Guide de bonnes pratiques du développement JAVA

Application SIRH (Y2 && 150)

Date	Auteur	N° Version	Objet de la modification
05/04/2020	Jamal BOUJAOUD	1.0	Création

Table des matières

1. INTRODUCTION

1.1. Objectif du document

Le présent document est un recueil des bonnes et mauvaises pratiques de développement.

Il constitue une suite de recommandations dont l'objectif est d' :

- Améliorer la performance, surtout la vitesse d'exécution, des « big » traitements.
- Optimiser le code du projet SIRH (150 && Y2), améliorer sa lisibilité et sa compréhension
  Par tout le monde.
- Proposer des pistes permettant de faire re-factorings sans douleur.

La plupart des bonnes pratiques présentées dans ce document ont été réunies à la suite de mon travail sur les chantiers de performance (GénérationDSN, Xémelios,Hopyra, Ecart M-M-1, etc…).

Les autres bonnes pratiques sont de type général tirés des référentiels java connus.

Le contenu est destiné à évoluer, tout le monde peut y contribuer et y apporter des améliorations qu'on voit pertinentes

➔ Le but c'est que ça soit un travail participatif 🙂.

2. LISTE DES BONNES PRATIQUES

L'amélioration de la vitesse d'exécution du code est la pratique qui consiste à modifier un code correct pour le rendre plus efficace du coup le but n'est plus seulement d'écrire du code qui marche, mais qui soit performant 😉.

On peut considérer ce problème d'optimisation de la vitesse à deux niveaux : technique ou métier.

- Au niveau du métier : il s'agit de modifications obtenues par la refonte de la conception de tout ou certaines parties de la logique métier (algorithmes de calcul, règles de gestion, requêtes SQL/HQL, etc…).

- Au niveau technique : il s'agit de modifications obtenues par la refonte des implémentations techniques (connexion à la BDD, persistance des données, gestions des transaction, couplage des couches, sécurité, implémentation et utilisation des design pattern, etc…) Ou par le changement au niveau de la configuration de l'application (niveaux des logs, configuration Hibernate, configurations des caches…).

Ce document traitera que le deuxième niveau.

Voici alors la liste de bonnes pratiques que vous pouvez suivre pour faire un code propre et performant :

2.1. Ne pas optimiser avant de savoir si c'est vraiment nécessaire

⇨ Quelquefois, l'optimisation prématurée prend beaucoup de temps et rend le code difficile à lire et à maintenir. Et on peut même empirer les choses surtout si nos optimisations n'offrent aucun avantage car on a passé beaucoup de temps à optimiser les parties non critiques de l'application.

Alors, comment savoir si vous avez besoin d'optimiser quelque chose ??

⇨ Utilisez JProfiler pour définir la vitesse de votre code et identifier les parties qui sont trop lentes et doivent être améliorées.

2.2. Business Delegate && Dynamic proxy.

Dynamic proxy :

L'intérêt du DynamicProxy dans le projet est de permettre à une seule classe (fr.civitas.framework.metier.BusinessProxy) avec sa méthode (invoke (…)) de desservir tous les appels de méthode à des classes services (rh-metier) autrement dit BusinessProxy est là pour router tous les appels de méthodes services à un seul gestionnaire qu'est la méthode invoke (..).

Business Delegate :

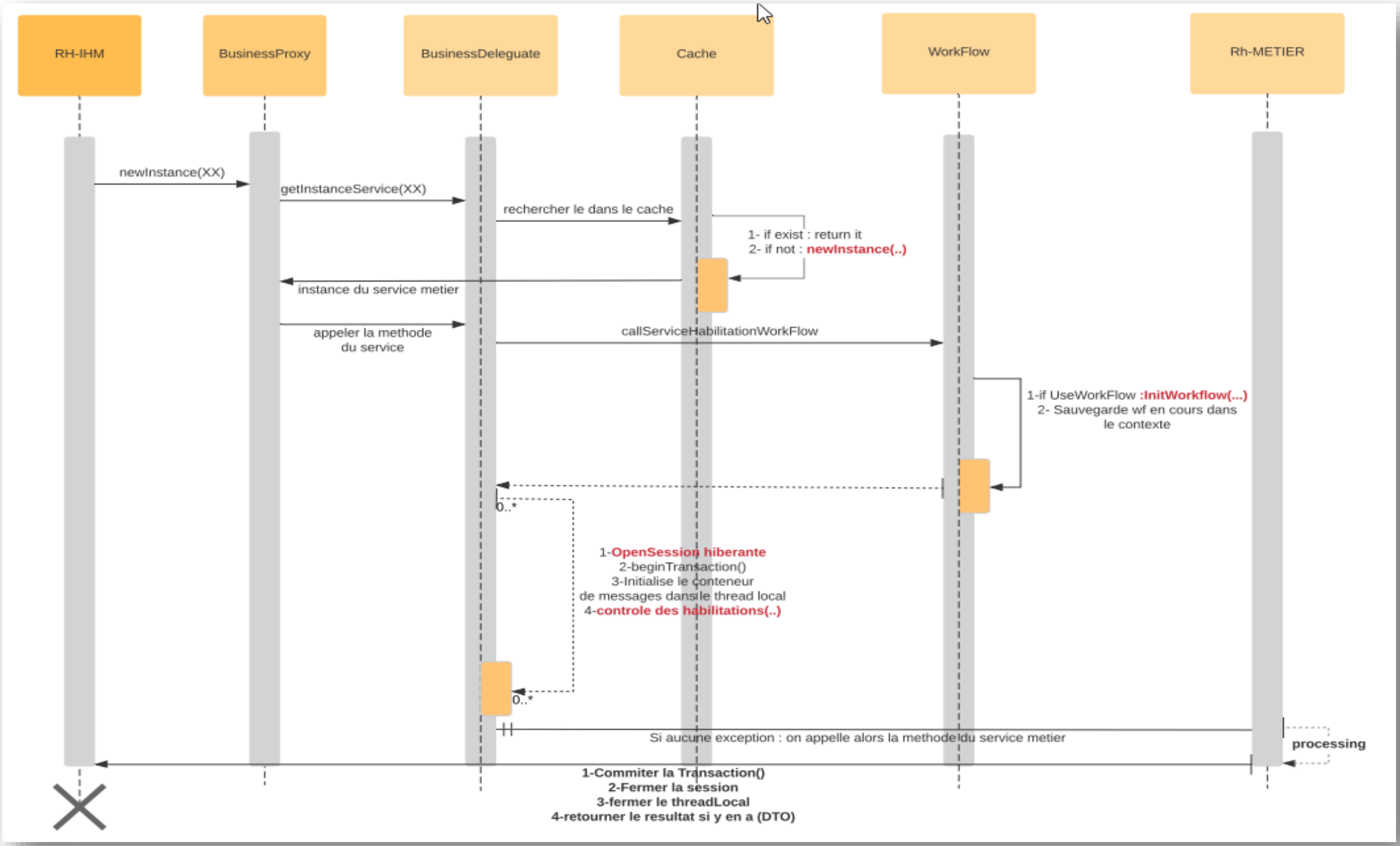
Le business Delegate est utilisé pour réduire le couplage entre la couche service (rh-metier) et la couche présentation (rh-ihm), il agit comme une abstraction des services métier coté client masquant ainsi les détails des implémentations techniques et la complexité des exigences nécessaire à chaque appel aux services métiers tel que la recherche, gestion de transactions, gestion d'accès, gestion du cache, etc…

