Development of a Sound Plugin for SetlX

and Implementation of a Composition Algorithm

Student Research Project

Course of Studies *Applied Computer Science*

Cooperative State University Mannheim

by

Lukas Retschmeier

Hand over:	June 4, 2018
Time of Editing:	12 Wochen
Matriculation Number, Course:	1339518, TINF15/AI-BI
Company:	Atos It Solutions and Services GmbH
Scientific tutor:	Prof. Dr. Karl Stroetmann

Abstract

This paper documents the work of *Lukas Retschmeier* during his student research project. The document shows how a sound plugin for the high-level programming language SetlX was developed. Therefore an external music framework was tethered to SetlX and can now be accessed with new user-defined functions from the interpreter. In order to demonstrate the capabilities of these new functionalities the *Mozart Dicing Game* was implemented in native SetlX . This document also provides detailed documentation about how this plugin can be used.

Contents

List of Figures

1 Introduction

The idea of using algorithms to compose and build music started long before it got more interesting by our modern high performance computers. We will list some milestones of this field of research:

The **greek** philosophers (Pythagoras, Ptolemy) around 100 AD believed, that there is an absolute relation between single tones. According to them, the harmony of music can be explained with pure mathematics. (Which was right!) So, they started to derive formulas, that were the foundation of our understanding of music with modes, scales and intervals.

Soon after **Mozart**s' early death, "Ein Musikalisches Würfelspiel" was published posthumously. It contains a list of fragments for a twelve-measure waltz, which can be arranged using a dice. Such games were pretty popular during that time and we will implement his algorithm at the end of this work.

John Cage was best known for his experiments with randomness in music. For example, his *Atlas Ecliptica* (1961) was composed by laying a score paper on top of astronomical charts and simply placing notes where the stars occured. This is called a **stochastic** approach. (see [5])

On the other side, there are **rule-based** schemes, that try to extract formulas out of existing music sheet. Based on these sources, it is possible to create new music. For example *Kemal Ebcioglu* wrote a system called `CHORAL`, which generates four-part chorales[1] in the style of J. S. Bachs' music according to 250 first-order predicates. (See [3])

The newest development is the usage of **Artificial Intelligence (AI)** for composing music. For example the startup *AIVA Technolgies* concentrates on generating soundtracks for movies and video games using AI. In 2017 they supplied the whole soundtrack for the game *Battle Royal* by *Pixelfield*. (see. [7])

[1]a choral

1.1 Scope of the Document

The scope of this document is to illustrate the work of the student *Lukas Retschmeier* during his student research project at the *Cooperative State University Baden Wuerttemberg* in Mannheim. It shows the development process of a soundplugin for the SETLX language and describes how it can be used. Furthermore, this document shows how the *Mozart Dicing Game* algorithm was implemented in SETLX using this new plugin.

1.2 Purpose of the Document

The purpose of this document is to explain the soundplugin for SETLX and support the development of further add ons to the SETLX Interpreter. Furthermore, it serves as a documentation for the new features and functions that were added to the programming language and how they can be used.

1.3 Prerequisites

In order to understand the musical background, it can help to have at least a little knowledge about music theory. If you want to refresh your skills in reading music sheet, you can find a summary of the most important things here.

Furthermore, for a deeper dive-in into the advanced topic of harmonies I can recommend the German book ”Harmonielehre im Selbststudium” (Harmonies in Selfstudy) by *Thomas Krämer* to anyone.

2 Theory

In this chapter, we will give more information about two theoretical topics:

1. The free **JFugue Java Framework**
2. The usage of a **MusicString** that can be parsed by JFugue and is used to write music

2.1 What is SetlX ?

According to the official webpage ”setlx is an interpreter for the high level programming language SETLX (set language eXtendend)”. [6] As you might already guess from this name, the language focuses on a clever and sophisticated support for lists and sets. SETLX is an extension for the programming language `setl` by Jack Schwartz designed by *Hermann Tom*.

Prof. Dr. Karl Stroetmann, a professor at the *DHBW Mannheim* and advisor for this research project also uses it in his lectures about mathematics and logics, because it is a good and easy way to demonstrate the concepts of discrete mathematics.

The latest SETLX version is *2.7.0*, which was released in October 2017, can be downloaded here.[1] It depends on the Java Runtime Environment (JRE) 1.7.

2.2 MIDI File Format

”MIDI (Musical Instrument Digital Interface) is a protocol developed in the 1980’ which allows electronic instruments and other digital musical tools to communicate with each other.”[1]

[1] `https://randoom.org/Software/SetlX`

A MIDI file itself does not produce any sound. It just gives instructions to an interpreter on how to produce tones with a series of messages. Consequently, the real sounding can vary from device to device. You can find a full specification of the MIDI file format here.

2.3 The JFugue 5.0 Java Framework

JFugue is an open-source Java framework, that allows an easy way to program and play music in Java. It was first released in 2002 by David Koelle.(see. [4])

It extends the basic java midi classes *Java.sound.midi* and provides an easy-to-use interface for a high-level access, that brings "music programming to the masses".[4, p. 13] So, you do not have to care about the bits and bytes in the background, but can concentrate on the thing, that matters most: the music itself.

In 2015 he published a completely rewritten version simply called *JFugue 5.0*, which is a from-scratch revision of the whole code base in order to get a cleaner software architecture of the framework.

2.3.1 Functionalities

JFugue provides a ton of functionalities for interacting with music in *Java*. These include:

1. Writing and playing music using special JFugue *MusicStrings*
2. Usage of functions based on music theory
3. Real-Time processing of music
4. Loading and saving of MIDI-files

2.3.2 Advantages over JMusic

There is another famous Java framework for music called JMusic. *JMusic* provides similar to *JFugue* methods for writing and playing music.

In an early prototype, many parts of this work had been first implemented in *JMusic*. But there were some problems, that lead the author switching to *JFugue*.

JMusic stopped maintanance and further development The last (major) release of JMusic was in 2009.(see. [2]). This was 10 years ago and since then no bugs have been fixed any more. Officially just Java versions up to 1.6 are supported and we can not hope for an optimization for newer runtime environments.

Simplicity of JFugue MusicString A *JFugue MusicString* is a simple character sequence, that can be parsed by a *JFugue Staccato Parser*. It provides a natural way to prompt music information. This can contain voices, tones as well as velocities and volumes. In other words, it contains everything you can find in a typical music sheet or midi file. It can be compared to well-known markup languages like *Markdown* or *Lilypond*[2]. We do not have the ability to write music in that natural and simple way as *JFugue* provides us in *JMusic*. In section 2.4 we will dive deeper into the possibilities of a *MusicString*.

These two points were the reason, why the decision was finally made in favor of *JFugue*.

2.3.3 A Simple JFugue Example

```java
import org.jfugue.player.Player;

public class HelloWorld2 {
        public static void main(String[] args) {
                Player player = new Player();
                player.play("V0 I[Piano] Eq Ch. | Eq Ch. | Dq Eq Dq Cq V1 I[Flute]
                     Rw | Rw | GmajQQQ CmajQ");
        }
}
```

Code 2.1: A Simple JFugue Example

Figure 2.1 shows an example for the *JFugue* Framework in Java playing a simple sequence of tones. First, a `player` object is created, that can play a *MusicString* and

[2]A special MD language for creation of sheet of music, free project

then we simply call the *play()*-method with some notes to play.

