Octopus Music API

Version 1.2 (Late 2009)Table of Contents

Abstract 3

Contents Overview 5

Table of Contents 6

List of Illustrations 7

List of Tables 8

Chapter 1 Guitar Performance Modelling 11

1.1 Octopus Music API (Application Programming Interface) 15

1.1.1 Octopus Project Design 17

1.1.2 Musical Data Structures 20

1.1.2.1 Class octopus.Note 21

1.1.2.2 Class octopus.Chord 22

1.1.2.3 Class octopus.Bar 23

1.1.2.4 Class octopus.RhythmPattern 25

1.1.2.5 Class octopus.Arpeggio 27

1.1.2.6 Class octopus.Scale 29

1.1.2.7 Class octopus.HarmonicProgression 30

1.1.2.8 Class octopus.Melody 30

1.1.2.9 Class octopus.Harmony 30

1.1.2.10 Class octopus.Music 32

1.1.3 Musical Data Interpreters 35

1.1.3.1 Class octopus.Musician 38

1.1.3.2 Class octopus.instument.Performer 38

1.1.3.3 Class octopus.instument.fretted.Guitarist 39

1.1.4 Instrument Classes 44

1.1.4.1 Class octopus.instument.Instrument 46

1.1.4.2 Class octopus.instument.string.fretted.FrettedInstrument 46

1.1.5 Communication Classes 47

1.1.5.1 Class octopus.communication.MusicalEvent 49

1.1.5.2 Class octopus.communication.MusicalEventSequence 51

1.1.5.3 Class octopus.communication.SynthesizerController 51

1.1.6 Final Considerations of the Octopus Music API 52

Bibliography 54

List of Illustrations

Figure 63: Simplified class diagram for the Musical Interpreters. 18

Figure 64: Class diagram for Musical Data Structure. 20

Figure 65: Bar internal data structure. 23

Figure 66: Example of the score representation for a Tuplet. 23

Figure 67: *RhythmPattern* internal data structure. 25

Figure 68: Class Arpeggio internal data structure. 27

Figure 69: Arpeggio time-stretching feature applied to the Harmony. 28

Figure 70: Class diagram for the *MusicComponents*. 31

Figure 71: Internal musical components organisation over the time. 32

Figure 72: Class diagram for the Musical Data Interpreters. 36

Figure 73: Am chord shape. 40

Figure 74: I*nstrumentGraphicalInterface*: a graphical interface of the Guitar class. 45

Figure 75: Conventional MIDI setup. 47

Figure 76: Octopus Music download figures. 52

List of Tables

Table 10: Chord’s notes. 21

Table 11: Am to G similarity calculation. 39

# Guitar Performance Modelling

“Error is discipline through which we advance.” (William Ellery Channing).

The idea of developing computational models of music performance dates back to the first computer applications. These first models were mainly dedicated to music production and experimentation (Gareth and Curtis, 1985).

Like any other musical application, these models had to handle common musical elements, such as a note, duration ratios between successive notes, ascending lines, melodic leaps, melodic and harmonic change, phrase structure, et cetera. Although these elements are essential to the formalisation of any musical application, each developer used to implement their own solution based on the particular needs of the application. It did not take long for them to realise that it would be more productive to have generic library of musical structures which they could reuse to write their application.

All the main programming languages started to provide libraries with some support for music production, even if it is as basic as a note production based on a frequency. This, however, was not enough and in the mid 1980’s dedicated programming language for music started to be produced. One of the first languages to become popular was the HMSL – Hierarchical Music Structure Language (Polansky and Rosenboom, 1985; Polansky *et al.*, 1987).

A computer programming language presents an abstract model of computation that allows one to write a program without worrying about details that are not relevant to the problem domain of the program (McCartney, 2002). These languages are designed to provide a set of abstractions that makes expressing compositional ideas as easy and direct as possible (McCartney, 2002). Examples of such languages are CSound (Vercoe, 1986), MAX/MSP (Puckette, 2002), SuperCollider (McCartney, 2002), Nyquist (Dannenberg, 1997) to name a few. Each specialised into some type of musical task: composition, performance, synthesis and so on.

Still, there is one particular area that has not been explored by these languages: musical performance modelling. When a language or application claims to be designed for musical performance purposes, it is purely a performance tool that is controlled in performance-time by the performer, similar to a musician playing an instrument. This means that the performance actions are still the performer’s responsibility and not something embedded in language, as one would expect in a performance ‘modelling’ language. In Section 2.5 (p. 55) we have seen an example of that with the PwSynth and ENP (Laurson, 2000; Laurson *et al.*, 2001).

In a more conventional setup, composers transcribe musical ideas into a written score in a way it can be understood and interpreted by another human, the instrumentalist/interpreter. Ultimately, it is the interpreter’s role to convey all the emotion intended by the composer to the listeners. This is a task that requires a great deal of sensibility and intelligence that is difficult to be mimicked by a machine.

To perform a similar task, a machine would require a much more detailed specification of the music performance, however as Gareth and Curtis (1985, p.237) explains, there is just a certain amount of information that can be formalised.

“The importance of formal thinking, in music or in any other area, cannot meaningfully be separated from the development of formal languages through which that thinking is officially expressed, and unofficially explained.”

In music, the active explicit creation of formal languages which is used to express aspects of theory, performance, or composition, did not come until the computer made it possible to automate some aspects of the processing of formal languages (Gareth and Curtis, 1985). The limitations of rules as constraints to a purely formal approach leave some gaps in the domain of creative, artistic (and ipso facto informal) endeavour.

This is the point where Artificial Intelligence (AI) meets Music Performance. Computers can now perform musical tasks that were formerly associated exclusively with naturally intelligent musicians (Roads, 1985). However, most of the Expressive Music Performance models (EMP) do not consider the bodily limitations behind performance actions. Even if they did, it would not be possible to formalise these physical actions using a conventional programming language for music performance because they do not provide support for this type of modelling.

To exemplify what has already been discussed, suppose we want to model a G note using a guitar. In a normal language for music performance, we have to select one of many available guitar timbres, the note’s frequency (G4), intensity (velocity), and duration.

In a language for music performance modelling, we would probably have to specify among other things: the string and fret used, the string tuning, gauge and tension, the guitar scale length, the fretboard inter-fret spacing, the finger used to stop the string and the force applied to do so, the finger used to pluck the string, nail the plectrum shape and hardness, the direction of the stroke, its intensity and region.

The example above shows that such approach is just not practical from the programmer’s point of view. If this level of detail is going to be modelled, then the language needs to provide a way to do it effortlessly. This, however, is just part of the problem. One must also find a Sound Generation Unit capable of taking full advantage of all of this detailed information and produce a realistic performance.

In summary, in order to model the results of the performer’s physical actions during a musical performance, two main fronts of development need to be approached:

1. A programming language/library that offers the level of abstraction necessary to detail the performer and the instrument constraints as well as supporting common musical tasks;
2. An intelligent algorithm capable of learning and predicting the biomechanical limitations of the human body during a music performance. This algorithm together with the environment mentioned above would save the programmer the time-consuming task of having to specify every single task of the performance;

Each of these two topics has its own challenges and they are described in the first two sections of this chapter. The third section will discuss the integration of the machine learning algorithms with the Octopus Music API: a Java library designed to model musical performances.

## Octopus Music API (Application Programming Interface)

Computers have been used to perform musical-related tasks in many different areas such as audio signal processing, score representation, compositional assistance, and real-time control of the complex processes that go into creating, performing, and analysing music. However the development of programming languages specifically for musical applications seems to have concentrated on the areas of sound synthesis and musical composition (Loy and Abbott, 1985).

