Chapter 1

An Introduction To Sound

In this chapter we will present an uncommon facet of Pharo: sound synthesis. Pharo offers a nice library to create and play various sounds. As soon as we need to put some sound in an application one think either to mp3 or sampled sounds. In Pharo we get several tools and libraries to create synthetized sounds from scratch as well as to play samples. We will show how we can do that with a couple of lines of code. This version is an english translation of an original french version. The author thanks Kilon for his first translation of this chapter.

1.1 Getting Started

Getting Pharo

First we need to install the latest version of Pharo from http://www.pharo.org/. The easiest method is to run the following in a shell:

```
curl get.pharo.org/30 vmLatest + | bash
```

or (if curl is not available):

```
wget−O−get.pharo.org/30 vmLatest + | bash
```

Other scripts are available for download at http://get.pharo.org. The GUI starts with /.pharo-ui pharo.image

Loading Sound

Once Pharo installed, you must download the package PharoSound using:

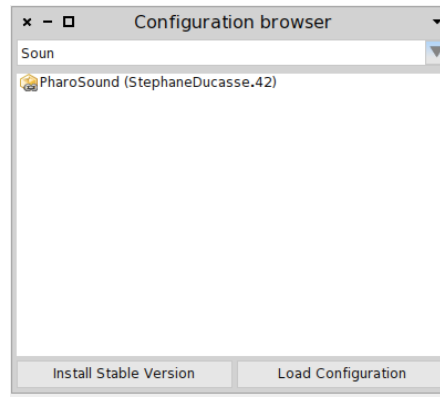

Figure 1.1: Configuration browser: a package manager

- either the ConfigurationBrowser (as shown in Figure) which is found in the Tools menu. Enter Sound in the text box to select the package, then press the 'Install Stable Version' button.

- or Gofer the package manager:

```
Gofer new
   smalltalkhubUser: 'PharoExtras' project: 'Sound';
   package: 'ConfigurationOfPharoSound';
   load.
(#ConfigurationOfPharoSound asClass project version: #stable) load
```

Setup verification

We now will verify that our environment can play a sound properly. To do this simply open a workspace (from the Tools menu) and evaluate the following line of code by selecting the expression and then using the 'Do it' menu (we will explain that later). Note that, in a workspace, to run a line it is enough to move the mouse cursor and press alt-d (this is equivalent to selecting the line and using the menu). To select a block of instructions quickly position the mouse at the beginning or at the end of the block and click, this has the effect of selecting all.

```
FMSound organ1 play.
```

If you have sound you can skip the next paragraph, if not here are some ways to get sound working.

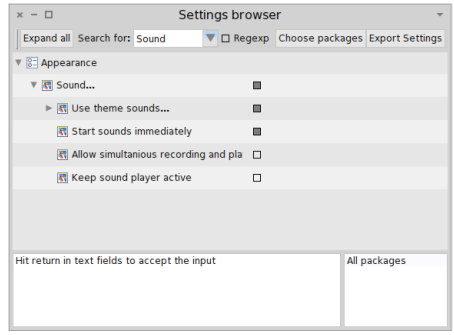

Figure 1.2: Settings

- Enable sound in Pharo. Check that the sound setting is enabled in your Pharo. To set it, open the Settings Browser from the Tools menu. Then type sound in the bar and click the radio buttons to get close to the image shown in Figure .

- The volume of your system is too low or muted
- Pulseaudio is enabled (on virtually all current distributions)

By default the Linux archive does not carry a pulseaudio plugin, only the Alsa plugin is available. We will disable the sound server temporarily. It should not be restarted automatically when we stop.

1. Create the file /home/[user]/.pulse/client.conf if it does not exist
2. Add parameter autospawn = no
3. Delete the current process with the command pulseaudio -k from your user account. To restart when you want, run the command pulseaudio -D.

- Restart the internal sound player process. It may also happen that the sound process no longer works. You can use the following commands to reset

```
SoundPlayer stopPlayerProcess.
SoundPlayer Startup
```

Here's a little tip before we go any further with sound rendering. Default reverb is added to the sound played. Often this adds a pleasant softness to the rendering, but in our case it can also distort what we hear.

