre de méthodes)) : caractère informatif, variable selon l'architecture.
- Poids des méthodes par classe (somme des complexités cyclomatiques des méthodes d'une classe) : caractère informatif.
- Manque de cohésion des méthodes : métrique d'intérêt discutable en Java du fait des *getters* et *setters*.
- Les Couplages : varient selon la nature de la classe (utilitaire ou métier), et leur rôle dans l'architecture. On distingue deux types de couplage :
 - Le couplage par dépendance ascendante (Ca ou Afférent Coupling) : nombre de paquetages tiers utilisant un paquetage donné. Indicateur de la responsabilité d'un paquetage, il peut mettre en évidence qu'un paquetage est au centre de l'applicatif ou mettre en relief une mauvaise gestion des paquetages.
 - Le couplage par dépendance descendante (Ce ou Efferent Coupling) : nombre de paquetages tiers utilisés par un paquetage donné. C'est un indicateur d'indépendance du code.
- Instabilité ($I = Ce / (Ca + Ce)$) ^[20] : varie selon la nature de la classe (utilitaire ou métier).
- Abstraction ($A = \text{nombre de classes et interfaces abstraites} / \text{nombre total de types dans un paquetage}$) ^[20] : caractère informatif.
- Distance normalisée ($Dn = A + I - 1$) ^[20] : caractère informatif.

Dans la famille des indicateurs de qualité listés ci-dessus, seul un a été retenu dans cette charte et utilisé pour une règle :

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-CYCLIC-DEP]	Toute dépendance cyclique entre paquetages est interdite.	SonarQube		Bloquante

8.2. Longueur des noms

Dans un but de lisibilité et de rapide compréhension, il faut éviter de donner aux variables et méthodes des noms trop courts ou trop longs.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-LONG-NOM-1]	Un nom de variable (champ, locale, paramètre) doit être constitué d'au moins trois caractères, à l'exception des index dans les boucles (qui seront nommés i, j, k ...).	PMD	ShortVariable	Majeure
[METR-LONG-NOM-2]	Un nom de variable (champ, locale, paramètre) doit être constitué d'au plus 35 caractères.	PMD	LongVariable	Majeure
[METR-LONG-NOM-3]	Un nom de méthode doit être constitué d'au moins trois caractères.	PMD	ShortMethodName	Majeure
[METR-LONG-NOM-4]	Un nom de classe doit être constitué d'au moins cinq caractères.	PMD	ShortClassName	Mineure

Contre-exemples :

```
public class MetrLongNom
{
    private int q = 15;    // VIOLATION - Nom de variable "q" (champs) trop court

    int reellementTropTropTropTropTropTropLong = -3;    // VIOLATION - Nom de
variable (champs) trop long

    /* VIOLATION - Noms de variables (paramètres) trop court et trop long */
    public static void main(String[] co, String[] beaucoupTropTropLong)
    {
        int r = 20 + q;    // VIOLATION - Nom de variable "r" (locale) trop court
        int otherReallyLongLongLongLongLongName = -5;    // VIOLATION - Nom de
variable (locale) trop longue
        for (int i = 0; i < 10; i++)    // Exception : pas de violation (dans
boucle FOR)
        {
            r += q;
        }

        for (int indexBeaucoupTropTropTropTropTropLong = 0;    // VIOLATION - Nom
```

```

de variable trop long
    indexBeaucoupTropTropTropTropTropLong < 10;
    indexBeaucoupTropTropTropTropTropLong ++ ) {}
}

public void a(int i) {}    // VIOLATION - Nom de méthode "a" trop court
}

```

8.3. Longueur et complexité des classes et interfaces

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-CLASS-1]	400 lignes de code effectif est un maximum pour une classe Java.	PMD	ExcessiveClassLength =1000 (commentaires inclus)	Critique
[METR-CLASS-2]	Une ligne ne doit pas dépasser 140 caractères.	SonarQube	S00103	Mineure
[METR-CLASS-3]	Le nombre d'imports doit être inférieur à 30 par classe.	SonarQube	S1200	Critique
[METR-CLASS-4]	Le degré d'héritage d'une classe ou interface doit être limité à 5.	SonarQube	MaximumInheritanceDepth	Majeure
[METR-CLASS-5]	Une classe ne doit pas avoir plus de 15 champs.	PMD	TooManyFields	Critique
[METR-CLASS-6]	Une classe ne doit pas avoir plus de 25 méthodes.	SonarQube	S1448	Majeure

8.3.1. [METR-CLASS-1] 400 lignes de code effectif **doit** être un maximum pour une classe Java.

Est considéré comme code effectif tout code visible sur l'arbre syntaxique abstrait (ou en anglais *Abstract Syntax Tree*, AST) généré par le compilateur `javacc`. Pour une définition de l'AST, se référer à http://fr.wikipedia.org/wiki/Arbre_syntaxique_abstrait. Le code effectif correspond donc à l'ensemble des lignes du fichier contenant le code source, moins les lignes de commentaires et les lignes blanches.

Une violation de cette règle indique généralement que la classe « en fait trop ». Essayer de **réduire** la taille de la classe en la décomposant en plusieurs, ou en supprimant les éventuels « copier-coller ».

Si la détection du code effectif n'est pas possible, la valeur maximum est repoussée à 1000 ligne de codes.

8.3.2. [METR-CLASS-2] Une ligne ne **doit** pas dépasser 140 caractères.

La longueur des lignes doit permettre une lecture aisée des fichiers dans l'environnement de développement standard, sur un écran de résolution standard.

Il est pour cela préférable d'**éviter** les lignes trop longues (plus de 140 caractères) qui rendraient difficile la relecture du code.

8.3.3. [METR-CLASS-3] Le nombre d'imports **doit** être inférieur à 30 par classe.

Un grand nombre d'imports peut indiquer un fort degré de couplage pour certains objets.

8.3.4. [METR-CLASS-4] Le degré d'héritage d'une classe ou interface doit être limité à 5.

Un héritage trop profond révèle une complexité de construction qui rendra difficile la prédiction et compréhension des comportements possibles.

8.3.5. [METR-CLASS-5] Une classe ne **doit** pas avoir plus de 15 champs.

Les classes ayant trop de champs **devraient** être sujettes à *refactoring*, en regroupant si possible certains champs en objets.

8.3.6. [METR-CLASS-6] Une classe ne **doit** pas avoir plus de 25 méthodes.

Un nombre important de méthodes peut indiquer que la classe **nécessite** d'être découpée en plusieurs, afin notamment de diminuer la difficulté de réalisation des tests.

Exception : les classes de données ne sont pas soumises à la règle.

8.4. Longueur et complexité des méthodes

8.4.1. Finalité, critères de sélection, sources et contraintes

Comme introduit dans la règle [NOM-17] il est préférable que chaque méthode ne fasse qu'une seule chose, le nom de chacune reflétant cela de manière précise. De même, il faudra éviter d'écrire des méthodes longues et compliquées : le traitement réalisé par une méthode doit être simple et fonctionnel, si possible réutilisable, modulaire.

Le nombre de lignes que l'on peut considérer comme raisonnable pour une méthode dépend de sa complexité. Cependant, on veillera à ce que chaque bloc, chaque méthode soit construit de manière à posséder la plus forte cohésion et la plus grande indépendance possible par rapport à son environnement.

Ces démarches s'inscrivent dans une volonté d'éviter la redondance de code, de faciliter la maintenance.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-METH-1]	Une méthode ne devrait pas dépasser 50 lignes de code effectif.	SonarQube	S138	Majeure
[METR-METH-2]	Une méthode ne devrait pas avoir plus de 7 paramètres	SonarQube	S00107	Majeure
[METR-METH-3]	La complexité cyclomatique d'une méthode devrait être inférieure à 12.	SonarQube	MethodCyclomaticComplexity	Critique
[METR-METH-4]	La densité des instructions switch doit être inférieure à 10	SonarQube	S1151	Critique

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-METH-5]	3 instructions if, for, while et try imbriquées sont un maximum	SonarQube	S134	Majeure

8.4.2. [METR-METH-1] Une méthode ne doit pas dépasser 50 lignes de code effectif.

Pour rappel, est considéré comme code effectif tout code visible sur l'arbre syntaxique abstrait (ou en anglais *Abstract Syntax Tree*, AST) généré par le compilateur `javacc`. Pour une définition de l'AST, se référer à http://fr.wikipedia.org/wiki/Arbre_syntaxique_abstrait. Le code effectif correspond donc à l'ensemble des lignes du fichier contenant le code source, moins les lignes de commentaires et les lignes blanches.

Une violation de cette règle indique généralement que la méthode « en fait trop ». Essayer de **réduire** la taille de la méthode en la décomposant en plusieurs, ou en supprimant les éventuels « copier-coller ».

8.4.3. [METR-METH-2] Une méthode ne doit pas avoir plus de 7 paramètres

Une longue liste de paramètres peut indiquer la nécessité de créer un nouvel objet regroupant ces paramètres.

8.4.4. [METR-METH-3] La complexité cyclomatique d'une méthode **doit** être inférieure à 12.

La complexité cyclomatique (de McCabe) est déterminée par le nombre de points de décision dans une méthode. Un point de décision correspond à l'occurrence d'un opérateur `if`, `while`, `do`, `for`, `case`, `catch`, `?:`, `&&`, `||`. Généralement, une méthode à complexité cyclomatique de 1 à 4 est considérée comme peu complexe, de 5 à 7 moyennement complexe, de 8 à 10 complexe, et de plus de 10 très complexe.

Ainsi, une méthode à complexité cyclomatique supérieure à 11 **devrait** être décomposée en plusieurs méthodes, afin de faciliter la compréhension de l'algorithme.

Contre-exemple :

```
public void complexiteCyclomatique()
{
    /* Complexité cyclomatique = 12 */
    if (a == b)    // 2
    {
        if (a1 == b1)    // 3
        {
            traiteLeCas();
        }
        else
        {
            if (a2 == b2)    // 4
            {
                traiteLeCas();
            }
            else
            {
                traiteLeCas();
            }
        }
    }
    else
    {
        if (c == d)    // 5
        {
```

```
        while (c == d)    // 6
        {
            traiteLeCas();
        }
    }
    else
    {
        if (e == f)    // 7
        {
            for (int n = 0; n < h; n++)    // 8
            {
                traiteLeCas();
            }
        }
        else
        {
            switch (z)
            {
                case 1:    // 9
                    traiteLeCas();
                    break;
                case 2:    // 10
                    traiteLeCas();
                    break;
                case 3:    // 11
                    traiteLeCas();
                    break;
                default:    // 12
                    traiteLeCas();
                    break;
            }
        }
    }
}
```

8.4.5. [METR-METH-4] La densité des intructions **switch** **doit** être inférieure à 10

Un nombre moyen de lignes de code par case supérieur révèle que l'instruction **switch** « en fait trop ».

Il **faudra** alors considérer la création de nouvelles méthodes, qui pourront être appelées depuis les case pour en diminuer le nombre de lignes.

Contre-exemple :

```
public class MetrMethQuatre
{
    public void bar(int x)
    {
        switch (x)
        {
            case 1:
            {
                /* beaucoup de traitements */
                break;
            }
            case 2:
            {
                /* beaucoup de traitements */
                break;
            }
            default:
            {
                /* traitements */
            }
        }
    }
}
```



```

    }
}

```

8.4.6. **[METR-METH-5]** 3 instructions if, for, while et try imbriquées **sont** un maximum

Cette règle a pour objectif d'améliorer la lisibilité et la bonne compréhension du code source.

Contre-exemple :

```

public void bar(int w, int x, int y, int z)
{
    if (w < x)
    {
        if (x < y)
        {
            if (y < z)
            {
                if (x == y)
                {
                    /* whew, trop profond */
                }
            }
        }
    }
}

```

8.5. Code dupliqué

8.5.1. Finalité, critères de sélection, sources et contraintes

Les règles relatives au code dupliqué, ou « copié/collé », ont pour but d'améliorer la modularité du code source.

Le respect de ces règles est un bon indicateur de la qualité du code source.

8.5.2. **[METR-DUPL-1]** 25 lignes de code dupliqué devrait être un maximum

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[METR-DUPL-1]	25 lignes de code dupliqué est un maximum.	SonarQube	DuplicatedBlocks	Majeure

On veillera en plus à éviter les « copiés-collés » servant à la libération de ressources. On préférera l'utilisation d'un *framework* ou de classes/méthodes utilitaires.

9. Règles de gestion des exceptions

9.1. Généralités, finalité, critères de sélection, sources et contrainte

Ce chapitre regroupe toutes les règles relatives à l'implémentation et à l'utilisation des exceptions. Elles complètent les règles précédemment introduites sur leur formatage (cf. [FORM-14]), leur nommage (cf. [NOM-7]), et leur documentation (cf. [JDOC-7.1] et [JDOC-7.2]).

Ce chapitre débute sur un ensemble de règles qui tendent à éviter la perte ou la corruption des exceptions ou des informations sur les erreurs. On veillera en plus à reporter toutes les variables (le contexte) qui sont à l'origine de l'exception et aideront au diagnostic de l'erreur quand elle survient.

Suivent des règles sur le bon usage des exceptions. Une bonne pratique qui ne pouvait pas être transposée en règle peut être résumée en ces quelques mots : « n'utiliser une exception que dans des situations exceptionnelles ».

Contre-exemple :	Exemple correct :
<pre>/* Abus terrible des exceptions! */ try { int i = 0; while(true) a[i++].f(); } catch(ArrayIndexOutOfBoundsException e) { /* traitements */ }</pre>	<pre>for (int i = 0; i < a.length; i++) { a[i].f(); }</pre>

On veillera également à ne pas abuser des exceptions vérifiées. En effet, une API ne devrait pas forcer ses clients à utiliser systématiquement les exceptions lors d'un contrôle ordinaire :

Contre-exemple :	Exemple correct :
<pre>/* Invocation avec une exception vérifiée */ try { obj.action(args); } catch(LexceptionVerifiee e) { /* Traitement de l'exception */ ... }</pre>	<pre>/* Invocation avec test de l'état et exception non vérifiée */ if (obj.actionPermitted(args)) { obj.action(args); } else { /* Traitement de la condition exceptionnnelle */ ... }</pre>

Enfin, on s'efforcera d'utiliser des conditions vérifiées (*checked*) pour les erreurs récupérables et des exceptions non vérifiées (*unchecked* ou *run time*) pour les erreurs de programmation, d'utilisation inadéquate d'API.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[EXCEP-3.1]	Les blocs <code>catch</code> vides sont interdits	SonarQube	S00108	Critique
[EXCEP-3.2]	Les blocs <code>try</code> vides sont interdits.	SonarQube	S00108	Critique
[EXCEP-3.3]	Les blocs <code>finally</code> vides sont interdits.	SonarQube	S00108	Critique
[EXCEP-4]	Ne pas utiliser <code>InstanceOf</code> dans un bloc <code>catch</code> .	SonarQube	S1193	Bloquante
[EXCEP-5]	Préserver la <i>stacktrace</i>	SonarQube	S1166	Majeure
[EXCEP-6.1]	Interdiction de déclarer des méthodes générant <code>Exception</code> , <code>Throwable</code> ou <code>Error</code> .	SonarQube	S00112	Critique
[EXCEP-6.2]	Interdiction de générer <code>RuntimeException</code> , <code>Throwable</code> , <code>Exception</code> ou <code>Error</code>	SonarQube	S00112	Bloquante
[EXCEP-6.3]	<code>catch Throwable</code> , <code>catch Exception</code> , <code>catch Error</code> , <code>catch RuntimeException</code> et <code>IllegalMonitorStateException</code> sont interdits.	CheckStyle •SonarQube	Illegal Catch •S1181 •S2221 •S2235	Bloquante
[EXCEP-7.1]	Ne pas générer de <code>NullPointerException</code>	SonarQube	S1695	Critique
[EXCEP-7.2]	<code>catch NullPointerException</code> est interdit.	SonarQube	S1696	Critique
[EXCEP-8]	Ne pas utiliser l'instruction <code>return</code> dans un bloc <code>finally</code> .	SonarQube	S1143	Bloquante
[EXCEP-1]	Ne pas utiliser les exceptions comme des contrôleurs de flux.	SonarQube	S1141	Bloquante
[EXCEP-2]	Préférer l'utilisation des exceptions standards.			Majeure
[EXCEP-10]	Libérer les ressources dans un bloc <code>finally</code>			Bloquante
[EXCEP-11]	Les exceptions doivent être immuables.	SonarQube	S1165	Majeure
[EXCEP-12]	Les exceptions ne doivent pas hériter de <code>java.lang.Error</code>	SonarQube	S1194	Bloquante
[EXCEP-13]	Une classe qui implémente l'interface <code>Iterator</code> , doit avoir une méthode <code>next()</code> pouvant remonter une exception <code>NoSuchElementException</code> .	SonarQube	S2272	Critique

9.2. Éviter la perte ou la corruption des exceptions ou des informations sur les erreurs

9.2.1. [EXCEP-3] Les blocs vides sont interdits

9.2.1.1. [EXCEP-3.1] Les blocs `catch` vides sont interdits

Un bloc `catch` vide va à l'encontre du but des exceptions. Ignorer une exception c'est arrêter une alarme qui vient de se déclencher pour que personne n'ait une chance de voir s'il y a vraiment le feu.

Contre-exemple :

```
try
{
    /* traitements */
}

/* Le bloc catch vide ci-dessous ignore l'exception. Hautement suspect ! */
catch (UneException e) {}
```

A minima, si l'exception doit être ignorée, mais cela doit rester rare, ce bloc devrait contenir un commentaire expliquant pourquoi l'exception peut-être ignorée. La règle peut aussi s'écrire : ne jamais capturer une exception sans y apporter un traitement pertinent.

Écrire un `catch` vide est **interdit**, et cela vaut pour les exceptions vérifiées et non-vérifiées. Particulièrement les exceptions non-vérifiées car alors, pratiquer ce genre de code, revient à faire continuer un programme silencieusement après une erreur et peut entraîner la panne ultérieure, panne qui pourra alors se montrer particulièrement difficile à diagnostiquer.

C'est pour cette même raison que l'utilisation de `catch Throwable` est interdite (cf. [EXCEP-6.3]).

9.2.1.2. [EXCEP-3.2] Les blocs `try` vides sont interdits

Un bloc `try` vide ne présente aucun intérêt, puisqu'il ne pourra générer aucune exception. Il **doit** ainsi être supprimé.

9.2.1.3. [EXCEP-3.3] Les blocs `finally` vides sont interdits

S'il n'y a aucun traitement final à apporter, le bloc `finally` n'a pas lieu d'exister, et **doit** être supprimé.

9.2.2. [EXCEP-4] Ne pas utiliser `InstanceOf` dans un bloc `catch`

Chaque exception traitée **doit** posséder son propre bloc `catch`.