- ⇨ Le but est que l'application soit fermée à la modification et ouverte à l'extension : il ne faut pas qu'un changement dans l'implémentation des services métier (rh-metier) implique forcément un changement dans le code d'implémentation exposé au niveau de la couche présentation (rh-ihm).

- ⇨ NB : Dans le projet, le DynamincPproxy s'appuie sur le BusinessDelegate donc on reprend tous les comportements de ce dernier dans le businessProxy 👆.

Ci-dessous un diagramme de séquence représentant les différentes actions effectuées quand on passe par le BusinessProxy pour appeler une méthode du service métier (la fameuse instruction).

```
XXXService serv = (XXXService)  BusinessProxy.newInstance(XXXService.class);
Serv.callMyMethode(…) ;
```


RH-IHM
BusinessProxy
BusinessDeleguate
Cache
WorkFlow
Rh-METIER
newInstance(XX)
getInstanceService(XX)
rechercher le dans le cache
1- if exist : return it
2- if not : newInstance(..)
instance du service metier
appeler la methode
du service
callServiceHabilitationWorkFlow
1-if UseWorkFlow :InitWorkflow(...)
2- Sauvegarde wf en cours dans
le contexte
0..*
1-OpenSession hiberante
2-beginTransaction()
3-Initialise le conteneur
de messages dansle thread local
4-controle des habilitations(..)
0..*
Si aucune exception : on appelle alors la methode du service metier
processing
1-Commiter la Transaction()
2-Fermer la session
3-fermer le threadLocal
4-retourner le resultat si y en a (DTO)

Mauvaise pratique

⇨ Utiliser le BusniessProxy dans un service de la couche RH-métier pour appeler une méthode d'un autre service.

```
@Override
public void traitement(GenerationDonneesDSNBean bean) throws MetierException {
    logger = new LoggerMetier("generationDonneesDsn.log", this.getClass(), true);
    logger.put("------- Generation données contrat -------");
    SrvMessageTraitementAsynchrone.getCurrent().setMessage("Début du traitement");
    try {
        traitementDonneesContrat(listeAgents);
        //GENERATION DES DONNEES REMUNERATION
        GenerationDonneesDSNRemunerationService remuService = BusinessProxy.newInstance
                (GenerationDonneesDSNRemunerationServiceImpl.class);
        remuService.execute(bean, listeAgents, logger);
        //GENERATION DES DONNEES ENTERPRISE
        GenerationDonneesDSNEntrepriseService donneesDSNEntrepriseService = BusinessProxy.newInstance
                (GenerationDonneesDSNEntrepriseServiceImpl.class);
        donneesDSNEntrepriseService.execute(bean, listeAgents, logger);
    } catch (Exception e){
        logger.put("Erreur technique lors de la génération des données DSN ");
        logger.put(e);
    }

    logger.put("Fin de traitement " + SrvDate.getDateHeureMinuteSecondeToString(SrvDate.getDateCourante()));
    SrvMessageTraitementAsynchrone.getCurrent().setMessage("Génération des données DSN - Terminé");
    SrvMessageTraitementAsynchrone.getCurrent().setLog(logger.getFichierLog());
}
```

⇨ On perd énormément de temps dans l'exécution des bouts de code qui ne sont pas censé être lancés (réouvertures des session Hibernate, gestion transactionnelle, contrôles des habilitations, gestion d'accès ….)

Bonne pratique

⇨ Utiliser le BusniessProxy que depuis la couche présentation (rh-ihm) pour appeler une méthode de la couche service (rh-metier).

```
@Override
public void traitement(GenerationDonneesDSNBean bean) throws MetierException {
    logger = new LoggerMetier("generationDonneesDsn.log", this.getClass(), true);
    logger.put("------- Generation données contrat -------");
    SrvMessageTraitementAsynchrone.getCurrent().setMessage("Début du traitement");
    try {
        traitementDonneesContrat(listeAgents);
        //GENERATION DES DONNEES REMUNERATION
        getDsnRemunerationBS().execute(bean, listeAgents, logger);
        //GENERATION DES DONNEES ENTERPRISE
        getDsnEntrepriseBS().execute(bean, listeAgents, logger);
    } catch (Exception e){
        logger.put("Erreur technique lors de la génération des données DSN ");
        logger.put(e);
    }

    logger.put("Fin de traitement " + SrvDate.getDateHeureMinuteSecondeToString
            (SrvDate.getDateCourante()));
    SrvMessageTraitementAsynchrone.getCurrent().setMessage
    ("Génération des données DSN - Terminé");
    SrvMessageTraitementAsynchrone.getCurrent().setLog(logger.getFichierLog());
}
```

```
private GenerationDonneesDSNEntrepriseService getDsnEntrepriseBS()
        throws MetierException {
 if(generationDonneesEntrepriseBS==null)
     generationDonneesEntrepriseBS= new GenerationDonneesDSNEntrepriseServiceImpl();
    return generationDonneesEntrepriseBS;
 }
```

