AUDIO SIGNAL PROCESSING AND OBJECT-ORIENTED SYSTEMS

Victor Lazzarini

Music Technology Laboratory
Department of Music
National University of Ireland, Maynooth
victor.lazzarini@may.ie

ABSTRACT

Object-oriented programming (OOP) has been for many years now one of the most important programming paradigms used in a variety of applications. Digital audio signal processing can benefit largely from this approach for systems development. In this paper a number of approaches to using object-orientation in audio processing systems are reviewed. Existing systems of audio processing are introduced and discussed in detail. The paper also draws attention to the different OOP techniques enabled and supported by these systems. Comparative code and tutorial examples are included, providing an insight into the development of signal processing applications using objects.

1. INTRODUCTION: OBJECT-ORIENTED PROGRAMMING

The OOP paradigm is one in a long line of development of computer languages. *Procedural* (sometimes called *Functional)* programming was the original paradigm, whose formal roots can be found on Churchill's λ-calculus[1]. Module decomposition provided a step towards the organisation of data, breaking the problem into independent sets, composed of procedures and related data. The development of the concept of *data abstraction* also provided more support for structured programming. This involves the modelling of real-world problems into user-definable data types[2].

Object-oriented programming extends the concept of user-defined types including the possibility of creating hierarchies of derived data types by inheritance. It also supports the idea of overridable elements, which is also known as *polymorphism*. This enables the development of abstract data objects which can be specialised into different concrete forms, sharing the same interface. In audio and music processing development, this feature is highly desirable. Processes can be modeled to serve as the basis from which more specific ones can be created, including some processes not previously anticipated.

Object-oriented languages are normally divided into class-based and object-based[3]. Class-based languages introduce the differentiation between the concepts of class (the description of an object, its 'kind') and object (a distinct instance of a class). The class description involves mainly two elements: attributes and methods. The former defines what objects of class are composed of, whereas the latter defines the interface to the class (and what objects can perform). In some OOP systems, the class interface includes the concept of *messages*. These, when sent to objects, will invoke their associated methods.

Examples of class-based languages are C++[4], Java[5] and the music processing language PD[6]. Object-based languages do not support the concept of classes, instead providing constructs for the creation of individual objects. Originated in the artificial intelligence community, these languages have not had any significant impact on audio signal processing development.

Another important aspect of programming with objects is the idea of *encapsulation*, whereby access to attributes of an object is mediated by methods defined within that type. Because it enables a black-box-style development, where the user is concerned only with what an object is and what it does, this is very well suited to audio programming.

A number of programming techniques arise from the OOP model. Common ones include *composition, refinement and abstraction*[7]. Composition is based on the reuse of existing classes as attributes of a new class. Refinement provides a derived class with extra support for a number of features not present in the base class. Abstraction is the technique used for developing a model (or abstract) class that can serve as the basis for a number of complex, specialised, classes.

2. AUDIO PROCESSING SYSTEMS

Audio processing systems such as sound compilers (e.g. Csound, CLM, Nyquist, Cmix) have had a long history of development, which can be traced back to Max Mathew's MUSIC series of programs. Some of these can be seen, in a very loose way, as object-oriented. For instance, in Csound, a class concept appears in the form of *instrument* definitions [8]. Instrument are instantiated when 'play a note' messages are sent to them. Typically, instruments are composed of smaller objects called unit generators (UGs), or opcodes (in reference to a syntax similarity with assembly languages). UGs are instantiated objects, with their related internal data and two associated methods. These are invoked automatically by the system, one when the object is created (a init method) and the other when signal is processed (a perform method). The provision of at least one perform method is a typical feature of all audio processing systems (for obvious reasons). Nevertheless, in many of these systems, there is not a lot of support for more complex forms of OOP-style development, or even for a more structured way of programming.

