Computer Composition of Popular Music using Elementary Cellular Automata

by

Jeffrey Clint Hamblin

A thesis submitted to Florida Institute of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Melbourne, Florida December, 2008
We the undersigned committee hereby approve the attached thesis

Computer Composition of Popular Music using Cellular Automata

by

Jeffrey Clint Hamblin

Gregory Harrison, Ph.D.
Adjunct Professor, Computer Science
Thesis Advisor and Committee Chair

Dennis Kulonda, Ph.D., P.E.
Professor and Site Director, Management Committee Chairperson

Richard James, Ph.D.
Associate Professor, Computer Science
Rollins College
Committee Chairperson

William Schoaff, Ph.D.
Professor and Head, Computer Science
Abstract

TITLE: Computer Composition of Popular Music using Elementary Cellular Automata

AUTHOR: Jeffrey Clint Hamblin

MAJOR ADVISOR: Gregory Harrison, Ph.D.

This thesis explores generating popular music using a simple system called Frond that relies on minimal user feedback. An approach is described to autonomously select song characteristics, chords, and cellular automata parameters and how to generate music based on these selections. Finally, a naïve Bayes classifier is discussed that allows the system to learn which parameters are favored by the user.

Songs generated by Frond exhibit several common behaviors, most of which are simplistic themselves. The resulting music is thus either pleasant but simple or complex but lost. Suggestions and possible improvements to the approach taken in Frond are then discussed.
Table of Contents

Table of Contents ........................................................................................................................................ iv
Table of Figures ............................................................................................................................................... vii
Table of Tables ............................................................................................................................................... ix
Table of Equations ......................................................................................................................................... x
Chapter 1......................................................................................................................................................... 1
  Statement of the Problem .......................................................................................................................... 1
  Definition of Terms .................................................................................................................................... 1
  Review of Literature ................................................................................................................................. 4
  Description of Remaining Chapters ......................................................................................................... 6
Chapter 2......................................................................................................................................................... 8
  Requirements and Specifications ............................................................................................................... 8
  Algorithms and Data Structures ............................................................................................................. 10
Chapter 3......................................................................................................................................................... 11
  Parts .......................................................................................................................................................... 11
  Key .......................................................................................................................................................... 12
  Time ......................................................................................................................................................... 14
  Structure .................................................................................................................................................. 15
  Dynamics .................................................................................................................................................. 16
Chapter 4......................................................................................................................................................... 18
  Chords .................................................................................................................................................... 18
    Type ..................................................................................................................................................... 19
    Scale Degree ......................................................................................................................................... 23
  Selecting Progressions ............................................................................................................................ 25
Chapter 5......................................................................................................................................................... 29
  Cellular Automata ................................................................................................................................. 29
  Initial Conditions ................................................................................................................................. 32
Chord.cs .................................................................................................................. 160
ChordInterval.cs ................................................................................................. 171
Dynamics.cs ........................................................................................................... 173
Key.cs ....................................................................................................................... 175
KeyInterval.cs ....................................................................................................... 183
Note.cs ....................................................................................................................... 185
NoteDerivation.cs .................................................................................................... 191
Song.cs ....................................................................................................................... 194
SongBuilder.cs ......................................................................................................... 205
Time.cs ...................................................................................................................... 209
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Treble and bass clefs.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2</td>
<td>C major and A minor scales.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Majors and relative minor signatures.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Time signature.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Example song structure.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Dynamics in Frond.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Progressions with scale degree and type.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Selection of scale degree.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Selection of chord type.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 10</td>
<td>A binary cellular automaton.</td>
<td>30</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Rule 30 cellular automaton.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Initial conditions affecting behavior.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Initial conditions having little effect on behavior.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Elementary rule numbering.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Example mobile automaton rules.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Example mobile automaton.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Fracturing cellular automaton.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Fracturing rule numbering.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Repetition in rule 246.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Nesting in rule 18.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Randomness in rule 86.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Localized structures in rule 124.</td>
<td>42</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Two dimensional cellular automaton.</td>
<td>43</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Dimension in music vs. one-dimensional CAs.</td>
<td>44</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Rule 30, size 7, cyclic edges.</td>
<td>45</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Song profile tree.</td>
<td>48</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Steps of note derivation.</td>
<td>49</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Mapping notes to columns.</td>
<td>51</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Completed note assignment.</td>
<td>53</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Out-of-range conditions.</td>
<td>54</td>
</tr>
<tr>
<td>Figure 31</td>
<td>Bass/melody intersection.</td>
<td>55</td>
</tr>
<tr>
<td>Figure 32</td>
<td>Rhythm example.</td>
<td>57</td>
</tr>
</tbody>
</table>
Figure 33 - Repeating whole notes. .................................................................67
Figure 34 - Repeating whole notes in the cellular automaton. .........................67
Figure 35 - Repeating quarter notes. ................................................................67
Figure 36 - Repeating quarter notes in the cellular automaton. .........................68
Figure 37 - Trilling ..........................................................................................68
Figure 38 - Trilling in the cellular automaton .......................................................69
Figure 39 - Descending notes ..........................................................................69
Figure 40 - Descending notes in the cellular automaton ......................................70
Figure 41 - Ascending notes ............................................................................70
Figure 42 - Ascending notes in the cellular automaton ......................................71
Figure 43 - Seemingly random behavior. ..............................................................71
Figure 44 - Seemingly random behavior in the cellular automaton ....................72
Figure 45 - Repeating melody and bass for Superman (It's Not Easy) .................74
Figure 46 - Random melody, repeating bass for Superman (It's Not Easy) ..........75
Figure 47 - Random melody and bass for Superman (It's Not Easy) .................75
Figure 48 - Repeating melody and bass for Walking In Memphis ......................75
Figure 49 - Random melody, repeating bass for Walking In Memphis ...............76
Figure 50 - Random melody and bass for Walking In Memphis .......................76
Figure 51 - Repeating melody and bass for Clocks. ..........................................77
Figure 52 - Random melody, repeating bass for Clocks .....................................77
Figure 53 - Random melody and bass for Clocks. ..........................................77
## Table of Tables

Table 1 - Dynamics levels. ................................................................. 17
Table 2 - Chord types found in popular music. .................................. 20
Table 3 - Intervals between notes. ..................................................... 22
Table 4 - Scales and degrees. ............................................................. 24
Table 5 - Interval calculations for C minor from G5. .......................... 52
Table 6 - Timing and duration of notes. ............................................. 57
Table 7 - Timing and duration with rests. ......................................... 58
Table 8 - Notional set of song data. .................................................. 61
Table 9 - Notional song with all good attributes. .............................. 61
Table 10 - Notional song with one attribute that is not good. .......... 62
Table of Equations

Equation 1 - Naïve Bayes classifier.................................................................61
Radio, television, and the Internet are dominated by popular music, which is a loose subset of music known for its broad appeal and mass-marketing to the general public. Currently, popular music is created by teams of artists, engineers, producers, and others with the assistance of computer programs such as Pro Tools. However, it is unlikely that computers, without the assistance of humans, have created songs similar to those found in popular music. This paper explores creating an application to generate popular music autonomously using simple algorithms with only minimal human feedback for learning purposes. This application is referred to as Frond.

**Definition of Terms**

Common terms used throughout this document are provided below with definitions.
- Cellular automata – A system containing cells whose states change over time according to a set of rules.

- Chord – Three or more notes played in harmony.

- Dynamics – The volume of notes played within a passage of music.

- Element – The highest level structure contained in a popular song, such as the verse, chorus, and bridge.

- Elementary rule – Defines the behavior and states of the cells in an elementary cellular automaton.

- Fracturing rule – Defines the behavior of the active cell in a fracturing elementary cellular automata.

- Frond – The name of the project created to demonstrate the principles outlined in this research.

- Interval – The distance between any two notes in a song.

- Key – Consists of a root note and scale and defines the notes that are typically played in a song.

- Measure – A grouping of notes used to organize a song based which typically represents one whole note of time.

- Naïve Bayes classifier – A learning algorithm that searches a set of data to determine the likelihood that an object with specific independent attributes belongs to a specific class within that data. A naïve Bayes
classifier can be used to determine how “good” a song is given that it consists of certain attributes.

- Note assignment – A process in Frond that maps a note derivation to cells within a cellular automaton.
- Note derivation – A process in Frond that combines the notes in the key with those of the chord currently being played in the song.
- Part – A subset of a song that is typically played by a single instrument.
- Pattern – A structure that tends to repeat as part of an element in a song.
- Profile – The sequence of chords in the order they are played in a song.
- Rhythm – The timing and duration of notes played within a song.
- Scale – A series of notes and the intervals between each that defines the notes typically played in a song.
- Scale degree – A number referring to the distance a given note in a scale is from the root note.
- Structure – The order and repetition of various elements, patterns, and measures in a song.
- Tempo – The rate that a song moves forward in time. Tempo is generally measured in beats per minute.
• Time signature – Indicated by two numbers, where the top number defines the number of beats per measure and the bottom defines the note whose duration is one beat.

**Review of Literature**

The following literature contributed to the research of and is cited in this thesis:

  o A website dedicated to explaining the basics of music theory.

• BPM Database. (n.d.).
  o An Internet database containing the tempos of over 23,000 user-submitted songs.

  o An article describing the rules used and patterns found in Conway’s game of “life”.

  o A book that contains sheet music and chords for 36 popular songs from the last several decades.
  o A book with an in-depth explanation of music theory topics, such as notation, scales, chords, and keys.

  o A website which provides an application that demonstrates an example naïve Bayes classifier as well as a method for dealing with zero probabilities for attributes that have not been classified.

  o A book describing various approaches to generating music using computers, including Markov chains, neural networks, and cellular automata.

  o A book that provides a comprehensive overview of artificial intelligence topics, including knowledge, reasoning, learning, and communicating.

  o A book describing the common elements of popular songs and providing a database of chord progressions used in popular songs predominately from the 20th century.

A book emphasizing the use of simple systems like cellular automata that produce complex results. Provides a thorough explanation of cellular automata and their rules, behaviors, and possible applications to the real world.


- A website that defines a list of chord notations and the intervals for each note in the chord.

**Description of Remaining Chapters**

Below is a list of chapters that follow and a brief description of each:

- **Methods Used** – Describes the requirements for Frond and provides a brief description of any algorithms used.

- **Song Characteristics** – Explains several traits of popular songs including parts, tempo, time signature, key, and dynamics which define the attributes of songs in Frond.

- **Chord Selection** – Describes chords, chord types, and scale degrees as defined within Frond. Also describes a process that could be used to select chord progressions that are favored by the user.
- **Music Generation** – Describes cellular automata and the rules that define those used in Frond. Also covers the process of generating music using the states of cells in the cellular automaton and how those states relate to the song characteristics and chords.

- **Learning using a Naïve Bayes Classifier** – Illustrates a method by which a music generation system such as Frond could learn what attributes make favorable songs.

- **Conclusions** – States the results of generating music using Frond as well as the problems left unsolved.
Chapter 2
Methods Used

Requirements and Specifications

The following requirements document Frond’s input, algorithms, and output:

Input

- Frond shall generate music that contains a melody part and a bass part.
- Frond shall allow a user to define the following attributes of the song:
  - Tempo
  - Time signature
  - Key
  - Structure
  - Chords
  - Cellular automaton rules (elementary and fracturing)
  - Cellular automaton initial conditions
  - Dynamics
**Algorithms**

- Frond shall generate music using an elementary cellular automaton.
- Frond shall determine pitches that map to each cell in the cellular automaton.
- Frond shall determine when each note is played using the states of cells in the cellular automaton.
- Frond shall determine the duration of each note using the states of cells in the cellular automaton.

**Output**

- Frond shall output the song in a MusicXML 2.0 compatible format.
- Frond shall display the state of the cellular automaton.
- Frond shall display the note mapping used by the cellular automaton.
**Algorithms and Data Structures**

The following algorithms are discussed in this document:

- Probablistic chord selection – selects a chord progression using a series of lookup tables using the conditional probability that chord Y is chosen given that chord X was already chosen.
- Elementary cellular automata with fracturing – propagates the states of cells over a series of discrete time steps according to a set of elementary rules. Fracturing adds an additional rule set that governs the movement of an active cell through each time step in the cellular automaton.
- Note mapping – derives the notes that are playable within the cellular automaton for each chord played in the song. The note that is currently played is the note that is mapped to the active cell.
- Rhythm rules – determines when a note is struck and how long it is held. Can also determine if a rest is occurring.
- Naïve Bayes learning – defines a process by which songs are classified as “good” or “bad” and the characteristics of each class that are considered when autonomously selecting the characteristics of a new song.
Chapter 3
Song Characteristics

There are several characteristics of a song that can be chosen without knowledge of the chords, notes, and rhythm. These characteristics are parts, key, time, structure, and dynamics. Parts provide organization for individual instruments, key is essential in determining the notes in the song, time in establishing rhythm, structure in establishing patterns within the music, and dynamics in expressing volume. In an autonomous music generation system such as Frond, song characteristics could be chosen probabilistically using lookup tables.

Parts

Parts are used in music to separate individual instruments from one another. For example, a song may have a drum part, two guitar parts, a bass part, and a voice part. Piano parts are typically divided into two separate parts: melody and bass. Since Frond uses only the piano as an instrument, the most logical representation by which a song could be expressed on a piano is with a melody part and a bass part.
The notation for the melody and bass parts are a treble clef and bass clef, respectively. The horizontal lines are the staff, which is used to read what notes to play. This is illustrated in Figure 1.

![Treble and bass clefs](image)

Figure 1 - Treble and bass clefs.

Knowledge of reading and writing music is not required to understand the approach taken with Frond, so that will not be discussed within this paper. It is, however, helpful and thus this paper assumes the reader has a basic understanding of music and its notation.

**Key**

The key of a song indicates the notes that are typically played throughout the course of a song. The key is defined by a root note and a scale. The root note is the pitch where the scale begins. The scale defines what notes can be played throughout the song relative to the root and to adjacent notes. While many scales exist, Frond uses only the major and minor scales which are shown below in Figure 2.
The major and minor scales are diatonic scales composed of seven notes each with five whole steps and two half steps. Two semitones make up a whole step and one semitone a half step. A semitone is the smallest step that can be taken between notes with there being a total of twelve semitones in an octave. Besides root note, the scales in the figure differ only in the number of semitones between notes in the scale. The semitones between notes in the major scale are 2-2-1-2-2-2-1 and in the minor scale are 2-1-2-2-1-2-2.

The scales in Figure 2 are part of the C major and A minor keys. There are a total of twelve major and twelve minor keys, each with different signatures. Each major key has an equivalent relative minor key which shares the same signature; for example, C major and A minor. Each major and its relative minor key are shown below in Figure 3 (Jones, 1974, p. 30).
Figure 3 - Majors and relative minor signatures.

The notes defined by the key are combined with chords during music generation to produce the playable notes for each portion of the song. This process is called note derivation and is described on page 49.

**Time**

Time is used to establish a sense of rhythm in music. Two components to time that can be applied to an entire song are time signature and tempo. Some music varies these parameters as the song progresses, however they are held constant throughout a song in Frond.

A time signature defines the number of beats within each measure and the note duration that is equivalent to one beat. In Figure 4, 4/4 is the time signature.
The top 4 means that there are four beats per measure and the lower 4 means that a quarter note (1/4) is equivalent to one beat.

![Figure 4 - Time signature.](image)

While there are a variety of time signatures in music, Frond uses only 4/4 time as it is commonly found within popular music.

The tempo of a song defines how many beats are played per minute. More beats per minute increases the rate at which the song progresses, and vice versa. Frond permits a range of tempos typically found in popular music: 32 to 192 (BPM Database). This range is based on the mean tempo (112 bpm) of 23,884 entries in the database within a range of three standard deviations (±80 bpm).

**Structure**

Structure organizes a song into distinct sections, some of which repeat throughout the song. These sections are referred to as elements. Scott defines several elements of structure found in popular music: intro (I), verse (V), prechorus (PC), chorus (C), turnaround (TA), bridge (B), outro (O) (Scott, 2000, p.
xx). Frond allows a user to define the structure of the song using any combination of these elements. An example song structure is shown in Figure 5.

<table>
<thead>
<tr>
<th>Measures</th>
<th>(any number of measures can be defined per pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patterns</td>
<td>V1  V2  V3  V4  C1  C2  V1  V2  V3  V4  C1  C2  B1  C1  C2</td>
</tr>
<tr>
<td>Elements</td>
<td>Verse  Chorus  Verse  Chorus  Bridge  Chorus</td>
</tr>
<tr>
<td>Structure</td>
<td>Song</td>
</tr>
</tbody>
</table>

**Figure 5 - Example song structure.**

Each element is defined by one or more patterns in Frond. These patterns each contain one or more measures, each which contain a chord from which notes are derived. The process of generating a profile of chords based on structure is described on page 47.

**Dynamics**

Dynamics are used to express the loudness and changes in loudness of a particular passage of music (Jones, 1974, pp. 64-65). There are several common levels of dynamics as defined in Table 1.
Like structure, dynamics are user-selectable in Frond and are applied to each pattern within the song. An example of dynamics is shown in Figure 6.

<table>
<thead>
<tr>
<th>Dynamics</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pianissimo</td>
<td><em>pp</em></td>
<td>Very quiet</td>
</tr>
<tr>
<td>Piano</td>
<td><em>p</em></td>
<td>Quiet</td>
</tr>
<tr>
<td>Mezzo-piano</td>
<td><em>mp</em></td>
<td>Medium quiet</td>
</tr>
<tr>
<td>Mezzo-forte</td>
<td><em>mf</em></td>
<td>Medium loud</td>
</tr>
<tr>
<td>Forte</td>
<td><em>f</em></td>
<td>Loud</td>
</tr>
<tr>
<td>Fortissimo</td>
<td><em>ff</em></td>
<td>Very loud</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamics</th>
<th><em>mf</em></th>
<th><em>f</em></th>
<th><em>ff</em></th>
<th><em>mf</em></th>
<th><em>f</em></th>
<th><em>ff</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Patterns</td>
<td>V1</td>
<td>V2</td>
<td>V3</td>
<td>V4</td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>Elements</td>
<td>Verse</td>
<td>Chorus</td>
<td>Verse</td>
<td>Chorus</td>
<td>Bridge</td>
<td>Chorus</td>
</tr>
<tr>
<td>Structure</td>
<td>Song</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 6 - Dynamics in Frond.*
Chapter 4
Chord Selection

Chords are the basis from which notes are generated in Frond. Chords are combined into pairs called progressions, each of which can be chosen when creating a song. The selection of chords for the song completes the prerequisite information needed for the music generation process.

Chord selection does not explicitly take place in Frond. However, it is discussed in the sections that follow to reinforce the concept that all aspects of a song could be automatically chosen without input from a user.

Chords

Chords are a combination of three or more notes and are built off of a single note, called the root (Adams, 2005). In Frond, chords are defined by type and scale degree. The type of a chord determines which notes are played in harmony. The scale degree determines the root note of the chord relative to the key and scale of the song. While there are many other details to chords, type and
scale degree are powerful enough to represent the vast majority of those found in popular music (Scott, 2000).

**Type**

There are many different types of chords. A list of chords found in popular music (Scott, 2000) with the notations and spelling of each (Wright, 2006) is shown below in Table 2. There are a total of 34 chord types available in Frond.
<table>
<thead>
<tr>
<th>Chord type</th>
<th>Notation</th>
<th>Spelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>M or (none)</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>Minor</td>
<td>m</td>
<td>1, b3, 5</td>
</tr>
<tr>
<td>Diminished</td>
<td>d</td>
<td>1, b3, b5</td>
</tr>
<tr>
<td>Diminished 7th</td>
<td>d7</td>
<td>1, b3, b5, bb7</td>
</tr>
<tr>
<td>Half diminished</td>
<td>m7b5</td>
<td>1, b3, b5, b7</td>
</tr>
<tr>
<td>Augmented</td>
<td>A</td>
<td>1, 3, #5</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>5</td>
<td>1, 5</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7</td>
<td>1, 3, 5, b7</td>
</tr>
<tr>
<td>Minor 7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m7</td>
<td>1, b3, 5, b7</td>
</tr>
<tr>
<td>Major 7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>M7</td>
<td>1, 3, 5, 7</td>
</tr>
<tr>
<td>Minor/major 7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m/Maj7</td>
<td>1, b3, 5, 7</td>
</tr>
<tr>
<td>Suspended 4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>sus4</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>Suspended 2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>sus2</td>
<td>1, 2, 5</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; suspended 4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7sus4</td>
<td>1, 4, 5, b7</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; suspended 2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>7sus2</td>
<td>1, 2, 5, b7</td>
</tr>
<tr>
<td>Added 2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>add2</td>
<td>1, 2, 3, 5</td>
</tr>
<tr>
<td>Added 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>add9</td>
<td>1, 3, 5, 9</td>
</tr>
<tr>
<td>Added 4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>add4</td>
<td>1, 3, 4, 5</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>6</td>
<td>1, 3, 5, 6</td>
</tr>
<tr>
<td>Minor 6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m6</td>
<td>1, b3, 5, 6</td>
</tr>
<tr>
<td>6/9</td>
<td>6/9</td>
<td>1, 3, 5, 6, 9</td>
</tr>
<tr>
<td>9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>9</td>
<td>1, 3, 5, b7, 9</td>
</tr>
<tr>
<td>Minor 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m9</td>
<td>1, b3, 5, b7, 9</td>
</tr>
<tr>
<td>Major 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>M9</td>
<td>1, 3, 5, 7, 9</td>
</tr>
<tr>
<td>11&lt;sup&gt;th&lt;/sup&gt;</td>
<td>11</td>
<td>1, 3, 5, b7, 9, 11</td>
</tr>
<tr>
<td>Minor 11&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m11</td>
<td>1, b3, 5, b7, 9, 11</td>
</tr>
<tr>
<td>Major 11&lt;sup&gt;th&lt;/sup&gt;</td>
<td>M11</td>
<td>1, 3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>13&lt;sup&gt;th&lt;/sup&gt;</td>
<td>13</td>
<td>1, 3, 5, b7, 9, 11, 13</td>
</tr>
<tr>
<td>Minor 13&lt;sup&gt;th&lt;/sup&gt;</td>
<td>m13</td>
<td>1, b3, 5, b7, 9, 11, 13</td>
</tr>
<tr>
<td>Major 13&lt;sup&gt;th&lt;/sup&gt;</td>
<td>M13</td>
<td>1, 3, 5, 7, 9, 11, 13</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; sharp 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7#9</td>
<td>1, 3, 5, b7, #9</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; flat 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7b9</td>
<td>1, 3, 5, b7, b9</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; sharp 5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7#5</td>
<td>1, 3, #5, b7</td>
</tr>
<tr>
<td>7&lt;sup&gt;th&lt;/sup&gt; flat 5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>7b5</td>
<td>1, 3, b5, b7</td>
</tr>
</tbody>
</table>
Chord types are generally named according to the interval between the root note and the highest note in the chord. An interval is defined as the number of steps it takes to get from one note to another in the current scale. Intervals can either be major (M), minor (m), augmented (A), diminished (d), and perfect (P). Intervals 1, 4, 5, and 8 can be perfect, augmented, or diminished while intervals 2, 3, 6, and 7 can be either major, minor, augmented, or diminished (Jones, 1974, pp. 31-35). Table 3 illustrates the various intervals between two notes using the C Major scale and a starting note of C. Intervals greater than eight follow the same pattern as outlined below.
### Table 3 - Intervals between notes.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Type</th>
<th>Abbreviation</th>
<th>Notes</th>
<th>#</th>
<th>Spelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unison</td>
<td>Diminished</td>
<td>d1</td>
<td>C, Cb (B)</td>
<td>1</td>
<td>1, b1</td>
</tr>
<tr>
<td>Unison</td>
<td>Perfect</td>
<td>P1</td>
<td>C, C</td>
<td>0</td>
<td>1, 1</td>
</tr>
<tr>
<td>Unison</td>
<td>Augmented</td>
<td>A1</td>
<td>C, C#</td>
<td>1</td>
<td>1, #1</td>
</tr>
<tr>
<td>Second</td>
<td>Diminished</td>
<td>d2</td>
<td>C, Db</td>
<td>0</td>
<td>1, bb2</td>
</tr>
<tr>
<td>Second</td>
<td>Minor</td>
<td>m2</td>
<td>C, Db</td>
<td>1</td>
<td>1, b2</td>
</tr>
<tr>
<td>Second</td>
<td>Major</td>
<td>M2</td>
<td>C, D</td>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>Second</td>
<td>Augmented</td>
<td>A2</td>
<td>C, D#</td>
<td>3</td>
<td>1, #2</td>
</tr>
<tr>
<td>Third</td>
<td>Diminished</td>
<td>d3</td>
<td>C, Ebb</td>
<td>2</td>
<td>1, bb3</td>
</tr>
<tr>
<td>Third</td>
<td>Minor</td>
<td>m3</td>
<td>C, Eb</td>
<td>3</td>
<td>1, b3</td>
</tr>
<tr>
<td>Third</td>
<td>Major</td>
<td>M3</td>
<td>C, E</td>
<td>4</td>
<td>1, 3</td>
</tr>
<tr>
<td>Third</td>
<td>Augmented</td>
<td>A3</td>
<td>C, E# (F)</td>
<td>5</td>
<td>1, #3</td>
</tr>
<tr>
<td>Fourth</td>
<td>Diminished</td>
<td>d4</td>
<td>C, Fb (E)</td>
<td>4</td>
<td>1, b4</td>
</tr>
<tr>
<td>Fourth</td>
<td>Perfect</td>
<td>P4</td>
<td>C, F</td>
<td>5</td>
<td>1, 4</td>
</tr>
<tr>
<td>Fourth</td>
<td>Augmented</td>
<td>A4</td>
<td>C, F#</td>
<td>6</td>
<td>1, #4</td>
</tr>
<tr>
<td>Fifth</td>
<td>Diminished</td>
<td>d5</td>
<td>C, Gb</td>
<td>6</td>
<td>1, b5</td>
</tr>
<tr>
<td>Fifth</td>
<td>Perfect</td>
<td>P5</td>
<td>C, G</td>
<td>7</td>
<td>1, 5</td>
</tr>
<tr>
<td>Fifth</td>
<td>Augmented</td>
<td>A5</td>
<td>C, G#</td>
<td>8</td>
<td>1, #5</td>
</tr>
<tr>
<td>Sixth</td>
<td>Diminished</td>
<td>d6</td>
<td>C, Abb</td>
<td>7</td>
<td>1, bb6</td>
</tr>
<tr>
<td>Sixth</td>
<td>Minor</td>
<td>m6</td>
<td>C, Ab</td>
<td>8</td>
<td>1, b6</td>
</tr>
<tr>
<td>Sixth</td>
<td>Major</td>
<td>M6</td>
<td>C, A</td>
<td>9</td>
<td>1, 6</td>
</tr>
<tr>
<td>Sixth</td>
<td>Augmented</td>
<td>A6</td>
<td>C, A#</td>
<td>10</td>
<td>1, #6</td>
</tr>
<tr>
<td>Seventh</td>
<td>Diminished</td>
<td>d7</td>
<td>C, Bbb</td>
<td>9</td>
<td>1, bb7</td>
</tr>
<tr>
<td>Seventh</td>
<td>Minor</td>
<td>m7</td>
<td>C, Bb</td>
<td>10</td>
<td>1, b7</td>
</tr>
<tr>
<td>Seventh</td>
<td>Major</td>
<td>M7</td>
<td>C, B</td>
<td>11</td>
<td>1, 7</td>
</tr>
<tr>
<td>Seventh</td>
<td>Augmented</td>
<td>A7</td>
<td>C, B# (C)</td>
<td>12</td>
<td>1, #7</td>
</tr>
<tr>
<td>Octave</td>
<td>Diminished</td>
<td>d8</td>
<td>C, Cb (B)</td>
<td>11</td>
<td>1, b8</td>
</tr>
<tr>
<td>Octave</td>
<td>Perfect</td>
<td>P8</td>
<td>C, C</td>
<td>12</td>
<td>1, 8</td>
</tr>
<tr>
<td>Octave</td>
<td>Augmented</td>
<td>A8</td>
<td>C, C#</td>
<td>13</td>
<td>1, #8</td>
</tr>
</tbody>
</table>

Note that intervals are merely the distance between two notes in a scale and, in Table 3, do not depend on the C note being fixed. For example, a minor seventh interval is listed as “C, Bb” but could also be “C#, B”. This is because the
interval between the notes is the same (a seventh) and the distance in semitones is equal (10).

A comparison between Table 2 and Table 3 confirms that each chord type is typically named using the interval between the root note and the highest note in the chord. In cases where this is not true, a major chord is changed slightly to create a different chord type (i.e. a minor chord is a major with a minor third note).

Scale Degree

Scale degrees in Frond are used to reference the root notes of chords relative to the key of the song. These root notes are numbered from one to seven using Roman numerals and ascend with the notes in the scale. Two scales are shown below in Table 4 to illustrate referring to notes in a scale using scale degrees.
Table 4 - Scales and degrees.

<table>
<thead>
<tr>
<th>C Major</th>
<th>F# Minor</th>
<th>Scale Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>F#</td>
<td>I</td>
</tr>
<tr>
<td>D</td>
<td>G#</td>
<td>II</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>III</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>IV</td>
</tr>
<tr>
<td>G</td>
<td>C#</td>
<td>V</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>VI</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>VII</td>
</tr>
</tbody>
</table>

The advantage of defining chords using scale degrees is that a chord becomes independent of the notes in the scale. This allows chords to be easily transposed from one key to another. For example, a V major chord in the key of C major is equivalent to a V major chord in any other major key. These chords can be considered equivalent because the intervals between the notes and the scale degree between the key and root note of the chord are the same. The only difference between the chords are the notes that are played.

Scale degrees in chords are also useful in simplifying combinations of chord progressions. Twelve combinations per chord type exist when defining a chord by all possible root notes. This number can be reduced to seven using scale degrees. The reduction in combinations would benefit chord selection by increasing the probability that preferable chord progressions are chosen due to fewer possibilities being available.
Selecting Progressions

Chords are grouped with other chords to form progressions. Progressions consist of two or more chords played in succession within the song. More complex progressions can be built using smaller progressions composed of pairs of chords. Within Frond, these pairs of chords are selected first by scale degree and then by type. Figure 7 shows a complex chord progression composed of paired chords in the key of E major.

![Figure 7 - Progressions with scale degree and type.](image)

In a chord pair, there is the current chord and the next chord. The current chord could be used to probabilistically select the next chord in a progression. In the case where a current chord does not yet exist, such as the beginning of the song, the I major chord could be chosen as the current chord.

A uniform random number between 0.0 and 1.0 would be chosen to select the scale degree of the next chord from a table of probabilities. The probabilities in the table are summed in sequential order until meeting or exceeding the random number, as illustrated in Figure 8. It is worthwhile to note that the
probabilities can be adjusted using a learning algorithm, such as that described on page 59. Doing so would allow the system to be trained to select progressions preferable to a user’s tastes.

<table>
<thead>
<tr>
<th>Current Scale Degree</th>
<th>Next Scale Degree</th>
<th>Probability</th>
<th>Running Sum of Probabilities</th>
<th>Random Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>I</td>
<td>0.10</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>II</td>
<td>0.25</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>III</td>
<td>0.05</td>
<td>0.40</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>IV</td>
<td>0.10</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>V</td>
<td>0.30</td>
<td>0.80</td>
<td>0.6832</td>
</tr>
<tr>
<td>I</td>
<td>VI</td>
<td>0.15</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>VII</td>
<td>0.05</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8 - Selection of scale degree.

In the example above, the I-V pair is chosen because the random number, 0.6832, is greater than the running sum for IV but less than the running sum for V (0.50 ≤ 0.6832 < 0.80). There are a total of 49 ((# scale degrees)$^2 = 7^2$) pairs that can be selected for all scale degrees.

The chord type of the next chord would be chosen in the same manner as shown in Figure 9. There are a total of 1,156 ((# chord types)$^2 = 34^2$) pairs that can be selected for all chord types.
<table>
<thead>
<tr>
<th>Current Chord Type</th>
<th>Next Chord Type</th>
<th>Probability</th>
<th>Running Sum of Probabilities</th>
<th>Random Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>m</td>
<td>0.20</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>d</td>
<td>0.15</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>d7</td>
<td>0.10</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>d7</td>
<td>m7b5</td>
<td>0.07</td>
<td>0.52</td>
<td></td>
</tr>
<tr>
<td>m7b5</td>
<td>A</td>
<td>0.05</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>0.03</td>
<td>0.61</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0.04</td>
<td>0.65</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>m7</td>
<td>0.03</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>m7</td>
<td>M7</td>
<td>0.04</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>M7</td>
<td>m/Maj7</td>
<td>0.03</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>m/Maj7</td>
<td>sus4</td>
<td>0.02</td>
<td>0.77</td>
<td></td>
</tr>
<tr>
<td>sus4</td>
<td>sus2</td>
<td>0.02</td>
<td>0.79</td>
<td></td>
</tr>
<tr>
<td>sus2</td>
<td>7sus4</td>
<td>0.01</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>7sus4</td>
<td>7sus2</td>
<td>0.01</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>7sus2</td>
<td>add2</td>
<td>0.01</td>
<td>0.82</td>
<td>0.8129</td>
</tr>
<tr>
<td>add2</td>
<td>add9</td>
<td>0.01</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>add9</td>
<td>add4</td>
<td>0.01</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>add4</td>
<td>6</td>
<td>0.01</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>m6</td>
<td>0.01</td>
<td>0.86</td>
<td></td>
</tr>
<tr>
<td>m6</td>
<td>6/9</td>
<td>0.01</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>6/9</td>
<td>9</td>
<td>0.01</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>m9</td>
<td>0.01</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td>m9</td>
<td>M9</td>
<td>0.01</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>M9</td>
<td>11</td>
<td>0.01</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>m11</td>
<td>0.01</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>m11</td>
<td>M11</td>
<td>0.01</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>M11</td>
<td>13</td>
<td>0.01</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>m13</td>
<td>0.01</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>m13</td>
<td>M13</td>
<td>0.01</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>M13</td>
<td>7#9</td>
<td>0.01</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>7#9</td>
<td>7b9</td>
<td>0.01</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>7b9</td>
<td>7#5</td>
<td>0.01</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td>7#5</td>
<td>7b5</td>
<td>0.01</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9 - Selection of chord type.**

Similar to scale degree, the chord type chosen by the random number, 0.8129, is between the running sum for 7sus2, 0.81, and the running sum for
add2, 0.82. Combining the selected scale degree with the chord type yields a new chord of Vadd2. This chord can then be used to determine the next chord in the progression.