2.4 Using JFugue MusicString

This section will explain all you need to know about creating music in SETLX . Because the SETLX -soundplugin will directly implement JFugue, it is important to know, how you can write down music in a single string. This section will just cover the basics on what can be done with it. For more information please have a look into the official manual[3].

The following shows how a note is specified in *JFugue*. If you want to play a rest, you can use the rest character 'R'.

```
regex := ['C','D','E','F','G','A','B','R']['#','b']?
       [octave]?[duration]*[dotted]?[chord]?
octave := {0,1,..,10}
duration := {'w','h','q','i','s','t','x','o'}
dotted := {'.'}
chord := {maj, min, aug, dim, ...}
       // For a complete list see Chords section
```

Examples For example "C","Eb4q" or "Ab6w.maj" are valid Notes/chords.

2.4.1 Octaves

You may optionally specify an octave for the note represented by a number ranging from 0 to 10. Figure 2.1 shows the main octaves.

2.4.2 Duration

The duration indicates how long a note should be played. Duration is indicated by one of the letters in the table below. If duration is not specified, a quarter serves as default value.

[3]Chapter Two

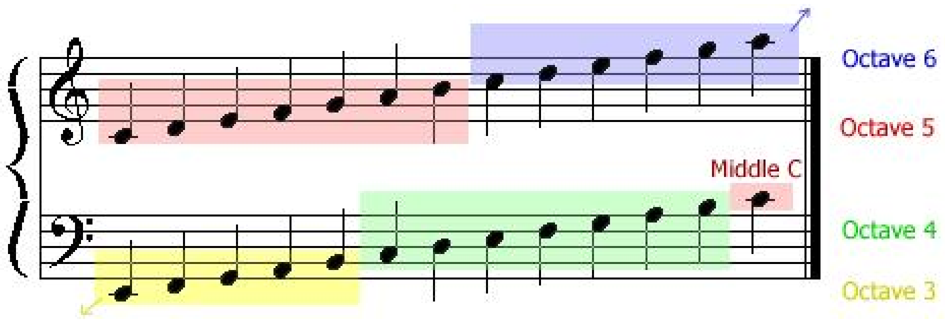

Figure 2.1: JFugue Octave Values

Duration	Character
whole	w
half	h
quarter	q
e**i**ghth	i
sixteenth	s
thirty-second	t
si**x**ty-fourth	x
one-twenty-eighth	o

A dot increases the duration of a note by half of its value. So, for example a dotted half note (Rh.) is equal to the duration of a half note plus a quarter note.

Examples Here are some examples of durations:

```
"Aw" // A5 whole note
"Aw." //A5 whole+half note
"E7h" //E7 half note
"[60]wq // C (Numerical 60 whole+quarter)
"G8i" // G8 dotted-eighth note
```

2.4.3 Chords

JFugue supports a variety of chords, each of which is described in the table 2.1 on page 10. If you add this chord postfix to a note, *JFugue* will generate the chord based on the note as root.

2.4.4 MusicString

If you want to create music and not just single notes, you have the know about *MusicStrings*. Simply, a *MusicString* is a pattern in form of a string that contains a sequence of notes/rests and additional behaviors.

They can also be combined which is useful, when you want to reuse and structure parts of your music - for example splitting up the music into *Intro*, *Verse* and *Bridges* and repeat these parts.

In a *MusicString* you just write the notes one after another. This can look like one of the following examples, which are all valid *MusicStrings*:

```
"C D E F G A B C6" // 'B' == German 'H'
"Cmaj Fmaj Gmaj Cmaj"
"Cq Dw Ew. Gh."
```

2.4.4.1 Additional Information

You can add additional information to the *MusicString*. Therefore, the letters `T`, `I` and `V` were added in order to parse the *Tempo*, *Instrument* and *Voice*. These settings are valid until they are explicit changed again.

Tempo The tempo in BPM (Beats per Minute) or tempo markings (Largo, Andante, Allegro, Presto, ...)

```
"T[Andante] Cq Eq T[120] Cq Eq T[Presto] Cq Eq"
       // Increasing tempo using three different tempi
```

Instrument The instrument that is used by the MIDI player. You can find a list of all supported instruments in the manual (see Figure 2.13).

```
"I[Piano] Cq Gq I[Violin] Cq Gq"
       // plays a quint and changes instrument from piano to violin
```

Voice The line in which the notes are played. If you want to play more than one note at the same time (polyphony), you have to use voices.

```
"V1 C6h V2 E6q G6q"
       // plays E and G at the same time as C
```

Of course, you can use these information together! Therefore, an instrument subordinates to the voice, which subordinates to the tempo. This means that for example an instrument information always refers to the last voice indication:

`Tempo >> Voice >> Instrument`

```
"T120 V1 I[Piano] C6h V2 I[Violin] E6q G6q"
// plays E and G (Violin) at the same time as C (Piano)
```

2.4.4.2 Examples For MusicString

Here are some examples of complete MusicStrings:

```
"C5q C5q G5q G5q A5q A5q Gh"
       // Twinkle, twinkle, little star
"[V1] D6qmaj7 G6qmaj7 C6hmaj7"
       //Simple Jazz II-V-I progression on Voice 1
```

Inversions You can change the root note of a chord by shifting up by one octave. Then the second note will become the bass note. This process is called *inverting*. Especially in Jazz we also speak from the *voicing* of the chord.

You can invert a chord by adding a '^' to the end of the chord string. ('^^' for second inversion). For example C6maj^ will be equivalent to E6+G6+C7 (instead of C6+E6+G6)

2.4.5 Progressions

A chord progression is a succession of musical chords. The complexity of a chord progression varies from genre to genre and over different historical periods. Pop and Rock songs have in general easier progressions than Jazz, Blues or Funk music.

Common Name	**JFugue Name**	**Intervals (0 = root)**
major	maj	0, 4, 7
minor	min	0, 3, 7
augmented	aug	0, 4, 8
diminished	dim	0, 3, 6
7th (dominant)	dom7	0, 4, 7, 10
major 7th	maj7	0, 4, 7, 11
minor 7th	min7	0, 3, 7, 10
suspended 4th	sus4	0, 5, 7
suspended 2nd	sus2	0, 2, 7
6th (major)	maj6	0, 4, 7, 9
minor 6th	min6	0, 3, 7, 9
9th (dominant)	dom9	0, 4, 7, 10, 14
major 9th	maj9	0, 4, 7, 11, 14
minor 9th	min9	0, 3, 7, 10, 14
diminished 7th	dim7	0, 3, 6, 9
add9	add9	0, 4, 7, 14
minor 11th	min11	0, 7, 10, 14, 15, 17
11th (dominant)	dom11	0, 7, 10, 14, 17
13th (dominant)	dom13	0, 7, 10, 14, 16, 21
minor 13th	min13	0, 7, 10, 14, 15, 21
major 13th	maj13	0, 7, 11, 14, 16, 21
7-5 (dominant)	dom7<5	0, 4, 6, 10
7+5 (dominant)	dom7>5	0, 4, 8, 10
major 7-5	maj7<5	0, 4, 6, 11
major 7+5	maj7>5	0, 4, 8, 11
minor major 7	minmaj7	0, 3, 7, 11
7-5-9 (dominant)	dom7<5<9	0, 4, 6, 10, 13
7-5+9 (dominant)	dom7<5>9	0, 4, 6, 10, 15
7+5-9 (dominant)	dom7>5<9	0, 4, 8, 10, 13
7+5+9 (dominant)	dom7>5>9	0, 4, 8, 10, 15

Table 2.1: Chord You Can Use

Notation There are two ways to notate a progression. In modern music sheets you mostly find the ABC-notation, where you write the exact chords including the tone letter and further tones or inversions with additional parameters. (e.g. "CMaj7/E" for an C major chord with an additional sept on base tone E). The problem of this notation is, that it is bound to a scale and not to the general cases. Remember: If you keep the relative distances, you can start on every key. Converting music from one scale into another is called *transposing*.

