# Maintenance of the supervisor of *Code_Aster*

**Summary:**

This document is with the use of the developers of the supervisory core of Aster. It clarifies the interpretation of the data file of the user (construction of the command set) and the sequence of the executions.

**Nota bene important:**

This document was produced for version 6 of Code_Aster and was not updated since. It is preserved here for archive.

## Contents

## Contents

# 1 Introduction

**Digital study**

To carry out a digital study with *Code_Aster*, the end-user starts the features of the code and provides the necessary information to the execution of these features.

A functionality is selected via an order. An order is initially a name i.e. a character string or an identifier **external** (known of the user) of the functionality. It also incorporates attributes and in first place the identifier **intern** functionality (exploitable by code FORTRAN).

The necessary information with the launching and the execution of the order otherwise-known as the parameters of the digital processing, are data introduced by means of keywords the identifier. For a given order, a certain number of parameters must be defined.

The description of the orders and the keywords is carried out by the developers of *Code_Aster*. in a file called “catalogues orders”.

The end-user uses the code via a file called “command file”. He provides to it orders whose composition must be compatible with their description in the catalogue of the orders. The orders are numerous but the number of associated data is much more important with possible combinations themselves very many.

Finally the computer code stores information (orders and keywords) in a structure of data interns called “command set”.

**The role of the supervisor**

The supervisor is the part of *Code_Aster* who manages the command set. In particular:

1) it imports the catalogue of the orders in the memory Python,
2) it charges the orders and their keywords in the memory Python,
3) it carries out the treatment orders by order,
4) it provides - at the request of FORTRAN – the value of the parameters to the features of *Code_Aster*. stored in the memory Python. For that the supervisor proposes API C.

**The Eficas graphic interface**

It is possible to define “manually” its command file, for example by means of a text editor. However, the user nonfamiliar with syntax of the orders which it handles but also the language python will be able to use Eficas.

The Eficas graphic interface is intended to the end-user of *Code_Aster*. It enables him to build valid orders with statically such validated associated keywords, then to generate a command file bound for the code. The user has the assurance which the file produced by Eficas has a correct syntax.

**The core**

Sources common to the Eficas interface-graph and the supervisor of *Code_Aster* are organized (identified and gathered) in a specific space of development called the “Core”. The comprehension of these sources is obviously a requirement to undertake the maintenance of the two tools which use them.

**Main actors**

The principal types of actors evolving in the environment of Eficas and the supervisor are listed Ci - below.

1) The end-user: he rather knows the physical part of the problem to be solved and he has as a task to introduce valid data into the code, of launching calculations and stripping the results; he can use the Eficas graphic interface for that.
2) The developer of features in *Code_Aster* : it practises primarily programming FORTRAN of the digital algorithms and, in the case of an macro-order, the drafting of script corresponding Python; it enriches and/or modifies the catalogue of order and it uses API C of the supervisor to recover in his code FORTRAN, the values provided by the end-user.
3) The developer of the supervisor in *Code_Aster* : it writes scripts of reading and interpretation of the catalogue of orders and the command set; it also deals with the module python `aster` writing out of C (`astermodule.c`) who serves as API to the supervisor.
4) The person in charge of the maintenance of the code, the environment of use and management of configuration: he centralizes the sources. Besides code FORTRAN, these sources understand those of the catalogue (Python) orders, scripts (Python) of the supervisor and those of the module `aster`.
5) The developer of the Eficas graphic interface develops a man-machine interface. Its role is to conceive and program dialog boxes as well as sequences of event allowing the taking into account of the requests of the end-user and the checking – at every moment - validity of the command set in the course of construction.

In this document, one rather focusses on the principles of the design of the core that on the detail of scripts which one cannot save the examination.

**Covered issues**

The first chapter proposes a summary definition of the terms used abundantly in the continuation of the document.

The chapter 2 present few conventions and the organization of the source files in repertoires: packages. It also describes the environment necessary for the development and the operation of the supervisor of *Code_Aster* and of the Eficas graphic interface.

In chapter 3, some recalls are carried out on the language of script Python. They must direct the future person in charge of maintenance towards certain techniques that it will be necessary for him to control to carry out his task.

The important subject of the factory of order is evoked in the chapter 4. It is a base necessary to the comprehension of all the structure of the command set.

Lastly, an answer is given to the chapter 5, with some questions which one can provide that all future person in charge of maintenance will be able to be posed.

## 1.1 Definitions

**Mode “by batch”**

The treatment of the orders user by *Code_Aster* can be carried out according to two modes.

In the first mode, the file order is charged in memory to create the command set. This creation of the jdc makes it possible to validate syntax python (brackets, commas), syntax Aster (coherence with the catalogue) and to validate the last concepts in argument. Finally after this checking, the command set is traversed to carry out the corresponding digital processings.
This first mode is called “mode by batch”. The end-user selects this mode by specifying the value 'YES' for the keyword PAR_LOT under the order BEGINNING.

In the second mode, the command file is charged in memory to create the command set. Then the stages (equivalent to the orders) are built and carried out sequentially.
The end-user selects this mode by specifying the value 'NOT' for the keyword PAR_LOT under the order BEGINNING.

By default, the mode used is PAR_LOT=' OUI'.

**Operator of *Code_Aster***

An operator is a unit of *Code_Aster* dealing with functionality of the code. Concretely it is a subroutine FORTRAN whose name is numbered, for example the subroutine OP0001 who charges a grid in the memory with the application. The classification of the operators facilitates association between their internal representation (subroutine FORTRAN) and their external representation for the end-user (order).

**Order**

An order is a character string identifying a digital operator. It thus makes it possible to the end-user to start the execution of this operator starting from a data file called “command file”.

There exist 4 types of orders: OPER, PROC, MACRO and FORMULA.

The developer of the digital operator defines - in the catalogue of the order - those and control characteristics of the keyword corresponding to the parameters of the digital operator:

1) its name (the character string usable by the end-user),
2) its rules of composition in keywords,
3) an explanatory French and/or English comment,
4) the key defining the handbook and the chapter devoted to this keyword in the documentation of *Code_Aster*.

**Order OPER**

Besides the attributes enumerated above, an ordering of type OPER has the following attribute:

1) The type of the structure of data Aster produced by the operator and turned over by the order;

**Order PROC**

An ordering of type PROC has the characteristic not to turn over a value. This characteristic put except for, it has the same attributes as an ordering of type OPER.

**Order MACRO**

Macro is a function written in Python – by the developer Aster - which calls orders – i.e. operators - of *Code_Aster*. It stores results which could be recovered via the supervisor.

The text the macro one can be public; in this case it is stored in a specific file of under - Macro repertoire. If it is deprived, it is placed or imported in the command file.

An ordering of MACRO type makes it possible to the end-user to use the macro one. For example the order ASSEMBLY allows to launch the macro public one assemblage_ops.

**Catalogue of an order**

The catalogue of an order is the whole of the instructions Python describing the definition of the order i.e. the values assigned to the attributes of the order.

The catalogue of an order is written by the developer of the digital operator associated with the order.

**Catalogue general**

The catalogue general is a file Python containing the description of all the orders otherwise-known as containing the catalogues of all the orders.

**Command set**

The command set is the structure of data - organized in a Python object – containing the unit of the furnished information by the end-user, in the command file.

**Command file**

The command file makes it possible to the end-user to start the digital operators carrying the features of *Code_Aster* via the orders.

**Structure of data Aster**

A structure of data Aster is an organization of data produced by a digital operator of *Code_Aster* . It is identified by a type itself declared at the beginning of the catalogue (cata.py); what makes it possible to use it – symbolically_– in the command file although it is produced by FORTRAN.

**Simple keyword**

A simple keyword is a character string identifying a data used as starter by an operator (a digital functionality of *Code_Aster*). A simple keyword is thus defined inside an order of *Code_Aster*.

The end-user will be able to provide one **value** with the parameter of an order via the name of the simple keyword corresponding in the command file.