According to Loy and Abbott (1985), three strategies have been commonly employed in the development of musical tools:

1. modifying a composition program written in an existing programming language;
2. writing a programming language as the embodiment of a musical paradigm;
3. developing libraries of utility subroutines that implement common operations on musical data structures then writing composition programs in some standard programming language

The Octopus Music API is an example of the third approach and, as one can imagine, it is not the only programming library with a musical purpose. Numerous software packages have been written for applications in music composition, music analysis, sound synthesis, and sound manipulation (Pennycook, 1985). Unlike other Java APIs’ for musical software development, the Octopus Music API is specifically designed to model the interaction of the musical performance elements, mainly the performer and musical instrument.

As a programming library the Octopus can be used to write any application that requires dealing with musical structures, such as composition or musical educational software. As an example of the possible use for the Octopus Music API, suppose that software has to play a simple harmonic progression composed of the chords ‘C – F7 – G’. This is a very simple task; all that the software has to do is to determine the notes of the chords and play them in a specified tempo and timbre (i.e. GM guitar). Any musical programming language can do this with ease. This is what we call the 1st layer of abstraction.

Now, let us add another feature to this software. The harmonic sequence needs to be played with the ‘Admira Concerto Classical’ guitar timbre (standard tuning) using a particular chord fingering. In addition, a particular guitar strumming should be applied. This 2nd layer of abstraction will require a base (harmonic) guitar playing knowledge and the ability to communicate with a Sound Generation Unit able to render the particular timbre of that guitar. This layer is much more specialised than the first, but still possible to be achieved using most musical programming languages as long as the programmer is experienced and is prepared to work hard.

The 3rd layer is where the specialisation reaches another level. The software is now requested to play the same sequence using not only the timbre of ‘Admira Concerto Classical’ but also considering its playability and mechanics (string gauge, dimensions, tuning, etc). Additionally, the sequence should be played by a certain flamenco guitarist named B.B Queen who is a left-handed and famous for his peculiar use of the rhythmic-hand; B.B Queen likes to play the guitar placed on his right-leg and he is a slightly anxious with this gig because he never played this particular guitar before. Facing a demanding audience, B.B Queen is wondering if he should have rehearsed more rather than dedicated his last 3 months trying to learn wakeboard in the Caribbean Islands.

The use of layers illustrates the different levels of complexity, and consequently the amount of coding required in order to achieve the goals set. Imagine the amount of recoding that would be necessary in the event that the guitar or performer had to be replaced. Even worse, imagine if the harmonic sequence has to be played on a different instrument or even in an ‘extended instrument’ [[1]](#footnote-1)without a formal performance technique.

The Octopus Music API was designed to deal with the 3rd layer making the most of the Object-Oriented Programming (OOP) concepts, such as encapsulation, polymorphism, and inheritance (Booch *et al.*, 2007). The API was organised in a way that contemplates its extension so in the future it could be extended to model all sorts of musical performance. The Octopus API design is explained in the next section.

### Octopus Project Design

One of the first decisions to be made once the full requirements of a software (including a library) has been identified is the selection of the most suitable technology for the job, in this case, a Object-oriented Programming (OOP) language with basic support to audio and MIDI functionalities. We have decided to use Java SDK (version 1.5).

One could argue that, compared to languages specially designed for Computer Music, Java suffers from slower performance. Indeed Java sacrifices performance in order to be flexible and portable, allowing the integration of music and sound processing with features like networking, graphics rendering, mobile devices programming, database, etc. (Costalonga et al., 2005). Furthermore, Java is free and widely used in academics.

The Octopus Music API is composed of 80 classes (52 publics) totalling over 16,000 lines of code. The packages are physically organised as follows:

**Package octopus**: The classes in this package represent general musical structures, such as *Note, Chord, Melody*, etc.

**Package octopus.instrument**: This package contains general classes that are useful to any and all musical instruments modelled within Octopus. The classes on in this level must be generic enough to allow the expansion of the API to new instruments.

It is in this level that a more abstract musical structure such as *Melody* (package octopus) becomes the more practical *PerformableMelody*, so rather than only notes in a musical score, the *PerformableMelody* contains instructions of how to play the *Melody* in a particular *Instrument*.

**Package octopus.instrument.fretted**: These are specialised classes for fretted instruments, normally from the Luto family. It is in this level that we solve some problems related to guitar performance modelling. For instance how to pinpoint a note position in the guitar fretboard (class *GuitarNotePosition).*

**Package octopus.communication:** These classes prepare Octopus for the next step of the research: integration with a Sound Generation Unit capable of rendering a performance with the level of detail that Octopus allows. These classes work as middle-tier communication protocol to external devices. They are internal descriptors that can be easily parsed to whatever communication protocol the device (and Java) supports such as MIDI or OpenSound Control (OSC).

**Package octopus.communication.midi:** Communication classes *parse* the internal performance descriptor into MIDI messages.

**Package octopus.util**: Miscellaneous utility classes;

More important than the physical arrangement of the classes into packages is the classes’ conceptual classification. Four categories are used:

1. Musical Data Structures classes (playable): Classes that represent musical concepts and can be played;
2. Musical Data Interpretation Classes (musicians): Classes capable of translate ‘playable’ objects (instance of a classes) into sound;
3. Instrument Classes: Classes that allow the modelling of the musical instrument playability attributes (mechanical/ergonomic);
4. Communication classes: Bridge between the Musical Data Interpretation classes and the Sound Generator Units;

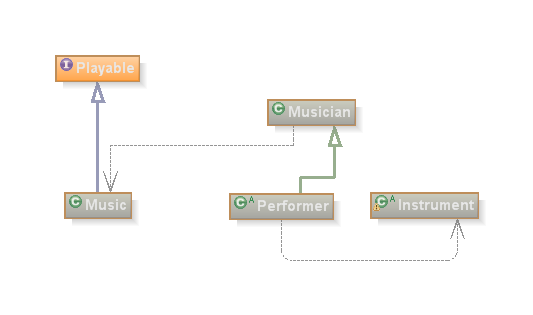


Figure 63: Simplified class diagram for the Musical Interpreters.

Figure 63 illustrates with a sample class diagram the conceptual organisation of the classes. The *Music* class (Data Structure) realises the *Playable* Interface and, as a result, becomes ‘playable’ to the Musician (Musical Data Interpretation). The *Performer* is a subclass of *Musician*, therefore inherits all the musical ‘knowledge’ which is extended to contemplate the *Instrument* handling*.*

The main classes of each of these categories will be discussed in the next sections.

The full documentation and API (including tutorials) can be found online at <http://sourceforge.net/projects/octopusmusic/> or in the CD Annex.

### Musical Data Structures

Generative composition languages usually come with descriptive musical data structures but emphasise compositional processing (Loy and Abbott, 1985). With many aspects of music, we know what to represent, but the issue is how to represent it (Dannenberg, 1993).

Musical Data Structures are a computational formalisation of musical concepts that are compatible with other classes of the API. It includes basic classes such as *Note, Melody and* *RhythmPattern*. These classes could be used as part of a hard-coded composition, as part of an application that generates music algorithmically, or any other application that benefits from the structural relations of musical elements.

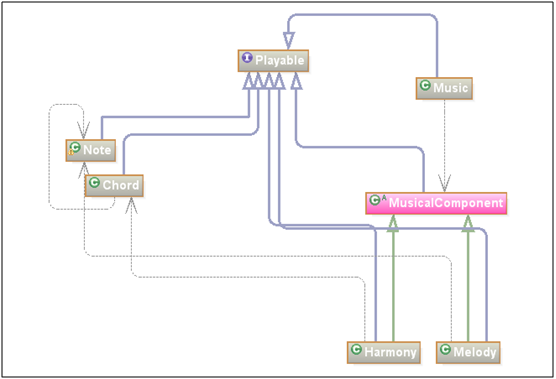


Figure 64: Class diagram for Musical Data Structure.