To disable rendering with reverb, run the command

```
SoundPlayer stopReverb
```

Or restart if you prefer with

```
SoundPlayer startReverb
```

1.2 FM synthesis basis by example

Now that we have verified that the sound system is functional, we will explore how it works. Consider the method that defines the instrument organ1:

```
FMSound class>>organ1
   "FMSound organ1 play"
   "(FMSound majorScaleOn: FMSound organ1) play"

   | sound |
   sound := self new.
   sound addEnvelope: (VolumeEnvelope
                        points: { 0@0 . 60@1.0 . 110@0.8 . 200@1.0 . 250@0.0}
                        loopStart: 2 loopEnd: 4).
   sound setPitch: 440.0 duration: 1.0 loudness: 0.5.
   ^ snd
```

Let's break this method down:

First, it creates a new instance of the class FMSound using the message new as follows:

```
   sound: = self new.
```

It then defines a set of points to create a volume envelope. For each point, the first datum expresses a duration in milliseconds, followed by a volume expressed from 0 to 1 (1 being equal to 100% of volume).

```
sound addEnvelope: (VolumeEnvelope
                    points: { 0@0 . 60@1.0 . 110@0.8 . 200@1.0 . 250@0.0}
                    loopStart: 2 loopEnd: 4).
```

We note that an envelopg is characterized by a set of points and a loop. The associated method is points:loopStart:loopEnd: . A loop can either use the set of points defined or only 1.

For example by adding to an envelope with its loop indices ranging 2-4 (1 being the index value of the first table). The following sequence is performed for varying between 0.8 and 1 values. Sound volume seems to vibrate.

```
60@1.0. 110@0.8. 200@1.0
```

If we want to keep the volume at a certain level, we would write the envelope like this

```
sound addEnvelope: (VolumeEnvelope
                points: { 0@0 . 60@1.0 . 110@0.8 . 200@1.0 . 250@0.0}
                loopStart: 5 loopEnd: 5).
```

The sound ends with the value 250@0.0 and a loop now the volume to 0 throughout its duration.

Finally, the method returns the sound with the note as the first parameter (here expressed in Hz, see table), the duration second, and finally the overall volume of the note. This volume is then modulated by the envelope as defined.

```
^ sound setPitch: 440.0 duration: 1.0 loudness: 0.5
```

Note	**Octave**								
	-1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
C	16.3	32.7	65	131	262	523	1046.5	2093	4186
C#	17.3	34.6	69	139	277	554	1109	2217	4435
D	18.3	36.7	74	147	294	587	1175	2349	4698
D#	19.4	38.9	78	156	311	622	1244.5	2489	4978
E	20.5	41.2	83	165	330	659	1318.5	2637	5274
F	21.8	43.6	87	175	349	698.5	1397	2794	5588
F#	23.1	46.2	92.5	185	370	740	1480	2960	5920
G	24.5	49.0	98	196	392	784	1568	3136	6272
G#	26.0	51.9	104	208	415	831	1661	3322	6645
A	27.5	55.0	110	220	440	880	1760	3520	7040
A#	29.1	58.0	117	233	466	932	1865	3729	7458
B	30.8	62.0	123	247	494	988	1975	3951	7902

1.3 How to play your own sound?

We'll show you how we will create our own sound. Start by playing a A follows:

```
(FMSound new setPitch: 440 duration: 1.0 loudness: 1) play
```

The sound is played at a frequency of 440 Hz for 1 second with a maximum volume. The class FMSound provides generation of sine tones. It gives a fairly smooth and even sound. Figure below shows how the curve looks like.

One thing to know is that instead of defining a note by its frequency

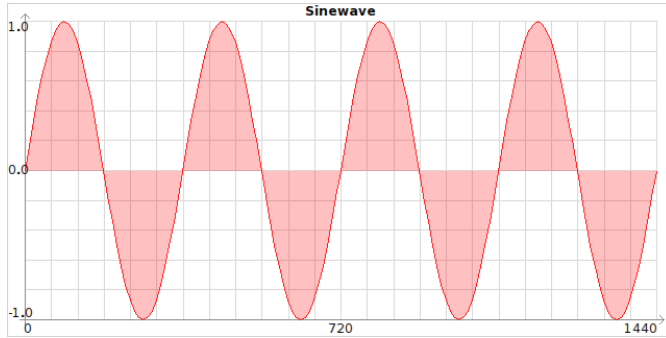

Figure 1.3: sinewave

we can write in the English notation. Table @frequence shows that an A at octave 4 has a frequence of 440Hz.