The progression that is selected would then used to select the next progression in the song. This process would be repeated until chords have been selected for all patterns within each element. The chords are then used to generate the notes during the music generation process.
Chapter 5
Music Generation

The characteristics and chords of the song are now assembled to create the notes played in the song itself. These pieces are put together by applying them to various steps in the music generation process, which makes use of cellular automata. As Miranda notes, “Since cellular automata produce large amounts of patterned data and if we assume that music composition can be thought of as being based on pattern propagation and the formal manipulation of its parameters... cellular automata could be associated to some sort of music representation in order to generate compositional material.” (Miranda, 2001, p. 124) A specialized adaptation of elementary cellular automata with fracturing rules is used in this process.

Cellular Automata

In Frond, cellular automata (CA) are used to generate the notes that are played in the melody and bass parts of the song. A cellular automaton in its simplest form is a one-dimensional grid of cells that changes state by evaluating its
current state against a set of predefined rules to determine the state of the automaton at the next discrete time step. This type of cellular automaton is referred to as elementary. The state of an elementary cellular automaton is determined by \( n \) number of cells, each of which have a value of either 0 (off) or 1 (on). The elementary cellular automaton shown in Figure 10 illustrates this with an initial state where only one cell is on, as indicated by the shaded portion of the first row in the example. The cells that are off are indicated by their lack of shading. Each row indicates one time step of the cellular automaton.

![Figure 10 - A binary cellular automaton.](image)

A set of rules are assigned that determine the state of the CA at the next time step. A rule notation adopted from Wolfram (Wolfram, 2002, pp. 53-60) is used to identify a specific elementary CA rule and is described on page 33 of this text. For illustrative purposes, the first row is labeled 1 to indicate that this is the first step (initial state) of the automaton while subsequent rows are numbered in sequential order. Columns are ordered A through S which, when combined with row numbers, allows naming specific cells and referencing them to more easily
explain how a CA functions. It is typical, however, to forgo naming cells as referencing a specific cell is generally not necessary. Further examples in this text also refrain from naming cells.

The first cell to be evaluated is J2 as it is immediately underneath the only cell with an on state in the previous time step. The value of J2 is determined by looking at its nearest neighbors from the previous time step which are the three cells to the upper left (I1), immediately above (J1), and upper right (K1) of J2. These cells contain the values 0, 1, and 0 which can be looked up in the rule set to determine that J2 has a value of 1. The same method can be used for cells I2 and K2 to yield values of 1 and 1. Similarly, the rules can be applied to the remaining cells for this step and then re-evaluated for each subsequent step. Figure 11 illustrates ten steps of a rule 30 cellular automaton.
The sections that follow elaborate further on various topics applicable to cellular automata, including initial conditions, rules, dimensions, and size.

**Initial Conditions**

The initial conditions given to a cellular automaton can be critical in determining the behavior of the automaton as a whole. In some cases, initial conditions can dramatically alter the behavior of a CA while in other cases, initial conditions can have very little effect on the it. An example of initial conditions having a large effect on CA behavior is illustrated in the first five steps of rule 118 shown in Figure 12.

![Figure 12 - Initial conditions affecting behavior.](image)

An example of initial conditions having little effect on the overall behavior of a cellular automaton is shown in Figure 13 using rule 160, which exhibits similar behavior given different initial conditions.
Only four cells on

All but four cells on

Figure 13 - Initial conditions having little effect on behavior.

Any set of initial conditions can be used in Frond, which permits evaluating the effects initial conditions have when generating music. An example of drastic effects would be where a song with many different notes instead plays the same note repetitively. An example of minor effects would be a song where the notes in the first few beginning measures change but the rest of the song is largely the same.

Rules

The rules that govern elementary cellular automata are simple and were first described by Wolfram (Wolfram, 2002, pp. 53-60). There are 256 possible sets of rules for Wolfram’s numbering scheme. This numbering scheme was derived from three neighboring cells being considered by the rule: one to the upper left, one above, and one to the upper right. There are two states for each of these cells: either 0 or 1. This yields eight possible configurations that a cell's nearest neighbors can be in (two states ^ three cells = eight configurations). The
resulting value for the cell being evaluated is either 0 or 1, depending on the configuration that applies to that cell. As a result, there are a total of 256 (two resulting values \( \times \) eight configurations) possible rules for Wolfram's numbering scheme. The rules are assigned numbers by treating the outcome of the rule as a base-2 integer. Rules and derivation of their numbers are shown in Figure 14 in increasing sequential order.

<table>
<thead>
<tr>
<th>Rule 0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rule 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Rule 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

... 

| Rule 255 | 128 | 64  | 32  | 16  | 8  | 4  | 2  | 1  |

Figure 14 - Elementary rule numbering.

Wolfram's numbering scheme for elementary cellular automata illustrates the most simplistic rules; however, rules can become as complex as one would like. Rules can be expanded to consider additional cells from previous time steps. With multi-state rules, cells can contain values from 0 to \( n \), where these values are
often mapped to gray scale or color values for visualization purposes. Rules exist that are even more simplistic such as those in Conway's Game of Life, a two-dimensional cellular automaton, which has the following rules (Gardner, 1970):

- Cells with two or three neighbors survive
- Cells with four or more neighbors die
- Cells with less than two neighbors die
- An empty cell with exactly three neighbors births a new cell

An example of Conway’s Game of Life is illustrated on page 43.

While more complex rules exist, simple rules yield surprisingly complex and interesting behavior. Frond uses a variation of Wolfram’s elementary rules based on the concepts of fracturing and active cells.

Fracturing

Fracturing is a term used to describe a cellular automaton which contains the appearance of a crack or fracture running through it (Wolfram, 2002, p. 375). Fracturing makes use of the elementary rule set specified in 0 but with one slight addition: an active cell. The concept of an active cell was adapted from mobile automata where only the active cell is evaluated for each time step rather than
every cell (Wolfram, 2002, pp. 71-77). An example of mobile automaton rules are shown in Figure 15.

![Figure 15 - Example mobile automaton rules.]

Note that the active cell is indicated by an X. In addition, the rules show what position the active cell is in at the next time step. The mobile automaton generated by these rules is illustrated in Figure 16.

![Figure 16 - Example mobile automaton.]

In this example, note that the states of the previous cells continue to propagate through the system until affected by an active cell.
The active cell concept used for fracturing in Frond is slightly different than those found in mobile automata. In Frond, all cells are evaluated for a given step regardless of which cell is the active cell as was the case in previous examples. The active cell is instead used to determine which cell in the automaton is the note that will be played. Figure 17 reproduces the automaton in Figure 16 using fracturing, where all cells are evaluated for each time step as opposed to just the active cell.
The repetitive behavior of the CA is a result of the rule used. Fracturing cellular automata with different behavior are shown throughout the section titled Note Assignment.

Fracturing rules can also be assigned numbers using a scheme similar to Wolfram’s numbering scheme for elementary automata. The number assigned to a fracturing rule is independent of the elementary rule. Like elementary rules, fracturing rules have eight configurations. However, unlike elementary rules which allow each configuration to be in only two states (on or off), fracturing configurations can be in any of three states based on the placement of the active cell. These states are left, middle, or right. The numbering scheme for fracturing rules is thus ternary, allowing for a total of 6,561 possible fracturing rules (three states \(^8\) configurations). Fracturing rules are numbered in the same order as elementary rules as shown in Figure 18.
Behavior

After analyzing the results of all 256 possible rules from his numbering scheme, Wolfram identifies four distinct behaviors exhibited by the one-dimensional elementary cellular automata that use these rules: repetition, nesting, randomness, and localized structures (Wolfram, 2002, p. 52). Repetition exhibits behavior with a simple recurring pattern throughout, such as a checkboard. Nesting shows behavior where a shape contains smaller instances of
the same shape within itself. Randomness is seen in cellular automata where the behavior does not appear to fit a specific pattern. In these cases, the automata typically contains nested shapes of varying sizes at apparently random intervals. Finally, localized structures contain areas that appear random within a mostly repetitive pattern. Each class of behavior is shown in Figure 19, Figure 20, Figure 21, and Figure 22.

Figure 19 - Repetition in rule 246.
Figure 20 - Nesting in rule 18.

Figure 21 - Randomness in rule 86.
These types of behavior are exhibited in simple one-dimensional cellular automata using Wolfram's numbering scheme. Surprisingly, one-dimensional CAs with more complex rules do not necessarily exhibit more complex behavior as Wolfram observed (Wolfram, 2002, p. 62).

The cellular automaton used in Frond can use any of the 256 elementary rules that exhibit the four categories of behavior specified above.

**Dimension**

The dimension of a cellular automaton is defined as the number of axes used to define the state of a CA at a given step. The examples presented thus far are one-dimensional in that each step is defined by a single line of cells. Though
one-dimensional CAs and their steps are represented by two-dimensional grids, the dimension of a CA does not count the steps made as a dimension.

While only one-dimensional cellular automata have been presented thus far, CAs can be two-, three-, or even n-dimensional and similarly, fairly simplistic rules could be used to govern the behavior of these CAs. An example of a two-dimensional CA using Conway’s Game of Life (discussed in the section titled Rules) is shown below in Figure 23.

![Figure 23 - Two dimensional cellular automaton.](image)

One-dimensional elementary cellular automata are used in Frond to emphasize the idea that a simple system can produce complex behavior, such as the patterns found in popular music. One-dimensional space is used to represent notes that can be played while each step in the CA is used to represent time.
Figure 24 illustrates one way that dimension in music can correspond to that in one-dimensional cellular automata.

Size

The size of a cellular automaton is defined as the maximum number of cells each step contains. In the simple examples presented thus far, the size of the CA has been unbounded. A CA with unbounded size evaluates, in effect, an infinite number of cells at each step. Yet this calculation is trivial, as the infinite number of cells being evaluated all have the same state. Cellular automata could also be fixed size where the CA is not allowed to grow beyond its fixed size. A fixed size cellular automaton is shown in Figure 25.
This cellular automata is cyclic in that the cells on the left edge affect those on the right. The state of the CA also repeats every four time steps, which means it has a period of four. The period of a CA is defined as the number of time steps before the system repeats a previous state. The period of a CA also tends to increase with the size.

In Frond, two cyclic cellular automata with size 23 are used for the bass and melody parts. A fixed size bounds the range of notes that can be played during a measure to roughly one and a half octaves above or below the previous note (±11 scale degrees). The cyclic edges create a period, which permits repetitive behavior similar to that found in most music.
Musical Application of Cellular Automata

Wolfram's Principle of Computational Equivalence states "that all processes, whether they are produced by human effort or occur spontaneously in nature, can be viewed as computations." (Wolfram, 2002, p. 715). This broad yet simple principle can be applied to Frond as well. It can be interpreted to mean that popular music, though for the most part created by humans, can be viewed as computations and thus be created by a computer. The idea that simple programs run by a computer can generate complex behavior was also emphasized by Wolfram throughout A New Kind of Science (Wolfram, 2002, p. 4). This idea is central to the design philosophy employed in Frond.

Note Mapping

Note mapping is the term used in Frond to map the state of the cellular automaton to notes that will be played in the song. This process involves creating a profile of the song, deriving notes from each chord in the profile, and then assigning the notes to specific cells in the cellular automaton. Using this method, Frond makes use of two cellular automata; one for the bass part and one for the
melody. However, the implementation of the note mapping method is the same between parts.

Song Profile

A song profile is a tree containing every pattern and measure played in a song organized by element. The song profile can be used to derive the sequencing of every measure in the song. For example, assume the structure of a song consists of some permutation of elements A and B:

- A, B, B, A

Where each element contains some number of patterns, each of which may repeat more than once:

- Element A
  - Pattern x (repeat 2x)
  - Pattern y (repeat 1x)
- Element B
  - Pattern z (repeat 4x)

And each pattern contains some number of measures:

- Pattern x
The definition of elements, patterns, and measures form a tree which can be traversed to derive the sequencing of every measure in the song as shown in Figure 26.

This is done by starting with the first element of the structure: A. The first pattern in A is x which contains measures 1 and 2 that are repeated twice. The second pattern is y, which contains measures 3, 4, and 5. This completes all the
measures in element A. The rest of the structure can be traversed in the same manner to yield the following sequence of measures:

- 1, 2, 1, 2, 3, 4, 5; 6, 6, 6, 6; 6, 6, 6, 6; 1, 2, 1, 2, 3, 4, 5

The sequence of measures is then used to derive the notes for each measure that map to the columns in the cellular automaton.

Note Derivation

Note derivation is a process which combines the notes in a chord with the key of the song to determine which notes are playable. This process effectively creates a new scale that is playable only for the measure that the chord is active. Note derivation is accomplished by taking the notes specified by the key and combining them with those in the chord. The notes in the chord that differ replace those in the key. This is illustrated in Figure 27 with the C major key and F major, C minor, and B major chords.

<table>
<thead>
<tr>
<th>Key (C Major)</th>
<th>C, D, E, F, G, A, B</th>
<th>C, D, E, F, G, A, B</th>
<th>C, D, E, F, G, A, B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chord</td>
<td>F, A, C</td>
<td>C, Eb, G</td>
<td>B, D#, F#</td>
</tr>
<tr>
<td>F Major</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C Minor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B Major</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note Derivation</td>
<td>C, D, E, F, G, A, B</td>
<td>C, D, Eb, F, G, A, B</td>
<td>C, D#, E, F#, G, A, B</td>
</tr>
</tbody>
</table>

Figure 27 - Steps of note derivation.
The notes are derived for every chord in the profile and then assigned to columns in the cellular automaton.

Note Assignment

Note assignment is the final step that occurs in the note mapping process. Using the chord profile and the notes that were derived, note assignment allocates a note to each column in the CA for the duration of each chord.

For example, assume a key of C Major and the following chord profile:

- F Major, C Minor, B Major, C Minor

With a note derivation for each chord:

- F Major: C, D, E, F, G, A, B
- C Minor: C, D, Eb, F, G, A, B
- B Major: C, D#, E, F#, G, A, B

At the start of the song, Fruin assigns the column containing the active cell, identified by the X, to the root note of the first chord in the chord profile. In this case, this is an F note as shown in Figure 28:
The nominal octave for this part is assumed to be 5. Since F major is the first chord in the song, the root note of the chord is centered on the active cell and the nominal octave. Thus, the first note played in the song will be F in the 5th octave (F5). The eleven notes left of F5 decrease in pitch and octave while the eleven to the right increase. The note mapping does not change until the next measure when the next chord is played, so the remaining notes are determined by the position of the active cell.

At the chord change, the note map changes to use the note derivation for the next chord, C minor. The interval measured in semitones is calculated from the last played note, G5, to each note in C minor within one octave above and below the most recent octave, 5. The intervals are shown below in Table 5.
Table 5 - Interval calculations for C minor from G5.

<table>
<thead>
<tr>
<th>Chord</th>
<th>Octave</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>19</td>
</tr>
<tr>
<td>Eb</td>
<td>16</td>
</tr>
<tr>
<td>G</td>
<td>12</td>
</tr>
</tbody>
</table>

The note map is centered on the active cell with note G5 since it has the smallest interval in semitones. The notes that follow are determined by the active cell until the next chord change occurs and a new note map is generated. This process is repeated for all chords in the song. The resulting cellular automaton is shown in Figure 29.
Exception cases can occur in the mapping process depending on the state of the cellular automaton and the previous mappings. One exception case is an out-of-range condition, where the notes played by the cellular automaton are either too low or too high in octave. The other exception case is an intersection between the bass and melody parts. Both of these conditions are checked for each time a chord change occurs.

Out-of-range conditions occur when the most recent note played is less than one octave below or greater than one octave above the key and nominal octave. If an out-of-range condition occurs, the mapping is incremented or decremented by one octave to bring the pitch within range. The nominal octaves
for the bass and melody parts are three and four, respectively. Out-of-range conditions are illustrated in Figure 30 with a key of C Major and nominal octave of four.

In this example, the note mapping would have normally chosen E5 as the note to map to the active cell. However, since E5 is greater than one octave away from the key (C major) and nominal octave (four), the entire note map is decremented one octave to keep in-range.

An intersection occurs when the most recent melody note has a pitch less than or equal to the the pitch of the most recent bass note. If an intersection between bass and melody parts occurs, both bass and melody mappings are incremented or decremented an octave to ensure the intersection is resolved.
Checks for intersections occur each time the note map changes. An example of an intersection is shown in Figure 31.

The intersection above is caused when the bass plays a note with higher pitch (G₄) than the melody note (D₄). Both bass and melody parts are corrected by one octave when the note map changes.

Rhythm

Rhythm in Frond is defined by rules and consists of two parts: timing and duration. Timing is defined as the point in time when a note is struck. Duration is the length of time that a note is held after it has been struck. The rules that determine rhythm are as follows:
• At the start of a measure:
  o Active cell is on: note strike
  o Active cell is off: note strike
• Elsewhere in the measure:
  o Active cell is on: note strike
  o Active cell is off: note hold

Each time step in the cellular automaton determines when a note strike occurs within a measure. In Frond, a measure can be divided into eight, sixteen, thirty-two, or sixty-four equal spans of time allowing for whole, half, and quarter notes and additionally eighth, sixteenth, thirty-second, and sixty-fourth notes, respectively. These spans of time are constant throughout the song. Each time span corresponds to exactly one step in the cellular automaton. Figure 32 illustrates a notional cellular automaton with constant \( \frac{3}{4} \) time spans and a 4/4 timing signature.
The resulting notes with rhythm are shown in Table 6. Timing is expressed in terms of the beat number when the note occurs. Duration is in terms of how many beats occur while the note is held.

Table 6 - Timing and duration of notes.

<table>
<thead>
<tr>
<th>Note</th>
<th>Measure</th>
<th>Timing (beat #)</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>G5</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>G5</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A5</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>B5</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>D#6</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Rests can be added by modifying the rules used for rhythm. For instance, these rules could be modified as follows:

- At the start of a measure:
  - Active cell is on: note strike
  - Active cell is off: rest

- Elsewhere in the measure:
  - Active cell is on: note strike
  - Active cell is off: note hold

Applying the rules above to Figure 32 yields a new table of notes and rhythm shown in Table 7.

<table>
<thead>
<tr>
<th>Note</th>
<th>Measure</th>
<th>Timing (beat #)</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>G5</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>(rest)</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A6</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(rest)</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D#6</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

After a song is generated, the song is then output to a file using MusicXML. This output contains every aspect of the song that was selected by the user, the notes pitches generated during note mapping, and the timing and durations of
each note. Each measure of the song is output according to the profile with repeated measures being output more than once. The MusicXML-compatible file generated by Frond can be read into any number of MusicXML supporting players, such as Finale Reader. MusicXML also allows users to view the notes, measures, dynamics, time signature, key, and chords of songs generated by Frond.
Chapter 6
Learning using a Naïve Bayes Classifier

The quality of songs generated by Frond could, in theory, be improved by using a naïve Bayes classifier. A naïve Bayes classifier could be used in two ways. First, it could determine how good a song is, based on a database of good and bad songs. Second, it could learn how to generate better songs by retaining knowledge of new songs evaluated by the user as being either good or bad. Both of these topics are discussed in further detail below.

Classification

A naïve Bayes classifier searches a set of data to determine the likelihood that an object with specific independent attributes belongs to a specific class within that data. By applying this concept to generating music, two classes are defined: good songs ($C = +$) and bad songs ($C = -$). A song generated by Frond with independent attributes $x_1, x_2, ..., x_n$, could be classified by a naïve Bayes classifier to determine the likelihood that the song is good given its attributes,
\( P(C = + | x_1, ..., x_n) \), based on a set of data that defines both good and bad
songs. Equation 1 defines a naïve Bayes classifier (Russell & Norvig, 2002, p. 718) using terms defined above:

\[
P(C = + | x_1, ..., x_n) = P(C = +) \prod_{i=1}^{n} P(x_i | C = +)
\]

Equation 1 - Naïve Bayes classifier.

As an example, assume the set of data in Table 8 identifies good and bad songs which are composed of three attributes: tempo, a chord progression, and a cellular automata rule:

<table>
<thead>
<tr>
<th>Song Class ((C))</th>
<th>Tempo ((X_1))</th>
<th>Chord Progression ((X_2))</th>
<th>Cellular Automata Rule ((X_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>80</td>
<td>I-IV</td>
<td>90</td>
</tr>
<tr>
<td>Good</td>
<td>80</td>
<td>IV-V</td>
<td>90</td>
</tr>
<tr>
<td>Good</td>
<td>90</td>
<td>I-IV</td>
<td>30</td>
</tr>
<tr>
<td>Bad</td>
<td>230</td>
<td>bIIIV-II</td>
<td>187</td>
</tr>
<tr>
<td>Bad</td>
<td>10</td>
<td>IV-V</td>
<td>193</td>
</tr>
</tbody>
</table>

Now assume a song has been generated with all good attributes as in Table 9 below:

<table>
<thead>
<tr>
<th>Tempo ((X_1))</th>
<th>Chord Progression ((X_2))</th>
<th>Cellular Automata Rule ((X_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>I-IV</td>
<td>90</td>
</tr>
</tbody>
</table>
Using the set of known data, the naïve Bayes classifier defined in Equation 1 can be used to determine the probability that the song is good:

\[
P(C = + | x_1, ..., x_n) = P(C = +) \prod_{i=1}^{3} P(x_i | C = +)
\]

\[= p(C = +) \left[ p(X_1 = 80 | C = +) * \right]
\[= p(C = +) \left[ p(X_2 = I - IV | C = +) * \right]
\[= p(C = +) \left[ p(X_3 = 90 | C = +) * \right]
\]

\[= \frac{3}{5} * \left[ \frac{2}{3} * \frac{2}{3} * \frac{2}{3} \right]
\]

\[= 0.177
\]

Thus, the probability that the song is good (i.e. belongs to class \( C = + \)) given attributes \( X_1 = 80, X_2 = I - IV, \) and \( X_3 = 90 \) is \( 0.177 \), or roughly 17.7%.

While this result may seem low, it is in fact the song with the greatest probability of being good. Other songs composed of only good attributes have probabilities ranging from \( 0.02 \) to \( 0.08 \).

As another example, assume the same data set specified in Table 8. A new song has been created with two good attributes and one attribute that is not good, as defined in Table 10.

<table>
<thead>
<tr>
<th>Tempo ((X_1))</th>
<th>Chord Progression ((X_2))</th>
<th>Cellular Automata Rule ((X_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>I-IV</td>
<td>30</td>
</tr>
</tbody>
</table>
Classifying this new song using Equation 1 yields the following steps:

\[
p(C = +) \left[ p(X_1 = 230 \mid C = +) \cdot p(X_2 = I - IV \mid C = +) \cdot p(X_3 = 30 \mid C = +) \right] \\
= \frac{3}{5} \cdot \left[ \frac{0}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} \right] \\
= 0.0
\]

Herein lies an issue with naïve Bayes classifiers. In the case where an attribute, in this example tempo, has never been classified as good in the data set, the resulting probability is always 0.0. To deal with this, Langley suggests replacing cases where \( p(X_i = x_i \mid C = +) = 0.0 \) with a small constant number (Langley, 2006). For songs, this constant could represent an assumption that, under some rare circumstances, any attribute might be considered good.

Reclassifying the example song above assuming a constant of 0.001 yields a probability of 0.00013. While not a large number, this allows the song to be more appropriately compared to other songs containing attributes not classified as good.

Naïve Bayes classification coupled with Langley’s replacement of zero cases could allow Frond to generate any number of songs at random, classify each, and then present the best song to the user. It can also be used to find the song
with the maximum likelihood of being good by choosing only attributes that each have the highest probability of being good.

**Learning**

Coupling the classification process with an additional learning step could further improve the quality of songs. Learning would entail adding additional data to the set that defines good and bad songs. This data could be captured using songs generated by Frond and feedback from the user that is either simple or detailed.

With simple feedback, a single data set would exist that would consist of all the good and bad songs and their attributes. As a song is generated, the user would be asked whether they liked the song or disliked it. The song and its feedback would be added to the data set and, as a result, be considered when generating future songs.

With detailed feedback, multiple data sets would exist consisting of groups of attributes that make up a song; for instance, song characteristics, chord progressions, and cellular automaton traits. After a song was generated, the user would be asked to evaluate whether they liked or disliked each individual group.
This would allow a user to say, for example, that while they enjoyed the chords and song characteristics, the cellular automaton traits were not enjoyable. The idea is that better results would be obtained by providing more detailed feedback.

Whether using simple or detailed feedback, the end result would be that the user could in theory train Frond to generate music catered to their preferences.
Chapter 7
Conclusions

Statement of the Results

The results are divided into two categories which are discussed in detail in the sections that follow. The first category identifies common behaviors in songs generated by Frond. The second category generates music based on the chords and characteristics of several real-world popular songs.

Behaviors

Frond was used to generate approximately 200 songs with random cellular automata rules to classify the different behaviors exhibited by the songs produced. These behaviors, which are influenced by the note mapping process, can be broken into several categories: repeating note type (whole, half, etc.), trilling, ascending/descending notes, and seemingly random. Each behavior is described in further detail below with figures to illustrate the notes and state of the cellular automaton that causes the behavior.
Repeating note types occur when a note strike occurs at a specific period for the active cell. Figure 33 and Figure 34 illustrate a case where whole notes are repeated.

![Figure 33 - Repeating whole notes.](image)

Repeating note types can also occur with other notes, such as quarter notes, as illustrated in Figure 35 and Figure 36.

![Figure 35 - Repeating quarter notes.](image)
Trilling, which is playing two adjacent notes in rapid succession, behavior becomes apparent with certain rules as well. Trilling is shown in Figure 37 and Figure 38.
Ascending and descending notes occur when the active cell in the cellular automaton moves only in one direction. Ascending and descending notes can be obvious, such as the example shown in Figure 39 and Figure 40.
Ascending and descending notes can also be less obvious, as shown in Figure 41 and Figure 42. Note the second measure where an out-of-range condition occurs.
Figure 42 - Ascending notes in the cellular automaton.

Perhaps the most interesting behavior exhibited is that which appears random. The previously mentioned behaviors have all occurred within cellular automata that were repetitive. Random behavior is seen in cellular automata that are either contain localized structures, exhibit nesting behavior, or exhibit random behavior. An example is shown in Figure 43 and Figure 44.

Figure 43 - Seemingly random behavior.
The behavior of the song is largely dependent on the underlying cellular automaton rules as well as the note mapping process. Trying additional elementary and fracturing rules with the current note mapping process may yield new kinds of behavior not yet discovered. Modifying the note mapping process may also yield new behavior using the same elementary and fracturing rules. The combinations of rules and note maps are practically endless which complicates searching for new behaviors.

Evaluation of Songs

The parameters of three different popular songs were put into Frond and music generated for each. Each song was generated using three approaches:
• *Repeating melody and bass*: Repeating notes in the melody and bass parts. Bass always plays the root note of the chord.


• *Random melody and bass*: Random behavior in both the melody and bass parts.

Each approach is a combination of the random and repeating behaviors described in the previous section. Other behaviors were not identified as they did not produce results that sounded as favorable. The three songs that were evaluated are (Hal Leonard Corporation, 2006):

• Five For Fighting – “Superman (It’s Not Easy)”

• Marc Cohn – “Walking In Memphis”

• Coldplay – “Clocks”

The tempos in beats per minute for each song are 107, 137, and 131 respectively (BPM Database).

While the entire structure of each song was placed into Frond, only the first four measures of each song will be shown as they are representative of the
notes throughout the entire song. Note that there are parts of each song that may not have been accurately represented in Frond. For instance, some songs have multiple chord changes per measure. To deal with this, the chords progressions were kept intact and additional measures were added. Songs may also have elements whose chords are different when repeated later in the song. In this case, the chords from the first instance of the element were maintained. The measures for each song and a brief impression of each is included below.

*Superman (It’s Not Easy)*

- Repeating melody and bass (Figure 45). Very recognizable, good rhythm, sounds pleasant. Very primitive.

- Random melody, repeating bass (Figure 46). Barely recognizable, good rhythm, sounds pleasant. Feels wandering and lost.
Random melody, repeating bass for Superman (It's Not Easy).

- Random melody and bass (Figure 47). Not recognizable, okay rhythm, sounds okay. Feels like both parts wander and can’t figure out which should lead.

Walking In Memphis

- Repeating melody and bass (Figure 48). Hard to recognize (due to two chords per measure in real song), good rhythm, sounds pleasant. Very primitive.
• Random melody, repeating bass (Figure 49). Not recognizable, good rhythm, sounds pleasant. Feels wandering and lost.

![Figure 49 - Random melody, repeating bass for Walking In Memphis.](image)

• Random melody and bass (Figure 50). Not recognizable, okay rhythm, sounds okay. Both parts feel disjoint and wandering.

![Figure 50 - Random melody and bass for Walking In Memphis.](image)

*Clocks*

• Repeating melody and bass (Figure 51). Very recognizable, good rhythm, sounds pleasant. Very primitive.
Random melody, repeating bass (Figure 52). Barely recognizable, good rhythm, sounds pleasant. Melody feels lost and not very strong.

Random melody and bass (Figure 53). Not recognizable, okay rhythm, sounds okay. Both parts feel like they’re from different songs.

All three songs had roughly the same results. Repeating melody and bass is very recognizable and is the most pleasant sounding because its only plays notes that are part of the chords. Random melody and repeating bass is recognizable and doesn’t sound unpleasant, but it definitely doesn’t sound pleasant either.
Random melody and bass isn’t recognizable and tends to sound like two completely different songs.

**Final Conclusion**

The behavior of songs produced by Frond appears limited to only four types: repeating note types, trilling, ascending/descending notes, and seemingly random. Of the four, three produce simple melodies that are very repetitive. In particular, repeating note types in both melody in bass produces recognizable, pleasant songs that are, however, extremely simple and not as representative of popular music. Only one behavior – seemingly random – seems capable of producing songs that resemble those in popular music. However, when seemingly random behavior is introduced into only the melody, the songs become less recognizable and tend to feel lost or wandering. When added to the bass, the songs can not be recognized and the parts feel mostly disjoint. Thus, the conclusion for songs generated by Frond is that while all songs are built using the basic characteristics of popular music, none possess the enjoyment and mass appeal of those composed by humans.

Further exploration in song behavior is necessary to produce music that is more recognizable, more pleasant to listen to, has a well-defined rhythm, and
bears more resemblance to popular music. The next section explores areas left unsolved by the conclusions and makes suggestions for exploration in each area.

**Statements of the Problems Left Unsolved**

Several problems remain unsolved, each of which are posed as questions below. Suggestions for exploring each unsolved problem are also provided.

- Popular music generally plays multiple notes simultaneously within a single part. Could the active cell and note mapping approach be scaled to effectively play multiple notes while remaining simplistic?
  - One approach is to use the cells neighboring the active cell. If a neighboring cell is on, its corresponding note is struck in unison and held for the same duration as the active cell’s note.
- Popular music is composed using more than one instrument. Can this approach be scaled to support multiple instruments?
  - In theory, a cellular automaton could be used for each instrument. Note mapping would probably need to be altered to reflect the characteristics of the specific instrument. For instance, guitar parts tend to consist of chords that are strummed frequently in
repetition. Drum parts do not have notes and instead would have to map to the individual pieces that make up the drum kit.

- **What note mapping scheme is optimal in Frond?**
  - It’s unlikely that the mapping scheme used in Frond, where notes in the map are always a second interval apart, is optimal. Using different intervals such as thirds or fifths may produce better results.

- **What elementary and fracturing rule combinations are optimal in Frond?**
  - Many elementary rules have repetitive behavior that produces correspondingly repetitive musical results. Likewise, many fracturing rules tend to favor one direction or the other which produces music that is always ascending or descending. Better results would likely be produced more consistently if unfavorable rules were not chosen to generate songs.

- **What cellular automaton behavior class is optimal in Frond?**
  - Using a piano its unlikely that the repetitive class of behavior is useful. The other three classes – localized structures, nesting, and random – could be explored further. Different instruments may also work better with different behavior classes. For instance, the repetitive class may work really well for percussive instruments.
• How well does naïve Bayes classification perform?
  
  o It would be interesting to determine how well naïve Bayes classification would work for teaching a system such as Frond what makes a “good” song versus a “bad” song. It would also be interesting to see if, instead of classifying a song as “good” or “bad, songs could be classified as belonging to a particular genre such as “latin” or “not latin”.