Contre-exemple :

```
try
{
    /* traitements */
}
catch (Exception excep)
{
    if (excep instanceof IOException)
```

Exemple correct :

```
try
{
    /* traitements */
}
catch (IOException excep)
{
    nettoyage();
```

<pre> { nettoyage(); } </pre>	<pre> } </pre>
-----------------------------------	----------------

9.2.3. [EXCEP-5] Préserver la *stacktrace*

Générer une exception depuis un bloc **catch** sans passer la première exception à la nouvelle exception entraîne la perte de la *stacktrace*, ce qui pourra poser des problèmes lors du débogage de l'applicatif.

Contre-exemple :	Exemple correct :
<pre> try { facture.calcul(client, date); } catch (ProfilClientInconnuException excep) { throw new CompteClientInconnuException (excep.getMessage()); } </pre>	<pre> try { facture.calcul(client, date); } catch (ProfilClientInconnuException excep) { throw new CompteClientInconnuException(excep.getMe ssage(), excep); } </pre>

9.2.4. [EXCEP-6] Exceptions « mères » interdites

9.2.4.1. [EXCEP-6.1] Interdiction de déclarer des méthodes générant Exception, Throwable ou Error

Il est rappelé qu'il **faudrait** indiquer dans chaque méthode la liste exhaustive des types d'exceptions pouvant remonter.

Cependant, au même titre qu'il faut importer spécifiquement les classes utilisées dans les imports et éviter les imports avec une étoile (cf. [FORM-14]), il faut indiquer précisément et uniquement les types d'exceptions pouvant être émis par une classe. Il est donc **interdit** de déclarer qu'une méthode **throws Exception**, **throws Throwable** ou **throws Error**. Il est aussi également interdit de déclarer des clauses **throws** inutilisées.

Contre-exemple :
<pre> public void methodeThrowingException() throws Exception { /* traitements */ } </pre>

9.2.4.2. [EXCEP-6.2] Interdiction de générer RuntimeException, Throwable, Exception ou Error

Plutôt que de générer ces exceptions, il **faudrait** utiliser les sous-classes de celles-ci, de façon à produire des exceptions plus précises.

Contre-exemple :
<pre> public class ExcepSixDeux { public void bar() throws Exception { throw new Exception(); } } </pre>

```
}  
}
```

9.2.4.3. [EXCEP-6.3] catch Throwable, catch Exception, catch Error, catch RuntimeException et IllegalMonitorStateException sont interdits

L'exception **Throwable** est la mère de toutes les exceptions. Capturer la catégorie **Throwable** est **interdit**, car tout y passe :

- les erreurs de format des fichiers **.class** ;
- la saturation de la mémoire de l'applicatif ;
- l'utilisation malheureuse de pointeur **null** (**NullPointerException**) ;
- les débordements dans l'utilisation des tableaux ;
- les récursivités infinies et les erreurs applicatives.

Si une fuite mémoire entraîne l'invocation de l'exception **OutOfMemoryError**, l'applicatif va capturer le problème et continuer son traitement. La cause réelle du problème sera masquée, entraînant des erreurs en cascades qu'il sera difficile d'analyser.

De même, attraper une **RuntimeException** empêche la propagation des exceptions *Runtime* vers le conteneur pour la libération des ressources.

Les exceptions **Exception**, **Error** et **RuntimeException**, qui héritent en premier ou deuxième niveau de **Throwable**, sont trop génériques pour apporter une réponse particulière à l'exception. Il est par conséquent également **interdit** de les rattraper.

IllegalMonitorStateException est généralement « jetée » seulement dans le cas d'un défaut de conception dans le code (appel de **wait** ou **notify** sur un objet sur lequel il n'a pas été placé de verrou).

Contre-exemple :

```
public class ExcepSixTrois  
{  
    public void bar()  
    {  
        try  
        {  
            /* traitements */  
        }  
        catch (Throwable th)    // Ne devrait pas catch throwable  
        {  
            /* traitement de l'exception */  
        }  
    }  
}
```

Solution : capturer uniquement et si nécessaire les exceptions vérifiées pouvant être émises. Pour cela, il faut indiquer une à une, dans chaque méthode, les exceptions vérifiées pouvant remonter. Le compilateur se chargera de signaler tous les chemins de l'applicatif où il est nécessaire de capturer ou de propager les exceptions.

Rattraper les exceptions non-vérifiées n'est pas recommandé mais n'est pas pour autant interdit : on peut concevoir ainsi une « traduction » d'exception non-vérifiée dépendante de l'implémentation en une autre, vérifiée, plus appropriée pour l'appelant. On recommande de porter une attention particulière à la documentation (cf. [JDOC-7.2]).

9.2.1. [EXCEP-7] Interdiction de NullPointerException

9.2.1.1. [EXCEP-7.1] Ne pas générer de NullPointerException

Générer une `NullPointerException` peut porter à confusion, car la plupart des développeurs vont penser que celle-ci a été générée par la machine virtuelle. Il **faudra** utiliser à la place `IllegalArgumentException`, qui sera clairement identifiée comme une exception initiée par le développeur.

Contre-exemple :

```
int getMontantFacture(client, date);
{
    if (client.getCompte == null)
    {
        throw new NullPointerException();
    }
    ...
}
```

9.2.1.2. [EXCEP-7.2] catch `NullPointerException` est à éviter

Une `NullPointerException` ne devrait jamais être générée (cf. [EXCEP-7.1]). Un bloc `catch` pourrait masquer l'erreur originale et causer d'autres erreurs plus subtiles dans son sillage.

Contre-exemple :

```
public class ExcepSeptDeux
{
    void bar()
    {
        try
        {
            /* traitements */
        }
        catch (NullPointerException npe)
        {
            /* ne jamais attraper NullPointerException ! */
        }
    }
}
```

9.2.2. [EXCEP-8] Ne pas utiliser l'instruction `return` dans un bloc `finally`

Un bloc `finally` **ne doit pas** contenir d'instruction `return`, puisqu'il est appelé en cas d'exception ou de `return` dans le bloc `try` correspondant, ce qui fait que l'instruction `return` éventuelle du bloc `try` sera ignorée.

Pour information, l'utilisation de l'instruction `return` dans un bloc `finally` donne lieu au *warning* suivant dans la plupart des IDEs : `finally clause cannot complete normally`.

9.3. Du bon usage des exceptions

9.3.1. [EXCEP-1] Ne pas utiliser les exceptions comme des contrôleurs de flux

Utiliser les exceptions comme des contrôleurs de flux revient à écrire des « GOTO ».

Contre-exemple :

```

public class ExcepUn
{
    void bar()
    {
        try
        {
            try
            {
                /* traitements */
            }
            catch (Exception excep)
            {
                throw new WrappedRuntimeException(excep);
                /* Ceci ressemble à un GOTO WrappedRuntimeException ! */
            }
            catch (WrappedRuntimeException e)
            {
                /* traitement de l'exception */
            }
        }
    }
}

```

9.3.2. [EXCEP-2] Préférer l'utilisation des exceptions standards

Réutiliser des exceptions standards a de grand bénéfices : cela facilite grandement l'apprentissage et la lisibilité de l'API (et cela amène des gains en empreinte et temps de chargement).

Voici les recommandations principales de réutilisation :

- `IllegalArgumentException` : quand la valeur d'un paramètre est inappropriée ;
- `IllegalStateException` : quand l'état de l'objet est inapproprié ;
- `IndexOutOfBoundsException` : quand l'index du paramètre est hors limite ;
- `ConcurrentModificationException` : quand une modification concurrente a été détectée ;
- `UnsupportedOperationException` : quand l'objet ne supporte pas la méthode.

9.3.3. [EXCEP-10] Libérer les ressources dans un bloc finally

Veiller à toujours **fermer** les flux et libérer les ressources dans un bloc **finally**.

Code interdit :	Code correct :
<pre> InputStream ins = null; try { ins = new FileInputStream("monIn"); ... } catch (FileNotFoundException e) { /* Traitement de l'exception */ } </pre>	<pre> InputStream ins = null; try { ins = new FileInputStream("monIn"); ... } catch (FileNotFoundException e) { /* Traitement de l'exception */ } finally { if (ins != null) { try </pre>

	<pre> { ins.close(); } catch (IOException e) { /* Traitement de l'exception */ } }</pre>
--	---

9.3.4. [EXCEP-11] Les exceptions doivent être immuables

Les exceptions **doivent** être immuables, c'est-à-dire qu'elles ne doivent définir, entre autres, que des champs **final**.

En effet, des champs autres que **final** permettraient non seulement de modifier l'état de l'exception et donc de masquer l'erreur originale, mais permettraient également d'oublier d'initialiser l'état, et ainsi d'en tirer lors du traitement de l'exception des conclusions incorrectes basées sur cet état (cf. §7.2.6).

9.3.5. [EXCEP-12] Les exceptions ne doivent pas hériter de java.lang.Error

Error **doit** être réservé aux erreurs systèmes et les exceptions de votre système ne doivent pas en hériter.

9.3.6. [EXCEP-13] Une classe qui implémente l'interface Iterator, doit avoir une méthode next() pouvant remonter une exception throw NoSuchElementException.

Exemple :

```
Public class MyIterator implements Iterator
{
    ...

    public Object next() throws NoSuchElementException
    {
        if (position >= data.length)
        {
            throw new NoSuchElementException();
        }
        return data[position++];
    }
}
```

10. Règles de programmation pour la performance

10.1. Finalité, critères de sélection, et généralités sur la performance

La finalité de ce chapitre et de ses règles est, comme son nom l'indique, la performance.

On pourra noter que la plupart des règles qui y apparaissent ont attrait à ce qu'il ne faut pas faire, ce qu'il est interdit ou déconseillé de faire pour garantir performance, stabilité et qualité du code.

En effet, il est recommandé avant toute chose d'écrire du code clair, simple, correct, et laisser la phase d'optimisation du code en dernier. Il faudra éviter les décisions de conception qui limitent le performance en particulier les interactions entre les modules et avec le monde extérieur à l'applicatif (en particulier les accès réseaux et la gestion de persistance).

Les API sont particulièrement à surveiller, il faut toujours considérer avec attention les conséquences de vos choix de conception d'API sur la performance. Par exemple, déclarer public un type mutable peut demander de nombreuses copies défensives inutiles (cf. 7.2.6 et 7.5.2). De même utiliser abusivement l'héritage plutôt que la composition peut artificiellement limiter les performances de la sous-classe (cf. 7.2.7).

L'optimisation de code demande un effort particulier qui ne se justifie pas toujours. Il y a de nombreuses raisons pour ne pas optimiser un code. Par exemple :

- s'il fonctionne, l'optimiser entraîne nécessairement de nouvelles erreurs subtiles ;
- un code optimisé est plus difficile à maintenir et à comprendre ;
- il faut consommer beaucoup de temps pour optimiser un code, pour un gain pas souvent nécessaire.

Le code doit être optimisé uniquement quand les mesures de performance montrent que cela est nécessaire. Il est donc recommandé de re-mesurer la performance systématiquement après chacun des essais d'optimisation.

10.2. Règles de gestion de la mémoire : fuites mémoires et ramasse-miettes

10.2.1. Finalité, critères de sélection, sources et contraintes

L'objectif des règles suivantes est l'amélioration de la performance à travers la diminution de l'empreinte mémoire des programmes, tout comme l'abandon de certaines instructions couramment utilisées mais en réalité contre-productives.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outil	Criticité
[MEM-1]	Mettre les références des objets à null devrait rester une exception plutôt que la règle.	PMD	NullAssignment	Majeure
[MEM-2]	Éviter la création de pools d'objets.			Critique
[MEM-3]	Motiver et documenter la création de caches d'objets.			Bloquante
[MEM-4]	Interdiction de forcer le ramasse-miettes	SonarQube	S1215	Bloquante
[MEM-5]	Interdiction d'implémenter et d'appeler finalize	SonarQube	ObjectFinalizeCheck	Bloquante

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[MEM-6]	Les fabriques et méthodes utilitaires devraient être statiques.			Majeure
[MEM-7]	Utilisation de ThreadLocal fortement déconseillée	FindBugs	PMB_INSTANCE_BASED_THREAD_LOCAL	Critique

10.2.2. [MEM-1] Dé-référencement des objets.

Les fuites mémoires des langages à ramasse-miettes sont insidieuses. Si un objet est mal-intentionnellement retenu, non seulement cet objet est exclu du ramassage de miettes, mais tous les objets qu'il référence le sont aussi. Ainsi même si peu de références d'objets sont malencontreusement retenus, nombreux peuvent être les objets qui resteront encombrer la mémoire et détériorer les performances.

Une des façons de résoudre rapidement ce problème est de mettre les références des objets à `null` quand ils deviennent obsolètes. Cependant, on constate alors souvent une certaine propension à le surfaire, ce qui peut réduire la lisibilité, la maintenabilité, et même la performance du code à terme.

Ainsi, mettre les références des objets à `null` devrait rester une exception plutôt que la règle. La meilleure façon d'éliminer une référence obsolète est de réutiliser/recycler la variable dans laquelle elle était contenue, ou de la laisser tomber en dehors du périmètre. Cela se fera donc naturellement s'il on a pris soin de définir chaque variable dans le plus petit périmètre possible. On rappelle en effet que les implémentations modernes de JVM ont un mécanisme de ramasse-miettes hautement optimisé.

10.2.3. [MEM-2] Éviter la création de pools d'objets.

Le coût d'établissement d'une connexion est suffisamment élevé pour justifier une réutilisation de ces objets au travers d'un pool. Cependant, de façon plus générale, maintenir son propre pool d'objets peut polluer le code, augmenter l'empreinte mémoire et détériorer la performance. Les implémentations modernes de JVM ont une gestion mémoire hautement optimisée qui surpasse aisément tout pool d'objets dit « léger ».

Il est donc **demandé** d'éviter la création de pool d'objets, sauf pour les objets de type connexion et tous les objets associés à l'utilisation de ressources, à fort coût d'instanciation. Toute création et/ou utilisation de pool **seront** donc motivées par une note d'architecture préalable et sera commentée dans le code source.

10.2.4. [MEM-3] Motiver et documenter la création de caches d'objets.

Une autre source fréquente de fuites mémoires sont les caches d'objets. Une fois la référence d'un objet mise en cache, il est facile d'oublier qu'il y est, et de l'y laisser longtemps après qu'elle soit devenue inutile et/ou inapplicable. Il y a deux solutions possible à ce problème :

- Si vous êtes assez chanceux pour implémenter un cache pour lequel une entrée est exploitable aussi longtemps qu'il existe des références à ses clefs en dehors du cache, représentez le cache en tant que **WeakHashMap** ; les entrées seront effacées automatiquement après qu'elles deviennent obsolète.
- De façon plus fréquente, la période pour laquelle une entrée de cache reste exploitable n'est pas très bien définie, on sait juste qu'avec le temps ces entrées deviennent moins précieuses. Dans de telles circonstances le cache pourra être occasionnellement nettoyé des entrées devenues inutiles (à travers l'utilisation de Timer par exemple).

C'est pourquoi il est **demandé** de motiver et documenter l'utilisation et la création de caches d'objets par une note d'architecture préalable, qui explicitera en particulier la politique de gestion mémoire choisie.

10.2.5. [MEM-4] Interdiction de forcer le ramasse-miettes

Les appels aux méthodes `System.gc()`, `Runtime.getRuntime().gc()`, et `System.runFinalization()` sont **interdits**. Ces méthodes peuvent augmenter les probabilités d'exécution du ramasse-miettes et des finaliseurs mais ne le garantissent pas. Les seules méthodes qui annonçaient une finalisation garantie (`System.runFinalizersOnExit` et `Runtime.runFinalizersOnExit`) sont dépréciées (*deprecated*, et donc interdites cf. [PROG-INTERDIT-4]).

Le code doit avoir exactement le même comportement si les appels au ramasse-miettes sont désactivés (pour cela, l'option `-Xdisableexplicitgc` peut être utilisée pour désactiver les appels à `System.gc()`).

10.2.6. [MEM-5] Interdiction d'implémenter et d'appeler `finalize`

Les finaliseurs sont imprévisibles, souvent dangereux, et rarement nécessaires. Leur utilisation peut provoquer des comportements erratiques, des pertes de performance et des problèmes de portabilité. Même si les finaliseurs ont quelques raisons valables d'utilisation, abordées ci-après, il est **interdit** d'implémenter des méthodes `finalize`.

Il n'y a en effet aucune garantie que les finaliseurs soient exécutés rapidement. Cela peut prendre un temps arbitrairement long entre le moment où l'objet est devenu non-atteignable et le moment où la méthode `finalize` est exécutée. Cela implique que rien de critique (comme la libération de ressources par exemple) ne doit être réalisé par un finaliseur.

C'est par exemple une grave erreur que de dépendre d'un finaliseur pour fermer des fichiers ouverts, car les descripteurs de fichiers ouverts constituent une ressource limitée. Si de nombreux fichiers sont laissés ouverts parce que la JVM exécute tardivement les finaliseurs, un programme peut échouer parce qu'il ne peut plus ouvrir de fichiers. La promptitude d'exécution des finaliseurs est principalement fonction de l'algorithme du ramasse-miettes, qui varie fortement d'une implémentation de JVM à une autre.

Non seulement il n'existe aucune garantie sur la rapidité d'exécution des finaliseurs, mais il n'existe pas non plus de garantie de leur exécution même. Il est en effet probable qu'un programme subisse une interruption sans exécution des finaliseurs sur certains objets qui ne sont plus atteignables. C'est pourquoi il ne faut pas dépendre d'un finaliseur pour mettre à jour des états critiques (comme par exemple libérer un verrou ou une ressource partagée).

De plus, si une exception est levée lors de la finalisation, elle est ignorée.

Méthode de « terminaison » : si l'algorithme le nécessite, on remplacera l'utilisation de `finalize` par une méthode explicite de fin de vie (de « terminaison »). Cette méthode pourra ainsi le cas échéant libérer les ressources de l'objet. Elle devra être explicitement appelée par le client/propriétaire de l'objet. L'instance de l'objet doit alors savoir si la méthode de « terminaison » a été appelée, et une exception `IllegalStateException` doit être propagée si l'objet est malencontreusement sollicité après « terminaison ».

Les appels natifs : il n'existerait qu'une seule utilisation légitime pour l'implémentation de `finalize`, qui concernerait les objets avec des pairs natifs. Mais l'utilisation d'objets natifs étant elle-même interdite par la règle (cf. [PROG-INTERDIT-3]), cette exception n'a pas lieu d'être, et l'implémentation de `finalize` est donc strictement interdite. Pour information tout de même, un pair natif est un objet natif à qui un objet normal délègue à travers des méthodes natives. Parce qu'un objet natif n'est pas un objet normal, le ramasse-miettes ignore son existence. Le finaliseur serait alors un moyen approprié pour forcer la désallocation de l'objet natif. Cependant, si l'objet natif détient des ressources, elles doivent être libérées par une méthode explicite de terminaison telle qu'introduit ci-dessus.

10.2.7. [MEM-6] Les fabriques et méthodes utilitaires devraient être statiques

Il est recommandé de déclarer les fabriques (*factory*) de façon statique, cela permet en effet aux clients de ces fabriques de partager des instances préexistantes plutôt que d'en créer des nouvelles. Cela réduit l'empreinte mémoire et les coûts de fonctionnement du ramasse-miettes.

