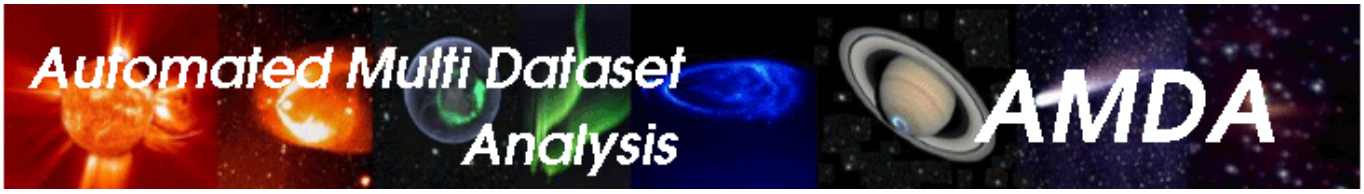
Agence ou Service : NTIC

Projet : AMDA-Kernel



DOSSIER DE
CONCEPTION DU
NOYAU AMDA-NG

Rédigé par : *Architecte* : *Gaëtan SCHNELLER*	Diffusé à : CNES/ IRAP
Approuvé par : *Chef de projet AKKA : Freddy CASIMIR* Chef de projet CNES : DUFOURG Nicolas	

LISTE DES MODIFICATIONS DU DOCUMENT

Vers.	Date	Paragraphe	Description de la modification
01.00	05/10/2012		Création du document
01.01	24/10/2012	Tous	Prise en compte du sprint 2 Release 1
01.02	05/11/2012		Prise en compte du sprint 3 Release 1
01.03	22/11/2012		Prise en compte de sprint4 Release 1
01.04	10/12/2012		Prise en compte Sprint 1 Release 2
01.05	21/12/2012		Prise en compte Sprint 2 Release 2 - Process o Standard o Resampling o TimeShifting - Ajout de fonction custom et utilisation de fonction standard (cos,sin, …) - Surcharge des opérateurs - Traitement des Interval de temps Prise en compte Sprint 3 Release 2 - Création d'un nouveau type de parameter - Création d'un nouveau type d'output - Création d'un nouveau type getParam
01.06	11/02/2013		Prise en compte Sprint 1 Release 3 Optimisation dans le but de ne compiler que si nécessaire. Prise en compte des remarques CNES et IRAP
01.07	22/02/2013	P25	Diagramme correction

SOMMAIRE

1 INTRODUCTION

L'objectif du document est de présenter les grandes lignes de la conception de AMDA-Kernel, de préciser les points clefs, sensibles.

Cette conception sera basée sur les principes des langages Objets (Le code sera implémenté en C++). Elle est basée sur la décomposition indiquée dans le dossier d'architecture du noyau d'AMDA-NG [A1]. Chaque module du dossier d'architecture sera détaillé en termes de structuration, d'interface et de comportement.

2 ARCHITECTURE

Voir le dossier d'architecture du noyau d'AMDA-NG, en particulier le paragraphe 3.

3 CONCEPTION DETAILLEE

3.1 DESCRIPTION DÉTAILLÉE DE CHAQUE COMPOSANT

3.1.1 Couche service

3.1.1.1 Composant KernelServices

AD

3.1.2 Couche métier

3.1.2.1 Composant Parameters

Ce composant a pour but :

- de gérer
    - La construction de paramètres composés.
    - L'accès aux données des paramètres basiques de la couche Persistance.
- de déléguer
    - La validation de la formule de composition au composant EvalFormule.
    - Le calcul des valeurs du paramètre composé au module EvalFormule.
    - La génération d'output aux composants Plot, DownLoad.

Le composant Parameters est principalement représenté par la classe `Parameter`.

Cette classe pilote tous les accès possibles à un paramètre :

- Récupération des données ;
- Ecriture des données ;

Les paramètres sont créés via la classe `ParameterManager`.

3.1.2.1.1 Description de la classe **`ParameterManager`**

La classe **`ParameterManager`** est l'orchestrateur et la mémoire de l'exécution d'une requête. Elle est la seule à pouvoir créer des **Parameter** et en détient la liste. Une classe qui veut un **`Parameter`** doit le lui demander.

Une classe qui veut créer un `Parameter` d'identifiant X doit demander à `ParameterManager` de le faire. Celui-ci ne le fera que si l'identifiant n'est pas déjà attribué.

Cas particulier du resampling :

La classe ParamOutput à la possibilité de demander un `Parameter` « resamplé », Si celui n'existe pas ParameterManager va le créer.

Elle détient également la liste des travaux à lancer une fois les composants de la requête construits (**Parameter, DataClient, DataWriter, …).** Ces travaux sont les **ParamOutput.** La méthode qui lance ces travaux est **« execute ».**

La méthode « execute » de **ParameterManager :**

- boucle (gestion multi-requêtes) sur les **ParamOutput** et leur demande d'établir leurs connexions (establishConnection). Toutes les dépendances entre objets se font à ce moment là.
- opère une seconde boucle (init) afin que les objets échangent leur configuration (StartTime , TimeInt, type du ParamData, Info de Calibration, , …)
- lance dans un thread différent chaque méthode « apply » des différents **ParamOutput**,
- attend la fin de ces threads.

3.1.2.1.2 Description de la classe **Parameter**

La classe **Parameter** se comporte comme un serveur. Elle a pour but de récupérer des données via son dataWriter quand un de ses clients (DataClient) en a besoin.

La classe Parameter est composée :

- d'un identificateur de parameter : _id
- d'une liste de ParamInterval. Objet contenant un startTime, un timeInt et une liste de DataClient intéressé..
- d'une liste d'information de calibration InfoValues et de producteur d'info de calibration CalibrationInfoWriter

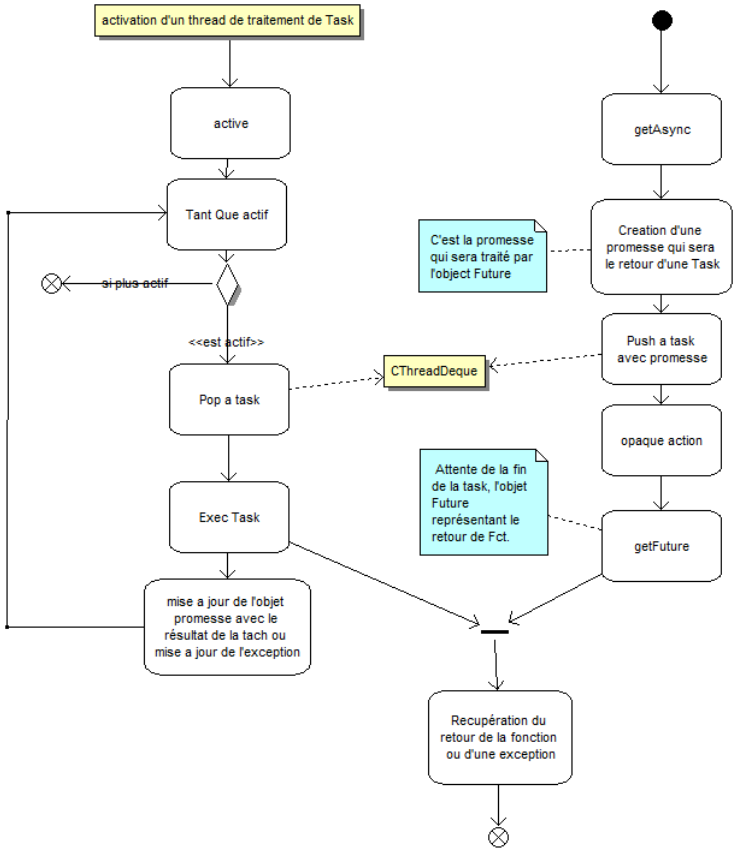
La classe Parameter agrège des interfaces :

- **DataWriter** : dont l'implémentation est responsable d'acquérir des données. Cette acquisition peut être une récupération de donnée en base (voir l'implémentation ParamGet) ou le résultat d'un calcul sur d'autres paramètres (voir les implémentations de Process). C'est elle qui référence les données acquises ou calculées via une instance de ParamData.
  - **ParamData** : son but est de stocker, de décrire et de mettre à disposition les données sous la forme la plus simple possible. Elle permet un accès direct aux valeurs de temps et a une implémentation par type de données stockées sous la forme d'un template **ParamDataSpec<typename Type>**.
- Une liste de **DataClient** : Ces DataClient sont utilisateurs des ParamData. Pour chaque DataClient, Parameter mémorise les données utilisées par ce client. Les clients s'abonnent au Parameter par l'intermédiaire de la méthode « openConnection » dans un premier temps. Ensuite ils fournissent à Parameter leurs informations de configuration (StartTime, TimeInterval) via la méthode « init ».

Lorsque le dernier client a passé ses paramètres de configuration, le paramètre met ses ParamInterval dans un mode actif. C'est à dire qu'il lance un thread par ParamInterval dans lequel ils répondent séquentiellement à toutes les demandes de données de leurs DataClient.

