

# OPENMUSIC: DESIGN AND IMPLEMENTATION ASPECTS OF A VISUAL PROGRAMMING LANGUAGE

Carlos Agon, Jean Bresson, Gérard Assayag  
IRCAM - CNRS UMR STMS  
Music Representations Research Group

## 1. INTRODUCTION

OpenMusic (OM) is a computer-aided composition environment developed at Ircam since the end of the 90s [1] [6] [7]. It is a complete functional programming language, which extends Common Lisp with a visual specification.

Thanks to graphical tools and protocols, the user / programmer can create functions and programs using arithmetic or logic operations, or make use of other programming concepts like functional abstraction and application, iteration or recursion. As we shall demonstrate in this article, he/she can also benefit from the powerful object protocol of CLOS (Common Lisp Object System [10]).

The musical issues and compositional relevance of this environment, widely discussed in various related publications, are voluntarily left aside; in the present paper we shall rather go through different aspects of the programming language design and features.

## 2. LANGUAGE ARCHITECTURE

The elements of the visual language can be divided in two categories: the *meta-objects* are the “traditional” language primitives (functions, programs, classes, instances, types, etc.) and the *visual components* (or *visual meta-objects*) constitute the visual part of the language and provide its graphical representation and user interactions.

### 2.1. Meta-Objects

In CLOS, the classes, generic functions, methods, and other element of the language are meta-object classes, instances of the class *standard-class* [11]. These classes (respectively *standard-class*, *standard-generic-function*, *standard-method*, etc.) can therefore be subclassed and extended by new classes. This what is systematically done in OM for defining the language meta-objects (e.g. *OMClass*, *OMGenericFunction*, *OMMethod*, etc.) *OMClass*, for instance, is defined as a subclass of *standard-class* as follows:

```
(defclass omclass (standard-class) ())  
  
> #<standard-class omclass>
```

In order to make new classes instances of *OMClass* instead of *standard-class*, one can then use the *:metaclass* *initarg* in class definition as follows:

```
(defclass my-class ()  
  ((slot1 :initarg :slot1 :accessor slot1)  
   (slot2 :initarg :slot2 :accessor slot2))  
  (:metaclass omclass))  
  
> #<omclass my-class>
```

That way, it is possible to extend *standard-class* in order to set particular behaviours and properties related to specific aspects of the visual language (icons specification, documentation, persistence, behaviours in the graphical user interface, etc.) Every class defined as an *OMClass* instead of *standard-class* will therefore possibly be handled in the visual part of the language.

Specific protocols are established for the creation of OM meta-objects. The macros *defclass!* and *defmethod!* expand as calls to *defclass* et *defmethod* that specify the appropriate metaclass and allow for the setting of the particular attributes of the corresponding meta-object. For instance, *OMClass* has a slot called *icon* containing an icon ID converted as a picture icon when it is represented in the visual language. An *OMClass* can therefore be created directly as follows:

```
(defclass! my-class2 () ()  
  (:icon 21))  
  
> #<omclass my-class2>
```

The same principle applies for generic functions and methods. In the following example, the keywords *icon*, *initvals* and *indocs* allow to specify the attributes of the *OMMethod* instance which is created, that will be used to determine respectively the icon for the graphical representation of this methods, the default values, and a documentation for each of its arguments:

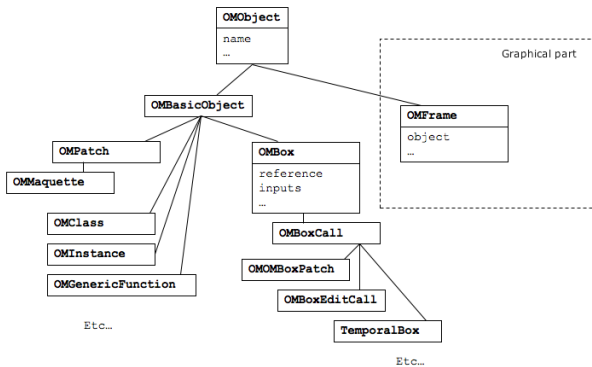
```
(defmethod! my-method (arg1 arg2)  
  :icon 123  
  :initvals '(0 0)  
  :indocs '("argument1" "argument2")  
  (+ arg1 arg2))  
  
> #<ommethod my-method>
```