The developer of functionality of *Code_Aster* will define as for him, them **characteristics** simple keyword in the catalogue of the order containing the simple keyword:

1) its name (the character string usable by the end-user and the digital operator),
2) the type of the parameter (whole, real, text, concept,…),
3) the statute of the simple keyword (optional or obligatory),
4) the value by default to be assigned to the parameter,
5) the minimum number of data which the end-user will have to provide behind the simple keyword,
6) the maximum number of data which the end-user will have to provide behind the simple keyword,
7) an explanatory French and/or English comment.

The supervisor of *Code_Aster* load in the memory of the application, the characteristics of the simple keyword, starting from the catalogue of the orders. Then it charges (and checks) possibly the value with the parameter of the order starting from the command file provided to the application by the end-user.

The digital operator of *Code_Aster* question the supervisor via the API one `getvxx` to recover the value of the parameter starting from the name of the keyword. The supervisor turns over the value provided by the user or the value by default of the parameter.

**Keyword factor**

A keyword factor is a semantically associated character string identifying a group of keyword simple. A keyword factor is defined inside an order. An order can contain several keywords possibly optional factors., each keyword factor containing itself of the simple keywords of the same name.

The end-user will be able to define in his command file, a keyword factor by specifying his name then its **value** i.e. the value of the definite digital parameters behind the simple keywords of the keyword factor.

The developer of the functionality of *Code_Aster* defines them **characteristics** keyword factor in the catalogue of the order containing the keyword factor:

1) its name (the character string usable by the end-user and the digital operator),
2) its rules of composition in simple keywords,
3) the statute of the keyword factor (optional or obligatory),
4) the minimum number of repetition of the keyword factor,
5) the maximum number of repetition of the keyword factor,
6) an explanatory French and/or English comment,
7) the key defining the handbook and the chapter devoted to this keyword in the documentation of *Code_Aster*.

**Conditional block**

A conditional block (a block), associates:

1) simple keywords,
2) keywords factors,
3) and of the conditional blocks.

The occurrence of the block in its order, depends on a condition expressed at the time of the definition of the order by the developer of the digital functionality.

The developer of the operator corresponding to the order containing the block, specifies the characteristics of the block:

1) its name,
2) its condition,
3) its rules of composition in simple keywords,
4) an explanatory French and/or English comment.

The end-user will be able to give a value to the parameters of the treatment by using the keywords factors and the keywords simple associates in the conditional block but without specifying the name of the block.

**Rule of composition**

Composite entities of the catalogue of orders such as “command set”, orders, keyword factor and conditional block, structures of other entities while possibly following one or more rules of composition among the following ones:

AU_MOINS_UN
: The rule AU_MOINS_UN express that one at least entities whose names passed in arguments must be present in the composite entity in which figure the rule.

UN_PARMI
: The rule UN_PARMI express that one and only one of the entities of the entities whose names passed in arguments must be present in the composite entity in which figure the rule.

EXCLUDED
: The rule EXCLUDED express that if one of the entities whose name passed in argument, is present, the entities corresponding to the other arguments must miss in the made up entity in which figure the rule. Otherwise-known as if several entities of the group are present the rule is violated.

TOGETHER
: The rule TOGETHER then express that if one of the entities whose name passed in argument is present in the composed entity, all those corresponding to the other names will owe the being too. The order of the occurrences does not have importance. And if none the entities represented in the rule is present in the composite entity, the composite entity is valid.

PRESENT_PRESENT
: The rule PRESENT_PRESENT express that if the entity corresponding to **first** name is present, then all those corresponding to the other names will also owe the being in the current composite entity. The order of occurrence of the other entities does not have importance. If none the entities represented in the rule is present, the composite entity is valid.

PRESENT_ABSENT

The rule PRESENT_ABSENT express that if the entity corresponding to **first** name is present, then all those corresponding to the other names will have to miss in the current composite entity. The order of occurrence of the other entities does not have importance. If none the entities represented in the rule is present, the composite entity is valid.

Each rule of composition (called also simply “rule”) is a class (see the module regle.py).

## 1.2 Assumption of responsibility of maintenance

The following approach is proposed to the candidate with the maintenance of the Eficas graphic interface and/or the supervisor of *Code_Aster*.

1) To study “ Accas ” i.e. what is common to the Eficas graphic interface and the supervisor of *Code_Aster* ;
2) To study the structure of the catalogue general of the orders: in the file catalogues and the memory of the application. For that:

- to familiarize itself with the techniques of programming in Python, used in Accas;
- to develop a small model of factory (cf [§4.1]) for integrating well the basic mechanism of the loading of the keywords.

1) To study the structure of the command set (in its file) and memory; in particular the question of the loading of the command set (mechanism and zones of codes concerned) must be considered;
2) To examine scripts Python or the sources C concerned at the time of requests for modification or treatment of the errors detected by the users.

# 2 The organization Python sources

## 2.1 Conventions

Following conventions the purpose of which are to facilitate the reading of scripts, are imperfectly applied

1) a name of class starts with a capital letter;
2) the identifier of an object of the Python type list start with the prefix l_ (this rule is used in Eficas);
3) in the packages used by the supervisor (Core, Execution, Validation, Build and Accas) only one class is defined by module i.e. by file *.py ;
4) in the packages used by the supervisor (Core, Execution, Validation, Build and Accas) the name of each module starts with a prefix indicating the name of the package.
    1) N_ for Core
    2) V_ for Validation
    3) E_ for Execution
    4) B_ for Build
    5) A_ for Accas

## 2.2 Typology of the modules Python

Each class is defined in a module: for example, the class MCSIMP is defined in the module N_MCSIMP.py where the prefix N_ indicate the name of the package (Core) containing the module.

The technique of the packages makes it possible to cut out the modules according to the sphere of activity in which it acts. With each field a package Python corresponds

For example, class MCSIMP exists in each of the five packages,

1) ·Core: N_MCIMP.SIMP ;
2) ·Validation: V_MCSIMP.SIMP ;
3) ·Ihm: I_MCIMP.SIMP ;
4) ·Accas: A_MCIMP.SIMP ;
5) ·Build: B_MCIMP.SIMP ;

Core

This package contains primarily the system of class of the factory of the command set.

Validation

This package contains the modules carrying out the checks of validity of the objects (conditional orders, blocks, keywords,…).

Build

This package is present only in the supervisor contains the modules treating the orders of the macro type and the methods of request with the command set since the API-C: the GETVxxx interfaces.

Accas

This package is most important. It contains – in particular - the classes more including used as well by the interface – Eficas graph as by the supervisor of *Code_Aster*. It is in this package that it is necessary to search the objects and the methods specialized in the treatment – not graph – orders:

1) loading of the catalogue;
2) loading of a command set;
3) execution of the command set.

The classes girls defined in this package are it by heritage of classes relationships having the same name that the classes girls but being located in a different package.

**Ihm**

The classes of this package enrich the classes by the Core of methods – nonrelated to the graphic aspect – used by the Eficas graphic interface.

**Editor**

This package contains the modules of graphic treatment of the command set.