2.3. Ne pas instancier les dépendances directes d'une classe service dans ses méthodes métiers.

Mauvaise pratique

```
private void traitementContratsNonModifies(List<ContratDsnPersistant> contratsTraites,
        AgentPersistant agent, CotisationPersistant regimeMSA, List<Double> codeRubAT,
        List<ContratDsnPersistant> contratsNonTraites) throws MetierException {

        for (ContratDsnPersistant c : contratsNonTraites) {

            new UtilitaireDSNServiceImpl(bean.getEts_cod()).
                    completerDonneesContrat(bean, null, regimeMSA, codeRubAT, logger);

            // reconduire les données contrats de la période précédente
            DonneesContratDsnPersistant donneesContrat ;
            if (null == donneesContrat) {
                logger.put(agent.getAgt_num()
                        + " "
                        + agent.getOpc_etatcivil().getEtc_nomusu()
                        + " "
                        + agent.getOpc_etatcivil().getEtc_prenom()
                        + " "
                        + c.getCdn_code()
                        + " Aucune donnée contrat trouvée sur la période précédente");
                continue;
            }
```

⇨ Ici en plus de faire l'instanciation de la classe UtilitaireDSNServiceImpl à l'intérieur de la méthode métier, on l'a mis encore à l'intérieur de la boucle for, ça sera donc instancié autant de fois que le taille de la liste parcourue (x fois).

⇨ Imaginons que la méthode est appelée par un client (depuis RH-IHM ou par autre service) qui fait lui aussi une boucle (Y fois)

```
for(AgentPersistant a : bean.getListAgentsPersistant()) {
    traitementContratsNonModifies(contratsTraites, agent,
                    regimeMSA, codeRubAT, contratsNonTraites);
}
```

⇨ Donc au total UtilitaireDSNServiceImpl sera instancié (x * y) fois alors que c'est censé être instancié une seule fois ➔ Une Consommation de mémoire et CPU (vitesse) sans besoin (goulot d'étranglement)

Bonne pratique

⇨ Ajouter une variable d'instance nomeServiceBS et créer sa méthode getter (pour chaque dépendance), la méthode getter doit créer et retourner une instance de la classe service s'il n'existait pas déjà :

```
private UtilitaireDSNService utilitaireDsnBS;

UtilitaireDSNService getUtilitaireDsnBS() throws TechniqueMineureException {
    if(utilitaireDsnBS==null) {
        utilitaireDsnBS=new UtilitaireDSNServiceImpl(bean.getEts_cod());
    }
    return utilitaireDsnBS;
}
```

⇨ La méthode métier devient :

```
private void traitementContratsNonModifies(List<ContratDsnPersistant> contratsTraites,
        AgentPersistant agent, CotisationPersistant regimeMSA, List<Double> codeRubAT,
        List<ContratDsnPersistant> contratsNonTraites) throws MetierException {

        for (ContratDsnPersistant c : contratsNonTraites) {

            getUtilitaireDsnBS().completerDonneesContrat(bean, null, regimeMSA, codeRubAT, logger);
                    // reconduire les données contrats de la période précédente
            DonneesContratDsnPersistant donneesContrat ;
            if (null == donneesContrat) {
                logger.put(agent.getAgt_num()
                        + " "
                        + agent.getOpc_etatcivil().getEtc_nomusu()
                        + " "
                        + agent.getOpc_etatcivil().getEtc_prenom()
                        + " "
                        + c.getCdn_code()
                        + " Aucune donnée contrat trouvée sur la période précédente");
                continue;
            }
```

⇨ Là on est sûr que UtilitaireDSNServiceImpl n'aura qu'une unique instance pour chaque instance de la classe service.

⇨ Juste cette modification m'a fait gagner (44 minutes) dans l'exécution du traitement Hopyra (pour 3000 agents).

2.4. Ne pas initialiser les dépendances d'une classe ni au moment de la déclaration ni l'intérieur du constructeur :

Mauvaise pratique

```
public class GenerationDonneesDSNServiceImpl extends ServiceAsynchrone
                                implements GenerationDonneesDSNService {
    AgentDao agentDao = new AgentDaoImpl();
    ParametreApplicatifDao paramApplDao = new ParametreApplicatifDaoImpl();
    ParametreDao paramDao = new ParametreDaoImpl();
    ContratDsnDao contratDao = new ContratDsnDaoImpl();
    DossierStatutaireDao dossStatDao = new DossierStatutaireDaoImpl();
    DossierRemunerationDao dossRemuDao = new DossierRemunerationDaoImpl();


    private HistoriqueOrganismeFWDao histoOrganismeDao;
    private VersementIndividuDao versementDao;

    public GenerationDonneesDSNServiceImpl() {
        this.histoOrganismeDao = new HistoriqueOrganismeFWDaoImpl();
        this.versementDao = new VersementIndividuDaoImpl();

    }
```

⇨ Ici la plupart des dépendances de la classe GenerationDonneesDSN est instancié au moment de déclaration et le reste est instancié à l'intérieur du constructeur, chose qui n'est pas bon car on n'est pas sûr qu'ils seront toutes utilisées à chaque fois on fait appel aux méthodes de cette classe.