Therefore to produce an A we can now write it as the following:

```
(FMSound new setPitch: 'a4' duration: 1.0 loudness: 1) play
```

We can also generate other notes.

```
(FMSound new setPitch: 'g#4' duration: 1.0 loudness: 1) play
```

1.4 Playing several sounds in sequence

Playing a sound is good, but two is even better. For that we have at our disposal a class named SequentialSound. As the name suggests we can play sounds in sequence.

```
| seq |
seq := SequentialSound new.
seq add: (FMSound new setPitch: 'g#4' duration: 0.5 loudness: 1).
seq add: (FMSound new setPitch: 'a3' duration: 0.5 loudness: 1).
seq add: (FMSound new setPitch: 'g#4' duration:0.5 loudness: 1).
seq play.
```

If we listen to the sounds generated, we can perceive a small clap transitions between each note. To avoid this we will apply a volume envelope on each note. This is done as follows:

```
points := { 0@0 . 10@1 . 400@0.4 . 500@0}.
```

The played note is at its maximum at the beginning and then gradually returns to 0 after 500ms. The envelope is defined for a period equal to the duration of the note transition occurs naturally and sounds better to the ear.

See what gives our previous code with a volume envelope

```
| seq snd1 snd2 snd3 points |
"Sounds creation"
seq := SequentialSound new.
snd1 := FMSound new.
snd2 := FMSound new.
snd3 := FMSound new.

"Volume creation"
points := { 0@0 . 10@1 . 400@0.4 . 500@0}.

"Generation of notes"
snd1 setPitch: 'g#4' duration:0.5 loudness: 1.
snd2 setPitch: 'a3' duration:0.5 loudness: 1.
snd3 setPitch: 'g#4' duration:0.5 loudness: 1.

"Application  of envelope to sounds"
snd1 addEnvelope: (VolumeEnvelope points: points loopStart: 4  loopEnd: 4).
snd2 addEnvelope: (VolumeEnvelope points: points loopStart: 4  loopEnd: 4).
snd3 addEnvelope: (VolumeEnvelope points: points loopStart: 4  loopEnd: 4).

seq add: snd1.
seq add: snd2.
seq add: snd3.

"Playing"
seq play.
```

The sequence is now more pleasant to listen to.

1.5 Playing several sounds simultaneously

Play sounds in sequence but it is still not enough to give a soul to our production. For this we will use class MixedSound. It will literally mixing several sounds at the same time.

```
| mix snd1 snd2 points |
mix := MixedSound new.
snd1 := FMSound new.
snd2 := FMSound new.

points := { 0@0 . 10@1 . 400@0.4 . 500@0}.
```

```
snd1 setPitch: 'a4' duration:1 loudness: 1.
snd2 setPitch: 'a3' duration:1 loudness: 1.

snd1 addEnvelope: (VolumeEnvelope points: points loopStart: 4  loopEnd: 4).
snd2 addEnvelope: (VolumeEnvelope points: points loopStart: 4  loopEnd: 4).

mix add: snd1 pan: 0; add: snd2 pan: 1.

mix play.
```

Let's modify some values of the attack using the method setPitch: duration: loudness: for different notes and familiarize ourselves with the messages. The volume change of the envelope curve of the sound over time and can be used to obtain special effects.

For example if we wanted to add an echo effect to our sound should change the points of the envelope volume

```
points := { 0@1 . 100@0.6  . 200@0 . 300@0.2 . 400@0.1 . 500@0 }.
```

and envelope definitions

```
snd1 addEnvelope: (VolumeEnvelope points: points loopStart: 6  loopEnd: 6).
snd2 addEnvelope: (VolumeEnvelope points: points loopStart: 6  loopEnd: 6).
```

In our example we also use the message pan: that will position the sound in stereo. 0 for left and 1 for right. A value of 0.5 places the sound in the middle.