Exemples :

```
public static final Complex ZERO = new Complex(0, 0);  
public static final Complex ONE = new Complex(1, 0);  
public static final Complex I = new Complex(0, 1);
```

De même, une méthode qui ne manipule pas d'attributs ni de variables statiques est un utilitaire. Elle devrait être déclarée **static**, cela améliore la vitesse d'exécution. Par exemple, la méthode `copyValueOf(char[])` de la classe `String` est déclarée **static** car elle ne manipule pas d'instance.

On notera également que déclarer une méthode **final** est éventuellement bon d'un point de vue modélisation mais n'a aucun impact sur les performances (cf. 7.2.6).

10.2.8. [MEM-7] Eviter l'utilisation de ThreadLocal

Chaque *thread* détient une référence implicite à sa copie de `ThreadLocal` aussi longtemps que le *thread* est en vie et que l'objet `ThreadLocal` est accessible. Après la mort du *thread*, toutes les copies des variables du `ThreadLocal` sont sujettes au ramasse-miettes (à moins que d'autres références à ces copies subsistent ailleurs).

`ThreadLocal` peut ainsi comporter de sérieux risques (potentiellement des fuites mémoires), particulièrement quand il est utilisé dans un environnement multi-threadé et avec de multiples *classloaders*. C'est pourquoi il est **fortement déconseillé** d'utiliser `ThreadLocal`. Toute exception devra être commentée dans le code, en prenant notamment en compte les avertissements suivants.

Il faut en effet noter que les *threads* sont recyclés par les serveurs web Java, et par les serveurs d'applications J2EE. Ainsi par exemple pour Tomcat, par défaut 25 *threads* sont toujours prêts à répondre aux requêtes entrantes. Utiliser `ThreadLocal` peut donc entraîner par exemple :

- des fuites mémoires : l'objet `ThreadLocal` doit absolument, quelles que soient les exceptions en cours d'exécution, être nettoyé à la fin de chaque requête pour s'assurer qu'il n'y aura pas de fuites entre les requêtes. Si l'implémentation ne peut le garantir, les fuites mémoires sont inévitables.
- un arrêt problématique de l'applicatif : ces *threads* sont chargés par le *classloader* de plus haut niveau, utiliser `ThreadLocal` liera les objets créés par le contexte du *classloader* de l'applicatif web avec celui de plus haut niveau. Comme les *threads* sont sans-cesse recyclés par le serveur, ils ne sont pas dé-référencés par le ramasse-miette, les variables du `ThreadLocal` ne sont pas nettoyées et cela pourrait empêcher l'applicatif web d'être correctement déréférencé et donc arrêté.

Aussi `ThreadLocal` ne doit jamais être utilisé directement, et s'il doit l'être, il sera :

- utilisé à travers un *wrapper* (couplé à `ServletFilter` par exemple dans le cas d'un applicatif web) et on devra s'assurer des appels systématiques aux méthodes de libération de mémoires et ressources locales au *thread* ;
- ou déclaré en tant que variable **statique**.

10.3. Règles de gestion des blocs critiques (*synchronized*) et des *threads*

10.3.1. Finalité, critères de sélection, sources et contraintes

L'objectif des règles suivantes est de gagner en performance, ou plutôt d'éviter de perdre en performance, dans la manipulation de *threads* et blocs critiques. Cela se traduira notamment par l'abandon de certaines formulations couramment utilisées, mais en réalité contre-productives.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outillage	Criticité
[THR-1.1]	Éviter l'utilisation excessive de blocs critiques (<i>synchronized</i>)			Majeure
[THR-1.2]	Règle fusionnée avec [THR-1.1] depuis la version 2.1.0.			
[THR-1.3]	Recommandations d'écriture de blocs critiques pour éviter les <i>deadlocks</i> .	SonarQube	S1860	Bloquante
[THR-1.4]	Les blocs critiques vides sont interdits.	SonarQube	S00108	Critique
[THR-2]	Utilisation du modificateur <i>volatile</i> interdite.	PMD	AvoidUsingVolatile	Majeure
[THR-3.1]	Le <i>double check locking</i> est interdit.	FindBugs	DC_DOUBLECHECK	Bloquante
[THR-3.2]	Préférer <i>static</i> à <i>synchronized</i> pour l'implémentation du singleton.			Critique
[THR-4.1]	Justifier l'utilisation de Thread par une note d'architecture préalable.	PMD	DoNotUseThreads	Bloquante
[THR-4.2]	La création de Thread est interdite dans le conteneur d'EJB.			Bloquante
[THR-4.3]	Utiliser <i>NotifyAll()</i> plutôt que <i>Notify()</i> .	SonarQube	S2446	Majeure
[THR-5]	Ne jamais invoquer <i>wait</i> en dehors d'une boucle <i>while</i> et d'un bloc synchronisé.	SonarQube	S2274pass	Bloquante
[THR-6]	Éviter les groupes de <i>threads</i> .	PMD	AvoidThreadGroup	Critique

10.3.2. [THR-1] Règles d'utilisation des blocs critiques, de *synchronized*

10.3.2.1. Rappels

Le mot-clef (l'adjectif clef) *synchronized* de Java permet d'obtenir un accès exclusif à objet et de créer ainsi des sections critiques :

- Tout objet Java est équipé d'un verrou d'exclusion mutuelle. Ainsi, pour assurer qu'une seule activité accède à un objet `unObj` d'une classe quelconque, on définit les actions sur l'objet dans une section critique par la syntaxe :

```
synchronized (unObj)
{
    /* Section critique */
}
```

- Une méthode peut aussi être qualifiée de `synchronized` :

```
synchronized T uneMethode(...)
{
    ...
}
```

Ceci est équivalent à :

```
T uneMethode(...)
{
    synchronized (this)
    {
        ...
    }
}
```

Il y a donc exclusion d'accès de l'objet sur lequel on applique la méthode, pas de la méthode elle-même, qui peut être exécutée concurremment sur des objets différents.

- Chaque classe possède aussi un verrou exclusif qui s'applique aux méthodes de classe (méthodes statiques) :

```
class X
{
    static synchronized T foo()
    { ... }
    static synchronized T' bar()
    { ... }
}
```

L'utilisation de `synchronized` assure ici l'exécution en exclusion mutuelle pour toutes les méthodes *statiques* synchronisées de la classe X.

Les verrous sont qualifiés de récursifs ou réentrants : si une activité possède un verrou, une deuxième demande provenant de cette activité est satisfaite sans causer d'auto-interblocage, et le code suivant s'exécute sans problème :

```
synchronized(o)
{
    ...
    synchronized(o)
    { ... }
    ...
}
```

La synchronisation est également la garantie que chaque *thread* entrant dans un bloc ou une méthode `synchronized` voit les effets de toutes les transitions d'états précédents contrôlées par le même verrou. Après la sortie d'un premier *thread* d'une région `synchronized`, tout *thread* qui entre dans une région `synchronized` par le même verrou voit la transition d'état généré par le premier *thread*.

Dire que l'utilisation de ces blocs critiques est à éviter est dangereusement faux. La synchronisation est requise pour garantir à la fois l'exclusion mutuelle et une communication sûre entre *threads*. C'est un aspect complexe mais essentiel du modèle de mémoire du langage Java.

Lorsque l'on rédige une classe Java celle-ci doit être utilisable en multi-tâches. L'implémentation et l'utilisation de tels blocs critiques est donc inévitable, chacun doit être cependant conscient que les méthodes **synchronized** sont plus lentes que les méthodes classiques même si les dernières générations de machines virtuelles améliorent considérablement cet aspect.

10.3.2.2. [THR-1.1] Éviter l'utilisation excessive de blocs critiques, de **synchronized**.

Il est donc recommandé d'éviter l'utilisation excessive de blocs critiques, de **synchronized**. On privilégiera :

- les classes immuables (non mutables) : (cf. 7.2.5) il n'y aura pas de conflit d'accès ;
- une approche pragmatique dans le cas d'utilisation de classes mutables : on évitera au maximum l'utilisation simultanée d'une instance, et le cas échéant on veillera à faire des copies défensives d'objets (cf. 7.5.2). Toutefois il faudra garantir qu'à chaque fois que plusieurs threads partagent une donnée mutable, chaque thread qui lit ou écrit la donnée doit obtenir un verrou ;
- les méthodes réentrantes : c'est à dire ayant la propriété de pouvoir être appelées de façon récursive et simultanément par plusieurs tâches (*thread*) utilisatrices sans effet de bord : le traitement de la méthode par un thread est indépendant de tout traitement précédent. Pour être réentrante une méthode ne doit tenir aucune donnée statique (sauf final), doit travailler uniquement avec les données fournies par l'appelant, et ne doit pas appeler d'autres fonctions réentrantes ;
- et enfin, à l'inverse, on évitera l'écriture d'objet modal, c'est à dire dont le traitement est dépendant des traitements précédents. De tels objets exigent évidemment une bonne documentation.

C'est pourquoi, il est également demandé de motiver et documenter l'utilisation de **synchronized** dans le code, en explicitant la politique de gestion en multi-tâches exigée par le contexte applicatif.

10.3.2.3. [THR-1.3] Recommandations d'écriture de blocs critiques pour éviter les *deadlocks*.

Enfin, quelques recommandations dans l'écriture de blocs critiques, afin d'éviter les *deadlocks* et la corruption de données :

- ne jamais appeler une méthode inconnue depuis un bloc critique ;
- et plus généralement limiter la quantité de travail effectuée à l'intérieur.

Contre-exemple :

```
public class ExempleDeDeadLock
{
    public static void main(String[] args)
    {
        final Object ressource1 = "ressource1";
        final Object ressource2 = "ressource2";
        /* t1 essaye de verrouiller ressource1 puis ressource2 */
        Thread t1 = new Thread()
        {
            public void run()
            {
                /* Verrouille ressource 1 */
                synchronized (ressource1)
                {
                    System.out.println("Thread 1 a verrouillé la ressource 1");

                    try
                    {
                        Thread.sleep(50);
                    }
                }
            }
        };
    }
}
```



```
        catch (InterruptedException e)
        {}

        synchronized (ressource2)
        {
            System.out.println("Thread 1 a verrouillé la ressource 2");
        }
    }
};

/* t2 essaye de verrouiller ressource2 puis ressource1 */
Thread t2 = new Thread()
{
    public void run()
    {
        synchronized (ressource2)
        {
            System.out.println("Thread 2 a verrouillé la ressource 2");

            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException e)
            {}

            synchronized (ressource1)
            {
                System.out.println("Thread 2 a verrouillé la ressource 1");
            }
        }
    }
};

/*
 * Si tout se passe comme prévu, le deadlock se produit,
 * et le programme ne s'arrête jamais.
 */
t1.start();
t2.start();
}
```

La synchronisation sur des types primitifs «boxed» peut amener à des *deadlocks* et notamment pour :

- les boolean ;
- les primitifs encapsulés (string, integer, etc.) ;
- les constantes partagées.

Contre-exemple : Boolean

```
private static Boolean initied = Boolean.FALSE; ...
synchronized(initied)
{
    if (!initied)
    {
        init();
        initied = Boolean.TRUE;
    }
}
```

Contre-exemple : les primitifs encapsulés

```
private static Integer count = 0; ...
    synchronized(count)
    {
        count++;
    }
```

Contre-exemple : les constantes partagées

```
private static String LOCK = "LOCK"; ...
    synchronized(LOCK)
    {
        ...
    }
```

Contre-exemple : les primitifs encapsulés

```
private static final Integer fileLock = new Integer(1); ...
    synchronized(fileLock)
    {
        .. do something ..
    }
```

De manière très générale, on préférera synchroniser sur un objet plutôt qu'un type primitif « boxed ».

Exemple : déclaration d'un objet

```
private static final Object fileLock = new Object(); ; ...
    synchronized(fileLock)
    {
        .. do something ..
    }
```

10.3.2.4. [THR-1.4] Les blocs `synchronized` vides sont interdits

Les blocs critiques vides sont **interdits**, car ils sont inutiles.

Contre-exemple :

```
public class ThrUnQuatre
{
    public void bar()
    {
        synchronized (this)
        {
            /* vide ! */
        }
    }
}
```

10.3.3. [THR-2] Utilisation du modificateur `volatile` est à éviter

L'utilisation du mot-clef, du modificateur `volatile` peut constituer une alternative viable à une synchronisation ordinaire.

En effet, les variables précédées par le modificateur `volatile` obligent la machine virtuelle Java à relire la valeur des variables au sein de la mémoire partagée à chaque fois qu'elles sont accédées. Les compilateurs Java s'autorisent à mettre en cache les variables membres dans les registres mémoires. Autrement dit, les *threads* utilisant une variable membre partagée, n'accèdent qu'à des copies de cette variable dont les modifications ne peuvent être visibles pour ces *threads* concurrents. Une incohérence de la variable a de fortes probabilités de déclencher un fonctionnement inattendu du programme. Les variables possédant le modificateur `volatile` sont chargées (*load*) à partir de la mémoire centrale avant chaque utilisation. Suite à leur exploitation (*read* ou *write*), les variables volatiles sont stockées (*stored*) en mémoire centrale. Par ce moyen, la valeur réelle d'une telle variable est assurée d'être cohérente à l'intérieur de chaque *thread*.

Ainsi, si plusieurs *threads* sont susceptibles d'accéder à une variable partagée mutable, on peut alors choisir :

- de faire en sorte que ces *threads* n'utilisent que des méthodes ou des blocs synchronisés pour lire et écrire cette donnée ;
- ou déclarer cette variable `volatile` afin de s'assurer que tous les *threads* puissent voir ses modifications de valeur.

Mais c'est une technique particulièrement avancée. Son utilisation requière une expertise forte du JMM (Java Memory Model), de plus son applicabilité est en général peu connue. C'est pourquoi, principalement pour des raisons de maintenabilité et de portabilité, l'utilisation du modificateur `volatile` est **interdite**. Toute dérogation ou exception devra être documentée et validée, et devra faire l'objet d'une note d'architecture préalable.

10.3.4. [THR-3] Patterns et anti-patterns dans l'écriture d'un Singleton

10.3.4.1. [THR-3.1] Le double check locking est interdit

Le *double check locking* (DCL) a vu le jour suite à la volonté d'optimiser l'implémentation du pattern singleton. Avec le DCL, l'implémentation de la méthode `getInstance()` est la suivante :

Contre-exemple :

```
public static Singleton getInstance()
{
    if (instance == null)
    {
        synchronized (Singleton.class)    // 1
        {
            if (instance == null)        // 2
            {
                instance = new Singleton(); // 3
            }
        }
    }
    return instance;
}
```

La théorie derrière le DCL est que la seconde vérification en ligne 2 rend impossible la création de deux Singleton comme vu précédemment. Cette théorie semble parfaite. Malheureusement la réalité est complètement différente. Le DCL n'apporte aucune garantie de fonctionnement. L'échec du DCL n'est pas dû à un défaut d'implémentation de la JVM, mais au modèle de mémoire de la JVM. Ce modèle de mémoire autorise ce qui est connu comme le "out of order writes", ou écriture dans le désordre. Pour plus d'informations sur la gestion de la mémoire en Java, se référer aux spécifications de la JVM. Tout code de type DCL est donc évidemment complètement **interdit**.

10.3.4.2. [THR-3.2] Préférer `static` à `synchronized` pour l'implémentation du Singleton

L'utilisation de Singleton en environnement multi-thread doit faire l'objet d'un certain nombre de précautions. Deux solutions sont retenues pour l'implémentation d'un Singleton :

- synchroniser toute la méthode `getInstance()` ;
- abandonner la synchronisation et utiliser un initialiseur `static`.

La deuxième solution est celle recommandée.

Cette solution présente l'avantage, non négligeable, d'éliminer la synchronisation lors de chaque accès. Cela fonctionne car la machine virtuelle garantit qu'un objet d'une classe ne peut être accédé tant que la classe n'est pas complètement chargée.

Toutefois :

- cette solution n'est viable que si toutes les informations nécessaires à la création du Singleton sont disponibles au moment du chargement, ce qui ne peut être toujours garanti. En effet, la machine virtuelle charge les classes comme elle le veut et il n'y a aucune garantie que le chargement soit différé jusqu'au premier appel de la méthode `getInstance()` ;
- en cas de sérialisations puis de désérialisations multiples, il est possible de se retrouver avec plusieurs instances, à moins d'implémenter `readResolve`, comme dans l'exemple suivant.

Exemple :

```
public class Singleton implements java.io.Serializable
{
    public static Singleton INSTANCE = new Singleton();

    protected Singleton()
    {
        /* Exists only to defeat instantiation. */
    }
    private Object readResolve()
    {
        return INSTANCE;
    }
}
```

- si l'initialisation est coûteuse, il est intéressant de ne la réaliser qu'à la demande :

Exemple :

```
/* The initialize-on-demand holder class idiom */
private static class FooHolder
{
    static final Foo foo = new Foo();
}

public static Foo getFoo()
{
    return FooHolder.foo;
}
```

- et enfin, il est bon de noter que même si le constructeur est privé, il peut être insidieusement rendu accessible par réflexion (en combinant `getDeclaredConstructors` et `setAccessible` (cf. [PROG-INTERDIT-2]) .

10.3.5. [THR-4] Règles de création de thread

10.3.5.1. [THR-4.1] Justifier l'utilisation de Thread par une note d'architecture préalable

Les problématiques exposées précédemment montrent que la gestion multi-tâches est complexe. C'est pourquoi il est également **demandé** de motiver et documenter l'utilisation de *threads* par une note d'architecture préalable, qui explicitera toute création de *thread* de traitement spécifique qui ne soit pas géré par le conteneur/serveur d'applications.

10.3.5.2. [THR-4.2] La création de Thread est interdite dans le conteneur d'EJB

J2EE définit 4 types de conteneurs : EJB, *servlet*, *applet*, *application client*. Il est rappelé ici que la création de Thread est **interdite** dans le conteneur d'EJB, mais pas dans les autres conteneurs.

Cette règle est un rappel des spécifications EJB-2.0 ^[5b] (p.495) :

"The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enter-prise bean must not attempt to manage thread groups."

10.3.5.3. [THR-4.3] Utiliser notifyAll() plutôt que notify()

Thread.notify() notifie un *thread* monitorant un objet (objet « verrou », verrouillé par l'utilisation de synchronized). Si plus d'un *thread* monitore cet objet, alors un seul est arbitrairement choisi pour être notifié.

Pour cette raison il est **recommandé** d'appeler notifyAll() plutôt que notify().

Exception : L'appel de notifyAll() réveille tous les *threads*. Cela ne pose aucun problème s'il n'est pas nécessaire de contrôler l'ordre d'exécution de leur « réveil ». Cependant, si l'ordre dans lequel ces *threads* se réveillent doit être contrôlé, la même problématique que celle inhérente à l'appel de notify() se pose.