Bonne pratique

⇨ Pour plus d'optimisation de la mémoire ainsi que du CPU et pour garantir une instanciation qu'à la demande (lazyInstantiation 🙂) , il faut procéder comme dans la bonne pratique précédant :

- Créer une méthode getter pour chaque dépendance qui se charge de créer et retourne une instance de la dépendance s'il n'existait pas.

⇨ Comme ça on est sûr que chaque dépendance ne sera instanciée qu'après sa première utilisation.

```
/** The liste dossier paie service impl. */
private ListeDossierPaieServiceImpl listeDossierPaieServiceImpl;

/** The parametre dao impl. */
private ParametreDaoImpl parametreDaoImpl;

/**
 * Gets the liste dossier paie service impl.
 *
 * @return the liste dossier paie service impl
 */
private ListeDossierPaieServiceImpl getListeDossierPaieServiceImpl() {
    if (listeDossierPaieServiceImpl == null)
        listeDossierPaieServiceImpl = new ListeDossierPaieServiceImpl();
    return listeDossierPaieServiceImpl;
}

/**
 * Gets the parametre dao impl.
 *
 * @return the parametre dao impl
 */
private ParametreDaoImpl getParametreDaoImpl() {
    if (parametreDaoImpl == null)
        parametreDaoImpl = new ParametreDaoImpl();
    return parametreDaoImpl;
}
```

2.5. Utiliser le design pattern Singleton pour les classes dont le fonctionnement est global.

⇨ Parfois, une classe, par son fonctionnement et son utilisation, ne nécessite être instanciée qu'une seule fois durant l'exécution de l'application d'où vient l'utilité du DP singleton dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet ➔ donc un gain de performance (car on économise de la mémoire et du CPU).

⇨ Le singleton doit être utilisé avec précaution, il faut tenir en compte que l'application SIRH est multi-thread et en Saas l'application est aussi multi-environnements : donc dans cas du Saas l'application est multi-thread et au même temps multibases de données et si une classe est « singleton » sa seule instance sera commune à tous les environnements 👆.

⇨ Un exemple concret est celui de la classe ***AuthentificationNetProxy*** : le rôle est de s'authentifier auprès du serveurs ***netEntreprise*** et récupère le token(image ci-dessous) :

```java
import static fr.civitas.metier.commun.dsn.ws.utils.JAXBUtils.marshalObjectToXML;

// TODO: Auto-generated Javadoc
/**
 * The Class AuthentificationNetProxyImpl.
 * @author jboujaou
 *
 */
public class  AuthentificationNetProxy extends RestClientConfiguration implements AuthentificationNetAPI{
    /** The logger. */
    private Log logger = LoggerDecorator.Of(LoggerFactory.getLogger(this.getClass()));
    /**
     * Instantiates a new authentification net proxy impl.
     */
    private AuthentificationNetProxy() {}
    /**
     * The Class AuthentificationNetProxySingletonHolder.
     */
    private static class AuthentificationNetProxySingletonHolder {
        /** Instance unique non préinitialisée. */
        private final static AuthentificationNetProxy instance = new AuthentificationNetProxy();
    }
    /**
     * Point d'accès pour l'instance unique du singleton.
     *
     * @param endpoint the endpoint
     * @param version  the version
     * @return the authentification net proxy
     */
    public static AuthentificationNetProxy getInstance() {
        return AuthentificationNetProxySingletonHolder.instance;
    }
    /**
     * Authentifier.
     *
     * @author jboujaou
     * @param authNetRequest the auth net request
     * @return the auth net response
     * @throws TechniqueMajeureException the technique majeure exception
     */
    @Override
    public AuthNetResponse authentifier(AuthNetRequest authNetRequest) throws TechniqueMajeureException{
        logger.debug("[AUTHENTIFICATION A NET ENTREPRISE][TRAME D ENTREE] : {0}",marshalObjectToXML(authNetRequest));
        Response response =getWebTarget()
                                    .path(format(DSNServicesURIs.AUTH_URI, getApiVersion().getVersion()))
                                    .request(MediaType.APPLICATION_XML)
                                    .headers(getHeaders(ServicesConstantes.AUTHENTIFICATION,StringUtils.EMPTY,Declaration.NET))
                                    .post(Entity.entity(authNetRequest, MediaType.APPLICATION_XML));

        return AuthNetResponse.of(response);
    }

    public AuthentificationNetProxy setup(String endpoint, APIVersion version) {
        setApiURL(endpoint);
        setApiVersion(version);
        initConfiguration();
        return this;
    }
```

2.6. Utiliser l'api NIO au lieu de l'api IO (Y2) et Bufferisation (150) dans les traitements entrées-sorties

- ⇨ Java NIO est considéré comme plus rapide que java IO car nio supporte le mode non bloquant qui est plus rapide que le mode bloquant sur lequel est basé l'api IO standard de java. En plus NIO optimise la copie de données en prenant en charge le mécanisme de bufferisation est bien connu.

- ⇨ Le mécanisme de bufferisation consiste à stocker ce qui est écrit dans une zone mémoire à accès rapide (en tout cas plus rapide qu'un disque), et de ne déclencher l'écriture effective sur le disque que lorsque cette zone mémoire arrive à saturation, ou à intervalles de temps réguliers.

