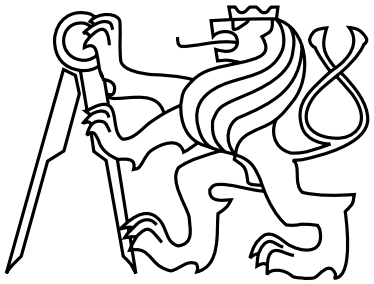
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

Salzella - A Declarative Language for Music Generation

Štěpán Volf

Supervisor: doc. Ing. Karel Richta, CSc.

Study Programme: Open Informatics, Master's Program

Field of Study: Software Engineering

January 11, 2016

Aknowledgements

I would like to express my gratitude to doc. Ing. Karel Richta, CSc. for the time he invested in supervising my thesis. I would also like to thank the participants of the usability tests and interviews whom I annoyed relentlessly and without mercy. Special thanks goes to Petr Blšťák and Jan Herzán for helping me set up and perform these tests. Last but definitely not least, thanks to my family for supporting me throughout my studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 15, 2015 .....

Abstract

Salzella is a domain specific declarative language. Its primary focus lies in the field of music generation. The key principle upon which the language is built can be summarized as follows: For any existing piece of music it must be possible to create a Salzella program which will describe it in a way that running this program will output a piece of music similar to the original. Salzella programs are presumed to be generated rather then written by hand. The sole purpose of Salzella is to make creation of music generating tools easier. Bundling Salzella interpreter with music generating software and using it as an engine can provide a significant level of abstraction and thus make the development of such software a lot easier. Tools built on top of Salzella can generate Salzella programs and let the interpreter worry about the actual music generation. To prove that Salzella interpreter can indeed be used as internal engine of music generating tools, a prototype of an algorithm for converting musical pieces into Salzella programs was created.

Apart from creating Salzella specification, the following tools were developed as part of this thesis: Salzella interpreter, Salzella development environment and Salzella extensions.

Abstrakt

Salzella je doménově specifický deklarativní jazyk, jehož využití lze nalézt především v oblasti generování hudby. Klíčový princip, na kterém je jazyk postaven, lze shrnout následovně: Pro jakékoli existující hudební dílo musí být možné vytvořit program v jazyku Salzella, který toto dílo popíše tak, že spuštěním tohoto programu dojde k vygenerování hudební díla podobného hudebnímu dílu originálnímu. Hlavním cílem jazyka Salzella je usnadnit vývoj nástrojů pro generování hudby. Nástroje postavené nad platformou Salzella mohou místo hudby generovat programy v jazyku Salzella a samotné generování hudby ponechat na interpretovi tohoto jazyka. Pro ilustraci možnosti využití interpreta jazyka Salzella jakožto interní komponenty nástroje pro generování hudby byl vytvořen prototyp algoritmu pro převod hudebních děl do programů zapsaných v jazyku Salzella.

Krom specifikace jazyka Salzella byly v rámci této práce vytvořeny následující nástroje: interpret jazyka Salzella, vývojové prostředí pro jazyk Salzella a několik ukázkových rozšíření jazyka Salzella.

Contents

Chapter 1

Introduction

1.1 Fundamental requirements

1.1.1 Descriptive nature

Salzella is a domain specific declarative language. Its primary focus lies in the field of music generation. The key principle upon which the language is built can be summarized as follows: For any existing piece of music it must be possible to create a Salzella program which will describe it in a way that running this program will output a piece of music similar to the original. The extent to which the resulting piece of music will resemble to the original can vary and is entirely in the hands of the creator. Note that running the same Salzella program repeatedly may or may not produce different outputs. However, as illustrated by Figure 1.1, producing different outputs upon repeated execution is presumed to be the typical behavior. The fact that Salzella programs must be capable of capturing the essence of arbitrary musical piece implies music genre independence.

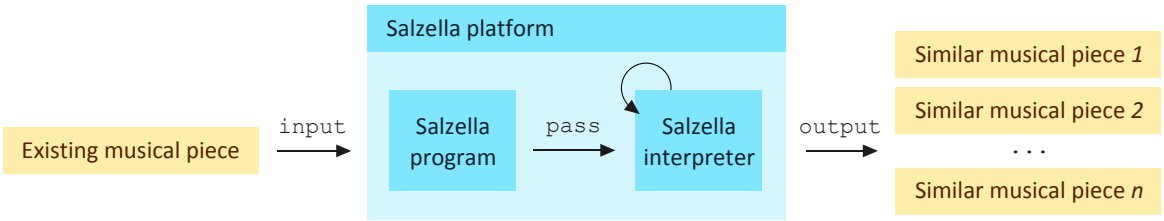


Figure 1.1: Using Salzella to capture the essence of an existing musical piece

1.1.2 Bundle requirement

The sole purpose of Salzella is to make creation of music generating tools easier. Bundling Salzella interpreter with music generating software and using it as an engine can provide a significant level of abstraction and thus make the development of such software a lot easier. Tools built on top of Salzella platform are not assumed to generate music directly. They can generate Salzella programs and let the interpreter worry about the actual music generation.

The fact that Salzella programs are presumed to be generated rather than written by hand had great impact on design of the language. Neither human readability nor ease of manual creation can be found on the list of requirements. Figure 1.2 illustrates the typical workflow of application built on top of Salzella platform.

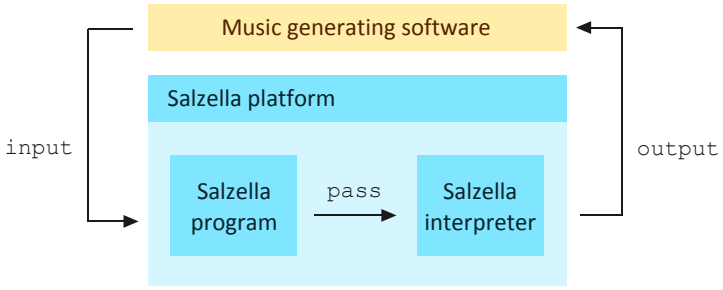


Figure 1.2: Salzella interpreter as internal component of a music generating tool

1.1.3 Absorption requirement

The most important architectural aspect of Salzella platform is extensibility. Absorption requirement states that Salzella must be able to integrate any existing music generating algorithm in a form of a plugin. As illustrated by Figure 1.3, arbitrary number of external algorithms can be run during execution of a single Salzella program. Capability of absorbing external algorithms is a key structural feature which allows Salzella users to customize its behavior.

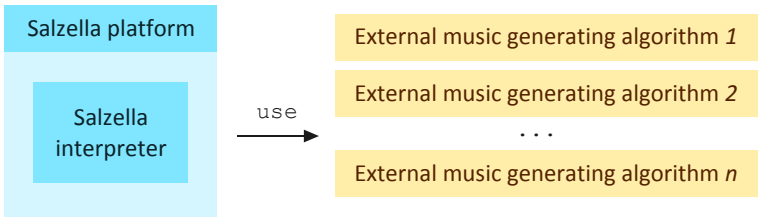


Figure 1.3: Salzella can leverage already existing music generating algorithms

1.2 Goal declaration

1.2.1 Specification and interpreter

Primary goal of this thesis was to create a specification which would satisfy the fundamental requirements presented in the previous chapter. In attempt to make Salzella more versatile in terms of satisfying the music genre independence requirement, I have decided to design the language without explicit support for abstract structures like chords, scales and keys. Instead of relying on built-in support of these structures, Salzella users are meant to define these

manually within individual programs. Secondary goal of this thesis was to test feasibility of this approach. And finally, in order to be able to test the designed language in practice, implementing its interpreter was necessary.

1.2.2 Development environment

To make creation and execution of Salzella programs easier, realization of Salzella development environment was added on the list of goals. The key requirements regarding its functionality were possibility of manual creation and execution of Salzella programs and ability to visualize and play back the generated snippets of music.

1.2.3 Extensions and conversion tool

To prove that absorption requirement was met, it was necessary to implement several Salzella extensions. To show that Salzella interpreter can indeed be used as internal engine of music generating tools, I decided to create a prototype of an algorithm for converting musical pieces into Salzella programs. By encapsulating the essence of an already existing musical piece by means of a Salzella program, this tool should be able to generate musical contents similar to the original piece.

1.3 Platform overview

Figure 1.4 illustrates the so far implied structural relations of Salzella platform components. Both the development environment and conversion tool use Salzella interpreter. Salzella interpreter implements Sazlella specification and uses external plugins. Conversion tool can be executed directly from the development environment.

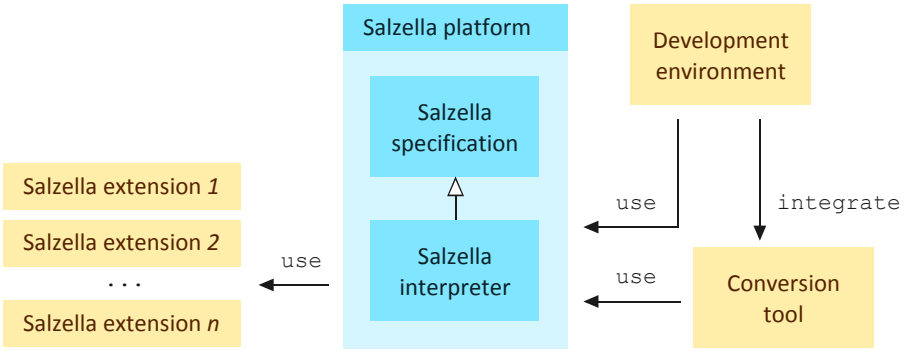


Figure 1.4: Overview of the so far implied structural relations

1.4 Related works

There are dozens of music generating programs and algorithms. Some solutions are robust and aim to generate whole compositions [1], some are designed to perform specific tasks such

as generating jazz guitar solos [2]. To some extent, all of them leverage existing principles described in music theory. This often includes outlining the bounds in which a certain level of randomness is introduced. But there are also more exotic approaches like deriving music from cellular automaton patterns [3]. Salzella is designed in a way which makes it possible to integrate any existing music generating algorithm in a form of a plugin. In other words, Salzella doesn't compete with other solutions, it is capable of absorbing them. No attempts to make such framework seem to have taken place in the past. In order to make the process of absorption of these algorithms as painless as possible, possibilities of adopting some standardized computer representation of musical content were researched.