### 2.2. Visual Components

The visual aspects of the language principally manifest themselves through the *boxes* (class *OMBox*) and the *editors* (class *Editor*). These are however still “non-graphic” objects in the environment: a box is a “visual meta-object”, i.e. a syntactic specification of the visual language. It forms part of the “visual MOP” (VMOP), as a special

class of "environment" meta-object. Visual components generally refer to other elements of the language (e.g. functions, classes or programs) and represent them in the user interface. The actual graphic level (i.e. the GUI elements) encapsulate these non-graphic VMOP components via the class *OMFrame* and its subclasses (*BoxFrame*, *InputFrame*, *EditorFrame*, etc.)

The visual aspects are thus not simple interfaces on the language but a set of meta-object properties and graphical components taking part in the language specification. Figure 1 shows the class architecture corresponding to the main elements of the language.

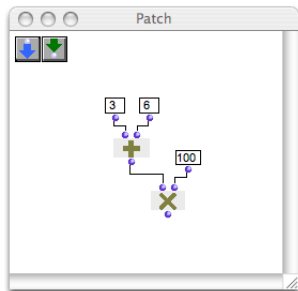


**Figure 1.** Simplified architecture of the main classes in the OM visual programming language.

### 3. VISUAL PROGRAMMING

Two main visual components were cited above: boxes and editors. The boxes are represented by frames surrounded by inputs and outputs, according to the object they refer to. They are possibly connected with one another within a visual program via these inputs and outputs. Most of the objects are also associated to an editor, which allow for their “manual” building and edition.

*Patches* represent visual programs and are the main entry-point in the OM programming framework (class *OMPatch* in Figure 1). They are associated to an editor in which the user / programmer creates functional units represented by graphical boxes. Figure 2 shows a patch which performs simple arithmetic operations on numbers.



**Figure 2.** A patch implementing the operation  $(3 + 6) \times 100$ .

Each box refers to a functional object: in the example of Figure 2, the boxes refer to functions (+ and  $\times$ ) or to constant values (3, 6 and 100). The functions can be classical Common Lisp functions (*standard-function*, *standard-method*, etc.) or OM functions (*OMMethod*). They can be built-in functions, included in the Lisp image, or user defined functions created or loaded dynamically while using OM.

As mentioned above, a box has a variable number of inlets and outlets so that it can be connected to other boxes. Inlets are visible at the top of the boxes, and outlets at the bottom. A set of connected boxes therefore constitutes an acyclic graph that corresponds to a functional expression. The patch in Figure 2 corresponds to the following Lisp expression:

```
(* (+ 3 6) 100)
```

The graph defined in a patch can be evaluated at any point, as a Lisp expression does. The evaluation of a box, triggered by a user action, is a call to the referred function. The arguments of this function call are the results of the evaluation of the boxes connected to the various inlets of this box. A recursive sequence of evaluations therefore occurs in order to reduce the Lisp expression according to the functional composition defined by the connections, which corresponds to the execution of the program.

The evaluation of the box  $\times$  in Figure 2 starts this reduction process: the result of box 100 (i.e. the value itself) is multiplied to that of box +, and so forth. The Lisp listener prints the final result:

```
> 900
```

The function responsible for the evaluation of the boxes is the method *omng-box-value*:

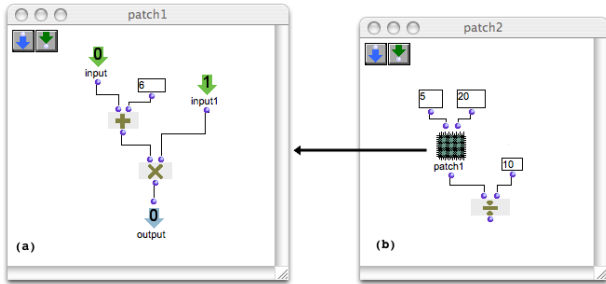
```
; Eval the output indexed by 'numout'
; for the box 'self'
(defmethod omNG-box-value ((self OMBoxCall)
  &optional (numout 0))
  (let ((args (mapcar #'(lambda (input)
    (omNG-box-value input)
    (inputs self))))
    (nth numout (multiple-value-list
    (apply (reference self) args)))
  ))
```

This method is specialized for the different types of boxes (subclasses of *OMBox*). Here, *OMBoxCall* is a box that refers to a function. Note that *omng-box-value* called on a box input reports the call on the box connected to this input, following the links established by the connections in the patch.