• Do elementary rules of the same behavior class produce substantially different results?
  
  o Comparing the results of different elementary rules might reveal that the rules themselves have little effect on the outcome and it is the behavior class that makes the most difference.

  o The path the active cell takes through the cellular automaton and the states it encounters along the way can be thought of as random. One thought would be to try eliminating elementary rules all together and instead randomizing the states of the cellular automaton. Perhaps this would produce results that are just as good or better than those found with elementary rules.
List of References


Appendix

Source Code

Below is the source code for each file that is part of Frond. All source code was written in C# and is compilable under Visual C# 2008 Express Edition.

Program.cs

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Frond
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        /// <STAThread>
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainUserInterface());
        }
    }
}
```

Utilities.cs

```csharp
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Linq;
```
using System.Text;
using System.Reflection;

namespace Frond
{
    public class Utilities
    {
    
        // Can't use generic type constraints on value types,
        // so have to do check like this.
        public static IEnumerable<T> EnumToList<T>()
        {
            Type enumType = typeof(T);
            Array enumValArray = Enum.GetValues(enumType);
            List<T> enumValList = new List<T>(enumValArray.Length);
            foreach (int val in enumValArray)
            {
                enumValList.Add((T)Enum.Parse(enumType, val.ToString()));
            }
            return enumValList;
        }

        public static string GetEnumDescription(Enum value)
        {
            FieldInfo fi = value.GetType().GetField(value.ToString());
            DescriptionAttribute[] attributes = (DescriptionAttribute[])fi.GetCustomAttributes((typeof(DescriptionAttribute), false));
            if ((attributes != null) && (attributes.Length > 0))
            {
                return attributes[0].Description;
            }
            return value.ToString();
        }

        public static int CountCharactersInString(string s, char character)
        {
            int count = 0;
            foreach (char c in s)
            {
                if (c == character)
                {
                    count++;
                }
            }
            return count;
        }
    }
}
MainUserInterface.cs
	namespace Frond
{
    public partial class MainUserInterface : Form
    {
        // Output file constants.
        private const string fileDelimiter = "//";
        private const string outputFileFormat = "{0}" + fileDelimiter + "{1}";
        private const string longOutputFileFormat = "{0}" + fileDelimiter + "{1}" + fileDelimiter + "{2}" + fileDelimiter + "{3}";
        private const string timeBeatsFileToken = "TimeBeats";
        private const string timeTypeFileToken = "TimeType";
        private const string keyRootFileToken = "KeyRoot";
        private const string keyScaleFileToken = "KeyScale";
        private const string tempoBPMFileToken = "TempoBPM";
        private const string stepsPerMeasureFileToken = "StepsPerMeasure";
        private const string melodyElementaryRuleFileToken = "MelodyElementaryRule";
        private const string melodyFracturingRuleFileToken = "MelodyFracturingRule";
        private const string bassElementaryRuleFileToken = "BassElementaryRule";
        private const string bassFracturingRuleFileToken = "BassFracturingRule";
        private const string bassPlayRootFileToken = "BassPlayRoot";
        private const string initialConditionsFileToken = "InitialConditions";
        private const string structureFileToken = "Structure";
        private const string elementFileToken = "Element";

        // Other private data members.
        private Song song = null;
        private const string defaultPath = "C:\Frond";

        // Methods.
        public MainUserInterface()
        {
            InitializeComponent();
        }

        private void MainUserInterface_Load(object sender, EventArgs e)
        {
            // Check the melody and bass part check boxes.
            melodyPartCheckBox.Checked = true;
            bassPartCheckBox.Checked = true;
        }
    }
}
// Populate the beats and beat types combo boxes.
timeBeatsComboBox.Items.Add(Time.DEFAULT_BEAT);
timeBeatsComboBox.SelectedIndex = 0;
timeTypeComboBox.Items.Add(Time.DEFAULT_BEAT_TYPE);
timeTypeComboBox.SelectedIndex = 0;

// Populate the key root and scale combo boxes.
int keyRootSelectedIndex = 0;
foreach (Note.Pitch notePitch in Utilities.EnumToList<Note.Pitch>())
{
  keyRootComboBox.Items.Add(Utilities.GetEnumDescription(notePitch));
  if (notePitch == Key.DEFAULT_ROOT)
  {
    keyRootComboBox.SelectedIndex = keyRootSelectedIndex;
  }
  keyRootSelectedIndex++;

  int keyScaleSelectedIndex = 0;
  foreach (Key.KeyScale keyScale in Utilities.EnumToList<Key.KeyScale>())
  {
    keyScaleComboBox.Items.Add(Utilities.GetEnumDescription(keyScale));
    if (keyScale == Key.DEFAULT_SCALE)
    {
      keyScaleComboBox.SelectedIndex = keyScaleSelectedIndex;
    }
    keyScaleSelectedIndex++;
  }
}

// Populate the tempo text box.
tempoBPMTextBox.Text = Time.DEFAULT_TEMPO.ToString();

// Populate the cellular automaton rule text boxes and check box.
stepsPerMeasureComboBox.Text =
  Song.DEFAULT_STEPS_PER_MEASURE.ToString();
stepsPerMeasureComboBox.SelectedIndex =
  Song.DEFAULT_STEPS_PER_MEASURE.ToString();
melodyElementaryRuleTextBox.Text =
  CellularAutomaton.DEFAULT_ELEMENTARY_RULE.ToString();
melodyFracturingRuleTextBox.Text =
  CellularAutomaton.DEFAULT_FRACTURING_RULE.ToString();
bassElementaryRuleTextBox.Text =
  CellularAutomaton.DEFAULT_ELEMENTARY_RULE.ToString();
bassFracturingRuleTextBox.Text =
  CellularAutomaton.DEFAULT_FRACTURING_RULE.ToString();
bassAlwaysPlayRootCheckBox.Checked = true;

// Populate the initial conditions.
Cell[] cells = CellularAutomaton.DEFAULT_INITIAL_CONDITIONS;
CellularAutomata.CellState[] cellStates = new
  CellularAutomata.CellState[cells.Length];
for (int i = 0; i < cells.Length; i++)
{
  cells[i].State = cellStates[i];
}

initialConditionsControl.Reset(CellularAutomaton.DEFAULT_CA_WIDTH, 1);
initialConditionsControl.SetInitialRow(cellStates);
// Populate the element types.
foreach (Element.ElementType elementType in Utilities.EnumToList<Element.ElementType>())
{
    elementTypesListBox.Items.Add(Utilities.GetEnumDescription(elementType));
}

// Populate the element tree.
foreach (Element.ElementType elementType in Utilities.EnumToList<Element.ElementType>())
{
    TreeNode node = elementsTreeView.Nodes.Add(Utilities.GetEnumDescription(elementType));
    node.ContextMenuStrip = elementContextMenu;
    node.Name = Utilities.GetEnumDescription(elementType);
}

// Populate the pattern context menu with dynamics info.
foreach (Dynamics.DynamicsType dynamicsType in Utilities.EnumToList<Dynamics.DynamicsType>())
{
    string description = Utilities.GetEnumDescription(dynamicsType);
    string notation = Dynamics.LookupDynamicsNotation(dynamicsType);
    ToolStripItem toolStripItem = new ToolStripMenuItem();
    toolStripItem.Text = description + "(" + notation + ")";
    dynamicsMenuItem.DropDownItems.Add(toolStripItem);
}

// Set the current directory and populate the path text box.
if (Directory.Exists(defaultPath) == false)
{
    Directory.CreateDirectory(defaultPath);
}
Directory.SetCurrentDirectory(defaultPath);
pathTextBox.Text = Directory.GetCurrentDirectory();

// Set the current part to melody on the results tab.
partComboBox.SelectedIndex = 0;

private void chordMenuItem_Click(object sender, EventArgs e)
{
    // Display the chord selection dialog box.
    ChordDialog chordDialog = new ChordDialog();
    // Show the dialog to the user.
    DialogResult dialogResult = chordDialog.ShowDialog();
    if (dialogResult == DialogResult.OK)
    {
        // If the user selected OK, get the data from the dialog.
        Chord.ChordType chordType = chordDialog.ChordType;
        Chord.ScaleDegree scaleDegree = chordDialog.ScaleDegree;
        Chord chord = new Chord(chordType, scaleDegree);

        // Put the data into the elements tree.
        TreeNode treeNode = elementsTreeView.SelectedNode;
        char[] delimiters = { '"' };
string[] treeNodeTextTokens =
    treeNode.Text.Split(delimiters, StringSplitOptions.RemoveEmptyEntries);
if (treeNodeTextTokens.Length == 1 ||
    treeNodeTextTokens.Length == 2)
{
    // Remove any trailing spaces.
    treeNode.Text = treeNodeTextTokens[0];
    treeNode.Text = treeNode.Text.Replace(" ", ":");

    // Append the chord string.
    treeNode.Text += " (" + chord.ToString() + ")";

    // Set the name.
    treeNode.Name = treeNode.Text;
}
else
{
    throw new Exception("Error adding chord to element tree.
    Error tokenizing node text.");
}

// Dispose of the form in case the user closed it.
chordDialog.Dispose();

private void generateSongButton_Click(object sender, EventArgs e)
{
    string validationResults = validateInputs();
    // If there are no validation errors, proceed.
    if (validationResults == ":")
    {
        try
        {
            // Initialize the song builder with most of the data.
            SongBuilder songBuilder = new SongBuilder();
            songBuilder.Time =
                Time.FromString(timeBeatsComboBox.Text, timeTypeComboBox.Text,
                    tempoBPMTextBox.Text);
            songBuilder.Key = Key.FromString(keyRootComboBox.Text,
                keyScaleComboBox.Text);
            songBuilder.StepsPerMeasure =
                Convert.ToInt32(stepsPerMeasureComboBox.Text);
            songBuilder.MelodyElementaryRule =
                Convert.ToInt32(melodyElementaryRuleTextBox.Text);
            songBuilder.MelodyFracturingRule =
                Convert.ToInt32(melodyFracturingRuleTextBox.Text);
            songBuilder.BassElementaryRule =
                Convert.ToInt32(bassElementaryRuleTextBox.Text);
            songBuilder.BassFracturingRule =
                Convert.ToInt32(bassFracturingRuleTextBox.Text);
            songBuilder.BassAlwaysPlayRootNote =
                Convert.ToBoolean(bassAlwaysPlayRootCheckBox.Checked);

            // Copy the initial conditions into the song builder.
            CellularAutomata.CellState[] cellStates =
                initialConditionsControl.GetCellStates()[0];
            Cell[] initialConditions = new Cell[cellStates.Length];
            for (int i = 0; i < cellStates.Length; i++)
            {
                initialConditions[i] = new Cell();
                initialConditions[i].State = cellStates[i];
            }
        }
    }
songBuilder.InitialConditions = initialConditions;

// Copy the structure into a string array.
string[] structure = new string[structureListBox.Items.Count];
for (int i = 0; i < structureListBox.Items.Count; i++)
{
    structure[i] = (string) structureListBox.Items[i];
}
songBuilder.Structure = structure;

// Add the elements, patterns, dynamics, measures, and chords.
TreeNodeCollection elementNodes = elementsTreeView.Nodes;
foreach (TreeNode elementNode in elementNodes)
{
    string element = elementNode.Text;
    // Traverse the patterns for this element.
    TreeNodeCollection patternNodes = elementNode.Nodes;
    foreach (TreeNode patternNode in patternNodes)
    {
        char[] separators = new char[] {' ', '(', ')', '[', ']'};
        string[] patternTokens = patternNode.Text.Split(separators, StringSplitOptions.RemoveEmptyEntries);
        string pattern = patternTokens[0];
        Dynamics.DynamicsType dynamicsType = Dynamics.LookupDynamicsType(patternTokens[1]);
        int repeats = Convert.ToInt32(patternTokens[2].Replace("x", "))
        Dynamics.LookupDynamicsType(patternTokens[1]);
        int repeats = Convert.ToInt32(patternTokens[2].Replace("x", "))
        // Add the pattern and dynamics.
        songBuilder.AddPattern(element, pattern, dynamicsType, repeats);
        // Traverse the measures for this pattern.
        TreeNodeCollection measureNodes = patternNode.Nodes;
        foreach (TreeNode measureNode in measureNodes)
        {
            string[] measureTokens = measureNode.Text.Split(separators, StringSplitOptions.RemoveEmptyEntries);
            string measure = measureTokens[0];
            Chord.ScaleDegree scaleDegree = Chord.LookupScaleDegree(measureTokens[1]);
            Chord.ChordType chordType = Chord.LookupChordType(measureTokens[2]);
            Chord chord = new Chord(chordType, scaleDegree);
            // Add the measure and chord.
            songBuilder.AddMeasure(pattern, measure, chord);
        }
    }
}
// Generate the song.
song = songBuilder.GenerateSong();

// Automatically output the song to a MusicXML file.
string songXML = song.ToXML();
saveXMLFile(songXML);

// Populate the results tab with patterns.
List<string> patterns = song.RetrievePatterns();
patternListBox.Hide();
patternListBox.Items.Clear();
foreach (string pattern in patterns)
{
    patternListBox.Items.Add(pattern);
}
patternListBox.Show();
}

private string validateInputs()
{
    string validationResults = "";

    // Validate parts.
    if (melodyPartCheckBox.CheckState == CheckState.Indeterminate)
    {
        validationResults += "Melody part must be checked or unchecked!\n";
    }
    if (bassPartCheckBox.CheckState == CheckState.Indeterminate)
    {
        validationResults += "Bass part must be checked or unchecked!\n";
    }

    // Validate time.
    if (timeBeatsComboBox.SelectedItem == null)
    {
        validationResults += "Time beats must be selected!\n";
    }
    if (timeTypeComboBox.SelectedItem == null)
    {
        validationResults += "Time type must be selected!\n";
    }

    // Validate key.
    if (keyRootComboBox.SelectedItem == null)
    {
        validationResults += "Key root must be selected!\n";
    }
    if (keyScaleComboBox.SelectedItem == null)
{ validationResults += "Key scale must be selected!\n";
}

// Validate tempo.
try
{
    int tempo = Convert.ToInt32(tempoBPMTextBox.Text);
    if (tempo < 32)
    { validationResults += "Tempo must be greater than or
        equal to 32!\n";
    }
    else if (tempo > 192)
    { // Provide a warning but allow the tempo anyway.
        validationResults += "Tempo must be less than or equal
to 192!\n";
    }
    catch (Exception e)
    { validationResults += "Tempo can not be converted to an
        integer!\n";
    }

    // Validate steps per measure.
    try
    {
        int stepsPerMeasure =
            Convert.ToInt32(stepsPerMeasureComboBox.Text);
    }
    catch (Exception e)
    { validationResults += "Steps per measure can not be converted
to an integer!\n";
    }

    // Validate melody rules.
    try
    {
        int melodyElementaryRule =
            Convert.ToInt32(melodyElementaryRuleTextBox.Text);
        if (melodyElementaryRule < 0 || melodyElementaryRule > 255)
        { validationResults += "Melody elementary rule must be
            between 0 and 255!\n";
        }
    }
    catch (Exception e)
    { validationResults += "Melody elementary rule can not be
        converted to an integer!\n";
    }

    try
    {
        int melodyFracturingRule =
            Convert.ToInt32(melodyFracturingRuleTextBox.Text);
        if (melodyFracturingRule < 0 || melodyFracturingRule > 6560)
validationResults += "Melody fracturing rule must be between 0 and 6560!\n";
} catch (Exception e) {
  validationResults += "Melody fracturing rule can not be converted to an integer!\n";
}

// Validate bass rules.
try {
  int bassElementaryRule = Convert.ToInt32(bassElementaryRuleTextBox.Text);
  if (bassElementaryRule < 0 || bassElementaryRule > 255) {
    validationResults += "Bass elementary rule must be between 0 and 255!\n";
  }
} catch (Exception e) {
  validationResults += "Bass elementary rule can not be converted to an integer!\n";
}

try {
  int bassFracturingRule = Convert.ToInt32(bassFracturingRuleTextBox.Text);
  if (bassFracturingRule < 0 || bassFracturingRule > 6560) {
    validationResults += "Bass fracturing rule must be between 0 and 6560!\n";
  }
} catch (Exception e) {
  validationResults += "Bass fracturing rule can not be converted to an integer!\n";
}
if (bassAlwaysPlayRootCheckBox.CheckState == CheckState.Indeterminate) {
  validationResults += "Always play root note must be checked or unchecked!\n";
}

// Validate initial conditions.
CellularAutomata.CellState[][] cellStates = initialConditionsControl.GetCellStates();
int activeCellCount = 0;
for (int i = 0; i < cellStates.Length; i++) {
  for (int j = 0; j < cellStates[i].Length; j++) {
    CellularAutomata.CellState cellState = cellStates[i][j];
    // Count active cells.
{  
    activeCellCount++;  
}  
}  
}  
if (activeCellCount != 1)  
{  
    validationResults += "Initial conditions must contain only one active cell!\n";  
}  
if (cellStates.Length > 1)  
{  
    validationResults += "Initial conditions must be only one row in length!\n";  
}  
// Validate structure.  
if (structureListBox.Items.Count <= 0)  
{  
    validationResults += "Structure must contain at least one element!\n";  
}  
// Validate elements.  
ListBox.ObjectCollection structureList = structureListBox.Items;  
for (int i = 0; i < structureList.Count; i++)  
{  
    string structure = (string) structureList[i];  
    TreeNode[] elementNodes = elementsTreeView.Nodes.Find(structure, false);  
    if (elementNodes.Length == 1)  
    {  
        // See if a pattern exists for this element.  
        TreeNodeCollection patternNodes = elementNodes[0].Nodes;  
        if (patternNodes.Count > 0)  
        {  
            foreach (TreeNode patternNode in patternNodes)  
            {  
                // Check if the pattern has at least one measure.  
                string pattern = patternNode.Text;  
                if (patternNode.Nodes.Count > 0)  
                {  
                    // Check if each measure has a chord.  
                    TreeNodeCollection measureNodes = patternNode.Nodes;  
                    foreach (TreeNode measureNode in measureNodes)  
                    {  
                        string measure = measureNode.Text;  
                        char[] delimiters = { '(' };  
                        string[] measureNodeTokens = measureNode.Text.Split(delimiters, StringSplitOptions.RemoveEmptyEntries);  
                        if (measureNodeTokens.Length == 1)  
                        {  
                            validationResults += "Measure " + measure + " is missing a chord!\n";  
                        }  
                        else if (measureNodeTokens.Length != 2)  
                        {  
                            validationResults += "Measure " + measure + " can not be parsed for a chord!\n";  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
if (patternNodeTokens.Length == 1)
    validationResults += "Pattern '" + pattern + "' is missing dynamics and repeats!
    n";
else if (patternNodeTokens.Length == 2)
    validationResults += "Pattern '" + pattern + "' is missing dynamics or repeats!
    n";
else if (patternNodeTokens.Length != 3)
    validationResults += "Pattern '" + pattern + "' can not be parsed for dynamics and repeats!
    n";
    }
private void structureUpButton_Click(object sender, EventArgs e)
{
    // Determine which item was selected.
    ListBox.SelectedIndexCollection selectedIndices =
    structureListBox.SelectedIndices;
    if (selectedIndices.Count == 1) {
        // Get the index to the item and add it to the structure
        box.
        int index = selectedIndices[0];
        if (index > 0) {
            string elementType =
            (string)structureListBox.Items[index];
            structureListBox.Items.RemoveAt(index);
            structureListBox.Items.Insert(index - 1, elementType);
            structureListBox.SelectedIndex = index - 1;
        }
    }
    else {
        MessageBox.Show("Invalid number of structures selected.
Please select only one.", "Error", MessageBoxButtons.OK);
    }
}

private void structureDownButton_Click(object sender, EventArgs e) {
    // Determine which item was selected.
    ListBox.SelectedIndexCollection selectedIndices =
    structureListBox.SelectedIndices;
    if (selectedIndices.Count == 1) {
        // Get the index to the item and add it to the structure
        box.
        int index = selectedIndices[0];
        if (index < structureListBox.Items.Count - 1) {
            string elementType =
            (string)structureListBox.Items[index];
            structureListBox.Items.RemoveAt(index);
            structureListBox.Items.Insert(index + 1, elementType);
            structureListBox.SelectedIndex = index + 1;
        }
    }
    else {
        MessageBox.Show("Invalid number of structures selected.
Please select only one.", "Error", MessageBoxButtons.OK);
    }
}

private void elementsTreeView_MouseClick(object sender, MouseEventArgs e) {
    // If a user right-clicks on a node, attempt to select it.
    if (e.Button == MouseButtons.Right) {
        elementsTreeView.SelectedNode =
        elementsTreeView.GetNodeAt(e.X, e.Y);
    }
}
private void addPatternMenuItem_Click(object sender, EventArgs e)
{
    // Grab the selected node.
    TreeNode selectedNode = elementsTreeView.SelectedNode;
    
    // Name the new node.
    string newNodeName = selectedNode.Text.Substring(0, 1);
    int count = selectedNode.Nodes.Count + 1;
    newNodeName += count.ToString();
    
    // Add the pattern to the currently selected node.
    TreeNode newNode = selectedNode.Nodes.Add(newNodeName);
    newNode.ContextMenuStrip = patternContextMenu;
    newNode.Name = newNodeName;
    
    // Expand the selected node.
    selectedNode.Expand();
}
private void removePatternMenuItem_Click(object sender, EventArgs e)
{
    // Grab the selected node and store the parent.
    TreeNode selectedNode = elementsTreeView.SelectedNode;
    TreeNode elementNode = selectedNode.Parent;
    
    // Remove the pattern from the tree.
    selectedNode.Remove();
    
    // Rename the remaining nodes and keep their dynamics intact.
    string name = elementNode.Text.Substring(0, 1);
    int count = 1;
    foreach (TreeNode node in elementNode.Nodes)
    {
        char[] separators = new char[] {' ', '(', ')', ' [', ']'};
        string[] nodeTokens = node.Text.Split(separators,
        StringSplitOptions.RemoveEmptyEntries);
        string dynamicsText = ""
        string repeatsText = ""
        if (nodeTokens.Length > 1)
        {
            dynamicsText = nodeTokens[1];
        }
        if (nodeTokens.Length > 2)
        {
            repeatsText = nodeTokens[2];
        }
        node.Text = name + count.ToString();
        if (dynamicsText != "")
        {
            node.Text += " (" + dynamicsText + ")";
        }
        if (repeatsText != "")
        {
            node.Text += " [" + repeatsText + "]";
        }
        count++;
    }
}
private void addMeasureMenuItem_Click(object sender, EventArgs e)
{
    // Grab the selected node.
    TreeNode selectedNode = elementsTreeView.SelectedNode;
// Name the new node.
char[] separators = new char[] { ' ', '(', ')', '[', ']' };
string[] patternTokens = selectedNode.Text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
string newNodeName = patternTokens[0] + "_Measure";
int count = selectedNode.Nodes.Count + 1;
newNodeName += count.ToString();

// Add the pattern to the currently selected node.
TreeNode newNode = selectedNode.Nodes.Add(newNodeName);
newNode.ContextMenuStrip = measureContextMenu;
newNode.Name = newNodeName;

// Expand the selected node.
selectedNode.Expand();
}
private void dynamicsMenuItem_Click(object sender, EventArgs e)
{
    // Parse the text of the menu item that was clicked on.
    ToolStripDropDownItem dropDownItem = (ToolStripDropDownItem)sender;
    char[] separators = new char[] { ' ', '(', ')', '[', ']' };
    string[] dynamicsTokens = dropDownItem.Text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
    string dynamicsText = dynamicsTokens[1];
    string nodeText = selectedNode.Text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
    string[] nodeTokens = nodeText.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
    string repeatsText = ";
    if (nodeTokens.Length == 2)
    {
        if (nodeTokens[1].Contains('x'))
        {
            repeatsText = nodeTokens[1];
        }
    }
    if (nodeTokens.Length == 3)
    {
        repeatsText = nodeTokens[2];
    }
    
    // Update the selected node and update its text with the new
    dynamics.
    selectedNode.Text = nodeText + " (" + dynamicsText + ");
    if (repeatsText !="
    {
        selectedNode.Text += " [" + repeatsText + "])";
    }
    
    // Update the node name.
    selectedNode.Name = selectedNode.Text;
}
private void removeMeasureMenuItem_Click(object sender, EventArgs e)
{
    // Grab the selected node and store the parent.
    TreeNode selectedNode = elementsTreeView.SelectedNode;
    TreeNode patternNode = selectedNode.Parent;
    
    // Remove the measure from the tree.
selectedNode.Remove();

// Rename the remaining nodes and keep their chords intact.
char[] separators = new char[] { ' ', '(', ')' };
string[] patternTokens = patternNode.Text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
string name = patternTokens[0] + "_Measure";
int count = 1;
foreach (TreeNode node in patternNode.Nodes)
{
    string[] nodeTokens = node.Text.Split(separators,
    StringSplitOptions.RemoveEmptyEntries);
    string scaleDegreeText = "";
    string chordTypeText = "";
    if (nodeTokens.Length > 2)
    {
        scaleDegreeText = nodeTokens[1];
        chordTypeText = nodeTokens[2];
    }
    node.Text = name + count.ToString();
    if (scaleDegreeText != "" && chordTypeText != "")
    {
        node.Text += " (" + scaleDegreeText + " " +
        chordTypeText + ")";
    }
    count++;
}

private void loadSetupButton_Click(object sender, EventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Frond setup files (*.frond)|*.frond|All
    files (*.*)|*.*";
    openFileDialog.DefaultExt = "frond";
    openFileDialog.AddExtension = true;
    openFileDialog.Multiselect = false;
    openFileDialog.InitialDirectory = defaultPath;
    DialogResult dialogResult = openFileDialog.ShowDialog();
    if (dialogResult == DialogResult.OK)
    {
        string filename = openFileDialog.FileName;
        loadFile(filename);
    }
}

private void loadFile(string filename)
{
    // Open the file.
    using (TextReader tr = new StreamReader(filename))
    {
        string[] fileDelimiters = new string[] { fileDelimiter };

        // Clear our structures.
        structureListBox.Items.Clear();
        elementsTreeView.Nodes.Clear();

        // Read the line and get the tokens.
        string line = tr.ReadLine();
        while (line != null)
        {
            string[] tokens = line.Split(fileDelimiters,
            StringSplitOptions.RemoveEmptyEntries);
        }
    }
}
// Parse the file.
if (tokens.Length >= 0)
{
    string inputType = tokens[0];
    // Switch on the type of the line read in.
    if (inputType == timeBeatsFileToken)
    {
        if (tokens.Length == 2)
        {
            timeBeatsComboBox.Text = tokens[1];
            timeBeatsComboBox.SelectedItem = tokens[1];
        }
        else
        {
            throw new Exception("Error reading file on line " + line + ".");
        }
    }
    else if (inputType == timeTypeFileToken)
    {
        if (tokens.Length == 2)
        {
            timeTypeComboBox.Text = tokens[1];
            timeTypeComboBox.SelectedItem = tokens[1];
        }
        else
        {
            throw new Exception("Error reading file on line " + line + ".");
        }
    }
    else if (inputType == keyRootFileToken)
    {
        if (tokens.Length == 2)
        {
            keyRootComboBox.Text = tokens[1];
            keyRootComboBox.SelectedItem = tokens[1];
        }
        else
        {
            throw new Exception("Error reading file on line " + line + ".");
        }
    }
    else if (inputType == keyScaleFileToken)
    {
        if (tokens.Length == 2)
        {
            keyScaleComboBox.Text = tokens[1];
            keyScaleComboBox.SelectedItem = tokens[1];
        }
        else
        {
            throw new Exception("Error reading file on line " + line + ".");
        }
    }
    else if (inputType == tempoBPMFileToken)
    {
        if (tokens.Length == 2)
        {
            tempoBPMTextBox.Text = tokens[1];
        }
    }
}
else if (inputType == stepsPerMeasureToken)
{
    if (tokens.Length == 2)
    {
        stepsPerMeasureComboBox.Text = tokens[1];
    }
    else
    {
        throw new Exception("Error reading file on line "+ line + ".");
    }
}
else if (inputType == melodyElementaryRuleFileToken)
{
    if (tokens.Length == 2)
    {
        melodyElementaryRuleTextBox.Text = tokens[1];
    }
    else
    {
        throw new Exception("Error reading file on line "+ line + ".");
    }
}
else if (inputType == melodyFracturingRuleFileToken)
{
    if (tokens.Length == 2)
    {
        melodyFracturingRuleTextBox.Text = tokens[1];
    }
    else
    {
        throw new Exception("Error reading file on line "+ line + ".");
    }
}
else if (inputType == bassElementaryRuleFileToken)
{
    if (tokens.Length == 2)
    {
        bassElementaryRuleTextBox.Text = tokens[1];
    }
    else
    {
        throw new Exception("Error reading file on line "+ line + ".");
    }
}
else if (inputType == bassFracturingRuleFileToken)
{
    if (tokens.Length == 2)
    {
        bassFracturingRuleTextBox.Text = tokens[1];
    }
    else
    {
        throw new Exception("Error reading file on line "+ line + ".");
    }
}
else
    {
        throw new Exception("Error reading file on line " + line + ".");
    }
}  
else if (inputType == bassPlayRootFileToken)
{
    if (tokens.Length == 2)
    {
        bassAlwaysPlayRootCheckBox.Checked = Convert.ToBoolean(tokens[1]);
    }
    else
    {
        throw new Exception("Error reading file on line " + line + ".");
    }
}  
else if (inputType == initialConditionsFileToken)
{
    if (tokens.Length == 4)
    {
        int row = Convert.ToInt32(tokens[1]);
        int column = Convert.ToInt32(tokens[2]);

        // Determine what the cell state is.

        // Set the cell state.
        initialConditionsControl.SetCellState(row, column, cellState);
    }
    else
    {
        throw new Exception("Error reading file on line " + line + ".");
    }
}  
else if (inputType == structureFileToken)
{
    if (tokens.Length == 2)
    {
        structureListBox.Items.Add(tokens[1]);
    }
    else
    {
        throw new Exception("Error reading file on line " + line + ".");
    }
}  
else if (inputType == elementFileToken)
{
    if (tokens.Length == 4)
    {
        // If necessary, readd the base elements to the tree.
        // Populate the element tree.
        foreach (Element.ElementType elementType in Utilities.EnumToList<Element.ElementType>())
        {
            // See if this element already exists.
Utilities.GetEnumDescription(elementType);

string element =

TreeNode[] existingElementNodes =
elementsTreeView.Nodes.Find(element, false);
if (existingElementNodes.Length == 0)
{
    // The element doesn't exist, so add
    TreeNode node =
        elementsTreeView.Nodes.Add(element);
    node.ContextMenuStrip =
        elementContextMenu;
    node.Name = element;
}
else if (existingElementNodes.Length == 1)
{
    // Do nothing, the element already
    exists.
}
else
{
    throw new Exception("Multiple
    element nodes found when parsing input file.");
}

// Now add the new node(s).
TreeNode[] elementNodes =
elementsTreeView.Nodes.Find(tokens[1], false);
if (elementNodes.Length == 1)
{
    // See if a pattern exists for this
    element.
    TreeNode[] patternNodes =
elementNodes[0].Nodes.Find(tokens[2], false);
    if (patternNodes.Length == 0)
    {
        // The pattern doesn't exist yet, so
        add it.
        elementNodes[0].Nodes.Add(tokens[2]);
        patternContextMenu;
        patternNode.Name = tokens[2];
        patternNode.ContextMenuStrip =
            patternContextMenu;
        patternNode.Nodes.Add(tokens[3]);
        measureContextMenu;
    }
    else if (patternNodes.Length == 1)
    {
        // The pattern exists, add the
        measure.
        patternNodes[0].Nodes.Add(tokens[3]);
        measureContextMenu;
    }
    else
    {

throw new Exception("Error parsing pattern on line '" + line + "'.");
}
else{
throw new Exception("Error parsing element on line '" + line + "'.");
}
}
else{
throw new Exception("Error reading file on line '" + line + "'.");
}
else{
throw new Exception("Invalid input token '" + inputType + "'.");
}
// Read the next line of the input file.
line = tr.ReadLine();

private void saveSetupButton_Click(object sender, EventArgs e)
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    saveFileDialog.Filter = "Frond setup files (*.frond)|*.frond|All files (*.*)|*.*";
    saveFileDialog.DefaultExt = "frond";
    saveFileDialog.AddExtension = true;
    saveFileDialog.InitialDirectory = defaultPath;
    DialogResult dialogResult = saveFileDialog.ShowDialog();
    if (dialogResult == DialogResult.OK)
    {
        string filename = saveFileDialog.FileName;
        saveFile(filename);
    }
}
private void saveFile(string filename)
{
    // Open the file.
    using (TextWriter tw = new StreamWriter(filename))
    {
        // Output time.
        tw.WriteLine(outputFileFormat, timeBeatsFileToken,
        timeBeatsComboBox.Text);
        tw.WriteLine(outputFileFormat, timeTypeFileToken,
        timeTypeComboBox.Text);
        // Output key.
        tw.WriteLine(outputFileFormat, keyRootFileToken,
        keyRootComboBox.Text);
        tw.WriteLine(outputFileFormat, keyScaleFileToken,
        keyScaleComboBox.Text);
        // Output tempo.
tw.WriteLine(outputFileFormat, tempoBPMFileToken, tempoBPMTextBox.Text);

// Output CA rules.
tw.WriteLine(outputFileFormat, stepsPerMeasureToken, stepsPerMeasureComboBox.Text);
tw.WriteLine(outputFileFormat, melodyElementaryRuleFileToken, melodyElementaryRuleTextBox.Text);
tw.WriteLine(outputFileFormat, melodyFracturingRuleFileToken, melodyFracturingRuleTextBox.Text);
tw.WriteLine(outputFileFormat, bassElementaryRuleFileToken, bassElementaryRuleTextBox.Text);
tw.WriteLine(outputFileFormat, bassFracturingRuleFileToken, bassFracturingRuleTextBox.Text);
tw.WriteLine(outputFileFormat, bassAlwaysPlayRootFileToken, bassAlwaysPlayRootCheckBox.Checked);

// Output initial conditions.
CellularAutomata.CellState[,] cellStates = initialConditionsControl.GetCellStates();
for (int i = 0; i < cellStates.Length; i++)
{
    for (int j = 0; j < cellStates[i].Length; j++)
    {
        tw.WriteLine(outputFileFormat, initialConditionsFileToken, i, j, cellStates[i][j]);
    }
}

// Output structure.
ListBox.ObjectCollection structureList = structureListBox.Items;
for (int i = 0; i < structureList.Count; i++)
{
    tw.WriteLine(outputFileFormat, structureFileToken, (string)structureList[i]);
}

// Output elements.
TreeNodeCollection treeNodes = elementsTreeView.Nodes;
foreach (TreeNode elementNode in treeNodes)
{
    // Output patterns.
    TreeNode patternNode = elementNode.FirstNode;
    while (patternNode != null)
    {
        // Output measures.
        TreeNode measureNode = patternNode.FirstNode;
        while (measureNode != null)
        {
            tw.WriteLine(outputFileFormat, elementFileToken, elementNode.Text, patternNode.Text, measureNode.Text);

            // Get the next measure.
            measureNode = measureNode.NextNode;
        }

        // Get the next pattern.
        patternNode = patternNode.NextNode;
    }
}
// Close the file.
tw.Close();
private void saveXMLFile(string songXML)
{
    // Grab the path and determine the filename.
    string path = pathTextBox.Text;
    string filename = "song0.xml";
    for (int i = 0; File.Exists(path + "\" + filename) == true; i++)
    {
        filename = "song" + i + ".xml";
    }

    // Output the file.
    string fullName = path + "\" + filename;
    using (StreamWriter sw = new StreamWriter(fullName))
    {
        sw.Write(songXML);
        sw.Close();
    }

    // Update the filename text box.
    filenameTextBox.Text = filename;
}

private void patternListBox_SelectedIndexChanged(object sender, EventArgs e)
{
    ListBox.SelectedItems = patternListBox.SelectedItems;
    if (selectedItems.Count == 1)
    {
        // Retrieve the cells for the selected item.
        string part = partComboBox.Text;
        string pattern = (string)selectedItems[0];
        // Start updating the CA control.
        UpdateCellularAutomatonControl(part, pattern);
    }
    else if (selectedItems.Count == 0)
    {
        MessageBox.Show("Only one pattern can be selected at a time.", "Error", MessageBoxButtons.OK);
    }
    else
    {
        MessageBox.Show("Only one pattern can be selected at a time.", "Error", MessageBoxButtons.OK);
    }
}

private void partComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    ListBox.SelectedItems = patternListBox.SelectedItems;
    if (selectedItems.Count == 1)
    {
        // Retrieve the cells for the selected item.
        string part = partComboBox.Text;
        string pattern = (string)selectedItems[0];
        // Start updating the CA control.
        UpdateCellularAutomatonControl(part, pattern);
    }
    else
    {
        // If no patterns are selected, do nothing.
    }
}
private void UpdateCellularAutomatonControl(string part, string pattern)
{
    // Lock incase a pattern is selected before the previous one has loaded.
    lock (cellularAutomatonControl)
    {
        List<Cell[]> cells = song.RetrieveCells(part, pattern);
        List<string> measures = song.RetrieveMeasures(pattern);
        int stepsPerMeasure = song.StepsPerMeasure;
        int caLength = cells.Count + measures.Count;
        int numHeadersAdded = 0;

        // Populate the cellular automaton control.
        cellularAutomatonControl.Hide();
        if (cells.Count > 0)
        {
            cellularAutomatonControl.Reset(CellularAutomaton.DEFAULT_CA_WIDTH, caLength);
            for (int i = 0; i < cells.Count; i++)
            {
                // If we're starting a new measure, print the header for that measure.
                if (i % stepsPerMeasure == 0)
                {
                    int measureIndex = (int) (i / stepsPerMeasure);
                    string measure = measures[measureIndex];
                    string[] measureHeader = song.RetrieveMeasureHeader(part, measure);
                    if (measureHeader.Length == cells[i].Length)
                    {
                        for (int j = 0; j < cells[i].Length; j++)
                        {
                            cellularAutomatonControl.SetCellHeader(i + numHeadersAdded, j, measureHeader[j]);
                        }
                        numHeadersAdded++;
                    }
                    else
                    {
                        string message = "Measure '" + measure + "' header differs in size '";
                        message += measureHeader.Length + '" from CA width '";
                        message += cells[i].Length + '".';
                        throw new Exception(message);
                    }
                }
            }

            // Set the cellstate for this row.
            for (int j = 0; j < cells[i].Length; j++)
            {
                cellularAutomatonControl.SetCellState(i + numHeadersAdded, j, cells[i][j].State);
            }
        }
    }
}
else {
    throw new Exception("Cell retrieval returned no cells!");
}
cellularAutomatonControl.Show();
}

private void repeatsStripMenuItem_Click(object sender, EventArgs e) {
    ToolStripDropDownItem dropDownItem = (ToolStripDropDownItem)sender;
    string repeatsText = dropDownItem.Text + "x";
    // Parse the text of the selected node to get the pattern text.
    char[] separators = new char[] { '[', ','] ;
    TreeNode selectedNode = elementsTreeView.SelectedNode;
    string[] nodeTokens = selectedNode.Text.Split(separators, StringSplitOptions.RemoveEmptyEntries);
    string nodeText = nodeTokens[0];
    // Update the selected node and update its text with the new dynamics.
    selectedNode.Text = nodeText;
    if (nodeText[nodeText.Length - 1] == ' ') {
        selectedNode.Text += "[" + repeatsText + "]";
    } else {
        selectedNode.Text += " [" + repeatsText + "]";
    }
    // Update the node name.
    selectedNode.Name = selectedNode.Text;
}

private void openInFinaleButton_Click(object sender, EventArgs e) {
    string filename = pathTextBox.Text + "\" + filenameTextBox.Text;
    ProcessStartInfo startInfo = new ProcessStartInfo();
    startInfo.Arguments = filename;
    startInfo.WorkingDirectory = "C:\Program Files\Finale Reader";
    startInfo.FileName = "Finale Reader.exe";
    Process finaleProcess = new Process();
    finaleProcess.StartInfo = startInfo;
    finaleProcess.Start();
}

private void elementTypesRemoveButton_Click(object sender, EventArgs e) {
    // Determine which item was selected.
    ListBox.SelectedIndexCollection selectedIndices = structureListBox.SelectedItems;
    if (selectedIndices.Count == 1) {
        // Get the index to the item and add it to the structure box.
        int index = selectedIndices[0];
    }
structureListBox.Items.RemoveAt(index);

else
{
    MessageBox.Show("Invalid number of elements selected for removal. Please select only one.", "Error", MessageBoxButtons.OK);
}

private void melodyPartRandomButton_Click(object sender, EventArgs e)
{
    melodyElementaryRuleTextBox.Text = (new Random()).Next(255).ToString();
    melodyFracturingRuleTextBox.Text = (new Random()).Next(6560).ToString();
}

private void bassPartRandomButton_Click(object sender, EventArgs e)
{
    bassElementaryRuleTextBox.Text = (new Random()).Next(255).ToString();
    bassFracturingRuleTextBox.Text = (new Random()).Next(6560).ToString();

MainUserInterface.Designer.cs

namespace Frond
{
    partial class MainUserInterface
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

    }
}
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.measureContextMenu = new
        System.Windows.Forms.ContextMenuStrip(this.components);
    this.removeMeasureMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.measureSeparator = new
        System.Windows.Forms.ToolStripSeparator();
    this.chordMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.patternContextMenu = new
        System.Windows.Forms.ContextMenuStrip(this.components);
    this.addMeasureMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.removePatternMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.patternSeparator = new
        System.Windows.Forms.ToolStripSeparator();
    this.dynamicsMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeatsToolStripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats1StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats2StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats3StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats4StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats5StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats6StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats7StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.repeats8StripMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.elementContextMenu = new
        System.Windows.Forms.ContextMenuStrip(this.components);
    this.addPatternMenuItem = new
        System.Windows.Forms.ToolStripMenuItem();
    this.keyGroupBox = new System.Windows.Forms.GroupBox();
    this.keyScaleComboBox = new System.Windows.Forms.ComboBox();
    this.keyScaleLabel = new System.Windows.Forms.Label();
    this.keyRootComboBox = new System.Windows.Forms.ComboBox();
    this.keyRootLabel = new System.Windows.Forms.Label();
    this.songCharacteristicsGroupBox = new
        System.Windows.Forms.GroupBox();
    this.tempoGroupBox = new System.Windows.Forms.GroupBox();
    this.bpmLabel = new System.Windows.Forms.Label();
    this.tempoBPMTextBox = new System.Windows.Forms.TextBox();
    this.timeSignatureGroupBox = new
        System.Windows.Forms.GroupBox();
    this.timeTypeComboBox = new System.Windows.Forms.ComboBox();
    this.beatTypeLabel = new System.Windows.Forms.Label();
    this.beat1Label = new System.Windows.Forms.Label();
    this.beat2Label = new System.Windows.Forms.Label();
    this.beat3Label = new System.Windows.Forms.Label();
    this.beat4Label = new System.Windows.Forms.Label();
    this.beat5Label = new System.Windows.Forms.Label();
    this.beat6Label = new System.Windows.Forms.Label();
    this.beat7Label = new System.Windows.Forms.Label();
    this.beat8Label = new System.Windows.Forms.Label();
    this.beat9Label = new System.Windows.Forms.Label();
    this.beat10Label = new System.Windows.Forms.Label();
    this.beat11Label = new System.Windows.Forms.Label();
    this.beat12Label = new System.Windows.Forms.Label();
    this.beat13Label = new System.Windows.Forms.Label();
    this.beat14Label = new System.Windows.Forms.Label();
    this.timeBeatsComboBox = new System.Windows.Forms.ComboBox();
    this.timeBeatsLabel = new System.Windows.Forms.Label();
}
112