In order to serve the general case, that is not bound to a tonality, we can use a representation using roman letters from `I` to `VII`, that describe the relative relation between the different chords. Of course, if you want to play the progression, you have to set a base key (=`I`).

A JFugue *MusicString* can parse such progressions, with a class called *ChordProgression*, that accepts a progression and can convert it to the chords based on a base key (scale).

2.4.5.1 Example: Pop Progression

One of the most important progressions is the *pop Progression*. This progression is defined by the following sequence.

$$\mathtt{I} - \mathtt{V} - \mathtt{vi} - \mathtt{IV}$$

In the scale of C Major the chords would be C-G-Am-F.

The effect of this simple four chords is astonishing. An australian comedy group called 'Axis Of Awesome' has arranged a medley of short snippets of 50 famous pop songs just using these four chords. You can listen to it here.

Writing this in JFugue using a *MusicString* is pretty the same. We will use the function *addChordProgression()* we will later implement in SetlX .

```
addChordProgression("chords", "I V vi IV");
play("chords");
```

2.4.6 Triplets

You can also use tuplets in your music. Tuplets are groups of notes in which the duration of the notes is adjusted such that the duration of the group of notes is consistent with the duration of the next larger note duration.

You can write that in JFugue like the following:

```
addPattern("1:2", "Eq*3:2 Fq*3:2 Gq*3:2");
play("1:2");
```

You can also modify the attac and decay values:

```
addPattern("ad", "C5qa0d127");
       // Sharp attack (a0), long decay (d127)
play("ad");
```

For more detailed information about them please have look into chapter two the manual.

3 Method

Now, we want to show how the soundplugin that connects *JFugue* with SETLX was developed. Therefore new *User Defined Functions* and a *JFugue* backend were added to the SETLX core and can now be called by the interpreter.

3.1 Development Environment

For the development of the soundplugin *IDEA*s Java Development Environment Platform *IntelliJ* was used, but of course you can also develop or compile it using *your* preferred IDE or text editor (vim of course!).

In order to build the interpreter you need the following tools:

- Java JDK 8+
- Maven 3+

These are the versions that were used in the soundplugin development. So, older Maven versions might probabily not work correctly. *JFugue* requires at least Java 8, so we need to compile SETLX with Java 8, too.

3.1.1 Setting up Directories for the Sound Plugin

We add two folder structures to the existing directories:

- *example_setlX_sound_code* contains some example code and the *Mozart Dice Algorithm*
- *sound_addon* the source and test code for the plugin itself.

These folders are marked in blue in the following complete directory tree:

```
/
├── Tutorial
├── Documentation
├── example_SetlX_code
├── example_SetlX_sound_code
│   ├── documentation
│   └── example_code
├── example_SetlX_stat_code
├── grammar_pure
├── interpreter
│   └── sound_addon
│       ├── lib
│       │   └── jfugue
│       └── src
│           ├── main
│           │   └── (...)
│           └── test
│               └── (...)
└── syntax_highlighting
```

3.1.2 Add Sound-Plugin as Maven Subproject

A maven project can contain sub projects and we will add the sound plugin as a separate project to the Maven main *pom.xml*.

So, we just have to add one line to include a new sub project `sound_addon`.

```
[...]
<modules>
        <module>core</module>
        <module>pc_ui</module>
        <module>sound_addon</module> <!-- Including the sound plugin-->
        <module>gfx_addon</module>
        <module>plot_addon</module>
        <module>stat_addon</module>
        <module>tools</module>
</modules>
```

```
[...]
```

Next, we create another *pom.xml* in the `sound_plugin` directory, that contains build information for this subproject.

3.1.3 Modifiying the Compilation Scripts

There is a script called *createDistributions.sh* located in the top directory, which invokes maven and compiles the whole SETLX language and documentations. We have to modify a few lines in order to include the new sound plugin. This is also important for every new add on that is being implemented for in the future.

```
# [...] l. 30
if [[ -f "$OVERRIDE_setlXJarDirectory/setlX.jar" &&
    "$OVERRIDE_setlXJarDirectory/setlX-gfx.jar" &&
    "$OVERRIDE_setlXJarDirectory/setlX-plot.jar" &&
    "$OVERRIDE_setlXJarDirectory/setlX-stat.jar" &&
    "$OVERRIDE_setlXJarDirectory/setlX-sound.jar" ]];
then

# [...] l. 156

cp changelog.txt license.txt manual.pdf tutorial.pdf
    $OVERRIDE_setlXJarDirectory/setlX
    $OVERRIDE_setlXJarDirectory/setlX.cmd
    $OVERRIDE_setlXJarDirectory/setlX.jar
    $OVERRIDE_setlXJarDirectory/setlX-gfx.jar
    $OVERRIDE_setlXJarDirectory/setlX-plot.jar
    $OVERRIDE_setlXJarDirectory/setlX-stat.jar
    $OVERRIDE_setlXJarDirectory/setlX-sound.jar zipContents/
```

After these two lines (30 and 156) are modified, it should be possible to compile SETLX and the new sound plugin by executing the script with

```
$ ./createDistributions.sh
```

which simple does some tests and then invokes the maven build process. If you add a *–notests* as parameter, you can compile without all of the time consuming tests.

3.2 Integration of JFugue into SetlX

SETLX uses *Apache Maven* to manage the external binaries and libraries. In order to include the JFugue 5.0 Framework into SETLX , we have to add it as a dependency to maven and it will be automatically added to the accessible libraries.

3.2.1 JFugue 5.0 and maven

Maven Central is the official default maven repository, where maven automatically pulls external binaries. The problem is that the new version 5.0 of *JFugue* is not available in *Maven Central* yet (in March 2018). Maybe it will be added in the future, but at the moment, we need a workaround.

A solution for that is to use a local repository and place the *JFugue* *.jar there. Then maven can resolve the path to a local file and imports the *JFugue 5.0* dependency.

This is how it looks in maven *pom.xml* file for the soundplugin subproject:

```
1  <repositories>
2          <repository>
3                  <id>lcl</id>
4                  <url>file://${basedir}/lib</url>
5          </repository>
6  </repositories>
7
8  <dependencies>
9          [...]
10         <dependency>
11         <groupId>jfugue</groupId>
12         <artifactId>jfugue</artifactId>
13         <version>5.0.9</version>
14         </dependency>
15 </dependencies>
```

You can see, that in line 4 a local file path is added to the maven repositories and is marked as local (line 3: 'lcl').

3.2.2 Architecture

In this section, we want to analyze the general architecture of the plugin.

3.2.2.1 Packages

Excluding the unit tests, there are **five** main packages integrating the major functionalities of JFugue into SETLX . We will give a short description about what their purposes are.

