UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Primer

Jean-Paul Barthès

BP 349 COMPIÈGNE
Tel +33 3 44 23 44 66
Fax +33 3 44 23 44 77

Email: barthes@utc.fr

N219
May 2008

Warning

This document is a primer to help the inexperienced user with the **MOSS 7.xx** programming environment that uses the PDM4 model for representing knowledge, by building ontologies and associated knowledge bases. The document describes only elementary operations. It is an upgrade of the following manual:

- `UTC/GI/DI/N200 - MOSS 6 : A Primer`

More advanced users should refer to more detailed manuals, namely:

- `UTC/GI/DI/N204 - MOSS 6 : Syntax`

A current research version of MOSS 7 runs in a MacIntosh Common Lisp environment (MCL 5.1 or 5.2 for OSX) and in an Allegro Common Lisp environment (ACL 6.1 and 8.1 running under Windows XP).

Keywords

Object representation, object-oriented programming environment, ontology formalism

Revisions

Version	Date	Author	Remarks
1.0	Jun 06	Barthès	Initial issue
1.1	Oct 06	Barthès	Smoothing the vocabulary
1.2	Nov 07	Barthès	Removing irrelevant technical data
1.3	Jan 08	Barthès	Adding XP info
1.4	Jul 08	Barthès	Upgrade to MOSS v7

MOSS documents

Related documents

- UTC/GI/DI/N196L - PDM4
- UTC/GI/DI/N206L - MOSS 7 : User Interface (Macintosh)
- UTC/GI/DI/N218L - MOSS 7 : User Interface (Windows)
- UTC/GI/DI/N219L - MOSS 7 : Primer
- UTC/GI/DI/N220L - MOSS 7 : Syntax
- UTC/GI/DI/N221L - MOSS 7 : Advanced Programming
- UTC/GI/DI/N222L - MOSS 7 : Query System
- UTC/GI/DI/N223L - MOSS 7 : Kernel Methods
- UTC/GI/DI/N224L - MOSS 7 : Low Level Functions
- UTC/GI/DI/N225L - MOSS 7 : Dialogs
- UTC/GI/DI/N228L - MOSS 7 : Paths to Queries

Readers unfamiliar with MOSS should read first N196 and N219 (Primer), then N220 (Syntax), N218 (User Interface). N223 (Kernel) gives a list of available methods of general use when programming. N222 (Query) presents the query system and gives its syntax. For advanced programming N224 (Low level Functions) describes some useful MOSS functions. N209 (Dialogs) describes the natural language dialog mechanism.

Contents

1 Introduction

1.1 History

The MOSS system is a prototype object environment I designed and developed at UTC starting in 1986. As a research system its main goal was to study the various object mechanisms proposed in the literature, to evaluate their relevance and efficiency, so as to keep the best ones and to design new ones. MOSS draws on past experiences with semantic representations, and object oriented systems like VORAS. Objects use the PDM4 format (PDM stands for Property Driven Model).

With MOSS, I tried to develop an homogeneous environment while keeping a great flexibility, with a main goal of supporting multi-user interactive AI applications. The approach is built upon a simple but powerful object representation around which an object-oriented language, and a multi-user object manager for storing data permanently on secondary storage were developed. The first overall MOSS architecture is shown on Fig.1.

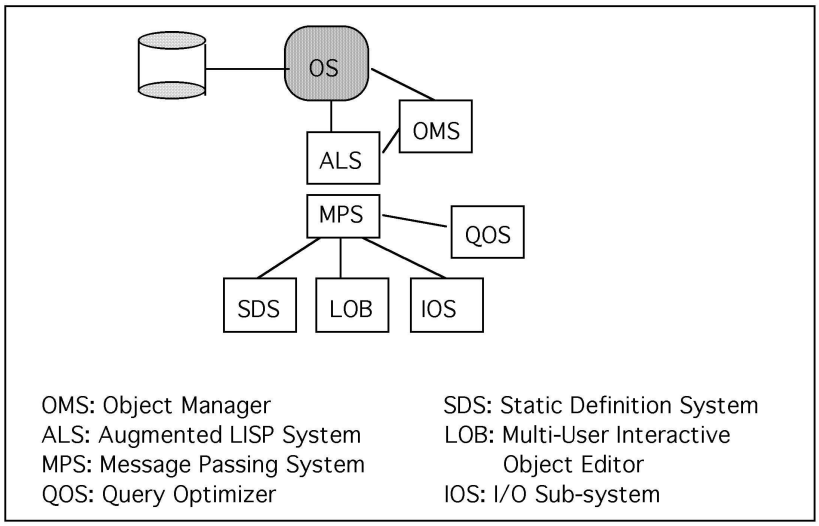

Figure 1: Overall architecture of the MOSS system

1.2 MOSS Today

MOSS can be used in standalone for representing knowledge through the development of ontologies and associated knowledge bases. MOSS can also be used in combination. For example, it has been integrated into the OMAS multi-agent platform for describing and implementing each agent knowledge. A more specialized version named SOL (Simple Ontology Language) can be used to translate an ontology into the OWL format.