```csharp
this.measureContextMenu.SuspendLayout();
this.patternContextMenu.SuspendLayout();
this.elementContextMenu.SuspendLayout();
this.keyGroupBox.SuspendLayout();
this.songCharacteristicsGroupBox.SuspendLayout();
this.tempoGroupBox.SuspendLayout();
this.timeSignatureGroupBox.SuspendLayout();
this.partsGroupBox.SuspendLayout();
this.structureGroupBox.SuspendLayout();
this.caResultsGroupBox.SuspendLayout();
this.cellularAutomatonGroupBox.SuspendLayout();
this.tabControl1.SuspendLayout();
this.setupPage.SuspendLayout();
this.elementsGroupBox.SuspendLayout();
this.initialConditionsGroupBox.SuspendLayout();
this.melodyPage.SuspendLayout();
this.patternsGroupBox.SuspendLayout();
this.groupBox1.SuspendLayout();
this.resultsGroupbox.SuspendLayout();
this.SuspendLayout();

// measureContextMenu

this.measureContextMenu.Items.AddRange(new
System.Windows.Forms.ToolStripItem[]{
  this.removeMeasureMenuItem,
  this.measureSeparator,
  this.chordMenuItem});
this.measureContextMenu.Name = "measureContextMenu";
this.measureContextMenu.Size = new System.Drawing.Size(166, 54);
// removeMeasureMenuItem
  this.removeMeasureMenuItem.Name = "removeMeasureMenuItem";
  this.removeMeasureMenuItem.Size = new System.Drawing.Size(165, 22);
  this.removeMeasureMenuItem.Text = "&Remove Measure";
  this.removeMeasureMenuItem.Click += new
System.EventHandler(this.removeMeasureMenuItem_Click);
// measureSeparator
  this.measureSeparator.Name = "measureSeparator";
  this.measureSeparator.Size = new System.Drawing.Size(162, 6);
// chordMenuItem
  this.chordMenuItem.Name = "chordMenuItem";
  this.chordMenuItem.Size = new System.Drawing.Size(165, 22);
  this.chordMenuItem.Text = "&Select Chord...";
  this.chordMenuItem.Click += new
System.EventHandler(this.chordMenuItem_Click);