We will reuse the example above and add stereo to it.

```
| mix seq1 seq2 seq3 snd1 snd2 snd3 points rest rest2 |

mix := MixedSound new.
rest := RestSound new.
rest2 := RestSound new.
seq1 := SequentialSound new.
seq2 := SequentialSound new.
seq3 := SequentialSound new.
snd1 := FMSound new.
snd2 := FMSound new.
snd3 := FMSound new.

points := { 0@1 . 400@0.4 . 500@0}.

snd1 setPitch: 'g#4' duration:0.5 loudness: 1.
snd2 setPitch: 'a3' duration:0.5 loudness: 1.
snd3 setPitch: 'g#4' duration:0.5 loudness: 1.
rest duration: 0.5.
rest2 duration: 0.5.
```

```
snd1 addEnvelope: (VolumeEnvelope points: points loopStart: 3  loopEnd: 3).
snd2 addEnvelope: (VolumeEnvelope points: points loopStart: 3  loopEnd: 3).
snd3 addEnvelope: (VolumeEnvelope points: points loopStart: 3  loopEnd: 3).

seq1 add: snd1.
seq2 add: rest; add: snd2.
seq3 add: rest; add: rest2; add: snd3.

mix add: seq1 pan: 0.
mix add: seq2 pan: 0.5.
mix add: seq3 pan: 1.

mix play.
```

In this new example, we introduced a new class of its RestSound to create a silence. It has only one message duration: expressed in seconds.

The class SequentialSound has a limitation. Look at the following example to see what happens

```
| seq snd1 snd2 snd3 |
seq := SequentialSound new.
snd1 := FMSound new.
snd2 := FMSound new.
snd3 := FMSound new.

snd1 setPitch: 'g#4' duration:0.5 loudness: 1.
snd2 setPitch: 'a3' duration:0.5 loudness: 1.
snd3 setPitch: 'g#4' duration:0.5 loudness: 1.

seq add: snd1.
seq add: snd2.
seq add: snd3.
seq add: snd2.

seq play.
```

Normally we should hear 4 sounds played in sequence, or there are only 3. Obviously we can not reuse the already used. This approache is not suitable for what we want to do. For this there is another class that will meet our needs: QueueSound

Replace the line seq := SequentialSound by seq := QueueSound and listen again our example. We listen well our 4 sounds. Voila!

1.6 Envelopes

We introduced a first envelope which acts on the volume of a sound. There is another that changes the note or rather its frequency as it is played. This is called the pitch in English.

In the same way as for the volume we will define a set of points which indicates the time of change of the sound frequency. The higher the value, the higher the note will be. If a negative number is indicated, the rating drops below the starting note. We define an envelope but this time using the class PitchEnvelope.

```
| snd p |
p := { 0@0 . 100@0.1 . 200@0.2 . 300@(−0.5) . 400@0.7 . 800@0.8 . 1000@1 }.
snd := FMSound new.
snd addEnvelope: (PitchEnvelope points: p loopStart: 7  loopEnd:7).
(snd setPitch: 'a4' duration: 2 loudness: 0.5) play.
```

The transition between different frequencies is smooth and without cracks. We will add to this a volume envelope to produce a more interesting sound .

```
| snd p p2 |
p := { 0@0 . 100@0.1 . 200@0.2 . 300@(−0.5) . 400@0.7 . 800@0.8 . 1000@1 }.
p2 := { 0@0 . 100@0.1 . 200@0.2 . 300@(−0.5) . 400@0.7 . 800@0.8 . 1000@0 }.
snd := FMSound new.
snd addEnvelope: (PitchEnvelope points: p loopStart: 7  loopEnd: 7).
snd addEnvelope: (VolumeEnvelope points: p2 loopStart: 7  loopEnd: 7).
(snd setPitch: 'a4' duration: 2 loudness: 0.5) play.
```

The volume varies a lot in our example because all the points have been set manually. Now imagine that we wanted to gradually reduce to 0. We would have to define a number of points to produce a decreasing curve. It is doable, but it is tedious and the curve may not produce the desired effect. Instead of this manual approach, we will use a message called exponentialDecay: of VolumeEnvelope for the volume or class PitchEnvelope for the pitch. The curve exponentially starts with the highest volume and arrive smoothly to 0.