The current MOSS architecture for version 7 contains only the object manager OMS and the message passing system MPS. The disc interface has been disabled, as the multi-user LOB system. The I/O system is that of Lisp. An interactive editor MES is now available.

1.3 Who should Read the Manual

This manual is intended for those who would like to get familiar with MOSS by exercising the MOSS environment. It describes how to use the MPS (Message Passing System) and the API (Programming Interface), which is good enough for prototyping small applications and catching a glimpse of what it is all about.

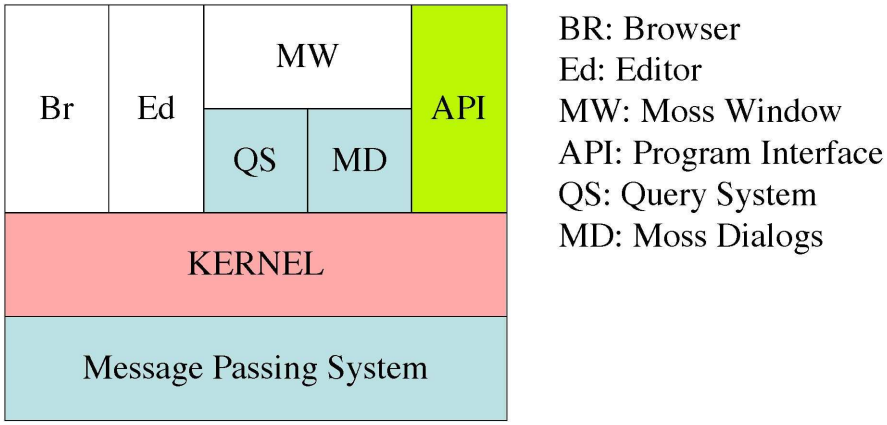

Figure 2: The MOSS components

2 How to Get Started

2.1 Macintosh Environment

MOSS 6 was tested in a MCL 5.1 and 5.2 MacIntosh OS X 10.4 PPC environment and in an ACL 6.1 and 8.1 environment running under Wirndows XP. The examples are taken from the MCL version.

You start the MCL LISP environment as usual and get the welcoming message:

```
Welcome to Macintosh Common Lisp Version 5.2!
?
```

Then, you need only load the `moss-load.lisp` file, using the "Load File" entry of the File menu. Some messages appear like:

```
#P"Odin:Users:barthes:MCL:OMAS-MOSS 7.0.1:MOSS:moss-load.lisp"
#P"Odin:Users:barthes:MCL:OMAS-MOSS 7.0.1:MOSS:MOSS-version-info.lisp"
;*** MOSS v7.0.1 - Macros loaded ***
;*** MOSS v7.0.1 - Service loaded ***
;*** MOSS v7.0.1 - Boot loaded ***
;*** MOSS v7.0.1 - Engine loaded ***
;*** MOSS v7.0.1 - Kernel loaded ***
;*** MOSS v7.0.1 - Definitions loaded ***
;*** MOSS v7.0.1 - Browser loaded ***
;*** Instances of sub-classes are also included, this can be changed
;     by setting the global variable *query-allow-sub-classes* to nil ***
;*** MOSS v7.0.1 - Query loaded ***
;*** MOSS v7.0.1 - Editor loaded ***
;*** MOSS v7.0.1 - MOSS window loaded ***
;*** MOSS v7.0.1 - dialog classes loaded ***
;*** MOSS - dialog engine loaded (control) ***
;*** MOSS v7.0.1 - Paths loaded ***
;*** MOSS v7.0.1 - Time loaded ***
;*** MOSS v7.0.1 - Patches 0 loaded ***
;*** MOSS v7.0.1 - Moss dialogs loaded ***
;*** MOSS v7.0.1 - online doc loaded (control) ***
;*** MOSS version 7.0.1 loaded ***
;*** Methods are cached: *cache-methods* set to T ***
;*** Fancy inheritance turned off *lexicographic-inheritance* set to T ***
?
```

The messages should not bother you.

Then the MOSS window shows up (Fig.3).

Close the MOSS window and keep the Lisp Listener. You are then ready to work.

2.2 Windows XP Environment

In the Microsoft XP environment the MOSS window looks like Fig.4.