Les clients préviennent qu'ils ont besoin de nouvelles données grâce à la méthode « getAsync ».Ils se mettent en attente de l'objet `future` (**`ParamDataIndexInfo`**) retourné par cette méthode. Cet objet `future` contient une description des données accessible par le client (premier index de données et nombre de données suivant cet index).

En programmation, les notions de futures, promises ou delay font référence à des techniques de synchronisation pour certains langages concurrents. Il s'agit d'abstractions qui servent de proxy pour un résultat non-connu au moment où il est référencé pour la première fois, car son calcul et/ou son obtention se fera « plus tard » à l'exécution. Voir le diagramme d'activité suivant :



Les implémentations de DataWriter sont :

- **`ParamGet`** : dont une implémentation doit être capable de récupérer les données du paramètre pour l'intervalle de temps spécifié. Ces implémentations seront décrites dans les composants respectifs (Couche Persistance).
- **`Process`** : c'est une classe abstraite qui a la particularité de devoir implémenter DataWriter et DataClient. Elle contient une expression (string) décrivant le `Process` et une **Operation** dont une implémentation sera chargée des calculs spécifiques.
    - **`ProcessStandard`** en est la principale implémentation. Cette classe doit « parser » l'expression de process et générer un source qui doit être compilé, chargé et enfin exécuté dans le but de changer les Data de ParamData. C'est lui qui représente le composant EvalFormule.
    - D'autres implémentations sont à consulter dont **`ProcessResampling`** `et` **`ProcessTimeShifted.`**

Les implémentations de DataClient

- **`ParamOutput`** : dont une implémentation doit générer l'output. Ces implémentations seront décrites dans les composants respectifs.

  Ce sont les `ParamOutput` qui sont les clients primaires des paramètres, ce sont eux qui pilotent la chaine `establishConnection`, `init`, `getAsync`, `write`.
- **`Process :`** Après avoir « parser » l'expression, l'objet de classe Process détermine de quels paramètres il est client (DataClient) et s'enregistre auprès d'eux dans la phase `establishConnection,` puis pilote ses serveurs de données avec les méthodes `init`, `getAsync`, `write.`

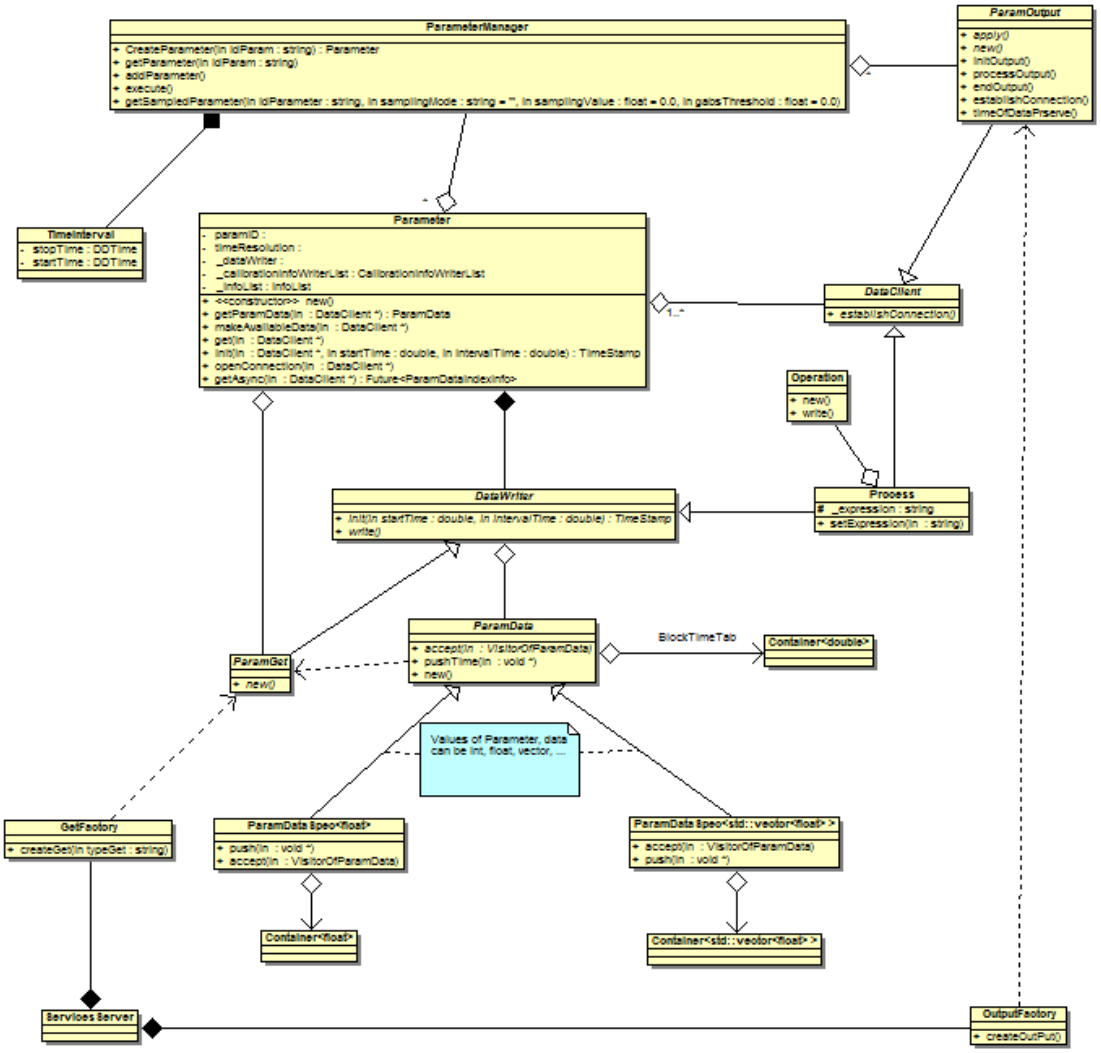


Figure 1 Diagramme de classe Parameter

3.1.2.1.3 Processus de construction de Parameter

La construction d'objet de la classe `Parameter` est de la responsabilité de la Classe `ParameterManager`.

`ParameterManager` devra donc configurer un `Pamareter` à l'aide d'un `ParameterConfigurator`.

XMLParameterConfigurator est l'implémentation de ParameterConfigurator dont le but est de lire dans un fichier XML (voir [A2]) décrivant le paramètre. L'abstraction ParameterConfigurator permet ainsi de prévoir de futures évolutions où un paramètre ne serait pas décrit par un fichier XML mais par un autre moyen (base de données, …).

Le « parse » du fichier XML est fait à l'aide d'un parcours de l'arbre xml via la librairie libxml2. Il utilise les différents `NodeCfg` préenregistrés (l'enregistrement est effectué dans les `registerPluging` des différents plugins de « parse ») pour « parser » les différentes parties du fichier XML (partie commune au paramètre, partie spécifique au type de GET).

Lors de la lecture de la balise « <process> », l'expression lue est modifiée pour traduire la balise « <time_resolution> ». Les fonctions resampling (correspondant au time_resolution) sont ajoutées à chaque parameter.

Exemple : l'expression « $imf - #time_shift($imf) » donne « #resampling($imf;60;0) - #time_shift(#resampling($imf;60;0)) » si le fichier contient la balise time_resolution et que celle-ci indique 60 secondes.

Attention, c'est de la responsabilité de `ParameterManager` de s'assurer qu'il n'y a jamais, à un instant donné, deux mêmes objets `Parameter` créés.

Le diagramme suivant a pour but de montrer comment les classes présentées ci-dessus se comportent dans le cas d'une demande de création d'un fichier ASCII contenant les valeurs du **sinus** d'un paramètre simple nommé **« A »**. Il met en évidence le processus de création d'un objet de type `Parameter`, la récupération des données et l'écriture du fichier ASCII.