Figure 64 presents a class diagram of some of the structures modelled in the API. All the Music Data Structures must realise the Interface *Playable*, meaning that it can be played by a *Musician*. The most obvious examples are: *Note*, *Chord,* and *Music*. A less trivial *Playable* structure is the *MusicalComponent*, which is the abstract class that any Musical Structure must implement in order to be compatible with *Music* internal data structure. Although *Note(s)* and *Chord(s)* are found in *Music*, they must first be grouped into *Harmony* and *Melody* structures that implement the *MusicalComponent*.

#### Class octopus.Note

Most computer music notations define a musical note as the specification of an acoustic event. In the traditional music notation a *Note* specifies a human gesture toward an instrument (Loy and Abbott, 1985). For us, the *Note* is the smallest audible element that can be intentionally played or grouped in a musical structure.

Although the *Note* is the simplest musical element of the API, it has attributes like any other object in OOP paradigm. The notes attributes are:

* Name: C Sharp;
* Symbol: C#
* Pitch Value: 64 (midi);
* Octave: C4.
* Accidents: (sharp, double sharp, flat, double flat);

It would be unproductive if every time a Note is required the programmer had to fill in values to all these attributes. So instead, we used a software design pattern known as Factory.

Factories are static classes that return highly demanding objects in a simple form, reducing code overhead. The factory class used to create and perform computations over Note(s) objects is the NoteFactory. Code Example 1 shows two ways of instantiate Note objects.

|  |
| --- |
| Note A = NoteFactory.getA( );  Note Ab = NoteFactory.get(“Ab”); |

Code Example 1: Instantiation of a Note object using the NoteFactory static class.

#### Class octopus.Chord

A *Chord* is a set of *Notes* played together or *arpeggiated*. There are several ways to instantiate a *Chord* object but the recommended one is based on the chord musical notation. The chord musical notation can be seen as a language with a well defined semantic and syntax to describe *Chords*. The API, just like a compiler, runs a lexical analysis over the text describing the *Chord* and validates or refuses based on the alphabet in use. (More information in (Costalonga *et al.*, 2008)

Unfortunately, the chord notation is not standardised all over the world, meaning that the chord names (symbols) might vary among different communities of musicians. The default *ChordNotation* using the API is based on Brazilian Bossa Nova musical genre known by its complex harmonies. If the *ChordNotation* is not adequate for software that is being developed than it might be necessary to load another notation (file).An example of *Chord* instantiation can be seen in Code Example 2.

|  |
| --- |
| Chord chord = new Chord(“C#m7(add11)”); |

Code Example 2: Chord instantiation.

The chord object in the Code Example 2 is populated by the *Note(s)* objects linked to the *Interval* that describes their role, as shown in Table 10.

|  |  |  |
| --- | --- | --- |
| **Index** | **Interval** | **Note** |
| **0** | Fundamental(root) | Note C# |
| **1** | Minor 3rd | Note E |
| **2** | Perfect 5th | Note G# |
| **3** | Minor 7th | Note B |
| **4** | Major 11th | Note F# |

Table 10: Chord’s notes.

Two additional pointers are used to indicate the root and the bass note, which is not the same note in inversion cases. In the example given, both pointers are on *index 0*.

#### Class octopus.Bar

A *Bar* is simply a rhythmic phrase. It is a collection of the smallest rhythmic structure designed in the API. The *Bar.RhythmEvent* can be either note or rest with values between 0 and 1 for duration, dynamic, and accentuation. The tie attribute is used to indicate whether the duration of the *RhythmEvent* should be linked with the next one in the sequence.

The interaction between real time, measured in seconds, and metrical time, measured in beats, is frequently addressed in music representation schemes (Dannenberg, 1993). The time signature of the *Bar* is written in the form of a fraction given by the number of rhythmic units divided by the measuring of the unit. The *Bar* will not prevent the input of *RhythmEvents* that exceeds the time signature instead, the *bar.*[*getSignatureDistance*](file:///\\127.0.0.1\Application%20Data\Local%20Settings\Temporary%20Internet%20Files\lcostalonga\Application%20Data\My%20Documents\code\api\Octopus\doc\octopus\Bar.html#getSignatureDistance%28%29)*( )* method was implemented to inform the programmer if the R*hythmEvents* are in accordance with the *metre* or not. Figure 63 shows the internal structure of the Bar with an example.

Figure 65: Bar internal data structure.

RhythmEvents do not always have to obey the metre. A *tuplet* allows the organisation of two of more *RhythmEvents* in a time frame (duration) smaller than the total duration of the events in the *tuplet*. Code Example 3 shows how to implement the *tuplet* illustrated by Figure 66.

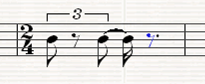
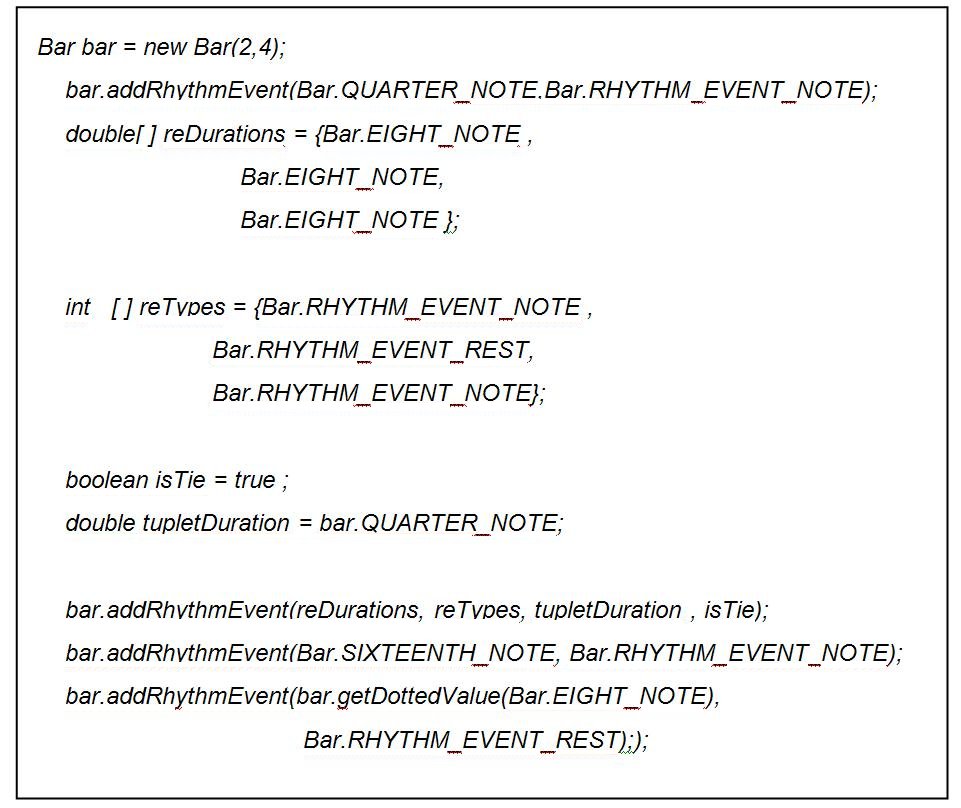


Figure 66: Example of the score representation for a Tuplet.



Code Example 3: Tuplet.

#### Class octopus.RhythmPattern

In the Octopus Music API the rhythmic line is defined independently of the *Melody* or *Harmony* and it is represented by the *RhythmPattern.*

Both the *Melody* and the *Harmony* can be linked to *RhythmPattern.* In *Melody*, the *RhythmPattern* is mapped to the *Notes* while in *Harmony* it is mapped to the *Chords*. If the *ChordNotes* required individual rhythmic manipulation (different start time and/or duration) then the *Arpeggio* Class must be used, as explained in the next section.

The *RhythmPattern* is composed of *Bars*, *Marks* and *Returning* *Points*. The *Bars* are inserted sequentially so the order of input must be observed. Like the *Bar*, a *Mark* is placed in a certain position of the *RhythmPattern*. Every time that a *ReturnPoint* is reached, the pointer goes back to its respective *Mark*. This loop lasts while the number of repetitions specified in the *Return Point* is not achieved.