Close the MOSS window and keep the Lisp Listener (Debug Window, CG-USER package). You are then ready to work[1].

[1] Alternatively, it is possible to work in the MOSS Window using the Data mode.

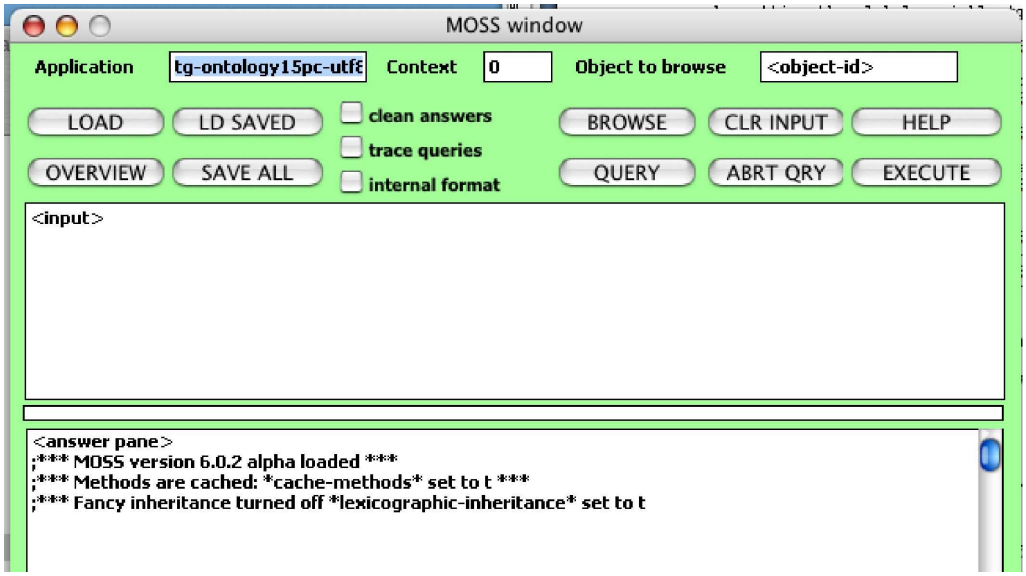

Figure 3: MCL MOSS window

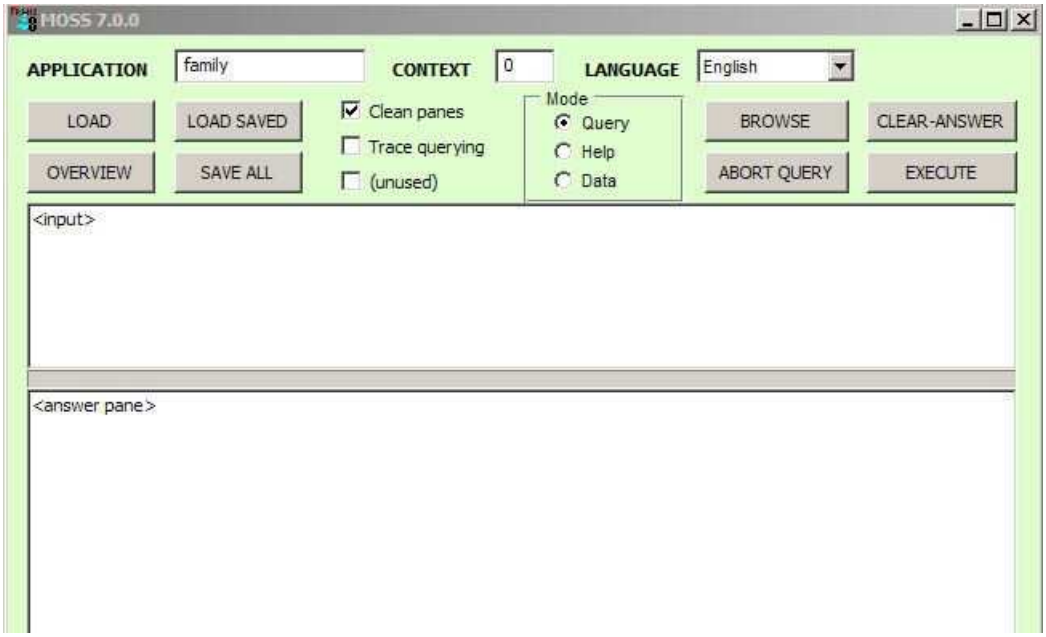

Figure 4: Microsoft XP MOSS window

3 Working with MOSS

Working with MOSS consists in creating objects and giving them behaviors. MOSS being a representation language allows modeling ontologies. Thus, one tends to use concepts rather then classes and individual concepts rather then instances.