1.4.1 Open source music libraries

Open source libraries such as [4] or [5] provide their own data types not only for elementary entities such as notes, but also for more complex structures like chords and scales. These libraries could be useful when creating actual music generating algorithms. But there would be little to no benefit from building Salzella on top of any of them. As mentioned above, Salzella does not provide explicit support for high level structures such as chords, scales and keys. This means that the only relevant content of open source music libraries would include several data types and enumerations related to representing a note. Developing these structures from scratch is not a challenging task and the added benefit of being in control over the APIs of these structures is indisputable.

1.4.2 MIDI specification

MIDI specification [6] is primarily a communication protocol designed to deal with real time communication between synthesizers. Several years after its original release in 1983, storage of this communication by means of time stamping individual messages was standardized. Using MIDI specification as basis for structural representation of music seems to be most beneficial. Its popularity and platform independence increases the likelihood of compatibility with musical content representations used by the already existing music generating algorithms. Furthermore, possibility of storing the generated musical contents in MIDI compatible format would allow Salzella users to import the generated snippets of music into external software such as music notation editors or digital audio workstations.

Note For more detailed description of the MIDI specification, see the article I wrote when working on semestral project from Y14TED. This article can be found on the enclosed CD, see Appendix B.

1.5 Early attempts

Before diving into specification of the language, let me shortly explain the motivation behind the topic of this thesis. When working on my bachelor thesis, I created a music generating tool called M-Architect. M-Architect was a full featured music editor with additional capabilities such as generating harmonies and bass lines under existing melodies. Shortly after submitting the bachelor thesis, a series of usability tests was performed. In attempt to fix the

discovered usability issues, various modifications were made. These included adding support for copy/paste functionality, allowing the user to store/load settings of music generating algorithms or implementing more responsive editor controls. Perhaps the most important improvement, however, was introduction of concept of scenarios. Scenarios allowed the user to predefine complex sequences of actions and store them as macros. The existence of scenarios allowed the application's functionality to be wrapped up and presented in a form of a list of predefined macros and a single generate button. Figure 1.5 illustrates the process of converting low fidelity paper prototypes to a fully functional application.

Note For full documentation on designing the user interface of M-Architect, see A4M39NUR semestral project documentation 'Assistive tool for music composers'. For more details about the related qualitative research, see A4M39PUR semestral project documentation 'Habits of music composers'. Both documents can be found on the enclosed CD, see Appendix B.

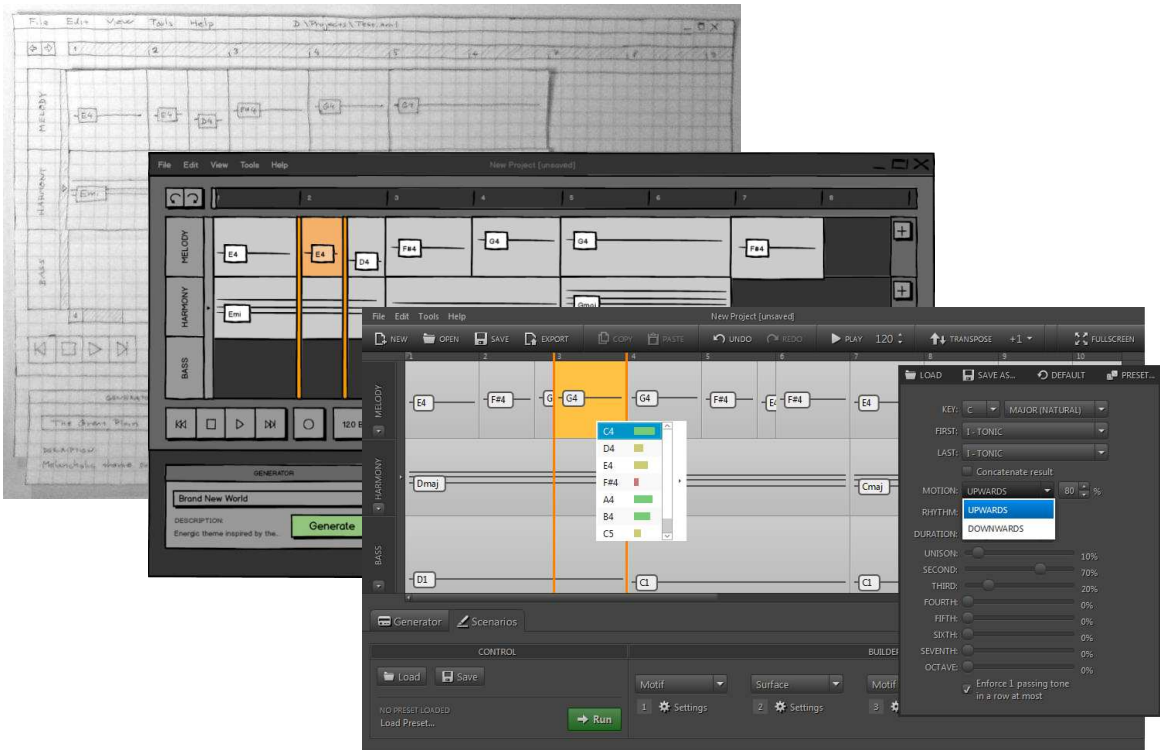
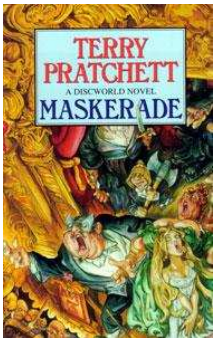


Figure 1.5: Converting low fidelity prototypes to a fully functional application

The aforesaid modifications solved the most critical usability issue: the overall complexity. Unfortunately, where one problem disappeared, another arose. Once advantageous definition of music as triplet <melody, harmony, bass> was suddenly a cause of unnecessary limitations in terms of variety of outputted music. Instead of redesigning the very basis of this particular music generating tool, I decided to create a platform upon which music generating tools could be built. For now, M-Architect is in a dormant state. But in the future, it could be made less restrictive and rebuilt to use Salzella internally.

1.6 Name of the language

The language is named after Mr. Sazella, a character from Terry Pratchett's novel Maskerade [7]. As a reference to the book, an arbitrary number of exclamation marks can be inserted anywhere in a Salzella program without affecting its functionality, see Figure 1.6.



´What sort of person,´ said Salzella patiently, ´sits down and writes a maniacal laugh? And all those exclamation marks, you notice? Five? A sure sign of someone who wears his underpants on his head. Opera can do that to a man.´

Terry Pratchett
Maskerade

Figure 1.6: A quote from Terry Pratchett's book Maskerade

Chapter 2

Specification

2.1 Program structure

Salzella program is a list of key-value pairs. Each key-value pair is terminated by a semicolon and the key/value parts are separated by a colon. Key-value pairs are divided into segments. Segments are separated by three consecutive dashes. A snippet shown in Figure 2.1 consists of three segments. First two contain three key-value pairs, third one contains two key-value pairs. Note that Salzella is case sensitive and whitespace characters outside the value part of key-value pairs have no significance. When a program is parsed, all whitespace characters which are not inside the value part of a key-value pair are removed. The contents of the value part of all key-value pairs are trimmed and all sequences of whitespace characters are replaced with a single space character.

Convention Because the value parts of key-value pairs will be typically parsed by external algorithms, format of the actual string representations of values is not predetermined by Salzella specification. However, for sake of uniformity, the following convention was established: Should the contents of the value part of some key-value pair be complex enough to require usage of delimiters, white space character should be used as primary delimiter and comma should be used as secondary delimiter.

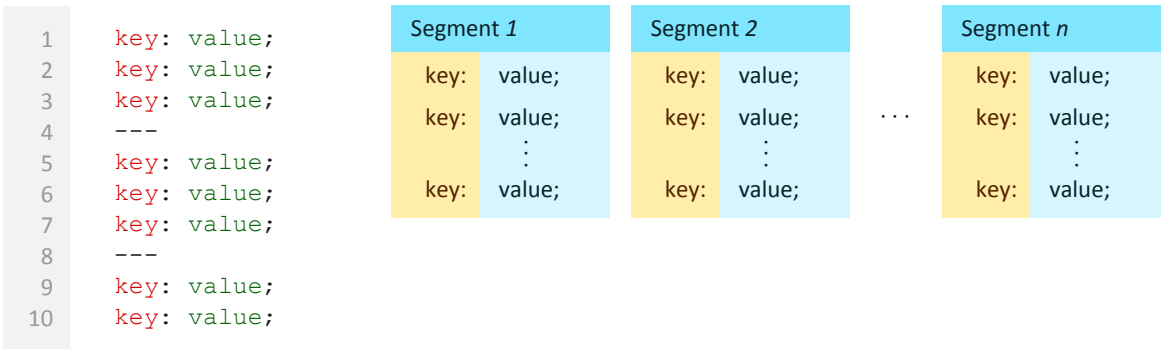


Figure 2.1: Salzella program is a list of key-value pairs

2.2 Pitch literal

Pitch literals represent frequencies of tones. Table 2.1 shows a complete list of pitch literals. With the total of 127 pitch literals, Salzella programs can address pitches ranging from C0 to G10. As mentioned earlier, Salzella is case sensitive. The names of tones must be written in lower case.