3. OBJECT-ORIENTED AUDIO PROCESSING

Object-oriented features are supported in a more comprehensive way by a number of graphic programming languages which

include Max/MSP, JMax and PD, as well as by systems such as RTcmix and the Synthesis Toolkit. Object-oriented development of audio signal processing applications is also supported by a number of public-domain libraries written in OOP languages such as C++. This paper will be concentrating on three cases: PD, the RTcmix[9] system and the Sound Object Library[10]. In addition, the possibilities provided by the Java Sound API[11] are examined as yet another approach to object-oriented audio programming.

3.1. PD

Pure Data (PD) is a graphical programming language developed originally by Miller Puckette. A PD program is edited graphically as a 'patch' of interconnected objects. These feature any number of inlets and outlets, which are connected with lines acting as patchcords. Objects are instances of PD classes or patches, and there is very little, if any, functional difference between compiled PD classes and patch-defined classes. As a programming language, PD lacks the more advanced features of inheritance and polymorphism. Classes cannot be created as specialised versions of pre-existing ones, but only through composition and delegation (passing arguments and messages to its components).

```
typedef struct CLASS_EX_DATA {
          t_object x_obj;   // base (inherited) attributes
          (...)                     // other attributes
} class_example_data;
```

Figure 1. *PD Class attributes*

Another important aspect of PD is its support for customization and extensibility, which is demonstrated by the possibility of implementation of new classes as part of dynamically-loadable libraries. Classes are usually created in C, which has no support for full-fledged OOP development. In this way, effectively, classes are coded as C-language modules, which makes for awkward coding in certain situations. Class attributes are usually wrapped in a data structure, as shown in fig.1.. It includes a required `t_object` data field, which makes the class a subclass of a basic abstract type. The base class contains attributes relating to how an object is displayed and how it interacts with other aspects of the system.

```
static t_class*  class_example;

void lib_example_setup(void) {
class_example = class_new(gensym("example~"),
   (t_newmethod) class_example_new,
   (t_method) class_example_free,
   sizeof(class_example_data),
   CLASS_DEFAULT, A_DEFFLOAT, 0);

class_addmethod(class_example,
   (t_method) class_example_dsp, gensym("dsp"), 0);
}
```

Figure 2. *Registering a PD class*

Classes, their constructors, destructors and methods are registered with the system using a special setup function, executed when a class library is loaded by PD. The convention used is that one such function is present on each library, with the library name followed by an underscore as a prefix to `setup()` as shown in fig.2. System functions, such as `class_new()`, `class_addmethod()` etc., are used to register and define the class interface. A pointer to the class is returned by `class_new()` to be used by PD to invoke the constructor, destructors and other methods when a new object is instantiated.

```
void*
class_example_new(t_floatarg arg){
 // allocate class dataspace
 class_example_data *data =
            pd_new(class_example_data);

 // set 2 signal inlets & 1 signal outlet
 inlet_new(&data->x_obj, &data->x_obj.ob_pd,
                          &s_signal, &s_signal);
 outlet_new(&data->x_obj, &s_signal);

 // other initialisations (args etc)
 // if necessary
 (...)
 return (void *) data;
}
```

Figure 3. *PD class constructor*

An unspecified number of methods to deal with control messages can be implemented, and, in the case of signal-processing classes ('tilde' classes) one perform method is required. Using `class_addmethod()`, these are registered with the system. Each method is associated with a message, so that it can be invoked when such message is passed to an object. The message-passing system is one of the most important OOP features in this language. A number of standard messages are predefined, such as 'float', which signals the arrival of a single floating-point number at an inlet. Any message can be defined for an object, e.g. 'reset', followed or not by numerical arguments

```
void class_example_dsp(class_example_data *x,
                  t_signal **sp){
dsp_add(class_example_perform,  // method name
              5, // number of args,
              x, // pointer to object data
             sp[0]->s_vec, // signal vectors
             sp[1]->s_vec,
             sp[2]->s_vec,
             sp[0]->s_n);  // size of vectors
}

t_int* class_example_perform(t_int *argtab){
    int i; // counter
    class_example_data *x =
     (class_example_data *) argtab[1];  // not used

           // signal vector pointers (2 inputs, 1 output)
   t_sample *input1 = (t_sample *) argtab[2]; // sig input 1
   t_sample *input2 = (t_sample *) argtab[3]; // sig input 2
   t_sample *out = (t_sample *) argtab[4];    // sig output
        // size of vectors in samples
   int vecsize = (int) argtab[5];

        // processing loop
   for(i=0;i < vecsize; i++)
    out[i] = (t_sample) (input1[i]+input2[i]); // mix
    return (argtab+5);
}
```