Using concepts and individuals rather than classes and instances also avoids possible conflicts with CLOS (the internal Common Lisp Object System).

Thus, we will use mostly the terms concept and individual rather than class and instance. Examples are given in the context of MCL. They are identical in the ACL context except that the prompt is not a question mark but something like:

```
CG-USER(2):
```

meaning that you are in the CG-USER package and this is the second expression that you want the system to evaluate.

3.1 Creating a Concept

Creating a concept is simple and can be done with the `defconcept` macro as follows:

```
? (defconcept "Person")
$E-PERSON
```

This creates an object whose concept name is PERSON in your environment (usually `CG-USER` or `COMMON-LISP-USER`). The macro returns `$E-PERSON`, the internal name or identifier (**object-id**) referring to the concept you have just created. This identifier will be used as a radix when you create individuals, i.e., the individual internal names will be `$E-PERSON.1`, `$E-PERSON.2`, `$E-PERSON.3`, and so on. A concept has an associated internal counter for building such names that starts counting at 1.

3.1.1 Internal and external names: object-ids and entry points

Objects in MOSS are unique entities. As such they exist, even if nothing is known about them, which is quite different from the relational model for example, in which an object (tuple) cannot exist if one knows nothing about it. The internal or system name for an object is called its identifier or **object-id**. An object identifier is unique and designates the corresponding object directly. However, such names are often meaningless and difficult to use. Furthermore, they are assigned by the system dynamically and depend on the order in which defining files are loaded. Consequently, as a facility to the user, external names can be specified. External names are called **entry points**, and are in practice indexes for accessing corresponding objects directly [2]. They are somewhat analogous to keys in traditional databases; but, contrary to keys, *they need not be unique*. Some of the defining macros provide external names automatically in a normalized way. Examples will be given as we go along. Recovering an internal name from an external entry point is sometimes not easy, due to possible ambiguities. Furthermore, for certain kinds of objects (e.g. concept names), external names must be unique within the application name space.

In the example, `$E-PERSON`, is the internal name or object-id of the concept PERSON. The concept PERSON is a first class object [3].

The symbol `PERSON` on the other hand is defined as the external name or entry point for the concept. *It is a reserved name and should not be used as the name of a of a special (global) variable, otherwise the structure of your application will break.*

[2] In practice the object-id is implemented as a LISP symbol, the value of which is an a-list representing the object itself. Hence one can view the internal structure of a MOSS object simply by typing its object-id.

```
? $E-PERSON
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
(MOSS::$ENAM (0 (:EN "PERSON"))) (MOSS::$RDX (0 $E-PERSON))
(MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-PERSON.CTR)))
```

[3] Concepts, properties (attributes and relations), entry points are all first class objects

3.1.2 Cases

Because MOSS is defined in a traditional Lisp environment, cases are not important when using symbols and are not retained by the system, except within strings. Thus:

```
? (defconcept PERSON)
? (defconcept Person)
? (defconcept persSON)
? (defconcept PeRsOn)
```

will give the same result and the name of the concept will be printed as PERSON, using capital letters. If one wants to retain the case[4], the definition using the string will keep the case when printing the external name of the concepts. It is advised to use the string version of the definition starting concept names with a capital letter, which is a somewhat standard approach for class names in OO languages.

A composed class name like "Capital Town" will yield the normalized external name CAPITAL-TOWN. Note the hyphenation mark that is not an underscore.

3.1.3 Documentation

One can also attach some documentation information to a concept by using the option `:doc`, e.g.

```
? (defconcept "Person"
  (:doc "This describes a PERSON, bla, bla,..."))
```

3.1.4 Redefining Concepts

Note also that if you want to redefine the concept PERSON using the defconcept macro, then MOSS issues an error. Indeed it is an error to redefine a concept. Thus, if you want to test a different definition without reinitializing your session, you have to execute it with a different concept name.

From now on, you could create individuals of the concept PERSON using this model. However, since the concept has no attributes, this is not very interesting. Next section indicates how to add properties to concepts.

3.2 Creating Concept Properties

MOSS objects have two kinds of properties: immediate values that qualify the object, and links to other objects. The first kind is called an **attribute**, the second kind is called a relationship or **relation**. Many people use the terms attribute, property, and relation to mean quite different things. We will use them strictly as shown in the next paragraphs.

3.2.1 Attributes

Within the `defconcept` macro, you can specify that a PERSON has a name, and a first name with the `:att` (attribute) option.

```
? (defconcept "Person"
  (:att "name" (:min 1)(:max 3)(:entry))
(:att "first name"))
```

Because *attributes can be multivalued*, you may want to specify the minimal and the maximal cardinality. In the above example a PERSON will have at least 1 name and at most 3. The `:entry` option means that you want to be able to get access to the persons (instances) by giving their name (in case of multiple names, any one of them will do). Hence the value of the name property for an instance

[4]This is only useful when printing

will generate one or more **entry point(s)**.
For the first name, no cardinality restriction is imposed: a person can have zero or an unlimited number of first names.
Notice also that no specific type has been imposed on the attributes.