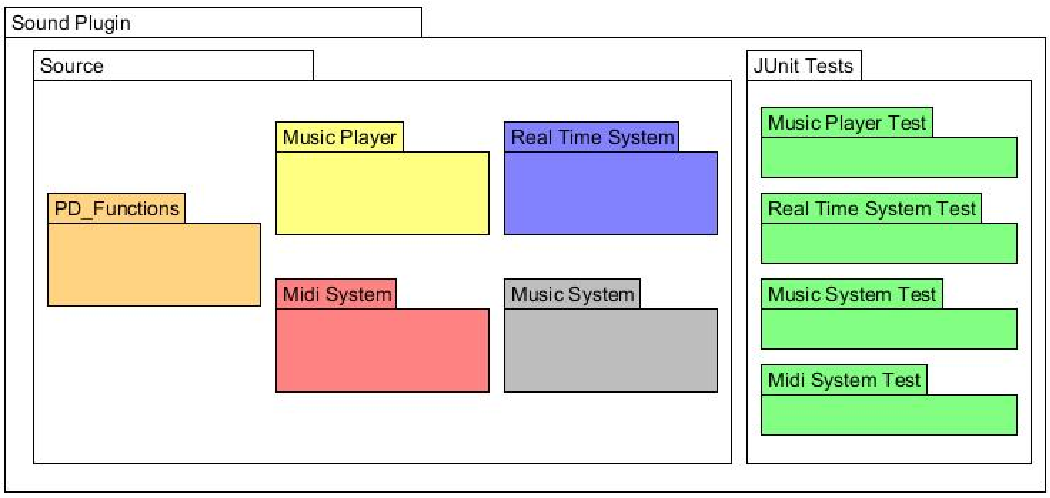

Figure 3.1: The Coarse Architecture of the Plugin

PD_Functions In order to make the new music operations accessible, we have to declare some new functions to the interpreter. They are called `PreDefinedProcedures` and will be discussed in 3.2.4.

Music Player This package contains all classes that are necessary to play the patterns of music that are stored in SETLX during runtime.

Real Time System This package adds programming methodologies for processing (and playing) music in real time without any intermediate steps of creating objects for just a simple playback. Basically it provides the backend for a *playTone* function, which you can be directly used from the SETLX interpreter to play a tone without needing to store information about it. The tone will immediately be played after the function is called.

Midi System This package manages everything that is related to midi file in- and output. We can export the music written in SETLX into a valid midi file and play it in other media players that support the midi-codec (like *timidity*, *vlc*, ...). On the other side, you can also import a midi file, modify and play it in SETLX . For instance, you can download some midi files of *Bach* music, load them into SETLX and then play them.

Music System The music system can store and manage *JFugue PatternProducer* objects. A *PatternProducer* is an JFugue interface implemented by classes that create actual music content. In other words, it can save actual music information and operate with them.

JUnit tests This package contains the unit tests for the plugin. The playback of music is pretty difficult to test, so we just focus on the *Music System* component.

Furthermore, it is also difficult to test `PreDefinedProcedures`. The reason is that they depend on the runtime of the interpreter.

3.2.2.2 Data Flow Graph

We have to connect our `PreDefinedProcedures` with our sound plugin backend. Because every `PreDefinedProcedure` stands completely alone, we need a class that manages the whole backend and gives the `PreDefinedProcedure` access to the subsystems it needs. Therefore we implement a global class that serves as the entry-point providing getter methods returning references to the subsystems.

After the interpreter is started, this class automatically initializes all of the sound plugin components.

Figure 3.2 shows how this process works. A user calls a `PreDefinedProcedure` from the interpreter. This function has access to the main instance of the soundplugin, which returns the required subsystem. After that, the `PreDefinedProcedure` has access to the methods it needs.

An example data flow, which is also visualized in 3.2 might be:

1. The user calls *playTone(40)* from the interpreter
2. The interpreter connects to the soundplugins' *Main* class
3. If the subsystems are not initialized yet, it will first create the objects of the subsystems
4. The *Main* class returns the requested *Real Time System* to the interpreter
5. The interpreter can now call the *play*-method of the *Real Time System*
6. The interpreter returns the success of the operation to the user and is ready for next function calls.

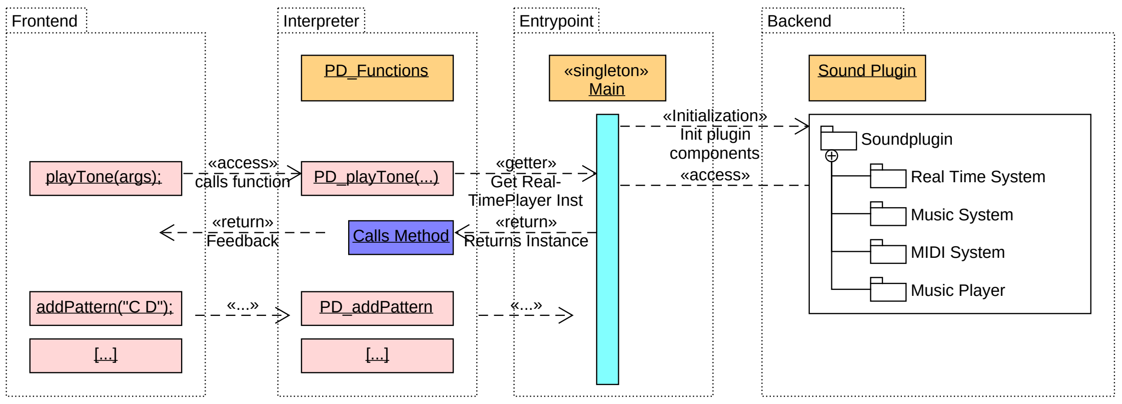

Figure 3.2: The Connection between the Interpreter and the Soundplugin

3.2.2.3 Storage for Music Information

Now we want to temporary store music information supplied by a user during runtime. At the moment, JFugue has three different classes for music information: A *ChordProgression* class containing a sequence of chords, a *Rhythm* class for playing percussions in a simplified way and a *Pattern* class that can hold music songs. All of these JFugue classes inherit from a super class *PatternProducer*.

So, if we want to store them, we can generate a generic class that takes objects implementing the *PatternProducer* interface. The following codesnippet (3.1) shows the main methods of this class. It simply provides an interface to a map of elements.

The implementation of this interface specification then just uses a *HashMap* from the Java Collection API in order to save a key-value. A unique String that can later be used to find music elements in the storage.

```java
public interface iMusicStorage<T extends PatternProducer> {
       void addElement(String name, T element)
              throws NullArgumentsException;
       boolean checkExisting(String name);
       T getElement(String name)
              throws PatternNotFoundException;
       HashMap<String, T> getAllElements();
}
```

Code 3.1: Storage for Music Information

3.2.2.4 Main Class

As shown in 3.2 the main class provides an interface to all parts of the soundplugin. Therefore it creates objects for all different systems at startup: The *Real Time System*, *Music Systen* and the *Music Player*.