// patternContextMenu

this.patternContextMenu.Items.AddRange(new
System.Windows.Forms.ToolStripItem[]{
  this.addMeasureMenuItem,
  this.removePatternMenuItem,
  this.patternSeparator,
  this.dynamicsMenuItem,
  this.repeatsToolStripMenuItem});
this.patternContextMenu.Name = "elementContextMenu";
this.patternContextMenu.Size = new System.Drawing.Size(166, 54);
```

this.patternContextMenu.Size = new System.Drawing.Size(159, 98);
//
// addMeasureMenuItem
//
this.addMeasureMenuItem.Name = "addMeasureMenuItem";
this.addMeasureMenuItem.Size = new System.Drawing.Size(158, 22);
this.addMeasureMenuItem.Text = "&Add Measure";
this.addMeasureMenuItem.Click += new System.EventHandler(this.addMeasureMenuItem_Click);

//
// removePatternMenuItem
//
this.removePatternMenuItem.Name = "removePatternMenuItem";
this.removePatternMenuItem.Size = new System.Drawing.Size(158, 22);
this.removePatternMenuItem.Text = "Remove &Pattern";
this.removePatternMenuItem.Click += new System.EventHandler(this.removePatternMenuItem_Click);

//
// patternSeparator
//
this.patternSeparator.Name = "patternSeparator";
this.patternSeparator.Size = new System.Drawing.Size(155, 6);
//
// dynamicsMenuItem
//
this.dynamicsMenuItem.Name = "dynamicsMenuItem";
this.dynamicsMenuItem.Size = new System.Drawing.Size(158, 22);
this.dynamicsMenuItem.Text = "&Dynamics";
//
// repeatsToolStripMenuItem
//
this.repeatsToolStripMenuItem.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[] {
    this.repeats1StripMenuItem,
    this.repeats2StripMenuItem,
    this.repeats3StripMenuItem,
    this.repeats4StripMenuItem,
    this.repeats5StripMenuItem,
    this.repeats6StripMenuItem,
    this.repeats7StripMenuItem,
    this.repeats8StripMenuItem});
this.repeatsToolStripMenuItem.Name = "repeatsToolStripMenuItem";
this.repeatsToolStripMenuItem.Size = new System.Drawing.Size(158, 22);
this.repeatsToolStripMenuItem.Text = "&Repeats";
//
// repeats1StripMenuItem
//
this.repeats1StripMenuItem.Name = "repeats1StripMenuItem";
this.repeats1StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats1StripMenuItem.Text = "1";
this.repeats1StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats2StripMenuItem
//
this.repeats2StripMenuItem.Name = "repeats2StripMenuItem";
this.repeats2StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats2StripMenuItem.Text = "2";
this.repeats2StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats3StripMenuItem
//
this.repeats3StripMenuItem.Name = "repeats3StripMenuItem";
this.repeats3StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats3StripMenuItem.Text = "3";
this.repeats3StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats4StripMenuItem
//
this.repeats4StripMenuItem.Name = "repeats4StripMenuItem";
this.repeats4StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats4StripMenuItem.Text = "4";
this.repeats4StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats5StripMenuItem
//
this.repeats5StripMenuItem.Name = "repeats5StripMenuItem";
this.repeats5StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats5StripMenuItem.Text = "5";
this.repeats5StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats6StripMenuItem
//
this.repeats6StripMenuItem.Name = "repeats6StripMenuItem";
this.repeats6StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats6StripMenuItem.Text = "6";
this.repeats6StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats7StripMenuItem
//
this.repeats7StripMenuItem.Name = "repeats7StripMenuItem";
this.repeats7StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats7StripMenuItem.Text = "7";
this.repeats7StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// repeats8StripMenuItem
//
this.repeats8StripMenuItem.Name = "repeats8StripMenuItem";
this.repeats8StripMenuItem.Size = new System.Drawing.Size(80, 22);
this.repeats8StripMenuItem.Text = "8";
this.repeats8StripMenuItem.Click += new System.EventHandler(this.repeatsStripMenuItem_Click);
//
// elementContextMenu
//
this.elementContextMenu.Items.AddRange(new System.Windows.Forms.ToolStripItem[] { this.addPatternMenuItem });
this.elementContextMenu.Name = "elementContextMenu";
this.elementContextMenu.Size = new System.Drawing.Size(138, 26);
    //
    // addPatternMenuItem
    //
    this.addPatternMenuItem.Name = "addPatternMenuItem";
    this.addPatternMenuItem.Size = new System.Drawing.Size(137, 22);
    this.addPatternMenuItem.Text = "&Add Pattern";
    this.addPatternMenuItem.Click += new System.EventHandler(this.addPatternMenuItem_Click);
    //
    // keyGroupBox
    //
    this.keyGroupBox.Controls.Add(this.keyScaleComboBox);
    this.keyGroupBox.Controls.Add(this.keyScaleLabel);
    this.keyGroupBox.Controls.Add(this.keyRootComboBox);
    this.keyGroupBox.Controls.Add(this.keyRootLabel);
    this.keyGroupBox.Location = new System.Drawing.Point(6, 102);
    this.keyGroupBox.Name = "keyGroupBox";
    this.keyGroupBox.Size = new System.Drawing.Size(228, 77);
    this.keyGroupBox.TabIndex = 0;
    this.keyGroupBox.TabStop = false;
    this.keyGroupBox.Text = "Key";
    //
    // keyScaleComboBox
    //
    this.keyScaleComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
    this.keyScaleComboBox.FormattingEnabled = true;
    this.keyScaleComboBox.Location = new System.Drawing.Point(43, 43);
    this.keyScaleComboBox.Name = "keyScaleComboBox";
    this.keyScaleComboBox.Size = new System.Drawing.Size(87, 21);
    this.keyScaleComboBox.TabIndex = 3;
    //
    // keyScaleLabel
    //
    this.keyScaleLabel.AutoSize = true;
    this.keyScaleLabel.Location = new System.Drawing.Point(6, 46);
    this.keyScaleLabel.Name = "keyScaleLabel";
    this.keyScaleLabel.Size = new System.Drawing.Size(37, 13);
    this.keyScaleLabel.TabIndex = 2;
    this.keyScaleLabel.Text = "Scale:";
    //
    // keyRootComboBox
    //
    this.keyRootComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
    this.keyRootComboBox_FormattingEnabled = true;
    this.keyRootComboBox.Location = new System.Drawing.Point(43, 16);
    this.keyRootComboBox.Name = "keyRootComboBox";
    this.keyRootComboBox.Size = new System.Drawing.Size(63, 21);
    this.keyRootComboBox.TabIndex = 1;
    //
    // keyRootLabel
    //
    this.keyRootLabel.AutoSize = true;
    this.keyRootLabel.Location = new System.Drawing.Point(10, 19);
    this.keyRootLabel.Name = "keyRootLabel";
    this.keyRootLabel.Size = new System.Drawing.Size(33, 13);
    this.keyRootLabel.TabIndex = 0;
    this.keyRootLabel.Text = "Root:";
// songCharacteristicsGroupBox
//
this.songCharacteristicsGroupBox.Controls.Add(this.tempoGroupBox);
this.songCharacteristicsGroupBox.Controls.Add(this.timeSignatureGroupBox);
this.songCharacteristicsGroupBox.Controls.Add(this.partsGroupBox);
this.songCharacteristicsGroupBox.Controls.Add(this.keyGroupBox);
this.songCharacteristicsGroupBox.Location = new System.Drawing.Point(6, 6);
this.songCharacteristicsGroupBox.Name = "songCharacteristicsGroupBox";
this.songCharacteristicsGroupBox.Size = new System.Drawing.Size(241, 242);
this.songCharacteristicsGroupBox.TabIndex = 1;
this.songCharacteristicsGroupBox.TabStop = false;
this.songCharacteristicsGroupBox.Text = "Song Characteristics";
//
// tempoGroupBox
//
this.tempoGroupBox.Controls.Add(this.bpmLabel);
this.tempoGroupBox.Controls.Add(this.tempoBPMTextBox);
this.tempoGroupBox.Location = new System.Drawing.Point(6, 185);
this.tempoGroupBox.Name = "tempoGroupBox";
this.tempoGroupBox.Size = new System.Drawing.Size(228, 50);
this.tempoGroupBox.TabIndex = 5;
this.tempoGroupBox.TabStop = false;
this.tempoGroupBox.Text = "Tempo";
//
// bpmLabel
//
this.bpmLabel.AutoSize = true;
this.bpmLabel.Location = new System.Drawing.Point(10, 22);
this.bpmLabel.Name = "bpmLabel";
this.bpmLabel.Size = new System.Drawing.Size(33, 13);
this.bpmLabel.TabIndex = 0;
this.bpmLabel.Text = "BPM:";
//
// tempoBPMTextBox
//
this.tempoBPMTextBox.Location = new System.Drawing.Point(43, 19);
this.tempoBPMTextBox.Name = "tempoBPMTextBox";
this.tempoBPMTextBox.Size = new System.Drawing.Size(60, 20);
this.tempoBPMTextBox.TabIndex = 6;
//
// timeSignatureGroupBox
//
this.timeSignatureGroupBox.Controls.Add(this.timeTypeComboBox);
this.timeSignatureGroupBox.Controls.Add(this.beatTypeLabel);
this.timeSignatureGroupBox.Controls.Add(this.label4);
this.timeSignatureGroupBox.Controls.Add(this.timeBeatsComboBox);
this.timeSignatureGroupBox.Location = new System.Drawing.Point(103, 19);
this.timeSignatureGroupBox.Name = "timeSignatureGroupBox";
this.timeSignatureGroupBox.Size = new System.Drawing.Size(131, 77);
this.timeSignatureGroupBox.TabIndex = 5;
this.timeSignatureGroupBox.TabStop = false;
this.timeSignatureGroupBox.Text = "Time Signature";
//
// timeSignatureGroupBox.Controls.Add(this.timeTypeComboBox);
this.timeSignatureGroupBox.Controls.Add(this.beatTypeLabel);
this.timeSignatureGroupBox.Controls.Add(this.label4);
this.timeSignatureGroupBox.Controls.Add(this.timeBeatsComboBox);
this.timeSignatureGroupBox.Location = new System.Drawing.Point(103, 19);
this.timeSignatureGroupBox.Name = "timeSignatureGroupBox";
this.timeSignatureGroupBox.Size = new System.Drawing.Size(131, 77);
this.timeSignatureGroupBox.TabIndex = 5;
this.timeSignatureGroupBox.TabStop = false;
this.timeSignatureGroupBox.Text = "Time Signature";
//
// timeTypeComboBox

117

//
this.timeTypeComboBox.DisplayMember = "0";
this.timeTypeComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
this.timeTypeComboBox.Enabled = false;
this.timeTypeComboBox.FormattingEnabled = true;
this.timeTypeComboBox.Location = new System.Drawing.Point(49, 44);
this.timeTypeComboBox.Name = "timeTypeComboBox";
this.timeTypeComboBox.Size = new System.Drawing.Size(63, 21);
this.timeTypeComboBox.TabIndex = 3;
this.timeTypeComboBox.ValueMember = "0";
//
// beatTypeLabel
//
this.beatTypeLabel.AutoSize = true;
this.beatTypeLabel.Location = new System.Drawing.Point(9, 47);
this.beatTypeLabel.Name = "beatTypeLabel";
this.beatTypeLabel.Size = new System.Drawing.Size(34, 13);
this.beatTypeLabel.TabIndex = 2;
this.beatTypeLabel.Text = "Type:";
//
// label4
//
this.label4.AutoSize = true;
this.label4.Location = new System.Drawing.Point(6, 20);
this.label4.Name = "label4";
this.label4.Size = new System.Drawing.Size(37, 13);
this.label4.TabIndex = 0;
this.label4.Text = "Beats:";
//
// timeBeatsComboBox
//
this.timeBeatsComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
this.timeBeatsComboBox.Enabled = false;
this.timeBeatsComboBox.FormattingEnabled = true;
this.timeBeatsComboBox.Location = new System.Drawing.Point(49, 17);
this.timeBeatsComboBox.Name = "timeBeatsComboBox";
this.timeBeatsComboBox.Size = new System.Drawing.Size(63, 21);
this.timeBeatsComboBox.TabIndex = 1;
//
// partsGroupBox
//
this.partsGroupBox.Controls.Add(this.melodyPartCheckBox);
this.partsGroupBox.Controls.Add(this.bassPartCheckBox);
this.partsGroupBox.Location = new System.Drawing.Point(6, 19);
this.partsGroupBox.Name = "partsGroupBox";
this.partsGroupBox.Size = new System.Drawing.Size(91, 77);
this.partsGroupBox.TabIndex = 2;
this.partsGroupBox.TabStop = false;
this.partsGroupBox.Text = "Parts";
//
// melodyPartCheckBox
//
this.melodyPartCheckBox.AutoSize = true;
this.melodyPartCheckBox.Enabled = false;
this.melodyPartCheckBox.Location = new System.Drawing.Point(6, 19);
this.melodyPartCheckBox.Name = "melodyPartCheckBox";
this.melodyPartCheckBox.Size = new System.Drawing.Size(82, 17);
this.melodyPartCheckBox.TabIndex = 3;
this.melodyPartCheckBox.Text = "Melody Part";
this.melodyPartCheckBox.UseVisualStyleBackColor = true;
//
// bassPartCheckBox
//
this.bassPartCheckBox.AutoSize = true;
this.bassPartCheckBox.Enabled = false;
this.bassPartCheckBox.Location = new System.Drawing.Point(6, 46);
this.bassPartCheckBox.Name = "bassPartCheckBox";
this.bassPartCheckBox.Size = new System.Drawing.Size(71, 17);
this.bassPartCheckBox.TabIndex = 2;
this.bassPartCheckBox.Text = "Bass Part";
this.bassPartCheckBox.UseVisualStyleBackColor = true;
//
// structureGroupBox
//
this.structureGroupBox.Controls.Add(this.elementTypesRemoveButton);
this.structureGroupBox.Controls.Add(this.structureDownButton);
this.structureGroupBox.Controls.Add(this.structureUpButton);
this.structureGroupBox.Controls.Add(this.elementTypesAddButton);
this.structureGroupBox.Controls.Add(this.structureListBox);
this.structureGroupBox.Controls.Add(this.elementTypesListBox);
this.structureGroupBox.Location = new System.Drawing.Point(253, 48);
this.structureGroupBox.Name = "structureGroupBox";
this.structureGroupBox.Size = new System.Drawing.Size(361, 200);
this.structureGroupBox.TabIndex = 2;
this.structureGroupBox.TabStop = false;
this.structureGroupBox.Text = "Structure";
//
// structureDownButton
//
this.structureDownButton.Location = new System.Drawing.Point(132, 48);
this.structureDownButton.Name = "structureDownButton";
this.structureDownButton.Size = new System.Drawing.Size(52, 23);
this.structureDownButton.TabIndex = 10;
this.structureDownButton.Text = "Down";
this.structureDownButton.UseVisualStyleBackColor = true;
this.structureDownButton.Click += new System.EventHandler(this.structureDownButton_Click);
//
// structureUpButton
//
this.structureUpButton.Location = new System.Drawing.Point(132, 19);
this.structureUpButton.Name = "structureUpButton";
this.structureUpButton.Size = new System.Drawing.Size(52, 23);
this.structureUpButton.TabIndex = 9;
this.structureUpButton.Text = "Up";
this.structureUpButton.UseVisualStyleBackColor = true;
this.structureUpButton.Click += new System.EventHandler(this.structureUpButton_Click);
//
// elementTypesAddButton
//
this.elementTypesAddButton.Location = new System.Drawing.Point(280, 18);
this.elementTypesAddButton.Name = "elementTypesAddButton";
this.elementTypesAddButton.Size = new System.Drawing.Size(75, 23);
this.elementTypesAddButton.TabIndex = 8;
this.elementTypesAddButton.Text = "Add";
this.elementTypesAddButton.UseVisualStyleBackColor = true;
this.elementTypesAddButton.Click += new System.EventHandler(this.elementTypesAddButton_Click);

//
// structureListBox
//
this.structureListBox.FormattingEnabled = true;
this.structureListBox.Location = new System.Drawing.Point(6, 19);
this.structureListBox.Name = "structureListBox";
this.structureListBox.Size = new System.Drawing.Size(120, 173);
this.structureListBox.TabIndex = 7;

//
// elementTypesListBox
//
this.elementTypesListBox.FormattingEnabled = true;
this.elementTypesListBox.Location = new System.Drawing.Point(203, 19);
this.elementTypesListBox.Name = "elementTypesListBox";
this.elementTypesListBox.Size = new System.Drawing.Size(71, 95);
this.elementTypesListBox.TabIndex = 6;

//
// elementsTreeView
//
this.elementsTreeView.Location = new System.Drawing.Point(6, 19);
this.elementsTreeView.Name = "elementsTreeView";
this.elementsTreeView.Size = new System.Drawing.Size(347, 263);
this.elementsTreeView.TabIndex = 0;
this.elementsTreeView.MouseClick += new System.Windows.Forms.MouseEventHandler(this.elementsTreeView_MouseClick);

//
// caResultsGroupBox
//
this.caResultsGroupBox.Controls.Add(this.cellularAutomatonControl);
this.caResultsGroupBox.Location = new System.Drawing.Point(212, 6);
this.caResultsGroupBox.Name = "caResultsGroupBox";
this.caResultsGroupBox.Size = new System.Drawing.Size(376, 537);
this.caResultsGroupBox.TabIndex = 6;
this.caResultsGroupBox.TabStop = false;
this.caResultsGroupBox.Text = "Cellular Automaton Results";

//
// cellularAutomatonControl
//
this.cellularAutomatonControl.AutoScroll = true;
this.cellularAutomatonControl.Location = new System.Drawing.Point(6, 19);
this.cellularAutomatonControl.Name = "cellularAutomatonControl";
this.cellularAutomatonControl.Size = new System.Drawing.Size(364, 514);
this.cellularAutomatonControl.TabIndex = 9;
//
// generateSongButton
//
this.generateSongButton.Location = new System.Drawing.Point(6, 19);
this.generateSongButton.Name = "generateSongButton";
this.generateSongButton.Size = new System.Drawing.Size(120, 23);
this.generateSongButton.TabIndex = 5;
this.generateSongButton.Text = "Generate Song";
this.generateSongButton.UseVisualStyleBackColor = true;
this.generateSongButton.Click += new System.EventHandler(this.generateSongButton_Click);

// cellularAutomatonGroupBox

this.cellularAutomatonGroupBox.Controls.Add(this.stepsPerMeasureComboBox);
this.cellularAutomatonGroupBox.Controls.Add(this.stepsPerMeasureLabel);
this.cellularAutomatonGroupBox.Controls.Add(this.stepsPerMeasureComboBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsAlwaysPlayRootCheckBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsPartRandomButton);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsFracturingRuleTextBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsFracturingRuleLabel);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsElementaryRuleTextBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsElementaryRuleLabel);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsPartRandomButton);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsFracturingRuleTextBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsFracturingRuleLabel);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsElementaryRuleTextBox);
this.cellularAutomatonGroupBox.Controls.Add(this.cellsElementaryRuleLabel);
this.cellularAutomatonGroupBox.Location = new System.Drawing.Point(6, 254);
this.cellularAutomatonGroupBox.Name = "cellularAutomatonGroupBox";
this.cellularAutomatonGroupBox.Size = new System.Drawing.Size(241, 288);
this.cellularAutomatonGroupBox.TabIndex = 7;
this.cellularAutomatonGroupBox.TabStop = false;
this.cellularAutomatonGroupBox.Text = "Cellular Automaton";

this.stepsPerMeasureComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
this.stepsPerMeasureComboBox.FormattingEnabled = true;
this.stepsPerMeasureComboBox.Items.AddRange(new object[] {
"8",
"16",
"32",
"64"});
this.stepsPerMeasureComboBox.Location = new System.Drawing.Point(99, 19);
this.stepsPerMeasureComboBox.Name = "stepsPerMeasureComboBox";

// stepsPerMeasureComboBox

this.stepsPerMeasureComboBox.DropDownStyle = System.Windows.Forms.ComboBoxStyle.DropDownList;
this.stepsPerMeasureComboBox.FormattingEnabled = true;
this.stepsPerMeasureComboBox.Items.AddRange(new object[] {
"8",
"16",
"32",
"64"});
this.stepsPerMeasureComboBox.Location = new System.Drawing.Point(99, 19);
this.stepsPerMeasureComboBox.Name = "stepsPerMeasureComboBox";
this.stepsPerMeasureComboBox.Size = new System.Drawing.Size(63, 21);
this.stepsPerMeasureComboBox.TabIndex = 6;
// stepsPerMeasureLabel 
//
this.stepsPerMeasureLabel.AutoSize = true;
this.stepsPerMeasureLabel.Location = new System.Drawing.Point(6, 22);
this.stepsPerMeasureLabel.Name = "stepsPerMeasureLabel";
this.stepsPerMeasureLabel.Size = new System.Drawing.Size(83, 13);
this.stepsPerMeasureLabel.TabIndex = 5;
this.stepsPerMeasureLabel.Text = "Steps/Measure:";
//
// bassAlwaysPlayRootCheckBox
//
this.bassAlwaysPlayRootCheckBox.AutoSize = true;
this.bassAlwaysPlayRootCheckBox.Location = new System.Drawing.Point(20, 194);
this.bassAlwaysPlayRootCheckBox.Name = "bassAlwaysPlayRootCheckBox";
this.bassAlwaysPlayRootCheckBox.Size = new System.Drawing.Size(134, 17);
this.bassAlwaysPlayRootCheckBox.TabIndex = 9;
this.bassAlwaysPlayRootCheckBox.Text = "Always Play Root Note";
this.bassAlwaysPlayRootCheckBox.UseVisualStyleBackColor = true;

System.Drawing.Point(20, 194);
this.bassAlwaysPlayRootCheckBox.Name = "bassAlwaysPlayRootCheckBox";
this.bassAlwaysPlayRootCheckBox.Size = new System.Drawing.Size(134, 17);
this.bassAlwaysPlayRootCheckBox.TabIndex = 9;
this.bassAlwaysPlayRootCheckBox.Text = "Always Play Root Note";
this.bassAlwaysPlayRootCheckBox.UseVisualStyleBackColor = true;
//
// label11

this.label11.AutoSize = true;
this.label11.Location = new System.Drawing.Point(6, 123);
this.label11.Name = "label11";
this.label11.Size = new System.Drawing.Size(61, 13);
this.label11.TabIndex = 17;
this.label11.Text = "Bass Part";
//
// bassPartRandomButton
//
this.bassPartRandomButton.Location = new System.Drawing.Point(160, 140);
this.bassPartRandomButton.Name = "bassPartRandomButton";
this.bassPartRandomButton.Size = new System.Drawing.Size(75, 23);
this.bassPartRandomButton.TabIndex = 13;
this.bassPartRandomButton.Text = "Random";
this.bassPartRandomButton.UseVisualStyleBackColor = true;
this.bassPartRandomButton.Click += new System.EventHandler(this.bassPartRandomButton_Click);

// bassFracturingRuleTextBox
//
this.bassFracturingRuleTextBox.Location = new System.Drawing.Point(99, 168);
this.bassFracturingRuleTextBox.Name = "bassFracturingRuleTextBox";
this.bassFracturingRuleTextBox.Size = new System.Drawing.Size(55, 20);
this.bassFracturingRuleTextBox.TabIndex = 16;
this.melodyFracturingRuleTextBox.Size = new System.Drawing.Size(55, 20);
this.melodyFracturingRuleTextBox.TabIndex = 10;
//
// melodyFracturingRuleTextBox
//
this.melodyFracturingRuleLabel.AutoSize = true;
this.melodyFracturingRuleLabel.Location = new System.Drawing.Point(11, 96);
this.melodyFracturingRuleLabel.Name = "melodyFracturingRuleLabel";
this.melodyFracturingRuleLabel.Size = new System.Drawing.Size(82, 13);
this.melodyFracturingRuleLabel.TabIndex = 9;
this.melodyFracturingRuleLabel.Text = "Fracturing Rule:";
//
// melodyElementaryRuleTextBox
//
this.melodyElementaryRuleTextBox.Location = new System.Drawing.Point(99, 67);
this.melodyElementaryRuleTextBox.Name = "melodyElementaryRuleTextBox";
this.melodyElementaryRuleTextBox.Size = new System.Drawing.Size(55, 20);
this.melodyElementaryRuleTextBox.TabIndex = 7;
//
// melodyElementaryRuleLabel
//
this.melodyElementaryRuleLabel.AutoSize = true;
this.melodyElementaryRuleLabel.Location = new System.Drawing.Point(6, 70);
this.melodyElementaryRuleLabel.Name = "melodyElementaryRuleLabel";
this.melodyElementaryRuleLabel.Size = new System.Drawing.Size(87, 13);
this.melodyElementaryRuleLabel.TabIndex = 8;
this.melodyElementaryRuleLabel.Text = "Elementary Rule:";
//
// tabControl1
//
this.tabControl1.Controls.Add(this.setupPage);
this.tabControl1.Controls.Add(this.melodyPage);
this.tabControl1.Location = new System.Drawing.Point(0, 0);
this.tabControl1.Name = "tabControl1";
this.tabControl1.SelectedIndex = 0;
this.tabControl1.Size = new System.Drawing.Size(628, 574);
this.tabControl1.TabIndex = 8;
//
// setupPage
//
this.setupPage.Controls.Add(this.elementsGroupBox);
this.setupPage.Controls.Add(this.initialConditionsGroupBox);
this.setupPage.Controls.Add(this.songCharacteristicsGroupBox);
this.setupPage.Controls.Add(this.cellularAutomatonGroupBox);
this.setupPage.Controls.Add(this.structureGroupBox);
this.setupPage.Location = new System.Drawing.Point(4, 22);
this.setupPage.Name = "setupPage";
this.setupPage.Padding = new System.Windows.Forms.Padding(3);
this.setupPage.Size = new System.Drawing.Size(628, 574);
this.setupPage.TabIndex = 0;
this.setupPage.Text = "Setup";
this.setupPage.UseVisualStyleBackColor = true;
//
// elementsGroupBox
//
this.elementsGroupBox.Controls.Add(this.elementsTreeView);
this.elementsGroupBox.Location = new System.Drawing.Point(253, 254);
this.elementsGroupBox.Name = "elementsGroupBox";
this.elementsGroupBox.Size = new System.Drawing.Size(361, 288);
this.elementsGroupBox.TabIndex = 9;
this.elementsGroupBox.TabStop = false;
this.elementsGroupBox.Text = "Elements";
//
// initialConditionsGroupBox
//
this.initialConditionsGroupBox.Controls.Add(this.initialConditionsControl);
this.initialConditionsGroupBox.Location = new
System.Drawing.Point(253, 6);
this.initialConditionsGroupBox.Name = "initialConditionsGroupBox";
this.initialConditionsGroupBox.Size = new
System.Drawing.Size(361, 42);
this.initialConditionsGroupBox.TabIndex = 8;
this.initialConditionsGroupBox.TabStop = false;
this.initialConditionsGroupBox.Text = "Initial Conditions";
//
// initialConditionsControl
//
this.initialConditionsControl.AutoScroll = true;
this.initialConditionsControl.BackColor = System.Drawing.Color.LightSteelBlue;
this.initialConditionsControl.Location = new
System.Drawing.Point(6, 19);
this.initialConditionsControl.Name = "initialConditionsControl";
this.initialConditionsControl.Size = new
System.Drawing.Size(347, 17);
this.initialConditionsControl.TabIndex = 9;
//
// melodyPage
//
this.melodyPage.Controls.Add(this.patternsGroupBox);
this.melodyPage.Controls.Add(this.caResultsGroupBox);
this.melodyPage.Location = new System.Drawing.Point(4, 22);
this.melodyPage.Name = "melodyPage";
this.melodyPage.Padding = new System.Windows.Forms.Padding(3);
this.melodyPage.Size = new System.Drawing.Size(620, 548);
this.melodyPage.TabIndex = 1;
this.melodyPage.UseVisualStyleBackColor = true;
//
// patternsGroupBox
//
this.patternsGroupBox.Controls.Add(this.patternListBox);
this.patternsGroupBox.Controls.Add(this.patternLabel);
this.patternsGroupBox.Controls.Add(this.partComboBox);
this.patternsGroupBox.Controls.Add(this.partLabel);
this.patternsGroupBox.Location = new System.Drawing.Point(8, 6);
this.patternsGroupBox.Name = "patternsGroupBox";
this.patternsGroupBox.Size = new System.Drawing.Size(198, 537);
this.patternsGroupBox.TabIndex = 7;
this.patternsGroupBox.TabStop = false;
this.patternsGroupBox.Text = "Patterns";
//
// patternListBox
//
this.patternListBox.FormattingEnabled = true;
this.patternListBox.Location = new System.Drawing.Point(56, 46);
this.patternListBox.Name = "patternListBox";
this.patternListBox.Size = new System.Drawing.Size(120, 485);
this.patternListBox.SelectedIndexChanged += new System.EventHandler(this.patternListBox_SelectedIndexChanged);
//
// patternLabel
//
this.patternLabel.AutoSize = true;
this.patternLabel.Location = new System.Drawing.Point(6, 46);
this.patternLabel.Name = "patternLabel";
this.patternLabel.Size = new System.Drawing.Size(44, 13);
this.patternLabel.TabIndex = 10;
this.patternLabel.Text = "Pattern:";
//
// partComboBox
//
this.partComboBox.FormattingEnabled = true;
this.partComboBox.Items.AddRange(new object[] { "Melody", "Bass" });
this.partComboBox.Location = new System.Drawing.Point(56, 19);
this.partComboBox.Name = "partComboBox";
this.partComboBox.Size = new System.Drawing.Size(121, 21);
this.partComboBox.TabIndex = 9;
this.partComboBox.SelectedIndexChanged += new System.EventHandler(this.partComboBox_SelectedIndexChanged);
//
// partLabel
//
this.partLabel.AutoSize = true;
this.partLabel.Location = new System.Drawing.Point(21, 22);
this.partLabel.Name = "partLabel";
this.partLabel.Size = new System.Drawing.Size(29, 13);
this.partLabel.TabIndex = 8;
this.partLabel.Text = "Part:";
//
// filenameTextBox
//
this.filenameTextBox.Enabled = false;
this.filenameTextBox.Location = new System.Drawing.Point(6, 100);
this.filenameTextBox.Name = "filenameTextBox";
this.filenameTextBox.Size = new System.Drawing.Size(120, 20);
this.filenameTextBox.TabIndex = 9;
//
// filenameLabel
//
this.filenameLabel.AutoSize = true;
this.filenameLabel.Location = new System.Drawing.Point(6, 84);
this.filenameLabel.Name = "filenameLabel";
this.filenameLabel.Size = new System.Drawing.Size(52, 13);
this.filenameLabel.TabIndex = 11;
this.filenameLabel.Text = "Filename:";
this.pathLabel.AutoSize = true;
this.pathLabel.Location = new System.Drawing.Point(6, 45);
this.pathLabel.Name = "pathLabel";
this.pathLabel.Size = new System.Drawing.Size(32, 13);
this.pathLabel.TabIndex = 12;
this.pathLabel.Text = "Path:";

this.pathTextBox.Enabled = false;
this.pathTextBox.Location = new System.Drawing.Point(6, 61);
this.pathTextBox.Name = "pathTextBox";
this.pathTextBox.Size = new System.Drawing.Size(246, 20);
this.pathTextBox.TabIndex = 13;

this.openInFinaleButton.Location = new System.Drawing.Point(132, 98);
this.openInFinaleButton.Name = "openInFinaleButton";
this.openInFinaleButton.Size = new System.Drawing.Size(120, 35);
this.openInFinaleButton.TabIndex = 14;
this.openInFinaleButton.Text = "Open in Finale Reader";
this.openInFinaleButton.UseVisualStyleBackColor = true;
this.openInFinaleButton.Click += new System.EventHandler(this.openInFinaleButton_Click);

this.loadSetupButton.Location = new System.Drawing.Point(6, 19);
this.loadSetupButton.Name = "loadSetupButton";
this.loadSetupButton.Size = new System.Drawing.Size(120, 23);
this.loadSetupButton.TabIndex = 16;
this.loadSetupButton.UseVisualStyleBackColor = true;
this.loadSetupButton.Click += new System.EventHandler(this.loadSetupButton_Click);

this.saveSetupButton.Location = new System.Drawing.Point(132, 19);
this.saveSetupButton.Name = "saveSetupButton";
this.saveSetupButton.Size = new System.Drawing.Size(120, 23);
this.saveSetupButton.TabIndex = 18;
this.saveSetupButton.UseVisualStyleBackColor = true;
this.saveSetupButton.Click += new System.EventHandler(this.saveSetupButton_Click);

this.groupBox1.Controls.Add(this.loadSetupButton);
this.groupBox1.Controls.Add(this.saveSetupButton);
this.groupBox1.Location = new System.Drawing.Point(630, 17);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(260, 53);
this.groupBox1.TabIndex = 19;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Setup";
    // resultsGroupbox
    //
    this.resultsGroupbox.Controls.Add(this.generateSongButton);
    this.resultsGroupbox.Controls.Add(this.filenameLabel);
    this.resultsGroupbox.Controls.Add(this.filenameTextBox);
    this.resultsGroupbox.Controls.Add(this.openInFinaleButton);
    this.resultsGroupbox.Controls.Add(this.pathLabel);
    this.resultsGroupbox.Controls.Add(this.pathTextBox);
    this.resultsGroupbox.Location = new System.Drawing.Point(630, 76);
    this.resultsGroupbox.Name = "resultsGroupbox";
    this.resultsGroupbox.Size = new System.Drawing.Size(260, 141);
    this.resultsGroupbox.TabIndex = 20;
    this.resultsGroupbox.TabStop = false;
    this.resultsGroupbox.Text = "Results";
    //
    // elementTypesRemoveButton
    //
    this.elementTypes RemoveButton.Location = new System.Drawing.Point(132, 77);
    this.elementTypesRemoveButton.Name = "elementTypesRemoveButton";
    this.elementTypesRemoveButton.Size = new System.Drawing.Size(65, 23);
    this.elementTypesRemoveButton.TabIndex = 11;
    this.elementTypesRemoveButton.Text = "Remove";
    this.elementTypesRemoveButton.UseVisualStyleBackColor = true;
    this.elementTypesRemoveButton.Click += new System.EventHandler(this.elementTypesRemoveButton_Click);
    //
    // MainUserInterface
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.ClientSize = new System.Drawing.Size(898, 574);
    this.Controls.Add(this.resultsGroupbox);
    this.Controls.Add(this.groupBox1);
    this.Controls.Add(this.tabControl1);
    this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;
    this.Name = "MainUserInterface";
    this.Text = "Frond";
    this.Load += new System.EventHandler(this.MainUserInterface_Load);
    this.measureContextMenu.ResumeLayout(false);
    this.patternContextMenu.ResumeLayout(false);
    this.elementContextMenu.ResumeLayout(false);
    this.keyGroupBox.ResumeLayout(false);
    this.keyGroupBox.PerformLayout();
    this.songCharacteristicsGroupBox.ResumeLayout(false);
    this.tempoGroupBox.ResumeLayout(false);
    this.tempoGroupBox.PerformLayout();
    this.timeSignatureGroupBox.ResumeLayout(false);
    this.timeSignatureGroupBox.PerformLayout();
    this.partsGroupBox.ResumeLayout(false);
    this.partsGroupBox.PerformLayout();
    this.structureGroupBox.ResumeLayout(false);
    this.caResultsGroupBox.ResumeLayout(false);
    this.cellularAutomatonGroupBox.ResumeLayout(false);
    this.cellularAutomatonGroupBox.PerformLayout();
    this.tabControl1.ResumeLayout(false);
    this.setupPage.ResumeLayout(false);
    this.elementsGroupBox.ResumeLayout(false);
    this.initialConditionsGroupBox.ResumeLayout(false);
this.melodyPage.ResumeLayout(false);
this.patternsGroupBox.ResumeLayout(false);
this.patternsGroupBox.PerformLayout();
this.groupBox1.ResumeLayout(false);
this.resultsGroupBox.ResumeLayout(false);
this.resultsGroupBox.PerformLayout();
this.ResumeLayout(false);
}
#region
gion
private System.Windows.Forms.GroupBox keyGroupBox;
private System.Windows.Forms.GroupBox songCharacteristicsGroupBox;
private System.Windows.Forms.Label keyScaleLabel;
private System.Windows.Forms.ComboBox keyRootComboBox;
private System.Windows.Forms.Label keyRootLabel;
private System.Windows.Forms.ComboBox keyScaleComboBox;
private System.Windows.Forms.CheckBox bassPartCheckBox;
private System.Windows.Forms.CheckBox melodyPartCheckBox;
private System.Windows.Forms.GroupBox partsGroupBox;
private System.Windows.Forms.GroupBox timeSignatureGroupBox;
private System.Windows.Forms.Label label4;
private System.Windows.Forms.ComboBox timeTypeComboBox;
private System.Windows.Forms.Label beatTypeLabel;
private System.Windows.Forms.ComboBox timeBeatsComboBox;
private System.Windows.Forms.GroupBox tempoGroupBox;
private System.Windows.Forms.Label bpmLabel;
private System.Windows.Forms.TextBox tempoBPMTextBox;
private System.Windows.Forms.GroupBox structureGroupBox;
private System.Windows.Forms.TreeView elementsTreeView;
private System.Windows.Forms.GroupBox caResultsGroupBox;
private System.Windows.Forms.Button generateSongButton;
private System.Windows.Forms.GroupBox cellularAutomatonGroupBox;
private System.Windows.Forms.TabControl tabControl1;
private System.Windows.Forms.TabPage setupPage;
private System.Windows.Forms.TabPage melodyPage;
private CellularAutomata.CellularAutomatonControl cellularAutomatonControl;
private System.Windows.Forms.GroupBox initialConditionsGroupBox;
private CellularAutomata.CellularAutomatonControl initialConditionsControl;
private System.Windows.Forms.TextBox melodyFracturingRuleTextBox;
private System.Windows.Forms.Label melodyFracturingRuleLabel;
private System.Windows.Forms.TextBox filenameTextBox;
private System.Windows.Forms.Label filenameLabel;
private System.Windows.Forms.TextBox pathTextBox;
private System.Windows.Forms.Button openInFinaleButton;
private System.Windows.Forms.Label label11;
private System.Windows.Forms.TextBox bassFracturingRuleTextBox;
private System.Windows.Forms.Label bassFracturingRuleLabel;
private System.Windows.Forms.TextBox bassElementaryRuleTextBox;
private System.Windows.Forms.Label bassElementaryRuleLabel;
private System.Windows.Forms.CheckBox bassAlwaysPlayRootCheckBox;
private System.Windows.Forms.ListBox structureListBox;
private System.Windows.Forms.ListBox elementTypesListBox;
private System.Windows.Forms.GroupBox elementsGroupBox;
private System.Windows.Forms.ToolStripMenuItem addMeasureMenuItem;
private System.Windows.Forms.ToolStripMenuItem dynamicsMenuItem;
private System.Windows.Forms.ToolStripMenuItem addPatternMenuItem;
private System.Windows.Forms.ToolStripSeparator patternSeparator;
private System.Windows.Forms.ToolStripMenuItem chordMenuItem;
private System.Windows.Forms.ToolStripMenuItem removePatternMenuItem;
private System.Windows.Forms.ToolStripSeparator measureSeparator;
private System.Windows.Forms.GroupBox patternsGroupBox;
private System.Windows.Forms.ComboBox partComboBox;
private System.Windows.Forms.Label partLabel;
private System.Windows.Forms.Label patternLabel;
private System.Windows.Forms.ListBox patternListBox;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.GroupBox resultsGroupBox;
private System.Windows.Forms.ComboBox stepsPerMeasureComboBox;
private System.Windows.Forms.Label stepsPerMeasureLabel;
private System.Windows.Forms.ToolStripMenuItem repeatsToolStripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats1StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats2StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats3StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats4StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats5StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats6StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats7StripMenuItem;
private System.Windows.Forms.ToolStripMenuItem repeats8StripMenuItem;
private System.Windows.Forms.Button elementTypesRemoveButton;
}
}

ChordDialog.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
namespace Frond
{
    public partial class ChordDialog : Form
    {
        private Chord.ChordType chordType = Chord.ChordType.Major;
        private Chord.ScaleDegree scaleDegree = Chord.ScaleDegree.I;
        private bool chordTypeSelected = false;
        private bool scaleDegreeSelected = false;

        public Chord.ChordType ChordType
        {
            get
            {
                if (chordTypeSelected == true)
                {
                    return chordType;
                }
                else
                {
                    throw new Exception("User did not select a chord type in the chord dialog.");
                }
            }
        }

        public Chord.ScaleDegree ScaleDegree
        {
            get
            {
                if (scaleDegreeSelected == true)
                {
                    return scaleDegree;
                }
                else
                {
                    throw new Exception("User did not select a scale degree in the chord dialog.");
                }
            }
        }

        public ChordDialog()
        {
            InitializeComponent();
        }

        private void ChordDialog_Load(object sender, EventArgs e)
        {
            // Populate the scale degrees.
            foreach (Chord.ScaleDegree scaleDegree in Utilities.EnumToList<Chord.ScaleDegree>())
            {
                scaleDegreeListBox.Items.Add(Utilities.GetEnumDescription(scaleDegree));
            }

            // Populate the chord types.
            foreach (Chord.ChordType chordType in Utilities.EnumToList<Chord.ChordType>())
            {
chordTypeListView.Items.Add(Utilities.GetEnumDescription(chordType));

private void chordDialogCancelButton_Click(object sender, EventArgs e)
{
    // The user cancelled. Close the form.
    Close();
}

private void chordDialogOKButton_Click(object sender, EventArgs e)
{
    // Grab the selected scale degree.
    int scaleDegreeIndex = scaleDegreeListBox.SelectedIndex;
    if (scaleDegreeIndex >= 0 && scaleDegreeIndex <
        scaleDegreeListBox.Items.Count)
    {
        scaleDegree =
            Chord.LookupScaleDegree(scaleDegreeListBox.Items[scaleDegreeIndex].ToString());
        scaleDegreeSelected = true;
    }
    else
    {
        // Nothing was selected or an invalid index was returned.
        MessageBox.Show("No scale degree selected.", "Error",
            MessageBoxButtons.OK);
        return;
    }

    // Grab the selected chord type.
    ListView.SelectedItemsCollection
        chordTypeSelectedIndexCollection = chordTypeListView.SelectedItems;
    if (chordTypeSelectedIndexCollection.Count == 1)
    {
        int index = chordTypeSelectedIndexCollection[0];
        chordType =
            Chord.LookupChordType(chordTypeListView.SelectedItems[index].Text);
        chordTypeSelected = true;
    }
    else if (chordTypeSelectedIndexCollection.Count == 0)
    {
        MessageBox.Show("No chord type selected.", "Error",
            MessageBoxButtons.OK);
        return;
    }
    else
    {
        MessageBox.Show("Multiple chord types selected.", "Error",
            MessageBoxButtons.OK);
        return;
    }

    // If we successfully got a scale degree and chord type, close
    // the dialog.
    if (scaleDegreeSelected == true && chordTypeSelected == true)
    {
        // Close the form.
        this.DialogResult = DialogResult.OK;
        Close();
    }
else
{
    this.DialogResult = DialogResult.Cancel;
}

namespace Frond{

    partial class ChordDialog
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {            this.chordTypeListView = new System.Windows.Forms.ListView();
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.scaleDegreeListBox = new System.Windows.Forms.ListBox();
            this.chordDialogOKButton = new System.Windows.Forms.Button();
            this.chordDialogCancelButton = new System.Windows.Forms.Button();
            this.SuspendLayout();

            this.chordTypeListView.FullRowSelect = true;
            this.chordTypeListView.Location = new System.Drawing.Point(93, 25);
this.chordTypeListView.MultiSelect = false;
this.chordTypeListView.Name = "chordTypeListView";
this.chordTypeListView.Size = new System.Drawing.Size(314, 189);
this.chordTypeListView.TabIndex = 0;
this.chordTypeListView.UseCompatibleStateImageBehavior = false;
this.chordTypeListView.View = System.Windows.Forms.View.List;

//
// label1
//
this.label1.AutoSize = true;
this.label1.Location = new System.Drawing.Point(93, 9);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(34, 13);
this.label1.TabIndex = 1;
this.label1.Text = "Type:";

//
// label2
//
this.label2.AutoSize = true;
this.label2.Location = new System.Drawing.Point(12, 9);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(75, 13);
this.label2.TabIndex = 2;
this.label2.Text = "Scale Degree:";

//
// scaleDegreeListBox
//
this.scaleDegreeListBox.FormattingEnabled = true;
this.scaleDegreeListBox.Location = new System.Drawing.Point(12, 25);
this.scaleDegreeListBox.Name = "scaleDegreeListBox";
this.scaleDegreeListBox.Size = new System.Drawing.Size(75, 95);
this.scaleDegreeListBox.TabIndex = 3;

//
// chordDialogOKButton
//
this.chordDialogOKButton.Location = new System.Drawing.Point(413, 25);
this.chordDialogOKButton.Name = "chordDialogOKButton";
this.chordDialogOKButton.Size = new System.Drawing.Size(75, 23);
this.chordDialogOKButton.TabIndex = 4;
this.chordDialogOKButton.Text = "OK";
this.chordDialogOKButton.UseVisualStyleBackColor = true;
this.chordDialogOKButton.Click += new System.EventHandler(this.chordDialogOKButton_Click);

//
// chordDialogCancelButton
//
this.chordDialogCancelButton.Location = new System.Drawing.Point(413, 54);
this.chordDialogCancelButton.Name = "chordDialogCancelButton";
this.chordDialogCancelButton.Size = new System.Drawing.Size(75, 23);
this.chordDialogCancelButton.TabIndex = 5;
this.chordDialogCancelButton.Text = "Cancel";
this.chordDialogCancelButton.UseVisualStyleBackColor = true;
this.chordDialogCancelButton.Click += new System.EventHandler(this.chordDialogCancelButton_Click);

//
// ChordDialog
//
private System.Windows.Forms.ListView chordTypeListView;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.ListBox scaleDegreeListBox;
this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;
this.Name = "ChordDialog";
this.Text = "Chords";
this.Load += new System.EventHandler(this.ChordDialog_Load);
this.ResumeLayout(false);
this.PerformLayout();

namespace CellularAutomata
{
    public enum CellState
    {
        Off,
        Off_Active,
        On,
        On_Active
    }
    public partial class CellControl : UserControl
    {
        // Private data members.
        private CellState state = CellState.Off;
        private string text = "\";

        // Setters/getters.
    }
}
public CellState State
{
    get { return state; }
    set
    {
        state = value;
        UpdateAppearance();
    }
}

public string Text
{
    get { return text; }
    set { text = value; }
}

// Methods
public CellControl()
{
    InitializeComponent();
    UpdateAppearance();
}

public override string ToString()
{
    return state.ToString();
}

private void UpdateAppearance()
{
    switch (State)
    {
        case CellState.Off:
            BackColor = Color.White;
            ForeColor = Color.White;
            break;
        case CellState.Off_Active:
            BackColor = Color.White;
            ForeColor = Color.Red;
            break;
        case CellState.On:
            BackColor = Color.FromArgb(191, 191, 191);
            ForeColor = Color.FromArgb(191, 191, 191);
            break;
        case CellState.On_Active:
            BackColor = Color.FromArgb(191, 191, 191);
            ForeColor = Color.Red;
            break;
        default:
            throw (new Exception("Invalid cell state encountered."));
    }
}

private void Cell_MouseClick(object sender, MouseEventArgs e)
{
    switch (State)
    {
        case CellState.Off:
            State = CellState.Off_Active;
            break;
        case CellState.Off_Active:
            State = CellState.On;
            break;
        case CellState.On:
            State = CellState.On_Active;
            break;
    }
}
break;
case CellState.On_Active:
    State = CellState.Off;
    break;
default:
    throw (new Exception("Invalid cell state encountered."));
}

private void Cell_Paint(object sender, PaintEventArgs e)
{
    if (ForeColor != BackColor)
    {
        int borderWidth = 1;
        Color borderColor = ForeColor;
        ControlPaint.DrawBorder(e.Graphics, e.ClipRectangle,
            borderColor, borderWidth, ButtonBorderStyle.Solid,
            borderColor, borderWidth, ButtonBorderStyle.Solid,
            ButtonBorderStyle.Solid,
            borderColor, borderWidth, ButtonBorderStyle.Solid);
    }
    if (text != "")
    {
        System.Drawing.Font drawFont = new System.Drawing.Font("Arial", 5);
        float x = 0.0f;
        float y = 1.0f;
        RectangleF drawRect = new RectangleF(x, y, Size.Width, Size.Height);
        StringFormat stringFormat = new StringFormat();
        stringFormat.Alignment = StringAlignment.Center;
        stringFormat.LineAlignment = StringAlignment.Center;
        stringFormat.Trimming = StringTrimming.None;
        e.Graphics.DrawString(text, drawFont, drawBrush, drawRect, stringFormat);
    }
}

CellControl.Designer.cs

namespace CellularAutomata
{
    partial class CellControl
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;
    }
}
/// <summary>
/// Clean up any resources being used.
/// </summary>
/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}

#region Component Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    // CellControl
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.CausesValidation = false;
    this.DoubleBuffered = true;
    this.Margin = new System.Windows.Forms.Padding(0);
    this.Name = "CellControl";
    this.Size = new System.Drawing.Size(15, 15);
    this.Paint += new System.Windows.Forms.PaintEventHandler(this.Cell_Paint);
    this.MouseClick += new System.Windows.Forms.MouseEventHandler(this.Cell_MouseClick);
    this.ResumeLayout(false);
}
#endregion

CellularAutomatonControl.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace CellularAutomata
{
    public partial class CellularAutomatonControl : UserControl
    {
        delegate void ResetCallback(int width, int height);

        private CellControl[,] cells = null;
        private int height = 0;
        private int width = 0;
        private object cellLock = new object();

        public CellularAutomatonControl()
        {
            InitializeComponent();
        }

        private void CellularAutomaton_Load(object sender, EventArgs e)
        {
            Reset(width, height);
        }

        public void SetInitialRow(CellState[] initialCells)
        {
            lock (cellLock)
            {
                if (height > 0)
                {
                    if (initialCells.Length == width)
                    {
                        for (int i = 0; i < width; i++)
                        {
                            cells[0][i].State = initialCells[i];
                        }
                    }
                    else
                    {
                        throw new Exception("Initial row set failed!  Rows have different widths.");
                    }
                }
                else
                {
                    throw new Exception("Initial row set failed!  Height is not greater than 0.");
                }
            }
        }

        public void Reset(int width, int height)
        {
            lock (cellLock)
            {
                this.width = width;
                this.height = height;

                // Hide the current cells.
                if (cells != null)
                {
                    foreach (CellControl[] row in cells)
                    {
                        if (row != null)
                        {
                            foreach (CellControl cell in row)
// Create a new set of cells.
cells = null;
if (width > 0 && height > 0)
{
cells = new CellControl[height][];
for (int i = 0; i < height; i++)
{
cells[i] = new CellControl[width];
for (int j = 0; j < width; j++)
{
cells[i][j] = new CellControl();
this.Controls.Add(cels[i][j]);
cells[i][j].BackColor = System.Drawing.Color.White;
cells[i][j].ForeColor = System.Drawing.Color.White;
cells[i][j].Location = new System.Drawing.Point(15 * j, 15 * i);
cells[i][j].Margin = new System.Windows.Forms.Padding(0);
cells[i][j].Name = "cell1";
cells[i][j].Size = new System.Drawing.Size(15, 15);
cells[i][j].State = CellularAutomata.CellState.Off;
cells[i][j].TabIndex = 0;
}
}
}

public CellState[][] GetCellStates()
{
lock (cellLock)
{
int height = cells.Length;
CellState[][] states = new CellState[height][];
for (int i = 0; i < height; i++)
{
int width = cells[i].Length;
states[i] = new CellState[width];
for (int j = 0; j < width; j++)
{
states[i][j] = new CellState();
states[i][j] = cells[i][j].State;
}
}
return states;
}

public void SetCellState(int row, int column, CellState cellState)
{
lock (cellLock)
{
public void SetCellHeader(int row, int column, string text)
{
    lock (cellLock)
    {
        cells[row][column].State = CellState.Off;
        cells[row][column].ForeColor = Color.SteelBlue;
        cells[row][column].BackColor = Color.SteelBlue;
        cells[row][column].Text = text;
    }
}

public static CellState FromString(string state)
{
    CellState cellState = CellState.Off;
    if (state == "Off")
    {
        cellState = CellularAutomata.CellState.Off;
    }
    else if (state == "Off_Active")
    {
        cellState = CellularAutomata.CellState.Off_Active;
    }
    else if (state == "On")
    {
        cellState = CellularAutomata.CellState.On;
    }
    else if (state == "On_Active")
    {
        cellState = CellularAutomata.CellState.On_Active;
    }
    else
    {
        throw new Exception("Error converting CellState from string " + state + ".");
    }

    return cellState;
}

private int GetWidth()
{
    return width;
}

private int GetHeight()
{
    return height;
}

CellularAutomatonControl.Designer.cs

namespace CellularAutomata
partial class CellularAutomatonControl
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Component Designer generated code

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.SuspendLayout();
        //
        // CellularAutomatonControl
        //
        this.AutoScroll = true;
        this.DoubleBuffered = true;
        this.Name = "CellularAutomatonControl";
        this.Size = new System.Drawing.Size(296, 327);
        this.Load += new System.EventHandler(this.CellularAutomaton_Load);
        this.ResumeLayout(false);
    }

    #endregion
}

Cell.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using CellularAutomata;

namespace Frond
{
    public class Cell
    {
        private CellState state = CellState.Off;
        public CellState State
        {
            get { return state; }
            set { state = value; }
        }
        public override string ToString()
        {
            return Utilities.GetEnumDescription(state);
        }
    }
}

CellularAutomaton.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using CellularAutomata;

namespace Frond
{
    public class CellularAutomaton
    {
        // Constants
        public const int DEFAULT_CA_WIDTH = 23;
        public const int DEFAULT_CA_LENGTH = 1;
        public const int DEFAULT_ELEMENTARY_RULE = 90;
        public const int DEFAULT_FRACTURING_RULE = 4521;
        public static Cell[] DEFAULT_INITIAL_CONDITIONS
        {
            get
            {
                Cell[] defaultInitialConditions = new Cell[CellularAutomaton.DEFAULT_CA_WIDTH];
                for (int i = 0; i < CellularAutomaton.DEFAULT_CA_WIDTH; i++)
                {
                    defaultInitialConditions[i] = new Cell();
                }
                return defaultInitialConditions;
            }
        }
    }
}
if (i == Convert.ToInt32(CellularAutomaton.DEFAULT_CA_WIDTH / 2))
{
    defaultInitialConditions[i].State = CellState.On_Active;
}
else
{
    defaultInitialConditions[i].State = CellState.Off;
}
return defaultInitialConditions;

// Private data members.
private int width = DEFAULT_CA_WIDTH;
private List<Cell[]> cells = new List<Cell[]>();
private Rule rule = new Rule(DEFAULT_ELEMENTARY_RULE,
DEFAULT_FRACTURING_RULE);

// Setters/getters.
public int Width
{
    get { return width; }
}
public int Length
{
    get { return cells.Count; }
    set
    {
        // Reinitialize the CA.
        if (cells.Count > 1)
        {
            cells.RemoveRange(1, cells.Count);
        }
        for (int i = cells.Count; i < value; i++)
        {
            cells.Add(new Cell[width]);
            for (int j = 0; j < width; j++)
            {
                cells[i][j] = new Cell();
            }
        }
    }
}
public Cell[] InitialConditions
{
    get
    {
        if (cells.Count > 0)
        {
            return cells[0];
        }
        else
        {
            throw new Exception("Initial conditions get when CA
height was <= 0.");
        }
    }
    set
    {
    }
if (value.Length == Width)
{
    if (cells.Count > 0)
    {
        for (int i = 0; i < Width; i++)
        {
            cells[0][i] = value[i];
        }
    }
    else
    {
        throw new Exception("Initial conditions set when CA
height was <= 0.");
    }
} else
{
    throw new Exception("Invalid size specified for initial
conditions.");
}

public Rule Rule
{
    get { return rule; }
    set
    {
        rule.ElementaryRule = value.ElementaryRule;
        rule.FracturingRule = value.FracturingRule;
    }
}

public List<Cell[]> Cells
{
    get { return cells; }
}

// Methods
public CellularAutomaton()
{
    for (int i = 0; i < DEFAULT_CA_LENGTH; i++)
    {
        cells.Add(new Cell(Width));
        for (int j = 0; j < Width; j++)
        {
            cells[0][j] = new Cell();
        }
    }
    InitialConditions = DEFAULT_INITIAL_CONDITIONS;
}

public void Process()
{
    // Proceed through calculating the states for all cells in the
    // CA.
    for (int i = 1; i < Length; i++)
    {
        for (int j = 0; j < Width; j++)
        {
            // Determine the index of the second left neighbor.
            int xLeftIndex2 = j - 2;
            if (xLeftIndex2 < 0)
            {
                xLeftIndex2 += Width;
            }
// Determine the index of the left neighbor.
int xLeftIndex = j - 1;
if (xLeftIndex < 0)
{
    xLeftIndex += Width;
}

// Determine the index of the middle neighbor.
int xMiddleIndex = j;

// Determine the index of the right neighbor.
int xRightIndex = j + 1;
if (xRightIndex >= Width)
{
    xRightIndex -= Width;
}

// Determine the index of the second right neighbor.
int xRightIndex2 = j + 2;
if (xRightIndex2 >= Width)
{
    xRightIndex2 -= Width;
}

// Get the state of the cells.
CellState xLeft2 = cells[i - 1][xLeftIndex2].State;
CellState xLeft = cells[i - 1][xLeftIndex].State;
CellState xMiddle = cells[i - 1][xMiddleIndex].State;
CellState xRight = cells[i - 1][xRightIndex].State;
CellState xRight2 = cells[i - 1][xRightIndex2].State;

// Calculate the state of the current cell.
rule.ExecuteRule(xLeft2, xLeft, xMiddle, xRight, xRight2);

Rule.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using CellularAutomata;

namespace Frond
{
    public class Rule
    {
        private int elementaryRule = 0;
        public int ElementaryRule
        {
private int fracturingRule = 0;
public int FracturingRule
{
    get { return fracturingRule; }
    set { fracturingRule = value; }
}

public Rule(int elementaryRule, int fracturingRule)
{
    ElementaryRule = elementaryRule;
    FracturingRule = fracturingRule;
}

public CellState CalculateState(CellState xLeft2, CellState xLeft,
CellState xMiddle, CellState xRight, CellState xRight2)
{
    // Calculate the cell state.
    int elementaryRuleIndex = 0;
    if (xLeft == CellState.On || xLeft == CellState.On_Active)
    {
        elementaryRuleIndex += 4;
    }
    if (xMiddle == CellState.On || xMiddle == CellState.On_Active)
    {
        elementaryRuleIndex += 2;
    }
    if (xRight == CellState.On || xRight == CellState.On_Active)
    {
        elementaryRuleIndex += 1;
    }
    bool state = Convert.ToBoolean(elementaryRule &
Convert.ToInt32(Math.Pow(2.0, Convert.ToDouble(elementaryRuleIndex))));

    // Calculate if the cell is active.
    bool active = false;
    if (xLeft == CellState.Off_Active || xLeft ==
CellState.On_Active)
    {
        int fracturingRuleIndex = 0;
        if (xLeft2 == CellState.On || xLeft2 == CellState.On_Active)
        {
            fracturingRuleIndex += 4;
        }
        if (xLeft == CellState.On || xLeft == CellState.On_Active)
        {
            fracturingRuleIndex += 2;
        }
        if (xMiddle == CellState.On || xMiddle ==
CellState.On_Active)
        {
            fracturingRuleIndex += 1;
        }
        int result = Convert.ToInt32(Math.Pow(3.0,
Convert.ToDouble(fracturingRuleIndex)));
        for (int i = 7; i >= 0; i--)
        {
            if (i != result)
            {
                result -= Convert.ToInt32(Math.Pow(3.0,
Convert.ToDouble(i)));
            }
        }
    }
    // Return the cell state.
    return State; // Use a state enum or custom type for state representation.
}
// Perform the equivalent of a ternary mask.
int leftResult = FracturingRule;
for (int i = 7; i >= 0; i--)
{
    double powerResult = Math.Pow(3.0, Convert.ToDouble(i));
    int dividend = Convert.ToInt32(Math.Floor(leftResult / powerResult));
    if (i != fracturingRuleIndex)
    {
        leftResult -= dividend * Convert.ToInt32(powerResult);
    } else
    {
        if (dividend == 0)
        {
            active = true;
        } else
        {
            active = false;
        }
    }
}
else if (xMiddle == CellState.Off_Active || xMiddle == CellState.On_Active)
{
    int fracturingRuleIndex = 0;
    if (xLeft == CellState.On || xLeft == CellState.On_Active)
    {
        fracturingRuleIndex += 4;
    }
    if (xMiddle == CellState.On || xMiddle == CellState.On_Active)
    {
        fracturingRuleIndex += 2;
    }
    if (xRight == CellState.On || xRight == CellState.On_Active)
    {
        fracturingRuleIndex += 1;
    }
    int result = Convert.ToInt32(Math.Pow(3.0, Convert.ToDouble(fracturingRuleIndex)));
    for (int i = 7; i >= 0; i--)
    {
        if (i != result)
        {
            result -= Convert.ToInt32(Math.Pow(3.0, Convert.ToDouble(i)));
        }
    }
}

// Perform the equivalent of a ternary mask.
int middleResult = FracturingRule;
for (int i = 7; i >= 0; i--)
{
    double powerResult = Math.Pow(3.0, Convert.ToDouble(i));
    int dividend = Convert.ToInt32(Math.Floor(middleResult / powerResult));
    if (i != fracturingRuleIndex)
    {
        }
middleResult = dividend * Convert.ToInt32(powerResult);
} else {
    if (dividend == 1)
        active = true;
    else
        active = false;
}
}

else if (xRight == CellState.Off_Active || xRight == CellState.On_Active)
{
    int fracturingRuleIndex = 0;
    if (xMiddle == CellState.On || xMiddle == CellState.On_Active)
    {
        fracturingRuleIndex += 4;
    }
    if (xRight == CellState.On || xRight == CellState.On_Active)
    {
        fracturingRuleIndex += 2;
    }
    if (xRight2 == CellState.On || xRight2 == CellState.On_Active)
    {
        fracturingRuleIndex += 1;
    }
    // Perform the equivalent of a ternary mask.
    int rightResult = FracturingRule;
    for (int i = 7; i >= 0; i--)
    {
        double powerResult = Math.Pow(3.0, Convert.ToDouble(i));
        int dividend = Convert.ToInt32(Math.Floor(rightResult / powerResult));
        if (i != fracturingRuleIndex)
        {
            rightResult = dividend * Convert.ToInt32(powerResult);
        } else {
            if (dividend == 2)
                active = true;
            else
                active = false;
        }
    }
}

// Determine the cell state.
CellState cellState = CellState.Off;
if (state == true)
{
    if (active == true)
    {
        cellState = CellState.On_Active;
    } else
    {
        cellState = CellState.On;
    }
} else
{
    if (active == true)
    {
        cellState = CellState.Off_Active;
    } else
    {
        cellState = CellState.Off;
    }
}
return cellState;
}
[DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
private ElementType type;
[DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
private List<Pattern> patterns = new List<Pattern>();

// Setters/getters
public ElementType Type
{
    get { return type; }
}

public string Name
{
    get
    {
        return Utilities.GetEnumDescription(type);
    }
}

public List<Pattern> Patterns
{
    get { return patterns; }
}

// Constructor
public Element(string element)
{
    type = LookupElementType(element);
}

public Element(ElementType elementType)
{
    type = elementType;
}

// Public methods
public void AddPattern(Pattern pattern)
{
    patterns.Add(pattern);
}

public static ElementType LookupElementType(string elementType)
{
    ElementType type = ElementType.Intro;
    bool found = false;
    foreach (ElementType enumeratedType in Utilities.EnumToList<ElementType>())
    {
        string enumeratedTypeString = Utilities.GetEnumDescription(enumeratedType);
        if (elementType == enumeratedTypeString)
        {
            type = enumeratedType;
            found = true;
        }
    }
    if (found == false)
    {
        throw new Exception("Unknown element type "+ type + ".");
    }
    return type;
}
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using CellularAutomata;

namespace Frond
{
    public class Measure
    {
        // Private members.
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Chord chord = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private NoteDerivation noteDerivation = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private string measureName = "";
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dynamics dynamics = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private CellularAutomaton cellularAutomaton = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int startingRow = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int numRows = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int divisions = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int staff = 0;

        // Setters/getters.
        public NoteDerivation NoteDerivation
        {
            get { return noteDerivation; }
            set { noteDerivation = value; }
        }
        public string Name
        {
            get { return measureName; }
        }
        public Dynamics Dynamics
        {
            get { return dynamics; }
            set { dynamics = value; }
        }
        public Chord Chord
        {
            get { return chord; }
        }
        public CellularAutomaton CellularAutomaton
        {
            get { return cellularAutomaton; }
        }
    }
}

{
    set { cellularAutomaton = value; }
    get { return cellularAutomaton; }
}
public int StartingRow
{
    set { startingRow = value; }
    get { return startingRow; }
}
public int NumRows
{
    set { numRows = value; }
    get { return numRows; }
}
public int Divisions
{
    get { return divisions; }
    set { divisions = value; }
}
public int Staff
{
    set { staff = value; }
    get { return staff; }
}
public List<Note> Notes
{
    get
    {
        List<Note> notes = new List<Note>();

        // Traverse each row of the CA.
        Note newNote = null;
        int newNoteDuration = 0;
        for (int i = startingRow; i < numRows + startingRow; i++)
        {
            // Traverse each cell in the row to find the active cell.
            int activeCellIndex = -1;
            CellState cellState = CellState.Off;
            for (int j = 0; j < cellularAutomaton.Cells[i].Length; j++)
            {
                cellState = cellularAutomaton.Cells[i][j].State;
                if (cellState == CellState.Off_Active || cellState == CellState.On_Active)
                {
                    activeCellIndex = j;
                    break;
                }
            }
            if (activeCellIndex == -1)
            {
                throw new Exception("Active cell not found in row "+ i.ToString() + " of cellular automaton.");
            }

            // Create a new note if its the start of the measure or the active cell is "on".
            if (i == startingRow || cellState == CellState.On_Active)
            {
                // Use the active cell index to get the note string

