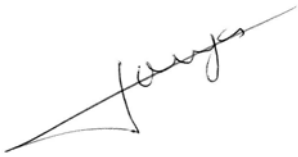
IASI

THALES INFORMATION SYSTEMS

IA-CP-2100-9552-THA
Edition : 02 Date : 18/09/2002
Révision : 02 Date : 06/04/2011
Réf. : - MT : X Code diffusion : E

DOSSIER DE CONCEPTION PRELIMINAIRE
DOSSIER DE CONCEPTION PRELIMINAIRE DU LOGICIEL DE L'OPS IASI

Rédigé par :		le : 11/04/2011	
BRANET Pascal	THALES SERVICES		
CABANE Philippe	THALES SERVICES		
MERIGUET Christelle	THALES SERVICES		
BRUNEL Samuel	THALES SERVICES		
Validé par :		le : 11/04/2011	
TOUJAS Chantal	THALES SERVICES		
BOTELLA Christine	THALES SERVICES		
Pour application :		le : 29/04/2011	
LONJOU Vincent	DCT/ME/EI		

18, avenue Edouard Belin 31401 Toulouse CEDEX 9 Tél. 05 61 27 31 31

BORDEREAU D'INDEXATION

CONFIDENTIALITE : NC	**MOTS CLES : Conception, IASI, algorithmes, Traitement d'images**

TITRE DU DOCUMENT : DOSSIER DE CONCEPTION PRELIMINAIRE
DOSSIER DE CONCEPTION PRELIMINAIRE DU LOGICIEL DE L'OPS IASI

AUTEUR(S) : BRANET Pascal THALES SERVICES
CABANE Philippe THALES SERVICES
MERIGUET Christelle THALES SERVICES
BRUNEL Samuel THALES SERVICES

RESUME : Ce document constitue le dossier de conception du logiciel OPS IASI et est conforme à ses spécifications techniques.

DOCUMENTS RATTACHES : Ce document vit seul.	**LOCALISATION :**

VOLUME : 1	**NBRE TOTAL DE PAGES : 138** **DONT PAGES LIMINAIRES : 7** **NBRE DE PAGES SUPPL. : 0**	**DOCUMENT COMPOSITE : N**	**LANGUE : FR**

GESTION DE CONF. : F	**RESP. GEST. CONF. : GOMEZ MH**

CAUSE D'EVOLUTION : Mise à jour suite à :
- IA-FT-2526 (DM-2100-OPS)
- IA-FT-2521 (DM-2100-OPS)

CONTRAT : 01/8937

SYSTEME HOTE :
Microsoft Word 11.0 (11.0.6568)
G:\Thales\prive\Projets\iasi-ops\Modèles CNES\GDOC 4.0.5\ModeleGDOCIndus.dot
Version GDOC : v4.0.5
Base projet : G:\Thales\prive\Projets\iasi-ops\Modèles CNES\IASI-thales01

DIFFUSION INTERNE

Nom	Sigle	BPi	Observations
BLUMSTEIN Denis	DCT/PO/EV	2504	
CHALON Gilles	DCT/PO/EV	2504	
PONCE Ghislaine	DCT/PO/EV	2504	
SEGALEN Barbara	DCT/PO/EV	2504	
DUPLAA Michel	DTS/MID/VM/D	1502	
MARQUIER Henry	DCT/PO/EV	1321	
MORENO Richard	DCT/PS/TIS	1321	
GOMEZ Marie-Hélène	DCT/PS/TIS	1321	
BAILLY Isabelle	DCT/PS/TIS	1321	
RAYSSIGUIER Michel	DTS/OT/QTIS/VP	811	
MATHIEU Nathalie	EUROGICIEL	1415	
RICHARD Pascal	DSI/EP/SL	3517	
FJORTOFT Roger	DCT/SI/EI	1219	
VANNET Carole	EUROGICIEL		

DIFFUSION EXTERNE

Nom	Sigle	Observations
AYER Patrick	THALES SERVICES	
BOBIN Serge	THALES SERVICES	
BRANET Pascal	THALES SERVICES	
PASCAL Jean-Luc	THALES SERVICES	
CABANE Philippe	THALES SERVICES	
MERIGUET Christelle	THALES SERVICES	
TOUJAS Chantal	THALES SERVICES	
BOTELLA Christine	THALES SERVICES	

MODIFICATION

Ed.	Rév.	Date	Référence, Auteur(s), Causes d'évolution
02	02	06/04/2011	- BRANET Pascal THALES SERVICES CABANE Philippe THALES SERVICES MERIGUET Christelle THALES SERVICES BRUNEL Samuel THALES SERVICES Mise à jour suite à : - IA-FT-2526 (DM-2100-OPS) - IA-FT-2521 (DM-2100-OPS)
02	01	13/09/2007	MERIGUET Christelle THALES SERVICES Traitement de l'action 127 : mise à jour suite à PKCD
02	00	18/09/2002	- CABANE Philippe THALES IS PASCAL Jean-Luc THALES IS Mise au format GDOC et prise en compte des actions de la RCP
01	00	14/06/2002	- BRANET Pascal THALES IS CABANE Philippe THALES IS PASCAL Jean-Luc THALES IS Création du document

SOMMAIRE

GLOSSAIRE ET LISTE DES PARAMETRES AC & AD

API	Applicative Program Interface
AVHRR	Advanced Very High Resolution Radiometer : radiomètre avancé à très haute résolution (visible et infrarouge) sur les satellites polaires
Canal IASI	Un canal IASI est tout simplement un échantillon de spectre IASI.
CCD	Corner Cube Direction : direction du coin de cube (miroir mobile de l'interféromètre)
CCTP	Cahier des Clauses Techniques Particulières
CDR-CGS	Revue Critical Design Review du CGS
CET	Centre d'Expertise Technique : basé au CNES Toulouse, il constitue la boucle longue du segment-sol et génère la configuration de l'OPS et reçoit le produit données technologiques
CFI	Customer Furnished Item
CGS	Core Ground Segment : segment-sol développé par ALCATEL sous contrat d'EUMETSAT, et dans lequel l'OPS ira s'insérer
DDR (ou PKCD)	Detailed Design Review
DIF	Dissemination Facility : sous-système du CGS-EPS
DPC	Data Processing Chain : chaîne de traitement bord des données (une par pixel-sondeur)
DPS	Data Processing Software : logiciel bord de traitement des données.
DR	Document de Référence
Dump	Un dump est une ensemble de données correspondant à un intervalle entre 2 téléchargements. En moyenne il s'agit d'une orbite. Les produits METOP sont structurés autour de cette notion de dump : un produit=un dump
EPS	EUMETSAT Pol ar System : Système 'Polaire d'EUMETSAT
EUMETSAT	EUMETSAT est une organisation intergouvernementale regroupant 17 nations européenne, dont l'objectif est l'établissement, le maintien et l'exploitation des systèmes européens de satellites météorologiques opérationnels
FEPS	Fiche d'Etude des Problème Soulevés
FLTS	File Transfert Service : services du DIF
FOV sondeur	Field Of View : champ de vue
GDD	Gestionnaire de Données et de Diffusion : sous-système du CNES
GSP	Ground Segment Palnification : sous-système du Centre de Mission ENVISAT
HRPT	High Resolution Picture Transmission : centres de traitements locaux qui exploitent les données METOP qui descendent en bande L.
IASI	Infrared Atmospheric Sounding Interferometer : interferomètre de sondage atmosphériq'ue dans l'infrarouge. L'objet de ce document est de spécifier le logiciel opérationnel du traitement sol des données IASI.
IHM	Interface Homme Machine
IPSF	Instrument Point Spread Function : forme du pixel IASI
ISRFEM	Instrument Spectral Response Function Estimation Model
MCS	Monitoring and Control Segment
METOP	Série de satellites météorologiques opérationnels en orbite polaire. IASI est l'un des instruments de METOP.
MLA	MCS Local Agent
MNP	Modèle Numérique de Performance.
NAMS	Naming Services : services du DIF

NOAA	Administration nationale océanique et atmosphérique (Etats-Unis)
NRT	
NZPD	Number of sample of Zero Path Difference : échantillon qui définit la différence de marche nulle sur l'interférogramme
OPS	Logiciel Opérationnel (Operational Software) : correspond au IASI level 1 PPS dans les glossaires d'EUMETSAT. PPS=Product Processing Software
OS	Operating System
PAR-CGS	Preliminary Acceptance Review du CGS, couplée avec la revue FAR de l'OPS (Final Acceptance Review) qui traduit l'acceptation définitive de l'OPS par EUMETSAT
PDR (ou RCP)	Preliminary Design Review
PDR-CGS	Revue Preliminary Design Review du CGS
PDS	Payload Data Segment : Centre de Mission ENVISAT
PGE	Product Generation Element : fournit des services aux PPS [DA1]
PGF	Product Generation Facility
Pixel	Un pixel désigne un des 4 détecteurs du spectromètre infrarouge. A ne pas confondre avec Pixel-Image
Pixel-Image	Un pixel-image est l'un des détecteurs de l'imageur IASI. On en compte 64x64.
PKCD (ou DDR)	Point-Clef de Conception Détaillée
PKPV	Point-Clef Préparatoire à la Validation
PPF	Product Processing Facility
PPS	Product Processing Software
RCP (ou PDR)	Revue de Conception Préliminaire
Scan	Une ligne de mesure IASI est décomposée en 37 sous-cycles également appelés scans ou visées
SGC	Service de Gestion de Configuration
SIF	Simulteurs d'Interfaces du F-PAC ENVISAT (sous système du CNES)
SIP	Sub Instruction Performer : process de l'architecture du SIF
SMTS	Small Message Transfert Service : services du DIF
Sondeur	Par sondeur on désigne le spectromètre infrarouge, par opposition à l'imageur.
SP	Scan Position : position du miroir de changement de visée
TEC	Technical Expertise Center : CET en anglais
U-MARF	Centre unifié d''archivage et de consultation des produits météorologiques
UML	Unified Modelling Language
WO	Working Order

Liste des paramètres AC :

Liste des paramètres AD :

1.GENERALITES

1.1.DOCUMENTS DE REFERENCE ET APPLICABLES

La liste des documents constituant le référentiel du projet OPS-IASI est détaillée dans la « Liste Unique » du Logiciel OPS-IASI [DR100].

Afin de ne pas retarder les travaux techniques, la conception préliminaire de l'OPS a été effectué à partir des documents applicables/de référence disponibles, sans attendre la fin des négociations contractuelles concernant les évolutions des documents suivants :

[DA7] V4.0,

[DA17] V2.3,

[DA18] V2.2

2.INTRODUCTION

2.1.OBJECTIF

Ce document a pour but de présenter la Conception préliminaire de l'OPS-IASI.

La décomposition du sous-système OPS en processus est décrite dans le chapitre 3 Architecture Logicielle.

Le chapitre 4 Architecture de Classes présente les classes constituant les processus identifiés. Chaque groupe de classes est décrit par un diagramme de classes. Les diagrammes de séquence présentant l'implémentation des principaux Use Cases de l'OPS sont ensuite détaillés.

2.2.DOMAINE D'APPLICATION

Ce présent document s'applique aux différentes phases de la réalisation du logiciel OPS IASI décrites dans le Plan d'Application [DA102].

Il est livré au CNES en version définitive pour la RCP.

2.3.NOTATION

La notation UML a été retenue pour décomposer l'architecture logicielle. Nous utilisons l'outil Borland Together dont nous présentons rapidement le symbolisme à travers l'exemple de la figure suivante.

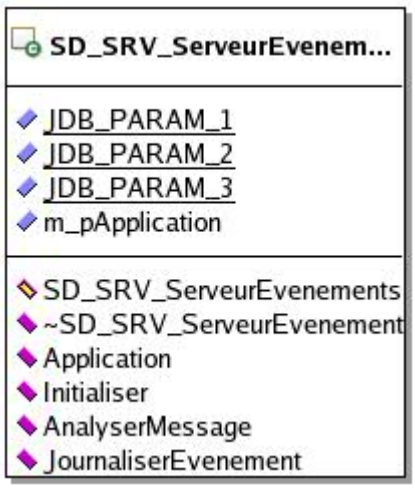


Figure 1 : Exemple de Classe

Dans la notation UML, une classe est représentée par une table découpée en 3 zones. De haut en bas leur contenu est le suivant :

le nom de la classe,

la liste des attributs,

la liste des méthodes,

3.PRESENTATION

3.1.DIAGRAMME DE CONTEXTE

Le diagramme de contexte présente les interactions de l'OPS avec les systèmes externes à travers les interfaces échangées. Le schéma ci-dessous présente le diagramme de contexte de l'OPS.

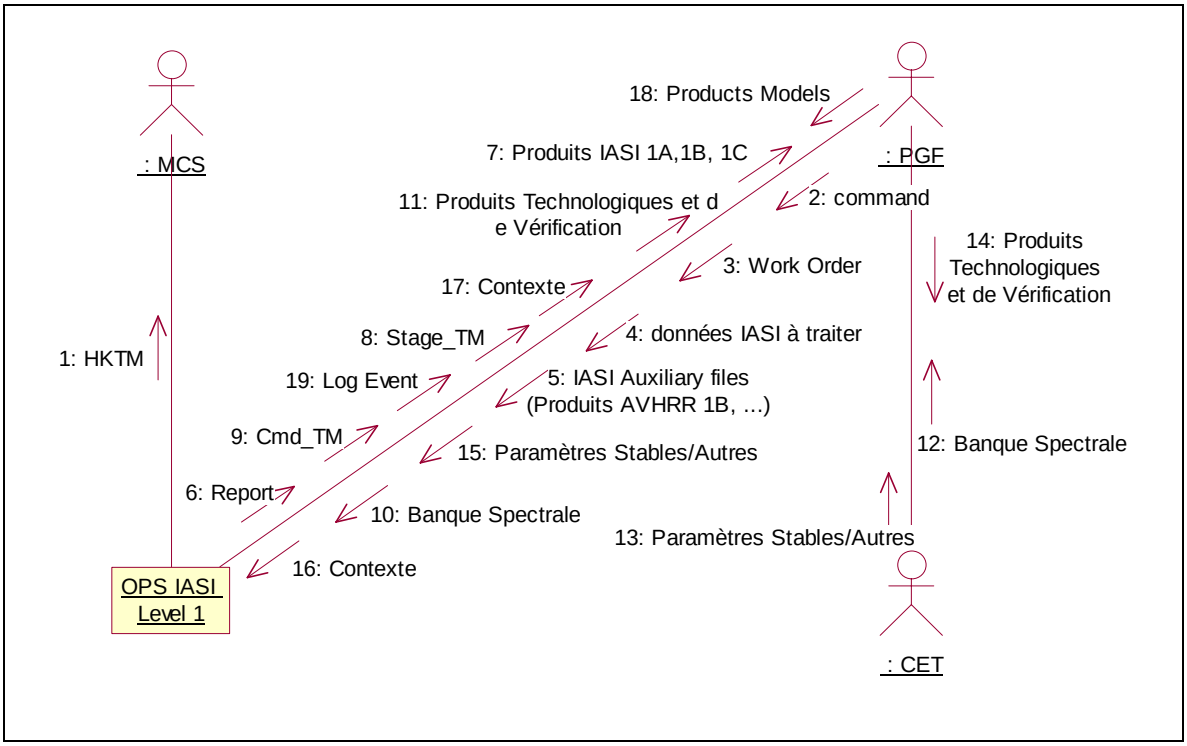


Figure 2 : Diagramme de Contexte de l'OPS-IASI

L'OPS IASI est interfacé avec 2 autres sous-systèmes du CGS :

le MCS qui est le système de Monitoring&Control des sous-systèmes du CGS. Le rôle du MCS est de surveiller le fonctionnement et la disponibilité des divers sous-systèmes du CGS. Pour ceci, les sous-systèmes dont l'OPS doivent lui fournir régulièrement ou sur anomalie des informations de suivi : les HK TM statuts (1).

le PGF qui est le système de Monitoring&Control des chaînes de traitement. Son rôle est de répartir et surveiller les traitements des données des charges utiles EPS en fonction d'un plan de travail sur les calculateurs disponibles. Le PGF a en charge de mettre à disposition l'ensemble des données nécessaires à l'OPS pour effectuer un traitement : work order (3), données à traiter (4), données auxiliaires (Produits AVHRR, …) (5), Contexte(16), Paramètres Stables/Autres (15), Banque Spectrale (10), Product Models (18) ; puis, de contrôler à travers les commandes (2) la génération des

produits. Le traitement d'une commande par l'OPS donne lieu à l'émission d'un compte rendu : cmde_TM (9). En fin de traitement, l'OPS génére un rapport (6) et met les données produites à disposition du PGF qui a en charge de les récupérer et de les diffuser : Produits IASI 1A/1B/1C (7), Contexte (17), Produits Technologiques et de Vérification (11). L'OPS prévient le PGF de la disponibilité d'un nouveau produit par l'envoi d'un message Stage_TM (8). Les événements de l'OPS sont signalés à l'aide de messages Journal de Bord (19).

3.2.RAPPEL DES FONCTIONNALITES

L'analyse des exigences de l'OPS met en évidence les principales fonctions suivantes :

la fonction de génération des produits IASI L1 :

l'initialisation des données de configuration (Banque Spectrale, …),

l'exécution des algorithmes de la chaîne de traitement,

la génération des produits,

les services :

la surveillance et le contrôle de l'OPS,

la réception et le traitement des work orders,

la gestion des messages d'informations,

la gestion des anomalies.

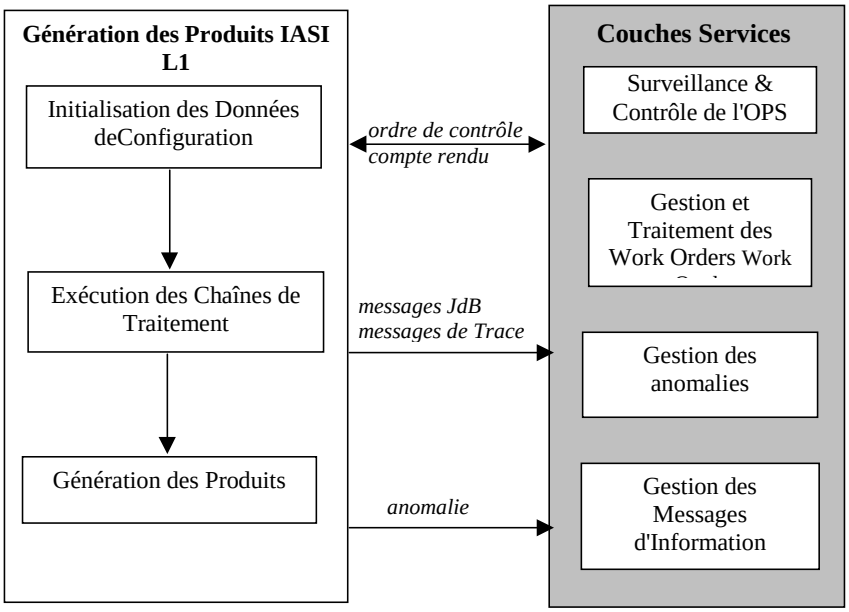


Figure 3 : Fonctions de l'OPS IASI L1

4.ARCHITECTURE LOGICIELLE

4.1.LE DECOUPAGE EN PROCESSUS

4.1.1.Présentation

Le choix de la réutilisation de composants existants, a naturellement guidé la conception de l'OPS. L'OPS est basé sur l'architecture logicielle du SIF; ceci nous a conduit à identifier l'architecture de processus dès le début de la conception.

Cette approche pragmatique ne suit pas à la lettre la méthode UML, qui préconise de commencer par modéliser le problème en déterminant les objets nécessaires et de se préoccuper à la fin du "déploiement des composants". Cette démarche ne permet d'identifier les processus informatiques qu'en fin de conception.

La reprise de l'architecture du SIF, nous a conduit à réutiliser et adapter les **processus de services** du SIF :

- le Serveur de Messages (MSGS) qui centralise et banalise les communications inter-processus,
- le Serveur d'Evénement de Temps (TES) qui génère des événements Timer pour les processus applicatifs qui les ont programmés,
- le Serveur de Messages Journal de Bord (JDBS) qui est chargé de collecter les messages JdB générés par l'ensemble des processus de l'OPS, de les formater puis le transmettre au PGF.

L'architecture est complétée par les **processus applicatifs** spécifiques à l'OPS :

- le Main Process (MP) qui centralise le traitement des interfaces de Monitoring & Control avec le MCS et le PGF. Cette centralisation est imposée par l'architecture du CGS,
- le Work Order Manager (WOM) qui est chargé de gérer et contrôler la production des données IASI level 1 en fonction les ordres de traitement et les commandes envoyées par le PGF,
- le Serveur de Données (SD) qui produit les données IASI Level 1; il encapsule les algorithmes de traitement.

Deux types de processus peuvent être exécutés dans une architecture logicielle :

des processus permanents qui sont toujours présents lorsque le système est en exécution,

des processus déclenchés qui sont exécuté à la demande.

L'avantage des processus permanents est qu'ils sont prêts à traiter immédiatement les événements, de plus ils s'affranchissent du temps d'initialisation.

Par définition, les processus de service et les processus en interface directe avec les systèmes externes doivent être permanents.

Le WOM et le SD pourraient être des processus déclenchés. Mais, les contraintes de performance qui préconisent le chargement des données d'initialisation en une seule fois pour le traitement d'un dump, impliquent que ces processus soient des processus permanents.

Tous les processus de l'OPS sont donc des processus permanents.

4.1.2.L'Architecture et les Interfaces

La figure suivante présente l'architecture des processus de l'OPS ainsi que leurs interfaces.

Afin de faciliter la lisibilité du schéma, les éléments suivants n'ont pas été représentés :

- le processus MSGS qui collecte et diffuse les messages entre les processus,
- les interfaces de type messages JDB (*m_LogEvent*) : chaque processus de l'OPS envoie des messages au JDBS en fonction des événements d'exécution,
- les interfaces de type message de trace (*pgf_traceInfo*) envoyées en mode debug par les processus applicatifs de l'OPS.

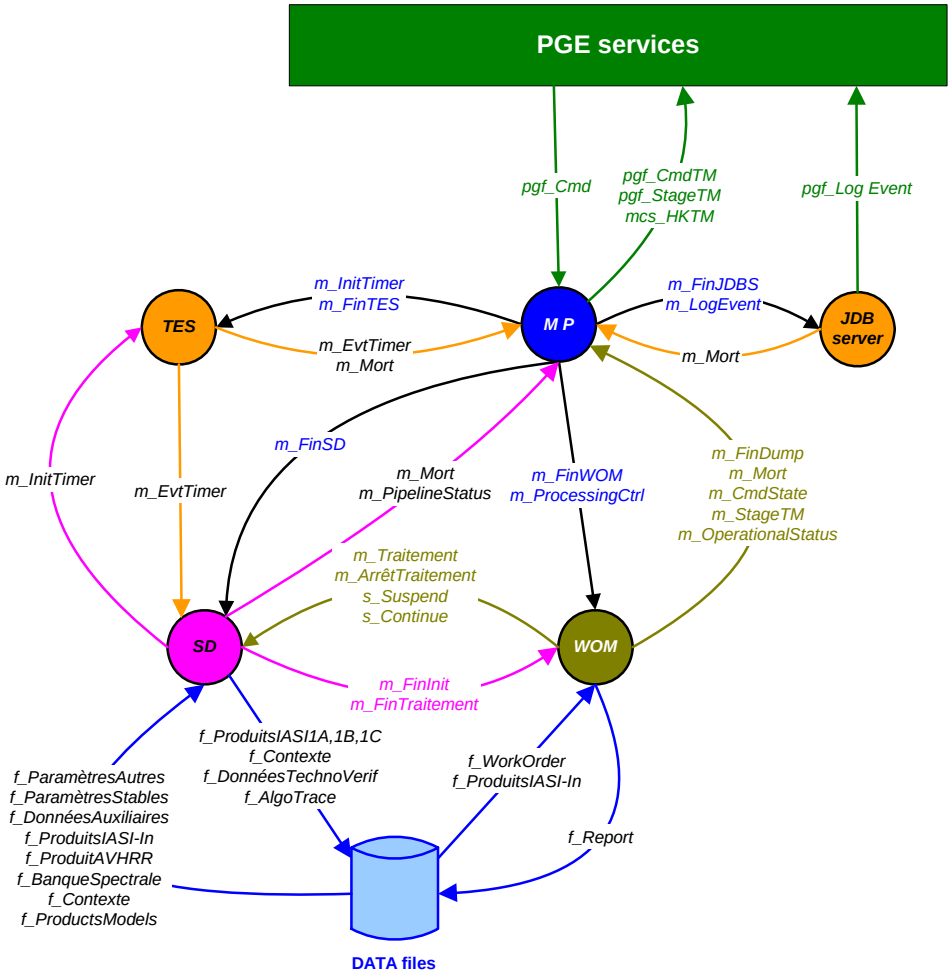


Figure 4 : Architecture Logicielle de l'OPS

La dynamique de l'échange des interfaces est présentée dans la suite de ce chapitre.

Les préfixes des noms des interfaces ont la signification suivante :

m_ : interface interne de type message échangé via le MSGS,

s_ : interface interne de type signal UNIX envoyé au processus,

f_ : interface externe de type fichier.

pgf_ : interface externe avec le PGF via les PGE services (librairie MLA),

mcs_ : interface externe avec le MCS via les PGE services (librairie MLA).

Les Interfaces Internes :

Le tableau suivant présente les interfaces internes échangées entre les processus de l'OPS.

Producteur / Consommateur	MP	WOM	SD	TES	JDBS
MP		*m_Mort* *m_FinDump* *m_CmdState* *m_StageTM* *m_OperationalStatus*	*m_Mort* *m_PipelineStatus*	*m_Mort* *m_EvtTimer*	*m_Mort*
WOM	*m_ProcessingCtrl* *m_FinWOM*		*m_FinInit* *m_FinTraitement*		
SD	*m_FinSD*	*m_Traitement* *m_ArrêtTraitement* *s_Suspend* *s_Continue*		*m_EvtTimer*	
TES	*m_InitTimer* *m_FinTES*		*m_InitTimer*		
JDBS	*m_LogEvent* *m_FinJDBS*	*m_LogEvent*	*m_LogEvent*	*m_LogEvent*	

Tableau 1 : Tableau des Interfaces Internes de l'OPS

m_ArrêtTraitement : message contenant l'ordre d'annulation du traitement en cours.
m_CmdState : message contenant le statut d'exécution d'une commande.
m_EvtTimer : message signalant le déclenchement d'un timer.
m_FinDump : message indiquant la fin du traitement du dump.
m_FinInit : message indiquant la fin d'initialisation du SD.
m_FinTraitement : message indiquant la fin du traitement du granule.
m_FinXX : message contenant l'ordre d'arrêt immédiat du processus XX.
m_InitTimer : message contenant une demande d'initialisation de timer.
m_LogEvent : message contenant un message JdB (LogEvent) à transmettre au PGF
m_Mort : message indiquant la fin anormale d'un processus.

m_PipelineStatus : message contenant un Statut de type HKTM Pipeline Status.
m_OperationalStatus : message contenant un Statut de type HKTM Readiness Status.
m_ProcessingCtrl : message contenant un ordre de contrôle des traitements (Stop/Step/Break/Suspend/Resume).
m_StageTM : message contenant le descriptif du fichier livré.
m_Traitement : message contenant le descriptif du traitement à effectuer.
s_Suspend : signal imposant une interruption du traitement courant.
s_Continue : signal imposant une reprise du traitement courant.

Les informations transmises par les interfaces de type message sont décrites dans le tableau ci-dessous.

Nom Interface	Contenu
m_ProcessingCtrl	Type de la commande : Stop/Step/Break/Suspend/Resume Command_id : 0 dans le cas d'un Stop simulé suite à un arrêt sur anomalie SD process_id (toujours fourni sauf si le process SD est mort : -1) Work_order_filename (seulement dans le cas du Step)
m_CmdState	Type (Start/Stop/Step/Break/Suspend/Resume) Command_id Statut (coded/rejected/completed/aborted)
m_Traitement	Description du traitement à effectuer : informations extraites du work order, temps de début et de fin dans le fichier de données à traiter, mode de traitement (granule/dump) positionnement du traitement (First, Middle, End).
m_FinTraitement	Etat de fin (OK/NOK) Cause d'une fin NOK Dates début/fin du traitement Elapsed time et user time du traitement Liste des fichiers utilisés en entrée pour le traitement Liste des produits générés PCD Message d'anomalie le cas échéant
M_StageTM	Stage_id Command_id Category_product Filename
m_ArrêtTraitement	Aucun
m_FinDump	Statut (OK, NOK)
m_FinInit	Statut (OK, NOK)
m_FinXX	Aucun
m_Mort	Nom du processus
m_LogEvent	Code du message JdB Paramètres du message JdB

Nom Interface	Contenu
m_InitTimer	Type de timer (périodique, date) Dates début/fin validité Période éventuelle Nom logique du timer
m_EvtTimer	Nom logique du timer
m_PipelineStatus	Type déclenchement (fin production granule, timer) PDU sensing start time Sensing time en cours de traitement Numéro d'orbite du granule Instantaneous quality value Stage information (ok/failed)
m_OperationalStatus	Disponibilité de l'OPS (working, idle,unrecoverable error)

Les Interfaces Externes :

Le tableau suivant présente les interfaces externes échangées entre les processus de l'OPS et les sous-systèmes en interface.

Producteur / Consommateur	MP	JDBS	WOM	SD	PGF
MP					*pgf_Cmd*
WOM					*f_WorkOrder* *f_ProduitIASI-In*
SD					*f_ParametresAutres* *f_ParametreStables* *f_BanqueSpectrale* *f_DonnéesAuxiliaires* *f_Contexte* *f_ProductsModels* *f_ProduitAVHRR* *f_ProduitIASI-In*
JDBS					
PGF	*pgf_CmdTM* *pgf_StageTM* *pgf_TraceInfo*	*pgf_LogEvent*	*f_Report* *pgf_TraceInfo*	*f_Contexte* *f_ProduitsIASI-1A,1B,1C* *f_DonnéesTechnoVerif* *f_AlgoTrace* *pgf_TraceInfo*	
MCS	*mcs_HKTM*				

Tableau 2 : Tableau des Interfaces Externes

f_AlgoTrace : fichier des traces des algorithmes de calcul en mode trace .
f_BanqueSpectrale : fichier de Banque Spectrale.
f_Contexte : fichier de Contexte.
f_DonnéesAuxiliaires : fichiers des Données Auxiliaires.
f_DonnéesTechnoVerif : fichier des Données Technologiques et de Vérification.
f_ParametresAutres : fichiers des Paramètres Autres.
f_ParametresStables : fichiers des Paramètres Stables.
f_ProductsModels : fichiers des Products Models.
f_ProduitAVHRR : fichier Produit AVHRR.
f_ProduitIASI-In : fichier produit IASI à traiter (granule ou dump).
f_ProduitsIASI-1A,1B,1C : fichiers produit IASI produit (granule ou dump).
f_Report : fichier Rapport de traitement.
f_Work Order : fichier Work Order.
mcs_HKTM : HKTM statut renvoyée au MCS.
pgf_Cmd : Commande envoyée par le PGF.
pgf_CmdTM : Compte rendu de prise en compte ou d'exécution d'une commande.
pgf_LogEvent : Message de Log Event envoyé au PGF.
pgf_StageTM : Message de mise à disposition d'un fichier de données.
pgf_TraceInfo : Message de trace émis par l'OPS en mode DEBUG

Les PGE services :

Toutes les interfaces (hors fichier) entre l'OPS, le PGF et le MCS s'échangent à l'aide des PGE services (MLA librairie, ..). La figure suivante liste les services du PGE utilisés par chacun des processus de l'OPS.

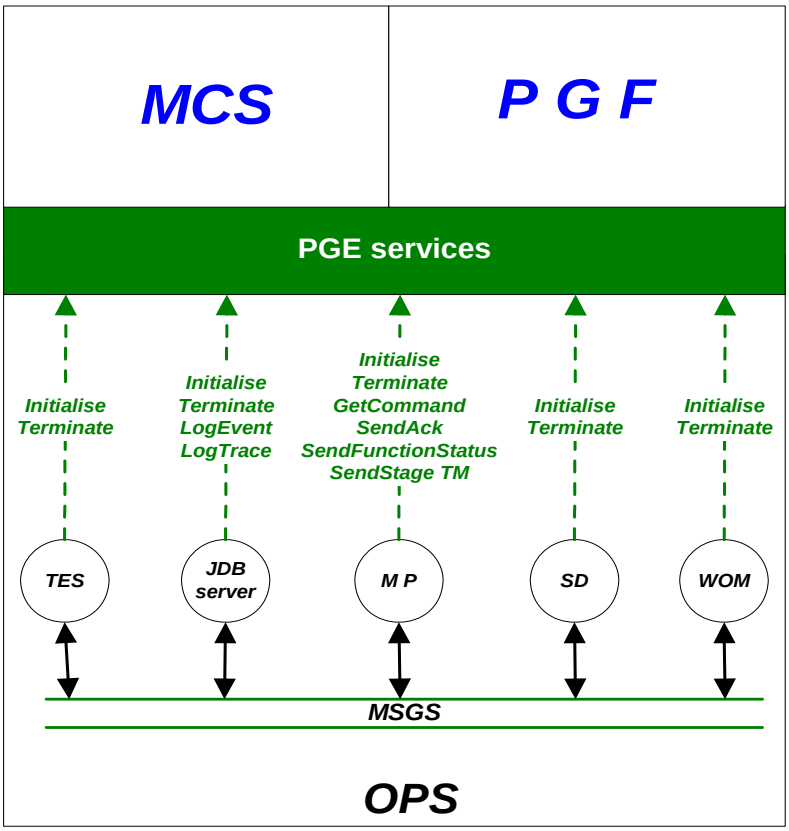


Figure 5 : Les PGE services utilisés par les processus de l'OPS

4.1.3.Mise en Exécution de l'OPS

Tous les processus de l'OPS doivent être lancés par le MP (contrainte de l'architecture du CGS), le MP étant lui-même lancé par le PGF via l'exécution de son shell UNIX.

Le contexte d'exécution d'un processus OPS est composé :

- de son shell UNIX de lancement (*.sh),
- de son fichier des variables d'environnement (*.env),
- d'un exécutable et éventuellement d'un second pour le mode DEBUG,
- éventuellement d'un fichier de configuration (*.cfg).

Le shell UNIX de lancement exécute le fichier de variables d'environnement (*.env), lance l'exécutable du processus en fonction du mode (nominal/debug) et renvoie l'id du processus (sauf dans le cas du MP).

Les paramètres suivants sont passés lors de la mise en exécution du shell du MP :

- le *command_Id* qui est transmis comme argument de lancement de l'exécutable,
- le *Working_Root_Directory* et le *Mode* qui sont stockés dans des variables d'environnement de telle manière que tous les process lancés par le MP puissent les récupérer au besoin.

En fonction du mode (debug/nominal) le shell met en exécution l'exécutable approprié.

Les tableaux suivants listent les arguments (arg) et les variables d'environnement (ve) utilisées par les processus de l'OPS. Cette liste sera complétée en phase de Conception Détaillée.

	Origine	Consommateur
arg_CommandId	paramètre de lancement de MP.sh	MP
ve_WorkingRootDirectory	paramètre de lancement de MP.sh	tous les processus
ve_Mode	paramètre de lancement de MP.sh	tous les processus
ve_NbreThreads	définit dans SD.env	SD
ve_ListeProcessus	définit dans MP.env	MP
ve_NbreLigneGranule	définit dans WOM.env	WOM

Suite à sa mise en exécution, le MP a en charge de mettre en exécution tous les processus de l'OPS, via leur shell, suivant un ordre prédéfini (*ve_ListeProcessus* :MSGS, JDBS, TES, WOM, SD). Il reçoit en retour :

- soit l'id du processus lancé (cet id est utilisé dans le cas d'envoi de signaux),
- soit un compte rendu d'erreur de lancement, dans ce cas, il avertit le PGF et exécute la procédure d'arrêt de l'OPS.

Chaque processus lancé doit impérativement exécuter les services du PGE *Initialise* au démarrage et *Terminate* en fin d'exécution.

La phase d'initialisation du SD est longue ; en effet, celle-ci consiste entre autres à stocker en mémoire le contexte d'exécution : *f_Contexte*, *f_BanqueSpectrale*. Pendant cette phase, l'OPS n'est pas disponible. En fin d'initialisation, le SD avertit le WOM de la fin de son initialisation (*m_FinInit*), celui-ci transmet alors au MP un message de type *m_CmdState* pour la commande Start :

si Fin Init = OK => *m_CmdState (Start, coded),*

si Fin Init = NOK => *m_CmdState (Start, rejected).*

Si l'initialisation se termine en anomalie, le MP exécute la procédure d'arrêt de l'OPS (cf §4.1.4)

Sinon, le MP exécute les traitements suivants :

envoi du message *pge_CmdTM* de type Acknowlegde pour la commande Start (p_command_Id),

transmission du statut HKTM de type Operational Readiness pour avertir le CGS de la disponibilité de la fonction de traitement OPS-IASI level1,

mise en attente d'événements (message ou commandes).

Durant leur phase d'initialisation, les processus nécessitant d'être averti par événements timer doivent définir les conditions (*m_InitTimer*) de déclenchement des timers.

Le schéma ci-dessous montre les interfaces échangées lors du démarrage du processus TES (les processus MSGS et JDBS étant préalablement mis en exécution).

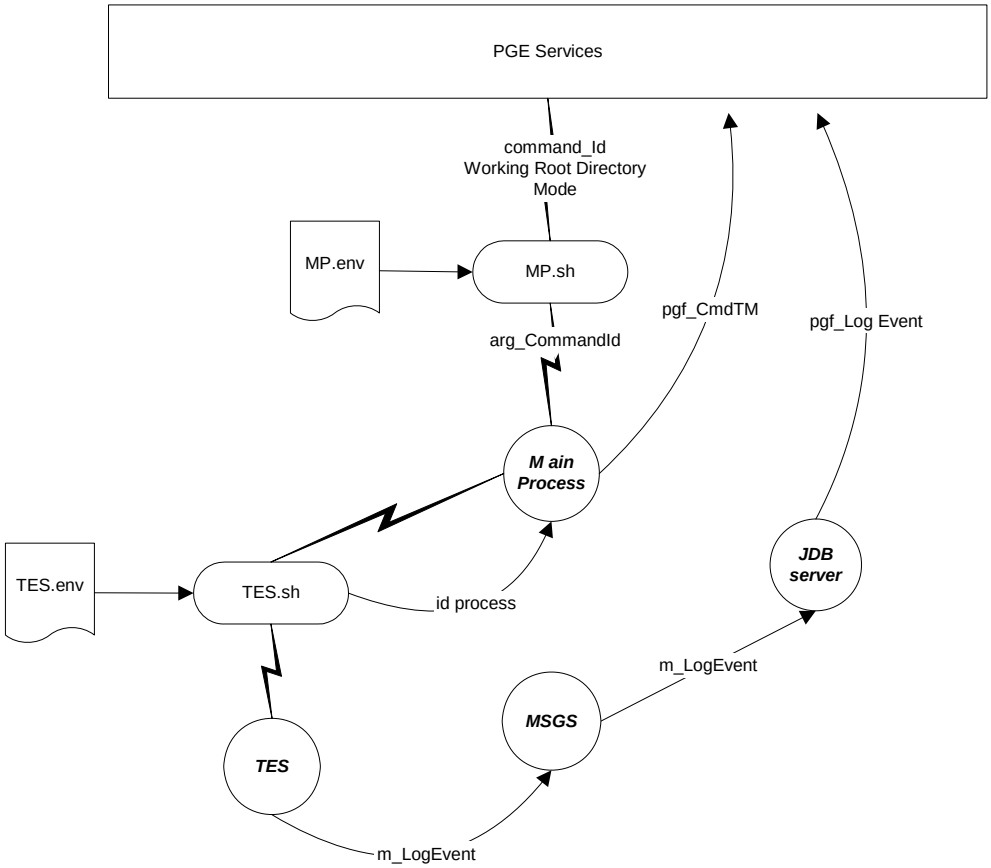


Figure 6 : Démarrage du processus TES

4.1.4.Arrêt de l'OPS

L'OPS doit traiter 3 types d'arrêt :

- l'arrêt demandé par le PGF via la commande Stop,
- l'arrêt nominal en fin de traitement du dump,
- l'arrêt sur anomalie irréversible.

Dans le cas de la fin de traitement d'un dump, la procédure d'arrêt complète de l'OPS est exécutée par le MP sur réception du message *m_FinDump*. La procédure d'arrêt consiste en l'envoi d'une demande de fin individuellement à tous les processus actifs suivant un ordre prédéfini (inverse de l'ordre de démarrage). La demande est transmise à l'aide d'un message (*m_FinXX*) individuellement à chacun des processus en exécution de l'OPS. Puis, le MP transmet au PGF le message de fin d'exécution pour la commande Start (statut completed).