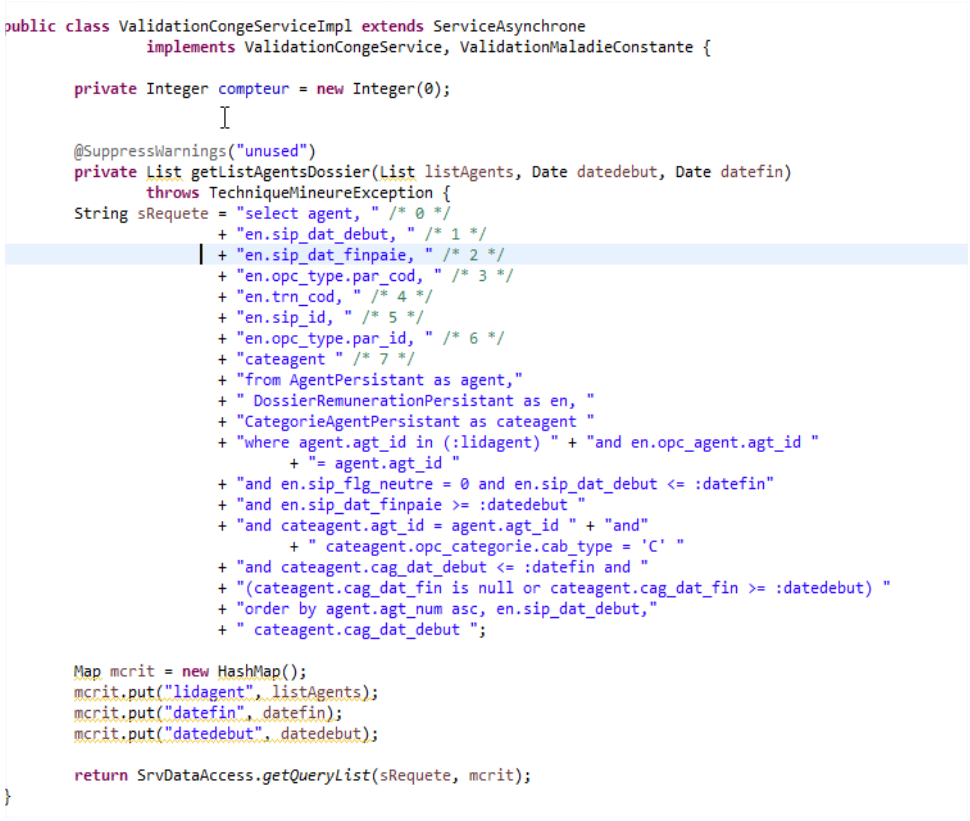
- ⇨ **Contrainte :** l'api NIO n'est disponible qu'à partir du JDK 1.7 mais on peut coder nous-même la bufferisation (par les classes bufferedReader et BufferedOutputStream) pour le projet 150 🤌.

Mauvaise pratique	Bonne pratique
`` public void copier(File fichier_source, File fichier_dest, LoggerCommun logger) throws IOException { try (InputStream is = new FileInputStream(fichier_source); OutputStream os = new FileOutputStream(fichier_dest);) { byte[] buffer = new byte[2048]; int length; while ((length = is.read(buffer)) > 0) { os.write(buffer, 0, length); } logger.put("Copie Réussie de " + fichier_source + " vers " + fichier_dest); } catch (IOException e) { logger.put("Copie KO de " + fichier_source + " vers " + fichier_dest); } } ```	- Y2 : ``` /** * @author jboujaou * Instantiates a new tools. */ @UtilityClass public class Tools { /** * Pour améliorer les performances des flux , on utilise Le mécanisme de * bufferisation . il consiste à stocker ce qui est écrit dans une zone * mémoire(tampon) à accès rapide (plus rapide qu'un disque) et de ne déclencher * l'écriture effective sur le disque que lorsque cette zone mémoire arrive à * saturation. **/ public static final BiConsumer<File, File> copyWithBuffering = (fichierSource, fichierDest) -> { try { Files.copy(fichierSource.toPath(), fichierDest.toPath()); } catch (IOException e) { } }; ``` - 150 :

```java
/**
 * @author jboujaou
 * The Class Utils.
 */
public class Utils {

    public static void copyWithBuffering(File fichierSource, File fichierdest) throws IOException {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = new BufferedInputStream(new FileInputStream(fichierSource));
            out = new BufferedOutputStream(new FileOutputStream(fichierdest));
            byte[] buffer = new byte[2048];
            int length;
            while ((length = in.read(buffer)) > 0) {
                out.write(buffer, 0, length);
                out.flush();
            }
        } finally {
            if (null != in) {
                in.close();
            }
            if (null != out) {
                out.close();
            }
        }
    }
```

2.7. Eviter d’écrire les requêtes SQL ou HQL dans la couche métier (rh-metier).

Mauvaise pratique	Bonne pratique
	⇨ Pour bien respecter le principe du pattern DAO, il faut veiller à séparer la couche gérant les traitements métier de la couche gérant la persistance des données (CRUD). ⇨ Donc il est nécessaire d'encapsuler toutes les méthodes dao ou les instructions qui construisent les requêtes SQL ou HQL dans la couche DAO. ⇨ Bref, il faut masquer le code responsable de la persistance des données au code « extérieur (services) », et l'exposer uniquement via des interfaces.

```
public class ValidationCongeServiceImpl extends ServiceAsynchrone
        implements ValidationCongeService, ValidationMaladieConstante {

    private Integer compteur = new Integer(0);

    @SuppressWarnings("unused")
    private List getListAgentsDossier(List listAgents, Date datedebut, Date datefin)
        throws TechniqueMineureException {
    String sRequete = "select agent, " /* 0 */
                + "en.sip_dat_debut, " /* 1 */
                + "en.sip_dat_finpaie, " /* 2 */
                + "en.opc_type.par_cod, " /* 3 */
                + "en.trn_cod, " /* 4 */
                + "en.sip_id, " /* 5 */
                + "en.opc_type.par_id, " /* 6 */
                + "cateagent " /* 7 */
                + "from AgentPersistant as agent,"
                + " DossierRemunerationPersistant as en, "
                + "CategorieAgentPersistant as cateagent "
                + "where agent.agt_id in (:lidagent) " + "and en.opc_agent.agt_id "
                    + "= agent.agt_id "
                + "and en.sip_flg_neutre = 0 and en.sip_dat_debut <= :datefin"
                + "and en.sip_dat_finpaie >= :datedebut "
                + "and cateagent.agt_id = agent.agt_id " + "and"
                    + " cateagent.opc_categorie.cab_type = 'C' "
                + "and cateagent.cag_dat_debut <= :datefin and "
                + "(cateagent.cag_dat_fin is null or cateagent.cag_dat_fin >= :datedebut) "
                + "order by agent.agt_num asc, en.sip_dat_debut,"
                + " cateagent.cag_dat_debut ";

    Map mcrit = new HashMap();
    mcrit.put("lidagent", listAgents);
    mcrit.put("datefin", datefin);
    mcrit.put("datedebut", datedebut);

    return SrvDataAccess.getQueryList(sRequete, mcrit);
}
```