Table 2.1: A complete list of pitch literals

Oct.	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	c0	c#0	d0	d#0	e0	f0	f#0	g0	g#0	a0	a#0	b0
1	c1	c#1	d1	d#1	e1	f1	f#1	g1	g#1	a1	a#1	b1
2	c2	c#2	d2	d#2	e2	f2	f#2	g2	g#2	a2	a#2	b2
3	c3	c#3	d3	d#3	e3	f3	f#3	g3	g#3	a3	a#3	b3
4	c4	c#4	d4	d#4	e4	f4	f#4	g4	g#4	a4	a#4	b4
5	c5	c#5	d5	d#5	e5	f5	f#5	g5	g#5	a5	a#5	b5
6	c6	c#6	d6	d#6	e6	f6	f#6	g6	g#6	a6	a#6	b6
7	c7	c#7	d7	d#7	e7	f7	f#7	g7	g#7	a7	a#7	b7
8	c8	c#8	d8	d#8	e8	f8	f#8	g8	g#8	a8	a#8	b8
9	c9	c#9	d9	d#9	e9	f9	f#9	g9	g#9	a9	a#9	b9
10	c10	c#10	d10	d#10	e10	f10	f#10	g10				

2.3 Duration literal

Duration literals represent relative durations of tones. Each duration literal consists of two parts: duration base and duration multiplicity. Duration bases represent standard duration symbols used in musical scores. Table 2.2 shows a complete list of duration bases.

Table 2.2: A complete list of duration bases

Type	Regular	Triplet	Dotted
whole	1	1t	1d
half	2	2t	2d
quarter	4	4t	4d
eighth	8	8t	8d
sixteenth	16	16t	16d
thirty-second	32	32t	32d
sixty-fourth	64	64t	64d

To create a duration literal, duration base must be prefixed with a duration multiplicity. Duration multiplicity is an integer number from interval [0..255] followed by symbol 'x'. The actual duration represented by a duration literal can be computed by multiplying the duration defined by duration base by duration multiplicity. Table 2.3 shows a few examples of duration literals.

Table 2.3: A few examples of duration literals

Literal	Meaning
2x1	two whole notes
1x2	one half note
4x8	four eighth notes
1x64	one sixty-fourth note
1x2t	one half triplet
3x1d	three dotted whole notes

2.4 Signature matrix

When creating a Salzella program, one must always define the overall rhythmic structure to which the generated musical piece will be bound to adhere. This can be achieved by creating a signature matrix. Each line of signature matrix consists of two integer numbers followed by a duration literal. The second integer and the duration literal denote the top and bottom part of a time signature, respectively. The first number denotes how many times should be this time signature applied in the sequence of measures. For example, the signature matrix shown in Figure 2.2 says the following: There are 3 measures of 4/4 time signature, then a single measure of 7/8, and then 4 measures of 4/4 time signature.

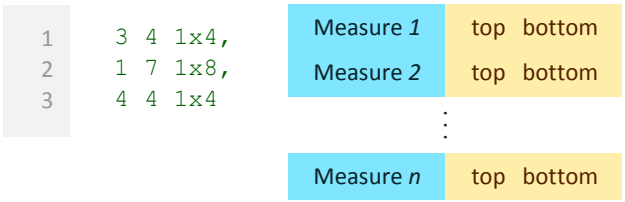


Figure 2.2: Example of a signature matrix

2.5 Surface matrix

Another type of matrix Salzella specification defines is surface matrix. This type of matrix allows the user to specify the tonal surface upon which the musical piece will be built. Surface matrix divides the musical piece into one or more snippets. Each line of this type

of matrix consists of a duration literal followed by twelve integer numbers from the interval [0...255]. The duration literal defines how long the snippet is. The twelve integer numbers label individual pitch classes. There are 12 pitch classes and their implicit order goes as follows: C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, B. The actual meaning of these integer labels may vary among individual Salzella programs and is in no way predetermined by Salzella specification.

Convention In all example programs created in this thesis, surface matrix is used to determine which scale/chord should be used within individual snippets. To achieve this, the following convention was established: Pitch classes labeled with 0 are not contained in the scale. Pitch classes labeled with 1 are contained in the scale. Pitch classes labeled with 2 are contained in the scale and are also contained in the chord. Pitch class labeled with 3 is a root of the chord and is also contained in the scale. Surface matrix shown in Figure 2.3 divides the musical piece into four snippets. All of them are one whole note long and all of them are built on top of the e minor scale. The chord progression defined by this particular surface matrix is: Emi, Dmaj, Cmaj, Bmaj.

```
1  1x1 1 0 1 0 3 0 1 2 0 1 0 2,
2  1x1 1 0 3 0 1 0 2 1 0 2 0 1,
3  1x1 3 0 1 0 2 0 1 2 0 1 0 1,
4  1x1 1 0 0 2 1 0 2 1 0 1 0 3
```

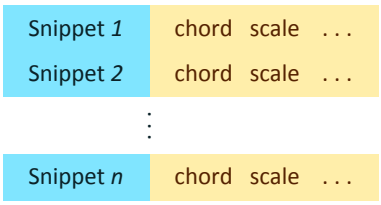


Figure 2.3: Example of a surface matrix

2.6 Track matrix

The last type of matrix Salzella defines is track matrix. This matrix defines a list of tracks of which the musical piece will be made. Each track is described in a form of a quintuplet defining a unique identifier of the track, instrument, volume and mute and solo flags. Instrument and volume are integers from the interval [0..127], solo and mute are boolean values. Instrument values are interpreted as defined by the MIDI specification [6]. Note that if value 128 is specified as instrument, the affected track will be considered to be percussive. Track matrix

```
1  vocal  73 127 false false,
2  piano   0 127 false false,
3  drums 128 127 false false
```

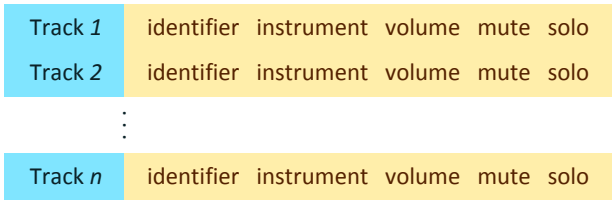


Figure 2.4: Example of a track matrix

shown in Figure 2.4 defines three tracks. All of them are set to be as loud as possible and their solo/mute flags are switched off. The first track uses instrument 73 (flute) and its identifier is ‘vocal‘. Second track uses instrument 0 (piano) and its identifier is ‘piano‘. The last track uses instrument 128 (percussion) and its identifier is ‘drums‘.

2.7 Header segment

Every Salzella program must begin with a header segment. Within this segment, the general structural properties to which the generated musical piece should adhere must be defined. Signature, surface and track matrices must be specified within this segment. Also, an integer number from the interval [1..999] must be provided to determine tempo of the resulting musical piece. Table 2.4 shows a complete list of properties of header segment. Figure 2.5 shows an example of a header segment.

Table 2.4: Header segment has four mandatory properties

Property	Expected value	Example
tempo:	number from [1, 999]	`120`
signature:	signature matrix	`4 4 1x4`
surface:	surface matrix	`1x1 1 0 1 0 3 0 1 2 0 1 0 2`
tracks:	track matrix	`piano 0 127 false false`

```
1     tempo: 120;
2   surface: 1x1 1 0 1 0 3 0 1 2 0 1 0 2,
3            2x1 1 0 3 0 1 0 2 1 0 2 0 1,
4            1x1 0 2 1 0 2 0 1 1 0 3 0 1,
5            1x1 2 0 1 0 2 0 1 1 0 3 0 1,
6            2x1 1 0 1 0 3 0 1 2 0 1 0 2,
7            3x4 1 0 3 0 1 0 2 1 0 2 0 1,
8            1x1 1 0 1 0 3 0 1 2 0 1 0 2,
9            1x1 1 0 3 0 1 0 2 1 0 2 0 1,
10           1x1 0 2 1 0 2 0 1 1 0 3 0 1,
11           1x1 2 0 1 0 2 0 1 1 0 3 0 1,
12           7x8 1 0 1 0 3 0 1 2 0 1 0 2,
13           7x8 1 0 3 0 1 0 2 1 0 2 0 1;
14 signature: 7 4 1x4,
15           1 3 1x4,
16           4 4 1x4
17           2 7 1x8;
18    tracks: vocal  73 127 false false,
19           guitar 25 127 false false,
20           drums 128 127 false false;
```

Figure 2.5: Example of a header segment

2.8 Filter segment

Salzella programs can contain zero or more filter segments. Each filter segment has exactly four mandatory properties. The source property decides which plugin should be used when execution of this filter is requested. The input property contains a list of track identifiers determining which tracks will be passed at the input of the filter. Start and duration properties contain duration literals defining a time window at which the filter should be applied. Table 2.5 shows a complete list of mandatory properties of filter segment. Figure 2.6 shows a few examples of filter segments.

Custom properties Note that each filter may also define a set of custom properties. Since parsing of custom properties is handled by filters themselves, there are almost no limitations as to what can be passed to the value part of a custom key-value pair. However, creators of filters are encouraged to use standard pitch and duration literals whenever possible and adhere to space/comma character delimiter convention.

Restrictions The following characters/strings have special meaning in Salzella and therefore can not appear in either part of a key-value pair: `;`, `:`, `---`. Also, keep in mind that requirement stated in 1.6 renders exclamation marks effectively useless. Finally, in case multiple key-value pairs with the same key are declared within one segment, the last occurring pair counts.