Figure 67 shows the internal structure of a *RhythmPattern*. In this particular example there is a returning point placed after the third *Bar* that goes back to *Mark* M1 three times. The Code Example 3 defines the *RhythmPattern* illustrated by Figure 3.

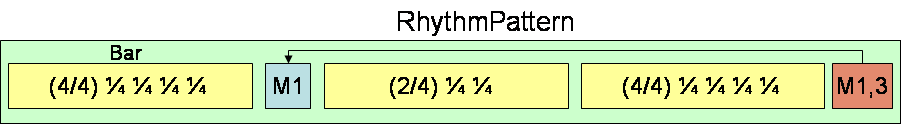


Figure 67: *RhythmPattern* internal data structure.

|  |
| --- |
| RhythmPattern rhythmpattern = new RhythmPattern();  Bar bar1 = new Bar(4,4);  bar1.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar1.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar1.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar1.addRhythmEvent(Bar.QUARTER\_NOTE,1);  Bar bar2 = new Bar(2,4);  bar2.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar2.addRhythmEvent(Bar.QUARTER\_NOTE,1);  Bar bar3 = new Bar(4,4);  bar3.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar3.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar3.addRhythmEvent(Bar.QUARTER\_NOTE,1);  bar3.addRhythmEvent(Bar.QUARTER\_NOTE,1);  //placing the bar and setting the return point  rhythmpattern.insertBar(bar1);  rhythmpattern.insertMark("M1");  rhythmpattern.insertBar(bar2);  rhythmpattern.insertBar(bar3);  rhythmpattern.insertReturn("M1",3); |

Code Example 4: RhythmPattern.

#### Class octopus.Arpeggio

An *Arpeggio* is a set of *RhythmPatterns* played simultaneously; it is used to spread the notes of the *Chord* throughout its overall duration (voicing). Often the *Arpeggio* information is omitted in more popular musical notations (i.e. guitar tablature) and its use varies upon to the technique and expressivityof the *Performer*.

Inside the *Arpeggio*, the *RhythmPatterns* (called voices) are organised in vertical parallel lines, as seen Figure 68. The lowest voice (index 0) is linked to ChordNote with the lowest pitch, the second lowest to the second lowest ChordNote pitch and so on.

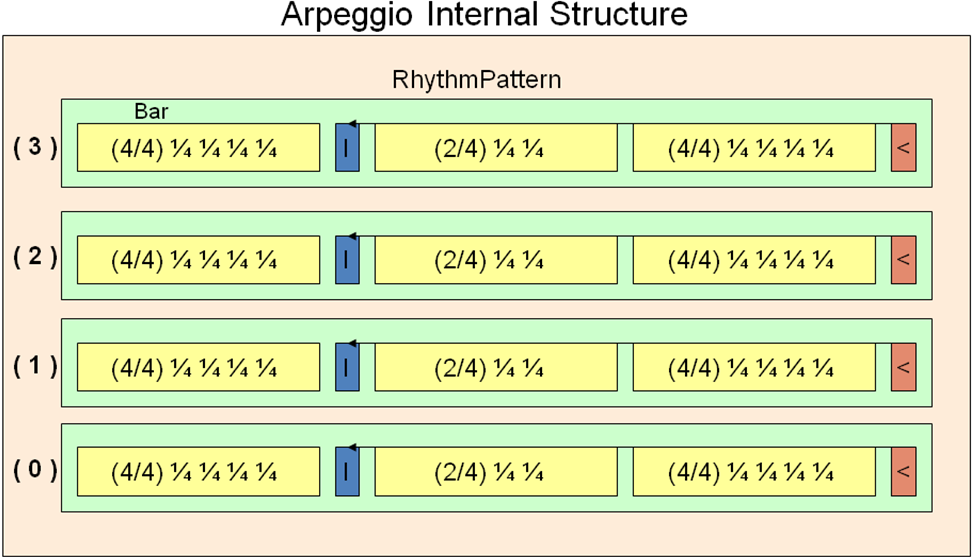


Figure 68: Class Arpeggio internal data structure.

When a *Musician* (Musical Data Interpretation Class) is requested to play a *Harmony* using a particular *Arpeggio*, it will adapt the *Arpeggio* to the *Harmony*, repeating or stretching its duration to match the duration of the *Chords*. Figure 69 illustrates the ‘time stretching’ feature. Note that even though the *C Chord* uses the same Arpeggio as *F* and *G chords,* its duration is twice as long.



Figure 69: Arpeggio time-stretching feature applied to the Harmony.



Code Example 5: Arpeggio.

The Code Example 5 shows an example of an *Arpeggio* with 4 voices, each of them composed of a 4/8 *Bar*.

#### Class octopus.Scale

*Scale* is a set of *Notes* that maintains a pre-determined interval (mode) between them. The *Scale* class allows the programmer to automatically instantiate several notes of a diatonic (Code Example 6) or pentatonic scale, which could later be used as melodic fragments.

|  |
| --- |
| Scale s = Scale.getDiatonicScale(NoteFactory.getNote(“B”),  Scale.MODE\_MAJOR); |

Code Example 6: B Major diatonic scale.

#### Class octopus.HarmonicProgression

*A HarmonicProgression* is to *Chords* what a *Scale* is to N*otes*. It is a set of *Chords* generated according to the degrees (given by Roman numeral) of a user-defined harmonic progression and its key.

|  |
| --- |
| HarmonicProgression harmonicprogression = new HarmonicProgression("I -ii -V7");  harmonicprogression.addScaleDegree("I");  harmonicprogression.addScaleDegree("ii");  harmonicprogression.addScaleDegree("V", IntervalFactory.getMajorSecond());  Chord[] chords = harmonicprogression.getChords(NoteFactory.getC()); |

Code Example 7: HarmonicProgression in C (key) composed by the chords respectively represented by the tonic, supertonic (minor), and dominant seventh degrees.

The *HarmonicProgression* class allows the programmer to automatically instantiate several *Chords* following a harmonic structure in any key. Code Example 7 shows how to model a Jazz harmonic progression in the key of C.

#### Class octopus.Melody

*Melody* is a set of *Notes* played sequentially according to a certain *RhythmPattern*. The Code Example 8 creates a simple melody.



Code Example 8: Melody.

#### Class octopus.Harmony

*Harmony* is a set of *Chords* played sequentially according to an overall *RhythmPattern* but respecting the *Arpeggios assigned to each chord*.

If an *Arpeggio* is not assigned to a *Chord* then all *Notes* of the *Chord* will sound simultaneously and lasts for the duration of the *Chord*. The duration of each *Chord* is the same as the *RhythmEvent* associated with it, as previously illustrated in Figure 69.

|  |
| --- |
| // Instantiate a harmony object with a demo RythmPattern ;  Harmony harmony = new Harmony(RhythmPattern.getDemoRhythmPattern());  // Creates the chords of the harmony  Chord[] chords = new Chord[2];  chords[0] = Chord.getChord("C");  chords[1] = Chord.getChord("F");  Chord chord = Chord.getChord("G");  // Creates the arpeggio to the vector of chords.  Arpeggio arpeggio1 = ArpeggioLibrary.getDemoArpeggio();  arpeggio1.setTimeStratch(true);  //Creates the arperggio for G chords  Arpeggio arpeggio2 = ArpeggioLibrary.getDemoArpeggio2();  //Assing the chords to the harmony.  harmony.addChord(chord, arpeggio1);  harmony.addChord(chords, arpeggio2); |

Code Example 9: Coding a *Harmony* with 3 Chords that uses two different *Arpeggios*.

Code Example 9 shows how to code a *Harmony* composed of three chords and two different Arpeggios. Note that there is no information on the *Harmony* regarding the fingering of C*hords* (i.e. chord shapes). This knowledge belongs to the *Performer*.