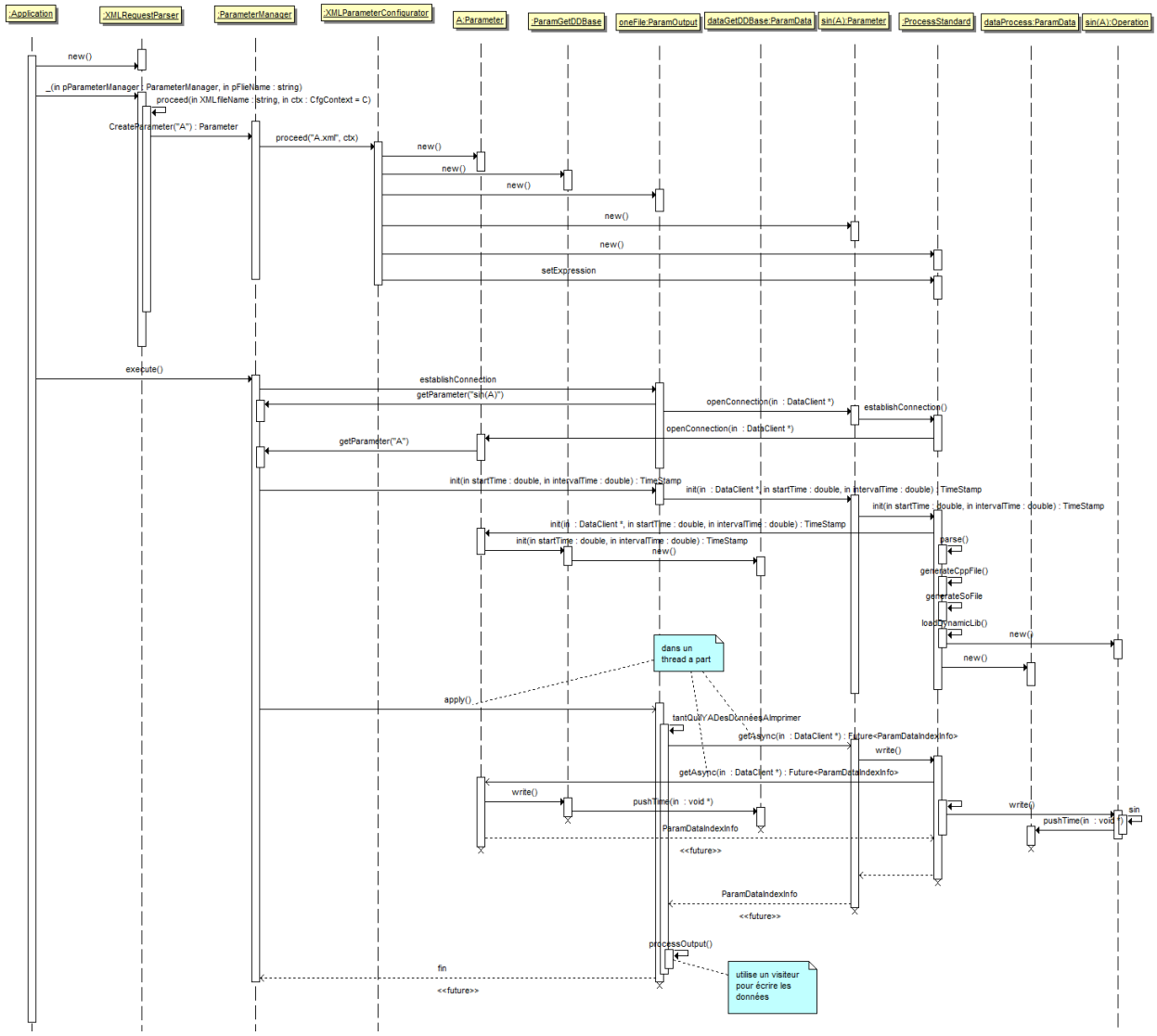


Figure 2 Diagramme de Creation de Parameter

Ce diagramme fait apparaître une classe supplémentaire :

- `Application` qui représente la couche service (fonction `main()`)

3.1.2.1.4 Processus de récupération des données

La récupération des données se fait par l'intermédiaire de l'interface `ParamGet`.

`ParamOutput` demande un bloc de données à `Process` (le `DataWriter` de `ParamOutput`), qui demande (si nécessaire) à son tour des données à ParamGetDDBase (implémentation de `ParamGet`, le `DataWriter` de `Process`) .Celui-ci récupère un bloc de données sur `DD_server`. Les données qui ne sont plus utilisées par un `DataClient` sont libérées.

3.1.2.1.5 Processus de «Process»

L'objet de type `ProcessStandard` est mis à jour avec l'expression de process lors du « parse » du fichier XML décrivant le paramètre par la classe `ProcessNode`.

Lors du « parse » de l'expression, si l'objet ProcessStandard rencontre le caractère # ,il comprend que la fonction qui suit ce caractère est une fonction Process. C'est-à-dire un nouveau Parameter à créer avec pour DataWriter l'implémentation du nom qui suit le '#'. (Exemple Resampling et TimeShifted)

L'objet de class `Process` créer un nouveau `ParamData` d'un type résultant de l'opération appliqué `ParamData` existant.

Exemple si l'objet ProcessStandard contient l'expression $IMF*$DST ou IMF et DST sont des vectors de float, à la compilation de notre classe opération spécifique, le compilateur déterminera que le type du ParamData résultat sera float puisque cette opération est un produit scalaire.

3.1.2.1.6 Processus de génération de output

La création d'un output dépend du type de paramètre et de la sortie voulue.

Il faudrait que l'on puisse ajouter aux classes `ParamData` des méthodes pour chaque type d'output. Cela rendrait l'application rapidement impossible à maintenir. Nous avons donc choisi d'utiliser le design pattern « visiteur » pour pouvoir ajouter dynamiquement des comportements aux classes `ParamData`.

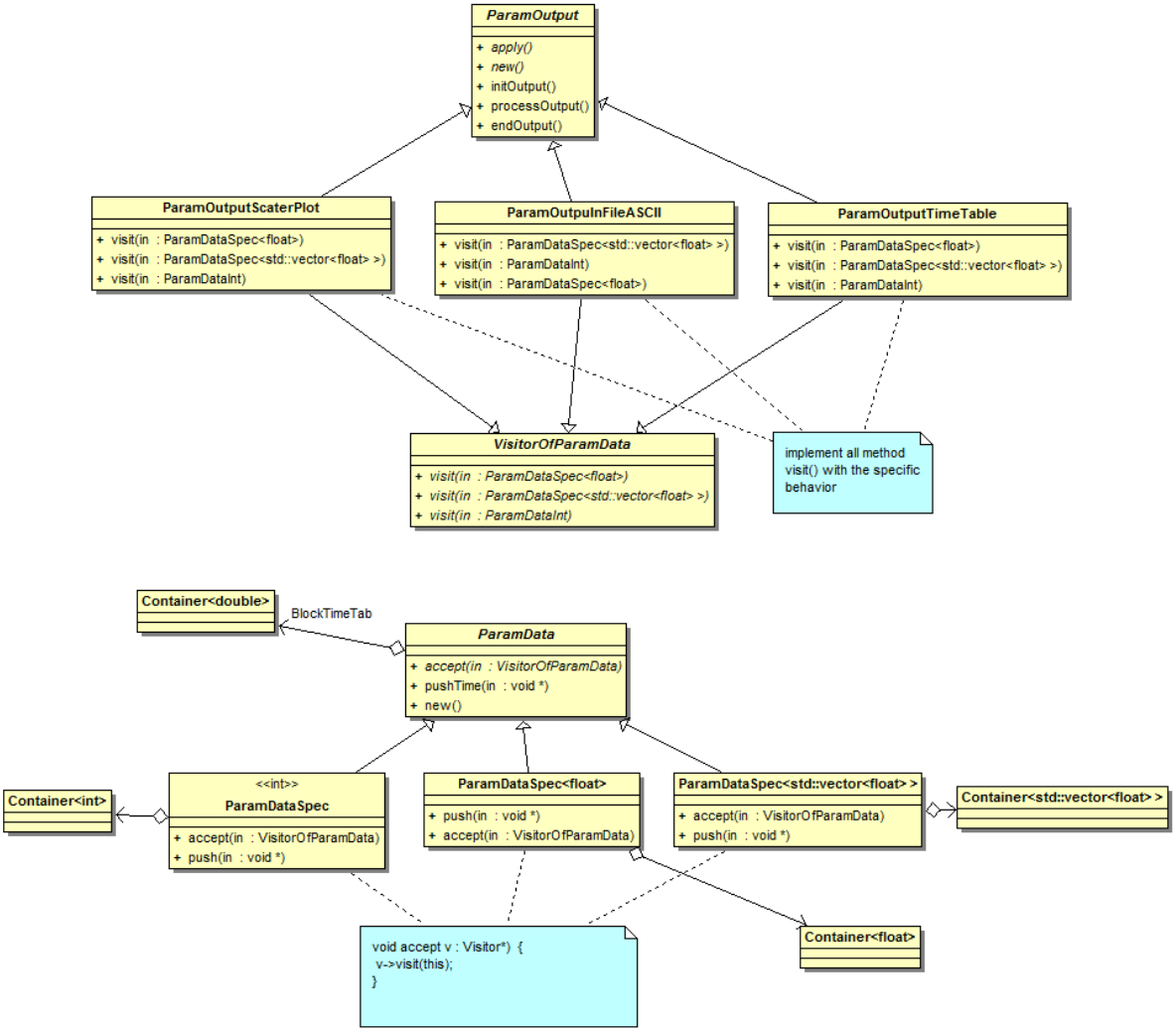


Figure 3 Design pattern visitor pour ParamOutPut

On pourra ainsi « demander » à un objet de type `ParamData` de se transformer en output par l'intermédiaire de sa méthode `accept()`. Le code de cette transformation sera bien entendu contenue dans une implémentation de `ParamOutput`, dans les différentes méthodes `visitor()`. Ainsi l'ajout d'un nouveau type de sortie se ramène à l'écriture d'une nouvelle classe implémentant `ParamOutput`. Pour chaque type de `ParamData` existant, (double, float, tableau de float, matrice de float, …) on implémente la méthode `visitor()` associée.