Table 2.5: Each filter segment has four mandatory properties

Property	Expected value	Example
source:	plugin indentifier	`com.example.CustomPlugin`
input:	one or more track identifiers	`vocal guitar`
start:	duration literal	`0x1`
duration:	duration literal	`1x1`

```
1     source: cz.stepanvolf.salzella.plugin.GeneralMelody;
2      input: vocal;
3      start: 2x1;
4   duration: 2x1;
```

```
1     source: cz.stepanvolf.salzella.plugin.SimpleDrums;
2      input: drums;
3      start: 0x1;
4   duration: 4x1;
```

Figure 2.6: A few examples of filter segments

2.9 Extensions

Sazella program will typically contain more than one filter. When Salzella program is executed, all filters will be applied one by one in order in which they were declared. Note that all Salzella filters are distributed in form of plugins. To illustrate structural capabilities of Salzella, three simple filters were created as part of this thesis. In the rest of this chapter, these will be described in greater detail.

2.9.1 SimpleDrums

SimpleDrums filter generates simple kick/snare/hihat drum beats. Table 2.6 shows a complete list of custom properties of this filter. Hihat property defines rate at which hihat should be played. Crash property defines probability of substituting closed hihat hit with open hihat hit. Kick property defines approximate kick usage in relation to the hihat rhythm grid. Snare property defines list of duration literals determining exact time marks at which snare should be played. Loop property defines duration of loop. Variety property defines degree of syncopation. Figure 2.7 shows an example usage of SimpleDrums filter.

Table 2.6: List of SimpleDrums properties

Property	Expected value	Example
loop:	single duration literal	`1x1`
hihat:	single duration literal	`1x8`
snare:	one or more duration literals	`1x4 3x4`
kick:	number from [0.0, 1.0]	`0.3`
crash:	number from [0.0, 1.0]	`0.1`
variety:	number from [0.0, 1.0]	`0.2`

```
1     source: cz.stepanvolf.salzella.plugin.SimpleDrums;
2      input: inputTrack;
3      start: 0x1;
4   duration: 1x1;
5       loop: 1x1;
6      hihat: 1x8;
7      snare: 1x4 3x4;
8       kick: 0.3;
9      crash: 0.2;
10   variety: 0.1;
```

Figure 2.7: Example usage of SimpleDrums filter

2.9.2 GeneralMelody

GeneralMelody filter generates random melodies. Table 2.7 shows a complete list of custom properties of this filter. Low/high properties define the lowest and highest allowed pitch. Min/max properties define size of the smallest and biggest allowed step within the scale. Chord property defines probability of chord consonance enforcement. If a generated tone is not contained in the underlaying chord, its pitch will be changed to the closest acceptable pitch based on the aforesaid probability. Grid property uses a single duration literal to create evenly distributed time marks at which tones in this melody are allowed to start/end. Relax property defines a probability of prolonging a note by postponing its end to match start of the next note in melody. Finally, structure property determines the overall structure of the melody in terms of note density and directional curve. Figure 2.8 shows an example usage of GeneralMelody filter.

Table 2.7: List of GeneralMelody properties

Property	Expected value	Example
grid:	single duration literal	`1x1`
chord:	number from [0.0, 1.0]	`0.8`
relax:	number from [0.0, 1.0]	`0.5`
min:	number from [0, 127]	`0`
max:	number from [0, 127]	`3`
low:	single pitch literal	`c4`
high:	single pitch literal	`f#5`
structure:	one or more numbers from [0.0, 1.0],	`0.3 0.8 0.9,`
	one or more elements from {UP, DOWN, STEADY}	`UP DOWN`

```
1     source: cz.stepanvolf.salzella.plugin.GeneralMelody;
2      input: inputTrack;
3      start: 0x1;
4   duration: 1x1;
5       grid: 1x8;
6  structure: 0.7 0.8 0.5,
7             UP STEADY DOWN UP;
8      chord: 0.5;
9      relax: 0.8;
10       min: 0;
11       max: 1;
12       low: e4;
13      high: e5;
```

Figure 2.8: Example usage of GeneralMelody filter

2.9.3 RhythmGuitar

RhythmGuitar filter generates simple guitar complement. Table 2.8 shows a complete list of custom properties of this filter. Mode property determines the type of complement. Grid property defines rate at which should the virtual guitar player move his hand up and down. Figure 2.9 shows an example usage of RhythmGuitar filter.

Table 2.8: List of RhythmGuitar properties

Property	Expected value	Example
mode:	one element from {RHYTHM, STRUM}	`RHYTHM`
grid:	single duration literal	`1x8`

```
1     source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
2      input: inputTrack drumTrack;
3      start: 0x1;
4   duration: 1x1;
5       mode: RHYTHM;
6       grid: 1x8;
```

Figure 2.9: Example usage of RhythmGuitar filter

Chapter 3

Realization

3.1 Interpreter

3.1.1 Architecture overview

Figure 3.1 shows an overview of Salzella platform architecture. This is how individual components work together: Program execution begins by invoking the parser which creates an instance of Salzella object model. Once the parsing is complete and the Salzella object model created, execution of the program is a fairly straight forward process. The execution begins by creating a lightweight object representation of a MIDI sequence. The overall structure of this sequence is derived from the track matrix. After the sequence is created, filters are applied to it one by one. Given the external nature of these filters, the algorithm responsible for execution is very careful to make defensive copies of the original sequence and checks for unexpected runtime exceptions. In case an error occurs during execution of the program, appropriate exception is thrown. Salzella defines its own exceptions which hold information not only about the kind of error, but about what segment of program caused it.

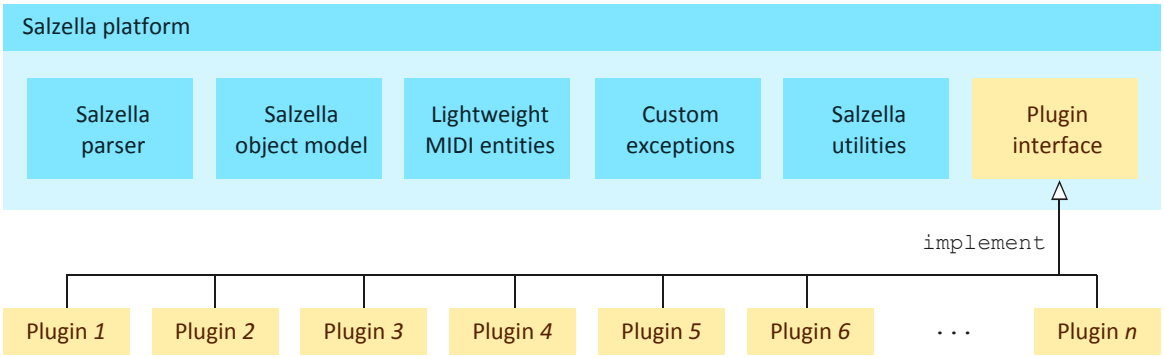


Figure 3.1: An overview of Salzella platform architecture

3.1.2 Salzella object model

Prior to execution, each program is converted into Salzella object model. The structure of this model closely copies the program structure presented in the previous chapter. Diagram shown in Figure 3.2 should therefore not require a more detailed description. Note however, that some of the classes provide convenience methods which can be used when implementing the actual music generating algorithms. For example, the `Segment` class provides a method which can determine whether the given beat is off-beat, on-beat or down-beat.

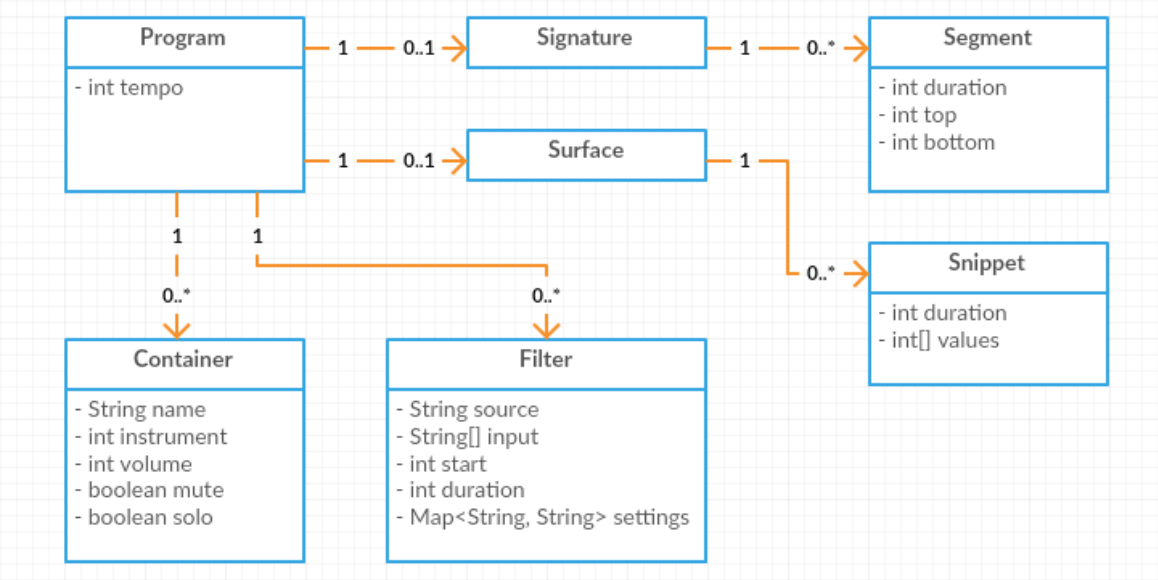


Figure 3.2: Salzella object model

3.1.3 Lightweight MIDI entities

Salzella provides its own lightweight implementation of elementary MIDI entities, see Figure 3.3. Convenience methods for converting the Salzella lightweight MIDI entities to standard MIDI format and vice versa can be found in the `Converter` class.