On conseille alors deux solutions d'implémentation : si chaque *thread* possède son propre objet « verrou », utilisez notify(), sinon utilisez notifyAll() (cf. « Apply the specific notification pattern to control the order of thread execution »).

10.3.6. [THR-5] Ne jamais invoquer wait() en dehors d'une boucle while et d'un bloc synchronisé

Le méthode Object.wait() est utilisée pour faire attendre par un *thread* la levée d'une certaine condition. Elle doit **doit** être appelée à l'intérieur d'un bloc synchronisé, pour verrouiller l'objet qui l'invoque.

Exemple :

```
synchronized (obj)
{
    while (<condition non respectée>)
        obj.wait();
    /* traitements appropriés à la condition */
}
```

Il faut toujours **invoquer** la méthode wait() à l'intérieur d'une boucle while. En effet, la boucle sert à tester la condition avant et après l'attente :

- avant, car si la condition est levée et notifiée (via notifyAll) avant l'attente, il n'y a pas de garantie que le *thread* soit réveillé de son attente ;

- après, car si le *thread* est réveillé alors que la condition n'est pas levée, il peut détruire les invariants protégés par le verrou.

10.3.7. [THR-6] Eviter les groupes de threads

Ne pas utiliser `ThreadGroup` : bien qu'il soit destiné à des environnements multi-threadés, il contient certaines méthodes qui ne sont pas *thread safe*.

Contre-exemple :

```
public class ThrSix
{
    void buz()
    {
        ThreadGroup tgp = new ThreadGroup("Mon threadgroup");
        tgp = new ThreadGroup(tgp, "mon thread group");
        tgp = Thread.currentThread().getThreadGroup();
        tgp = System.getSecurityManager().getThreadGroup();
    }
}
```

10.4. Optimisation dans la manipulation de chaînes de caractères

10.4.1. Finalité, critères de sélection, sources et contraintes

L'objectif des règles suivantes est le gain de performance dans la manipulation des chaînes de caractères, tout comme la simplification de certaines formulations couramment utilisées mais en réalité contre-productives.

Réf.	Descriptif	Contrôle automatisé		
		Outils	Réf. règle outillage	Criticité
[PERF-STR-1]	Ne pas utiliser <code>new String</code> avec une constante chaîne de caractères, utiliser l'affectation.	FindBugs	DM_STRING_VOID_CTOR DM_STRING_CTOR	Majeure
[PERF-STR-2]	Ne pas utiliser <code>String</code> pour la concaténation de chaînes de caractères.	PMD	UseStringBufferForStringAppends	Critique
[PERF-STR-3]	Préciser la taille des <code>StringBuffer</code> de plus de 16 caractères.	PMD	InsufficientStringBufferDeclaration	Majeure
[PERF-STR-4]	Ne pas concaténer dans un appel au constructeur de <code>StringBuffer</code>	PMD	InefficientStringBuffering	Majeure
[PERF-STR-5]	Ne pas concaténer des caractères en tant que <code>String</code> avec <code>StringBuffer.append()</code>	PMD	AppendCharacterWithChar	Majeure
[PERF-STR-6]	Ne pas réaliser des appels successifs à <code>StringBuffer.append()</code>	PMD	ConsecutiveLiteralAppends	Majeure
[PERF-STR-7]	Placer la chaîne de caractères (<i>literal</i>) en premier lors d'une comparaison de <code>String</code> avec <code>equals()</code>	Squid	S1132	Critique

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[PERF-STR-8]	<u>Ne pas</u> utiliser <code>String.indexOf()</code> avec des caractères en tant que <code>String</code>	PMD	UseIndexOfChar	Majeure
[PERF-STR-9]	Règle supprimée depuis la version 1.1			
[PERF-STR-10]	Pour les comparaisons non dépendantes de la casse, <u>utiliser</u> <code>equalsIgnoreCase()</code>	SonarQube	S1157	Majeure

10.4.2. **[PERF-STR-1]** Ne pas utiliser `new String` avec une constante chaîne de caractères, utiliser l'affectation.

Ci-dessous, la chaîne « hello » est déjà une instance de `String`.

La constante chaîne de caractère doit être directement affectée à la variable `String`, sans passer par `new`.

Code interdit :	Code correct :
<code>String str = new String("hello");</code>	<code>String str = "hello";</code>

10.4.3. **[PERF-STR-2]** Ne pas utiliser `String` pour la concaténation de chaînes de caractères.

En Java, une instance de la classe `String` est immuable, c'est à dire qu'après avoir été créée, la chaîne ne peut plus être modifiée. Cela s'avère très pratique dans beaucoup de situations : inutile par exemple de dupliquer une instance de `String` pour s'assurer qu'elle restera constante (comme c'est le cas en C++ par exemple). Mais cette propriété se révèle désastreuse avec l'emploi de l'opérateur '+' pour la concaténation de chaîne, car chaque étape de la concaténation implique la construction d'une nouvelle instance de `String`.

Contre-exemple :	Exemple correct :
<pre>String result = ""; // création d'une chaîne vide for (int i = 0; i < 10; i++) { result = result + i; } System.out.println(result); // "0123456789"</pre>	<pre>StringBuffer result = new StringBuffer(10); // 10 = taille maximale de la chaîne for (int i = 0; i < 10; i++) { result.append(i); } System.out.println(result.toString()); // "0123456789"</pre>

Lors de l'exécution de ce programme, chaque itération de la boucle construit une nouvelle instance de `String`. Chaque itération oblige donc la JVM à trouver de la place en mémoire, instancier l'objet, copier le contenu des deux chaînes dans la nouvelle, libérer la mémoire, recommencer à l'itération suivante. Cela revient à créer dix instances de `String` pour les résultats intermédiaires.

La classe `java.lang.StringBuffer` est une classe qui gère une chaîne modifiable. Cette classe a été spécialement conçue pour manipuler des chaînes de caractères.

Ce code produit exactement le même résultat que le précédent, sauf qu'il instancie un seul objet là où dix étaient nécessaires.

Depuis J2SE 5.0 (également appelée *Tiger*), il est possible d'utiliser la classe `java.lang.StringBuilder`. Son fonctionnement est identique à celui de `StringBuffer` à la différence qu'il n'est pas *thread-safe*. S'il n'y a pas de besoin de synchronisation (comme c'est souvent le cas), préférer `StringBuilder` car il sera plus performant.

Il est donc **fortement déconseillé** d'utiliser `String` pour réaliser la concaténation de chaînes de caractères, on utilisera `StringBuffer` ou (quand le contexte applicatif le permet) `StringBuilder`.

10.4.4. [PERF-STR-3] Préciser la taille des `StringBuffer` de plus de 16 caractères

Par défaut, un `StringBuffer` est initialisé avec une taille de 16 caractères. A chaque fois que celle-ci est dépassée, `StringBuffer` redimensionne automatiquement son tableau de caractères pour pouvoir accueillir le texte. Mais cette option est aussi coûteuse que la création d'une instance de `String`, et il convient donc de **préciser** intelligemment cette valeur.

Contre-exemple :	Exemple correct :
<pre>public class PerfStrTrois { void bar() { StringBuffer bad = new StringBuffer(); bad.append("Ceci est une longue chaîne de caractères, qui excède les 16 caractères par défaut"); } }</pre>	<pre>public class PerfStrTrois { void bar() { StringBuffer good = new StringBuffer(81); good.append("Ceci est une longue chaîne de caractères, qui excède les 16 caractères par défaut"); } }</pre>

10.4.5. [PERF-STR-4] Ne pas concaténer de non littéraux dans un appel au constructeur de `StringBuffer`

Pour des raisons de performance et de lisibilité du code source, ne pas concaténer de non littéraux dans un appel au constructeur de `StringBuffer`, utiliser `append()`.

Contre-exemple :	Exemple correct :
<pre>public class PerfStrQuatre { void bar() { StringBuffer bad = new StringBuffer("tmp = "+System.getProperty("java.io.tmpdir")) ; } }</pre>	<pre>public class PerfStrQuatre { void bar() { StringBuffer good = new StringBuffer("tmp = "); good.append(System.getProperty("java.io.t mpdir")); } }</pre>

10.4.6. [PERF-STR-5] Ne pas concaténer des caractères en tant que `String` avec `StringBuffer.append`

Ne pas concaténer des caractères en tant que `String` avec `StringBuffer.append`, mais les concaténer en tant que caractères.

Contre-exemple :	Exemple correct :
<pre>public class PerfStrCinq</pre>	<pre>public class PerfStrCinq</pre>

<pre> { void bar() { StringBuffer sbr = new StringBuffer(); sbr.append("a"); } } </pre>	<pre> { void bar() { StringBuffer sbr = new StringBuffer(); sbr.append('a'); } } </pre>
---	---

10.4.7. [PERF-STR-6] Ne pas réaliser des appels successifs à StringBuffer.append

Ne pas réaliser des appels successifs à **StringBuffer.append** avec des chaînes de caractères ; tenter plutôt de les regrouper en un seul appel à **append**, avec une seule chaîne de caractères.

Contre-exemple :	Exemple correct :
<pre> public class PerfStrSix { void bar() { StringBuffer sbr = new StringBuffer(); sbr.append("Hello").append(" ").append("World"); } } </pre>	<pre> public class PerfStrSix { void bar() { StringBuffer sbr = new StringBuffer(); sbr.append("Hello World"); } } </pre>
<pre> public class PerfStrSix { void bar() { StringBuffer sbr = new StringBuffer(); sbr.append("Hello "); sbr.append("World") } } </pre>	<pre> public class PerfStrSix { void bar() { StringBuffer sbr = new StringBuffer(); String var = "World" sbr.append("Hello"); sbr.append(var); } } </pre>

10.4.8. [PERF-STR-7] Placer la chaîne de caractères (*literal*) en premier lors d'une comparaison de String avec equals

Contre-exemple :	Exemple correct :
<pre> public class PerfStrSept { boolean bar(String str) { return str.equals("2"); } } </pre>	<pre> public class PerfStrSept { boolean bar(String str) { return "2".equals(str); } } </pre>

Placer la chaîne de caractères (*literal*) en premier lors d'une comparaison de **String** avec **equals** permet, si l'objet **String** est null, de ne pas obtenir de **NullPointerException**, mais d'avoir un retour à **false**.

10.4.9. **[PERF-STR-8]** Ne pas utiliser `String.indexOf()` avec des caractères en tant que `String`.

Ne pas utiliser `String.indexOf()` avec comme paramètre des caractères en tant que `String`. En effet, `String.indexOf()` est plus rapide avec un simple caractère.

Contre-exemple :	Exemple correct :
<pre>public class PerfStrHuit { void bar() { String str = "hello world"; int index = str.indexOf("d"); } }</pre>	<pre>public class PerfStrHuit { void bar() { String str = "hello world"; int index = str.indexOf('d'); } }</pre>

10.4.10. **[PERF-STR-9]** Règle supprimée depuis la version 1.1

10.4.11. **[PERF-STR-10]** Pour les comparaisons non dépendantes de la casse, utiliser `equalsIgnoreCase()`

Utiliser `equalsIgnoreCase()` est plus rapide que `toUpperCase / toLowerCase().equals()`

Contre-exemple	Exemple correct
<pre>public class PerfStrDix { public boolean bar(String buz) { return buz.toUpperCase().equals("baz"); } }</pre>	<pre>public class PerfStrDix { public boolean bar(String buz) { return buz.equalsIgnoreCase("baz"); } }</pre>

10.5. Autres règles pour la performance

10.5.1. Finalité, critères de sélection, sources et contraintes

L'objectif des règles suivantes est toujours le gain de performance.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[PERF-AUTR-1]	Favoriser les <i>buffers</i>			
[PERF-AUTR-2]	Ne pas utiliser le modificateur <code>final</code> pour des motifs de performance			
[PERF-AUTR-3]	Copier les tableaux avec <code>System.arraycopy</code>	PMD	AvoidArrayLoops	Majeure
[PERF-AUTR-4]	Utiliser la taille de la collection lors des appels à <code>Collection.toArray</code>	PMD	OptimizableToArrayCall	Critique
[PERF-AUTR-5]	Préférer <code>List</code> à <code>Vector</code>	SonarQube	S1149	Majeure
[PERF-AUTR-6]	Préférer <code>Map</code> à <code>HashTable</code>	SonarQube	S1149	Majeure

10.5.2. [PERF-AUTR-1] Favoriser les Buffers

La lecture et l'écriture, en utilisant le comportement par défaut des classes du paquetage `java.io` (byte par byte), donnent de mauvaises performances.

L'emploi de classes « bufferisées », comme `BufferedInputStream` et `BufferedOutputStream`, est recommandé, car il améliore les performances de manière significative.

Contre-exemple, code lent :	Code rapide, recommandé :
<pre>public class PerfAutrUn { public static void main(String args[]) { try { /* Sans buffering */ Reader lecteur = new FileReader(args[0]); int compteur; while ((compteur = lecteur.read()) != -1); lecteur.close(); } catch (IOException e) </pre>	<pre>public class PerfAutrUn { public static void main(String args[]) { try { /* Avec buffering */ Reader lecteur = new FileReader(args[0]); lecteur = new BufferedReader(lecteur); int compteur; while ((compteur = lecteur.read()) != -1); lecteur.close(); </pre>

<pre> { /* Traitement de l'exception */ } } } </pre>	<pre> } catch (IOException e) { /* Traitement de l'exception */ } } </pre>
--	--

10.5.3. [PERF-AUTR-2] Ne pas utiliser le modificateur `final` pour des motifs de performance

Il ne faut pas utiliser le modificateur `final` pour des motifs de performance.

En particulier, avec les JVM récentes (depuis la version 1.3), le compilateur JIT (*Just In Time*) est suffisamment intelligent pour *inliner* sans avoir à déclarer des éléments `final` (ce qui constitue un choix de conception assez structurant).

10.5.4. [PERF-AUTR-3] Copier les tableaux avec `System.arraycopy`

Certaines méthodes de l'API Java sont rédigées en C et permettent une nette amélioration des performances. Par exemple, la méthode `System.arraycopy()` permet de copier un tableau très rapidement.

Il est donc préférable de copier les tableaux avec la méthode `arrayCopy()`, plutôt qu'avec la méthode `clone()` ou une boucle `for`.

10.5.5. [PERF-AUTR-4] Utiliser la taille de la collection lors des appels à `Collection.toArray`

Les appels à `Collection.toArray` **doivent** utiliser la taille de la collection à la place d'un tableau vide.

Contre-exemple, code lent :	Code rapide, recommandé :
<pre> public class PerfAutrQuatre { void bar(Collection coll) { // Pourrait être optimisé coll.toArray(new PerfAutrQuatre[0]); } } </pre>	<pre> public class PerfAutrQuatre { void bar(Collection coll) { // Beaucoup mieux coll.toArray(new PerfAutrQuatre[coll.size()]); } } </pre>

10.5.6. [PERF-AUTR-5] Préférer `List` à `Vector`

Pour des raisons de performance, `java.util.List` (introduit depuis Java 1.2) sera préféré à `Vector`.

Contre-exemple :
<pre> void bar() { Vector v = new Vector(); } </pre>

10.5.7. [PERF-AUTR-6] Préférer Map à HashTable

Pour des raisons de performance, `java.util.Map` (introduit depuis Java1.2) sera préféré à HashTable.

Contre-exemple :

```
void bar()
{
    Hashtable h = new Hashtable();
}
```

10.6. Micro optimisations

10.6.1. Finalité, critères de sélection, sources et contraintes

L'objectif des règles suivantes est toujours le gain de performance, même si dans ce chapitre celui-ci sera moins flagrant que pour les règles précédentes.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[MICRO-OPTIM-1]	Optimiser l'implémentation de <code>equals()</code>			Majeure
[MICRO-OPTIM-2]	Préférer <code>MaClass.class</code> à <code>Class.forName</code>			Majeure
[MICRO-OPTIM-3]	Règle supprimée depuis la version 1.0			
[MICRO-OPTIM-4]	Ne pas utiliser le type <code>short</code>	PMD	AvoidUsingShortType	Majeure
[MICRO-OPTIM-5]	Préférer <code>valueOf()</code> à <code>new</code> pour l'instantiation de types primitifs	PMD FindBugs	IntegerInstantiation ByteInstantiation ShortInstantiation LongInstantiation BooleanInstantiation DM_FP_NUMBER _CTOR	Majeure

10.6.2. [MICRO-OPTIM-1] Optimiser l'implémentation de equals

Ci-dessous, si le paramètre est égal à `this`, la méthode peut retourner immédiatement `true` sans avoir à tester chaque attribut.

Exemple :

```
public boolean equals(Object obj)
{
    if (this == obj) return true;
```

```
    /* ... */  
}
```

Rappel : si la méthode `equals()` est redéfinie, la méthode `hashCode()` doit l'être également.

10.6.3. [MICRO-OPTIM-2] Préférer `MaClass.class` à `Class.forName`

Préférer `MaClass.class` à `Class.forName("MaClass")`, la première formulation est plus rapide.

10.6.4. [MICRO-OPTIM-4] Ne pas utiliser le type `short`

Java utilise le type `short` pour réduire l'espace mémoire occupé par les instances, et non pour optimiser le calcul. Au contraire. La JVM ne possède pas d'arithmétique en `short`. Le P-code doit convertir les `short` en `int`, faire le calcul et convertir le résultat en `short`.

Il est donc déconseillé d'utiliser `short` en Java, sauf si l'impact mémoire est très important.

10.6.5. [MICRO-OPTIM-5] Préférer `valueOf()` à `new` lors de l'instantiation de types primitifs

Depuis Java5, appeler les constructeurs de types primitifs provoque une allocation mémoire. Il est donc préférable d'utiliser la méthode `valueOf()` pour : `Integer`, `Byte`, `Short`, `Long`, `Boolean` et `Double`.

Contre-exemple :

```
private Integer i = new Integer(0);
```

Exemple correct :

```
private Integer i = Integer.valueOf(0);
```

11. Règles J2EE

De la spécification J2EE ^[5], n'ont été retenus pour cette charte que les chapitres relatifs à JDBC (version 2.0), aux JSP (version 1.2) et aux Servlets (version 2.3), aux EJB Session et Message-Driven Beans (version 2.0).

11.1. Règles JDBC

11.1.1. Finalité, critères de sélection, sources et contraintes

Rappel de la spécification JDBC (Java DataBase Connectivity) :

« *The JDBC 2.0 extension includes APIs for row sets, connection naming via JNDI, connection pooling, and distributed transaction support. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server.* »

Remarque : l'utilisation d'un *framework* de persistance listé dans la matrice technologique (par exemple : Hibernate) permet par défaut de ne pas avoir à manipuler de JDBC directement. Dans ce cas, les règles JDBC définies ci-dessous n'ont pas lieu d'être vérifiées. Par contre, il est parfois nécessaire avec ces *frameworks* de manipuler directement du JDBC, et dans ce cas les règles suivantes s'appliquent.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[JDBC-DS]	Les accès au SGBD se font via le DataSource , et non via le DriverManager .			Bloquante
[JDBC-POOL]	Ne pas créer de connexions au fur et à mesure des besoins mais utiliser le pool de connexions fourni par le serveur d'applications.			Bloquante
[JDBC-RES]	Fermer, dans l'ordre : les ResultSet , Statement et Connection JDBC dans un bloc finally .	SonarQube	S2095	Bloquante
[JDBC-PS]	Les PreparedStatement sont préférés aux Statement .	SonarQube	S2077	Critique
[JDBC-CS]	Les CallableStatement sont interdits.			Critique
[JDBC-RS]	Toujours vérifier la valeur de retour ResultSet .	PMD	CheckResultSet	Bloquante
[JDBC-TX-1]	Une modification du niveau d'isolation par défaut doit être documentée.			Bloquante
[JDBC-TX-2]	Pour des mises à jour groupées, utiliser des <i>batch update</i> .			Critique

11.1.2. [JDBC-DS] Les accès au SGBD se font via le DataSource, et non via le DriverManager

Le DataSource est un objet J2EE qui joue le rôle de pool de connexion vers une instance de base de données.