```csharp
string[] noteDerivationHeader = GetHeader();
if (activeCellIndex >= noteDerivationHeader.Length)
    throw new Exception("Active cell index exceeds note derivation header length.");
string noteString = noteDerivationHeader[activeCellIndex];

    // Set the duration and add the old note to the list of notes.
    if (newNote != null)
    {
        newNote.Duration = newNoteDuration;
        newNote.Staff = staff;
        notes.Add(newNote);
    }
    // Create the note using the note string.
    newNote = new Note(noteString);
    newNoteDuration = 1;
    } else {
    // It's not the starting row or the cell is not "on", so just increment the duration instead.
    newNoteDuration++;
    }
    // Add the last note to the list of notes.
    if (newNote != null)
    {
        newNote.Duration = newNoteDuration;
        newNote.Staff = staff;
        notes.Add(newNote);
    }
    return notes;
}

public Note LastNotePlayed
{
    get
    {
        List<Note> notes = Notes;
        Note lastNote = notes[notes.Count - 1];
        return lastNote;
    }
}

    // Constructors.
    public Measure(string measureName, Chord chord)
    {
        this.measureName = measureName;
        this.chord = chord;
    }
    public Measure(Measure measure)
    {
        if (measure.chord != null)
        {
            this.chord = new Chord(measure.chord);  // Deep copy.
        }
    }
```
else
    {this.chord = null;
    }this.measureName = measure.measureName;
if (measure.dynamics != null)
    {this.dynamics = new Dynamics(measure.dynamics); // Deep
    }else
    {this.dynamics = null;
    }this.noteDerivation = measure.noteDerivation;
}

// Public methods.
public Note.Pitch CalculateRootNotePitch(Key key)
{
    ChordInterval[] chordIntervals =
    chord.CalculateAbsoluteIntervals(key.KeyIntervals);
    return (key.GetNoteUsingSemitone(chordIntervals[0].Semitone));
}

public string[] GetHeader()
{
    string[] header = new string[CellularAutomaton.DEFAULT_CA_WIDTH];

    // Determine where the active cell is at.
    Cell[] row =
    cellularAutomaton.Cells[startingRow].ToArray<Cell>();
    int activeCellIndex = -1;
    for (int i = 0; i < row.Length; i++)
    {
        if (row[i].State == CellState.On_Active || row[i].State ==
            CellState.Off_Active)
        {
            activeCellIndex = i;
            break;
        }
    }
    if (activeCellIndex == -1)
    {
        throw new Exception("Failed to get header for measure.
        Active cell not found in row.");
    }

    // Copy the notes to the string array, starting from left to
    right.
    int shift = NoteDerivation.CENTER_NOTE_INDEX - activeCellIndex;
    shift *= -1; // Cells left of the active cell are negative,
    right positive.
    for (int i = 0; i < CellularAutomaton.DEFAULT_CA_WIDTH; i++)
    {
        int sourceIndex = (i + NoteDerivation.CENTER_NOTE_INDEX) %
            CellularAutomaton.DEFAULT_CA_WIDTH;
        int destinationIndex = (i + shift +
            NoteDerivation.CENTER_NOTE_INDEX) % CellularAutomaton.DEFAULT_CA_WIDTH;
        header[destinationIndex] =
            noteDerivation[sourceIndex].ToString();
public string ToXML()
{
    string xmlOutput = "";
    List<Note> notes = Notes;
    foreach (Note note in notes)
    {
        xmlOutput += note.ToXML();
    }
    return xmlOutput;
}

Part.cs
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class Part
    {
        // Private data members.
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private string name = "";
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private List<Measure> measures = new List<Measure>();
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private CellularAutomaton cellularAutomaton = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private bool alwaysPlayRootNote = false;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Key key = null;

        // Setters/getters
        public CellularAutomaton CellularAutomaton
        {
            set
            {
                cellularAutomaton = value;
                cellularAutomaton.Process();
            }
            get { return cellularAutomaton; }
        }
        public string Name
        {
            get { return name; }
        }
    }
}
```csharp
public List<Measure> Measures
{
    get { return measures; }
}

public bool AlwaysPlayRootNote
{
    get { return alwaysPlayRootNote; }
    set { alwaysPlayRootNote = value; }
}

public Key Key
{
    get { return key; }
    set { key = value; }
}

// Constructors.
public Part(string name)
{
    this.name = name;
}

// Public methods.
public void AddMeasure(Measure measure)
{
    measures.Add(measure);
}

public void CalculateNoteDerivations(Dictionary<string, KeyInterval[]> keyIntervalMap)
{
    int count = measures.Count;
    // Determine the note derivations for each measure.
    Measure previousMeasure = null;
    for (int i = 0; i < count; i++)
    {
        Measure measure = measures[i];
        // Only calculate a note derivation if the measure doesn't
        // already have one.
        if (measure.NoteDerivation == null)
        {
            KeyInterval[] keyIntervals = keyIntervalMap[measure.Name];
            // Check the always play root note flag.
            if (alwaysPlayRootNote == true)
            {
                // Determine the root note of the chord.
                int octave = Note.CalculateDefaultOctave(name);
                Note.Pitch rootNotePitch = measure.CalculateRootNotePitch(key);
                Note rootNote = new Note(rootNotePitch, octave);
                // Create the note derivation.
                NoteDerivation noteDerivation = new
                NoteDerivation(rootNote);
                measure.NoteDerivation = noteDerivation;
            }
            else
            {
                // This is the first measure, so center the note
derivation on the root note of the chord.
            }
        }
    }
    