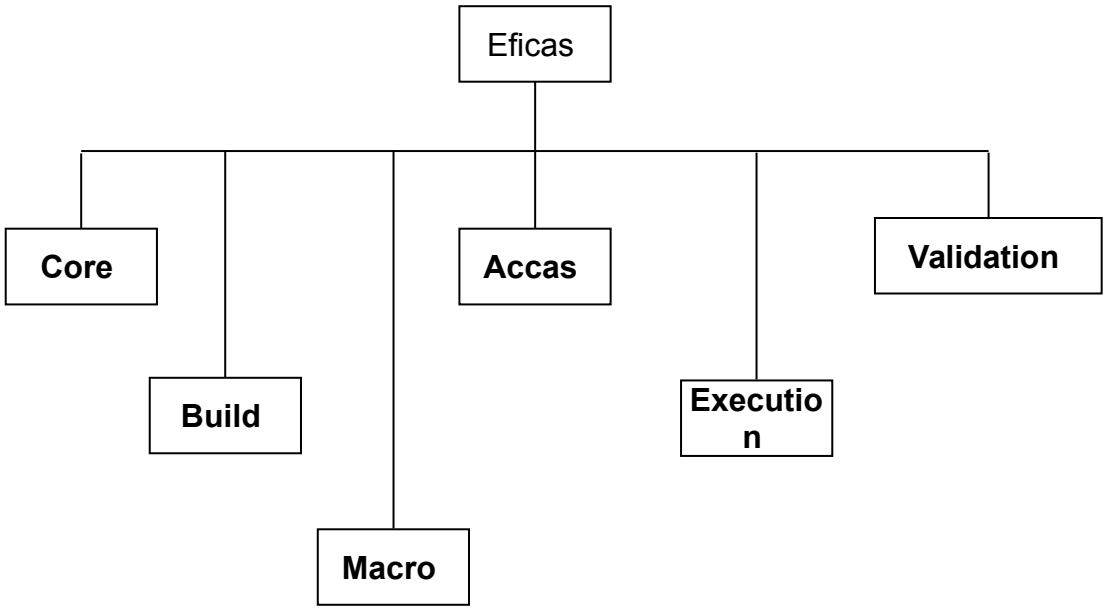
## 2.3 Hierarchy of the repertoires of the sources

### 2.3.1 The python sources of the supervisor

The supervisor of *Code_Aster* is composed of modules python writings:

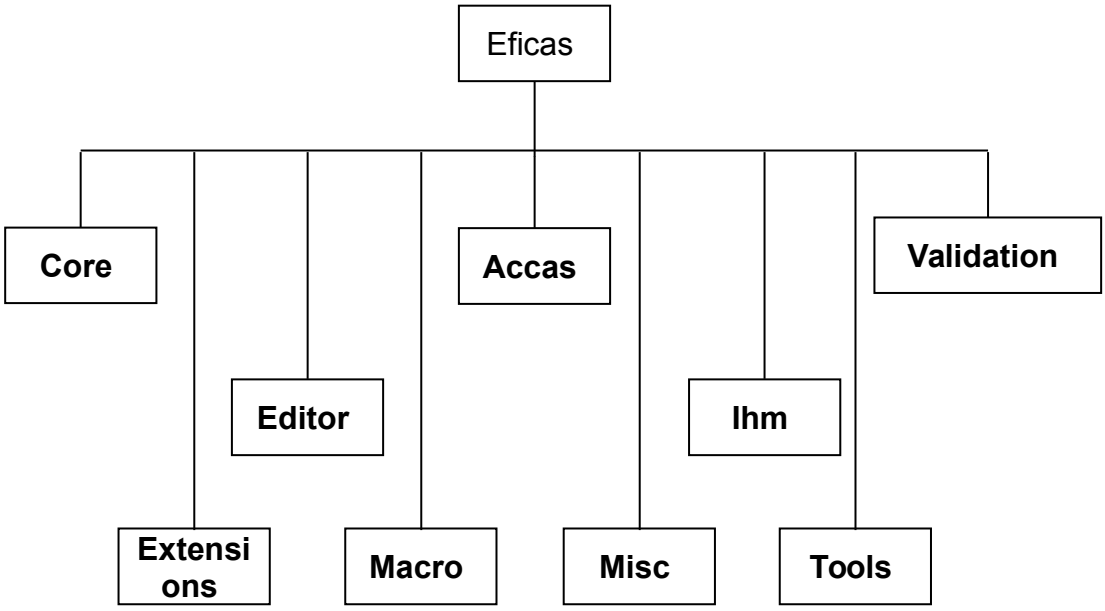
1) out of C for the application interface of the supervisor: the module `aster` (`astermodule.c`);
2) in Python for the management of the command set (loading, organization,…).

The diagram below presents the repertoires containing the Python sources of the supervisor.



### 2.3.2 Sources of the Eficas interface-graph

The sources are organized in repertoires under the repertoire of installation `Eficas` :




## 2.4 Environment

This paragraph describes the pre-necessary conditions with the operation of the Supervisor and the Eficas graphic interface.

### 2.4.1 Installation of the interpreter Python

To inform

### 2.4.2 Shell parameters of configuration

To inform

### 2.4.3 Installation of the Eficas interface-graph

To inform

### 2.4.4 Update of the supervisor

To inform

# 3 Use of the language Python in Eficas and the supervisor of *Code_Aster*

One presents here the techniques of programming in language Python, whose control is necessary to any candidate with the maintenance of the Eficas graphic interface and the supervisor of *Code_Aster*.

## 3.1 Call of a function with a variable number of arguments

The end-user of *Code_Aster* use a script Python to start the features and to provide values to the parameters of the functionality. The supply of these values being sometimes optional, the Accas core uses the mechanism envisaged in the language Python to pass a variable number of arguments to a function.

To maintain Accas, it is, consequently, necessary to control this technique whose we present a small example below.

```
# script main.py
def fonc (number, *tup_arguments, ** d_arguments):
    print number
    print repr (tup_args)
    print repr (d_args)

fonc (11111, "arg 2", "arg 3", 4, n=5, j=6)
```

Under Unix, the interpretation of script `main.py` is done by:

```
$ python main.py
```

It gives the following result on the standard exit:

```
11111
('arg 2', 'arg 3', 4)
{': 5, 'I: 6}
```

Only the argument number is obligatory. The possible positional arguments according to are stored in a tuple (which can be empty) and the possible arguments passed by keyword are stored in a dictionary.

This technique is used in particular, by the objects which build the command set in the memory then which initialize it.

## 3.2 Use of spaces of name

### 3.2.1 Concept of space of names

One *space of names* (see [bib1], page 97) is one *dictionary Python* containing a set of couples name/value. The name is in general a character string and the value can be a digital value, a function or an object.

In a module Python, each instruction is carried out in a called specific space of names **local space of names** whose contents can be displayed by the function `locals ()`. The instructions also have access to**total space of names** whose contents can be displayed with the function `globals ()`.

Accas uses in particular, a space of names to store the dictionary of the definitions which will be used to interpret the keywords and to charge their value in the memory.

### 3.2.2 The module __builtin

A remarkable module is the standard module __builtin who – except typical case ([bib2], page 100) – is referred in each module user by the attribute __builtins **in ReadOnly mode only**.

The interpretation of the following sequence:

```
# script main.py
print globals ()
```

poster

```
{'__doc': Nun, '__name': '__main', '__builtins': <module'__builtin '(built-in) >}
```

The module __builtin by default is imported; all the data and the functions which it contains are thus accessible by defaults, in all the modules of the application. Data and functions defined in the module __builtin can thus be regarded as most total with the application.

Inter alia, one finds in this module, the following tools:

1) functions locals (). and globals () ;
2) the variable __debug who conditions interpretation according to his value 1 (defect) or 0;
3) the function “builtin” __import.

```
# script main.py
# One installs the module context in the total space of the interpreter
# (__builtin)
# under name CONTEXT in order to have access to the functions
# get_current_step, set_current_step and unset_current_step of anywhere
importation context
importation __builtin
__builtin__.CONTEXT=context
```

It is possible to import and enrich the module __builtin ; this technique is used in Eficas in script Noyau/__init__.py.

### 3.2.3 Execution of an order Python in a space of names

In the following example, the variable GB defined at the beginning of script is accessible in all the units from the module: it is definite within the space of names total.

```
# script main.py
gb=2
def fonc (a):
    b=gb*a
    print "locals () =", locals ()
    print "globals () =", globals ()
    return B
x=123
z=-1
u=fonc (X)
print z+u
```

The interpretation of script main.py poster:

```
locals () = {'B': 246, 'has': 123}
globals () = {'__doc': Nun, 'fonc': <function fonc At 0x810aee4>, 'Z': -1, 'X': 123,
'__builtins': <module '__builtin' (built-in) >, '__name': '__main', 'GB' : 2}
```

A specific space of names can be created and used to carry out instructions stored in a character string.

```
# script main.py
d_contexte= {'has': 1, 'B': 2}
print d_contexte
s_commande=' x=a+b'
exec s_commande in d_contexte
print d_contexte
```

When he interprets script main.py, the interpreter enriches space by names d_contexte with 'X': 3 the result of the instruction s_commande. But the local space of names does not contain the result the instruction s_commande.