The definition of volume becomes this:

```
snd addEnvelope: (VolumeEnvelope exponentialDecay: 0.9).
```

The value must be strictly greater than 0 and smaller than 1. To know the points generated just select the following expression and select the print it menu item.

```
(VolumeEnvelope exponentialDecay: 0.2) points.
```

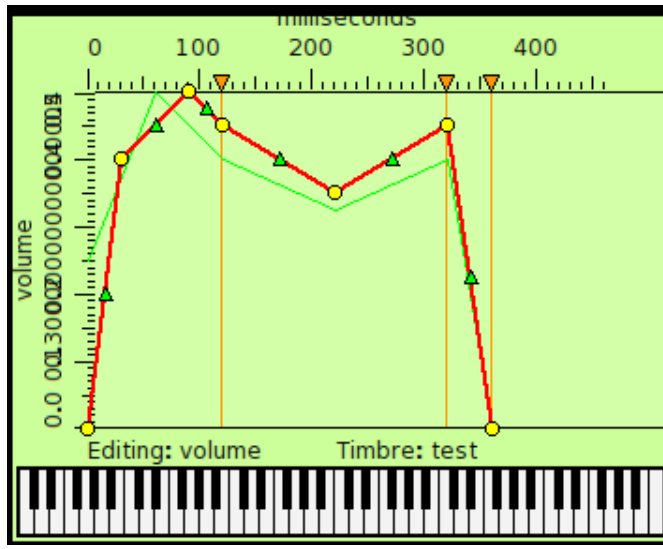

Figure 1.4: Envelope Editor

We get the following result

```
{(0@0.0). (10@1.0). (20@0.2). (30@0.04000000000000001). (50@0.0)}
```

The higher the value, the greater the number of values is. We can visualize what envelope looks like by opening the EnvelopeEditorMorph like this:

```
EnvelopeEditorMorph new openInWorld
```

Select the entire line and do **right click**/**Do it**. A window appears as shown in Figure .

You can enlarge it by selecting it with **shift and right click** then moving the anchor at the bottom right of the window as shown in Figure

The Editor exposes the curves of the instrument and allows us to change the points. You can switch from one curve to the other is by selecting directly in the window, or by selecting it from the menu editing at the bottom left of the editor.

In abscissa we have the time in milliseconds. The ordinate is modified in accordance with the selected curve. We also have 3 boundaries represented by the vertical orange axes. The two on the left are used for loop when the note is held, are the parameters loopStart:loopEnd:. We can also move according to our needs using the orange triangles above the curve abscissa. Note that it is not possible with the editor to have a loop of less than 50ms

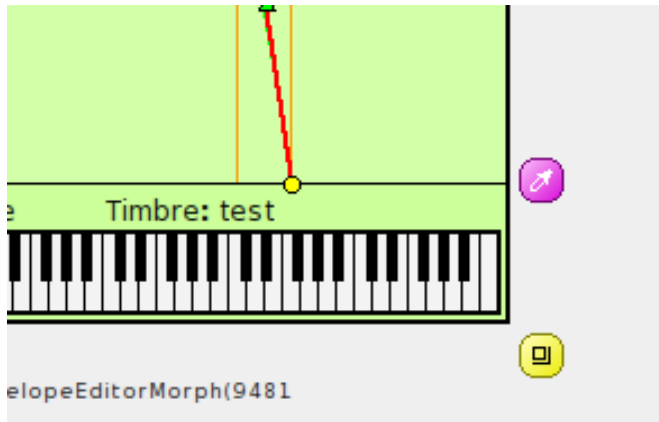

Figure 1.5: Anchor

duration, as it is not possible to have a loop start value different from the value end. This makes sense if we want to get a smooth loop.

To add a point on a portion of the curve just click on the little green triangles to display the corresponding point. This point can then be positioned at the desired value. We invite you to test all these possibilities and listen to the result obtained with the piano keyboard available. Envelopes complement our knowledge on the generation of sound. So we know play a sound, several sounds one after the other or several played simultaneously.

