GUITARDSP Project v1.0
Report

Guitar Effects Processor in Java

Final project
of the course of

Object-Oriented Programming

Submitted by
657291 - Francesco Del Duchetto
francesc.delduchetto@studio.unibo.it

Date:
April 2, 2014

School of Science
University of Bologna
Cesena, FC, 47521

Abstract

GuitarDSP is an application written in Java that allows to modify a sound file in real time, applying on it some effects.
The ultimate purpose is to simulate a multi-effect pedal for guitar but due for time this version of the project allows user to modify an audio source taken by a .wav file stored on the pc.

Contents

1 Introduction

With the advent of the digital world the way in which the sound is managed has been revolutioned increasing the versatility and the capability of the audio effects. Now a guitarist doesn't have to spend a lot of money, buying pedals, to use the effects and find its own sound: there are on the market a lot of pedals that are able to emulate (pretty well) the most famous analog effects of the major manufacturers and that are more powerful than their corresponding analog.

Digital effects allow to experiment new processing techniques creating new sounds that in the analog domain are impossible to obtain; also they guarantees more safety because the musician must not have to deal with cables and possible damages of the circuits.

With this project I try to create an application that is able to process an audio stream with some DSP algorithm. It use the Java Sound API[1], particularly the package javax.sound.sampled to obtain the input and output lines from the system.
It is equipped with a graphical interface developed using the Swing library.

The ultimate aim of the project is to allow user to plug its guitar jack in the mic input of the pc and to start playing modifying the sound in real-time with the effects provided by the application. Now the application can process an input stream given from an audio file stored on the PC because the real-time functionality involve in many difficulties, such as noise and latency, that due for time I have not yet been able to overcome.

2 Work Done

2.1 Application design

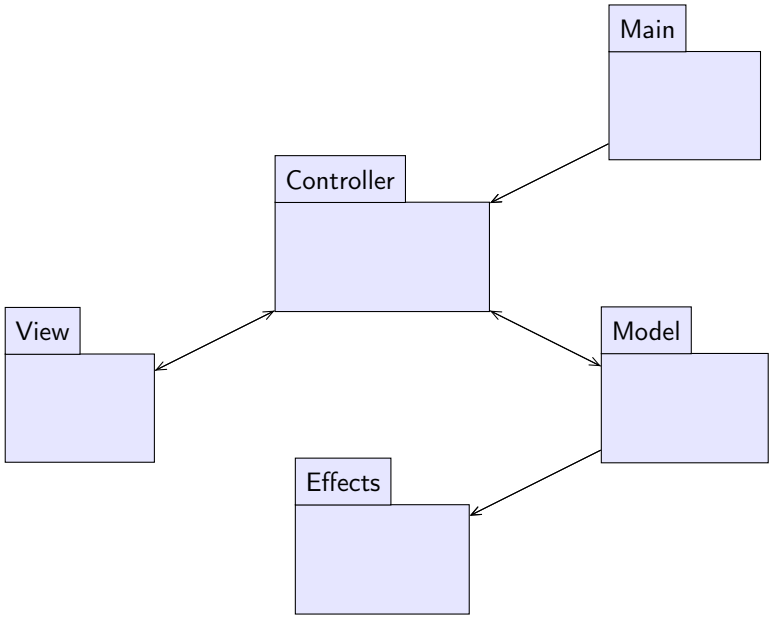
In this stage of the work was designed the structure of the application, studying the best way to organize the entities and the operations between them.

The global operation is based on the Model-View-Controller pattern that indicates the guidelines to write a good GUI-based application. This pattern is very useful because it helps the programmer to write highly reusable and extensible code when he has to work with a user interface. Its philosophy is to split the general operation into three parts: the management of the graphical interface (View), the management of data and functionalities (Model) and the communications between the View and the Model (Controller).

The application is composed by the following packages:

Main: contains only the main class that launches the application.
View: contains the classes that compose the user interface.
Model: contains the classes that manage the internal operation, namely the streaming and processing of the sound.
Controller: contains a class that translates user interactions with the View to commands for the Model, and also it modifies the user interface according to signals from the Model.
Effects: contains the effects that implement the DSP algorithms to modify the input signal.