During the use of the instruction exec , one can specify that the data created owe the local being within the space of names as the example shows it below.

```
# script main.py
# script main.py
d_contexte= {'has': 1, 'B': 2}
print d_contexte
s_commande=' x=a+b'
exec s_commande in d_contexte, locals ()
print 'x=', X
```

what displays:

```
{'B': 2, 'has': 1}
x= 3
```

The instruction Python exec allows to create a space of names by the interpretation of a script Python - stored in a character string - by specifying a space of names behind the keyword in, for example: exec s_script in space

## 3.3 A second manner of importing a module

The function "builtin" __import ([bib2], p. 100), allows to import a module starting from its name stored in a character string. In fact, this function is called by the interpreter Python at the time of the importation of a module by the instruction importation ([bib3], p629). This technique is used on several occasions in Eficas (packages Editor and Extensions).

```
nom_module=' string'
print to dir ()
module_string =__import (nom_module)
print to dir ()
print to dir (module_string)
print module_string.lower ('ABCD')
```

It is noticed that the function __import turn over in module_string, a reference on the module G-string from which it is possible to reach the tools contained in the module.

The function "builtin" `__import` can also be used to import a specific module starting from a parcelling (package). The following example imports the catalogue general of the orders of *Code_Aster* then poster on the standard exit the name of each order.

```
importation sys
TOP='/local/yessayan/Eficas/EficasPourSalome' # repertoire of installation
sys.path.append (TOP)
sys.path.append (TOP+'/Aster')
from Catastrophes importation catastrophes
for order in cata.JdC.commandes:
    print commande.nom
```

It can be written:

```
importation sys
TOP='/local/yessayan/Eficas/EficasPourSalome'
sys.path.append (TOP)
sys.path.append (TOP+'/Aster')
package=__import ('Cata.cata')
module_cata= getattr (package, 'catastrophes')
for order in module_cata.JdC.commandes:
    print commande.nom
```

As it is indicated in the documentation of python (http://www.python.org/doc/current/lib/built-in-funcs.html), the instruction

```
__import ('Cata.cata')
```

import the package Catastrophes.

It thus remains to recover the module of the catalogue starting from the package for example:

```
module=getattr (package, 'catastrophes'))
```

The order `__import` allows to import a module or a package whose name is known only at the time of the interpretation of current script.

## 3.4 Module Python written in language C

To inform

# 4 The catalogue general of the orders of *Code_Aster*

## 4.1 Example 1: a factory to build the simple keywords

### 4.1.1 Principle of the factory

The end-user of *Code_Aster* provides values to the features via keywords. A keyword makes it possible to introduce a value (keyword simple) or another keyword (keyword factor).

The mechanism of loading and especially of use of the value of the keyword is complicated by the fact that the type of this value is not single. This type can be for example: float, int, G-string,… In the following example:

```
MASSE_VOLUMIQUE=7800.0
FICHIER_MAILLAGE=' maillage.unv'
```

The keyword simple MASSE_VOLUMIQUE and FICHIER_MAILLAGE receive the value respectively 7800.0 and 'maillage.unv'

Some share thus should be described the characteristics of the simple keyword (name, standard, value by default, unit, optional or obligatory statute,…). The answer to this question is founded on the separation characteristics/following value:

1) the main role of an object of the type MCSIMP is to wrap **value of a simple keyword** otherwise-known as the principal attribute of an object of the type MCSIMP is a value;
2) an object of the type SIMP plays two leading roles:

- to wrap the definition of a simple keyword: an object SIMP contains the whole of the characteristics of a simple keyword: its name, the type of its value,…;
- but this object also has a function __call who allows to generate a simple keyword starting from his characteristics; an object SIMP can be regarded as a machine to manufacture an object of MCSIMP from where the term *factory*

Each object MCSIMP contains its value and a reference on the object SIMP who describes his characteristics.

The technique of the factory is presented Ci below by reducing the classes MCSIMP and SIMP with their minimum.

1) The class MCSIMP has two attributes:

- definition : its definition (a reference on the object SIMP who created the object MCSIMP) from which it is possible to recover the text part of the keyword: definition.nom or its type definition.type ;
- valley : its value

1) the class SIMP has two attributes:
   1) name : the part text (character string which can contain a white space) keyword MCSIMP that the object SIMP will build;
   2) type : the type of the value introduced by the keyword MCSIMP that the object SIMP will build.

The function __call class SIMP create an object of the type MCSIMP. That implies that class MCSIMP is defined before the definition of this function.

| In the model of design factory applied to the command set of *Code_Aster*, the class manufactured (example MCSIMP) must be defined before the class producer (SIMP). |
|---|

```
# script main.py
importation sys
# G-string in python 2.1 becomes str starting from python 2.2
s_version=str (sys.version_info [0]) + '. '+str (sys.version_info [1])
assert (float (s_version) <=2.1), 'the version '+s_version+' of python is
INVALID'

class MCSIMP:
    def __init (coil, definition, valley, parent=None):
        self.definition = definition
        self.val        = valley
        self.parent     = Nun

class SIMP:
    def __init (coil, name, type):
        self.nom = name
        self.type =type
    def __call (coil, valley, parent=None):
        assert (str (standard (valley))== " <type '"+self.type+" '>")
        return MCSIMP (definition=self, val=val, parent=parent)

d_context= {
        'MASSE_VOLUMIQUE'  : SIMP (nom=' MASS VOLUMIQUE', type=' float'),
        'FICHIER_MAILLAGE': SIMP (nom=' FILE MAILLAGE', type=' string')
}

s_commande = 'rho=MASSE_VOLUMIQUE (7800.0)'
exec s_commande in d_context, locals () #  rho  is  added  to  the  space  of
names locals ()
print rho
print rho.definition.nom
print rho.val


s_commande = 'mail=FICHIER_MAILLAGE ("maillage.d")'
exec s_commande in d_context, locals () # e-mail is added to the space of
names locals ()
print e-mail
print mail.definition.nom
print mail.val

sys.stderr.write ("FINE NORMAL of main.py " + ' \)
```

The interpretation of script `main.py` above gives following posting:

```
<__main__.MCSIMP authority At 0x810c4ac>
DENSITY
7800.0
<__main__.MCSIMP authority At 0x810c4d4>
FILE GRID
maillage.d
```

Let us consider the lines

```
s_commande = 'rho=MASSE_VOLUMIQUE (7800.0)'
exec s_commande in d_context, locals ()
```

The mechanism of construction of the object rho, within the space of names room, is a mechanism of factory including the following stages:

1) Within the space of names d_context, the order becomes:
rho=SIMP (nom=' MASS VOLUMIQUE', type=' float') (7800.0)
then it is carried out;
1) An object of the type SIMP is built with the name 'DENSITY';
2) method __call is called with val=7800.0 in argument;
3) starting from the name (self.nom) and of the value (valley), method __call create and turns over an object of the type MCSIMP ;
4) the turned over object is affected with the variable rho within the space of names room.

**Important:**

*In the order s_commande , the keyword should not contain of white space:*
*" rho=MASSE_VOLUMIQUE (7800.0) "is a valid instruction Python.*

A white space in "rho=MASSE VOLUMINAL (7800.0)"would generate an error with interpretation.

In the dictionary of the keywords the character strings used for the keys, must obey the writing rules of an identifier Python: no white space.

### 4.1.2 The organization of the factory in files

The organization in files, presented below, described that which is used for the Eficas graphic interface and the supervisor. It also describes the procedure of loading of the command set in the memory starting from the furnished information – in the command file - by the end-user.