Figure 4. *PD perform method example*

A PD class constructor, as usual, allocates memory for the object data (using the PD system function `pd_new()`) and

initialises it. It also creates all the inlets/outlets needed by the object and performs any other data allocation and initialisation (fig.3). PD class destructors are only needed if any extra memory was allocated by the constructor (other than by `pd_new()`). The 'dsp' method in PD adds the perform method to a list kept by a scheduler. The perform method will produce a vector of samples at the output on each DSP cycle. It can take a number of input/output vectors (depending on the number of signal inlets/outlets), a pointer to the object data and the size of the vectors. A code example is given on fig.4, implementing a simple mixing of two signal inputs (a class with two inlets and one outlet, as defined in the constructor, fig.3).

In terms of system architecture, PD has two layers: a Tcl/Tk-based graphic interface and a processing engine. These two layers communicate over a socket connection (usually a TCP/IP stream). Signal-processing developers do not need to concern themselves with this side of the system. PD classes interface transparently with the rest of the system, provided that they follow a number programming guidelines. This tends to put some constraints on class development. Also the use of the C-language to implement object-oriented features is very awkward, especially the constant need to pass pointers to the class data structure to its methods. Nevertheless, it is possible to use C++, a genuinely OOP language, to develop PD classes. This has been done using a C++ wrapper, *flext*, developed by Thomas Grill [12].

3.2. RTcmix

RTcmix is a version of the original Cmix system developed at Columbia University. It is basically a C++ library of sound-processing 'instruments', which can be driven by a score parser (Minc, originally developed for Cmix) or, similarly to PD, by a TCP/IP socket connection. The system provides a scheduler that calls the signal-processing method of an instrument when it is invoked by Minc or the socket connection. Any interface can be developed to interact with a RTcmix instrument using TCP/IP.

```
class INST_EXAMPLE : public Instrument {
  // any class attributes
  // declared here
  public:
  INST_EXAMPLE();
  virtual ~INST_EXAMPLE();
  int init(float *, int);
   int run();
};
```

Figure 5. *RTcmix instrument class example*

Instruments can be added to the library using the C++ language. An RTcmix instrument is a class which inherits a basic set of attributes and methods from a base class (Fig.5). Two basic methods are overridable and need to be implemented in the derived class: an initialisation method (which gets parameters from the parser or socket, `init()`) and, as in PD, a 'perform' method (which generates audio, `run()`). The base class also provides methods to output the signal vector (`rtaddout()` and `addout()`) and other utilities. It also contains all general attributes of an instrument, such as number of input/output channels, sampling rate etc..

```
INST_EXAMPLE::init(float p[], int n_args)
{
  float outskip, inskip, dur;
  // standard p-fields
  outskip = p[0]; // out skiptime (start time)
  inskip = p[1];  // in skiptime
  dur = p[2]; // duration
  amp = p[3]; // amplitude

  // Handle additional p-fields
   (...)

   // set the input/output bus pointers
   nsamps = rtsetoutput(outskip, dur, this);
   rtsetinput(inskip, this);

  // check p-field consistency
   (...)

   // set function tables if used
   (...)

   // Set control rate counter.
   skip = (int) (SR / (float) resetval);
   return nsamps; // number of sample frames to be written
}
```