Dans le cas d'une commande Stop envoyée par le PGF : la commande est transmise au WOM (*m_ProcessingCtrl*). Le WOM contrôle la fin d'exécution des traitements en cours (cf Figure 8). Puis, il envoie au MP un message de compte rendu pour la commande Start (*m_CmdState* => Start aborted) puis un message de fin de traitement (*m_FinDump* => NOK). Sur réception du message *m_FinDump,* le MP exécute la procédure d'arrêt complet de l'OPS puis renvoie au PGF le statut aborted pour la commande Start.

Dans le cas d'une anomalie irréversible :

anomalie d'initialisation : le WOM envoi au MP un message de compte rendu pour la commande Start (*m_CmdState* => Start rejected) puis un message de fin de traitement (*m_FinDump* => NOK). Le MP exécute la procédure d'arrêt complet de l'OPS puis renvoie au PGF le statut rejected pour la commande Start.

mort du processus TES, JDBS, ou SD : le MP reçoit un message *m_Mort*, il envoie alors une commande Stop "simulé" (*m_ProcessingCtrl*) au WOM qui exécute la procédure d'arrêt définie pour traiter la commande Stop.

mort du processus WOM : la procédure d'arrêt exécutée sur la réception du message *m_FinDump* est immédiatement exécutée par le MP, puis celui-ci renvoie au PGF le statut rejected pour la commande Start.

mort du processus MSGS ou MP : aucun traitement n'est possible, l'arrêt de l'OPS est une procédure opérationnelle (voir ci-dessous les procédures d'arrêt opérationnelles).

Sur réception d'un message d'arrêt, les processus doivent se terminer "proprement" c'est-à-dire:

fermer la connexion avec le MSGS,

fermer tous les fichiers en cours d'utilisation,

purger les fichiers temporaires,

sauvegarder le contexte courant (ensemble des données d'entrée) sous le répertoire dédié (repertoire /debug de l'arborescence d'execution).

exécuter le service *Terminate* du PGE.

Une **procédure d'arrêt opérationnelle** est à la disposition des opérateurs :

procédure déclenchée par le PGF basée sur la commande Abort.

4.1.5.Traitement des Commandes

Une fois l'OPS en exécution, le MP est en attente d'un événement : commande du PGF ou message en provenance d'un autre processus.

Sur réception d'une commande du PGF, le MP transmet la commande au WOM via un message (*m_ProcessingCtrl*) puis se remet en attente d'un événement.

Le WOM est en charge de gérer l'exécution des traitements en fonction des commandes transmises par le MP : Step, Break, Suspend, Resume, Stop.

Pour ceci, le WOM gère une liste des travaux à effectuer, cette liste de type FIFO est enrichie à chaque réception d'un message *m_ProcessingCtrl* de type Step valide.

Le WOM contrôle d'abord la pertinence de la commande reçue par rapport à son contexte d'exécution et renvoie un message *m_CmdState* (Step, coded ou rejected) au MP qui a en charge de le retransmettre au PGF.

Les contrôles effectués sont les suivants :

Step : le WOM contrôle la disponibilité du work order. Si ce n'est pas le cas, la commande est rejetée par l'OPS. Sinon, la commande est insérée dans la liste des travaux à effectuer.

Break : le WOM vérifie que la liste des travaux n'est pas vide. Si c'est le cas, la commande est rejetée. Sinon, un message d'arrêt du traitement courant *m_ArrêtTraitement* est envoyé au SD. La séquence des interfaces échangées pour le traitement d'un break est présentée sur la Figure 7.

Suspend : le WOM vérifie que la liste des travaux n'est pas vide. Si c'est le cas, la commande est rejetée; sinon, un signal UNIX *s_Suspend* est envoyé au SD.

Resume : le WOM vérifie que l'état courant du SD est suspend. Si ce n'est pas le cas, la commande est rejetée; sinon, un signal UNIX *s_Continue* est envoyé au SD.

Une fois, la commande traitée, le WOM renvoie un message *m_CmdState* (completed ou aborted) au MP qui a en charge de le retransmettre au PGF.

Si la liste des travaux n'est pas vide et si le SD n'effectue aucun traitement de granule, le WOM sélectionne le 1^{ier} travail de la liste puis transmet au SD un message *m_Traitement* décrivant le traitement à effectuer.

En fin de traitement nominal ou non (anomalie, break ou stop) le SD renvoie un message *m_FinTraitement* au WOM indiquant les caractéristiques de fin du traitement (anomalie, …).

Si un travail est dans la liste, le WOM le met en exécution puis traite le message de fin :

génération du fichier rapport d'exécution : *f_Report*,

génération de *m_StageTM* pour chaque type de fichier devant être rapatrié par le PGF. Ce message est transmis au MP qui a en charge de le transmettre au PGF.

génération et transmission du *m_CmdState*,

dans le cas d'une anomalie : transfert des données d'entrée sous un répertoire de stockage temporaire. L'opérateur a en charge de récupérer ces données.

La figure suivante présente les interfaces échangées entre les processus de l'OPS pour le traitement d'un Break, d'un Stop et en fin de traitement nominal d'un Step.

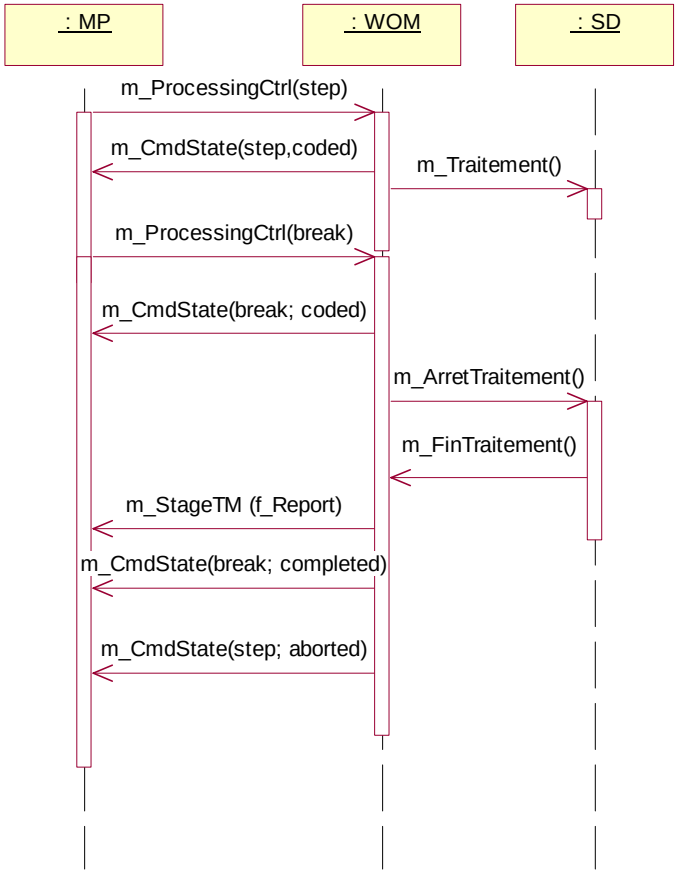


Figure 7 : Interfaces échangées pour le traitement de la Commande Break

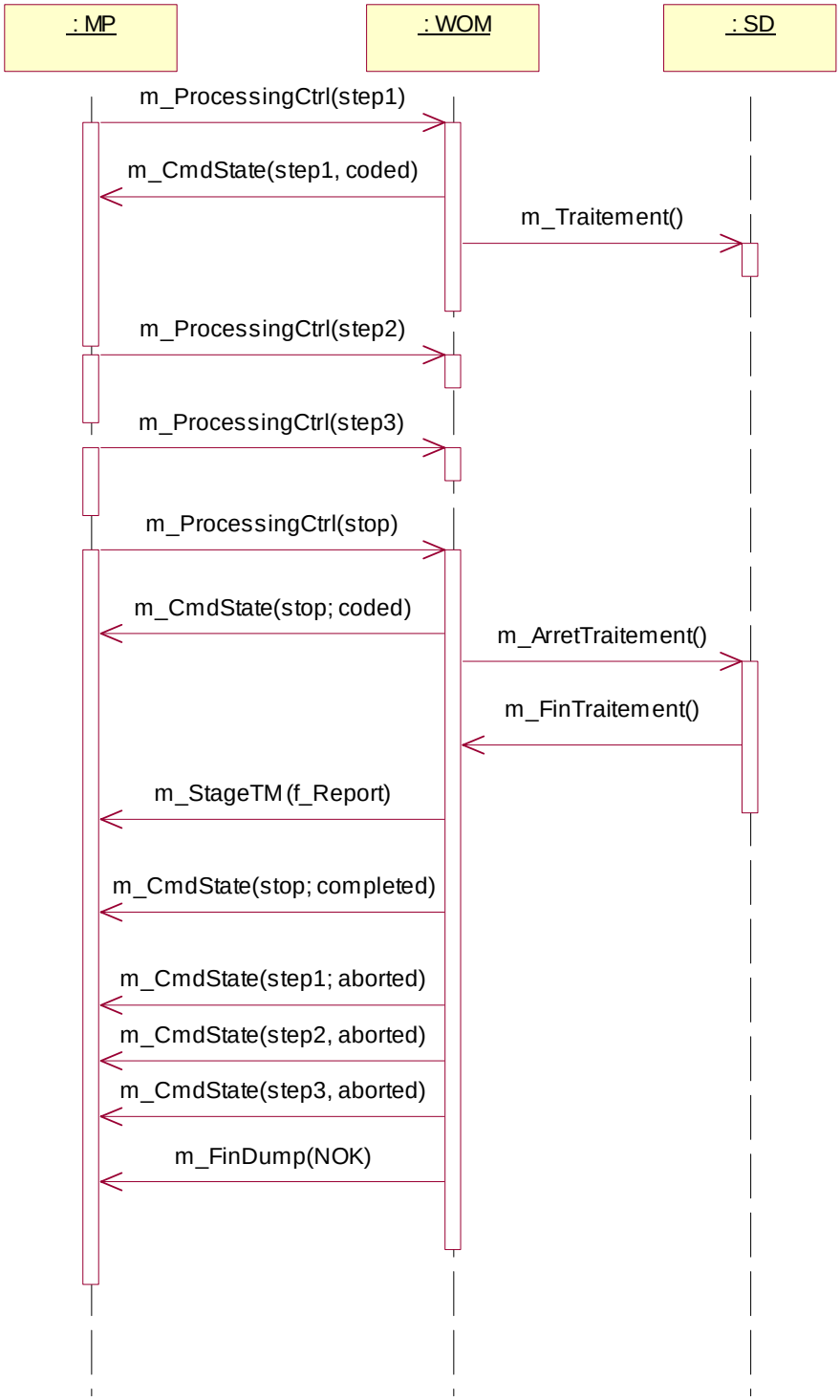


Figure 8 : Interfaces échangées pour le traitement de la Commande Stop

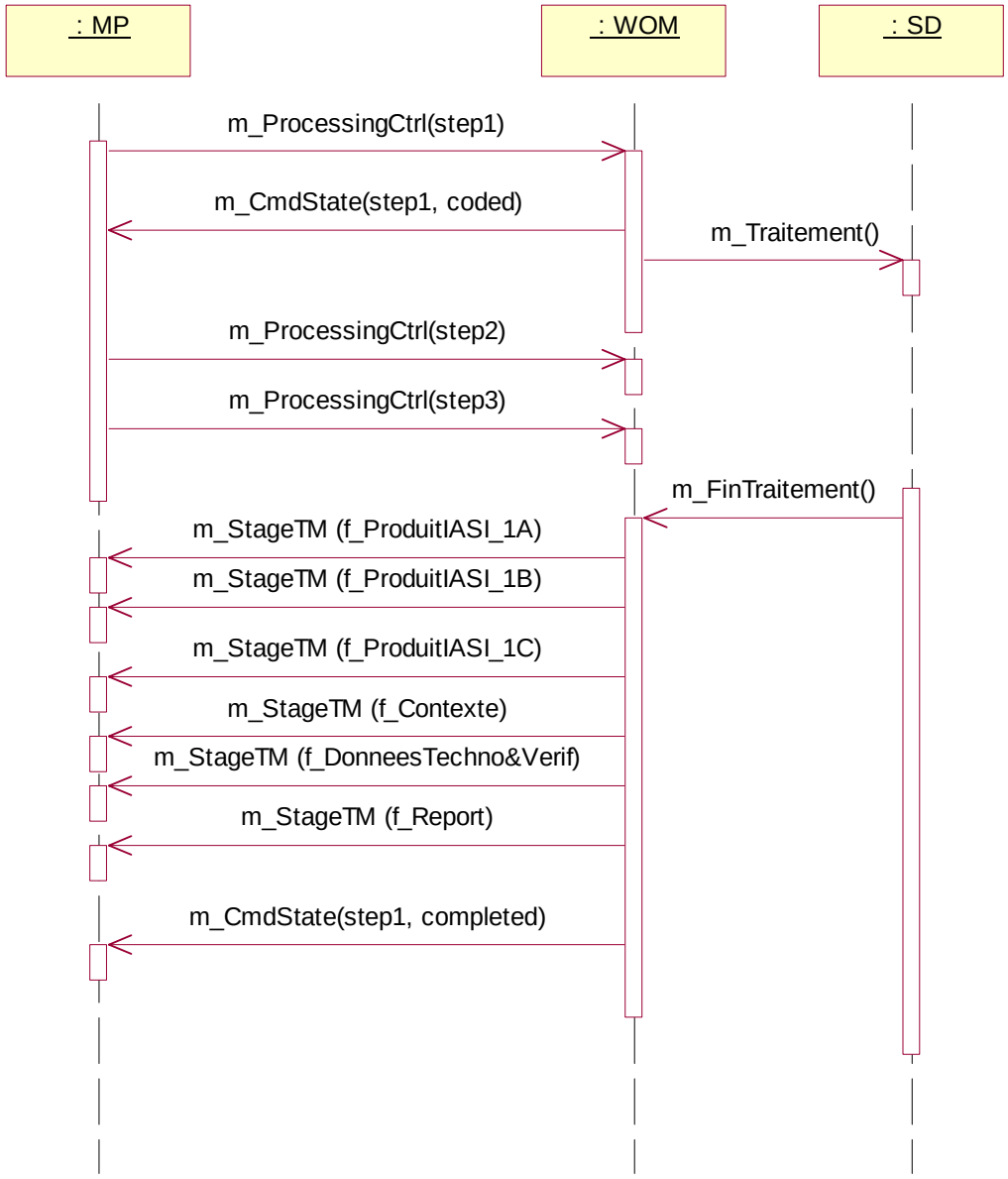


Figure 9 : Interfaces échangées en fin de traitement nominal d'un granule

4.1.6.Surveillance de l'OPS

L'état fonctionnel de l'OPS est surveillé par le MCS à l'aide des HKTM statuts. Le MP a en charge de collecter ces statuts :

soit directement dans le cas des statuts de type Ressource et Performance,

soit via le traitement des messages *m_PipelineStatus* envoyés par le SD,

soit via le traitement des messages *m_OperationalStatus* envoyés par le WOM.

La collecte de ces statuts s'effectue soit de façon périodique soit sur événement (la détection de certain événement pouvant s'effectuer de façon périodique).

Les statuts autres que de type Pipeline sont collectés par le MP soit sur événement, soit sur timer (*m_EvtTimer*) et stockés. Sur réception d'un statut de type Pipeline, tous les statuts stockés par le MP sont envoyés au MCS.

4.1.7.Gestion des Log events

L'envoi des log events au PGF est centralisé par le processus JDBS qui a en charge de les collecter, les formatter et les transmettre au PGF via le PGE services.

Les log events sont transmis des processus de l'OPS au JDBS par un message *m_LogEvent*.

4.1.8.Gestion des Log Traces

En mode DEBUG, les Log Traces émis par chaque processus sont centralisés par le processus JDBS qui les formate et les transmet au PGF via le PGE services. En outre, le SD crée pour des besoins algorithmiques des fichiers de trace volumineux qui sont référencés dans le Log Trace.

4.2.ARCHITECTURE DES REPERTOIRES

L'OPS doit implémenter l'architecture générique de répertoires imposée aux sous-systèmes du CGS.

Cette architecture adaptée à l'OPS est présentée sur la figure suivante.

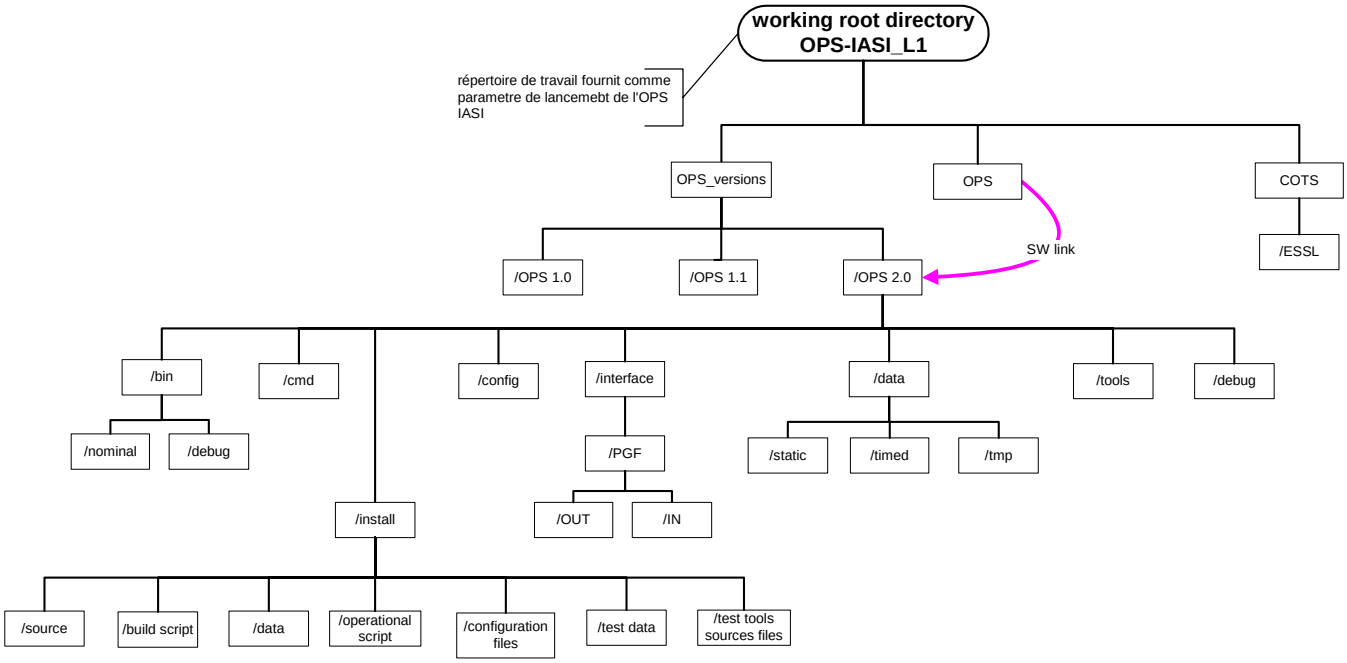


Figure 10 : Architecture des répertoires de l'OPS

Répertoires	Commentaires
OPS	Version opérationnelle de l'OPS. Le PGF gère ce repértoire et positionne le lien logique sur la version opérationnelle.
COTS	Répertoire d'installation des COTS, pour l'OPS, un seul COTS est nécessaire (ESSL).
OPS_version	Répertoires d'installation des versions disponibles de l'OPS
OPS_version/OP2.0	Ce répertoire est crée par la procédure d'installation pour chaque nouvelle version.
OPS_version/OP2.0/bin	Répertoire de dépôt des exécutables de l'OPS. Un répertoire est dédié à la version nominale et

Répertoires	Commentaires
	un à la version de debug. Accès : procédure d'installation de l'OPS.
OPS_version/OP2.0/cmd	Ce répertoire contient les shells de démarrage de tous les processus ainsi que le shell d'arrêt de l'OPS. Accès : procédure d'installation de l'OPS
OPS_version/OP2.0/config	Ce répertoire contient les fichier de configuration et de variables d'environnement. Accès : procédure d'installation de l'OPS, opérateur pour la mise à jour des fichiers d'environnement.
OPS_version/OP2.0/tools	Ce répertoire contient l'outillage nécessaire à la validation, l'exploitation et ou la maintenance de l'OPS. Accès : procédure d'installation de l'OPS.
OPS_version/OP2.0/debug	Répertoire sous lequel sont déposées les traces en mode débug. Les fichiers de log pour la maintenance sont générés sous ce répertoire. Accès : par l'OPS en cours d'exécution.
OPS_version/OP2.0/interface/PGF/IN	Répertoire de dépôt de toutes les données d'entrée de l'OPS. Accès : la gestion de ce répertoires est sous la responsabilité du PGF (dépôt/purge,…), l'OPS accède à ce répertoire en lecture
OPS_version/OP2.0/interface/PGF/OUT	Répertoire de dépôt es données produites par l'OPS. Accès : l'OPS accède à ce répertoire en écriture la purge de ce répertoires est sous la responsabilité du PGF.
OPS_version/OP2.0/install	Répertoire sous lequel sont descendues les données du média de livraison de l'OPS. Accès : procédure de génération de l'OPS.
OPS_version/OP2.0/install/source	Fichiers sources en C/C++ de l'OPS
OPS_version/OP2.0/install/built script	Script de génération des exécutables de l'OPS
OPS_version/OP2.0/install/data	Données opérationnelles : Dans le cas de l'OPS ce répertoire contient le fichier d'Initialisation à froid.

Répertoires	Commentaires
OPS_version/OP2.0/install/operational script	Script et shell opérationnels recopiés dans le répertoire cmd lors de l'installation.
OPS_version/OP2.0/install/configuration files	Fichiers de configuration de l'OPS, ces fichiers sont recopiés sous le répertoire /config par la procédure d'installation
OPS_version/OP2.0/install/test data	Données de validation de l'OPS.
OPS_version/OP2.0/install/tests tools sources files	Fichiers sources des outils de validation
OPS_version/OP2.0/data/static	Répertoire de dépôt des données statiques pour le traitement d'un dump. Afin d'éviter des transferts de fichier pénalisant en temps, nous proposons de positionner des liens logiques sur les fichiers disponibles dans /PGF/IN (répertoire directement accessible).
OPS_version/OP2.0/data/timed	Répertoire de dépôt des données temporelles correspondant au traitement d'un granule. En début de traitement de chaque dump, l'OPS transfère les données à traiter depuis /PGF/IN (répertoire directement accessible).sous ce répertoire.
OPS_version/OP2.0/data/tmp	Répertoire de travail de l'OPS. Ce répertoire contient les fichiers produits par l'OPS. En fin de traitement l'OPS se charge de copier ces fichiers dans le répertoire /PGF/OUT (répertoire directement accessible).

Les transferts de données entre les répertoires de travail /data et les répertoires externes /PGF sont présentés sur la figure suivante dans le cas d'un traitement nominal.

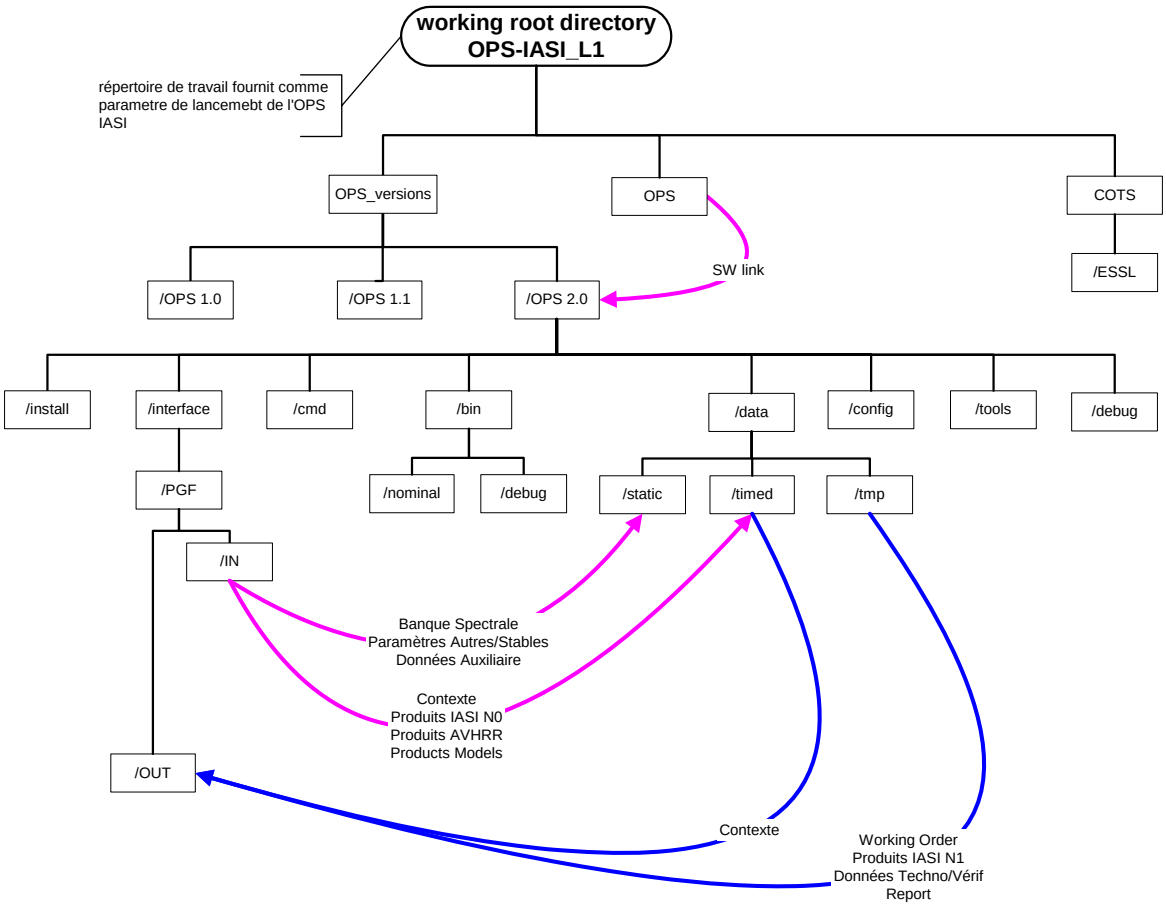


Figure 11 : Transfert de Données pour le Traitement Nominal d'un Granule

4.3.LES SOLUTIONS D'IMPLEMENTATION

4.3.1.La Parallélisation des Traitements

4.3.1.1.Présentation

Le composant Serveur de Données, est spécialisé dans l'exécution et le suivi des traitements algorithmiques. La solution retenue pour atteindre les performances demandées, est la parallélisation par ligne de l'exécution des algorithmes. Le mécanisme du multithreading a été retenu pour implémenter cette solution.

L'ensemble des threads est créé au lancement du processus. L'un des threads est spécialisé dans la scrutation des messages externes; les autres, appelés threads de calcul, sont dédiés aux traitements proprement dits. Par défaut, les threads de calcul sont dans l'état suspendu. Le nombre de threads de calcul est paramétrable : ce paramètre est lu au lancement du process.

Lorsqu'un traitement est demandé par le WOM, le SD recherche le premier thread de calcul libre (état en attente) dans l'ensemble des threads, lui affecte un traitement puis demande au thread de s'exécuter. Lorsque le traitement se termine, le thread repasse dans « l'état en attente » et devient disponible pour un autre traitement.

L'intérêt de cette approche est d'être à la fois simple tout en étant très efficace. En effet, elle permet de maîtriser les threads qui peuvent s'exécuter en parallèle tout en évitant les risques de création et destruction permanente de threads consommateurs de ressource et de temps.

4.3.1.2.Distribution des Traitements sur les Threads

Le Serveur de Données contrôle le traitement d'un granule suivant 3 niveaux hiérarchiques :

le **traitement** : ce niveau gère l'ordre de production transmis par le WOM. Cet ordre est relatif à un granule.

la **tâche** : est l'activité à exécuter, qui est affectée à un thread de calcul.

l'**action** : la tâche est constituée d'une séquence d'actions ; l'action constitue l'élément de granularité minimale géré par le SD.

Ces trois abstractions sont liées entre elles de la façon suivante : le Serveur de Données exécute un traitement ; ce traitement est décrit à l'aide d'un enchaînement de tâches séquencées et/ou parallélisées. Chaque tâche est constituée de une ou plusieurs actions. La figure suivante présente sous forme UML la modélisation des relations entre ces abstractions.

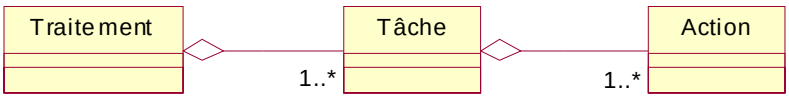


Figure 12 : Association Traitement/Tâche/Action

Le traitement IASI Level1, les tâches, les actions et leur association sont décrits dans les 2 fichiers de configuration suivants :

ModelesTraitements.txt : fichier de description des associations <traitement-tâche>,

ModelesTaches.txt : fichier de description des taches logiques (une tache logique = une liste d'actions).

Ces fichiers de configuration sont chargés au lancement du Serveur de Données. Le contenu de ces fichiers est décrit au §4.4.

4.3.2.L'Encapsulation

Le principe d'encapsulation mis en place dans le système OPS est issu des spécifications techniques de besoin. Celles ci demandent d'englober tous les appels aux librairies externes afin de faciliter la maintenance et l'évolutivité du code (stratégie de Portable Application Module). Dans le cas de l'OPS cela concerne les librairies ESSL, METOP et aux fonctions système.

Afin de rendre l'OPS moins dépendant de ses interfaces externes, lorsque cela a été possible l'accès à ces interfaces a été centralisé dans un processus dédié à leur traitement, comme par exemple pour :

les commandes qui permettent le contrôle de l'OPS par le PGF,

les HK-TM statuts qui permettent de transférer au MCS des informations périodiques sur l'état de la chaîne,

les messages d'information qui sont de type Event ou Trace.

Plus généralement, les fonctions externes appelées par l'OPS sont regroupées dans des classes communes identifiées dans ce document. Ceci est le cas en particulier pour les librairies METOP qui sont utilisées pour les calculs géométriques dans les algorithmes.

De plus, cette approche nous offre une solution pour traiter 2 particularités d'implémentation de l'OPS :

l'utilisation de librairie externes non thread-safe. L'encapsulation permet alors de gérer les appels concurrents de plusieurs threads à une même fonction par la mise en place de mutex assurant que les ressources de la librairie sont accédées de manière unique à un instant donnée. Ce mécanisme doit être mis en place en particulier pour

l'utilisation de la librairie METOP.

le fonctionnement en stand–alone. l'encapsulation permet un comportement différent du logiciel vis à vis des interfaces suivant qu'il est appelé dans un contexte ou dans un autre.

Plus généralement, pour les raisons évoquées précédemment, le processus de conception de l'OPS a permis de regrouper les fonctions de base et les fonctions d'accès aux interfaces externes dans des librairies centralisées spécifiques. Ceci est le cas pour les accès aux produits et aux données de configuration mais aussi pour les fonctions algorithmiques de base.

4.3.3.Fonctionnement en Standalone

Le mode standalone de l'OPS correspond à la possibilité d'activer le système en dehors du contexte du PGF. Ce fonctionnement implique donc d'inactiver ou de simuler l'accès aux interfaces externes suivantes : HKTM, LogEvent, LogTrace, Commandes, Cmd_TM et Stage_TM. Pour cela, les classes associées à ces interfaces doivent, dans ce contexte avoir un comportement radicalement différent du comportement opérationnel.

Les traitements associés à ces interfaces en mode standalone sont les suivants :

les commandes sont transmises à l'OPS à l'aide de fichier ASCII. L'OPS en mode standalone scrute l'apparition d'un fichier particulier contenant la commande. Dès l'apparition du fichier, celui-ci est analysé afin d'en extraire la commande qui est exécutée par l'OPS. Le message émis suite à l'acceptation de la commande (Command_TM) donne lieu lui aussi à la création d'un fichier dans ce répertoire. De façon similaire, le message Stage_TM donné lieu à création d'un fichier.

appels aux services *initialise* et *terminate*, utilisés systématiquement au début et à la fin de chaque processus sont dans ce contexte remplacés par des muets,

émissions de HK-TM et de messages (LogEvent, LogTrace), sont redirigées dans des fichiers locaux. Ceci permet ainsi la surveillance de l'OPS dans un contexte simple au travers de la consultation de fichiers. On considère ici qu'un seul fichier est crée pour un run et pour chaque type d'interface.

L'opérateur de l'OPS en mode standlone a en charge de purger les fichiers créés par l'OPS.

La figure ci-dessous illustre dans cet environnement, la solution proposée par THALES IS.

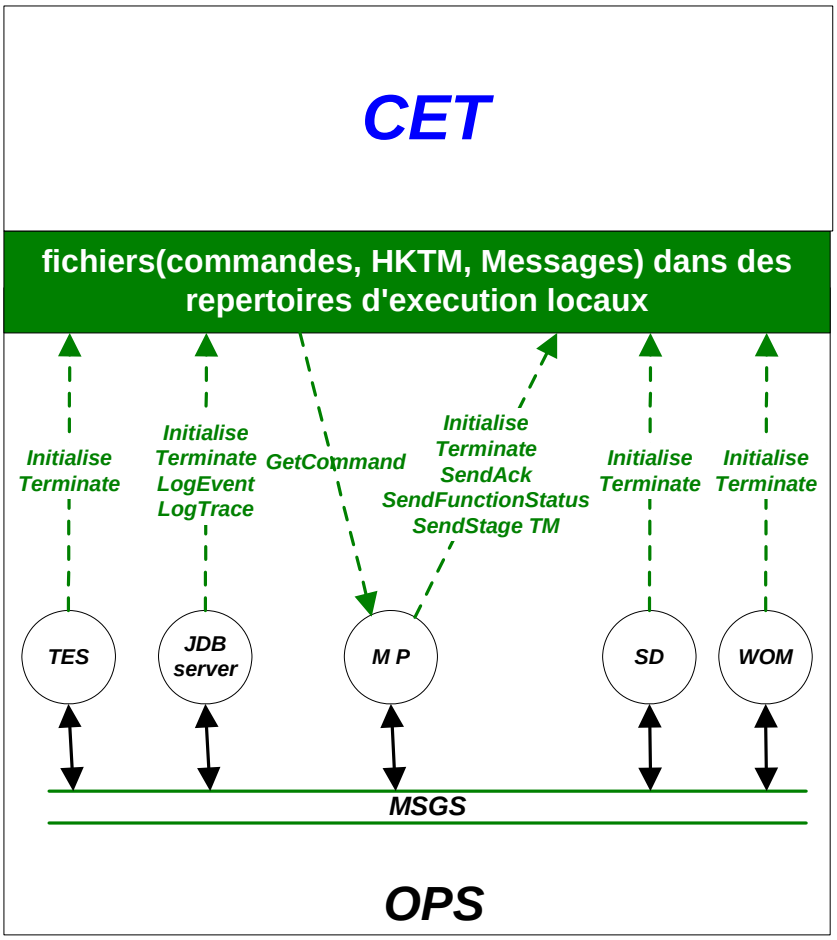


Figure 13 : Interfaces du mode Standalone

Le fonctionnement en standalone est activé en fonction de la valeur d'une variable d'environnement dédiée. Cette variable permet de configurer le fonctionnement des couches de services de l'OPS. La stratégie d'encapsulation décrite au paragraphe précédent permet de faciliter l'implémentation de cette solution.

Il faut cependant souligner que hors du contexte du CGS, certains services sont cependant indispensables au fonctionnement :

la librairie mathématique ESSL.

4.3.4.La Gestion des Traces

En mode DEBUG, l'OPS génère des traces (*pgf_TraceInfo*) supplémentaires qui pourront être utilisées en phase de maintenance. Afin de ne pas dégrader les performances de l'OPS par le fonctionnement en mode Debug, nous avons choisi de gérer le code spécifique mis en place pour les traces à l'aide d'une variable de compilation.

Par conséquent, pour chaque processus de l'OPS, 2 exécutables sont installés :

un exécutable compilé avec traces (mode Debug),

un exécutable compilé sans trace (mode Nominal).

Le shell de démarrage de chaque processus lance l'exécutable adéquat en fonction du mode d'exécution (variable d'environnement). On peut ainsi considérer dans ce mode de fonctionnement qu'une fois le traitement activé, les messages de traces inclus dans le code seront émis.

L'analyse des exigences nous a permis d'identifier des besoins distincts selon les processus :

- le volume des traces générées par les processus « de support » (MP, WOM), est peu important; ces traces correspondent à des informations permettant un premier niveau d'investigation en cas d'anomalie,

- pour le processus de traitement algorithmique SD , il est primordial d'avoir accès aux résultats intermédiaires issus des algorithmes à des fins de validation et d'investigation. Dans ce cas, le volume des informations à générer peut être très important ce qui pourrait entraîner la saturation du serveur de messages MSGS.

Ces deux besoins nous conduisent à proposer deux solutions distinctes :

pour les traces non algorithmiques, le volume et la nature des traces permet l'utilisation de l'API LogTrace formaté suivant le format décrit dans le document [DA17]. Notons ici que pour des raisons de synchronisation, le processus JDBs centralise les appels à cette fonction et récupère les traces émises par l'OPS via le bus logiciel. On retrouve à ce niveau, les informations suivantes (numéro de ligne, nom du fichier) ainsi qu'une chaîne de caractère formaté en début de chaque message. Le type des messages émis permet d'amener des précisions par rapports aux autres messages (LogEvent). Ainsi en cas de messages de niveau erreur ou warning on pourra préciser ici le contexte de l'évènement (numéro de ligne, pixel, arbre d'appel ...).

pour les traces algorithmiques un mécanisme spécifique a été choisi. Il consiste à référencer via l'API LogTrace, un fichier de debug algorithmique au format définit dans [DA107]. Ainsi toutes les informations volumineuses sont directement écrites par le processus de traitement en utilisant un mécanisme simple permettant de filtrer les traces choisies. Le principe de base consiste à prévoir l'écriture de toutes les entrées/sortie des algorithmes identifiés dans le document [DA7].

Le Fichier de debug algorithmique est conçu de manière à être utilisé de façon intensive en validation, il ne contient aucun message issus d'erreurs ou de warning, ceux-ci étant centralisés dans le fichier de trace. Un répertoire dédié est dés à présent prévu dans l'arborescence locale (/debug) pour stocker ce type de fichiers.

Pour une utilisation de la chaîne en mode granule nous proposons de créer :

un fichier unique de debug via l'API LogTrace, ce fichier est enrichi pour chaque granule,

un fichier de traces algorithmiques par granule, ce fichier est crée lors de l'initialisation du traitement.

Dans le cas du mode dump, on génère ainsi plusieurs fichiers dont le nom contient le numéro du granule.

Remarque : Le nettoyage du répertoire est une procédure opérationnelle.

4.4.LA CONFIGURATION DE L'OPS

4.4.1.Présentation

La configuration de l'OPS utilise 2 mécanismes :

des variables d'environnement,

des fichiers de paramètres.

Les variables d'environnement sont utilisées pour stocker les paramètres étant susceptibles d'être modifiés de façon opérationnelle entre deux exécutions de l'OPS. Il s'agit, pour la plupart, de variables à caractère informatique. Le stockage par fichier est plutôt utilisé pour stocker les valeurs des paramètres sur lesquelles un opérateur n'a pas à agir.

La liste et la valeur des variables d'environnement seront définies au cours de la Conception Détaillée.

Les fichiers de configuration identifiés en Conception Préliminaire est la suivante :

les fichiers de paramétrages de la parallélisation des traitements.

4.4.2.Fichiers de Paramétrage de la Parallélisation du Traitement

Le SD utilise 2 fichiers de configuration pour décrire la parallélisation des algorithmes de traitement sur les threads de calcul :

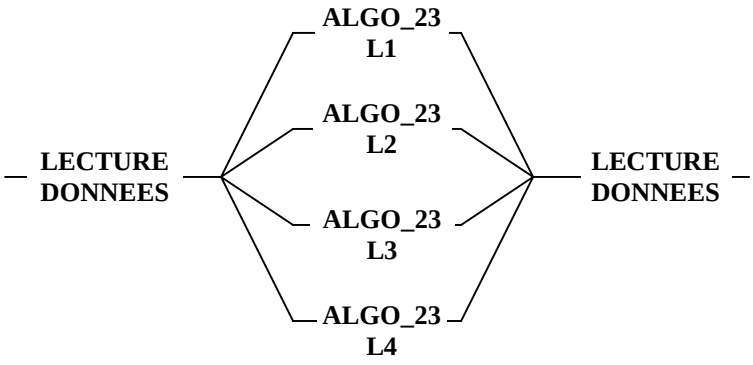
ModeleTraitement.txt : fichier de description découpage du traitement en tâches,

ModelesTaches.txt : fichier de description des taches logiques (une tache logique = une liste d'actions) ;

4.4.2.1.ModeleTraitement.txt

Ce fichier décrit le diagramme d'enchaînement des tâches nécessaire pour effectuer le traitement dans une architecture multithreadée. Ce diagramme permet de modéliser des exécutions en séquence, en parallèle ainsi que la synchronisation de tâches.