1.7 Playing a melody

What may look like good music generated by Pharo? We evaluate the following line of code in a workspace and listen.

```
FMSound bachFugue play
```

The music is composed of a sequence of notes. We now know how to write such a sequence but for an entire melody is not suitable. It is necessary to do otherwise. We will describe all the notes in a table. The format is (note length volume).

The note can be written in its literal form or with its frequency in hertz. The duration is expressed in milliseconds. The volume meanwhile is a value between 0 and 1000. A value higher than 1000 may saturate the sound. It can be can be useful in some cases to give an effect. Here is an example of declaring a sequence

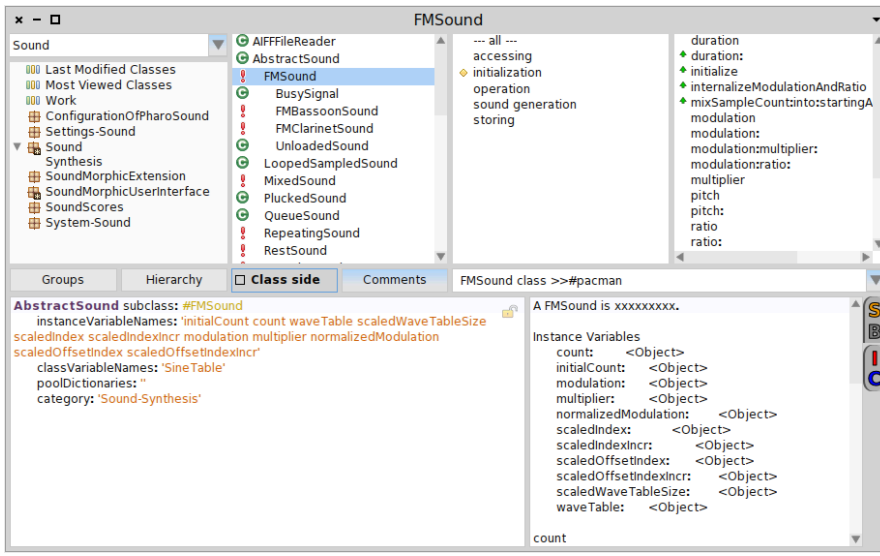

Figure 1.6: System browser: a class navigator

```
notes := #((a3 200 150) (a4 100 150) (g3 150 150))
```

We will write our first song with an air should remember good memories to all those who love the beginning of the air video games. We will define the melody introduction of pacman game.

Define a new instrument

Start by defining a new instrument. In order not to complicate our exercise, we will use the FMSound class in which we will define some new methods. Note that to be cleaner and more modular, we should define our own package and classes.

To define a method we will use a class browser: For this

- Open the Tools menu from the SystemBrowser
- In the left pane, then right-click Find Class
- Enter in the text box FMSound , then select it

You should get a browser like in figure

As methods that we define are methods that create instances of FMSound= as opposed to methods that operate on instances of the class FMSound, we define them on the class side of FMSound:

- Click on the radio button 'Class side'
- Using the menu in the third list from the left (list of protocol methods), add a protocol (Add protocol) named for example 'new instruments'.
- Select the new protocol and then add the following method fm1. To add a method, type the body of the method in the panel provided for this purpose (bottom left) and then use the entry 'Accept' menu (right click) compiles the method and then use the shortcut Alt s.

```
fm1
   | snd |
   snd := self new.
   snd addEnvelope: (VolumeEnvelope points: { 0@1 . 300@0 }
      loopStart: 2
i     loopEnd: 2).
   snd modulation: 1 ratio: 2.
   ^ snd
```

You should obtain a situation similar to the one presented in figure

Verify that your new instrument operates by executing FMSound fm1 play

We define another instrument by copying it to allow you to experiment.

```
fm2
   | snd |
   snd := self new.
   snd addEnvelope: (VolumeEnvelope points: { 0@1 . 300@0 }
      loopStart: 2
      loopEnd: 2).
   snd modulation: 1 ratio: 2.
   ^ snd
```