Because we only need one main class per runtime, we implement it using the *Singleton* design pattern.

```java
public class SoundPlugin implements iSoundPlugin {
       private static SoundPlugin setlxSoundPlugin;
       private iMusicPlayer musicPlayer;
       private iMusicManager musicManager;
       private iRealTimePlayer realTimePlayer;
       private iAtomFactory atomFactory;
       // ...
       private SoundPlugin() throws MidiNotAvailableException {
              initializeComponents();
       }
```

```
11      private void initializeComponents() throws
            MidiNotAvailableException {
12              atomFactory = new AtomFactory()
13              // ...
14              musicManager = new MusicManager();
15              musicPlayer = new MusicPlayer(musicManager);
16              realTimePlayer = new RealTimerPlayer(noteFactory,
                    atomFactory);
17      }
18      @Override
19      public iMusicManager getMusicManager() {
20              return musicManager;
21      }
22      // More getter methods here...
```

We will shortly discuss this code:

1. In line 1, we implement the interface *iSoundPlugin*. Every class in the project does implement an interface that specifies the method signatures of the implementing class.

2. In line 2, we define a static variable that holds the reference to the *SoundPlugin* singleton object.

3. The constructor is set to private in line 9, so that is impossible to create new objects from outside that class.

4. The *initializeComponents()* method from line 12 to 18 creates the subsystem objects and is called by the constructor.

5. All others methods of this class just returns the reference to the subsystem objects. (Line 19)

The following static *getInstance()* method finishes the implementation of the Singleton pattern. It is globally accessible and takes care that only one object can be created at the same time. An advantage is that the soundplugin objects are only created, when they are really needed. As long as no soundplugin specific function is called by the interpreter, we do not have to create the sound plugin objects.

```java
	public static SoundPlugin getInstance() {
		if (setlxSoundPlugin == null) {
			try {
				setlxSoundPlugin = new SoundPlugin();
			} catch (MidiNotAvailableException e) {
				e.printStackTrace();
			}
		}
		return setlxSoundPlugin;
	}
}
```

3.2.2.5 User Defined Functions

You can add own `PreDefinedProcedures` to SetlX . The main idea behind is that every function to be added get its own java class inheriting from `PreDefinedProcedure` insides a *functions* folder. This class has to follow certain conventions in order to be recognized as a function:

1. Located in *functions* folder
2. "PD_" prefix in file- and classname
3. Inheritance from `PreDefinedProcedure`

The following code snippets shows the class *PD_addChordProgression* that adds a new `ChordProgression` to a runtime storage.

```java
package org.randoom.setlx.functions;
import org.jfugue.theory.ChordProgression;
// ...
public class PD_addChordProgression extends PreDefinedProcedure {
```

Because we also want to pass parameters to the function, they are defined by a global `createParameter`-method and then stored to a variable. We later bind this parameter to an object in the constructor.

```
        private final static ParameterDefinition PATTERN_NAME =
            createParameter("patternName");
        private final static ParameterDefinition CHORD_PROGRESSION =
            createParameter("chordProgression");
        private final static ParameterDefinition KEY =
            createOptionalParameter("key", SetlString.
            newSetlStringFromSB(new StringBuilder("C")));
        // With default value
```

In order to later access our function, we need to call a default constructor for our `PreDefinedProcedure`.

```
        public final static PreDefinedProcedure DEFINITION = new
            PD_addChordProgression();
```

Because our plugin has a main class that can be used to access all components of the Soundplugin, we can get a reference to that object by a *getInstance()*-method. This follows the conventions of the *Singleton*-pattern

```
        SoundPlugin root = SoundPlugin.getInstance();
```

Of course, we also have to declare the default constructor, where we add the parameters that were defined globally to the *ChordProgression* object.

```
protected PD_addChordProgression() {
        super();
        addParameter(PATTERN_NAME);
        addParameter(CHORD_PROGRESSION);
        addParameter(KEY);
}
```

The magic happens in the inherited method `execute`, which is being executed every time the method is called. The parameters are passed by a HashMap, which holds all of the values passed by the caller.

```
        @Override
        protected Value execute(final State state, final HashMap<
           ParameterDefinition, Value> args) throws SetlException {
                final Value patternName = args.get(PATTERN_NAME);
                final Value chordProgression = args.get(
                   CHORD_PROGRESSION);
                final Value key = args.get(KEY);
```

Now we can use these values to call the backend as shown in 3.2. In this case, we use a `ChordProgressionFactory` to generate a new `ChordProgression` object and then add it to the *MusicManager*. At the end, we simply return that the operation was successful. Of course, we could also return a real value that was calculated by the execute method. A use case for that would be implementing new mathematical functions.

```
        ChordProgression cp = root.getChordProgressionFactory()
        .createChordProgression(
        chordProgression.getUnquotedString(state), key.
           getUnquotedString(state));
        root.getMusicManager().add(patternName.getUnquotedString(state)
           , cp);
        return SetlBoolean.TRUE;
        }
}
```

3.2.3 Exception Handling

Runtime errors can be caught by SETLX and displayed without leaving the interpreter environment. Therefore, a new defined exception must inherit from the class *CatchableInSetlXException*.

Example The following example shows the implementation of an exception that is thrown, when a given key is already in use. This can for example happen, when you try to add two patterns with the same name.

```
package org.randoom.setlx.setlXMusic.musicSystem.exceptions;
import org.randoom.setlx.exceptions.CatchableInSetlXException;
public class KeyAlreadyInUseException extends
    CatchableInSetlXException {
        public KeyAlreadyInUseException() {
                super("The key/name you want to use is already in use!")
                    ;
        }
}
```

3.2.4 Application Programming Interface (API)

The following pages list all of the new functions of the soundplugin that can directly be called by the interpreter and give some examples on how to use them.

A question mark in the parameter list means that it is optional and predefined if you do not pass it.

3.2.4.1 addChordProgression

```
addChordProgression(patternName: string, chordProgression: string,
    key?: string = "C")
```

Parameter	Description
patterName	A unique identifier for this *ChordProgression*
chordProgression	A string containing a *ChordProgression*
key	The base key (=Tonic) of the *ChordProgression*

Creates a new chord progression from a given *progression string* **chordProgression** and adds it to the the current SETLX runtime environment. A *ChordProgression* is a sequence of roman numbers. For minor chords use lower capitals. You can find more information about *ChordProgression* strings in 2.4.5. The **patternName** is the identifier for the pattern that is used, whenever you refer to this pattern.

The **key** identifies the tonic chord. Because a progression just shows the relative distances of the chords, you need to specify a base key that is used for the tonic chord. For example the progression `I-IV-V-I` would play the chords *"Cmaj Fmaj Gmaj Cmaj"* on 'C' as base key or *"Fmaj Bbmaj Cmaj Fmaj"* on 'F'.

You can also modify a chord progression using the functions *allChordsAs* and *eachChordAs*.

```
$ addChordProgression("prog1", "I IV V I");
        // Simply adds I IV V I progression
$ addChordProgression("prog2", "I IV V I", "F");
        // Same progression as ex1, but on F-Key
& addChordProgression("prog3", "I V vi iii IV I IV V")
        // The famous Canon progression by Pachelbel using minor chords
$ play("prog1");
        // In order to actually play the progression
```

3.2.4.2 addPattern

```
addPattern(patternName: string, pattern: string, voice?: int = 0,
    tempo?: int = 0, instrument?: int = 0)
```

This is the most general and recommended way to add music. The function takes a pattern name and the pattern itself and saves it to the current SetlX -runtime.