Creating Attributes Directly It is important to realize that *properties can be created independently from any concept.* For example:

```
? (defattribute "sex")
$T-SEX
```

defines a single-valued generic property [5]. The system returns the internal name, e.g. $T-SEX, of the property.

The attribute can however be attached to a concept as follows:

```
? (defattribute "sex" (:min 1) (:max 1) (:concept PERSON))
$T-PERSON-SEX
```

Note that the internal name is tagged with the concept name. The attribute is defined as a local attribute related to the concept PERSON, rather than a global attribute that exists independently from any concept. The external name of the attribute in both cases is constructed by adding the prefix HAS: `HAS-NAME`.

3.2.2 Relations

Relations, also known as links, can be defined likewise, using the `defrelation` macro:

```
? (defrelation "brother" PERSON PERSON)
$S-PERSON-BROTHER
```

meaning that a person may have another person as a brother. The actual meaning of the notation is more like that of an integrity constraint: PERSON appears twice: the first time to specify the concept it is a property of (sometimes called the domain e.g. for a function), the second time to specify the type of the "successor" (sometimes called its range).

Creating Relations Directly It is important to realize that *properties can be created independently from any concept.* For example:

```
? (defrelation "neighbor" :any :any)
$T-NEIGHBOR
```

creates a relation object meaning that any object can be the neighbor of any other object. However, in general relations are defined between individuals (i.e. instances of concepts).

In order to beef up our example we now define additional concepts:

```
? (defconcept "Department"  (:att "name"))
$E-DEPARTMENT

? (defconcept "Course"
    (:att "code" (:min 1)(:max 3)(:entry))
    (:rel "department" DEPARTMENT))
$E-COURSE
```

[5] `defattribute` does not require that the created property be attached to a specific concept. It is thus possible to create properties independently (like mixins in the flavor system).

Note that, while defining the concept COURSE, we used DEPARTMENT with two meanings in the last line: one for a relation, the second one for the concept name. This situation happens quite frequently, hence MOSS creates distinct external names for concepts and for properties, removing the ambiguity. Here `DEPARTMENT` will be the external name of the concept, and `HAS-DEPARTMENT` the external name of the property.

3.3 Defining Sub-Concepts

Like in other representation languages, one can define sub-concepts as refinements of concepts. For example:

```
? (defconcept "Student"
      (:is-a "Person")
   (:rel "course" "course"))
$E-STUDENT
```

or

```
? (defconcept "Teacher"
   (:is-a "person")
  (:rel "lecture" "course")
  (:rel "department" "department"))
$E-TEACHER
```

As expected, students or teachers will inherit the properties of persons.

3.4 Creating Individual Concepts

Concepts are used to create individual concepts that will respect the structure specified by the concept. To create individuals one uses the `defindividual` macro.

```
? (defindividual "person")
```

will return the internal name of the instance, e.g. the first time you will get

```
$E-PERSON.1
```

To store the internal name in a variable, one can do a regular assignment or else use the `:var` keyword option, i.e.

```
? (setq _p1 (defindividual PERSON))
or
? (defindividual PERSON (:var _p1))
```

Note that by convention a variable name should start with an underscore, e.g., `_P1`. Note also that we can define individuals by referring to the class by means of the corresponding string[6] ("person") or by using the internal name (PERSON).

Of course, it is interesting to assign values to properties of individuals. This is done by using the external names of the properties in the `defindividual` macro.

```
? (defindividual PERSON
(:var _p1)
(has-name "Dupond" "Durand")
(has-first-name "Jean")
(has-sex "M"))
```

[6]The case here is not important. We can use "name" or "Name" or "NAME".

or

```
? (defindividual PERSON
(:var _p2)
("name" "Dubois" "Dupond")
("sex" "F")
("brother" _p1))
```

Here the internal object-ids are kept in the variables `_p1` and `_p2`. Property names can be specified by their definitional name prefixed with HAS-. Remember, this is to distinguish attributes from concepts: `HAS-DEPARTMENT` indicates a property and `DEPARTMENT` a concept name. Values can be multiple like the 2 names: Dupond/Durand or Dubois/Dupond.

Note that the `HAS-XXX` symbols can be replaced with the string `"XXX"` as shown in the second definition.