2.8. Utiliser les nouveautés du (java8) quand c'est possible.

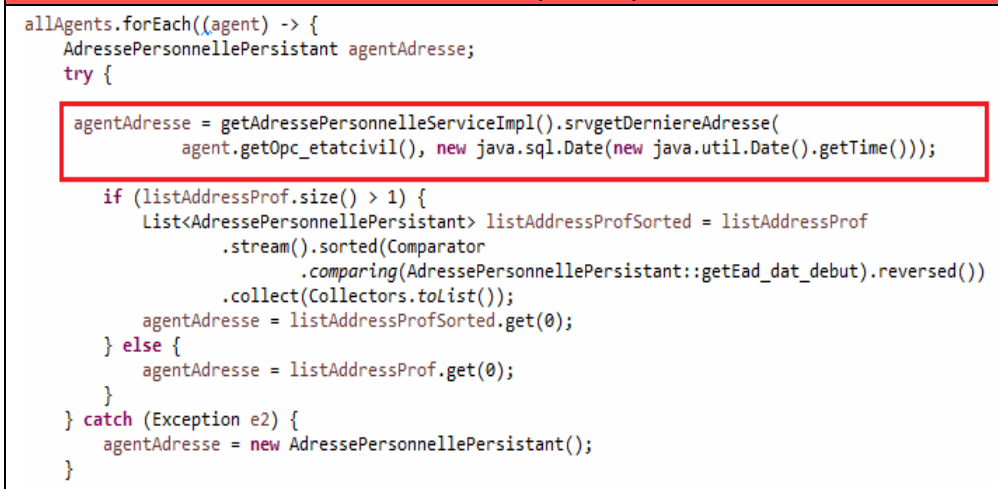
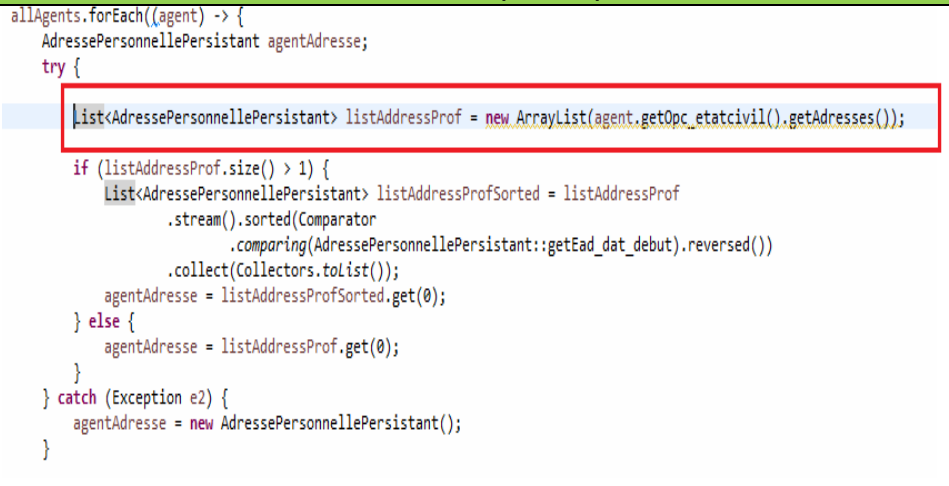
⇨ Un benchmark comparatif a été réalisé avec (JMH) entre Java 7 et Java 8 et on a constaté que Java 8 est plus performant.

⇨ La Migration en java 8, même sans adaptation de code (Juste JRE), améliore la performance (gain de l'ordre de 4 % en moyenne).

2.9. Hibernate : Traitement par lot && **OutOfMemoryException**

Mauvaise pratique	Bonne pratique
`` Session session = sessionFactory.openSession(); Transaction tx = session.beginTransaction(); for (int i=0; i<100000; i++) { AgentPersistant agent= new AgentPersistant(.....); session.save(agent); } tx.commit(); session.close(); ```	``` Session session = sessionFactory.openSession(); Transaction tx = session.beginTransaction(); for (int i=0; i<100000; i++) { AgentPersistant agent = new AgentPersistant(.....); session.save(agent); if (i % 20 == 0) { //20, same as the JDBC batch size //flush a batch of inserts and release memory: session.flush(); session.clear(); } } tx.commit(); session.close(); ```
⇨ Ceci devrait s'écrouler avec une OutOfMemoryException quelque part aux alentours de la 50 000 -ème ligne ➔ C'est parce que Hibernate cache toutes les instances d'Agent nouvellement insérées dans le cache session.	⇨ Lorsque vous rendez des nouveaux objets persistants, vous devez régulièrement appeler flush() et puis clear() sur la session, pour contrôler la taille du cache de premier niveau. ⇨ Même chose pour la mise à jours et suppression.

2.10. Hibernate : éviter le chargement individuel « *super tardif » d'une association quand le chargement par « proxy » est possible.*

Mauvaise pratique	Bonne pratique
	
⇨ Ici l'adresse d'un agent est récupérée individuellement depuis la base de données avec une méthodes qui lance une nouvelle requête vers la BDD.	⇨ Puisque l'adresse est une association et puisque le persistant agent est déjà chargé de la base, Hibernate a déjà généré un proxy pour cette association qu'on peut récupérer en appelant son getter sans passer par une nouvelle requête individuelle.