#### 3.1. Functional Abstraction

Functional abstraction basically consists in making some elements of a program become variables. Inputs and outputs, also represented by boxes, can be introduced in an OM patch. They will represent these possible variables in the program defined in this patch.

Starting from the example in Figure 2, it is possible to create the function  $f(x, y) = (x + 6) \times y$  by adding two inputs and an output connected to the program as shows *patch1* in Figure 3 (a).



**Figure 3.** (a) Definition et (b) application of a function. Abstraction is carried out by making variable some elements of the program.

Then *patch1* now corresponds to a function definition. The corresponding Lisp expression would be:

```
(lambda (x y) (* (+ x 6) y))
```

As a function definition, this expression can also be expressed as follows:

```
(defun myfunction (x y) (* (+ x 6) y))
```

*Patch1* can then be used in another patch, and is then considered as a function with 2 arguments and 1 output value, as in *patch2* on Figure 3 (b). In this patch can be set the values of the abstraction variables (functional application).

In this example, we see a new type of box, which refers to the patch (the box labelled *patch1*). Its evaluation corresponds to the Lisp call:

```
(myfunction 5 20)
```

The new program (*patch2*) therefore corresponds to the expression:

```
(/ (myfunction 5 20) 10)
```

In these abstraction/application mechanisms, the patch is converted into a Lisp function. A function called *compile-patch* carries out this conversion by a recursive call to a code-generating method (called *gen-code*) on the functional graph that constitutes the patch. During this recursive call to *gen-code*, each box generates the Lisp code corresponding to its referring object. The newly generated Lisp expression is then compiled and the resulting function is attached to the patch (the class *OMPatch* has a dedicated slot called *code*).

The evaluation of a box referring to this patch thus consists in the application of the values connected to its inputs to the compiled function:

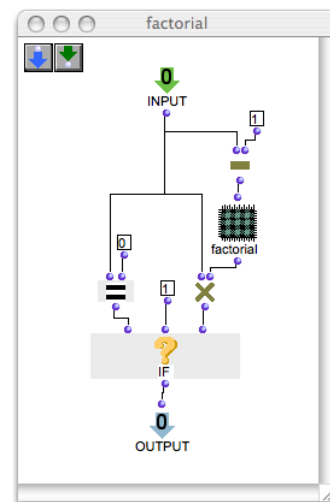
```
; Eval the output indexed by 'numout'
; for the box 'self'
(defmethod omNG-box-value ((self OMPatch)
                           &optional (numout 0))
```

```
(let ((args (mapcar #'(lambda (input)
                       (omNG-box-value input)
                       (inputs self))))
      (unless (compiled? (reference self))
              (compile-patch (reference self)))
      (nth numout (multiple-value-list
                    (apply (code (reference self)) args)))
      ))
```

Within a patch editor, the program can thus be modified and partially executed. From the outside, however, it is a box corresponding to an abstract function. This function can therefore be used later on in different contexts and purposes. The multiple occurrences of a patch box in other patches will all refer to the same function.

Abstractions can also be used in their own definitions, hence implementing the notion of recursion. Figure 4 shows a patch corresponding to the recursive function “factorial”:

```
(defun factorial (x)
  (if (= x 0) 1
      (* x (factorial (- x 1)))))
```

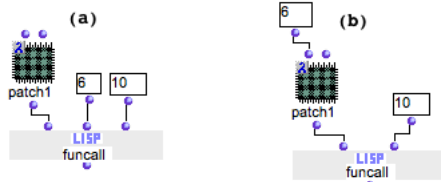


**Figure 4.** The recursive factorial function in OM.

### 3.2. Lambda Functions

In functional languages data and functions are equally considered as “first-class citizens”. A Lisp function can thus be considered as data and inspected or constructed in the calculus. This allows for the creation of “higher-level functions”, i.e. functions that take other functions as arguments, or producing functions as output values.