It is not recommend to use the voice, tempo and instrument parameters, because it can have negative side effects, when the pattern also contains changes in them. It is better to use the additional information in the *MusicString* like described in 2.4.4.1

```
addPattern("pat1","C D E F G A B");
        // C-major scale
addPattern("pat2", "V1 C D E F V2 E F G B");
        // thirds played simultanous
addPattern("pat3", "T60 V0 F6i D6i G6i V1 F4i D4i G4i V2 Rq.");
        //A more complex pattern
addPattern("pat4","V0 I[Piano] Eq Ch. | Eq Ch. | Dq Eq Dq Cq V1 I[Flute] Rw | Rw
    | GmajQQQ CmajQ");
        // ~~~ " ~~~
addPattern("pat5","C D E F", 1, 120, 5)
        // Using more parameters. A mapping table for num <-> instruments can be
            found in the manual
```

3.2.4.3 addPatternsToPattern

`addPatternsToPattern(patternFrom: string, patternTo: string)`

Parameter	Description
patternFrom	A list of pattern names that will be added to the destination
patternTo	The destination pattern in which to insert

If you want to add a pattern at the end of another, you can use this function.

patternFrom is a list with the names of the patterns you want to copy into the end of **patternTo**. If you specify more than one pattern in patternFrom, it will add the patterns in the same order as in the string.

```
addPattern("A","C D E F");
addPattern("B","G H B C6");

addPatternsToPattern("B","A");
        // Adds pattern "B" to "A"
addPatternsToPattern("B B","A");
        // Adds "B" twice to "A"
showMusic();
```

3.2.4.4 addRhythm

`addRhythm(rhythmName: string, pattern: string)`

Parameter	Description
rhythmName	The name of the rhythm element
pattern	A *RhythmString*

*RhythmicProgression*s can be used to add percussion to the music.

```
addRhythm("rhy1","xxxxxxxx");
addRhythm("rhy2","x.x.x.x.x.");

play("rhy1");
```

3.2.4.5 addToPattern

`addToPattern(name: string, pattern: string)`

Parameter	Description
name	The name of the existin pattern
pattern	A MusicString that contains the musical information

Adds a new pattern at the end of an existing pattern. Therefore you can specify a *MusicString*, which will directly be added at the end of the existing pattern.

```
addPattern("add1","C D E F");
addToPattern("add1","V1 G V2 Rw C");
        // Adds the MusicString "V1 G [...]" to pattern "add1"
showMusic(); // To see the effect
```

3.2.4.6 allChordsAs

`allChordsAs(chordPatternName: string, modifyString: string)`

Parameter	Description
chordPatternName	The name of the *ChordProgression*
modifyString	A string that defines the modification

Can be used to specify the duration of the chords of a *ChordProgression*. By default, every Chord has the duration of a whole note.

You refer to the chords with the separator char '#' followed by the number of the chord. "#1w" will be interpreted as: "The first Chord has a duration of a **whole** note".

```
addChordProgression("mdC1","I IV V I");
allChordsAs("mdC1","#1w #2h #3q #4a");
        // First Chord: whole note, Second: half, ...
showMusic();
```

3.2.4.7 duplicatePattern

`duplicatePattern(patternSourceName: string, patternNewName: string)`

Parameter	Description
patternSourceName	The name of the pattern to be copied
patternNewName	The name of the new pattern

Creates a deep copy of a pattern. Technically, this means that the *MusicString* that defines an existing pattern is hard-copied into a new one.

```
addPattern("A","C D E F G A B C");
duplicatePattern("A","A_copy");
        // Copies the pattern "A" into a new one called "A_copy"
showMusic(); // To see the effect
```

3.2.4.8 eachChordAs

`eachChordAs(patternName: string, modifyString: string)`

Parameter	Description
patternName	The name of the *ChordProgression* you want to modify
modifyString	A string that defines the modification

Requires passing a string that has '#' followed by an index, in which case each hashsign+index will be replaced by the indexed note of the chord for each chord in the progression. Using the underscore character instead of an index will result in the chord itself added to the string.

```
addChordProgression("alb","I IV V I");

eachChordAs("alb","#1e #3e #2e #3e #1e #3e #2e #3e");
        // The famous Alberti bass
```

3.2.4.9 getPatternStats

`getPatternStats(patternName: string)`

Parameter	Description
patternName	The name of the pattern you want to see statistics

Prints some interesting statistics about an existing pattern. They contain information about the number of notes, rests and measures.

It will also print calculated N, Average, SD and Range values of harmonics, rests, pitches, duration and intervals, which can be useful for music analysis.

```
addPattern("stt","Cq Eh Gq Rq");
getPatternStats("stt");
```

3.2.4.10 loadMidi

`loadMidi(fileName: string, patternName: string)`

Parameter	Description
fileName	The name of the MIDI file to be parsed. Can also contain paths, if the file is located in another directory.
patterName	A name for the new pattern with the parsed MIDI information

You can load existing MIDI files into SetlX . It will create a new pattern and fill it with the MIDI-information parsed from an external MIDI file.
You can find some demo midi files here.

```
loadMidi("../someBach.mid", "I<3Bach");
showMusic("I<3Bach");
```

3.2.4.11 modifyPatternProperty

```
modifyPatternProperty(patternName: string, property: string,
    value: double)
```

Parameter	Description
patterName	The name of the MIDI file to be parsed. Can also contain paths, if the file is located in another directory.
property	A name for the new pattern with the parsed MIDI information
value	A real value for the property.

Modifies one of the following properties of a whole pattern: voice, tempo, instrument. This function has the same effect as setting the properties directly in *addPattern*(3.2.4.2).

Just use this function when you really need it! Setting one of these parameters can have interferences with music information given in the string itself. For example playing the pattern can have side effects, you properly do not see at the first glimpse.

```
        addPattern("mod1","C D E F G A B C");
        modifyPatternProperty("mod1","voice",1)
                // Sets VOICE to 1
        modifyPatternProperty("mod1","TEMPO",120)
                // Sets tempo to 120 BPM
        modifyPatternProperty("mod1","instrument",1)
                // Sets instrument to piano
```

For more information about instrument values again please have a look into the JFugue manual.

3.2.4.12 play

`play(patternNames: string)`

Parameter	Description
patternNames	The names of the pattern to be played in a single string. More patterns are separated by a *blank* like "pattern1 pattern2"

Maybe, **this** is the most essential function directly after *addPattern()*, because it allows to actually *play* existing patterns. All you have to do is passing the name of the *Pattern*, *ChordProgression* or *RhythmProgression* that you had added to the runtime storage.

```
addPattern("ply1", "V1 C D E F G A B C6"); // C-maj scale
addPattern("ply2", "V2 E F G A B C6 D6 E6");
        // Another voice in a seperate pattern
play("ply1");
        // Plays one pattern
play("ply1 ply");
        // plays pattern ply1 two times, because it just uses one Voice (1)
play("ply1 ply2");
        // plays pattern "ply1" and "ply2" simultaniously, because they use
           different voices (1 & 2)
```

3.2.4.13 playTone

`playTone(note?: double = 0, duration?: double = 0, instrument?:`

```
double = 0, voice?: double = 1, layer?: double = 1)
```

Parameter	Description
note	An integer representation of the note to be played. You can find some important mapping values in the table below.
duration	The tempo of the tone BPM
instrument	An instrument for this single tone
voice	A voice of this single tone. If you want to play multiple tones at the same time, you have to use this argument.
layer	The layer of the tone

This function adds real time audio processing to SETLX . It can be used to generate tones and directly play them using audio interfaces. Of course, you can combine the parameters with variable values.