#### Class octopus.Music

Music is normally expressed in terms of [pitch](http://en.wikipedia.org/wiki/Pitch) (i.e. [melody](http://en.wikipedia.org/wiki/Melody)), [rhythm](http://en.wikipedia.org/wiki/Rhythm) (i.e [tempo](http://en.wikipedia.org/wiki/Tempo)), and the quality of sound which includes [timbre](http://en.wikipedia.org/wiki/Timbre), [articulation](http://en.wikipedia.org/wiki/Articulation_%28music%29), [dynamics](http://en.wikipedia.org/wiki/Dynamics_%28music%29), and [texture](http://en.wikipedia.org/wiki/Texture_%28music%29) (Dannenberg, 1993).

In the Octopus Music API, *Harmony* and *Melody* (both containing rhythmic information) are implementations of the *octopus*.*MusicalComponent* abstracted class.

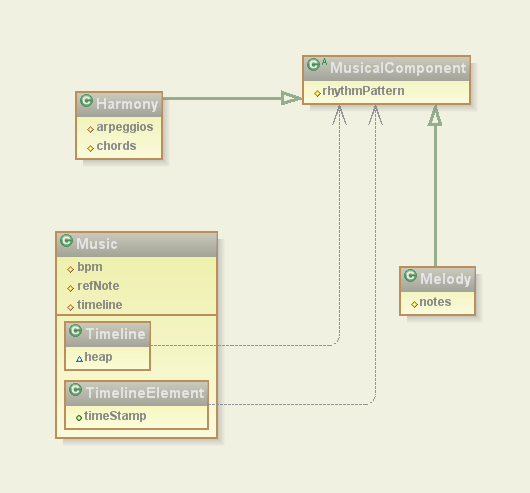


Figure 70: Class diagram for the *MusicComponents*.

*Music* is a set of *MusicalComponents* scheduled in time. The timeline is represented by the *Music.Timeline* internal class, which is responsible for indexing the *MusicalComponents* throughout the duration of the *Music*. A class diagram showing the relationship between the Music and the *MusicalComponents* is presented in Figure 70.

Figure 71 shows an internal representation of *Music* composed with 3 *Harmonies* and 3 *Melodies*. Note that H1 and M3 start in the beginning of the music; hence the index is zero for both objects in the timeline. The same does not happen to H2 and M1 which was scheduled to start in a latter time requiring exclusive pointers for them in timeline table.

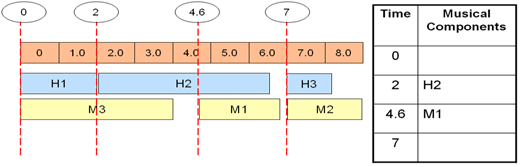


Figure 71: Internal musical components organisation over the time.

|  |
| --- |
| //Create and empty music object  Music music = new Music();  //Generate the chords based of on a harmonic progression of I -ii -V7 in C major.  HarmonicProgression harmonicprogression = new HarmonicProgression("Jazz");  harmonicprogression.addScaleDegree("I");  harmonicprogression.addScaleDegree("ii");  harmonicprogression.addScaleDegree("V", IntervalFactory.getMajorSeventh());  Chord[] chords = harmonicprogression.getChords(NoteFactory.getC());  //Create a demo RhythmPattern  RhythmPattern rhythmPattern = RhythmPatternLibrary.getConstantRhythmPattern();  //Assign the Chords and the RhythmPattern to the harmony  Harmony jazzyHarmony = new Harmony(chords,rhythmPattern) ;  //Create a melody based on the the freeSoloNotes array;  String[] freeSoloNotes = ;  Melody melody = new Melody(freeSoloNotes,rhythmPattern);  //Insert the MusicalComponents. Harmony starts in the beginning followed by harmony.  music.insertMusicalComponent(jazzyHarmony,0.0);  music.insertMusicalComponent(melody,jazzyHarmony.getDuration()); |

Code Example 10: Creating a Music with a harmony from a HarmonicProgression and “free notes” melody.

Code Example 10 demonstrated how *Music* can be ‘assembled’ using a Harmonic Progression in C Major and free notes soloing. Note that the melody will only start after the Harmony has finished, with the timestamp returned by *jazzyHarmony.getDuration* method.

### Musical Data Interpreters

A satisfactory realisation of an encoded work can be reconstituted through the interpretive practice of trained performers, but the knowledge that enables human performers to interpret music notation is extremely difficult to represent in a formal way (Sundberg, 1980).

Historically, the Western musical tradition has developed what we now refer to as Common Music Notation (CMN) to provide a written representation of musical compositions.

One of the key problems is that music notation is not just a mechanical transformation of performance information. Performance nuance is lost going from performance to notation, and symbolic structure is lost in the translation from notation to performance. It seems that music notation rules are made to be broken (Dannenberg, 1993) even if it was designed to serve the needs and processing abilities of humans (Loy and Abbott, 1985).

As previously mentioned, the Octopus Music API has its focus on the modelling of elements involved in a musical performance, mainly on the *Performer* and its *Instrument*. However, rather than coding each action involved in the performance, the *Performer* was programmed to interpret the Musical Structures by itself. In other words, the programmer does not have to code every minor detail as illustrated by the B.B Queen example. Instead, the Musical Data Interpreters classes are used to play the Musical Data Structures.

Mathews (1970, p.272) once said that:

“The desired relationship between the performer and the computer is not that between a player and his instrument, but rather that between the conductor and his orchestra”.

If the above quote is true, then who should be responsible to add expressivity and interpretation to a musical piece? The conductor may be responsible for directing the overall mood of the piece but surely he can not oversee every single action of the performers. In addition, as seen in chapter 2, performers have their own means of interpretation.

We believe that the interpretation of the musical information varies according to the knowledge of the *Musician* and the musical context where it has been applied. A *Musician* does not need to play a musical *Instrument* to be able to understand music. In the same way, a percussionist is not less of a *Musician* for concentrating more on the rhythmic aspect of the music and less on the melody or harmony.

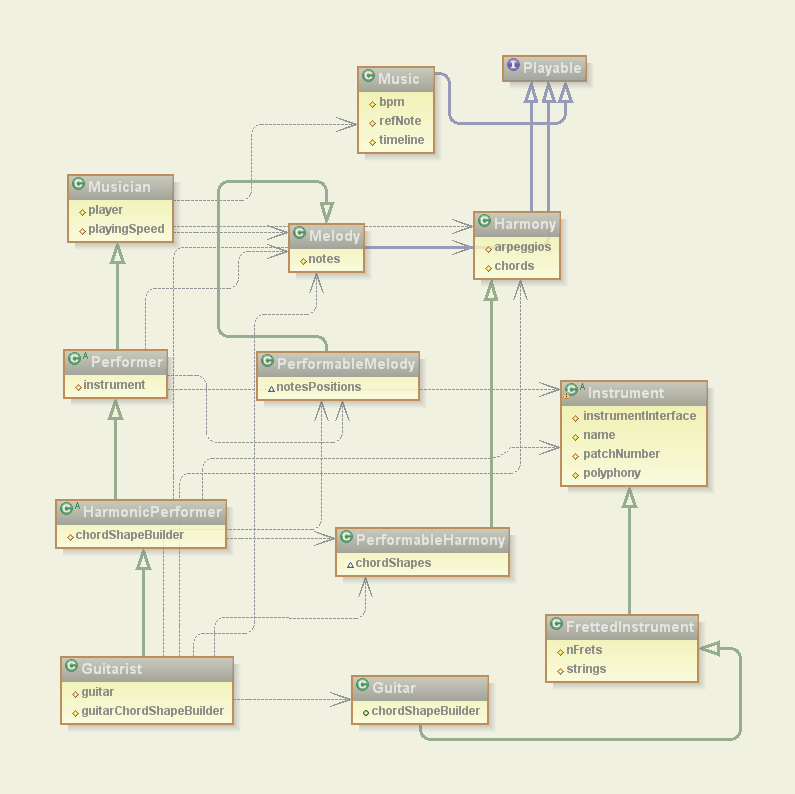


Figure 72: Class diagram for the Musical Data Interpreters.