OM boxes can be set to a “lambda” state so as to return not the result of its reference’s functional application, but its reference as a function object. When a patch box is in mode “lambda”, a small *lambda* icon is displayed on it. A box like *patch1* in Figure 3 (b), for example, if it is set to this lambda mode, will not return a value anymore (22, in this example), but the functional definition  $(\lambda (x y) (* (+ x 6) y))$  which will be used and eventually called as such in the continuation of the program execution (see Figure 5).



**Figure 5.** (a) Creation of a lambda form in a visual program. The patch box *patch1* is in mode “lambda” and returns a function, called using the *funcall* box and arguments 6 and 10. (b) Curryfication: the previous function is converted to a function of one single argument by explicitly setting one of the input values in the lambda form.

The curryfication (i.e. transformation of a function of  $n$  arguments into a function of  $n - 1$  arguments) can also be carried out using the lambda mode, by connecting explicitly one value to some inputs of the patch box as shown in Figure 5 (b).

### 3.3. Local Functions

Complementarily to the abstraction mechanism detailed above, it is possible to create sub-programs (or sub-patches) which actually are local functions, defined only within the local context of a patch. These sub-patches are graphically differentiated with the color of their referring boxes.

For instance in the example of Figure 3 *myfunction* is defined in the environment, which would correspond to the following expressions:

```
; patch1
(defun myfunction (x) (* (+ x 6) 100))

; patch2
(defun myprogram (x) (/ (myfunction x) 10))
```

with a local function, we have the possibility to obtain something similar to:

```
(defun myprogram (x)
  (flet ((myfunction (x) (* (+ x 6) 100)))
    (/ (funcall myfunction x) 10)))
```

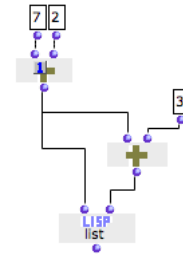
In this case, *myfunction* does not exist outside *myprogram*. As a consequence while all boxes referring to an abstraction point to a unique patch, local functions can be duplicated and edited independently in each of their occurrences. It is also possible to detach patches from their abstraction (by creating a local copy of the function) or conversely to define a global abstraction from a local function.

### 3.4. Local Variables

In order to simulate the Lisp *let* statement and allow for the creation of local variables, it is possible to set the function call boxes to a third state called *ev-once*. In this mode the box is evaluated only once during an evaluation process.

The example in Figure 6 shows a patch in which the topmost box  $+$  would be called various times, each time producing the same result :

```
(list (+ 7 2) (+ (+ 7 2) 3))
```



**Figure 6.** Creation of local variable. The box in *ev-once* mode (at the top) is evaluated only once during a global eval process.

As the box  $+$  in mode *ev-once* (see little icon at the top-left of the box), this example actually corresponds to :

```
(let ((var (+ 7 2)))
  (list var (+ var 3)))
```

This case is a simple example, but the factorization of the call  $(+ 7 2)$  in a local variable can be crucial in more complex processes. In case of nondeterministic processes (e.g. involving a random call), the *ev-once* state will also ensure that the box provides the same results to its different callers.

It is also possible to completely lock a box so that it computes and stores its result once and keep it for all the following calls until the box is unlocked. This would rather correspond to a global variable.

### 3.5. Control Structures

Various control structures commonly used in programming languages (conditionals, iterations, etc.) are available in the OM visual programming framework.

Figure 7 shows an example of an *omloop*, which represents an iterative process (the *loop* Lisp macro). The *omloop* box visible on the left is associated to a special editor allowing one to define the behaviour of the program during this iteration.