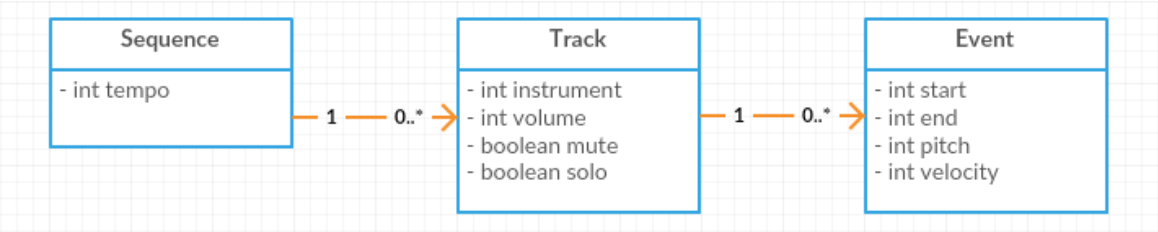


Figure 3.3: Salzella implementation of elementary MIDI entities

3.1.4 Implementation

Salzella is implemented using the Java programming language. Both the source code and javadoc documentation can be found on the enclosed CD, see Appendix B. Figure 3.4 shows the package overview as can be seen on the javadoc landing page. Source codes were version controlled using Git. Figure 3.5 shows last few commits.

Packages

Package	Description
cz.stepanvolf.salzella	Public APIs for invoking the interpreter.
cz.stepanvolf.salzella.entity	Lightweight implementation of elementary MIDI entities.
cz.stepanvolf.salzella.exception	Custom exceptions.
cz.stepanvolf.salzella.parser	Implementation of the parser and Salzella object model.
cz.stepanvolf.salzella.plugin	Interface for plugins.
cz.stepanvolf.salzella.util	A helper library for filter creators.

Figure 3.4: Salzella interpreter package overview

Add support for invoking the interprete	Stepan <stepan@stepanvolf.cz>	2015-12-23 11:23:29
Add javadoc comments to all public A	Stepan <stepan@stepanvolf.cz>	2015-12-19 19:11:29
Create javadoc for packages	Stepan <stepan@stepanvolf.cz>	2015-12-19 12:33:09
Fix import for boundary enumeration	Stepan <stepan@stepanvolf.cz>	2015-12-19 12:20:04
Rename boundary enumeration	Stepan <stepan@stepanvolf.cz>	2015-12-15 11:55:49
Make snippet API more user friendly	Stepan <stepan@stepanvolf.cz>	2015-12-14 12:08:23
Create method for generating randon	Stepan <stepan@stepanvolf.cz>	2015-12-12 14:28:00
Implement beat classification functior	Stepan <stepan@stepanvolf.cz>	2015-12-08 15:00:34
Update surface and signature APIs	Stepan <stepan@stepanvolf.cz>	2015-12-08 13:15:52
Implement method for pitch class retr	Stepan <stepan@stepanvolf.cz>	2015-12-08 12:40:36
Add snippet/segment retrieval metho	Stepan <stepan@stepanvolf.cz>	2015-12-08 12:21:43
Fix sculptor bug	Stepan <stepan@stepanvolf.cz>	2015-12-08 11:18:20
Add support for sequence sculpting	Stepan <stepan@stepanvolf.cz>	2015-12-08 11:03:30
Allow exclamation marks to be inserte	Stepan <stepan@stepanvolf.cz>	2015-12-06 15:14:43
Improve exception handling mechanis	Stepan <stepan@stepanvolf.cz>	2015-12-06 12:21:27
Detect drum tracks when converting M	Stepan <stepan@stepanvolf.cz>	2015-12-06 11:33:40
Change min/max boundaries for num	Stepan <stepan@stepanvolf.cz>	2015-12-06 10:34:15
Fix MIDI channel assignment bug	Stepan <stepan@stepanvolf.cz>	2015-12-05 19:39:45
Use proper resolution when convertin	Stepan <stepan@stepanvolf.cz>	2015-12-05 19:00:55
Add support for MIDI->Salzella conver	Stepan <stepan@stepanvolf.cz>	2015-12-05 15:48:27
Add support for Salzella->MIDI conver	Stepan <stepan@stepanvolf.cz>	2015-12-04 11:13:19

Figure 3.5: Last few Git commits

3.1.5 Extenision APIs

As mentioned earlier, Salzella filters are distributed in form of plugins. Creation of a custom plugin is a simple matter of subclassing the `cz.stepanvolf.salzella.plugin.Plugin` class and implementing the abstract `run()` method. Figure 3.6 shows an example source code for a very simple plugin which adds a note at the given position. Duration and velocity of this note will be passed to the plugin as part of custom settings. The pitch of the added note will be derived from root note of surface snippet in which the start property lies.

```
1  package cz.stepanvolf.salzella.plugin;
2
3  import cz.stepanvolf.salzella.entity.Event;
4  import cz.stepanvolf.salzella.entity.Sequence;
5  import cz.stepanvolf.salzella.parser.Parser;
6  import cz.stepanvolf.salzella.parser.Signature;
7  import cz.stepanvolf.salzella.parser.Surface;
8  import java.util.Map;
9
10 public final class SamplePlugin extends Plugin {
11
12   private static final String NAME = "Sample plugin";
13   private static final String AUTHOR = "Stepan Volf";
14   private static final String DESCRIPTION = "Use this plugin to...";
15   private static final String TEMPLATE = "source: cz.stepanvo...";
16
17   public SamplePlugin() {
18      super(NAME, AUTHOR, DESCRIPTION, TEMPLATE);
19   }
20
21   @Override
22   public Sequence run(Sequence sequence, Surface surface,
23                       Signature signature, int[] input, int start,
24                       int duration, Map<String, String> settings) {
25
26      // Parse settings
27      int pitch = surface.getSnippet(start).relevantIndexes(3)[0] + 60;
28      int velocity = Integer.parseInt(settings.get("velocity"));
29      int length = Parser.parseDuration(settings.get("length"));
30      int end = start + Math.min(duration, length);
31
32      // Add new event
33      Event event = new Event(start, end, pitch, velocity);
34      sequence.getTracks().get(input[0]).getEvents().add(event);
35      return sequence;
36   }
37 }
```

Figure 3.6: Source code of a very simple plugin

Figure 3.7 shows a Salzella program which uses the sample plugin. In this particular example, the filter will ensure that eighth note E5 will be played at the beginning of first measure and half note D5 will be played at the beginning of second measure. Both notes will be played by the guitar.

```
    tempo: 120;
  surface: 1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 3 0 1 0 2 1 0 2 0 1,
           1x1 0 2 1 0 2 0 1 1 0 3 0 1,
           1x1 2 0 1 0 2 0 1 1 0 3 0 1,
           1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 3 0 1 0 2 1 0 2 0 1,
           1x1 0 2 1 0 2 0 1 1 0 3 0 1,
           1x1 2 0 1 0 2 0 1 1 0 3 0 1,
           1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 3 0 1 0 2 1 0 2 0 1,
           1x1 0 2 1 0 2 0 1 1 0 3 0 1,
           1x1 2 0 1 0 2 0 1 1 0 3 0 1;
signature: 12 4 1x4;
   tracks: guitar 24 127 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.SamplePlugin;
    input: guitar;
    start: 0x1;
 duration: 1x8;
 velocity: 127;
   length: 1x8;
           ---
   source: cz.stepanvolf.salzella.plugin.SamplePlugin;
    input: guitar;
    start: 1x1;
 duration: 1x2;
 velocity: 127;
   length: 1x2;
```

Figure 3.7: Using the sample plugin within a Salzella program

3.2 Development environment

3.2.1 User guide

A development environment for creating Salzella programs was created as part of this thesis. The user interface of Salzella development environment consists of a source code editor, playback controls and three Salzella control buttons. These buttons allow the user to run a Salzella program, load an existing Salzella program from a built-in database and generate a program based on contents of a MIDI file. Figures 3.8 - 3.12 illustrate the basic use case scenarios.