If you want to wait for the music to be finished, you can use the *sleep()* function and wait for the end of the playback.

You can calculate the duration of multiple tones using the following formula

$$\sum_{\texttt{note} \in \texttt{QueuedNotes}} 60/\texttt{BPM}(note)$$

where $BPM : note \rightarrow Int$ function that returns the speed of a given note.

If all notes have the same tempo *BPM* you can use the following formula:

$$n \cdot 60/BPM$$

Where n is #QueuedNotes.

```
print("Simple Demo for Real Time Music");
playTone(40); // A single tone

for(x in {40..50}){
        playTone(x, 120); // Just a few notes; ~5 seconds: 120 BPM with 11 notes
}
for(x in {40..50}){
        playTone(x, 120,x-39); // Using different instruments; ~5 Seconds
}
```

```
for(x in {30..90}){
        playTone(x, 7*x); // Accelerando; ~13 Seconds, 61 tones on Average of
            90*7/2=315 BPM
}
for(x in {40..55}){
        playTone(x, 120,2,1); // Two Voices at the same time; 13 Seconds
        playTone(x+3, 180,2,2); // Playing a minor third; Polyrhythmical
}
sleep(36000); // wait for playback to be finished for 36 seconds
```

3.2.4.14 removeMusic

`removeMusic(patternName: string)`

Parameter	Description
patternName	The name of the *Pattern*, *ChordProgression* or *RhythmProgression* to be removed.

Removes a *Pattern*, *ChordProgression* or *RhythmProgression* from the runtime storage.

```
addPattern("rem1","C D E F G A B C");
removeMusic("rem1"); // removes the pattern from runtime storage
showMusic(); // To see the effect
```

3.2.4.15 saveAsMidi

`saveAsMidi(patternName: string, fileName: string)`

Parameter	Description
patternName	The name of the *Pattern*, *ChordProgression* or *RhythmProgression* to be removed.
fileName	The name and path of the MIDI output file

If you want to share a pattern created with SetlX , you can create a MIDI output file. MIDI is a standardized format allowing you to use your music everywhere. Many digital instruments also support the MIDI-format and you can load your music there.

```
addPattern("mid1","C D E F G A B C6");
saveAsMidi("mid1","output.mid");
```

3.2.4.16 saveAsPattern

`saveAsPattern(elementName: string)`

Parameter	Description
elementName	The name of the *ChordProgression* or *RhythmProgression* to be converted

Because *ChordProgressions* and *RhythmProgressions* are often difficult to handle with, you can convert them into a valid pattern and add it into the pattern storage. The reason, why this works is that *ChordProgressions* and *RhythmProgressions* are just subsets of patterns, with some special functions.

It will automatically get the ending '[oldName]_conv'.

```
addChordProgression("cpr1","I IV V I");
addRhythmProgression("rhy1" ,"xxxxxxxx");

saveAsPattern("cpr1");
saveAsPattern("rhy1");
showMusic(); // New name: "cpr1_conv" and "rhy1_conv"
```

3.2.4.17 setKeyForChordProgression

`setKeyForChordProgression(elementName: string, key: string)`

Parameter	Description
elementName	The name of the *ChordProgression* you want to change the base key
key	A character representing the tonic chord

You can change the base key that is equal to the tonic chord of a *ChordProgression*. It has the same effect as setting the optional key parameter in *addChordProgression*.

```
addChordProgression("KFC1", "I IV V I","C");
play("KFC1"); // plays it in C-Major (Base Key 'C')
setKeyForChordProgression("KFC1","F");
play("KFC1", "F"); // plays it in F-Major (Base Key 'F')
```

3.2.4.18 showMusic

`showMusic()`

Prints a list showing all *Patterns*, *ChordProgressions* and *RhythmProgressions* that are currently stored in the SETLX interpreter.

```
addChordProgression("SMu1", "I IV V I","C");
addPattern("SMu2","C D E F G A B C6");
       // Adds some music...
showMusic();
```

3.2.4.19 stopTones

`stopTones()`

This function immediately stops the playback of the real time player. It also removes all queued notes.

```
for(i in {40..100}){
       playTone(x);
} // Fills the queue with some tones ...

stopTones(); // ... aaand stop
```

3.3 The Mozart Dicing Game

As a simple demonstration of what the SETLX -soundplugin can do, we want to implement a musical dicing game by one of the most famous composers of all times: *J. W. Mozart*[1].

3.3.1 A Description of the Game

Music dicing games were pretty popular during the *Classical Period*. Even Haydn and other famous composers tried to construct such games. The principle is pretty simple: A composer constructs a harmonic progression and than writes down different melodies/motives for every bar that are based upon this progression. As a

[1]Exact name: Johannes Chrysostomus Wolfgangus Theophilus Mozart

consequence, you have more choices for every bar fulfilling the same function in the piece of music: The music always sounds good and coherent.

At the heart of the dice game is a large collection of musical fragments. Each fragment is a single 3/8 measure, consisting of a treble voice and a bass voice. Traditionally, these fragments are stored in a "score", or "table of measures", and located via two tables of measure numbers, which act as lookups, indexing into that collection. The "player" then uses dices to randomly create his own personal piece of music!

Figure 3.3 is an extraction showing fragments of Mozarts' dicing game.

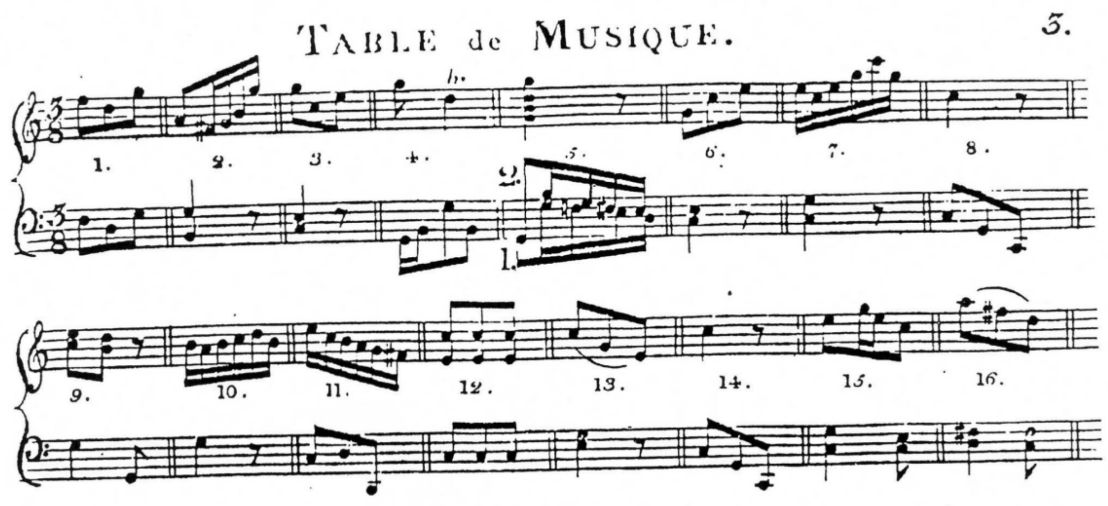

Figure 3.3: Extraction from Mozarts' "Ein musikalisches Würfelspiel" (A musical Dicing game)

Figure 3.4 on page 37 shows, how the measures are linked to the sum of the two dices on the nth roll. For instance, if you throw a **5** and a **6** in the first roll, it means that you play fragment 3 (A simple tonic chord) in the first (A) measure. You can see that fragment in figure 3.3.