```
if (previousMeasure == null)
{
    // Determine the root note of the chord.
    int octave = Note.CalculateDefaultOctave(name);
    int nominalOctave =
    Note.CalculateDefaultOctave(name);
    Note rootNote = new Note(rootNotePitch, octave);
    Note rootNote = new Note(rootNotePitch, octave);
    int rootNoteSemitone =
    key.GetSemitoneOfNote(rootNote);
    // Create the note derivation.
    NoteDerivation noteDerivation = new
    NoteDerivation(rootNoteSemitone, octave, nominalOctave,
    key, keyIntervals,
    measure.Chord.CalculateAbsoluteIntervals(key.KeyIntervals));
    measure.NoteDerivation = noteDerivation;
}
else
{
    // Get the last note played of the previous
    measure.
    lastNotePlayed =
    previousMeasure.LastNotePlayed;
    int nominalOctave =
    Note.CalculateDefaultOctave(name);
    int lastNoteSemitone =
    key.GetSemitoneOfNote(lastNotePlayed);
    int octave = lastNotePlayed.Octave;
    // Create the note derivation.
    NoteDerivation noteDerivation = new
    NoteDerivation(lastNoteSemitone, octave, nominalOctave,
    key, keyIntervals,
    measure.Chord.CalculateAbsoluteIntervals(key.KeyIntervals));
    measure.NoteDerivation = noteDerivation;
}

// Set the previous measure.
previousMeasure = measure;
}
public bool MeasureExists(string measureName)
{
    bool measureExists = false;
    foreach (Measure measure in measures)
    {
        if (measure.Name == measureName)
        {
            measureExists = true;
            break;
        }
    }

    return measureExists;
}

namespace Frond
{
    public class Pattern
    {
        // Private data members.
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private List<Measure> measures = new List<Measure>();
        private string name;
        private Dynamics dynamics;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int repeats;

        // Setters/getters.
        public string Name
        {
            get { return name; } 
        }
        public List<Measure> Measures
        {
            get { return measures; } 
        }
        public Dynamics Dynamics
        {
            get { return dynamics; } 
        }
        public int Repeats
        {
            get { return repeats; } 
        }
    }
}
get { return repeats; }
}

// Constructors.
public Pattern(string name, string dynamics, int repeats)
{
    this.name = name;
    this.dynamics = new Dynamics(dynamics);
    this.repeats = repeats;
}
public Pattern(string name, Dynamics dynamics, int repeats)
{
    this.name = name;
    this.dynamics = dynamics;
    this.repeats = repeats;
}

// Public methods.
public void AddMeasure(Measure measure)
{
    measures.Add(measure);
}

Profile.cs

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class Profile : System.Collections.IEnumerable
    {
    // Private data members.
    private List<Element> elements = new List<Element>();

    // Constructors.
    public Profile()
    {
    }

    // Public methods.
    public Element FindElement(string elementName)
    {
        Element elementRef = null;
        foreach (Element element in elements)
        {
            if (element.Name == elementName)
            {
                elementRef = element;
            }
        }
    }
}
if (elementRef == null)
{
    throw new Exception("Unable to find element with name '" + elementName + ":";)
}

return elementRef;

public void AddElement(Element element)
{
    elements.Add(element);
}

public System.Collections.IEnumerator GetEnumerator()
{
    for (int i = 0; i < elements.Count; i++)
    {
        yield return elements[i];
    }
}

Chord.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Frond
{
    public class Chord
    {
        public enum ChordType
        {
            [Description("Major")]
            Major,
            [Description("Minor")]
            Minor,
            [Description("Diminished")]
            Diminished,
            [Description("Diminished 7th")]
            DiminishedSeventh,
            [Description("Half diminished")]
            HalfDiminished,
            [Description("Augmented")]
            Augmented,
            [Description("5th")]
            Fifth,
            [Description("7th")]
            Seventh,
            [Description("Minor 7th")]
        }
    }
}
MinorSeventh,
[Description("Major 7th")]
MajorSeventh,
[Description("Minor/major 7th")]
MinorMajorSeventh,
[Description("Suspened 4th")]
SuspendedFourth,
[Description("Suspened 2nd")]
SuspendedSecond,
[Description("7th suspended 4th")]
SeventhSuspendedFourth,
[Description("7th suspended 2nd")]
SeventhSuspendedSecond,
[Description("Added 2nd")]
AddedSecond,
[Description("Added 9th")]
AddedNinth,
[Description("Added 4th")]
AddedFourth,
[Description("6th")]
Sixth,
[Description("Minor 6th")]
MinorSixth,
[Description("6/9")]
SixNine,
[Description("9th")]
Ninth,
[Description("Minor 9th")]
MinorNinth,
[Description("Major 9th")]
MajorNinth,
[Description("11th")]
Eleventh,
[Description("Minor 11th")]
MinorEleventh,
[Description("Major 11th")]
MajorEleventh,
[Description("13th")]
Thirteenth,
[Description("Minor 13th")]
MinorThirteenth,
[Description("Major 13th")]
MajorThirteenth,
[Description("7th sharp 9th")]
SeventhSharpNinth,
[Description("7th flat 9th")]
SeventhFlatNinth,
[Description("7th sharp 5th")]
SeventhSharpFifth,
[Description("7th flat 5th")]
SeventhFlatFifth
}

public enum ScaleDegree
{
    I,
    II,
    III,
    IV,
    V,
    VI,
    VII
}
private ChordType type;
private ScaleDegree degree;
private static Dictionary<ChordType, string> notations = new Dictionary<ChordType, string>();
private static Dictionary<ChordType, string[]> spellings = new Dictionary<ChordType, string[]>();
private static Dictionary<ChordType, ChordInterval[]> intervals = new Dictionary<ChordType, ChordInterval[]>();

public ChordType Type
{
    get { return type; }
    set { type = value; }
}

public ScaleDegree Degree
{
    get { return degree; }
    set { degree = value; }
}

public int DegreeAsInt
{
    get
    {
        int scaleDegree = 0;
        switch (degree)
        {
            case ScaleDegree.I:
                scaleDegree = 1;
                break;
            case ScaleDegree.II:
                scaleDegree = 2;
                break;
            case ScaleDegree.III:
                scaleDegree = 3;
                break;
            case ScaleDegree.IV:
                scaleDegree = 4;
                break;
            case ScaleDegree.V:
                scaleDegree = 5;
                break;
            case ScaleDegree.VI:
                scaleDegree = 6;
                break;
            case ScaleDegree.VII:
                scaleDegree = 7;
                break;
            default:
                throw new Exception("Unable to convert scale degree to integer.");
        }
        return scaleDegree;
    }
}

public string[] Intervals
{
    get { return spellings[type]; }
}

public ChordInterval[] RelativeIntervals
{
public override string ToString()
{
    string notation = "";
    notation += degree.ToString();
    notation += " ";
    notation += notations[Type];
    return notation;
}

// Constructors
static Chord()
{
    // Initialize the notation dictionary.
    #region Initialize notation dictionary
    notations.Add(ChordType.Major, "M");
    notations.Add(ChordType.Minor, "m");
    notations.Add(ChordType.Diminished, "d");
    notations.Add(ChordType.DiminishedSeventh, "d7");
    notations.Add(ChordType.HalfDiminished, "m7b5");
    notations.Add(ChordType.Augmented, "A");
    notations.Add(ChordType.Fifth, "5");
    notations.Add(ChordType.Seventh, "7");
    notations.Add(ChordType.MinorSeventh, "m7");
    notations.Add(ChordType.MajorSeventh, "M7");
    notations.Add(ChordType.MinorMajorSeventh, "m/Maj7");
    notations.Add(ChordType.SuspendedFourth, "sus4");
    notations.Add(ChordType.SuspendedSecond, "sus2");
    notations.Add(ChordType.SeventhSuspendedFourth, "7sus4");
    notations.Add(ChordType.SeventhSuspendedSecond, "7sus2");
    notations.Add(ChordType.AddedSecond, "add2");
    notations.Add(ChordType.AddedNinth, "add9");
    notations.Add(ChordType.AddedFourth, "add4");
    notations.Add(ChordType.Sixth, "6");
    notations.Add(ChordType.MinorSixth, "m6");
    notations.Add(ChordType.SixNine, "6/9");
    notations.Add(ChordType.Ninth, "9");
    notations.Add(ChordType.MinorNinth, "m9");
    notations.Add(ChordType.MajorNinth, "M9");
    notations.Add(ChordType.Eleventh, "11");
    notations.Add(ChordType.MinorEleventh, "m11");
    notations.Add(ChordType.MajorEleventh, "M11");
    notations.Add(ChordType.Thirteenth, "13");
    notations.Add(ChordType.MinorThirteenth, "m13");
    notations.Add(ChordType.MajorThirteenth, "M13");
    notations.Add(ChordType.SeventhSharpNinth, "7#9");
    notations.Add(ChordType.SeventhFlatNinth, "7b9");
    notations.Add(ChordType.SeventhSharpFifth, "7#5");
    notations.Add(ChordType.SeventhFlatFifth, "7b5");
    #endregion

    // Initialize the spelling dictionary.
    #region Initialize spelling dictionary
    spellings.Add(ChordType.Major, new string[] { "1", "3", "5" });
    spellings.Add(ChordType.Minor, new string[] { "1", "b3", "5" });
    spellings.Add(ChordType.Diminished, new string[] { "1", "b3", "b5" });
    spellings.Add(ChordType.DiminishedSeventh, new string[] { "1", "b3", "b5", "bb7" });
    spellings.Add(ChordType.HalfDiminished, new string[] { "1", "b3", "b5", "b7" });
    //endregion
}
spellings.Add(ChordType.Augmented, new string[] { "1", "3", "#5" });
spellings.Add(ChordType.Fifth, new string[] { "1", "5" });
spellings.Add(ChordType.Seventh, new string[] { "1", "3", "5", "b7" });
spellings.Add(ChordType.MinorSeventh, new string[] { "1", "b3", "5", "b7" });
spellings.Add(ChordType.MajorSeventh, new string[] { "1", "3", "5", "7" });
spellings.Add(ChordType.MinorMajorSeventh, new string[] { "1", "b3", "5", "7" });
spellings.Add(ChordType.SuspendedFourth, new string[] { "1", "4", "5" });
spellings.Add(ChordType.SuspendedSecond, new string[] { "1", "2", "5" });
spellings.Add(ChordType.SeventhSuspendedFourth, new string[] { "1", "4", "5", "b7" });
spellings.Add(ChordType.SeventhSuspendedSecond, new string[] { "1", "2", "5", "b7" });
spellings.Add(ChordType.AddedSecond, new string[] { "1", "2", "3", "5" });
spellings.Add(ChordType.AddedNinth, new string[] { "1", "3", "5", "9" });
spellings.Add(ChordType.AddedFourth, new string[] { "1", "3", "4", "5" });
spellings.Add(ChordType.Sixth, new string[] { "1", "3", "5", "6" });
spellings.Add(ChordType.MinorSixth, new string[] { "1", "b3", "5", "6" });
spellings.Add(ChordType.SixNine, new string[] { "1", "3", "5", "6", "9" });
spellings.Add(ChordType.Ninth, new string[] { "1", "3", "5", "b7", "9" });
spellings.Add(ChordType.MinorNinth, new string[] { "1", "b3", "5", "b7", "9" });
spellings.Add(ChordType.MajorNinth, new string[] { "1", "3", "5", "7", "9" });
spellings.Add(ChordType.Eleventh, new string[] { "1", "3", "5", "b7", "9", "11" });
spellings.Add(ChordType.MinorEleventh, new string[] { "1", "b3", "5", "b7", "9", "11" });
spellings.Add(ChordType.MajorEleventh, new string[] { "1", "3", "5", "7", "9", "11" });
spellings.Add(ChordType.MinorThirteenth, new string[] { "1", "b3", "5", "b7", "9", "11", "13" });
spellings.Add(ChordType.MajorThirteenth, new string[] { "1", "3", "5", "7", "9", "11", "13" });
spellings.Add(ChordType.SeventhSharpNinth, new string[] { "1", "3", "5", "7", "9", "11", "13", "#5" });
spellings.Add(ChordType.SeventhFlatNinth, new string[] { "1", "3", "5", "7", "9", "11", "13", "b9" });
spellings.Add(ChordType.SeventhSharpFifth, new string[] { "1", "3", "5", "7", "9", "11", "13", "#5", "b7" });
spellings.Add(ChordType.SeventhFlatFifth, new string[] { "1", "3", "5", "7", "9", "11", "13", "b9", "b7" });

// Initialize the intervals dictionary.
#endregion
#region Initialize intervals dictionary
new ChordInterval(1, 0, 0);
intervals.Add(ChordType.Major, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Minor, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Diminished, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Augmented, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Fifth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Seventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorSeventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MajorSeventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorMajorSeventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SuspendedFourth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SuspendedSecond, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhSuspendedFourth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhSuspendedSecond, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.AddedSecond, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.AddedNinth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.AddedFourth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Sixth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorSixth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SixNine, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Ninth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorNinth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MajorNinth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Eleventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorEleventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MajorEleventh, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.Thirteenth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MinorThirteenth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.MajorThirteenth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhSharpNinth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhFlatNinth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhSharpFifth, new ChordInterval[] { {1, 0, 0} });
intervals.Add(ChordType.SeventhFlatFifth, new ChordInterval[] { {1, 0, 0} });
#endregion
}
new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0));
intervals.Add(ChordType.Minor, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.Diminished, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 1, 0)});
intervals.Add(ChordType.DiminishedSeventh, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 1, 0), new ChordInterval(7, 2, 0)});
intervals.Add(ChordType.HalfDiminished, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 1, 0), new ChordInterval(7, 1, 0)});
intervals.Add(ChordType.Augmented, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 1)});
intervals.Add(ChordType.Fifth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.Seventh, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0)});
intervals.Add(ChordType.MinorSeventh, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0)});
intervals.Add(ChordType.MajorSeventh, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(7, 0, 0)});
intervals.Add(ChordType.MinorMajorSeventh, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new ChordInterval(5, 0, 0), new ChordInterval(7, 0, 0)});
intervals.Add(ChordType.SuspendedFourth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(4, 0, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.SuspendedSecond, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(2, 0, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.SeventhSuspendedFourth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(4, 0, 0), new ChordInterval(7, 0, 0)});
intervals.Add(ChordType.SeventhSuspendedSecond, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(2, 0, 0), new ChordInterval(7, 1, 0)});
intervals.Add(ChordType.AddedSecond, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(2, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.AddedFourth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(4, 0, 0), new ChordInterval(5, 0, 0)});
intervals.Add(ChordType.AddedNinth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(9, 0, 0)});
intervals.Add(ChordType.AddedSixth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(6, 0, 0)});
intervals.Add(ChordType.AddedNinth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(9, 0, 0)});
intervals.Add(ChordType.AddedSixth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(6, 0, 0)});
intervals.Add(ChordType.MinorSixth, new ChordInterval[]{
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new ChordInterval(5, 0, 0), new ChordInterval(6, 0, 0)});
new ChordInterval(9, 0, 0));
intervals.Add(ChordType.Ninth, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0),
    new ChordInterval(9, 0, 0)});
intervals.Add(ChordType.MinorNinth, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0),
    new ChordInterval(9, 0, 0)});
intervals.Add(ChordType.MajorNinth, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 0, 0),
    new ChordInterval(9, 0, 0)});
intervals.Add(ChordType.Eleventh, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0),
    new ChordInterval(9, 0, 0), new ChordInterval(11, 0, 0)});
intervals.Add(ChordType.MinorEleventh, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 1, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0),
    new ChordInterval(9, 0, 0), new ChordInterval(11, 0, 0)});
intervals.Add(ChordType.MajorEleventh, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 0, 0),
    new ChordInterval(9, 0, 0), new ChordInterval(11, 0, 0),
    new ChordInterval(13, 0, 0)});
intervals.Add(ChordType.SeventhSharpNinth, new ChordInterval[] {
    new ChordInterval(1, 0, 0), new ChordInterval(3, 0, 0), new
    ChordInterval(5, 0, 0), new ChordInterval(7, 1, 0),
    new ChordInterval(9, 0, 1)});
{  type = chordType;
   degree = scaleDegree;
}
public Chord(Chord chord)
{
   this.degree = chord.degree;
   this.type = chord.type;
}

// Public methods
public static ChordType LookupChordType(string chordType)
{
   ChordType type = ChordType.Major;
   bool found = false;
   foreach (ChordType enumeratedType in Utilities.EnumToList<ChordType>())
   {
      string enumeratedTypeString = Utilities.GetEnumDescription(enumeratedType);
      if (chordType == enumeratedTypeString || chordType == notations[enumeratedType])
      {
         type = enumeratedType;
         found = true;
      }
   }
   if (found == false)
   {
      throw new Exception("Unknown chord type "+ type + ",."");
   }
   return type;
}
public static string LookupChordTypeNotation(string chordType)
{
   ChordType type = LookupChordType(chordType);
   return LookupChordTypeNotation(type);
}
public static string LookupChordTypeNotation(ChordType chordType)
{
   return notations[chordType];
}
public static ScaleDegree LookupScaleDegree(string scaleDegree)
{
   ScaleDegree degree = ScaleDegree.I;
   bool found = false;
   foreach (ScaleDegree enumeratedType in Utilities.EnumToList<ScaleDegree>())
   {
      string enumeratedTypeString = Utilities.GetEnumDescription(enumeratedType);
      if (scaleDegree == enumeratedTypeString)
      {
         degree = enumeratedType;
         found = true;
      }
   }
   if (found == false)
   {
      throw new Exception("Unknown scale degree "+ scaleDegree + ",."");
   }
   return degree;
}
if (found == false) {
    throw new Exception("Unknown scale degree "+ degree + ":");
}

return degree;

public ChordInterval[] CalculateAbsoluteIntervals(KeyInterval[] keyIntervals) {
    ChordInterval[] relativeIntervals = RelativeIntervals;
    int count = relativeIntervals.Length;

    // Calculate the offset between relative and absolute.
    int scaleDegree = DegreeAsInt;
    int keyIntervalIndex = scaleDegree - 1;
    if (scaleDegree > keyIntervals.Length || scaleDegree <= 0) {
        throw new Exception("Scale degree "+ scaleDegree.ToString() + ":" + keyIntervals.Length + scaleDegree <= 0);}
    int offset = keyIntervals[keyIntervalIndex].Semitone;

    // Create the absolute intervals.
    ChordInterval[] absoluteIntervals = new ChordInterval[count];
    for (int i = 0; i < count; i++) {
        int intervalNumber = relativeIntervals[i].IntervalNumber + (scaleDegree - 1);
        if (intervalNumber > Key.NOTES_IN_SCALE) {
            intervalNumber = intervalNumber % Key.NOTES_IN_SCALE;
        }
        int semitone = (relativeIntervals[i].Semitone + offset) % Key.NUM_SEMITONES_IN_SCALE;
        absoluteIntervals[i] = new ChordInterval(intervalNumber, semitone);
    }
    return absoluteIntervals;
}

public static int ChordSpellingToKeySemitone(string spelling, int scaleDegree, Key key) {
    int flatAdjustment = -1 * Utilities.CountCharactersInString(spelling, 'b');
    int sharpAdjustment = Utilities.CountCharactersInString(spelling, '#');
    int totalAdjustment = flatAdjustment + sharpAdjustment;

    // Ensure the total adjustment doesn't exceed one octave above or below. If so, throw an exception.
    if (totalAdjustment >= Key.NUM_SEMITONES_IN_SCALE ||
        totalAdjustment < (-1 * Key.NUM_SEMITONES_IN_SCALE)) {
        throw new Exception("Semitonal adjustment of "+ totalAdjustment.ToString() + ":");
    }

    // Determine any adjustments in semitone due to flats and sharps.
    int intervalNumber = (scaleDegree - 1) % Key.NOTES_IN_SCALE;
    int semitone = (relativeIntervals[i].Semitone + offset) % Key.NUM_SEMITONES_IN_SCALE;
    absoluteIntervals[i] = new ChordInterval(intervalNumber, semitone);
    return absoluteIntervals;
}
// Remove any flats/sharps from the spelling, since they're already accounted for.
spelling = spelling.Replace("b", ");
spelling = spelling.Replace("#", ");

// Calculate the scale degree of this note using the chord scale degree and the interval of the note.
int semitone = -1;
try {
    // Ensure the scale degree doesn't exceed the max number of scale degrees.
    scaleDegree = scaleDegree % Utilities.EnumToList<ScaleDegree>().ToList<ScaleDegree>().Count;
    try {
        int interval = Convert.ToInt32(spelling);
        if (interval > Key.NOTES_IN_SCALE) {
            // Ensure the interval does not exceed 7.
            interval = interval % Key.NOTES_IN_SCALE;
        }
        int intervalAdjustment = 0;
        switch (interval) {
            case 1:
                intervalAdjustment = 0;
                break;
            case 2:
                intervalAdjustment = 2;
                break;
            case 3:
                intervalAdjustment = 4;
                break;
            case 4:
                intervalAdjustment = 5;
                break;
            case 5:
                intervalAdjustment = 7;
                break;
            case 6:
                intervalAdjustment = 9;
                break;
            case 7:
                intervalAdjustment = 11;
                break;
            default:
                throw new Exception("Unable to converting chord interval '" + spelling + "' to semitones.");
        }
        int semitone = Key.GetSemitoneOfScaleDegree(scaleDegree);
        semitone += intervalAdjustment;
        totalAdjustment += semitone;
        if (semitone < 0) {
            semitone += Key.NUM_SEMITONES_IN_SCALE;
        }
    }
}
semitone = semitone % Key.NUM_SEMITONES_IN_SCALE;
} catch (Exception e)
{
    throw new Exception("Unable to convert chord spelling '\" + spelling + '\" to chord interval.");
} catch (Exception e)
{
    throw new Exception("Error converting chord spelling '\" + spelling + '\" into semitone.");
}
return semitone;
}

public Note.Pitch[] GetNotes(Key key)
{
    // Copy the scale degrees into a list.
    List<ScaleDegree> scaleDegrees = Utilities.EnumToList<ScaleDegree>().ToList<ScaleDegree>();
    // Determine the scale degree.
    int scaleDegree = scaleDegrees.IndexOf(degree);
    if (scaleDegree == -1)
    {
        throw new Exception("Unknown scale degree '\" + scaleDegree.ToString() + '\" during chord GetNotes.");
    }
    // Determine the number of notes in our chord.
    string[] chordSpellings = spellings[type];
    int numNotesInChord = chordSpellings.Length;
    // Create an array of notes for the chord.
    Note.Pitch[] notes = new Note.Pitch[numNotesInChord];
    for (int i = 0; i < numNotesInChord; i++)
    {
        string spelling = chordSpellings[i];
        int semitone = ChordSpellingToKeySemitone(spelling, scaleDegree, key);
        notes[i] = key.GetNoteUsingSemitone(semitone);
    }
    return notes;
}

public int GetRelativeOffset()
{
    int scaleDegree = 0;
    switch (degree)
    {
        case ScaleDegree.I:
            scaleDegree = 0;
            break;
        case ScaleDegree.II:
            scaleDegree = 1;
            break;
        case ScaleDegree.III:
            scaleDegree = 2;
            break;
        case ScaleDegree.IV:
            scaleDegree = 3;
            break;
        // Add cases for other scale degrees here...
    }
    return scaleDegree;
}
case ScaleDegree.V:
    scaleDegree = 4;
    break;
case ScaleDegree.VI:
    scaleDegree = 5;
    break;
case ScaleDegree.VII:
    scaleDegree = 6;
    break;
default:
    throw new Exception("Unable to convert scale degree to integer.");
}

return scaleDegree;

ChordInterval.cs

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class ChordInterval
    {
        // Private data members.
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int intervalNumber = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int semitone = 0;

        // Setters/getters.
        public int IntervalNumber
        {
            get { return intervalNumber; }
        }
        public int Semitone
        {
            get { return semitone; }
        }

        // Constructors.
        public ChordInterval(int intervalNumber, int semitone)
        {
            this.intervalNumber = intervalNumber;
            this.semitone = semitone;
        }
        public ChordInterval(int intervalNumber, int numFlats, int numSharps)
this.intervalNumber = intervalNumber;

// Determine the semitone.
switch (intervalNumber) {
    case 1:
    case 8:
        semitone = 0;
        break;
    case 2:
    case 9:
        semitone = 2;
        break;
    case 3:
    case 10:
        semitone = 4;
        break;
    case 4:
    case 11:
        semitone = 5;
        break;
    case 5:
    case 12:
        semitone = 7;
        break;
    case 6:
    case 13:
        semitone = 9;
        break;
    case 7:
    case 14:
        semitone = 11;
        break;
    default:
        throw new Exception("Error converting chord interval
into semitone.");
}

this.semitone += (-1 * numFlats) + numSharps;

// Public methods.
public override string ToString()
{
    string buffer = "";

    buffer += "Interval: " + intervalNumber.ToString();
    buffer += ", Semitone: " + semitone.ToString();
    return buffer;
}
Dynamics.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class Dynamics
    {
        // Private data members.
        public enum DynamicsType
        {
            Pianissimo,
            Piano,
            [Description("Mezzo-piano")]
            MezzoPiano,
            [Description("Mezzo-forte")]
            MezzoForte,
            Forte,
            Fortissimo,
        }
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private DynamicsType type;
        private static Dictionary<DynamicsType, string> notations = new Dictionary<DynamicsType, string>();
        private static Dictionary<DynamicsType, string> volumes = new Dictionary<DynamicsType, string>();

        // Setters/getters.
        public DynamicsType Type
        {
            get { return type; }
            set { type = value; }
        }

        // Constructors.
        static Dynamics()
        {
            // Initialize dynamics dictionary.
            #region Initialize dynamics dictionary
            notations.Add(DynamicsType.Pianissimo, "pp");
            notations.Add(DynamicsType.Piano, "p");
            notations.Add(DynamicsType.MezzoPiano, "mp");
            notations.Add(DynamicsType.MezzoForte, "mf");
            notations.Add(DynamicsType.Forte, "f");
            notations.Add(DynamicsType.Fortissimo, "ff");
            #endregion

            // Initialize volumes dictionary.
            #region Initialize volumes dictionary
            volumes.Add(DynamicsType.Pianissimo, "30");
            volumes.Add(DynamicsType.Piano, "45");
            volumes.Add(DynamicsType.MezzoPiano, "60");
            volumes.Add(DynamicsType.MezzoForte, "70");
            volumes.Add(DynamicsType.Forte, "85");
            volumes.Add(DynamicsType.Fortissimo, "100");
            #endregion
        }
    }
}
public Dynamics(DynamicsType dynamicsType)
{
    type = dynamicsType;
}
public Dynamics(string dynamicsType)
{
    type = LookupDynamicsType(dynamicsType);
}
public Dynamics(Dynamics dynamics)
{
    this.type = dynamics.type;
}

// Public methods
public static DynamicsType LookupDynamicsType(string dynamicsType)
{
    DynamicsType type = DynamicsType.Pianissimo;
    bool found = false;
    foreach (DynamicsType type in Utilities.EnumToList<DynamicsType>())
    {
        string enumeratedTypeString = Utilities.GetEnumDescription(type);
        if (dynamicsType == enumeratedTypeString || dynamicsType == notations[enumeratedType])
        {
            type = enumeratedType;
            found = true;
        }
    }
    if (found == false)
    {
        throw new Exception("Unknown dynamics type "+ type + ".");
    }
    return type;
}
public static string LookupDynamicsNotation(string dynamicsType)
{
    return LookupDynamicsNotation(type);
}
public static string LookupDynamicsNotation(DynamicsType dynamicsType)
{
    return notations[dynamicsType];
}
public static string LookupDynamicsVolume(DynamicsType dynamicsType)
{
    return volumes[dynamicsType];
}
public override string ToString()
{
    return Utilities.GetEnumDescription(type);
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Frond
{
    public class Key
    {
        // Enums
        public enum KeyScale
        {
            Major,
            Minor
        }

        // Constants.
        public const Note.Pitch DEFAULT_ROOT = Note.Pitch.C;
        public const KeyScale DEFAULT_SCALE = KeyScale.Major;
        public const int NOTES_IN_SCALE = 7;
        public const int NUM_SEMITONES_IN_SCALE = 12;

        // Private data members.
        private Note.Pitch root = DEFAULT_ROOT;
        private KeyScale scale = DEFAULT_SCALE;
        private int[] majorSemitones = new int[] { 0, 2, 4, 5, 7, 9, 11 };
        private string[] majorIntervals = new string[] { "1", "2", "3", "4", "5", "6", "7" };
        private int[] minorSemitones = new int[] { 0, 2, 3, 5, 7, 8, 10 };
        private string[] minorIntervals = new string[] { "1", "2", "b3", "4", "5", "b6", "b7" };
        private static Dictionary<KeyScale, KeyInterval[]> intervals = new Dictionary<KeyScale, KeyInterval[]>();

        // Constructors
        static Key()
        {
            intervals.Add(KeyScale.Major, new KeyInterval[] {
                new KeyInterval(1, KeyScale.Major), new KeyInterval(2, KeyScale.Major), new KeyInterval(3, KeyScale.Major),
                new KeyInterval(4, KeyScale.Major), new KeyInterval(5, KeyScale.Major),
                new KeyInterval(6, KeyScale.Major), new KeyInterval(7, KeyScale.Major)});
            intervals.Add(KeyScale.Minor, new KeyInterval[] {
                new KeyInterval(1, KeyScale.Minor), new KeyInterval(2, KeyScale.Minor),
                new KeyInterval(3, KeyScale.Minor), new KeyInterval(4, KeyScale.Minor),
                new KeyInterval(5, KeyScale.Minor), new KeyInterval(6, KeyScale.Minor),
                new KeyInterval(7, KeyScale.Minor)});
        }

        public Key(Note.Pitch root, KeyScale scale)
{  
    this.root = root;
    this.scale = scale;
}

// Setters/getters
public Note.Pitch Root
{
    get
    {
        return root;
    }
    set
    {
        root = value;
    }
}

public KeyScale Scale
{
    get
    {
        return scale;
    }
    set
    {
        scale = value;
    }
}

public Note.Pitch[] Notes
{
    get
    {
        // Get the semitones for the scale.
        int[] semitonesInScale = GetSemitonesUsedInScale();
        // Convert the semitones into a boolean array.
        bool[] semitonesUsedInScale = new bool[NUM_SEMITONES_IN_SCALE];
        for (int i = 0; i < semitonesUsedInScale.Length; i++)
        {
            // Set to true if the semitone is used. Set to false otherwise.
            semitonesUsedInScale[i] = semitonesInScale.Contains<int>(i);
        }
        // Get a list of pitches starting with root note.
        List<Note.Pitch> pitchList = GetAllPitchesFromRoot();
        // Create the array of notes.
        Note.Pitch[] notes = new Note.Pitch[NOTES_IN_SCALE];
        if (pitchList.Count == semitonesUsedInScale.Length)
        {
            int noteIndex = 0;
            for (int i = 0; i < pitchList.Count; i++)
            {
                if (semitonesUsedInScale[i] == true)
                {
                    notes[noteIndex] = pitchList[i];
                    noteIndex++;
                }
            }
        }
    }
}
else
{
    throw new Exception("Pitches differ from semitones used
in scale!");
}
return notes;
}

public string[] Intervals
{
    get
    {
        string[] intervals = null;
        switch (scale)
        {
            case KeyScale.Major:
                intervals = majorIntervals;
                break;
            case KeyScale.Minor:
                intervals = minorIntervals;
                break;
            default:
                throw new Exception("Invalid key scale on intervals
call.");
        }
        return intervals;
    }
}

public KeyInterval[] KeyIntervals
{
    get { return intervals[scale]; }
}

// Public methods.
public static Key FromString(string root, string scale)
{
    Key key = new Key(Note.FromString(root), FromString(scale));
    return key;
}

public static KeyScale FromString(string scale)
{
    KeyScale keyScale = KeyScale.Major;
    bool found = false;
    foreach (KeyScale enumeratedType in
Utilities.EnumToList<KeyScale>())
    {
        string enumeratedTypeString =
Utilities.GetEnumDescription(enumeratedType);
        if (scale == enumeratedTypeString)
        {
            keyScale = enumeratedType;
            found = true;
        }
    }
    if (found == false)
    {
        throw new Exception("Error converting string '|' + scale + '|' to key scale!");
    }
return keyScale;
}
public override string ToString()
{
    Note.Pitch[] notes = Notes;
    string buffer = "";
    foreach (Note.Pitch note in notes)
    {
        if (buffer != "")
        {
            buffer += " | ";
        }
        buffer += Utilities.GetEnumDescription(note);
    }
    return buffer;
}
public Note.Pitch GetNoteUsingScaleDegree(int scaleDegree)
{
    // Get the note for a given scale degree.
    Note.Pitch[] notes = Notes;
    return notes[scaleDegree];
}
public int GetSemitoneOfScaleDegree(int scaleDegree)
{
    // Get the semitones for the scale.
    int[] semitonesInScale = GetSemitonesUsedInScale();
    return semitonesInScale[scaleDegree];
}
public int GetSemitoneOfNote(Note note)
{
    Note.Pitch notePitch = note.NotePitch;
    // Get all possible pitches starting with the root note.
    List<Note.Pitch> pitches = new List<Note.Pitch>();
    for (int i = 0; i < NUM_SEMITONES_IN_SCALE; i++)
    {
        pitches.Add(GetNoteUsingSemitone(i));
    }
    // Remove pitches from the list until we find the notePitch.
    int semitone = 0;
    int count = pitches.Count;
    for (int i = 0; i < count; i++)
    {
        Note.Pitch currentPitch = pitches[0];
        if (currentPitch == notePitch)
        {
            break;
        }
        else
        {
            pitches.RemoveAt(0);
            semitone++;
        }
    }
    return semitone;
}
public Note.Pitch GetNoteUsingSemitone(int semitone)
```csharp
List<Note.Pitch> pitches = GetAllPitchesFromRoot();
return pitches[semitone];

public int GetIntervalFromC()
{
    int interval = 0;
    if (scale == KeyScale.Major)
    {
        switch (root)
        {
        case Note.Pitch.CsDb:
            interval = 2;
            break;
        case Note.Pitch.GsAb:
            interval = 6;
            break;
        case Note.Pitch.DsEb:
            interval = 3;
            break;
        case Note.Pitch.AsBb:
            interval = 7;
            break;
        case Note.Pitch.F:
            interval = 4;
            break;
        case Note.Pitch.C:
            interval = 1;
            break;
        case Note.Pitch.G:
            interval = 5;
            break;
        case Note.Pitch.D:
            interval = 2;
            break;
        case Note.Pitch.A:
            interval = 6;
            break;
        case Note.Pitch.E:
            interval = 3;
            break;
        case Note.Pitch.B:
            interval = 7;
            break;
        case Note.Pitch.FsGb:
            interval = 4;
            break;
        default:
            throw new Exception("Unknown key root note in
GetDistanceFromC.");
            break;
        }
    }
    else if (scale == KeyScale.Minor)
    {
        switch (root)
        {
        case Note.Pitch.CsDb:
            interval = 2;
            break;
        case Note.Pitch.GsAb:
            interval = 6;
            break;
        case Note.Pitch.DsEb:
            interval = 3;
            break;
        case Note.Pitch.AsBb:
            interval = 7;
            break;
        case Note.Pitch.F:
            interval = 4;
            break;
        case Note.Pitch.C:
            interval = 1;
            break;
        case Note.Pitch.G:
            interval = 5;
            break;
        case Note.Pitch.D:
            interval = 2;
            break;
        case Note.Pitch.A:
            interval = 6;
            break;
        case Note.Pitch.E:
            interval = 3;
            break;
        case Note.Pitch.B:
            interval = 7;
            break;
        case Note.Pitch.FsGb:
            interval = 4;
            break;
        default:
            throw new Exception("Unknown key root note in
GetDistanceFromC.");
            break;
        }
    }
```
case Note.Pitch.DsEb:
    interval = 3;
    break;
case Note.Pitch.AsBb:
    interval = 7;
    break;
case Note.Pitch.F:
    interval = 4;
    break;
case Note.Pitch.C:
    interval = 1;
    break;
case Note.Pitch.G:
    interval = 5;
    break;
case Note.Pitch.D:
    interval = 2;
    break;
case Note.Pitch.A:
    interval = 6;
    break;
case Note.Pitch.E:
    interval = 3;
    break;
case Note.Pitch.B:
    interval = 7;
    break;
case Note.Pitch.FsGb:
    interval = 4;
    break;
default:
    throw new Exception("Unknown key root note in GetDistanceFromC.");
    break;
}
else {
    throw new Exception("Unknown key scale on GetDistanceFromC!");
}

// Subtract one to make it zero-based.
interval--;

return interval;

// Private methods.
private int[] GetSemitonesUsedInScale()
{
    // Get the semitones for the scale.
    int[] semitonesInScale = null;
    if (scale == KeyScale.Major)
    {
        semitonesInScale = majorSemitones;
    }
    else if (scale == KeyScale.Minor)
    {
        semitonesInScale = minorSemitones;
    }
    else
    {

throw new Exception("Cannot convert scale "+ scale + " to semitones in key.");
}

return semitonesInScale;

private List<Note.Pitch> GetAllPitchesFromRoot()
{
    // Get a list of pitches and arrange it so the root note is first.
    List<Note.Pitch> pitchList = Utilities.EnumToList<Note.Pitch>().ToList<Note.Pitch>();
    for (int i = 0; i < pitchList.Count; i++)
    {
        Note.Pitch currentPitch = pitchList[0];
        if (currentPitch != root)
        {
            pitchList.RemoveAt(0);
            pitchList.Add(currentPitch);
        }
        else
        {
            // The list is sorted properly, exit the loop.
            break;
        }
    }

    return pitchList;
}

public string ToXML()
{
    string xmlOutput = "";
    xmlOutput += "<key>
";
    xmlOutput += "<fifths>" + RootToFifthsInt() + "</fifths>
";
    xmlOutput += "<mode>" + Utilities.GetEnumDescription(scale) + "</mode>
";
    xmlOutput += "</key>\n"
    return xmlOutput;
}

private int RootToFifthsInt()
{
    int fifthsInt = 0;
    if (scale == KeyScale.Major)
    {
        switch (root)
        {
            case Note.Pitch.A:
                fifthsInt = 3;
                break;
            case Note.Pitch.AsBb:
                fifthsInt = -2;
                break;
            case Note.Pitch.B:
                fifthsInt = 5;
                break;
            case Note.Pitch.C:
                fifthsInt = 0;
                break;
            case Note.Pitch.CsDb:
                fifthsInt = -5;
                break;
            case Note.Pitch.D:
                fifthsInt = 2;
break;
case Note.Pitch.DsEb:
fifthsInt = -3;
break;
case Note.Pitch.E:
fifthsInt = 4;
break;
case Note.Pitch.F:
fifthsInt = -1;
break;
case Note.Pitch.FsGb:
fifthsInt = 6;
break;
case Note.Pitch.G:
fifthsInt = 1;
break;
case Note.Pitch.GsAb:
fifthsInt = -4;
break;
default:
throw new Exception("Unknown pitch ' +
Utilities.GetEnumDescription(root) + '" when converting key to XML.");
} }
else if (scale == KeyScale.Minor) |
switch (root) |
| } |
| case Note.Pitch.A:
| fifthsInt = 0;
| break;
case Note.Pitch.AsBb:
| fifthsInt = -5;
| break;
case Note.Pitch.B:
| fifthsInt = 2;
| break;
case Note.Pitch.C:
| fifthsInt = -3;
| break;
case Note.Pitch.CsDb:
| fifthsInt = 4;
| break;
case Note.Pitch.D:
| fifthsInt = -1;
| break;
case Note.Pitch.DsEb:
| fifthsInt = 6;
| break;
case Note.Pitch.E:
| fifthsInt = 1;
| break;
case Note.Pitch.F:
| fifthsInt = -4;
| break;
case Note.Pitch.FsGb:
| fifthsInt = 3;
| break;
case Note.Pitch.G:
| fifthsInt = -2;
| break;
case Note.Pitch.GsAb:
| fifthsInt = 5;
KeyInterval.cs

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Frond
{
    public class KeyInterval
    {
        // Private data members.
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int intervalNumber = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int semitone = 0;

        // Setters/getters.
        public int IntervalNumber
        {
            get { return intervalNumber; }
        }
        public int Semitone
        {
            get { return semitone; }
        }

        // Constructors.
        public KeyInterval(int intervalNumber, int semitone)
        {
            this.intervalNumber = intervalNumber;
            this.semitone = semitone;
        }
        public KeyInterval(int intervalNumber, Key.KeyScale scale)
        {
this.intervalNumber = intervalNumber;

semitone = 0;
switch (intervalNumber)
{
    case 1:
        semitone = 0;
        break;
    case 2:
        semitone = 2;
        break;
    case 3:
        if (scale == Key.KeyScale.Major)
        {
            semitone = 4;
        } else if (scale == Key.KeyScale.Minor)
        {
            semitone = 3;
        } else
        {
            throw new Exception("Error converting key scale and interval to semitone.");
        }
        break;
    case 4:
        semitone = 5;
        break;
    case 5:
        semitone = 7;
        break;
    case 6:
        if (scale == Key.KeyScale.Major)
        {
            semitone = 9;
        } else if (scale == Key.KeyScale.Minor)
        {
            semitone = 8;
        } else
        {
            throw new Exception("Error converting key scale and interval to semitone.");
        }
        break;
    case 7:
        if (scale == Key.KeyScale.Major)
        {
            semitone = 11;
        } else if (scale == Key.KeyScale.Minor)
        {
            semitone = 10;
        } else
        {
            throw new Exception("Error converting key scale and interval to semitone.");
        }
        break;
    default:
throw new Exception("Error converting key interval to semitone.");
}

// Public methods.
public override string ToString()
{
    string buffer = "";
    buffer += "Interval: " + intervalNumber.ToString();
    buffer += ", Semitone: " + semitone.ToString();
    return buffer;
}
}

\---

Note.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class Note
    {
        // Constants.
        public const int DEFAULT_BASS_OCTAVE = 3;
        public const int DEFAULT_MELODY_OCTAVE = 4;
        public const int OCTAVE_RANGE = 1;
        // Private data members.
        public enum Pitch
        {
            C,
            [Description("C# / Db")]
            CsDb,
            D,
            [Description("D# / Eb")]
            DsEb,
            E,
            F,
            [Description("F# / Gb")]
            FsGb,
            G,
            [Description("G# / Ab")]
            GsAb,
            A,
            [Description("A# / Bb")]
            AsBb,
            B
        }
    }
}
private Pitch pitch;
private int octave;
private int duration;
private int staff;

// Setters/getters.
public Pitch NotePitch
{
    get { return pitch; }
}
public int Octave
{
    get { return octave; }
    set { octave = value; }
}
public int Duration
{
    get { return duration; }
    set { duration = value; }
}
public int Staff
{
    set { staff = value; }
    get { return staff; }
}

// Constructors.
public Note(string notation)
{
    // Determine the pitch.
    char[] pitchSeparators = new char[] { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
    string[] pitchTokens = notation.Split(pitchSeparators, StringSplitOptions.RemoveEmptyEntries);
    if (pitchTokens.Length != 1)
    {
        throw new Exception("Unable to determine pitch from new note " + notation + ", ");
    }
    string pitchString = pitchTokens[0];
    if (pitchString == "C")
    {
        pitch = Pitch.C;
    }
    else if (pitchString == "C#" || pitchString == "Db")
    {
        pitch = Pitch.CsDb;
    }
    else if (pitchString == "D")
    {
        pitch = Pitch.D;
    }
    else if (pitchString == "D#" || pitchString == "Eb")
    {
        pitch = Pitch.DsEb;
    }
    else if (pitchString == "E")
    {

pitch = Pitch.E;

} else if (pitchString == "F")
{
    pitch = Pitch.F;
}
else if (pitchString == "F#" || pitchString == "Gb")
{
    pitch = Pitch.FsGb;
}
else if (pitchString == "G")
{
    pitch = Pitch.G;
}
else if (pitchString == "G#" || pitchString == "Ab")
{
    pitch = Pitch.GsAb;
}
else if (pitchString == "A")
{
    pitch = Pitch.A;
}
else if (pitchString == "A#" || pitchString == "Bb")
{
    pitch = Pitch.AsBb;
}
else if (pitchString == "B")
{
    pitch = Pitch.B;
}
else
{
    throw new Exception("Unable to determine pitch from new note 
" + notation + ".");
}

// Determine the octave.
char[] octaveSeparators = new char[] { 'A', 'B', 'C', 'D', 'E', 
'F', 'G', '#', 'b' };
string[] octaveTokens = notation.Split(octaveSeparators, 
StringSplitOptions.RemoveEmptyEntries);
if (octaveTokens.Length != 1)
{
    throw new Exception("Unable to determine octave from new note 
" + notation + ".");
}
string octaveString = octaveTokens[0];
octave = Convert.ToInt32(octaveString);

// Set the duration to 0.
duration = 0;
staff = 0;

public Note(Pitch pitch, int octave)
{
    this.pitch = pitch;
    this.octave = octave;
    duration = 0;
    staff = 0;
}
public Note(string pitch, int octave)
{
    this.pitch = FromString(pitch);
this.octave = octave;
duration = 0;
staff = 0;
}

// Public methods.
public static Pitch FromString(string pitch)
{
    Pitch newPitch = Pitch.C;
    bool found = false;
    foreach (Pitch enumeratedType in Utilities.EnumToList<Pitch>())
    {
        string enumeratedTypeString =
        Utilities.GetEnumDescription(enumeratedType);
        if (pitch == enumeratedTypeString)
        {
            newPitch = enumeratedType;
            found = true;
        }
    }
    if (found == false)
    {
        throw new Exception("Error converting string '") + pitch + ";
        to pitch!");
    }
    return newPitch;
}
public static int CalculateDefaultOctave(string partName)
{
    int octave = 0;
    if (partName == "Melody")
    {
        octave = DEFAULT_MELODY_OCTAVE;
    }
    else if (partName == "Bass")
    {
        octave = DEFAULT_BASS_OCTAVE;
    }
    else
    {
        throw new Exception("Unknown part '") + partName + ";
        when calculating default octave.");
    }
    return octave;
}
public int ToInteger(Key key)
{
    int semitone = key.GetSemitoneOfNote(this);
    int noteInteger = semitone + (octave * 
    Key.NUM_SEMITONES_IN_SCALE));
    return noteInteger;
}
public override string ToString()
{
    string buffer = ";
    switch (pitch)
    {

case Pitch.AsBb:
    buffer += "A#";
    break;
case Pitch.CsDb:
    buffer += "C#";
    break;
case Pitch.DsEb:
    buffer += "D#";
    break;
case Pitch.FsGb:
    buffer += "F#";
    break;
case Pitch.GsAb:
    buffer += "G#";
    break;
default:
    buffer += Utilities.GetEnumDescription(pitch);
    break;
}
buffer += octave.ToString();
return buffer;
}

public string ToXML()
{
    String xmlOutput = "";
    // Print out the note.
    xmlOutput += "<note>
    xmlOutput += "<pitch>
    // Determine the pitch step and alter.
    string pitchStep = "";
    string pitchAlter = "";
    switch (pitch)
    {
        case Pitch.A:
            pitchStep = "A";
            pitchAlter = "0";
            break;
        case Pitch.AsBb:
            pitchStep = "A";
            pitchAlter = "1";
            break;
        case Pitch.B:
            pitchStep = "B";
            pitchAlter = "0";
            break;
        case Pitch.C:
            pitchStep = "C";
            pitchAlter = "0";
            break;
        case Pitch.CsDb:
            pitchStep = "C";
            pitchAlter = "1";
            break;
        case Pitch.D:
            pitchStep = "D";
            pitchAlter = "0";
            break;
        case Pitch.DsEb:
            pitchStep = "D";
            pitchAlter = "1";
            break;
    }
case Pitch.E:
    pitchStep = "E";
    pitchAlter = "0";
    break;
case Pitch.F:
    pitchStep = "F";
    pitchAlter = "0";
    break;
case Pitch.FsGb:
    pitchStep = "F";
    pitchAlter = "1";
    break;
case Pitch.G:
    pitchStep = "G";
    pitchAlter = "0";
    break;
case Pitch.GsAb:
    pitchStep = "G";
    pitchAlter = "1";
    break;
default:
    throw new Exception("Unable to convert note '" + Utilities.GetEnumDescription(pitch) + '" to XML.");
    break;
}

xmlOutput += "<step>" + pitchStep + "</step>\n";
xmlOutput += "<alter>" + pitchAlter + "</alter>\n";
xmlOutput += "<octave>" + octave.ToString() + "</octave>\n";
xmlOutput += "<pitch>\n";
xmlOutput += "<duration>" + duration.ToString() + "</duration>\n";

if (pitchAlter == "-2")
    { xmlOutput += "<accidental>flat-flat</accidental>"; }
else if (pitchAlter == "-1")
    { xmlOutput += "<accidental>flat</accidental>"; }
else if (pitchAlter == "1")
    { xmlOutput += "<accidental>sharp</accidental>"; }
else if (pitchAlter == "2")
    { xmlOutput += "<accidental>sharp-sharp</accidental>"; }
xmOutput += "<staff>" + staff + "</staff>\n";
xmOutput += "</note>\n";

return xmlOutput;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Frond
{
    public class NoteDerivation
    {
        // Constants.
        public const int CENTER_NOTE_INDEX = (int)(CellularAutomaton.DEFAULT_CA_WIDTH / 2);
        // Private data members.
        private Note[] notes = new Note[CellularAutomaton.DEFAULT_CA_WIDTH];
        // Setters/getters.
        public Note this[int i]
        {
            get { return notes[i]; } } } } // Constructors.
    public NoteDerivation(Note rootNote)
    {   // Set all notes in the map to the root note.
        for (int i = 0; i < notes.Length; i++)
        {
            notes[i] = new Note(rootNote.NotePitch, rootNote.Octave);
        }
    }
    public NoteDerivation(int lastNoteSemitone, int lastNoteOctave, int nominalOctave, Key key, KeyInterval[] keyIntervals, ChordInterval[] chordIntervals)
    {   // Determine the octave, semitone, and interval number of the note in the center of the derivation.
        int centerOctave = 0;
        int centerSemitone = CalculateCenterNote(lastNoteSemitone, lastNoteOctave, chordIntervals, out centerOctave);
        int centerIntervalIndex = CalculateCenterIntervalIndex(centerSemitone, keyIntervals);
        // Create the list of intervals above and below the center interval.
        List<int> intervalIndices = new List<int>();
        for (int i = (CENTER_NOTE_INDEX * -1); i < 0; i++)
        {
            int newIndex = centerIntervalIndex + i;
            while (newIndex < 0)
            {
                newIndex += Key.NOTES_IN_SCALE;
            }
            intervalIndices.Add(newIndex);
        }
        intervalIndices.Add(centerIntervalIndex);
        for (int i = 1; i <= CENTER_NOTE_INDEX; i++)
        {
int newIndex = (centerIntervalIndex + i) % Key.NOTES_IN_SCALE;
intervalIndices.Add(newIndex);

// Create the notes for the intervals specified.
for (int i = 0; i < intervalIndices.Count; i++)
{
    int intervalIndex = intervalIndices[i];

    // Determine the pitch of the note.
    Note.Pitch notePitch = key.GetNoteUsingSemitone(keyIntervals[intervalIndex].Semitone);

    // Determine the octave of the note.
    int keyRootInterval = key.GetIntervalFromC();
    int distance = (i - CENTER_NOTE_INDEX);
    int noteOctave = CalculateOctaveFromCenterNote(centerIntervalIndex, centerOctave, keyRootInterval, distance);

    // Create the note.
    notes[i] = new Note(notePitch, noteOctave);
}

// Check for out of bounds conditions in the derivation.
Note keyRootNote = new Note(key.Root, nominalOctave);
Note activeNote = notes[CENTER_NOTE_INDEX];
int outOfRange = Note.OCTAVE_RANGE * Key.NUM_SEMITONES_IN_SCALE;

// If the difference between the key root and the active note is greater than
// the out of range variable, adjust the note derivation accordingly.
int difference = keyRootNote.ToInteger(key) -
activeNote.ToInteger(key);
if (difference >= outOfRange)
{
    // Exceeded lower range. Increment each note in the
derivation by 1 octave.
    ShiftOctave(1);
}
if (difference <= (-1 * outOfRange))
{
    // Exceeded upper range. Decrement each note in the
derivation by 1 octave.
    ShiftOctave(-1);
}

public void ShiftOctave(int shift)
{
    for (int i = 0; i < notes.Length; i++)
    {
        notes[i].Octave = notes[i].Octave + shift;
    }
}

public static int CalculateCenterNote(int lastNoteSemitone, int lastNoteOctave,
ChordInterval[] chordIntervals, out int octave)
{
    int selectedOctave = 0;
    int selectedSemitone = 0;
// Calculate the distance between the semitones in the chord and
the last note's semitone.
    int smallestDistance = 100000;
    int lastNoteSemitoneWithOctave = lastNoteSemitone + 
(lastNoteOctave * Key.NUM_SEMITONES_IN_SCALE);
    foreach (ChordInterval chordInterval in chordIntervals)
    {
        int chordSemitone = chordInterval.Semitone;
        int intervalNumber = chordInterval.IntervalNumber;

        // Calculate the distance within +/- the octave range.
        int lowerOctave = lastNoteOctave - Note.OCTAVE_RANGE;
        if (lowerOctave < 0)
        {
            throw new Exception("Negative lower octave encountered
during note derivation!");
        }
        int upperOctave = lastNoteOctave + Note.OCTAVE_RANGE;
        for (int i = lowerOctave; i <= upperOctave; i++)
        {
            int distance = Math.Abs(lastNoteSemitoneWithOctave -
(chordSemitone + (i * Key.NUM_SEMITONES_IN_SCALE)));
            if (distance < smallestDistance)
            {
                smallestDistance = distance;
                // This is the closest semitone so far, so save the
                // octave and semitone.
                selectedSemitone = chordInterval.Semitone;
                selectedOctave = i;
            }
        }
    }

    // Determine the octave of the semitone.
    octave = selectedOctave;
    return selectedSemitone;
}

private int CalculateCenterIntervalIndex(int centerNoteSemitone,
KeyInterval[] keyIntervals)
{
    int centerIntervalIndex = -1;
    for (int i = 0; i < keyIntervals.Length; i++)
    {
        if (centerNoteSemitone == keyIntervals[i].Semitone)
        {
            centerIntervalIndex = i;
            break;
        }
    }

    if (centerIntervalIndex == -1)
    {
        throw new Exception("Center interval for semitone " +
centerNoteSemitone.ToString() + " not found in note derivation!");
    }

    return centerIntervalIndex;
}

private int CalculateOctaveFromCenterNote(int centerNoteIntervalIndex, int centerNoteOctave, int keyRootInterval, int distance)
```csharp
int centerNoteWithOctave = centerNoteIntervalIndex +
keyRootInterval + (centerNoteOctave * Key.NOTES_INSCALE);
int noteWithOctave = centerNoteWithOctave + distance;
int noteOctave = (int)(noteWithOctave / Key.NOTES_INSCALE);
return noteOctave;
```

### Song.cs