Now declare the same way the method pacman to mix all the tracks that will be created using methods pacmanV10: and pacmanV20:.

```
pacman
   "comment stating purpose of message"

   ^ MixedSound new
      add: (self pacmanV1On: self fm1) pan: 0.5;
      add: (self pacmanV2On: self fm2) pan: 0.5.
```

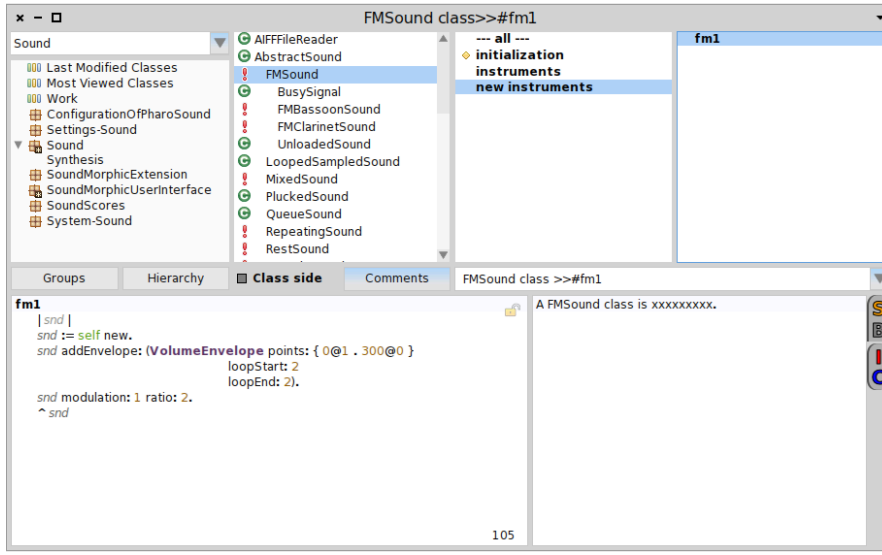

Figure 1.7: System browser sur une méthode de classe

When you compile the method, Pharo will check that all invoked methods exist. If they do not exist they will appear in red in the code. A dialog box will ask you to confirm your entry. Simply select the first choice in the list. Repeat this as many times as there are unknown methods.

The following two methods contain all the notes to be played with the triplet note, duration and volume.

```
pacmanV1On: aSound

    | notes |
    notes := #(
        (b4 0.144 150) (b5 0.144 150) ('f#5' 0.144 150) ('d#5' 0.144 150)
        (b5 0.072 150) ('f#5' 0.144 150) ('d#5' 0.288 150) (c5 0.144 150)
        (c6 0.144 150) (g5 0.144 150) (e5 0.144 150) (c6 0.072 150)
        (g5 0.216 150) (e5 0.288 150) (b4 0.144 150) (b5 0.144 150)
        ('f#5' 0.144 150) ('d#5' 0.144 150) (b5 0.072 150) ('f#5' 0.216 150)
        ('d#5' 0.288 150) ('d#5' 0.072 150) (e5 0.072 150) (f5 0.144 150)
        (f5 0.072 150) ('f#5' 0.072 150) (g5 0.144 150) (g5 0.072 150)
        ('g#5' 0.072 150) (a5 0.144 150) (b5 0.216 150)).

        ^self noteSequenceOn: aSound from: notes.
```

The method noteSequenceOn:from: generates a sequence of notes (e.g. SequentialSound) from a table of notes for a given instrument. It will use as

seen above the triplet notes, duration, volume.

```
pacmanV2On: aSound
   | notes |
   notes := #(
      (b1 0.432 500) (b2 0.144 500) (b1 0.432 500) (b2 0.144 500) (c2 0.432 500)
      (c3 0.144 500) (c2 0.432 500) (c3 0.144 500) (b1 0.432 500) (b2 0.144 500)
      (b1 0.432 500) (b2 0.144 500) ('f#2' 0.288 500) ('g#2' 0.288 500)
      ('a#2' 0.288 500) (b2 0.288 500)).

      ^self noteSequenceOn: aSound from: notes.
```

If your melody notes use low frequency volume it will be difficult to hear. To remedy to this, increase the volume of notes. Here we increase the volume to 500, unlike the previous melody where we stayed at 150. Now we just have to play the melody: FMSound play pacman.