In this example the iteration is done via the *list-loop* iterator (other available iterators include *while*, *for*, *on-list*, etc.), on a list given as the input of the loop. At each step of the iteration, hence for each element in the list, another control structure is used: conditional structure *omif*, which corresponds to the Lisp *if* statement (also present in the previous example of Figure 4). Here, the values from the list are incremented if they are inferior to a given threshold. The successive results are collected in a new list which is returned as the result of the iteration. This loop thus corresponds to the following Lisp expression:

```
(lambda (list)
  (loop for x in list
        collect (if (>= x 5) x (+ x 5))))
```

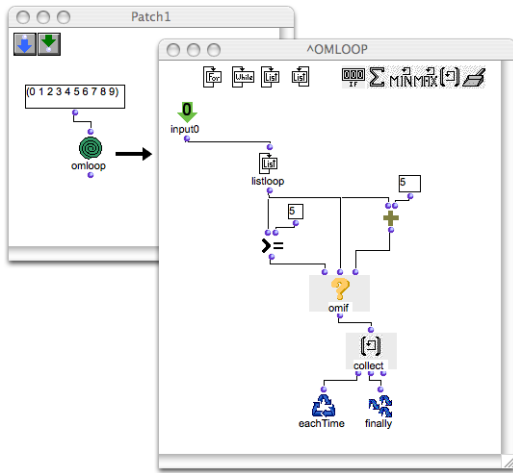


Figure 7. *omloop*: iterative process.

The *file-box* tool is another example of a visual iteration, performing the equivalent of an *omloop* within a *with-open-file* statement, i.e. with an input and/or output access to a file stream. Figure 8 shows an example, corresponding to the expression:

```
(lambda (path list)
  (with-open-file (s path)
    (loop for item in list do
      (write-line item s))))
```

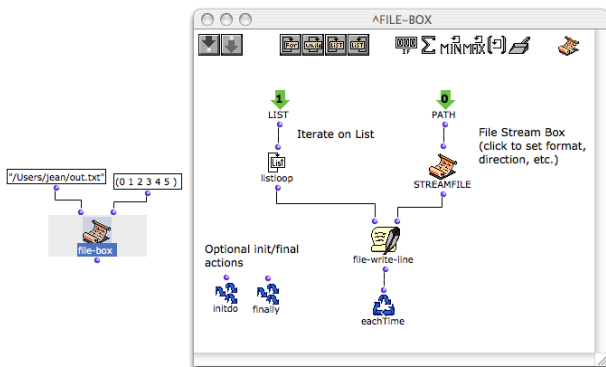


Figure 8. *filebox*: i/o access on file streams. The *stream-file* box represents the stream declaration, initialized with a pathname.

#### 4. OBJECT-ORIENTED PROGRAMMING

In addition to the functional programming features presented above, OM is also offers object-oriented programming facilities.

The main use that is actually done by composers of the object-oriented programming is generally to create instances of in-built classes and methods. OM includes some predefined musical or general-purpose classes and functions organized in a hierarchical package architecture. Figure 9 shows the OM packages browser window.

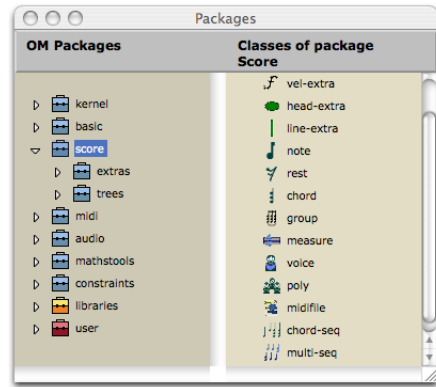


Figure 9. OM packages window.

As we shall demonstrate forthwith, however, users can also define their own classes and methods in the visual language.

They are also provided with means to get further in object-oriented programming with meta-object programming tools. Indeed, as mentioned in the first part of the paper, the meta-object protocol provides reflexive properties, so that the elements that constitute the language can become the objects of processing of this same language. The meta-objects and visual meta-objects that constitute a program (classes, functions, methods, boxes) can therefore be created and modified dynamically while running this program (see [3] for a detailed description of meta-object programming in OM).

#### 4.1. Classes

Figure 10 shows the class tree of one of the OM subpackages: the *score* package. This editor is accessible via the corresponding icon on the package browser window. It shows the different classes defined in this package and their possible inheritance relationships.