Le schéma suivant montre un exemple de diagramme d'enchaînement à modéliser : une lecture des données suivie d'une exécution en parallèle de l'Algo_23 sur 4 threads de calcul puis de l'écriture des données produites.



La syntaxe du fichier de configuration utilisé pour stocker un diagramme est la suivante :

[NOM DU TRAITEMENT]
NOMBRE TACHES=xxx
TACHE1=xxx
TYPE TACHE1={ RDV | LIGNE | AUCUN }
…
TACHEn=xxx
TYPE TACHEn={ RDV | LIGNE | AUCUN }

Le paramètre <TYPE TACHE> définit le mécanisme de synchronisation entre les tâches :

une tâche de type RDV (RenDez-Vous) est une tâche qui n'autorise pas le lancement d'autres tâches en parallèle,

une tâche de type LIGNE est une tâche dupliquée en autant d'occurrences que de lignes à traiter ; ces tâches sont exécutées en parallèle,

une tâche de type AUCUN est une tâche d'occurrence 1 et qui peut être exécutée en parallèle d'autres tâches (de même type ou de type LIGNE).

A l'aide de ce formalisme, le diagramme présenté en exemple se modélise par 3 tâches :

1 tâche LECTURE_DONNEES de type RDV,

1 tâche ALGO_23 de type LIGNE,

1 tâche ECRITURE_DONNEES de type RDV,

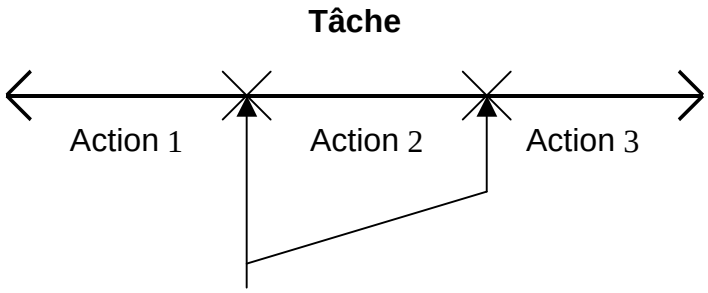
qui sont décrites de la façon suivante dans le fichier de configuration :

```
[0-1A]
NOMBRE TACHES=3
TACHE1=LECTURE_DONNEES
TYPE TACHE1=RDV
TACHE2=TACHE_ALGO_23_CNE
TYPE TACHE2=LIGNE
TACHE3=ECRITURE_DONNEES

TYPE TACHE3=RDV
```

4.4.2.2.ModelesTaches.txt

Ce fichier décrit le découpage d'une tâche en une séquence d'actions. Ce découpage permet de positionner des points de contrôle entre les actions pour traiter le cas échéant des événements en provenance d'un autre thread.



Point de contrôle si fin de
la tâche demandée

Dans le contexte de l'OPS, une action implémentera un algorithme ou une séquence de plusieurs algorithmes, les points de contrôle permettront par exemple de traiter au plutôt les demandes d'arrêt en provenance du WOM.

La syntaxe de ce fichier de configuration est la suivant :

```
[NOM DE LA TÂCHE]
NOMBRE ACTIONS=xxx
ACTION1=xxx
…
ACTIONn=xxx
```

Dans le cadre de l'OPS, seules des taches comprenant une action sont envisagées.

La table suivante présente un exemple de fichier de configuration de type ModelesTaches.

```
[TACHE_ALGO_1]
OCCURRENCE=Aucune
NOMBRE ACTIONS=1
ACTION1=Algo_00_INI
```

5.ARCHITECTURE DE CLASSES

5.1.L'ARCHITECTURE GENERALE

L'OPS est structuré en paquetages suivant le découpage en processus applicatifs décrit au § 4.1.1. Pour chaque processus un paquetage est ainsi crée, ce paquetage peut éventuellement être re-découpé en sous paquetages (cas du SD) en fonction de la complexité. On rajoute ici un paquetage commun permettant de regrouper les services utilisés de manière globale par plusieurs processus.

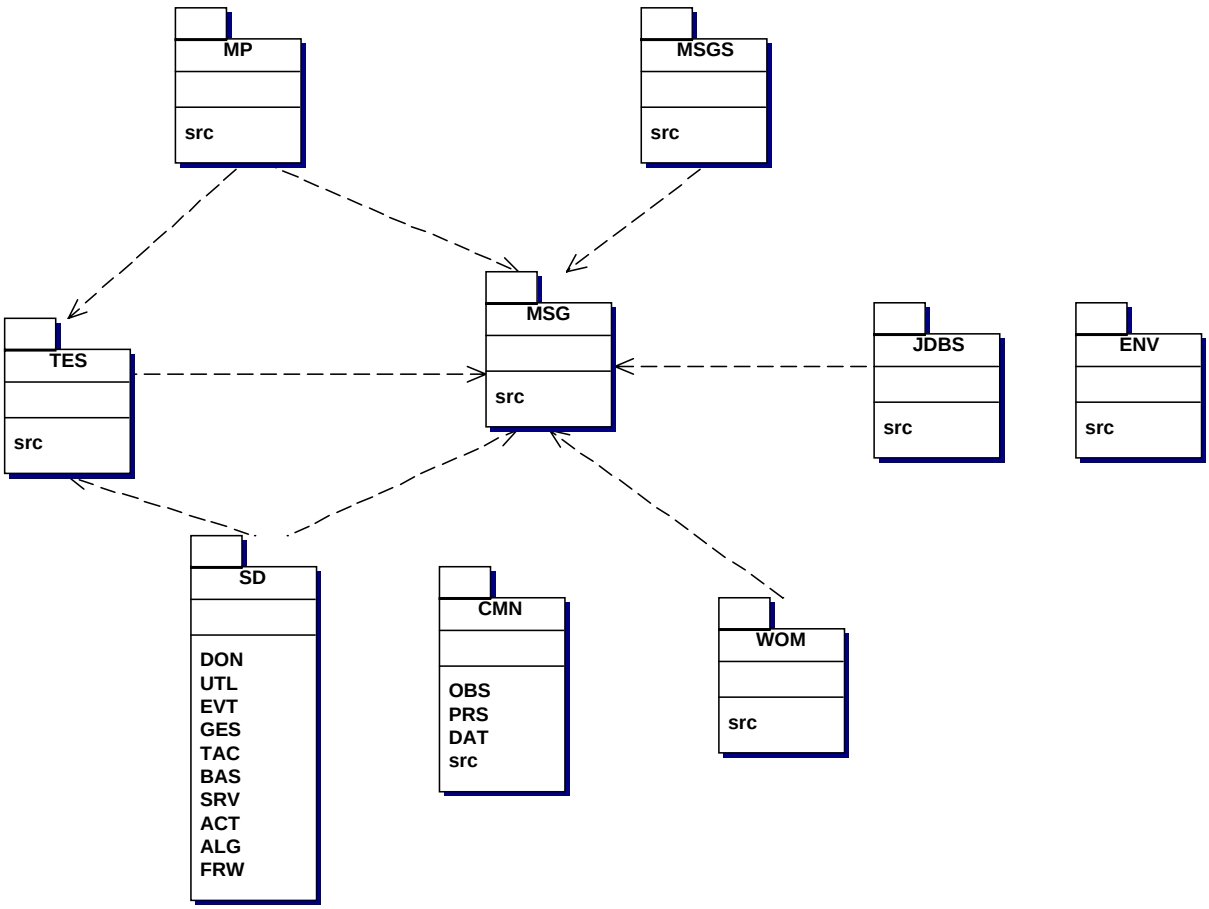


Figure 14 : Architecture Globale de l'OPS

5.2.LES PROCESS APPLICATIFS

5.2.1.Le Main Process

5.2.1.1.Description

La présence d'un MP est imposée dans l'architecture de chacun des sous-systèmes du CGS. Le « processus principal » MP (Main Process) est chargé de faire l'interface entre le MLA (Main Local Agent), processus du PGF qui assure la collecte et la diffusion du flot de Monitoring & Control et le sous-système à contrôler.

Lors de sa mise en exécution, le MP récupère dans des variables d'environnement le mode de traitement courant (*Nominal* ou *Debug)* ainsi que le sous-système qui le contrôle (*CGS* ou *Standalone)*.

Le MP a la responsabilité :

de lancer les processus nécessaires pour assurer les fonctionnalités du sous-système,

de collecter et de diffuser les commandes émises par le PGF,

de renvoyer les acquittements des commandes issues du PGF,

de prévenir le PGF de la mise à disposition de fichiers de données (produits, rapport, ..), afin que celui-ci les rapatrie,

de collecter et de envoyer la HKTM au MCS,

de coordonner l'arrêt des processus à la fin du traitement du dump, sur commande du PGF ou sur anomalie grave.

Gestion des commandes et des compte-rendus :

Le MP diffuse les commandes émises par le PGF (STEP, RESUME, SUSPEND, BREAK, STOP) vers le processus WOM.

Les compte-rendus d'acquittement d'activation (Coded/Rejected) et d'exécution (Completed/Aborted) des commandes (STEP, RESUME, SUSPEND, BREAK, STOP) sont envoyés par le WOM ; ces compte-rendus sont transmis au PGF.

Calculs de l'état des ressources :

Le MP collecte les informations nécessaires à la constitution des statuts soit directement, soit via des messages spécifiques envoyés par le WOM et le SD.

La collecte directe s'effectue périodiquement, toutes les n secondes (configurable), sur réception d'un « événement timer » programmé.

La liste des informations nécessaires pour construire les statuts est la suivante :

PPF version : stocké en permanence au niveau du MP.

Operational Readiness : transmis par le processus WOM. Le WOM envoie un message *m_OperationalStatus* pour informer le MP de l'état fonctionnel de la chaîne de traitement (working, idle,..), soit suite au traitement d'une commande, soit à la suite de la réception d'un message de fin d'initialisation ou de traitement du SD.

les informations extraites des données au cours de traitement : **Start_time, Current sensing time, Instantaneous quality, Start orbit number, OK or FAILURE , Instantaneous cpu usage** : transmis par le SD. Le SD envoie au MP un message *m_PipelineStatus* toutes les n secondes (configurable).

5.2.1.2.Diagramme de Classes

Le diagramme de classes de la figure suivante présente l'architecture de classes qui implémente le processus MP.

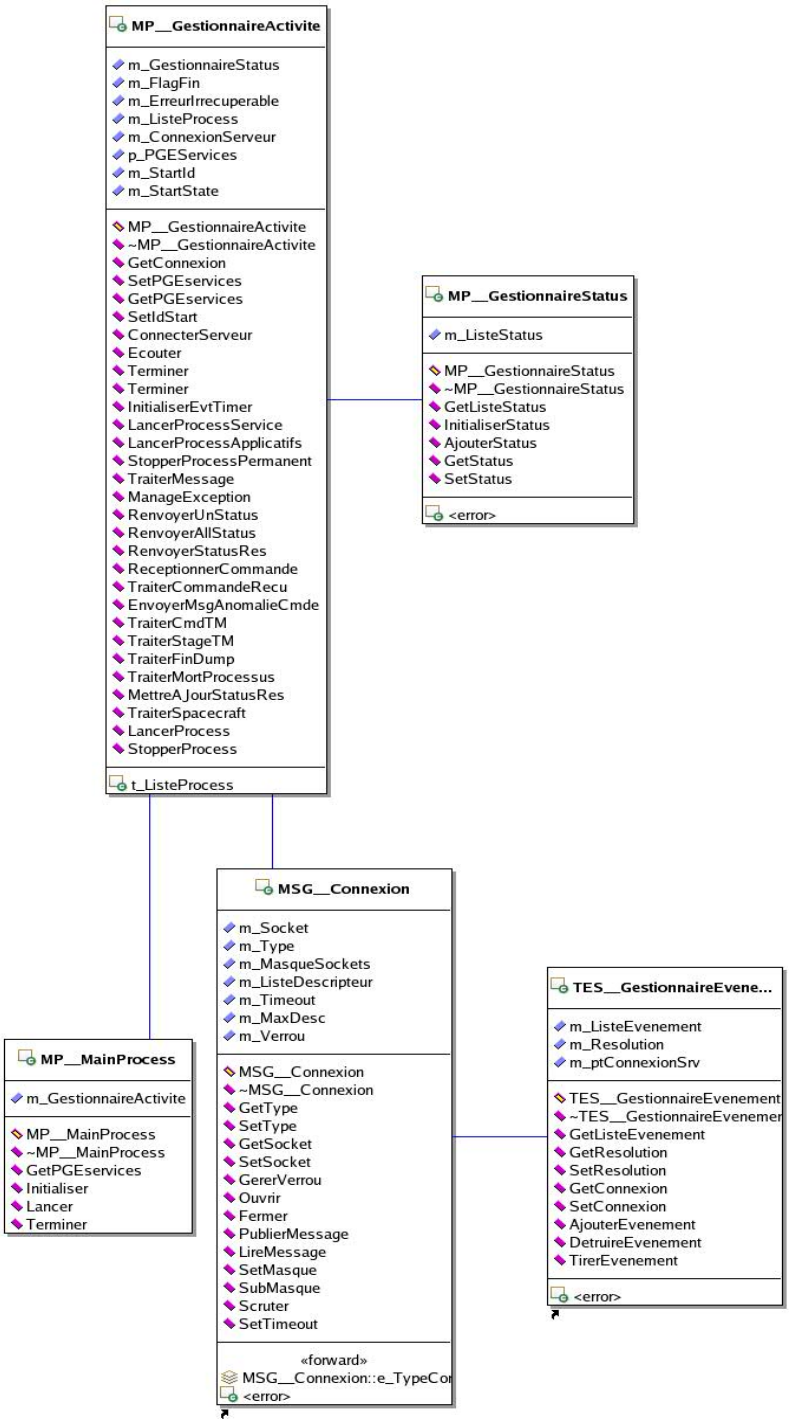


Figure 15 : Diagramme de classes du Main Process

Une instance de la classe **MP__MainProcess** est créée lors du lancement du processus MP. La méthode **MP__MainProcess.Initialiser** permet :

de créer le gestionnaire des activités du MP **MP__GestionnaireActivité** ;

de lire le numéro de port serveur et de créer la connexion serveur avec **MP__GestionnaireActivité.ConnecterServeur**.

La méthode **MP__MainProcess.Lancer** permet de lancer la boucle infinie d'écoute de l'activité sur la connexion avec le serveur qui est assuré par **MP__GestionnaireActivite.Ecouter**.

MP__GestionnaireActivite.Ecouter utilise **MSG__Connexion.Scruter** pour traiter les messages qui arrivent qui peuvent être de plusieurs types :

de type PGF_CMD pour diffuser les commandes en provenance du PGF ;

de type MSG_CMD_STATE pour renvoyer les acquittements des commandes au PGF ;

de type MSG_STAGE_TM pour informer le PGF de la mise à disposition de produits et rapports ;

de type MSG_PIPELINE_STATUS pour signifier l'évolution des statuts en provenance du SD ;

de type MSG_READINESS pour signifier l'évolution des statuts en provenance du WOM ;

de type MSG_FIN_DUMP en provenance du WOM pour signifier la fin du traitement complet ;

de type MSG_EVENT_TIMER qui permet de renvoyer la valeur courante des statuts de ressource au MCS ;

de type MSG_MORT en provenance de n'importe quel process afin de signifier son arrêt sur erreur ou anomalie grave.

Tous les messages qui arrivent sont traités par **MP__GestionnaireActivite.TraiterMessage** qui aiguille les traitements en fonction du type de message reçu.

MP__GestionnaireActivite.MettreAJourStatusRes utilise **MP__GestionnaireStatus** pour mettre a jour les valeurs courantes des statuts de ressources. Un statut reçu pour la première fois (**MP__GestionnaireStatus.InitialiserStatus** à l'état faux par défaut) est déclaré avec **MP__GestionnaireStatus.AjouterStatus**.

La méthode **MP__GestionnaireActivite.RenvoyerAllStatus** est activée par l'événement « EVT_EVENT_TIMER » qui cadence l'envoi des statuts de ressources vers le MCS. Cette méthode utilise la routine **MP__GestionnaireStatus.GetListeStatus** pour avoir la liste courante des statuts.

Les diagrammes de séquence des figures suivantes présentent :

le démarrage de l'OPS ;

l'arrêt de l'OPS ;

le traitement d'une commande ;

la génération de la HKTM.

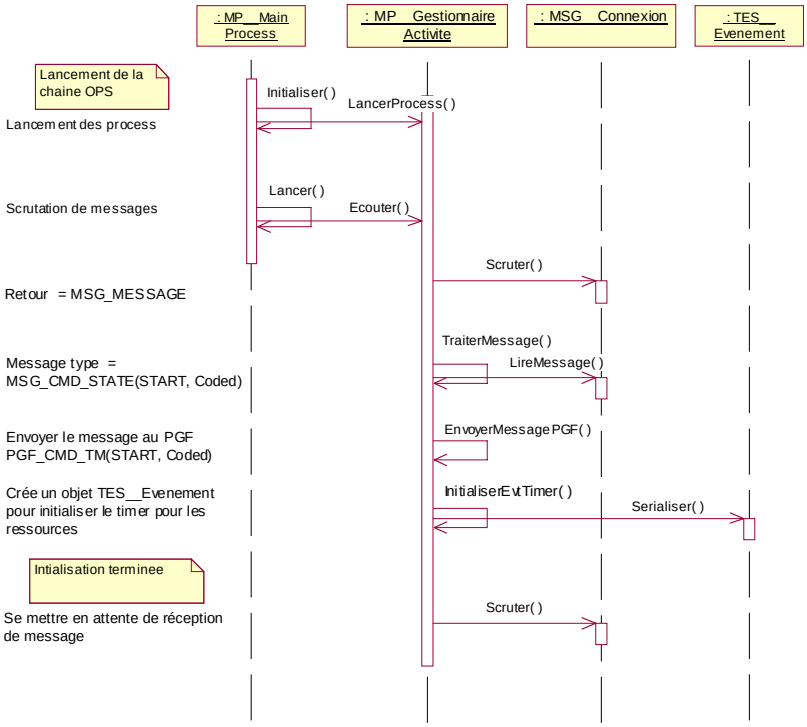


Figure 16 : Diagramme de séquence du démarrage de l'OPS

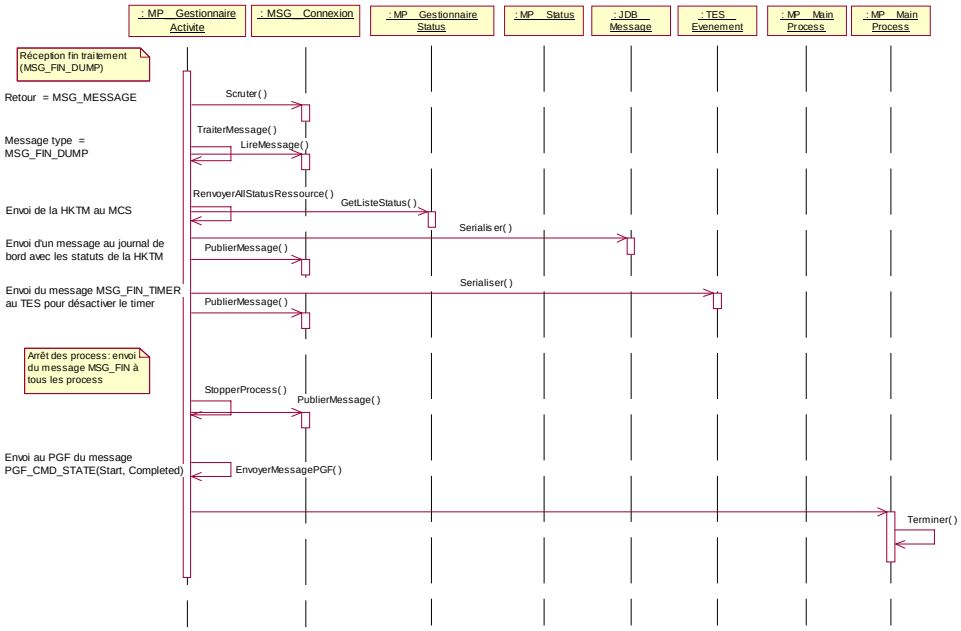


Figure 17 : Diagramme de séquence de l'arrêt de l'OPS en fin de traitement

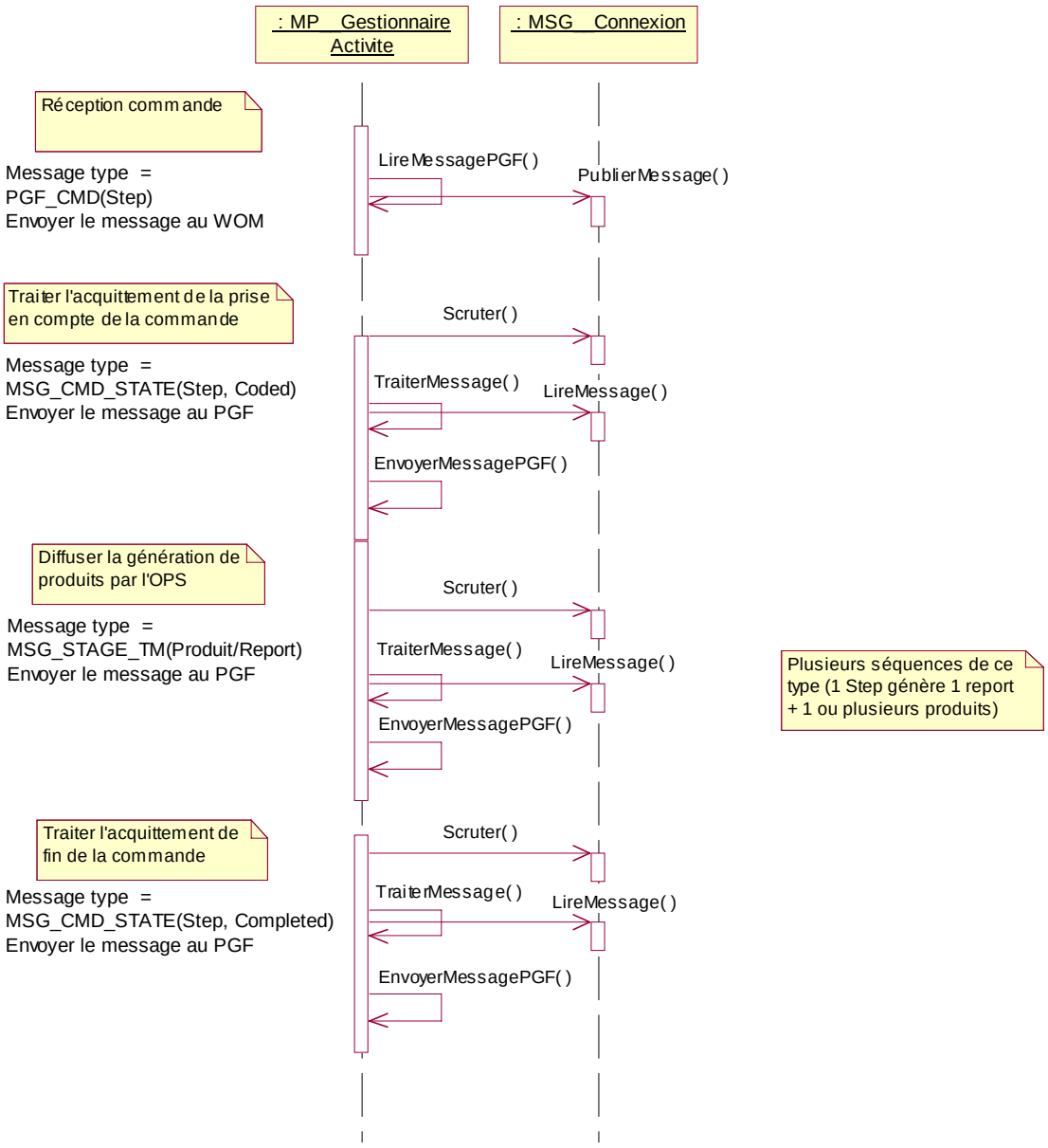


Figure 18 : Diagramme de séquence du traitement d'une commande

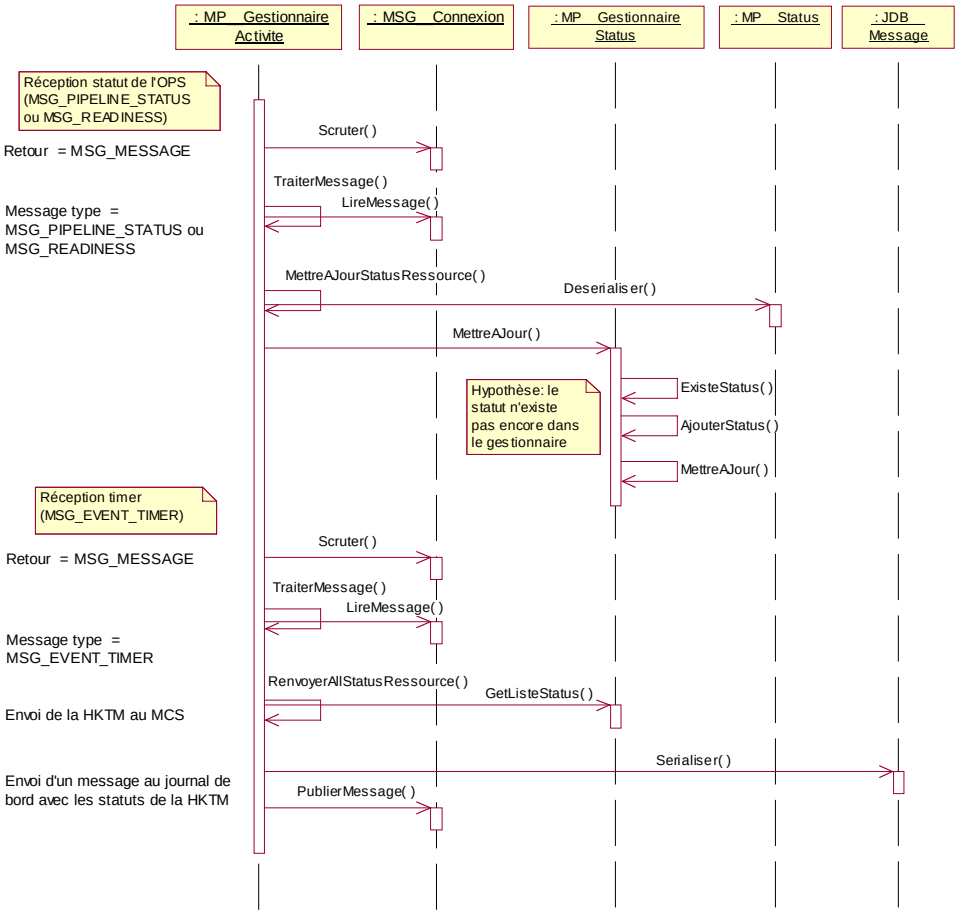


Figure 19 : Diagramme de séquence de la génération de la HKTM

5.2.2.Le Work Order Manager

5.2.2.1.Description

Le processus *Work Order Manager* (WOM) est chargé de gérer les Work Orders transmis par le PGF via le MP. Le processus WOM est un processus qui est lancé par le Main Process.

Le WOM a la responsabilité :

de nettoyer au démarrage l'espace de travail ;

d'analyser les fichiers Work Order transmis par le MP ;

de traiter les Work Orders et de contrôler leur exécution par le SD ;

d'informer le MP de la génération des produits et de la fin des traitements.

Le WOM reçoit et traite les commandes gérant la production transmises par le MP (STEP, RESUME, SUSPEND, BREAK, STOP). Dans le cas du STEP, la commande fait référence à un fichier particulier, le Work Order. Ce fichier contient les paramètres nécessaires à une production de données IASI Level 1.

A sa réception, le fichier Work Order est parsé, les contrôles sémantiques sont effectués et les informations relatives à l'ordre de production sont récupérées. L'ordre de production (ou Work Order) se décompose en un ou plusieurs traitements (cas du mode dump). Un traitement est l'entité élémentaire envoyée au processus produisant les données, le Serveur de Données. Le WOM supervise cette production.

Tous les Work Orders sont stockés dans un « gestionnaire de Work Orders », et sont gérés comme une file de type FIFO. Le WOM utilise ce gestionnaire pour mettre en exécution un traitement, l'interrompre, le suspendre ou le relancer.

Dès qu'un traitement est terminé, le WOM en est averti par le Serveur de Données et il enchaîne :

la mise en exécution du traitement suivant de la liste FIFO;

le traitement des produits générés précédemment (Report, Stage-TM,…).

Le WOM informe le MP, à l'aide du message MSG_READINESS, à chaque changement de son état fonctionnel (working, idle, error). Ce changement dépend des commandes reçus et/ou de l'état de la production.

De plus, le WOM détecte le dernier granule d'un dump dont la fin de traitement implique l'arrêt de l'OPS. Une fois ce granule traité, le WOM envoie un message MSG_FIN_DUMP au MP qui se charge ensuite de l'arrêt de l'OPS.

5.2.2.2.Diagramme de Classes

Le diagramme de classes de la figure suivante représente l'architecture du processus WOM en terme de classes.

Figure 20 : Diagramme de classes du Main Process

Une instance de la classe **WOM__ServeurWorkOrder** est créée lors du lancement du processus WOM. La méthode **WOM__ServeurWorkOrder.Initialiser** permet :

de créer le gestionnaire des activité du WOM : **WOM__GestionnaireActivite** ;

de lire le numéro de port serveur et de créer la connexion serveur avec **WOM__GestionnaireActivite.ConnecterServeur**.

La méthode **WOM__ServeurWorkOrder.Lancer** permet de lancer la boucle infinie d'écoute de l'activité sur la connexion avec le serveur qui est assurée par **WOM__GestionnaireActivite.Ecouter**.

WOM__GestionnaireActivite.Ecouter utilise **MSG__Connexion.Scruter** pour traiter les messages qui arrivent qui peuvent être de plusieurs types :

de type MSG_CTRL pour traiter les commandes en provenance du MP ;

de type MSG_FIN_INIT en provenance du SD pour signifier la fin de sa phase d'initialisation ;

de type MSG_FIN_TRAITEMENT en provenance du SD pour signifier la fin d'un traitement ;

de type MSG_FIN en provenance du MP pour signaler une demande d'arrêt du processus.

Tous les messages qui arrivent sont traités par la routine **WOM__GestionnaireActivite.TraiterMessage** qui active les traitements en fonction du type de message reçu.

Les traitements détectés dans le fichier Work Order sont déclarés au gestionnaire de traitements avec **WOM__GestionnaireWorkOrder.AjouterWorkOrder**.

La classe utilitaire **WOM__WorkOrder** est utilisée par tous les ordres de production pour générer et formater le rapport de production correspondant. **WOM__WorkOrder.CreerRapport** permet de charger le modèle du rapport qui contient toutes les structures fixes du rapport. Les éléments variables repérés par $<nom du paramètre> dans le modèle sont renseignés par les Work Orders lors du formatage du rapport en utilisant la routine **WOM__WorkOrder.MettreA jour**.

Les diagrammes de séquence des figures suivantes présentent :

la réception d'un Work Order ;
le traitement d'une fin de production ;
la gestion de la production en mode Dump.

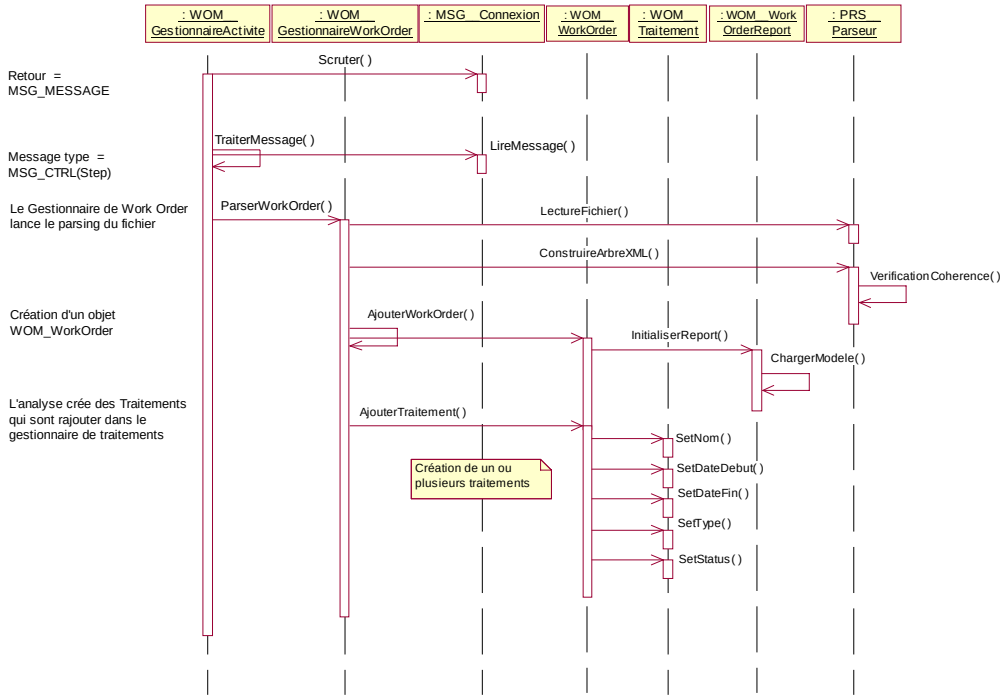


Figure 21 : Diagramme de séquence de réception d'un Work Order

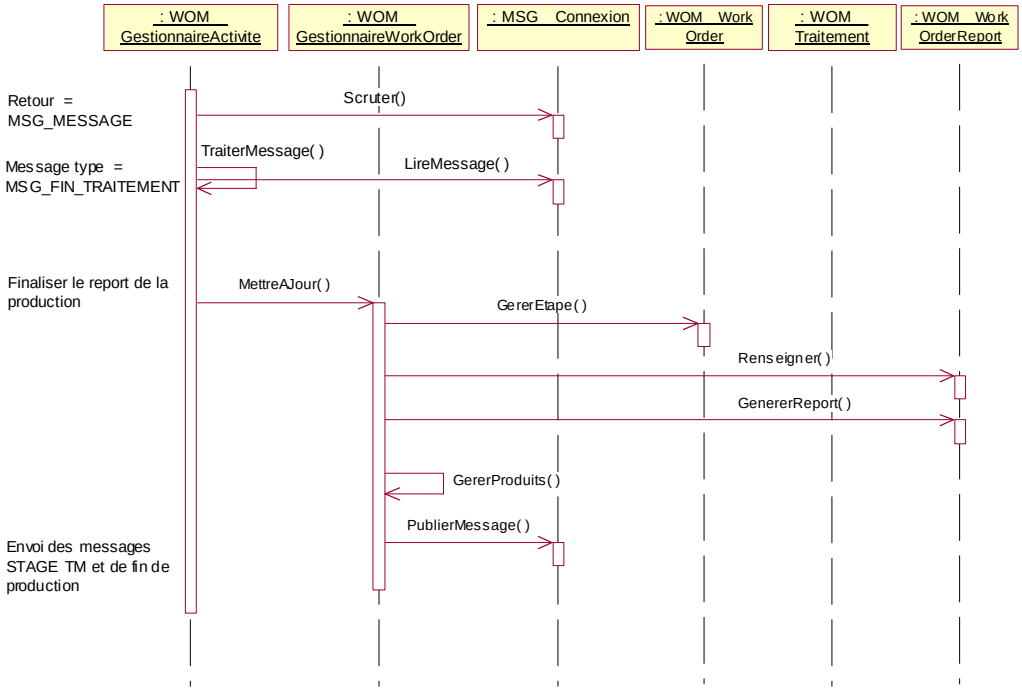


Figure 22 : Diagramme de séquence du traitement d'une fin de production

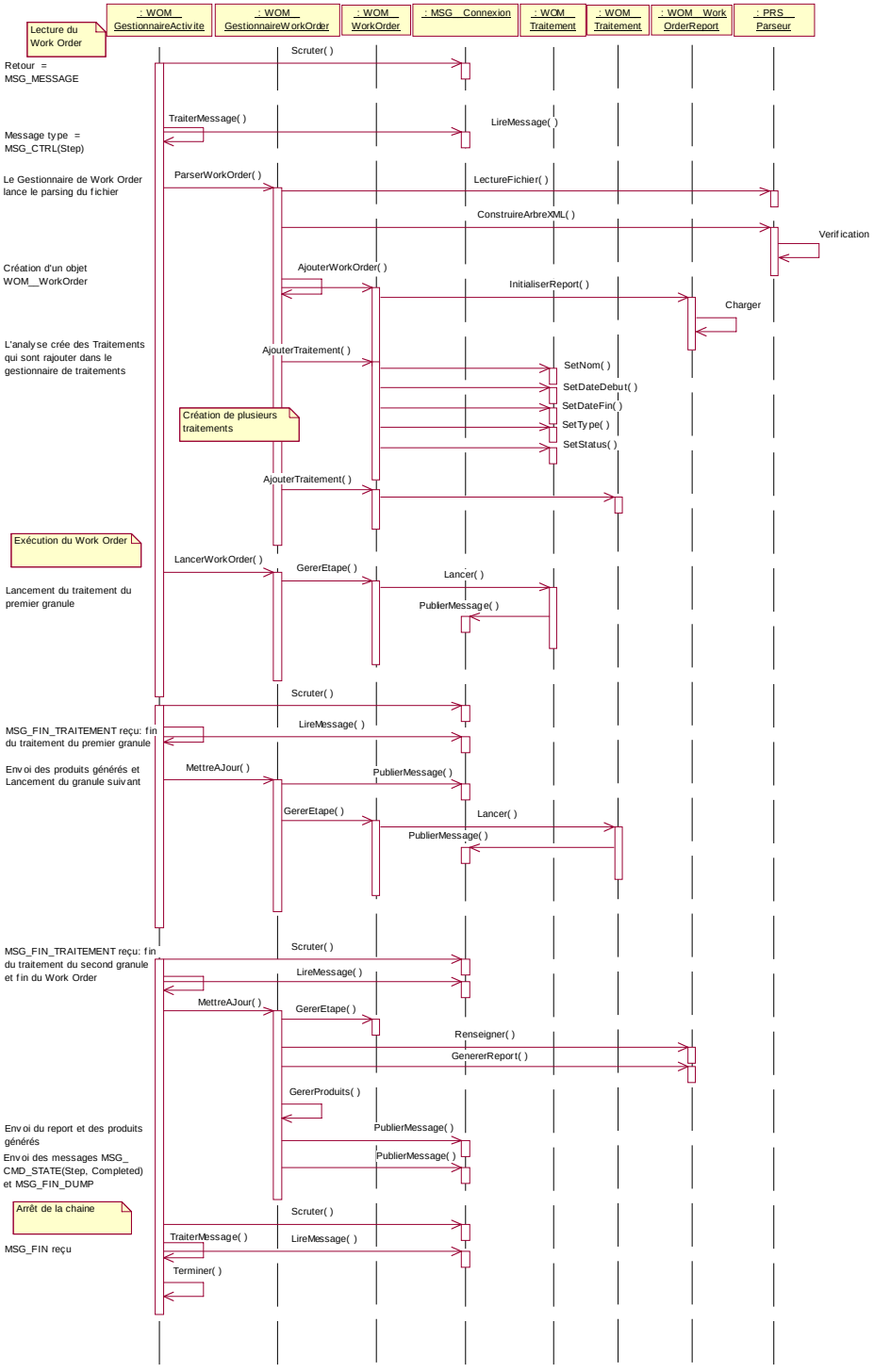


Figure 23 : Diagramme de séquence de la production en mode Dump

5.2.3.Le Serveur de Données

5.2.3.1.Description

Le composant *Serveur de Données* a en charge le traitement d'un granule de données IASI. Il encapsule les fonctionnalités de parallèlisation du traitement d'un granule et la chaîne de traitement IASI de niveau 1. Il reçoit les ordres de production et les ordres de traitement du composant *WO Manager*.

Il consiste en un processus multithreadé. Au lancement du processus, un ensemble de threads est créé, un des threads est spécialisé dans l'écoute et le traitement des interfaces externes (thread monitoring), les autres threads (thread principal et threads de calcul [nombre paramètrable]) sont dédiés aux tâches opérationnelles et sont dans l'état suspendu par défaut.

Lors de la réception d'un ordre de production, le *Serveur de Données* découpe cet ordre en tâches suivant le diagramme d'enchaînement associé (cf §4.4.2) et les place dans un échéancier. Lorsqu'une tâche doit être lancée, le processus principal recherche un thread libre dans l'ensemble des threads, lui affecte la tâche puis demande au thread de s'exécuter. Lorsque la tâche se termine, le thread repasse dans l'état suspendu et devient disponible pour une autre tâche.