Figure 72 shows the class diagram of the Musical Data Interpreters interacting with the Musical Data Structures in different levels. The first and most basic level of interpretation is the *Musician* with musical knowledge but who does not know anything about playing an instrument.

The Performer inherits all the skills of the *Musician* and implements some general monophonic instrumental knowledge. *HarmonicPerfomer* inherits all the skills of its *superclasses* and adds polyphonic instrument handling (*Harmony*). The *Guitarist* is a *HarmonicPerformer* with expertise in *Guitar* playing.

#### Class octopus.Musician

The *Musician* is basically an interpreter of the playable musical structures such as: *Scale, Melody, Harmony, Music, RhythmPattern* and so on. He knows how to read and play these structures in the simplest possible way. No instrument restriction is considered in this computation.

#### Class octopus.instument.Performer

The information contained in a traditional score is interpreted in performance time according to a set of rules that is formally incoherent, known as performance interpretation (Loy and Abbott, 1985).

A *Performer* uses the rules of musical interpretation to reconstitute an acceptable facsimile of the musical idea during performance (Loy and Abbott, 1985). This suggests that Performers actually extends the knowledge of the Musicians with contextualised information about the *Instrument*.

As a subclass of *Musician*, *Performers* are also capable of interpreting musical structure but they have to adjust these *MusicalComponents* to the characteristics of its *Instrument*. For instance, when a *Guitarist* plays a *Harmony* he will play it respecting the limitation of the specific *Guitar* that is being used, which may sound slightly different when ‘played’ by the *Musician*, although a *Guitarist* is ultimately a *Musician*. This is known as polymorphism in the OOP paradigm.

All performers are able to play *Melody* but the same is not true in regards to the *Harmony*. A *Performer* that is able to play *Harmony* is represented by the class of *HarmonicPerformer*. Both *Performer* and *HarmonicPerformer* are abstract classes, so they need to be specialised for a particular *Instrument*.

As previously explained, when a *Performer* is asked to play *Music* it adjusts the musical information to its particular instrument. This ‘learning’ process generates enriched versions of the Harmony and Melody musical components, respectively represented by the classes *PerformableMelody* and *PerformableHarmony.* These two classes contain all the information relating to the performance of the *Music* using a particular *Instrument* such as: articulation, plucking point, chord shapes, et cetera.

#### Class octopus.instument.fretted.Guitarist

The *Guitarist* is a *HarmonicPerformer* that knows how to play *Guitar.*

*Since the Guitar* used in the performance *can have* direct influence in the way the *Music* is played, the *Guitarist* needs to ‘know’ beforehand which *Guitar* they will be playing. In OOP terms, the *Guitar* needs to be informed in the *Guitarist* constructor. Code Example 11 demonstrates a *Guitarist* being instantiated.

|  |
| --- |
| …  Guitar guitar = new Guitar(); //create a classical guitar  Guitarist BB\_Queen = new Guitarist(guitar); //create the guitarist and assign the guitar to it  //request the performer to show the InstrumentGraphicalInterface whislt playing.  BB\_Queen. showInstrumentLayout();    BB\_Queen.play(music) //play the music |

Code Example 11 Guitarist instantiation.

As a *HarmonicPerformer*, the *Guitarist* knows to how to play a *Chord*. As previously stated, a *Chord* can be played in different regions of the guitar (*ChordShape*), using different fingering and *Arpeggios*. This knowledge was programmed into the Guitarist using a chord shape similarity function.

The similarity function compares the previous chord shape with the candidates chord shapes of the next chord in the harmonic sequence. In theory, the more similar the chord shape is to the previous chord, the lower the effort to move from one to the other (travel-cost). The similarity function returns a value between 0 and 1, where 1 means the same chord (Costalonga and Miranda, 2006). The similarity functions used in this work are given by equations system bellow:

|  |  |
| --- | --- |
|  | (1) |
|  |  |
|  | (2) |
|  | (3) |
|  | (4) |
|  | (5) |

Equation 1 represents a matrix of *m x n* elements where m = number of positions in the chord shape and n = number of positions in the next chord shape. Similarities of each position are given by Equations (2) and (3), where e = finger span value. Equation (4) is used to calculate the distance in non-open chords whilst Equation (5) is used with open chords, where qtOS represents the number of open strings used in the chord shapes.

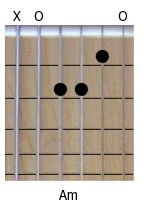
To exemplify, consider an Am chord shape (Figure 73) composed by the positions[string,fret] [(5,0),(4,2);(3,2);(2,1);(1,0)] which is going to be followed by a G chord. The first step is to find candidate chord shapes for the G chord. Some of them are: [(6,3);(4,0);(2,0);(6,3);(4,0);(3,4)], [(6,3);(5,2);(4,0);(5,10)], [(4,9);(1,10);(6,3);(3,4);(2,3)].

Figure 73: Am chord shape.

The black dots represent the position the fingers should be placed in. The hollow circle marks indicated the sting that must he plucked ‘openly’ and ‘x’ the string that should not.

Every candidate chord shape for the G chord will be compared with the Am chord shape, generating a matrix as shown in Table 11.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **G\A** | **-** | **(5,0)** | **(4,2)** | **(3,2)** | **(2,1)** | **(1,0)** |
| **(6,3)** | - | .31 |  |  |  |  |
| **(4,0)** |  | .31 | .62 | .32 |  |  |
| **(2,0)** |  |  |  | .31 | .62 | .31 |

Table 11: Am to G similarity calculation.

For every position of G (e.g. pos (6,3)), a value is calculated in relation to the positions of Am at the same string (Equation 3), one string above and one string below (Equation 4). For example, the position (4,0) (4th string open) of the chord G will be compared with the positions of the 5th and 3rd strings of Am chord shape, respectively position (5,0) and position (3,2).

The similarity between the chords is given by the average of the highest values of each row. In this example, the similarity is (0.31 + 0.62 + 0.62)/3 = 0.52. The candidate chords are then sorted by similarity.

Similarity is the main criteria for the selection of a chord shape but not the only one. The chord shape must also comply with the rhythmic pattern that is being used. For instance, in case of a strumming, the chord shapes with open string notes that do not belong to the basic structure of the chord must be ignored.

Obviously, the first chord shape can not be compared. Therefore, the selection is made on the basis of the proximity of chord shape to the guitar head (the lowest fret average values).

It is important to highlight that even though the similarity function used by the current implementation of the Guitarist is backed up by the biomechanical theory travel cost proposed by Rosenbaum (1995), it is not derived from any data collected from the experiments described in chapter 4. This topic will be discussed in the second part of this Chapter.

The biomechanical study of the guitarist’s right-hand (pluck hand) is not in the scope this research. Nevertheless, the Octopus does provide two classes to allow a detailed formalisation of the right-hand techniques (arpeggios): *GuitarBar* and *GuitarArpeggio (*Code Example 12*)*, respectively extending *Bar* and *Arpeggio.*

|  |
| --- |
| GuitarArpeggio gpr = new GuitarArpeggio(4);  gpr.setBpm(240);    GuitarBar bs1 = new GuitarBar(4,4);  bs1.addSingleRhythmEvent(bs1.WHOLE\_NOTE,Bar.RHYTHM\_EVENT\_NOTE,1,0,127,  GuitarBar.FINGERPICKING\_THUMB\_FINGER);  gpr.insertBar(bs1,0);    GuitarBar bs2 = new GuitarBar(4,4);  bs2.addSingleRhythmEvent(bs1.HALF\_NOTE,1,2 ,bs1.DIRECTION\_UP\_STROKE,bs1.REGION\_INDEX\_FINGER,  bs1.FINGERPICKING\_INDEX\_FINGER);  gpr.insertBar(bs2,1);  GuitarBar bs3 = new GuitarBar(4,4);  bs3.addSingleRhythmEvent(bs1.HALF\_NOTE,1,3,bs1.DIRECTION\_UP\_STROKE,  bs1.REGION\_MIDDLE\_FINGER,  bs1.FINGERPICKING\_MIDDLE\_FINGER);  gpr.insertBar(bs3,2);  GuitarBar bs4 = new GuitarBar(4,4);  bs4.addSingleRhythmEvent(bs1.QUARTER\_NOTE,1,4,  bs1.DIRECTION\_UP\_STROKE,  bs1.REGION\_RING\_FINGER,  bs1.FINGERPICKING\_RING\_FINGER);  gpr.insertBar(bs4,3); |

Code Example 12: Guitar Arpeggios.