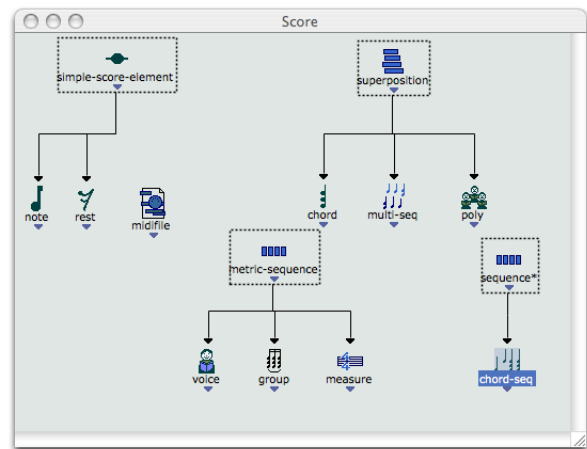
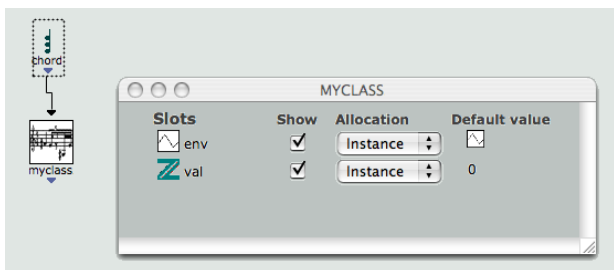


Figure 10. Class tree editor of package *score*. The dotted frames indicate *alias* boxes that refer to classes from other packages.

Users can create their own classes in the *user* package: inheritance relationships can be dynamically created and edited as well by setting/modifying the arrow-shaped graphical connections between user class boxes and/or OM predefined class boxes (see Figure 11).



**Figure 11.** Creation of a user class *mychord*, extending the OM class *chord* (in this example, *chord* is an alias, since the *chord* class is not in the *user* package). The class editor is open at the right of the figure: two additional slots are created.

Figure 11 also shows the editor for the newly created class. New slots can be added to the class, which types and initialisation values are set graphically in this editor. The equivalent Lisp code, automatically generated in this situation, is:

```
(defclass! myclass (chord)
  ((env :accessor env :initarg :env
        :allocation :instance
        :initform (make-instance 'bpf))
   (val :accessor val :initarg :val
        :allocation :instance
        :initform 0))
  (:icon 212)
  (:documentation ""))
```

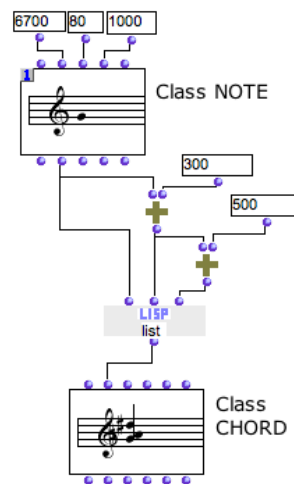
## 4.2. Instances and Factories

In order to use classes in visual programs, a special kind of box is used: the *factory* box. A factory is a box attached to a given class, which represents a functional call generating instances of this class (*make-instance*) and allows to set / get the values of the different slots of this instance. At the top of Figure 12 is visible the *note* class factory. The various inlets/outlets of the factory box represent get / set accesses on the instance itself (first in/outlet from left) and to the different slots of the class (for example, pitch, velocity, duration, etc. for the class *note*).

Figure 12 corresponds to the following expression:

```
(let ((note (make-instance 'note :pitch 6700
                          :vel 80
                          :dur 1000)))
  (make-instance 'chord
    :pitches (list (pitch note)
                  (+ (pitch note) 300)
                  (+ (+ (pitch note) 300) 500))))
```

The predefined musical object classes provided in OM have dedicated editors (e.g. score editors) associated to the corresponding factory boxes, and which allow to edit or just visualize the current instance contained in these boxes (i.e. the last created instance). The *factories* therefore make it possible to generate and/or store the state of a



**Figure 12.** The use of factory boxes in OM. A *note* object is instantiated from integer values corresponding to its pitch, velocity and duration. The pitch is used and processed in order to create a list of three values that are used to instantiate a *chord* object.

data set or structure at a given moment in the calculus (i.e. at a given position in the graph defined in the patch), and at the same time provide a direct access to these data via the editor [2]. That way, they constitute privileged entry-points for the introduction of data and the interaction of the user / programmer with the program [5].