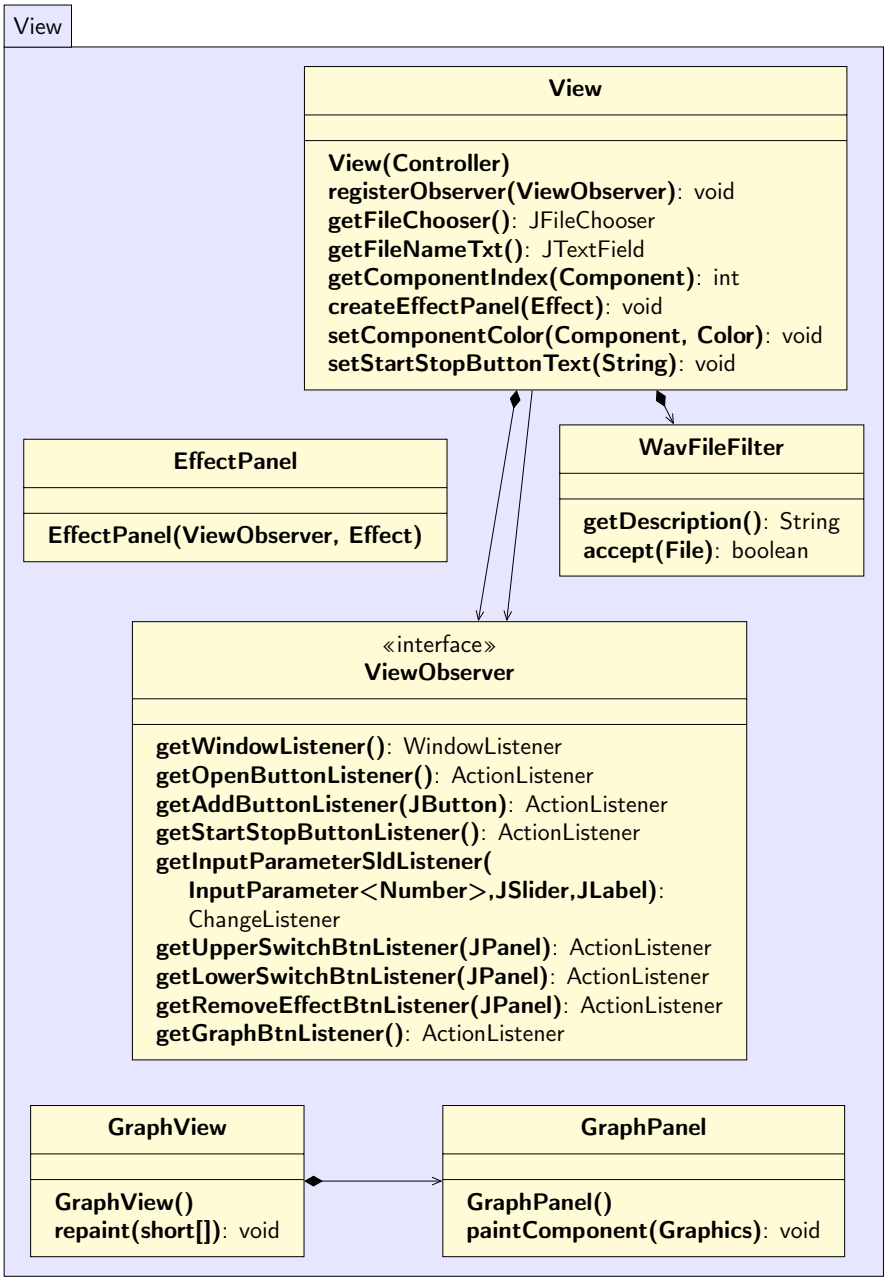
Below there's a simple diagram that broadly shows how conceptually the application works with the packages:



Main starts Controller which shows the View and initializes the Model. User interactions are delivered from View to Model and vice versa by Controller. The Model uses Effects to modify the input.

2.1.1 MVC pattern

As we said previously most of the entities of the application are grouped into the three packages that make up the MVC pattern.
Below are shown these packages with the classes composing them.



View: extends JFrame. It shows the graphical interface on which the user can interact with the application.
The Controller, to receive the user actions, must implement the ViewObserver interface and must register itself as an Observer of this class.