3.1.2.1.7 Principales implémentation des classes abstraite du composant Paramétrers

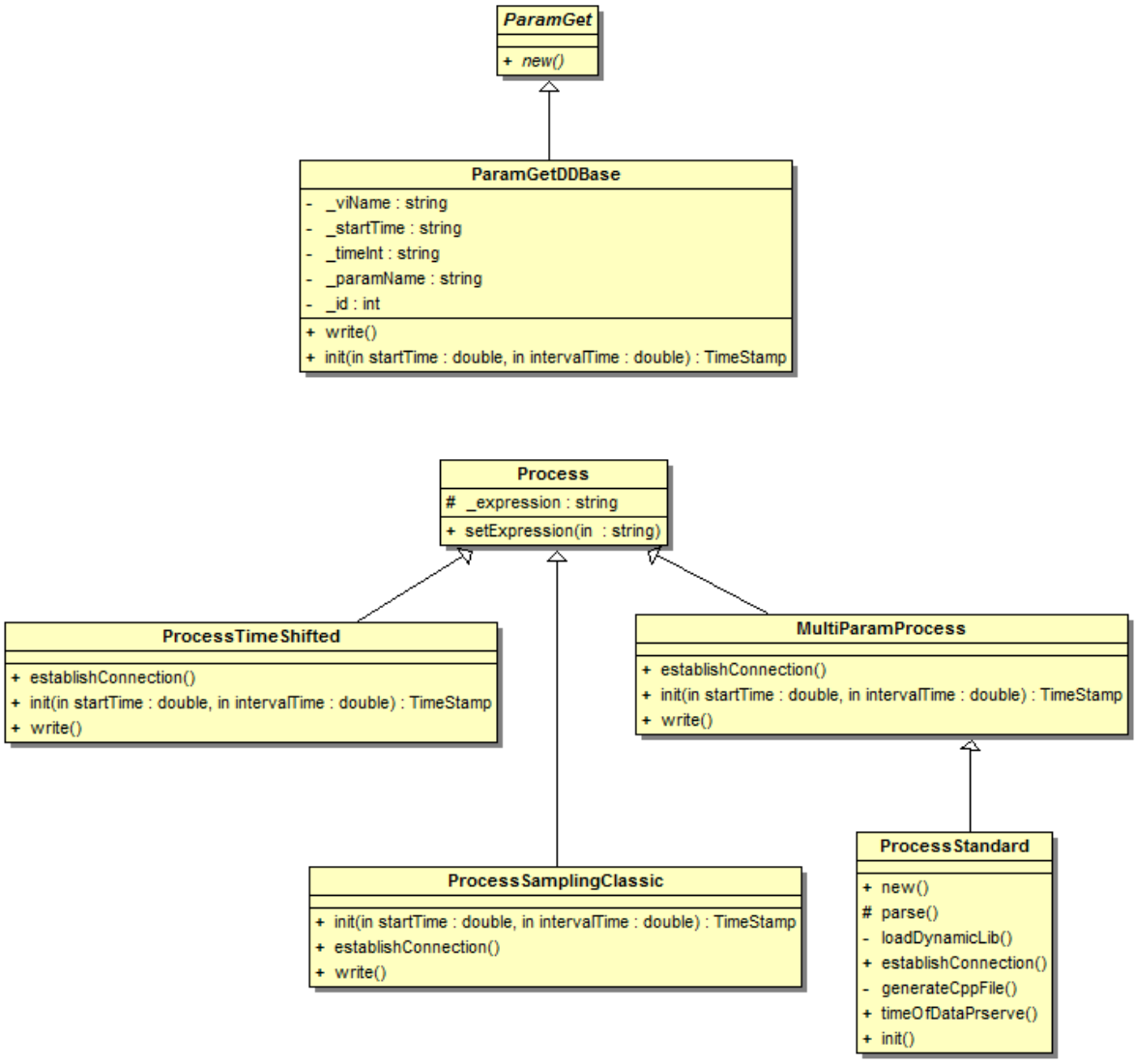


Figure 4 Principales implémentations de Process et de ParamGet

Les principales classes abstraites du composant sont `ParamGet` (accès aux données), `Process` (calculs) et `ParamOutput` (génération d'un résultat)

`ParamOutput` et ses implémentations sont décrits dans le paragraphe précédent ainsi qu'au §3.1.2.8 : « Composant DownLoad »..

`ParamGet` a comme principale implémentation : `ParamGetDDBase` qui est chargée de l'aquisition des données dans DDServer. Sa description est données au §3.1.3.1: « Composant DD server Interface ».

`Process` est dédié à avoir de nombreuses implémentations comme, `ProcessTilmeShifted`, `ProcessDeriv`, …

Une implémentation est nécessaire dès qu'il y a besoin de créer une fonction de calcul qui modifie le temps ou le nombre de données.

La principale implémentation est `ProcessStandard`. Il a pour responsabilite d'évaluer la formule présente dans la balise process du fichier xml parameter. Sa description est données au §3.1.2.5 :« Composant EvalFormule ».

3.1.2.2 Composant Plugin

L'application accepte actuellement l'ajout de plugin sous 2 formes :

- Une librairie « so »
- Les fonctions custom utilisables par la classe ProcessStandard

3.1.2.2.1 Librairie « *.so »

Ces librairies « *.so » doivent être placées sous le répertoire indiqué par le fichier propriété de l'application « app.plugin ».

Et elles devront contenir les deux fonctions C suivantes :

```
#include "ParameterManager.hh"
#include "PluginManager.hh"
#include "Operation.hh"

using namespace AMDA::Parameters;

namespace AMDA {
namespace Parameters {



/**
   Retrieve the Plugin version we're going to expect
*/
extern "C" const char* getPluginVersion()
{
      return "1.0.0";
}

/**
   Tells us to register our functionality to an engine kernel
*/
```

```
extern "C" void registerPlugin(AMDA::Plugins::PluginManager & pm)
{
      ParameterManager::getInstance()-
>getOperationsList()["Magnitude"]=Base::OperationSPtr(new Magnitude);}
```

Si ces fonctions ne sont pas présentes, la librairie n'est pas chargée et les éventuelles erreurs apparaissent en warning dans la journalisation. Le programme ne s'arrête pas si un plugin n'est pas chargé mais les effets de la non disponibilité des fonctionnalités peuvent amener à l'arrêt du programme. Dans ce cas un log d'erreur apparait dans la journalisation.

La fonction « registerPlugin » est la fonction clé. Elle sert à insérer de nouvelles fonctionnalités dans le programme AMDA-Kernel. Dans l'exemple ci-dessus on a mis à disposition la fonctionnalité Magnitude qui peut être utilisée dans la balise « process » du fichier XML de description d'un paramètre.

Cela est possible via le point d'entrée principal qui est le singleton ServicesServer. A partir de celui-ci on peut atteindre les fabriques de Process, de ParamGet, de ParamOutput, XMLRequestConfigurator et de l'XMLParameterConfigurator. Cf. les composants DDServerInterface et ParamOutput qui sont codés sous forme de plugin.

La Classe **ServicesServer** permet de mémoriser tous les plugins et se définit comme offrant des services au reste de l'application. **ServicesServer** peut offrir des implémentations du service de ParamGet, ParamOutput ou de Process

3.1.2.2.2 Les fonctions mathématiques utilisables par la classe ProcessStandard

Toute fonction détectée pendant le « parse » de l'expression d'un processStandart déclenchera la recherche d'un fichier d'interface de nom nomFct.hh dans un répertoire du nom de la fonction (nomFct). Si ce fichier est trouvé, il sera ajouté au fichier généré implémentant l'interface `Operation`.Les fichiers de librairie dynamique .so seront ajoutés à l'édition des liens de la complilation du fichier généré. Il est ainsi possible d'ajouter des fonctions mathématiques dynamiquement.

Cf annex 6.2

3.1.2.3 Composant Condition

AD

3.1.2.4 Composant Plot

AD

3.1.2.5 Composant EvalFormule

Ce composant créé une méthode compilée à partir d'une formule écrite dans un fichier XML balise « <process> ».