2.11. Instruction try-with-resources

⇨ Des ressources comme des fichiers, des flux, des connexions, ... doivent être fermées explicitement pour libérer les ressources sous-jacentes qu'elles utilisent. Généralement cela est fait en utilisant un bloc Try / finally pour garantir leur fermeture dans la quasi-totalité des cas.

⇨ De plus, la nécessité de fermer explicitement la ressource implique un risque potentiel d'oubli de fermeture qui entraine généralement une fuite de ressources.

⇨ Depuis Java 7, l'instruction try avec ressource permet de définir une ressource qui sera automatiquement fermée à la fin de l'exécution du bloc de code de l'instruction.

Exemple :

```java
/**
 * Copier.
 *
 * @param fichier_source
 *            the fichier source
 * @param fichier_dest
 *            the fichier dest
 * @param logger
 *            the logger
 * @throws IOException
 *             Signals that an I/O exception has occurred.
 */
public void copier(File fichier_source, File fichier_dest, LoggerCommun logger) throws IOException {
    try (InputStream is = new BufferedInputStream(new FileInputStream(fichier_source)); OutputStream os = new BufferedOutputStream( new FileOutputStream(fichier_dest));) {
        byte[] buffer = new byte[2048];
        int length;
        while ((length = is.read(buffer)) > 0) {
            os.write(buffer, 0, length);
        }
        logger.put("Copie Réussie de " + fichier_source + " vers " + fichier_dest);
    } catch (IOException e) {
        logger.put("Copie KO de " + fichier_source + " vers " + fichier_dest);
    }
}
```

2.12. Utilisez StringBuilder pour concaténer les chaînes

Mauvaise pratique	Bonne pratique
⇨ Concaténer les strings avec + : les String sont immuables et le résultat de chaque concaténation chaîne est stocké dans un nouvel objet String. Cela requiert de la mémoire supplémentaire et ralentit considérablement l'application, en particulier si on concaténé plusieurs chaînes dans une boucle.	⇨ Utiliser StringBuilder. Il est facile à utiliser et offre de meilleures performances que la concaténation avec + ou avec StringBuffer. Mais gardez à l'esprit que StringBuilder n'est pas thread-safe et peut ne pas convenir à tous les cas d'utilisation.

2.13. Utiliser des primitives lorsque cela est possible

⇨ Un autre moyen simple et rapide permettant d'éviter toute surcharge et d'améliorer les performances consiste à utiliser des types primitifs au lieu de leurs classes Wrapper.

⇨ Donc, il est préférable d'utiliser un int au lieu d'un Integer, ou un double au lieu d'un Double. Cela permet à la JVM de stocker la valeur dans la pile au lieu du tas pour réduire la consommation de mémoire et la gérer globalement plus efficacement.

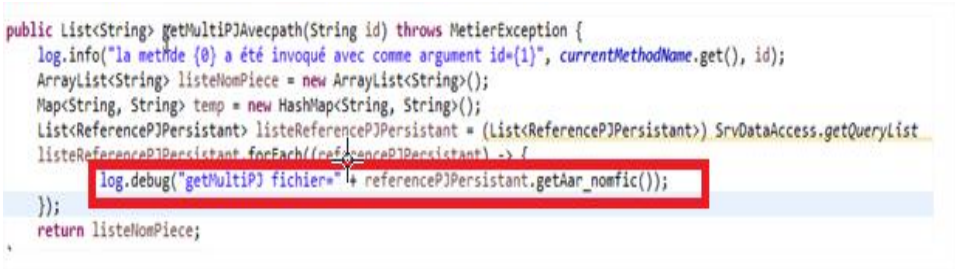
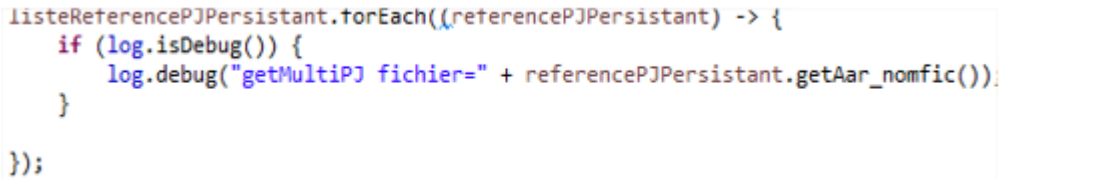
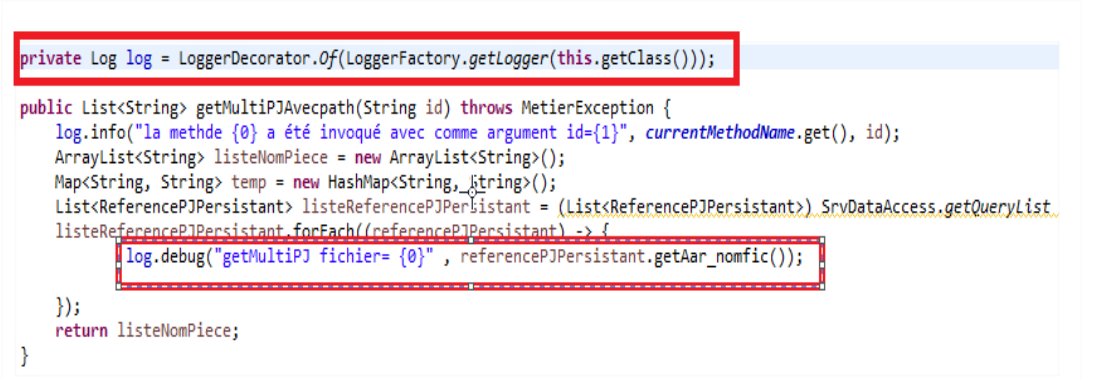
2.14. Eviter BigInteger et BigDecimal

⇨ BigInteger et BigDecimal nécessitent beaucoup plus de mémoire qu'un simple long ou double et ralentissent considérablement tous les calculs.