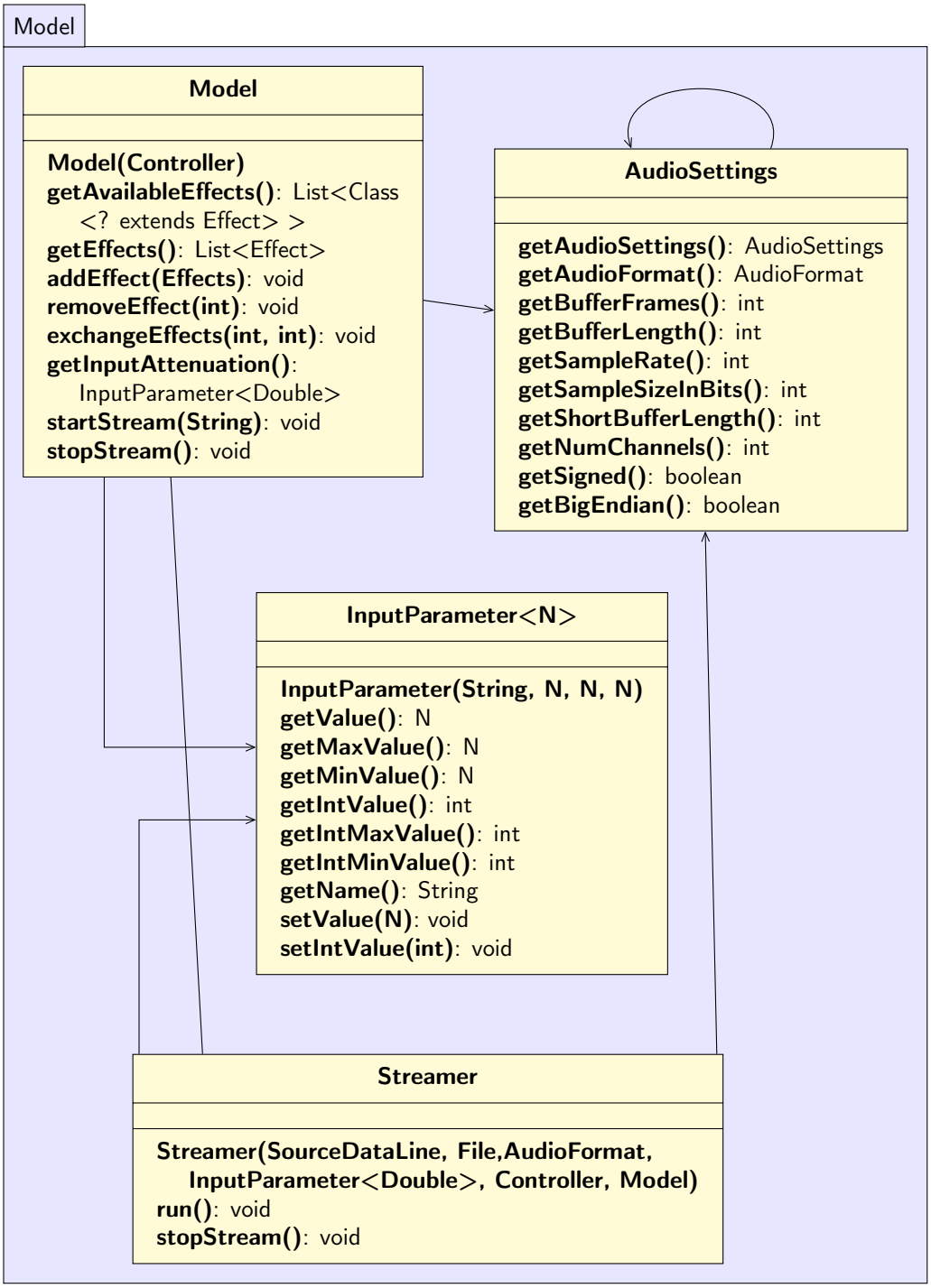
EffectPanel: extends JPanel. It is the panel that is added when you click the button to add an effect. It allows the user to vary the parameters of the Effect thus customizing the output sound.

WavFileFilter: extends FileFilter. It is used by the JFileChooser to filter out, in the window that allows user to select a file, the files that doesn't have the .wav extension.

ViewObserver: it is an interface that must be implemented by the Controller. Therefore the Controller must provide the Listeners of the View's components.

GraphView: it shows the graph of the audio signal processed with the Effects. It contains only a GraphPanel that shows the signal. It refreshes the graph every time that the repaint(short[]) method is called, taking the new buffer that must be shown.

GraphPanel: it's the JPanel that contains the graph of the signal. It draws the output signal according to the size of the JFrame which can be resized by the user. This function is made by overriding the paintComponent(Graphics) method.