L'intérêt de cette approche est d'être à la fois simple tout en étant très efficace. Elle permet de maîtriser le nombre de threads qui peuvent s'exécuter en parallèle tout en évitant les risques de création et destruction permanente de threads consommateurs de ressource et de temps.

Le composant *Serveur de Données* est donc implémenté à l'aide d'un processus multi-threadé, basé sur les threads POSIX dont l'architecture est reprise du logiciel SSALTO. Les threads mis en place sont les suivants :

le thread principal lance et supervise les différents traitements ;

le thread de monitoring traite les requêtes émanant du process *WO Manager* ;

un ensemble (pool) de threads de calcul qui sont dédiés aux tâches proprement dites. Ce nombre de thread est paramétrable au lancement du process.

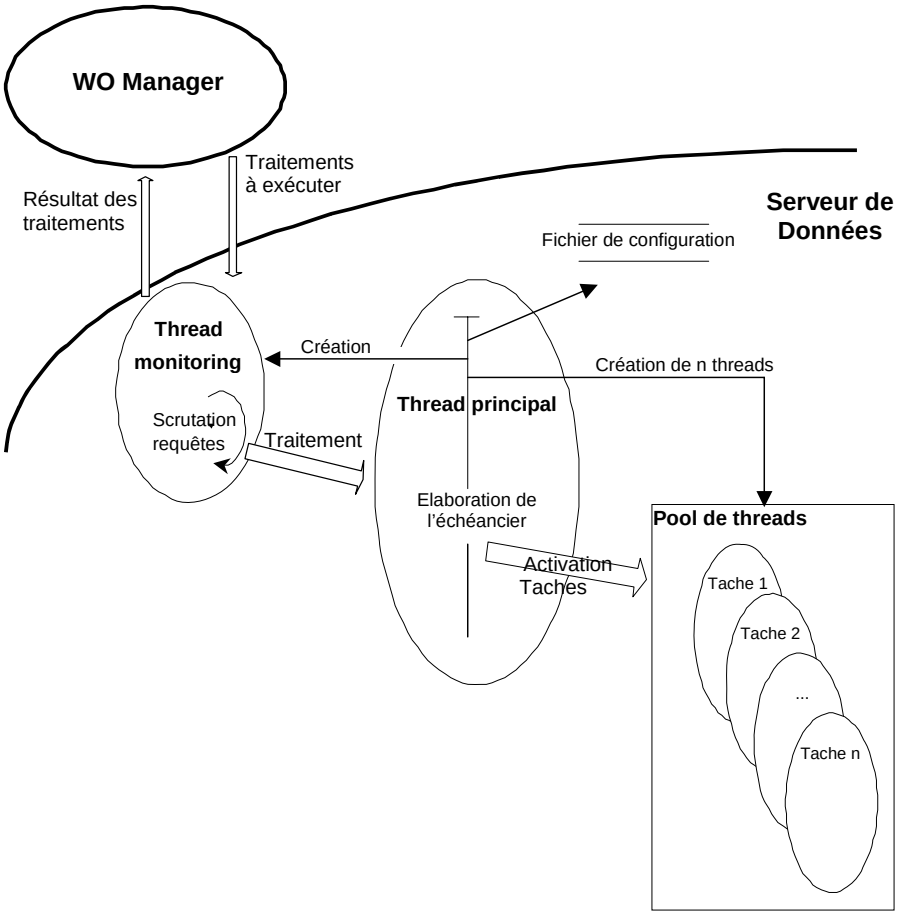


Figure 24 : Architecture du Serveur de Données

5.2.3.2.Diagramme de Classes

5.2.3.2.1.Diagramme des Paquetages

Le SD est constitué de 11 paquetages :

7 paquetages sont dédiés à la gestion des tâches génériques , des actions génériques et à la surveillance de l'exécution des tâches dans l'architecture multithreading. Ces paquetages sont récupérés de l'architecture multithreading du SD-SSALTO :

ACT : gestion des actions composant les tâches ;

EVT : modules de gestion des événements ;

FRW : base (framework) du SD ;

SRV : modules des serveurs associés au SD ;

TAC : gestion des tâches ;

UTL : modules regroupant des utilitaires (simulateur d'un client, ...)

l'utilisation de l'architecture SSALTO, implique de rajouter des paquetages en charge de spécialiser les concepts génériques de SSALTO à l'OPS ainsi que des paquetages spécifiques pour la gestion des données OPS :

GES : spécialisation par dérivation des actions et tâches génériques de SSALTO en taches algorithmiques OPS,

ALG : algorithmes de bases au sens du document [DA7],

DON : classes de gestion des accès aux données,

BAS : modules et classes de services communs spécifiques à l'OPS.

La figures suivantes présente l'architecture de des paquetages du SD et leur interfaçage.

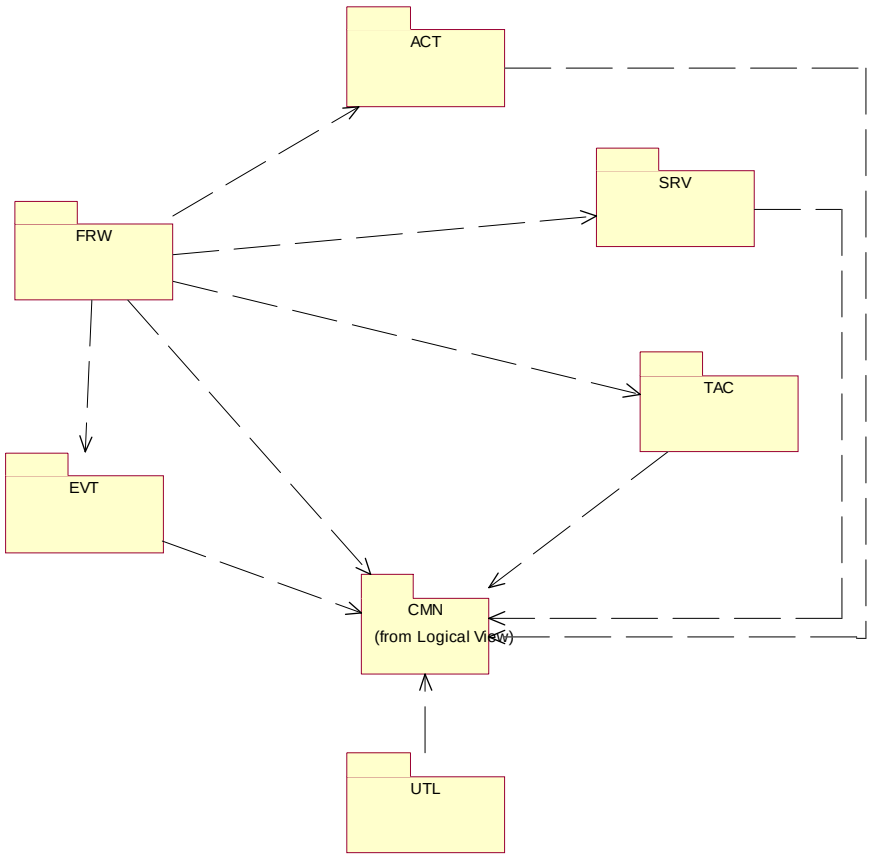


Figure 25 : Paquetages Génériques de Gestion des Tâches/Actions

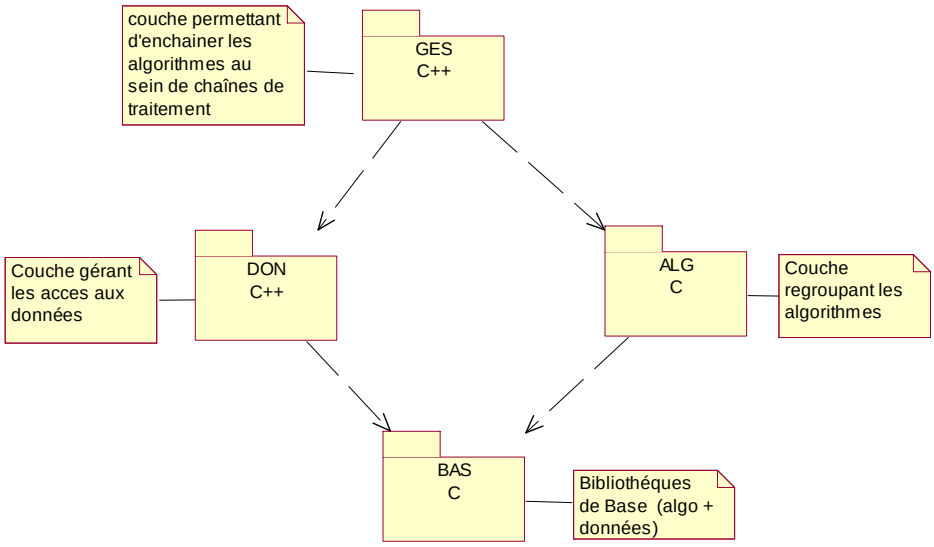


Figure 26 : Paquetages des Traitements Algorithmiques OPS

La liste exhaustives des classes constituant ces paquetages sont présentés au §5.2.3.5.

5.2.3.2.2.Diagramme des Principales Classes du SD

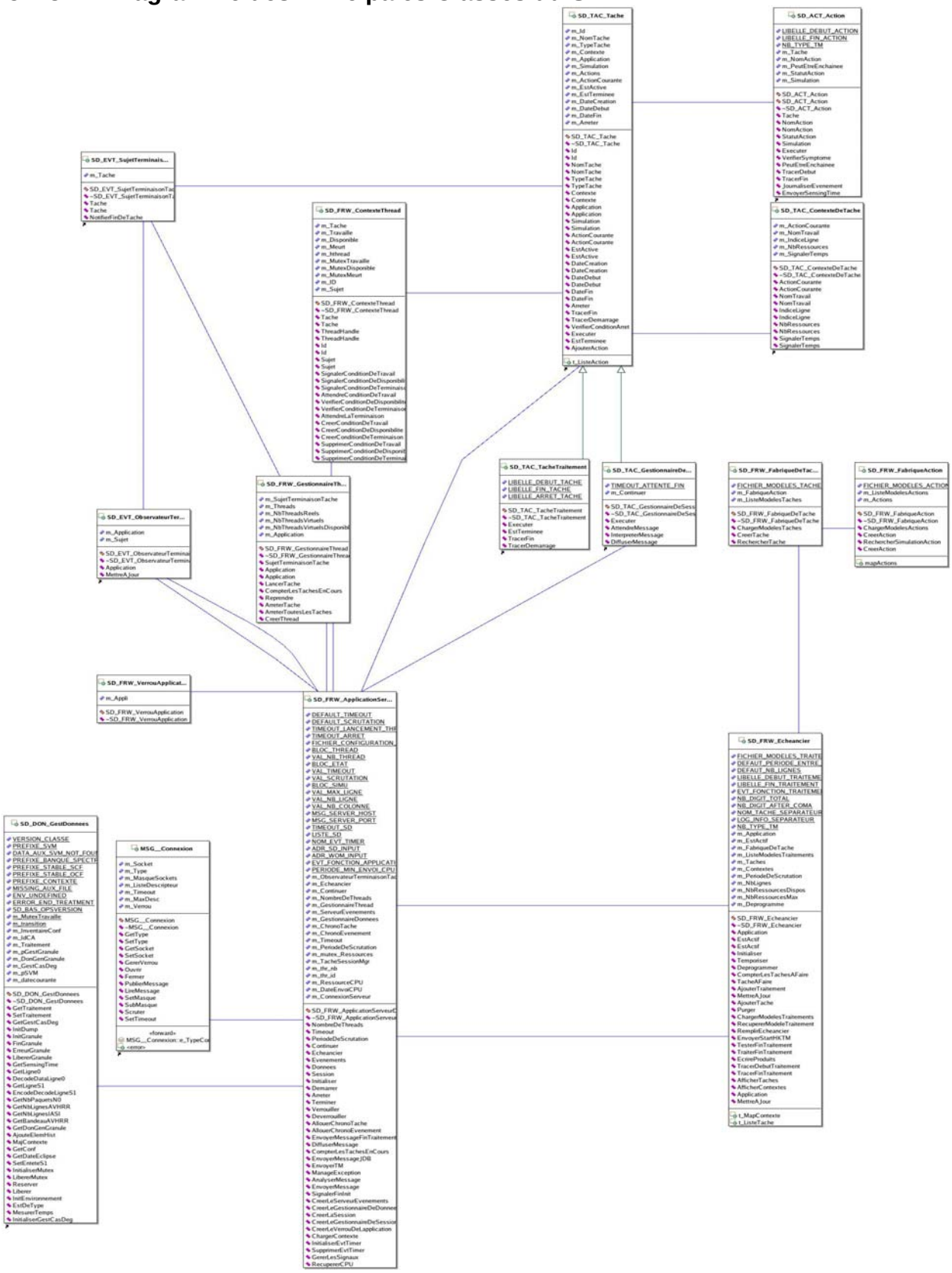


Figure 27 : Diagramme de classes général du Serveur de Données

Ce diagramme représente le canevas du composant **Serveur de Données**.

La classe **ApplicationServeurDonnee** est instanciée au démarrage de l'application. Elle se charge de créer le gestionnaire de threads **GestionnaireThreads**, le serveur d'événements **ServeurEvenements** et l'échéancier **Echeancier**.

La classe **GestionnaireThreads** gère un ensemble de contexte de threads **ContexteThread** qui encapsule toute la logique du multi-threading et de la synchronisation associée.

La classe **ApplicationServeurDonnee** crée également un observateur d'événements **ObservateurTerminaisonTache** attaché au **SujetTerminaisonTache** du gestionnaire de thread **GestionnaireThread**. Par ce mécanisme, chaque fois qu'une tâche opérationnelle se termine, l'observateur est notifié et reçoit par la méthode **MettreAJour** son sujet d'intérêt. L'instance **SujetTerminaisonTache** reçue en paramètre permet d'accéder via la méthode **Tache** aux propriétés de la tâche qui vient de se terminer. Le rôle de l'observateur est entre autre de diffuser un message de terminaison de tâche.

Après cette séquence d'initalisation, la classe **ApplicationServeurDonnee** crée une instance de **GestionnaireDeSession** qui est une tâche spécialisée dans la veille et l'interprétation des messages en provenance d'autres process, puis demande au gestionnaire de thread **GestionnaireThread** de lui affecter un thread de traitement via la méthode **LancerTache**.

Le **GestionnaireDeSession** se met en écoute des messages sur la session via la méthode **AttendreMessage**.

L'application principale du serveur de données est à partir de ce moment libérée des problèmes de surveillance des requêtes externes, une tâche étant spécialement dédiée à cette activité.

Nous présentons plus loin dans le document les diagrammes de séquence :

de lancement d'un traitement et d'une tâche à partir de l'échéancier ;

de terminaison d'un traitement ;

de traitement d'une requête d'arrêt d'un traitement.

5.2.3.2.3.Diagramme de Classes du Serveur d'Evénements

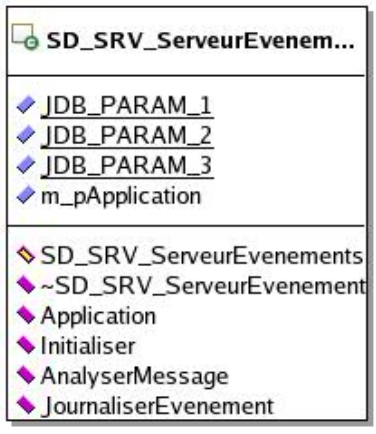


Figure 28 : Serveur d'Evénements

La classe **ServeurEvenements** est la classe qui permet la centralisation de la gestion des événements dans le Serveur de Données.

La méthode **JournaliserEvenement** est chargée d'envoyer un message au journal de bord**.** L'application serveur de donnée peut soumettre un message au serveur d'événements via la méthode **AnalyserMessage.**

5.2.3.2.4.Diagramme de Classes des Tâches et Actions

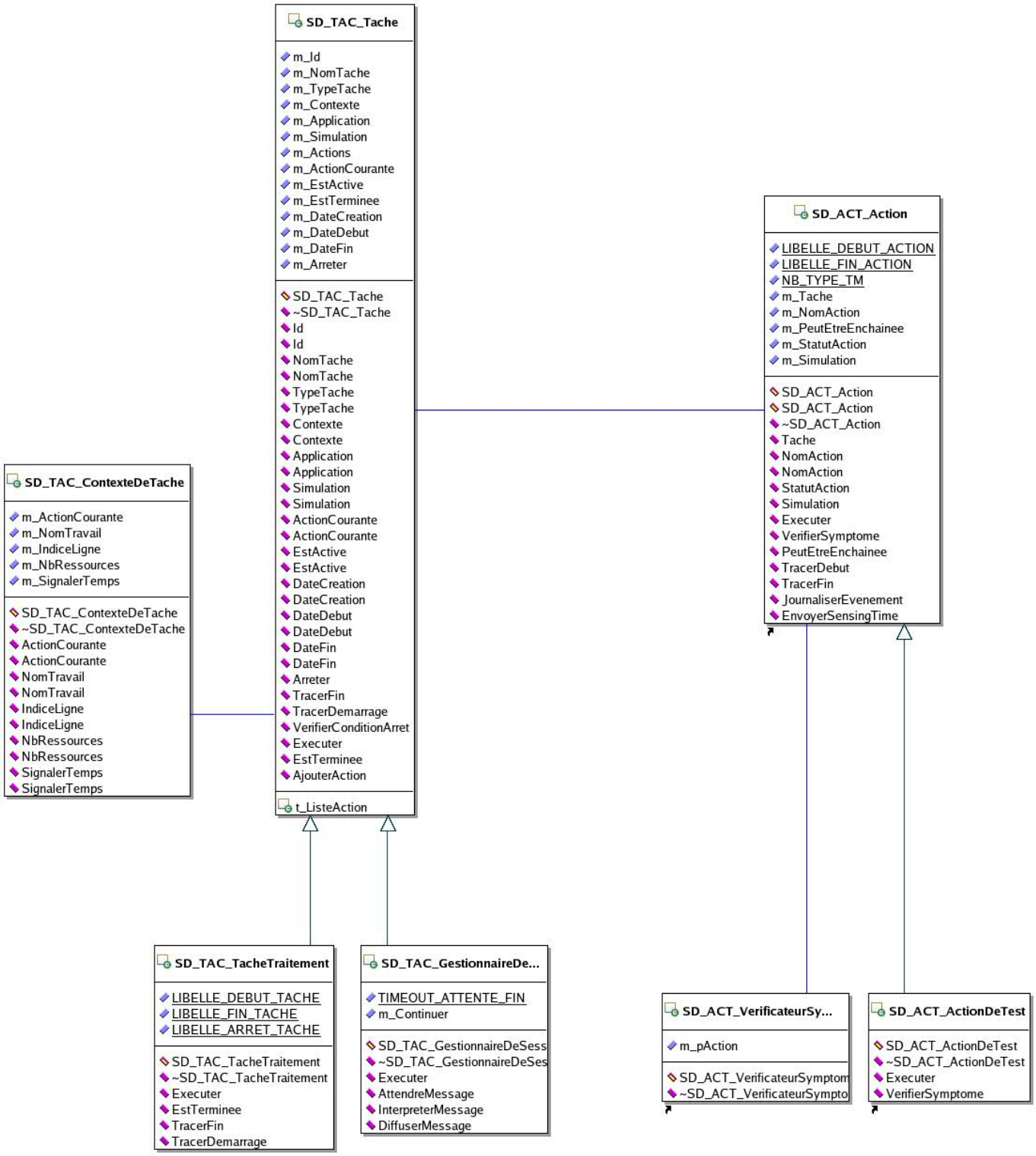


Figure 29 : Tâches et Actions

Tous les types de tâche héritent d'une classe commune et abstraite **Tache** qui :

trace dans le journal d'événement son démarrage et sa terminaison via les méthodes **TracerDemarrage** et **TracerFin** ;

effectue un traitement via la méthode virtuelle **Executer**.

L'accesseur **Application** permet d'invoquer les méthodes de l'application particulièrement **Verrouiller** et **Deverrouiller** qui encapsulent l'exclusion mutuelle des différents threads aux ressources de l'application. Une fois l'application verrouillée, l'instance de tâche utilise l'accesseur **Evenements** et invoque les méthodes appropriées des objets correspondants puis déverrouille l'application.

La classe **GestionnaireDeSession** évoquée précédemment est chargée de gérer et traiter les messages transitant par la session. La méthode **Executer** est constituée d'une boucle **AttendreMessage**, **InterpreterMessage**, **DiffuserMessage**. Cette classe délègue l'exploitation proprement dite des messages à la classe **ApplicationServeurDonnee**. La méthode **AttendreMessage** est bloquée sur la méthode **AttendreMessage** de l'objet **Session.** Comme la tâche **GestionnaireDeSession** est associée à un thread, le reste des activités du serveur de données se poursuit normalement.

La classe **TacheTraitement** est la classe de base de tous les traitements opérationnels. Elle est composée d'un ensemble d'actions opérationnelles à exécuter (**Action**). La méthode **Executer** de la classe **TacheTraitement** consiste à parcourir la collection des actions à exécuter et d'invoquer la méthode **Executer** associée à chaque action tant que la méthode **EstTerminee** retourne un statut correct.

Pour s'exécuter une instance de **TacheTraitement** s'appuie sur un **ContexteDeTache** préparé par la classe **Echeancier**. Le contexte de tâche contient principalement le numéro d'action courante (**ActionCourante**), l'indice de ligne à traiter et la liste des fichiers. Ces informations sont exploitées et mises à jour par chaque action. La méthode **Executer** d'une action peut faire appel à la méthode virtuelle **VerifierSymptome** qui permet d'effectuer un certain nombre de vérifications. Un symptôme peut être déclenché et faire passer l'action en erreur dans son **CompteRendu**.

L'échéancier détermine les tâches à exécuter sur réception d'un message MSG_TRAITEMENT en provenance du WOM, à l'aide de la méthode **AjouterTraitement**.

5.2.3.2.5.Diagramme des Actions pour les Traitements Algorithmiques

La classe **Action** est la classe de base de tous les traitements algorithmiques de la chaîne de traitement IASI OPS. En effet dans notre cas, une instance de la classe **Tache traitement** est associée à une instance de la classe **Action** suite à l'analyse préalable par la classe **Echéancier.** Ainsi les Taches/Actions identifiées dans le cadre de ce projet sont liées aux possibilités de parallélisation de la chaîne de traitement. Deux cas sont prévus selon que l'on fonctionne en mode intermédiaire (1A-1C, 1B-1C) ou en mode normal (0-1C). Les tableaux ci-dessous présentent les taches identifiées dans chaque cas.

Nom tache	Parallelisation	Contenu	Synchro
SD_GES_ChaineImageDeb	non	Algorithmes 30_FTB, 38_ICC, 110_DPT	
SD_GES_ChaineImagISRFEM	par ligne	algorithmes46_HIP, 39_IRC, 20_DOC, 22,SOS, 23_SSD, 21_SSS, 24_IAX	attente fin 1
SD_GES_FiltrageAxeInterf	non	algorithme 25_FAX	attente fin 2
SD_GES_ChaineProd0-1C	par ligne	algorithmes 09_PLK, 33_SME, 43_ISF, 20_DOC, 31_SCR, 32_HEC, 34_SMC, 22_SOS, 35_S1B, 37_S1C, 40_IAC, 41_CCS, 42_PCH, 44_GEO, 100_EXS, 111_MCX, 45_QIS	attente fin 3

Tableau 30 : Plan de Travail du Traitement 0-1C

Nom tache	Parallelisation	Contenu	Synchro
SD_GES_FiltrageAxeInterf	Non	algorithme 25_FAX	
SD_GES_ChaineProd1A-1C	par ligne	algorithmes 43_ISF, 46_HIP, 22_SOS, 35_S1B, 37_S1C, 40_IAC, 41_CCS, 42_PCH, 44_GEO, 100_EXS, 111_MCX, 45_QIS	attente fin 1

Tableau 31 : Plan de Travail du Traitement 1A-1C

Nom tache	Parallelisation	Contenu	Synchro
SD_GES_FiltrageAxeInterf	Non	algorithme 25_FAX	
SD_GES_ChaineProd1B-1C	par ligne	algorithmes 43_ISF, 46_HIP, 37_S1C, 40_IAC, 41_CCS, 42_PCH, 44_GEO, 100_EXS, 111_MCX, 45_QIS	attente fin 3

Tableau 32 : Plan de Travail du Traitement 1B-1C

Pour plus de précisions sur le contenu de chacune des taches on se reportera au § 5.2.3.3.4.

Remarque importante : On note ici que les taches de lecture/écriture des données ne sont pas inclues dans les taches de traitement. Ceci est la conséquence de 2 constats :

il est possible, pour des tailles de granules raisonnables de stocker globalement l'ensemble des données E/S à un granule afin de simplifier et augmenter la performance des phases de lecture/écriture fichier.

dans le cas ou une erreur majeure sur les données fondamentales survient, on doit être capable de s'arrêter sans générer de produits. Notre solution concentrant ce type d'erreur à l'extérieur des taches de traitement, permet facilement de répondre à ce besoin.

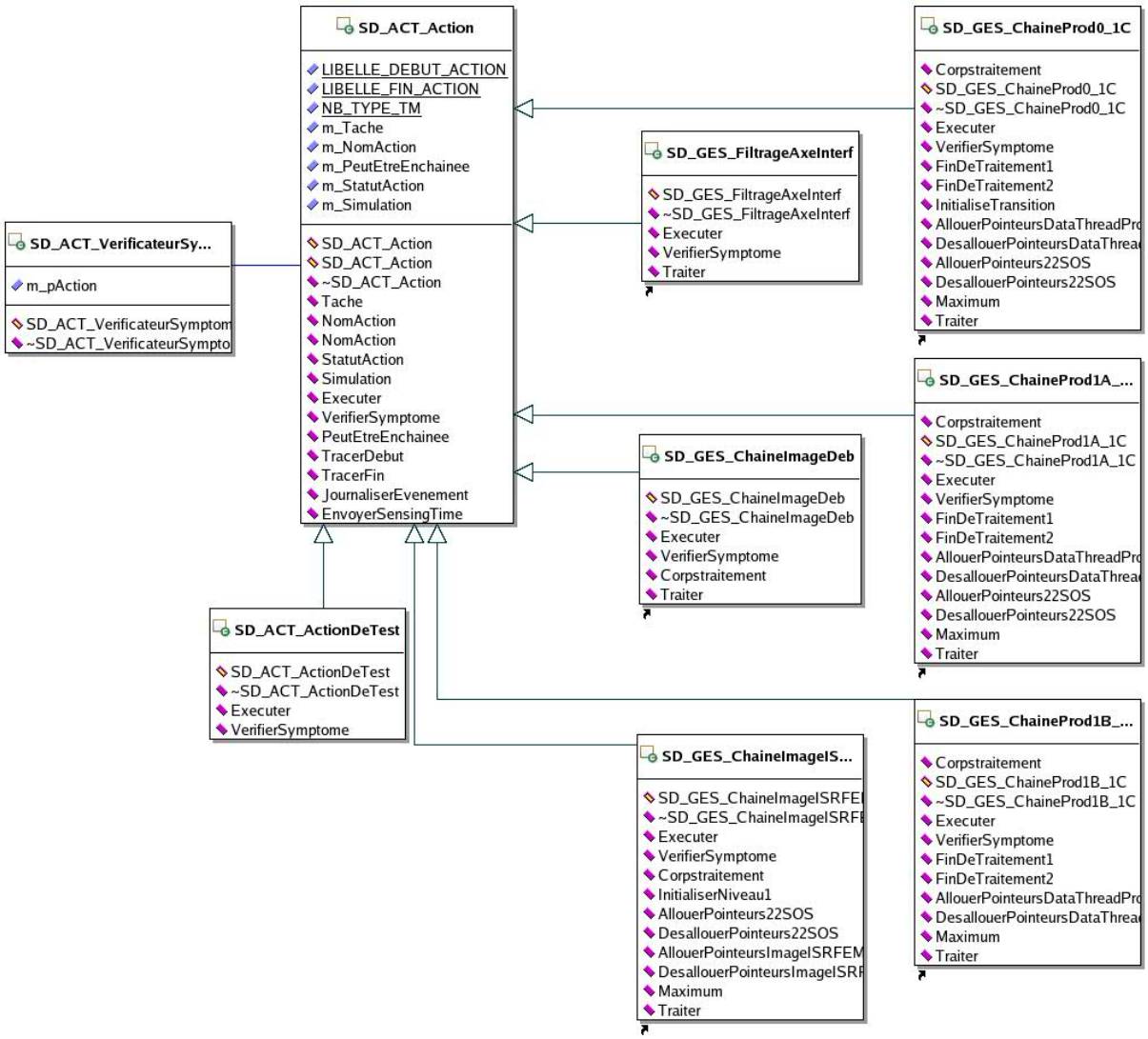


Figure 33 : Actions pour les traitements algorithmiques

Les actions (**SD_GES_XXX**) implémentent les enchaînements d'algorithmes tels que décrits dans le diagramme d'enchaînement. Ces classes utilisent les modules C encapsulant les algorithmes (**SD_ALG_XX**).

Dans la pratique l'activation du plan de travail se traduit par l'exécution séquentielle des méthodes Exécuter de chacun objet Actions instancié. L'enchaînement des appels est fonction de la chaîne choisie définie dans le work-order : 0-1C, 1A-1C, 1B-1C. Cet enchaînement est défini dans le fichier de configuration asoocié.

Le diagramme de classe met en évidence une architecture logicielle reposant sur :

des classes de haut niveau enchaînant les appels aux fonctions algorithmiques (**SD_GES_XX**) qui héritent de la classe générique **SD_ACT_Action**,

un gestionnaire de données (**SD_DON_GestDonnees**) qui offre un service d'accès centralisé aux données nécessaires aux chaînes algorithmiques :

produits (niveau 0, Niveau 1, AVHRR),

données de configuration (Stables, Autres, Banque spectrale, données auxiliaires),

données de contexte.

des modules C implémentant chacun des algorithmes spécifiés dans le document [DA7].

5.2.3.2.6.Diagramme de Classes pour le Gestionnaire de Données

Le Gestionnaire de données du SD permet d'accéder à l'ensemble des données utilisées par les traitements. Celles ci sont séparées en 2 types :

les données dynamiques qui sont issues de l'analyse du Work-order et qui sont rechargées pour chaque granule. On retrouve ici les produits (niveau 0, niveau1, AVHRR) mais aussi le Work-Order. Chacune est représentée dans le diagramme qui suit par une ou plusieurs classes suivant la complexité de sa gestion.

les données statiques qui sont connues dès le lancement de la chaîne. Elles sont ainsi chargées en mémoire ou référencées. On retrouve ici les fichiers de configuration (stable, Autres, Banque spectrale, Auxiliaire) mais aussi le Contexte. Chacune est représentée dans le diagramme qui suit par une ou plusieurs classes suivant la complexité de sa gestion.

Les Données Dynamiques :

La figure suivante présence l'ensemble des classes en charge de la gestion des données dynamiques.

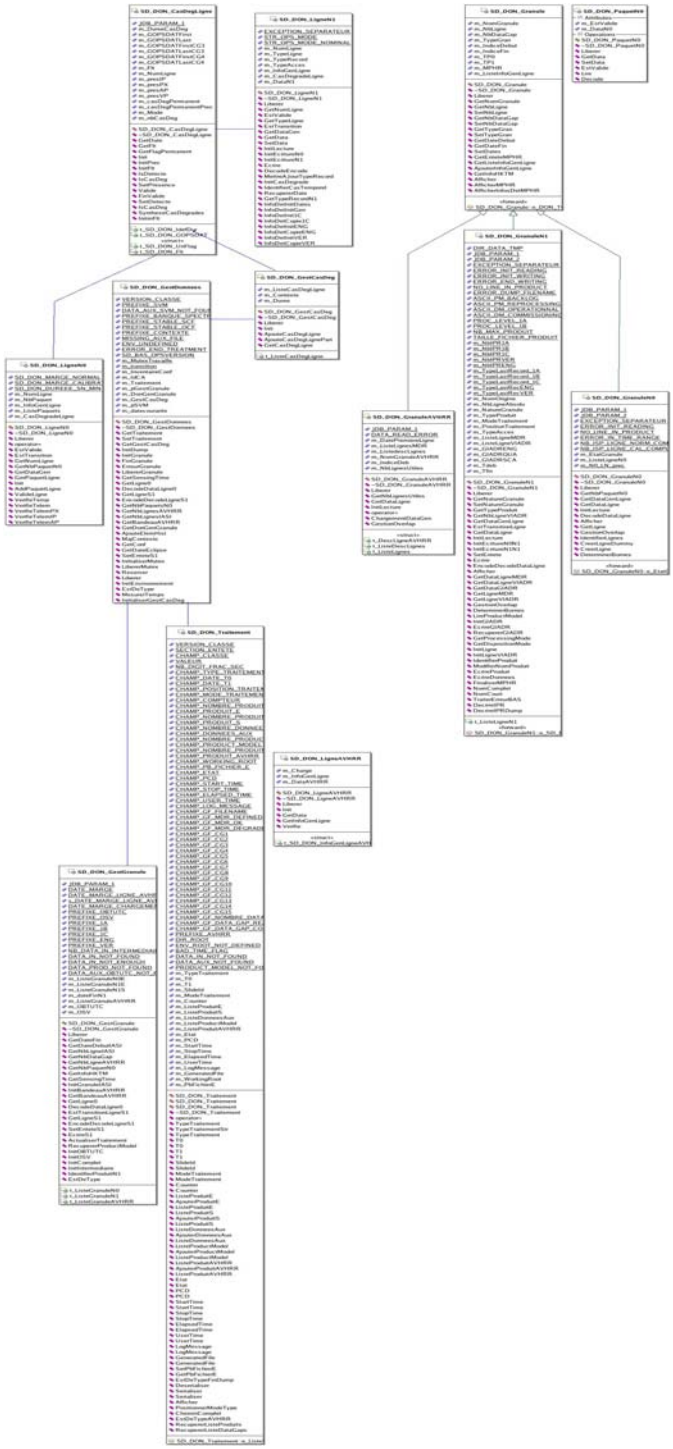


Figure 34 : Gestionnaire de Données (données dynamiques)

SD_DON_GestGranule permet d'accéder aux produits IASI par l'intermédiaires des granules (ensemble de lignes d'un même produit) : **SD_DON_GranuleN1** et **SD_DON_GranuleN0** . Ceux ci reposent eux même sur les entités de base de stockage que sont : **SD_DON_LigneN1, SD_DON_LigneN0** et **SD_DON_PaquetN0**.

Les accès AVHRR sont optimisés au travers de l'objet **SD_DON_BandeauAVHRR** centralisant les informations issues de plusieurs produits (**SD_DON_GranuleAVHRR**) et stockées dans des lignes **SD_DON_LigneAVHRR.**

A ces données évidentes s'ajoute l'objet **SD_DON_Traitement** qui regroupe l'ensemble services d'accès aux informations permettant d'initialiser le traitement. Cet objet est instancié par le Work Order Manager (WOM) qui le construit et l'envoie au serveur de données après avoir complété certaines informations dans le cas du mode dump (cf § 5.2.3.3.3). Finalement notons ici, la présence de 3 classes dédiées à la gestion des cas dégradés liées aux données : **SD_DON_CasDegLigne**, **SD_DON_CasDegradé**, **SD_DON_GestCasDeg.**

La classe **SD_DON_FinTraitement** offre l'ensemble des services pour la constitution du report (utilisé par le WOM).

Les Données Statiques :

La figure suivante présence l'ensemble des classes en charge de la gestion des données statiques.

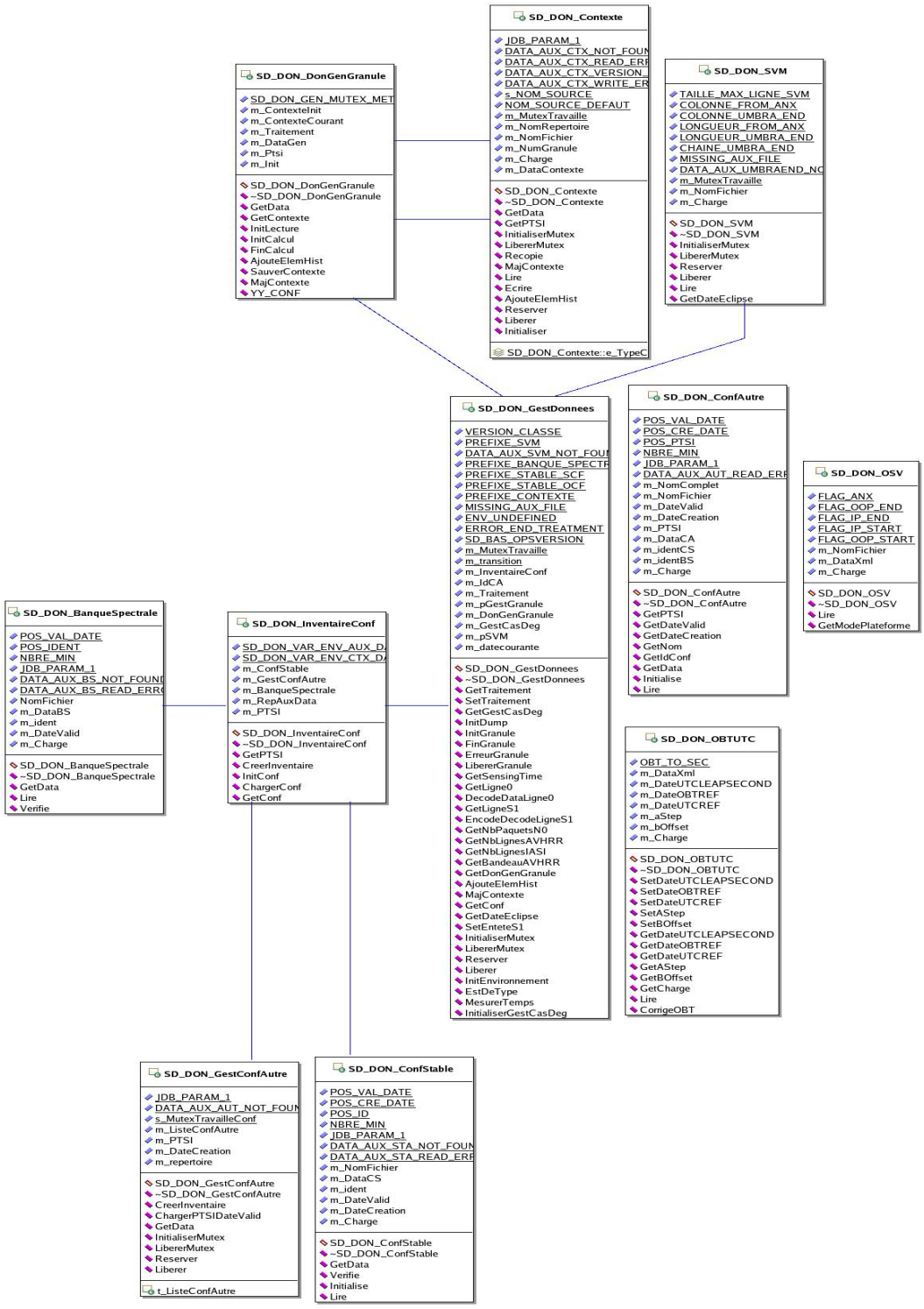


Figure 35 : Gestionnaire de Données (données statiques)

SD_DON_InventaireConf offre les services d'accès aux données de configuration et permet de gérer leur cohérence globale. Chaque type de données de configuration est pris en charge par une ou 2 classes spécialisées :

SD_DON_GestConfAutre et SD_DON_ConfAutre,

SD_DON_GestConfStable et SD_DON_ConfStable,

SD_DON_BanqueSpectrale,

SD_DON_GeomEvent, SD_DON_OrbitPredict, SD_DON_OBTUTC.

La Gestion du contexte (**SD_DON_Contexte**) est quant à elle particulière du fait du besoin de sauvegarde fichier de la donnée pour chaque granule. Le chargement depuis un fichier se fait comme pour les autres données statiques lors du lancement. Elle est associée à l'initialisation d'une instance de l'objet **SD_DON_DonGenGranule** qui regroupe les informations globales non inclues dans le contexte.