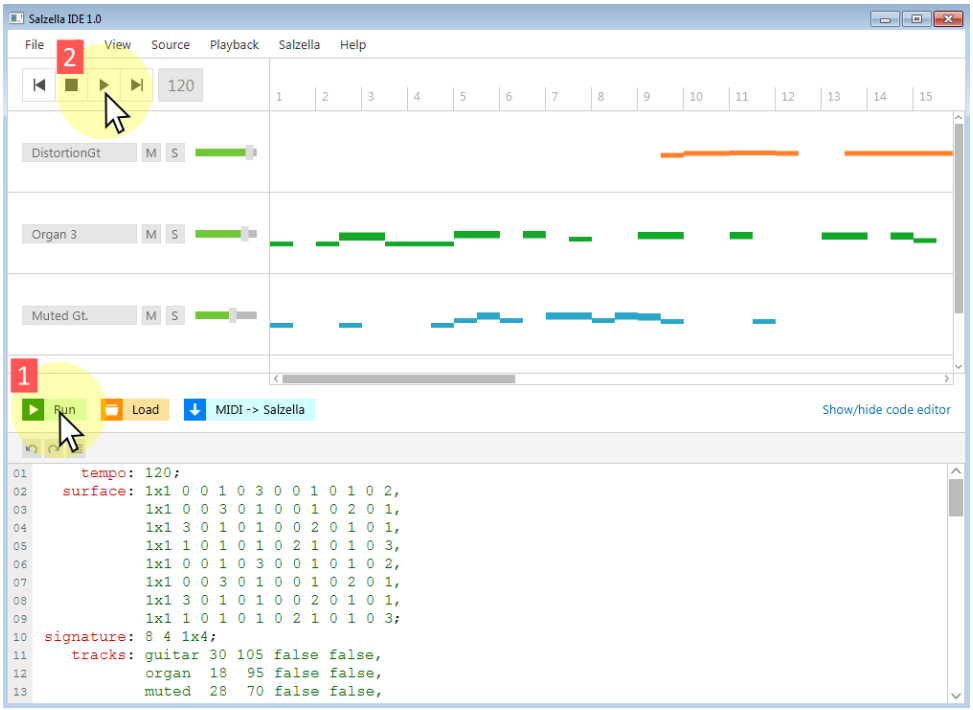


Figure 3.8: Use case 1 - Run program and playback the result

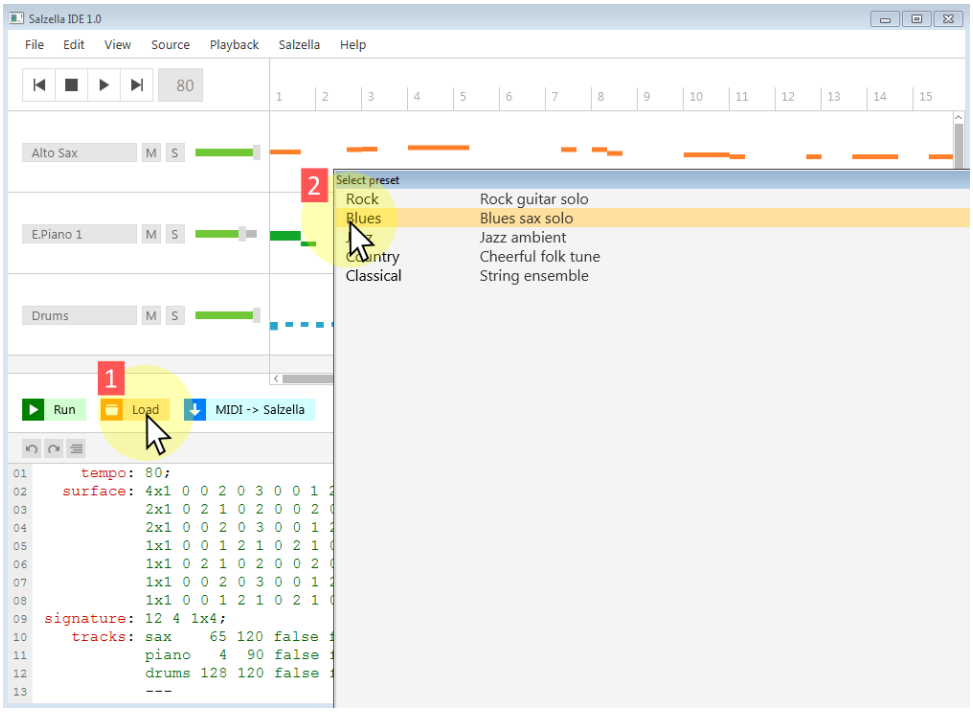


Figure 3.9: Use case 2 - Load program from the built-in database

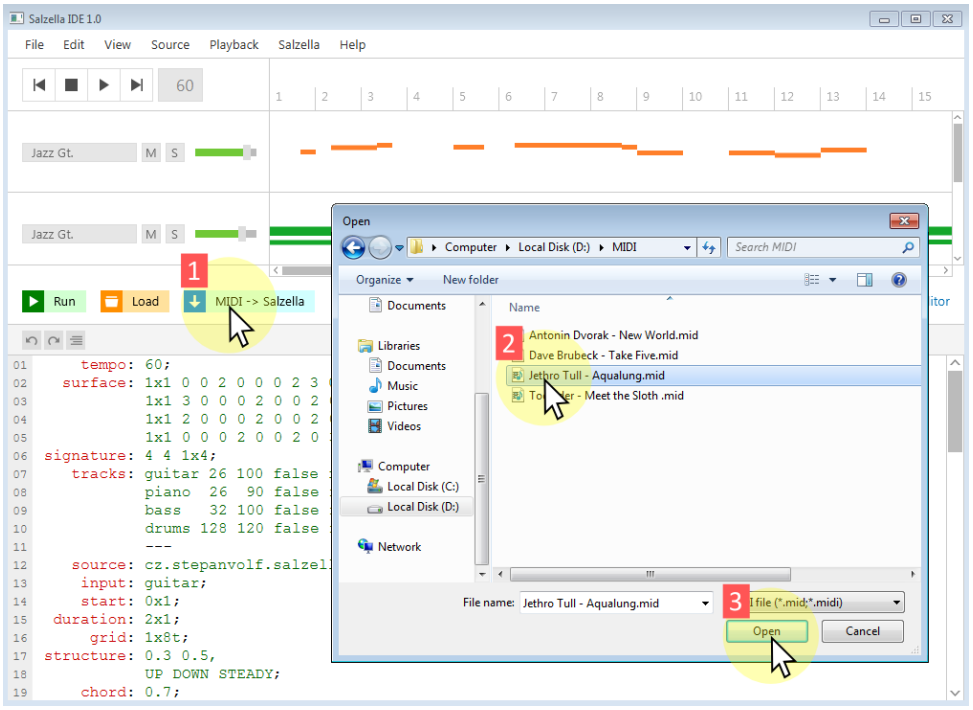


Figure 3.10: Use case 3 - Generate program based on contents of a MIDI file

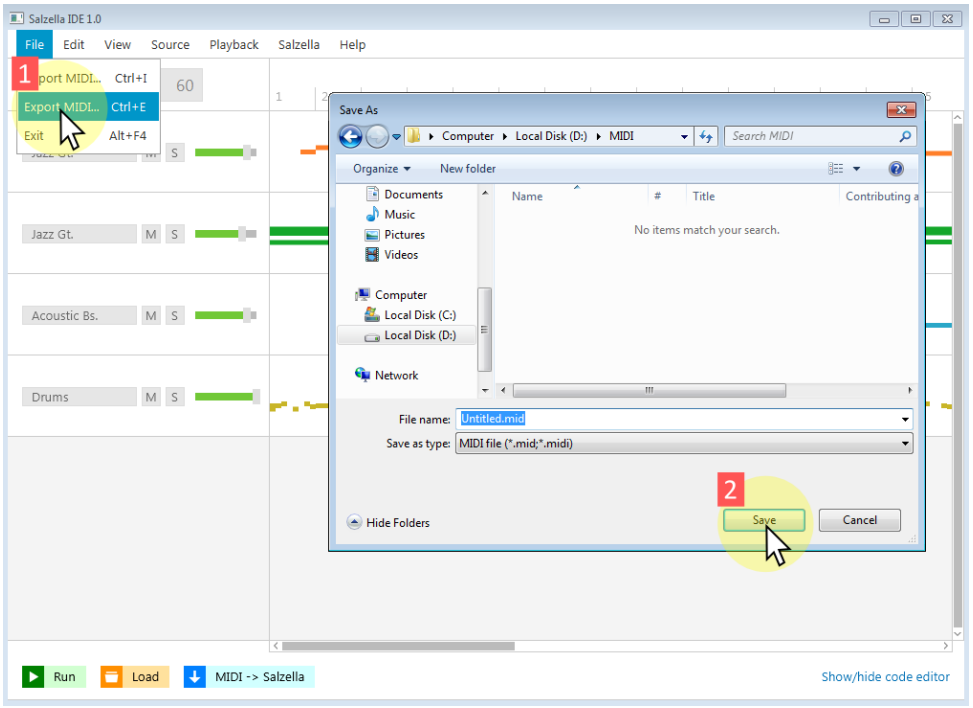


Figure 3.11: Use case 4 - Export the generated snippet of music to a MIDI file

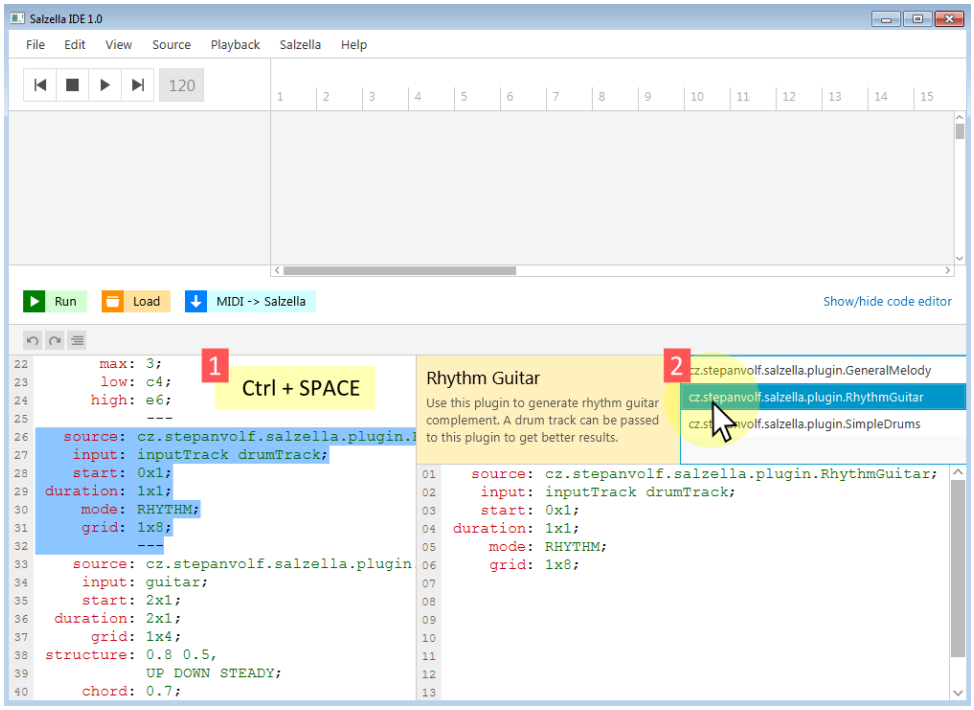


Figure 3.12: Use case 5 - Use code completion

3.2.2 Implementation notes

Salzella development environment is built on top of the MVC architecture. The communication between layers is handled by standard design patterns. Undoable changes in Model are performed using the command pattern and Model-View synchronization is achieved by means of the observer pattern. Figure 3.13 shows the package overview as can be seen on the javadoc landing page. Executable version of the development environment can be found on the enclosed CD, see Appendix B.