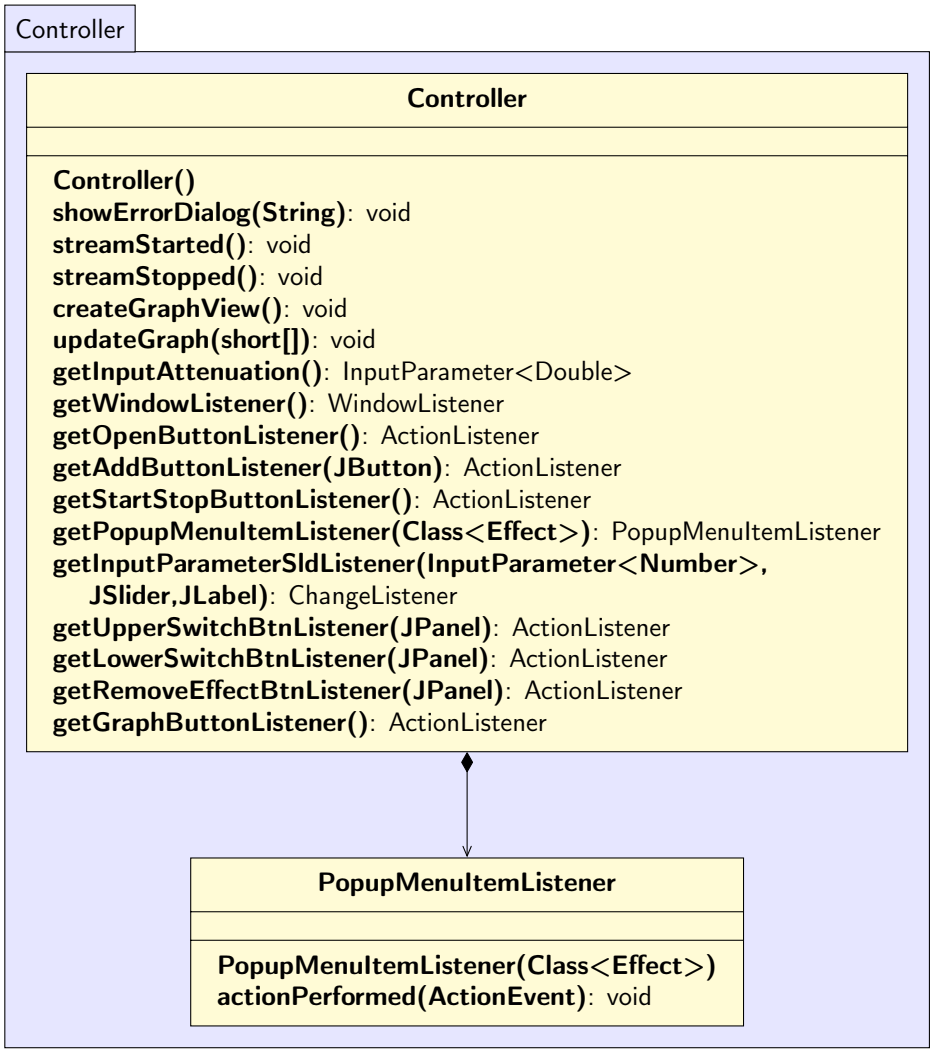

Model
Model
Model(Controller)
getAvailableEffects(): List<Class <? extends Effect> >
getEffects(): List<Effect>
addEffect(Effects): void
removeEffect(int): void
exchangeEffects(int, int): void
getInputAttenuation(): InputParameter<Double>
startStream(String): void
stopStream(): void
AudioSettings
getAudioSettings(): AudioSettings
getAudioFormat(): AudioFormat
getBufferFrames(): int
getBufferLength(): int
getSampleRate(): int
getSampleSizeInBits(): int
getShortBufferLength(): int
getNumChannels(): int
getSigned(): boolean
getBigEndian(): boolean
InputParameter<N>
InputParameter(String, N, N, N)
getValue(): N
getMaxValue(): N
getMinValue(): N
getIntValue(): int
getIntMaxValue(): int
getIntMinValue(): int
getName(): String
setValue(N): void
setIntValue(int): void
Streamer
Streamer(SourceDataLine, File,AudioFormat, InputParameter<Double>, Controller, Model)
run(): void
stopStream(): void

Model: it manages the whole operation of the application. It is able to start a new Streamer and to stop it, also it manages the Effects allowing the user to add, remove and exchange them.