The slightest of the variations in the strokes of a *Guitar* *Arpeggio* is enough to create a whole new *Arpeggio*. Some of the stroke’s properties are: direction of the stroke, fingerstyle (*PIMA*) or pickstyle modes, plucking point, plectrum hardness and shape, body slap region, percussive muting, string slap intensity, and plectrum attack angle.

The programmer has the option to write the *GuitarArpeggio* himself *(*Code Example 12*)* or let the Guitarist automatically ‘learn’ the Arpeggio. The automatic conversion of the *Arpeggio* into *GuitarArpeggio* takes into consideration if the *Arpeggio* is meant to be strummed, arpeggiated, played with a plectrum or using the fingers. The default is the Classical Finger Style Arpeggio (PIMA).

At the moment, the *Guitarist* is the only full implementation of a *Performer* in the API but this does not mean that the Octopus API can only be used to model guitar performances. The priority for the implementation of the *Guitarist* is in accordance with the ultimate goal of our research. Nevertheless, the extension to the *Performer* class to embrace other *Instruments* would not be difficult for a Java programmer.

### Instrument Classes

In the real world, musical instruments are classified by different criteria such as the note range or the way they generate the sound (what vibrates in the instrument to produce the [sound](http://en.wikipedia.org/wiki/Sound)). For example, in an orchestra the instruments are split into woodwind, [brass](http://en.wikipedia.org/wiki/Brass_instrument), [percussion](http://en.wikipedia.org/wiki/Percussion_instrument), strings.

Hornbostel-Sachs (or Sachs-Hornbostel) is a system of [musical instrument classification](http://en.wikipedia.org/wiki/Musical_instrument_classification) devised by [Erich Moritz von Hornbostel](http://en.wikipedia.org/wiki/Erich_Moritz_von_Hornbostel) and [Curt Sachs](http://en.wikipedia.org/wiki/Curt_Sachs) (von Hornbostel and Sachs, 1961), and first published in the Zeitschrift für Ethnologie in 1914. It is the most widely accepted system for classifying [musical instruments](http://en.wikipedia.org/wiki/Musical_instruments) by [ethnomusicologists](http://en.wikipedia.org/wiki/Ethnomusicology) and [organologists](http://en.wikipedia.org/wiki/Organology) (Lysloff and Matson, 1985).

The Hornbostel-Sachs system is based on one devised in the late [19th century](http://en.wikipedia.org/wiki/19th_century) by [Victor Mahillon](http://en.wikipedia.org/w/index.php?title=Victor_Mahillon&action=edit), the curator of [Brussels](http://en.wikipedia.org/wiki/Brussels) Conservatory's musical instrument collection. Mahillon's system was the first to classify musical instruments based on what vibrated to produce its [sound](http://en.wikipedia.org/wiki/Sound); however this system was limited to western instruments used in [classical music](http://en.wikipedia.org/wiki/European_classical_music). The Hornbostel-Sachs system is an expansion on Mahillon's in which it is possible to classify any instrument from any culture.

This API adopted the way a *Performer* interacts with the *Instrument* as the classification criteria, which resembles the Sachs-Hornbostel system. However, instead of classifying the instruments based on how they produce the sound (what vibrates), we classify how the performer manipulates the instrument in order to produce the sound with it.

For instance, the ergonomics of string-fretted instruments are very similar. It does not matter whether it is an acoustic classical guitar or a mandolin. The way a performer (normally) interacts with these instruments in order to produce the sound is by stopping the strings against the fingerboard, consequently changing the effective length of the strings, which in turn changes the frequency at which the [string vibrate](http://en.wikipedia.org/wiki/Vibrating_string)s when plucked.

This form of categorisation is useful in the context of this thesis because it favours the reuse of code. For example, the skills to play a Guitar are quite different from the skills to play a piano but it is very similar to the skills to play a mandolin, since both are from the Lute family. Therefore, once an instrument of the Lute family is modelled, the rest demand little effort to be modelled.

#### Class octopus.instument.Instrument

The abstract class *Instrument* establish the minimum requirements that a new *Instrument* must implement in order to be able to interact with the other classes of API. This assures the scalability of the API to contemplate new *Instruments* during its development.

For example, all *Instruments* of the API must implement an *InstrumentGraphicalInterface* that basically provides a visual feedback of the performance. Figure 74 shows an implementation of such graphical interface for the Guitar (class *GuitarGraphicalInterface*).

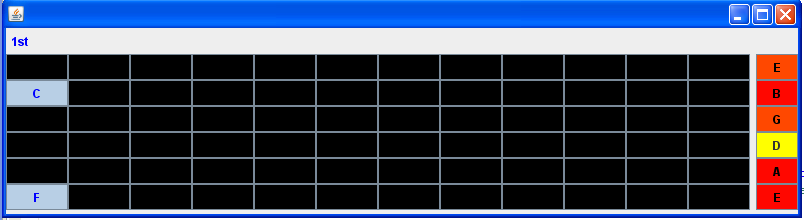


Figure 74: I*nstrumentGraphicalInterface*: a graphical interface of the Guitar class.

The 6 rows of the matrix represent the string and the columns the frets. On the right-hand side, the labels show the strings open tuning. The darker the red, harder the string has been plucked.

#### Class octopus.instument.string.fretted.FrettedInstrument

The *FrettedInstrument* Class represents the category of *Instruments* of the Lute family. Most plucked string instruments belong to the [Lute](http://en.wikipedia.org/wiki/Lute) family (such as [guitar](http://en.wikipedia.org/wiki/Guitar), [bass guitar](http://en.wikipedia.org/wiki/Bass_guitar), [mandolin](http://en.wikipedia.org/wiki/Mandolin), [banjo](http://en.wikipedia.org/wiki/Banjo), [balalaika](http://en.wikipedia.org/wiki/Balalaika), [sitar](http://en.wikipedia.org/wiki/Sitar), and [pipa](http://en.wikipedia.org/wiki/Pipa)). The lute refers to plucked [string instrument](http://en.wikipedia.org/wiki/String_instrument)s with a fretted [neck](http://en.wikipedia.org/wiki/Neck_%28music%29) and a deep round back (Lysloff and Matson, 1985).

The *Guitar* class is a subclass of *FrettedInstrument* with an overridden constructor to model an acoustic classical guitar with 6 strings the standard tuning (E, A, D, G, B, E) and 12 frets clear frets.

### Communication Classes

Once a musical representation is adopted, issues of transmitting and storing the representation arise. Transmission, especially in real time, raises questions of network protocols, the conventions by which information is transmitted and received. Storage raises the question of coding, or how the abstract information is converted into specific bit patterns.

MIDI is the most prevalent protocol for the real-time transmission of music information, but it has many weaknesses (Dannenberg, 1993). In a conventional MIDI setup the controller interface (e.g. instrument) might not incorporate the Sound Generator Unit (e.g. synthesiser) itself, meaning they are two distinct devices. This separation makes the architecture more flexible once the units can be replaced to better suit a particular musical task. For example, a guitarist will most certainly find it easier to play a Guitar-shaped MIDI controller than a keyboard, which is traditionally the preferred MIDI controller.