## 4.3. Generic Functions, Methods

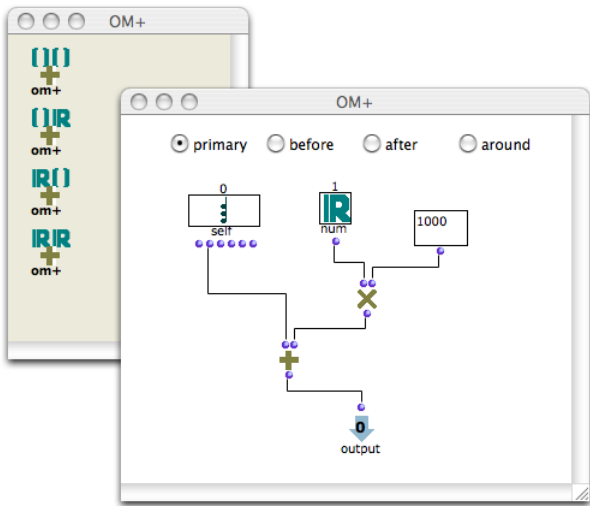
The polymorphism of the generic functions in CLOS is also integrated in the OM visual programming features. New generic functions can be created graphically, as well as their different methods specializing the different possible types of their arguments. It is also possible to add new methods to existing generic functions in order to specialize them for specific types.

Figure 13 shows the editor for the generic function *om+*, which lists its four existing methods. The editor of a new method being defined is also open. It allows to define a visual program corresponding to the method process, and is similar to a patch editor (except for the input types management and the possible *:before*, *:around* and *:after* statements).

The method created in Figure 13 corresponds to the following Lisp definition:

```
(defmethod! om+ ((c chord) (n number))
  (om+ (pitches chord) (om* n 1000)))
```

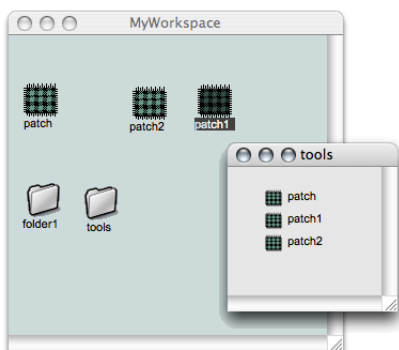
Among the different other features of the OM MOP, are also the possibility to define a specific processing function called at creating an instance of a class, or to redefine graphically the accessor methods of its different slots.



**Figure 13.** Method definition. A new method specializes the generic function *om+* for arguments of types *chord* and *number*

## 5. PERSISTENCE

The main window of the OM environment is called a *workspace* and is similar to a classical OS desktop. In this workspace the user creates programs (patches) and organizes them in a directory tree. Figure 14 shows an OM workspace. Each icon represents a patch or a folder containing patches or sub-folders.



**Figure 14.** A *workspace* in OM.

This organisation reflects a real file/folder architecture on the disk: patches are “persistent” objects. They are saved as Lisp files which evaluation recreate the original visual programs.

Similarly, user-defined meta-objects (classes or methods), organized in packages and accessible via the package tree window (Figure 9), are also saved as Lisp forms in files on the disk. That way, the user can save his workspace’s contents and later reload his programs, classes and functions as in a traditional programming environment.

## 6. CURRENT IMPLEMENTATION ISSUES

OM is one the successors of the PatchWork visual programming environment [13]. It has been initially developed for Macintosh computers using MCL Lisp compiler. In 2005, with version OM 5, the code was refuted so as to improve modularity and reduce Lisp and/or platform dependencies of the environment [8]. An API has been specified, gathering graphical features, user interfaces and non-ANSI CL parts of the code, in order to facilitate portability on new Lisp compilers and platforms. This API has been implemented on MCL and on Allegro CL for Windows. A Linux implementation using SBCL and GTK+ graphical toolkit is also currently in progress. The programming protocol defined by the OM API then allows to systematically interpret the OM code according to the targeted platforms.