3.5 Printing Information

While creating objects it may be interesting to print their content. This is done simply by sending a message to whatever object must be printed in the following way:

```
? (send '$E-person '=print-self)
```

or

```
? (send _person '=print-self)
```

Notice the "=" sign in front of the `=print-self` method[7]. We obtain:

```
----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 ATTRIBUTE : name/Person, first name/Person
 RELATION : brother/Person
 COUNTER: 4
-----
:DONE
```

Note that the symbol `_person` (variable) holds the internal object-id of the concept PERSON and was defined by MOSS automatically. However, if we write

```
? (send PERSON '=print-self)
```

then we get an error. For printing the content of person `_p1` we write

```
? (send _p1 '=print-self)
```

getting

```
----- $E-PERSON.3
 name: Dupond, Durand
 first name: Jean
 sex: M
-----
:done
```

[7]In some systems method names start with ":". In SMALLTALK they end with ":". In MOSS I chose to use the "=" sign to distinguish methods from other names (e.g. entry point names), and because the ":" sign in LISP is reserved for keywords and packages.

Similarly, for printing the content of the relation brother

```
? (send _has-brother '=print-self)
```

we get (generic brother relation)

```
----- $S-BROTHER
 INVERSE: (EN IS-BROTHER-OF)
 SUCCESSOR: UNIVERSAL-CLASS
 PROPERTY-NAME: brother
-----
:done
```

etc.

3.6 Object Behavior

In the previous sections, we used only definitional macros and predefined methods. The interesting part in an object oriented programming language is to define specific behaviors for various objects. MOSS uses three types of methods as presented in the next paragraphs.

3.6.1 General Behavior

Assume that we'd like the method `=print-self` to print first names and names of persons. Instead of a tabular format, then we can define a specific `=print-self` method for persons using the `definstmethod` macro, as:

```
? (definstmethod =print-self PERSON ()
"Prints a list of first names and names"
  (format t "~&~{~A~^-~} ~{~A~^-~}"
   (HAS-FIRST-NAME) (HAS-NAME)))
```

where `(HAS-FIRST-NAME)` is a short-hand notation for

```
(send *self* '=get 'HAS-FIRST-NAME)
```

where `*self*` is bound to the object that just received the message;

Writing the above definition creates an object method that is attached to the concept PERSON, but applies to its individual instances. Hence it is called an **instance-method**. It is used as follows:

```
? (send _p1 '=print-self)
Jean Dupond-Durand
NIL
```

Using the macro `definstmethod`, one can create any kind of instance method.

Note also that by convention methods begin with an = sign. This is not required, however it is good practice and helps separating method names from other names like variable names, property names, concept names, etc. MOSS uses this convention for predefined method names. Method names should not start with the ":" character, because Lisp uses it to indicate a keyword. Keywords cannot have values, hence cannot be used as entry points for methods.

3.6.2 Exceptions, Own Methods

Let us assume that we want to print Ms. Dubois differently by omitting her first name and inserting Ms for example. We can define a specific method, called **own-method** that we attach to the individual instance of person _p2. We do that using the special macro `defownmethod`:

```
? (defownmethod =print-self _p2 ()
"Prints the name with Ms. in front of it."
   (format t "~&Ms ~{~A~^-~} ~{~A~^-~}" (HAS-FIRST-NAME)(HAS-NAME)))
```

Note that the concept name argument has been replaced with a variable containing the object-id.

The specific method will be called in due time, i.e. when we state:

```
? (send _p2 '=print-self)
Ms Dubois-Dupond
NIL
```

which yields a different result from

```
? (send _p1 '=print-self)
Jean Dupond-Durand
NIL
```

The NIL value returned in each case is the value returned by the printing function. Traditionally Lisp printing functions always return nil.

3.6.3 Universal Methods

Some methods like `=get` or `=print-self` are predefined and apply to all objects. They are called **universal methods** and are not attached to any object in particular. They are used by default as explained in the next section.

3.6.4 Inheritance Mechanism

When an object receives a message, MOSS looks first for a local or own-method. If one is there, then it applies it. Otherwise, it looks for an instance method at the concept level. If none is found, then it tries to inherit some from super-concepts using a mechanism described in the section about advanced features. If none can be inherited, then it looks for a universal method. If none exists, then it quits.

3.7 Creating Orphans

Because we wanted to use MOSS in robotics we introduced the possibility of having *objects not specified by any concept* or **orphans**. An orphan is an object that does not have a concept but that may have any predefined property. It can be built using the `defobject` macro as follows.

```
? (defobject ("name" "Joe")(:var _ob1))
$0-0
```

defines an orphan with name Joe; `_ob1` is a variable containing its internal object id.