Streamer: extends Thread. When started it creates a new thread which flows the audio from the input stream to the output line applying the effects added by the user. Every time it reads a new buffer from the input line it asks to the Model the list of Effects that it must apply, allowing the user to add, remove and exchange effects during the streaming of the file. The Streamer also call the updateGraph(short[]) method of Controller every time that a new buffer is processed, refreshing the graph.

InputParameter<N>: extends Observable. It encapsulates the idea of parameter that can be varied at run-time modifying the state of an Effect. For example it is used by the OverdriveEffect to allow the user to modify the power of distortion.
It is a generic class where N must extends Number. It provides however methods to access the value as int number because the JSlider, which is used to vary its value, works with int numbers.
It also notifies its Observers when its value is changed. An Effect can register itself as Observer of their InputParameters.

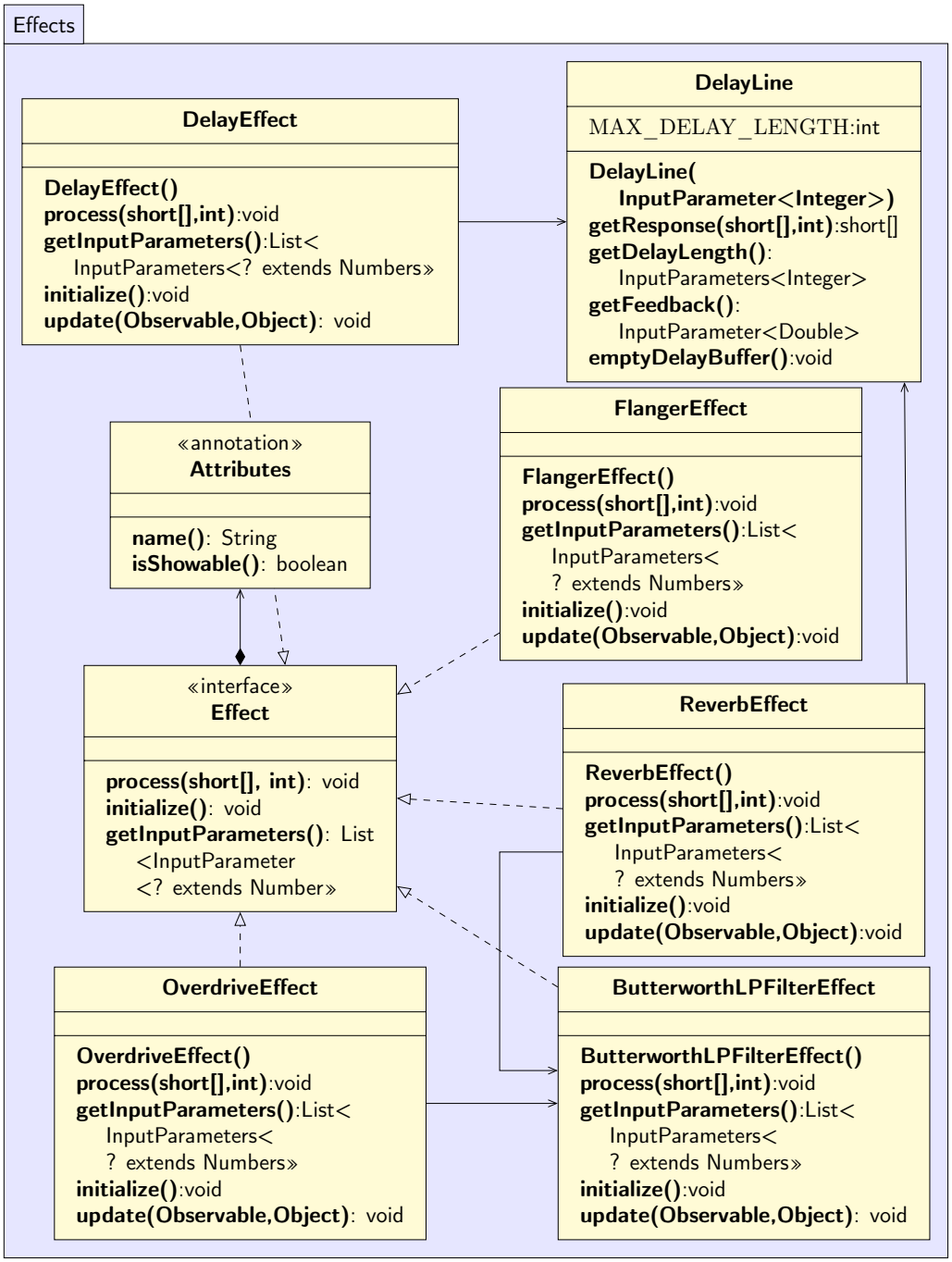
AudioSettings: it follows the agreement of the Singleton pattern, in fact its constructor is private and the only existing instance can be taken only by calling the AudioSettings.getAudioSettings method.
I chose to make it a Singleton pattern because there are no reasons to have more than one instance of this class at the same time and also to avoid having to pass its reference to every object that needs it.
It provides methods to retrieve the audio technical information, such as the frame size or the buffer length used.