L'accès au gestionnaire de données depuis les traitements (**SD_GES_xxxxxxxxxx**) se fait par l'intermédiaire de l'attribut m_Tache présent dans chaque action. Cette tache référence directement la classe principale du serveur de données (**SD_FRW_ApplicationServeurDonnées**) dans laquelle est stocké un lien vers l'objet **SD_DON_GestDonnées.**

5.2.3.3.Choix d'Implémentation

5.2.3.3.1.Interfaçage C/C++

Afin de préserver au maximum les performances de la chaîne de traitement il a été décidé de réaliser l'ensemble des fonctions algorithmiques en langage C. Cependant l'intégration des fonctions algorithmique dans l'architecture multithreadée du Serveur SSALTO réalisé en C++ impose des contraintes d'interface. La frontière entre les module C et les classes C++ a été définie de la manière suivante :

au niveau conceptuel, il semble beaucoup plus naturel de réaliser les couches de gestion des données en C++ afin de bénéficier de la puissance d'un langage Objet et de mettre naturellement en pratique les concepts essentiels que sont : l'encapsulation, l'héritage, le polymorphisme. Ainsi dans le cadre de la chaîne IASI OPS, les classes directement liées au serveur de données sont réalisées en C++.

afin de permettre l'utilisation des données gérées dans des objets C++ par des modules C, ces informations sont stockées dans des structures de données compatibles avec en langage C. Ainsi, lors de la création d'un objet, on alloue et on remplit cette structure, l'objet C++ ne fait que stocker l'adresse de cette structure. Pour utiliser la donnée (lecture/écriture) dans les algorithmes C, il suffit alors de transmettre cette adresse transmettre à la routine C de calcul. Nous stockons de cette manière les paquets de niveau 0, les lignes de niveau 1, les lignes AVHRR ainsi que les données de configuration.

Le schéma ci-dessous identifie le langage utilisé pour chacun des paquetages logicielles constituant les traitements algorithmiques.

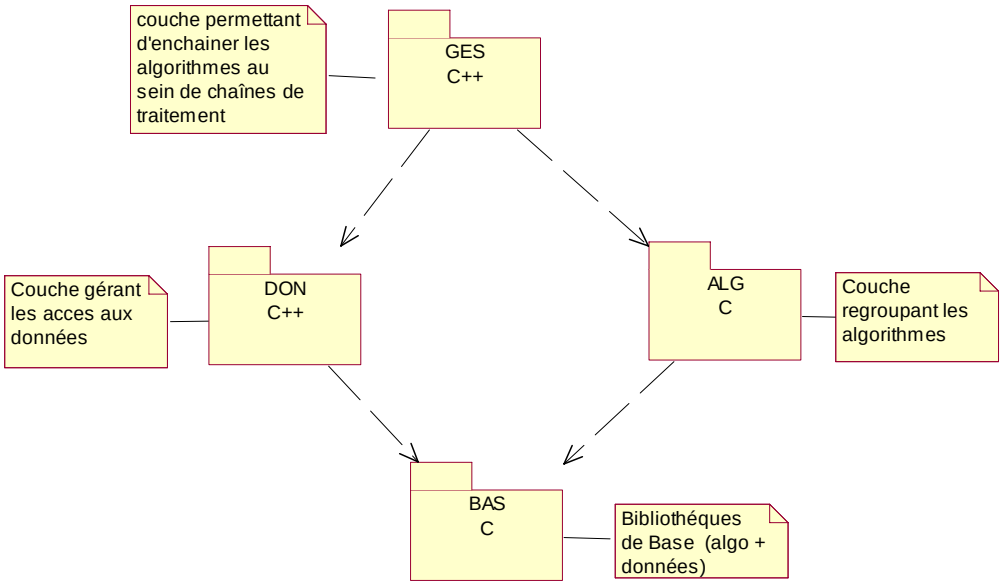


Figure 36 : Langage Informatiques des Paquetages de Traitements

NB : notons en outre que la couche des bibliothèques de base est réalisée en C afin de simplifier les appels depuis les algorithmes mais aussi car elle est constituée en grande partie de fonctions réutilisées (accès aux données (produits, configuration)).

5.2.3.3.2.Les Modes Intermédiaires

La chaîne de traitement doit pouvoir s'adapter au niveau des données en entrée pour pouvoir générer en sortie des produits de niveau 1C. Les cas pris en compte dans notre conception sont :

0-1C comprenant un produit de niveau 0 en entrée et 3 produits en sortie (niveau 1 + données technologiques + données de vérification),

1A-1C comprenant 2 produits de niveau 1 en entrée et 2 produits en sortie (niveau 1 + données technologiques),

1B-1C comprenant 2 produit de niveau 1 en entrée et 2 produits en sortie (niveau 1 + données technologiques).

Il est à noter que quelque soit le niveau des données d'entrée, le niveau des données de sortie attendues est toujours le même : niveau 1C.

Au niveau de l'initialisation, cela se traduit par l'utilisation de fonctions d'initialisation spécifiques en fonction du niveau des données d'entrée.

Cette phase d'initialisation se ramène à récupérer un certain nombre d'informations du produit d'entrée et à l'utiliser dans le produit en sortie :

dans le cas du traitement de niveau 0-1C, ceci est très succinct car la majorité des paramètres sont calculés en cours de traitement,

dans le cas des traitements 1A-1C ou 1B-1C, on doit à l'inverse récupérer un maximum d'informations qui sont recopiées depuis les données d'entrée dans les données de sortie.

Par contre, les traitements à effectuer sont distincts et diffèrent suivant les niveaux. Ils sont matérialisés dans notre solution par des taches/actions spécifiques à chaque niveau décrites au §5.2.3.3.4. Cependant les accès aux données est identique étant donné que la phase l'initialisation aboutit à un état interne semblable.

Finalement, l'écriture des produits fait appel à une service générique générant les mêmes produits (excepté pour les données de vérification).

5.2.3.3.3.Cas du Mode Dump

Le mode dump correspond à la possibilité de traiter un fichier orbite complet sans qu'un découpage préalable en granule ait été effectué par le PGF. Le mode est déterminé par le type de traitement défini dans le fichier Work-order. Afin de ne pas complexifier les interfaces d'entrée et le fonctionnement du SD, nous avons choisi de découper artificiellement le fichier orbite en entrée en granules « virtuels » de taille paramétrable. Ce travail est assuré par le Working Order Manager. Ce dernier génère ainsi autant d'ordres de traitement que de granules à traiter et demande au Serveur de Données de réaliser le traitement. Cette solution permet d'utiliser les interfaces WOM-SD existantes. Le message de traitement envoyé par le WOM indique le nom du fichier de donnée d'entrée et les dates de la 1^{ière} et de la dernière ligne à traiter.

La prise en compte du mode dump au niveau du processus SD se résume alors à :

parcourir les données d'entrée et les charger en mémoire en se restreignant à une fourchette de dates issue de l'ordre de traitement,

créer le fichier de données en sortie lors du traitement du premier granule puis l'enrichir lors des traitements suivants. Les entêtes sont alors également mises à jour au niveau des informations générales (nombre de lignes, indicateurs qualités, ...),

charger en mémoire pour chaque granule « virtuel » une partie des données AVHRR en entrée en se basant sur une fourchette de temps.

5.2.3.3.4.Descriptions des Tâches Algorithmiques

Ce chapitre détaille l'algorithme implémenté par chaque tâche et montre l'intégration des fonctions algorithmiques présentées au &5.2.3.2.5 avec les fonctions d'accès aux données en entrée / sortie. On considère de plus que l'action d'initialisation a été préalablement exécutée afin de charger en mémoire les données d'entrée et d'initialiser si besoin les données en sortie. L'écriture des produits est exécuté en dehors des actions de traitement afin de faciliter la gestion des cas dégradés.

Chaque appel à une fonction est précédée pour plus de clarté du mot « call ». Les regroupements réalisés ici sont ceux qui seront implémentés sur la chaîne opérationnelle.

La prise en compte des cas dégradés issu du document [DA104] est ici précisé en utilisant le formalisme suivant : « **Si (** CG5 **) Alors » qui doit se comprendre par « Si le cas dégradé CG5 n'est pas levé Alors ».**

5.2.3.3.4.1. SD_GES_ChaineImageDeb

/*----------*/
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte */
/*----------*/
/* filtrage des températures du corps noir chaud */
Mise à jour contexte avec les temperatures BBT de chaque ligne
Si (CG5 **) Alors** poids pour la température = 0

Call ***30_FTB***

Pour chaque ligne du granule
/*----------*/
/* récupérer les données de niveau 0 */
/* récupérer les données de niveau 1 initialisées */
/* récupérer les données de configuration en fonction du PTSI */
/* récupérer les données temporaire par ligne initialisées */
/*----------*/
/* Initialisation des données /ligne */
Si (CG2 ou CG12 **) Alors**

Si (CG7 ou CG8 **) Alors**
/*décodage du train binaire image dans des tableaux temporaires*/
/* calculer les coefficients de calibration des images (A1323) */
Call 38_ICC

/* calculer la table des pixels morts (A1322) */
Call 110_DPT
/* libération tableaux temporaires image */

Mise à jour contexte
Si numero absolu de ligne courante modulo 10 = 0
Mise à jour VIADR

Fin Si
Fin Si
Fin Pour chaque ligne du granule

5.2.3.3.4.2. SD_GES_ChaineImagISRFEM (par ligne)

```
/*-----------------------------------------------------------------*/
/* récupérer les données de configuration en fonction du PTSI */
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte */
/* récupérer les données de niveau 0 */
/* récupérer les données de niveau 1 initialisées */
/* récupérer les données temporaire par ligne initialisées */
/*-----------------------------------------------------------------*/
   Si ( CG2 ou CG12 ) Alors
      Pour chaque sous-cycle de la ligne ( de 1 à 30  ou de 3 à 30)
        Si ( CG6 ) Alors
            /*décodage du train binaire  image dans un tableau temporaire*/
      /* decodage image + calibration */
            Call 39_IRC
            Codage de l'image dans le produit 1A
            /* libération tableaux temporaires image  */

        Fin Si
        Si ( CG3) Alors
            Pour chaque pixel ( de 1 à 4 )
            /* reservation tableaux temporaires spectre  */
            /*decodage du train binaire  spectre dans un tableau temporaire*/
       Call 20_DOC
      Si Overflow ou underflow on lève CG4 et even,tuellement CG1 et CG3
            Si ( CG1 ou CG6 ou CG4 pour la fenêtre utilisée) Alors
                  /*Détermination du décalage spectral (A1332)*/
            Call 22_SOS
            Call 23_SSD
             Fin SI
            /* libération tableaux temporaires spectre  */
            Fin Pour chaque pixel
        Fin SI
      Fin Pour chaque sous-cycle de la ligne
      Pour chaque direction de coin de cube ( de 1 ,2 )
            Pour chaque pixel ( de 1 à 4 )
                  /*Sélection des décalages spectraux (A1333)*/
            Call 21_SSS
            Fin Pour chaque pixel
            /*Détermination de l'axe interférométrique (A1334)*/
      Call 24_IAX
      Fin Pour chaque direction de coin de cube
   Fin SI
```

5.2.3.3.4.3. SD_GES_FiltrageAxeInterf

/*----------*/
/* récupérer les données de configuration */
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte */
/* récupérer les données de niveau 1 initialisées */
/* récupérer les données temporaire par ligne initialisées */
/*----------*/

/*Filtrage de l'axe interférométrique (A1335)*/
Call 25_FAX

5.2.3.3.4.4. SD_GES_ChaineProd0-1C (Par ligne)

/*----------*/
/* récupérer les données de configuration en fonction du PTSI */
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte (historique axe interpol)*/
/* récupérer les données de niveau 0 (spectres décodés)*/
/* récupérer les données de niveau 1 initialisées */
/* récupérer les données AVHRR en fonction des dates de la ligne avec une marge /* une liste de pointeur est ici retournée*/
/* récupérer les données temporaire par ligne initialisées */
/*----------*/

Si (CG2 ou CG12 **) Alors**

/*Calcul fonction de Planck et température corps noir (A1342)*/
Call 09_PLK
Call 33_SME
Pour chaque direction de coin de cube (de 1 ,2)
Pour chaque pixel (de 1 à 4)
/*Interpolation de la banque spectrale (A1343)*/
Call 43_ISF
Fin Pour chaque pixel
Fin Pour chaque direction de coin de cube
Call *XX_INI*

/* Homogénéité inter-pixel du NZPD */
Call 46_HIP
/* reservation tableaux temporaires spectre (0, 1A, 1B ,1C) */**Pour** chaque sous-cycle de la ligne (de 1 à 30)
Pour chaque pixel (de 1 à 4)
/*decodage du train binaire spectre dans un tableau temporaire*/
/*decodage du spectre*/
Call 20_DOC
Si Overflow ou underflow on lève CG4 et eventuellement CG1 et CG3

Si (CG1 **) Alors**
/*surveillance télémesure (A1361)*/
Si (visee terre **) Alors**

Call 100_EXS
Fin Si
Mise à jour contexte
Si numero absolu de ligne courante modulo 10 = 0

Mise à jour VIADR

Pour chaque bande (de 1 à 3)
Call 111_MCXFin Pour
/*Calibration radiométriques et corrections (A1344)*/
Call 31_SCR
Call 32_HEC
Call 34_SMC
Codage du spectre dans le produit 1A
/*Sur-echantillonnage ,re-echantillonnage du spectre (A1345)*/
Call 22_SOS
Call 35_S1B
Codage du spectre dans le produit 1B
/*Apodisation du spectres du spectre A1346) */
Call 37_S1C
Codage du spectre dans le produit 1C
Fin Si
Fin Pour chaque pixel
// recuperation des indices lignes disponible entre TSN – marge et
// TSN + marge , le tableau AVHRR est passé à la fonction *40_IAC*
Si (CG9 **) Alors**
Si (CG6 **) Alors**
/*corregistration Imageur / AVHRR (A1351)*/
Call 40_IAC
sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/*analyse des radiances dans les FOV sondeur (A1352)*/
Call 41_CCS
sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/* Construction des pseudo-canaux IASI-AVHRR ***/**
Pour chaque pixel (de 1 à 4)
Call 42_PCH
Fin chaque pixel
/*géolocalisation (A1353)*/
Call 44_GEO
Fin Pour chaque sous-cycle de la ligne
/* libération tableaux temporaires spectre (0, 1A, 1B ,1C) */

/*calcul flag et indices de qualité (A1362)*/
Call 45_QIS
Constitution du PCD
Fin Si

5.2.3.3.4.5. SD_GES_ChaineProd1A-1C (par ligne)

/*----------*/
/* récupérer les données de configuration en fonction du PTSI */
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte (historique axe interpol)*/
/* récupérer les données de niveau 1 initialisées (spectres 1A décodés)*/
/* récupérer les données AVHRR en fonction des dates */
/* récupérer les données temporaire par ligne initialisées */
/*----------*/
Si (CG2 ou CG12 **) Alors**

Pour chaque direction de coin de cube (de 1 ,2)
Pour chaque pixel (de 1 à 4)
/*Interpolation de la banque spectrale (A1343)*/
Call 43_ISF
Fin Pour chaque pixel
Fin Pour chaque direction de coin de cube
/* Initialisation des données /ligne */
Call *XX_INI*

/* Homogénéité inter-pixel du NZPD */
Call 46_HIP

/* reservation tableaux temporaires spectre (1A, 1B ,1C) */
Pour chaque sous-cycle de la ligne (de 1 à 30)
Pour chaque pixel (de 1 à 4)

Si (CG1 **) Alors**
Decodage du spectre dans le produit 1A
/*Sur-echantillonnage ,re-echantillonnage du spectre (A1345)*/
Call 22_SOS
Call 35_S1B
Codage du spectre dans le produit 1B
/*Apodisation du spectres du spectre A1346) */
Call 37_S1C
Codage du spectre dans le produit 1C
Fin Si
Fin Pour chaque pixel

Si (CG9 **) Alors**
Si (CG6 **) Alors**
/*corregistration Imageur / AVHRR (A1351)*/
Call 40_IAC

sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/*analyse des radiances dans les FOV sondeur (A1352)*/
Call 41_CCS
sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/* Construction des pseudo-canaux IASI-AVHRR */
Pour chaque pixel (de 1 à 4)
Call 42_PCH
Fin chaque pixel
/*géolocalisation (A1353)*/
Call 44_GEO
Fin Pour chaque sous-cycle de la ligne
/* libération tableaux temporaires spectre (1A, 1B ,1C) */

/*calcul flag et indices de qualité (A1362)*/
Call 45_QIS
Constitution du PCD

Fin SI

5.2.3.3.4.6. SD_GES_ChaineProd1B-1C

/*----------*/
/* récupérer les données de configuration en fonction du PTSI */
/* récupérer les données temporaire globales initialisées */
/* récupérer les données du contexte (historique axe interpol)*/
/* récupérer les données de niveau 1 initialisées (spectres 1B décodés)*/
/* récupérer les données AVHRR en fonction des dates */
/* récupérer les données temporaire par ligne initialisées */
/*----------*/
Si (CG2 ou CG12 **) Alors**
/* Initialisation des données /ligne */

Pour chaque direction de coin de cube (de 1 ,2)
Pour chaque pixel (de 1 à 4)
/*Interpolation de la banque spectrale (A1343)*/
Call 43_ISF
Fin Pour chaque pixel
Fin Pour chaque direction de coin de cube
Call *XX_INI*

/* Homogénéité inter-pixel du NZPD */
Call 46_HIP

/* reservation tableaux temporaires spectre (1B ,1C) */
Pour chaque sous-cycle de la ligne (de 1 à 30)
Pour chaque pixel (de 1 à 4)

Si (CG1 **) Alors**
Decodage du spectre dans le produit 1B
/*Apodisation du spectres du spectre A1346) */
Call 37_S1C
Codage du spectre dans le produit 1C
Fin Si
Fin Pour chaque pixel
Si (CG9 **) Alors**
Si (CG6 **) Alors**
/*corregistration Imageur / AVHRR (A1351)*/
Call 40_IAC
sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/*analyse des radiances dans les FOV sondeur (A1352)*/

Call 41_CCS
sinon
on prend le dernier offset estimé (ligne précédente si possible)
Fin SI
/* Construction des pseudo-canaux IASI-AVHRR ***/**
Pour chaque pixel (de 1 à 4)
Call 42_PCH
Fin chaque pixel
/*géolocalisation (A1353)*/
Call 44_GEO
Fin Pour chaque sous-cycle de la ligne
/* Libération tableaux temporaires spectre (1B ,1C) */

/*calcul flag et indices de qualité (A1362)*/
Call 45_QIS
Constitution du PCD

Fin SI

5.2.3.3.5.Chargement et Stockages des Données Dynamiques

Un choix majeur pris dans notre architecture est de stocker en mémoire dès le début du granule, les données nécessaires au traitement, c.a.d :

la banque spectrale,

les fichiers de configuration,

le contexte,

les données de niveau 0 décodées,

les données de niveau 1 décodées,

les données de calcul par ligne,

les données AVHRR.

Dans la configuration du CGS, nous savons que chaque nœud de traitement possède 2 Go de mémoire vive. Il est donc nécessaire de valider que la taille des données en mémoire est compatible avec ces valeurs.

Le tableau ci-dessous illustre notre démarche et permet de valider le choix effectué.

Notons de plus que les données internes sont prises en compte au travers d'une marge de 30 % sur les données dynamiques.

Remarque : il faut noter que la stratégie actuelle de traitement consiste à décoder et à recoder à la volées les données images et spectre ce qui réduit de manière significative les évaluations (on suppose une parallélisation dans un facteur 4).

hypothéses	
taille granule	22
nombre de sous-cycles dans un granule	30
nombre de colonnes image IASI	64
nombre de Lignes image IASI	64
nombre de pixels sondeurs	4
nombre d'echantillons du spectre N0 N1	8500
nombre de Bandes spectrales	3
nombre de paquets spectres dans une ligne	120
nombre de paquets image dans une ligne	34
nombre de paquets de vérification dans une ligne	5
nombre de lignes AVHRR / lignes IASI	48,00
nombre de canaux AVHRR	5,00
nb colonnes AVHRR	2048,00
marge pour les données internes	30,00%
nombre de lignes traitées en parallèle	4

Figure 37 : Hypothèses de Dimensionnement

	taille unitaire	codage	nombre	taille globale
Banque spectrale	330 522 062	1	1	330 522 062
Fichier de configuration stable	3 343 786	1	1	3 343 786
Fichier de configuration autre	120 270	1	1	120 270
contexte	0	1	1	0
				0
				0
données de niveau 0				0
paquets spectres				0
entete	160	2	2640	844 800
spectre codé	8 960	1	2640	23 654 400
spectre décodé	8 500	8	480	32 640 000
paquets images				0
entete	25	2	748	37 400
images codées	6 202	1	748	4 639 096
images décodées	1 920	8	480	7 372 800
paquets de verification				0
MDR	27 000	2	110	5 940 000
paquets auxiliaires				0
MDR	390	2	22	17 160
				0
données internes				22 543 697
				0
données de niveau 1				0
Ligne N1C				0
données toutes les 10 lignes	625 178	1	2,2	1 375 392
entete	25 753	1	22	566 566
spectre 1A decodé	1 020 000	8	4	32 640 000
spectre 1A codé	1 020 000	2	22	44 880 000
spectre 1B décodé	1 020 000	8	4	32 640 000
spectre 1B codé	1 020 000	2	22	44 880 000
spectre 1C décodé	1 020 000	8	4	32 640 000
spectre 1C codé	1 020 000	2	22	44 880 000
Ligne N1 technologiques	6 795	8	22	1 195 920
Ligne N1 vérification	27 000	2	110	5 940 000
données internes (images calibrées ...)				72 491 363
				0
données AVHRR de niveau 2				0
Lignes	10 240	2	1056	21 626 880
données internes				6 488 064

volume total=	773 919 656

Figure 38 : Taille RAM estimée (cas 0-1C)

Dans le cas du mode dump, ce raisonnement est encore valide dans la mesure ou tous se passe comme si le dump était découpé virtuellement en granules.

Le même calcul a été réalisé dans le cas des modes intermédiaires et montre une occupation mémoire inférieure.

5.2.3.4.Diagrammes de séquence

5.2.3.4.1.Diagrammes de Séquence lors du Démarrage de la Chaîne OPS

Initialisation du Serveur de Données

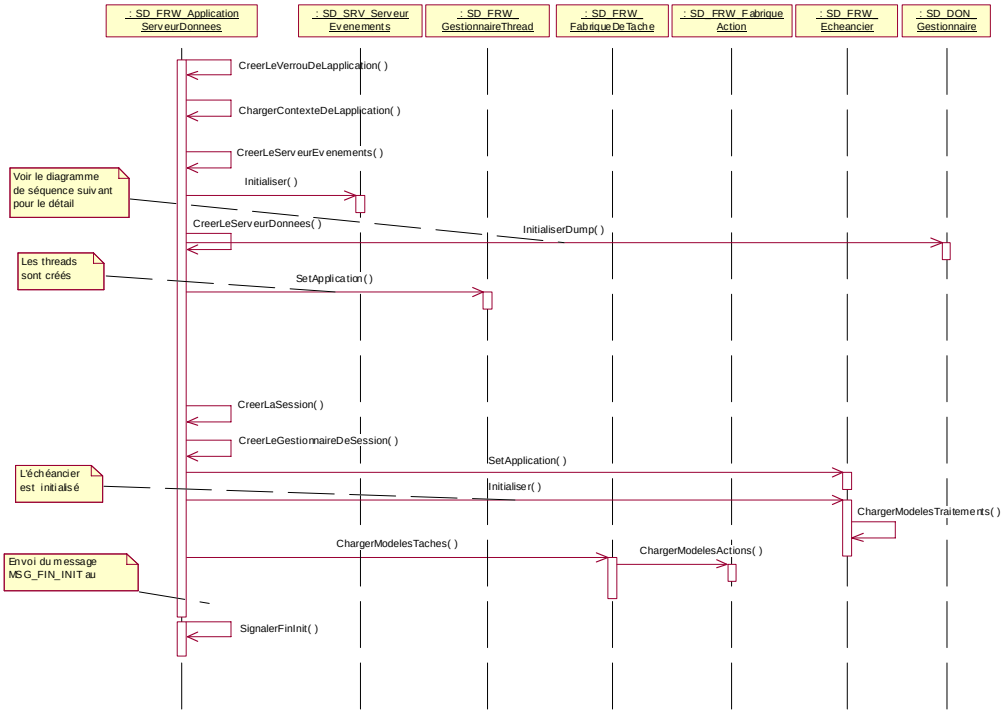


Figure 39 : Diagramme de séquence Initialisation du Serveur de Données

Au lancement du Serveur de Données, le serveur d'événement est créé ainsi que les gestionnaires de données. La méthode **SD_FRW_GestionnaireThread.SetApplication** permet de créer le pool de thread. Grâce à **CreerLeGestionnaireDeSession**, le premier thread est réservé pour le gestionnaire de la session créée préalablement (**CreerLaSession**).

Ensuite, les fichiers de configuration (traitement, tache et action) sont lues et chargés en mémoire. Enfin, un message de fin d'initialisation est envoyé au WOM (**SignalerFinInit**).

Initialisation du Gestionnaire de Données

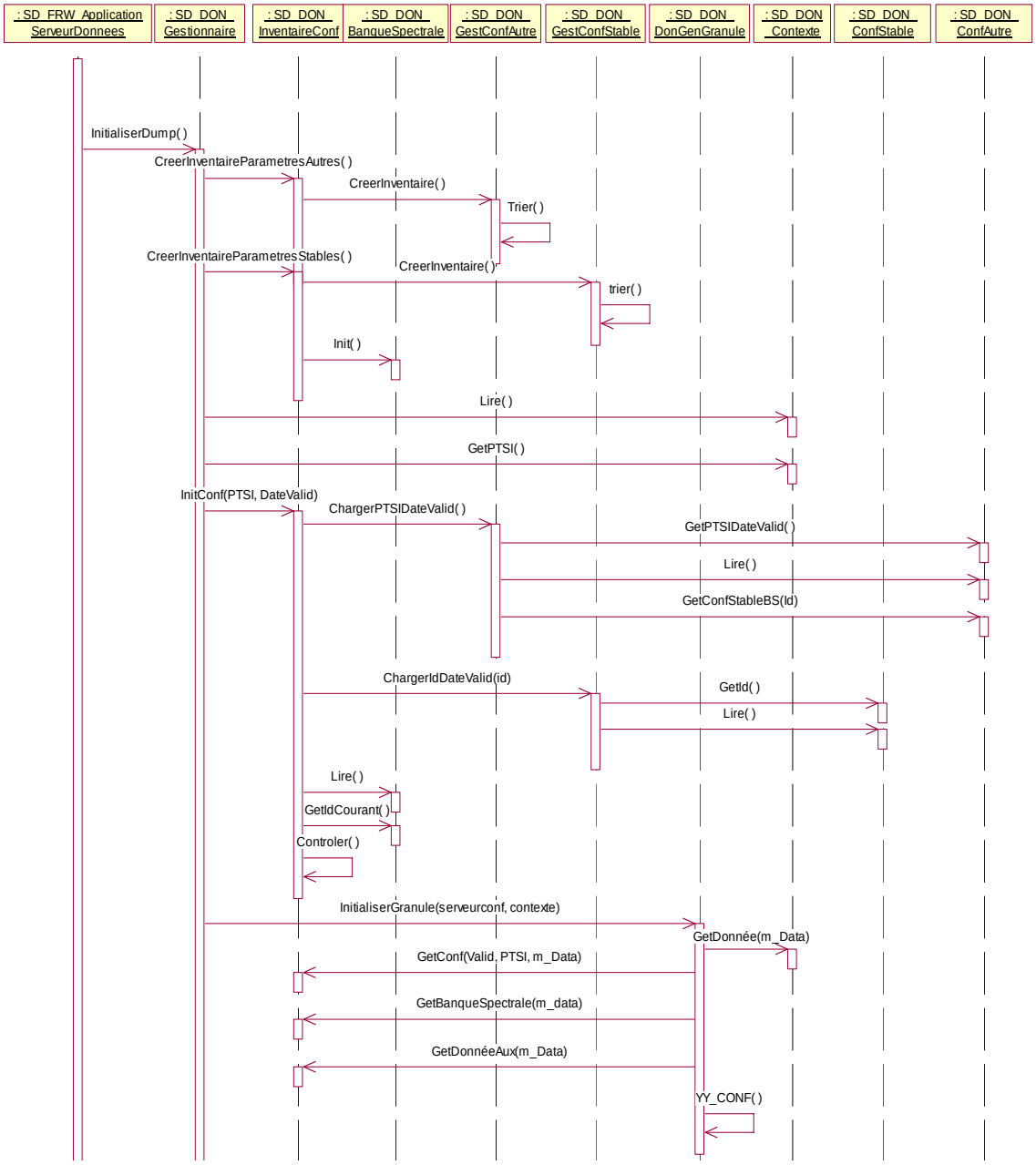


Figure 40 : Diagramme de Séquence de l'Initialisation des Données Statiques

Après création par la classe **SD_FRW_ApplicationServeurDonnees** d'une instance de la classe **SD_DON_GestionnaireDonnees** la méthode **initialiserDump** est appelée.

Pour les paramètres de configuration, un objet **SD_DON_InventaireConf** est créé afin de regrouper les accès aux données de configuration, puis les objets de configuration nécessaires au traitement sont mises en place, c.a.d :

création de l'inventaire des paramètres de configuration au travers de la classe **SD_DON_InventaireConf** et du gestionnaire **SD_DON_GestConfAutre** (méthode **CreerInventaireParametresAutres**),

création de l'inventaire des paramètres de configuration au travers de la classe **SD_DON_InventaireConf** et du gestionnaire **SD_DON_GestConfStable** (méthode **CreerInventaireParametresStables**),

lecture du fichier de contexte disponible via la méthode **Lire** de l'objet **SD_DON_Contexte**.

A la fin de cette phase, on peut alors se baser sur le contenu du contexte (PTSI) pour charger la configuration. La méthode **InitConf** de l'objet inventaire est ainsi appelée et elle regroupe la lecture en cascade des fichiers de configuration autres (méthode **ChargerPTSIDateValid**), stables (méthode **ChargerIdDateValid**), ainsi que de la banque spectrale (méthode **Lire**). Finalement un contrôle interne de cohérence est effectué.

Cette phase se termine avec la création et l'initialisation d'un l'objet **SD_DON_DonGenGranule** qui permet grâce à l'accès aux données précédentes de réaliser les calculs globaux sur le granule via la méthode interne **YY_CONF**.

5.2.3.4.2.Diagrammes de Séquence du Traitement d'un Granule

Initialisation de l'échéancier

Cette initialisation se déroule lors de la réception d'un message MSG_TRAITEMENT en provenance du WOM. Elle consiste en découper le traitement en tâches et initialiser le gestionnaire de données.

Le découpage en tâche est effectué par la méthode **SD_FRW_Echeancier.AjouterTraitement**. A chaque tâche est associé un contexte de traitement.

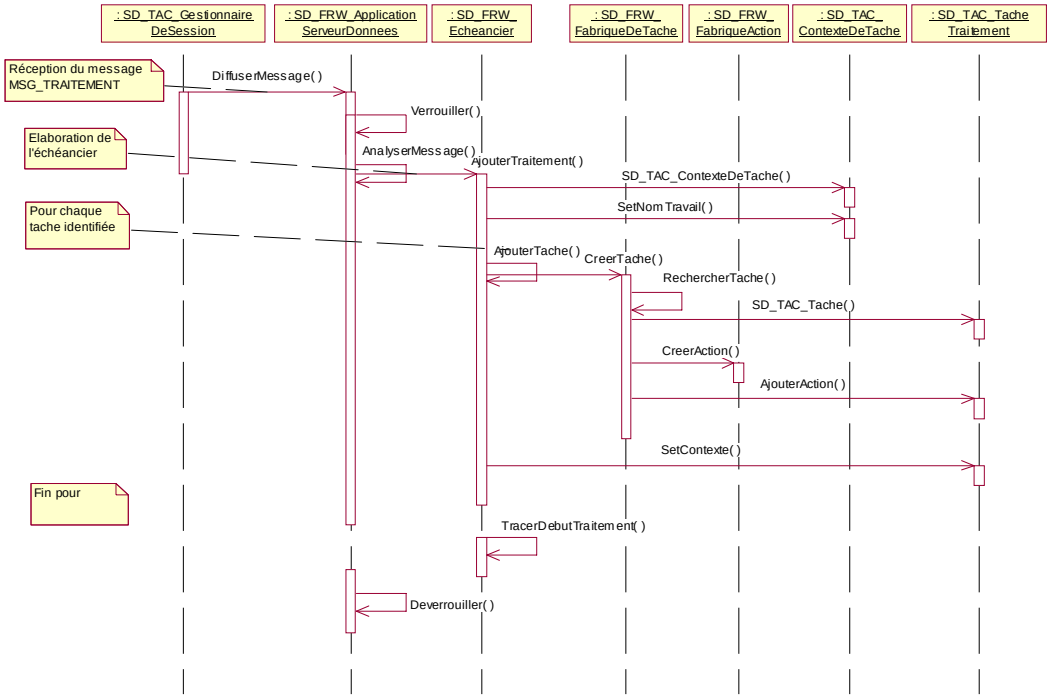


Figure 41 : Diagramme de Séquence Initialisation de l'échéancier

Initialisation du gestionnaire de données lors du traitement d'un granule

La phase d'initialisation exécutée lors du lancement du traitement d'un granule est décrite ci-dessous.

Celle-ci se décompose en 2 étapes majeures :

l'initialisation des données dynamiques : initialisation et chargement des données utilisées en entrée/sortie de la chaîne : produits IASI et AVHRR, données auxiliaires,

la mise à jour des données statiques, en fonction des données dynamiques précédentes, qui permet d'actualiser les paramètres de configuration.

La première étape consiste, après récupération des informations issues du work-order (objet **SD_DON_Traitement**) à initialiser le granule par la méthode **InitGranule** de l'objet **SD_DON_GestGranule**. L'initialisation de l'objet **SD_DON_GranuleN0** consiste alors à charger l'entête du fichier granule. La gestion de l'overlap peut ainsi être mise en œuvre (méthode **GestionOverlap**) afin de déterminer la première et la dernière ligne du produit à traiter (on récupère pour cela les informations générales dans les paquets instruments : type paquet, mode instrument, date ligne, numéro de sous-cycle). Ce parcours est réalisé en chargeant en mémoire les entêtes des paquets instruments (méthode ***GetInFoGen***).

Après ceci, le chargement complet du produit de niveau 0 est réalisé par le granule de niveau 0 **SD_DON_GranuleN0**. Les enregistrements de type MDR sont alors lus par la fonction **Lire** de l'objet SD_DON_PaquetN0. Les paquets sont alors regroupés en lignes (**SD_DON_LigneN1**) puis analysés (chronologie et qualité) par des méthodes internes. Pour chaque Ligne, un gestionnaire de cas dégradé est crée (**SD_DON_CasDegLigne**) par la méthode **AjouteCasDegLigne** de l'objet **SD_DON_GestCasDeg** et suivant l'analyse précédente un cas dégradé peut être levé.

A partir de ce granule, 3 granules de niveau 1 (objets **SD_DON_GranuleN1**) sont créés par la méthode **InitEcritureN0N1** de l'objet **SD_DON_GestGranule**. Ceux ci s'initialisent à partir de l'entête de niveau 0 ainsi qu'a partir du modèle donné en entrée. Pour chaque ligne de niveau 0 créée en entrée une ligne en sortie **SD_DON_LigneN1** est créée et initialisée par la méthode **Init**.

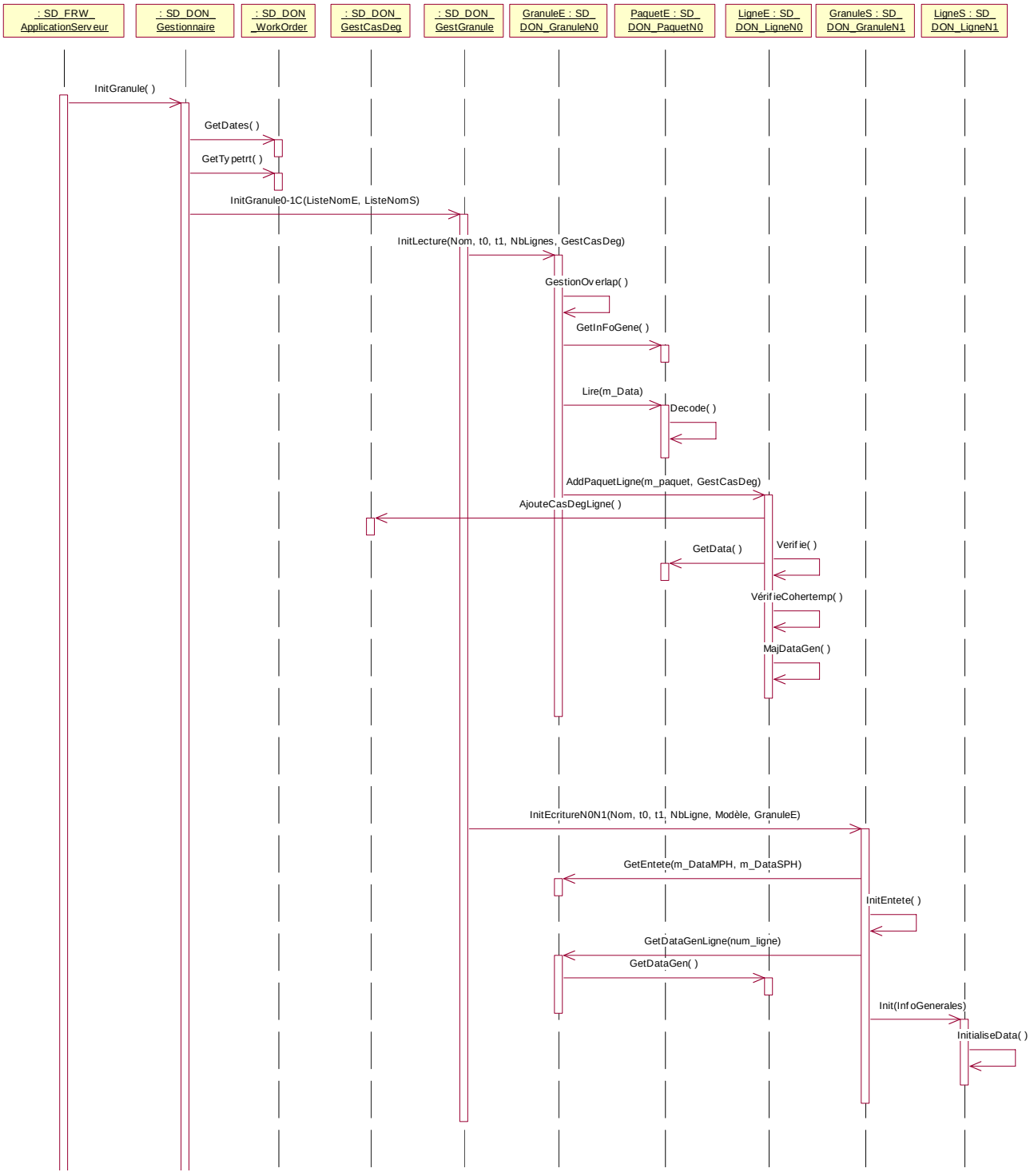


Figure 42 : Diagramme de Séquence de l'Initialisation des Données Dynamiques du Granule

Finalement, le bandeau AVHRR est initialisé (objet **SD_DON_BandeauAVHRR**) par analyse de l'ensemble des fichiers AVHRR en entrée (objet **SD_DON_GranuleAVHRR**). Ceux – ci sont lus (**InitLecture**) s'ils correspondent à une fourchette de date issue du produit en entrée. L'ensemble des lignes comprises dans cette fourchette est alors monté en mémoire par la méthode **LireLigne** de l'objet **SD_DON_LigneAVHRR** puis elles sont ajoutées au Bandeau courant (**AjouteLigne**) et décodées.

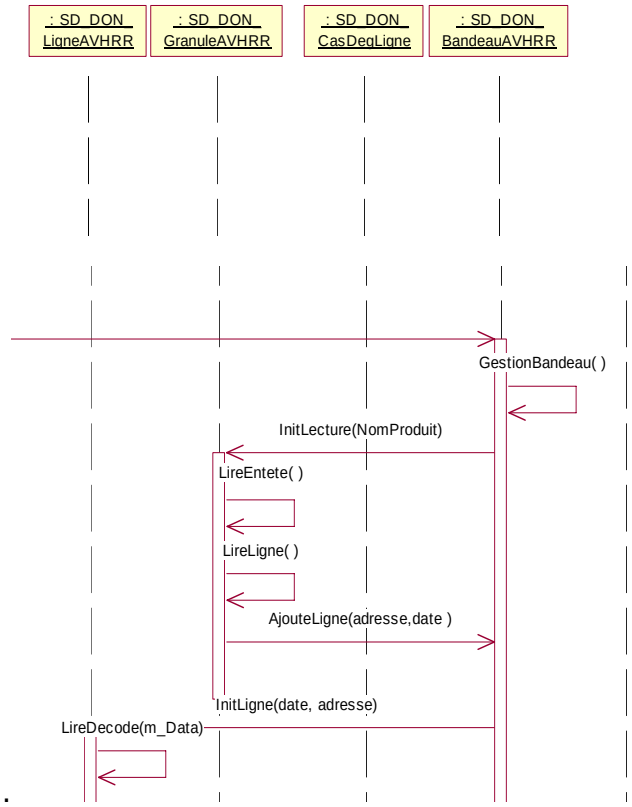


Figure 43 : Diagramme de Séquence de l'Initialisation des Données AVHRR du Granule

Pour les données statiques, une mise à jour est effectuée suite à l'initialisation des données dynamiques.