Of course we can attach (own-)methods to such an object to specify its behavior.

```
?(defownmethod <method-name> ob1 <args> <doc><body>)
```

Prototyping Furthermore we can specify that an orphan behaves as another known object by using the keyword `:is-a`. E.g.,

```
? (defobject (:var _p4)("NAME" "George")(:is-a _p2))
```

We don't know exactly what George may be, except that it behaves like `_p2`. Methods will be inherited along the is-a link. `_p2` is thus a **prototype** for the object we just defined. Thus, if we write

```
? (send _p4 '=print-self)
Ms   George
NIL
```

We notice that the special method defined previously for the person `_p2` has been used for the orphan `_p4` that we just created.

4 Programming

4.1 Message Passing

As with any object-oriented programming language one builds an application by defining objects and associated methods implementing the object behavior. Programming is done by sending messages. The main function to use is **send** as shown previously. A **broadcast** function is also available for sending the same message to a set of objects. However, it uses the Lisp `mapcar` primitive and thus *is not a parallel primitive.*

4.2 Service Functions

The programmer can use any of the Common LISP primitives in the methods. In addition a number of service functions are available. Some of them are very general and should actually be LISP primitives, others are related to the structure of the objects (PDM). They are efficient, but should be used sparingly and are mentioned in a separate document.

4.3 Predefined Methods

There are several kinds of predefined methods. Two kinds are of interest here:

basic methods like `=get-id`, `=get`, etc., that temper with the physical structure of the object representation. Such methods, except for `=get`, should not be used directly, because they can destroy the overall PDM structure (logical consistency).

PDM or kernel methods are defined at a higher level. They include `=add`, `=delete`, `=print-value`, etc.; such methods are referenced in the MOSS 6 Kernel Manual and should be used whenever possible. They respect the PDM structure (logical integrity constraints). In addition `=check` methods are provided to verify if an object still follows the PDM format.

Among the universal methods, some are of interest:

=get-properties returns the list of all properties as specified by the object concept, including the inherited ones. The resulting list looks strange; it is the list of internal ids of the properties.

```
? (send _p2 '=get-properties)
($T-PERSON-NAME $T-PERSON-FIRST-NAME $T-PERSON-SEX $S-PERSON-BROTHER)
```

=print-methods prints all instance methods associated with a concept. For the concept person, we defined only one method: `=print-self`.

```
? (send _person '=print-methods)
LOCAL INSTANCE METHODS
======================

=PRINT-SELF
Prints a list of first names and names
:DONE
```

=what? prints a summary of the type of object which received the message. If it is a concept, then prints the contents of the DOCUMENTATION field if it was filled (`:doc` option).

```
? (send _person '=what?)

The object is an instance of CONCEPT
*Sorry, no specific documentation available*
```

```
T
? (send _p4 '=what?)

The object is an orphan (i.e. has no class).
... has prototypes (depth first)
$E-PERSON.3
*Sorry, no specific documentation available*
T
```

4.4 Writing a Method

4.4.1 Global Variables

Methods are written in Common LISP. Several global variables are accessible when the method is executing, i.e. they can be used within the method.

self contains the internal name of the object that just received the message.

args contains the list of arguments of incoming message.

sender contains the identity of the sender. By default the initial sender is *user*.

After a method is executed it returns control to the sender, eventually with a value. The returned values are available in the ***answer*** variable.

4.4.2 Redefining a method

A method is redefined by re-executing the `definstmethod` or `defownmethod` macro. This automatically cleans the environment.

4.5 Debugging Helps

When writing new methods, it is sometimes of interest to visualize what is going on by monitoring the traffic of messages. MOSS allows doing several things:

print all traffic : all messages and answers are printed in an indented fashion,

```
(trace-message) or (ton)
(untrace-message) or (toff)
```

trace specific methods , e.g.

```
(trace-method <method-id>)
(untrace-method <method-id>)
```

monitor the traffic to specific objects

```
(trace-object <object-id>)
(untrace-object <object-id>)
```

Example of tracing messages :
When tracing messages, we monitor all messages that are exchanged in the system.

```
? (trace-message)
T
? (send _p1 '=print-self)
1 from: *USER* to: $E-PERSON.2, request: =PRINT-SELF arg-list: NIL
Jean Dupond-Durand
 1 $E-PERSON.2 for: =PRINT-SELF, to: *USER* returns: NIL
NIL
```