The files all – scripts Python - must be defined in the following order:

- MCSIMP. py
- SIMP.py
- dictio.py : the dictionary of the keywords MASSE_VOLUMIQUE and FICHIER_MAILLAGE
- valeurs.py : the file of the values provided by the end-user (for Aster, the command file user)
- main.py : the code charging the values provided by the end-user by reading and interpreting the file valeurs.py (for Aster the achievable one, also interpreter python: aster.exe)

Let us recall that the definition of the class MCSIMP must be carried out before that of the class SIMP. What leads to an organization starting with the definition of the class MCSIMP.

```
# MCSIMP.py script

class MCSIMP:
    def __init (coil, definition, valley, parent=None):

        assert (definition. __class. __name__==' SIMP')
        self.definition = definition

        assert (standard (valley). __name__==definition.type)
        self.val = valley
        self.parent =None

        return
```

The attribute `self.parent` useless here, will be used when the keyword is defined inside an order. It will then contain a reference on this order.

The developer of Accas can then define the class `SIMP`.

```
# SIMP.py script

importation MCSIMP
importation standards

class SIMP:
    def __init (coil, name, type):
        self.nom = name
        self.type =type
    def __call (coil, valley, parent=None):
        assert (str (standard (valley))== " <type '"+self.type+" '>")
        return MCSIMP.MCSIMP (definition=self, val=val, parent=parent)
```

Once two modules `SIMP` and MCSIMP placed at its disposal, the developer of digital features of the code can now define a "application catalogue of keywords 'in the file `dictio.py`. This script defines in a dictionary python, description (standard, possible values, field of definition,…) values associated with the keyword with kind to allow the reading of these values.

For example:

```
# script dictio.py

from SIMP importation SIMP
dict= {'MASSE_VOLUMIQUE': SIMP (nom=' MASS VOLUMIQUE', type=' float',
           'FICHIER_MAILLAGE': SIMP (nom= 'FILE MAILLAGE', type=' string')}
```

And the end-user can finally use the features by providing a script, for example `valeurs.py`.

```
# script valeurs.py

rho=MASSE_VOLUMIQUE (7800.0)
mail=FICHIER_MAILLAGE ('maillage.d')
```

To charge in memory the abundant data by the user, the code will read the command file as follows:

```
# script main.py

from SIMP importation *


d_context= {'MASSE_VOLUMIQUE': SIMP (nom=' MASS VOLUMIQUE', type=' float'),
            'FICHIER_MAILLAGE': SIMP (nom=' FILE MAILLAGE', type=' string')}


nom_script_valeurs = 'valeurs.py'
f=open (nom_script_valeurs, 'R')
string_parametres = f.read ()
f.close ()

exec string_parametres in d_context, locals ()

print rho,  rho.definition.nom, rho.val
```

To charge in memory, the value associated with a simple keyword, should be interpreted script python – text of the keyword simple and value (S) associated (S) provided by the end-user - within the space of names (the dictionary `d_context`) keyword.

## 4.2 Example 2: a factory to build an order

In practice, the keywords are not separately but obligatorily defined inside an order. What complicates the process of construction of the keywords in the memory. We now present an example – always simplified – intended to facilitate the comprehension of this process.

For that we will consider that a command set is a list of orders of type “procedure” and that each order is parameterized by keyword simple and only by simple keywords (not of conditional block, not of put-key factor). This simplification increases legibility while preserving all the categories of difficulties of facing to charge the command set in memory.

One thus starts by adding a factory of orders of the type `PROC_ETAPE`.

### 4.2.1 Addition of the classes PROC and PROC_ETAPE with the factory

The class PROC_ETAPE who models an ordering of standard “procedure”, is very close to the class MCSIMP. So much so that both could inherit a class common mother (it is the case in Accas besides).

```
# PROC_ETAPE.py script

print "\ tImport of "+__name

class PROC_ETAPE:
    def __init (coil, definition, arguments= {}):
        print 2* ' \ t'+ " PROC_ETAPE __init: creation of an object "+ \
        coil. __class. __name
              print  3*  '  \  t'+'  PROC_ETAPE  __init:  definition.nom=',
definition.nom
        print 3* ' \ t'+' PROC_ETAPE __init: arguments=', arguments

        assert (definition. __class. __name__==' PROC')
        self.definition = definition
        self.valor = arguments

        return
```

An object PROC_ETAPE is manufactured by an object of the type PROC. Its attribute self.definition is a reference on the object PROC who created it.

```
# PROC.py script
print "\ tImport of "+__name
importation PROC_ETAPE
from SIMP importation *

class PROC:
    def __init (coil, name, op, ** arguments):
        print 1* ' \ t'+ " PROC __init: creation of an object "+ \
        coil. __class. __name
        print 2* ' \ t'+' PROC __init: nom=', name
        print 2* ' \ t'+' PROC __init: arguments=', arguments
        self.nom = name          # text of the order
        self.entites = arguments     # dictionary of the manufacturers
        self.op = op            # number of operator FORTRAN
        return

    def __call (coil, ** arguments):
            # arguments contains the definition of the values of MCSIMP
(MASSE_VOLUMIQUE,
        # FICHIER_MAILLAGE)
        print 1* ' \ t'+' PROC __call: arguments=', arguments
        print 1* ' \ t'+' PROC __call: self.entites=', self.entites
        # construction of the simple keywords of the order and addition in
        # the dictionary of the keywords of order PROC in the course of
        # construction
        dict = {}
        for K, v in args.items ():
            dict [K] = self.entites [K] (val=v, parent=self)
        print 1* ' \ t'+' PROC __call: dict=', dict
        return PROC_ETAPE.PROC_ETAPE (coil, dict)
```

During its creation carried out starting from the catalogue, object PROC memorizes in the dictionary `self.entites`, the composition of the order in simple keywords; this information will be used in the second time – to build the simple keywords of the order – when the object `PROC` will be called upon via its method `__call`.

It is also the method `__call` who will initialize the keyword located inside the order with the values provided in the command file (the module `SIMP` is exactly that presented in the first example).

### 4.2.2 Concept of catalogue of orders

In the first example ([§4.1]), the description of the keywords was made in the dictionary (the space of names) `d_context`. But to facilitate the task of the developers of *Code_Aster*, it is preferable to describe the orders and their contents via a script Python then to convert this script into a dictionary which will be used as space of names for the loading of the orders.

For our second example, the catalogue can S' to write thus;

```
# script cata.py

print 1* ' \ t'+ " Importation of "+__name

from SIMP importation *
from PROC importation *

AFFE_MATERIAU=PROC (nom=' AFFE_MATERIAU',
                    op=10,
                    MASSE_VOLUMIQUE=SIMP (nom= " DENSITY", type=' float'),
                    FICHIER_MAILLAGE=SIMP (nom= " FILE GRID", type=' string'))
```

This catalogue contains only one order: `AFFE_MATERIAU` the use will start the call to routine FORTRAN `op0010`. This routine will use two parameters `MASSE_VOLUMIQUE` and `FICHIER_MAILLAGE`

The conversion of the catalogue into a dictionary is done into important simply the catalogue in a space of names. What does the following sequence:

```
d_context= {}
string_cata= " from catastrophes importation *"
exec string_cata in d_context
```

### 4.2.3 Concept of command file

The command file is also a script Python, very simple intended to be interpreted within the space of names of the catalogue of orders. For example:

```
# script commandes.py

AFFE_MATERIAU (MASSE_VOLUMIQUE=7800.0, FICHIER_MAILLAGE= " maillage.unv")
```

In script commandes.py above, the end-user asks for the execution of routine FORTRAN op0010 with MASSE_VOLUMIQUE=7800.0 and FICHIER_MAILLAGE= " maillage.unv". He is interpreted by the following sequence:

```
f_commandes=open ('commandes.py', 'R')
string_commandes = f_commandes.read ()
f_commandes.close ()
exec string_commandes in d_context
```

At the end of which command set (here reduced to the only order AFFE_MATERIAU) is defined within the space of names d_context.