Packages

Package	Description
cz.stepanvolf.salzie	Initialization related classes.
cz.stepanvolf.salzie.commands	Both the command interface and concrete commands.
cz.stepanvolf.salzie.controller	Implementation of controller.
cz.stepanvolf.salzie.model	Classes responsible for application state representation.
cz.stepanvolf.salzie.observers	Interfaces for model observers.
cz.stepanvolf.salzie.view	Implementation of user interface.

Figure 3.13: Salzella development environment package overview

Chapter 4

Experiments

4.1 Genre independence

To illustrate that Sazella is capable of generating music of various music genres, five Salzella programs addressing the following musical styles were created: Rock, Jazz, Blues, Folk and Classical music. Snippets of these programs can be found in Appendix A. Full versions of these programs are stored in the built-in database of Salzella development environment. To test whether the aforesaid programs work as intended, five experienced musicians were asked to listen to 10 snippets of music generated by these programs (2 snippets per program). Upon listening to a snippet of music, each participant was asked to classify this snippet in terms of music genre. Each participant was presented with a different, newly generated, randomly ordered sequence of snippets. Figure 4.1 shows recognition success rates for all tested music genres. Note that the participants were not presented with a list of options and only exact matches were accepted as correct answers.

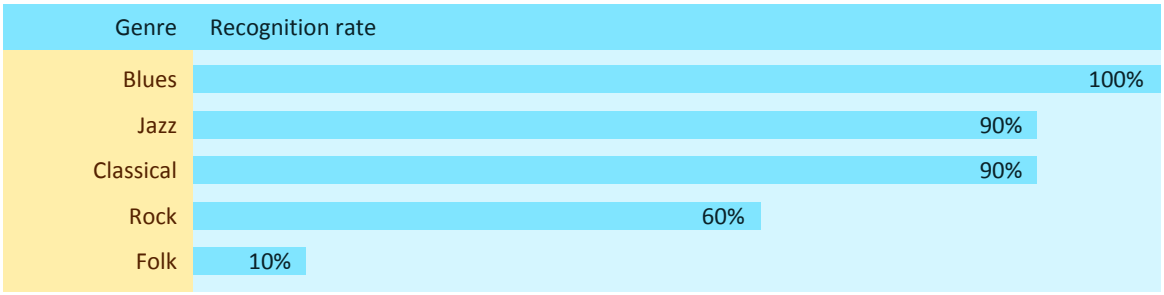


Figure 4.1: Recognition success rates for the tested music genres

4.2 Conversion tool

4.2.1 Problem statement

To illustrate that Salzella interpreter can indeed be used as internal engine of music generating tools, a prototype of an algorithm for converting musical pieces into Salzella programs

was created as part of this thesis. Figure 4.2 illustrates the way this algorithm is meant to be used. The user supplies a standard MIDI file containing an arbitrary musical piece. Based on the contents of this file, a Salzella program is generated. Since this program encapsulates the essence of the original musical piece, passing this program to Salzella interpreter results in producing musical piece which is bound to be musically similar to the contents of the file at the input.

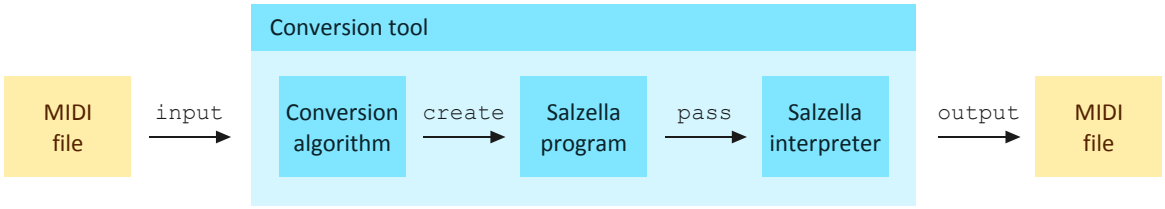


Figure 4.2: Converting MIDI files to Salzella programs

4.2.2 Limitations

Since the algorithm presented in the rest of this chapter is a prototype, it has several limitations. For example, it assumes 3/4 signature throughout the whole song. Or, to put it more precisely, it assumes that changes in harmony occur only at intervals of three quarter notes. It also assumes that second track of the original MIDI sequence contains a melody. And even though it takes into account all input tracks, it always produces only two tracks. First track contains a melody similar to the melody from the second input track. Second track contains a sequence of chords harmonizing the melody. Also, the conversion algorithm only supports natural form of major/minor scale and major/minor triad based harmonies.

4.2.3 Generating header segment

As illustrated by Figure 4.3, deriving contents of header segment from an already existing musical piece is simple. Tempo will always be the same as the tempo of the original MIDI sequence. Because the conversion tool will always limit the output to contain exactly two tracks, generation of track matrix is trivial. And because 3/4 signature is assumed throughout the whole song, creating signature matrix is a simple matter of counting how long the original sequence is. Creating a surface matrix is a bit harder, see the note below.

```
1      tempo: <same as tempo of the input track>;
2    surface: <provided by the harmony analysis algorithm>;
3  signature: <derived from length of the input sequence> 3 1x4;
4     tracks: melody  0 127 false false,
5             harmony 0 127 false false;
```

Figure 4.3: Generating header segment

Note In order to create a surface matrix, harmony of the original piece must be determined. To solve this task, I used a melody harmonization algorithm I came up with when working on A4M35KO semestral project. For more detailed description of this algorithm, see A4M35KO semestral project documentation 'Melody harmonization'. This document can be found on the enclosed CD, see Appendix B.

4.2.4 Applying filters

When creation of the header segment is complete, RhythmGuitar filter is used to generate a simple guitar complement. As shown in Figure 4.4, duration property is the only property which may vary between individual executions of the conversion algorithm. Value of this property will always match duration of the whole musical piece.

```
    source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
     input: harmony;
     start: 0x1;
  duration: <derived from length of the input sequence>;
      mode: STRUM;
      grid: 3x4;
```

Figure 4.4: Applying the RhythmGuitar filter

Contents of the melody track are generated by repeated application of GeneralMelody filter. The size of time window assigned to each GeneralMelody filter application is two measures and the filter itself is applied as many times as necessary to cover the whole input track. As shown in Figure 4.5, all properties are statistically derived from the relevant portion of the input track. Note that making the time window smaller or larger results in increase or decrease in overall similarity.

```
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: melody;
     start: <multiple of 6x4>;
  duration: 6x4;
      grid: <statistically derived from the input sequence>;
 structure: <statistically derived from the input sequence>;
     chord: <statistically derived from the input sequence>;
     relax: <statistically derived from the input sequence>;
       min: <statistically derived from the input sequence>;
       max: <statistically derived from the input sequence>;
       low: <statistically derived from the input sequence>;
      high: <statistically derived from the input sequence>;
```

Figure 4.5: Applying the GeneralMelody filter

Chapter 5

Conclusion

5.1 Evaluation

As declared at the very beginning of this document, I have decided to take advantage of the fact that Salzella programs are presumed to be generated rather than written by hand and designed the language without explicit support for abstract structures like chords, scales and keys. Instead, I attempted to design the language in a way which would allow Salzella users to manually define these structures within individual programs. In retrospect, this approach had the following advantages and disadvantages:

Advantages The most obvious advantage of adopting the low level approach was overall simplification of the language. Possibility of manual definition of arbitrary scales/chords was achieved by providing specification of a single type of matrix, the surface matrix. Note that the concept of surface matrices plays the key role in satisfying the music genre independence requirement. By even the slightest modification of program's surface matrix, contents of the outputted musical piece can change dramatically. For example, program for generating rock music, see A.1, uses the surface matrix to define a chord progression made of power chords and declares usage of pentatonic scale. Program for generating folk music, see A.4, uses the same data structure to define harmony based on major/minor triads and declares usage of diatonic scale.

Disadvantages The biggest disadvantage of the low level approach is, of course, delegating the responsibility of defining the high level structures to the user. Adopting the low level approach also implies growth of Salzella programs in terms of number of lines. Since ease of manual creation of Salzella programs was not a priority, neither of these disadvantages presented a problem. In fact, Salzella programs can be as long as tens of thousands of lines of code and still satisfy the fundamental requirements stated at the very beginning of this document.

5.1.1 Descriptive nature

As illustrated by the conversion tool described in the previous chapter, describing an existing musical piece by means of a Salzella program can be achieved by a fairly simple statistical

analysis. The amount of control over the generated music will, however, always be limited by abilities of Salzella extensions used for the actual generation of musical contents. The three simple Salzella extensions created as part of this thesis are certainly not sufficient to fully satisfy the descriptive nature requirement. For example, program for generating classical music, see A.5, generates three melodies. Since neither of the three existing Salzella extensions provides enough control over generating multiple melody lines, this program generates three independent melodies. Sure, these melodies share the same harmony surface and the resulting musical piece will often sound acceptable. However, by creating Salzella extension for generating melodies able to respect the already existing musical contents, creators of Salzella programs would be granted control over the contrapuntal motion between multiple melody lines.

5.1.2 Bundle requirement

As shown by creating the conversion tool, Salzella interpreter can be easily integrated with music generating applications. Note, however, that Salzella can be useful even in software which is not strictly related to music generation. For example, the development environment created as part of this thesis uses Salzella lightweight implementation of MIDI entities to represent musical contents.