Controller: its constructor creates a new View and a new Model acting as a link between them. It implements ViewObserver and therefore provides the Listeners for the View. It also supplies methods to allow the Model to report when a stream is started or stopped and methods to create and to update the GraphView.

PopupMenuItemListener: it handles the operations to do when an item, contained in the pop-up that allows the user to select an effect, is pressed. It says the Model to create a new Effect and it says the View to add a new EffectPanel for the Effect created.

2.1.2 Effects


Effects
DelayEffect
DelayEffect()
process(short[],int):void
getInputParameters():List<
InputParameters<? extends Numbers»
initialize():void
update(Observable,Object): void
DelayLine
MAX_DELAY_LENGTH:int
DelayLine(
InputParameter<Integer>)
getResponse(short[],int):short[]
getDelayLength():
InputParameters<Integer>
getFeedback():
InputParameter<Double>
emptyDelayBuffer():void
«annotation»
Attributes
name(): String
isShowable(): boolean
FlangerEffect
FlangerEffect()
process(short[],int):void
getInputParameters():List<
InputParameters<
? extends Numbers»
initialize():void
update(Observable,Object):void
«interface»
Effect
process(short[], int): void
initialize(): void
getInputParameters(): List
<InputParameter
<? extends Number»
ReverbEffect
ReverbEffect()
process(short[],int):void
getInputParameters():List<
InputParameters<
? extends Numbers»
initialize():void
update(Observable,Object):void
OverdriveEffect
OverdriveEffect()
process(short[],int):void
getInputParameters():List<
InputParameters<
? extends Numbers»
initialize():void
update(Observable,Object): void
ButterworthLPFilterEffect
ButterworthLPFilterEffect()
process(short[],int):void
getInputParameters():List<
InputParameters<
? extends Numbers»
initialize():void
update(Observable,Object): void

Effect: it is an Interface that extends Observer because it can register itself to receive a notification by some of its InputParameter.
Each class that implement this interface must be able to:

- process a given buffer of shorts;
- initialize itself resetting its environment;
- return the InputParameters that the user can modify.

Attributes: it is an Annotation that stores the name of the Effect and also it allows to know if it's show-able to the user since some effects are for inner use and the user can't use it directly. This annotations is necessary because allows, through reflection, to know which effects the user can select before they are instantiated.
This Annotation allow, in the future, to add new Effects without modifying the View.

DelayLine: it isn't an effect because it doesn't modifies the original sound but it simply stores, in a circular buffer, an infinite series of the given sounds according to these parameters:

Delay length: it defines the length of the circular buffer, changing the perception of the sound reflection's distance.

Feedback: it defines the amount of the original sound that must be stored. Its value can vary in the range $[0, 1]$, if the value is 0 the effect doesn't store any delay, if the value is 1 it stores the sound with the same volume as the original.

To get the dealy response associated to the original buffer you must call getResponse(originalBuffer, bufferLength).

DelayEffect: it is a simple effect that use a DelayLine to perform an echo effect on the original sound. It allows the user to modify the length of the delay and the gain value of the echoes. [2].

FlangerEffect: is an Effect that mixes the original sound with a delayed copy of the original signal. The delay time is continuously varied by a low frequency.
This effect doesn't use any delay lines because the DelayLine keeps an infinite series of previously echoes while it needs only the previous delayed sound.[3] The parameters that the user can modify are:

Frequency: the frequency of the sin function that continuously modify the delay length.

Excursion: the range in which the delay can vary.

Depth: modifies the percentage of delayed sound over the original sound.