On peut alors se baser sur le contenu du produit d'entrée (PTSI) pour charger la configuration. La méthode **ChargerParametresConf** de l'objet inventaire est ainsi appelée pour chaque ligne et elle regroupe la lecture du fichier de configuration autres (méthode **ChargerPTSIDateValid**), ainsi qu'un contrôle interne de cohérence. Les données auxiliaires sont alors rechargées par la méthode **LireDonAux** de l'objet **SD_DON_InventaireConf** et le contexte est finalement initialisé.

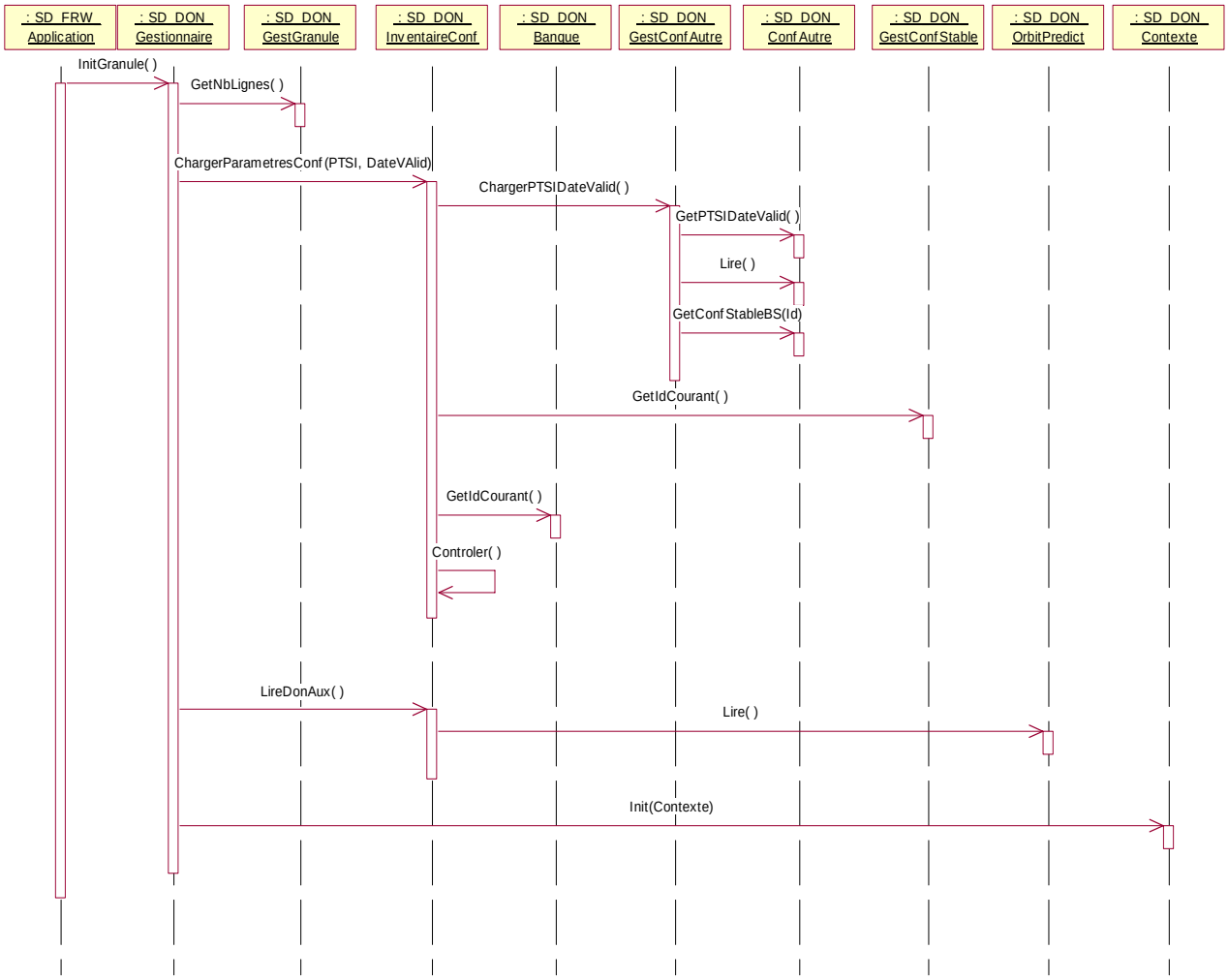


Figure 44 : Diagramme de Séquence Mise à jour des Données Statiques

Lancement des tâches

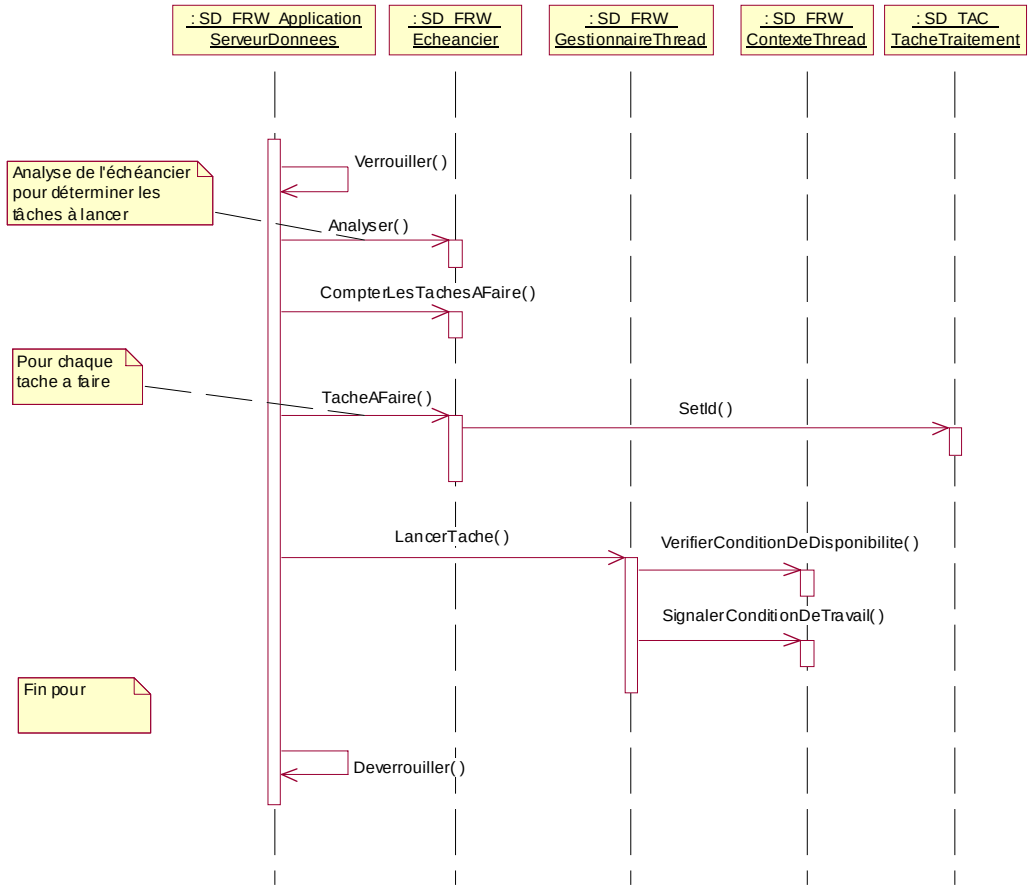


Figure 45 : Diagramme de Séquence de Lancement des tâches

Ce diagramme décrit les collaborations de classes mis en œuvre lors du lancement des taches à partir de l'analyse de l'échéancier.

L'application invoque la méthode **Analyser** de l'échéancier à intervalle régulier qui calcule les tâches à lancer . L'application recueille le nombre de tâche à lancer (déterminée par l'analyse de l'échéancier) via la méthode **CompterLesTachesAFaire,** puis chaque tâche via la méthode **TacheAFaire**. Un identifiant est associé à la tache par la méthode **SetId**. Enfin, l'application demande au gestionnaire de thread de trouver un thread disponible pour exécuter une tâche via la méthode **LancerTache**.

La méthode **VerifierConditionDeDisponibilite** permet d'identifier un thread libre pour affecter une tâche. L'invocation de la méthode **SignalerConditionDeTravail** permet de faire passer le thread dans l'état actif pour réaliser la tâche.

5.2.3.4.3.Diagramme de Séquence de Fin de Traitement d'un Granule

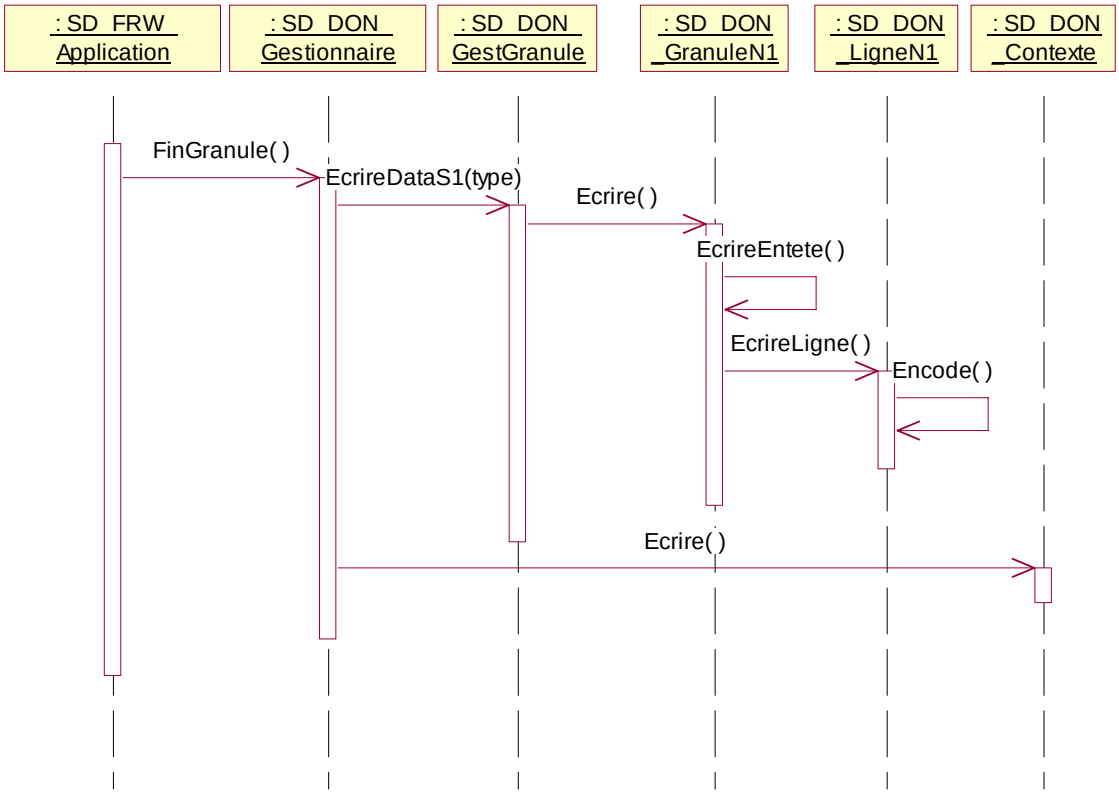


Figure 46 : Diagramme de Séquence de Fin de Traitement d'un Granule

Lors de la fin de traitement d'un granule, la méthode **FinGranule** est appelée sur le gestionnaire de granule afin d'écrire l'entête (**EcrireEntete**) et chaque ligne (**EcrireLigne**) pour chacun des granules de sortie (**SD_DON_GranuleN1**).

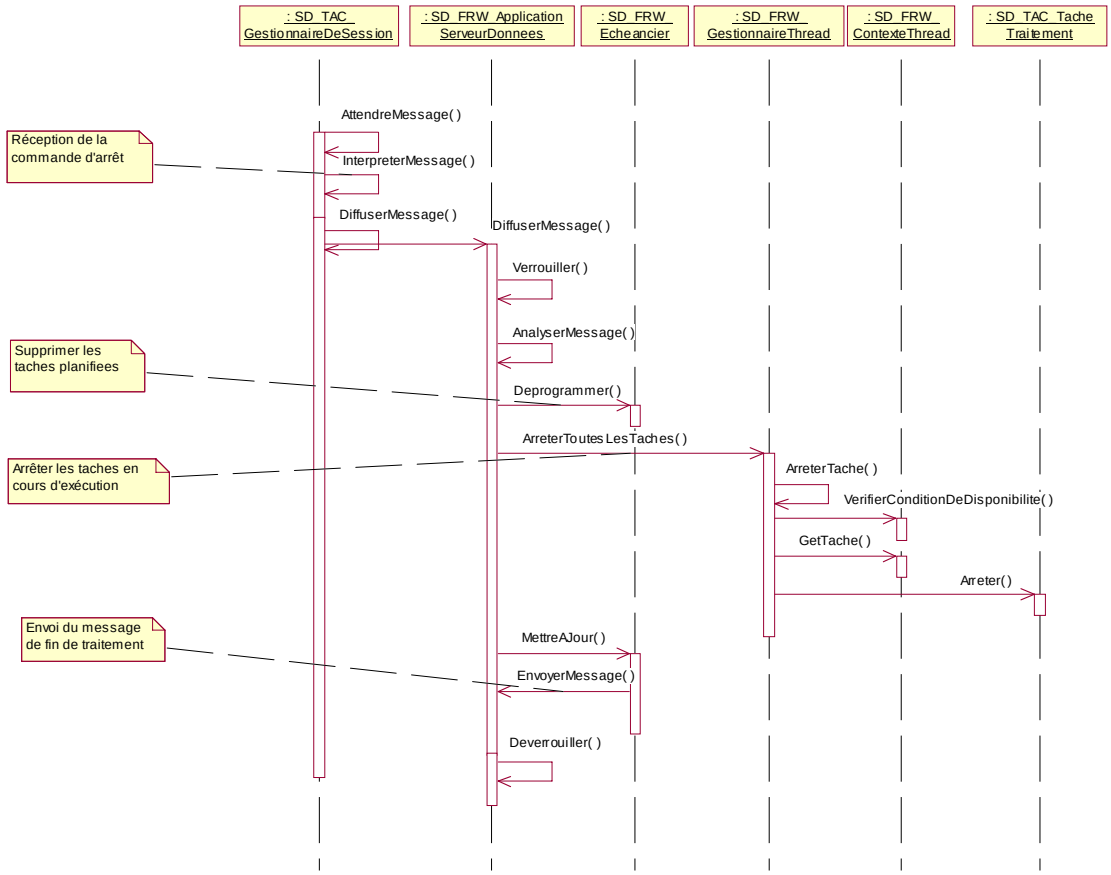
Finalement, le contexte courant (objet **SD_DON_Contexte**) est sauvegardé dans un fichier par la méthode **Ecrire.**

L'objet **SD_DON_FinTraitement** est alors renseigné via tous les accesseurs de manière à préparer les informations utiles au WOM pour la génération du ficher report puis la méthode **sérialiser** est utilisée au préalable de l'envoie du message sur le bus.

Remarque : c'est à ce stade que tous les objets associés à des données dynamiques sont détruits avant toute nouvelle exécution

Dans le cas d'une erreur majeure de traitement, c'est la méthode **ErreurGranule** qui est appelée ; celle ci effectue le même traitement mais ne génère pas le produit en sortie et ne sauvegarde pas le fichier contexte.

5.2.3.4.4. Diagramme de Séquence de Traitement d'une Commande d'Arrêt du Traitement en Cours



Ce diagramme décrit les collaborations de classes mis en œuvre lors de la réception d'une demande d'arrêt d'un traitement en cours.

Le gestionnaire de session reçoit la commande d'arrêt par le biais de la méthode **InterpreterMessage** et transmet avec sa routine **DiffuserMessage** cette commande à SD_FRW_ApplicationSDerveurDonnees.

L'application invoque la méthode **Déprogrammer** de l'échéancier qui supprime les tâches à lancer. L'application via la méthode **ArreterToutesLesTaches** demande au gestionnaire de threads d'arrêter les tâches en cours. Le gestionnaire teste si chacun de ses threads est actif (**VérifierConditionDeDisponibilié**) ; si c'est le cas, le gestionnaire informe la tâche en cours qu'elle doit s'arrêter (**Arrêter**).

Une fois les tâches arrêtées, l'échéancier est mis à jour, les traitements de fin granule exécutés et le message de fin de traitement est envoyé au WOM.

5.2.3.4.5.Diagramme de séquence de Production Nominale

Cf §5.2.3.3.4

5.2.3.4.6.Diagramme de séquence de Production en Mode Intermédiaire

Cf §5.2.3.3.4

5.2.3.5.Liste des Classes constituant le SD

Ce paragraphe a pour but lister de façon exhaustive toutes les classes constituant le Serveur de données (SD).

Paquetage ACT (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage ACT de gestion des actions génériques.

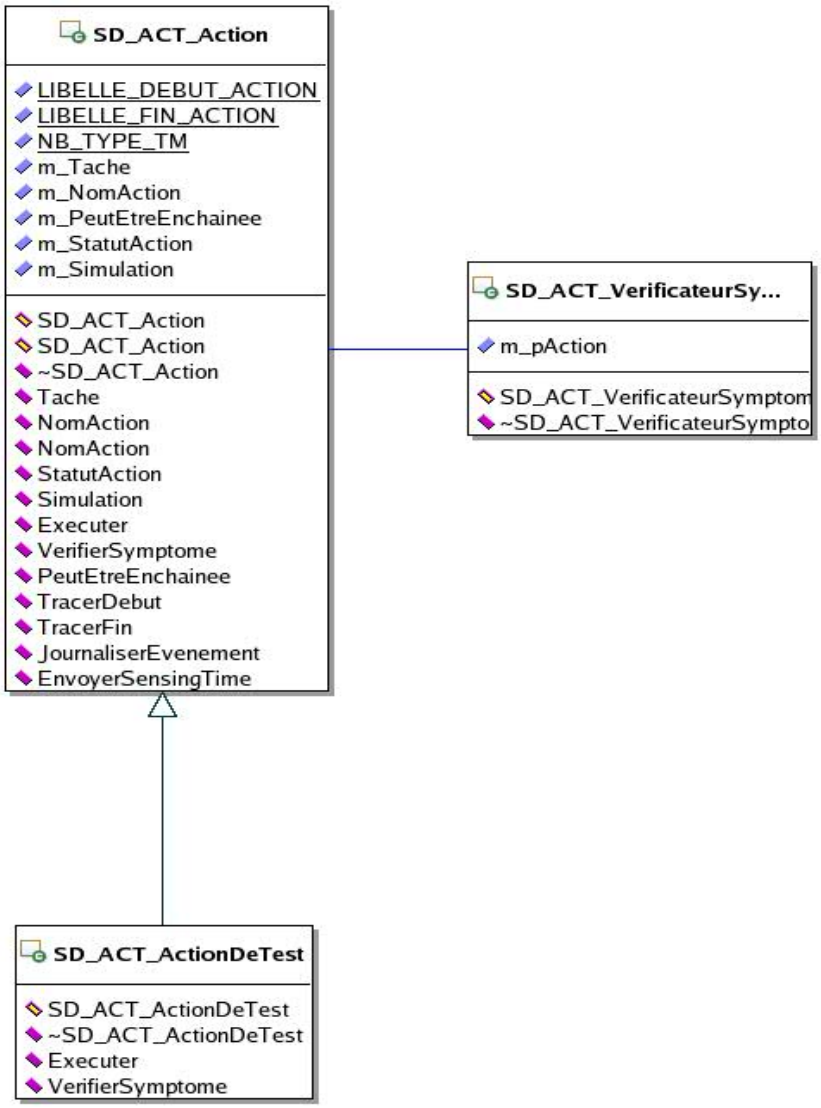


Figure 47 : Paquetage ACT

Paquetage EVT (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage EVT de gestion des événements.

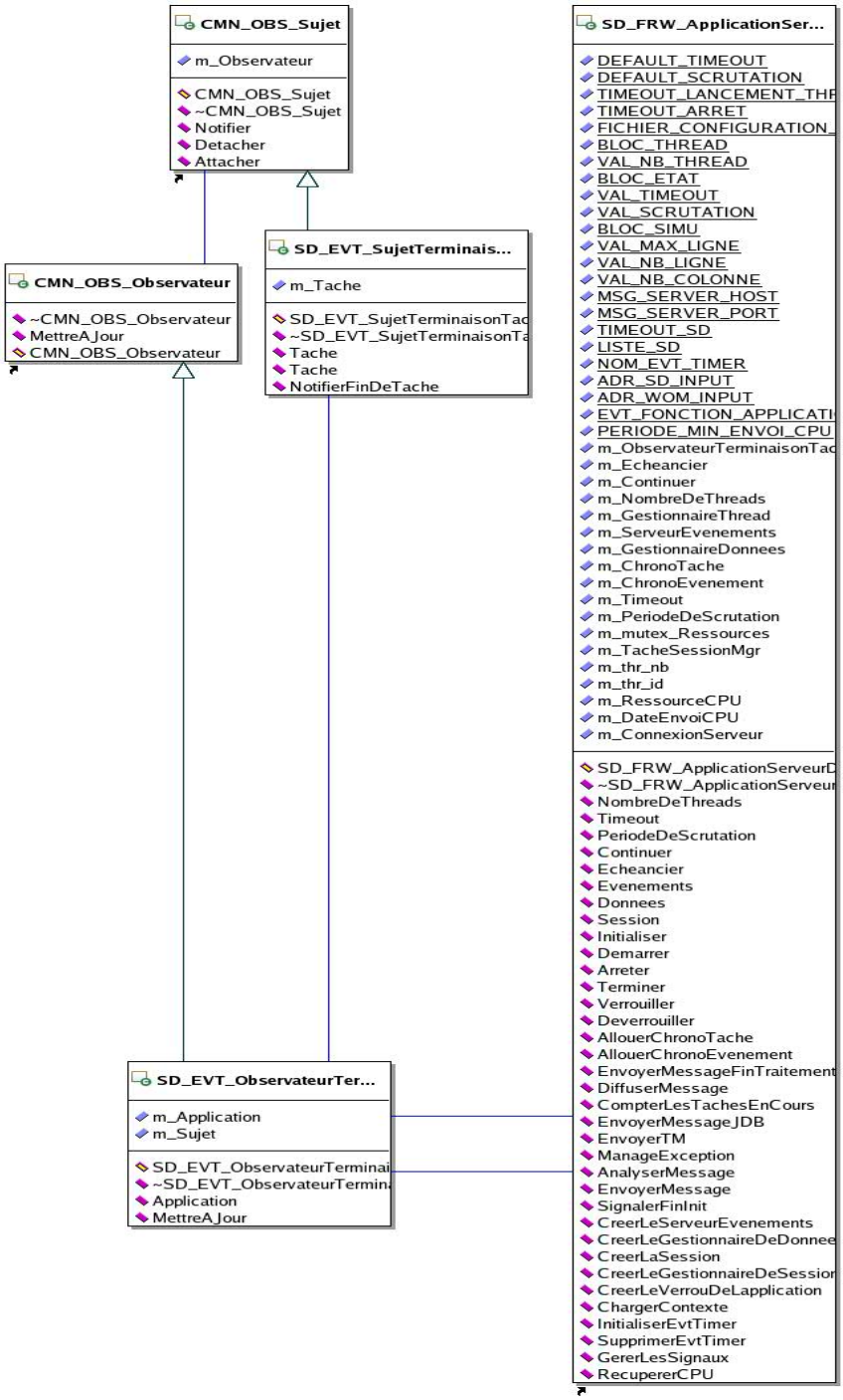


Figure 48 : Paquetage EVT

Paquetage FRW (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage framework FRW.

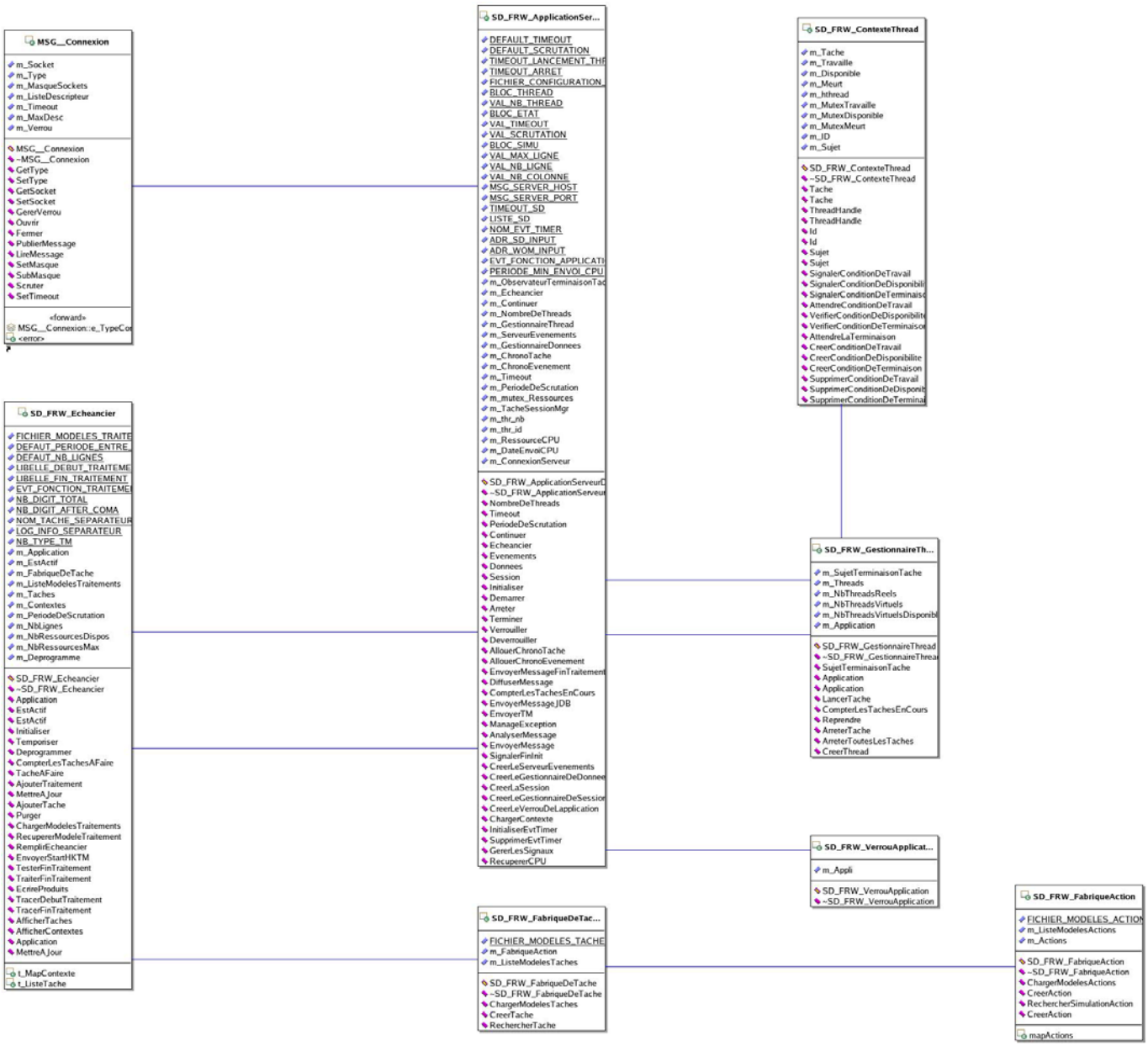


Figure 49 : Paquetage FRW

Paquetage SRV (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage SRV Serveur d'Evénements.

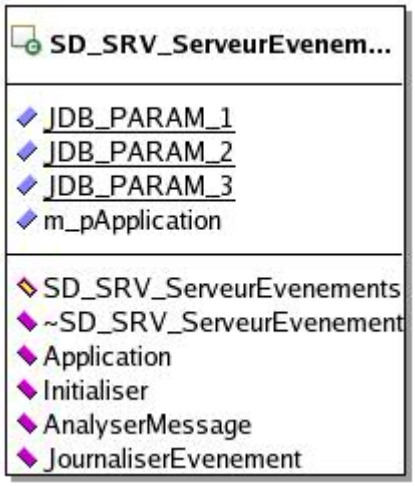


Figure 50 : Paquetage SRV

Paquetage TAC (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage TAC de gestion de tâches génériques.

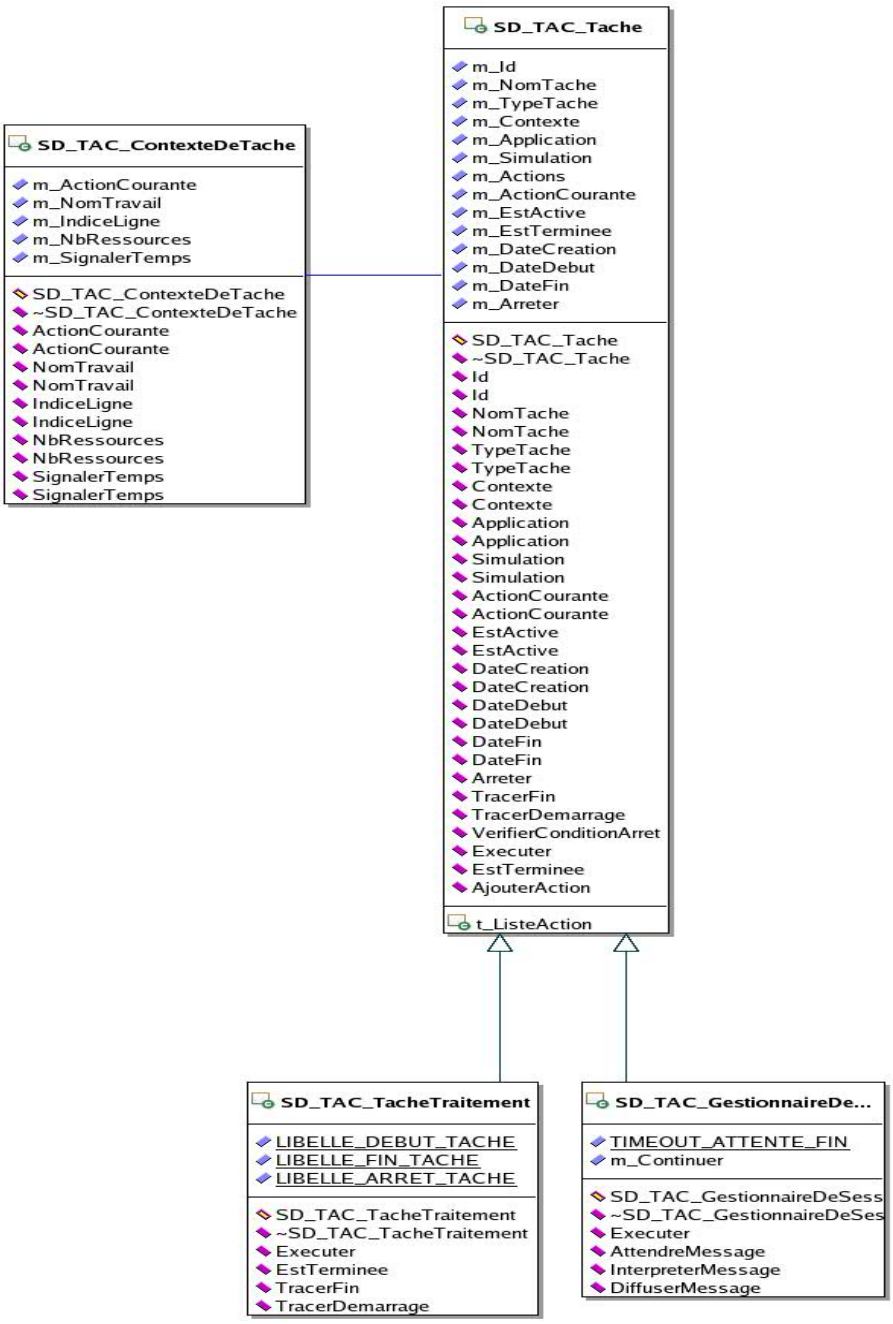


Figure 51 : Paquetage TAC

Paquetage UTL (récupéré de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage des utilitaires UTL.

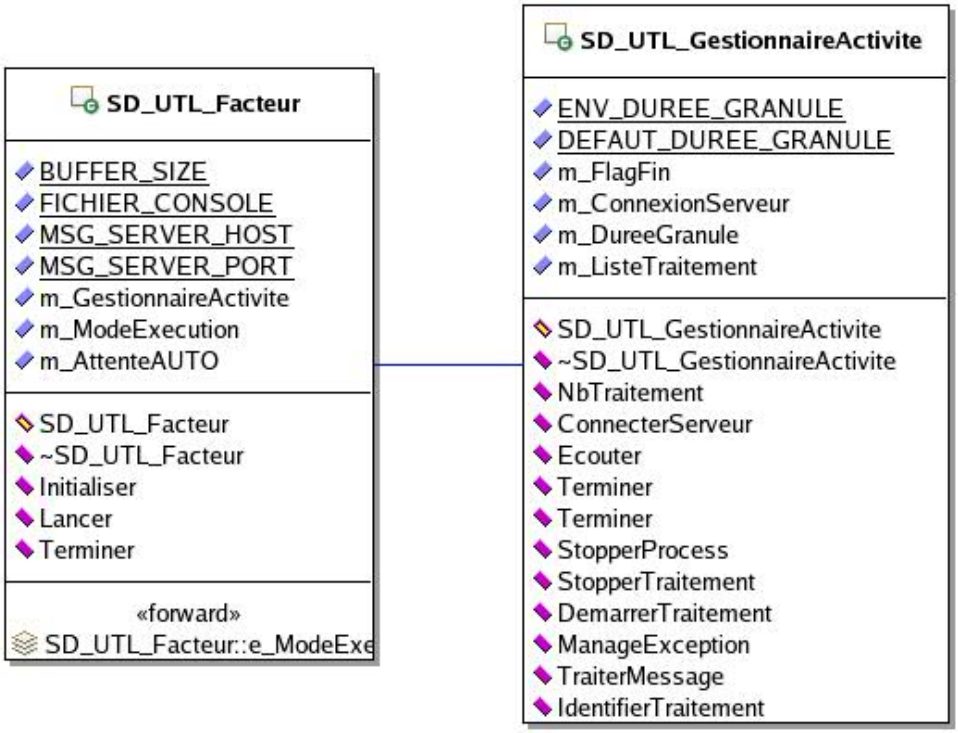


Figure 52 : Paquetage UTL

Paquetage DON (spécifique à l'OPS)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant le paquetage DON de gestion des données de l'OPS IASI.

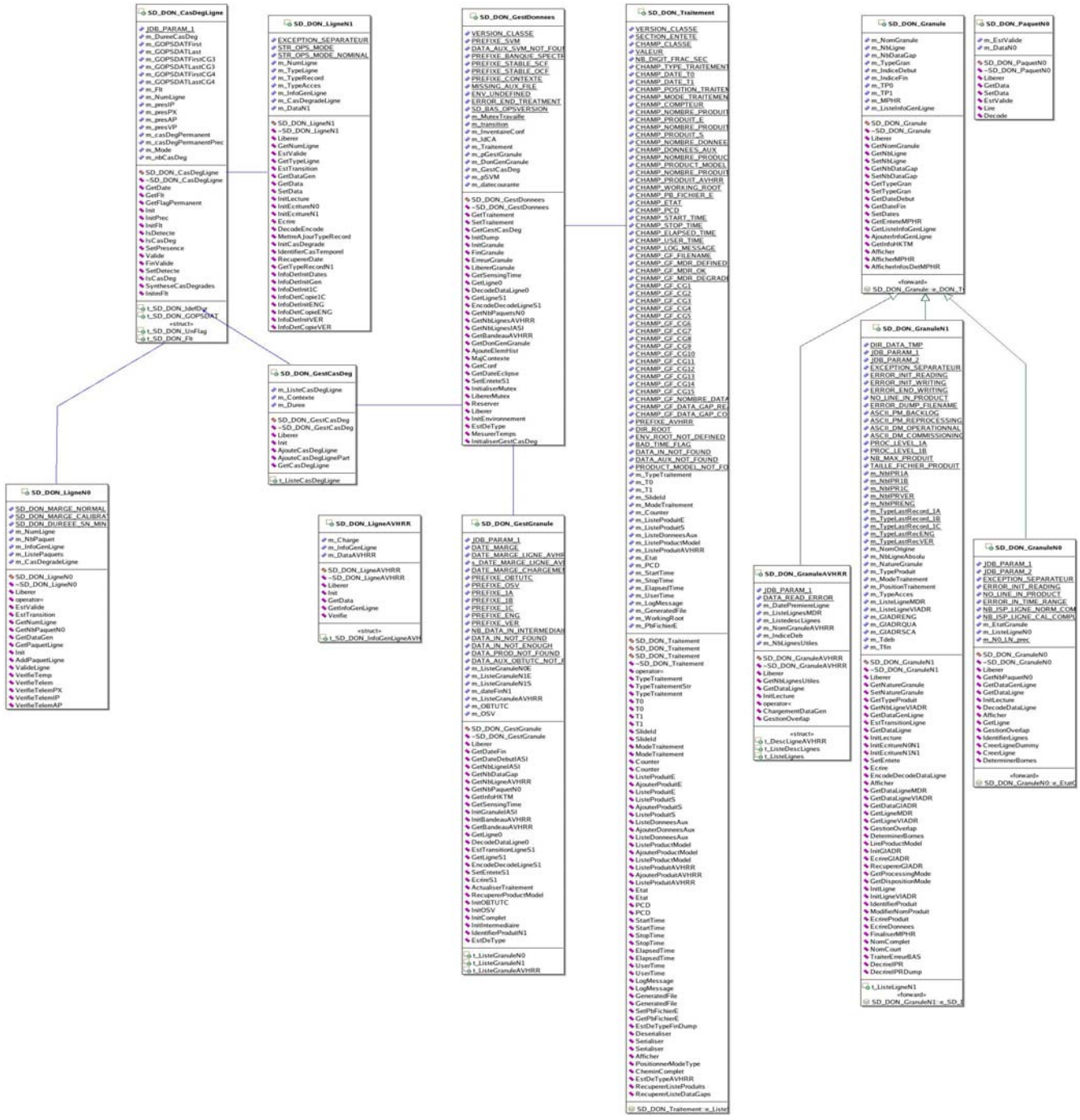


Figure 53 : Paquetage DON (données dynamiques)

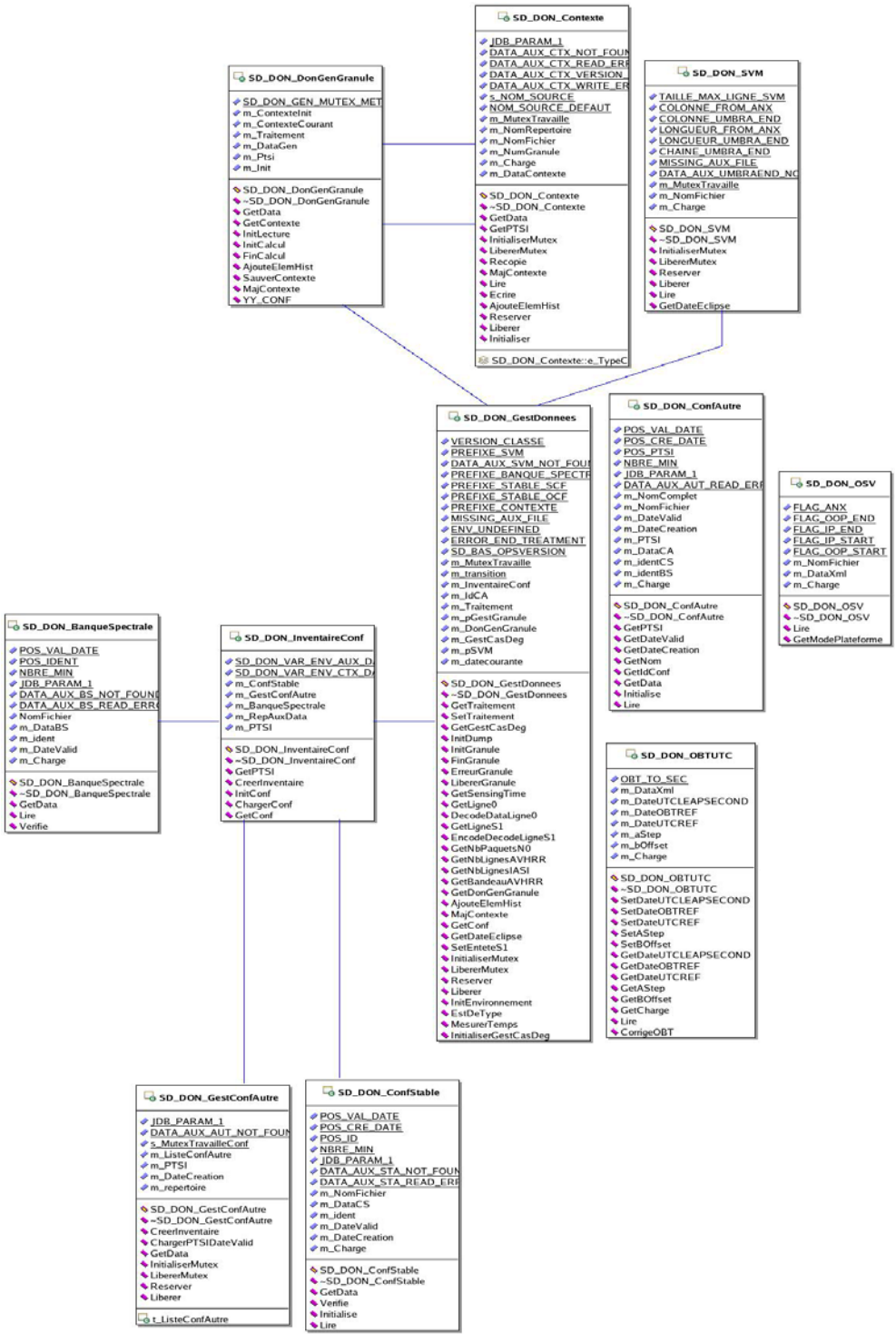


Figure 54 : Paquetage DON (données statiques)

Paquetage GES (dérivé des classes génériques de SSALTO)

Le diagramme de classe ci-dessous présente l'ensemble des classes constituant les tâches et les actions spécifiques de l'OPS IASI.