5.1.3 Absorbtion requirement

To prove that external music generating algorithms can be integrated into Sazlella platform, three Salzella extensions were created. Note that Salzella extensions will be typically responsible for generating specific parts of musical piece such as drum beats, vocal lines, complementary melodies etc. However, it is possible to use a single Salzella extension to generate the whole musical piece.

5.2 Future work

5.2.1 Verse and chorus support

The current version of Salzella works best when dealing with musical ideas few bars long. Even though it would be possible to create an extension which would allow Salzella users to work with concepts like verses and choruses, adding explicit support for defining custom meaning of manually selected time windows should be considered.

5.2.2 Surface matrix convention

The twelve-integer surface matrix convention doesn't seem to be sufficient when dealing with more complex music. One possible course of action could be using twenty four integers per row. This would allow users to store information about scales and chords separately. Going even further and using thirty six integers per row would make it possible to use different scales based on direction of melodies. This would be particularly useful when dealing with melodic minor scales. No matter what approach will be chosen, limiting the size of surface matrix rows is unnecessary and should be definitely removed from Salzella specification.

5.2.3 Manual creation of surface matrices

Even though Salzella programs are presumed to be generated rather than written by hand, developers of Salzella extension will often want to create surface matrices manually to test their algorithms. Manual creation of surface matrices is a tedious and error prone task. Once surface matrix convention is stable enough, a surface matrix generation tool should be added to the development environment to make manual creation of surface matrices easier.

Bibliography

[1] Biedrzycki, Maciej. "Can Computers Create Music?" CgMusic. May 19, 2008. Accessed December 23, 2015. http://codeminion.com/blogs/maciek/2008/05/cgmusic-computers-create-music/.

[2] "User-Customized Jazz Improvisation Generation." Jazzerbot. Accessed December 23, 2015. https://code.google.com/p/jazzerbot/.

[3] "An Experiment in a New Kind of Music." WolframTones. Accessed December 23, 2015. http://tones.wolfram.com/.

[4] "Music Programming for Java™ and JVM Languages." JFugue. Accessed December 23, 2015. http://www.jfugue.org/.

[5] "Computer Music Composition in Java." JMusic. Accessed December 23, 2015. http://explodingart.com/jmusic/.

[6] "MIDI Specification." MIDI Manufacturers Association. Accessed December 23, 2015. http://midi.org/.

[7] Pratchett, Terry. Maskerade: A Novel of Discworld Series. New York: HarperPrism, 1997.

Appendix A

Example programs

A.1 Program 1 - Rock

```
      tempo: 120;
    surface: 1x1 0 0 1 0 3 0 0 1 0 1 0 2,
             1x1 0 0 3 0 1 0 0 1 0 2 0 1,
             1x1 3 0 1 0 1 0 0 2 0 1 0 1,
             1x1 1 0 1 0 1 0 2 1 0 1 0 3,
             1x1 0 0 1 0 3 0 0 1 0 1 0 2,
             1x1 0 0 3 0 1 0 0 1 0 2 0 1,
             1x1 3 0 1 0 1 0 0 2 0 1 0 1,
             1x1 1 0 1 0 1 0 2 1 0 1 0 3;
signature: 8 4 1x4;
   tracks: guitar 30 105 false false,
           organ  18  95 false false,
           muted  28  70 false false,
           drums 128 120 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: guitar;
    start: 2x1;
 duration: 2x1;
     grid: 1x8;
structure: 0.7 0.8 0.5,
           UP STEADY DOWN UP;
    chord: 0.5;
    relax: 0.8;
      min: 0;
      max: 1;
      low: e4;
     high: e5;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;

+ 35 more lines
```

Figure A.1: Salzella program for generating rock music

A.2 Program 2 - Blues

```
      tempo: 80;
    surface: 4x1 0 0 2 0 3 0 0 1 2 1 0 2,
             2x1 0 2 1 0 2 0 0 2 0 3 0 1,
             2x1 0 0 2 0 3 0 0 1 2 1 0 2,
             1x1 0 0 1 2 1 0 2 1 0 2 0 3,
             1x1 0 2 1 0 2 0 0 2 0 3 0 1,
             1x1 0 0 2 0 3 0 0 1 2 1 0 2,
             1x1 0 0 1 2 1 0 2 1 0 2 0 3;
signature: 12 4 1x4;
   tracks: sax    65 120 false false,
           piano   4  90 false false,
           drums 128 120 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: sax;
    start: 0x1;
 duration: 12x1;
     grid: 1x8t;
structure: 0.3 0.5 0.5,
           UP DOWN DOWN UP STEADY STEADY STEADY UP UP;
    chord: 0.4;
    relax: 0.2;
      min: 0;
      max: 3;
      low: e4;
     high: b5;
           ---
   source: cz.stepanvolf.salzella.plugin.SimpleDrums;
    input: drums;
    start: 0x1;
 duration: 12x1;
     loop: 1x1;
    hihat: 1x8t;
    snare: 1x4 3x4;
     kick: 0.3;
    crash: 0.0;
  variety: 0.0;
           ---
   source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
    input: piano;
    start: 0x1;
 duration: 12x1;
     mode: RHYTHM;
     grid: 1x8t;
```

Figure A.2: Salzella program for generating blues music

A.3 Program 3 - Jazz

```
      tempo: 60;
    surface: 1x1 0 0 2 0 0 0 2 3 0 0 0 2,
             1x1 3 0 0 0 2 0 0 2 0 0 0 2,
             1x1 2 0 0 0 2 0 0 2 0 3 0 0,
             1x1 0 0 0 2 0 0 2 0 3 0 0 2;
signature: 4 4 1x4;
   tracks: guitar 26 100 false false,
             piano  26  90 false false,
             bass   32 100 false false,
             drums 128 120 false false;
             ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: guitar;
    start: 0x1;
 duration: 2x1;
     grid: 1x8t;
structure: 0.3 0.5,
             UP DOWN STEADY;
    chord: 0.7;
    relax: 0.3;
      min: 0;
      max: 3;
      low: c4;
     high: e6;
             ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: guitar;
    start: 2x1;
 duration: 2x1;
     grid: 1x4;
structure: 0.8 0.5,
             UP DOWN STEADY;
    chord: 0.7;
    relax: 0.3;
      min: 0;
      max: 3;
      low: c4;
     high: e6;
             ---
   source: cz.stepanvolf.salzella.plugin.SimpleDrums;
    input: drums;
    start: 0x1;

+ 26 more lines
```

Figure A.3: Salzella program for generating jazz music

A.4 Program 4 - Folk

```
      tempo: 120;
    surface: 1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 1 0 3 0 1 2 0 1 0 2,
             1x1 3 0 1 0 2 0 1 2 0 1 0 1,
             1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 1 0 3 0 1 2 0 1 0 2,
             2x1 1 0 3 0 1 0 2 1 0 2 0 1;
signature: 8 4 1x4;
   tracks: flute  73 100 false false,
           guitar 25 100 false false,
           drums 128 120 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: flute;
    start: 0x1;
 duration: 4x1;
     grid: 1x4;
structure: 0.7 0.5,
           UP DOWN UP STEADY;
    chord: 1.0;
    relax: 1.0;
      min: 0;
      max: 3;
      low: g4;
     high: e6;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: flute;
    start: 4x1;
 duration: 4x1;
     grid: 1x4;
structure: 0.9 0.3 0.5 0.9,
           DOWN DOWN UP UP;
    chord: 1.0;
    relax: 0.5;
      min: 0;
      max: 3;
      low: g4;
     high: f#6;
           ---
   source: cz.stepanvolf.salzella.plugin.SimpleDrums;

+ 14 more lines
```

Figure A.4: Salzella program for generating folk music

A.5 Program 5 - Classical

```
    tempo: 120;
  surface: 1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 2 0 1 0 2 1 0 1 0 3,
           1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 2 0 1 0 2 1 0 1 0 3,
           1x1 3 0 1 0 2 0 1 2 0 1 0 1,
           1x1 1 0 3 0 1 0 2 1 0 2 0 1,
           2x1 1 0 1 0 3 0 1 2 0 1 0 2;
signature: 8 4 1x4;
   tracks: violin     40 105 false false,
           cello      42  95 false false,
           contrabass 43  95 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: violin;
    start: 2x1;
 duration: 6x1;
     grid: 1x8;
structure: 0.2,
           STEADY;
    chord: 0.9;
    relax: 1.0;
      min: 0;
      max: 3;
      low: e5;
     high: e6;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: cello;
    start: 1x1;
 duration: 7x1;
     grid: 1x4;
structure: 0.3,
           STEADY;
    chord: 0.2;
    relax: 1.0;
      min: 0;
      max: 12;
      low: e3;
     high: e4;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;

+ 10 more lines
```

Figure A.5: Salzella program for generating classical music

Appendix B

Contents of the enclosed CD

```
├── javadoc
│   ├── converter-javadoc.zip
│   ├── ide-javadoc.zip
│   ├── plugins-javadoc.zip
│   └── salzella-javadoc.zip
├── latex
│   ├── documentation.pdf
│   └── documentation.zip
├── samples
│   ├── greensleeves.mid
│   ├── jethro.mid
│   └── scarborough.mid
├── semestral
│   ├── A4M35KO.pdf
│   ├── A4M39NUR.pdf
│   ├── A4M39PUR.pdf
│   └── Y14TED.pdf
├── source
│   ├── converter-source.zip
│   ├── ide-dependencies.zip
│   ├── ide-source.zip
│   ├── plugins-source.zip
│   └── salzella-source.zip
├── ide.jar
└── presentation.mp4
```