ButterworthLPFilterEffect: it's a low pass filter that filters the frequencies above a given bound which can be varied by the user[4]. It isn't showable to the user, it's only used by the $\mathrm{OverdriveEffect}$ and by the $\mathrm{ReverbEffect}$.

OverdriveEffect: it applies a non-linear distortion function to the original sound[5]. The "sound" of the distortion is varied through two parameters:

Drive: modifies the power of the distortion, if its value is equal to 0 the output sound is the same as the original signal and higher is the value more the sound is amplified and like a square wave.

Cutoff: manages the $\mathrm{ButterworthLPFilterEffect}$ that cut the higher frequencies that in a real situation doesn't occur and allows user to change the "color" of the distortion.

Using the $\mathrm{GraphView}$ we can clearly see the effect of these two parameters on the original signal.

ReverbEffect: it simulates the echoes that occurs for example in a church or in a concert hall. Often this effect is added to a registration to make the sound less dry and more pleasant.
The algorithm implemented is based on the *Schroeder Reverberator*[6] that suggest to apply two stages of delay to the incoming sound:

Early reflections: is implemented using some delay lines chained in series. The delay lengths are very small to increase the density of the audio impulses and different from each other to make a good spatialization of the echoes.

Late reflections: is implemented using some delay lines chained in parallel. Their delay lengths are longer than the previous and varying them we can simulate the size of the rooms in which the reverb occurs.
The late reflections are filtered with a $\mathrm{ButterworthLPFilterEffect}$ to simulate the air absorption of the high frequencies.

3 Considerations

3.1 Workflow

1. The design of the project started with a study of the Java Sound API to understand how it works and how its tools could meet my needs.
2. Then I designed an initial architecture of the entities starting to write some code.
3. At this point I had to search some resources on the web that helped me to understand how I actually had to implement the effects.

3.1.1 In-development changes

During the development of the project there was few changes from the initial intentions:

1. I implemented the effects extending the abstract class `Control` provided by the `javax.sound.sampled` package which represents a control that the line requested can have but then I choose to refactor the effects under the new interface `Effect` to have more movement's freedom and because conceptually, in my opinion, it is better a choice.
2. I had decided to make the `View` and the `Model` unique using the `Singleton` pattern but then I changed this setting to allow in the future to add some of these if necessary.
3. I decided to remove the audio settings in the `Model` refactoring them in the entity `AudioSettings` in which I will be able to add, in the future, the methods to modify these settings according to user's needs.
4. Lately I added the functionality that allows the user to see the output audio signal in a graph.

3.2 Possible troubles

It is recommended not to use an overdrive effect before a delay or a reverb effect because the overdrive saturation, that is replicated many time by the delay, will overflow the buffers producing only noise.

4 Future Work

As already said my goal is to allow real-time processing of the sound coming from the mic line of the PC.
These are the next features needed:

- Make the application able to take the input stream from the mic line and to allow the user to change the audio settings adjusting them according to its needs and to the machine capabilities.
- Implement a noise reduction algorithm due to interferences from the PC.
- Add the ability to save a specific configuration of the effects allowing the user to resume it when he wants.
- Create a kind of foot-switcher that handle the commands of the application, maybe using Arduino.

5 Conclusion

This project gave me the possibility to enhance my knowledge of the Object Oriented Programming paradigm by dealing with the development of an exciting application.

The development of the project took me quite some time that maybe slightly exceeded the limit of one hundred hours considering the time to understand the Java Sound API, to study the DSP algorithms and to make this report.

References

[1] Java Sound API
http://docs.oracle.com/javase/tutorial/sound/index.html

[2] Smith, J.O. "Feedback comb filters", in Physical Audio Signal Processing,
https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html

[3] Smith, J.O. "Flanging", in Physical Audio Signal Processing,
https://ccrma.stanford.edu/~jos/pasp/Flanging.html

[4] Baum Dev Blog "Butterworth low pass filter",
http://baumdevblog.blogspot.it/2010/11/butterworth-lowpass-filter-coefficients.html

[5] Cheng-Hao Chang "A Guitar Overdrive/Distortion Effect of Digital Signal Processing ",
http://ses.library.usyd.edu.au/bitstream/2123/7624/2/DESC9115_DAS_Assign02_310106370.pdf

[6] Smith, J.O. "Schroeder Reverberators", in Physical Audio Signal Processing,
https://ccrma.stanford.edu/~jos/pasp/Schroeder_Reverberators.html