⇨ Donc, il faut réfléchir à deux fois avant de les utiliser (surtout si on n'a pas besoin de la précision supplémentaire, ou si les numéros utilisés ne dépassent pas la gamme d'un long par exemple).

2.15. Vérifiez toujours le niveau log actuel ou utiliser le loggerDecorator.

⇨ Avant de créer un message de débogage, vous devez toujours vérifier le niveau de log en cours. Sinon, vous pouvez utiliser le *loggerDecorator.java* que j'ai mis en place pour le faire implicitement.

Mauvaise pratique	Bonne pratique
	 ⇨ Ou en utilisant **loggerDecorator** 

```
public List<String> getMultiPJAvecpath(String id) throws MetierException {
    log.info("la methde {0} a été invoqué avec comme argument id={1}", currentMethodName.get(), id);
    ArrayList<String> listeNomPiece = new ArrayList<String>();
    Map<String, String> temp = new HashMap<String, String>();
    List<ReferencePJPersistant> listeReferencePJPersistant = (List<ReferencePJPersistant>) SrvDataAccess.getQueryList
    listeReferencePJPersistant.forEach((referencePJPersistant) -> {
        log.debug("getMultiPJ fichier=" + referencePJPersistant.getAar_nomfic());
    });
    return listeNomPiece;
```

```
listeReferencePJPersistant.forEach((referencePJPersistant) -> {
    if (log.isDebug()) {
        log.debug("getMultiPJ fichier=" + referencePJPersistant.getAar_nomfic());
    }

});
```

```
private Log log = LoggerDecorator.Of(LoggerFactory.getLogger(this.getClass()));

public List<String> getMultiPJAvecpath(String id) throws MetierException {
    log.info("la methde {0} a été invoqué avec comme argument id={1}", currentMethodName.get(), id);
    ArrayList<String> listeNomPiece = new ArrayList<String>();
    Map<String, String> temp = new HashMap<String, String>();
    List<ReferencePJPersistant> listeReferencePJPersistant = (List<ReferencePJPersistant>) SrvDataAccess.getQueryList
    listeReferencePJPersistant.forEach((referencePJPersistant) -> {
        log.debug("getMultiPJ fichier= {0}" , referencePJPersistant.getAar_nomfic());
    });
    return listeNomPiece;
}
```

2.16. Paramétrer les collections.

⇨ Pour plus de lisibilité et pour faciliter le débogage, il faut paramétrer les collections avec une déclaration de type. Cela permet aussi au compilateur Java de vérifier si vous essayez d'utiliser votre collection avec le bon type d'objets

2.17. Le parallélisme et pool de threads.

⇨ Un code générique et un exemple ont été fait en 150 afin de donner la possibilité de lancer des traitements avec un certain nombre de threads exécutés soit les uns après les autres, de façon séquentielle ou en parallèle.

⇨ Je vous laisse le voir, ça peut vous être utile dans certaines circonstances mais il faut être prudent avant de l'utiliser.

```
package fr.civitas.metier.commun.dsn.asynchronoustasks;
```

```java
import static fr.civitas.framework.metier.BusinessDeleguate.beginTransaction;
/**
 * The Class FutureTask.
 *
 * @author jboujaoud@cegid.com
 * @param <T> the generic type
 */
public class FutureTask<T extends Task> implements Callable<Long> {
    /** The business service. */
    private T   businessService;
    /** The context. */
    private Map context;
    /**
     * Of.
     *
     * @param <T>             the generic type
     * @param businessService the business service
     * @param context         the context
     * @return the future task
     */
    public static <T> FutureTask<Task> of(T businessService, Map context) {
        return new FutureTask<Task>((Task) businessService, context);
    }
    /**
     * Call.
     *
     * @return the integer
     * @throws Exception the exception
     */
    @Override
    public Long call() throws Exception {
        Long startTime = System.nanoTime();
        try {
            propagateContextToNewThreads(context);
            beginTransaction(false);
            businessService.run();
        } catch (Exception e) {
            throw e;
        } finally {
            commitTransaction();
        }
        Long endTime = System.nanoTime();
        return endTime - startTime;
    }
    public static String getStatus(Future<Long> ft1) {
        if (ft1.isDone())
            return "Le traitement s'est déroulé correctement";
        if (ft1.isCancelled())
            return "Le traitement a été annulé";
        return "";
    }
    /**
     * Gets the business service.
     * @return the business service
     */
    public T getBusinessService() {
        return businessService;
    }
```

```
package fr.civitas.metier.commun.dsn.service.impl;
```

```
Future<Long> generationDonnesContratsFuturTask = threadsManagerExecutor.submit(FutureTask.of(this, Contexte.getAll()));
Future<Long> generationDonneesRemunerationFuturTask = threadsManagerExecutor.submit(FutureTask.of(getGenerationDonneesRemunerationBS(), Contexte.getAll()));
Future<Long> generationDonneesEnrepriseFuturTask = threadsManagerExecutor.submit(FutureTask.of(getGenerationDonneesEnrepriseBS(), Contexte.getAll()));
    try {
          duration1 = generationDonnesContratsFuturTask.get();
          duration2 = generationDonneesRemunerationFuturTask.get();
          duration3 = generationDonneesEnrepriseFuturTask.get();
     } catch (Exception e) {
          logger.put("Erreur technique lors de la génération des données DSN ");
          logger.put(e);
     }
    ExecutorsTools.shutdown(threadsManagerExecutor);
    logger.put( "[GENERATION_DSN] EXECUTOR IS SHUTDOWN: " + threadsManagerExecutor.isShutdown() );
    logger.put( "[GENERATION_DSN] All TASKS ARE TERMINATED: " + threadsManagerExecutor.isTerminated() );
```