3.3.2 Implementation in SetlX

Now we will give a possible implementation in SETLX using the new awesome functionalities provided by the SETLX Soundplugin.

```
// ~~~ MOZART DICING GAME ~~~
addPattern("1","V0 F6i D6i G6i V1 F4i D4i G4i V2 Rq.");
```

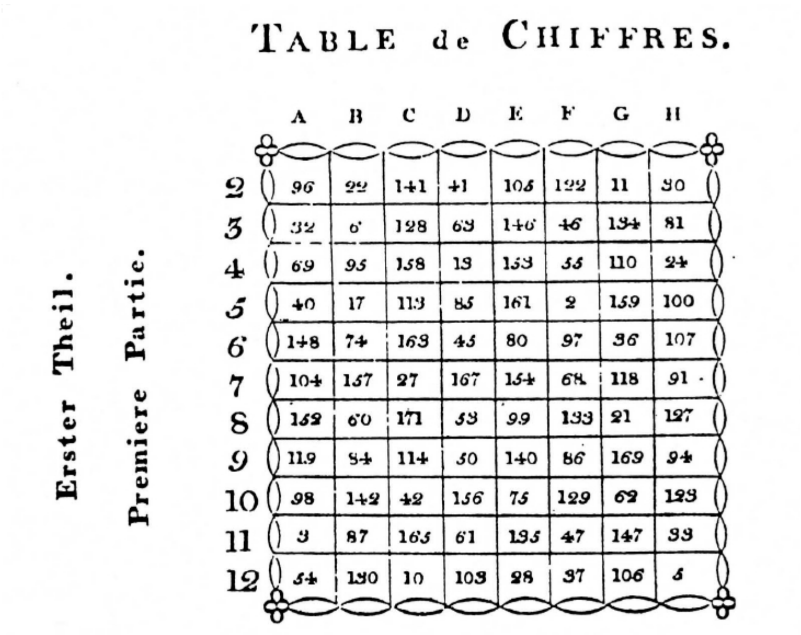

TABLE de CHIFFRES.

Erster Theil.

Premiere Partie.

	A	B	C	D	E	F	G	H
2	96	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

Figure 3.4: Extraction from the Map Table

```
addPattern("2","V0 A5i F#5s G5s B5s G6s V1 B3q Ri V2 G4q Ri");
// [...]
addPattern("174","V0 G5i C5i E5i V1 E4s G4s E4s G4s E4s G4s V2 C4s Rs C4s Rs C4s
    Rs");
addPattern("175","V0 E6s C6s B5s D6s G6i V1 G4i G3i Ri V2 Rq.");
mapTable_firstPart := [
        [96, 32, 69, 40, 148, 104, 158, 119, 98, 3, 54],
        [22,6,95,17,74,157,60,84,142,87,130],
        [141,128,158,113,163,27,171,114,42,165,10],
        [41,69,19,85,45,167,53,50,156,61,103],
        [108,146,159,161,80,154,99,140,75,135,28],
        [122,46,55,2,97,68,133,86,129,47,37],
        [11,134,110,159,36,118,21,169,62,147,106],
        [30,81,24,100,107,91,127,94,123,33,8]
];
mapTable_secondPart := [
        [70,117,66,90,25,138,16,120,65,102,85],
        [121,39,139,176,143,71,155,88,77,4,29],
        [26,126,15,7,64,150,86,48,19,91,108],
        [9,56,132,94,125,29,175,166,82,164,92],
        [112,174,73,67,76,101,43,51,137,144,12],
23        [49,1858,160,136,162,168,115,39,59,124],
24        [109,116,145,58,1,23,89,72,149,178,44],
25        [14,83,79,170,99,151,172,111,8,78,191]
26  ];
27  choices := []; choices_sec := [];
28  for(x in {1..8}){
29        rand := rnd({1..12});
30        randP2 := rnd({1..12});
31        choices += [mapTable_firstPart[x][rand]];
32        choices_sec += [mapTable_secondPart[x][randP2]];
33  }
34  choices += choices_sec; // Adds the second part to the first one
35  addPattern("Song","T60");
36  for(x in choices){ addPatternsToPattern(x,"Song"); }
37  play("Song");
38  saveAsMidi("Song");
```

Figure 3.3.2 on page 36 shows an extraction from the implementation of the *Mozart Dicing Game* in SETLX . We discuss this implementation line by line:

1. In line 2 to 6, we directly define the patterns Mozart has developed using the *addPattern()*-function. Note: 180 other patterns are hidden. You can compare the *MusicString*s of the patterns to the original sheet of music shown in 3.3 on page 36. In order to be able to play the left and right hand at the same time, we have to use multiple voices. (`V0` and `V1`). For example, fragment 1 can directly be written using a *MusicString* shown in figure 3.5.

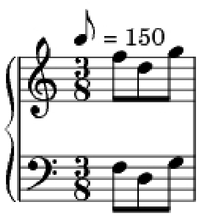

Figure 3.5: "`V0 F6i D6i G6i V1 F4i D4i G4i V2 Rq.`"

2. In line 7 and 17, we define two map tables. We just need a two dimensional list for this: The first entry is for the measure, the second for the bijective mapping *DiceSum* $\leftrightarrow$ *FragmentNumber*

3. Lines 28 to 33 generate the random waltz. Therefore we iterate over eight bars and create two random numbers in every iteration: One for the first part and the other for the second.

   Note: This implementation is a bit different from reality, because choosing a random number from the set $\{1..12\}$ guarantees that every pattern has the same probability to occur. In order to be nearer to the original concept, we could simply use two random numbers from 1 to 6 (dices) and add them. But the author thinks that it is more interesting to keep the statistical probabilities equal.

4. In line 31 and 32 the fragments that belong to the random numbers are added to `choices`.

5. In line 34 we add the second part to the end of the first one.

6. Line 35 creates a new container pattern `Song` with the meta information of *tempo 60 BPM*.

7. In line 36, we add all selected patterns to our song. Therefore, we iterate over the `choices`-list and add each element to `Song`.

8. Last but not least, we want to listen to our personal waltz. Line 37 uses the *play()*-function to play the song and saves it as a MIDI-file to the local file system.

Bibliography

[1] amandaghassaei. What is midi? http://www.instructables.com/id/What-is-MIDI/, 2012. (16/04/18).

[2] Andrew Sorensen & Andrew Brown. jmusic - music composition in java. http://explodingart.com/jmusic/jmNews.html, 2009. (16/04/18).

[3] Kemal Ebcioglu. An expert system for chorale harmonization. http://charlesames.net/pdf/KemalEbcioglu/choral.pdf, 1999. (28/03/18).

[4] David Koelle. The complete guide to jfugue. http://www.jfugue.org/4/jfbmrkklprpp/TheCompleteGuideToJFugue-v1.pdf, 2015. (16/04/18).

[5] John A. Maurer. A brief history of algorithmic composition. https://ccrma.stanford.edu/~blackrse/algorithm.html#mozart, 1999. (28/03/18).

[6] Prof. Karl Stroetmann. setlx. http://download.randoom.org/setlX/tutorial.pdf, 2017. (26/04/18).

[7] Aiva Technologies. Aiva company homepage. http://www.aiva.ai/, 2018. (28/03/18).