L'accès à une base de données via une DataSource est préféré au DriverManager, car :

- les drivers ne sont pas obligés de s'enregistrer eux-mêmes, comme ils le font avec le DriverManager ;
- les implémentations de DataSource permettent de facilement changer les propriétés des sources de données. Par exemple, il n'est plus nécessaire de modifier du code applicatif lors de changements concernant la base de données, son emplacement physique ou encore un driver ;
- les instances de Connection fournies par les DataSource ont des capacités étendues (pool de connexions, transactions distribuées, etc).

Exemple :

```
Connection connection = null;
try
{
    /* Récupération de la DataSource à partir du contexte */
    Context ctx = new InitialContext();
    DataSource source = (DataSource)ctx.lookup("jdbc/MaDataSource");

    /* Récupération d'une Connection */
    connection = source.getConnection();
    /* ... */
}
catch (NamingException ex)
{
    /* traitement de l'exception */
}
catch (SQLException ex)
{
    /* traitement de l'exception */
}
return connection;
```

11.1.3. [JDBC-POOL] Ne pas créer de connexions au fur et à mesure des besoins mais utiliser le pool de connexions fourni par le serveur d'applications

Exemple de DataSource *OracleDS* configurant un pool de connexions Oracle pour JBoss :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- NOTE:
    Fichier de config pour la prise en compte de la base de donnees Oracle. Ce
    fichier doit etre mis dans le repertoire $JBoss_HOME/server/appli/deploy/.
    Les mots en italiques sont a substituer avec les valeurs courantes du projet.
-->
<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
    <connection-url>jdbc:oracle:thin:@adresse_BDD:1521:SID_BDD</connection-url>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <user-name>nomUtilisateur</user-name>
    <password>motDePasseUtilisateur</password>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <max-pool-size>40</max-pool-size>
```



```
<min-pool-size>10</min-pool-size>
<!-- Use the security domain defined in conf/login-config.xml -->
<!-- <security-domain>EncryptDBPassword</security-domain> -->
</local-tx-datasource>
</datasources>
```

11.1.4.[JDBC-RES] Fermer, dans l'ordre : les ResultSet, Statement et Connection JDBC dans un bloc finally

Toutes les ressources JDBC (ResultSet, Statement, Connection) **doivent** être fermées quand elles ne sont plus utilisées. Cela permet d'éviter des fuites de ressources, même si une exception se produit, et de gêner le bon fonctionnement de la base de données.

Les ressources JDBC **doivent** être fermées dans un bloc finally.

Exemple :

```
Connection connection = null;
Statement statement = null;
ResultSet resultat = null;
try
{
    /* initialisation des trois éléments, divers traitements sur la base de
    données */
}
catch (Exception ex)
{
    /* gestion des différents types d'erreur pouvant survenir */
}
finally
{
    /* Tentative de fermeture du ResultSet */
    try
    {
        if (resultat != null)
        {
            resultat.close();
        }
    }
    catch (SQLException e)
    {
        /* traitement de l'exception */
    }

    /* Tentative de fermeture du Statement */
    try
    {
        if (statement != null)
        {
            statement.close();
        }
    }
    catch (SQLException e)
    {
        /* traitement de l'exception */
    }

    /* Tentative de fermeture de la Connection */
    try
    {
        if (connection != null)
```

```

        {
            connection.close();
        }
    }
    catch (SQLException e)
    {
        /* traitement de l'exception */
    }
}

```

Remarque : le bloc `finally` ci-dessous est volumineux du fait de la nécessité de traiter les `SQLException` pouvant intervenir lors de la fermeture des ressources. Une méthode utilitaire pourra être créée à cette fin.

11.1.5. [JDBC-PS] Les `PreparedStatement` sont préférés aux `Statement`

Les instances de `PreparedStatement` contiennent une instruction SQL déjà compilée, ce qui améliore les performances si cette instruction doit être appelée plusieurs fois.

Les `PreparedStatement` sont aussi **préférés** pour des raisons de sécurité. En effet, ils sont moins vulnérables aux attaques par injection de SQL.

11.1.6. [JDBC-CS] Les `CallableStatement` sont interdits

Pour rappel, l'interface `CallableStatement` étend `PreparedStatement`, et permet de faire appel aux procédures stockées et aux fonctions (procédures stockées renvoyant un résultat) de manière standard pour tous les SGBD.

Les procédures stockées étant interdites par l'architecture Copernic, les `CallableStatement` **n'ont pas lieu** d'être utilisés.

11.1.7. [JDBC-RS] Toujours vérifier la valeur de retour `ResultSet`

Toujours **vérifier** la valeur de retour après avoir appelé l'une des méthodes de navigation de `ResultSet` ; ne jamais assumer qu'un résultat sera présent.

Pour information, les méthodes de navigation de `ResultSet` incluent : `next`, `first`, `last` et `previous`.

Contre-exemple :	Exemple correct :
<pre> Statement stat = conn.createStatement(); ResultSet rst = stat.executeQuery("SELECT name FROM person"); rst.next(); String firstName = rst.getString(1); </pre>	<pre> Statement stat = conn.createStatement(); ResultSet rst = stat.executeQuery("SELECT name FROM person"); if (rst.next()) { String firstName = rst.getString(1); } else { /* traitement de l'erreur */ } </pre>

11.1.8. [JDBC-TX-1] Une modification du niveau d'isolation par défaut **doit** être documentée

Pour information, la modification du niveau d'isolation (TRANSACTION_READ_COMMITTED par défaut pour Oracle) peut être réalisée de plusieurs façons :

- dans le code via la méthode `setTransactionIsolation` ;
- dans la configuration de la `DataSource`. Par exemple sous JBoss, via le paramètre `<transaction-isolation>`.

11.1.9. [JDBC-TX-2] Pour des mises à jour groupées, **utiliser** des *batch update*

Les *batch updates* réduisent le nombre d'appels JDBC, et améliorent d'autant les performances.

Exception : pour certains *drivers* JDBC (comme par exemple celui fourni avec Oracle 9i), une dégradation des performances peut être constatée au-delà d'un certain nombre (~50) de *statements*. Un test de performance est préconisé pour évaluer la pertinence de volumineux *batch updates*. La règle reste fortement recommandée pour les *batch updates* inférieurs à ce seuil.

Exemple :

```
Connection connection = ... ;
Statement statement = ... ;
try
{
    if (connection.getMetaData().supportsBatchUpdates())
    {
        connection.setAutoCommit(false);
        statement.clearBatch(); //on supprime les anciens batch
        /* Ajout des commandes au statement */
        statement.addBatch("INSERT ...");
        statement.addBatch("UPDATE ...");
        statement.addBatch("...");

        /* Voir les différents types de retour possibles */
        int[] resultat = statement.executeBatch();
        if(!ExecutionEstSatisfaisante)
        {
            connection.commit();
        }
        else
        {
            connection.rollback();
        }
    }
}
catch (BatchUpdateException bue)
{
    int[] commandesExecutees = bue.getUpdateCounts();
    System.out.println(commandesExecutees.length+
        " commandes ont déjà été exécutées avant l'erreur ");
    connection.rollback();
}
catch (SQLException sqle)
{
    try
    {
        connection.rollback();
    }
    catch (Exception e)
    {
        /* traitement de l'exception */
    }
}
```

```
    }  
}  
finally  
{  
    /* Si l'on ne remet pas AutoCommit à true, les prochaines connexions ne  
     * seront pas en auto-commit.  
     */  
    try  
    {  
        connection.setAutoCommit(true);  
    }  
    catch (Exception e)  
    {  
        /* traitement de l'exception */  
    }  
  
    /* Fermeture des ressources (Statement et Connection) */  
    try  
    {  
        statement.close();  
    }  
    catch (SQLException e)  
    {  
        /* traitement de l'exception */  
    }  
    try  
    {  
        connection.close();  
    }  
    catch (SQLException e)  
    {  
        /* traitement de l'exception */  
    }  
}
```

11.2. Règles JSP/Servlet

11.2.1. Finalité, critères de sélection, sources et contraintes

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[JSP-JAVA]	Par défaut, les règles Java s'appliquent aux JSP.			
[JSP-NOM]	Les noms des fichiers JSP commencent par une minuscule.			
[JSP-ORG]	Les pages JSP doivent être utilisées.			
[JSP-ENC]	L'encodage des fichiers JSP et Servlet doit être UTF-8.			
[JSP-COM]	Les commentaires doivent être écrit sous forme JSP et non HTML.	SonarQube Web plugin	Web:AvoidHtmlCommentCheck	Majeure
[JSP-CODE-1]	Aucune instruction SQL ne doit apparaître dans la couche de présentation (ni dans une page JSP, ni dans un taglib)	SonarQube Web plugin	Web:LibraryDependencyCheck Web:IllegalTagLibsCheck	Critique
[JSP-CODE-2]	Aucun code applicatif Java (aucun scriptlet) ne doit apparaître dans une page JSP.	SonarQube Web plugin	Web:JspScriptletCheck	Majeure
[JSP-EXC]	Chaque page JSP doit utiliser une page d'erreur.			
[JSP-LIBEL]	La déclaration des libellés utilisés dans une page JSP ou JSF ne doit pas être inscrit en dur dans la page ou le fragment de page.	SonarQube Web plugin	Web:InternationalizationCheck	Majeure
[HTTPSESS-1]	Désactiver la création de HttpSession dans les JSP si celle-ci n'est pas utilisée.			
[HTTPSESS-2]	La taille de la session doit rester raisonnable.			
[HTTPSESS-3]	Implémenter Serializable pour toutes les instances présentes dans la session HTTP.	SonarQube	S2441	Bloquante

11.2.2. [JSP-JAVA] Par défaut, les règles Java s'appliquent aux JSP

Par défaut, les règles Java **s'appliquent** aux JSP.

Exemple d'application : à l'intérieur d'un fichier JSP, la règle de base est d'utiliser les règles d'indentation HTML sur la partie HTML du code et d'utiliser les conventions Java pour la partie Java.

Les exceptions à cette règle [JSP-JAVA] sont précisées par la suite (par exemple : [JSP-NOM]).

11.2.3. [JSP-NOM] Les noms des fichiers JSP commencent par une minuscule

Il s'agit d'une redéfinition de [NOM-5](les noms des fichiers Java commencent par une majuscule) pour les JSP: les noms des fichiers **doivent** commencer par une minuscule.

Mais, comme pour les noms des fichiers Java [NOM-5], les caractères minuscules et majuscules sont ensuite alternés : la première lettre de chaque champ sémantique (hormis le premier) le composant doit être en majuscule, et le reste en minuscules. L'usage de caractères non alphabétiques est interdit. L'usage de chiffres est toléré quand il s'agit d'un besoin métier. Aucun chiffre ne pourra cependant être utilisé comme préfixe, mais plutôt comme suffixe.

De même, à moins que la classe désigne une abréviation très courante (ex. URL), on évitera les acronymes.

A noter également que le caractère *underscore* '_' est **interdit** dans les noms de fichiers, tout comme les caractères accentués et les caractères spéciaux.

Les noms de fichiers JSP **doivent** ainsi respecter l'expression régulière : `^[a-z][a-zA-Z]*[0-9]*$`

11.2.4. [JSP-ORG] Les pages JSP **doivent** être utilisées

Les pages JSP **doivent** donc être utilisées, ou supprimées.

11.2.5. [JSP-ENC] L'encodage des fichiers JSP et Servlets **doit** être UTF-8.

Les navigateurs Internet devraient être capables d'interpréter les deux en-têtes suivantes :

Contre-exemples :

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
```

Dans la pratique, tous les navigateurs ne respectent pas correctement ces instructions.

La solution suivante, implémentée côté serveur mais indépendante de ce dernier, sera préférée :

Exemple de fichier WEB-INF/web.xml en utilisant un filtre de servlet :

```
<filter>
  <filter-name>Encoding Filter</filter-name>
  <filter-class>EncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Encoding Filter</filter-name>
  <url-pattern>/*</url-pattern>
  <!-- ou pour une servlet nommée :
  <servlet-name>MyServlet</servlet-name> -->
</filter-mapping>
```

Exemple de code pour la classe qui gère l'encodage :

```
import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class EncodingFilter implements javax.servlet.Filter
{
    private String encoding = "UTF-8" ;

    public void init(FilterConfig filterConfig) throws ServletException
    {
        this.encoding = filterConfig.getInitParameter("encoding");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain filterChain)
        throws IOException, ServletException
    {
        request.setCharacterEncoding(encoding);
        filterChain.doFilter(request, response);
    }

    public void destroy()
    {
    }
}
```

11.2.6. [JSP-COM] Les commentaires **doivent** être écrits sous forme JSP et non HTML

Les finalités de cette règle sont les suivantes :

- ne pas montrer à l'utilisateur les commentaires lorsque ce dernier demande l'affichage du code source depuis son navigateur ;
- cette technique permet également de limiter le flux sur le réseau.

Contre-exemple (commentaire HTML) :	Exemple correct (commentaire JSP) :
<!-- commentaire [<%= expression %>] -->	<%-- commentaire --%>

Exception : cette règle ne s'applique pas si la page JSP est écrit en xhtml, car dans ce cas les commentaires doivent être entourés d'un commentaire xml du type <!-- -->.

11.2.7. Règles de programmation des JSP

11.2.7.1. [JSP-CODE-1] Aucune instruction SQL ne doit apparaître dans la couche de présentation

Tout code SQL est **interdit** dans les pages JSP, et dans les *taglibs*.

En effet, la JSP fait partie de la couche présentation : elle ne doit être utilisée que pour afficher des formulaires de saisie ou des vues. Si un accès à la base SQL doit être effectué ou si une règle de gestion doit être appliquée, il faut impérativement respecter la charte d'architecture, utiliser les couches applicatives adéquates autres que la couche présentation et respecter à minima le découpage Modèle Vue Contrôleur.

11.2.7.2. [JSP-CODE-2] Aucun code applicatif Java (aucun *scriptlet*) ne doit apparaître dans une page JSP

Tout code Java est **fortement déconseillé** dans les pages JSP. En effet :

- les *scriptlets* ne sont pas réutilisables ;
- les *scriptlets* encouragent le « copier-coller » ;
- les *scriptlets* rendent difficiles la lecture et la maintenance des pages JSP ;
- les *scriptlets* sont difficiles à déboguer et à tester.

Le code Java de la couche de présentation devra faire l'objet d'implémentation de *taglibs*.

Contre-exemple :	Exemple correct :
<pre><html> <body> <table> <% for (int i = 0; i < employees.length; i++) { %> out.println("<TD>" + employees.getName(i) + "</TD>"); out.println("<TD>" + employees.getProfile(i) + "</TD>"); <% } %> </table> </body> </html></pre>	<pre><%@ taglib uri="/WEB-INF/jcs.tld" prefix="jcs" %> <html> <body> <table> <jcs:employeeelist id="employee_list"> <tr> <td><jcs:employee attribute="name"/></td> <td><jcs:employee attribute="profile"/></td> </tr> </jcs:employeeelist> </table> </body> </html></pre>

11.2.8. [JSP-EXC] Chaque page JSP doit utiliser une page d'erreur.

Voici ci-dessous un exemple de fichier WEB-INF/web.xml définissant une page d'erreur par défaut pour les JSP et Servlets :

```
...
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/erreur.jsp</location>
</error-page>
...
```

En cas de besoin d'une page d'erreur différente pour une JSP spécifique, il est possible de surcharger le fichier web.xml par l'utilisation de l'attribut `errorPage` au début de la page JSP :

```
<%@ page errorPage="/autre_page_erreur.jsp" contentType="text/html" %>
```


11.2.9. **[JSP-LIBEL]** La déclaration des libellés utilisés dans une page JSP ou JSF ne doit pas être inscrite en dur dans la page ou le fragment de page.

En effet, utiliser des libellés en dur, les rendent très dépendants de l'encodage de l'élément dans lequel ils se trouvent.

Il est conseillé de récupérer les libellés via un fichier de propriétés, un fichier XML, un mappage avec une classe en mémoire ou tout autre moyen dynamique.