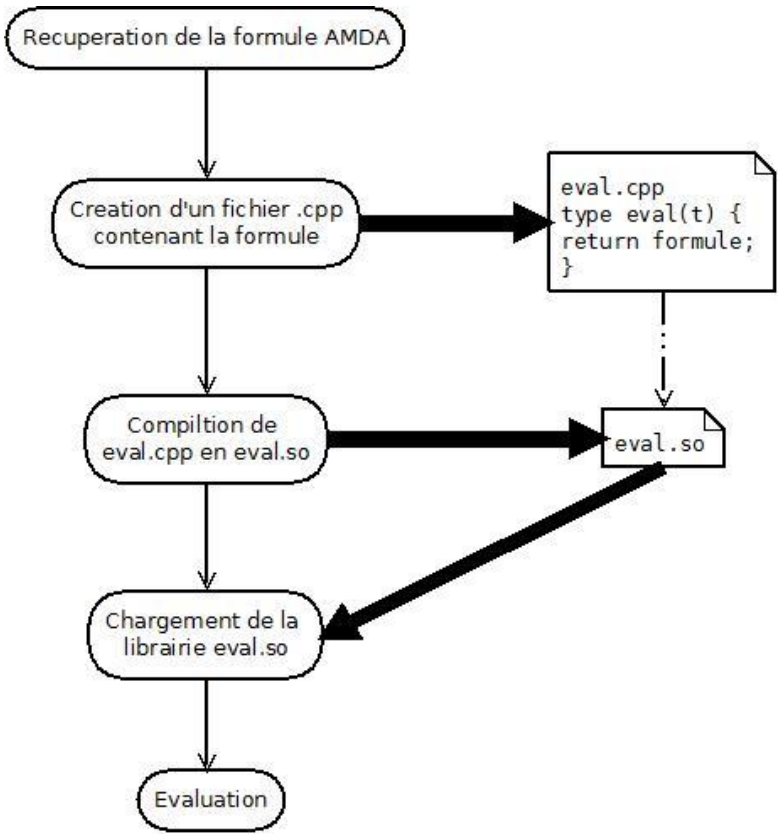
Ce composant est principalement porté par la classe `ProcessStandard`.

Cette formule est transformée du format AMDA en une chaîne de caractères au format C++.

- ram_presure($density,$speed) => ram_pressure(_density.get(i),_speed.get(i)) ;

Ensuite on génère un fichier temporaire C++ contenant une fonction de calcul dont le code est la chaîne générée. Ce fichier est ensuite compilé sous forme d'une librairie dynamique qui sera chargée dynamiquement par le serveur. Cette fonction pourra alors être exécutée pour chaque pas de temps.

Dans le but d'optimiser, la compilation n'aura lieu que si nécessaire. Pour déterminer si la compilation est nécessaire, le code se base sur le `TimeStamp` retourné par les méthodes `init` de `DataWriter`. Cette méthode doit retourner l'heure du fichier le plus vieux pouvant avoir un impact sur le type du `ParamData` output de `DataWriter`. La génération du .cc et sa compilation n'aura lieu que si cette date et celles des interfaces (.hh) utilisées est plus vieille que la date du « .so » à générer.



Ce composant est entièrement contenue dans la class `Process`.

Le fichier généré implémente une fonction C de prototype :

```
extern "C" void processInit(AMDA::Parameters::ProcessStandard *pProcess);
```

Cette fonction installe au ProcessStandart l'implémentation de l'opération avec un paramOutput de type réel.

Voici un exemple de fichier généré :

```
/*  File generated by AMDA */

#include "Operation.hh"
```

```
#include "ServicesServer.hh"
#include "Parameter.hh"
#include "ParamData.hh"
#include "DataTypeMath.hh"
#include "ProcessStandard.hh"

#include "ram_pressure.hh"

typedef float  _T_DENSITY_Element_Type;
typedef AMDA::Parameters::ParamDataSpec<float>  _T_DENSITY;
typedef float  _T_SPEED_Element_Type;
typedef AMDA::Parameters::ParamDataSpec<float>  _T_SPEED;
extern "C" void processInit(AMDA::Parameters::ProcessStandard *pProcess);

template <typename TParamData>
class _Operationram_pressure : public AMDA::Parameters::Operation {
public:

_Operationram_pressure(AMDA::Parameters::ProcessStandard &pProcess) :
AMDA::Parameters::Operation(pProcess), _processStandard(pProcess), _paramOutput(new
TParamData())
{
      _paramDataOutput = _paramOutput;
      density = dynamic_cast<_T_DENSITY*>( pProcess.getParameterList()["density"]-
>getParamData(&pProcess).get());
      speed = dynamic_cast<_T_SPEED*>( pProcess.getParameterList()["speed"]-
>getParamData(&pProcess).get());
}

virtual ~_Operationram_pressure() {}

void  write(AMDA::Parameters::ParamDataIndexInfo &pParamDataIndexInfo) {
      for (unsigned int index = pParamDataIndexInfo._startIndex; index <
pParamDataIndexInfo._startIndex + pParamDataIndexInfo._nbDataToProcess; ++index) {
```

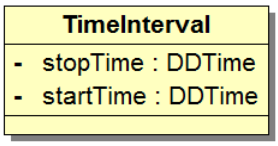
```
            _paramOutput->pushTime(_processStandard.getParameterList().begin()-
>second->getParamData(&_process)->getTime(index));
            _paramOutput->push(ram_pressure(density->get(index),speed->get(index)));
      }
}

AMDA::Parameters::ProcessStandard&  _processStandard;
TParamData *_paramOutput;
AMDA::Parameters::ParamDataIndexInfo _paramDataIndexInfo;
 _T_DENSITY *density;
 _T_SPEED *speed;

};
void processInit(AMDA::Parameters::ProcessStandard *pProcess){
      pProcess->setOperation(new
_Operationram_pressure<decltype(AMDA::Parameters::generate(ram_pressure(_T_DENSITY_El
ement_Type(),_T_SPEED_Element_Type())))>(*pProcess));
      pProcess->getParamData().reset( pProcess->getOperation()->getParamOutput());
}
```

3.1.2.6 Composant TimeTable

Ce composant a pour but de gérer les Time Tables :

- mémorisation via la couche de persistance
- fusion de `TimeTable`

Dans un premier temps, ce composant est représenté par une simple classe nommée `TimeInterval`



AD

3.1.2.7 Composant UpLoad-Data

AD

3.1.2.8 Composant DownLoad

Ce composant a pour but

- de mettre à disposition des fichiers de données sous forme ASCII.

Ce composant s'appuie sur l'implémentation de `ParamOutput` : `ParamOutputFileASCII`.

Il implémente les méthodes etablishConnection() et init() pour initialiser le processus. Puis via l'implémentation de la méthode apply(), il demande de nouvelles données qu'il écrit dans un fichier ASCII.

Comme indiqué au paragraphe 3.1.2.1.6, `ParamOutputFileASCII` implémente le `VisitorOfParamData` pour différentier les types de données et faire une sortie adéquate.

`ParamOutputFileASCII` est capable via l'attribut _typeFormatTime d'écrire le temps au format DD, ISO ou Double.

Le format de sortie est de la forme :

En entête les valeurs des informations de calibration : une information par ligne, chaque valeur séparée par des espaces.

Puis pour chaque date, sur une ligne, la date au format demandé et les valeurs du paramètre séparé par des espaces.

3.1.3 Couche Persistance

3.1.3.1 Composant DD server Interface

Ce composant a pour but d'unifier l'accès à DD server.

Il est piloté par l'implémentation de ParamGet : ParamGetDDBase.

Il utilise une librairie (DDClient) qui communique via des sockets TCP/IP. Quatre principales étapes sont nécessaires pour récupérer des données:

- OPENINSREQ : Ouvre un dataset et retourne un identifiant unique.
- CLOSEINSREQ : ferme un dataset (identifiant en paramètre).
- TIMESETREQ: Positionne un pointeur sur le Start Time (Start Time et identifiant en paramètre)
- DATAGETREQ: A partir du start Time positionné, déplace le pointeur durant la requête par intervalle de temps (indiqué en paramètre). Le retour se fait sous forme d'un flux les données dans la socket. En en-tête est indiqué le nombre d'enregistrements et la taille des paramètres.

Dans un souci d'optimisation DDServer permet de récupérer des paramètres d'un même VirtualInstrument en une seule requête. Un mécanisme a donc été mis en place pour utiliser cette méthode de DDClient getMultiParam.

L'idée est qu'une classe `VirtualInstrumentManager` assure l'unicité des objets `VirtualInstrument`. Ceci sont responsables de fournir au `ParamGetDDBase` un flow de data en fonction du nom de paramètre et de l'intervalle de temps demandé. `VirtualInstrument` assure qu'un seul objet `VirtualInstrumentInterval` fournit le `ParamFlow` pour un intervalle de temps donnée et procure toutes les données commune à un `VirtualInstrument` (MinSampling, MaxSampling, FillValue, GlobalStartTime). Ci-dessous quelques diagrammes UML décrivant ce composant.

Le diagramme d'objet et le diagramme de séquence décrivent une Requête A dans laquelle on demande les sorties ASCII des paramètres « imf », « speed » et « density ».