Figure 6. *RTcmix init() method*

The `init()` method takes as arguments a number of input numeric fields from the score (pfields), in the form of an array of floats and its size (fig.6). Its task is fourfold: to read, check and store pfields; set input/output bus pointers; set any function tables and other instrument-specific elements; and, finally, set the control rate counter, which is used to update slow-varying parameters, such as envelope amplitude and sub-audio modulating frequency. The `init()` method is called by the scheduler when the instrument is to be initialised.

```
int INST_EXAMPLE :: run() {
 int   samps;
 float out[2]; // 2-chan max output

   // base class run()
   Instrument::run();
   // samps is the total number of samples in the vector
   samps = chunksamps * inputchans;

   // get the input samples into the buffer *in*
   // which should have been previously allocated
   rtgetin(in, this, samps);

   // the processing loop
   for (int n = 0; i < samps; n += inputchans) {
      if (--branch < 0) {
         // any control rate variables should be
         // updated here
         branch = skip;
      }
      // copy input scaled by amp
   out[0] = in[n]*amp;
      if(outputchans == 2) out[1] = in[n+1]*amp;

     // write the frame to the output buffer
      rtaddout(out);
      cursamp++; // number of written frames
   }
   return chunksamps;
}
```

Figure 7. *RTcmix run() method*

The `run()` method is the business end of an instrument class. It is called by the scheduler every time-slice in which the instrument should run. It process a vector full of input frames into the output. The instrument developer needs to take care of

any memory allocation that is needed for input sound buffering etc. . Also, there is a requirement that the base class `run()` method is called at the top of the function. Typically, here the developer will (in similar fashion to PD) provide a loop where the vector samples are processed. For efficiency, inside the loop, control rate values can be updated less often, using the control rate counter. As seen in figure 7, the perform method for RTcmix is very straightforward, if compared to PD, especially in terms of how to obtain the input signal and how to write to output. In addition to this, any method can be included in an instrument to instantiate independent socket connections for message processing. In general, as opposed to PD, RTcmix tends to put very few constraints on how instruments should be designed, so that a range of OOP techniques supported by the C++ language can be employed in their development.

3.3. The Sound Object Library

The Sound Object (SndObj) Library is an object-oriented cross-platform audio signal processing framework. As opposed to the previous two cases, it is not an audio processing system, but a set of programming tools for software development. As such, it supports all OOP techniques described above, enabling fast development of standalone applications, as well as the development of new signal processing algorithms. There are no constraints imposed on the developer, no need for implementation of any special method, or any type of identifier registration. Even if a developer does not provide a specialised 'perform' method (overriding the base class `DoProcess()`), a newly developed class can still be compiled and used (but of course with little functionality).

```
SndRTIO input(1, INPUT); // realtime IO
SndRTIO output(1, OUTPUT);
SndObj audio;            // sound object
DelayLine delay(0.3, &audio); // 0.3 sec delay

while(audio_on){

  audio << input; // get input
  delay.DoProcess(); // delay it
  audio *= 0.5; // scale direct signal
  delay *= 0.5; // scale delayed signal
  audio += delay; // mix
  audio >> output; // output

}
```

Figure 8. *Using the SndObj library*

The library uses some concepts of OOP to its full advantage. For instance, inheritance and polymorphism provide ways of creating a whole tree of derived classes, which can share the same interface, but effectively implement different signal processing algorithms. New classes can be derived from existing ones, maximizing code re-use. In this case, a new class will typically override the 'perform' method of its base class, but can use any of the other facilities provided by it. Several classes, dealing with some basic signal processing families (e.g., filters, delay lines, etc.), are present in the library, from which specialised versions can be created.

```
class ClassExample: public SndObj {
   protected:
   SndObj* m_input2;

   public:
   ClassExample();
   ClassExample(SndObj* input, SndObj* input2
      int vecsize=DEF_VECSIZE, float sr=DEF_SR);

   void SetInput2(SndObj* input){ m_input2 = input;}

   short DoProcess();
   char* ErrorMessage() { SndObj::ErrorMessage(); }
};
```