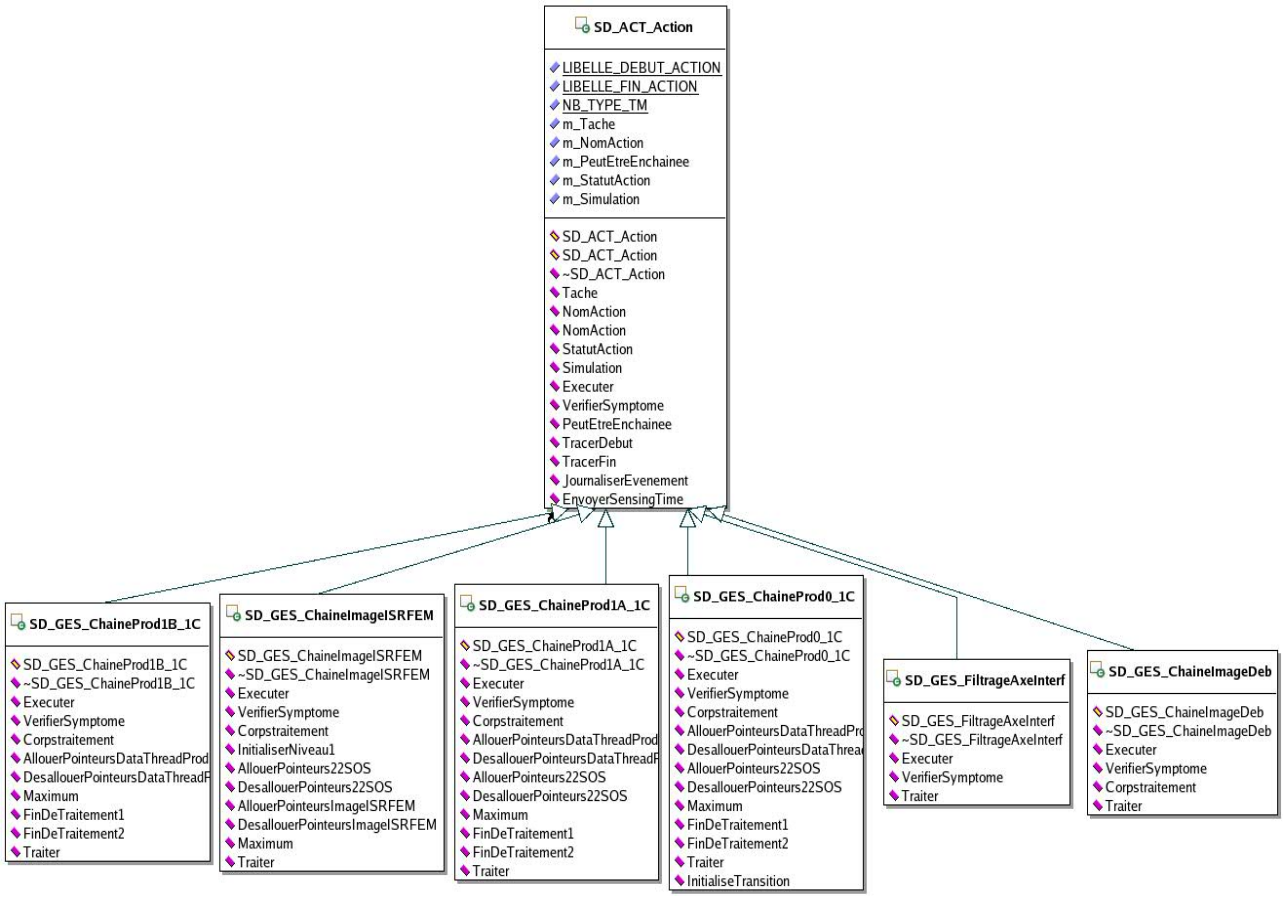


Figure 55 : Paquetage GES

Paquetage BAS (spécifique à l'OPS)

Ce paquetage regroupe l'ensemble des classes de services communs aux chaînes de traitement de l'OPS IASI.

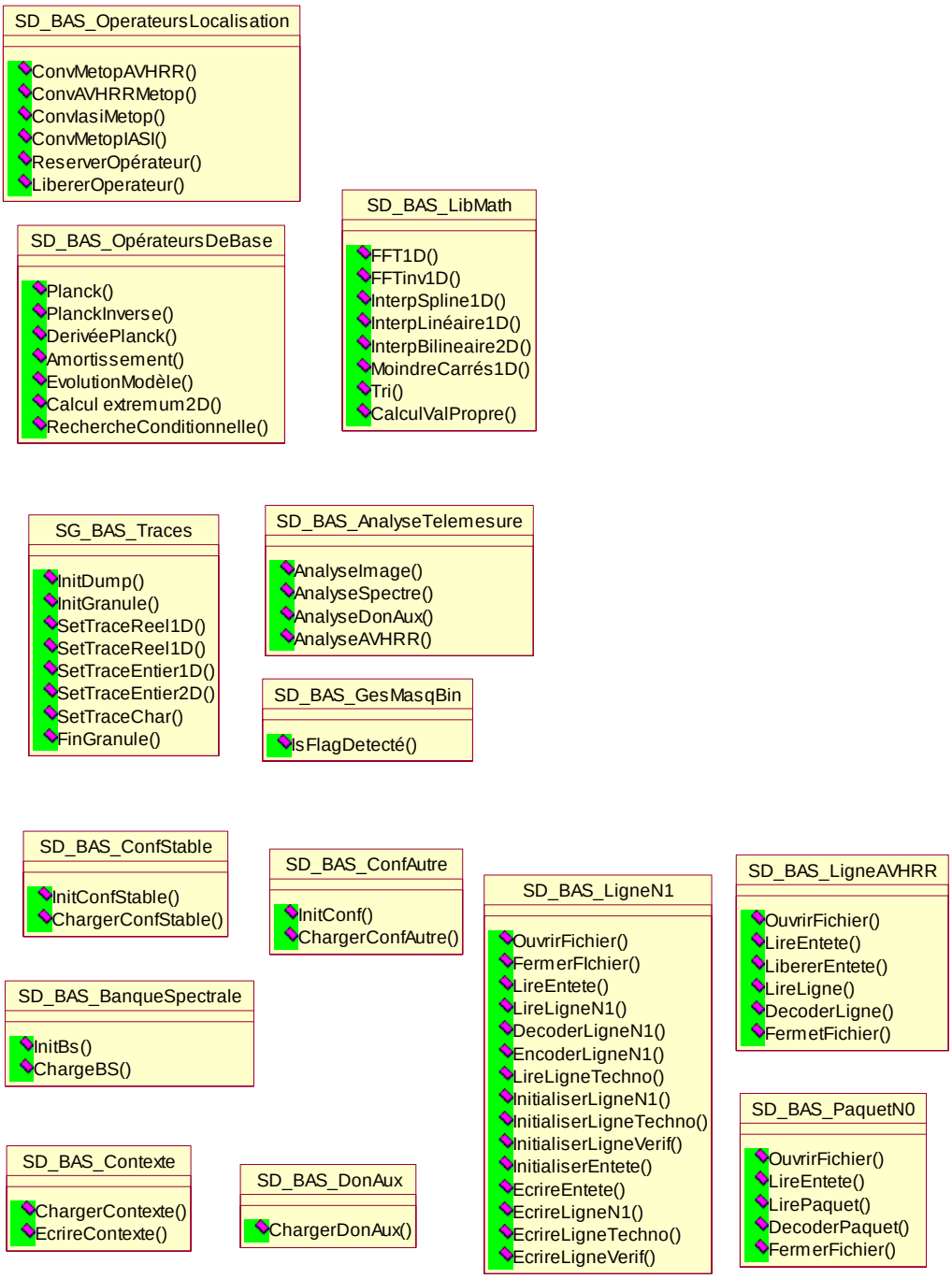


Figure 56 : Paquetage BAS

Paquetage ALG (spécifique à l'OPS)

Le diagramme de classe ci-dessous présente l'ensemble des modules constituant le paquetage ALG qui encapsule les algorithmes de l'OPS IASI L1.

Figure 57 : Paquetage ALG

5.3.LA COUCHE DE SERVICES

5.3.1.Présentation

La couche de services regroupe les processus ou les librairies qui fournissent des services utilisés par plusieurs processus.

La couche de services est composée :

des processus "Serveur de Messages", "Serveur d'événements Timer",et "JDB Server".

des librairies de Services Communs.

5.3.2.Le Serveur de Messages (MSGS)

5.3.2.1.Description Générale

Le serveur de messages a pour but de centraliser et banaliser les communications inter-processus de l'OPS. Il implémente les échanges de type point à point, 1 vers n ou n vers 1. Chaque processus s'abonne aux types de message auxquels il s'intéresse; dès qu'un message de ce type est émis, il lui est automatiquement transmis. Si plusieurs processus sont intéressés par un message, ce message est diffusé à tous les processus abonnés.

Le serveur de messages est le socle de l'architecture de l'OPS. Le processus MSGS (pour MeSsaGe Server) est un processus permanent lancé par le Main Process (contrainte CGS) au démarrage de l'OPS.

Au démarrage le "serveur de messages" se met à l'écoute (appel système **listen**) sur un numéro de port particulier qui est appelé PORT_CONNEXION, et écoute les connexions demandées par les clients. Les connexions clientes sont acceptées (appel système **accept**) et un nouveau socket est créé. Il est chargé de la communication entre le "serveur de messages" et le nouveau client. Ce nouveau socket est rajouté à la liste des sockets sur laquelle le serveur effectue un appel système **select** pour détecter des messages à lire éventuels. Ce socket reste actif tant que la connexion entre le client et le serveur est ouverte (ouverture de la connexion au démarrage du process et fermeture lors de la terminaison).

L'architecture du "serveur de messages" est présentée figure suivante :

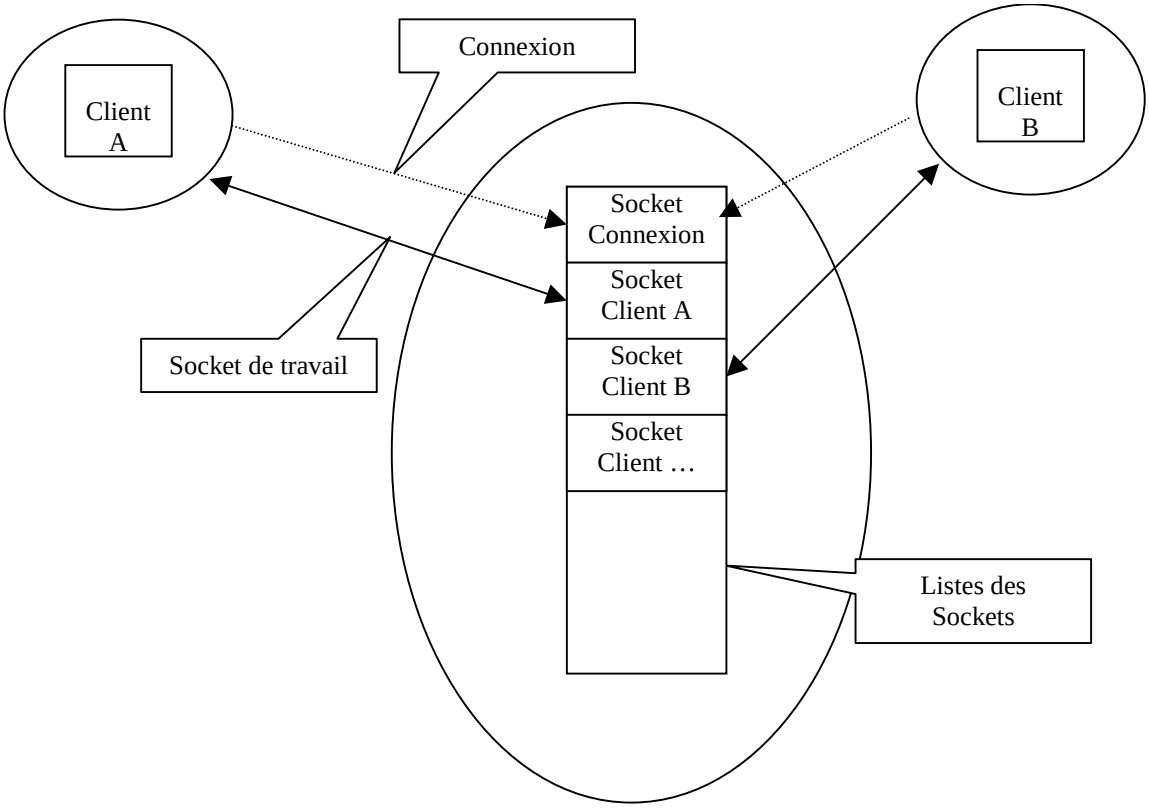


Figure 58 : Architecture du Serveur de Messages

A cette architecture, il faut ajouter la notion "d'adresse logique". Le "client A" qui se connecte au "serveur de message" s'abonne aux adresses logiques "adresse 1", "adresse 2", etc. L'abonnement est enregistré au niveau du serveur. Lorsque le "client A" publie un message sur l'adresse logique "adresse 1", le serveur reçoit ce message et le diffuse automatiquement aux "client B" et "client C" qui sont abonnés à cette adresse. Ce multiplexage est illustré par la figure suivante :

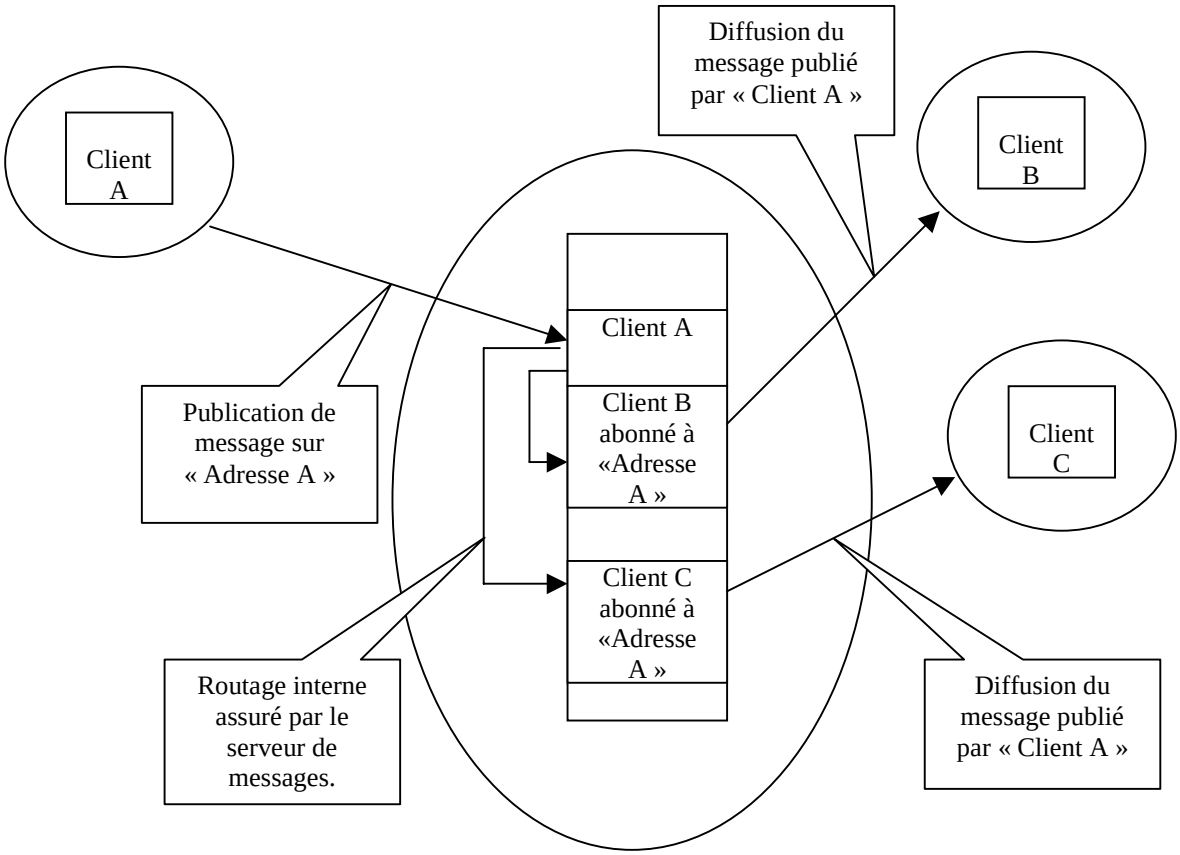


Figure 59 : Routage des Adresses Logiques

Les clients connectés au "serveur de messages" échangent deux types de messages :

les "messages de commandes" à destination du MSGS qui permettent au client de gérer ses communications : connexion, abonnement, …

les "messages applicatifs" dont la sémantique est libre qui sont échangés entre le producteur et le(s) consommateur(s) via le MSGS.

5.3.2.2.Diagramme de Classes

Le diagramme de classes de la Figure 60 représente l'architecture des classes qui composent le processus MSGS.

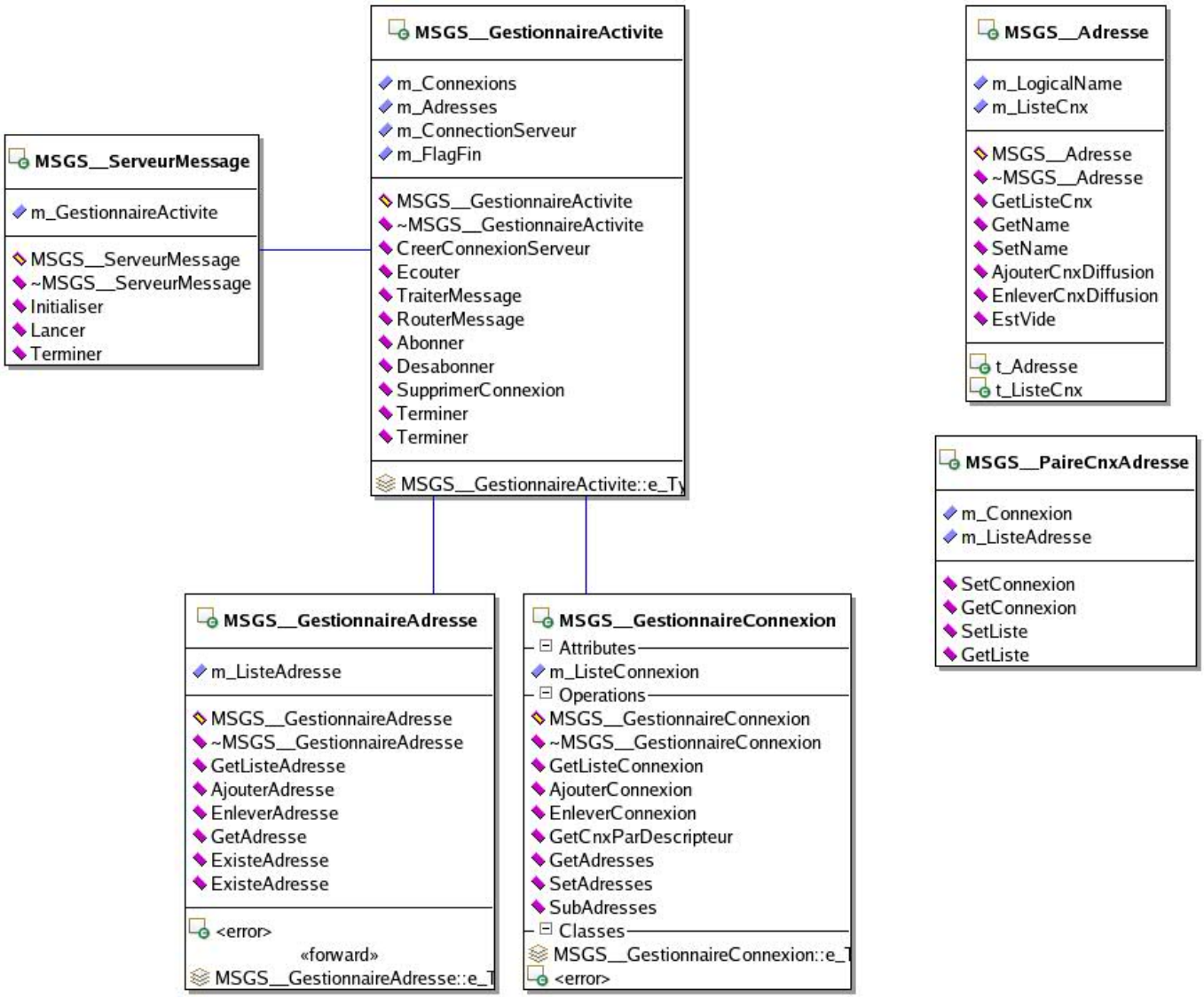


Figure 60 : Diagramme de Classes de MSGS

Une instance de la classe **MSG__ServeurMessage** est créée lors du lancement du processus MSGS.

La méthode MSGS__ServeurMessage.Initialiser permet :

de créer le gestionnaire de connexion **MSGS__GestionnaireActivite**,

de lire le numéro de port serveur et de créer la connexion serveur avec

MSGS__GestionnaireActivite.CreerConnexionServeur.

La méthode **MSGS__ServeurMessage.Lancer** permet de lancer la boucle infinie d'écoute de l'activité sur les sockets serveur et client qui est assurée par **MSGS__GestionnaireActivite.Ecouter**.

Le gestionnaire d'activité **MSGS__GestionnaireActivite** crée lors de son initialisation le gestionnaire des connexions **MSGS__GestionnaireConnexion** et le gestionnaire des adresses **MSGS__GestionnaireAdresse.**

La méthode **MSGS__GestionnaireActivite.Ecouter** permet d'écouter (appel système select) l'activité sur le masque courant des descripteur de socket (**m_MasqueSockets**).

Lorsqu'une activité est détectée sur un des descripteurs le traitement diffère selon le type de la connexion associé au descripteur.

S'il s'agit de la connexion **m_Type** = SERVEUR (port d'écoute du serveur) alors il s'agit d'une demande de connexion d'un nouveau client, qui est traité selon le diagramme de séquence de la figure suivante.

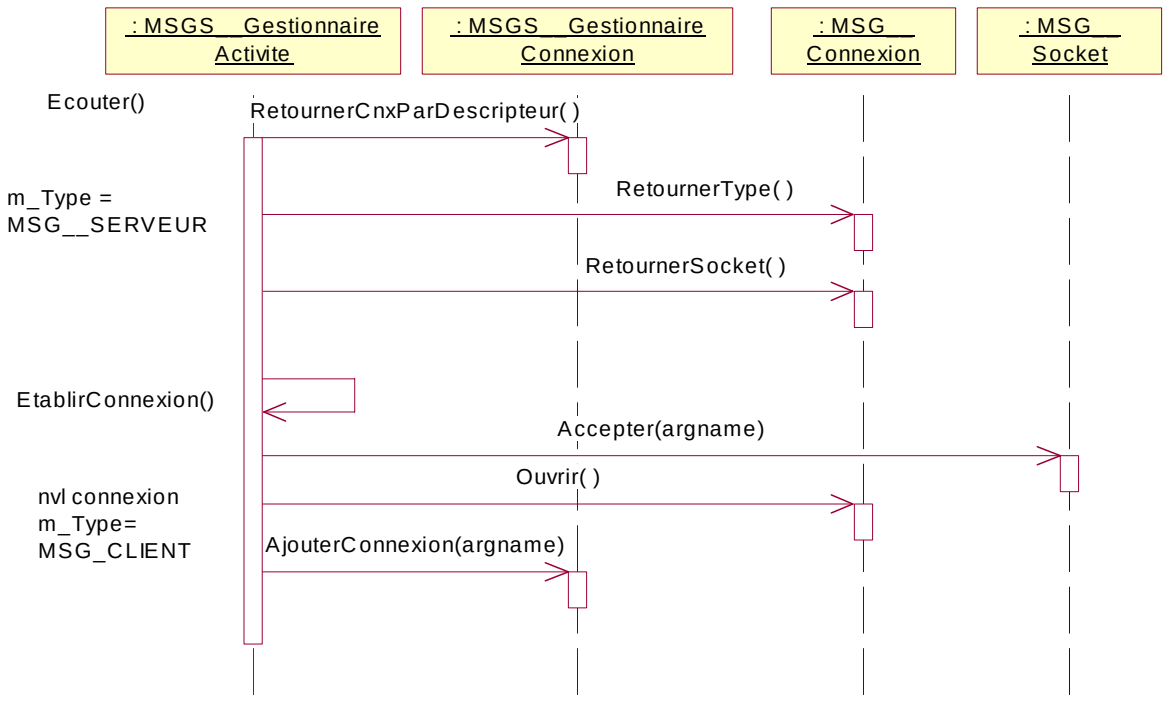


Figure 61 : Diagramme de Séquence d'une Connexion

Si c'est une connexion **m_Type** = CLIENT (liaison déjà établie entre serveur et client) alors il s'agit soit d'une requête d'abonnement/ désabonnement (cf Figure 62), soit d'une diffusion de message (cf Figure 63).

Diagramme de séquence d'une demande d'abonnement :

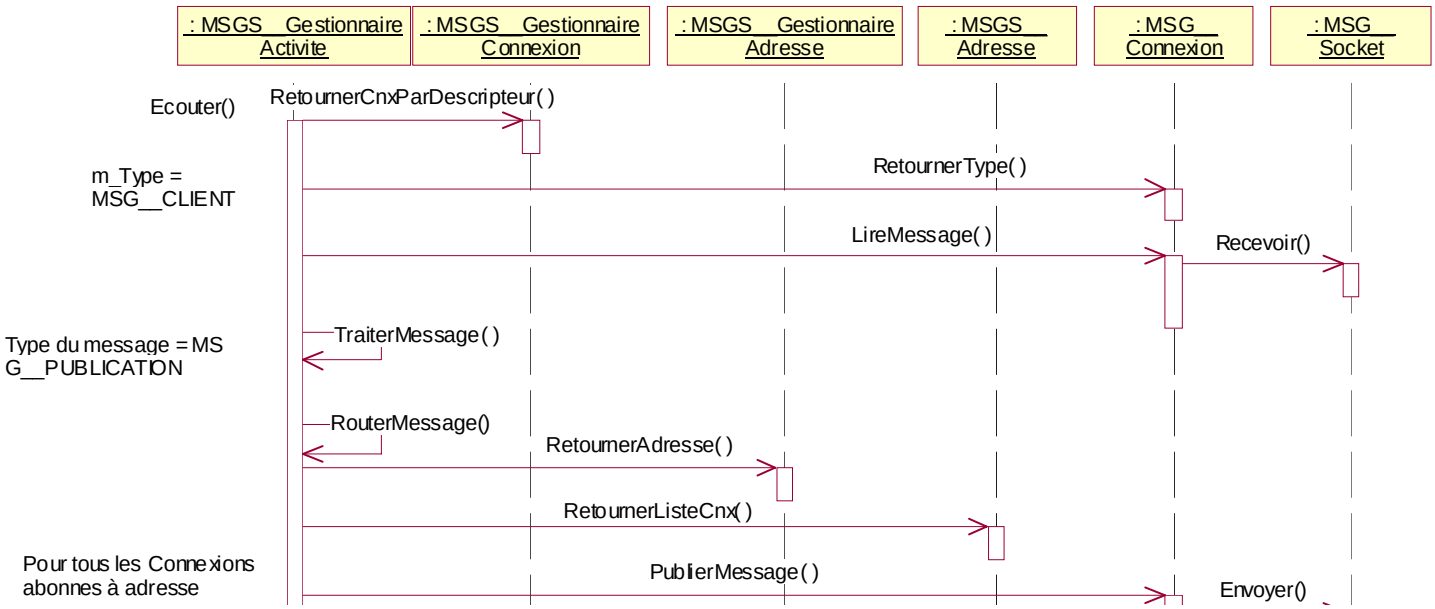


Figure 62 : Diagramme de Séquence d'un Abonnement

Diagramme de Séquence d'une Publication :

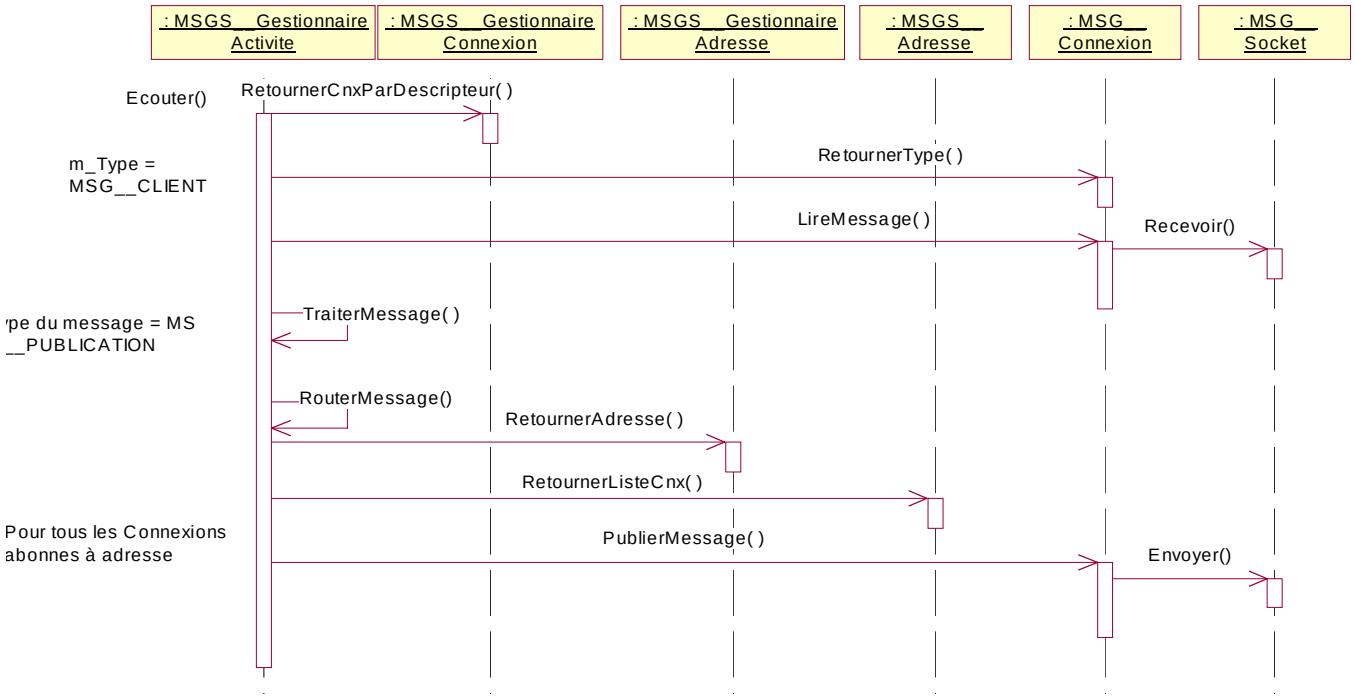


Figure 63 : Diagramme de Séquence d'une Publication

La classe MSG__Socket va toujours de paire avec MSG___Connection; afin de simplifier les schéma d'architecture, cette classe n'est pas représentées dans les schéma suivants.

5.3.3.Le Serveur d'Evénements Timer (TES)

5.3.3.1.Description

Le rôle du Serveur d'Evénements Timer (Timer Event Server TES) est de fournir un service d'alerte pour une fréquence ou à des dates programmées.

Les événements sont préalablement programmés par les processus applicatifs qui ont besoin de cadencer ou déclencher des travaux par rapport à des dates.

Dans le cadre de l'OPS, ce service est utilisé pour cadencer la collecte et la diffusion de la HKTM.

Le « serveur d'événement timer » est un processus permanent qui est lancé par le Main Process dés le démarrage de l'OPS, il se connecte au « serveur de messages » et attend que les processus applicatifs lui programment des événements à générer.

Les événements timer que peut générer le « serveur de d'événement timer » sont de deux types : des événements périodiques, des événements "date précise".

Le serveur bat en interne sur une base de temps unique basé sur un timeout au niveau du select. Tous les tops internes (déclenchement de la base de temps interne), le serveur va examiner la liste des événements programmés, et si un événement à sa date de début égale à la date courante du serveur à la résolution de la base de temps interne prés, alors l'événement est publié. Publier un événement signifie qu'un message de commande de type MESSAGE_TIMER est publié sur l'adresse logique indiquée par l'application lors de la programmation de l'événement.

La résolution du timer est donnée par la résolution de la base de temps interne. Avec une base de temps interne à 1 seconde, la résolution des événements ne pourra être meilleure que 1 seconde. Le but de ce serveur n'est pas d'offrir une précision de 1 micro-seconde et de faire du temps réel, mais plutôt de permettre aux applications de gérer simplement un grand nombre de timers logiques. Ainsi le code de l'application s'en trouve simplifié car il n'a pas à mettre en œuvre une gestion de signaux. De plus le nombre de timers logiques n'est pas limité alors que le nombre des timers systèmes est limité par processus.

Les événements périodiques sont auto-entretenus : Lorsqu'un événement périodique est publié, il programme dans la file des événements le prochain événement en tenant compte de la période.

Par ce mécanisme, la file des événements contient toujours le minimum d'information ce qui favorise la rapidité de son examen. Les événements sont triés par date d'activation dans la liste. Un événement qui est publié est retiré de la liste.

Pour le « serveur d'événement timer », les messages de commandes identifiés sont :

message de type *m_InitTimer* qui permet à un client de programmer un événement dans le serveur. Le message contient les informations suivantes : type de timer (périodique, un coup), date début validité, date fin validité, période éventuelle, nom logique du timer, adresse logique de publication de l'événement.

message de type *m_SupTimer* qui permet de détruire un événement précédemment programmé. Le message contient le nom logique de l'événement à détruire.

message de type *m_EvtTimer* qui permet de publier sur l'adresse logique le déclenchement du timer. Le message contient le nom logique de l'événement.

5.3.3.2.Diagramme de Classes

Le diagramme de classes de la Figure 64 représente l'architecture du processus TES en terme de classes.

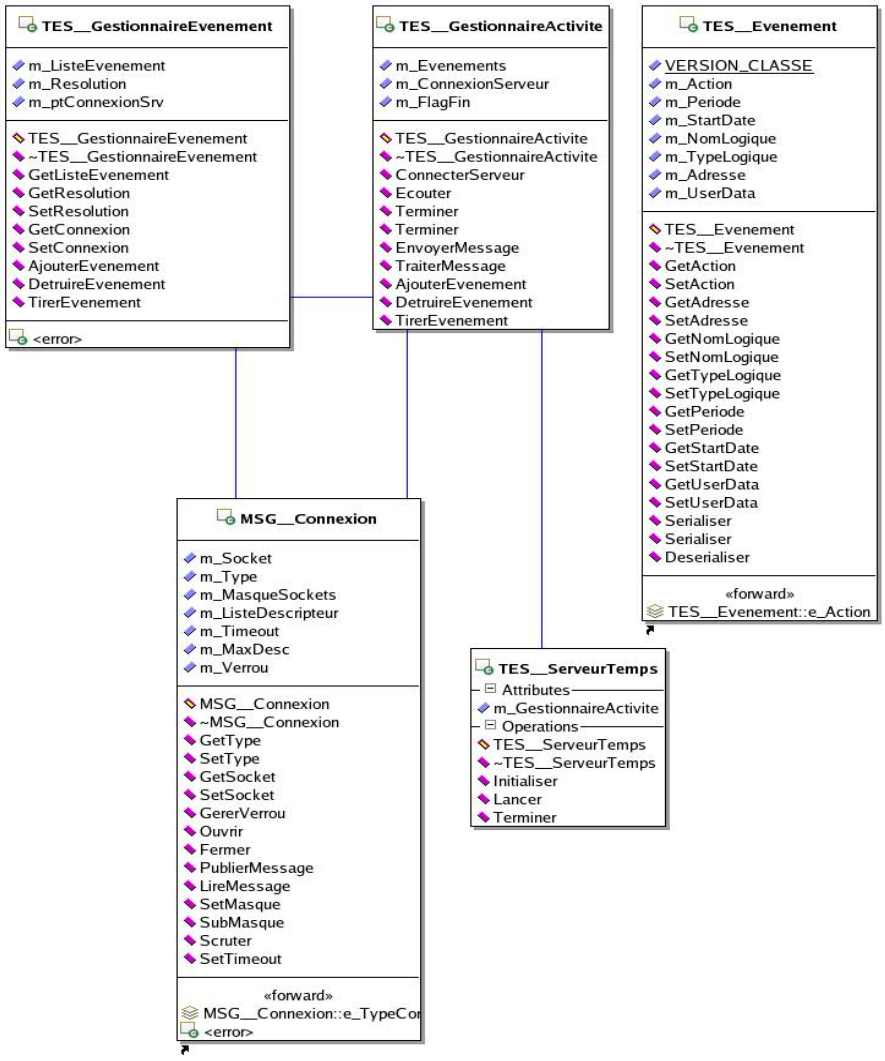


Figure 64 : Diagramme de classes du processus TES

Une instance de la classe **TES__ServeurTemps** est créée lors du lancement du processus TES.

La méthode **TES__ServeurTemps**.**Initialiser** permet :

de créer le gestionnaire des événements timers **TES__GestionnaireActivité**,

de lire le numéro de port serveur et de créer la connexion serveur avec **TES__GestionnaireActivité.ConnecterServeur,** un Timeout positionné sur la connexion avec le serveur permet gérer la base de temps interne.

La méthode **TES__ServeurTemps**.**Lancer** permet de lancer la boucle infinie d'écoute de l'activité sur la connexion avec le serveur qui est assuré par **TES__GestionnaireActivité.Ecouter**.

TES__GestionnaireActivité.Ecouter utilise **MSG__Connexion.Scruter** pour traiter la base de temps interne qui est déclenchée à l'expiration de **MSG__Connexion.m_Timeout** qui appelle la méthode **TES__GestionnaireActivité.TirerEvenement**. **TES__GestionnaireActivité.TirerEvenement** s'appuie sur **TES__GestionnaireEvenement** pour déterminer la liste des événements à publier.

TES__GestionnaireEvenement.TirerEvenement examine la liste des événements connus (triée par **TES__Evenement.m_DateDebut** croissante) pour voir si un événement doit être publié sur l'addresse **TES__Evenement.m_AdresseDiffusion**. Un événement périodique (m_Periode # 0) est auto alimenté par création du prochain événement dans la file.

Si **TES__GestionnaireActivité.Ecouter** détecte un message en provenance de MSGS, alors le message est traité avec **TES__GestionnaireActivité.TraiterMessage.** Les messages qui arrivent peuvent être de plusieurs types :

de type MSG__EvtTimer qui en fonction du contenu de **m_Action** permet d'ajouter ou d'enlever des événements,

de type MSG__EvtFin pour signaler une demande d'arrêt du processus.

Le diagramme de séquence de la figure suivante montre les enchaînements liés au traitement de la base de temps interne.

Le diagramme de séquence de la Figure 66 montre les enchaînements liés au traitement de la déclaration d'un événement timer.