Since MCL was not ported on the new Macintosh computers / Intel x86, the need for a new reliable, efficient Lisp with GUI creation toolkit led us to start a new port of OM on LispWorks, which will probably be used as a common support for Mac Intel, Mac PPC and Windows versions of OM. OM 6 / LispWorks for Mac has been distributed as a beta test version on February 2008.

## 7. CONCLUSION

We presented some aspects about the OpenMusic visual programming language, particularly concerning functional and object-oriented programming features. Many works have also been carried out in OM regarding constraint programming: various constraints solver are integrated in it, and were used in a large number of musical applications (see [14], [12], [9]).

Complementarily to these programming tools, OM provides an important library of classes, data structures and predefined functions allowing to head programming toward musical and compositional applications. The more general-purpose tools are integrated in the OM image, while more specific or aesthetically oriented ones are dynamically loaded via external user libraries.

Many musical works have been created with OM during the past 10 years (involving much more complex programs than those presented here). *The OM Composer’s Books* provide varied interesting examples of these applications of visual programming for music composition [4].

## 8. REFERENCES

- [1] Agon, C. *OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur*. PhD Thesis, Université Pierre et Marie Curie (Paris 6), 1998.
- [2] Agon, C. and Assayag, G. “Programmation visuelle et éditeurs musicaux pour la composition assistée par ordinateur”, *14ème Conférence Francophone sur l’Interaction Homme-Machine IHM’02*, Poitiers, France, 2002.

- [3] Agon, C. and Assayag, G. "OM: A Graphical extension of CLOS using the MOP", *Proceedings of ICL'03*, New York, USA, 2003.
- [4] Agon, C., Bresson, J. and Assayag, G. *The OM Composer's Book*, Volumes 1 and 2, IRCAM – Editions Delatour France, 2006 / 2008.
- [5] Assayag, G. and Agon, C. "OpenMusic Architecture", *Proceedings of the International Computer Music Conference*, Hong Kong, 1996.
- [6] Assayag, G., Agon, C., Fineberg, J. and Hanappe, P. "An Object Oriented Visual Environment for Musical Composition", *Proceedings of the International Computer Music Conference*, Thessaloniki, Greece, 1997.
- [7] Assayag, G., Rueda, C., Laurson, M., Agon, C. and Delerue, O. "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic", *Computer Music Journal*, 23(3), 1999.
- [8] Bresson, J., Agon, C. and Assayag, G. "OpenMusic 5: A Cross-Platform release of the Computer-Assisted Composition Environment", *Proceedings of the 10<sup>th</sup> Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brasil, 2005.
- [9] Bonnet, A and Rueda, C. "Situation: Un langage visuel basé sur les contraintes pour la composition musicale", in *Recherches et applications en informatique musicale*, Chemillier M. and Pachet, F. (Eds.), Hermès, 1998.
- [10] Gabriel, R. P., White, J. L. and Bobrow, D. G. "CLOS: Integration Object-oriented and Functional Programming", *Communications of the ACM*, 34(9), 1991.
- [11] Kiczales, G., des Rivières, J. and Bobrow, D. G. *The Art of the Metaobject Protocol*, MIT Press, 1991.
- [12] Laurson, M. "PWConstraints", *Symposium: Composition, Modélisation et Ordinateur*, IRCAM, Paris, 1996.
- [13] Laurson, M. and Duthen, J. "Patchwork, a Graphic Language in PreForm", *Proceedings of the International Computer Music Conference*, Ohio State University, USA, 1989.
- [14] Siskind, J. M. and McAllester, D. A. "Nondeterministic Lisp as a Substrate for Constraint Logic Programming", *Proceedings of the 11<sup>th</sup> National Conference on Artificial Intelligence*, AAAI Press, 1993.