Figure 9. *SndObj-derived class example*

Objects created from SndObj classes have a true OOP behaviour, as shown in figure 8. A SndObj object is defined by its internal state (sampling rate, vector size, inputs taken etc.) and what kind of processing it does. Its output is available to any other object of the family. Apart from producing vectors of samples when the `DoProcess()` method is invoked, using operator overloading, objects can be added, subtracted, and multiplied together or by numeric values. Shift operators can be used to input/output signal to SndIO classes (which deal with all aspects of input/output). On a higher-level, connections are created between objects, rather than by using signal patchcords or buffers.

```
ClassExample::ClassExample(){ // default constructor
    m_input2 = 0;
}

ClassExample::ClassExample(SndObj* input, Snd0vj* input2,
        int vecsize, float sr)
  :SndObj(input, vecsize, sr){
 // perform the initialisation
    m_input2 = input2;
}

short
ClassExample::DoProcess(){
if(!m_error){ // check for errors
 if(m_input && m_input2){ // check if inputs are set
     // processing loop
  for(m_vecpos = 0; m_vecpos < m_vecsize; m_vecpos++) {
  if(m_enable) {
  // m_output holds the object output signal
       m_output[m_vecpos] = m_input->Output(m_vecpos) +
                    m_input2->Output(m_vecpos);
  }
  // if processing is bypassed
     else m_output[m_vecpos] = 0.f;
  }
     return 1; // return 1 if succesful
 } else {
 m_error = 3;
 return 0;
 }
}
else return 0;
}
```

Figure 10. *ClassExample implementation*

In terms of applications, the SndObj library has been extensively used in the development of stand-alone sound processing software. It can be also used to provide high-quality audio to any application. The library can also be integrated with other music processing systems. For instance, it has been used to implement new PD classes, such as `syncgrain~`, by Frank

Barknecht[13]. Developed employing the homonymous SndObj class, it implements sampled-sound granular resynthesis.

One of the most important aspects of the library is its support for class development. Classes can be derived from any of the available ones, depending on the type of processing and the types of services provided by a base class. For instance, for the development of delay-based processes, it might be useful to derive a new class from either DelayLine or one of its derived classes. The simplest way of creating a new class is to specialise the base class, SndObj. In figure 9, the interface for a class that does exactly the same task as the PD example is shown. All is needed is to add the code for the constructor and the perform method, `DoProcess()`. The simplicity of the implementation is clear (fig.10).

3.4. Object-oriented audio in Java

Java™ is an object-oriented language in certain ways very similar to C++. One of the main differences is that, being an interpreted language, it is not as efficient for number-crunching as compiled binaries. Usually, Java tends to be used in the development of interfaces for systems (e.g. for RTcmix, or for a SndObj application). Nevertheless, the latest versions of Java have been much optimised and it is feasible to envisage the development of audio processing applications based on this language.

The introduction of the Java Sound API and the package javax.sound.sampled has provided some possibilities in the area. Currently, it provides Java classes and interfaces to access audio devices and soundfiles for reading and writing, as well as providing simple processing controls (e.g. pan, gain, reverberation etc.; implementation dependent). The audio input/output side of the API is based around the concepts of *Mixer* and *Lines*. The former are representations of audio devices, which can receive/produce formatted signal streams, and the later patchcords or pipelines in and out of Mixers and other objects (source and target lines). The `Line` interface hierarchy, which includes `Mixer` and other types is shown on fig.11.