Exemple 1 : utilisation d'un bundle Struts (fichier de propriétés) en utilisant la directive `<bean:message>` :

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<bean:message key="cle.dans.fichier.proprietes"/>
Le bundle doit alors être déclaré dans le descripteur Struts sous la forme :
<message-resources null="false" parameter="NomFichierProprietes"/>
```

Exemple 2 : utilisation de la directive JSTL `<fmt:bundle>` pour charger un bundle :

```
<fmt:bundle basename="package.NomFichierProprietes"/>
<fmt:message key="msg.error.key1"/>
```

Exemple 3 : utilisation d'une propriété d'un objet en mémoire :

```
${requestScope.monObjet.maPropriete} ou ${sessionScope.monObjet.maPropriete}
```

Contre-Exemple :

```
<a href="javascript:submitAction(document.${formName},
'rechercheDetaillee.do');void(0);" class="Bouton_Libelle2" onmousedown=""
onmouseup="">Recherche détaillée </a>
```

11.2.10. Règles de gestion pour la session HTTP

11.2.10.1. **[HTTPSESS-1]** Désactiver la création de `HTTPSession` dans les JSP si celle-ci n'est pas utilisée.

Par défaut, toutes les pages JSP créent une session si elle n'existe pas déjà. **Désactiver** la création de celle-ci quand elle n'est pas utilisée permet d'éviter un surcoût de sérialisation, et améliore ainsi les performances.

Exemple (à placer en haut de la page JSP n'utilisant pas de session) :

```
<%@ page session="false" %>
```

11.2.10.2. **[HTTPSESS-2]** La taille de la session doit rester raisonnable.

La gestion de la session HTTP est un élément fondamental pour les performances et la scalabilité des applicatifs développés dans une architecture J2EE.

Bien qu'il soit intéressant de maintenir en session des objets souvent accédés, la taille de la session doit rester raisonnable, c'est-à-dire inférieure à 100Ko.

11.2.10.3.[HTTPSESS-3] Implémenter `Serializable` pour toutes les instances présentes dans la session HTTP.

Les instances placées dans la session **doivent** être toutes sérialisables, car les serveurs d'applications J2EE ne gardent en mémoire qu'un nombre limité de contextes. S'il y en a plus, ils sérialisent les plus vieux sur disque et les remontent lorsque cela est nécessaire. Une instance non sérialisée disparaît lors du rechargement de la session, pouvant entraîner des erreurs difficiles à reproduire.

11.3. Règles EJB

11.3.1. Finalité, critères de sélection, sources et contraintes

Les règles suivantes constituent un rappel de la spécification des EJB. Elles ont pour but principal d'assurer la portabilité et scalabilité des composants développés.

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[EJB-NOM]	Les noms des EJB contiennent le suffixe Bean .	PMD	MDBAndSessionBeanNamingConvention	Mineure
[EJB-SPEC]	Tous les champs statiques des EJB doivent être déclarés final .	PMD	StaticEJBFieldShouldBeFinal	Critique

11.3.2. [EJB-NOM] Les noms des EJB contiennent le suffixe **Bean**.

Une classe qui étend **SessionBean** ou **MessageDrivenBean** **doit** avoir un nom suffixé par **Bean**, comme précisé dans la spécification EJB.

11.3.3. [EJB-SPEC] Tous les champs statiques des EJB doivent être déclarés **final**.

En effet, selon la spécification^[5b] (p.494), un EJB ne peut utiliser de champs statiques en lecture/écriture. Par contre, les champs statiques en lecture seule sont autorisés. Pour cette raison, il est **fortement recommandé** que tous les champs statiques d'une classe EJB soient déclarés **final**.

Cette règle permet d'assurer une sémantique d'exécution consistante, notamment quand les instances sont distribuées par le conteneur EJB sur plusieurs JVM.

11.3.4. Règles EJB Session

11.3.4.1. Règles de nommage des interfaces des EJB Session

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[EJB-SESS-NOM-1]	Le nom de l'interface Remote Home d'un EJB Session doit être suffixé par Home .	PMD	RemoteSessionInterfaceNamingConvention	Mineure
[EJB-SESS-NOM-2]	Le nom de l'interface locale d'un EJB Session doit être suffixé par Local .	PMD	LocalInterfaceSessionNamingConvention	Mineure
[EJB-SESS-NOM-3]	Le nom de l'interface locale Home d'un EJB Session doit être suffixé par LocalHome .	PMD	LocalHomeNamingConvention	Mineure
[EJB-SESS-NOM-4]	Le nom de l'interface Remote d'un EJB Session ne doit pas être suffixé.	PMD	RemoteInterfaceNamingConvention	Mineure

Exemple :

```

/* Classe d'implémentation de l'EJB Session (extends javax.ejb.SessionBean) */
FacticeBean.java

/* Interface Home ou Remote Home de l'EJB Session (extends javax.ejb.EJBHome) */
FacticeHome.java

/* Interface locale de l'EJB Session (extends javax.ejb.EJBLocalObject) */
FacticeLocal.java

/* Interface locale Home de l'EJB Session (extends javax.ejb.EJBLocalHome) */
FacticeLocalHome.java