### 4.2.4 Use of the catalogue and the command file

Script following Python carries out the loading of the catalogue, the loading of the command file and the examination of the command set in the memory.

```
# script main.py
importation traceback
trace=traceback.extract_stack ()
script_file=trace [0] [0]
prefixe=script_file+': '
print prefixe+ " BEGINNING of ", script_file

d_context= {}
# 1. Loading of the catalogue
# Creation - into important the catalogue catastrophes - of a space of
being useful name
# for the interpretation of the command set

print 3* ' \ n'+prefixe+ " importation of the catalogue"
string_cata= " from catastrophes importation *"
exec string_cata in d_context

# 2. Loading of the text of the orders
# Reading of the command file (the text of the orders east stores in one
# chains caratères)

print 3* ' \ n'+prefixe+ " reading of the text of the orders"
f_commandes=open ('commandes.py', 'R')
string_commandes = f_commandes.read ()
f_commandes.close ()

# 3. Creation of the command set
# Interpretation of the text of the order in the d_contexte of the
catalogue. The structure
# command set, produced, is stored within the space of name d_context
print 3* ' \ n'+prefixe+ \
"Conversion of the text of the orders (G-string) into a command set
(d_context)"

exec string_commandes in d_context

# 4. Course of the structure command set in the d_contexte

print 3* ' \ n'+prefixe+ " Posting of the command set"
importation standards
for K, v in d_context.items ():
```

```
          standard yew (v) ==types.InstanceType and v. __class. __name__==
'PROC_ETAPE':
                # if the attribute is an order, one examines his value
                # it be-have-to say its keywords
                print 1* ' \ t'+v.definition.nom+' \ t'+str (v. __class)
                for kk, vv in v.valeur.items ():
                        print 2* ' \ t'+kk, ': ', vv, '\ you, vv.val

print 2* ' \
print prefixe+ " FINE NORMAL of", script_file
```

## 4.3 The file catalogues

### 4.3.1 Catalogue of an order

The catalogue of an order contains the description of the order. Each order is an authority of class OPER, PROC or MACRO

| Attribute | Description |
|---|---|
| name | name of the order (character string without white space) |
| op | number of operator FORTRAN: entirety ranging between 1 and 199 |
| sd_prod | type of the result, for the orders of the type OPER |
| rule | list of the rules of composition of the order |
| Fr | French comment |
| Doc. | reference of documentation Aster |
| reentrant | |
| repetable | |
| entities | Composition of the order: arguments containing the description of the keywords used to provide values to the keyword of the order |

Example

```
ASSE_MAILLAGE=OPER (nom=' ASSE_MAILLAGE', op= 105, sd_prod=maillage,
                   fr=' To assemble two grids under only one nom',
                   docu=' U4.23.03-e', reentrant=',
         GRID        =SIMP (statut=' o', typ=maillage, min=2, max=2),
) ;
```

In this example, drawn from the true catalogue of orders from *Code_Aster* :

1) the described order is called ASSE_MAILLAGE ;
2) it makes it possible to start operator FORTRAN op0105 ;
3) it turns over a data of the type grid ; this type is defined in the beginning of catalogue cata.py ;
4) the use of operator FORTRAN op0105 requires obligatorily, the supply of 2 data of type grid

### 4.3.2 General structure of the file catalogues

The file catalogues orders of *Code_Aster* – the module `$TOP/Aster/Catastrophes/cata.py` - contains following information:

1) importation of all information of the module `Accas`, in particular `Accas.A_ASSD.ASSD`
2) declaration of the types deriving from the generic type `Accas.A_ASSD.ASSD`, used to typify the values of the keywords or the values turned over by the orders; for example types:

- `entirety`, `reality`, `complex`, `list`, `chain` ;
- the geometrical ones, `No` (node), `groupno`, `my` (mesh), `groupma` ;
- `grid`, `model`, `to subdue`
- *etc*

1) the list of the catalogues of the orders i.e. the description of all the orders, with for each order

## 4.4 Installation of the catalogue in the memory

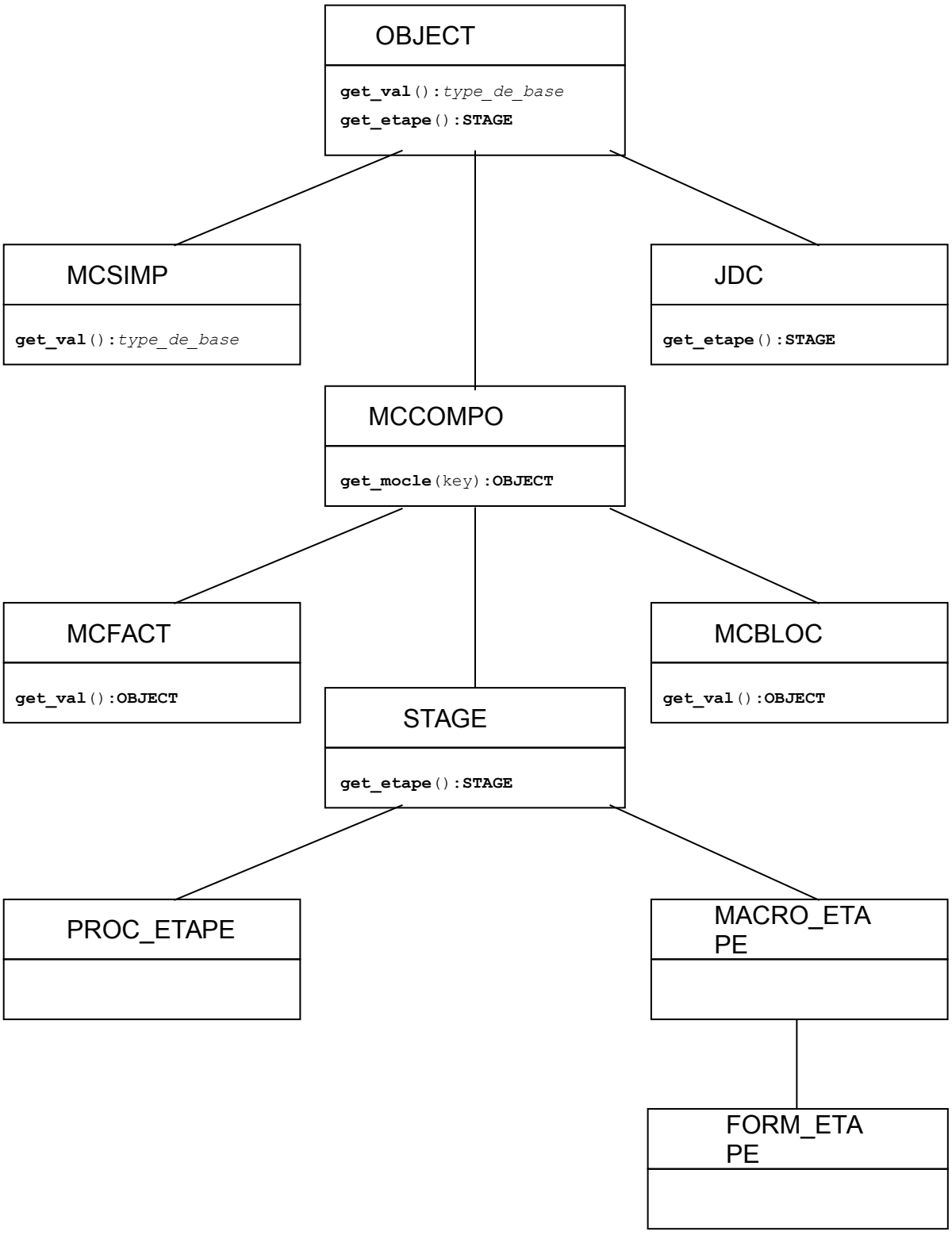
It is important to remember that a reference on the catalogue running is stored in the module whose reference is stored in `CONTEXT`. The reference `CONTEXT` is itself definite within the space of names total `__builtin`