Figure 75 illustrates a guitar-like MIDI controller sending MIDI messages to an independent external synthesiser that convert the MIDI messages into analogical wave signals that is passed to the speakers.

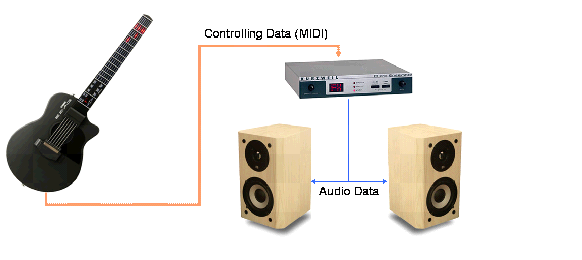


Figure 75: Conventional MIDI setup.

The specialisation of the units into different devices provides great flexibility but also draw attention to a fundamental element of the setup: the communication protocol that interfaces the units. The communication protocol must guarantee that the musical information generated in the controller is understood by the Sound Generation Unit, so it can render the sound accordingly.

Every musical instrument has its own means of producing sound (the mechanics of the instrument). Likewise, performers use different techniques when playing an instrument. So, how is it possible to have standard protocol that contemplates all instruments and playing styles? It is not. The MIDI was designed for keyboard instruments and this simplification costs the expressiveness of performers of other types of instruments. In summary, MIDI sequencers will always treat musical material identically regardless of the instrument.

It is important to clarify that the Octopus API does not aim to be a synthesizer or a communication protocol; instead, it does provide the programmer a way to communicate with external Sound Generation Units.

Normally, musical software would make direct use of programming language libraries to gain access to MIDI or audio devices. This option can be tricky since it involves low-level programming skills. Additionally, it would require the programmer to be familiar with every little detail of the Musical Data Structures classes, which goes against the concept of encapsulation of the OOP paradigm. This, however, is an inconvenience and not the main problem.

The main problem is that there is no commercial synthesizer capable of rendering expressive performances to the level of detail the Octopus API allows one to model them, as seen in Section 2.5 (p. 55). Consequently, there is no protocol either. So, what communication protocol should be used in the Octopus API to communicate with a Sound Generator Unit that does not even exist?

The solution found was the creation of a middle-tier layer that contains all the performance information but only transmits what the Sound Generation Unit is able to handle using whatever communication protocol it supports. This is where the *InstrumentGraphicalInterface* comes into play, providing visual feedback of what is not yet possible to sound.

#### Class octopus.communication.MusicalEvent

Musical data can be either continuous or discrete. Continuous information changes over time and is typically represented by digital sampling, splines, or arbitrary mathematical functions. In contrast to continuous data that fills time intervals, discrete information usually represents events at a point in time (Dannenberg, 1993).

These events are usually related to the production of a sound, but the parameters or sub actions involved in this activity vary from *Instrument* to *Instrument* and also from the capability of the synthesizer to process it.

What should a synthesizer know about a *Guitar* music performance? Is it the same knowledge necessary to render a saxophone performance? A subset of actions might be the same, but not all. How is it possible to tell the Sound Generation Unit that the force applied by the guitarist’s index finger is not enough to render a clean note or that the string was plucked using a triangular plectrum in an upward movement?

Even a synthesiser specialised in a particular instrument may not be able to produce all the sound nuances that performers can produce with real instruments. Whilst the shape and the hardness of the pick could be relevant information for a particular guitar synthesiser, to another it could be pointless.

The point we are trying to make is that in the real world the performer interacts freely with a musical instrument, even in non-musical ways. Who has not heard of the famous incident of Jimi Hendrix setting fire on his guitar 1967 during a concert in Finsbury Park Astoria?

In order to provide a way to describe any possible action the instrumentalist might want to perform, the Class *MusicalEvent* extends the *java.util.Properties*. In essence, it is an unlimited collection of attributes that describes the musical actions in any level of detail that the synthesizer requires to produce the sound with fidelity. Some of the parameters appear quite often so it was decided to make them permanent class attributes. They are: note, duration, timing and velocity

#### Class octopus.communication.MusicalEventSequence

Musical tasks can be described as a sequence of simple actions with specifiable goals (Pennycook, 1985). A *MusicalEvent* is no more than a single atomic musical task. An entire musical piece is likely to need more than one *MusicalEvent*.

The *MusicalEventSequence* is an array of *MusicalEvents*. It is the artefact that the *Musician* generates when requested to play something because it is the synthesizer that actually plays the *Music* through the *ShynthesizerController* class.

The *MusicalEventSequence* can also be used to group *MusicalEvents* in order to process them all together. For example, add delay in all the events of the sequence.

#### Class octopus.communication.SynthesizerController

The *SynthesizerController* is a parser from the *MusicalEventSequence* to whatever protocol is used to control the synthesizer. The standard protocol supported by *Java* is still MIDI, so *MidiSynthesizerController* realises the interface SynthesizerController to parse *MusicalEvent* into MIDI messages.

The *MidiSynthesizerController* is able to communicate with external devices through the MIDI ports. In Java, this is achieved by connecting transmitters (MIDI OUT) to receivers (MIDI IN), just like you would do it with a cable.

To deal with the particularities of a guitar performance, a *GuitarMidiSynthesizerController* was designed to convert guitar performance information into MIDI messages. This conversion implies loss of precision because, as previously explained, the MIDI was not designed for guitar usage.

Justice needs to be done: MIDI would not have been the standard communication protocol for digital musical instruments for the last 20 years if it was not a well designed solution. To overcome lack of native support for some instruments, the MIDI specification proposes the use of System Exclusive Messages (SysExMessages) to extend the functionalities of the protocol to specific devices/manufactures.

The GuitarMidiSynthesizerController makes wide use of SysExMessages to communicate all the performance actions. However, this is only helpful if the device (receiver) could interpret these messages. Unfortunately, such a device does not yet exist yet.

In order to verify the functioning of the solution, we implemented the *GraphicalGuitarMidiReceiver* class. This class is used to decode the SysExMessages and provide a visual feedback using the GuitarGraphicalInterface (Figure 74, p. 205).

### Final Considerations of the Octopus Music API

On March 2007 the first beta version of Octopus Music API was made available under Academic Free License at http://sourceforge.net/projects/octopusmusic/.

Up to this moment, 277 downloads are registered (Figure 76). This shows that despite lack of support for users, there is a continuous interest of the community for a tool like Octopus Music API.

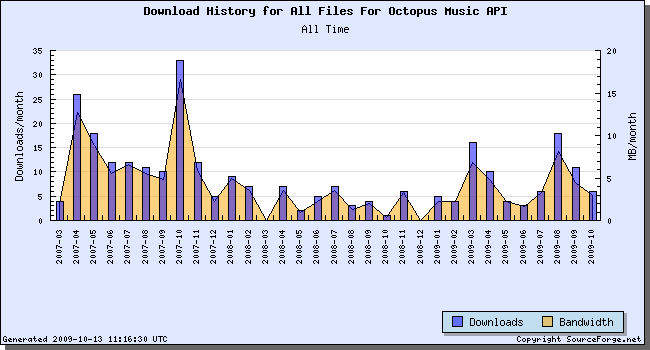


Figure 76: Octopus Music download figures.

The x-coordinates show the date and the y-coordinates the monthly downloads

We believe that the Octopus Music API is an important step towards a successful modelling of music performance, not only for this particular research, but for any other work that requires manipulation in low-level of performance actions, which by itself it is a significant contribution to the state of the art.

The version currently available has all the functionalities presented in this chapter so far. It does not contemplate any of the machine learning algorithms used to model speed, precision and force of the guitarist during a performance. This will be made available in a major update due to be released after the conclusion of this Thesis.

1. Extended instruments are acoustic instruments equipped with sensors and other customised electronic components designed to extend the instrument capabilities. [↑](#footnote-ref-1)