```csharp
using CellularAutomata;
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    public class Song
    {
        // Constants
        public const int DEFAULT_STEPS_PER_MEASURE = 32;

        // Private data members
        private Key key = null;
        private Time time = null;
        private bool bassAlwaysPlayRootNote = false;
        private int stepsPerMeasure = DEFAULT_STEPS_PER_MEASURE;
        private string[] structure = null;
        private List<Part> parts = null;
        private Dictionary<string, List<string>> patterns = null;
        private Dictionary<string, List<string>> measures = null;
        private List<string> allMeasures = null;
        private Dictionary<string, Dynamics.DynamicsType> dynamics = null;
        private Dictionary<string, Chord> chords = null;
        private Profile profile = null;

        // Setters/getters.
        public int StepsPerMeasure
        {
            get { return stepsPerMeasure; }
            set { stepsPerMeasure = value; }
        }
        public Key Key
        {
            set { key = value; }
            get { return key; }
        }
        public Time Time
        {
```csharp
    set { time = value; }
    get { return time; }
}

dictionary<string, string[]> Structure
    set { structure = value; }
    get { return structure; }

public Profile Profile
    get { return profile; }
    set { profile = value; }

public Dictionary<string, List<string>> Patterns
    set { patterns = value; }

public Dictionary<string, List<string>> Measures
    set
    {
        measures = value;
        allMeasures = new List<string>();
        // Populate the list of all measures.
        Dictionary<string, List<string>>.ValueCollection measureCollection = measures.Values;
        foreach (List<string> measureList in measureCollection)
        {
            foreach (string measure in measureList)
            {
                allMeasures.Add(measure);
            }
        }
    }

public Dictionary<string, Dynamics.DynamicsType> Dynamics
    set { dynamics = value; }

public Dictionary<string, Chord> Chords
    set { chords = value; }

    // Methods.
    public void AddPart(string partName, Rule rule, Cell[] initialConditions, int length)
    {
        // Ensure parts exist.
        if (parts == null)
        {
            parts = new List<Part>();
        }

        // Create a new part.
        Part part = new Part(partName);
```
// Create a new CA and add it to the part.
CellularAutomaton cellularAutomaton = new CellularAutomaton();
cellularAutomaton.Rule = rule;
cellularAutomaton.InitialConditions = initialConditions;
cellularAutomaton.Length = length;
part.CellularAutomaton = cellularAutomaton;

// Set the always play root note flag if necessary.
if (partName == "Bass")
{
    part.AlwaysPlayRootNote = bassAlwaysPlayRootNote;
}
else
{
    // It's not an option for any other part to only play the
    // root note.
    part.AlwaysPlayRootNote = false;
}

// Set the key into the part.
part.Key = key;

// Add it to our parts list.
parts.Add(part);

public void CreateNoteDerivations()
{
    // Create the key intervals for each chord.
    Dictionary<string, KeyInterval[]]> keyIntervalMap = CreateKeyIntervals();

    // Create all the measures for each part.
    CreateMeasuresForParts();

    // Determine the derivations for each part.
    foreach (Part part in parts)
    {
        part.CalculateNoteDerivations(keyIntervalMap);
    }

    // Check for intersections between the melody and bass parts.
    CheckNoteDerivationsForIntersections();
}

public List<string> RetrievePatterns()
{
    List<string> patternsList = new List<string>();
    Dictionary<string, List<string>>.KeyCollection patternCollection = measures.Keys;

    foreach (string pattern in patternCollection)
    {
        patternsList.Add(pattern);
    }

    return patternsList;
}

public List<string> RetrieveMeasures(string pattern)
{
    return measures[pattern];
}

public string[] RetrieveMeasureHeader(string partName, string measureName)
string[] header = null;
foreach (Part part in parts)
{
    if (part.Name == partName)
    {
        Measure measure = part.FindMeasure(measureName);
        header = measure.GetHeader();
        break;
    }
}
if (header == null)
{
    throw new Exception("Could not obtain measure header for measure " + measureName + " and part '' + partName + ".");
}
return header;

public List<Cell[]> RetrieveCells(string partName, string pattern)
{
    List<Cell[]> cells = new List<Cell[]>();
    // Ensure the part exists in our dictionary.
    Part part = null;
    foreach (Part p in parts)
    {
        if (p.Name == partName)
        {
            part = p;
            break;
        }
    }
    if (part == null)
    {
        throw new Exception("Part not found during cell retrieval! ");
    }
    // Retrieve the first measure with this pattern.
    List<string> measureList = measures[(pattern,
    if (measureList.Count <= 0)
    {
        throw new Exception("No measures found within pattern "+ pattern + " during cell retrieval.");
    }
    string firstMeasure = measureList[0];
    // Determine the number of previous measures and number of measures.
    // This is used to index into the appropriate portion of the CA.
    int numPreviousMeasures = allMeasures.IndexOf(firstMeasure);
    int numMeasures = measureList.Count;
    if (numPreviousMeasures >= 0)
    {
        // Retrieve the part to use.
        CellularAutomaton cellularAutomaton = part.CellularAutomaton;
        // Initialize our cells.
        List<Cell[]> caCells = cellularAutomaton.Cells;
        int startingIndex = numPreviousMeasures * stepsPerMeasure;
        int endingIndex = (numPreviousMeasures + numMeasures) * stepsPerMeasure;
    }
if (endingIndex <= caCells.Count)
{
    for (int i = startingIndex; i < endingIndex; i++)
    {
        cells.Add(caCells[i]);
    }
} else
{
    string message = "Ending index '" + endingIndex;
    message += '"' exceeds cellular automaton cell count of '" + caCells.Count;
    message += '"' during cell retrieval.";
    throw new Exception(message);
}
else
{
    throw new Exception("Unknown measure '" + firstMeasure + '" used during cell retrieval.");
}
return cells;
}
public string ToXML()
{
    string xmlOutput = "";
    // XML file header.
    xmlOutput += "<?xml version="1.0" encoding="UTF-8" standalone="no"?>\n";
    xmlOutput += "<!DOCTYPE score-partwise PUBLIC\n";
    xmlOutput += "-//Recordare//DTD MusicXML 2.0 Partwise//EN\n";
    xmlOutput += "http://www.musicxml.org/dtds/partwise.dtd">\n";
    // Print out the song.
    xmlOutput += "<score-partwise version="2.0">\n";
    xmlOutput += "<work>\n";
    xmlOutput += "<work-title>A Generated Song</work-title>\n";
    xmlOutput += "</work>\n";
    xmlOutput += "<identification>\n";
    xmlOutput += "<creator type="composer">" + "Key in integer\n";
    xmlOutput += Utilities.GetEnumDescription(key.Root) + ";\n";
    xmlOutput += Utilities.GetEnumDescription(key.Scale) + ";\n";
    xmlOutput += "</creator>\n";
    xmlOutput += "<rights>N/A</rights>\n";
    // Output encoding.
    string date = DateTime.Now.Year.ToString() + "-";
    date += DateTime.Now.Month.ToString() + "-";
    date += DateTime.Now.Day.ToString();
    xmlOutput += "<encoding>\n";
    xmlOutput += "<encoding-date>" + date + "</encoding-date>\n";
    xmlOutput += "<software>" + "Frond" + "</software>\n";
    xmlOutput += "</encoding>\n";
    xmlOutput += "</identification>\n";
    // Part list.
    xmlOutput += "<part-list>\n";
    xmlOutput += "<score-part id="P1">\n";
    xmlOutput += "<part-name>Melody</part-name>\n";
    xmlOutput += "<score-instrument id="P1-I1">\n";
}
xmlOutput += "<instrument-name>Grand Piano</instrument-name>\n";
xmlOutput += "</score-instrument>\n";
xmlOutput += "<midi-instrument id="P1-I1">\n";
xmlOutput += "<midi-channel>1</midi-channel>\n";
xmlOutput += "<midi-program>1</midi-program>\n";
xmlOutput += "</midi-instrument>\n";
xmlOutput += "</score-part>\n";
xmlOutput += "<score-part id="P2">\n";
xmlOutput += "<part-name>Bass</part-name>\n";
xmlOutput += "<instrument-name>Grand Piano</instrument-name>\n";
xmlOutput += "</score-instrument>\n";
xmlOutput += "<midi-instrument id="P2-I1">\n";
xmlOutput += "<midi-channel>1</midi-channel>\n";
xmlOutput += "<midi-program>1</midi-program>\n";
xmlOutput += "</midi-instrument>\n";
xmlOutput += "</score-part>\n";
xmlOutput += "</part-list>\n";

// Get our bass and melody parts.
Part melodyPart = null;
Part bassPart = null;
foreach (Part part in parts) {
    if (part.Name == "Melody") {
        melodyPart = part;
    } else if (part.Name == "Bass") {
        bassPart = part;
    } else {
        throw new Exception("Unknown part "+ part.Name + " encountered when checking intersections between melody and bass parts.");
    }
}
if (melodyPart == null || bassPart == null) {
    throw new Exception("Melody or bass part not found when checking intersections between melody and bass parts.");
}

// Output the measures of the melody part.
Frond.Dynamics.DynamicsType lastDynamicsType = Frond.Dynamics.DynamicsType.Forte;
bool dynamicsTypePrinted = false;
xmlOutput += "<part id="P1">\n";
for (int i = 0; i < melodyPart.Measures.Count; i++) {
    Measure melodyMeasure = melodyPart.Measures[i];
    // Output start of measure.
    xmlOutput += "<measure number=" + (i + 1) + "">\n";
    xmlOutput += "<attributes>\n";
    xmlOutput += "</divisions>\n";
    // Output key.
    xmlOutput += key.ToXML();
    // Output time.
xmlOutput += time.ToXML();
xmlOutput += "<staves">" + "1" + "</staves>
"
// Output clefs, which are always constant.
xmoutput += "<clef number="" + "1" + "</clef>
"
xmlOutput += "<sign>" + "G" + "</sign>
"
xmlOutput += "<line>" + "2" + "</line>
"
xmlOutput += "</attributes>
"
// Output the chord as text within the measure.
xmlOutput += "<direction placement="above">
"
xmlOutput += "<words default-y="26" font-size="11" font-weight="bold" relative-x="0" xml:lang="en">";
xmlOutput += Utilities.GetEnumDescription(key.GetNoteUsingScaleDegree(melodyMeasure.Chord.Degree - 1));
if (melodyMeasure.Chord.Type != Chord.ChordType.Major)
{
xmlOutput += Chord.LookupChordTypeNotation(melodyMeasure.Chord.Type);
}
xmOutput += "</words>
"
xmlOutput += "</direction>
"
// Set the tempo and dynamics for MIDI.
xmlOutput += "<direction placement="below">
"
xmlOutput += "<dynamics default-y="-90">
"
xmlOutput += "<" + Frond.Dynamics.LookupDynamicsNotation(melodyMeasure.Dynamics.Type) + "/>
"
xmlOutput += "</dynamics>
"
xmlOutput += "</direction>
"
// First output the melody part, then the bass part.
xmlOutput += melodyMeasure.ToXML();
xmlOutput += "</measure>
"
// Output the measures of the bass part.
xmlOutput += "</part>
"
// Output the measures of the bass part.
xmlOutput += "<part id="P2">
";
for (int i = 0; i < bassPart.Measures.Count; i++)
{
Measure bassMeasure = bassPart.Measures[i];

// Output start of measure.
xmOutput += "<measure number=" + (i + 1) + "">\n";
xmOutput += "<attributes>\n";
xmOutput += "<divisions>" + bassMeasure.Divisions + "</divisions>\n";

// Output key.
xmOutput += key.ToXML();

// Output time.
xmOutput += "<staves>" + "1" + "</staves>\n";

// Output clefs, which are always constant.
xmOutput += "<clef number=" + "1" + "">\n";
xmOutput += "<sign>" + "F" + "</sign>\n";
xmOutput += "<line>" + "4" + "</line>\n";
xmOutput += "</clef>\n";
xmOutput += "</attributes>\n";

// Set the tempo and dynamics for MIDI.
xmOutput += "<direction placement="below">\n";
xmOutput += "<direction-type>\n";
xmOutput += "<words default-y="26" font-size="11" font-weight="bold" relative-x="0" xml:lang="en">";
xmOutput += "</direction-type>\n";
xmOutput += "<staff>1</staff>\n";
xmOutput += "<sound dynamics=" + Frond.Dynamics.LookupDynamicsVolume(bassMeasure.Dynamics.Type) + " \n";
xmOutput += "<tempo=" + time.Tempo.ToString() + "/>\n";
xmOutput += "</direction>\n";

// First output the melody part, then the bass part.
xmOutput += bassMeasure.ToXML();
xmOutput += "</measure>\n";

xmlOutput += "</part>\n";

// Output the closing XML remarks.
xmOutput += "</score-partwise>\n";

return xmlOutput;

// Private methods.
private Dictionary<string, KeyInterval[]> CreateKeyIntervals()
{
    Dictionary<string, KeyInterval[]> keyIntervalMap = new
    Dictionary<string, KeyInterval[]>();

    KeyInterval[] keyIntervals = key.KeyIntervals;

    // Determine the new key intervals for each chord.
    foreach (KeyValuePair<string, Chord> measureChordPair in chords)
    {
        string measure = measureChordPair.Key;
        Chord chord = measureChordPair.Value;

        // Copy the existing key intervals.
        ChordInterval[] chordIntervals = chord.CalculateAbsoluteIntervals(keyIntervals);
        KeyInterval[] newKeyIntervals = new
        KeyInterval[keyIntervals.Length];

        // Determine the new key intervals for each chord.

for (int i = 0; i < keyIntervals.Length; i++)
{
    int keyIntervalNumber = keyIntervals[i].IntervalNumber;
    bool intervalFound = false;

    // Copy the chord interval semitone, if it exists.
    foreach (ChordInterval chordInterval in chordIntervals)
    {
        int chordIntervalNumber = chordInterval.IntervalNumber;
        if (chordIntervalNumber > keyIntervals.Length)
        {
            // Incase the interval is as high as an octave.
            chordIntervalNumber = chordIntervalNumber %
        }
        if (chordIntervalNumber == keyIntervalNumber)
        {
            int semitone = chordInterval.Semitone;
            newKeyIntervals[i] = new KeyInterval(keyIntervalNumber, semitone);
            intervalFound = true;
            break;
        }
    }

    // If the chord interval semitone doesn't exist for this
    key interval, just copy the key interval.
    if (intervalFound == false)
    {
        int semitone = keyIntervals[i].Semitone;
        newKeyIntervals[i] = new KeyInterval(keyIntervalNumber, semitone);
    }
}

    // Add the new key intervals for the measure to our map.
    keyIntervalMap.Add(measure, newKeyIntervals);
}
return keyIntervalMap;

private void CreateMeasuresForParts()
{
    foreach (string elementName in structure)
    {
        Element element = profile.FindElement(elementName);

        // For every pattern within an element...
        foreach (Pattern pattern in element.Patterns)
        {
            int repeats = pattern.Repeats;

            // Loop once for each time the pattern repeats.
            for (int i = 0; i < repeats; i++)
            {
                // For every measure within a pattern...
                foreach (Measure measure in pattern.Measures)
                {
                    // For every part of the song...
                    foreach (Part part in parts)
                    {
                        // For every chord in the part...
                        foreach (Chord chord in part.Chords)
                        {
                            // Calculate the new key intervals for each chord.
                            foreach (ChordInterval chordInterval in chord.ChordIntervals)
                            {
                                int chordIntervalNumber = chordInterval.IntervalNumber;
                                if (chordIntervalNumber > keyIntervals.Length)
                                {
                                    // Incase the interval is as high as an octave.
                                    chordIntervalNumber = chordIntervalNumber %
                                }
                                if (chordIntervalNumber == keyIntervalNumber)
                                {
                                    int semitone = chordInterval.Semitone;
                                    newKeyIntervals[i] = new KeyInterval(keyIntervalNumber, semitone);
                                    intervalFound = true;
                                    break;
                                }
                            }

                            // If the chord interval semitone doesn't exist for this
                            key interval, just copy the key interval.
                            if (intervalFound == false)
                            {
                                int semitone = keyIntervals[i].Semitone;
                                newKeyIntervals[i] = new KeyInterval(keyIntervalNumber, semitone);
                            }

                            // Add the new key intervals for the measure to our map.
                            keyIntervalMap.Add(measure, newKeyIntervals);
                        }
                    }
                }
            }
        }
    }
}
// See if the measure already exists in the part.
if (part.MeasureExists(measure.Name) == true)
{
    Measure newMeasure = part.FindMeasure(measure.Name);
    part.AddMeasure(newMeasure);
} else
{
    // Create a deep copy of the current measure.
    Measure newMeasure = new Measure(measure);
    // Set the dynamics of the pattern into the CA. (Hackish code)
    List<string> measureList = measures[pattern.Name];
    if (measureList.Count <= 0)
    {
        throw new Exception("No measures found within pattern "+ pattern + "," during cell retrieval.");
    }
    int numPreviousMeasures = allMeasures.IndexOf(measure.Name);
    newMeasure.StartingRow = numPreviousMeasures * stepsPerMeasure;
    newMeasure.NumRows = stepsPerMeasure;
    newMeasure.Divisions = stepsPerMeasure / time.Beat;
    // Set the staff for the part.
    // This value is always 1, since melody and bass are split into separate parts.
    newMeasure.Staff = 1;
    // Set the CA into the measure.
    // Add the measure to the part.
    part.AddMeasure(newMeasure);
}

private void CheckNoteDerivationsForIntersections()
{
    // Find our melody and bass parts.
    Part melodyPart = null;
    Part bassPart = null;
    foreach (Part part in parts)
    {
        if (part.Name == "Melody")
        {
            melodyPart = part;
        }
        else if (part.Name == "Bass")
        {
            bassPart = part;
        }
    }
    // Check for note derivations across parts.
    if (melodyPart != null && bassPart != null)
    {
        CheckNoteDerivationsAcrossParts(melodyPart, bassPart);
    }
}
{ melodyPart = part; }
else if (part.Name == "Bass")
{ bassPart = part; }
else
{
    throw new Exception("Unknown part "+ part.Name + ", encountered when checking intersections between melody and bass parts.");
}
if (melodyPart == null || bassPart == null)
{
    throw new Exception("Melody or bass part not found when checking intersections between melody and bass parts.");
}

// Check for intersections with each measure.
int count = melodyPart.Measures.Count;
if (count != bassPart.Measures.Count)
{
    throw new Exception("Melody and bass parts have different number of measures when checking intersections between melody and bass parts.");
}
for (int i = 0; i < count; i++)
{
    Measure melodyMeasure = melodyPart.Measures[i];
    Measure bassMeasure = bassPart.Measures[i];
    NoteDerivation melodyNoteDerivation = melodyMeasure.NoteDerivation;
    NoteDerivation bassNoteDerivation = bassMeasure.NoteDerivation;
    Note currentMelodyNote = melodyNoteDerivation[NoteDerivation.CENTER_NOTE_INDEX];
    Note currentBassNote = bassNoteDerivation[NoteDerivation.CENTER_NOTE_INDEX];

    // Check for an intersection.
    if (currentMelodyNote.ToInteger(key) <=
        currentBassNote.ToInteger(key))
    {
        // There's an intersection. Correct each derivation by 1 octave.
        melodyNoteDerivation.ShiftOctave(1);
        bassNoteDerivation.ShiftOctave(-1);
    }
}
using CellularAutomata;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace Frond
{
    class SongBuilder
    {
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Time time = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Key key = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int stepsPerMeasure = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int melodyElementaryRule = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int melodyFracturingRule = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int bassElementaryRule = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private int bassFracturingRule = 0;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private bool bassAlwaysPlayRootNote = false;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Cell[] initialConditions = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private string[] structure = null;
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dictionary<string, List<string>> patterns = new Dictionary<string, List<string>>();
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dictionary<string, List<string>> measures = new Dictionary<string, List<string>>();
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dictionary<string, int> repeats = new Dictionary<string, int>();
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dictionary<string, Dynamics.DynamicsType> dynamics = new Dictionary<string, Dynamics.DynamicsType>();
        [DebuggerBrowsableAttribute(DebuggerBrowsableState.Never)]
        private Dictionary<string, Chord> chords = new Dictionary<string, Chord>();

        // Setters/getters.
        public Time Time
        {
            set { time = value; }
            get { return time; }
        }
        public Key Key
        {
            set { key = value; }
            get { return key; }
        }
    }
}
public int StepsPerMeasure
{
    set { stepsPerMeasure = value; }
    get { return stepsPerMeasure; }
}
public int MelodyElementaryRule
{
    set { melodyElementaryRule = value; }
    get { return melodyElementaryRule; }
}
public int MelodyFracturingRule
{
    set { melodyFracturingRule = value; }
    get { return melodyFracturingRule; }
}
public int BassElementaryRule
{
    set { bassElementaryRule = value; }
    get { return bassElementaryRule; }
}
public int BassFracturingRule
{
    set { bassFracturingRule = value; }
    get { return bassFracturingRule; }
}
public bool BassAlwaysPlayRootNote
{
    set { bassAlwaysPlayRootNote = value; }
    get { return bassAlwaysPlayRootNote; }
}
public Cell[,] InitialConditions
{
    set { initialConditions = value; }
    get { return initialConditions; }
}
public string[,] Structure
{
    set { structure = value; }
    get { return structure; }
}
public Dictionary<string, List<string>> Patterns
{
    get { return patterns; }
}
public Dictionary<string, List<string>> Measures
{
    get { return measures; }
}
public Dictionary<string, Dynamics.DynamicsType> Dynamics
{
    get { return dynamics; }
}
public Dictionary<string, Chord> Chords
{
    get { return chords; }
}

// Methods.
public void AddPattern(string element, string pattern,
Dynamics.DynamicsType dynamicsType, int repeats)
{
    try
    {

if (patterns.ContainsKey(element) == false)
{
    List<string> patternList = new List<string>();
    patternList.Add(pattern);
    patterns.Add(element, patternList);
}
else
{
    patterns[element].Add(pattern);
}  

if (measures.ContainsKey(pattern) == false)
{
    List<string> measureList = new List<string>();
    measureList.Add(measure);
    measures.Add(pattern, measureList);
}
else
{
    measures[pattern].Add(measure);
}  

chords.Add(measure, chord);
}

for (KeyValuePair<string, List<string>> elementPatternsPair
    in patterns)
{
    try
    {
        if (patterns.ContainsKey(element) == false)
        {
            List<string> patternList = new List<string>();
            patternList.Add(pattern);
            patterns.Add(element, patternList);
        }
        else
        {
            patterns[element].Add(pattern);
        }
        dynamics.Add(pattern, dynamicsType);
        this.repeats.Add(pattern, repeats);
    }
    catch (Exception e)
    {
        string message = "Error importing pattern " + pattern + ", for element " + element + ".\n"
        message += "Error: " + e.ToString();
        throw new Exception(message);
    }
}

for (KeyValuePair<string, List<string>> patternMeasuresPair
    in measures)
{
    try
    {
        if (measures.ContainsKey(pattern) == false)
        {
            List<string> measureList = new List<string>();
            measureList.Add(measure);
            measures.Add(pattern, measureList);
        }
        else
        {
            measures[pattern].Add(measure);
        }
        chords.Add(measure, chord);
    }
    catch (Exception e)
    {
        string message = "Error importing measure " + measure + ", for pattern " + pattern + ".\n"
        message += "Error: " + e.ToString();
        throw new Exception(message);
    }
}

public Song GenerateSong()
{
    // Create a song.
    Song song = new Song();
    song.Key = key;
    song.Time = time;
    song.Patterns = patterns;
    song.Measures = measures;
    song.Dynamics = dynamics;
    song.Chords = chords;
    song.StepsPerMeasure = stepsPerMeasure;
    song.BassAlwaysPlayRootNote = bassAlwaysPlayRootNote;
    song.Structure = structure;

    // Create a profile of the song.
    Profile profile = new Profile();
    foreach (KeyValuePair<string, List<string>> elementPatternsPair
        in patterns)
Element newElement = new Element(elementPatternsPair.Key);
List<string> patternNames = elementPatternsPair.Value;

// Add each pattern to the element.
foreach (string patternName in patternNames)
{
    Dynamics newDynamics = new Dynamics(dynamics[patternName]);
    int newRepeats = repeats[patternName];
    Pattern newPattern = new Pattern(patternName, newDynamics, newRepeats);

    // Add each measure to the pattern.
    List<string> measureNames = measures[patternName];
    foreach (string measureName in measureNames)
    {
        Chord newChord = chords[measureName];
        Measure newMeasure = new Measure(measureName, newChord);
        newPattern.AddMeasure(newMeasure);
    }

    // Add the pattern to the element.
    newElement.AddPattern(newPattern);
}

// Add the element to the profile.
profile.AddElement(newElement);

song.Profile = profile;

// Calculate the length the CAs must be.
int caLength = chords.Keys.Count * stepsPerMeasure;

// Initialize the melody and bass CAs.
Rule melodyRule = new Rule(melodyElementaryRule, melodyFracturingRule);
song.AddPart("Melody", melodyRule, initialConditions, caLength);
Rule bassRule = new Rule(bassElementaryRule, bassFracturingRule);
song.AddPart("Bass", bassRule, initialConditions, caLength);

// Derive the note derivations for each chord.
song.CreateNoteDerivations();

// Return the song.
return song;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Frond
{
    public class Time
    {
        // Constants
        public const int DEFAULT_BEAT = 4;
        public const int DEFAULT_BEAT_TYPE = 4;
        public const int DEFAULT_TEMPO = 130;

        // Private data members.
        private int tempo = DEFAULT_TEMPO;
        private int beat = DEFAULT_BEAT;
        private int beatType = DEFAULT_BEAT_TYPE;

        // Setters/getters.
        public int Tempo
        {
            get { return tempo; }
            set { tempo = value; }
        }
        public int Beat
        {
            get { return beat; }
            set { beat = value; }
        }
        public int BeatType
        {
            get { return beatType; }
            set { beatType = value; }
        }

        // Public methods.
        public static Time FromString(string beat, string beatType, string tempo)
        {
            Time time = new Time();
            // Attempt to convert beat.
            try
            {
                time.Beat = Convert.ToInt32(beat);
            }
            catch (Exception e)
            {
                throw new Exception("Error converting beat '" + beat + '" to integer!");
            }

            // Attempt to convert beat type.
            try
            {
                time.BeatType = Convert.ToInt32(beatType);
            }
        }
    }
}
catch (Exception e)
{
    throw new Exception("Error converting beat type '" +
    beatType + '" to integer!");
}

// Attempt to convert tempo.
try
{
    time.Tempo = Convert.ToInt32(tempo);
}
catch (Exception e)
{
    throw new Exception("Error converting tempo '" + tempo + '"
    to integer!");
}

return time;
}

public string ToXML()
{
    string xmlOutput = "";
    xmlOutput += "<time>
    
    <beats>" + beat + "</beats>
    
    <beat-type>" + beatType + "</beat-type>
    
    </time>
    ";
    return xmlOutput;
}