/* Interface Remote de l'EJB Session (extends javax.ejb.EJBObject) */
Factice.java

```

11.3.4.2. Règles sur les passages de paramètres des EJB Session

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[EJB-SESS-THIS-1]	<u>Ne pas</u> passer la référence this comme paramètre d'une méthode ou constructeur d'un EJB Session.			
[EJB-SESS-THIS-2]	<u>Ne pas</u> donner la référence this en retour d'une méthode d'un EJB Session.			

Plutôt que **this**, il **faudra** utiliser le résultat de la méthode `getEJBObject()`, accessible depuis `SessionContext`.

11.3.4.3. Règles d'implémentation des EJB Session

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[EJB-SESS-SPEC]	Un EJB Session doit satisfaire les spécifications suivantes. La classe implémentant <code>javax.ejb.SessionBean</code> : <ul style="list-style-type: none"> • doit être public ; • ne doit être ni final ni abstract ; • doit contenir un constructeur public sans paramètre ; • ne doit pas définir de méthode <code>finalize()</code> ; • doit définir une méthode <code>ejbCreate()</code>, qui doit être public, non final, non static, et doit retourner void. 			

11.3.5. Règles d'implémentation des Message-Driven Beans (MDB)

Réf.	Descriptif	Contrôle automatisé		
		Outillage	Réf. règle outillage	Criticité
[MDB-SPEC]	<p>Un MDB doit satisfaire les spécifications suivantes. La classe implémentant <code>javax.ejb.MessageDrivenBean</code> et <code>javax.jms.MessageListener</code> :</p> <ul style="list-style-type: none">• <u>doit</u> être public ;• <u>ne doit</u> être ni final ni abstract ;• <u>doit</u> contenir un constructeur public sans paramètre ;• <u>ne doit pas</u> définir de méthode <code>finalize()</code> ;• <u>doit</u> définir une méthode <code>ejbCreate()</code>, qui doit être public, non final, non static, sans paramètres, et doit retourner void.			

12. Annexes

12.1. Liste des règles relatives au langage Java contrôlées par SonarQube dans le jeu de règles CoCA 2.3.1

Identifiant SonarQube	Règle
java:S2204	".equals()" should not be used to test the values of "Atomic" classes
java:S2757	"=+" should not be used instead of "+="
java:S1698	"==" and "!=" should not be used when "equals" is overridden
java:S4682	"@CheckForNull" or "@Nullable" should not be used on primitive types
java:S3753	"@Controller" classes that use "@SessionAttributes" must call "setComplete" on their "SessionStatus" objects
java:S5738	"@Deprecated" code marked for removal should never be used
java:S1874	"@Deprecated" code should not be used
java:S2637	"@NonNull" values should not be set to null
java:S1161	"@Override" should be used on overriding and implementing methods
java:S3751	"@RequestMapping" methods should not be "private"
java:S4602	"@SpringBootApplication" and "@ComponentScan" should not be used in the default package
java:S5301	"ActiveMQConnectionFactory" should not be vulnerable to malicious code deserialization
java:S3631	"Arrays.stream" should be used for primitive arrays
java:S2111	"BigDecimal(double)" should not be used
java:S2737	"catch" clauses should do more than rethrow
java:S4925	"Class.forName()" should not load JDBC 4.0+ drivers
java:S2975	"clone" should not be overridden
java:S2157	"Cloneables" should implement "clone"
java:S4087	"close()" calls should not be redundant
java:S1596	"Collections.EMPTY_LIST", "EMPTY_MAP", and "EMPTY_SET" should not be used
java:S2200	"compareTo" results should not be checked for specific values

Identifiant SonarQube	Règle
java:S4351	"compareTo" should not be overloaded
java:S2167	"compareTo" should not return "Integer.MIN_VALUE"
java:S2718	"DateUtils.truncate" from Apache Commons Lang library should not be used
java:S4524	"default" clauses should be last
java:S2308	"deleteOnExit" should not be used
java:S2127	"Double.longBitsToDouble" should not be used for "int"
java:S5261	"else" statements should be clearly matched with an "if"
java:S2864	"entrySet()" should be iterated when both the key and value are needed
java:S3066	"enum" fields should not be publicly mutable
java:S1201	"equals" method overrides should accept "Object" parameters
java:S4454	"equals" method parameters should not be marked "@Nonnull"
java:S2162	"equals" methods should be symmetric and work for subclasses
java:S1206	"equals(Object obj)" and "hashCode()" should be overridden in pairs
java:S1210	"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method
java:S2097	"equals(Object obj)" should test argument type
java:S2221	"Exception" should not be caught when not required by called methods
java:S2060	"Externalizable" classes should have no-arguments constructors
java:S2156	"final" classes should not have "protected" members
java:S2165	"finalize" should not set fields to "null"
java:S1994	"for" loop increment clauses should modify the loops' counters
java:S127	"for" loop stop conditions should be invariant
java:S3067	"getClass" should not be used for synchronization
java:S2116	"hashCode" and "toString" should not be called on array instances
java:S4601	"HttpSecurity" URL patterns should be correctly ordered
java:S2254	"HttpServletRequest.getRequestSessionId()" should not be used
java:S2692	"indexOf" checks should not be for positive numbers

Identifiant SonarQube	Règle
java:S4425	"Integer.toHexString" should not be used to build hexadecimal strings
java:S2142	"InterruptedException" should not be ignored
java:S4348	"iterator" should not return "this"
java:S1849	"Iterator.hasNext()" should not call "Iterator.next()"
java:S2272	"Iterator.next()" methods should throw "NoSuchElementException"
java:S1194	"java.lang.Error" should not be extended
java:S4042	"java.nio.Files#delete" should be preferred
java:S2442	"Lock" objects should not be "synchronized"
java:S2096	"main" should not "throw" anything
java:S3824	"Map.get" and value test should be replaced with single method call
java:S2446	"notifyAll" should be used
java:S2789	"null" should not be used with "Optional"
java:S1696	"NullPointerException" should not be caught
java:S1695	"NullPointerException" should not be explicitly thrown
java:S1174	"Object.finalize()" should remain protected (versus public) when overriding
java:S2274	"Object.wait(...)" and "Condition.await(...)" should be called inside a "while" loop
java:S1844	"Object.wait(...)" should never be called on objects that implement "java.util.concurrent.locks.Condition"
java:S2629	"Preconditions" and logging arguments should not require evaluation
java:S2695	"PreparedStatement" and "ResultSet" methods should be called with valid indices
java:S3398	"private" methods called only by inner classes should be moved to those classes
java:S1444	"public static" fields should be constant
java:S2119	"Random" objects should be reused
java:S2677	"read" and "readLine" return values should be used
java:S4929	"read(byte[],int,int)" should be overridden
java:S2675	"readObject" should not be "synchronized"
java:S2062	"readResolve" methods should be inheritable

Identifiant SonarQube	Règle
java:S2232	"ResultSet.isLast()" should not be used
java:S2151	"runFinalizersOnExit" should not be called
java:S2122	"ScheduledThreadPoolExecutor" should not have 0 core threads
java:S4347	"SecureRandom" seeds should not be predictable
java:S2057	"Serializable" classes should have a "serialVersionUID"
java:S2066	"Serializable" inner classes of non-serializable classes should be "static"
java:S4719	"StandardCharsets" constants should be preferred
java:S3252	"static" base class members should not be accessed via derived types
java:S2209	"static" members should be accessed statically
java:S4034	"Stream" call chains should be simplified when possible
java:S3864	"Stream.peek" should be used with caution
java:S6204	"Stream.toList()" method should be used instead of "collectors" when unmodifiable list needed
java:S3039	"String" calls should not go beyond their bounds
java:S5361	"String#replace" should be preferred to "String#replaceAll"
java:S1317	"StringBuilder" and "StringBuffer" should not be instantiated with a character
java:S1114	"super.finalize()" should be called at the end of "Object.finalize()" implementations
java:S1151	"switch case" clauses should not have too many lines of code
java:S131	"switch" statements should have "default" clauses
java:S1301	"switch" statements should have at least 3 "case" clauses
java:S1219	"switch" statements should not contain non-case labels
java:S1479	"switch" statements should not have too many "case" clauses
java:S1217	"Thread.run()" should not be called directly
java:S2925	"Thread.sleep" should not be used in tests
java:S3014	"ThreadGroup" should not be used
java:S5164	"ThreadLocal" variables should be cleaned up when no longer used
java:S4065	"ThreadLocal.withInitial" should be preferred

Identifiant SonarQube	Règle
java:S2438	"Threads" should not be used where "Runnables" are expected
java:S1130	"throws" declarations should not be superfluous
java:S3020	"toArray" should be passed an array of the proper type
java:S2225	"toString()" and "clone()" methods should not return null
java:S1858	"toString()" should never be called on a String object
java:S2112	"URL.hashCode" and "URL.equals" should be avoided
java:S3078	"volatile" variables should not be used with compound operators
java:S3046	"wait" should not be called when multiple locks are held
java:S2273	"wait", "notify" and "notifyAll" should only be called when a lock is obviously held on an object
java:S2276	"wait(...)" should be used instead of "Thread.sleep(...)" when a lock is held
java:S4349	"write(byte[],int,int)" should be overridden
java:S3042	"writeObject" should not be the only "synchronized" code in a class
java:S5413	'List.remove()' should not be used in ascending 'for' loops
java:S6219	'serialVersionUID' field should not be set to '0L' in records
java:S2251	A "for" loop update clause should move the counter in the right direction
java:S1264	A "while" loop should be used instead of a "for" loop
java:S1109	A close curly brace should be located at the beginning of a line
java:S3973	A conditionally executed single line should be denoted by indentation
java:S1700	A field should not duplicate the name of its containing class
java:S5876	A new session should be created during user authentication
java:S2115	A secure password should be used when connecting to a database
java:S118	Abstract class names should comply with a naming convention
java:S1610	Abstract classes without fields should be converted to interfaces
java:S3038	Abstract methods should not be redundant
java:S5324	Accessing Android external storage is security-sensitive
java:S3923	All branches in a conditional structure should not have exactly the same implementation

Identifiant SonarQube	Règle
java:S3752	Allowing both safe and unsafe HTTP methods is security-sensitive
java:S4434	Allowing deserialization of LDAP objects is security-sensitive
java:S5693	Allowing requests with excessive content length is security-sensitive
java:S5804	Allowing user enumeration is security-sensitive
java:S5850	Alternatives in regular expressions should be grouped when used with anchors
java:S1694	An abstract class should have both abstract and concrete methods
java:S4838	An iteration on a Collection should be performed on the type handled by the Collection
java:S1106	An open curly brace should be located at the beginning of a line
java:S1710	Annotation repetitions should not be wrapped
java:S1604	Anonymous inner classes containing only one method should become lambdas
pmd:AppendCharacterWithChar	Append Character With Char
java:S1195	Array designators "[]" should be located after the type in method signatures
java:S1197	Array designators "[]" should be on the type, not the variable
java:S3012	Arrays should not be copied using loops
java:S3878	Arrays should not be created for varargs parameters
java:S3415	Assertion arguments should be passed in the correct order
java:S5779	Assertion methods should not be used within the try block of a try-catch catching an Error
java:S5845	Assertions comparing incompatible types should not be made
java:S2970	Assertions should be complete
java:S5960	Assertions should not be used in production code
java:S5863	Assertions should not compare an object to itself
java:S5958	AssertJ "assertThatThrownBy" should not be used alone
java:S5841	AssertJ assertions "allMatch" and "doesNotContains" should also test for emptiness
java:S6103	AssertJ assertions with "Consumer" arguments should contain assertion inside consumers
java:S5831	AssertJ configuration should be applied
java:S5833	AssertJ methods setting the assertion context should come before an assertion

Identifiant SonarQube	Règle
java:S4274	Asserts should not be used to check the parameters of a public method
java:S3064	Assignment of lazy-initialized members should be the last step with double-checked locking
java:S1121	Assignments should not be made from within sub-expressions
java:S4165	Assignments should not be redundant
java:S5808	Authorizations should be based on strong decisions
java:S6288	Authorizing non-authenticated users to use keys in the Android KeyStore is security-sensitive
pmd:AvoidAccessibilityAlteration	Avoid Accessibility Alteration
pmd:AvoidArrayLoops	Avoid Array Loops
java:S5411	Avoid using boxed "Boolean" types directly in boolean expressions
pmd:AvoidUsingNativeCode	Avoid Using Native Code
pmd:AvoidUsingVolatile	Avoid Using Volatile
java:S6262	AWS region should not be set with a hardcoded String
java:S6001	Back references in regular expressions should only refer to capturing groups that are matched before the reference
findbugs:NM_CONFUSING	Bad practice - Confusing method names
findbugs:EQ_GETCLASS_AND_CLASS_CONSTANT	Bad practice - equals method fails for subtypes
findbugs:NM_WRONG_PACKAGE_INTENTIONAL	Bad practice - Method doesn't override method in superclass due to wrong package for parameter
findbugs:NM_VERY_CONFUSING_INTENTIONAL	Bad practice - Very confusing method names (but perhaps intentional)
java:S2647	Basic authentication should not be used
java:S2445	Blocks should be synchronized on "private final" fields
java:S1940	Boolean checks should not be inverted
java:S2589	Boolean expressions should not be gratuitous
pmd:BooleanGetMethodName	Boolean Get Method Name

Identifiant SonarQube	Règle
java:S1125	Boolean literals should not be redundant
java:S2153	Boxing and unboxing should not be immediately reversed
java:S5320	Broadcasting intents is security-sensitive
pmd:CallSuperInConstructor	Call Super In Constructor
java:S6068	Call to Mockito method "verify", "when" or "given" should be simplified
java:S1157	Case insensitive string comparisons should be made without intermediate upper or lower casing
java:S5866	Case insensitive Unicode regular expressions should enable the "UNICODE_CASE" flag
java:S2147	Catches should be combined
java:S5838	Chained AssertJ assertions should be simplified to the corresponding dedicated assertion
java:S6397	Character classes in regular expressions should not contain only one character
java:S5869	Character classes in regular expressions should not contain the same character twice
java:S5857	Character classes should be preferred over reluctant quantifiers in regular expressions
pmd:CheckResultSet	Check ResultSet
java:S2387	Child class fields should not shadow parent class fields
java:S2177	Child class methods named for parent class methods should be overrides
java:S5547	Cipher algorithms should be robust
java:S3329	Cipher Block Chaining IVs should be unpredictable
java:S5803	Class members annotated with "@VisibleForTesting" should not be accessed from production code
java:S101	Class names should comply with a naming convention
java:S2176	Class names should not shadow interfaces or superclasses
java:S1104	Class variable fields should not have public accessibility
java:S1258	Classes and enums with private members should have a constructor
java:S2134	Classes extending java.lang.Thread should override the "run" method
java:S1191	Classes from "sun.*" packages should not be used
java:S2166	Classes named like "Exception" should extend "Exception" or a subclass

Identifiant SonarQube	Règle
java:S2390	Classes should not access their own subclasses during initialization
java:S1872	Classes should not be compared by name
java:S1200	Classes should not be coupled to too many other classes (Single Responsibility Principle)
java:S2094	Classes should not be empty
java:S1448	Classes should not have too many methods
java:S1182	Classes that override "clone" should be "Cloneable" and call "super.clone()"
java:S2440	Classes with only "static" methods should not be instantiated
java:S2974	Classes without "public" constructors should be "final"
java:S3776	Cognitive Complexity of methods should not be too high
java:S1066	Collapsible "if" statements should be merged
java:S5329	Collection constructors should not be used as java.util.function.Function
java:S2250	Collection methods with O(n) performance should be used carefully
java:S3981	Collection sizes and array length comparisons should make sense
java:S1155	Collection.isEmpty() should be used to test for emptiness
java:S2114	Collections should not be passed as arguments to their own methods
java:S6208	Comma-separated labels should be used in Switch with colon case
java:S4488	Composed "@RequestMapping" variants should be preferred
java:S2583	Conditionally executed code should be reachable
java:S3972	Conditionals should start on new lines
java:S4792	Configuring loggers is security-sensitive
java:S5853	Consecutive AssertJ "assertThat" statements should be chained
pmd:ConsecutiveLiteralAppends	Consecutive Literal Appends
java:S115	Constant names should comply with a naming convention
java:S1214	Constants should not be defined in interfaces
java:S5993	Constructors of an "abstract" class should not be declared "public"
java:S2129	Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and

Identifiant SonarQube	Règle
	primitive-wrapper classes
java:S1699	Constructors should only call non-overridable methods
java:S3959	Consumed Stream pipelines should not be reused
java:S6244	Consumer Builders should be used
java:S134	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply
java:S121	Control structures should use curly braces
findbugs:EC_UNRELATED_INTERFACES	Correctness - Call to equals() comparing different interface types
findbugs:EC_UNRELATED_TYPES	Correctness - Call to equals() comparing different types
findbugs:EC_UNRELATED_CLASS_AND_INTERFACE	Correctness - Call to equals() comparing unrelated class and interface
findbugs:SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH	Correctness - Dead store due to switch statement fall through
findbugs:SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH_TO_THROW	Correctness - Dead store due to switch statement fall through to throw
findbugs:EQ_ALWAYS_FALSE	Correctness - equals method always returns false
findbugs:EQ_ALWAYS_TRUE	Correctness - equals method always returns true
fb-contrib:PMB_INSTANCE_BASED_THREAD_LOCAL	Correctness - Field is an instance based ThreadLocal variable
findbugs:ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND	Correctness - int value cast to float and then passed to Math.round
findbugs:ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	Correctness - Integral value cast to double and then passed to Math.ceil
findbugs:NM_WRONG_PACKAGE	Correctness - Method doesn't override method in superclass due to wrong package for parameter

Identifiant SonarQube	Règle
java:S6432	Counter Mode initialization vectors should not be reused
java:S3330	Creating cookies without the "HttpOnly" flag is security-sensitive
java:S2092	Creating cookies without the "secure" flag is security-sensitive
java:S6242	Credentials Provider should be set explicitly when creating a new "AwsClient"
java:S6437	Credentials should not be hard-coded
java:S4426	Cryptographic keys should be robust
java:S2061	Custom serialization method signatures should meet requirements
java:S5917	DateTimeFormatters should not use mismatched year and week numbers
java:S1319	Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"
java:S4507	Delivering code in production with debug features activated is security-sensitive
java:S6355	Deprecated annotations should include explanations
java:S1133	Deprecated code should be removed
java:S1123	Deprecated elements should have both the annotation and the Javadoc tag
java:S4970	Derived exceptions should not hide their parents' catch blocks
java:S5247	Disabling auto-escaping in template engines is security-sensitive
java:S4502	Disabling CSRF protections is security-sensitive
java:S5689	Disclosing fingerprints from web application technologies is security-sensitive
java:S2154	Dissimilar primitive wrappers should not be used with the ternary operator without explicit casting
pmd:DoNotUseThreads	Do Not Use Threads
java:S3599	Double Brace Initialization should not be used
java:S2168	Double-checked locking should not be used
java:S1168	Empty arrays and collections should be returned instead of null
java:S5846	Empty lines should not be tested with regex MULTILINE flag
java:S1116	Empty statements should be removed
java:S6363	Enabling file access for WebViews is security-sensitive

Identifiant SonarQube	Règle
java:S6362	Enabling JavaScript support for WebViews is security-sensitive
java:S5542	Encryption algorithms should be used with secure mode and padding scheme
java:S1150	Enumeration should not be implemented
java:S6218	Equals method should be overridden in records containing array fields
java:S5665	Escape sequences should not be used in text blocks
java:S1165	Exception classes should be immutable
java:S1166	Exception handlers should preserve the original exceptions
java:S5777	Exception testing via JUnit <code>@Test</code> annotation should be avoided
java:S5776	Exception testing via JUnit ExpectedException rule should not be mixed with other assertions
java:S1193	Exception types should not be tested using "instanceof" in catch blocks
java:S2139	Exceptions should be either logged or rethrown but not both
java:S3984	Exceptions should not be created without being thrown
java:S1989	Exceptions should not be thrown from servlet methods
java:S1163	Exceptions should not be thrown in finally blocks
java:S1215	Execution of the Garbage Collector should be triggered only by the JVM
java:S1147	Exit methods should not be called
java:S5042	Expanding archive files without controlling resource consumption is security-sensitive
java:S3346	Expressions used in "assert" should not produce side effects
java:S3305	Factory method injection should be used in "@Configuration" classes
java:S116	Field names should comply with a naming convention
java:S1948	Fields in a "Serializable" class should either be transient or serializable
java:S2065	Fields in non-serializable classes should not be "transient"
java:S2689	Files opened in append mode should not be used with ObjectOutputStream
java:S1244	Floating point numbers should not be tested for equality
java:S2077	Formatting SQL queries is security-sensitive
java:S4276	Functional Interfaces should be as specialised as possible

Identifiant SonarQube	Règle
java:S1190	Future keywords should not be used as names
java:S112	Generic exceptions should never be thrown
java:S1452	Generic wildcard types should not be used in return types
java:S4275	Getters and setters should access the expected fields
java:S2886	Getters and setters should be synchronized in pairs
java:S2068	Hard-coded passwords are security-sensitive
java:S6418	Hard-coded secrets are security-sensitive
java:S2053	Hashes should include an unpredictable salt
java:S5122	Having a permissive Cross-Origin Resource Sharing policy is security-sensitive
java:S1764	Identical expressions should not be used on both sides of a binary operator
java:S2235	IllegalMonitorStateException should not be caught
pmd:ImmutableField	Immutable Field
java:S2175	Inappropriate "Collection" calls should not be made
java:S2639	Inappropriate regular expressions should not be used
pmd:InefficientStringBuffering	Inefficient String Buffering
java:S110	Inheritance tree of classes should not be too deep
java:S2388	Inner class calls to super class methods should be unambiguous
java:S4517	InputStream.read() implementation should not return a signed byte
java:S5445	Insecure temporary file creation methods should not be used
java:S2696	Instance methods should not write to "static" fields
pmd:InsufficientStringBufferDeclaration	Insufficient String Buffer Declaration
java:S114	Interface names should comply with a naming convention
java:S3958	Intermediate Stream methods should not be left unused
java:S2183	Ints and longs should not be shifted by zero or more than their number of bits-1
java:S2110	Invalid "Date" values should not be used

Identifiant SonarQube	Règle
java:S4738	Java features should be preferred to Guava
java:S3626	Jump statements should not be redundant
java:S1143	Jump statements should not occur in "finally" blocks
java:S2186	JUnit assertions should not be used in "run" methods
java:S5785	JUnit assertTrue/assertFalse should be simplified to the corresponding dedicated assertion
java:S2924	JUnit rules should be used
java:S2188	JUnit test cases should call super methods
java:S1607	JUnit4 @Ignored and JUnit5 @Disabled annotations should be used to disable tests and should provide a rationale
java:S5790	JUnit5 inner test classes should be annotated with @Nested
java:S5786	JUnit5 test classes and methods should have default package visibility
java:S5810	JUnit5 test classes and methods should not be silently ignored
java:S5659	JWT should be signed and verified with strong cipher algorithms
java:S1119	Labels should not be used
java:S1602	Lambdas containing only one statement should not nest this statement in a block
java:S1612	Lambdas should be replaced with method references
java:S6246	Lambdas should not invoke other lambdas synchronously
java:S4433	LDAP connections should be authenticated
java:S103	Lines should not be too long
pmd:LocalHomeNamingConvention	Local Home Naming Convention
pmd:LocalInterfaceSessionNamingConvention	Local Interface Session Naming Convention
java:S117	Local variable and method parameter names should comply with a naming convention
java:S1488	Local variables should not be declared and then immediately returned or thrown
java:S1117	Local variables should not shadow class fields
java:S2222	Locks should be released on all paths
java:S3416	Loggers should be named for their enclosing classes

java:S2252	Loop conditions should be true at least once
java:S2189	Loops should not be infinite
java:S135	Loops should not contain more than a single "break" or "continue" statement
java:S1751	Loops with at most one iteration should be refactored
java:S109	Magic numbers should not be used
findbugs:MS_PKGPROTECT	Malicious code - Field should be package protected
java:S6104	Map "computeIfAbsent()" and "computeIfPresent()" should not be used to add "null" values.
java:S4143	Map values should not be replaced unconditionally
java:S1640	Maps with keys that are enum values should be replaced with EnumMap
java:S2184	Math operands should be cast before assignment
java:S6209	Members ignored during record serialization should not be used
pmd:MDBAndSessionBeanNamingConvention	Message Driven Bean And Session Bean Naming Convention
java:S100	Method names should comply with a naming convention
java:S2638	Method overrides should not change contracts
java:S1226	Method parameters, caught exceptions and foreach variables' initial values should not be ignored
java:S2236	Methods "wait(...)", "notify()" and "notifyAll()" should not be called on Thread instances
java:S1845	Methods and field names should not be the same or differ only by capitalization
java:S2140	Methods of "Random" that return floating point values should not be used in random integer generation
java:S3516	Methods returns should not be invariant
java:S5826	Methods setUp() and tearDown() should be correctly annotated starting with JUnit4
java:S1186	Methods should not be empty
java:S1221	Methods should not be named "toString", "hashCode" or "equal"
java:S1541	Methods should not be too complex

Identifiant SonarQube	Règle
java:S2229	Methods should not call same-class methods with incompatible "@Transactional" values
java:S4144	Methods should not have identical implementations
java:S138	Methods should not have too many lines
java:S107	Methods should not have too many parameters
java:S1142	Methods should not have too many return statements
java:S3400	Methods should not return constants
java:S3065	Min and max used in combination should not always return the same value
pmd:MissingStaticMethodInNonInstantiatableClass	Missing Static Method In Non Instantiatable Class
java:S6301	Mobile database encryption keys should not be disclosed
java:S5969	Mocking all non-private methods of a class should be avoided