A reference on the catalogue running is obtained by:

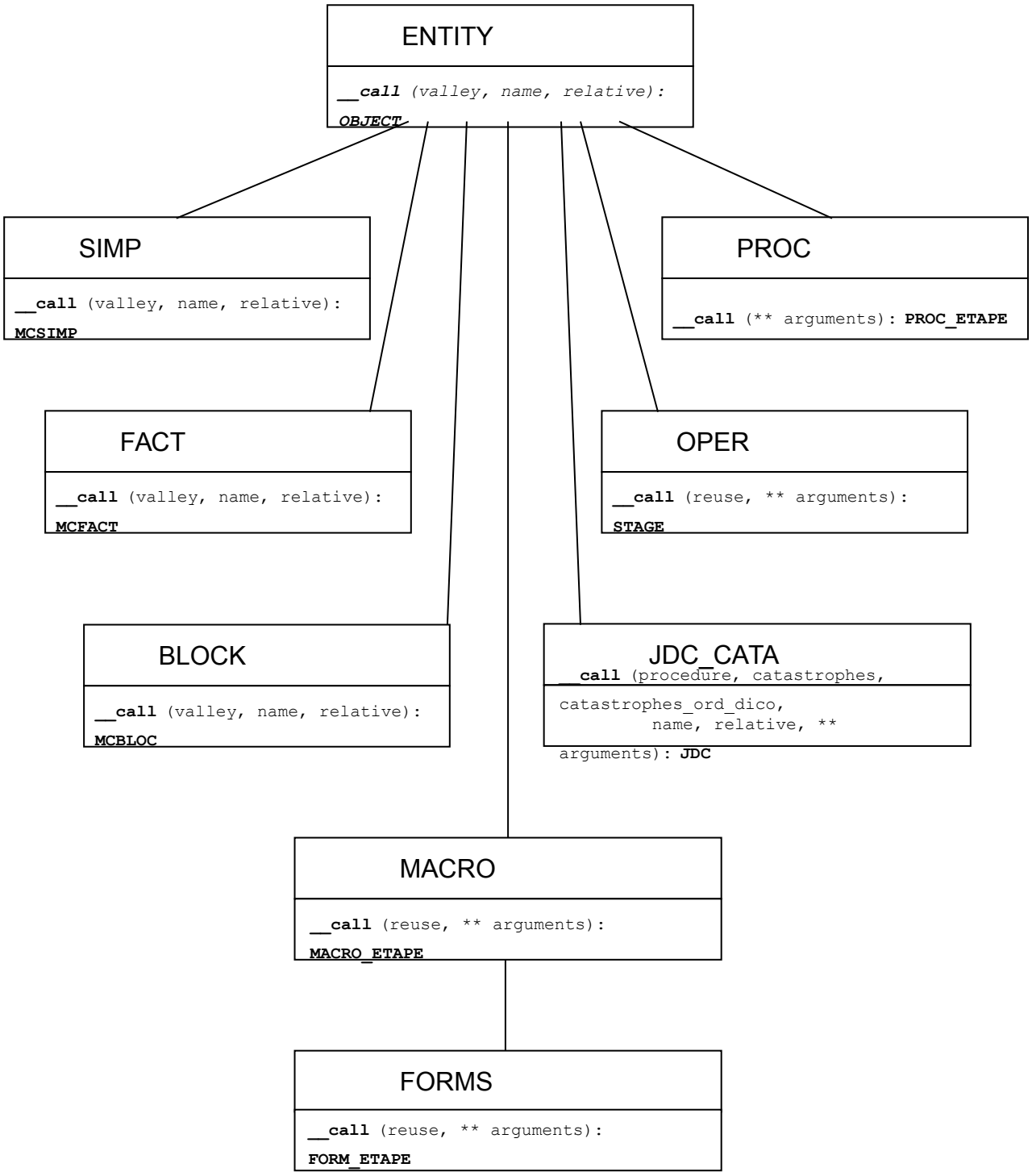
```
CONTEXT.get_current_cata ()
```

### 4.4.1 The structure of data in the package Core

The following classes have as a role to store the command set in the memory and to restore the value of the keywords to the request.



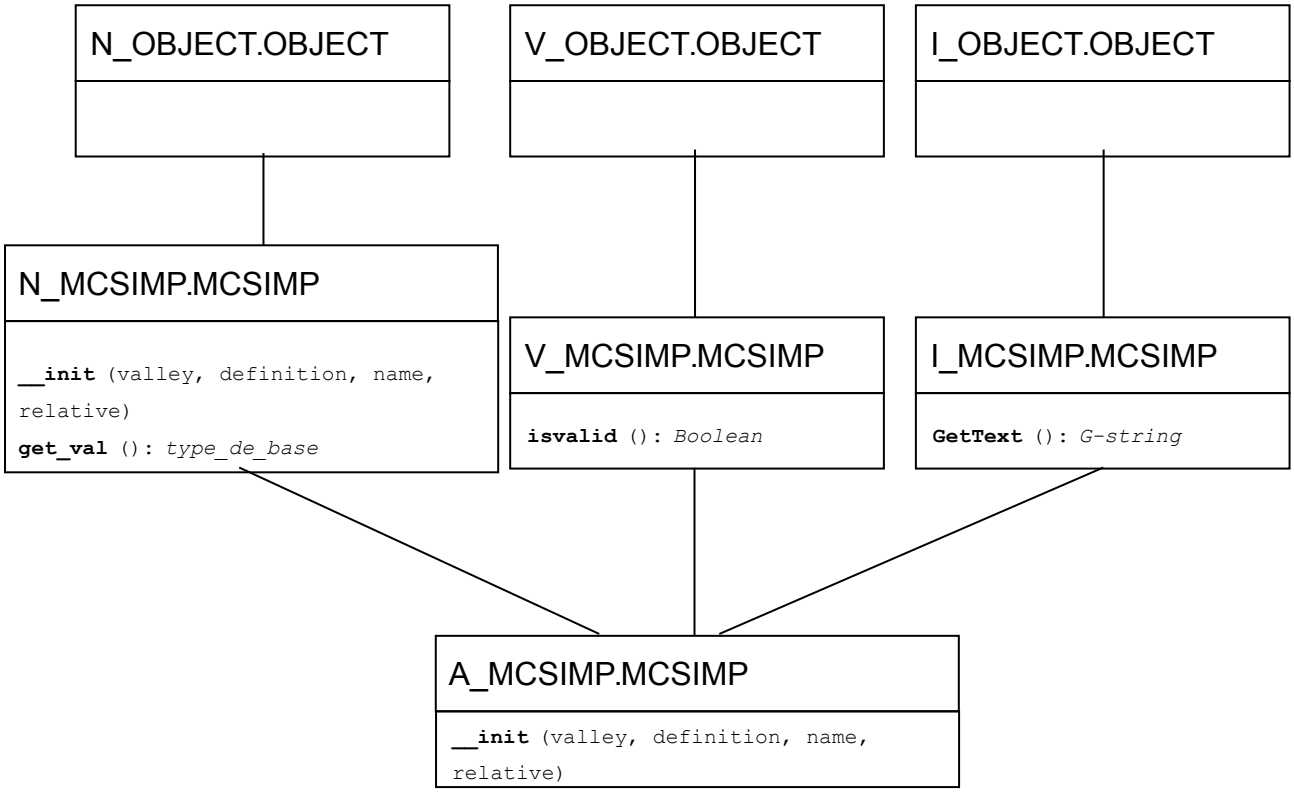
The whole of the classes presented below take part in the loading of the command set in the memory. They inherit all the abstract class ENTITY. And their principal function is the method __call who ensures the role of factory (factory) of object.



### 4.4.2 The Accas package

The Accas package is the principal package used by the supervisor of *Code_Aster* and by the Eficas graphic interface. It contains the modules corresponding to the classes obtained by assembly (use of the multiple heritage) of the classes of the other packages.