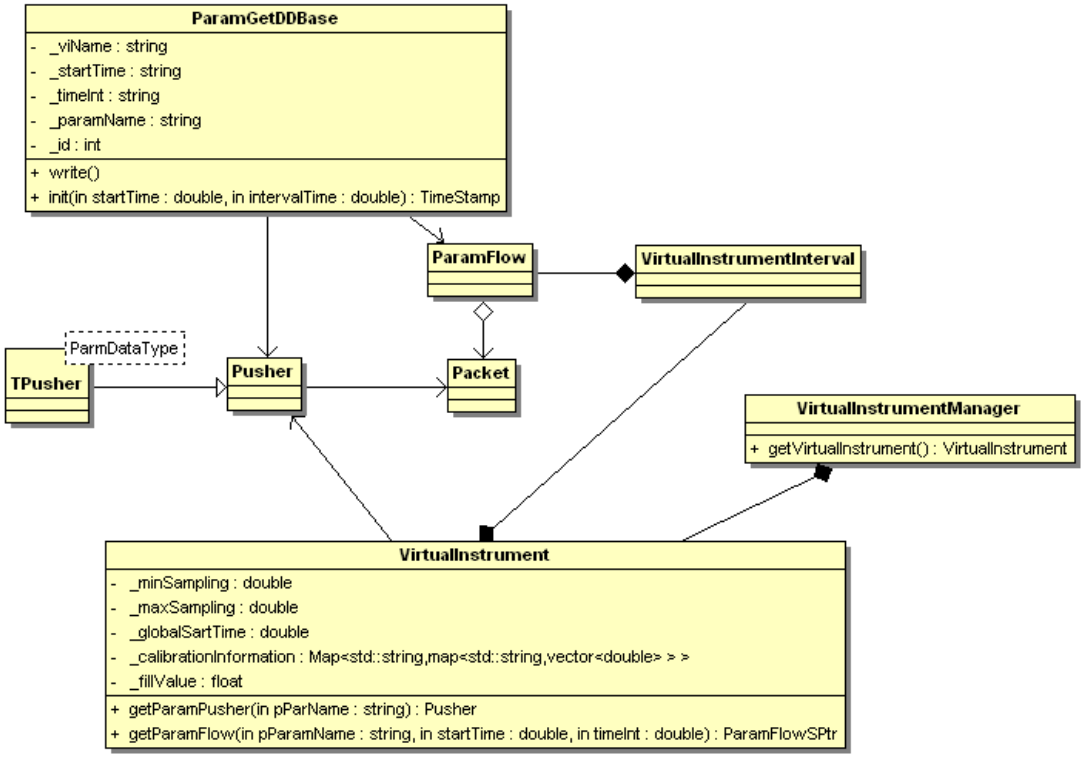


Figure 5 Diagramme de classe UML : Package DDServerInterface

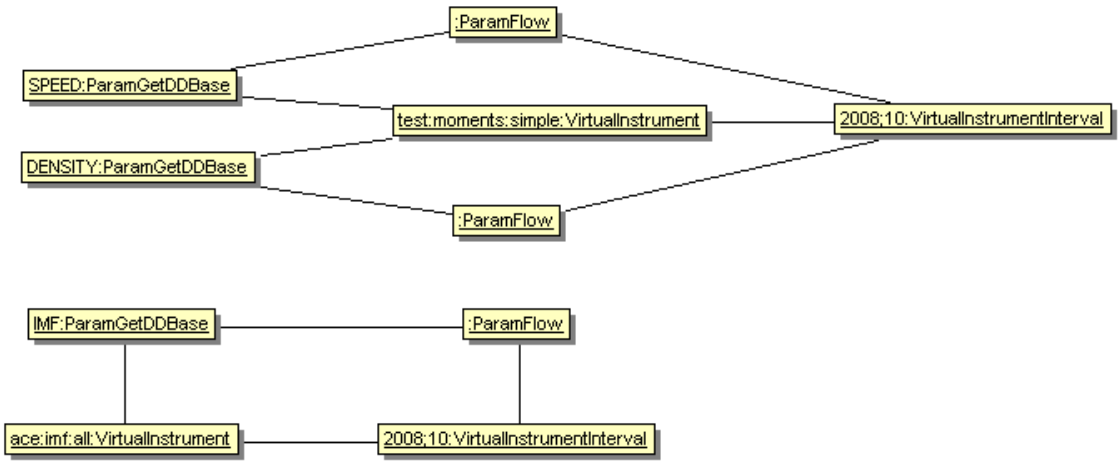


Figure 6 Diagramme Objet UML : Requête A objets instanciés

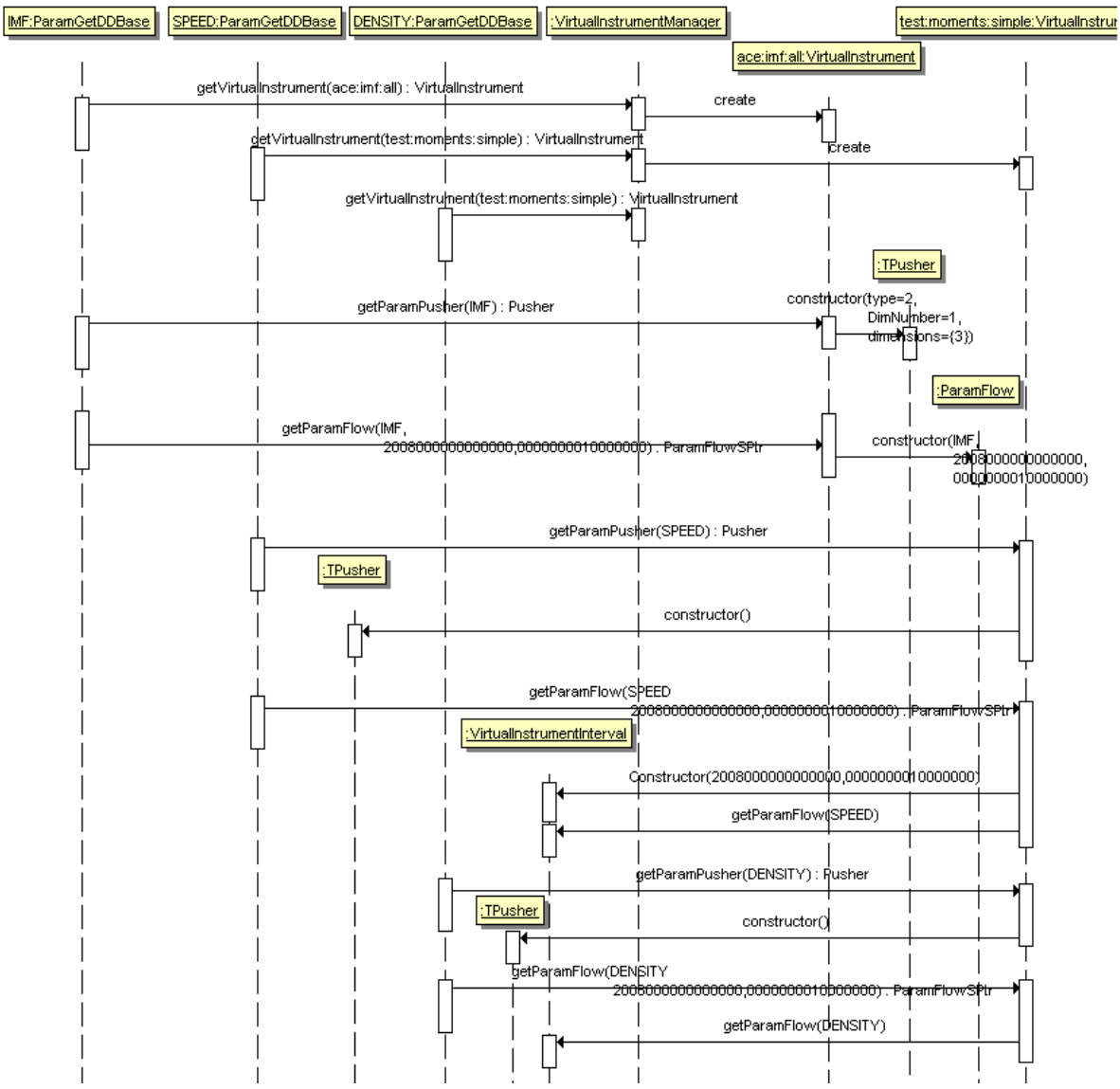


Figure 7 Diagramme de Séquence UML : Requête A phase d'init

3.1.3.2 Composant My Data

AD

3.1.3.3 Composant ReadData

AD

3.1.3.4 Composant WorkSpace

AD

3.1.4 Composant transverse de journalisation

3.1.4.1 Stratégie de journalisation

La stratégie de journalisation est basée sur 5 niveaux :

- FATAL : trace les erreurs qui obligent un arrêt de l'exécutable. A utiliser que dans les cas extrêmes, sur des problèmes system ne permettant pas de remettre le programme dans un état stable.
- ERROR : trace les erreurs fonctionnels et systèmes qui engendrent un retour d'erreur à l'utilisateur. Le code Erreur ainsi que la raison indiquée en anglais doivent être affichés.
- WARN : trace les warnings fonctionnels. Un warning étant une erreur qui dégrade la fonction mais n'empêche pas d'obtenir un résultat.
- INFO : est utilisé pour tracer les accès aux interfaces externes. Par exemple tracer la requête, tracer les accès à DD_server.
- DEBUG : est utilisé à discrétion du développeur.

La journalisation indique la date, la couche logicielle et composante ayant effectuée la trace et un message explicite en anglais.