Example of tracing objects :
A more complex trace is the following:

```
? (send _ob1 '=print-self)
 1 from: NIL to: $0-0, request: =PRINT-SELF arg-list: NIL
  2 from: $0-0 to: $0-0, request: =GET-PROPERTIES arg-list: NIL
  2 $0-0 for: =GET-PROPERTIES, to: $0-0 returns: (MOSS::$TYPE MOSS::$ID $T-NAME)
----- $0-0
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$TYPE)
  2 $0-0 for: =GET-ID, to: $0-0 returns: (MOSS::*NONE*)
  2 from: $0-0 to: $TYPE, request: =PRINT-VALUE arg-list: ((MOSS::*NONE*) :STREAM T)
 TYPE: *NONE*
  2 $0-0 for: =PRINT-VALUE, to: $TYPE returns: NIL
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$ID)
  2 $0-0 for: =GET-ID, to: $0-0 returns: ($0-0)
  2 from: $0-0 to: $ID, request: =PRINT-VALUE arg-list: (($0-0) :STREAM T)
 IDENTIFIER: $0-0
  2 $0-0 for: =PRINT-VALUE, to: $ID returns: NIL
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: ($T-NAME)
  2 $0-0 for: =GET-ID, to: $0-0 returns: ("Joe")
  2 from: $0-0 to: $T-NAME, request: =PRINT-VALUE arg-list: ((#) :STREAM T)
 name: Joe
  2 $0-0 for: =PRINT-VALUE, to: $T-NAME returns: NIL
-----
 1 NIL for: =PRINT-SELF, to: $0-0 returns: :DONE
:DONE
```

When tracing an object, we only monitor messages addressed to this object. Notice the difference with the previous trace.

```
? (trace-object _ob1)
T
? (send _ob1 '=print-self)
 1 from: NIL to: $0-0, request: =PRINT-SELF arg-list: NIL
  2 from: $0-0 to: $0-0, request: =GET-PROPERTIES arg-list: NIL
  2 $0-0 for: =GET-PROPERTIES, to: $0-0 returns: (MOSS::$TYPE MOSS::$ID $T-NAME)
----- $0-0
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$TYPE)
  2 $0-0 for: =GET-ID, to: $0-0 returns: (MOSS::*NONE*)
  2 from: $0-0 to: $TYPE, request: =PRINT-VALUE arg-list: ((MOSS::*NONE*) :STREAM T)
 TYPE: *NONE*
  2 $0-0 for: =PRINT-VALUE, to: $TYPE returns: NIL
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$ID)
  2 $0-0 for: =GET-ID, to: $0-0 returns: ($0-0)
```

```
  2 from: $0-0 to: $ID, request: =PRINT-VALUE arg-list: (($0-0) :STREAM T)
 IDENTIFIER: $0-0
  2 $0-0 for: =PRINT-VALUE, to: $ID returns: NIL
  2 from: $0-0 to: $0-0, request: =GET-ID arg-list: ($T-NAME)
  2 $0-0 for: =GET-ID, to: $0-0 returns: ("Joe")
  2 from: $0-0 to: $T-NAME, request: =PRINT-VALUE arg-list: ((#) :STREAM T)
 name: Joe
  2 $0-0 for: =PRINT-VALUE, to: $T-NAME returns: NIL
-----
 1 NIL for: =PRINT-SELF, to: $0-0 returns: :DONE
:DONE
```

Example of tracing methods :
When tracing a method, we only monitor messages that refer to the method name. Notice the difference with the previous traces.

```
? (trace-method '=get-id)
T
? (send _ob1 '=print-self)
----- $0-0
 1 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$TYPE)
 1 $0-0 for: =GET-ID, to: $0-0 returns: (MOSS::*NONE*)
 TYPE: *NONE*
 1 from: $0-0 to: $0-0, request: =GET-ID arg-list: (MOSS::$ID)
 1 $0-0 for: =GET-ID, to: $0-0 returns: ($0-0)
 IDENTIFIER: $0-0
 1 from: $0-0 to: $0-0, request: =GET-ID arg-list: ($T-NAME)
 1 $0-0 for: =GET-ID, to: $0-0 returns: ("Joe")
 name: Joe
```

4.6 Defining Test Files

When trying to get acquainted with MOSS it is a good idea to group all definitions in two separate files, one for defining the object static structures, the second one for defining their behavior (methods). They could be called e.g. `mytest-concepts.lisp`, and `mytest-methods.lisp`. They can be loaded after the MOSS environment. Thus, one does not need to retype everything all the time.

Note however that concepts can be loaded only once. Indeed, as a safety feature, MOSS refuses to redefine concepts. Hence when modifying the definition file, the environment has to be reloaded. On the other hand, methods can be redefined at any time, and one can use a separate file so that the file can be edited and reloaded any number of times. When the file becomes big, one can create a temporary file for debugging a particular method, and, once the method works satisfactorily, copy it into the larger file.

5 Advanced Features

They are described in the Memo

UTC/GI/DI/N 201 - MOSS 6: Programming, Advanced Features