1.8 Playing a sample

The sound of an application does not have to be done through sound synthesis. Sometimes it is necessary to make use of digitized sound reproducing sounds, voices or entire music. We'll show you how to play a sample from a file or directly from memory.

First we should download the sounds for our example. To do this run the following in a console or command then download the archive directly on your computer.

```
wget https://github.com/xmessner/PharoSoundTutorial/archive/master.zip
```

Unzip it in the directory of your choice. The folder sounds contains all digitized sounds we need. You have to adjust the repository directory based on your environment.

Let us try at first to play a sound from a file in wav format.

```
(SampledSound fromWaveFileNamed: '/home/messner/PharoSound/sounds/
     mouette.wav") play
```

The sound is played in background without blocking the rest of our environment. For example, we can use this method to sound to a specific action or alert.

If our goal was to design a game, loading the sound directly from the disk at key times could slow animations, making it look unplayable. To avoid these problems sounds can be loaded in memory and played when it will be necessary to gain response time.

Pharo provides us with a sound bank that we can supply according to our needs. To add a sound to this library we use the message addLibrarySoundNamed:samples:samplingRate:.

Copy the following code in a workspace and evaluate it to add the sample in the library

```
| spl spl2 |
spl := SampledSound fromWaveFileNamed: '/home/messner/sounds/guitar.wav'.

"Added sound in the library for play later"
SampledSound addLibrarySoundNamed: 'guitar'
        samples: spl samples
        samplingRate: spl originalSamplingRate.
```

Now we can play the sample frequency and the desired time.

```
| spl spl2 |
spl := SampledSound soundNamed: 'guitar'.
spl2 := (SampledSound samples: spl samples samplingRate: 2000).
spl2 duration: 0.1.
spl2 play.
```

We can control the duration and sampling rate. The example above just play the first milliseconds of the sample at a very low frequency. If we increase the samplingRate and duration we can get the complete sample played.

1.9 A last little effort

We will close this chapter with one last method to play several digitized sounds in sequence. First we will load into memory the different samples are used.

```
| spl spl1 spl2 |
"Loading sounds"
spl := SampledSound fromWaveFileNamed: '/home/messner/sounds/ocean.wav'.
spl1 := SampledSound fromWaveFileNamed: '/home/messner/sounds/mouette.wav'.

spl2 := SampledSound fromWaveFileNamed: '/home/messner/sounds/bateau.wav'.


SampledSound addLibrarySoundNamed: 'ocean'
        samples: spl samples
        samplingRate: spl originalSamplingRate.
SampledSound addLibrarySoundNamed: 'mouette'
        samples: spl1 samples
```

```
        samplingRate: spl1 originalSamplingRate.
SampledSound addLibrarySoundNamed: 'bateau'
        samples: spl2 samples
        samplingRate: spl2 originalSamplingRate.
```

Now that all sounds are loaded we will play in sequence with the class QueueSound seen previously. They are played one after the other at the desired frequency.

```
| mix q1 q2 spl1 spl2 spl3 |

q1 := QueueSound new.
q2 := QueueSound new.
mix := MixedSound new.
spl1 := (SampledSound samples: (SampledSound soundNamed: 'ocean')
                samples samplingRate: 22000).
spl1 duration: 20.
spl2 := (SampledSound samples: (SampledSound soundNamed: 'mouette')
                samples samplingRate: 22000).
spl2 duration: 10.
spl3 := (SampledSound samples: (SampledSound soundNamed: 'bateau')
                samples samplingRate: 22000).
spl3 duration: 5.

q1 add:spl2;add:spl2.
q2 add:spl3;add:spl3.
mix add: q1 pan: 0 volume: 0.100;
   add: q2 pan: 1 volume: 0.5;
   add: spl1 pan: 0.5 volume: 1.
mix play.
```

1.10 Conclusion

We showed you a small part of Pharo opportunities for the creation and manipulation of sounds. We hope that you will experiment and have fun with Pharo.