3.1.4.2 Log4cxx

La journalisation est basée sur Log4cxx. Les niveaux d'erreurs (Critical, Error, Warning, Info, Debug) sont mappés sur leurs équivalents en log4cxx.

La hiérarchie de trace proposée par Log4cxx est mappée sur les notions de couche logicielle / composant.

3.2 DESCRIPTION DÉTAILLÉE DES DONNÉES

Sans objet.

3.3 DESCRIPTION DES MESSAGES D'ERREUR

L'exécutable AMDA retourne 0 s'il n'y a pas d'erreur, comme le recommande UNIX. En cas d'erreur il retourne le code décrit dans [A2].

4 DOCUMENTS APPLICABLES ET DE REFERENCE (A/R)

A/R	Référence	Titre
A1	CDPP-AR-32500-382-SI	Dossier d'architecture du noyau d'AMAD-NG
A2	CDPP-IF-32500-438-SI	Dossier de contrôle des interfaces du noyau AMDA-NG

5 GLOSSAIRE ET ABREVIATIONS

5.1 GLOSSAIRE

Terme	Définition

5.2 ABRÉVIATIONS

Abréviation	Nom détaillé

6 ANNEXES

6.1 ANNEXE 1 : DOCUMENTATION GÉNÉRÉE PAR DOXYGEN

Doxygen.pdf

6.2 ANNEXE 2: CRÉATION D'UNE NOUVELLE FONCTION MATHÉMATIQUE

Note : Cette annexe est présente en ligne sur le wiki FitNesse.

6.2.1 Objectif

L'objectif de cette page est de créer une nouvelle fonction mathématique. Cela permettra de créer un fichier xml décrivant un paramètre contenant une balise process du type **newFunction($dst)**

6.2.2 Procédure

Il faut créer au moins un fichier .h d'interface.

De plus un fichier .so (librairie dynamique) contenant le code compilé de la fonction pourra être créé *cas classique*.
Ces 2 fichiers doivent être positionnés dans le répertoire {app.plugin}/newFunction

6.2.3 Exemple

6.2.3.1 Creation la fonction newFunction

Cette fonction prendra un tableau 1 D en paramètre et retournera un flottant.

1. **Créer le fichier .h**

Le fichier devra se nommer **newFunction.h**

Son contenue sera du type:

```
#ifndef NEWFUNCTION_HH_ //is required
#define NEWFUNCTION_HH_

#include <vector> //for the AMDA type tab 1 D

std::vector<float>newFunction(const float& el); //is the prototype at the new
function

#endif /* NEWFUNCTION_HH_ */
```

2. **Créer un fichier .cc**

Ce fichier devra implémenter l'interface newFunction.h.

Par exemple newFunction.cc, son contenu sera du type:

```
#include "newFunction.hh"

//add all necessary include
#include <math.h>

std::vector<float>newFunction(const float& el)
{
        std::vector<float> r;
        r.push_back(el);
        r.push_back(el);
        r.push_back(el);
        return r;
}
```

3. Compiler le fichier .cc en un fichier .so

```
c++   -DMagnitude_EXPORTS -g  -std=c++0x -fPIC -Wall -ggdb -DLINUX -Dlinux -
D_REENTRANT -malign-double -pthread -fPIC   -o newFunction.o -c newFunction.cc
```

```
c++  -fPIC -g  -std=c++0x -fPIC -Wall -ggdb -DLINUX -Dlinux -D_REENTRANT -malign-
double -pthread  -shared -Wl,-soname,libnewFunction.so -o libnewFunction.so
newFunction.o
```

4. Créer le plugin

Soit {app.plugin} le répertoire de plugin indiqué dans le fichier de properties *app.properties* d'AMDA. Ce répertoire est indiqué par la propriété app.plugin

Créer dans le répertoire *{app.plugin}* un répertoire du nom de la nouvelle fonction. i.e. *newFunction*. Puis dans ce nouveau répertoire: **{app.plugin}/newFunction** copier le fichier **newFunction.h** et le fichier **newFunction.so**

Si votre plugin dépend de librairie dynamique supplémentaire, vous pouvez copier (ou créer des liens symboliques) ces librairies dans le répertoire *{app.plugin}/newFunction*. Ces librairies seront prises en compte.

6.3 ANNEXE 3: CRÉATION D'UN NOUVEAU TYPE DE DONNÉES

Note : Cette annexe est présente en ligne sur le wiki FitNesse.

6.3.1 Objectif

L'objectif est de créer un nouveau type de données. La classe ParamData est l'interface de stockage des données et la classe template ParamDataSpec en est l'implémentation pour un type donné.

Aujourd'hui, il existe les ParamData de type suivant (fichier ParamData.hh):

```
typedef ParamDataSpec<int> ParamDataScalaireInt;

typedef ParamDataSpec<char> ParamDataScalaireChar;

typedef ParamDataSpec<double> ParamDataScalaireDouble;

typedef ParamDataSpec<float> ParamDataScalaireFloat;

typedef ParamDataSpec<long double> ParamDataScalaireLongDouble;

typedef ParamDataSpec<short> ParamDataScalaireShort;

typedef ParamDataSpec<std::vector<int> > ParamDataTab1DInt;

typedef ParamDataSpec<std::vector<char> > ParamDataTab1DChar;

typedef ParamDataSpec<std::vector<short> > ParamDataTab1DShort;

typedef ParamDataSpec<std::vector<float> > ParamDataTab1DFloat;

typedef ParamDataSpec<std::vector<double> > ParamDataTab1DDouble;

typedef ParamDataSpec<std::vector<long double> > ParamDataTab1DLongDouble;
```

6.3.2 Réalisation

Pour ajouter un nouveau type « LogicalData », il faut :

- Déclarer ce type ou vous le souhaitez,
- Ajouter les implémentations dans ParamData.hh :
  - typedef ParamDataSpec<LogicalData> ParamDataScalaireLogicalData ;
  - typedef ParamDataSpec<std::vector<LogicalData> > ParamDataTab1LogicalData ;
- déclarer les opérateurs si nécessaire dans ParamData.hh
- ajouter la méthode virtuelle pure de visite du nouveau type dans VisitorOfParamData.hh
- corriger à la compilation les erreurs remontées parce que les visiteurs n'implémentent pas cette nouvelle méthode. (Exemples : ParamOutputASCIIFile, TimeShiftedCreator, …)

6.4 ANNEXE 4: CRÉATION D'UN NOUVEAU TYPE D'OUTPUT

Note : Cette annexe est présente en ligne sur le wiki FitNesse.

6.4.1 Objectif

L'objectif de cette page est de créer un nouveau type de sortie. Cela permettra de gérer un fichier xml décrivant une requête contenant une balise output du type

```
<outputs>
  <OutputName>
    <param id="density_boxcar" />
    ...
  </OutputName>
</outputs>
```

6.4.2 Procédure

- Il faut créer un plugin cf. AMDA plugin enregistrant le nouveau **Output** *
- Il faut créer au moins une classe dérivant de **AMDA::Parameters::ParamOutput**.
- Il faut créer une classe dérivant de **AMDA::XMLConfigurator::NodeGrpCfg**.
- Il faut mettre à jour les schémas XSD permettant de valider les fichiers XML de requête

6.4.3 Exemple:

6.4.3.1 Création d'une nouvelle sortie: intervalTrue

Cette sortie écrit toutes les valeurs d'un paramètre sur une ligne. Le fichier xml de requête contiendra la balise output suivante (pamarmName)

```
<outputs>
  <intervalTrue>
    <param id="pamarmName" />
  </intervalTrue>
</outputs>
```

6.4.3.1.1 Suivre la procédure de création de plugin

cf. AMDA plugin

La fonction registerPlugin doit contenir les lignes suivantes:

```
extern "C" void registerPlugin(AMDA::Plugins::PluginManager & pm) {
      AMDA::XMLRequest::XMLRequestParser* lXMLRequestParser =
ServicesServer::getInstance()->getService<AMDA::XMLRequest::XMLRequestParser*>();
      if (lXMLRequestParser) {
            XMLRequestParser-
>addNodeParser("request/outputs/IntervalTrue",AMDA::XMLConfigurator::NodeCfgSPtr(new
OutputIntervalTrueNode));
        }
}
```

6.4.3.1.2 Créer la classe OutputIntervalTrueNode

Il faut dériver de **AMDA::XMLConfigurator::NodeGrpCfg** et implémenter la méthode:

- void process() : permet de lire le noeud intervalTrue du fichier de requête.