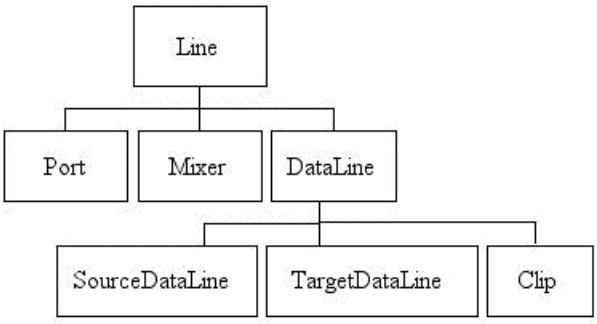

Figure 11. *Java Sound Line interface hierarchy*

The functionality of the base interface includes methods for opening/closing lines, setting up controls and generating events (related to the status of the line). The subinterface `Port` describes a line relating to a physical input/output (mic, speaker, line, etc.), whereas `Mixer`, as mentioned before, describes an audio device. `DataLine` is a subinterface that extends `Line` to give it extra features, such as an associated data format, a signal buffer, start/stop streaming controls etc.. `TargetDataLine` and `SourceDataLine` are specialised versions used for sound input and output, respectively. They include methods for reading/writing the object buffer. `Clip` is a simplified version of `SourceDataLine` which stores a single signal vector intended for output.

The Java Sound API seems to provide comprehensive support for audio input/output, as well as for soundfile operations. It is possible, therefore, to develop signal processing classes, in similar fashion to the SndObj library to provide OOP audio manipulation resources in Java. Other possibilities involve the use of dynamically loadable libraries ('native methods' in Java) to perform the more intensive number crunching aspects of a class (the perform method, for instance). A Java-based music processing object-oriented system is a definite possibility, using one of the strategies described above.

On an implementation level, particularly important is the provision of what is called a service provider interface (SPI) which would enable extensions to the basic services provided by the API. In general, it seems to be designed to implement file reading, writing and conversion to/from new formats, but there is also support for implementing new mixer interfaces An audio processing engine could possibly be designed as an implementation of such Java interface. This would enable the provision of signal processing algorithms which can be 'plugged-in' to an existing implementation of the API. The advantage, from a programmer's point of view, is that such services could be transparently integrated.

4. CONCLUSION

Object-oriented development can be very useful for audio processing applications and systems. Examples can be found on all levels of programming, from graphic music processing languages such as PD, to software libraries in C++ or Java. Although the different systems discussed in this paper vary in level of complexity, they share some important aspects which make OOP an important systems development paradigm. It was also shown that, the more a system adheres to object-orientation, the more straightforward the programming task can be. The comparison between PD class development and the other two C++ based systems, RTcmix and the SndObj library, clearly shows it. By extension, it is very opportune to note that Java, as thoroughly object-oriented language, is starting to offer very real possibilities for audio processing, as an alternative to C/C++-based software development.

5. REFERENCES

[1] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, Englewood Cliffs, 1990.

[2] Carlo Ghezi, Mehdi Javayeri, *Programming Language Concepts*, J Wiley & Sons, New York, second edition, 1987.

[3] Martin Abadi, Luca Cardelli, *A Theory of Objects*, Springer-Verlag, New York, 1996.

[4] Bjarne Stroustrop, *The C++ Programming Language*, Addison-Wesley, New York, second edition, 1991.

[5] Ken Arnold, James Gosling, *The Java Programming Language,* Addison-Wesley, New York, 1996.
[6] Puckette, M., "Pure Data", in *Proc. International Computer Music Conference*, San Francisco, pp. 269-272, 1996.
[7] Pope, S., "Machine Tongues XI: Object Oriented Design", in S. Pope (Ed.), *The Well Tempered Object*, MIT Press, Cambridge, Mass, pp.32-45, 1991.
[8] Ffitch, John, "What Happens When You Run Csound", in R. Boulanger (Ed.), *The Csound Book*. MIT Press, Cambridge, Mass, pp.99-121, 2000.
[9] Garton, B., Topper, D., "RTCmix Using CMIX in Realtime", http://www.music.columbia.edu/cmix/rtrealtime.html.
[10] Lazzarini, V,"The Sound Object Library", *Organised Sound 5 (1)*, Cambridge: Cambridge Univ. Press., 2000, pp 35-49.
[11] http://java.sun.com/products/java-media/sound
[12] Grill, Thomas. "Flext, C++ layer for MaxMSP and pd externals". http://www.parasitaere-kapazitaeten.net/Pd/ext/flext/
[13] Barknecht, Frank. "Syncgrain~". http://footils.org/