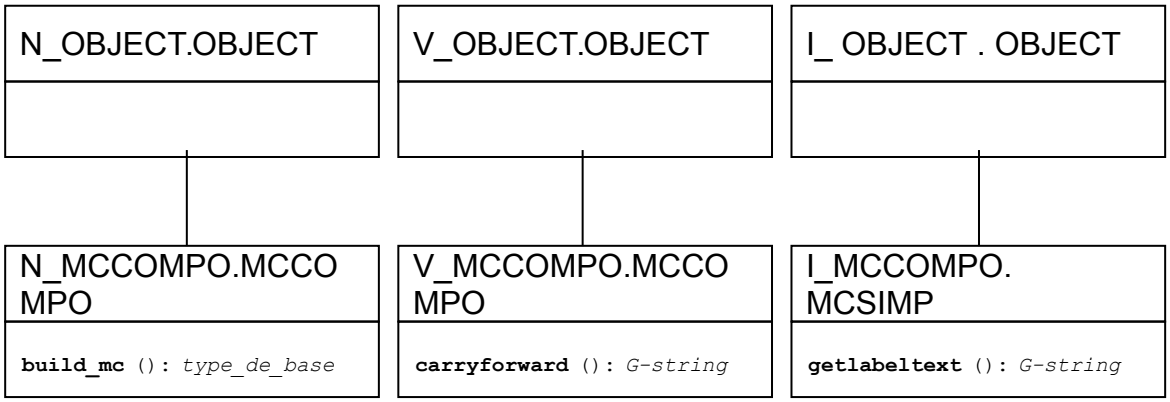
**The class MCSIMP**




Objects of the class MCSIMP are built with the manufacturer of the class A_MCSIMP.MCSIMP. The diagram above represents the principal contribution of each package to the features of the class MCSIMP
Method get_val () is used by the supervisor to recover the value of the simple keyword and to turn over it to the operator of *Code_Aster* who the request.

**The class MCCOMPO**

The class MCCOMPO – which models one OBJECT composite - does not exist in the Accas package but it is important because it is used as a basis for the classes MCFACT, MCBLOC.




The diagram above highlights behaviors justifying the existence of the class MCCOMPO.

1) method build_mc () in the package Core: it builds the objects located insideOBJECT ;
2) the method carryforward () which turns over the report of validation by applying the method isvalid () to all them OBJECT located in MCCOMPO.

# 5 The macro-orders Python

Description summary of the nature of the macro-orders:

1) An macro-order Python can produce one or more results (called concepts) whereas the simple orders produce zero (ordering of type PROC) or a result (ordering of type OPER);
2) An macro-order has parameters like an ordinary order; they are simple keywords factors and keywords;
3) The principal concept produced by macro is turned over by the macro one while the secondary produced concepts are arguments modified by the macro one;
4) The secondary produced concepts must be typified: that is done via a function provided by the developer of macro via the argument sd_prod the macro one;
5) The body of the macro-order is a function fascinating Python charges with it the treatment which includes the call to other orders (or even with other macro-orders).

To define an macro-order, its developer must thus define:

1) keywords of the order;
2) the type of the produced concepts;
3) the body of the macro-order.

# 6 Some questions

## 6.1 How to know which file catalogue is used by a calculation?

The situation is the following one:

1) a calculation was carried out with *Code_Aster* ;
2) several files of catalogue of the orders are present in the environment;
3) the user – or the developer – wants to know that which is actually used.

A solution can be as follows:

1) to import the catalogue in the command file, for example in `ahlv100a.comm` ;
2) to insert in the command file the following sequence which:

- import the catalogue,
- writing the reference of the catalogue on the standard exit,
- the treatment stops.

```
importation catastrophes, sys
print 'ahlv100a.comm: catastrophes=', catastrophes
sys.exit (0)
```

With this sequence, one gets a result of the following style:

```
catastrophes=<module 'catastrophes' from
'/home/salome/yessayan/Devel/Asterv7/bibpyt/Catastrophes/cata.pyc'>
```

From where it is deduced that the catalogue used can be in the file:

```
/home/salome/yessayan/Devel/Asterv7/bibpyt/Cata/cata.py
```

## 6.2 How mode DEBUG is managed?

In Eficas and the supervisor - in fact, in any script Python - mode DEBUG is managed via a definite standard variable within the space of names total `__builtin : __debug`. In normal mode of interpretation (`python main.py`), `__debug` is put at **1** (in `main.pyc`) but in optimized mode (`python -O main.py`) `__debug` is put at **0** (in `main.pyo`)

At any moment, in all the modules, the variable __debug can be used to condition the treatment.

## 6.3 Where the catalogue of order is it charged in the memory

Whether it is in the supervisor or the Eficas graphic interface, `JdC`, the Python object containing the catalogue is created in module `catastrophes` package `Catastrophes`. More exactly, `JdC` is created at the time when the module `catastrophes` is imported: the importation is carried out in the method `imports` class `SUPERV` module `Execution/E_SUPERV`. After the importation, the object `JdC` contains – in its attribute `orders`, of type `list` - the definition of all the orders available like that of all the associated keywords with each order.

### 6.3.1 Creation of the JdC object

At the beginning of script cata.py, it `JdC` is declared by the instruction:

```
JdC = JDC_CATA (code=' ASTER',
                execmodul=None,
                rules = (AU_MOINS_UN ('BEGINNING', 'CONTINUATION'),
                          AU_MOINS_UN ('FINE'),
                          A_CLASSER (('BEGINNING', 'CONTINUATION'), 'END')))
```

This instruction calls mainly on the method `__init` class `N_JDC_CATA.JDC_CATASTROPHES` (package `Core`). In this method, the object `JdC` created is recorded in total space `__builtins` , via the variable _catastrophes in the module `CONTEXT : __builtins ["CONTEXT"]. _catastrophes`.

**A reference on the catalogue running is always available within the space of names total `__builtins .`**

### 6.3.2 Loading of the entities of the catalogue in the JdC object

After creation the loading is always carried out at the time of the importation of the catalogue in the method `imports`, by creating objects of the types

1) `OPER:`
2) `PROC:`
3) `MACRO:`

## 6.4 Where the command set is carried out by the supervisor?

Command set `J` (object of the type `Accas.A_JDC.JDC`) is carried out in the method `Carry out` class `SUPERV` in the module `E_SUPERV` package `Execution`.

Two cases are possible:

1) In mode PAR_LOT=' OUI' (in script the attribute j.par_lot command set is positioned with 'YES'), the treatment is carried out by the call

```
j.exec_compile ();
```

1) In mode PAR_LOT=' NON' (in script the attribute j.par_lot command set is positioned with 'NOT'), the treatment is carried out by the call

```
ier= self.ParLotMixte (J).
```

## 6.5 For what the keyword is used _F used in the command file?

Into the command file, a keyword factor is introduced by the character string _F. In fact this character string is a name of class which deals with creation in memory of the dictionary corresponding to the keyword factor starting from a description using the equal sign '=' and of the brackets rather than the two points ': ' and accodances that it would be necessary to use with a dictionary standard Python.

For example:
ELAS= **_F (** E **=** 2.1E11, NAKED **=** 0.3, ALPHA **=** 1.E-5, RHO **=** 8000. **)**
is equivalent to:
ELAS= {E: 2.1E11, NAKED: 0.3, ALPHA: 1.E-5, RHO: 8000.}

This presentation is more adapted to the wishes of the end-users and the tradition of the process control language of *Code_Aster*.

## 6.6 Where the interface getvxx command set find-such?

Methods getvxx belong to the class STAGE defined in the module B_ETAPE package Build.

# 7 Bibliography

[1] Introduction to Python, Mark Lutz & David Ascher, O' REILLY, Paris, 2001
[2] Python, Essential Reference, David M. Beazley, New Riders, 2001
[3] Python 2.1 Bible, Dave Brueck & Stephen Tanner, 2001