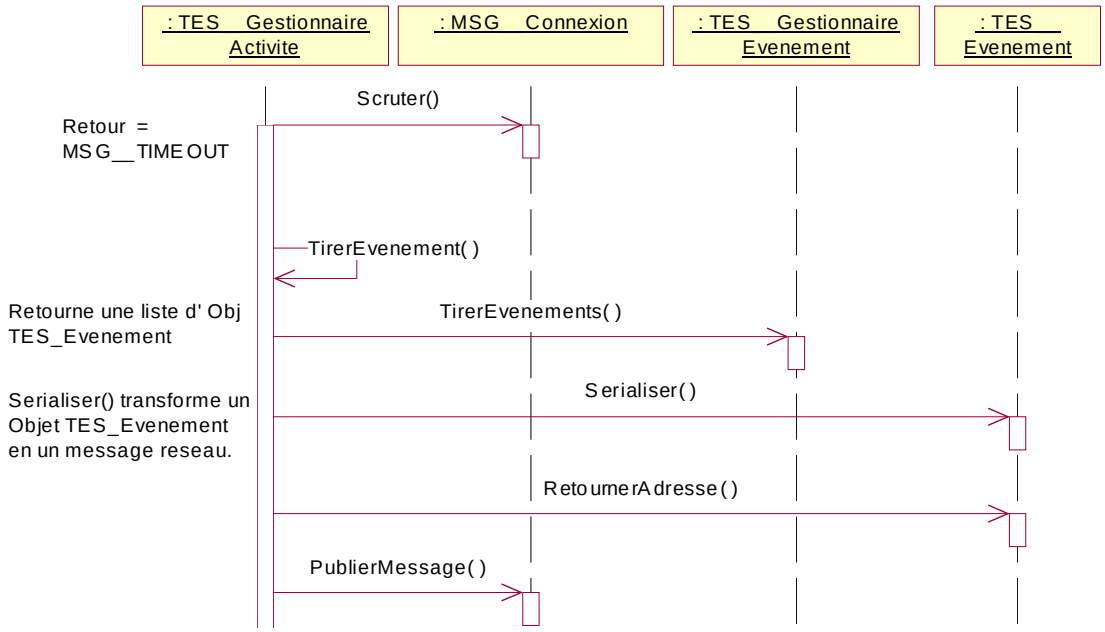


Figure 65 : Diagramme de séquence de traitement de la base de temps interne

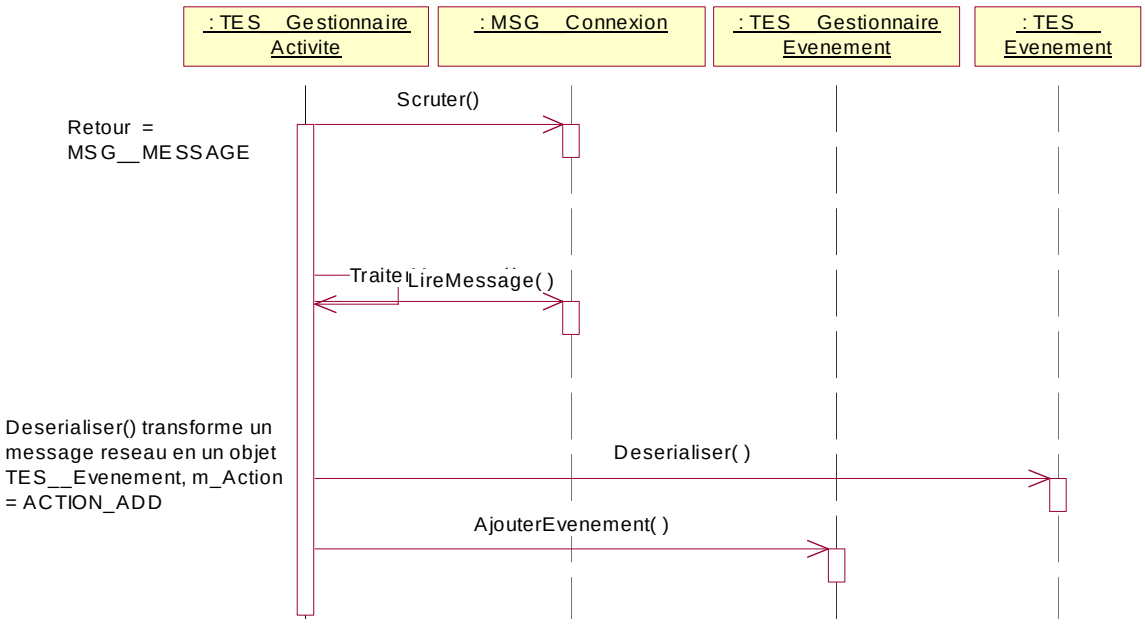


Figure 66 : Diagramme de séquence de la déclaration d'un événement timer

5.3.4.Le Serveur de Messages JdB (JDBS)

5.3.4.1.Description Générale

Le serveur de message de l'OPS a pour rôle de formater et d'émettre des messages de deux types différents via 2 API du PGE:

les messages de type *pgf_LogEvent* qui signalent les évènements importants**,**

les messages de type *pgf_TraceInfo* qui donnent une vue plus détaillée des traitements en cours.

Les messages de type Log trace ne sont émis que si l'application est compilée avec une option spécifique pour des problèmes de performances.

Ce Serveur JDB (JDBS pour JDB Serveur) est connecté à « API Message » qui reçoit tous les messages envoyés par les différents processus sur l'adresse logique « MessageJDB ». Le Serveur collecte ainsi tous les messages qui lui parviennent et les redirige vers le PGF via une API spécialisée du PGE. Cette approche permet de centraliser le formatage et l'émission de messages et ainsi supprime les problèmes de synchronisation.

Le processus JDBS est un processus permanent qui est lancé par le Main Process (contrainte CGS) dés le démarrage de l'OPS.

Les messages JDB de type *pgf_LogEvent* sont stockés dans des fichiers « Message ». Chaque message est repéré de manière unique par :

un numéro « d'ensemble de message »,

un numéro de message à l'intérieur d'un ensemble.

Chaque message contient :

le texte du message, avec des paramètres variables éventuels, renseignés par l'application lors de l'émission du message,

l'attribut de sévérité : WARNING, ALARM, INFO.

Une gestion analogue est mise en place pour les messages de type *pgf_TraceInfo.*

5.3.4.2.Diagramme de Classes

Le diagramme de classes ci-après présente l'architecture du processus JDBS en terme de classes.

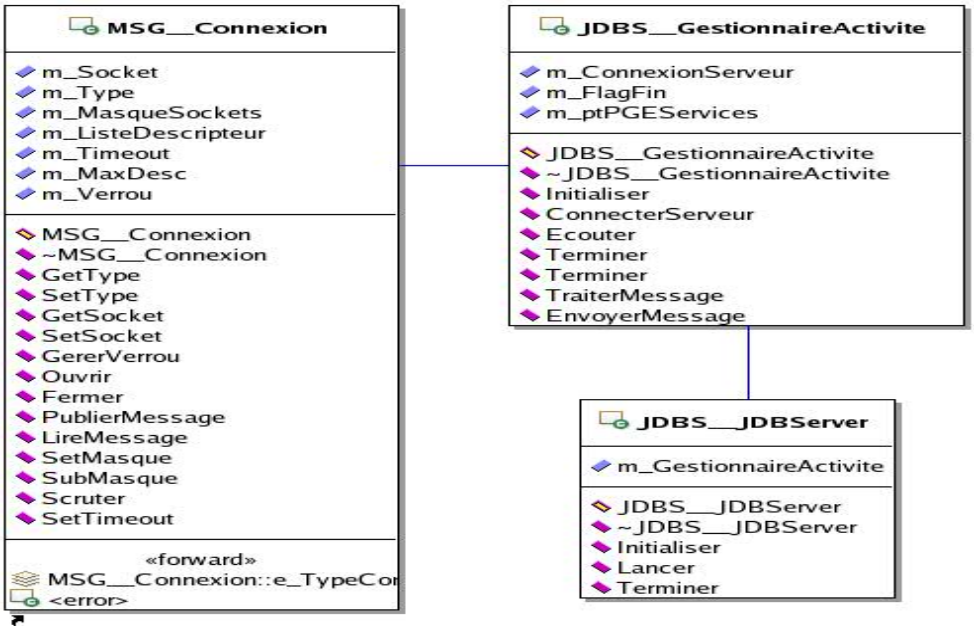


Figure 67 : Diagramme de Classes du Processus JDBS

Une instance de la classe **JDBS__JDBServeur** est créée lors du lancement du processus JDBS.

La méthode JDBS__JDBServeur.Initialiser permet :

de créer le gestionnaire des activités **JDBS__GestionnaireActivite**,

de lire le numéro de port serveur et de créer la connexion serveur avec **JDBS__GestionnaireActivite.ConnecterServeur**.

La méthode **JDBS__JDBServeur**.**Lancer** permet de lancer la boucle infinie d'écoute de l'activité sur la connexion avec le serveur qui est assurée par **JDBS__GestionnaireActivite.Ecouter**.

JDBS__GestionnaireActivite Ecouter utilise **MSG__Connexion.Scruter** pour traiter les messages qui arrivent qui peuvent être de plusieurs types :

de type MSG__EvtJDB pour signaler l'arrivé d'un message de type *pgf_LogEvent*,

de type MSG__EvtTRA pour signaler l'arrivé d'un message de type *pgf_TraceInfo*,

de type MSG__EvtFin pour signaler une demande d'arrêt du processus.

Tous les messages qui arrivent sont traités par **JDBS__GestionnaireActivite.TraiterMessage** qui aiguille les traitements en fonction du type de message reçu.

La figure ci-dessous illustre le diagramme de séquence associé au traitement de réception de message JDB.

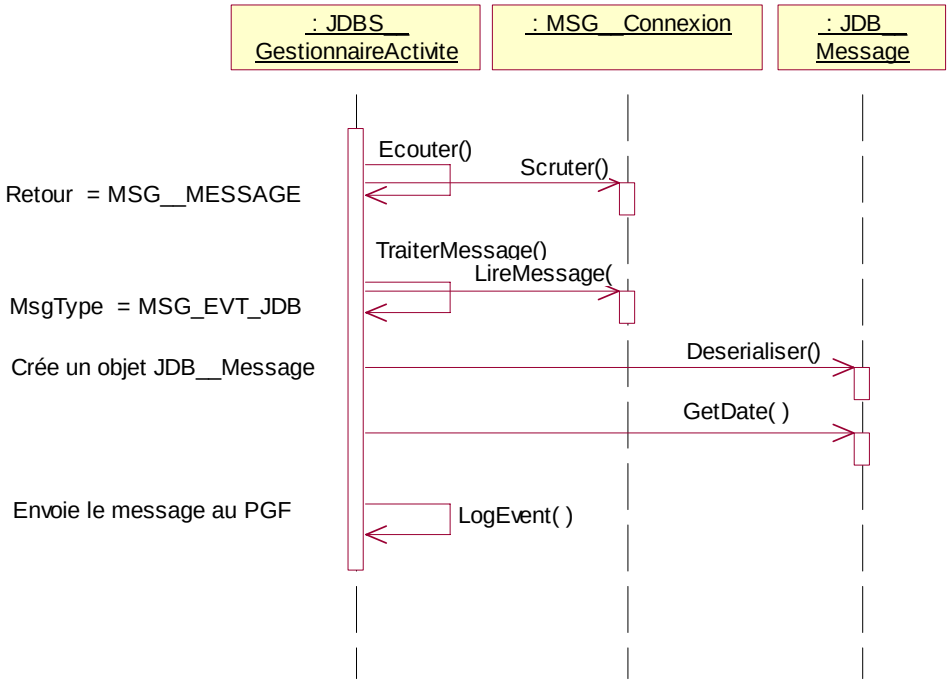


Figure 68 : Diagramme de Séquence du Traitement de l'Ecriture d'un Message JDB

5.3.5.La Librairie des Services Communs (CMN)

5.3.5.1.Description

La Librairie des Services Communs a pour but de centraliser et factoriser l'ensemble des services utilisés par plusieurs processus applicatifs de l'OPS. Elle est constituée d'un ensemble de paquetages :

GLB : classes offrant des services généraux récurrents ;

DAT : classes regroupant les modèles de données ;

OBS : classes regroupant les observateurs ;

MES : classes de gestion des messages ;

EVT : classes de gestion des événements ;

TEC : classes de gestion des tâches ;

PRS : classes offrant les services de parsing des fichiers Work Order.

La figure suivante présente l'architecture des paquetages des Services Communs.

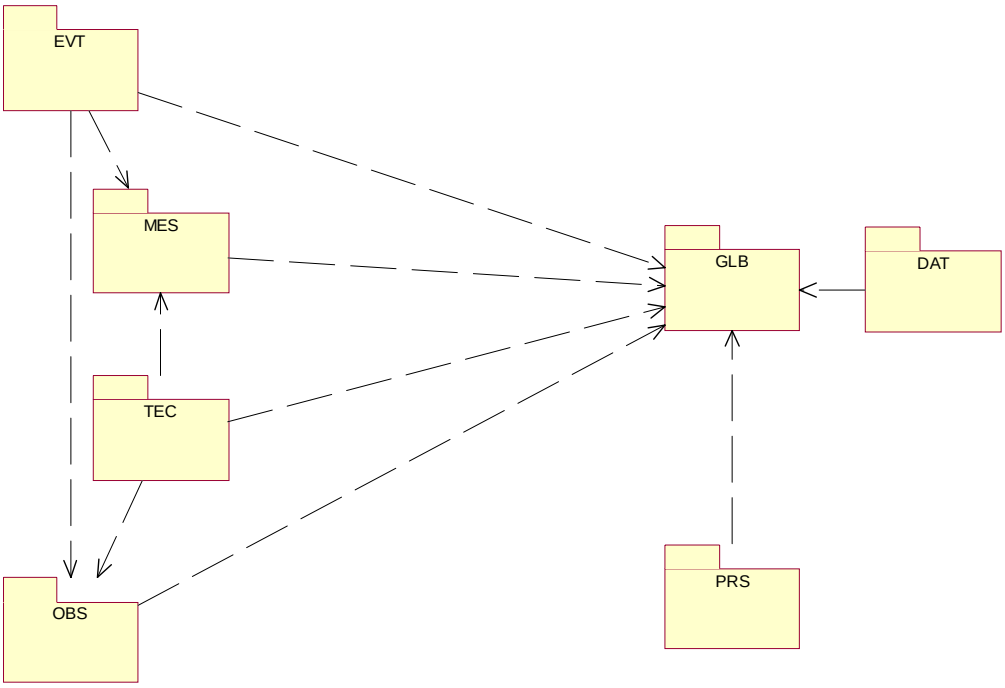


Figure 69 : Paquetage des Services Communs

5.3.5.2.Le paquetage GLB

Le paquetage GLB fournit quatre sous-ensembles de services :

Gestion des dates, des chaînes de caractères et des accès aux fichiers ;

Gestion des exceptions et des erreurs ;

Classes d'accès au journal de bord ;

Gestion des fichiers de configuration.

5.3.5.2.1.Classes de Gestion des Dates, des Chaînes et des Fichiers

Ce sous-ensemble fournit des classes pour gérer :

les dates : calculs et comparaisons sur les dates, manipulation des jours Julien standard et des jours Julien CNES ;

les chaînes de caractères : formatage, conversion en majuscule, en minuscule, … ;

les fichiers : recherche de fichiers dans un répertoire donné, manipulation de fichiers, obtention d'informations sur des fichiers, récupération de la valeur de variables

d'environnement.

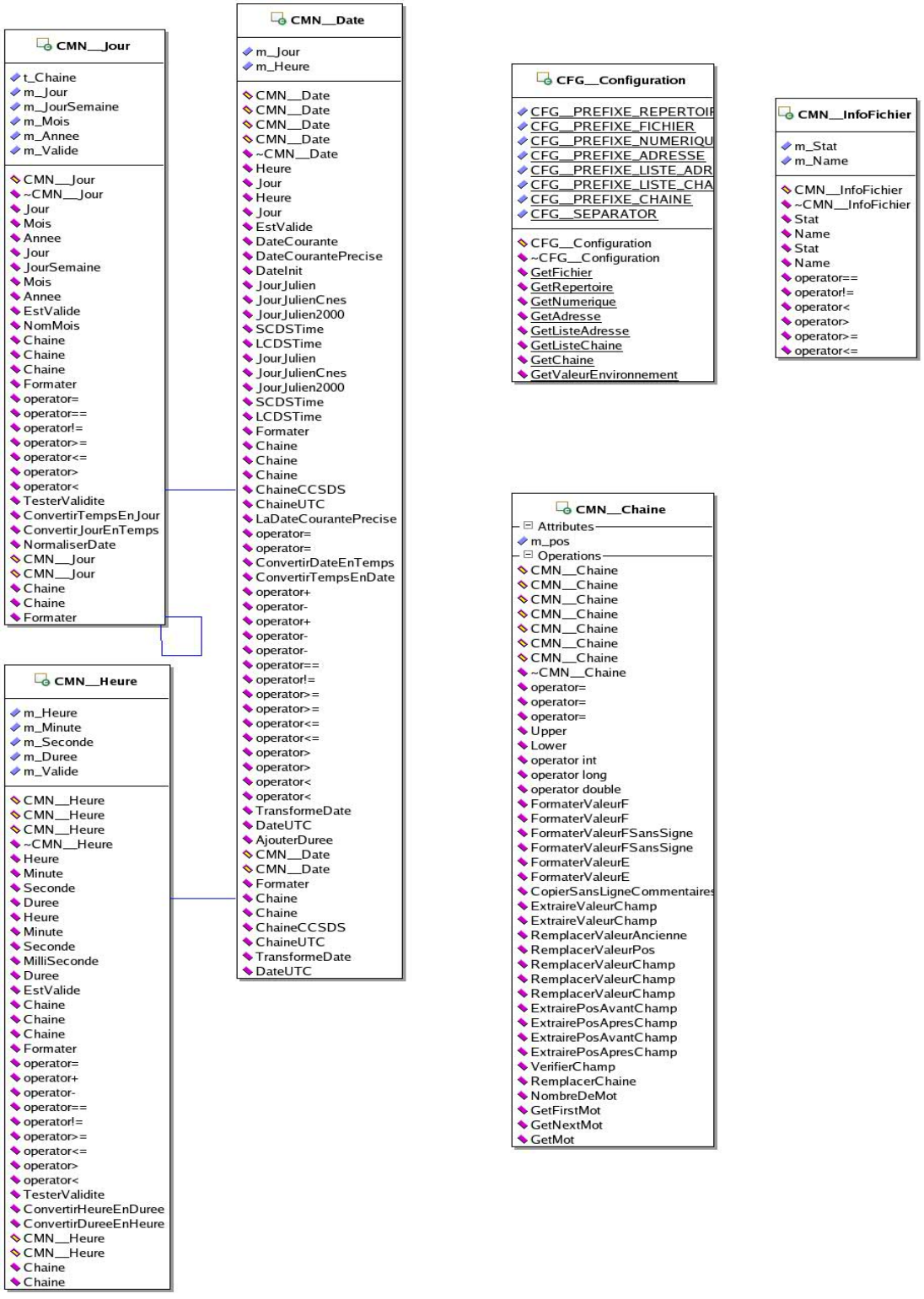


Figure 70 : Classes de Gestion des Dates, des Chaînes et des Fichiers

5.3.5.2.2.Classes de Gestion des Exceptions et des Erreurs

Ce sous-ensemble fournit des classes pour gérer les exceptions et les codes d'erreur. Le mécanisme des exceptions permet de résoudre le dilemme entre la détection des erreurs précises dans les couches basses et leur traitement spécifique à l'application au niveau des modules de niveau supérieur. Souvent, les codes d'erreur sont remontés de niveaux en niveaux et globalisés en perdant au passage de la précision dans le diagnostic.

De plus le code de traitement des erreurs « pollue » le code du traitement nominal en multipliant les branches, au point de faire perdre parfois la perception de l'enchaînement des actions. Le mécanisme des exceptions offre une solution élégante à ce problème :

lorsqu'un fonctionnement anormal est détecté (dans les couches basses généralement), un objet exception est lancé par **throw** ;

les modules supérieurs captent ces exceptions à l'intérieur de bloc **try …catch** au niveau approprié et décident de l'action à adopter (arrêt du processus, non-traitement de la données concernée etc…).

La difficulté consiste à être sûr de bien capter toutes les exceptions envoyées par les couches basses pour ne pas provoquer un arrêt inopiné du processus. Pour résoudre ce problème, nous utilisons la même **classe d'exception** pour tous les modules. Cette classe est présentée dans le schéma ci-dessous.

Elle permet de restituer précisément grâce aux attributs *m_File* et *m_Line*, qui sont remplis automatiquement lors de l'émission de l'exception, le fichier de code source et la ligne sur laquelle s'est produite l'erreur. L'attribut *m_ErrorCode* permet à l'application de positionner un numéro logique d'erreur, tandis que *m_Errno* contient la valeur de la variable Unix *errno* si l'erreur est survenue sur un appel système. L'attribut *m_Parametres* permet de stocker une liste de paramètres qui peut servir à remplir les champs d'un message JDB par exemple.

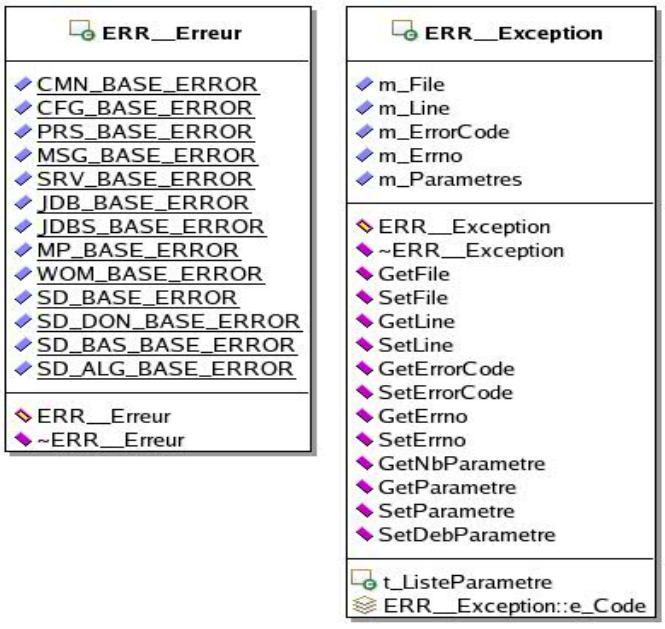


Figure 71 : Classes pour la Gestion des Exceptions et des Erreurs

Remarque : la gestion des exceptions n'est utilisée par l'OPS que dans le cadre du code SIF réutilisé. Dans le cas du SD, on utilisera pour des raisons d'optimisation une gestion plus classique.

5.3.5.2.3.Classes Journal de Bord

Ce sous-ensemble fournit des classes pour le formatage des messages et leur transmission au JDBS.

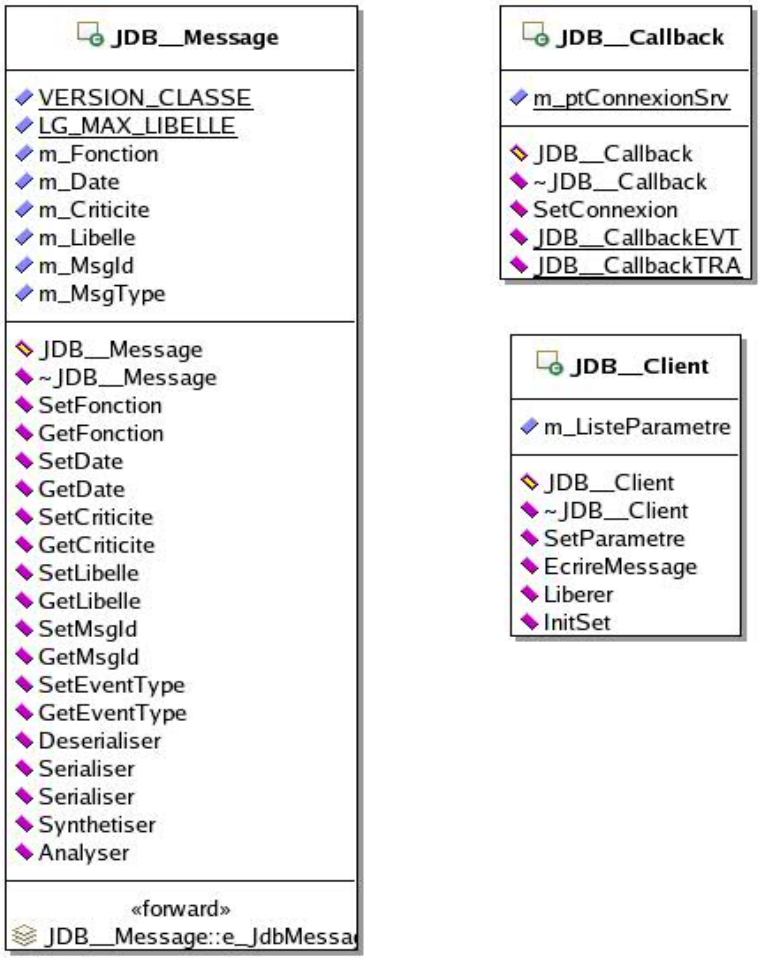


Figure 72 : Classes pour la Gestion des Messages Journal de Bord

5.3.5.2.4.Classes de Gestion des Données Configuration

La classe « Gestion des données de configuration » fournit des services pour exploiter les fichiers de paramètres, fichiers de configuration, et pour le transport d'objets.

Elle offre un service d'écriture et de lecture par rapport à un bloc de données respectant le formalisme Backus Nauer Form. Un exemple de ce formalisme est présenté ci-dessous :

```
<Configuration>          :: <Section>*
<Section>                :: [<Nom section>] <Champ>*
<Champ>                  :: <Nom champ> = <Valeur champ>
<Nom section>            :: <Chaîne de caractères sans blanc>
<Nom champ>              :: <Chaîne de caractères sans blanc>
<Valeur champ>           :: <Chaîne de caractères >| NULL
<Chaîne de caractères>   :: {A-Za-z0-9_:\/ }+
<Chaîne de caractères sans blanc> :: {A-Za-z0-9_:\/}+
```

Cette classe permet d'exploiter en entrée et en sortie un bloc de données à ce format dans un fichier ou dans une zone mémoire.

Pour le transport d'objets, il est nécessaire d'utiliser des mécanismes de sérialisation. Plusieurs classes de l'application OPS proposent des méthodes **Serialiser** (ou **Synthetiser**) et **deserialiser** (ou **Analyser**). L'action de **Sérialiser** consiste à traduire l'état interne d'un objet dans un format transportable (dans un message MSGS par exemple).

L'action de **Désérialiser** consiste à effectuer le traitement inverse et à construire un objet à partir de son format transportable.

Un objet « sérialisable » encapsule l'intelligence de sa sérialisation/désérialisation (encodage/décodage) qui ne peut être effectué que par un objet d'une même classe aux deux bouts du flot de transport.

Le format de transport retenu est l'ASCII structuré en section.

L'analyse de fichier de configuration et la sérialisation s'appuient sur la classe **CMN__Configuration**

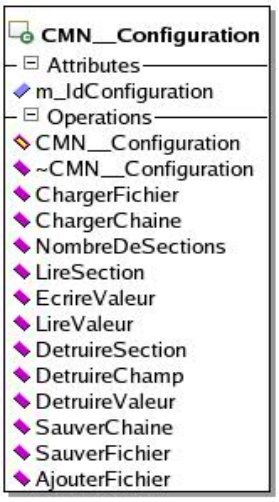


Figure 73 : Classe pour la Gestion des Fichiers de Configuration

La classe **CMN__Configuration** offre les différentes procédures de gestion d'un format configuration :

Ouvrir permet d'initialiser la gestion en rendant un identifiant de configuration à mentionner sur toutes les autres procédures.

ChargerFichier permet de renseigner le contenu d'une configuration se trouvant dans un fichier pour exploitation.

ChargerChaine permet de renseigner le contenu d'une configuration se trouvant en mémoire pour exploitation.

NombreDeSections permet d'avoir le nombre de sections se trouvant dans la configuration en gestion, c'est à dire la configuration éventuellement chargée ainsi que les sections éventuellement créées.

LireSection permet de rendre le nom d'une section de rang.

EcrireValeur permet d'écrire une valeur d'un champ dans une section, avec création éventuelle si le champ ou la section n'existe pas.

LireValeur permet de lire la valeur d'un champ dans une section.

DetruireSection permet de détruire une section avec tous les champs et les valeurs contenus.

DetruireChamp permet de détruire un champ et sa valeur dans une section, la section est détruite aussi si le champ était le dernier.

DetruireValeur permet de détruire la valeur d'un champ dans une section.

SauverChaine permet de sauver la configuration en gestion dans un espace mémoire.

SauverFichier permet de sauver la configuration en gestion dans un fichier.

AjouterFichier permet de sauver le contenu de la mémoire de configuration à la fin d'un fichier.

Exemple de fichier de configuration :

Le fichier de configuration du Serveur de Données est exploité à l'aide des services de cette classe. Un embryon de ce fichier est présenté ci-dessous.

```
[Chronos]
ChronoTache=78
ChronoEvenement=13

[EtatServeur]
Timeout=5
```

PeriodeDeScrutation=3

5.3.5.3.Le Paquetage de Gestion des Modèles de Données

5.3.5.3.1.Présentation

Pour la gestion des données, les différents composants du Serveur de Données manipulent des listes d'objets : liste d'événements, liste de tâches, ... Des classes se proposent de fournir des méthodes de construction de ces listes ainsi que des méthodes d'accès à leurs éléments.

Les classes CMN_DAT_ModeleTache, CMN_DAT_ModeleTraitement et CMN_DAT_ModeleAction sont définies lors de l'initialisation du Serveur de Données ; elles servent à stocker le contenu des fichiers de description des traitements, des tâches et des actions. Elles sont utilisées lors de la construction de l'échéancier sur la réception d'une demande de traitement de la part du WOM. Les classes CMN_DAT_Evenement sont utilisées en interne par le Serveur de Données.

5.3.5.3.2.Diagramme de Classes

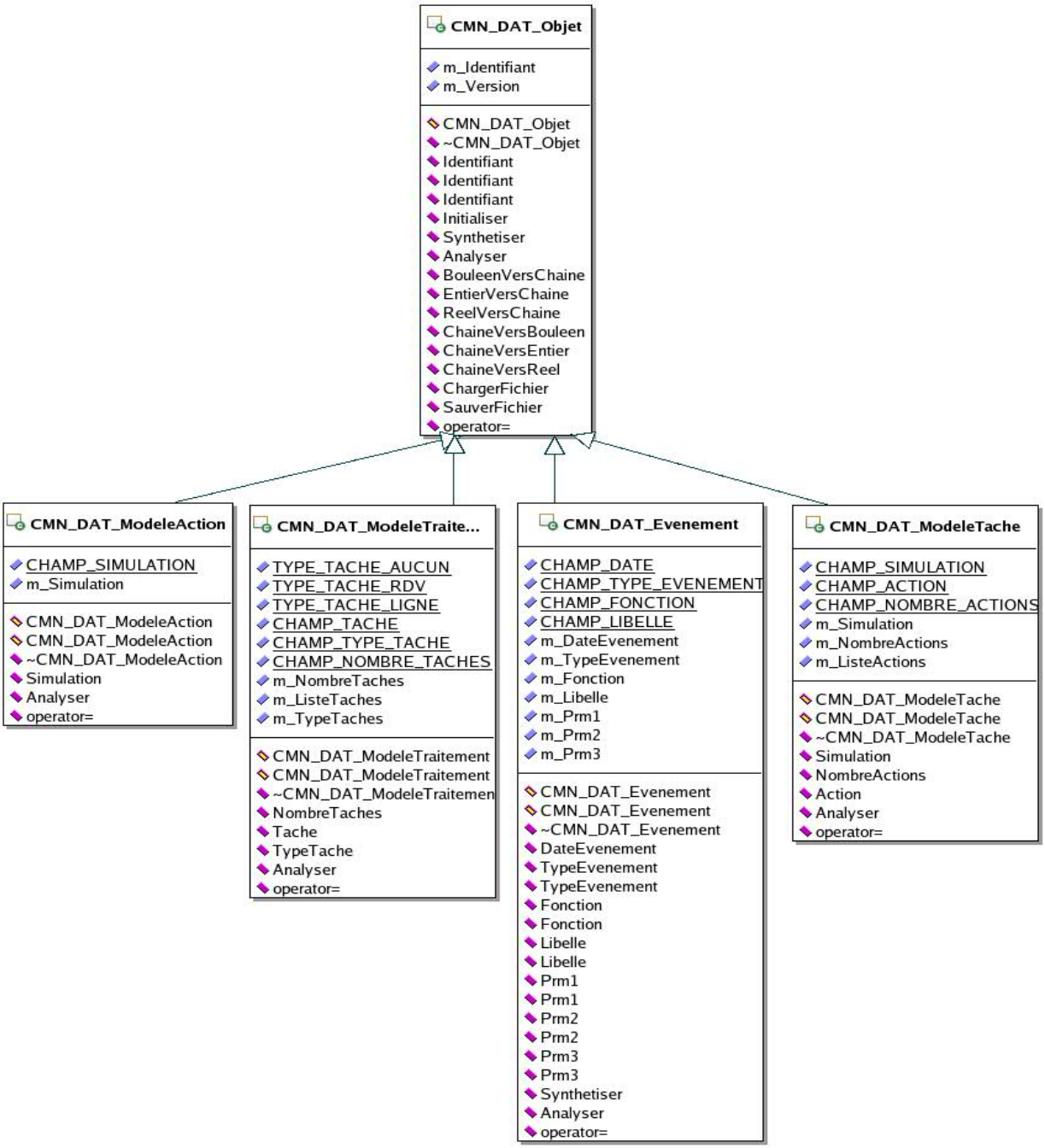


Figure 74 : Classes de Gestion des Modèles de données

La classe de base est la classe **CMN_DAT_Objet** attribuant un Identifiant à chacun des objets de configuration.

CMN_DAT_Evenement implémente les attributs des événements.

CMN_DAT_ModeleAction implémente les attributs des modèles pour les actions.

CMN_DAT_ModeleTache implémente les attributs des modèles pour les tâches.

CMN_DAT_ModeleTraitement implémente les attributs des modèles pour les traitements.

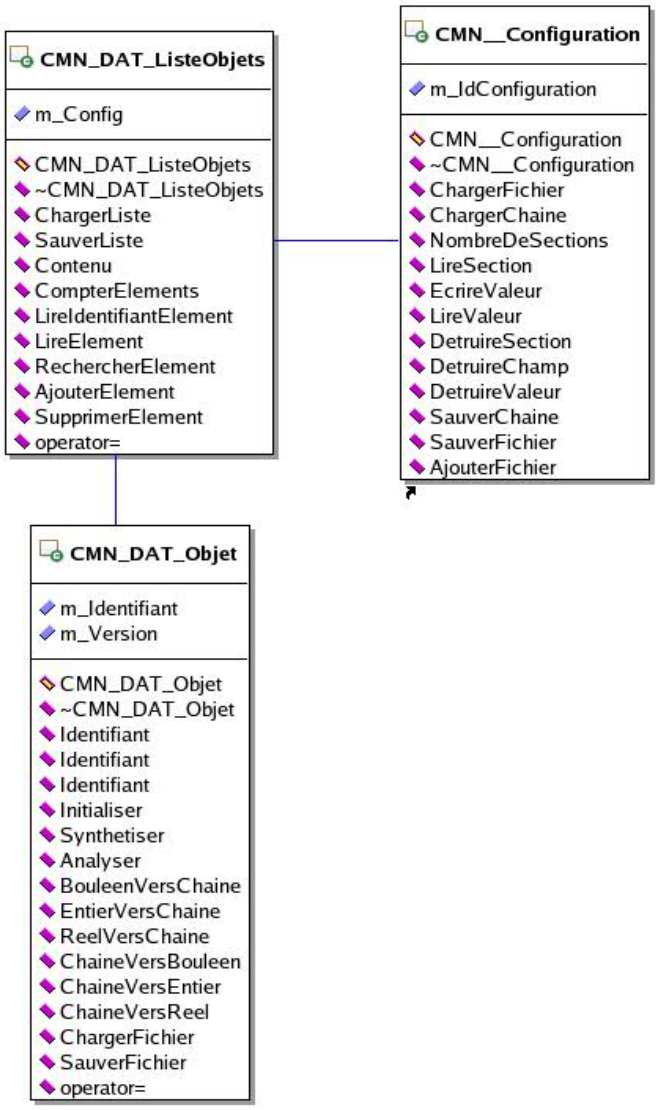


Figure 75 : Liste d'Objets

La classe **CMN_DAT_ListeObjets** s'appuie sur **CMN__Configuration** pour construire la liste des objets à partir des données d'un fichier de configuration. Chaque section du fichier de configuration correspond à un objet de la liste, le nom de la section étant l'identifiant de l'objet.

5.3.5.4.Le Mécanisme Sujet - Observateur

5.3.5.4.1.Présentation

Le mécanisme Sujet-Observateur est mis en place afin de rendre indépendant l'apparition d'un événement de ses consommateurs. Ceci est particulièrement utile dans le cas du suivi des tâches pour le serveur de données.

Le mécanisme est le suivant : les observateurs s'abonnent à un sujet. Ces observateurs sont ensuite réveillés par le sujet auquel ils se sont abonnés chaque fois que le sujet détecte un événement le concernant.

Dans le SD, ce mécanisme est utilisé par les threads lors de la terminaison d'une tâche afin d'en informer le thread principal du SD.

5.3.5.4.2.Diagramme de Classes

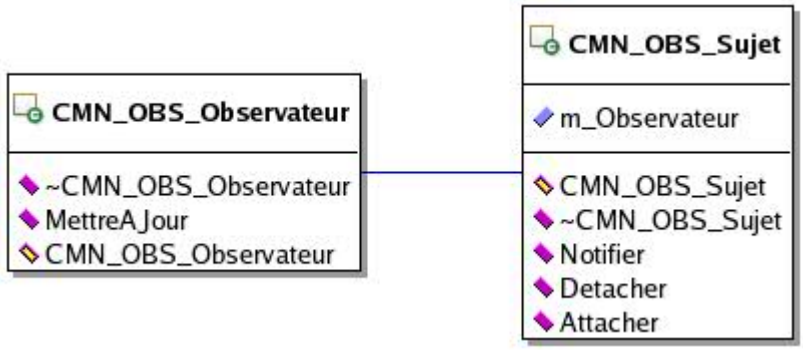


Figure 76 : Classes de Gestion du Mécanisme Sujet - Observateur

L'observateur **CMN_OBS_Observateur** s'abonne au sujet **CMN_OBS_Sujet** en appelant la méthode **Attacher** du sujet. Chaque fois qu'un événement concerne ce sujet, la méthode **Notifier** est activée. Le sujet se charge alors d'appeler la méthode **MettreAJour** de tous les observateurs qui se sont abonnés à lui.

5.3.5.5.Le Paquetage de Parsing des Fichiers XML

5.3.5.5.1.Présentation

Le « Parsing » est la procédure par laquelle on extrait des données pour les transformer dans le format interne de l'application.

Par exemple, les fichiers Work Order sont transmis et mis à disposition par le PGF dans un répertoire spécifique. Ils contiennent les informations permettant de paramétrer un traitement exécuté au niveau de l'OPS IASI Level 1. Ces fichiers sont au format XML. Les fichiers de type report sont également du même type.

Modèles de parsing :

Il existe deux modèles de "parsing" pour les fichiers XML :

Stream-parsing" ou SAX (Simple API for XML) : il s'agit d'un parsing de type "event-based" ou événementiel. L'accès aux éléments et attributs se fait à la demande, « au fil de l'eau ». Une source envoie le document dans un parseur qui fait appel à des callbacks (que l'utilisateur doit implémenter) et qui indiquent ce qui existe dans le document ; pour chaque type d'élément, la callback est différente. Ce mécanisme convient bien pour des gros documents ;

Tree-based parsing ou DOM (Document Object Model) : le fichier XML est traduit en un arbre informatique qui réside en mémoire. Cela permet d'obtenir une représentation objet du document et de son contenu. Tous les éléments et attributs sont disponibles en même temps. Ce mécanisme est pratique pour les fichiers peu volumineux.

Ces différents modèles de « parsing » sont mis en œuvre à l'aide d'API standards ; celles-ci permettent un accès normalisé aux contenus de documents XML à travers les différents parseurs du marché. Dans notre cas, nous nous appuyons sur l'API DOM.

Documents valides :

La structure d'un document XML peut être prédéfinie par une grammaire contenue dans une DTD (Document Type Definition). Un document XML est dit « valide » si :

une DTD existe ;

il respecte cette DTD (grammaire, élément racine, spécifications d'attributs)

il respecte l'intégrité référentielle : toutes les valeurs d'attributs sont distinctes et toutes les références sont valides.

5.3.5.5.2.Diagramme de Classes

La lecture et la vérification (syntaxique et sémantique) des Work Orders sont effectuées grâce à la classe **PRS__Parseur**.

La méthode **ConstruireArbreXML** permet de lire le fichier XML et de charger en mémoire l'arbre XML ; la méthode **VérificationCohérence** vérifie que le document est conforme à la DTD (*PPF_Work_Order.dtd* pour les Work Orders) et que les informations lues sont cohérentes.

Figure 77 : Classes de Parsing des Work Orders

FIN du DOCUMENT