Ce nœud comporte seulement un nœud param (qui est le paramètre qui sera affiché), il peut en comporter d'autres si besoin.

```
OutputIntervalTrueNode::OutputIntervalTrueNode() : NodeGrpCfg() {
      getChildList()["param"]=NodeCfgSPtr(new
ParamNode<ParamOutputIntervalTrueFile>());
}

void OutputIntervalTrueNode::proceed(xmlNodePtr pNode,
      const AMDA::Parameters::CfgContext& pContext) {
      LOG4CXX_DEBUG(gLogger,
            "OutputIntervalTrueNode::proceed: '" << pNode->name << "' node")
      ParameterManager* lParameterManager = pContext.get<ParameterManager*>();
      CfgContext lContext;
      ParamOutput* lParamOutputIntervalTrueFile = new ParamOutputIntervalTrueFile(
                  *lParameterManager);
      ParamOutputSPtr  lParamOutput(lParamOutputIntervalTrueFile);
```

```
      lContext.push<ParamOutputIntervalTrueFile*>(dynamic_cast<ParamOutputIntervalTru
eFile *>(lParamOutputIntervalTrueFile));
      NodeGrpCfg::proceed(pNode, lContext);
      lParameterManager->getParamOutputList().push_back(lParamOutput);
}
```

6.4.3.1.3 Créer la classe ParamOutputIntervalTrueFile

Il faut dériver de **AMDA::Parameters::ParamOutput** et implémenter au minimum les méthodes:

- **void apply()** : permet de générer la sortie, c'est le cœur du métier d'output, get data avec _parameter->getAsync()

et utilise le design pattern visiteur pour lire les valeur du paramètre.

```
void ParamOutputIntervalTrueFile::apply() {

      _paramDataIndexInfo = _parameter->getAsync(this).get();
      initOutput(); //create file
      _printIntervalTrueVisitor->setParamDataIndexInfo(_paramDataIndexInfo);

      while ( _paramDataIndexInfo._nbDataToProcess > 0) {
            try {
                  _parameter->getParamData(this)->accept(*_printIntervalTrueVisitor);
//visitor pattern
                  _paramDataIndexInfo = _parameter->getAsync(this).get();
                  _printIntervalTrueVisitor-
>setParamDataIndexInfo(_paramDataIndexInfo);
            } catch(AMDA::AMDA_exception & e) {
                  e << AMDA::errno_code(AMDA_PARAM_OUTPUT_ERR);
                  throw;
            }
      }
      endOutput(); //close file
}
```

- **void init()** : permet d'initialiser les paramètres à traiter

```
void ParamOutputIntervalTrueFile::init() {
	getParameter()->init(this,getStartTime(),getTimeInt());
}
```

- **void DataClient::establishConnection()** : permet de calculer des informations static (comme « parser » la formule de process) et doit ouvrir les connections avec son _parameterInput

```
void ParamOutputIntervalTrueFile::establishConnection() {
	_parameter = _parameterManager.getSampledParameter(_paramName, _samplingMode, _samplingValue, _gabsThreshold).get();
	if(_parameter == NULL)  {
		LOG4CXX_ERROR(_logger,"ParamOutput::init parameter : \""<< _paramName <<"\" Not Exist" );
		BOOST_THROW_EXCEPTION( ParamOutput_exception());
	}
	getParameter()->openConnection(this);
}
```

- **double DataClient::timeOfDataPreserve()** : doit retourner l'intervalle temps nécessaire pour assurer l'intégrité des calculs, ici la taille du boxcar: 1200

6.4.3.1.4 Mettre à jour le XSD

- Modifier le fichier request/all.xsd:

Y ajouter une ligne du type:

- Créer un fichier intervalTrue.xsd (voir nom du fichier référencer dans la ligne ajouté au fichier all.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:include schemaLocation="request.xsd" />
```

```
      <xs:element name="intervalTrue" substitutionGroup="OutputElement" >
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="param" minOccurs="1" maxOccurs="1">
                              <xs:complexType>
                                    <xs:sequence>
                                          <xs:element name="index" type="xs:integer"
minOccurs="0" maxOccurs="unbounded"/>
                                          <xs:element name="calibration_info"
type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
                                    </xs:sequence>
                                    <xs:attribute name="id" type="xs:string"
use="required"/>
                              </xs:complexType>
                        </xs:element>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
</xs:schema>
```

6.5 ANNEXE 5: CRÉATION D'UN NOUVEAU TYPE DE PARAMGET

Note : Cette annexe est présente en ligne sur le wiki FitNesse.

6.5.1 Objectif

L'objectif de cette page est de créer un nouveau type de ParamGet. Cela permettra de gérer un fichier xml décrivant un paramètre contenant une balise get du type

```
  <get>
   <newGetter name="density"/>
  </get>
```

l'attribut name du noeud fils de <get> est obligatoire

6.5.2 Procédure

- Il faut créer un plugin cf. AMDA plugin enregistrant le nouveau **ParamGet**
- Il faut créer au moins une classe dérivant de **AMDA::Parameters::ParamGet_CRTP**.
- Il faut créer une classe dérivant de **AMDA::XMLConfigurator::NodeGrpCfg**.
- Et il faut créer et/ou mettre à jour les schémas XSD permettant de valider les fichiers XML de paramètres

6.5.3 Exemple

*6.5.3.1 Création d'un nouveau ParamGet **lectFile***

Il devra lire les données dans un fichier ascii.

6.5.3.1.1 Suivre la procédure de création de plugin

cf. AMDA plugin

La fonction registerPlugin doit contenir les lignes suivantes:

```
      ParamGetFactory fact = boost::factory<ParamGetLectFile*>();
      ServicesServer *servicesServer = ServicesServer::getInstance();
      servicesServer->addParamGetFactory("LECT_FILE", fact);
      XMLParameterConfigurator* lXMLConf =
dynamic_cast<XMLParameterConfigurator*>(servicesServer->getConfigurator().get());
```

```
      lXMLConf-
>addNodeParser("param/get/lectFile",AMDA::XMLConfigurator::NodeCfgSPtr(new
GetLectFileNode));
```

6.5.3.1.2 Créer la classe GetLectFileNode

Il faut dériver de **AMDA::XMLConfigurator::NodeGrpCfg** et implémenter la méthode:

- void process() : permet de lire le noeud lectFile du fichier paramètre,

ce noeud peut comporter d'autres noeuds si besoin.

```
void GetLectFileNode::proceed(xmlNodePtr pNode,const AMDA::Parameters::CfgContext&
pContext) {
      const char *lSouceParamGet = "DDBASE";
      LOG4CXX_DEBUG(gLogger,  "NodeGetDDBase::proceed: '" << pNode->name << "' node")

      ServicesServer* lServicesServer = pContext.get<ServicesServer*>();
      Parameter* lParentParameter = pContext.get<Parameter*>();
      ParameterManager* lParameterManager =  pContext.get<ParameterManager*>();
      xmlChar* lParamName = NULL;

      try {

            if (!(lParamName = xmlGetProp(pNode, (const xmlChar *) PARAMNAME))) {
                  ERROR_EXCEPTION(
                              ERROR_MANDATORY_ATTRIBUTE_MISSING << pNode->name << "@"
<< PARAMNAME)
            }

            ParameterSPtr lParameter;
            if ( lParameterManager->addParameter(lParentParameter,std::string((const
char*) lParamName),lParameter)) {
```

```
                  ParamGetLectFileSPtr
lParamGet(dynamic_cast<ParamGetLectFile*>(lServicesServer->getParamGet("LECT_FILE",
*lParameter)));
                  DataWriterSPtr lDataWriter( lParamGet);
                  lParamGet->setParName((const char*) lParamName);
                  lParameter->setDataWriter(lDataWriter);
                  AMDA::Parameters::CfgContext lContext(pContext);
                  lContext.push<ParamGetDDBase*>(lParamGet.get());
                  NodeGrpCfg::proceed(pNode, lContext);
            }
      } catch (...) {
            if (lParamName) {
                  xmlFree(lParamName);
            }
            throw;
      }
      if (lParamName) {
            xmlFree(lParamName);
      }
}
```

6.5.3.1.3 Créer la classe ParamGetLectFile

Il faut dériver de **AMDA::Parameters::ParamGet_CRTP** et implémenter les méthodes:

- void DataWriter::init() :permet de calculer des informations static et créer les ParamDatad'output.

```
void ParamGetLectFile::init() {


      _paramData = new ParamDataSpec<float>();
}
```

- unsigned int DataWriter::write() :doit écrire les données dans le ParamData d'output et retourner le nombre de données qu'il a écrit.

```
unsigned int ParamGetLectFile::write() {
  vector<string> colonne;
```

```
  string ligne;
  ofstream fichier("data.txt" );
  // Tant que j'ai des lignes à lire
  double time = 0;
  while (getline (fichier, ligne))
  {
    float val = atof(ligne.str());
    dynamic_cast<ParamDataSpec<float>(_paramData)->getDataList().push_back(val);
     _paramData->getTimeList().push_back(time++);

  }
  return 0;
}
```

6.5.3.1.4 Mettre à jour le XSD

- Modifier le fichier request/all.xsd:

Y ajouter une ligne du type: * Créer un fichier newGetter .xsd (voir nom du fichier référencé dans la ligne ajoutée au fichier all.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:include schemaLocation="newGetter.xsd" />
    <xs:element name="newGetter" substitutionGroup="Getter" type="GetterType"/>
</xs:schema>
```