java:S1124	Modifiers should be declared in the correct order
findbugs:STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE	Multi-threading - Call to static DateFormat
findbugs:DC_DOUBLECHECK	Multi-threading - Possible double-check of field
findbugs:STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE	Multi-threading - Static DateFormat
java:S2681	Multiline blocks should be enclosed in curly braces
java:S1659	Multiple variables should not be declared on the same line
java:S2386	Mutable fields should not be "public static"
java:S5860	Names of regular expressions named groups should be used
java:S2676	Neither "Math.abs" nor negation should be used on numbers that could be "MIN_VALUE"
java:S2786	Nested "enum"s should not be declared static
java:S108	Nested blocks of code should not be left empty
java:S1199	Nested code blocks should not be used

Identifiant SonarQube	Règle
java:S6395	Non-capturing groups without quantifier should not be used
java:S1223	Non-constructor methods should not have the same name as the enclosing class
java:S3077	Non-primitive fields should not be "volatile"
java:S2230	Non-public methods should not be "@Transactional"
java:S2118	Non-serializable classes should not be written
java:S2441	Non-serializable objects should not be stored in "HttpSession" objects
java:S2885	Non-thread-safe fields should not be static
pmd:NullAssignment	Null Assignment
java:S4201	Null checks should not be used with "instanceof"
java:S2259	Null pointers should not be dereferenced
java:S2447	Null should not be returned from a "Boolean" method
java:S4449	Nullness of parameters should be guaranteed
java:S2133	Objects should not be created only to "getClass"
java:S5783	Only one method invocation is expected when testing checked exceptions
java:S5778	Only one method invocation is expected when testing runtime exceptions
java:S1171	Only static class initializers should be used
java:S5679	OpenSAML2 should be configured to prevent authentication bypass
java:S6202	Operator "instanceof" should be used instead of "A.class.isInstance()"
pmd:OptimizableToArrayCall	Optimizable To Array Call
java:S3655	Optional value should only be accessed after calling isPresent()
java:S3551	Overrides should match their parent class methods in synchronization
java:S1185	Overriding methods should do more than simply call the same method in the super class
java:S1598	Package declaration should match source file directory
java:S120	Package names should comply with a naming convention
java:S4032	Packages containing only "package-info.java" should be removed
java:S2234	Parameters should be passed in the correct order

Identifiant SonarQube	Règle
java:S1611	Parentheses should be removed from a single lambda input parameter when its type is inferred
java:S2130	Parsing should be used to convert "Strings" to primitives
java:S5344	Passwords should not be stored in plain-text or with a fast hashing algorithm
java:S6201	Pattern Matching for "instanceof" operator should be used instead of simple "instanceof" + cast
findbugs:DM_FP_NUMBER_CT OR	Performance - Method invokes inefficient floating-point Number constructor; use static valueOf instead
java:S6217	Permitted types of a sealed class should be omitted if they are declared in the same file
java:S4684	Persistent entities should not be used as arguments of "@RequestMapping" methods
java:S1158	Primitive wrappers should not be instantiated only for "toString" or "compareTo" calls
java:S3457	Printf-style format strings should be used correctly
java:S2275	Printf-style format strings should not lead to unexpected behavior at runtime
java:S1450	Private fields only used as local variables in methods should become local variables
java:S1170	Public constants and fields initialized at declaration should be "static final" rather than merely "final"
java:S1176	Public types, methods and fields (API) should be documented with Javadoc
java:S3034	Raw byte values should not be used in bitwise operations in combination with shifts
java:S3740	Raw types should not be used
java:S5322	Receiving intents is security-sensitive
java:S6206	Records should be used instead of ordinary classes when representing immutable data structure
java:S1905	Redundant casts should not be used
java:S6207	Redundant constructors/methods should be avoided in records
java:S1110	Redundant pairs of parentheses should be removed
java:S2109	Reflection should not be used to check non-runtime annotations
java:S3011	Reflection should not be used to increase accessibility of classes, methods, or fields
java:S6216	Reflection should not be used to increase accessibility of records' fields
java:S5855	Regex alternatives should not be redundant

Identifiant SonarQube	Règle
java:S5996	Regex boundaries should not be used in a way that can never be matched
java:S6002	Regex lookahead assertions should not be contradictory
java:S5994	Regex patterns following a possessive quantifier should not always fail
java:S5854	Regexes containing characters subject to normalization should use the CANON_EQ flag
java:S6241	Region should be set explicitly when creating a new "AwsClient"
java:S6353	Regular expression quantifiers and character classes should be used concisely
java:S5856	Regular expressions should be syntactically valid
java:S5843	Regular expressions should not be too complicated
java:S6331	Regular expressions should not contain empty groups
java:S6326	Regular expressions should not contain multiple spaces
java:S5998	Regular expressions should not overflow the stack
java:S1862	Related "if/else if" statements should not have the same condition
java:S6019	Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string
pmd:RemoteInterfaceNamingConvention	Remote Interface Naming Convention
pmd:RemoteSessionInterfaceNamingConvention	Remote Session Interface Naming Convention
java:S5842	Repeated patterns in regular expressions should not match the empty string
java:S2095	Resources should be closed
java:S6213	Restricted Identifiers should not be used as Identifiers
java:S1126	Return of boolean expressions should not be wrapped into an "if-then-else" statement
java:S2201	Return values from functions without side effects should not be ignored
java:S899	Return values should not be ignored when they contain the operation status code
java:S6243	Reusable resources should be initialized at construction time of Lambda functions
java:S4036	Searching OS commands in PATH is security-sensitive
java:S125	Sections of code should not be commented out
findbugs:SQL_PREPARED_STATEMENT	Security - A prepared statement is generated from a nonconstant String

Identifiant SonarQube	Règle
LEMENT_GENERATED_FROM _NONCONSTANT_STRING	
findbugs:PT_ABSOLUTE_PATH _TRAVERSAL	Security - Absolute path traversal in servlet
findsecbugs:BAD_HEXA_CONV ERSION	Security - Bad hexadecimal concatenation
findsecbugs:BLOWFISH_KEY_SI ZE	Security - Blowfish usage with short key
findsecbugs:PADDING_ORACLE	Security - Cipher is susceptible to Padding Oracle
findsecbugs:CIPHER_INTEGRIT Y	Security - Cipher with no integrity
findsecbugs:HTTPONLY_COOKI E	Security - Cookie without the HttpOnly flag
findsecbugs:INSECURE_COOKI E	Security - Cookie without the secure flag
findsecbugs:DES_USAGE	Security - DES is insecure
findsecbugs:ECB_MODE	Security - ECB mode is insecure
findbugs:DMI_EMPTY_DB_PAS SWORD	Security - Empty database password
findsecbugs:WEAK_FILENAME UTILS	Security - FilenameUtils not filtering null bytes
findsecbugs:HAZELCAST_SYM METRIC_ENCRYPTION	Security - Hazelcast symmetric encryption
findsecbugs:WEAK_HOSTNAME _VERIFIER	Security - HostnameVerifier that accept any signed certificates
findbugs:HRS_REQUEST_PARA METER_TO_COOKIE	Security - HTTP cookie formed from untrusted input
findbugs:HRS_REQUEST_PARA METER_TO_HTTP_HEADER	Security - HTTP Response splitting vulnerability
findsecbugs:CUSTOM_MESSAG E_DIGEST	Security - Message digest is custom
findbugs:SQL_NONCONSTANT_	Security - Nonconstant string passed to execute or addBatch method on an SQL statement

Identifiant SonarQube	Règle
STRING_PASSED_TO_EXECUTE	
findseccbugs:NULL_CIPHER	Security - NullCipher is insecure
findseccbugs:SQL_INJECTION_JDBC	Security - Potential JDBC Injection
findseccbugs:SQL_INJECTION_SPRING_JDBC	Security - Potential JDBC Injection (Spring JDBC)
findseccbugs:SQL_INJECTION_HIBERNATE	Security - Potential SQL/HQL Injection (Hibernate)
findseccbugs:SQL_INJECTION_JDO	Security - Potential SQL/JDOQL Injection (JDO)
findseccbugs:SQL_INJECTION_JPA	Security - Potential SQL/JPQL Injection (JPA)
findbugs:PT_RELATIVE_PATH_TRAVERSAL	Security - Relative path traversal in servlet
findseccbugs:RSA_KEY_SIZE	Security - RSA usage with short key
findseccbugs:RSA_NO_PADDING	Security - RSA with no padding is insecure
findbugs:XSS_REQUEST_PARAMETER_TO_SERVLET_WRITE	Security - Servlet reflected cross site scripting vulnerability
findbugs:XSS_REQUEST_PARAMETER_TO_SEND_ERROR	Security - Servlet reflected cross site scripting vulnerability in error page
findseccbugs:WEAK_MESSAGE_DIGEST_SHA1	Security - SHA-1 is a weak hash function
findseccbugs:STATIC_IV	Security - Static IV
findseccbugs:STRUTS_FORM_VALIDATION	Security - Struts Form without input validation
findseccbugs:WEAK_TRUST_MANAGER	Security - TrustManager that accept any certificates
findseccbugs:SERVLET_HEADER_REFERER	Security - Untrusted Referer header
findseccbugs:SERVLET_SESSION_ID	Security - Untrusted session cookie value

Identifiant SonarQube	Règle
findseccbugs:SERVLET_HEADER_USER_AGENT	Security - Untrusted User-Agent header
findseccbugs:XXE_DOCUMENT	Security - XML parsing vulnerable to XXE (DocumentBuilder)
findseccbugs:XXE_SAXPARSER	Security - XML parsing vulnerable to XXE (SAXParser)
findseccbugs:XXE_XMLREADER	Security - XML parsing vulnerable to XXE (XMLReader)
findseccbugs:XML_DECODER	Security - XMLDecoder usage
findseccbugs:XSS_REQUEST_WRAPPER	Security - XSSRequestWrapper is a weak XSS protection
java:S4830	Server certificates should be verified during SSL/TLS connections
java:S5527	Server hostnames should be verified during SSL/TLS connections
java:S2226	Servlets should not have mutable instance fields
java:S4512	Setting JavaBean properties is security-sensitive
java:S2612	Setting loose POSIX file permissions is security-sensitive
java:S2178	Short-circuit logic should be used in boolean contexts
java:S2437	Silly bit operations should not be performed
java:S2159	Silly equality checks should not be made
java:S2185	Silly math should not be performed
java:S2121	Silly String operations should not be made
java:S5976	Similar tests should be grouped in a single Parameterized test
pmd:SimpleDateFormatNeedsLocale	Simple Date Format Needs Locale
java:S5663	Simple string literal should be used for single line strings
pmd:SimplifyConditional	Simplify Conditional
java:S6035	Single-character alternations in regular expressions should be replaced with character classes
java:S1120	Source code should be indented consistently
java:S106	Standard outputs should not be used directly to log anything

Identifiant SonarQube	Règle
java:S122	Statements should be on separate lines
pmd:StaticEJBFieldShouldBeFinal	Static EJB Field Should Be Final
java:S3010	Static fields should not be updated in constructors
java:S3008	Static non-final field names should comply with a naming convention
pmd:StringInstantiation	String Instantiation
java:S1192	String literals should not be duplicated
java:S6126	String multiline concatenation should be replaced with Text Blocks
java:S4635	String offset-based methods should be preferred for finding substrings from offsets
java:S1153	String.valueOf() should not be appended to a String
java:S4973	Strings and Boxed types should be compared using "equals()"
java:S1132	Strings literals should be placed on the left side when checking for equality
java:S1643	Strings should not be concatenated using '+' in a loop
findbugs:UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	Style - Field not initialized in constructor but dereferenced without null check
java:S2160	Subclasses that add fields should override "equals"
java:S6396	Superfluous curly brace quantifiers should be avoided
java:S6205	Switch arrow labels should not use redundant keywords
java:S128	Switch cases should end with an unconditional "break" statement
java:S1860	Synchronization should not be done on instances of value-based classes
java:S1149	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used
java:S105	Tabulation characters should not be used
java:S3358	Ternary operators should not be nested
java:S3577	Test classes should comply with a naming convention
java:S5961	Test methods should not contain too many assertions
java:S2187	TestCases should contain tests
java:S5967	Tests method should not be annotated with competing annotations

Identifiant SonarQube

Règle

java:S5973	Tests should be stable
java:S2699	Tests should include assertions
java:S6203	Text blocks should not be used in complex expressions
java:S1220	The default unnamed package should not be used
java:S2293	The diamond operator (" \diamond ") should be used
java:S1213	The members of an interface or class declaration should appear in a pre-defined order
java:S2055	The non-serializable super class of a "Serializable" class should have a no-argument constructor
java:S1111	The Object.finalize() method should not be called
java:S1113	The Object.finalize() method should not be overridden
java:S6070	The regex escape sequence \cX should only be used with characters in the @- _ range
java:S1175	The signature of "finalize()" should match that of "Object.finalize()"
java:S1774	The ternary operator should not be used
java:S4968	The upper bound of type variables and wildcards should not be "final"
java:S2674	The value returned from a stream read should be checked
java:S1181	Throwable and Error should not be caught
pmd:TooManyFields	Too Many Fields
pmd:TooManyStaticImports	Too Many Static Imports
java:S1315	Track uses of "CHECKSTYLE:OFF" suppression comments
java:S1134	Track uses of "FIXME" tags
java:S1310	Track uses of "NOPMD" suppression comments
java:NoSonar	Track uses of "NOSONAR" comments
java:S1135	Track uses of "TODO" tags
java:S1141	Try-catch blocks should not be nested
java:S2093	Try-with-resources should be used
java:S1871	Two branches in a conditional structure should not have exactly the same implementation

Identifiant SonarQube	Règle
java:S119	Type parameter names should comply with a naming convention
java:S4977	Type parameters should not shadow other type parameters
java:S2761	Unary prefix operators should not be repeated
java:S5868	Unicode Grapheme Clusters should be avoided inside regex character classes
pmd:UnnecessaryFullyQualifiedName	Unnecessary Fully Qualified Name
java:S1128	Unnecessary imports should be removed
pmd:UnnecessaryReturn	Unnecessary Return
java:S3985	Unused "private" classes should be removed
java:S1068	Unused "private" fields should be removed
java:S1144	Unused "private" methods should be removed
java:S1854	Unused assignments should be removed
java:S1065	Unused labels should be removed
java:S1481	Unused local variables should be removed
java:S1172	Unused method parameters should be removed
java:S2326	Unused type parameters should be removed
java:S1075	URIs should not be hardcoded
pmd:UseIndexOfChar	Use Index Of Char
pmd:UseLocaleWithCaseConversions	Use Locale With Case Conversions
pmd:UseStringBufferForStringAppends	Use String Buffer For String Appends
pmd:UselessOperationOnImmutable	Useless Operation On Immutable
java:S6293	Using biometric authentication without a cryptographic solution is security-sensitive
java:S5332	Using clear-text protocols is security-sensitive
java:S1313	Using hardcoded IP addresses is security-sensitive
java:S6263	Using long-term access keys is security-sensitive

Identifiant SonarQube	Règle
java:S2257	Using non-standard cryptographic algorithms is security-sensitive
java:S2245	Using pseudorandom number generators (PRNGs) is security-sensitive
java:S5443	Using publicly writable directories is security-sensitive
java:S5852	Using slow regular expressions is security-sensitive
java:S6291	Using unencrypted databases in mobile applications is security-sensitive
java:S6300	Using unencrypted files in mobile applications is security-sensitive
java:S4544	Using unsafe Jackson deserialization configuration is security-sensitive
java:S4790	Using weak hashing algorithms is security-sensitive
java:S1118	Utility classes should not have public constructors
java:S3436	Value-based classes should not be used for locking
java:S2123	Values should not be uselessly incremented
java:S5669	Vararg method arguments should not be confusing
java:S1656	Variables should not be self-assigned
java:S4423	Weak SSL/TLS protocols should not be used
java:S3986	Week Year ("YYYY") should not be used for date formatting
java:S2479	Whitespace and control characters in literals should be explicit
java:S5664	Whitespace for text block indent should be consistent
java:S2208	Wildcard imports should not be used
java:S6373	XML parsers should not allow inclusion of arbitrary files
java:S6376	XML parsers should not be vulnerable to Denial of Service attacks
java:S2755	XML parsers should not be vulnerable to XXE attacks
java:S6374	XML parsers should not load external schemas
java:S6377	XML signatures should be validated securely
java:S3518	Zero should not be a possible denominator

12.2. Lexique

Lexique	
Terme	Définition
Charte	Une charte est l'ensemble de règles et principes fondamentaux d'une institution officielle (Définition Wikipedia).
Instanciation, instancier	Le fait de créer une instance d'un objet à partir de sa classe.
<code>ClassLoader</code>	La classe <code>ClassLoader</code> est un objet qui est responsable du chargement des classes.
<i>Thread</i>	Les processus légers (en anglais, <i>thread</i>) sont similaires aux processus en cela qu'ils représentent tous deux l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père partagent une même partie de sa mémoire virtuelle. Les <i>threads</i> sont définis dans le langage Java, sous forme d'objet.
Multi-threadé	On parlera de système multi-threadé quand il met en oeuvre plusieurs <i>threads</i> .
<i>Wrapper</i>	Classe « enveloppant » l'exécution d'un autre programme, pour lui préparer un environnement particulier.
Persistance	La gestion de la persistance des données et éventuellement des états de programme se réfère au mécanisme responsable de la sauvegarde et de la restauration de données, afin qu'un programme puisse se terminer sans que ses données ni son état d'exécution soient perdus.
Héritage	<i>Inheritance</i> en anglais : mécanisme permettant le partage et la réutilisation de propriétés entre les objets. La relation d'héritage est une relation de généralisation/spécialisation, qui organise les objets en une structure hiérarchique.
Atomique	Se dit d'opérations que l'on peut considérer comme indivisibles. Par exemple, une instruction assembleur du processeur l'est. Le fait de tester une variable et d'effectuer un branchement ne l'est pas forcément.
Atomicité	Caractéristique de certaines opérations complexes, en particulier les transactions, qui ne seront réellement effectuées que si toutes ses composantes peuvent être réalisées. Sinon, rien ne sera fait. Voir Atomique ci-dessus.
Mutable	Changeable: toutes les variables en Java sont pas défaut changeables, variantes (mutables). Pour rendre une variable non mutable, on peut utiliser le modificateur <code>final</code> ou <code>private</code> et ne fournir aucune méthode qui permette de la changer.
Immuable	Propriété de ne pas être mutable.
<i>Deadlock</i>	Un interblocage (<i>deadlock</i> en anglais, parfois appelé aussi «étreinte mortelle» ou encore «boucle létale») est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque deux processus légers (<i>thread</i>) concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique.

Bloc critique	On appellera bloc (ou section) critique, une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément. Cette section est contrôlée par une primitive de synchronisation (mutex, sémaphore). Cette synchronisation a pour fonction d'éviter que des ressources non partagées d'un système soient utilisées en même temps. Java propose la définition de sections critiques à l'aide du mot clé <code>synchronized</code> .
Sérialisation, désérialisation	<p>La sérialisation (ou <i>marshalling</i> en anglais) est un processus visant à encoder l'état d'un objet qui est en mémoire sous la forme d'une chaîne d'octets dans un flux de données. Cette chaîne d'octets pourra par exemple être utilisée pour la sauvegarde sur disque (persistance) ou le transport sur le réseau (proxy, RPC,...).</p> <p>L'activité inverse, visant à décoder la suite d'octets pour créer une copie conforme des objets d'origine, s'appelle la désérialisation (ou <i>unmarshalling</i>).</p>
Réflexion	<p>On entendra par réflexion : la découverte des caractéristiques d'une classe (classe mère, champs, méthodes, ...). Elle permet d'instancier des classes de manière dynamique, en agissant sur ses champs, en appelant ses méthodes . La réflexion est potentiellement intéressante pour l'écriture d'un générateur de code générique pour un ensemble de classes ou pour tout outil devant faire abstraction des spécificités d'un applicatif en proposant un service générique.</p> <p>Ce processus s'appuie sur l'API Java <i>Reflection</i> : cette API sert également dans le processus de sérialisation d'un objet Java ou dans la génération du code de création d'une table en base de données pour la persistance de la classe cible.</p>
Membre	Désigne indifféremment champ, méthodes, classe à l'intérieur d'une classe.
Modificateurs	<p>Désigne les :</p> <ul style="list-style-type: none"> attributs de visibilité des variables et des méthodes que sont <code>public</code>, <code>protected</code>. Sans précision, l'attribut de visibilité par défaut est <i>package</i> (ou <i>friendly</i>) ; ainsi que les autres attributs que sont <code>static</code>, <code>final</code>, <code>native</code>, <code>synchronized</code>, <code>volatile</code>, <code>abstract</code> et <code>transient</code>.
Membre de classe	Membre d'une classe pouvant être référencée sans création d'objet à partir de cette classe, c'est à dire membre déclaré avec le modificateur <code>static</code> . C'est pourquoi on utilisera aussi indifféremment dans ce document, les appellations membre de classe et membre statique.
Abstrait(e)	Abstrait(e) pourra désigner suivant le contexte le caractère <code>abstract</code> des classes et/ou des membres de classes qui ont été définis avec le modificateur <code>abstract</code> du langage Java.
Types énumérés	On désigne par types énumérés le nouveau type fourni par Java 5 le mot clé <code>enum</code> . Un type énuméré est un type dont la valeur fait partie d'un jeu de constantes définies.
<i>Package</i> (ou <i>friendly</i>)	L'attribut de visibilité par défaut est désigné indifféremment par <i>package</i> , <i>friendly</i> et <i>package friendly</i> à travers ce document.
Ramasse-miettes	Le ramasse-miettes (appelé aussi parfois récupérateur de mémoire, ou glaneur de cellules) (en anglais <i>garbage collector</i> , abrégé GC) est un sous-système de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.
Déréférencement	On parlera de déréférencement d'objet ou de déréférencer un objet pour désigner l'action

	qui consiste à effacer la référence de l'objet en mémoire.
Finaliseurs et Finalisation	Quand le ramasse-miettes est prêt à libérer la mémoire utilisée par un objet, il va d'abord appeler la méthode <code>finalize()</code> de cet objet et ce n'est qu'à la prochaine passe du ramasse-miettes que la mémoire de l'objet est libérée. C'est ce type de méthode qui sera dénommée finaliseur, et le processus finalisation.
IDE	En anglais, <i>Integrated Development Environment</i> : environnement de développement intégré. Exemples : Eclipse, NetBeans.
Réentrant	Propriété de pouvoir être appelé de façon simultanée par plusieurs <i>threads</i> sans effet de bord.
Transtypage	Fait de convertir une valeur d'un type (source) dans un autre (cible). On parle aussi de conversion de type, de coercion ou de <i>cast</i> .

12.3. Dernières notes de version

12.3.1. Modifications de la version 2.2.0 par rapport à la version 2.1.1

-
- Remplacement de règles obsolètes par leur équivalent SonarQube Squid, avec ajustement de la sévérité dans certains cas : [CONSTRUCT-1], [EXCEP-5], [EXCEP-6.3], [FORM-15], [FORM-18], [FORM-20], [JDBC-RES], [METR-CLASS-6], [PROG-AUTR-6], [PROG-EQUAL-UTIL], [PROG-NULL-1], [PROG-NULL-2], [VAR-5], [ENCAPS-0], [PROG-AUTR-STATIC-DATE], [THR-4.3]
- Suppression de l'implémentation SonarQube Squid de la règle [NOM-16]
- Suppression de l'implémentation CheckStyle de [JDOC-3.1]

12.3.2. Modifications de la version 2.1.1 par rapport à la version 2.1.0

- Remplacement de règles obsolètes PMD par leur équivalent SonarQube Squid : [CONSTRUCT-4], [EXCEP-7.1], [EXCEP-7.2], [HTTPSESS-3], [JDBC-PS], [METH-4], [NOM-7], [NOM-10], [NOM-15], [NOM-16], [PROG-AUTR-4], [PROG-CLONE-3], [PROG-EQUAL-COMP], [PROG-EQUAL-DEF], [VAR-2], [VAR-5], [VAR-6] ; avec ajustement de la sévérité dans certains cas : [ENCAPS-1], [EXCEP-13], [METH-11], [PROG-AUTR-3], [PROG-INTERDIT-4], [PROG-SWITCH-4], [THR-1.3]
- La sévérité de la règle [CONSTRUCT-1] passe de Critique à Majeure.

12.3.3. Modifications de la version 2.1.0 par rapport à la version 2.0.9

- Remplacement des contraintes Obligatoire, Fortement recommandé et Recommandé (et leurs étoiles associées) par les sévérités Bloquante, Critique, Majeur, Mineur et Info (sur le même modèle que SonarQube).
- Remplacement des règles obsolètes Checkstyle, PMD et PMDSoda par leur équivalent SonarQube Squid. Ajustement de la sévérité dans certains cas.
- [PROG-INTERDIT-1] Outillage de la règle avec SonarQube Squid
- [JSP-LIBEL] Outillage de la règle avec le SonarQube Web Plugin
- [METR-LONG-NOM-4] Création de la règle (bogue 52346)
- [MEM-7] Remplacement de la règle maison PMDSoda par la règle FindBugs PMB_INSTANCE_BASED_THREAD_LOCAL. Les variables ThreadLocal statiques sont désormais autorisées.
- [NOM-4] Suite à la fusion de la DGI et de la DGCP en DGFiP, le packaging de plus haut niveau fr.gouv est désormais fr.gouv.finances. Le packaging fr.gouv.impots est encore toléré pour les projets *legacy* de l'ex-DGI.
- [ENCAPS-2] Le nombre maximum d'imports statique passe de 1 à 4 (remplacement de l'implémentation PMDSoda par la règle PMD TooManyStaticImports).
- [EJB-NOM] Remplacement de l'implémentation maison PMDSoda par la règle PMD MDBAndSessionBeanNamingConvention. Au passage, le suffixe EJB n'est plus toléré.
- [EJB-SESS-NOM-1], [EJB-SESS-NOM-2], [EJB-SESS-NOM-3], [EJB-SESS-NOM-4], [JDBC-RS], [MICRO-OPTIM-4], [PROG-INTERDIT-2], [PROG-INTERDIT-3] Remplacement des implémentations maison PMDSoda par les règles PMD RemoteSessionInterfaceNamingConvention, LocalInterfaceSessionNamingConvention, LocalHomeNamingConvention, RemoteInterfaceNamingConvention, CheckResultSet, AvoidUsingShortType, AvoidAccessibilityAlteration, AvoidUsingNativeCode.
- Suppression de la règle [PROG-NULL-4], qui était doublonnée par [PERF-STR-7].
- Suppression de la règle [THR-1.2], fusionnée avec [THR-1.1].

12.3.4. Modifications de la version 2.0.9 par rapport à la version 2.0.8

- [METH-8] Le nombre maximum d'instructions **return** par méthode passe de un à trois (bogue 97557).

12.3.5. Modifications de la version 2.0.8 par rapport à la version 2.0.6

- [PROG-INTERDIT-5] Ajout d'une règle déconseillant l'utilisation de `File.deleteOnExit()` (bogue 64412)
- [PROG-SWITCH-3] Ajout du cas particulier sur les `enum` (bogue 48560).

12.3.6. Modifications de la version 2.3.1 par rapport à la version 2.2.0

- Ajout en annexe de la liste des règles contrôlées par SonarQube avec le jeu de règles CoCA 2.3.1

12.4. Licence du présent document



Paternité - Pas d'Utilisation Commerciale - Partage des Conditions Initiales à l'Identique 2.0 France

Vous êtes libres :

- ♦ de reproduire, distribuer et communiquer cette création au public
- ♦ de modifier cette création

Selon les conditions suivantes :



Paternité. Vous devez citer le nom de l'auteur original.



Pas d'Utilisation Commerciale. Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.



Partage des Conditions Initiales à l'Identique. Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

- ♦ A chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les conditions contractuelles de mise à disposition de cette création.
- ♦ Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits.

Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur : copies réservées à l'usage privé du copiste, courtes citations, parodie...)

Ceci est le Résumé Explicatif du [Code Juridique \(la version intégrale du contrat\)](#).

[Avertissement](#) 